4 Queues and Lists

This chapter introduces the queue and the priority queue, two important data structures often used to simulate real world situations. The concepts of the stack and queue are then extended to a new structure, the list. Various forms of lists and their associated operations are examined and several applications are presented.

4.1 THE QUEUE AND ITS SEQUENTIAL REPRESENTATION

A ***queue*** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (called the ***rear*** of the queue).

Figure 4.1.1a illustrates a queue containing three elements *A*, *B*, and *C*. *A* is at the front of the queue and *C* is at the rear. In Figure 4.1.1b an element has been deleted from the queue. Since elements may be deleted only from the front of the queue, *A* is removed and *B* is now at the front. In Figure 4.1.1c, when items *D* and *E* are inserted, they must be inserted at the rear of the queue.

Since *D* was inserted into the queue before *E*, it will be removed earlier. The first element inserted into a queue is the first element to be removed. For this reason a queue is sometimes called a ***fifo*** (first-in, first-out) list as opposed to a stack, which is a ***lifo***

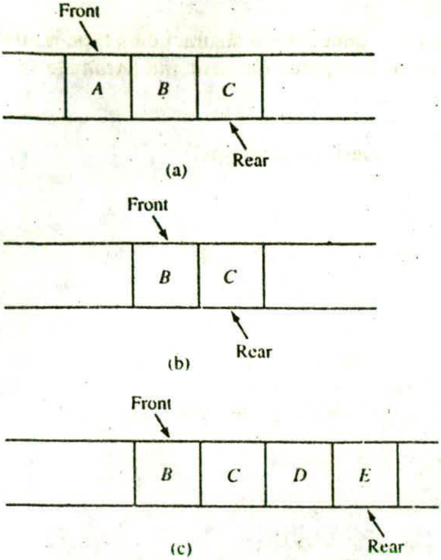


Figure 4.1.1 Queue

(last-in, first-out) list. Examples of a queue abound in the real world. A line at a bank or at a bus stop and a group of cars waiting at a toll booth are all familiar examples of queues.

Three primitive operations can be applied to a queue. The operation *insert(q,x)* inserts item *x* at the rear of the queue *q*. The operation *x = remove(q)* deletes the front element from the queue *q* and sets *x* to its contents. The third operation, *empty(q)*, returns *false* or *true* depending on whether or not the queue contains any elements. The queue in Figure 4.1.1 can be obtained by the following sequence of operations. We assume that the queue is initially empty.

```
insert(q, A);
insert(q, B);
insert(q, C);              (Figure 4.1.1a)
x = remove(q);             (Figure 4.1.1b; x is set to A)
insert(q, D);
insert(q, E);              (Figure 4.1.1c)
```

The *insert* operation can always be performed, since there is no limit to the number of elements a queue may contain. The *remove* operation, however, can be applied only if the queue is nonempty; there is no way to remove an element from a queue containing no elements. The result of an illegal attempt to remove an element from an empty queue is called ***underflow***. The *empty* operation is, of course, always applicable.

The Queue as an Abstract Data Type

The representation of a queue as an abstract data type is straightforward. We use *eltype* to denote the type of the queue element and parameterize the queue type with *eltype*.

```
abstract typedef <<eltype>> QUEUE(eltype);

abstract empty(q)
QUEUE(eltype) q;
postcondition        empty == (len(q) == 0);

abstract eltype remove(q)
QUEUE(eltype) q;
precondition         empty(q) == FALSE;
postcondition        remove == first(q');
                     q == sub(q', 1, len(q') - 1);

abstract  insert(q, elt)
QUEUE(eltype) q;
eltype elt;
postcondition        q == q' + <elt>;
```

C Implementation of Queues

How shall a queue be represented in C? One idea is to use an array to hold the elements of the queue and to use two variables, *front* and *rear*, to hold the positions within the array of the first and last elements of the queue. We might declare a queue *q* of integers by

```
#define MAXQUEUE 100
struct queue {
        int items[MAXQUEUE];
        int front, rear;
}q;
```

Of course, using an array to hold a queue introduces the possibility of ***overflow*** if the queue should grow larger than the size of the array. Ignoring the possibility of underflow and overflow for the moment, the operation *insert*(*q*, *x*) could be implemented by the statements

```
q.items [++q.rear] = x;
```

and the operation x = *remove*(*q*) could be implemented by

```
x = q.items [q.front++];
```

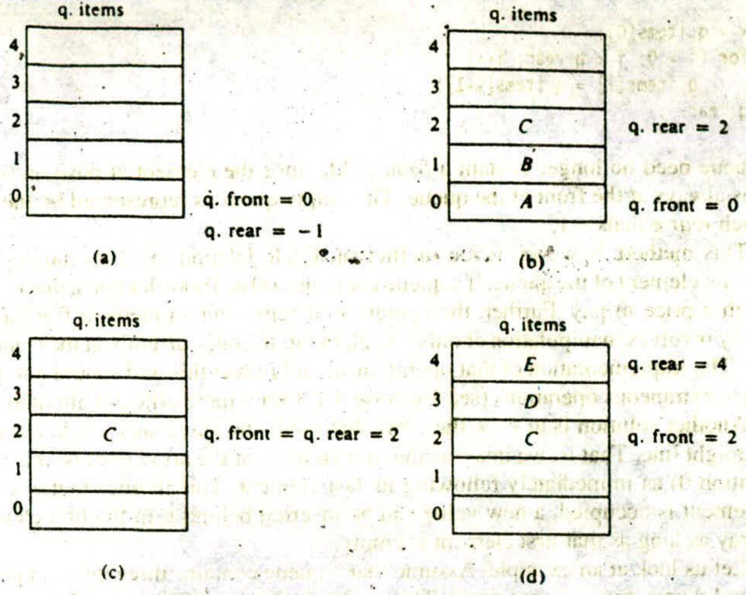


Figure 4.1.2

Initially, *q.rear* is set to −1, and *q.front* is set to 0. The queue is empty whenever *q.rear* < *q.front*. The number of elements in the queue at any time is equal to the value of *q.rear* − *q.front* + 1.

Let us examine what might happen under this representation. Figure 4.1.2 illustrates an array of five elements used to represent a queue (that is, *MAXQUEUE* equals 5). Initially (Figure 4.1.2a), the queue is empty. In Figure 4.1.2b items *A*, *B*, and *C* have been inserted. In Figure 4.1.2c two items have been deleted, and in Figure 4.1.2d two new items, *D* and *E*, have been inserted. The value of *q.front* is 2, and the value of *q.rear* is 4, so that there are only 4 − 2 + 1 = 3 elements in the queue. Since the array contains five elements, there should be room for the queue to expand without the worry of overflow.

However, to insert *F* into the queue, *q.rear* must be increased by 1 to 5 and *q.items*[5] must be set to the value *F*. But *q.items* is an array of only five elements, so that the insertion cannot be made. It is possible to reach the absurd situation where the queue is empty, yet no new element can be inserted (see if you can come up with a sequence of insertions and deletions to reach that situation). Clearly, the array representation outlined in the foregoing is unacceptable.

One solution is to modify the *remove* operation so that when an item is deleted the entire queue is shifted to the beginning of the array. The operation *x* = *remove*(*q*) would then be modified (again, ignoring the possibility of underflow) to

```
x = q.items[0];
for (i = 0; i < q.rear; i++)
     q.items[i] = q.items[i+1];
q.rear--;
```

The queue need no longer contain a *front* field, since the element at position 0 of the array is always at the front of the queue. The empty queue is represented by the queue in which *rear* equals −1.

This method, however, is too inefficient. Each deletion involves moving every remaining element of the queue. If a queue contains 500 or 1000 elements, this is clearly too high a price to pay. Further, the operation of removing an element from a queue logically involves manipulation of only one element: the one currently at the front of the queue. The implementation of that operation should reflect this and should not involve a host of extraneous operations (see Exercise 4.1.3 for a more efficient alternative).

Another solution is to view the array that holds the queue as a circle rather than as a straight line. That is, we imagine the first element of the array (that is, the element at position 0) as immediately following its last element. This implies that even if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty.

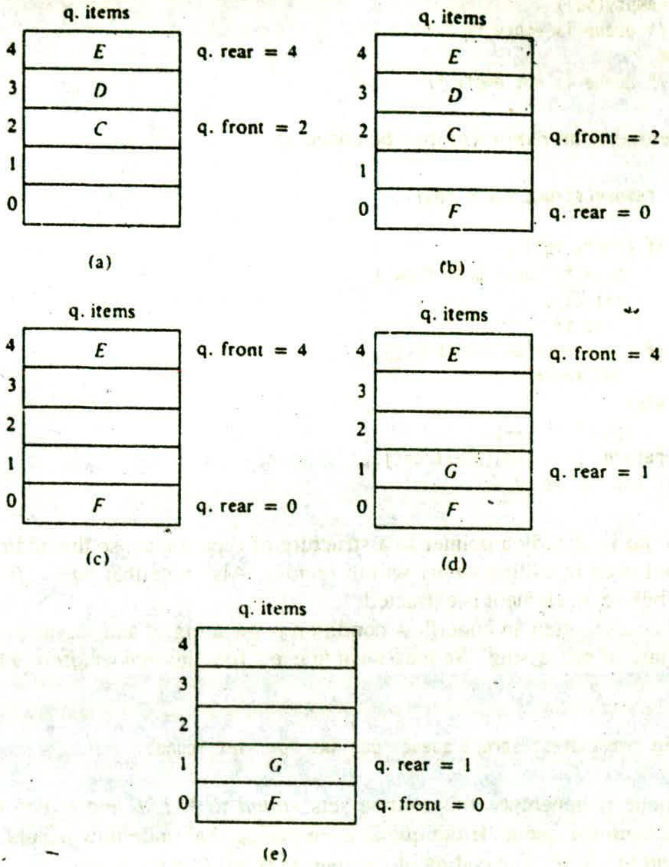
Let us look at an example. Assume that a queue contains three items in positions 2, 3, and 4 of a five element array. This is the situation of Figure 4.1.2d reproduced as Figure 4.1.3a. Although the array is not full, its last element is occupied. If item *F* is now inserted into the queue, it can be placed in position 0 of the array, as shown in Figure 4.1.3b. The first item of the queue is in *q.items*[2], which is followed in the queue by *q.items*[3], *q.items*[4] and *q.items*[0]. Figure 4.1.3c, d, and e show the status of the queue as first two items *C* and *D* are deleted, then *G* is inserted, and finally *E* is deleted.

Unfortunately, it is difficult under this representation to determine when the queue is empty. The condition *q.rear* < *q.front* is no longer valid as a test for the empty queue, since Figure 4.1.3b, c, and d all illustrate situations in which the condition is true yet the queue is not empty.

One way of solving this problem is to establish the convention that the value of *q.front* is the array index immediately preceding the first element of the queue rather than the index of the first element itself. Thus since *q.rear* is the index of the last element of the queue, the condition *q.front* == *q.rear* implies that the queue is empty. A queue of integers may therefore be declared and initialized by

```
#define MAXQUEUE 100
struct queue {
   int items[MAXQUEUE];
   int front, rear;
};
struct queue q;
q.front = q.rear = MAXQUEUE-1;
```

Note that *q.front* and *q.rear* are initialized to the last index of the array, rather than to −1 or 0, because the last element of the array immediately precedes the first one



(a)

(b)

(c)

(d)

(e)

Figure 4.1.3

within the queue under this representation. Since *q.rear* equals *q.front*, the queue is initially empty.

The *empty* function may be coded as

```
int empty(struct queue *pq)
{
    return ((pq->front == pq->rear) ? TRUE : FALSE);
} /* end empty */
```

Once this function exists, a test for the empty queue is implemented by the statement

```
if (empty(&q))
   /* queue is empty */
else
   /* queue is not empty */
```

The operation *remove(q)* may be coded as

```
int remove(struct queue *pq)
{
   if (empty(pq)) {
       printf("queue underflow");
       exit(1);
   } /* end if */
   if (pq->front == MAXQUEUE-1)
       pq->front = 0;
   else
      (pq->front)++;
   return (pq->items[pq->front]);
} /* end remove */
```

Note that *pq* is already a pointer to a structure of type *queue*, so the address operator "&" is not used in calling *empty* within *remove*. Also note that *pq*−> *front* must be updated before an element is extracted.

Of course, often an underflow condition is meaningful and serves as a signal for a new phase of processing. We may wish to use a function *remvandtest*, whose header is

```
void remvandtest(struct queue *pq, int *px, int *pund)
```

If the queue is nonempty, this routine sets **pund* to *FALSE* and **px* to the element removed from the queue. If the queue is empty, so that underflow occurs, the routine sets **pund* to *TRUE*. The coding of the routine is left to the reader.

insert Operation

The *insert* operation involves testing for overflow, which occurs when the entire array is occupied by items of the queue and an attempt is made to insert yet another element into the queue. For example, consider the queue of Figure 4.1.4a. There are three elements in the queue: *C*, *D*, and *E* in *q.items*[2], *q.items*[3], and *q.items*[4], respectively. Since the last item of the queue occupies *q.items*[4], *q.rear* equals 4. Since the first element of the queue is in *q.items*[2], *q.front* equals 1. In Figure 4.1.4b and c, items *F* and *G* are inserted into the queue. At that point, the array is full and an attempt to perform any more insertions causes an overflow. But this is indicated by the fact that *q.front* equals *q.rear*, which is precisely the indication for underflow. It seems that there is no way to distinguish between the empty queue and the full queue under this implementation. Such a situation is clearly unsatisfactory.

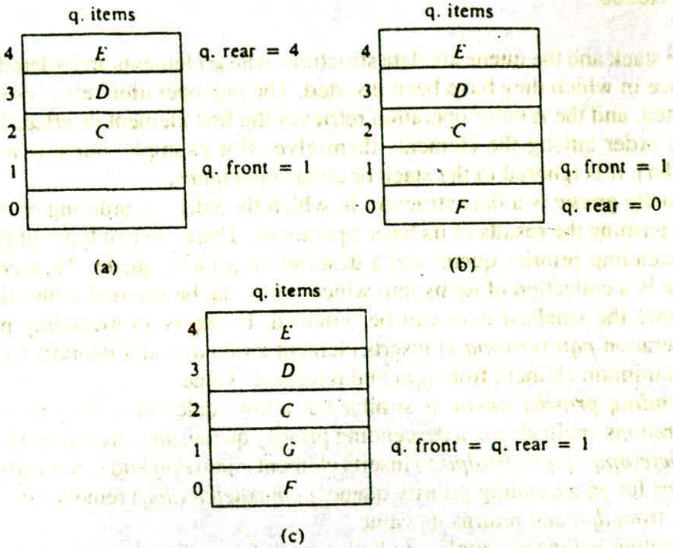


Figure 4.1.4

One solution is to sacrifice one element of the array and to allow a queue to grow only as large as one less than the size of the array. Thus, if an array of 100 elements is declared as a queue, the queue may contain up to 99 elements. An attempt to insert a hundredth element into the queue causes an overflow. The *insert* routine may then be written as follows:

```
void insert(struct queue *pq, int x)
{
   /* make room for new element */
   if (pq->rear == MAXQUEUE-1)
      pq->rear = 0;
   else
      (pq->rear)++;
   /* check for overflow */
   if (pq->rear == pq->front) {
      printf("queue overflow");
      exit(1);
   } /* end if */
   pq->items[pq->rear] = x;
   return;
} /* end insert */
```

The test for overflow in *insert* occurs after *pq*→*rear* has been adjusted, whereas the test for underflow in *remove* occurs immediately upon entering the routine, before *pq*→*front* is updated.

Priority Queue

Both the stack and the queue are data structures whose elements are ordered based on the sequence in which they have been inserted. The *pop* operation retrieves the last element inserted, and the *remove* operation retrieves the first element inserted. If there is an intrinsic order among the elements themselves (for example, numeric order or alphabetic order), it is ignored in the stack or queue operations.

The ***priority queue*** is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations. There are two types of priority queues: an ascending priority queue and a descending priority queue. An ***ascending priority queue*** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If *apq* is an ascending priority queue, the operation *pqinsert(apq,x)* inserts element *x* into *apq* and *pqmindelete(apq)* removes the minimum element from *apq* and returns its value.

A ***descending priority queue*** is similar but allows deletion of only the *largest* item. The operations applicable to a descending priority queue, *dpq*, are *pqinsert(dpq,x)* and *pqmaxdelete(dpq)*. *pqinsert(dpq,x)* inserts element *x* into *dpq* and is logically identical to *pqinsert* for an ascending priority queue. *pqmaxdelete(dpq)* removes the maximum element from *dpq* and returns its value.

The operation *empty(pq)* applies to both types of priority queue and determines whether a priority queue is empty. *pqmindelete* or *pqmaxdelete* can only be applied to a nonempty priority queue [that is, if *empty(pq)* is *FALSE*].

Once *pqmindelete* has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest, and so on. Thus the operation successively retrieves elements of a priority queue in ascending order. (However, if a small element is inserted after several deletions, the next retrieval will return that small element, which may be smaller than a previously retrieved element.) Similarly, *pqmaxdelete* retrieves elements of a descending priority queue in descending order. This explains the designation of a priority queue as either ascending or descending.

The elements of a priority queue need not be numbers or characters that can be compared directly. They may be complex structures that are ordered on one or several fields. For example, telephone-book listings consist of last names, first names, addresses, and phone numbers and are ordered by last name.

Sometimes the field on which the elements of a priority queue are ordered is not even part of the elements themselves; it may be a special, external value used specifically for the purpose of ordering the priority queue. For example, a stack may be viewed as a descending priority queue whose elements are ordered by time of insertion. The element that was inserted last has the greatest insertion-time value and is the only item that can be retrieved. A queue may similarly be viewed as an ascending priority queue whose elements are ordered by time of insertion. In both cases the time of insertion is not part of the elements themselves but is used to order the priority queue.

We leave as an exercise for the reader the development of an ADT specification for a priority queue. We now look at implementation considerations.

Array Implementation of a Priority Queue

As we have seen, a stack and a queue can be implemented in an array so that each insertion or deletion involves accessing only a single element of the array. Unfortunately, this is not possible for a priority queue.

Suppose that the *n* elements of a priority queue *pq* are maintained in positions 0 to $n - 1$ of an array *pq.items* of size *maxpq*, and suppose that *pq.rear* equals the first empty array position, *n*. Then *pqinsert(pq, x)* would seem to be a fairly straightforward operation:

```
if (pq.rear >= maxpq) {
   printf("priority queue overflow");
   exit(1);
} /* end if */
pq.items[pq.rear] = x;
pq.rear++;
```

Note that under this insertion method the elements of the priority queue are not kept ordered in the array.

As long as only insertions take place, this implementation works well. Suppose, however, that we attempt the operation *pqmindelete(pq)* on an ascending priority queue. This raises two issues. First, to locate the smallest element, every element of the array from *pq.items*[0] through *pq.items*[*pq.rear* − 1] must be examined. Therefore a deletion requires accessing every element of the priority queue.

Second, how can an element in the middle of the array be deleted? Stack and queue deletions involve removal of an item from one of the two ends and do not require any searching. Priority queue deletion under this implementation requires both searching for the element to be deleted and removal of an element in the middle of an array.

There are several solutions to this problem, none of them entirely satisfactory:

1. A special "empty" indicator can be placed into a deleted position. This indicator can be a value that is invalid as an element (for example, −1 in a priority queue of nonnegative numbers), or a separate field can be contained in each array element to indicate whether it is empty. Insertion proceeds as before, but when *pq.rear* reaches *maxpq* the array elements are compacted into the front of the array and *pq.rear* is reset to one more than the number of elements. There are several disadvantages to this approach. First, the search process to locate the maximum or minimum element must examine all the deleted array positions in addition to the actual priority queue elements. If many items have been deleted but no compaction has yet taken place, the deletion operation accesses many more array elements than exist in the priority queue. Second, once in a while insertion requires accessing every single position of the array, as it runs out of room and begins compaction.
2. The deletion operation labels a position empty as in the previous solution, but insertion is modified to insert a new item in the first "empty" position. Insertion then

involves accessing every array element up to the first one that has been deleted. This decreased efficiency of insertion is a major drawback to this solution.

3. Each deletion can compact the array by shifting all elements past the deleted element by one position and then decrementing *pq.rear* by 1. Insertion remains unchanged. On the average, half of all priority queue elements are shifted for each deletion, so that deletion becomes quite inefficient. A slightly better alternative is to shift either all preceding elements forward or all succeeding elements backward, depending on which group is smaller. This would require maintaining both *front* and *rear* indicators and treating the array as a circular structure, as we did for the queue.
4. Instead of maintaining the priority queue as an unordered array, maintain it as an ordered, circular array as follows:

```
#define MAXPQ 100
struct pqueue{
     int items[MAXPQ];
     int front, rear;
}
struct pqueue pq;
```

pq.front is the position of the smallest element. *pq.rear* is 1 greater than the position of the largest. Deletion involves merely increasing *pq.front* (for the ascending queue) or decreasing *pq.rear* (for a descending queue). However, insertion requires locating the proper position of the new element and shifting the preceding or succeeding elements (again, the technique of shifting whichever group is smaller is helpful). This method moves the work of searching and shifting from the deletion operation to the insertion operation. However, since the array is ordered, the search for the position of the new element in an ordered array is only half as expensive on the average as finding the maximum or minimum of the unordered array, and a binary search might be used to reduce the cost even more. Other techniques that involve leaving gaps in the array between elements of the priority queue to allow for subsequent insertions are also possible.

We leave the C implementations of *pqinsert*, *pqmindelete*, and *pqmaxdelete* for the array representation of a priority queue as exercises for the reader. Searching ordered and unordered arrays is discussed further in Section 7.1. In general, using an array is not an efficient method for implementing a priority queue. More efficient implementations are examined in the next section and in Sections 6.3 and 7.3.

EXERCISES

4.1.1. Write the function *remvandtest(pq, px, pund)* which sets **pund* to *FALSE* and **px* to the item removed from a nonempty queue **pq* and sets **pund* to *TRUE* if the queue is empty.

4.1.2. What set of conditions is necessary and sufficient for a sequence of *insert* and *remove* operations on a single empty queue to leave the queue empty without causing underflow? What set of conditions is necessary and sufficient for such a sequence to leave a nonempty queue unchanged?

4.1.3. If an array holding a queue is not considered circular, the text suggests that each *remove* operation must shift down every remaining element of a queue. An alternative method is to postpone shifting until *rear* equals the last index of the array. When that situation occurs and an attempt is made to insert an element into the queue, the entire queue is shifted down, so that the first element of the queue is in position 0 of the array. What are the advantages of this method over performing a shift at each *remove* operation? What are the disadvantages? Rewrite the routines *remove*, *insert*, and *empty* using this method.

4.1.4. Show how a sequence of insertions and removals from a queue represented by a linear array can cause overflow to occur upon an attempt to insert an element into an empty queue.

4.1.5. We can avoid sacrificing one element of a queue if a field *qempty* is added to the queue representation. Show how this can be done and rewrite the queue manipulation routines under that representation.

4.1.6. How would you implement a queue of stacks? A stack of queues? A queue of queues? Write routines to implement the appropriate operations for each of these data structures.

4.1.7. Show how to implement a queue of integers in C by using an array *queue*[100], where *queue*[0] is used to indicate the front of the queue, *queue*[1] is used to indicate its rear, and *queue*[2] through *queue*[99] are used to contain the queue elements. Show how to initialize such an array to represent the empty queue and write routines *remove*, *insert* and *empty* for such an implementation.

4.1.8. Show how to implement a queue in C in which each item consists of a variable number of integers.

4.1.9. A ***deque*** is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends of a deque *left* and *right*. How can a deque be represented as a C array? Write four C routines,

```
remvleft, remvright, insrtleft, insrtright
```

to remove and insert elements at the left and right ends of a deque. Make sure that the routines work properly for the empty deque and that they detect overflow and underflow.

4.1.10. Define an ***input-restricted deque*** as a deque (see Exercise 4.1.9) for which only the operations *remvleft*, *remvright*, and *insrtleft* are valid, and an ***output-restricted deque*** as a deque for which only the operations *remvleft*, *insrtleft*, and *insrtright* are valid. Show how each of these can be used to represent both a stack and a queue.

4.1.11. The Scratchemup Parking Garage contains a single lane that holds up to ten cars. Cars arrive at the south end of the garage and leave from the north end. If a customer arrives to pick up a car that is not the northernmost, all cars to the north of the car are moved out, the car is driven out, and the other cars are restored in the same order that they were in originally. Whenever a car leaves, all cars to the south are moved forward so that at all times all the empty spaces are in the south part of the garage. Write a program that reads a group of input lines. Each line contains an 'A' for arrival or a

'D' for departure, and a license plate number. Cars are assumed to arrive and depart in the order specified by the input. The program should print a message each time that a car arrives or departs. When a car arrives, the message should specify whether or not there is room for the car in the garage. If there is no room for a car, the car waits until there is room or until a departure line is read for the car. When room becomes available, another message should be printed. When a car departs, the message should include the number of times the car was moved within the garage, including the departure itself but not the arrival. This number is 0 if the car departs from the waiting line.

4.1.12. Develop an ADT specification for a priority queue.

4.1.13. Implement an ascending priority queue and its operations, *pqinsert*, *pqmindelete*, and *empty*, using each of the four methods presented in the text.

4.1.14. Show how to sort a set of input numbers using a priority queue and the operations *pqinsert*, *pqmindelete*, and *empty*.

4.1.15. Implement a C++ class for a queue using the sequential representation.

4.2 LINKED LISTS

What are the drawbacks of using sequential storage to represent stacks and queues? One major drawback is that a fixed amount of storage remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all. Further, no more than that fixed amount of storage may be allocated, thus introducing the possibility of overflow.

Assume that a program uses two stacks implemented in two separate arrays, *s1.items* and *s2.items*. Further, assume that each of these arrays has 100 elements. Then despite the fact that 200 elements are available for the two stacks, neither can grow beyond 100 items. Even if the first stack contains only 25 items, the second cannot contain more than 100.

One solution to this problem is to allocate a single array *items* of 200 elements. The first stack occupies *items*[0], *items*[1], ..., *items*[*top*1], while the second stack is allocated from the other end of the array, occupying *items*[199], *items*[198], ..., *items*[*top*2]. Thus when one stack is not occupying storage the other stack can use that storage. Of course, two distinct sets of *pop*, *push*, and *empty* routines are necessary for the two stacks, since one grows by incrementing *top*1, while the other grows by decrementing *top*2.

Unfortunately, although such a scheme allows two stacks to share a common area, no such simple solution exists for three or more stacks or even for two queues. Instead, one must keep track of the tops and bottoms (or fronts and rears) of all the structures sharing a single large array. Each time that the growth of one structure is about to impinge on the storage currently being used by another, neighboring structures must be shifted within the single array to allow for the growth.

In a sequential representation, the items of a stack or queue are implicitly ordered by the sequential order of storage. Thus, if *q.items*[*x*] represents an element of a queue, the next element will be *q.items*[*x* + 1] (or if *x* equals MAXQUEUE − 1, *q.items*[0]). Suppose that the items of a stack or a queue were explicitly ordered, that is, each item

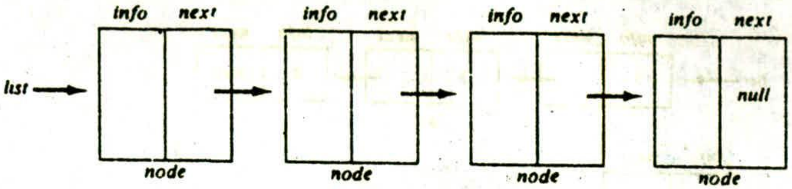


Figure 4.2.1 Linear linked list.

contained within itself the address of the next item. Such an explicit ordering gives rise to a data structure pictured in Figure 4.2.1, which is known as a ***linear linked list***. Each item in the list is called a ***node*** and contains two fields, an ***information*** field and a ***next address*** field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a ***pointer***. The entire linked list is accessed from an external pointer *list* that points to (contains the address of) the first node in the list. (By an "external" pointer, we mean one that is not included within a node. Rather its value can be accessed directly by referencing a variable.) The next address field of the last node in the list contains a special value, known as *null*, which is not a valid address. This ***null pointer*** is used to signal the end of a list.

The list with no nodes on it is called the ***empty list*** or the ***null list***. The value of the external pointer *list* to such a list is the null pointer. A list can be initialized to the empty list by the operation *list* = *null*.

We now introduce some notation for use in algorithms (but not in C programs). If p is a pointer to a node, *node*(p) refers to the node pointed to by p, *info*(p) refers to the information portion of that node, and *next*(p) refers to the next address portion and is therefore a pointer. Thus, if *next*(p) is not *null*, *info*(*next*(p)) refers to the information portion of the node that follows *node*(p) in the list.

Before proceeding with further discussion of linked lists, we should mention that we are presenting them primarily as a data structure (that is, an implementation method) rather than as a data type (that is, a logical structure with precisely defined primitive operations). We therefore do not present an ADT specification for linked lists here. In Section 9.1 we discuss lists as abstract structures and present some primitive operations for them.

In this section, we present the concept of a linked list and show how it is used. In the next section, we show how linked lists can be implemented in C.

Inserting and Removing Nodes from a List

A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and removed. The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

For example, suppose that we are given a list of integers, as illustrated in Figure 4.2.2a, and we desire to add the integer 6 to the front of that list. That is, we wish to

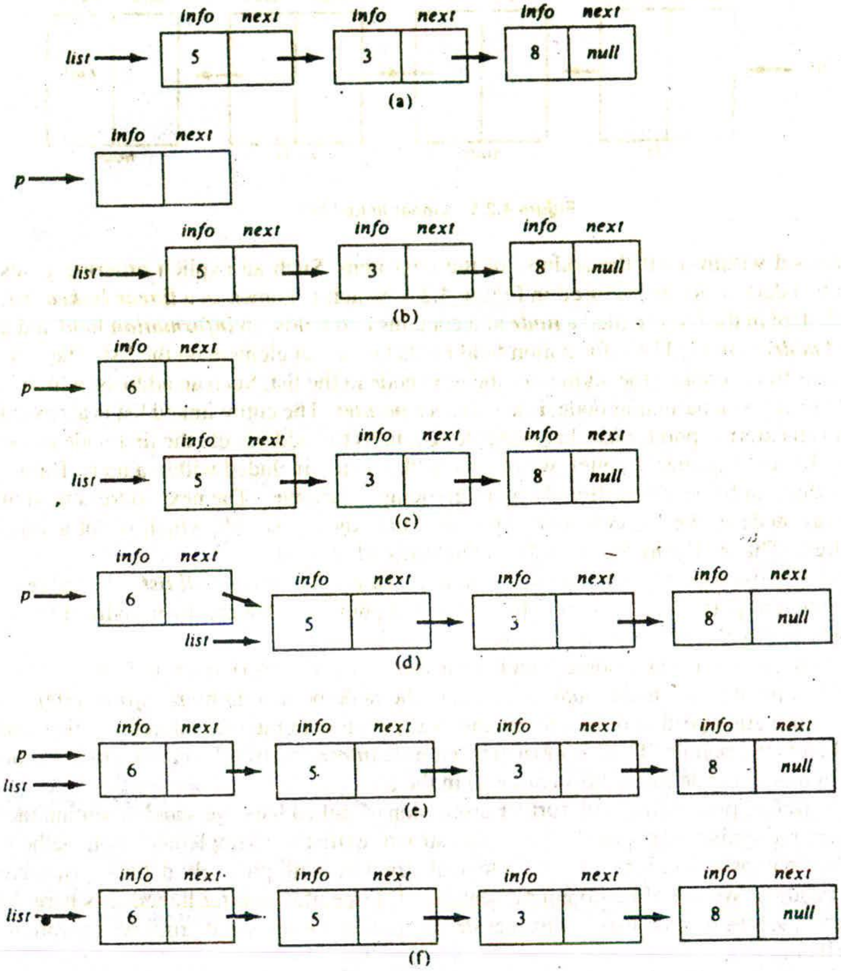


Figure 4.2.2 Adding an element to the front of a list.

change the list so that it appears as in Figure 4.2.2f. The first step is to obtain a node in which to house the additional integer. If a list is to grow and shrink, there must be some mechanism for obtaining empty nodes to be added onto the list. Note that, unlike an array, a list does not come with a presupplied set of storage locations into which elements can be placed.

Let us assume the existence of a mechanism for obtaining empty nodes. The operation

```
    p = getnode();
```

obtains an empty node and sets the contents of a variable named *p* to the address of that node. The value of *p* is then a pointer to this newly allocated node. Figure 4.2.2b illustrates the list and the new node after performing the *getnode* operation. The details of how this operation works will be explained shortly.

The next step is to insert the integer 6 into the *info* portion of the newly allocated node. This is done by the operation

```
info(p) = 6;
```

The result of this operation is illustrated in Figure 4.2.2c.

After setting the *info* portion of *node*(*p*), it is necessary to set the *next* portion of that node. Since *node*(*p*) is to be inserted at the front of the list, the node that follows should be the current first node on the list. Since the variable *list* contains the address of that first node, *node*(*p*) can be added to the list by performing the operation

```
next(p) = list;
```

This operation places the value of *list* (which is the address of the first node on the list) into the *next* field of *node*(*p*). Figure 4.2.2d illustrates the result of this operation.

At this point, *p* points to the list with the additional item included. However, since *list* is the external pointer to the desired list, its value must be modified to the address of the new first node of the list. This can be done by performing the operation

```
list = p;
```

which changes the value of *list* to the value of *p*. Figure 4.2.2e illustrates the result of this operation. Note that Figure 4.2.2e and f are identical except that the value of *p* is not shown in Figure 4.2.2f. This is because *p* is used as an auxiliary variable during the process of modifying the list but its value is irrelevant to the status of the list before and after the process. Once the foregoing operations have been performed, the value of *p* may be changed without affecting the list.

Putting all the steps together, we have an algorithm for adding the integer 6 to the front of the list *list*:

```
p = getnode();
info(p) = 6;
next(p) = list;
list = p;
```

The algorithm can obviously be generalized so that it adds any object *x* to the front of a list *list* by replacing the operation *info*(*p*) = 6 with *info*(*p*) = *x*. Convince yourself that the algorithm works correctly, even if the list is initially empty (*list* == *null*).

Figure 4.2.3 illustrates the process of removing the first node of a nonempty list and storing the value of its *info* field into a variable *x*. The initial configuration is shown in Figure 4.2.3a, and the final configuration is shown in Figure 4.2.3f. The process itself

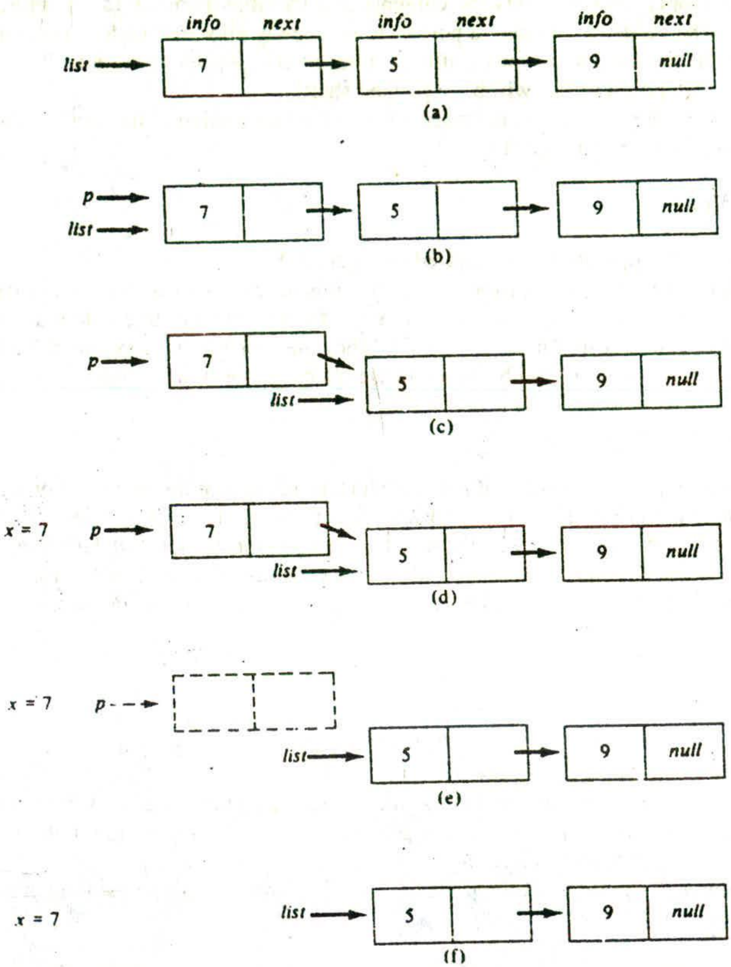


Figure 4.2.3 Removing a node from the front of a list.

is almost the exact opposite of the process to add a node to the front of a list. To obtain Figure 4.2.3d from Figure 4.2.3a, the following operations (whose actions should be clear) are performed:

```
p = list;            (Figure 4.2.3b)
list = next(p);      (Figure 4.2.3c)
x = info(p);         (Figure 4.2.3d)
```

At this point, the algorithm has accomplished what it was supposed to do: the first node has been removed from *list*, and *x* has been set to the desired value. However, the algorithm is not yet complete. In Figure 4.2.3d, *p* still points to the node that was formerly first on the list. However, that node is currently useless because it is no longer on the list and its information has been stored in *x*. (The node is not considered to be on the list despite the fact that *next*(*p*) points to a node on the list, since there is no way to reach *node*(*p*) from the external pointer *list*.)

The variable *p* is used as an auxiliary variable during the process of removing the first node from the list. The starting and ending configurations of the list make no reference to *p*. It is therefore reasonable to expect that *p* will be used for some other purpose in a short while after this operation has been performed. But once the value of *p* is changed there is no way to access the node at all, since neither an external pointer nor a *next* field contains its address. Therefore the node is currently useless and cannot be reused, yet it is taking up valuable storage.

It would be desirable to have some mechanism for making *node*(*p*) available for reuse even if the value of the pointer *p* is changed. The operation that does this is

```
    freenode(p);            (Figure 4.2.3e)
```

Once this operation has been performed, it becomes illegal to reference *node*(*p*), since the node is no longer allocated. Since the value of *p* is a pointer to a node that has been freed, any reference to that value is also illegal.

However, the node might be reallocated and a pointer to it reassigned to *p* by the operation $p = getnode()$. Note that we say that the node "might be" reallocated, since the *getnode* operation returns a pointer to some newly allocated node. There is no guarantee that this new node is the same as the one that has just been freed.

Another way of thinking of *getnode* and *freenode* is that *getnode* creates a new node, whereas *freenode* destroys a node. Under this view, nodes are not used and reused but are rather created and destroyed. We shall say more about the two operations *getnode* and *freenode* and about the concepts they represent in a moment, but first we make the following interesting observation.

Linked Implementation of Stacks

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack. In both cases, a new item is added as the only immediately accessible item in a collection. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similarly, the operation of removing the first element from a linked list is analogous to popping a stack. In both cases the only immediately accessible item of a collection is removed from that collection, and the next item becomes immediately accessible.

Thus we have discovered another way of implementing a stack. A stack may be represented by a linear linked list. The first node of the list is the top of the stack. If an external pointer *s* points to such a linked list, the operation *push*(*s*,*x*) may be implemented by

```
        p = getnode();
        info(p) = x;
        next(p) = s;
        s = p;
}
```

The operation *empty(s)* is merely a test of whether *s* equals *null*. The operation x = *pop(s)* removes the first node from a nonempty list and signals underflow if the list is empty:

```
if (empty(s)) {
   printf('stack underflow');
   exit(1);
}
else {
   p = s;
   s = next(p);
   x = info(p);
   freenode(p);
}  /* end if */
```

Figure 4.2.4a illustrates a stack implemented as a linked list, and Figure 4.2.4b illustrates the same stack after another element has been pushed onto it.

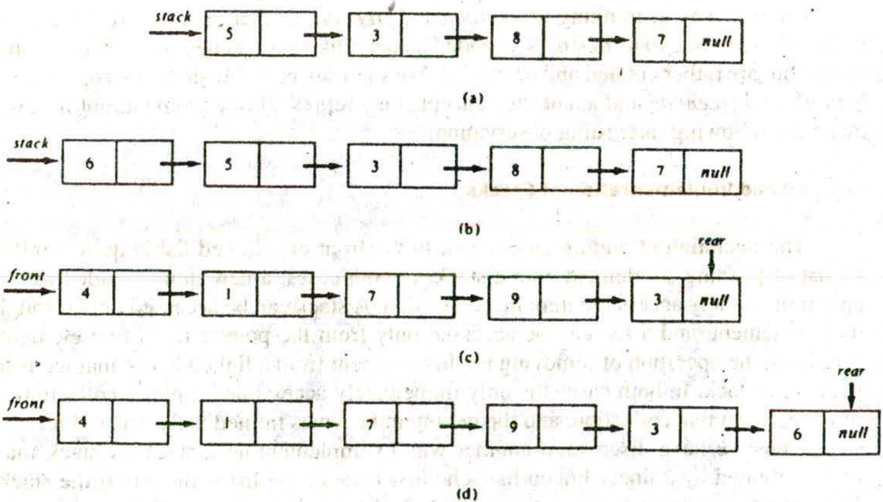


Figure 4.2.4 Stack and queue as linked lists.

The advantage of the list implementation of stacks is that all stacks being used by a program can share the same available list. When any stack needs a node, it can obtain it from the single available list. When any stack no longer needs a node, it returns the node to that same available list. As long as the total amount of space needed by all the stacks at any one time is less than the amount of space initially available to them all, each stack is able to grow and shrink to any size. No space has been preallocated to any single stack and no stack is using space that it does not need. Furthermore, other data structures such as queues may also share the same set of nodes.

getnode and *freenode* Operations

We now return to a discussion of the *getnode* and *freenode* operations. In an abstract, idealized world it is possible to postulate an infinite number of unused nodes available for use by abstract algorithms. The *getnode* operation finds one such node and makes it available to the algorithm. Alternatively, the *getnode* operation may be regarded as a machine that manufactures nodes and never breaks down. Thus, each time that *getnode* is invoked, it presents its caller with a brand new node, different from all the nodes previously in use.

In such an ideal world, the *freenode* operation would be unnecessary to make a node available for reuse. Why use an old second-hand node when a simple call to *getnode* can produce a new, never-before-used node? The only harm that an unused node can do is to reduce the number of nodes that can possibly be used, but if an infinite supply of nodes is available, such a reduction is meaningless. Therefore there is no reason to reuse a node.

Unfortunately, we live in a real world. Computers do not have an infinite amount of storage and cannot manufacture more storage for immediate utilization (at least, not yet). Therefore there are a finite number of nodes available and it is impossible to use more than that number at any given instant. If it is desired to use more than that number over a given period of time, some nodes must be reused. The function of *freenode* is to make a node that is no longer being used in its current context available for reuse in a different context.

We might think of a finite pool of empty nodes existing initially. This pool cannot be accessed by the programmer except through the *getnode* and *freenode* operations. *getnode* removes a node from the pool, whereas *freenode* returns a node to the pool. Since any unused node is as good as any other, it makes no difference which node is retrieved by *getnode* or where within the pool a node is placed by *freenode*.

The most natural form for this pool to take is that of a linked list acting as a stack. The list is linked together by the *next* field in each node. The *getnode* operation removes the first node from this list and makes it available for use. The *freenode* operation adds a node to the front of the list, making it available for reallocation by the next *getnode*. The list of available nodes is called the ***available list***.

What happens when the available list is empty? This means that all nodes are currently in use and it is impossible to allocate any more. If a program calls on *getnode* when the available list is empty, the amount of storage assigned for that program's data structures is too small. Therefore, overflow occurs. This is similar to the situation of a stack implemented in an array overflowing the array bounds.

As long as data structures are abstract, theoretical concepts in a world of infinite space, there is no possibility of overflow. It is only when they are implemented as real objects in a finite area that the possibility of overflow arises.

Assume that an external pointer *avail* points to a list of available nodes. Then the operation

```
p = getnode();
```

is implemented as follows:

```
if (avail == null) {
   printf("overflow");
   exit(1);
}
p = avail;
avail = next(avail);
```

Since the possibility of overflow is accounted for in the *getnode* operation, it need not be mentioned in the list implementation of *push*. If a stack is about to overflow all available nodes, the statement p = *getnode*() within the *push* operation results in an overflow.

The implementation of *freenode*(p) is straightforward:

```
next(p) = avail;
avail = p;
```

Linked Implementation of Queues

Let us now examine how to represent a queue as a linked list. Recall that items are deleted from the front of a queue and inserted at the rear. Let a pointer to the first element of a list represent the front of the queue. Another pointer to the last element of the list represents the rear of the queue, as shown in Figure 4.2.4c. Figure 4.2.4d illustrates the same queue after a new item has been inserted.

Under the list representation, a queue q consists of a list and two pointers, *q.front* and *q.rear*. The operations *empty*(q) and x = *remove*(q) are completely analogous to *empty*(s) and x = *pop*(s), with the pointer *q.front* replacing s. However, special attention is required when the last element is removed from a queue. In that case, *q.rear* must also be set to *null*, since in an empty queue both *q.front* and *q.rear* must be *null*. The algorithm for x = *remove*(q) is therefore as follows:

```
if (empty(q)) {
   printf("queue underflow");
   exit(1);
}
p = q.front;
x = info(p);
q.front = next(p);
if (q.front == null)
   q.rear = null;
freenode(p);
return(x);
```

The operation *insert(q, x)* is implemented by

```
p = getnode();
info(p) = x;
next(p) = null;
if (q.rear == null)
   q.front = p;
else
   next(q.rear) = p;
q.rear = p;
```

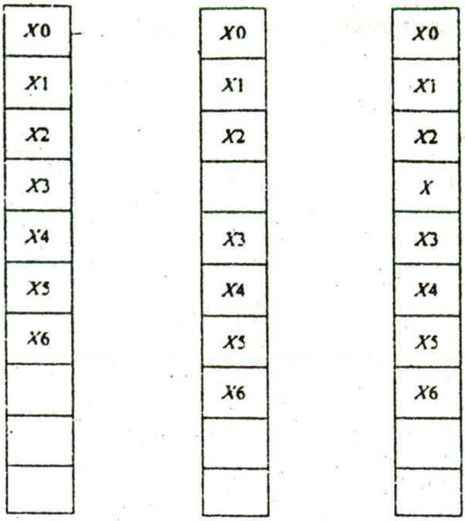
What are the disadvantages of representing a stack or queue by a linked list? Clearly, a node in a linked list occupies more storage than a corresponding element in an array, since two pieces of information per element are necessary in a list node (*info* and *next*), whereas only one piece of information is needed in the array implementation. However, the space used for a list node is usually not twice the space used by an array element, since the elements in such a list usually consist of structures with many subfields. For example, if each element on a stack were a structure occupying ten words, the addition of an eleventh word to contain a pointer increases the space requirement by only 10 percent. Further, it is sometimes possible to compress information and a pointer into a single word so that there is no space degradation.

Another disadvantage is the additional time spent in managing the available list. Each addition and deletion of an element from a stack or a queue involves a corresponding deletion or addition to the available list.

The advantage of using linked lists is that all the stacks and queues of a program have access to the same free list of nodes. Nodes not used by one stack may be used by another, as long as the total number of nodes in use at any one time is not greater than the total number of nodes available.

Linked List as a Data Structure

Linked lists are important not only as a means of implementing stacks and queues but as data structures in their own right. An item is accessed in a linked list by traversing the list from its beginning. An array implementation allows access to the *n*th item in a group using a single operation, whereas a list implementation requires *n* operations.



(a)

Figure 4.2.5a

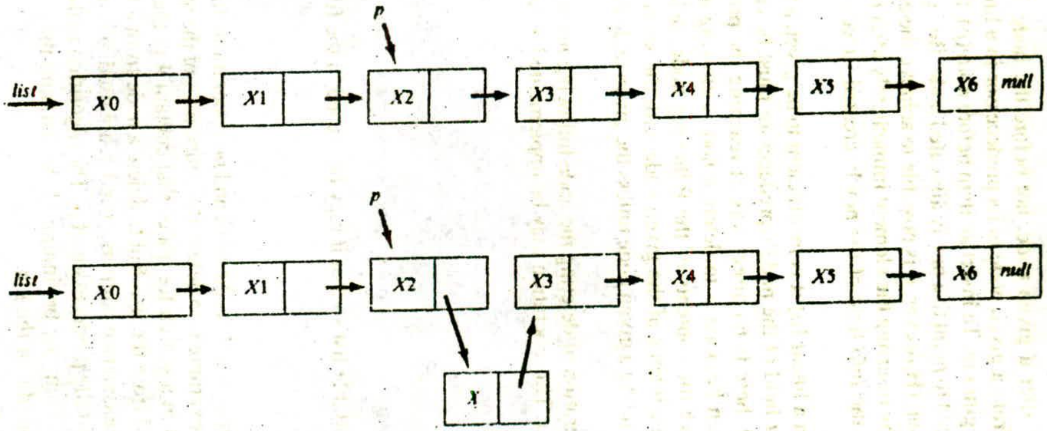
It is necessary to pass through each of the first $n - 1$ elements before reaching the nth element because there is no relation between the memory location occupied by an element of a list and its position within that list.

The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements. For example, suppose that we wished to insert an element x between the third and fourth elements in an array of size 10 that currently contains seven items ($x[0]$ through $x[6]$). Items 6 through 3 must first be moved one slot and the new element inserted in the newly available position 3. This process is illustrated by Figure 4.2.5a. In this case insertion of one item involves moving four items in addition to the insertion itself. If the array contained 500 or 1000 elements, a correspondingly larger number of elements would have to be moved. Similarly, to delete an element from an array without leaving a gap, all the elements beyond the element deleted must be moved one position.

On the other hand, suppose the items are stored as a list. If p points to an element of the list, inserting a new element after *node*(p) involves allocating a node, inserting the information, and adjusting two pointers. The amount of work required is independent of the size of the list. This is illustrated in Figure 4.2.5b.

Let *insafter*(p,x) denote the operation of inserting an item x into a list after a node pointed to by p. This operation is implemented as follows:

```
q = getnode();
info(q) = x;
next(q) = next(p);
next(p) = q;
```



(b)

Figure 4.2.5b

An item can be inserted only after a given node, not before the node. This is because there is no way to proceed from a given node to its predecessor in a linear list without traversing the list from its beginning. To insert an item before *node*(*p*), the *next* field of its predecessor must be changed to point to a newly allocated node. But, given *p*, there is no way to find that predecessor. (However, it is possible to achieve the effect of inserting an element before *node*(*p*) by inserting the element immediately after *node*(*p*) and then interchanging *info*(*p*) with the *info* field of the newly created successor. We leave the details for the reader.)

Similarly, to delete a node from a linear list it is insufficient to be given a pointer to that node. This is because the *next* field of the node's predecessor must be changed to point to the node's successor, and there is no direct way of reaching the predecessor of a given node. The best that can be done is to delete a node following a given node. (However, it is possible to save the contents of the following node, delete the following node, and then replace the contents of the given node with the saved information. This achieves the effect of deleting a given node unless the given node is last in the list.)

Let *delafter*(*p*,*x*) denote the operation of deleting the node following *node*(*p*) and assigning its contents to the variable *x*. This operation may be implemented as follows:

```
q = next(p);
x = info(q);
next(p) = next(q);
freenode(q);
```

The freed node is placed onto the available list so that it may be reused in the future.

Examples of List Operations

We illustrate these two operations, as well as the *push* and *pop* operations for lists, with some simple examples. The first example is to delete all occurrences of the number 4 from a list *list*. The list is traversed in a search for nodes that contain 4 in their *info* fields. Each such node must be deleted from the list. But to delete a node from a list, its predecessor must be known. For this reason two pointers, *p* and *q*, are used. *p* is used to traverse the list, and *q* always points to the predecessor of *p*. The algorithm makes use of the *pop* operation to remove nodes from the beginning of the list, and the *delafter* operation to remove nodes from the middle of the list.

```
q = null;
p = list;
while (p != null)  {
    if (info(p) == 4)
        if (q == null)  {
            /* remove first node of the list */
            x = pop(list);
            p = list;
        }
```

```
            else {
               /* delete the node after q and move up p */
               p = next(p);
               delafter(q, x);
            }  /* end if */
         else {
            /* continue traversing the list */
            q = p;
            p = next(p);
         } /* end if */
      } /* end while */
```

The practice of using two pointers, one following the other, is very common in working with lists. This technique is used in the next example as well. Assume that a list *list* is ordered so that smaller items precede larger ones. Such a list is called an ***ordered list***. It is desired to insert an item *x* into this list in its proper place. The algorithm to do so makes use of the *push* operation to add a node to the front of the list and the *insafter* operation to add a node in the middle of the list:

```
q = null;
for (p = list; p != null && x > info(p); p = next(p))
   q = p;
/* at this point, a node containing x must be inserted */
if (q == null) /* insert x at the head of the list      */
   push(list, x);
else
   insafter(q, x);
```

This is a very common operation and will be denoted by *place*(*list*, *x*).

Let us examine the efficiency of the *place* operation. How many nodes are accessed, on the average, in inserting a new element into an ordered list? Let us assume that the list contains n nodes. Then x can be placed in one of $n + 1$ positions; that is, it can be found to be less than the first element of the list, between the first and the second, . . . between the $(n - 1)$st and the nth, and greater than the nth. If x is less than the first, *place* accesses only the first node of the list (aside from the new node containing x); that is, it immediately determines that $x <$ *info*(*list*) and inserts a node containing x using *push*, If x is between the kth and $(k + 1)$st element, *place* accesses the first k nodes; only after finding x to be less than the contents of the $(k + 1)$st node is x inserted using *insafter*. If x is greater than the nth element, then all n nodes are accessed.

Now suppose that it is equally likely that x is inserted into any one of the $n + 1$ possible positions. (If this is true, we say that the insertion is ***random***.) Then the probability of inserting at any particular position is $1/(n + 1)$. If the element is inserted between the kth and the $(k + 1)$st position, the number of accesses is $k + 1$. If the element is inserted after the nth element, the number of accesses is n. The average number of nodes accessed, A, equals the sum, over all possible insertion positions, of the products of the probability of inserting at a particular position and the number of accesses required to insert an element at that position. Thus

$$A = \left(\frac{1}{n+1}\right) * 1 + \left(\frac{1}{n+1}\right) * 2 + \cdots + \left(\frac{1}{n+1}\right) * (n-1) + \left(\frac{1}{n+1}\right) * n$$
$$+ \left(\frac{1}{n+1}\right) * n$$

or

$$A = \left(\frac{1}{n+1}\right) * (1 + 2 + \cdots + n) + \frac{n}{n+1}$$

Now $1 + 2 + \cdots + n = n * \frac{n+1}{2}$. (This can be proved easily by mathematical induction.) Therefore,

$$A = \left(\frac{1}{n+1}\right) * \left(n * \frac{n+1}{2}\right) + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

When n is large, $n/(n + 1)$ is very close to 1, so that A is approximately $n/2 + 1$ or $(n + 2)/2$. For large n, A is close enough to $n/2$ that we often say that the operation of randomly inserting an element into an ordered list requires approximately $n/2$ node accesses on average.

List Implementation of Priority Queues

An ordered list can be used to represent a priority queue. For an ascending priority queue, insertion (*pqinsert*) is implemented by the *place* operation, which keeps the list ordered, and deletion of the minimum element (*pqmindelete*) is implemented by the *pop* operation, which removes the first element from the list. A descending priority queue can be implemented by keeping the list in descending, rather than ascending, order or by using *remove* to implement *pqmaxdelete*. A priority queue implemented as an ordered linked list requires examining an average of approximately $n/2$ nodes for insertion, but only one node for deletion.

An unordered list may also be used as a priority queue. Such a list requires examining only one node for insertion (by implementing *pqinsert* using *push* or *insert*) but always requires examining n elements for deletion (traverse the entire list to find the minimum or maximum and then delete that node). Thus an ordered list is somewhat more efficient than an unordered list in implementing a priority queue.

The advantage of a list over an array for implementing a priority queue is that no shifting of elements or gaps are necessary in a list. An item can be inserted into a list without moving any other items, whereas this is impossible for an array unless extra space is left empty. We examine other, more efficient implementations of the priority queue in Sections 6.3 and 7.3.

Header Nodes

Sometimes it is desirable to keep an extra node at the front of a list. Such a node does not represent an item in the list and is called a ***header node*** or a ***list header***. The *info* portion of such a header node might be unused, as illustrated in Figure 4.2.6a. More often, the *info* portion of such a node could be used to keep global information about the entire list. For example, Figure 4.2.6b illustrates a list in which the *info* portion of the header node contains the number of nodes (not including the header) in the list. In

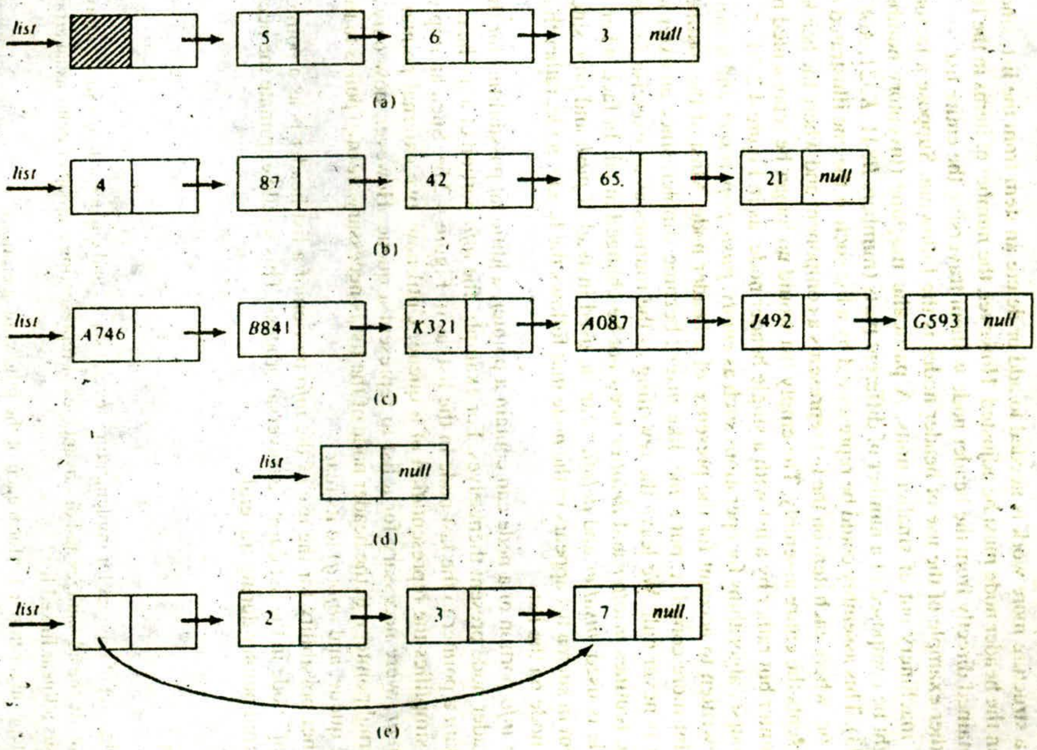


Figure 4.2.6 Lists with header nodes.

such a data structure more work is needed to add or delete an item from the list, since the count in the header node must be adjusted. However, the number of items in the list may be obtained directly from the header node without traversing the entire list.

Another example of the use of header nodes is the following. Suppose a factory assembles machinery out of smaller units. A particular machine (inventory number *A*746) might be composed of a number of different parts (numbers *B*841, *K*321, *A*087, *J*492, *G*593). This assembly could be represented by a list such as the one illustrated in Figure 4.2.6c, where each item on the list represents a component and where the header node represents the entire assembly. The empty list would no longer be represented by the null pointer but rather by a list with a single header node, as in Figure 4.2.6d.

Of course, algorithms for operations such as *empty*, *push*, *pop*, *insert*, and *remove* must be rewritten to account for the presence of a header node. Most of the routines become a bit more complex, but some, like *insert*, become simpler, since an external list pointer is never null. We leave the rewriting of the routines as an exercise for the reader. The routines *insafter* and *delafter* need not be changed at all. In fact, when a header node is used, *insafter* and *delafter* can be used instead of *push* and *pop*, since the first item in such a list appears in the node that follows the header node, rather than in the first node on the list.

If the *info* portion of a node can contain a pointer, additional possibilities for the use of a header node present themselves. For example, the *info* portion of a list header might contain a pointer to the last node in the list, as in Figure 4.2.6e. Such an implementation simplifies the representation of a queue. Until now, two external pointers, *front* and *rear*, were necessary for a list to represent a queue. However, now only a single external pointer *q* to the header node of the list is necessary. *next*(*q*) points to the front of the queue, and *info*(*q*) to its rear.

Another possibility for the use of the *info* portion of a list header is as a pointer to a "current" node in the list during a traversal process. This would eliminate the need for an external pointer during traveral.

EXERCISES

4.2.1. Write a set of routines for implementing several stacks and queues within a single array.

4.2.2. What are the advantages and disadvantages of representing a group of items as an array versus a linear linked list?

4.2.3. Write an algorithm to perform each of the following operations.

- **(a)** Append an element to the end of a list.
- **(b)** Concatenate two lists.
- **(c)** Free all the nodes in a list.
- **(d)** Reverse a list, so that the last element becomes the first, and so on.
- **(e)** Delete the last element from a list.
- **(f)** Delete the *n*th element from a list.
- **(g)** Combine two ordered lists into a single ordered list.
- **(h)** Form a list containing the union of the elements of two lists.
- **(i)** Form a list containing the intersection of the elements of two lists.
- **(j)** Insert an element after the *n*th element of a list.

(k) Delete every second element from a list.
(l) Place the elements of a list in increasing order.
(m) Return the sum of the integers in a list.
(n) Return the number of elements in a list.
(o) Move *node(p)* forward *n* positions in a list.
(p) Make a second copy of a list.

4.2.4. Write algorithms to perform each of the operations of the previous exercise on a group of elements in contiguous positions of an array.

4.2.5. What is the average number of nodes accessed in searching for a particular element in an unordered list? In an ordered list? In an unordered array? In an ordered array?

4.2.6. Write algorithms for *pqinsert* and *pqmindelete* for an ascending priority queue implemented as an unordered list and as an ordered list.

4.2.7. Write algorithms to perform each of the operations in Exercise 4.2.3, assuming that each list contains a header node containing the number of elements in the list.

4.2.8. Write an algorithm that returns a pointer to a node containing element *x* in a list with a header node. The *info* field of the header should contain the pointer that traverses the list.

4.2.9. Modify the C++ stack template implementation given at the end of Section 2.3 to use the pointer representation of stacks.

4.3 LISTS IN C

Array Implementation of Lists

How can linear lists be represented in C? Since a list is simply a collection of nodes, an array of nodes immediately suggests itself. However, the nodes cannot be ordered by the array ordering; each must contain within itself a pointer to its successor. Thus a group of 500 nodes might be declared as an array *node* as follows:

```
#define NUMNODES 500
struct nodetype  {
   int info, next;
};
struct nodetype node[NUMNODES];
```

In this scheme a pointer to a node is represented by an array index. That is, a pointer is an integer between 0 and *NUMNODES* − 1 that references a particular element of the array *node*. The null pointer is represented by the integer −1. Under this implementation, the C expression *node*[*p*] is used to reference *node*(*p*), *info*(*p*) is referenced by *node*[*p*].*info*, and *next*(*p*) is referenced by *node*[*p*].*next*. *null* is represented by −1.

For example, suppose that the variable *list* represents a pointer to a list. If *list* has the value 7, *node*[7] is the first node on the list, and *node*[7].*info* is the first data item on the list. The second node of the list is given by *node*[7].*next*. Suppose that *node*[7].*next* equals 385. Then *node*[385].*info* is the second data item on the list and *node*[385].*next* points to the third node.

The nodes of a list may be scattered throughout the array *node* in any arbitrary order. Each node carries within itself the location of its successor until the last node in the list, whose *next* field contains -1, which is the null pointer. There is no relation between the contents of a node and the pointer to it. The pointer, *p*, to a node merely specifies which element of the array *node* is being referenced; it is *node*[*p*].*info* that represents the information contained within that node.

Figure 4.3.1 illustrates a portion of an array *node* that contains four linked lists. The list *list*1 starts at *node*[16] and contains the integers 3, 7, 14, 6, 5, 37, 12. The nodes that contain these integers in their *info* fields are scattered throughout the array. The *next* field of each node contains the index within the array of the node containing the next element of the list. The last node on the list is *node*[23], which contains the integer 12 in its *info* field and the null pointer (-1) in its *next* field, to indicate that it is last on the list.

Similarly, *list*2 begins at *node*[4] and contains the integers 17 and 26, *list*3 begins at *node*[11] and contains the integers 31, 19, and 32, and *list*4 begins at *node*[3] and contains the integers 1, 18, 13, 11, 4, and 15. The variables *list*1, *list*2, *list*3, and *list*4 are integers representing external pointers to the four lists. Thus, the fact that the variable *list*2 has the value 4 represents the fact that the list to which it points begins at *node*[4].

		info	next
	0	26	−1
	1	11	9
	2	5	15
list4 =	3	1	24
list2 =	4	17	0
	5	13	1
	6		
	7	19	18
	8	14	12
	9	4	21
	10		
list3 =	11	31	7
	12	6	2
	13		
	14		
	15	37	23
list1 =	16	3	20
	17		
	18	32	−1
	19		
	20	7	8
	21	15	−1
	22		
	23	12	−1
	24	18	5
	25		
	26		

Figure 4.3.1 Array of nodes containing four linked lists.

Initially, all nodes are unused, since no lists have yet been formed. Therefore they must all be placed on the available list. If the global variable *avail* is used to point to the available list, we may initially organize that list as follows:

```
avail = 0;
for (i = 0; i < NUMNODES-1; i++)
    node[i].next = i + 1;
node[NUMNODES-1].next = -1;
```

The 500 nodes are initially linked in their natural order, so that *node*[*i*] points to *node*[*i* + 1]. *node*[0] is the first node on the available list, *node*[1] is the second, and so forth. *node*[499] is the last node on the list, since *node*[499].*next* equals −1. There is no reason other than convenience for initially ordering the nodes in this fashion. We could just as well have set *node*[0].*next* to 499, *node*[499].*next* to 1, *node*[1].*next* to 498, and so forth, until *node*[249].*next* is set to 250 and *node*[250].*next* to −1. The important point is that the ordering is explicit within the nodes themselves and is not implied by some other underlying structure.

For the remaining functions in this section, we assume that the variables *node* and *avail* are global and can therefore be used by any routine.

When a node is needed for use in a particular list, it is obtained from the available list. Similarly, when a node is no longer necessary, it is returned to the available list. These two operations are implemented by the C routines *getnode* and *freenode*. *getnode* is a function that removes a node from the available list and returns a pointer to it:

```
int getnode(void)
{
   int p;
   if (avail == -1)  {
      printf("overflow\n");
      exit(1);
   }
   p = avail;
   avail = node[avail].next;
   return(p);
} /* end getnode */
```

If *avail* equals − when this function is called, there are no nodes available. This means that the list structures of a particular program have overflowed the available space.

The function *freenode* accepts a pointer to a node and returns that node to the available list:

```
void freenode(int p)
{
   node[p].next = avail;
   avail = p;
   return;
} /* end freenode */
```

The primitive operations for lists are straightforward C versions of the corresponding algorithms. The routine *insafter* accepts a pointer *p* to a node and an item *x* as parameters. It first ensures that *p* is not null and then inserts *x* into a node following the node pointed to by *p*:

```
void insafter(int p, int x)
{
   int q;
   if (p == -1)  {
      printf("void insertion\n");
      return;
   }
   q = getnode();
   node[q].info = x;
   node[q].next = node[p].next;
   node[p].next = q;
   return;
} /* end insafter */
```

The routine *delafter(p, px)*, called by the statement *delafter(p, &x)*, deletes the node following *node(p)* and stores its contents in *x*:

```
void delafter(int p, int *px)
{
   int q;
   if ((p == -1) || (node[p].next == -1))  {
       printf ("void deletion\n");
       return;
   }
   q = node[p].next;
   *px = node[q].info;
   node[p].next = node[q].next;
   freenode(q);
   return;
} /* end delafter */
```

Before calling *insafter* we must be sure that *p* is not null. Before calling *delafter* we must be sure that neither *p* nor *node*[*p*].*next* is null.

Limitations of the Array Implementation

As we saw in Section 4.2, the notion of a pointer allows us to build and manipulate linked lists of various types. The concept of a pointer introduces the possibility of assembling a collection of building blocks, called nodes, into flexible structures. By altering the values of pointers, nodes can be attached, detached, and reassembled in patterns that grow and shrink as execution of a program progresses.

Under the array implementation, a fixed set of nodes represented by an array is established at the start of execution. A pointer to a node is represented by the relative position of the node within the array. The disadvantage of that approach is twofold. First, the number of nodes that are needed often cannot be predicted when a program is written. Usually, the data with which the program is executed determines the number of nodes necessary. Thus no matter how many elements the array of nodes contains, it is always possible that the program will be executed with input that requires a larger number.

The second disadvantage of the array approach is that whatever number of nodes are declared must remain allocated to the program throughout its execution. For example, if 500 nodes of a given type are declared, the amount of storage required for those 500 nodes is reserved for that purpose. If the program actually uses only 100 or even 10 nodes in its execution the additional nodes are still reserved and their storage cannot be used for any other purpose.

The solution to this problem is to allow nodes that are ***dynamic***, rather than static. That is, when a node is needed, storage is reserved for it, and when it is no longer needed, the storage is released. Thus the storage for nodes that are no longer in use is available for another purpose. Also, no predefined limit on the number of nodes is established. As long as sufficient storage is available to the job as a whole, part of that storage can be reserved for use as a node.

Allocating and Freeing Dynamic Variables

In Sections 1.1, 1.2, and 1.3, we examined pointers in the C language. If x is any object, $\&x$ is a pointer to x. If p is a pointer in C, $*p$ is the object to which p points. We can use C pointers to help implement dynamic linked lists. First, however, we discuss how storage can be allocated and freed dynamically and how dynamic storage is accessed in C.

In C a pointer variable to an integer can be created by the declaration

```
int *p;
```

Once a variable p has been declared as a pointer to a specific type of object, it must be possible to dynamically create an object of that specific type and assign its address to p.

This may be done in C by calling the standard library function *malloc(size)*. *malloc* dynamically allocates a portion of memory of size *size* and returns a pointer to an item of type *char*. Consider the declarations

```
extern char *malloc();
int *pi;
float *pr;
```

The statements

```
pi = (int  *) malloc(sizeof (int));
pr = (float *) malloc(sizeof (float));
```

dynamically create the integer variable **pi* and the float variable **pr*. These variables are called ***dynamic variables***. In executing these statements, the operator ***sizeof*** returns the size, in bytes, of its operand. This is used to maintain machine independence. *malloc* can then create an object of that size. Thus *malloc*(***sizeof***(***int***)) allocates storage for an integer, whereas *malloc*(***sizeof***(***float***)) allocates storage for a floating-point number. *malloc* also returns a pointer to the storage it allocates. This pointer is to the first byte (for example, character) of that storage and is of type ***char*** *. To coerce this pointer so that it points to an integer or real, we use the cast operator (***int*** *) or (***float*** *).

(The ***sizeof*** operator returns a value of type ***int***, whereas the *malloc* function expects a parameter of type ***unsigned***. To make the program "lint free" we should write

```
pi = (int *) malloc ((unsigned)(sizeof (int)));
```

However, the cast on the ***sizeof*** operator is often omitted.)

As an example of the use of pointers and the function *malloc*, consider the following statements:

```
1        int *p, *q;
2        int x
3        p = (int *) malloc(sizeof (int));
4        *p = 3;
5        q = p;
6        printf ("%d %d \n", *p,  *q);
7        x = 7;
8        *q = x;
9        printf("%d %d \n", *p  *q);
10       p = (int *) malloc (sizeof (int));
11       *p = 5;
12       printf("%d %d \n", *p  *q);
```

In line 3, an integer variable is created and its address is plaeed in *p*. Line 4 sets the value of that variable to 3. Line 5 sets *q* to the address of that variable. The assignment statement in line 5 is perfectly valid, since one pointer variable (*q*) is being assigned the value of another (*p*). Figure 4.3.2a illustrates the situation after line 5. Note that at this point, **p* and **q* refer to the same variable. Line 6 therefore prints the contents of this variable (which is 3) twice.

Line 7 sets the value of an integer variable, *x*, to 7. Line 8 changes the value of **q* to the value of *x*. However, since *p* and *q* both point to the same variable, **p* and **q* both have the value 7. This is illustrated in Figure 4.3.2b. Line 9 therefore prints the number 7 twice.

Line 10 creates a new integer variable and places its address in *p*. The results are illustrated in Figure 4.3.2c. **p* now refers to the newly created integer variable that has not yet been given a value. *q* has not been changed; therefore the value of **q* remains 7. Note that **p* does not refer to a single, specific variable. Its value changes as the value of *p* changes. Line 11 sets the value of this newly created variable to 5, as illustrated in Figure 4.3.2d, and line 12 prints the values 5 and 7.

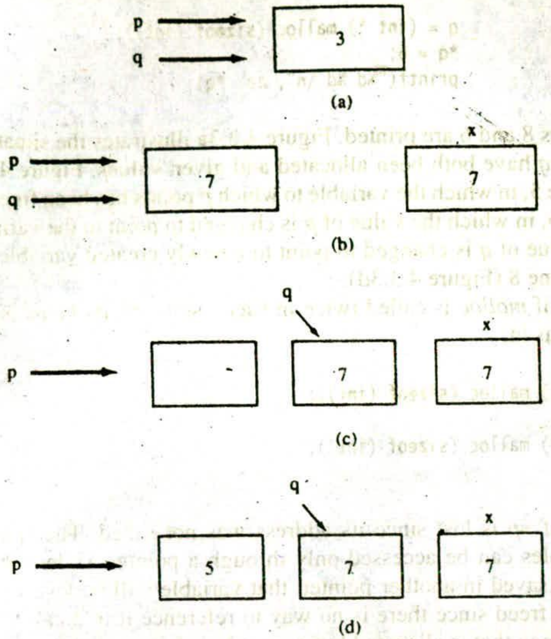


Figure 4.3.2

The function *free* is used in C to free storage of a dynamically allocated variable. The statement

```
free(p);
```

makes any future references to the variable **p* illegal (unless, of course, a new value is assigned to *p* by an assignment statement or by a call to *malloc*). Calling *free*(*p*) makes the storage occupied by **p* available for reuse, if necessary.

[*Note:* The *free* function, by default, expects a pointer parameter of type ***char*** *. To make the statement "lint free," we should write

```
free((char *) p);
```

However, in practice the cast on the parameter is often omitted.]

To illustrate the use of the *free* function, consider the following statements:

```
p = (int *) malloc (sizeof (int));
*P = 5;
q = (int *) malloc (sizeof (int));
*q = 8;
free(p);
p = q;
```

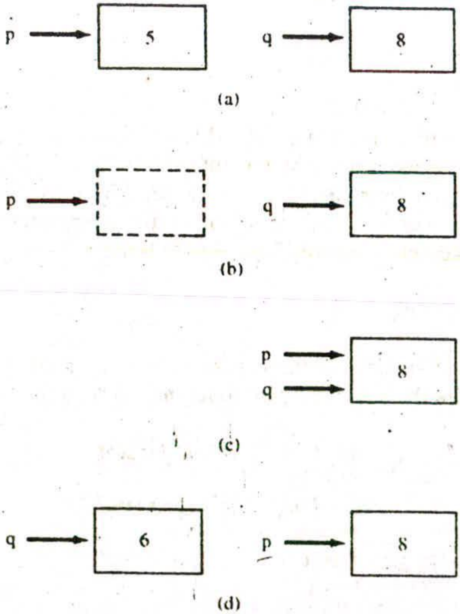
```
7          q = (int *) malloc (sizeof (int));
8          *q = 6;
9          printf("%d %d \n", *p. *q);
```

The values 8 and 6 are printed. Figure 4.3.3a illustrates the situation after line 4, where **p* and **q* have both been allocated and given values. Figure 4.3.3b illustrates the effect of line 5, in which the variable to which *p* points has been freed. Figure 4.3.3c illustrates line 6, in which the value of *p* is changed to point to the variable **q*. In lines 7 and 8, the value of *q* is changed to point to a newly created variable which is given the value 6 in line 8 (Figure 4.3.3d).

Note that if *malloc* is called twice in succession and its value is assigned to the same variable, as in:

```
p = (int *) malloc (sizeof (int));
*p = 3;
p = (int *) malloc (sizeof (int));
*p = 7;
```

the first copy of **p* is lost since its address was not saved. The space allocated for dynamic variables can be accessed only through a pointer. Unless the pointer to the first variable is saved in another pointer, that variable will be lost. In fact, its storage cannot even be freed since there is no way to reference it in a call to *free*. This is an example of storage that is allocated but cannot be referenced.



(a)

(b)

(c)

(d)

Figure 4.3.3

The value 0 (zero) can be used in a C program as the null pointer. Any pointer variable may be set to this value. Usually, a standard header to a C program includes the definition

```
#define NULL 0
```

to allow the zero pointer value to be written as *NULL*. This *NULL* pointer value does not reference a storage location but instead denotes the pointer that does not point to anything. The value *NULL* (zero) may be assigned to any pointer variable *p*, after which a reference to **p* is illegal.

We have noted that a call to *free*(*p*) makes a subsequent reference to **p* illegal. However, the actual effects of a call to *free* are not defined by the C language—each implementation of C is free to develop its own version of this function. In most C implementations, the storage for **p* is freed but the value of *p* is left unchanged. This means that although a reference to **p* becomes illegal, there may be no way of detecting the illegality. The value of *p* is a valid address and the object at that address of the proper type may be used as the value of **p*. *p* is called a ***dangling pointer***. It is the programmer's responsibility never to use such a pointer in a program. It is good practice to explicitly set *p* to *NULL* after executing *free*(*p*).

One other dangerous feature associated with pointers should be mentioned. If *p* and *q* are pointers with the same value, the variables **p* and **q* are identical. Both **p* and **q* refer to the same object. Thus, an assignment to **p* changes the value of **q*, despite the fact that neither *q* nor **q* are explicitly mentioned in the assignment statement to **p*. It is the programmer's responsibility to keep track of "which pointers are pointing where" and to recognize the occurrence of such implicit results.

Linked Lists Using Dynamic Variables

Now that we have the capability of dynamically allocating and freeing a variable, let us see how dynamic variables can be used to implement linked lists. Recall that a linked list consists of a set of nodes, each of which has two fields: an information field and a pointer to the next node in the list. In addition, an external pointer points to the first node in the list. We use pointer variables to implement list pointers. Thus, we define the type of a pointer and a node by

```
struct node {
    int info;
    struct node *next;
};
typedef struct node *NODEPTR;
```

A node of this type is identical to the nodes of the array implementation except that the *next* field is a pointer (containing the address of the next node in the list) rather than an integer (containing the index within an array where the next node in the list is kept).

Let us employ the dynamic allocation features to implement linked lists. Instead of declaring an array to represent an aggregate collection of nodes, nodes are allocated and freed as necessary. The need for a declared collection of nodes is eliminated.

If we declare

```
NODEPTR p;
```

execution of the statement

```
p = getnode();
```

should place the address of an available node into *p*. We present the function *getnode*:

```
NODEPTR getnode(void)
{
   NODEPTR p;
   p = (NODEPTR) malloc(sizeof(struct node));
   return(p);
}
```

Note that *sizeof* is applied to a structure type and returns the number of bytes required for the entire structure.

Execution of the statement

```
freenode(p);
```

should return the node whose address is at *p* to available storage. We present the routine *freenode*:

```
void freenode(NODEPTR p)
{
   free(p);
}
```

The programmer need not be concerned with managing available storage. There is no longer a need for the pointer *avail* (pointing to the first available node), since the system governs the allocating and freeing of nodes and the system keeps track of the first available node. Note also that there is no test in *getnode* to determine whether overflow has occurred. This is because such a condition will be detected during the execution of the *malloc* function and is system dependent.

Since the routines *getnode* and *freenode* are so simple under this implementation, they are often replaced by the in-line statements

```
p = (NODEPTR) malloc(sizeof (struct node));
```

and

```
      free(p);
```

The procedures *insafter(p,x)* and *delafter(p,px)* are presented below using the dynamic implementation of a linked list. Assume that *list* is a pointer variable that points to the first node of a list (if any) and equals *NULL* in the case of an empty list.

```
void insafter(NODEPTR p, int x)
{
   NODEPTR q;
   if (p == NULL) {
      printf("void insertion\n");
      exit(1);
   }
   q = getnode();
   q -> info = x;
   q -> next = p -> next;
   p -> next = q;
} /*  end insafter */


void delafter(NODEPTR p, int *px)
{
   NODEPTR q;
   if ((p == NULL) || (p -> next == NULL))  {
      printf("void deletion\n");
      exit(1);
   }
   q = p -> next;
   *px = q -> info;
   p -> next = q -> next;
   freenode(q);
} /* end delafter */
```

Notice the striking similarity between the preceding routines and those of the array implementation presented earlier in this section. Both are implementations of the algorithms of Section 4.2. In fact, the only difference between the two versions is in the manner in which nodes are referenced.

Queues as Lists in C

As a further illustration of how the C list implementations are used, we present C routines for manipulating a queue represented as a linear list. We leave the routines for manipulating a stack and a priority queue as exercises for the reader. For comparison purposes we show both the array and dynamic implementation. We assume that ***struct node*** and *NODEPTR* have been declared as in the foregoing. A queue is represented as a structure:

```
Array Implementation             Dynamic Implementation

struct queue {                   struct queue {
   int front, rear;                 NODEPTR front, rear;
};                               };
struct queue q;                  struct queue q;
```

front and *rear* are pointers to the first and last nodes of a queue presented as a list. The empty queue is represented by *front* and *rear* both equaling the null pointer. The function *empty* need check only one of these pointers since, in a nonempty queue, neither *front* nor *rear* will be *NULL*.

```
int empty(struct queue *pq)      int empty(struct queue *pq)
{                                {
   return ((pq->front == -1)        return ((pq->front == NULL)
      ? TRUE: FALSE);                     ? TRUE: FALSE);
} /* end empty */                } /* end empty */
```

The routine to insert an element into a queue may be written as follows:

```
void insert(struct queue *pq, int x)   void insert(struct queue *pq, int x)
 {                                     {
   int p;                                 NODEPTR p;
   p = getnode();
   node[p].info = x;                      p = getnode();
   node[p].next = -1;                     p->info = x;
   if (pq->rear == -1)                    p->next = NULL;
      pq->front = p;                      if (pq->rear == NULL)
   else                                      pq->front = p;
      node[pq->rear].next = p;            else
   pq->rear = p;                             (pq->rear)->next = p;
 } /* end insert */                       pq->rear = p;
                                       }  /* end insert */
```

The function *remove* deletes the first element from a queue and returns its value:

```
int remove(struct queue *pq)           int remove(struct queue *pq)
{                                      {
   int p, x;                              NODEPTR p;
                                          int x;
   if (empty(pq)) {                       if (empty(pq)) {
      printf("queue underflow\n");           printf("queue underflow\n");
      exit(1);                               exit(1);
   }                                      }
   p = pq->front;                         p = pq->front;
   x = node[p].info;                      x = p->info;
   pq->front = node[p].next;              pq->front = p->next;
   if (pq->front == -1)                   if (pq->front == NULL)
      pq->rear = -1;                         pq->rear = NULL;
   freenode(p);                           freenode(p);
   return(x);                             return(x);
} /* end remove */                     } /* end remove */
```

Examples of List Operations in C

Let us look at several somewhat more complex list operations implemented in C. We have seen that the dynamic implementation is often superior to the array implementation. For that reason the majority of C programmers use the dynamic implementation to implement lists. From this point on we restrict ourselves to the dynamic implementation of linked lists, although we might refer to the array implementation when appropriate.

We have previously defined the operation *place*(*list*, *x*), where *list* points to a sorted linear list and *x* is an element to be inserted into its proper position within the list. Recall that this operation is used to implement the operation *pqinsert* to insert i[illegible] priority queue. We assume that we have already implemented the stack operation *push*. The code to implement the *place* operation follows:

```
void place(NODEPTR *plist, int x)
{
   NODEPTR p, q;
   q = NULL;
   for (p = *plist; p != NULL && x > p->info; p = p->next)
      q = p;
   if (q == NULL)   /* insert x at the head of the list */
      push(plist, x);
   else
      insafter(q, x);
}  /* end place */
```

Note that *plist* must be declared as a pointer to the list pointer, since the value of the external list pointer is changed if *x* is inserted at the front of the list using the *push* routine. The foregoing routine would be called by the statement *place*(&*list*, *x*);.

As a second example, we write a function *insend*(*plist*,*x*) to insert the element *x* at the end of a list *list*:

```
void insend(NODEPTR *plist, int x)
{
   NODEPTR p, q;
   p = getnode();
   p->info = x;
   p->next = NULL;
   if (*plist == NULL)
      *plist = p;
   else {
      /* search for last node */
      for (q = *plist; q->next != NULL; q = q->next)
         ;
      q->next = p;
   } /* end if */
}  /* end insend */
```

We now present a function *search*(*list*, *x*) that returns a pointer to the first occurrence of *x* within the list *list* and the *NULL* pointer if *x* does not occur in the list:

```
NODEPTR search(NODEPTR list, int x)
{
   NODEPTR p;
   for (p = list; p != NULL; p = p->next)
      if (p->info == x)
         return (p);
   /* x is not in the list */
   return (NULL);
} /* end search */
```

The next routine deletes all nodes whose *info* field contains the value *x*:

```
void remvx(NODEPTR *plist, int x)
{
   NODEPTR p, q;
   int y;
   q = NULL;
   p = *plist;
   while (p != NULL)
      if (p -> info == x)  {
         p = p->next;
         if (q == NULL)  {
            /* remove first node of the list */
            freenode(*plist);
            *plist = p;
         }
         else
            delafter(q, &y);
      }
      else
         /* advance to next node of list   */
         q = p;
         p = p->next;
      } /* end if */
} /* end remvx */
```

Noninteger and Nonhomogeneous Lists

Of course, a node on a list need not represent an integer. For example, to represent a stack of character strings by a linked list, nodes containing character strings in their *info* fields are needed. Such nodes using the dynamic allocation implementation could be declared by

```
struct node {
        char info[100];
        struct node *next;
}
```

A particular application may call for nodes containing more than one item of information. For example, each student node in a list of students may contain the following information: the student's name, college identification number, address, grade point index, and major. Nodes for such an application may be declared as follows:

```
struct node  {
   char name[30];
   char id[9];
   char address[100];
   float gpindex;
   char major[20];
   struct node *next;
};
```

A separate set of C routines must be written to manipulate lists containing each type of node.

To represent nonhomogeneous lists (those that contain nodes of different types), a union can be used. For example,

```
#define INTGR      1
#define FLT        2
#define STRING     3
struct node {
   int etype; /* etype equals INTGR, FLT, or STRING   */
              /*  depending on the type of the        */
              /*         corresponding element.       */
   union {
      int ival;
      float fval;
      char *pval;  /* pointer to a string             */
   } element;
   struct node *next;
};
```

defines a node whose items may be either integers, floating-point numbers, or strings, depending on the value of the corresponding *etype*. Since a union is always large enough to hold its largest component, the ***sizeof*** and *malloc* functions can be used to allocate storage for the node. Thus the functions *getnode* and *freenode* remain unchanged. Of course, it is the programmer's responsibility to use the components of a node as appropriate. For simplicity, in the remainder of this section we assume that a linked list is declared to have only homogeneous elements (so that unions are not necessary). We examine nonhomogeneous lists, including lists that can contain other lists and recursive lists, in Section 9.1.

Comparing the Dynamic and Array Implementations of Lists

It is instructive to examine the advantages and disadvantages of the dynamic and array implementations of linked lists. The major disadvantage of the dynamic implementation is that it may be more time-consuming to call upon the system to allocate and free storage than to manipulate a programmer-managed available list. Its major advantage is that a set of nodes is not reserved in advance.for use by a particular group of lists.

For example, suppose that a program uses two types of lists: lists of integers and lists of characters. Under the array representation, two arrays of fixed size would immediately be allocated. If one group of lists overflows its array, the program cannot continue. Under the dynamic representation, two node types are defined at the outset, but no storage is allocated for variables until needed. As nodes are needed, the system is called upon to provide them. Any storage not used for one type of node may be used for another. Thus as long as sufficient storage is available for the nodes actually present in the lists, no overflow occurs.

Another advantage of the dynamic implementation is that a reference to *p does not involve the address computation that is necessary in computing the address of _node_[_p_]. To compute the address of _node_[_p_], the contents of _p_ must be added to the base address of the array _node_, whereas the address of *p is given by the contents of _p_ directly.

Implementing Header Nodes

At the end of Section 4.2 we introduced the concept of header nodes that can contain global information about a list, such as its length or a pointer to the current or last node on the list. When the data type of the header contents is identical to the type of the list-node contents, the header can be implemented simply as just another node at the beginning of the list.

It is also possible for header nodes to be declared as variables separate from the set of list nodes. This is particularly useful when the header contains information of a different type than the data in list nodes. For example, consider the following set of declarations:

```
struct node {
   char info;
   struct node *next;
};
struct charstr {
   int length;
   struct node *firstchar;
};
struct charstr s1, s2;
```

The variables _s_1 and _s_2 of type _charstr_ are header nodes for a list of characters. The header contains the number of characters in the list (_length_) and a pointer to the list (_firstchar_). Thus, _s_1 and _s_2 represent varying-length character strings. As exercises, you may wish to write routines to concatenate two such character strings or to extract a substring from such a string.

EXERCISES

4.3.1. Implement the routines *empty*, *push*, *pop*, and *popandtest* using the array and the dynamic storage implementations of a linked stack.

4.3.2. Implement the routines *empty*, *insert*, and *remove* using a dynamic storage implementation of a linked queue.

4.3.3. Implement the routines *empty*, *pqinsert*, and *pqmindelete* using a dynamic storage implementation of a linked priority queue.

4.3.4. Write C routines using both the array and dynamic variable implementations of a linked list to implement the operations of Exercise 4.2.3.

4.3.5. Write a C routine to interchange the *m*th and *n*th elements of a list.

4.3.6. Write a routine *inssub*(*l*1, *i*1, *l*2, *i*2, *len*) to insert the elements of list *l*2 beginning at the *i*2th element and continuing for *len* elements into the list *l*1 beginning at position *i*1. No elements of the list *l*1 are to be removed or replaced. If $i1 > length(l1) + 1$ (where *length*(*l*1) denotes the number of nodes in the list *l*1), or if *i*2 *length*(*l*2), or if $i1 < 1$, or if $i2 < 1$, print an error message. The list *l*2 should remain unchanged.

4.3.7. Write a C function *search*(*l*, *x*) that accepts a pointer *l* to a list of integers and an integer *x* and returns a pointer to a node containing *x*, if it exists, and the null pointer otherwise. Write another function, *srchinsrt*(*l*, *x*), that adds *x* to *l* if it is not found and always returns a pointer to a node containing *x*.

4.3.8. Write a C program to read a group of input lines, each containing one word. Print each word that appears in the input and the number of times that it appears.

4.3.9. Suppose that a character string is represented by a list of single characters. Write a set of routines to manipulate such lists as follows (in the following, *l*1, *l*2, and *list* are pointers to a header node of a list representing a character string, *str* is an array of characters, and *i*1 and *i*2 are integers):

(a) *strcnvcl*(*str*) to convert the character string *str* to a list. This function returns a pointer to a header node.

(b) *strcnvlc*(*list*, *str*) to convert a list into a character string.

(c) *strpsl*(*l*1, *l*2) to perform the *strpos* function of Section 1.2 on two character strings represented by lists. This function returns an integer.

(d) *strvrfyl*(*l*1, *l*2) to determine the first position of the string represented by *l*1 that is not contained in the string represented by *l*2. This function returns an integer.

(e) *strsbstr*(*l*1, *i*1, *i*2) to perform the *substr* function of Section 1.2 on a character string represented by list *l*1 and integers *i*1 and *i*2. This function returns a pointer to the header node of a list representing a character string, which is the desired substring. The list *l*1 remains unchanged.

(f) *strpsbl*(*l*1, *i*1, *i*2, *l*2) to perform a pseudo-*substr* assignment to list *l*1. The elements of list *l*2 should replace the *i*2 elements of *l*1 beginning at position *i*1. The list *l*2 should remain unchanged.

(g) *strcmpl*(*l*1, *l*2) to compare two character strings represented by lists. This function returns -1 if the character string represented by *l*1 is less than the string represented by *l*2, 0 if they are equal, and 1 if the string represented by *l*1 is greater.

4.3.10. Write a function *binsrch* that accepts two parameters, an array of pointers to a group of sorted numbers, and a single number. The function should use a binary search (see

Section 3.1) to return a pointer to the single number if it is in the group. If the number is not present in the group, return the value *NULL*.

4.3.11. Assume that we wish to form *N* lists, where *N* is a constant. Declare an array *list* of pointers by

```
#define N ...
struct  node  {
                int info
                struct node *next
};
typedef struct node *NODEPTR;
NODEPTR list [N];
```

Read two numbers from each input line, the first number being the index of the list into which the second number is to be placed in ascending order. When there are no more input lines, print all the lists.

4.4 EXAMPLE: SIMULATION USING LINKED LISTS

One of the most useful applications of queues, priority queues, and linked lists is in ***simulation***. A simulation program attempts to model a real-world situation in order to learn something about it. Each object and action in the real situation has its counterpart in the program. If the simulation is accurate—that is, if the program successfully mirrors the real world—the result of the program should mirror the result of the actions being simulated. Thus it is possible to understand what occurs in the real-world situation without actually observing its occurrence.

Let us look at an example. Suppose that there is a bank with four tellers. A customer enters the bank at a specific time ($t1$) desiring to conduct a transaction with any teller. The transaction may be expected to take a certain period of time ($t2$) before it is completed. If a teller is free, the teller can process the customer's transaction immediately, and the customer leaves the bank as soon as the transaction is completed, at time $t1 + t2$. The total time spent in the bank by the customer is exactly equal to the duration of the transaction ($t2$).

However, it is possible that none of the tellers are free; they are all servicing customers who arrived previously. In that case there is a line waiting at each teller's window. The line for a particular teller may consist of a single person—the one currently transacting business with the teller—or it may be a very long line. The customer proceeds to the back of the shortest line and waits until all the previous customers have completed their transactions and have left the bank. At that time the customer may transact his or her business. The customer leaves the bank at $t2$ time units after reaching the front of a teller's line. In this case the time spent in the bank is $t2$ plus the time spent waiting on line.

Given such a system, we would like to compute the average time spent by a customer in the bank. One way of doing so is to stand in the bank doorway, ask departing customers the time of their arrival and record the time of their departure, subtract the first from the second for each customer, and take the average over all customers. However, this would not be very practical. It would be difficult to ensure that no customer is overlooked leaving the bank. Furthermore, it is doubtful that most customers would remember the exact time of arrival.

Instead, we write a program to simulate the customer actions. Each part of the real-world situation has its analogue in the program. The real-world action of a customer arriving is modeled by input of data. As each customer arrives, two facts are known: the time of arrival and the duration of the transaction (since, presumably, when a customer arrives, he or she knows what he or she wishes to do at the bank). Thus the input data for each customer consists of a pair of numbers: the time (in minutes since the bank opened) of the customer's arrival and the amount of time (again, in minutes) necessary for the transaction. The data pairs are ordered by increasing arrival time. We assume at least one input line.

The four lines in the bank are represented by four queues. Each node of the queues represents a customer waiting on a line, and the node at the front of a queue represents the customer currently being serviced by a teller.

Suppose that at a given instant of time the four lines each contain a specific number of customers. What can happen to alter the status of the lines? Either a new customer enters the bank, in which case one of the lines will have an additional customer, or the first customer on one of the four lines completes a transaction, in which case that line will have one fewer customer. Thus there are a total of five actions (a customer entering plus four cases of a customer leaving) that can change the status of the lines. Each of these five actions is called an ***event***.

Simulation Process

The simulation proceeds by finding the next event to occur and effecting the change in the queues that mirrors the change in the lines at the bank due to that event. To keep track of events, the program uses an ascending priority queue, called the ***event list***. This list contains at most five nodes, each representing the next occurrence of one of the five types of events. Thus the event list contains one node representing the next customer arriving and four nodes representing each of the four customers at the head of a line completing a transaction and leaving the bank. Of course, it is possible that one or more of the lines in the bank are empty, or that the doors of the bank have been closed for the day, so that no more customers are arriving. In such cases the event list contains fewer than five nodes.

An event node representing a customer's arrival is called an ***arrival node***, and a node representing a departure is called a ***departure node***. At each point in the simulation, it is necessary to know the next event to occur. For this reason, the event list is ordered by increasing time of event occurrence, so that the first event node on the list represents the next event to occur. Thus the event list is an ascending priority queue represented by an ordered linked list.

The first event to occur is the arrival of the first customer. The event list is therefore initialized by reading the first input line and placing an arrival node representing the first customer's arrival on the event list. Initially, of course, all four teller queues are empty. The simulation then proceeds as follows: The first node on the event list is removed and the changes that the event causes are made to the queues. As we shall soon see, these changes may also cause additional events to be placed on the event list. The process of removing the first node from the event list and effecting the changes that it causes is repeated until the event list is empty.

When an arrival node is removed from the event list, a node representing the arriving customer is placed on the shortest of the four teller queues. If that customer is the only one on a queue, a node representing his or her departure is also placed on the event list, since he or she is at the front of the queue. At the same time, the next input line is read and an arrival node representing the next customer to arrive is placed on the event list. There will always be exactly one arrival node on the event list (as long as the input is not exhausted, at which point no more customers arrive), since as soon as one arrival node is removed from the event list another is added to it.

When a departure node is removed from the event list, the node representing the departing customer is removed from the front of one of the four queues. At that point the amount of time that the departing customer has spent in the bank is computed and added to a total. At the end of the simulation, this total will be divided by the number of customers to yield the average time spent by a customer. After a customer node has been deleted from the front of its queue, the next customer on the queue (if any) becomes the one being serviced by that teller and a departure node for that next customer is added to the évent list.

This process continues until the event list is empty, at which point the average time is computed and printed. Note that the event list itself does not mirror any part of the real-world situation. It is used as part of the program to control the entire process. A simulation such as this one, which proceeds by changing the simulated situation in response to the occurrence of one of several events, is called an ***event-driven simulation***.

Data Structures

We now examine the data structures necessary for this program. The nodes on the queues represent customers and therefore must contain fields representing the arrival time and the transaction duration, in addition to a *next* field to link the nodes in a list. The nodes on the event list represent events and therefore must contain the time that the event occurs, the type of the event, and any other information associated with the event, as well as a *next* field. Thus it would seem that two separate node pools are needed for the two different types of node. Two different types of node would entail two *getnode* and *freenode* routines and two sets of list manipulation routines. To avoid this cumbersome set of duplicate routines, let us try to use a single type of node for both events and customers.

We can declare such a pool of nodes and a pointer type as follows:

```
struct node  {
   int time;
   int duration;
   int type;
   struct node *next;
};
typedef struct node *NODEPTR;
```

In a customer node, *time* is the customer's arrival time and *duration* is the transaction's duration. *type* is unused in a customer node. *next* is used as a pointer to link the queue together. For an event node, *time* is used to hold the time of the event's occurrence; *duration* is used for the transaction duration of the arriving customer in an arrival node and is unused in a departure node. *type* is an integer between -1 and 3, depending on whether the event is an arrival (*type* $== -1$) or a departure from line 0, 1, 2, or 3 (*type* $==$ 0, 1, 2, or 3). *next* holds a pointer linking the event list together.

The four queues representing the teller lines are declared as an array by the declaration

```
struct queue {
   NODEPTR front, rear;
   int num;
};
struct queue q[4];
```

The variable *q*[*i*] represents a header for the *i*th teller queue. The *num* field of a queue contains the number of customers on that queue.

A variable *evlist* points to the front of the event list. A variable *tottime* is used to keep track of the total time spent by all customers, and *count* keeps count of the number of customers that have passed through the bank. These will be used at the end of the simulation to compute the average time spent in the bank by the customers. An auxiliary variable *auxinfo* is used to store temporarily the information portion of a node. These variables are declared by

```
NODEPTR evlist;
float count, tottime;
struct node auxinfo;
```

Simulation Program

The main routine initializes all lists and queues and repeatedly removes the next node from the event list to drive the simulation until the event list is empty. The event list is ordered by increasing value of the *time* field. The program uses the call *place*(*&evlist*, *&auxinfo*) to insert a node whose information is given by *auxinfo* in its proper place in the event list. The main routine also calls *popsub*(*&evlist*, *&auxinfo*) to remove the first node from the event list and place its information in *auxinfo*. This routine is equivalent to the function *pop*. These routines must, of course, be suitably modified from the examples given in the last section in order to handle this particular type of node. Note that *evlist*, *place*, and *popsub* are merely a particular implementation of an ascending priority queue and the operations *pqinsert* and *pqmindelete*. A more efficient representation of a priority queue (such as we present in Sections 6.3 and 7.3) would allow the program to operate somewhat more efficiently.

The main program also calls on the functions *arrive* and *depart*, which effect the changes in the event list and the queues caused by an arrival and a departure. Specifically, the function *arrive*(*atime*, *dur*) reflects the arrival of a customer at time *atime* with a transaction of duration *dur*, and the function *depart(qindx, dtime)* reflects the departure of the first customer from queue *q[qindx]* at time *dtime*. The coding of these routines will be given shortly.

```
#include <stdio.h>
#define NULL 0
struct node{
   int duration, time, type;
   struct node *next;
};
typedef struct node *NODEPTR;
struct queue {
   NODEPTR front, rear;
   int num;
};
struct queue q[4];
struct node auxinfo;
NODEPTR evlist;
int atime, dtime, dur, qindx;
float count, tottime;

void place(NODEPTR *, struct node *);
void popsub(NODEPTR *, struct node *);
void arrive(int, int);
void depart(int, int);
void push(NODEPTR *, struct node *);
void insafter(NODEPTR *, struct node *);
int empty(NODEPTR);
void insert(struct queue *, struct node *);
void remove(struct queue *, struct node *);
NODEPTR getnode(void);
void freenode(NODEPTR);

void main()
{
   /* initializations */
   evlist = NULL;
   count = 0;
   tottime = 0;
   for (qindx = 0; qindx < 4; qindx++) {
      q[qindx].num = 0;
      q[qindx].front = NULL;
      q[qindx].rear = NULL;
   } /* end for */
   /* initialize the event list with the first arrival */
   printf("enter time and duration\n");
   scanf("%d %d", &auxinfo.time, &auxinfo.duration);
   auxinfo.type = -1;  /* an arrival */
   place(&evlist, &auxinfo);
```

```
      /* run the simulation as long as the event list is not empty */
      while (evlist != NULL) {
         popsub(&evlist, &auxinfo);
         /*  check if the next event is an arrival or departure */
         if (auxinfo.type == -1) {
            /*        an arrival         */
            atime = auxinfo.time;
            dur = auxinfo.duration;
            arrive(atime, dur);
         }
         else {
            /*       a departure         */
            qindx = auxinfo.type;
            dtime = auxinfo.time;
            depart(qindx, dtime);
         }  /* end if */
      }  /* end while */
      printf("average time is %4.2f", tottime/count);
   }  /* end main */
```

The routine *arrive*(*atime*, *dur*) modifies the queues and the event list to reflect a new arrival at time *atime* with a transaction of duration *dur*. It inserts a new customer node at the rear of the shortest queue by calling the function *insert*(*&q*[*j*], *&auxinfo*). The *insert* routine must be suitably modified to handle the type of node in this example and must also increase *q*[*j*].*num* by 1. If the customer is the only one on the queue, a node representing his or her departure is added to the event list by calling on the function *place*(*&evlist*, *&auxinfo*). Then the next data pair (if any) is read and an arrival node is placed on the event list to replace the arrival that has just been processed. If there is no more input, the function returns without adding a new arrival node and the program processes the remaining (departure) nodes on the event list.

```
void arrive(int  atime, int dur)
{
   int i, j, small;
   /* find the shortest queue */
   j = 0;
   small = q[0].num;
   for (i = 1; i < 4; i++)
      if (q[i].num < small)  {
         small = q[i].num;
         j = i;
      }  /* end  for ... if */
   /* Queue j is the shortest. Insert a new customer node. */
   auxinfo.time = atime;
   auxinfo.duration = dur;
   auxinfo.type = j;
   insert(&q[j], &auxinfo);
```

```
      /* Check if this is the only node on the queue. If it  */
      /* is, the customer's departure node must be placed on */
      /*               the event list.                       */
      if (q[j].num == 1) {
         auxinfo.time = atime + dur;
         place(&evlist, &auxinfo);
      }
      /* If any input remains, read the next data pair and  */
      /*            place an arrival on the event list.     */
      printf("enter time\n");
      if (scanf("%d", &auxinfo.time) != EOF) {
         printf("enter duration\n");
         scanf("%d", &auxinfo.duration);
         auxinfo.type = -1;
         place(&evlist, &auxinfo);
      } /* end if */
   } /* end arrive */
```

The routine *depart(qindx, dtime)* modifies the queue *q[qindx]* and the event list to reflect the departure of the first customer on the queue at time *dtime*. The customer is removed from the queue by the call *remove(&q[qindx], &auxinfo)*, which must be suitably modified to handle the type of node in this example and must also decrement the queue's *num* field by 1. The departure node of the next customer on the queue (if any) replaces the departure node that has just been removed from the event list.

```
   void depart(int qindx, int dtime)
   {
      NODEPTR p;
      remove(&q[qindx], &auxinfo);
      tottime = tottime + (dtime - auxinfo.time);
      count++;
      /*   if there are any more customers on the queue,   */
      /*   place the departure of the next customer onto   */
      /* the event list after computing its departure time */
      if (q[qindx].num > 0)  {
         p = q[qindx].front;
         auxinfo.time = dtime + p->duration;
         auxinfo.type = qindx;
         place(&evlist, &auxinfo);
      } /* end if */
   } /* end depart */
```

Simulation programs are rich in their use of list structures. The reader is urged to explore the use of C for simulation and the use of special-purpose simulation languages.

EXERCISES

4.4.1. In the bank simulation program of the text, a departure node on the event list represents the same customer as the first node on a customer queue. Is it possible to use a single node for a customer currently being serviced? Rewrite the program of the text so that only a single node is used. Is there any advantage to using two nodes?

4.4.2. The program in the text uses the same type of node for both customer and event nodes. Rewrite the program using two different types of nodes for these two purposes. Does this save space?

4.4.3. Revise the bank simulation program of the text to determine the average length of the four lines.

4.4.4. Modify the bank simulation program to compute the standard deviation of the time spent by a customer in the bank. Write another program that simulates a single line for all four tellers with the customer at the head of the single line going to the next available teller. Compare the means and standard deviations of the two methods.

4.4.5. Modify the bank simulation program so that whenever the length of one line exceeds the length of another by more than two, the last customer on the longer line moves to the rear of the shorter.

4.4.6. Write a C program to simulate a simple multiuser computer system as follows: Each user has a unique ID and wishes to perform a number of transactions on the computer. However, only one transaction may be processed by the computer at any given moment. Each input line represents a single user and contains the user's ID followed by a starting time and a series of integers representing the duration of each of his or her transactions. The input is sorted by increasing starting time, and all times and durations are in seconds. Assume that a user does not request time for a transaction until the previous transaction is complete and that the computer accepts transactions on a first-come, first-served basis. The program should simulate the system and print a message containing the user ID and the time whenever a transaction begins and ends. At the end of the simulation it should print the average waiting time for a transaction. (The waiting time is the amount of time between the time that the transaction was requested and the time it was started.)

4.4.7. What parts of the bank simulation program would have to be modified if the priority queue of events were implemented as an array or as an unordered list? How would they be modified?

4.4.8. Many simulations do not simulate events given by input data but rather generate events according to some probability distribution. The following exercises explain how. Most computer installations have a random number generating function *rand*(x). (The name and parameters of the function vary from system to system. *rand* is used as an example only.) x is initialized to a value called a ***seed***. The statement x = *rand*(x) resets the value of the variable x to a uniform random real number between 0 and 1. By this we mean that if the statement is executed a sufficient number of times and any two equal-length intervals between 0 and 1 are chosen, approximately as many of the successive values of x fall into one interval as into the other. Thus the probability of a value of x falling in an interval of length $l <= 1$ equals l. Find out the name of the random number generating function on your system and verify that the foregoing is true. Given a random number generator *rand* consider the following statements:

```
x = rand(x);
y = (b-a)*x + a
```

(a) Show that, given any two equal-length intervals within the interval from *a* to *b*, if the statements are repeated sufficiently often, an approximately equal number of successive values of *y* fall into each of the two intervals. Show that if *a* and *b* are integers, the successive values of *y* truncated to an integer equal each integer between *a* and *b* − 1 an approximately equal number of times. The variable *y* is said to be a ***uniformly distributed random variable***. What is the average of the values of *y* in terms of *a* and *b*?

(b) Rewrite the bank simulation of the text, assuming that the transaction duration is uniformly distributed between 1 and 15. Each data pair represents an arriving customer and contains only the time of arrival. Upon reading an input line, generate a transaction duration for that customer by computing the next value according to the method just outlined.

4.4.9. The successive values of *y* generated by the following statements are called ***normally distributed***. (Actually, they are approximately normally distributed, but the approximation is close enough.)

```
float x[15];
float m, s, sum, y;
int i;
/* statements initializing the values of s, m and  */
/*              the array x go here                 */
while ( /* a terminating condition goes here */ ) {
    sum = 0;
    for (i = 0; i < 15; i++)  {
        x[i] = rand(x[i]);
        sum = sum + x[i];
    } /* end for */
    y = s * (sum - 7.5) / sqrt(1.25) + m;
    /* statments that  use the value of y go here */
} /* end while */
```

(a) Verify that the average of the values of *y* (the mean of the distribution) equals *m* and that the standard deviation equals *s*.

(b) A certain factory produces items according to the following process: an item must be assembled and polished. Assembly time is uniformly distributed between 100 and 300 seconds, and polishing time is normally distributed with a mean of 20 seconds and a standard deviation of 7 seconds (but values below 5 are discarded). After an item is assembled, a polishing machine must be used, and a worker cannot begin assembling the next item until the item he or she has just assembled has been polished. There are ten workers but only one polishing machine. If the machine is not available, workers who have finished assembling their items must wait for it. Compute the average waiting time per item by means of a simulation. Do the same under the assumption of two and three polishing machines.

4.5 OTHER LIST STRUCTURES

Although a linked linear list is a useful data structure, it has several shortcomings. In this section we present other methods of organizing a list and show how they can be used to overcome these shortcomings.

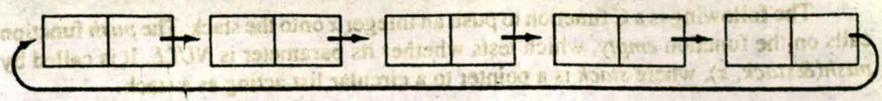


Figure 4.5.1 Circular list.

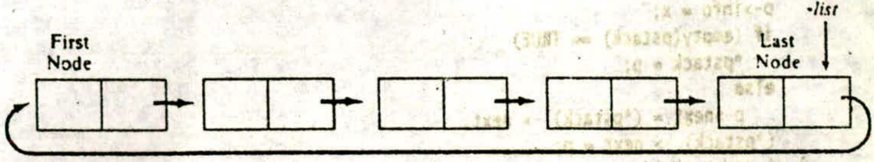


Figure 4.5.2 First and last nodes of a circular list.

Circular Lists

Given a pointer *p* to a node in a linear list, we cannot reach any of the nodes that precede *node*(*p*). If a list is traversed, the external pointer to the list must be preserved to be able to reference the list again.

Suppose that a small change is made to the structure of a linear list, so that the *next* field in the last node contains a pointer back to the first node rather than the null pointer. Such a list is called a ***circular list*** and is illustrated in Figure 4.5.1. From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point.

Note that a circular list does not have a natural "first" or "last" node. We must, therefore, establish a first and last node by convention. One useful convention is to let the external pointer to the circular list point to the last node, and to allow the following node to be the first node, as illustrated in Figure 4.5.2. If *p* is an external pointer to a circular list, this convention allows access to the last node of the list by referencing *node*(*p*) and to the first node of the list by referencing *node*(*next*(*p*)). This convention provides the advantage of being able to add or remove an element conveniently from either the front or the rear of a list. We also establish the convention that a null pointer represents an empty circular list.

Stack as a Circular List

A circular list can be used to represent a stack or a queue. Let *stack* be a pointer to the last node of a circular list and let us adopt the convention that the first node is the top of the stack. An empty stack is represented by a null list. The following is a C function to determine whether the stack is empty. It is called by *empty*(&*stack*).

```
int empty(NODEPTR *pstack)
{
   return ((*pstack == NULL) ? TRUE : FALSE);
} /* end empty */
```

The following is a C function to push an integer *x* onto the stack. The *push* function calls on the function *empty*, which tests whether its parameter is *NULL*. It is called by *push*(&*stack*, *x*), where *stack* is a pointer to a circular list acting as a stack.

```
void push(NODEPTR *pstack, int x)
{
   NODEPTR p;
   p = getnode();
   p->info = x;
   if (empty(pstack) == TRUE)
      *pstack = p;
   else
      p->next = (*pstack) -> next;
   (*pstack) -> next = p;
} /* end push */
```

Note that the *push* routine is slightly more complex for circular lists than it is for linear lists.

The C *pop* function for a stack implemented as a circular list calls the function *freenode* introduced earlier. *pop* is called by *pop*(&*stack*).

```
int pop(NODEPTR *pstack)
{
   int x;
   NODEPTR p;
   if (empty(pstack) == TRUE) {
      printf("stack underflow\n");
      exit(1);
   } /* end if */
   p = (*pstack) -> next;
   x = p->info;
   if (p == *pstack)
      /* only one node on the stack */
      *pstack = NULL;
   else
      (*pstack) -> next = p->next;
   freenode(p);
   return(x);
} /* end pop */
```

Queue as a Circular List

It is easier to represent a queue as a circular list than as a linear list. As a linear list, a queue is specified by two pointers, one to the front of the list and the other to its rear. However, by using a circular list, a queue may be specified by a single pointer *q* to that list. *node*(*q*) is the rear of the queue and the following node is its front.

The function *empty* is the same as for stacks. The routine *remove*(*pq*) called by *remove*(&*q*) is identical to *pop* except that all references to *pstack* are replaced by *pq*, a pointer to *q*. The C routine *insert* is called by the statement *insert*(&*q*, *x*) and may be coded as follows:

```
void insert(NODEPTR *pq, int x)
{
   NODEPTR p;
   p = getnode();
   p->info = x;
   if (empty(pq) == TRUE)
      *pq = p;
   else
      p->next = (*pq) -> next;
   (*pq) -> next = p;
   *pq = p;
   return;
} /* end insert */
```

Note that *insert*(&*q*, *x*) is equivalent to the code

```
push(&q,x);
q = q->next;
```

That is, to insert an element into the rear of a circular queue, the element is inserted into the front of the queue and the circular list pointer is then advanced one element, so that the new element becomes the rear.

Primitive Operations on Circular Lists

The routine *insafter*(*p*, *x*), which inserts a node containing *x* after *node*(*p*), is similar to the corresponding routine for linear lists as presented in Section 4.3. However, the routine *delafter*(*p*, *x*) must be modified slightly. Looking at the corresponding routine for linear lists as presented in Section 4.3, we note one additional consideration in the case of a circular list. Suppose that *p* points to the only node in the list. In a linear list, *next*(*p*) is null in that case, making the deletion invalid. In the case of a circular list, however, *next*(*p*) points to *node*(*p*), so that *node*(*p*) follows itself. The question is whether or not it is desirable to delete *node*(*p*) from the list in this case. It is unlikely that we would want to do so, since the operation *delafter* is usually invoked when pointers to each of two nodes are given, one immediately following another, and it is desired to delete the second. *delafter* for circular lists using the dynamic node implementation is implemented as follows:

```
void delafter(NODEPTR p, int *px)
{
   NODEPTR q;
   if ((p == NULL) || (p == p->next)) {
      /* the list is empty or contains only a single node */
      printf("void deletion\n");
      return;
   } /* end if */
```

```
    q = p->next;
    *px = q->info;
    p->next = q->next;
    freenode(q);
    return;
} /* end delafter */
```

Note, however, that *insafter* cannot be used to insert a node following the last node in a circular list and that *delafter* cannot be used to delete the last node of a circular list. In both cases the external pointer to the list must be modified to point to the new last node. The routines can be modified to accept *list* as an additional parameter and to change its value when necessary. (The actual parameter in the calling routine would have to be &*list*, since its value is changed.) An alternative is to write separate routines *insend* and *dellast* for these cases. (*insend* is identical to the *insert* operation for a queue implemented as a circular list.) The calling routine would be responsible for determining which routine to call. Another alternative is to give the calling routine the responsibility of adjusting the external pointer *list* if necessary. We leave the exploration of these possibilities to the reader.

If we are managing our own available list of nodes (as for example under the array implementation), it is also easier to free an entire circular list than to free a linear list. In the case of a linear list the entire list must be traversed, as one node at a time is returned to the available list. For a circular list, we can write a routine *freelist* that effectively frees an entire list by simply rearranging pointers. This is left as an exercise for the reader.

Similarly, we may write a routine *concat*(&*list*1, &*list*2) that concatenates two lists; that is, it appends the circular list pointed to by *list*2 to the end of the circular list pointed to by *list*1. Using circular lists, this can be done without traversing either list:

```
void concat(NODEPTR *plist1, NODEPTR *plist2)
{
    NODEPTR p;
    if (*plist2 == NULL)
        return;
    if (*plist1 == NULL)  {
        *plist1 = *plist2;
        return;
    }
    p = (*plist1) -> next;
    (*plist1) -> next = (*plist2) -> next;
    (*plist2) -> next = p;
    *plist1 = *plist2;
    return;
} /* end concat */
```

The Josephus Problem

Let us consider a problem that can be solved in a straightforward manner by using a circular list. The problem is known as the Josephus problem and postulates a group of soldiers surrounded by an overwhelming enemy force. There is no hope for victory without reinforcements, but there is only a single horse available for escape. The soldiers agree to a pact to determine which of them is to escape and summon help. They form a circle and a number *n* is picked from a hat. One of their names is also picked from a hat. Beginning with the soldier whose name is picked, they begin to count clockwise around the circle. When the count reaches *n*, that soldier is removed from the circle, and the count begins again with the next soldier. The process continues so that each time the count reaches *n*, another soldier is removed from the circle. Any soldier removed from the circle is no longer counted. The last soldier remaining is to take the horse and escape. The problem is, given a number *n*, the ordering of the soldiers in the circle, and, the soldier from whom the count begins, to determine the order in which soldiers are eliminated from the circle and which soldier escapes.

The input to the program is the number *n* and a list of names, which is the clockwise ordering of the circle, beginning with the soldier from whom the count is to start. The last input line contains the string "*end*" indicating the end of the input. The program should print the names in the order that they are eliminated and the name of the soldier who escapes.

For example, suppose that $n = 3$ and that there are five soldiers named *A*, *B*, *C*, *D*, and *E*. We count three soldiers starting at *A*, so that *C* is eliminated first. We then begin at *D* and count *D*, *E*, and back to *A*, so that *A* is eliminated next. Then we count *B*, *D*, and *E* (*C* has already been eliminated), and finally *B*, *D*, and *B*, so that *D* is the one who escapes.

Clearly, a circular list in which each node represents one soldier is a natural data structure to use in solving this problem. It is possible to reach any node from any other by counting around the circle. To represent the removal of a soldier from the circle, a node is deleted from the circular list. Finally, when only one node remains on the list, the result is determined.

An outline of the program might be as follows:

```
read(n);
read(name);
while (name != END)  {
      insert name on the circular list;
      read(name);
} /* end while */
while (there is more than one node on the list)  {
      count through n - 1 nodes on the list;
      print the name in the nth node;
      delete the nth node;
} /* end while */
print the name of the only node on the list;
```

We assume that a set of nodes has been declared as before except that the *info* field holds a character string (an array of characters) rather than an integer. We also assume at least one name in the input. The program uses the routines *insert*, *delafter*, and *freenode*. The routines *insert* and *delafter* must be modified, since the information portion of the node is a character string. Assignment from one character string variable to another is

accomplished via a loop. The program also makes use of a function *eqstr(str*1, *str*2), which returns *TRUE* if *str*1 is identical to *str*2, and *FALSE* otherwise. The coding of this routine is left to the reader.

```
void josephus(void)
{
   char *end = "end";
   char name[MAXLEN];
   int i, n;
   NODEPTR list = NULL;
   printf("enter n\n");
   scanf("%d", &n);
   /* read the names, placing each */
   /*    at the rear of the list   */
   printf("enter names\n");
   scanf("%s", &name);
   /* form the list */
   while (!eqstr(name, end)) {
      insert(&list, name);
      scanf("%s", name);
   } /* end while */
   printf("the order in which the soldiers are eliminated is:\n");
   /* continue counting as long as more */
   /* than one node remains on the list */
   while (list != list->next) {
      for (i = 1; i < n; i++)
         list = list->next;
      /* list->next points to the nth node */
      delafter(list, name);
      printf("%s\n", name);
   } /*  end while */
   /* print the only name on the list and free its node */
   printf("the soldier who escapes is: %s", list->info);
   freenode(list);
} /* end josephus */
```

Header Nodes

Suppose that we wish to traverse a circular list. This can be done by repeatedly executing *p* = *p*→*next*, where *p* is initially a pointer to the beginning of the list. However, since the list is circular, we will not know when the entire list has been traversed unless another pointer, *list*, points to the first node and a test is made for the condition *p* == *list*.

An alternative method is to place a header node as the first node of a circular list. This list header may be recognized by a special value in its *info* field that cannot be the valid contents of a list node in the context of the problem, or it may contain a flag marking it as a header. The list can then be traversed using a single pointer, with the traversal halting when the header node is reached. The external pointer to the list is to

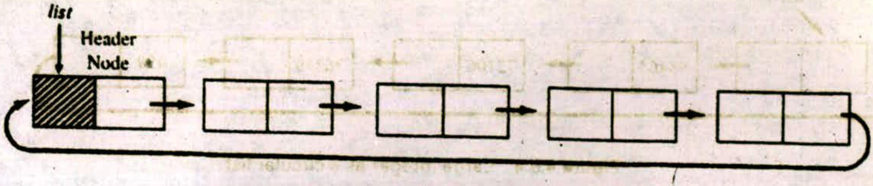


Figure 4.5.3 Circular list with a header node.

its header node, as illustrated in Figure 4.5.3. This means that a node cannot easily be added onto the rear of such a circular list, as could be done when the external pointer was to the last node of the list. Of course, it is possible to keep a pointer to the last node of a circular list even when a header node is being used.

If a stationary external pointer to a circular list is used in addition to the pointer used for traversal, the header node need not contain a special code but can be used in much the same way as a header node of a linear list to contain global information about the list. The end of a traversal would be signaled by the equality of the traversing pointer and the external stationary pointer.

Addition of Long Positive Integers Using Circular Lists

We now present an application of circular lists with header nodes. The hardware of most computers allows integers of only a specific maximum length. Suppose that we wish to represent positive integers of arbitrary length and to write a function that returns the sum of two such integers.

To add two such long integers, their digits are traversed from right to left, and corresponding digits and a possible carry from the previous digits' sum are added. This suggests representing long integers by storing their digits from right to left in a list so that the first node on the list contains the least significant digit (rightmost) and the last node contains the most significant (leftmost). However, to save space, we keep five digits in each node. (Long integer variables are used so that numbers as large as 99999 may be kept in each node. The maximum size of an integer is implementation-dependent; therefore you may have to modify the routines to hold smaller numbers in each node.) We may declare the set of nodes by

```
struct node   {
   long int info;
   struct node *next;
};
typedef struct node *NODEPTR;
```

Since we wish to traverse the lists during the addition but wish to eventually restore the list pointers to their original values, we use circular lists with headers. The header node is distinguished by an *info* value of −1. For example, the integer 459763497210698463 is represented by the list illustrated in Figure 4.5.4.

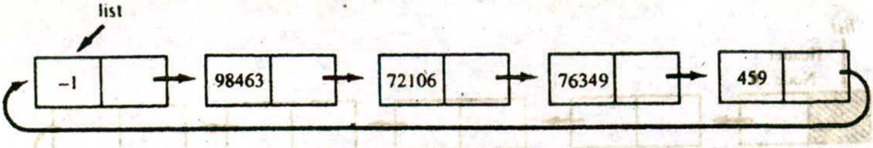


Figure 4.5.4 Large integer as a circular list.

Now let us write a function *addint* that accepts pointers to two such lists representing integers, creates a list representing the sum of the integers, and returns a pointer to the sum list. Both lists are traversed in parallel, and five digits are added at a time. If the sum of two five-digit numbers is *x*, the low-order five digits of *x* can be extracted by using the expression *x* % 100000, which yields the remainder of *x* on division by 100000. The carry can be computed by the integer division *x*/100000. When the end of one list is reached, the carry is propagated to the remaining digits of the other list. The function follows and uses the routines *getnode* and *insafter*.

```
NODEPTR addint(NODEPTR p, NODEPTR q)
{
   long int hunthou = 100000L;
   long int carry, number, total;
   NODEPTR s;
   /*  set p and q to the nodes following the headers */
   p = p->next;
   q = q->next;
   /* set up a header node for the sum */
   s = getnode();
   s->info = -1;
   s->next = s;
   /* initially there is no carry */
   carry = 0;
   while (p->info != -1 && q->info != -1)  {
      /* add the info of the two nodes */
      /*         and previous carry       */
      total = p->info + q->info + carry;
      /* Determine the low order five digits of */
      /*  the sum and insert into the list.     */
      number = total % hunthou;
      insafter(s, number);
      /*    advance the traversals              */
      s = s->next;
      p = p->next;
      q = q->next;
      /* determine whether there is a carry     */
      carry = total / hunthou;
   } /* end while */
   /* at this point, there may be nodes left in one of the */
   /*                two input lists                       */
   while (p->info != -1)  {
      total = p->info + carry;
      number = total % hunthou;
      insafter(s, number);
      carry = total / hunthou;
      s = s->next;
      p = p->next;
   } /* end while */
   while (q->info != -1)  {
      total = q->info + carry;
      number = total % hunthou;
      insafter(s, number);
      carry = total / hunthou;
      s = s->next;
      q = q->next;
   } /* end while */
   /*  check if there is an extra carry from the first */
   /*                 five digits                     */
   if (carry == 1)  {
      insafter(s, carry);
      s = s->next;
   } /* end if */
   /*  s points to the last node in the sum. s->next points to */
   /*       the header of the sum list.                        */
   return(s->next);
}  /* end addint */
```

Doubly Linked Lists

Although a circularly linked list has advantages over a linear list, it still has several drawbacks. One cannot traverse such a list backward, nor can a node be deleted from a circularly linked list, given only a pointer to that node. In cases where these facilities are required, the appropriate data structure is a ***doubly linked list***. Each node in such a list contains two pointers, one to its predecessor and another to its successor. In fact, in the context of doubly linked lists, the terms predecessor and successor are meaningless, since the list is entirely symmetric. Doubly linked lists may be either linear or circular and may or may not contain a header node, as illustrated in Figure 4.5.5.

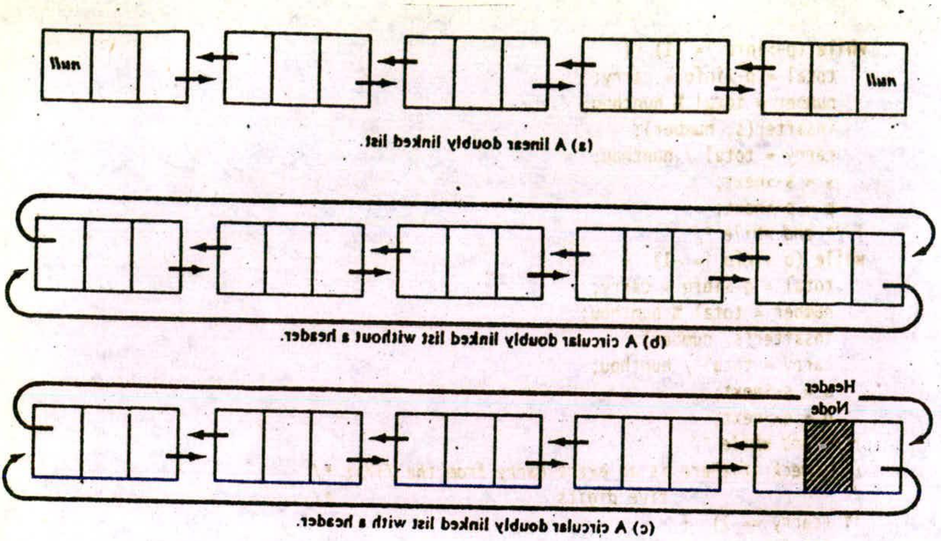
We may consider the nodes on a doubly linked list to consist of three fields: an *info* field that contains the information stored in the node, and *left* and *right* fields that contain pointers to the nodes on either side. We may declare a set of such nodes using either the array or dynamic implementation, by

Array Implementation

```
struct nodetype  {
   int info;
   int left, right;
};
struct nodetype node[NUMNODES];
```

Dynamic Implementation

```
struct node {
   int info;
   struct node *left, *right;
};
typedef struct node *NODEPTR;
```



(a) A linear doubly linked list.

(b) A circular doubly linked list without a header.

(c) A circular doubly linked list with a header.

Figure 4.5.5 Doubly linked lists.

Note that the available list for such a set of nodes in the array implementation need not be doubly linked, since it is not traversed bidirectionally. The available list may be linked together by using either the *left* or *right* pointer. Of course, appropriate *getnode* and *freenode* routines must be written.

We now present routines to operate on doubly linked circular lists. A convenient property of such lists is that if *p* is a pointer to any node, letting *left*(*p*) be an abbreviation for *node*[*p*].*left* or *p*-> *left*, and *right*(*p*) an abbreviation for *node*[*p*].*right* or *p*-> *right*, we have

```
left(right(p)) = p = right(left(p))
```

One operation that can be performed on doubly linked lists but not on ordinary linked lists is to delete a given node. The following C routine deletes the node pointed to by *p* from a doubly linked list and stores its contents in *x*, using the dynamic node implementation. It is called by *delete*(*p*, &*x*).

```
void delete(NODEPTR p, int *px)
{
   NODEPTR q, r;
   if (p == NULL)  {
      printf("void deletion\n");
      return;
   } /* end if */
```

```
      *px = p->info;
      q = p->left;
      r = p->right;
      q->right = r;
      r->left = q;
      freenode(p);
      return;
   } /* end delete */
```

The routine *insertright* inserts a node with information field *x* to the right of *node*(*p*) in a doubly linked list:

```
   void insertright(NODEPTR p, int x)
   {
      NODEPTR q, r;
      if (p == NULL)  {
         printf("void insertion\n");
         return;
      } /* end if */
      q = getnode();
      q->info = x;
      r = p->right;
      r->left = q;
      q->right = r;
      q->left = p;
      p->right = q;
      return;
   } /* end insertright */
```

A routine *insertleft* to insert a node with information field *x* to the left of *node*(*p*) in a doubly linked list is similar and is left as an exercise for the reader.

When space efficiency is a consideration, a program may not be able to afford the overhead of two pointers for each element of a list. There are several techniques for compressing the left and right pointers of a node into a single field. For example, a single pointer field *ptr* in each node can contain the sum of pointers to its left and right neighbors. (Here, we are assuming that pointers are represented in such a way that arithmetic can be performed on them readily. For example, pointers represented by array indexes can be added and subtracted. Although it is illegal to add two pointers in C, many compilers will allow such pointer arithmetic.) Given two external pointers, *p* and *q*, to two adjacent nodes such that $p == left(q)$, $right(q)$ can be computed as $ptr(q) - p$ and $left(p)$ can be computed as $ptr(p) - q$. Given *p* and *q*, it is possible to delete either node and reset its pointer to the preceding or succeeding node. It is also possible to insert a node to the left of *node*(*p*) or to the right of *node*(*q*) or to insert a node between *node*(*p*) and *node*(*q*) and reset either *p* or *q* to the newly inserted node. In using such a scheme, it is crucial always to maintain two external pointers to two adjacent nodes in the list.

Addition of Long Integers Using Doubly Linked Lists

As an illustration of the use of doubly linked lists, let us consider extending the list implementation of long integers to include negative as well as positive integers.

The header node of a circular list representing a long integer contains an indication of whether the integer is positive or negative.

To add a positive and a negative integer, the smaller absolute value must be subtracted from the larger absolute value and the result must be given the sign of the integer with the larger absolute value. Thus, some method is needed for testing which of two integers represented as circular lists has the larger absolute value.

The first criterion that may be used to identify the integer with the larger absolute value is the length of the integers (assuming that they do not contain leading 0s). The list with more nodes represents the integer with the larger absolute value. However, actually counting the number of nodes involves an extra traversal of the list. Instead of counting the number of nodes, the count could be kept as part of the header node and referenced as needed.

However, if both lists have the same number of nodes, the integer whose most significant digit is larger has the greater absolute value. If the leading digits of both integers are equal, it is necessary to traverse the lists from the most significant digit to the least significant to determine which number is larger. Note that this traversal is in the direction opposite that of the traversal used in actually adding or subtracting two integers. Since we must be able to traverse the lists in both directions, doubly linked lists are used to represent such integers.

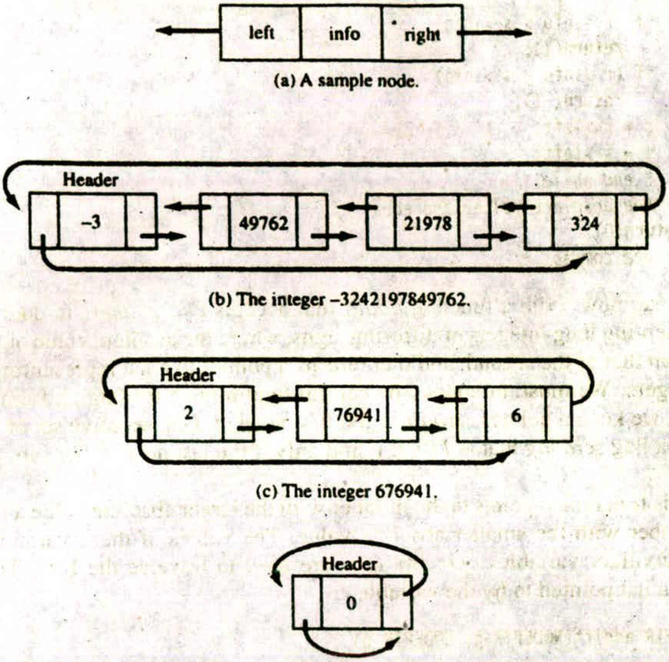
Consider the format of the header node. In addition to a right and left pointer, the header must contain the length of the list and an indication of whether the number is positive or negative. These two pieces of information can be combined into a single integer whose absolute value is the length of the list and whose sign is the sign of the number being represented. However, in so doing, the ability to identify the header node by examining the sign of its *info* field is destroyed. When a positive integer was represented as a singly linked circular list, an *info* field of -1 indicated a header node. Under the new representation, however, a header node may contain an *info* field such as 5 which is a valid *info* field for any other node in the list.

There are several ways to remedy this problem. One way is to add another field to each node to indicate whether or not it is a header node. Such a field could contain the logical value *TRUE* if the node is a header and *FALSE* if it is not. This means, of course, that each node would require more space. Alternatively, the count could be eliminated from the header node and an *info* field of -1 would indicate a positive number and -2 a negative number. A header node could then be identified by its negative *info* field. However, this would increase the time needed to compare two numbers, since it would be necessary to count the number of nodes in each list. Such space/time trade-offs are common in computing, and a decision must be made about which efficiency should be sacrificed and which retained.

In our case we choose yet a third option, which is to retain an external pointer to the list header. A pointer p can be identified as pointing to a header if it is equal to the original external pointer; otherwise *node*(p) is not a header.

Figure 4.5.6 indicates a sample node and the representation of four integers as doubly linked lists. Note that the least significant digits are to the right of the header and that the counts in the header nodes do not include the header node itself.

Using the preceding representation, we present a function *compabs* that compares the absolute values of two integers represented as doubly linked lists. Its two parameters



(a) A sample node.

(b) The integer −3242197849762.

(c) The integer 676941.

(d) The integer 0

Figure 4.5.6 Integers as doubly linked lists.

are pointers to the list headers and it returns 1 if the first has the greater absolute value, −1 if the second has the greater absolute value, and 0 if the absolute values of the two integers are equal.

```
int compabs(NODEPTR p, NODEPTR q)
{
   NODEPTR r, s;
   /* compare the counts    */
   if (abs(p->info) > abs(q->info))
      return(1);
   if (abs(p->info) < abs(q->info))
      return(-1);
   /* the counts are equal  */
   r = p->left;
   s = q->left;
   /* traverse the list from the most significant digits */
   while (r != p) {
      if (r->info > s->info)
         return(1);
      if (r->info < s->info)
         return(-1);
      r = r->left;
      s = s->left;
   } /* end while */
   /* the absolute values are equal */
   return(0);
} /* end compabs */
```

We may now write a function *addiff* that accepts two pointers to doubly linked lists representing long integers of differing signs, where the absolute value of the first is not less than that of the second, and that returns a pointer to a list representing the sum of the integers. We must, of course, be careful to eliminate leading 0s from the sum. To do this, we keep a pointer *zeroptr* to the first node of a consecutive set of leading-0 nodes and a flag *zeroflag* that is *TRUE* if and only if the last node of the sum generated so far is 0.

In this function, *p* points to the number with the larger absolute value and *q* points to the number with the smaller absolute value. The values of these variables do not change. Auxiliary variables *pptr* and *qptr* are used to traverse the lists. The sum is formed in a list pointed to by the variable *r*.

```
NODEPTR addiff(NODEPTR p, NODEPTR q)
{
   int count;
   NODEPTR pptr, qptr, r, s, zeroptr;
   long int hunthou = 100000L;
   long int borrow, diff;
   int zeroflag;
   /* initialize variables */
   count = 0;
   borrow = 0;
   zeroflag = FALSE;
   /* generate a header node for the sum */
   r = getnode();
   r->left = r;
   r->right = r;
   /* traverse the two lists
   pptr = p->right;
   qptr = q->right;
   while (qptr != q) {
      diff = pptr->info - borrow - qptr->info;
      if (diff >= 0)
         borrow = 0;
      else {
         diff = diff + hunthou;
         borrow = 1;
      } /* end if */
      /* generate a new node and insert it */
      /*   to the left of header in sum    */
      insertleft(r, diff);
      count += 1;
      /* test for zero node */
      if (diff == 0) {
         if (zeroflag == FALSE)
            zeroptr = r->left;
            zeroflag = TRUE;
      }
      else
         zeroflag = FALSE;
      pptr = pptr->right;
      qptr = qptr->right;
   } /* end while */
   /* traverse the remainder of the p list */
   while (pptr != p) {
      diff = pptr->info - borrow;
      if (diff >= 0)
         borrow = 0;
      else {
         diff = diff + hunthou;
         borrow = 1;
      } /* end if */
      insertleft(r, diff);
      count += 1;
      if (diff == 0) {
         if (zeroflag == FALSE)
            zeroptr = r->left;
         zeroflag = TRUE;
      }
      else
         zeroflag = FALSE;
       pptr = pptr->right;
   } /* end while */
   if (zeroflag == TRUE) /* delete leading zeros */
      while (zeroptr != r)  {
         s = zeroptr;
         zeroptr = zeroptr->right;
         delete(s, &diff);
         count -= 1;
      } /* end if...while */
   /* insert count and sign into the header */
   if (p->info > 0)
      r->info = count;
   else
      r->info = -count;
   return(r);
} /* end addiff */
```

We can also write a function *addsame* to add two numbers with like signs. This is very similar to the function *addint* of the previous implementation except that it deals with a doubly linked list and must keep track of the number of nodes in the sum.

Using these routines we can write a new version of *addint* that adds two integers represented by doubly linked lists.

```
NODEPTR addint(NODEPTR p, NODEPTR q)
{
   /* check if integers are of like sign */
   if (p->info * q->info > 0)
      return(addsame(p, q));
   /* check which has a larger absolute value */
   if (compabs(p, q) > 0)
      return(addiff(p, q));
   else
      return(addiff(q, p));
} /* end addint */
```

EXERCISES

4.5.1. Write an algorithm and a C routine to perform each of the operations of Exercise 4.2.3 for circular lists. Which are more efficient on circular lists than on linear lists? Which are less efficient?

4.5.2. Rewrite the routine *place* of Section 4.3 to insert a new item in an ordered circular list.

4.5.3. Write a program to solve the Josephus problem by using an array rather than a circular list. Why is a circular list more efficient?

4.5.4. Consider the following variation of the Josephus problem. A group of people stand in a circle and each chooses a positive integer. One of their names and a positive integer *n* are chosen. Starting with the person whose name is chosen, they count around the circle clockwise and eliminate the *n*th person. The positive integer that that person chose is then used to continue the count. Each time that a person is eliminated, the number that he or she chose is used to determine the next person eliminated. For example, suppose that the five people are *A*, *B*, *C*, *D*, and *E* and that they choose integers 3, 4, 6, 2, and 7, respectively, and that the integer 2 is initially chosen. Then if we start from *A*, the order in which people are eliminated from the circle is *B*, *A*, *E*, *C*, leaving *D* as the last one in the circle.

Write a program that reads a group of input lines. Each input line except the first and last contains a name and a positive integer chosen by that person. The order of the names in the data is the clockwise ordering of the people in the circle, and the count is to start with the first name in the input. The first input line contains the number of people in the circle. The last input line contains only a single positive integer representing the initial count. The program prints the order in which the people are eliminated from the circle.

4.5.5. Write a C function *multint*(*p*, *q*) to multiply two long positive integers represented by singly linked circular lists.

4.5.6. Write a program to print the 100th Fibonacci number.

4.5.7. Write an algorithm and a C routine to perform each of the operations of Exercise 4.2.3 for doubly linked circular lists. Which are more efficient on doubly linked than on singly linked lists? Which are less efficient?

4.5.8. Assume that a single pointer field in each node of a doubly linked list contains the sum of pointers to the node's predecessor and successor, as described in the text. Given pointers p and q to two adjacent nodes in such a list, write C routines to insert a node to the right of *node*(q), to the left of *node*(p), and between *node*(p) and *node*(q) modifying p to point to the newly inserted node. Write an additional routine to delete *node*(q), resetting q to the node's successor.

4.5.9. Assume that *first* and *last* are external pointers to the first and last nodes of a doubly linked list represented as in Exercise 4.5.8. Write C routines to implement the operations of Exercise 4.2.3 for such a list.

4.5.10. Write a routine *addsame* to add two long integers of the same sign represented by doubly linked lists.

4.5.11. Write a C function *multint*(p, q) to multiply two long integers represented by doubly linked circular lists.

4.5.12. How can a polynomial in three variables (x, y, and z) be represented by a circular list? Each node should represent a term and should contain the powers of x, y, and z as well as the coefficient of that term. Write C functions to do the following.

(a) Add two such polynomials.

(b) Multiply two such polynomials.

(c) Take the partial derivative of such a polynomial with respect to any of its variables.

(d) Evaluate such a polynomial for given values of x, y, and z.

(e) Divide one such polynomial by another, creating a quotient and a remainder polynomial.

(f) Integrate such a polynomial with respect to any of its variables.

(g) Print the representation of such a polynomial.

(h) Given four such polynomials $f(x,y,z)$, $g(x,y,z)$, $h(x,y,z)$ and $i(x,y,z)$, compute the polynomial $f(g(x,y,z), h(x,y,z), i(x,y,z))$.

4.6 LINKED LISTS IN C++

We now examine the implementation of linked lists in C++. We will look at singly linked linear lists and leave the details of circular and doubly linked lists to the reader.

Before going into details about lists in C++, we introduce the built-in C++ mechanism for allocating and freeing objects of a given type. If T is the name of a type, then the expression

```
new T
```

creates a new object of type T and returns a pointer to the newly created object. If T is a class with a constructor with no parameters, then the object created is also automatically initialized. If T is a class with a constructor with n parameters, then the expression

```
new T(p1, p2, ..., pn)
```

creates an object of type T, initializes it using the constructor with parameters $p1$ through pn, and returns a pointer to it.

If *p* points to an object created via use of the ***new*** operator, then the statement

```
delete p;
```

deallocates the object to which *p* was pointing. If the type has a destructor, the destructor is invoked prior to the deallocation.

Now we can turn to a discussion of lists in C++. We envision a linked list as a data structure, with a fixed set of public operations on the list. This means that the user accesses the list as a whole and is unable to access individual nodes within the list and individual pointers to those nodes. If a particular operation is desired on the list, it must be included in the public interface of the list class.

For example, the following might be the class definition for a linked list of integers, with the following operations:

1. Initialize a list to the empty list. This is a constructor, automatically invoked when a list is defined or created.
2. Free the nodes of a list. This is a destructor, automatically invoked when a list is freed or the block in which it is declared is exited.
3. Determine whether a list is empty.
4. Add a node with a given value into the list following the first node with another given value.
5. Add a node with a given value to the front of the list. This is the *push* operator.
6. Delete the first node with a given value from the list.
7. Delete the first node from the list. This is the *pop* operator.

The class definition follows:

```
class List {
protected:
     struct node {
          int info;
          struct node *next;
     }
     typedef struct node *NODEPTR;
     NODEPTR listptr;       // the pointer to the first node
                            // of the list
public:
     List();
     ~List();
     int emptylist();
     void insertafter(int oldvalue, int newvalue);
     void push(int newvalue);
     void delete(int oldvalue);
     int pop();
}
```

We now present the implementation of these routines:

List is a constructor that initializes a newly created list to the empty list.

```
List::List() {
    listptr = 0;
}
```

~ *List* is the destructor that traverses the nodes of a list, freeing them one by one.

```
List::~List() {
    NODEPTR p, q;
    if (emptylist())
        return 0;
    for (p = listptr, q = p->next; p! = 0; p = q, q = p->next)
        delete p;
}
```

emptylist determines if a list is empty.

```
int List::emptylist() {
    return(listptr == 0);
}
```

insertafter(oldvalue, newvalue) searches for the first occurrence of the value *oldvalue* in the list and inserts a new node with value *newvalue* following the node containing *oldvalue*.

```
List::insertafter(int oldvalue, int newvalue) {
    NODEPTR p, q;
    for (p = listptr; p != 0 && p->info !=  oldvalue; p  = p->next)
        ;
    if (p == 0)
        error("ERROR: value sought is not on the list.");
    q = new node;
    q->info = newvalue;
    q->next = p->next;
    p->next = q;
}
```

push(newvalue) adds a new node with a given value to the front of the list.

```
List::push(int newvalue) {
    NODEPTR p;
    p = new node;
    p->info = newvalue;
    p->next = listptr;
    listptr = p;
}
```

delete(oldvalue) deletes the first node containing the value *oldvalue* from the list.

```
List::delete(int oldvalue) {
      NODEPTR p, q;
      for (q=0, p=listptr; p!=0 && p->info!=oldvalue; q=p, p=p->next)
            ;
      if (p == 0)
            error("ERROR: value sought is not on the list.");
      if (q == 0)
          listptr=p->next;
      else
          q->next=p->next;
      delete p;
}
```

Finally, *pop* deletes the first node on the list and returns its contents.

```
int List::pop() {
      NODEPTR p;
      int x;
      if (emptytlist())
         error("ERROR: the list is empty.");
      p =listptr;
      listptr = p->next;
      x = p->info;
      delete p;
      return x;
}
```

Note that the *List* class does not permit the user to manipulate the nodes of the list; everything must be done via a method of *List* on the entire list.

EXERCISES

4.6.1. Modify the *List* class so that it uses a template and can be instantiated to implement a list of any type, not just integer. What problems may occur if you instantiate a list of lists?

4.6.2. Write a class *OrderedList* to implement a sorted list into which elements can only be inserted in their proper place. Can *OrderedList* be a descendant of *List*?

4.6.3. Add a method *insertafter*2(**int** *oldvalue*, **int** *n*, **int** *newvalue*) that inserts a node with value *newvalue* after the *n*th occurrence of *oldvalue*.

4.6.4. Write a class *CircList* to implement a circular list.

4.6.5. Write a class *DoubleList* to implement a doubly linked list.

5

Trees

In this chapter we consider a data structure that is useful in many applications: the tree. We define several different forms of this data structure and show how they can be represented in C and how they can be applied to solving a wide variety of problems. As with lists, we treat trees primarily as data structures rather than as data types. That is, we are primarily concerned with implementation, rather than mathematical definition.

5.1 BINARY TREES

A ***binary tree*** is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the ***root*** of the tree. The other two subsets are themselves binary trees, called the ***left*** and ***right subtrees*** of the original tree. A left or right subtree can be empty. Each element of a binary tree is called a ***node*** of the tree.

A conventional method of picturing a binary tree is shown in Figure 5.1.1. This tree consists of nine nodes with *A* as its root. Its left subtree is rooted at *B* and its right subtree is rooted at *C*. This is indicated by the two branches emanating from *A*: to *B* on the left and to *C* on the right. The absence of a branch indicates an empty subtree. For example, the left subtree of the binary tree rooted at *C* and the right subtree of the binary tree rooted at *E* are both empty. The binary trees rooted at *D*, *G*, *H*, and *I* have empty right and left subtrees.

Figure 5.1.2 illustrates some structures that are not binary trees. Be sure that you understand why each of them is not a binary tree as just defined.

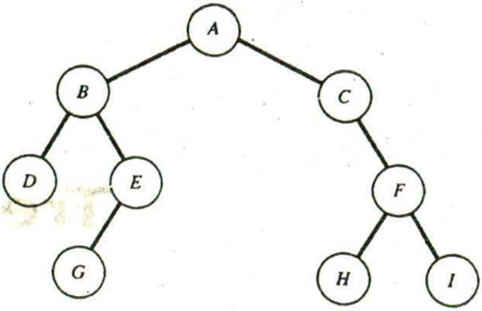
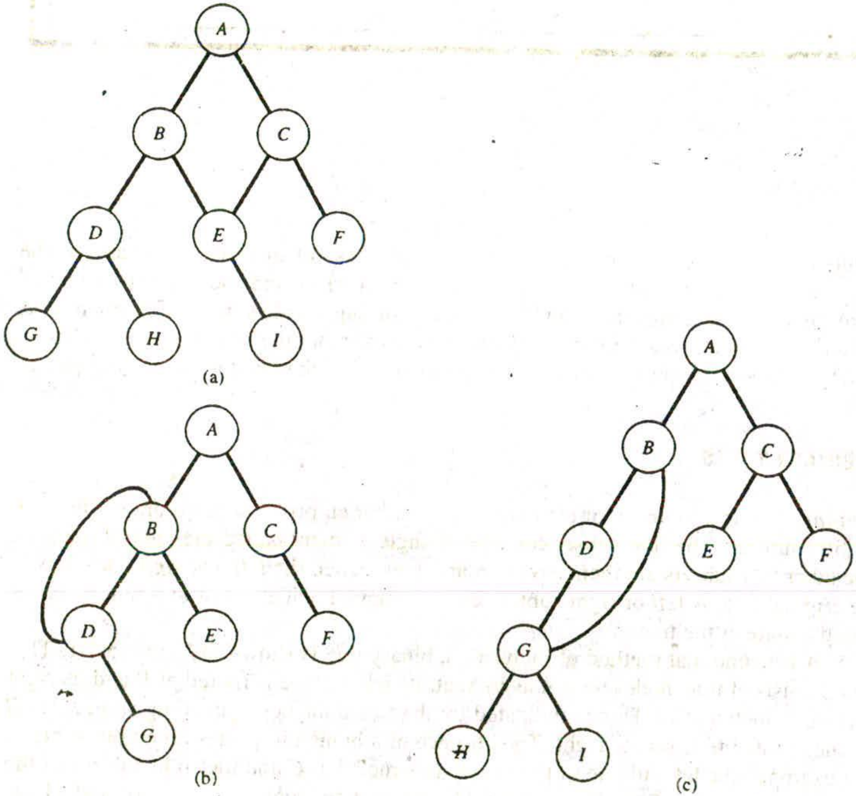


Figure 5.1.1 Binary tree.



(a)

(b)

(c)

Figure 5.1.2 Structures that are not binary trees.

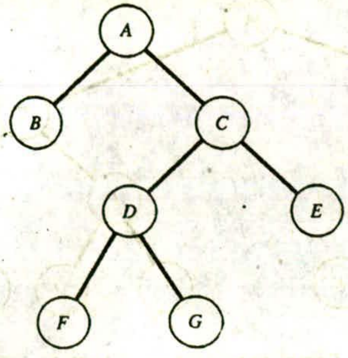


Figure 5.1.3 Strictly binary tree.

If A is the root of a binary tree and B is the root of its left or right subtree, then A is said to be the ***father*** of B and B is said to be the ***left*** or ***right son*** of A. A node that has no sons (such as D, G, H, or I of Figure 5.1.1) is called a ***leaf***. Node $n1$ is an ***ancestor*** of node $n2$ (and $n2$ is a ***descendant*** of $n1$) if $n1$ is either the father of $n2$ or the father of some ancestor of $n2$. For example, in the tree of Figure 5.1.1, A is an ancestor of G, and H is a descendant of C, but E is neither an ancestor nor a descendant of C. A node $n2$ is a ***left descendant*** of node $n1$ if $n2$ is either the left son of $n1$ or a descendant of the left son of $n1$. A ***right descendant*** may be similarly defined. Two nodes are ***brothers*** if they are left and right sons of the same father.

Although natural trees grow with their roots in the ground and their leaves in the air, computer scientists almost universally portray tree data structures with the root at the top and the leaves at the bottom. The direction from the root to the leaves is "down" and the opposite direction is "up." Going from the leaves to the root is called "climbing" the tree, and going from the root to the leaves is called "descending" the tree.

If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is termed a ***strictly binary tree***. Thus the tree of Figure 5.1.3 is strictly binary, whereas that of Figure 5.1.1 is not (because nodes C and E have one son each). A strictly binary tree with n leaves always contains $2n - 1$ nodes. The proof of this fact is left as an exercise for the reader.

The ***level*** of a node in a binary tree is defined as follows: The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father. For example, in the binary tree of Figure 5.1.1, node E is at level 2 and node H is at level 3. The ***depth*** of a binary tree is the maximum level of any leaf in the tree. This equals the length of the longest path from the root to any leaf. Thus the depth of the tree of Figure 5.1.1 is 3. A ***complete binary tree*** of depth d is the strictly binary tree all of whose leaves are at level d. Figure 5.1.4 illustrates the complete binary tree of depth 3.

If a binary tree contains m nodes at level l, it contains at most $2m$ nodes at level $l+1$. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at level l. A complete binary tree of depth d is the binary tree of depth d that contains exactly 2^l nodes at each level l between 0 and d. (This is equivalent to saying that it is the binary tree of depth d that contains exactly 2^d nodes at level d.) The total number of nodes in a complete binary tree of depth d, tn, equals the sum of the number of nodes at each level between 0 and d. Thus

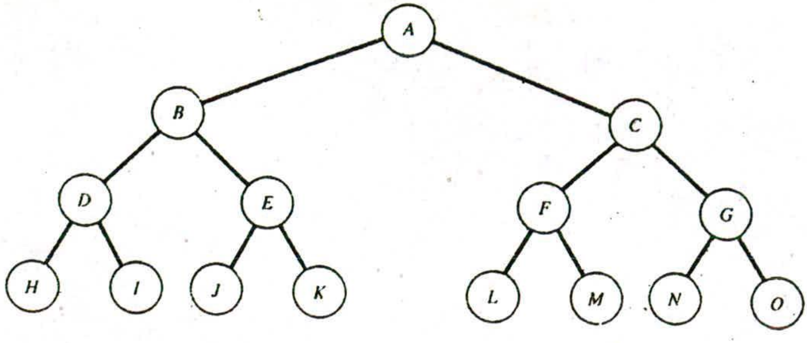


Figure 5.1.4 Complete binary tree of depth 3.

$$tn = 2^0 + 2^1 + 2^2 + \cdots + 2^d = \sum_{j=0}^{d} 2^j$$

By induction, it can be shown that this sum equals $2^{d+1} - 1$. Since all leaves in such a t ee are at level d, the tree contains 2^d leaves and, therefore, $2^d - 1$ nonleaf nodes.

Similarly, if the number of nodes, tn, in a complete binary tree is known, we can compute its depth, d, from the equation $tn = 2^{d+1} - 1$. d equals 1 less than the number of times 2 must be multiplied by itself to reach $tn + 1$. In mathematics, $\log_b x$ is defined as the number of times b must be multiplied by itself to reach x. Thus we may say that, in a complete binary tree, d equals $\log_2(tn + 1) - 1$. For example, the complete binary tree of Figure 5.1.4 contains 15 nodes and is of depth 3. Note that 15 equals $2^{3+1} - 1$ and that 3 equals $\log_2(15 + 1) - 1$. $\log_2 x$ is much smaller than x [for example, $\log_2 1024$ equals 10 and $\log_2 1000000$ is less than 20]. The significance of a complete binary tree is that it is the binary tree with the maximum number of nodes for a given depth. Put another way, although a complete binary tree contains many nodes, the distance from the root to any leaf (the tree's depth) is relatively small.

A binary tree of depth d is an ***almost complete binary tree*** if:

1. Any node nd at level less than $d - 1$ has two sons.
2. For any node nd in the tree with a right descendant at level d, nd must have a left son and every left descendant of nd is either a leaf at level d or has two sons.

The strictly binary tree of Figure 5.1.5a is not almost complete, since it contains leaves at levels 1, 2, and 3, thereby violating condition 1. The strictly binary tree of Figure 5.1.5b satisfies condition 1, since every leaf is either at level 2 or at level 3. However, condition 2 is violated, since A has a right descendant at level 3 (J) but also has a left descendant that is a leaf at level 2 (E). The strictly binary tree of Figure 5.1.5c satisfies both conditions 1 and 2 and is therefore an almost complete binary tree. The binary tree of Figure 5.1.5d is also an almost complete binary tree but is not strictly binary, since node E has a left son but not a right son. (We should note that many texts refer to such a tree as a "complete binary tree" rather than as an "almost complete binary tree."

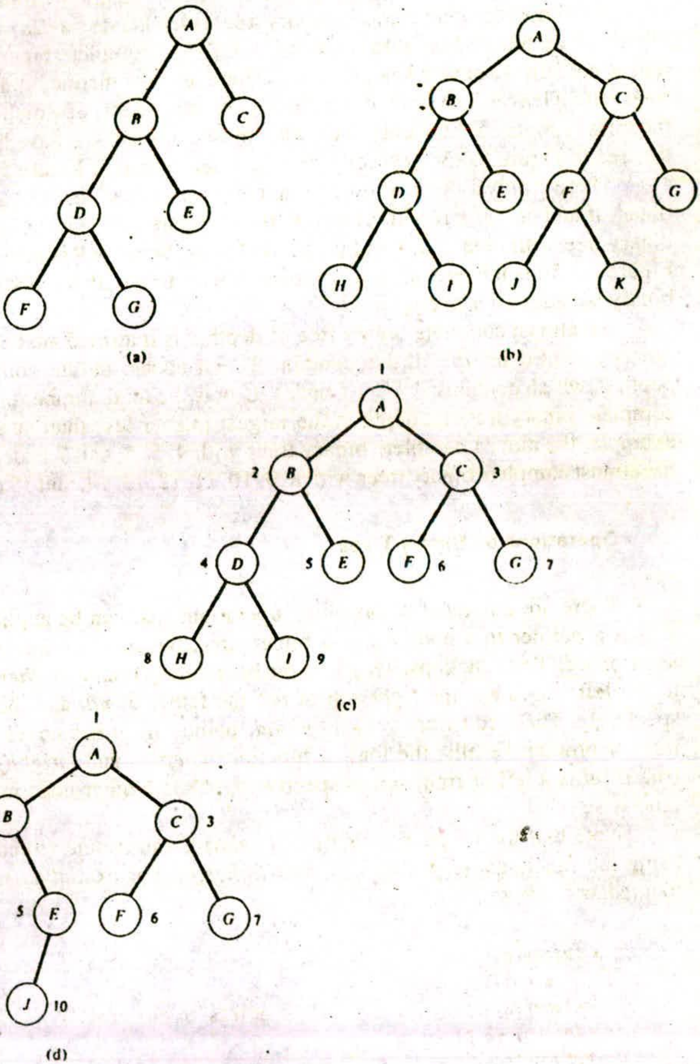


Figure 5.1.5 Node numbering for almost complete binary trees.

Still other texts use the term "complete" or "fully binary" to refer to the concept that we call "strictly binary." We use the terms "strictly binary," "complete," and "almost complete" as we have defined them here.)

The nodes of an almost complete binary tree can be numbered so that the root is assigned the number 1, a left son is assigned twice the number assigned its father, and a right son is assigned one more than twice the number assigned its father. Figure 5.1.5c and d illustrate this numbering technique. Each node in an almost complete binary tree is assigned a unique number that defines the node's position within the tree.

An almost complete strictly binary tree with n leaves has $2n-1$ nodes, as does any other strictly binary tree with n leaves. An almost complete binary tree with n leaves that is not strictly binary has $2n$ nodes. There are two distinct almost complete binary trees with n leaves, one of which is strictly binary and one of which is not. For example, the trees of Figure 5.1.5c and d are both almost complete and have five leaves; however, the tree of Figure 5.1.5c is strictly binary, whereas that of Figure 5.1.5d is not.

There is only a single almost complete binary tree with n nodes. This tree is strictly binary if and only if n is odd. Thus the tree of Figure 5.1.5c is the only almost complete binary tree with nine nodes and is strictly binary because 9 is odd, whereas the tree of Figure 5.1.5d is the only almost complete binary tree with ten nodes and is not strictly binary because 10 is even.

An almost complete binary tree of depth d is intermediate between the complete binary tree of depth $d-1$, that contains $2^d - 1$ nodes, and the complete binary tree of depth d, which contains $2^{d+1} - 1$ nodes. If tn is the total number of nodes in an almost complete binary tree, its depth is the largest integer less than or equal to $\log_2 tn$. For example, the almost complete binary trees with 4, 5, 6, and 7 nodes have depth 2, and the almost complete binary trees with 8, 9, 10, 11, 12, 13, 14, and 15 nodes have depth 3.

Operations on Binary Trees

There are a number of primitive operations that can be applied to a binary tree. If p is a pointer to a node nd of a binary tree, the function *info*(p) returns the contents of nd. The functions *left*(p), *right*(p), *father*(p), and *brother*(p) return pointers to the left son of nd, the right son of nd, the father of nd, and the brother of nd, respectively. These functions return the *null* pointer if nd has no left son, right son, father, or brother. Finally, the logical functions *isleft*(p) and *isright*(p) return the value *true* if nd is a left or right son, respectively, of some other node in the tree, and *false* otherwise.

Note that the functions *isleft*(p), *isright*(p), and *brother*(p) can be implemented using the functions *left*(p), *right*(p) and *father*(p). For example, *isleft* may be implemented as follows:

```
q = father(p);
if (q == null)
   return(false);                /* p points to the root */
if (left(q) == p)
   return(true);
return(false);
```

or, even simpler, as *father(p) && p == left(father(p))*. *isright* may be implemented in a similar manner, or by calling *isleft*. *brother(p)* may be implemented using *isleft* or *isright* as follows:

```
if (father(p) == null)
   return(null);                  /* p points to the root */
if (isleft(p))
   return(right(father(p)));
return(left(father(p)));
```

In constructing a binary tree, the operations *maketree, setleft,* and *setright* are useful. *maketree(x)* creates a new binary tree consisting of a single node with information field *x* and returns a pointer to that node. *setleft(p,x)* accepts a pointer *p* to a binary tree node with no left son. It creates a new left son of *node(p)* with information field *x*. *setright(p,x)* is analogous to *setleft* except that it creates a right son of *node(p)*.

Applications of Binary Trees

A binary tree is a useful data structure when two-way decisions must be made at each point in a process. For example, suppose that we wanted to find all duplicates in a list of numbers. One way of doing this is to compare each number with all those that precede it. However, this involves a large number of comparisons.

The number of comparisons can be reduced by using a binary tree. The first number in the list is placed in a node that is established as the root of a binary tree with empty left and right subtrees. Each successive number in the list is then compared to the number in the root. If it matches, we have a duplicate. If it is smaller, we examine the left subtree; if it is larger, we examine the right subtree. If the subtree is empty, the number is not a duplicate and is placed into a new node at that position in the tree. If the subtree is nonempty, we compare the number to the contents of the root of the subtree and the entire process is repeated with the subtree. An algorithm for doing this follows.

```
/* read the first number and insert it */
/*   into a single-node binary tree    */
scanf("%d", &number);
tree = maketree(number);
while (there are numbers left in the input) {
scanf("%d", &number);
p = q = tree;
while (number != info(p) && q != NULL) {
   p = q;
   if (number < info(p))
      q = left(p);
   else
      q = right(p);
} /* end while */
if (number==info(p))
   printf("%d %s\n", number, "is a duplicate");
/* insert number to the right or left of p */
```

```
    else if (number < info(p))
       setleft(p, number);
    else
       setright(p, number);
    }  /* end while */
```

Figure 5.1.6 illustrates the tree constructed from the input 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5.

Another common operation is to ***traverse*** a binary tree; that is, to pass through the tree, enumerating each of its nodes once. We may simply wish to print the contents of each node as we enumerate it, or we may wish to process it in some other fashion. In either case, we speak of ***visiting*** each node as it is enumerated.

The order in which the nodes of a linear list are visited in a traversal is clearly from first to last. However, there is no such "natural" linear order for the nodes of a tree. Thus, different orderings are used for traversal in different cases. We shall define three of these traversal methods. In each of these methods, nothing need be done to traverse an empty binary tree. The methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among the methods is the order in which these three operations are performed.

To traverse a nonempty binary tree in ***preorder*** (also known as ***depth-first order***), we perform the following three operations:

1. Visit the root.
2. Traverse the left subtree in oreorder.
3. Traverse the right subtree in preorder.

To traverse a nonempty binary tree in ***inorder*** (or ***symmetric order***):

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

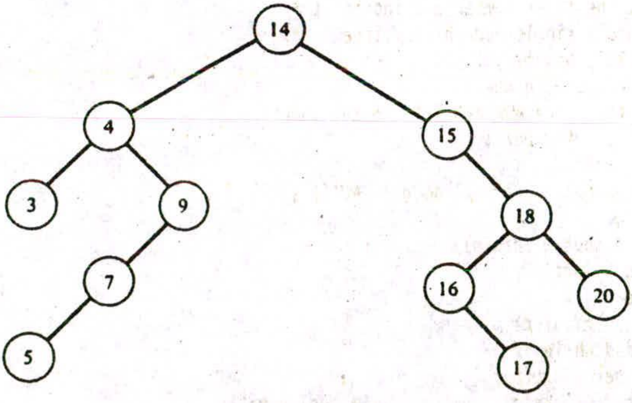
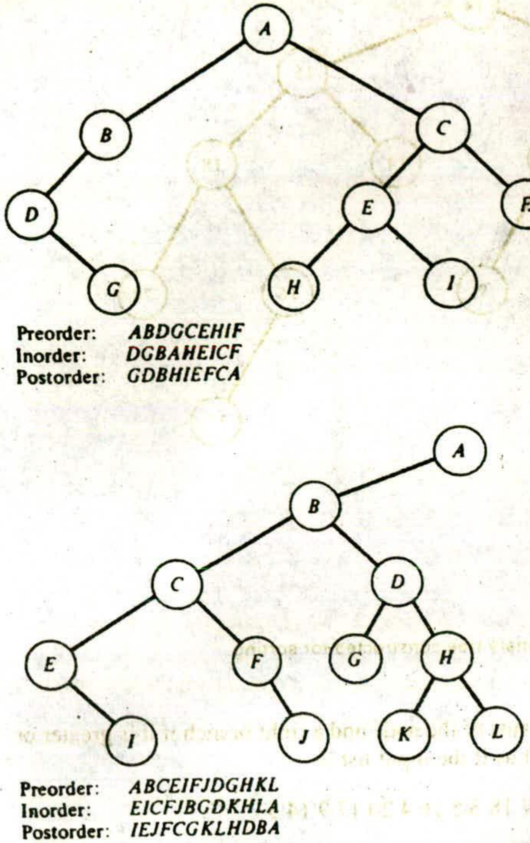


Figure 5.1.6 Binary tree constructed for finding duplicates.



Preorder: *ABDGCEHIF*
Inorder: *DGBAHEICF*
Postorder: *GDBHIEFCA*

Preorder: *ABCEIFJDGHKL*
Inorder: *EICFJBGDKHLA*
Postorder: *IEJFCGKLHDBA*

Figure 5.1.7 Binary trees and their traversals.

To traverse a nonempty binary tree in ***postorder***:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

Figure 5.1.7 illustrates two binary trees and their traversals in preorder, inorder, and postorder.

Many algorithms that use binary trees proceed in two phases. The first phase builds a binary tree, and the second traverses the tree. As an example of such an algorithm, consider the following sorting method. Given a list of numbers in an input file, we wish to print them in ascending order. As we read the numbers, they can be inserted into a binary tree such as the one of Figure 5.1.6. However, unlike the previous algorithm used to find duplicates, duplicate values are also placed in the tree. When a number is compared with the contents of a node in the tree, a left branch is taken if

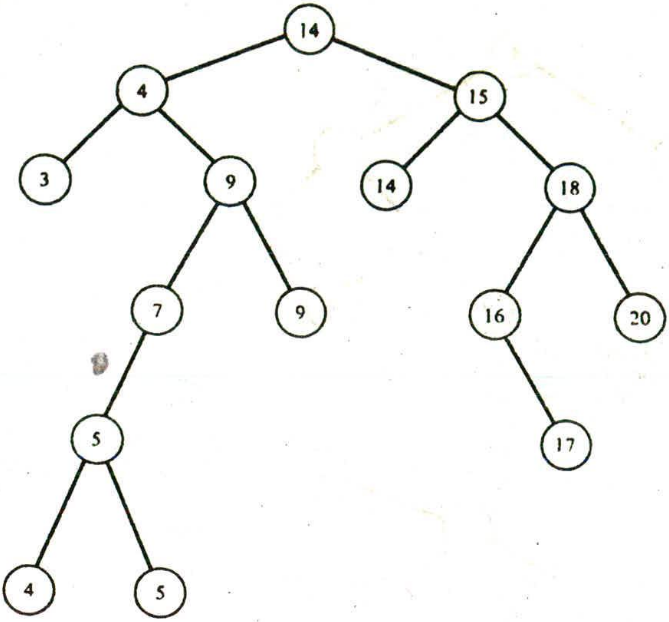


Figure 5.1.8 Binary tree constructed for sorting.

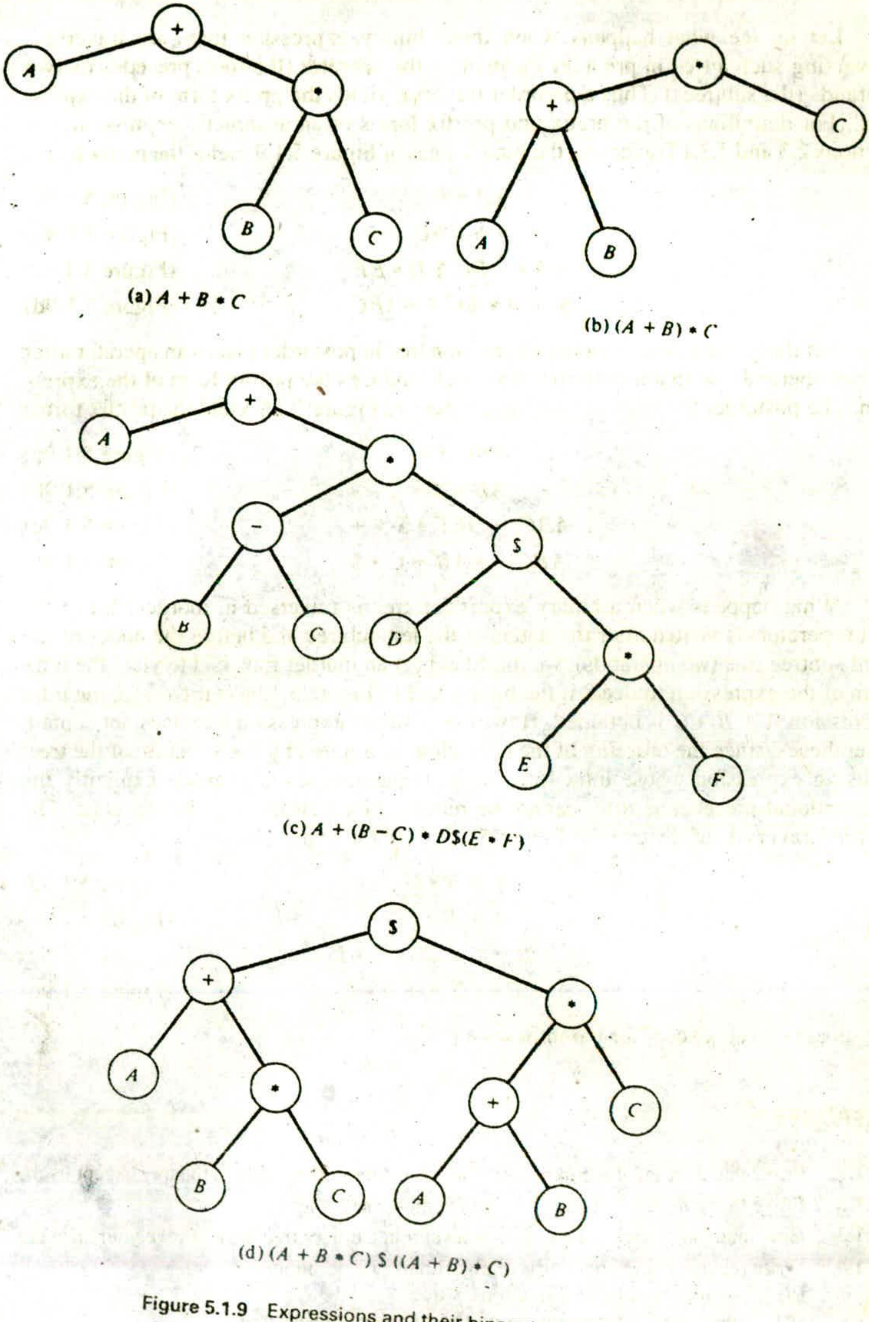
the number is smaller than the contents of the node and a right branch if it is greater or equal to the contents of the node. Thus if the input list is

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5

the binary tree of Figure 5.1.8 is produced.

Such a binary tree has the property that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree of n are greater than or equal to the contents of n. A binary tree that has this property is called a ***binary search tree***. If a binary search tree is traversed in inorder (left, root, right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order. Convince yourself that this is the case for the binary search tree of Figure 5.1.8. Binary search trees and their use in sorting and searching are discussed further in Sections 6.3 and 7.2.

As another application of binary trees, consider the following method of representing an expression containing operands and binary operators by a strictly binary tree. The root of the strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees. A node representing an operator is a nonleaf, whereas a node representing an operand is a leaf. Figure 5.1.9 illustrates some expressions and their tree representations. (The character "$" is again used to represent exponentiation.)



(a) A + B * C

(b) (A + B) * C

(c) A + (B − C) * D$(E * F)

(d) (A + B * C) $ ((A + B) * C)

Figure 5.1.9 Expressions and their binary tree representation.

Let us see what happens when these binary expression trees are traversed. Traversing such a tree in preorder means that the operator (the root) precedes its two operands (the subtrees). Thus a preorder traversal yields the prefix form of the expression. (For definitions of the prefix and postfix forms of an arithmetic expression, see Sections 2.3 and 3.3.) Traversing the binary trees of Figure 5.1.9 yields the prefix forms

$+A * BC$ (Figure 5.1.9a)

$* + ABC$ (Figure 5.1.9b)

$+A * -BC \$ D * EF$ (Figure 5.1.9c)

$\$ + A * BC * +ABC$ (Figure 5.1.9d)

Similarly, traversing a binary expression tree in postorder places an operator after its two operands, so that a postorder traversal produces the postfix form of the expression. The postorder traversals of the binary trees of Figure 5.1.9 yield the postfix forms

$ABC * +$ (Figure 5.1.9a)

$AB + C*$ (Figure 5.1.9b)

$ABC - DEF * \$ * +$ (Figure 5.1.9c)

$ABC * +AB + C * \$$ (Figure 5.1.9d)

What happens when a binary expression tree is traversed in inorder? Since the root (operator) is visited after the nodes of the left subtree and before the nodes of the right subtree (the two operands), we might expect an inorder traversal to yield the infix form of the expression. Indeed, if the binary tree of Figure 5.1.9a is traversed, the infix expression $A + B * C$ is obtained. However. a binary expression tree does not contain parentheses, since the ordering of the operations is implied by the structure of the tree. Thus an expression whose infix form requires parentheses to override explicitly the conventional precedence rules cannot be retrieved by a simple inorder traversal. The inorder traversals of the trees of Figure 5.1.9 yield the expressions

$A + B * C$ (Figure 5.1.9a)

$A + B * C$ (Figure 5.1.9b)

$A + B - C * D\$ * E * F$ (Figure 5.1.9c)

$A + B * C\$ A + B * C$ (Figure 5.1.9d)

which are correct except for parentheses.

EXERCISES

5.1.1. Prove that the root of a binary tree is an ancestor of every node in the tree except itself.

5.1.2. Prove that a node of a binary tree has at most one father.

5.1.3. How many ancestors does a node at level n in a binary tree have? Prove your answer.

5.1.4. Write recursive and nonrecursive algorithms to determine:

(a) The number of nodes in a binary tree

(b) The sum of the contents of all the nodes in a binary tree

(c) The depth of a binary tree

5.1.5. Write an algorithm to determine if a binary tree is

(a) Strictly binary

(b) Complete

(c) Almost complete

5.1.6. Prove that a strictly binary tree with n leaves contains $2n - 1$ nodes.

5.1.7. Given a strictly binary tree with n leaves, let *level(i)* for i between 1 and n equal the level of the ith leaf. Prove that

$$\sum_{i=1}^{n} \frac{1}{2^{level(i)}} = 1$$

5.1.8. Prove that the nodes of an almost complete strictly binary tree with n leaves can be numbered from 1 to $2n - 1$ in such a way that the number assigned to the left son of the node numbered i is $2i$ and the number assigned to the right son of the node numbered i is $2i + 1$.

5.1.9. Two binary trees are ***similar*** if they are both empty or if they are both nonempty, their left subtrees are similar, and their right subtrees are similar. Write an algorithm to determine if two binary trees are similar.

5.1.10. Two binary trees are ***mirror similar*** if they are both empty or if they are both nonempty and the left subtree of each is mirror similar to the right subtree of the other. Write an algorithm to determine if two binary trees are mirror similar.

5.1.11. Write algorithms to determine whether or not one binary tree is similar and mirror similar (see the previous exercises) to some subtree of another.

5.1.12. Develop an algorithm to find duplicates in a list of numbers without using a binary tree. If there are n distinct numbers in the list, how many times must two numbers be compared for equality in your algorithm? What if all n numbers are equal?

5.1.13. **(a)** Write an algorithm that accepts a pointer to a binary search tree and deletes the smallest element from the tree.

(b) Show how to implement an ascending priority queue (see Section 4.1) as a binary search tree. Present algorithms for the operations *pqinsert* and *pqmindelete* on a binary search tree.

5.1.14. Write an algorithm that accepts a binary tree representing an expression and returns the infix version of the expression that contains only those parentheses that are necessary.

5.2 BINARY TREE REPRESENTATIONS

In this section we examine various methods of implementing binary trees in C and present routines that build and traverse binary trees. We also present some additional applications of binary trees.

Node Representation of Binary Trees

As is the case with list nodes, tree nodes may be implemented as array elements or as allocations of a dynamic variable. Each node contains *info*, *left*, *right*, and *father* fields. The *left*, *right*, and *father* fields of a node point to the node's left son, right son, and father, respectively. Using the array implementation, we may declare

```
#define NUMNODES 500
struct nodetype {
   int info;
   int left;
   int right;
   int father;
};
struct nodetype node[NUMNODES];
```

Under this representation, the operations *info(p)*, *left(p)*, *right(p)*, and *father(p)* are implemented by references to *node[p].info*, *node[p].left*, *node[p].right*, and *node[p].father*, respectively. The operations *isleft(p)*, *isright(p)*, and *brother(p)* can be implemented in terms of the operations *left(p)*, *right(p)*, and *father(p)*, as described in the preceding section.

To implement *isleft* and *isright* more efficiently, we can also include within each node an additional flag *isleft*. The value of this flag is *TRUE* if the node is a left son and *FALSE* otherwise. The root is uniquely identified by a *NULL* value (-1) in its *father* field. The external pointer to a tree usually points to its root.

Alternatively, the sign of the *father* field could be negative if the node is a left son or positive if it is a right son. The pointer to a node's father is then given by the absolute value of the *father* field. The *isleft* or *isright* operations would then need only examine the sign of the *father* field.

To implement *brother(p)* more efficiently, we can also include an additional *brother* field in each node.

Once the array of nodes is declared, we could create an available list by executing the following statements:

```
int avail, i;

{
   avail = 1;
   for (i=0; i < NUMNODES; i++)
      node[i].left = i + 1;
   node[NUMNODES-1].left = 0;
}
```

The functions *getnode* and *freenode* are straightforward and are left as exercises. Note that the available list is not a binary tree but a linear list whose nodes are linked together by the *left* field. Each node in a tree is taken from the available pool when needed and returned to the available pool when no longer in use. This representation is called the ***linked array representation*** of a binary tree.

Alternatively, a node may be defined by

```
struct nodetype {
   int info;
   struct nodetype *left;
   struct nodetype *right;
   struct nodetype *father;
};
typedef struct nodetype *NODEPTR;
```

The operations *info(p)*, *left(p)*, *right(p)*, and *father(p)* would be implemented by references to *p->info*, *p->left*, *p->right*, and *p->father*, respectively. Under this implementation, an explicit available list is not needed. The routines *getnode* and *freenode* simply allocate and free nodes using the routines *malloc* and *free*. This representation is called the ***dynamic node representation*** of a binary tree.

Both the linked array representation and the dynamic node representation are implementations of an abstract ***linked representation*** (also called the ***node representation***) in which explicit pointers link together ihe nodes of a binary tree.

We now present C implementations of the binary tree operations under the dynamic node representation and leave the linked array implementations as simple exercises for the reaaer. The *maketree* function, which allocates a node and sets it as the root of a single-node binary tree, may be written as follows:

```
NODEPTR maketree(int x)
{
   NODEPTR p;

   p = getnode();
   p->info = x;
   p->left = NULL;
   p->right = NULL;
   return(p);
} /* end maketree */
```

The routine *setleft(p,x)* sets a node with contents *x* as the left son of *node(p)*:

```
void setleft(NODEPTR p, int x)
{
   if (p == NULL)
      printf("void insertion\n");
   else if (p->left != NULL)
      printf ("invalid insertion\n");
   else
      p->left = maketree(x);
} /* end setleft */
```

The routine *setright(p,x)* to create a right son of *node(p)* with contents *x* is similar and is left as an exercise for the reader.

It is not always necessary to use *father, left,* and *right* fields. If a tree is always traversed in downward fashion (from the root to the leaves), the *father* operation is never used; in that case, a *father* field is unnecessary. For example, preorder, inorder, and postorder traversal do not use the *father* field. Similarly, if a tree is always traversed in upward fashion (from the leaves to the root), *left* and *right* fields are not needed. The *isleft* and *isright* operations could be implemented even without *left* and *right* fields by using a signed pointer in the *father* field under the linked array representation, as discussed earlier: a right son contains a positive *father* value and a left son a negative *father* field. Of course, the routines *maketree, setleft,* and *setright* must then be suitably modified for these representations. Under the dynamic node representation, an *isleft* logical field is required in addition to *father* if *left* and *right* fields are not present and it is desired to implement the *isleft* or *isrighi* operations.

The following program uses a binary search tree to find duplicate numbers in an input file in which each number is on a separate input line. It closely follows the algorithm of Section 5.1. Only top-down links are used; therefore no *father* field is needed.

```
struct nodetype {
   int info;
   struct nodetype *left;
   struct nodetype *right;
};

typedef struct nodetype *NODEPTR;

main()
{
NODEPTR ptree;
NODEPTR p, q;
int number;

scanf("%d", &number);
ptree = maketree(number);
while (scanf("%d", &number) != EOF) {
   p = q = ptree;
   while (number != p->info && q != NULL) {
      p = q;
      if (number < p->info)
         q = p->left;
      else
         q = p->right;
   }  /* end while */
   if (number == p->info)
      printf("%d is a duplicate\n", number);
   else if (number < p->info)
      setleft(p, number);
   else
      setright(p, number);
   }  /* end while */
}  /* end main */
```

Internal and External Nodes

By definition leaf nodes have no sons. Thus, in the linked representation of binary trees, left and right pointers are needed only in nonleaf nodes. Sometimes two separate sets of nodes are used for nonleaves and leaves. Nonleaf nodes contain *info*, *left*, and *right* fields (often no information is associated with nonleaves, so that an *info* field is unnecessary) and are allocated as dynamic records or as an array of records managed using an available list. Leaf nodes do not contain a *left* or *right* field and are kept as a single *info* array that is allocated sequentially as needed (this assumes that leaves are never freed, which is often the case). Alternatively, they can be allocated as dynamic variables containing only an *info* value. This saves a great deal of space, since leaves often represent a majority of the nodes in a binary tree. Each (leaf or nonleaf) node can also contain a *father* field, if necessary.

When this distinction is made between nonleaf and leaf nodes, nonleaves are called ***internal nodes*** and leaves are called ***external nodes.*** The terminology is also often used even when only a single type of node is defined. Of course, a son pointer within an internal node must be identified as pointing to an internal or an external node. This can be done in C in two ways. One technique is to declare two different node types and pointer types and to use a union for internal nodes, with each alternative containing one of the two pointer types. The other technique is to retain a single type of pointer and a single type of node, where the node is a union that does (if the node is an internal node) or does not (if an external node) contain left and right pointer fields. We will see an example of this latter technique at the end of this section.

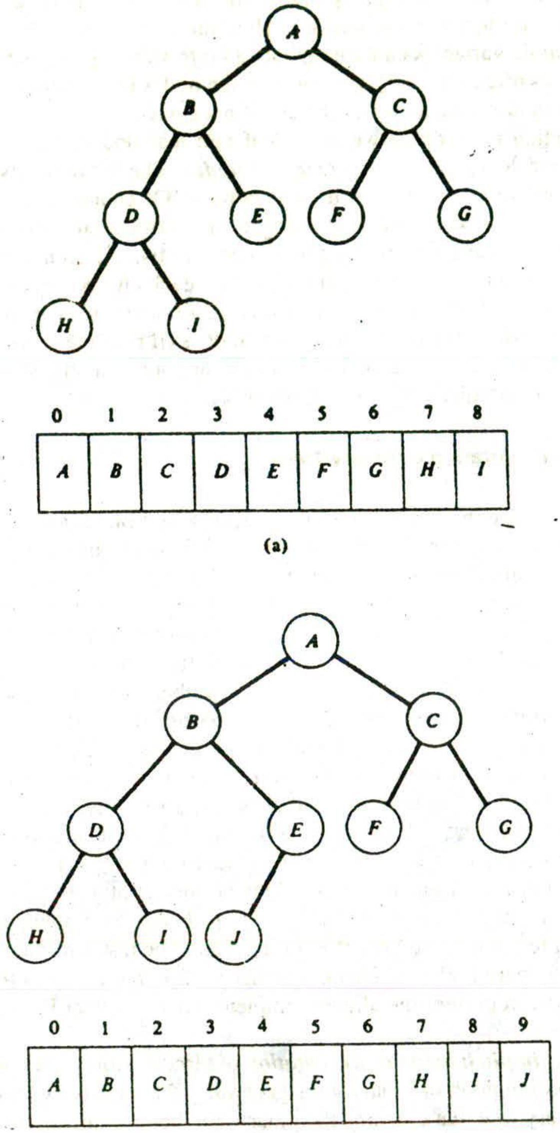
Implicit Array Representation of Binary Trees

Recall from Section 5.1 that the n nodes of an almost complete binary tree can be numbered from 1 to n, so that the number assigned a left son is twice the number assigned its father, and the number assigned a right son is 1 more than twice the number assigned its father. We can represent an almost complete binary tree without *father, left,* or *right* links. Instead, the nodes can be kept in an array *info* of size n. We refer to the node at position p simply as "node p." *info*[p] holds the contents of node p.

In C, arrays start at position 0; therefore instead of numbering the tree nodes from 1 to n, we number them from 0 to $n - 1$. Because of the one-position shift, the two sons of a node numbered p are in positions $2p + 1$ and $2p + 2$, instead of $2p$ and $2p + 1$.

The root of the tree is at position 0, so that *tree*, the external pointer to the tree root, always equals 0. The node in position p (that is, node p) is the implicit father of nodes $2p + 1$ and $2p + 2$. The left son of node p is node $2p + 1$ and its right son is node $2p + 2$. Thus the operation *left*(p) is implemented by $2 * p + 1$ and *right*(p) by $2 * p + 2$. Given a left son at position p, its right brother is at $p + 1$ and, given a right son at position p, its left brother is at $p - 1$. *father*(p) is implemented by $(p - 1)\ 2$. p points to a left son if and only if p is odd. Thus, the test for whether node p is a left son (the *isleft* operation) is to check whether $p\%2$ is not equal to 0. Figure 5.2.1 illustrates arrays that represent the almost complete binary trees of Figure 5.1.5c and d.

We can extend this ***implicit array representation*** of almost complete binary trees to an implicit array representation of binary trees generally. We do this by identifying an almost complete binary tree that contains the binary tree being represented. Figure 5.2.2a illustrates two (non-almost-complete) binary trees, and Figure 5.2.2b illustrates the smallest almost complete binary trees that contain them. Finally, Figure 5.2.2c illustrates the implicit array representations of these almost complete binary trees, and, by extension, of the original binary trees. The implicit array representation is also called the ***sequential representation***, as contrasted with the linked representation presented earlier, because it allows a tree to be implemented in a contiguous block of memory (an



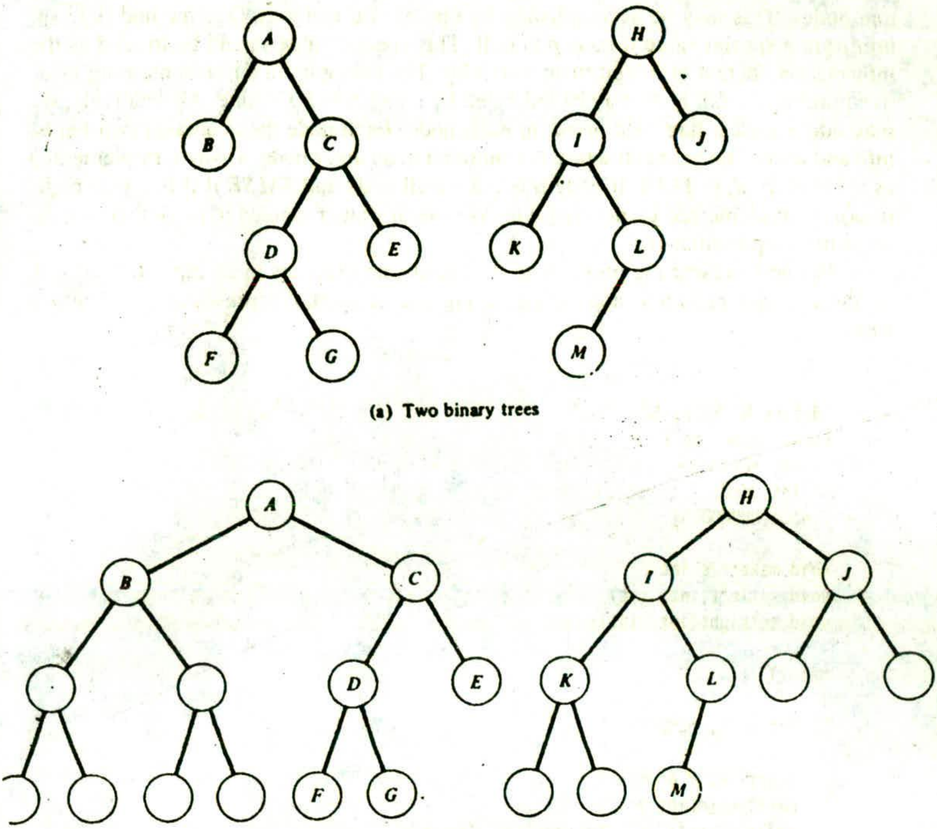
0	1	2	3	4	5	6	7	8
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*

(a)

0	1	2	3	4	5	6	7	8	9
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*

(b)

Figure 5.2.1



(a) Two binary trees

(b) Almost complete extensions

0	1	2	3	4	5	6	7	8	9	10	11	12
A	*B*	*C*			*D*	*E*					*F*	*G*

0	1	2	3	4	5	6	7	8	9
H	*I*	*J*	*K*	*L*					*M*

(c) Array representations

Figure 5.2.2

array) rather than via pointers connecting widely separated nodes. Under the sequential representation, an array element is allocated whether or not it serves to contain ~~a~~ node of a tree. We must, therefore, flag unused array elements as nonexistent, or ***null***, tree nodes. This may be accomplished by one of two methods. One method is to set *info*[*p*] to a special value if node *p* is null. This special value should be invalid as the information content of a legitimate tree node. For example, in a tree containing positive numbers, a null node may be indicated by a negative *info* value. Alternatively, we may add a logical flag field, *used*, to each node. Each node then contains two fields: *info* and *used*. The entire structure is contained in an array node. *used*(*p*), implemented as *node*[*p*].*used*, is *TRUE* if node *p* is not a null node and *FALSE* if it is a null node. *info*(*p*) is implemented by *node*[*p*].*info*. We use this latter method in implementing the sequential representation.

We now present the program to find duplicate numbers in an input list, as well as the routines *maketree* and *setleft*, using the sequential representation of binary trees.

```
#define NUMNODES 500
struct nodetype {
   int info;
   int used;
} node[NUMNODES];

void maketree(int);
void setleft(int, int);
void setright(int, int);

main()
{
   int p, q, number;

   scanf("%d", &number);
   maketree(number);
   while (scanf("%d", &number) != EOF) {
      p = q = 0;
      while (q < NUMNODES && node[q].used && number != node[p].info) {
         p = q;
         if (number < node[p].info)
            q = 2 * p + 1;
         else
            q = 2 * p + 2;
      } /* end while */
      /* if the number is in the tree it is a duplicate */
      if (number == node[p].info)
         printf("%d is a duplicate\n", number);
      else if (number < node[p].info)
         setleft(p, number);
         else
            setright (p, number);
      } /* end while */
   } /* end main */

   void maketree(int x)
   {
      int p;

      node[0].info = x;
      node[0].used = TRUE;
      /* The tree consists of node 0 alone. */
      /*   All other nodes are null nodes   */
      for (p=1; p < NUMNODES; p++)
         node[p].used = FALSE;
   } /* end maketree */

   void setleft(int p, int x)
   {
      int q;

      q = 2 * p + 1;          /* Q is the position of the left son */
      if (q >= NUMNODES)
         error("array overflow");
      else if (node[q].used)
         error("invalid insertion");
      else {
         node[q].info = x;
         node[q].used = TRUE;
      } /* end if */
   } /* end setleft */
```

The routine for *setright* is similar.

Note that under this implementation, the routine *maketree* initializes the fields *info* and *used* to represent a tree with a single node. It is no longer necessary for *maketree* to return a value, since under this representation the single binary tree represented by the *info* and *used* fields is always rooted at node 0. That is the reason that *p* is initialized to 0 in the main function before we move down the tree. Note also that under this representation it is always required to check that the range (*NUMNODES*) has not been exceeded whenever we move down the tree.

Choosing a Binary Tree Representation

Which representation of binary trees is preferable? There is no general answer to this question. The sequential representation is somewhat simpler, although it is necessary to ensure that all pointers are within the array bounds. The sequential representation clearly saves storage space for trees known to be almost complete, since it eliminates the need for the fields *left*, *right*, and *father* and does not even require a *used* field. It is also space efficient for trees that are only a few nodes short of being almost complete, or when nodes are successively eliminated from a tree that originates as almost complete, although a *used* field might then be required. However, the sequential representation can only be used in a context in which only a single tree is required, or where the number of trees needed and each of their maximum sizes is fixed in advance.

By contrast, the linked representation requires *left, right,* and *father* fields (although we have seen that one or two of these may be eliminated in specific situations) but allows much more flexible use of the collection of nodes. In the linked representation, a particular node may be placed at any location in any tree, whereas in the sequential representation a node can be utilized only if it is needed at a specific location in a specific tree. In addition, under the dynamic node representation the total number of trees and nodes is limited only by the amount of available memory. Thus the linked representation is preferable in the general, dynamic situation of many trees of unpredictable shape.

The duplicate-finding program is a good illustration of the trade-offs involved. The first program presented utilizes the linked representation of binary trees. It requires *left* and *right* fields in addition to *info* (the *father* field was not necessary in that program). The second duplicate-finding program that utilizes the sequential representation requires only an additional field, *used* (and this too can be eliminated if only positive numbers are allowed in the input, so that a null tree node can be represented by a specific negative *info* value). The sequential representation can be used for this example because only a single tree is required.

However, the second program might not work for as many input cases as the first. For example, suppose that the input is in ascending order. Then the tree formed by either program has all null left subtrees (you are invited to verify that this is the case by simulating the programs for such input). In that case the only elements of *info* that are occupied under the sequential representation are 0, 2, 6, 14, and so on (each position is two more than twice the previous one). If the value of *NUMNODES* is kept at 500, a maximum of only 16 distinct ascending numbers can be accommodated (the last one will be at position 254). This can be contrasted with the program using the linked representation, in which up to 500 distinct numbers in ascending order can be accommodated before it runs out of space. In the remainder of the text, except as noted otherwise, we assume the linked representation of a binary tree.

Binary Tree Traversals in C

We may implement the traversal of binary trees in C by recursive routines that mirror the traversal definitions. The three C routines *pretrav*, *intrav*, and *posttrav* print the contents of a binary tree in preorder, inorder, and postorder, respectively. The parameter to each routine is a pointer to the root node of a binary tree. We use the dynamic node representation of a binary tree:

```
void pretrav(NODEPTR tree)
{
   if (tree != NULL) {
      printf("%d\n", tree->info);     /* visit the root           */
      pretrav(tree->left);              /* traverse left subtree  */
      pretrav(tree->right);             /* traverse right subtree */
   } /* end if */
} /* end pretrav */

void intrav(NODEPTR tree)
{
   if (tree != NULL) {
      intrav(tree->left);               /* traverse left subtree  */
      printf("%d\n", tree->info);     /* visit the root           */
      intrav(tree->right);              /* traverse right subtree */
   } /* end if */
} /* end intrav */

void posttrav(NODEPTR tree)
{
   if (tree != NULL) {
      posttrav(tree->left);             /* traverse left subtree  */
      posttrav(tree->right);            /* traverse right subtree */
      printf("%d\n", tree->info);     /* visit the root           */
   } /* end if */
} /* end posttrav */
```

The reader is invited to simulate the actions of these routines on the trees of Figures 5.1.7 and 5.1.8.

Of course, the routines could be written nonrecursively to perform the necessary stacking and unstacking explicitly. For example, the following is a nonrecursive routine to traverse a binary tree in inorder:

```
#define MAXSTACK 100

void intrav2(NODEPTR tree)
{
   struct stack {
      int top;
      NODEPTR item[MAXSTACK];
   } s;
   NODEPTR p;

   s.top = -1;
   p = tree;
   do {
      /* travel down left branches as far as possible */
      /*       saving pointers to nodes passed        */
```

```
        while (p != NULL) {
           push (s, p);
           p = p->left;
        }  /* end while */
        /* check if finished */
        if (!empty(s)) {
           /* at this point the left subtree is empty */
           p = pop(s);
           printf("%d\n", p->info); /* visit the root */
           p = p->right;     /* traverse right subtree */
        }  /* end if */
     } while (!empty(s) || p != NULL);
  }  /* end intrav2 */
```

Nonrecursive routines to traverse a binary tree in postorder and preorder as well as nonrecursive traversals of binary trees using the sequential representation are left as exercises for the reader.

intrav and *intrav*2 represent an excellent contrast between a recursive routine and its nonrecursive counterpart. If both routines are executed, the recursive *intrav* generally executes much more quickly than the nonrecursive *intrav*2. This goes against the accepted "foll wisdom" that recursion is slower than iteration. The primary cause of the inefficiency of *intrav*2 as written is the calls to *push, pop*, and *empty*. Even when the code for these functions is inserted in-line into *intrav*2, *intrav*2 is still slower than *intrav* because of the often superfluous tests for overflow and underflow included in that code.

Yet, even when the underflow/overflow tests are removed, *intrav* is faster than *intrav*2 under a compiler that implements recursion efficiently! The efficiency of the recursive process in this case is due to a number of factors:

1. There is no "extra" recursion, as there is in computing the Fibonacci numbers, where $f(n-2)$ and $f(n-1)$ are both recomputed separately even though the value of $f(n-2)$ is used in computing $f(n-1)$.
2. The recursion stack cannot be entirely eliminated, as it can be in computing the factorial function. Thus the automatic stacking and unstacking of built-in recursion is more efficient than the programmed version. (In many systems, stacking can be accomplished by incrementing the value of a register that points to the stack top and moving all parameters into a new data area in a single block move. Program-controlled stacking as we have implemented it requires individual assignments and increments.)
3. There are no extraneous parameters and local variables, as there are, for exampie, in some versions of binary search. The automatic stacking of recursion does not stack any more variables than are necessary.

In cases of recursion that do not involve this excess baggage, such as inorder traversal, the programmer is well advised to use recursion directly.

The traversal routines that we have presented are derived directly from the definitions of the traversal methods. These definitions are in terms of the left and right sons of a node and do not reference a node's father. For that reason, both the recursive and nonrecursive routines do not require a *father* field and do not take advantage of such a field even if it is present. As we shall soon see, the presence of a *father* field allows us to develop nonrecursive traversal algorithms without using a stack. However, we first examine a technique for eliminating the stack in a nonrecursive traversal even if a *father* field is not available.

Threaded Binary Trees

Traversing a binary tree is a common operation, and it would be helpful to find a more efficient method for implementing the traversal. Let us examine the function *intrav*2 to discover the reason that a stack is needed. The stack is popped when *p* equals *NULL*. This happens in one of two cases. In one case, the ***while*** loop is exited after having been executed one or more times. This implies that the program has traveled down left branches until it reached a *NULL* pointer, stacking a pointer to each node as it was passed. Thus, the top element of the stack is the value of *p* before it became *NULL*. If an auxiliary pointer *q* is kept one step behind *p*, the value of *q* can be used directly and need not be popped.

The other case in which *p* is *NULL* is that in which the ***while*** loop is skipped entirely. This occurs after reaching a node with an empty right subtree, executing the statement $p = p->right$, and returning to repeat the body of the ***do while*** loop. At this point, we would have lost our way were it not for the stack whose top points to the node whose left subtree was just traversed. Suppose, however, that instead of containing a *NULL* pointer in its *right* field, a node with an empty right subtree contained in its *right* field a pointer to the node that would be on top of the stack at that point in the algorithm (that is, a pointer to its inorder successor.) Then there would no longer be a need for the stack, since the last node visited during a traversal of a left subtree points directly to its inorder successor. Such a pointer is called a ***thread*** and must be differentiable from a tree pointer that is used to link a node to its left or right subtree.

Figure 5.2.3 shows the binary trees of Figure 5.1.7 with threads replacing *NULL* pointers in nodes with empty right subtrees. The threads are drawn with dotted lines to differentiate them from tree pointers. Note that the rightmost node in each tree still has a *NULL* right pointer, since it has no inorder successor. Such trees are called ***right in-threaded*** binary trees.

To implement a right in-threaded binary tree under the dynamic node implementation of a binary tree, an extra logical field, *rthread*, is included within each node to indicate whether or not its right pointer is a thread. For consistency, the *rthread* field of the rightmost node of a tree (that is, the last node in the tree's inorder traversal) is also set to *TRUE*, although its *right* field remains *NULL*. Thus a node is defined as follows (recall that we are assuming that no *father* field exists):

```
struct nodetype {
   int info;
   struct nodetype *left;    /* pointer to left son  */
   struct nodetype *right;   /* pointer to right son */
   int rthread;              /*  rthread is TRUE if  */
                             /*  .right is NULL or   */
}                            /*  a non-NULL thread   */
typedef struct nodetype *NODEPTR;
```

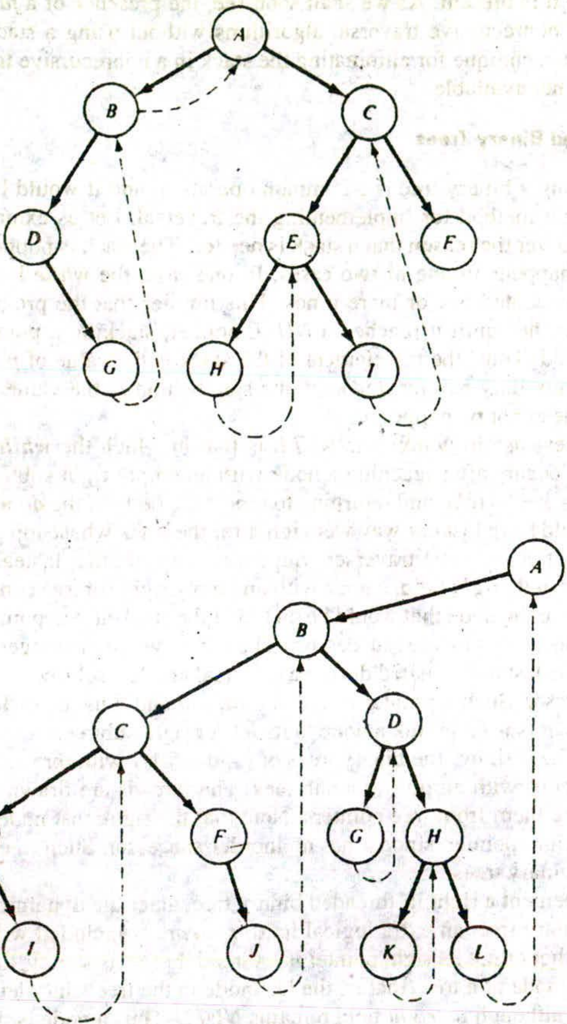


Figure 5.2.3 Right in-threaded binary trees.

We present a routine to implement inorder traversal of a right in-threaded binary tree.

```
void intrav3(NODEPTR tree)
{
   NODEPTR p, q;
```

```
    p = tree;
    do {
       q = NULL;
       while (p != NULL) {              /* Traverse left branch */
          q = p;
          p = p->left;
       }  /* end while */
       if ( q != NULL) {
          printf("%d\n", q->info);
          p = q->right;
          while (q->rthread && p != NULL) {
             printf("%d\n", p->info);
             q = p;
             p = p->right;
          }  /* end while */
       }  /* end if */
    }  while (q != NULL)
 }  /* end intrav3 */
```

In a right in-threaded binary tree the inorder successor of any node can be found efficiently. Such a tree can also be constructed in a straightforward manner. The routines *maketree, setleft*, and *setright* are as follows. We assume *info, left, right*, and *rthread* fields in each node.

```
NODEPTR maketree(int x)
{
   NODEPTR p;

   p = getnode();
   p->info = x;
   p->left = NULL;
   p->right = NULL;
   p->rthread = TRUE;
   return(p);
}  /* end maketree */

void setleft(NODEPTR p, int x)
{
   NODEPTR q;

   if (p == NULL)
      error("void insertion");
   else if (p->left != NULL)
      error("invalid insertion");
   else {
      q = getnode();
      q->info = x;
      p->left = q;
      q->left = NULL;
      /* The inorder successor of node(q) is node(p) */
```

```
        q->right = p;
        q->rthread = TRUE;
    }  /* end if */
}  /* end setleft */

void setright(NODEPTR p, int x)
{
    NODEPTR q, r;

    if (p == NULL)
        error("void insertion")
    else if (!p->rthread)
        error("invalid insertion");
    else {
        q = getnode();
        q->info = x;
        /*  save the inorder successor of node(p)      */
        r = p->right;
        p->right = q;
        p->rthread = FALSE;
        q->left = NULL;
        /* The inorder successor of node(q) is the     */
        /*      previous successor of node(p)          */
        q->right = r;
        q->rthread = TRUE;
    }  /* end else */
}  /* end setright */
```

In the linked array implementation, a thread can be represented by a negative value of *node*[*p*].*right*. The absolute value of *node*[*p*].*right* is the index in the array node of the inorder successor of *node*[*p*]. The sign of *node*[*p*].*right* indicates whether its absolute value represents a thread (minus) or a pointer to a nonempty subtree (plus). Under this implementation, the following routine traverses a right in-threaded binary tree in inorder. We leave *maketree*, *setleft*, and *setright* for the linked array representation as exercises for the reader.

```
void intrav4(int tree)
{
    int p, q;

    p = tree;
    do {
        /* travel down left links keeping q behind p */
        q = 0;
        while (p != 0) {
            q = p;
            p = node[p].left;
        }  /* end while */
```

```
      if (q != 0) {     /* check if finished */
         printf("%d\n", node[q].info);
         p = node[q].right;
         while (p < 0) {
            q = -p;
            printf("%d\n", node[q].info);
            p = node[q].right;
         }  /* end while */
      }  /* end if */
      /* traverse right subtree */
   }  while (q != 0);
}  /* end intrav4 */
```

Under the sequential representation of binary trees, the *used* field indicates threads by means of negative or positive values. If *i* represents a node with a right son, *node*[*i*].*used* equals 1, and its right son is at 2 * i + 2. However, if *i* represents a node with no right son, *node*[*i*].*used* contains the negative of the index of its inorder successor. (Note that use of negative numbers allows us to distinguish a node with a right son from a node whose inorder successor is the root of the tree.) If *i* is the rightmost node of the tree, so that it has no inorder successor, *node*[*i*].*used* can contain the special value +2. If *i* does not represent a node, *node*[*i*].*used* is 0. We leave the implementation of traversal algorithms for this representation as an exercise for the reader.

A ***left in-threaded*** binary tree may be defined similarly, as one in which each *NULL* left pointer is altered to contain a thread to that node's inorder predecessor. An ***in-threaded*** binary tree may then be defined as a binary tree that is both left in-threaded and right in-threaded. However, left in-threading does not yield the advantages of right in-threading.

We may also define right and left ***pre-threaded*** binary trees, in which *NULL* right and left pointers of nodes are replaced by their preorder successors and predecessors respectively. A right pre-threaded binary tree may be traversed efficiently in preorder without the use of a stack. A right in-threaded binary tree may also be traversed in preorder without the use of a stack. The traversal algorithms are left as exercises for the reader.

Traversal Using a *father* Field

If each tree node contains a *father* field, neither a stack nor threads are necessary for nonrecursive traversal. Instead, when the traversal process reaches a leaf node, the *father* field can be used to climb back up the tree. When *node*(*p*) is reached from a left son, its right subtree must still be traversed; therefore the algorithm proceeds to *right*(*p*). When *node*(*p*) is reached from its right son, both its subtrees have been traversed and the algorithm backs up further to *father*(*p*). The following routine implements this process for inorder traversal.

```
void intrav5(NODEPTR tree)
{
   NODEPTR p, q;
```

```
    q = NULL;
    p = tree;
    do {
        while (p != NULL) {
            q = p;
            p = p->left;
        }  /* end while */
        if (q != NULL) {
            printf("%d\n", q->info);
            p = q->right;
        }  /* end if */
        while (q != NULL && p == NULL) {
            do {
                /* node(q) has no right son. Back up until a */
                /* left son or the tree root is encountered  */
                p = q;
                q = p->father;
            } while (!isleft(p) && q != NULL);
            if (q != NULL) {
                printf("%d\n", q->info);
                p = q->right;
            }  /* end if */
        }  /* end while */
    } while (q != NULL);
}  /* end intrav5 */
```

Note that we write *isleft*(*p*) rather than *p*−>*isleft* because an *isleft* field is unnecessary to determine if *node*(*p*) is a left or a right son; we can simply check if the node is its father's left son.

In this inorder traversal a node is visited [printf ("%d\n", q−>info)] when its left son is recognized as *NULL* or when it is reached after backing up from its left son. Preorder and postorder traversal are similar except that, in preorder, a node is visited only when it is reached on the way down the tree and, in postorder, a node is visited only when its right son is recognized as *NULL* or when it is reached after backing up from its right son. We leave the details as an exercise for the reader.

Traversal using *father* pointers for backing up is less time efficient than traversal of a threaded tree. A thread points directly to a node's successor, whereas a whole series of *father* pointers may have to be followed to reach that successor in an unthreaded tree. It is difficult to compare the time efficiencies of stack-based traversal and father-based traversal, since the former includes the overhead of stacking and unstacking.

This backup traversal algorithm also suggests a stackless nonrecursive traversal technique for unthreaded trees, even if no *father* field exists. The technique is simple: simply reverse the *son* pointer on the way down the tree so that it can be used to find a way back up. On the way back up, the pointer is restored to its original value.

For example, in *intrav*5, a variable *f* can be introduced to hold a pointer to the father of *node*(*q*). The statements

```
q = p;
p = p->left;
```

in the first *while* loop can be replaced by

```
f = q;
q = p;
p = p->left;
if (p != NULL)
   q->left = f;
```

This modifies the left pointer of *node*(*q*) to point to the father of *node*(*q*) when going left on the way down [note that *p* points to the left son of *node*(*q*), so that we have not lost our way]. The statement

```
p = q->right;
```

in both of its occurrences can be replaced by

```
p = q->right;
if (p != NULL)
   q->right = f;
```

to similarly modify the right pointer of *node*(*q*) to point to its father when going right on the way down. Finally, the statements

```
p = q;
q = p->father;
```

in the inner ***do-while*** loop can be replaced by

```
p = q;
q = f;
if (q != NULL && isleft(p)) {
   f = left(q);
   left(q) = p;
}
else {
   f = right(q);
   right(q) = p;
}  /* end if */
```

to follow a modified pointer back up the tree and restore the pointer's value to point to its left or right son as appropriate.

However, now an *isleft* field is required, since the *isleft* operation cannot be implemented using a nonexistent *father* field. Also, this algorithm cannot be used in a multiuser environment if several users require access to the tree simultaneously. If one user is traversing the tree and temporarily modifying pointers, another user will be unable to use the tree as a coherent structure. Some sort of lockout mechanism is required to ensure that no one else uses the tree while pointers are reversed.

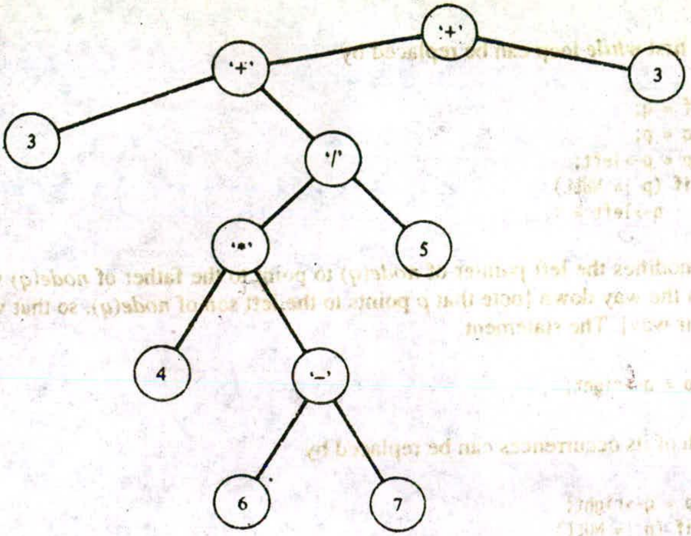


Figure 5.2.4 Binary tree representing 3 + 4 * (6 − 7) 5 + 3.

Heterogeneous Binary Trees

Often the information contained in different nodes of a binary tree is not all of the same type. For example, in representing a binary expression with constant numerical operands we may wish to use a binary tree whose leaves contain numbers but whose nonleaf nodes contain characters representing operators, Figure 5.2.4 illustrates such a binary tree.

To represent such a tree in C we may use a union to represent the information portion of the node. Of course, each tree node must contain within itself a field to indicate the type of object that its *info* field contains.

```
#define OPERATOR 0
#define OPERAND 1
struct nodetype {
     short int utype;   /* OPERATOR or OPERAND */
     union {
            char chinfo;
            float numinfo;
     } info;
     struct nodetype *left;
     struct nodetype *right;
};
typedef struct nodetype *NODEPTR;
```

Let us write a C function *evalbintree* that accepts a pointer to such a tree and returns the value of the expression represented by the tree. The function recursively evaluates the left and right subtrees and then applies the operator of the root to the two results. We use the auxiliary function *oper* (*symb*,*opnd*1,*opnd*2) introduced in Section 2.3. The first parameter of *oper* is a character representing an operator, and the last two parameters are real numbers that are the two operands. The function *oper* returns the result of applying the operator to the two operands.

```
float evalbintree (NODEPTR tree)
{
   float opnd1, opnd2;
   char symb;

   if (tree->utype == OPERAND) /* expression is a single operand */
      return (tree->numinfo);
   /* tree->utype == OPERATOR      */
   /* evaluate the left subtree    */
   opnd1 = evalbintree(tree->left);
   /* evaluate the right subtree   */
   opnd2 = evalbintree(tree->right);
   symb = tree->chinfo;            /* extract the operator */
   /*       apply the operator and return the result        */
   return(oper(symb, opnd1, opnd2));
} /* end evalbintree */
```

Section 9.1 discusses additional methods of implementing linked structures that contain heterogeneous elements. Note also that, in this example, all the operand nodes are leaves and all the operator nodes are nonleaves.

EXERCISES

5.2.1. Write a C function that accepts a pointer to a node and returns *TRUE* if that node is the root of a valid binary tree and *FALSE* otherwise.

5.2.2. Write a C function that accepts a pointer to a binary tree and a pointer to a node of the tree and returns the level of the node in the tree.

5.2.3. Write a C function that accepts a pointer to a binary tree and returns a pointer to a new binary tree that is the mirror image of the first (that is, all left subtrees are now right subtrees and vice versa).

5.2.4. Write C functions that convert a binary tree implemented using the linked array representation with only a *father* field (in which the left son's *father* field contains the negative of the pointer to its father and a right son's *father* contains a pointer to its father) to its representation using *left* and *right* fields, and vice versa.

5.2.5. Write a C program to perform the following experiment: Generate 100 random numbers. As each number is generated, insert it into an initially empty binary search tree. When all 100 numbers have been inserted, print the level of the leaf with the largest level and the level of the leaf with the smallest level. Repeat this process 50 times. Print out a table with a count of how many of the 50 runs resulted in a difference between the maximum and minimum leaf level of 0, 1, 2, 3, and so on.

5.2.6. Write C routines to traverse a binary tree in preorder and postorder.

5.2.7. Implement inorder traversal, *maketree, setleft*, and *setright* for right in-threaded binary trees under the sequential representation.

5.2.8. Write C functions to create a binary tree given:

(a) The preorder and inorder traversals of that tree

(b) The preorder and postorder traversals of that tree

Each function should accept two character strings as parameters. The tree created should contain a single character in each node.

5.2.9. The solution to the Towers of Hanoi problem for *n* disks (see Sections 3.3 and 3.4) can be represented by a complete binary tree of level $n - 1$ as follows.

(a) Let the root of the tree represent a move of the top disk on peg *frompeg* to peg *topeg*. (We ignore the identification of the disks being moved, as there is only a single disk [the top one] that can be moved from any peg to any other peg.) If *nd* is a leaf node (at level less than $n - 1$) representing the movement of the top disk from peg *x* to peg *y*, let *z* be the third peg that is neither the source or target of node *nd*. Then *left(nd)* represents a move of the top disk from peg *x* to peg *z* and *right(nd)* represents a move of the top disk from peg *z* to peg *y*. Draw sample solution trees as described previously for $n = 1, 2, 3$, and 4, and show that an inorder traversal of such a tree produces the solution to the Towers of Hanoi problem.

(b) Write a recursive C procedure that accepts a value for *n* and generates and traverses the tree as discussed previously.

(c) Because the tree is complete, it can be stored in an array of size $2^n - 1$. Show that the nodes of the tree can be stored in the array so that a sequential traversal of the array produces the inorder traversal of the tree, as follows: The root of the tree is in position $2^{n-1} - 1$; for any level *j*, the first node at that level is in position $2^{n-1-j} - 1$ and each successive node at level *j* is 2^{n-j} elements beyond the previous element at that level.

(d) Write a nonrecursive C program to create the array as described in part c and show that a sequential pass through the array does indeed produce the desired solution.

(e) How could the preceding programs be extended to include within each node the number of the disk being moved?

5.2.10. In Section 4.5 we introduced a method of representing a doubly linked list with only a single pointer field in each node by maintaining its value as the exclusive *or* of pointers to the node's predecessor and successor. A binary tree can be similarly maintained by keeping one field in each node set to the exclusive *or* of pointers to the node's *father* and *left* son [call this field *fleft(p)*] and another field in the node set to the exclusive *or* of pointers to the node's *father* and *right* son [call this field *fright(p)*].

(a) Given *father(p)* and *fleft(p)*, show how to compute *left(p)*.
Given *father(p)* and *fright(p)*, show how to compute *right(p)*.

(b) Given *fleft(p)* and *left(p)*, show how to compute *father(p)*.
Given *fright(p)* and *right(p)*, show how to compute *father(p)*.

(c) Assume that a node contains only *info, fleft, fright*, and *isleft* fields. Write algorithms for preorder, inorder, and postorder traversal of a binary tree, given an external pointer to the tree root, without using a stack or modifying any fields.

(d) Can the *isleft* field be eliminated?

5.2.11. An index of a textbook consists of major terms ordered alphabetically. Each major term is accompanied by a set of page numbers and a set of subterms. The subterms are printed on successive lines following the major term and are arranged alphabetically within the major term. Each subterm is accompanied by a set of page numbers.

Design a data structure to represent such an index and write a C program to print an index from data as follows: Each input line begins with an *m* (major term) or an *s* (subterm). An *m* line contains an *m* followed by a major term followed by an integer *n* (possibly 0) followed by *n* page numbers where the major term appears. An *s* line is similar except that it contains a subterm rather than a major term. The input lines appear in no particular order except that each subterm is considered to be a subterm of the major term which last precedes it. There may be many input lines for a single major term or subterm (all page numbers appearing on any line for a particular term should be printed with that term).

The index should be printed with one term on a line followed by all the pages on which the term appears in ascending order. Major terms should be printed in alphabetical order. Subterms should appear in alphabetical order immediately following their major term. Subterms should be indented five columns from the major terms.

The set of major terms should be organized as a binary tree. Each node in the tree contains (in addition to left and right pointers and the major term itself) pointers to two other binary trees. One of these represents the set of page numbers in which the major term occurs, and the other represents the set of subterms of the major term. Each node on a subterm binary tree contains (in addition to left and right pointers and the subterm itself, a pointer to a binary tree representing the set of page numbers in which the subterm occurs.

5.2.12. Write a C function to implement the sorting method of Section 5.1 that uses a binary search tree.

5.2.13. **(a)** Implement an ascending priority queue using a binary search tree by writing C implementations of the algorithms *pqinsert* and *pqmindelete*, as in exercise 5.1.13. Modify the routines to count the number of tree nodes accessed.

(b) Use a random number generator to test the efficiency of the priority queue implementation as follows: First, create a priority queue with 100 elements by inserting 100 random numbers in an initially empty binary search tree. Then call *pqmindelete* and print the number of tree nodes accessed in finding the minimum element, generate a new random number, and call *pqinsert* to insert the new random number and print the number of tree nodes accessed in the insertion. Note that after calling *pqinsert*, the tree still contains 100 elements. Repeat the delete/print/generate/insert/print process 1000 times. Note that the number of nodes accessed in the deletion tends to decrease, while the number of nodes accessed in the insertion tends to increase. Explain this behavior.

5.3 EXAMPLE: THE HUFFMAN ALGORITHM

Suppose that we have an alphabet of *n* symbols and a long message consisting of symbols from this alphabet. We wish to encode the message as a long bit string (a bit is either 0 or 1) by assigning a bit string code to each symbol of the alphabet and concatenating the individual codes of the symbols making up the message to produce an encoding for the message. For example, suppose that the alphabet consists of the four symbols *A*, *B*, *C*, and *D* and that codes are assigned to these symbols as follows:

Symbol	Code
A	010
B	100
C	000
D	111

The message *ABACCDA* would then be encoded as 010100010000000111010. Such an encoding is inefficient, since three bits are used for each symbol, so that 21 bits are needed to encode the entire message. Suppose that a two-bit code is assigned to each symbol, as follows:

Symbol	Code
A	00
B	01
C	10
D	11

Then the code for the message would be 00010010101100, which requires only 14 bits. We wish to find a code that minimizes the length of the encoded message.

Let us reexamine the above example. Each of the letters *B* and *D* appears only once in the message, whereas the letter *A* appears three times. If a code is chosen so that the letter *A* is assigned a shorter bit string than the letters *B* and *D*, the length of the encoded message would be small. This is because the short code (representing the letter *A*) would appear more frequently than the long code. Indeed, codes can be assigned as follows:

Symbol	Code
A	0
B	110
C	10
D	111

Using this code, the message *ABACCDA* is encoded as 0110010101110, which requires only 13 bits. In very long messages containing symbols that appear very infrequently, the savings are substantial. Ordinarily, codes are not constructed on the basis of the frequency of characters within a single message alone, but on the basis of their frequency within a whole set of messages. The same code set is then used for each message. For example, if messages consist of English words, the known relative frequency of occurrence of the letters of the alphabet in the English language might be used, although the relative frequency of the letters in any single message is not necessarily the same.

If variable-length codes are used, the code for one symbol may not be a prefix of the code for another. To see why, assume that the code for a symbol x, $c(x)$, were a prefix of the code of another symbol y, $c(y)$. Then when $c(x)$ is encountered in a left-to-right scan, it is unclear whether $c(x)$ represents the symbol x or whether it is the first part of $c(y)$.

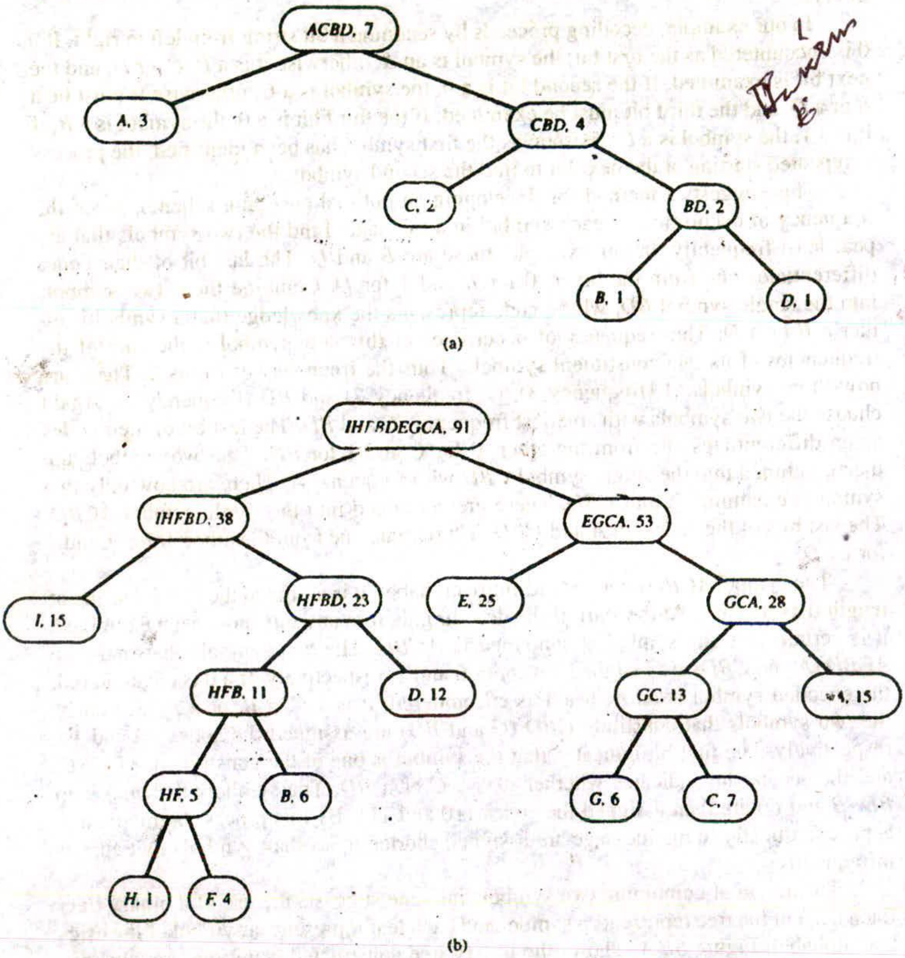
In our example, decoding proceeds by scanning a bit string from left to right. If a 0 is encountered as the first bit, the symbol is an *A*; otherwise it is a *B*, *C*, or *D*, and the next bit is examined. If the second bit is a 0, the symbol is a *C*; otherwise it must be a *B* or a *D*, and the third bit must be examined. If the third bit is a 0, the symbol is a *B*; if it is a 1, the symbol is a *D*. As soon as the first symbol has been identified, the process is repeated starting at the next bit to find the second symbol.

This suggests a method for developing an optimal encoding scheme, given the frequency of occurrence of each symbol in a message. Find the two symbols that appear least frequently. In our example, these are *B* and *D*. The last bit of their codes differentiates one from the other: 0 for *B* and 1 for *D*. Combine these two symbols into the single symbol *BD*, whose code represents the knowledge that a symbol is either a *B* or a *D*. The frequency of occurrence of this new symbol is the sum of the frequencies of its two constituent symbols. Thus the frequency of *BD* is 2. There are now three symbols: *A* (frequency 3), *C* (frequency 2) and *BD* (frequency 2). Again choose the two symbols with smallest frequency: *C* and *BD*. The last bit of their codes again differentiates one from the other: 0 for *C* and 1 for *BD*. The two symbols are then combined into the single symbol *CBD* with frequency 4. There are now only two symbols remaining: *A* and *CBD*. These are combined into the single symbol *ACBD*. The last bits of the codes for *A* and *CBD* differentiate one from the other: 0 for *A* and 1 for *CBD*.

The symbol *ACBD* contains the entire alphabet; it is assigned the null bit string of length 0 as its code. At the start of the decoding, before any bits have been examined, it is certain that any symbol is contained in *ACBD*. The two symbols that make up *ACBD* (*A* and *CBD*) are assigned the codes 0 and 1, respectively. If a 0 is encountered, the encoded symbol is an *A*; if a 1 is encountered, it is a *C*, a *B*, or a *D*. Similarly, the two symbols that constitute *CBD* (*C* and *BD*) are assigned the codes 10 and 11, respectively. The first bit indicates that the symbol is one of the constituents of *CBD*, and the second bit indicates whether it is a *C* or a *BD*. The symbols that make up *BD* (*B* and *D*) are then assigned the codes 110 and 111. By this process, symbols that appear frequently in the message are assigned shorter codes than symbols that appear infrequently.

The action of combining two symbols into one suggests the use of a binary tree. Each node of the tree represents a symbol and each leaf represents a symbol of the original alphabet. Figure 5.3.1a shows the binary tree constructed using the previous example. Each node in the illustration contains a symbol and its frequency. Figure 5.3.1b shows the binary tree constructed by this method for the alphabet and frequency table of Figure 5.3.1c. Such trees are called ***Huffman trees*** after the discoverer of this encoding method.

Once the Huffman tree is constructed, the code of any symbol in the alphabet can be constructed by starting at the leaf representing that symbol and climbing up to the root. The code is initialized to *null*. Each time that a left branch is climbed, 0 is appended to the beginning of the code; each time that a right branch is climbed, 1 is appended to the beginning of the code.



(a)

(b)

Symbol	Frequency	Code	Symbol	Frequency	Code	Symbol	Frequency	Code
A	15	111	*D*	12	011	*G*	6	1100
B	6	0101	*E*	25	10	*H*	1	01000
C	7	1101	*F*	4	01001	*I*	15	00

(c)

Figure 5.3.1 Huffman trees.

The Huffman Algorithm

The inputs to the algorithm are *n*, the number of symbols in the original alphabet, and *frequency*, an array of size at least *n* such that *frequency*[*i*] is the relative frequency of the *i*th symbol. The algorithm assigns values to an array *code* of size at least *n*, so that *code*[*i*] contains the code assigned to the *i*th symbol. The algorithm also constructs an array *position* of size at least *n* such that *position*[*i*] points to the node representing the *i*th symbol. This array is necessary to identify the point in the tree from which to start in constructing the code for a particular symbol in the alphabet. Once the tree has been constructed, the *isleft* operation introduced earlier can be used to determine whether 0 or 1 should be placed at the front of the code as we climb the tree. The *info* portion of a tree node contains the frequency of the occurrence of the symbol represented by that node.

A set *rootnodes* is used to keep pointers to the roots of partial binary trees that are not yet left or right subtrees. Since this set is modified by removing elements with minimum frequency, combining them and then reinserting the combined element into the set, it is implemented as an ascending priority queue of pointers, ordered by the value of the *info* field of the pointers' target nodes. We use the operations *pqinsert*, to insert a pointer into the priority queue, and *pqmindelete*, to remove the pointer to the node with the smallest *info* value from the priority queue.

We may outline Huffman's algorithm as follows:

```
/*   initialize the set of root nodes */
rootnodes = the empty ascending priority queue;
/*   construct a node for each symbol */

for (i = 0; i < n; i++) {
   p = maketree(frequency[i]);
   position[i] = p; /* a pointer to the leaf containing */
                    /*             the ith symbol        */
   pqinsert(rootnodes, p);
}  /* end for */
while (rootnodes contains more than one item) {
   p1 = pqmindelete(rootnodes);
   p2 = pqmindelete(rootnodes);
   /* combine p1 and p2 as branches of a single tree */
   p = maketree(info(p1) + info(p2));
   setleft(p, p1);
   setright(p, p2);
   pqinsert(rootnodes, p);
}  /* end while */

/* the tree is now constructed; use it to find codes */
root = pqmindelete(rootnodes);
for (i = 0;  i < n; i++) {
   p = position[i];
   code[i] = the null bit string;
```

```
      while (p != root) {
         /* travel up the tree */
         if (isleft(p))
            code[i] = 0 followed by code[i];
         else
            code[i] = 1 followed by code[i];
         p = father(p);
      } /* end while */
   } /* end for */
```

C Program

Note that the Huffman tree is strictly binary. Thus, if there are *n* symbols in the alphabet, the Huffman tree (which has *n* leaves) can be represented by an array of nodes of size $2n - 1$. Since the amount of storage needed for the tree is known, it may be allocated in advance in an array node.

In constructing the tree and obtaining the codes, it is only necessary to keep a link from each node to its father and an indication of whether each node is a left or a right son; left and right fields are unnecessary. Thus each node contains three fields: *father, isleft,* and *freq. father* is a pointer to the node's father. If the node is the root, its *father* field is *NULL.* The value of *isleft* is *TRUE* if the node is a left son and *FALSE* otherwise. *freq* (which corresponds to the *info* field of the algorithm) is the frequency of occurrence of the symbol represented by that node.

We allocate the array *node* based on the maximum possible symbols (a constant *maxsymbs*) rather than on the actual number of symbols, *n*. Thus the array *node*, that should be of size $2n - 1$, is declared as being of size $2 * MAXSYMBS - 1$. This means that some space is wasted. Of course, *n* itself could be made a constant rather than a variable, but then the program must be modified every time that the number of symbols differs. The nodes can also be represented by dynamic variables without wasting space. However, we present a linked array implementation. (We could also input the value of *n* and allocate arrays of the proper size using *malloc* dynamically during execution. Then, no space would be wasted using an array implementation.)

In using the linked array implementation, *node*[0] through *node*[$n - 1$] can be reserved for the leaves representing the original *n* symbols of the alphabet, and *node*[*n*] through *node*[$2 * n - 2$] for the $n - 1$ nonleaf nodes required by the strictly binary tree. This means that the array *position* is not required as a guide to the leaf nodes representing the *n* symbols, since the node containing the *i*th input symbol (where *i* goes from 0 to $n - 1$) is known to be *node*[*i*]. If the dynamic node representation were used, the array *position* would be required.

The following program encodes a message using Huffman's algorithm. The input consists of a number *n*, which is the number of symbols in the alphabet, followed by a set of *n* pairs, each of which consists of a symbol and its relative frequency. The program first constructs a string *alph*, consisting of all the symbols in the alphabet, and an array code such that *code*[*i*] is the code assigned to the *i*th symbol in *alph*. The program then prints each character, its relative frequency and its code.

Since the code is constructed from right to left, we define a structure *codetype* as follows:

```
#define MAXBITS 50

struct codetype {
   int bits[MAXBITS];
   int startpos;
};
```

MAXBITS is the maximum number of bits allowed in a code. If a code *cd* is null, *cd.startpos* is equal to *MAXBITS*. When a bit *b* is added to *cd* at the left, *cd.startpos* is decremented by 1 and *cd.bits*[*cd.startpos*] is set to *b*. When the code *cd* is completed, the bits of the code are in positions *cd.startpos* through *MAXBITS* − 1 inclusive.

An important issue is how to organize the priority queue of root nodes. In the algorithm, this data structure was represented as a priority queue of node pointers. Implementing the priority queue by a linked list, as in Section 4.2, would require a new set of nodes, each holding a pointer to a root node and a *next* field. Fortunately the *father* field of a root node is unused, so that it can be used to link together all the root nodes into a list. The pointer *rootnodes* could point to the first root node on the list. The list itself can be ordered or unordered, depending on the implementation of *pqinsert* and *pqmindelete*.

We make use of this technique in the following program, which implements the algorithm just presented.

```
#define MAXBITS 50
#define MAXSYMBS 50
#define MAXNODES 99   /* MAXNODES equals 2*MAXSYMBS-1 */

struct codetype {
   int bits[MAXBITS];
   int startpos;
};

struct nodetype {
   int freq;
   int father;         /* If node[p] is not a root node, father points   */
                       /* to the node's father; if it is, father points  */
                       /* to the next root node in the priority queue    */
   int isleft;
};

void pqinsert(int, int);
int pqmindelete(int);

main()
{
   struct codetype cd, code[MAXSYMBS];
   struct nodetype node[MAXNODES];
   int i, k, n, p, p1, p2, root, rootnodes;
   char symb, alph[MAXSYMBS];
```

```
for (i = 0; i < MAXSYMBS; i++)
   alph[i] = ' ';
rootnodes = 0;
/*  input the alphabet and frequencies  */
scanf("%d", &n);
for (i = 0; i < n; i++) {
   scanf("%s %d", &symb, &node[i].freq);
   pqinsert(rootnodes, i);
   alph[i] = symb;
}  /* end for */

/*  we now build the trees  */
for (p = n; p < 2*n-1; n++) {
   /* p points to the next available node. Obtain the */
   /*  root nodes p1 and p2 with smallest frequencies */
   p1 = pqmindelete(rootnodes);
   p2 = pqmindelete(rootnodes);
   /*  set left(p) to p1 and right(p) to p2 */
   node[p1].father = p;
   node[p1].isleft = TRUE;
   node[p2].father = p;
   node[p2].isleft = FALSE;
   node[p].freq = node[p1].freq + node[p2].freq;
   pqinsert(rootnodes, p);
}  /* end for */
/* There is now only one node left */
/*    with a null father field     */
root = pqmindelete(rootnodes);
/* extract the codes from the tree */
for (i = 0; i < n; i++) {
   /* initialize code[i] */
   cd.startpos = MAXBITS;
   /* travel up the tree */
   p = i;
   while (p != root) {
      --cd.startpos;
      if (node[p].isleft)
         cd.bits[cd.startpos] = 0;
      else
         cd.bits[cd.startpos] = 1;
      p = node[p].father;
   }  /* end while */
   for (k = cd.startpos; k < MAXBITS; k++)
      code[i].bits[k] = cd.bits[k];
   code[i].startpos = cd.startpos;
}  /* end for */
```

```
   /*  print results  */
   for (i = 0; i < n; i++) {
      printf("\n%c %d ", alph[i], nodes[i].freq);
      for (k = code[i].startpos; k < MAXBITS; k++)
         printf("%d", code[i].bits[k]);
      printf("\n");
   } /* end for */
} /* end main */
```

We leave to the reader the coding of the routine *encode(alph, code, msge, bitcode)*. This procedure accepts the string *alph*, the array *code* constructed in the foregoing program, and a message *msge* and sets *bitcode* to the bit string encoding of the message.

Given the encoding of a message and the Huffman tree used in constructing the code, the original message can be recovered as follows: Begin at the root of the tree. Each time that a 0 is encountered, move down a left branch, and each time that a 1 is encountered, move down a right branch. Repeat this process until a leaf is encountered. The next character of the original message is the symbol that corresponds to that leaf. See if you can decode 1110100010111011 using the Huffman tree of Figure 5.3.1b.

To decode it is necessary to travel from the root of the tree down to its leaves. This means that instead of *father* and *isleft* fields, two fields *left* and *right* are needed to hold the left and right sons of a particular node. It is straightforward to compute the fields *left* and *right* from the fields *father* and *isleft*. Alternatively, the values *left* and *right* can be constructed directly from the frequency information for the symbols of the alphabet using an approach similar to that used in assigning the value of *father*. (Of course, if the trees are to be identical, the symbol/frequency pairs must be presented in the same order under the two methods.) We leave these algorithms, as well as the decoding algorithm, as exercises for the reader.

EXERCISES

5.3.1. Write a C function *encode(alph, code, msge, bitcode)*. The function accepts the string *alph* and the array *code* produced by the program *findcode* in the text and a message *msge*. The procedure sets *bitcode* to the Huffman encoding of that message.

5.3.2. Write a C function *decode(alph, left, right, bitcode, msge)*, in which *alph* is the string produced by the program *findcode* in the text, *left* and *right* are arrays used to represent a Huffman tree, and *bitcode* is a bit string. The function sets *msge* to the Huffman decoding of *bitcode*.

5.3.3. Implement the priority queue *rootnodes* as an ordered list. Write appropriate *pqinsert* and *pqmindelete* routines.

5.3.4. Is it possible to have two different Huffman trees for a set of symbols with given frequencies? Either give an example in which two such trees exist or prove that there is only a single such tree.

5.3.5. Define the ***Fibonacci binary tree of order n*** as follows: If $n = 0$ or $n = 1$, the tree consists of a single node. If $n > 1$, the tree consists of a root, with the Fibonacci tree of order $n - 1$ as the left subtree and the Fibonacci tree of order $n - 2$ as the right subtree.

(a) Write a C function that returns a pointer to the Fibonacci binary tree of order n.

(b) Is such a tree strictly binary?

(c) What is the number of leaves in the Fibonacci tree of order *n*?

(d) What is the depth of the Fibonacci tree of order *n*?

5.3.6. Given a binary tree *t*, its ***extension*** is defined as the binary tree *e*(*t*) formed from *t* by adding a new leaf node at each *NULL* left and right pointer in *t*. The new leaves are called ***external*** nodes, and the original nodes (which are now all nonleaves) are called ***internal*** nodes. *e*(*t*) is called an ***extended binary tree***.

(a) Prove that an extended binary tree is strictly binary.

(b) If *t* has *n* nodes, how many nodes does *e*(*t*) have?

(c) Prove that all leaves in an extended binary tree are newly added nodes.

(d) Write a C routine that extends a binary tree *t*.

(e) Prove that any strictly binary tree with more than one node is an extension of one and only one binary tree.

(f) Write a C function that accepts a pointer to a strictly binary tree *t1* containing more than one node and deletes nodes from *t1* creating a binary tree *t2* such that *t1* = *e*(*t2*).

(g) Show that the complete binary tree of order *n* is the *n*th extension of the binary tree consisting of a single node.

5.3.7. Given a strictly binary tree *t* in which the *n* leaves are labeled as nodes 1 through *n*, let *level*(*i*) be the level of node *i* and let *freq*(*i*) be an integer assigned to node *i*. Define the ***weighted path length*** of *t* as the sum of *freq*(*i*) * *level*(*i*) over all leaves of *t*.

(a) Write a C routine to compute the weighted path length, given fields *freq* and *father*.

(b) Show that the Huffman tree is the strictly binary tree with minimum weighted path length.

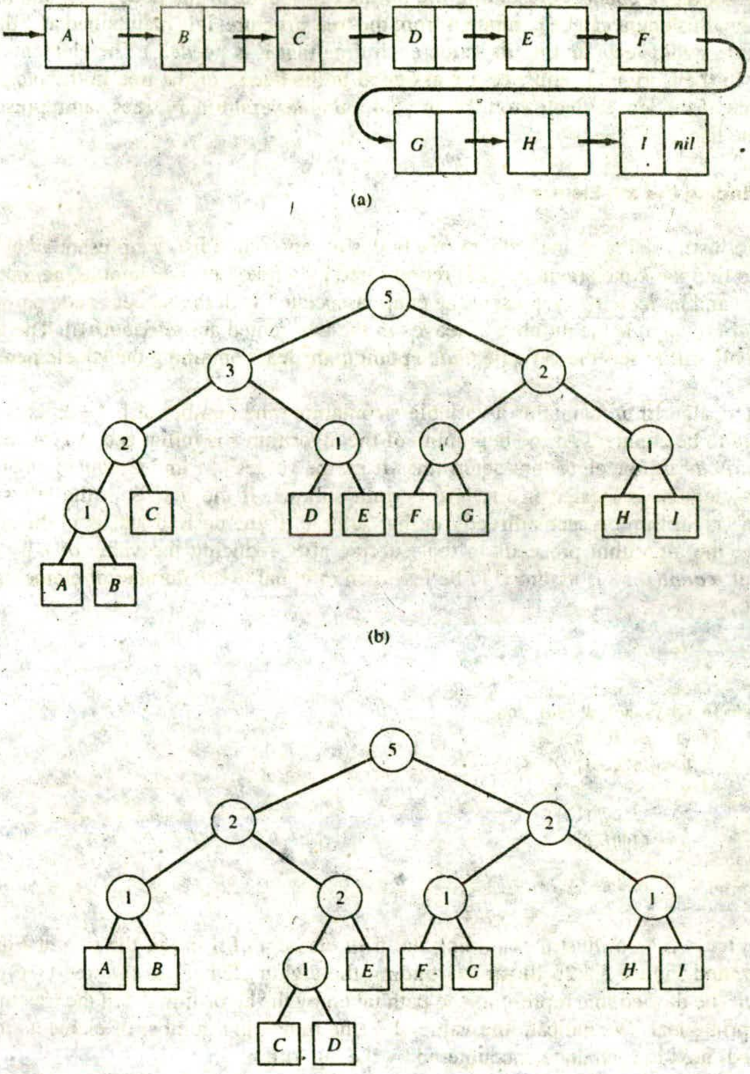
5.4 REPRESENTING LISTS AS BINARY TREES

Several operations can be performed on a list of elements. Included among these operations are adding a new element to the front or rear of the list, deleting the existing first or last element of the list, retrieving the *k*th element or the last element of the list, inserting an element following or preceding a given element, deleting a given element, and deleting the predecessor or successor of a given element. Building a list with given elements is an additional operation that is frequently required.

Depending on the representation chosen for a list, some of these operations may or may not be possible with varying degrees of efficiency. For example, a list may be represented by successive elements in an array or as nodes in a linked structure. Inserting an element following a given element is relatively efficient in a linked list (involving modifications to a few pointers aside from the actual insertion) but relatively inefficient in an array (involving moving all subsequent elements in the array one position). However, finding the *k*th element of a list is far more efficient in an array (involving only the computation of an offset) than in a linked structure(that requires passing through the first *k* − 1 elements). Similarly, it is not possible to delete a specific element in a singly linked linear list given only a pointer to that element, and it is only possible to do so inefficiently in a singly linked circular list (by traversing the entire list to reach the previous element, and then performing the deletion). The same operation, however, is quite efficient in a doubly linked (linear or circular) list.

In this section we introduce a tree representation of a linear list in which the operations of finding the *k*th element of a list and deleting a specific element are relatively efficient. It is also possible to build a list with given elements using this representation. We also briefly consider the operation of inserting a single new element.

A list may be represented by a binary tree as illustrated in Figure 5.4.1. Figure 5.4.1a shows a list in the usual linked format, while Figure 5.4.1b and c show two binary



(a)

(b)

(c)

Figure 5.4.1 List and two corresponding binary trees.

tree representations of the list. Elements of the original list are represented by leaves of the tree (shown as squares in the figure), whereas nonleaf nodes of the tree (shown as circles in the figure) are present as part of the internal tree structure. Associated with each leaf node are the contents of the corresponding list element. Associated with each nonleaf node is a count representing the number of leaves in the node's left subtree. (Although this count can be computed from the tree structure, it is maintained as a data element to avoid recomputing its value each time that it is needed.) The elements of the list in their original sequence are assigned to the leaves of the tree in the inorder sequence of the leaves. Note from Figure 5.4.1 that several binary trees can represent the same list.

Finding the *k*th Element

To justify using so many extra tree nodes to represent a list, we present an algorithm to find the *k*th element of a list represented by a tree. Let *tree* point to the root of the tree, and let *lcount(p)* represent the count associated with the nonleaf node pointed to by *p* [*lcount(p)* is the number of leaves in the tree rooted at *node*(*left*(*p*))]. The following algorithm sets the variable *find* to point to the leaf containing the *k*th element of the list.

The algorithm maintains a variable *r* containing the number of list elements remaining to be counted. At the beginning of the algorithm *r* is initialized to *k*. At each nonleaf *node*(*p*), the algorithm determines from the values of *r* and *lcount*(*p*) whether the *k*th element is located in the left or right subtree. If the leaf is in the left subtree, the algorithm proceeds directly to that subtree. If the desired leaf is in the right subtree, the algorithm proceeds to that subtree after reducing the value of *r* by the value of *lcount*(*p*). *k* is assumed to be less than or equal to the number of elements in the list.

```
r = k;
p = tree;
while (p is not a leaf node)
    if (r <= lcount(p))
        p = left(p);
    else {
        r -= lcount(p);
        p = right(p);
    } /* end if */
find = p;
```

Figure 5.4.2a illustrates finding the fifth element of a list in the tree of Figure 5.4.1b, and Figure 5.4.2b illustrates finding the eighth element in the tree of Figure 5.4.1c. The dashed line represents the path taken by the algorithm down the tree to the appropriate leaf. We indicate the value of *r* (the remaining number of elements to be counted) next to each node encountered by the algorithm.

The number of tree nodes examined in finding the *k*th list element is less than or equal to 1 more than the depth of the tree (the longest path in the tree from the root to a leaf). Thus four nodes are examined in Figure 5.4.2a in finding the fifth element of the

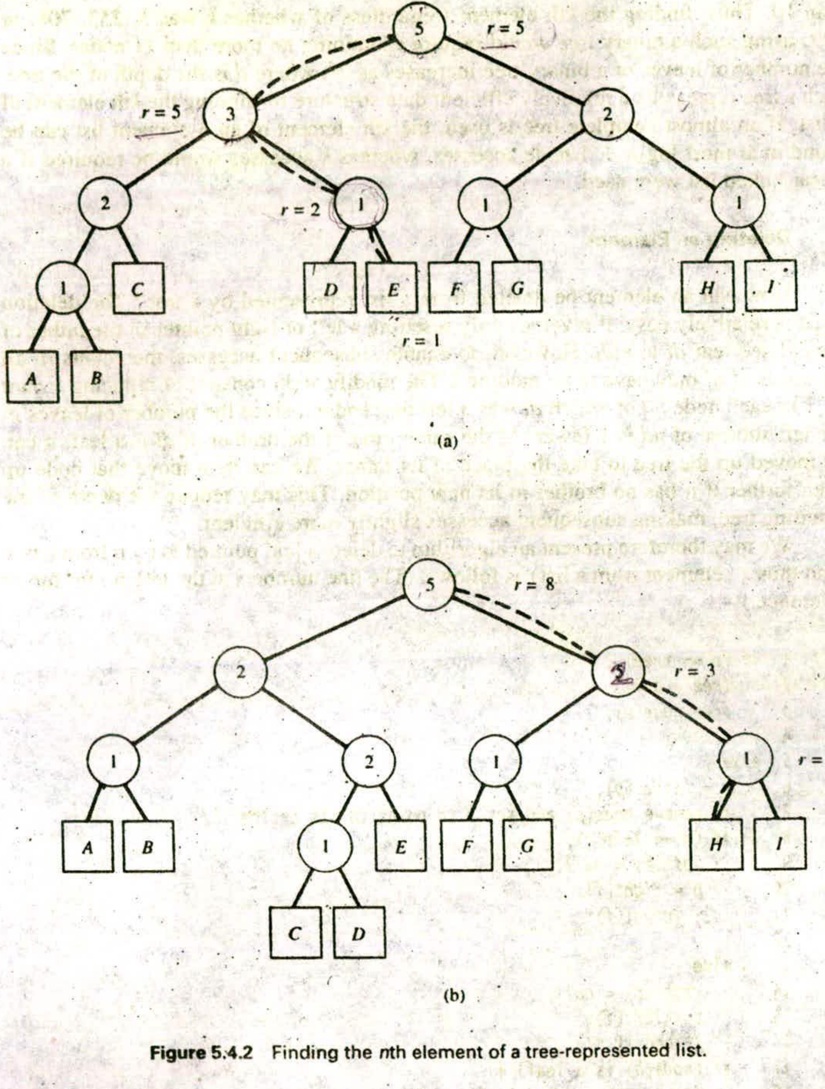


Figure 5.4.2 Finding the *n*th element of a tree-represented list.

list, and also in Figure 5.4.2b in finding the eighth element. If a list is represented as a linked structure, four nodes are accessed in finding the fifth element of the list [that is, the operation $p = next(p)$ is performed four times] and seven nodes are accessed in finding the eighth element.

Although this is not a very impressive saving, consider a list with 1000 elements. A binary tree of depth 10 is sufficient to represent such a list, since $\log_2 1000$ is less than 10. Thus, finding the *k*th element (regardless of whether *k* was 3, 253, 708, or 999) using such a binary tree would require examining no more than 11 nodes. Since the number of leaves of a binary tree increases as 2^d, where *d* is the depth of the tree, such a tree represents a relatively efficient data structure for finding the *k*th element of a list. If an almost complete tree is used, the *k*th element of an *n*-element list can be found in at most $\log_2 n + 1$ node accesses, whereas *k* accesses would be required if a linear linked list were used.

Deleting an Element

How can an element be deleted from a list represented by a tree? The deletion itself is relatively easy. It involves only resetting a left or right pointer in the father of the deleted leaf *dl* to *null*. However, to enable subsequent accesses, the counts in all ancestors of *dl* may have to be modified. The modification consists of reducing *lcount* by 1 in each node *nd* of which *dl* was a left descendant, since the number of leaves in the left subtree of *nd* is 1 fewer. At the same time, if the brother of *dl* is a leaf, it can be moved up the tree to take the place of its father. We can then move that node up even further if it has no brother in its new position. This may reduce the depth of the resulting tree, making subsequent accesses slightly more efficient.

We may therefore present an algorithm to delete a leaf pointed to by *p* from a tree (and thus an element from a list) as follows. (The line numbers at the left are for future reference.)

```
1   if (p == tree) {
2       tree = null;
3       free node(p);
4   }
5   else {
6       f = father(p);
7       /* remove node(p) and set b to point to its brother */
8       if (p == left(f)) {
9          left(f) = null;
10         b = right(f);
11         --lcount(f);
12      }
13      else {
14         right(f) = null;
15         b = left(f);
16      }  /* end if */
17      if (node(b) is a leaf) {
18         /* move the contents of node(b) up to its */
19         /*           father and free node(b)       */
20         info(f) = info(b);
21         left(f) = null;
22         right(f) = null;
23         lcount(f) = 0;
24         free node(b);
25      }  /* end if */
```

```
    free node(p);
    /* climb up the tree */
    q = f;
    while (q != tree) {
        f = father(q);
        if (q == left(f)) {
           /* the deleted leaf was a left descendant */
           /*.                  of node(f)          */
           --lcount(f);
           b = right(f);
        }
        else
           b = left(f);
        /* node(b) is the brother of node(q) */
        if (b == null && node(q) is a leaf){
           /* move up the contents of node(q) */
           /* to its father and free node(q) */
           info(f) = info(q);
           left(f) = null;
           right(f) = null;
           lcount(f) = 0;
           free node(q);
        } /* end if */
        q = f;
    } /* end while */
} /* end else */
```

Figure 5.4.3 illustrates the results of this algorithm for a tree in which the nodes *C*, *D*, and *B* are deleted in that order. Make sure that you follow the actions of the algorithm on these examples. Note that the algorithm maintains a 0 count in leaf nodes for consistency, although the count is not required for such nodes. Note also that the algorithm never moves up a nonleaf node even if this could be done. (For example, the father of *A* and *B* in Figure 5.4.3b has not been moved up.) We can easily modify the algorithm to do this (the modification is left to the reader) but have not done so for reasons that will become apparent shortly.

This deletion algorithm involves inspection of up to two nodes (the ancestor of the node being deleted and that ancestor's brother) at each level. Thus, the operation of deleting the *k*th element of a list represented by a tree (which involves finding the element and then deleting it) requires a number of node accesses approximately equal to three times the tree depth. Although deletion from a linked list requires accesses to only three nodes (the node preceding and following the deleted node as well as the deleted node), deleting the *k*th element requires a total of $k + 2$ accesses ($k - 1$ of which are to locate the node preceding the *k*th). For large lists, therefore, the tree representation is more efficient.

Similarly we can compare favorably the efficiency of tree-represented lists with array-represented lists. If an *n*-element list is maintained in the first *n* elements of an array, finding the *k*th element involves only a single array access, but deleting it requires shifting the $n - k$ elements that had followed the deleted element. If gaps are allowed in

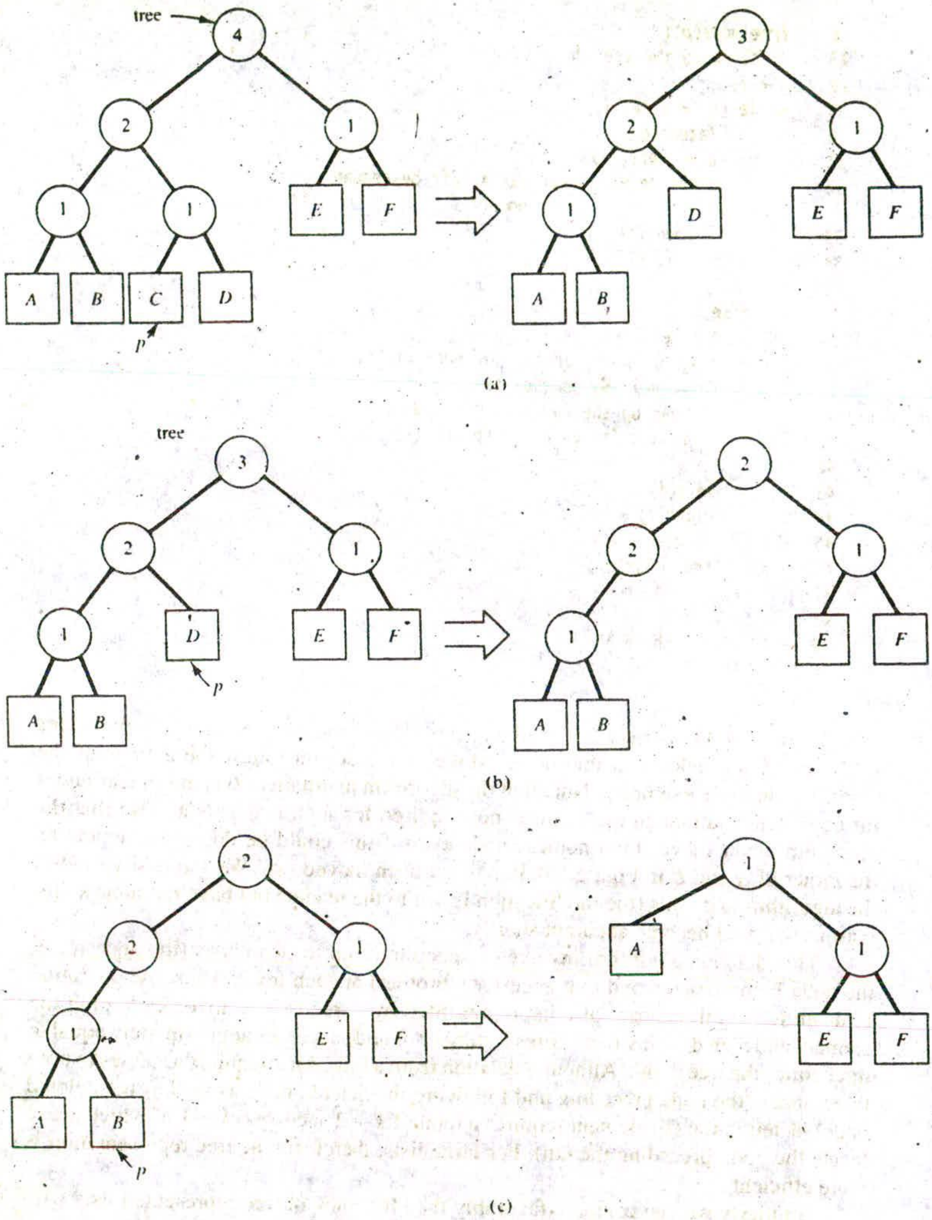


Figure 5.4.3 Deletion algorithm.

the array so that deletion can be implemented efficiently (by setting a flag in the array position of the deleted element without shifting any subsequent elements), finding the *k*th element requires at least *k* array accesses. The reason is that it is no longer possible to know the array position of the *k*th element in the list, since gaps may exist among the elements in the array. [We should note, however, that if the order of the elements in the list is irrelevant, the *k*th element in an array can be deleted efficiently by overwriting it with the element in position *n* (the last element) and adjusting the count to $n - 1$. However, it is unlikely that we would want to delete the *k*th element from a list in which the order is irrelevant, since there would then be no significance in the *k*th element over any of the others.]

Inserting a new *k*th element into a tree-represented list [between the $(k - 1)$st and the previous *k*th] is also a relatively efficient operation. The insertion consists of locating the *k*th element, replacing it with a new nonleaf that has a leaf containing the new element as its left son and a leaf containing the old *k*th element as its right son, and adjusting appropriate counts among its ancestors. We leave the details to the reader. (However, repeatedly adding a new *k*th element by this method causes the tree to become highly unbalanced, since the branch containing the *k*th element becomes disproportionately long compared with the other branches. This means that the efficiency of finding the *k*th element is not as great as it would be in a balanced tree in which all paths are approximately the same length. The reader is encouraged to find a "balancing" strategy to alleviate this problem. Despite this problem, if insertions into the tree are made randomly, so that it is equally likely for an element to be inserted at any given position, the resulting tree remains fairly balanced and finding the *k*th element remains efficient.)

Implementing Tree-Represented Lists in C

The C implementations of the search and deletion algorithms are straightforward using the linked representation of binary trees. However, such a representation requires *info, lcount, father, left,* and *right* fields for each tree node, whereas a list node requires only *info* and *next* fields. Coupled with the fact that the tree representation requires approximately twice as many nodes as a linked list, this space requirement may make the tree representation impractical. We could, of course, utilize external nodes containing only an *info* field (and perhaps a *father* field) for the leaves, and internal nodes containing *lcount, father, left,* and *right* fields for the nonleaves. We do not pursue that possibility here.

Under the sequential representation of a binary tree, the space requirements are not nearly so great. If we assume that no insertions are required once the tree is constructed and that the initial list size is known, we can set aside an array to hold an almost complete strictly binary tree representation of the list. Under that representation, *father, left,* and *right* fields are unnecessary. As we shall soon show, it is always possible to construct an almost complete binary tree representation of a list.

Once the tree has been constructed, the only fields required are *info, lcount*, and a field used to indicate whether or not an array element represents an existing or a deleted tree node. Also, as we have noted before, *lcount* is only required for nonleaf nodes of the tree, so that a structure could actually be used with either the *lcount* field or the *info* field.

depending on whether or not the node is a leaf. We leave this possibility as an exercise for the reader. It is also possible to eliminate the need for the *used* field at some expense to time efficiency (see Exercises 5.4.4 and 5.4.5). We assume the following definitions and declarations (assume 100 elements in the list):

```
#define MAXELTS 100      /* maximum number of list elements */
#define NUMNODES 2*MAXELTS - 1
#define BLANKS "                    "    /* 20 blanks */
struct nodetype {
   char info[20];
   int lcount;
   int used;
} node[NUMNODES];
```

A nonleaf node can be recognized by an *info* value equal to *BLANKS*. *father*(*p*), *left*(*p*), and *right*(*p*) can be implemented in the usual way as $(p - 1)/2$, $2 * p + 1$, and $2 * p + 2$, respectively.

A C routine to find the *k*th element follows. It uses the library routine *strcmp*, which returns 0 if two strings are equal.

```
int findelement(int k)
{
   int p, r;

   r = k;
   p = 0;
   while (strcmp(node[p].info, BLANKS) == 0)
      if (r <= node[p].lcount)
         p = p*2 + 1;
      else {
         r -= node[p].lcount;
         p = p*2 + 2;
      } /* end if */
   return(p);
} /* end findelement */
```

The C routine to delete the leaf pointed to by *p* using the sequential representation is somewhat simpler than the corresponding algorithm presented in the foregoing. We can ignore all assignments of *null* (lines 2, 9, 14, 21, 22, 43 and 44), since pointers are not used. We can also ignore the assignments of 0 to an *lcount* field (lines 23 and 45), since such an assignment is part of the conversion of a nonleaf to a leaf, and in our C representation the *lcount* field in leaf nodes is unused. A node can be recognized as a leaf (lines 17 and 39) by a nonblank *info* value, and the pointer *b* as *null* (line 39) by a *FALSE* value for *node*[*b*].*used*. Freeing a node (lines 3, 26, and 46) is accomplished by setting its *used* field to *FALSE*. The routine uses the library routine *strcpy*(*s*,*t*), which assigns string *t* to string *s*, and the routine *strcmp* to compare two strings for equality.

```
void delete(int p)
{
   int b, f, q;

   if (p == 0)
      node[p].used = FALSE;         /* Algorithm lines 1-4    */
   else {
      f = (p-1) / 2;                /* Algorithm line 6       */
      if (p % 2 != 0) {             /* Algorithm line 8       */
         b = 2*f + 2;
         --node[f].lcount;
      }
      else
          b = 2*f + 1;
      if (strcmp(node[b].info, BLANKS) != 0) {
                                     /* Algorithm lines 17-25 */
         strcpy(node[f].info, node[b].info);
         node[b].used = FALSE;
      }  /* end if */
      node[p].used = FALSE;         /* Algorithm line 26      */
      q = f;                        /* Algorithm line 28      */
      while (q != 0) {
         f = (q-1) / 2;             /* Algorithm line 30      */
         if (q % 2 != 0) {          /* Algorithm line 31      */
            --node[f].lcount;
            b = 2*f + 2;
         }
         else
            b = 2*f + 1;
         if (!node[b].used && strcmp(node[q].info, BLANKS) != 0) {
                                     /* Algorithm lines 39-47 */
            strcpy(node[f].info, node[q].info);
            node[q].used = FALSE;
         }  /* end if */
         q = f;
      }  /* end while */
   }  /* end if */
}  /* end delete */
```

Our use of the sequential representation explains the reason for not moving a nonleaf without a brother further up in a tree during deletion. Under the sequential representation, such a moving-up process would involve copying the contents of all nodes in the subtree within the array, whereas it involves modifying only a single pointer if the linked representation is used.

Constructing a Tree-Represented List

We now return to the claim that, given a list of *n* elements, it is possible to construct an almost complete strictly binary tree representing the list. We have already seen in Section 5.1 that it is possible to construct an almost complete strictly binary tree with n leaves and $2 * n - 1$ nodes. The leaves of such a tree occupy nodes numbered $n - 1$ through $2 * n - 2$. If d is the smallest integer such that 2^d is greater or equal to n (that is, if d equals the smallest integer greater than or equal to $\log_2 n$), d equals the depth of the tree. The number assigned to the first node on the bottom level of the tree is $2^d - 1$. The first elements of the list are assigned to nodes numbered $2^d - 1$ through $2 * n - 2$, and the remainder (if any) to nodes numbered $n - 1$ through $2^d - 2$. In constructing a tree representing a list with n elements, we can assign elements to the *info* fields of tree leaves in this sequence and assign a blank string to the *info* fields of the nonleaf nodes, numbered 0 through $n - 2$. It is also a simple matter to initialize the *used* field to *true* in all nodes numbered 0 to $2 * n - 2$.

Initializing the values of the *lcount* array is more difficult. Two methods can be used: one involving more time and a second involving more space. In the first method, all *lcount* fields are initialized to 0. Then the tree is climbed from each leaf to the tree root in turn. Each time a node is reached from its left son, 1 is added to its *lcount* field. After this process is performed for each leaf, all *lcount* values have been properly assigned. The following routine uses this method to construct a tree from a list of input data:

```
void buildtree(int n)
{
   int d, f, i, p, power, size;

   /* compute the tree depth d and the value of 2^d  */
   d = 0;
   power = 1;
   while (power < n) {
      ++d;
      power *= 2;
   } /* end while */
   /* assign the elements of the list, initialize the used flags, */
   /* and initialize the lcount field to 0 in all nonleaves       */
   size = 2*n - 1;
   for (i = power-1; i < size1; i++) {
      scanf("%d", &node[i].info);
      node[i].used = TRUE;
   } /* end for */
   for (i=n-1; i < power-1; i++){
      scanf("%s", node[i].info);
      node[i].used = TRUE;
   } /* end for */
   for (i=0: i < n-1; i++) {
      node[i].used = TRUE;
      node[i].lcount = 0;
      strcpy(node[i].info, BLANKS);
   } /* end for */
   /* set the lcount fields */
   for (i=n-1; i < size; i++) {
      /* follow the path from each leaf to the root */
      p = i;
      while (p != 0) {
         f = (p-1) / 2;
         if (p % 2 != 0)
            ++node[f].lcount;
         p = f;
      }  /* end while */
   }  /* end for */
}  /* end buildtree */
```

The second method uses an additional field, *rcount*, in each node to hold the number of leaves in the right subtree of each nonleaf node. This field as well as the *lcount* field is set to 1 in each nonleaf that is the father of two leaves. If n is odd, so that there is a node (numbered $(n - 3)/2$) that is the father of a leaf and a nonleaf, *lcount* in that node is set to 2 and *rcount* to 1.

The algorithm then goes through the remaining array elements in reverse order, setting *lcount* in each node to the sum of *lcount* and *rcount* in the node's left son, and *rcount* to the sum of *lcount* and *rcount* in the node's right son. We leave to the reader the C implementation of this technique. Note that *rcount* can be implemented as a local array in *buildtree* rather than as a field in every node, since its values are unused once the tree is built.

This second method has the advantage that it visits each nonleaf once to directly calculate its *lcount* (and *rcount*) value. The first method visits each nonleaf once for each of its leaf descendants, adding one to *lcount* each time that the leaf is found to be a left descendant. To counterbalance this advantage, the second method requires an extra *rcount* field, whereas the first method needs no extra fields.

The Josephus Problem Revisited

The Josephus problem of Section 4.5 is a perfect example of the utility of the binary tree representation of a list. In that problem it was necessary to repeatedly find the *m*th next element of a list and then delete that element. These are operations that can be performed efficiently in a tree-represented list.

If *size* equals the number of elements currently in a list, the position of the *m*th node following the node in position k that has just been deleted is given by $1+(k-2+m)$ % *size*. (Here we assume that the first node in the list is considered in position 1, not in position 0.) For example, if a list has five elements and the third element is deleted, and we wish to find the fourth element following the deleted element, $size = 4$, $k = 3$, and $m = 4$. Then $k - 2 + m$ equals 5 and $(k - 2 + m)$ % *size* is 1, so that the fourth element following the deleted element is in position 2. (After deleting element 3, we count elements 4, 5, 1, and 2.) We can therefore write a C function *follower* to find the *m*th node following a node in position k that has just been deleted and to reset k to its position. The routine calls the routine *findelement* presented earlier.

```
int follower(int size, int m, int *pk)
{
   int j, d;

   j = k - 2 + m;
   *pk = (j % size) + 1;
   return(findelement(*pk));
}  /* end follower */
```

The following C program implements the Josephus algorithm using a tree-represented list. The program inputs the number of people in a circle (n), an integer count (m), and the names of the people in the circle in order, beginning with the person from whom the count starts. The people in the circle are counted in order and the person at whom the input count is reached leaves the circle. The count then begins again from 1, starting at the next person. The program prints the order in which people leave the circle. Section 4.5 presented a program to do this using a circular list in which $(n - 1) * m$ nodes are accessed once the initial list is constructed. The following algorithm accesses fewer than $(n - 1) * \log_2 n$ nodes once the tree is built.

```
/* definitions of MAXELTS, NUMNODES, BLANKS, */
/*              and nodetype go here          */

void buildtree(int);
int follower(int, int, int *);
void delete(int);

main()
{
   int k, m, n, p, size;
   struct nodetype node[NUMNODES];

   scanf("%d%d", &n, &m);
   buildtree(n);
   k = n + 1; /* initially we have "deleted" the (n+1)st person */
   for (size = n; size > 2; --size) {
      /* repeat until one person is left */
      p = follower(size, m, &k);
      printf("%d\n", node[p].info);
      delete(p);
   }  /* end for */
   printf("%d", node[0].info);
}  /* end main */
```

EXERCISES

5.4.1. Prove that the leftmost node at level n in an almost complete strictly binary tree is assigned the number 2^n.

5.4.2. Prove that the extension (see Exercise 5.3.5) of an almost complete binary tree is almost complete.

5.4.3. For what values of *n* and *m* is the solution to the Josephus problem given in this section faster in execution than the solution given in Section 4.5? Why is this so?

5.4.4. Explain how we can eliminate the need for a *used* field if we elect not to move up a newly created leaf with no brother during deletion.

5.4.5. Explain how we can eliminate the need for a *used* field if we set *lcount* to -1 in a nonleaf that is converted to a leaf node and reset *info* to blanks in a deleted node.

5.4.6. Write a C routine *buildtree* in which each node is visited only once by using an *rcount* array as described in the text.

5.4.7. Show how to represent a linked list as an almost complete binary tree in which each list element is represented by one tree node. Write a C function to return a pointer to the *k*th element of such a list.

5.5 TREES AND THEIR APPLICATIONS

In this section we consider general trees and their representations. We also investigate some of their uses in problem solving.

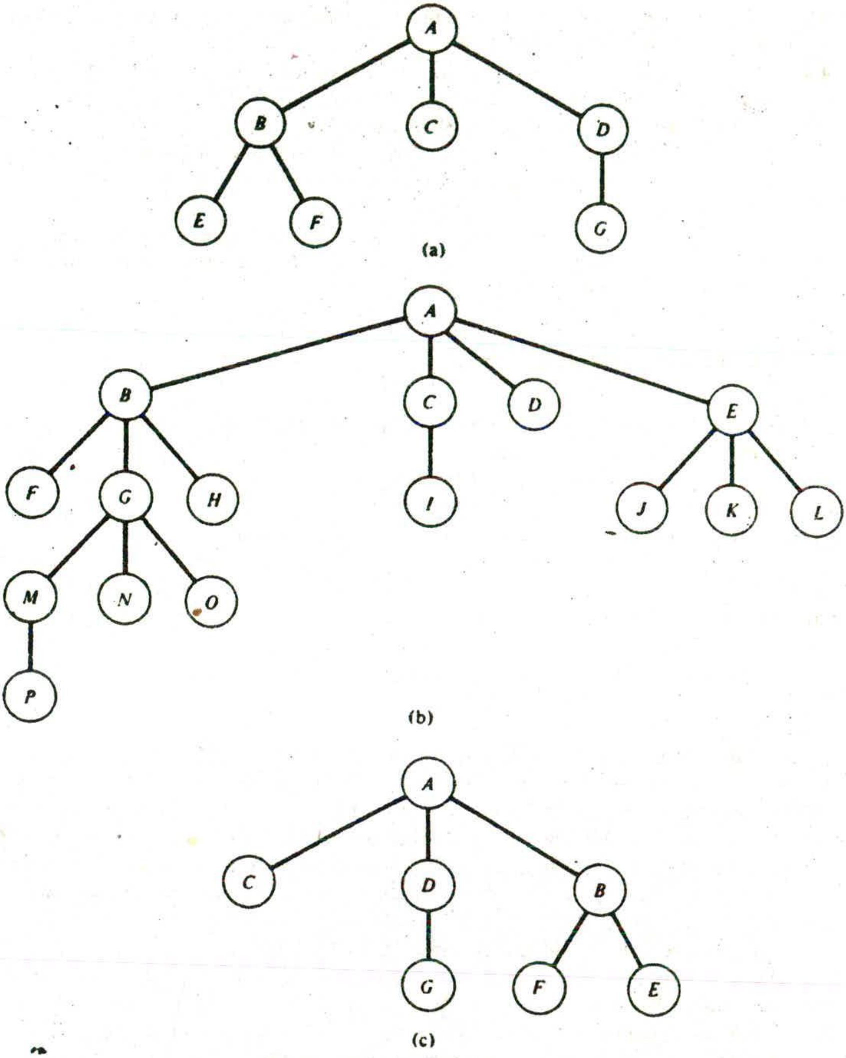
A ***tree*** is a finite nonempty set of elements in which one element is called the ***root*** and the remaining elements are partitioned into $m >= 0$ disjoint subsets, each of which is itself a tree. Each element in a tree is called a ***node*** of the tree.

Figure 5.5.1 illustrates some trees. Each node may be the root of a tree with zero or more subtrees. A node with no subtrees is a ***leaf***. We use the terms ***father, son, brother, ancestor, descendant, level,*** and ***depth*** in the same sense that we used them for binary trees. We also define the ***degree*** of a node in a tree as the number of its sons. Thus in Figure 5.5.1a, node *C* has degree 0 (and is therefore a leaf), node *D* has degree 1, node *B* has degree 2, and node *A* has degree 3. There is no upper limit on the degree of a node.

Let us compare the trees of Figure 5.5.1a and c. They are equivalent as trees. Each has *A* as its root and three subtrees. One of those subtrees has root *C* with no subtrees, another has root *D* with a single subtree rooted at *G*, and the third has root *B* with two subtrees rooted at *E* and *F*. The only difference between the two illustrations is the order in which the subtrees are arranged. The definition of a tree makes no distinction among subtrees of a general tree, unlike a binary tree, in which a distinction is made between the left and right subtrees.

An ***ordered tree*** is defined as a tree in which the subtrees of each node form an ordered set. In an ordered tree we may speak of the first, second, or last son of a particular node. The first son of a node in an ordered tree is often called the ***oldest*** son of that node, and the last son is called the ***youngest***. Although the trees of Figures 5.5.1a and c are equivalent as unordered trees, they are different as ordered trees. In the remainder of this chapter we use the word "tree" to refer to "ordered tree." A ***forest*** is an ordered set of ordered trees.

The question arises whether a binary tree is a tree. Every binary tree except for the empty binary tree is indeed a tree. However, not every tree is binary. A tree node may have more than two sons, whereas a binary tree node may not. Even a tree whose



(a)

(b)

(c)

Figure 5.5.1 Examples of trees.

nodes have at most two sons is not necessarily a binary tree. This is because an only son in a general tree is not designated as being a "left" or a "right" son, whereas in a binary tree, every son must be either a "left" son or a "right" son. In fact, although a nonempty binary tree is a tree, the designations of left and right have no meaning within the context of a tree (except perhaps to order the two subtrees of those nodes with two sons). A nonempty binary tree is a tree each of whose nodes has a maximum of two subtrees which have the added designation of "left" or "right."

C Representations of Trees

How can an ordered tree be represented in C? Two alternatives immediately come to mind: an array of tree nodes may be declared or a dynamic variable may be allocated for each node created. However, what should the structure of each individual node be? In the representation of a binary tree, each node contains an information field and two pointers to its two sons. But how many pointers should a tree node contain? The number of sons of a node is variable and may be as large or as small as desired. If we arbitrarily declare

```
#define MAXSONS 20

struct treenode {
   int info;
   struct treenode *father;
   struct treenode *sons[MAXSONS];
};
```

we are restricting the number of sons a node may have to a maximum of 20. Although in most cases this will be sufficient, it is sometimes necessary to create dynamically a node with 21 or 100 sons. Far worse than this remote possibility is the fact that twenty units of storage are reserved for each node in the tree even though a node may actually have only 1 or 2 (or even 0) sons. This is a tremendous waste of space.

One alternative is to link all the sons of a node together in a linear list. Thus the set of available nodes (using the array implementation) might be declared as follows:

```
#define MAXNODES 500

struct treenode {
   int info;
   int father;
   int son;
   int next;
};
struct treenode node[MAXNODES];
```

node[*p*].*son* points to the oldest son of *node*[*p*], and *node*[*p*].*next* points to the next younger brother of *node*[*p*].

Alternatively, a node may be declared as a dynamic variable:

```
struct treenode {
   int info;
   struct treenode *father;
   struct treenode *son;
   struct treenode *next;
};
typedef struct treenode *NODEPTR;
```

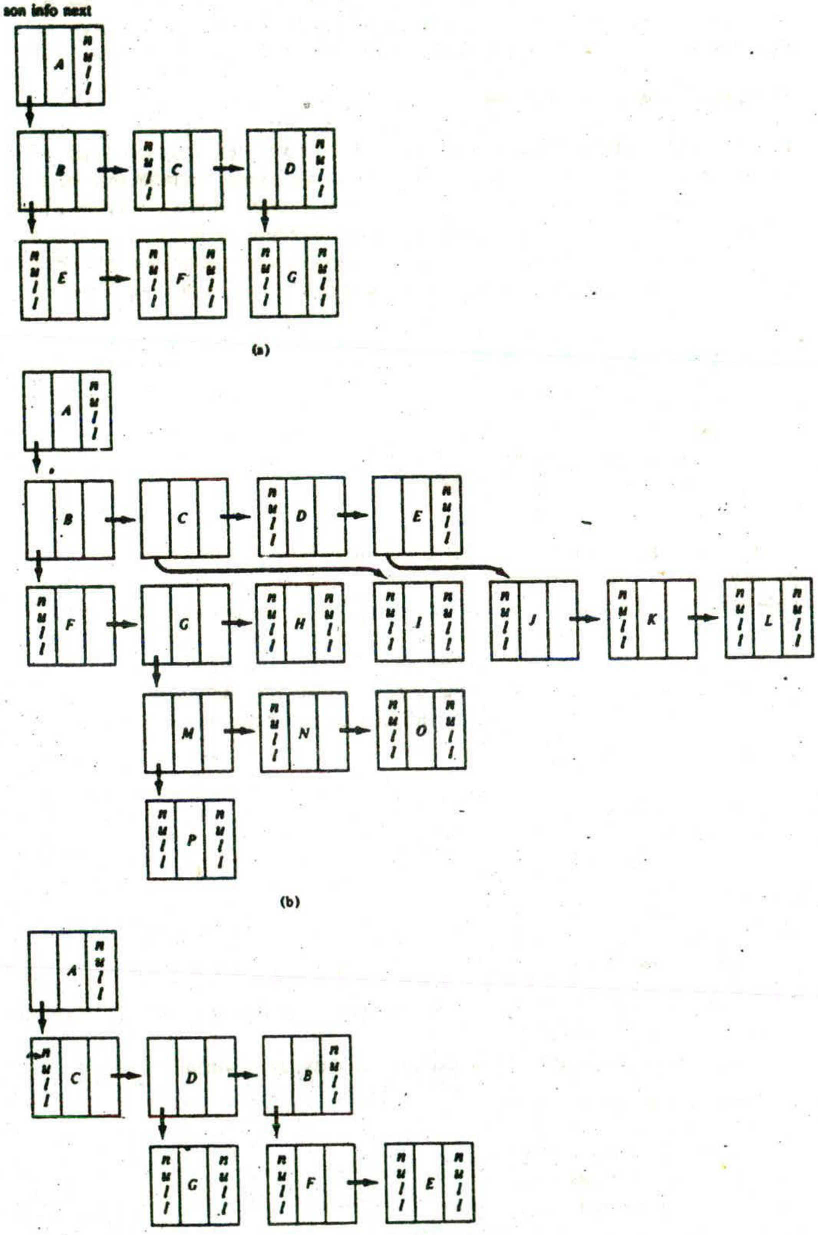


Figure 5.5.2 Tree representations. (See reference on page 309.)

If all traversals are from a node to its sons, the *father* field may be omitted. Figure 5.5.2 illustrates the representations of the trees of Figure 5.5.1 under these methods if no *father* field is needed.

Even if it is necessary to access the father of a node, the *father* field can be omitted by placing a pointer to the father in the *next* field of the youngest son instead of leaving it *null*. An additional logical field could then be used to indicate whether the *next* field points to a "real" next son or to the father. Alternatively (in the array of nodes implementation), the contents of the *next* field can contain negative as well as positive indices. A negative value would indicate that the *next* field points to the node's father rather than to its brother, and the absolute value of the *next* field yields the actual pointer. This is similar to the representation of threads in binary trees. Of course, in either of these two latter methods, accessing the father of an arbitrary node would require a traversal of the list of its younger sons.

If we think of *son* as corresponding to the *left* pointer of a binary tree node and *next* as corresponding to its *right* pointer, this method actually represents a general ordered tree by a binary tree. We may picture this binary tree as the original tree tilted 45 degrees with all father–son links removed except for those between a node and its oldest son, and with links added between each node and its next younger brother. Figure 5.5.3 illustrates the binary trees corresponding to the trees of Figure 5.5.1.

In fact, a binary tree may be used to represent an entire forest, since the *next* pointer in the root of a tree can be used to point to the next tree of the forest. Figure 5.5.4 illustrates a forest and its corresponding binary tree.

Tree Traversals

The traversal methods for binary trees induce traversal methods for forests. The preorder, inorder, or postorder traversals of a forest may be defined as the preorder, inorder, or postorder traversals of its corresponding binary tree. If a forest is represented as a set of dynamic variable nodes with *son* and *next* pointers as previously, a C routine to print the contents of its nodes in inorder may be written as follows:

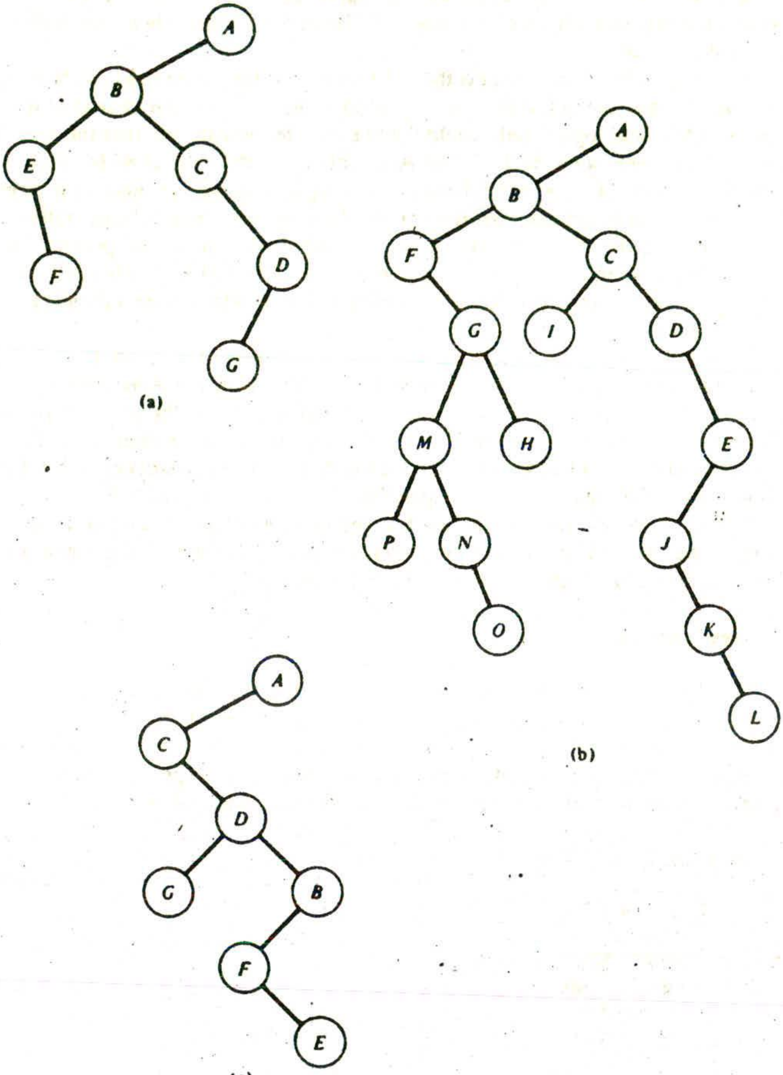
```
void intrav(NODEPTR p)
{
   if (p != NULL) {
      intrav(p->son);
      printf("%d\n", p->info);
      intrav(p->next);
   } /* end if */
} /* end intrav */
```

Routines for preorder and postorder traversals are similar.

These traversals of a forest may also be defined directly as follows:

PREORDER

1. Visit the root of the first tree in the forest.
2. Traverse in preorder the forest formed by the subtrees of the first tree, if any.
3. Traverse in preorder the forest formed by the remaining trees in the forest, if any.

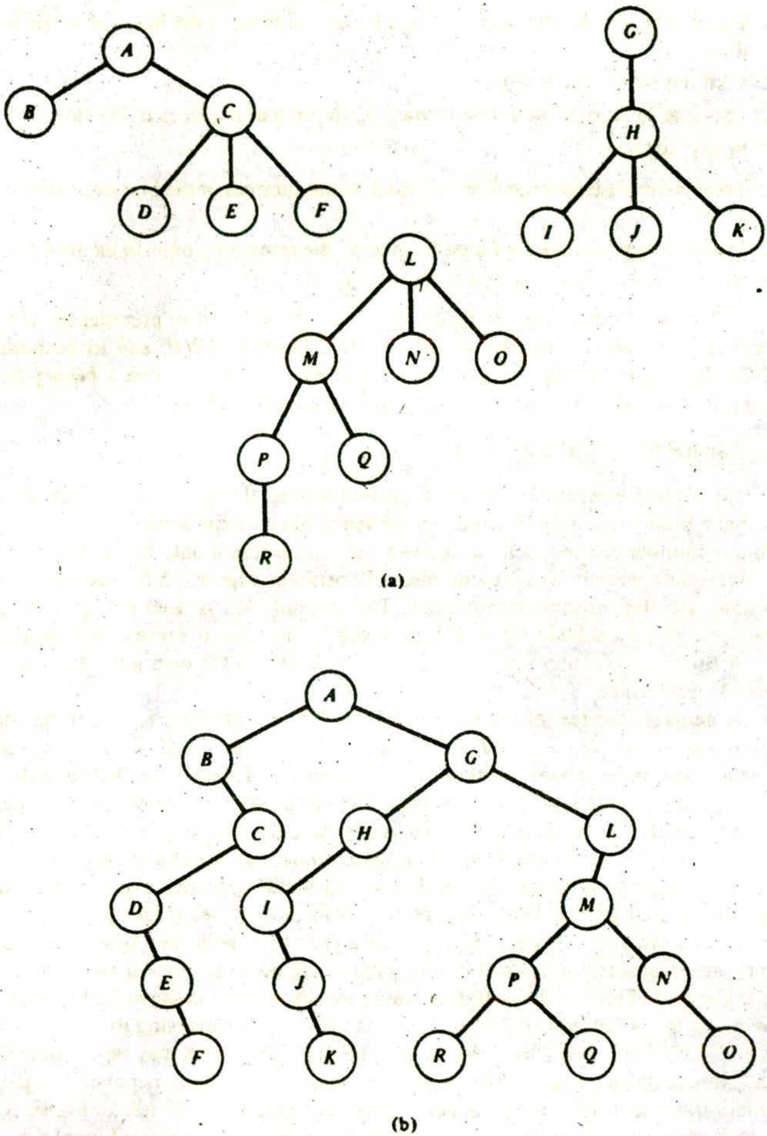


(a)

(b)

(c)

Figure 5.5.3 Binary trees corresponding to trees of Figure 5.5.1.



(a)

(b)

Figure 5.5.4 Forest and its corresponding binary tree.

INORDER

1. Traverse in inorder the forest formed by the subtrees of the first tree in the forest, if any.
2. Visit the root of the first tree.
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any.

POSTORDER

1. Traverse in postorder the forest formed by the subtrees of the first tree in the forest, if any.
2. Traverse in postorder the forest formed by the remaining trees in the forest, if any.
3. Visit the root of the first tree in the forest.

The nodes of the forest in Figure 5.5.4 a may be listed in preorder as *ABCDEFGHIJKLMPRQNO*, in inorder as *BDEFCAIJKHGRPQMNOL* and in postorder as *FEDCBKJIHRQPONMLGA*. Let us call a traversal of a binary tree a ***binary traversal***, and a traversal of an ordered general tree a ***general traversal***.

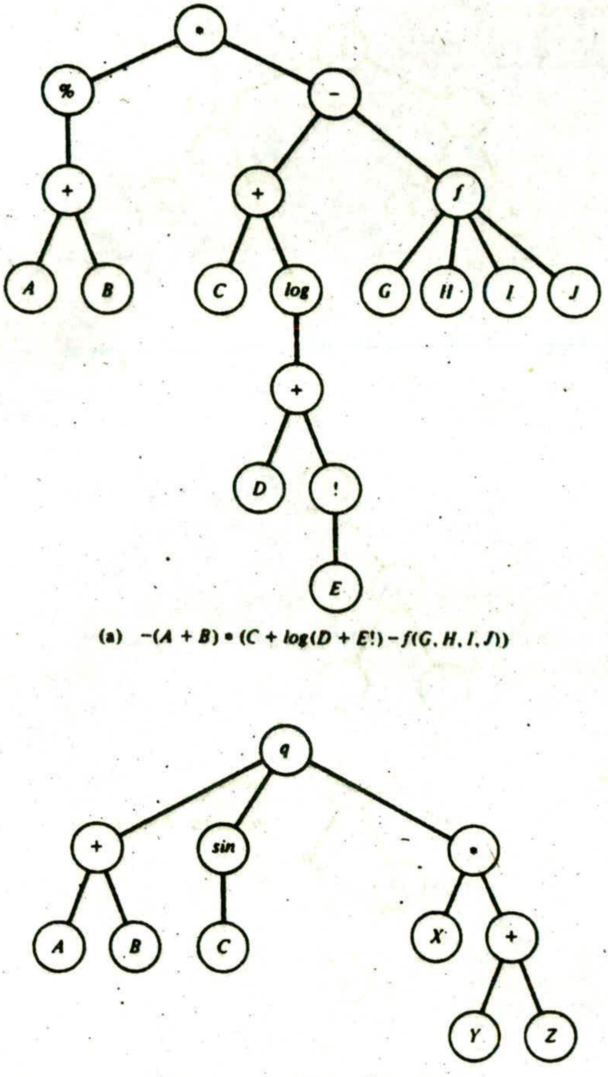
General Expressions as Trees

An ordered tree may be used to represent a general expression in much the same way that a binary tree may be used to represent a binary expression. Since a node may have any number of sons, nonleaf nodes need not represent only binary operators but can represent operators with any number of operands. Figure 5.5.5 illustrates two expressions and their tree representations. The symbol "%" is used to represent unary negation to avoid confusing it with binary subtraction that is represented by a minus sign. A function reference such as $f(g,h,i,j)$ is viewed as the operator f applied to the operands $g,h,i,$ and j.

A general traversal of the trees of Figure 5.5.5 in preorder results in the strings $* \% + AB - +C \log +D \,!\, EFGHIJ$ and $q + AB \; sinC \; * X + YZ$, respectively. These are the prefix versions of those two expressions. Thus we see that preorder general traversal of an expression tree produces its prefix expression. Inorder general traversal yields the respective strings $AB + \% \, CDE \,! \; + log + GHIJF - *$ and $AB + C \; sin \, XYZ + * q$, which are the postfix versions of the two expressions.

The fact that an inorder general traversal yields a postfix expression might be surprising at first glance. However, the reason for it becomes clear upon examination of the transformation that takes place when a general ordered tree is represented by a binary tree. Consider an ordered tree in which each node has zero or two sons. Such a tree is shown in Figure 5.5.6a, and its binary tree equivalent is shown in Figure 5.5.6b. Traversing the binary tree of Figure 5.5.6b is the same as traversing the ordered tree of Figure 5.5.6a. However, a tree such as the one in Figure 5.5.6a may be considered as a binary tree in its own right, rather than as an ordered tree. Thus it is possible to perform a binary traversal (rather than a general traversal) directly on the tree of Figure 5.5.6a. Beneath that figure are the binary traversals of that tree, and opposite Figure 5.5.6b are the binary traversals of the tree in that figure, which are the same as the traversals of the tree of Figure 5.5.6a if it is considered as an ordered tree.

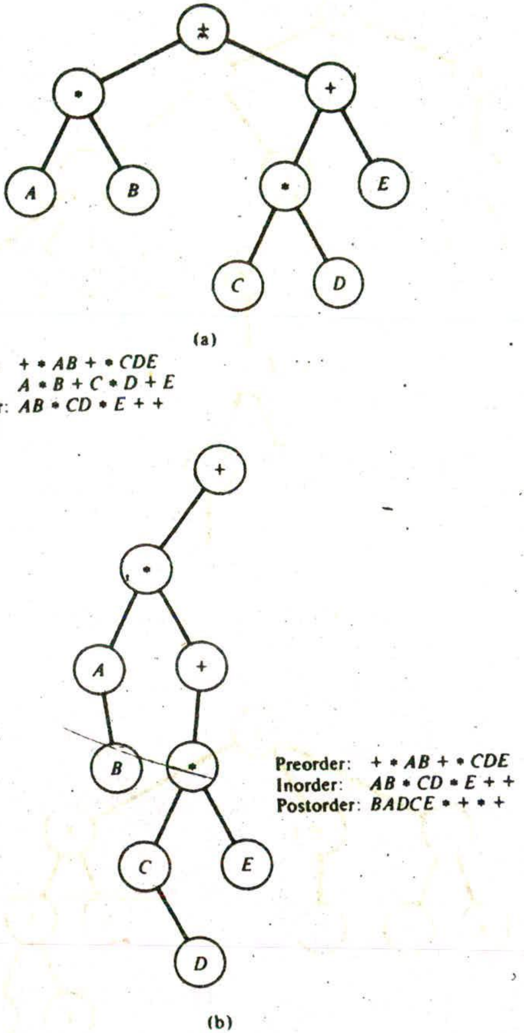
Note that the preorder traversals of the two binary trees are the same. Thus if a preorder traversal on a binary tree representing a binary expression yields the prefix of



(a) −(*A* + *B*) • (*C* + *log*(*D* + *E*!) − *f*(*G*, *H*, *I*, *J*))

(b) *q*(*A* + *B*, *sin*(*C*), *X* • (*Y* + *Z*))

Figure 5.5.5 Tree representation of an arithmetic expression.



(a)

Preorder: + * AB + * CDE
Inorder: A * B + C * D + E
Postorder: AB * CD * E + +

Preorder: + * AB + * CDE
Inorder: AB * CD * E + +
Postorder: BADCE * + * +

(b)

Figure 5.5.6

the expression, that traversal on an ordered tree representing a general expression that happens to have only binary operators yields prefix as well. However, the postorder traversals of the two binary trees are not the same. Instead, the inorder binary traversal of the second (which is the same as the inorder general traversal of the first if it is considered as an ordered tree) is the same as the postorder binary traversal of the first. Thus the inorder general traversal of an ordered tree representing a binary expression is equivalent to the postorder binary traversal of the binary tree representing that expression, which yields postfix.

Evaluating an Expression Tree

Suppose that it is desired to evaluate an expression whose operands are all numerical constants. Such an expression can be represented in C by a tree each of whose nodes is declared by

```
#define OPERATOR 0
#define OPERAND 1
struct treenode {
   short int utype;     /* OPERATOR or OPERAND */
   union {
      char operator[10];
      float val;
   } info;
   struct treenode *son;
   struct treenode *next;
};
typedef treenode *NODEPTR;
```

The *son* and *next* pointers are used to link together the nodes of a tree as previously illustrated. Since a node may contain information that may be either a number (operand) or a character string (operator), the information portion of the node is a union component of the structure.

We wish to write a C function *evaltree(p)* that accepts a pointer to such a tree and returns the value of the expression represented by that tree. The routine *evalbintree* presented in Section 5.2 performs a similar function for binary expressions. *evalbintree* utilizes a function *oper*, which accepts an operator symbol and two numerical operands and returns the numerical result of applying the operator to the operands. However, in the case of a general expression we cannot use such a function, since the number of operands (and hence the number of arguments) varies with the operator. We therefore introduce a new function, *apply(p)*, which accepts a pointer to an expression tree that contains a single operator and its numerical operands and returns the result of applying the operator to its operands. For example, the result of calling the function *apply* with parameter *p* pointing to the tree in Figure 5.5.7 is 24. If the root of the tree that is passed to *evaltree* represents an operator, each of its subtrees is replaced by tree nodes representing the numerical results of their evaluation so that the function *apply* may be called. As the expression is evaluated, the tree nodes representing operands are freed and operator nodes are converted to operand nodes.

We present a recursive procedure *replace* that accepts a pointer to an expression tree and replaces the tree with a tree node containing the numerical result of the expression's evaluation.

```
void replace(NODEPTR p)
{
   float value;
   NODEPTR q, r;
```

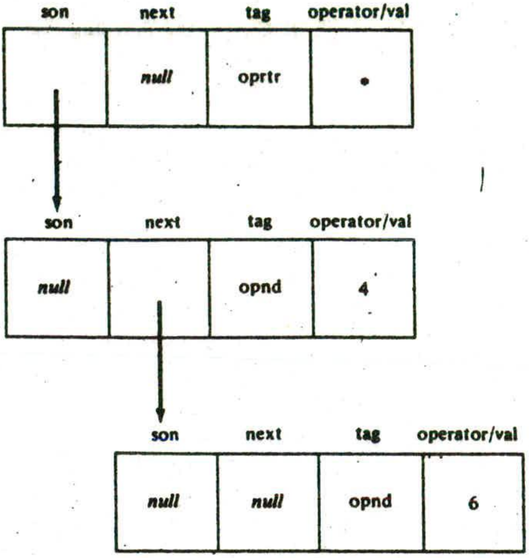


Figure 5.5.7 Expression tree.

```
if (p->utype == OPERATOR) {
   /* the tree has an operator */
   /*        as its root      */
   q = p->son;
   while (q != NULL) {
      /*   replace each of its subtrees  */
      /*          by operands            */
      replace(q);
      q = q->next;
   }  /* end while */
   /* apply the operator in the root to  */
   /*    the operands in the subtrees    */
   value = apply(p);
   /* replace the operator by the result */
   p->utype = OPERAND;
   p->val = value;
   /*         free all the subtrees      */
   q = p->son;
   p->son = NULL;
   while (q != NULL) {
      r = q;
      q = q->next;
      free(r);
   }  /* end while */
```

```
      } /* end if */
   } /* end replace */
```

The function *evaltree* may now be written as follows:

```
float evaltree(NODEPTR p)
{
   NODEPTR q;

   replace(p);
   return(p->val);
   free(p);
} /* end evaltree */
```

After calling *evaltree(p)* the tree is destroyed and the value of *p* is meaningless. This is a case of a ***dangling pointer*** in which a pointer variable contains the address of a variable that has been freed. C programmers who use dynamic variables should be careful to recognize such pointers and not to use them subsequently.

Constructing a Tree

A number of operations are frequently used in constructing a tree. We now present some of these operations and their C implementations. In the C representation, we assume that father pointers are not needed, so that the *father* field is not used and the *next* pointer in the youngest node is *null*. The routines would be slightly more complex and less efficient if this were not the case.

The first operation that we examine is *setsons*. This operation accepts a pointer to a tree node with no sons and a linear list of nodes linked together through the *next* field. *setsons* establishes the nodes in the list as the sons of the node in the tree. The C routine to implement this operation is straightforward (we use the dynamic storage implementation):

```
void setsons(NODEPTR p, NODEPTR list)
{
   /* p points to a tree node, list to a list  */
   /* of nodes linked together through their   */
   /*            next fields                   */
   if (p == NULL) {
      printf("invalid insertion\n");
      exit(1);
   } /* end if */
   if (p->son != NULL) {
      printf("invalid insertion\n");
      exit(1);
   } /* end if */
   p->son = list;
} /* end setsons */
```

Another common operation is *addson(p,x)*, in which *p* points to a node in a tree and it is desired to add a node containing *x* as the youngest son of *node(p)*. The C routine to implement *addson* is as follows. The routine calls the auxiliary function *getnode*, which allocates a node and returns a pointer to it.

```
void addson(NODEPTR p, int x)
{
   NODEPTR q;

   if (p == NULL) {
      printf("void insertion\n");
      exit(1);
   }  /* end if */
   /* the pointer q traverses the list of sons */
   /*       of p. r is one node behind q        */
   r = NULL;
   q = p->son;
   while (q != NULL) {
      r = q;
      q = q->next;
   }  /* end while */
   /* At this point, r points to the youngest */
   /*  son of p, or is null if p has no sons  */
   q = getnode();
   q->info = x;
   q->next = NULL;
   if (r == NULL)                 /* p has no sons */
      p->son = q;
   else
      r->next = q;
}  /* end addson */
```

Note that to add a new son to a node, the list of existing sons must be traversed. Since adding a son is a common operation, a representation is often used that makes this operation more efficient. Under this alternative representation, the list of sons is ordered from youngest to oldest rather than vice versa. Thus *son(p)* points to the youngest son of *node(p)*, and *next(p)* points to its next older brother. Under this representation the routine *addson* may be written as follows:

```
void addson(NODEPTR p, int x)
{
   NODEPTR q;

   if (p == NULL) {
      printf("invalid insertion\n");
      exit(1);
   }  /* end if */
```

```
      q = getnode();
      q->info = x;
      q->next = p->son;
      p->son = q;
   } /* end addson */
```

EXERCISES

5.5.1. How many trees exist with *n* nodes?

5.5.2. How many trees exist with *n* nodes and maximum level *m*?

5.5.3. Prove that if *m* pointer fields are set aside in each node of a general tree to point to a maximum of *m* sons, and if the number of nodes in the tree is *n*, the number of null son pointer fields is $n * (m - 1) + 1$.

5.5.4. If a forest is represented by a binary tree as in the text, show that the number of null right links is 1 greater than the number of nonleaves of the forest.

5.5.5. Define the ***breadth-first order*** of the nodes of a general tree as the root followed by all nodes on level 1, followed by all nodes on level 2, and so on. Within each level, the nodes should be ordered so that children of the same father appear in the same order as they appear in the tree and, if *n*1 and *n*2 have different fathers, *n*1 appears before *n*2 if the father of *n*1 appears before the father of *n*2. Extend the definition to a forest. Write a C program to traverse a forest represented as a binary tree in breadth-first order.

5.5.6. Consider the following method of transforming a general tree, *gt*, into a strictly binary tree, *bt*. Each node of *gt* is represented by a leaf of *bt*. If *gt* consists of a single node, *b* consists of a single node. Otherwise *bt* consists of a new root node and a left subtree *lt*, and a right subtree, *rt*. *lt* is the strictly binary tree formed recursively from the oldes subtree of *gt*, and *rt* is the strictly binary tree formed recursively from *gt* without its oldest subtree. Write a C routine to convert a general tree into a strictly binary tree.

5.5.7. Write a C function *compute* that accepts a pointer to a tree representing an expression with constant operands and returns the result of evaluating the expression without destroying the tree.

5.5.8. Write a C program to convert an infix expression into an expression tree. Assume that all nonbinary operators precede their operands. Let the input expression be represented as follows: an operand is represented by the character 'N' followed by a number, an operator by the character 'T' followed by a character representing the operator, and a function by the character 'F' followed by the name of the function.

5.5.9. Consider the definitions of an expression, a term, and a factor given at the end of Section 3.2. Given a string of letters, plus signs, asterisks and parentheses that forms a valid expression, a ***parse tree*** can be formed for the string. Such a tree is illustrated in Figure 5.5.8 for the string "$(A + B) * (C + D)$". Each node in such a tree represents a substring and contains a letter (*E* for expression, *T* for term, *F* for factor, or *S* for symbol) and two integers. The first is the position within the input string where the substring represented by that node begins, and the second is the length of the substring. (The substring represented by each node is shown below that node in the figure.) The leaves are all *S* nodes and represent single symbols of the original input. The root of the tree must be an *E* node. The sons of any non-*S* node *N* represent the substrings which make up the grammatical object represented by *N*. Write a C routine that accepts such a string and constructs a parse tree for it.

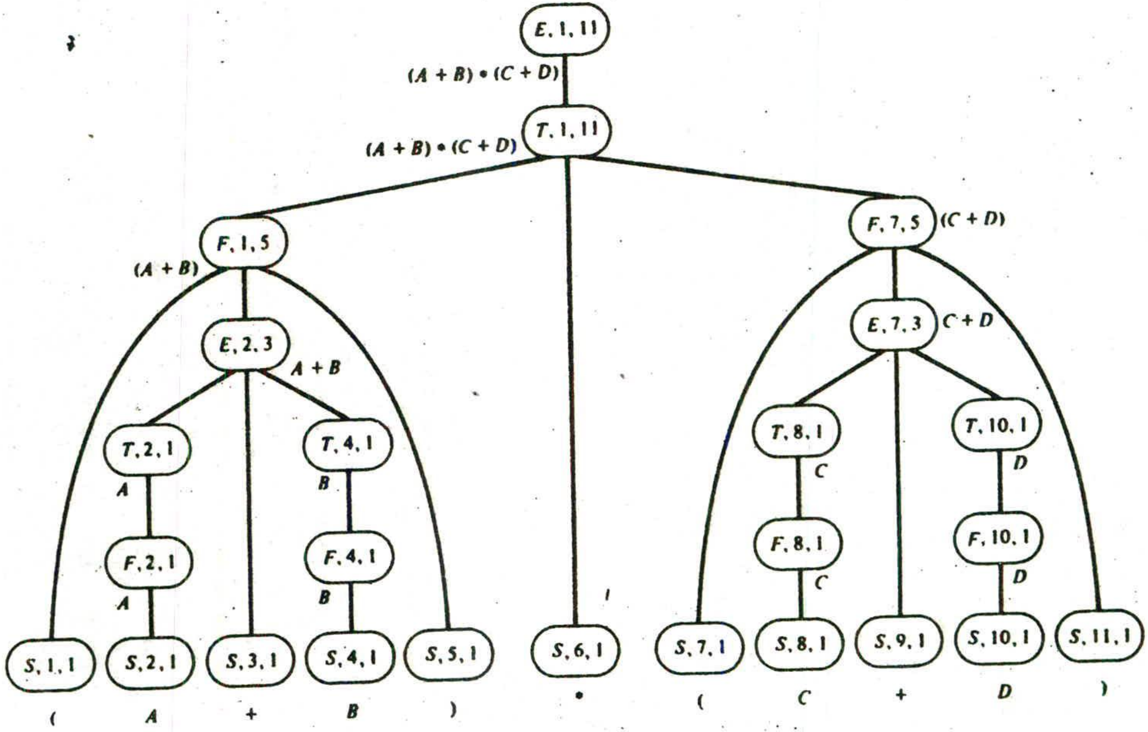


Figure 5.5.8 Parse tree for the string (A + B) • (C + D).

5.6 EXAMPLE: GAME TREES

One application of trees is to game playing by computer. We illustrate this application by writing a C program to determine the "best" move in tic-tac-toe from a given board position.

Assume that there is a function *evaluate* that accepts a board position and an indication of a player (X or O) and returns a numerical value that represents how "good" the position seems to be for that player (the larger the value returned by *evaluate*, the better the position). Of course, a winning position yields the largest possible value, and a losing position yields the smallest. An example of such an evaluation function for tic-tac-toe is the number of rows, columns, and diagonals remaining open for one player minus the number remaining open for his or her opponent (except that the value 9 would be returned for a position that wins, and −9 for a position that loses). This function does not "look ahead" to consider any possible board positions that might result from the current position; it merely evaluates a static board position.

Given a board position, the best next movc ould be determined by considering all possible moves and resulting positions. The mcve selected should be the one that results in the board position with the highest evaluation. Such an analysis, however, does not necessarily yield the best move. Figure 5.6.1 illustrates a position and the five possible moves that X can make from that position. Applying the evaluation function just described to the five resulting positions yields the values shown. Four moves yield the same maximum evaluation, although three of them are distinctly inferior to the fourth. (The fourth position yields a certain vicıory for X, whereas the other three can be drawn by O.) In fact, the move that yields the smallest evaluation is as good or better than the moves that yield a higher evaluation. The static evaluation function, therefore, is not good enough to predict the outcome of the game. A better evaluation function could easily be produced for the game of tic-tac-toe (even if it were by the brute-force method of listing all positions and the appropriate response), but most games are too complex for static evaluators to determine the best response.

Suppose that it were possible to look ahead several moves. Then the choice of a move could be improved considerably. Define the ***look ahead level*** as the number of future moves to be considered. Starting at any position, it is possible to construct a tree of the possible board positions that may result from each move. Such a tree is called a ***game tree***. The game tree for the opening tic-tac-toe position with a look-ahead level of 2 is illustrated in Figure 5.6.2. (Actually other positions do exist, but because of

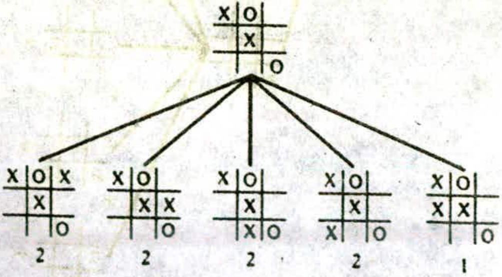


Figure 5.6.1

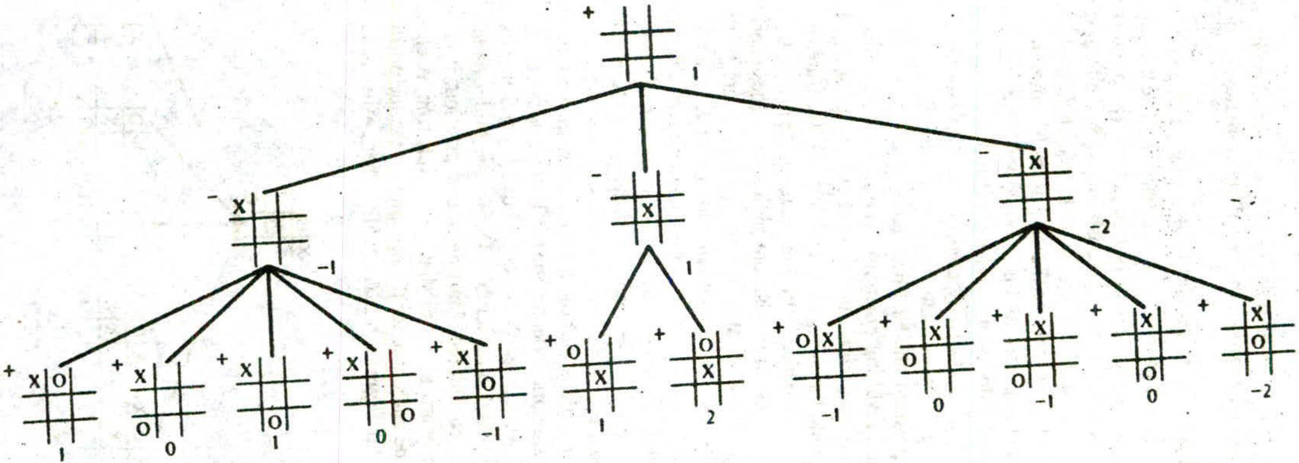


Figure 5.6.2 Game tree for tic-tac-toe.

symmetry considerations these are effectively the same as the positions shown.) Note that the maximum level (called the ***depth***) of the nodes in such a tree is equal to the look-ahead level.

Let us designate the player who must move at the root's game position as ***plus*** and his or her opponent as ***minus***. We attempt to find the best move for *plus* from the root's game position. The remaining nodes of the tree may be designated as ***plus nodes*** or ***minus nodes,*** depending upon which player must move from that node's position. Each node of Figure 5.6.2 is marked as a plus or as a minus node.

Suppose that the game positions of all the sons of a plus node have been evaluated for player *plus*. Then clearly, *plus* should choose the move that yields the maximum evaluation. Thus, the value of a *plus* node to player *plus* is the maximum of the values of its sons. On the other hand, once *plus* has moved, *minus* will select the move that yields the minimum evaluation for player *plus*. Thus the value of a *minus* node to player *plus* is the minimum of the values of its sons.

Therefore to decide the best move for player *plus* from the root, the positions in the leaves must be evaluated for player *plus* using a static evaluation function. These values are then moved up the game tree by assigning to each plus node the maximum of its sons' values and to each minus node the minimum of its sons' values, on the assumption that *minus* will select the move that is worst for *plus*. The value assigned to each node of Figure 5.6.2 by this process is indicated in that figure immediately below the node.

The move that *plus* should select, given the board position in the root node, is the one that maximizes its value. Thus the opening move for X should be the middle square, as illustrated in Figure 5.6.2. Figure 5.6.3 illustrates the determination of O's best reply. Note that the designation of "*plus*" and "*minus*" depends on whose move is being calculated. Thus, in Figure 5.6.2 X is designated as *plus*, whereas in Figure 5.6.3

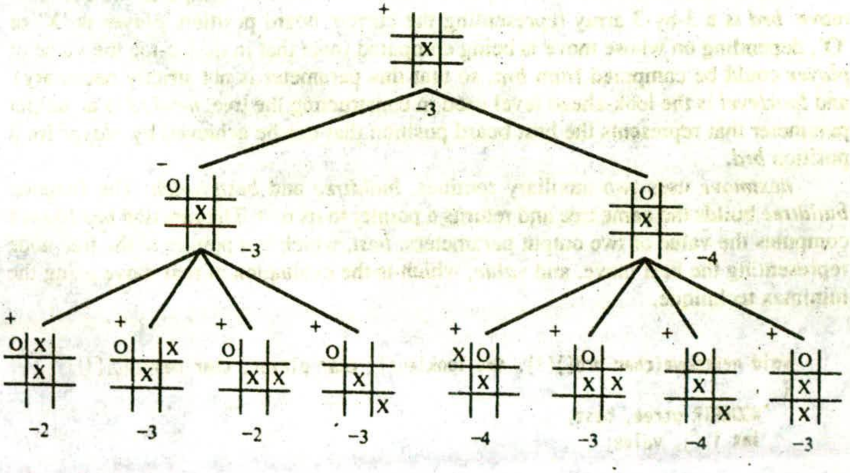


Figure 5.6.3 Computing O's reply.

O is designated as *plus*. In applying the static evaluation function to a board position, the value of the position to whichever player is designated as *plus* is computed. This method is called the ***minimax*** method, since, as the tree is climbed the maximum and minimum functions are applied alternately.

The best move for a player from a given position may be determined by first constructing the game tree and applying a static evaluation function to the leaves. These values are then moved up the tree by applying the minimum and maximum at minus and plus nodes, respectively. Each node of the game tree must include a representation of the board and an indication of whether the node is a plus node or a minus node. Nodes may therefore be declared by

```
struct nodetype {
   char board[3][3];
   int turn;
   struct nodetype *son;
   struct nodetype *next;
};

typedef struct nodetype *NODEPTR;
```

p− >*board*[*i*][*j*] has the value 'X', 'O', or ' ', depending on whether the square in row *i* and column *j* of that node is occupied by either of the players or is unoccupied. *p*− >*turn* has the value +1 or −1, depending on whether the node is a plus or minus node, respectively. The remaining two fields of a node are used to position the node within the tree. *p*− >*son* points to the oldest son of the node, and *p*− >*next* points to its next younger brother. We assume that the foregoing declaration is global, that an available list of nodes has been established, and that appropriate *getnode* and *freenode* routines have been written.

The C function *nextmove*(*brd, player, looklevel, newbrd*) computes the best next move. *brd* is a 3-by-3 array representing the current board position, *player* is 'X' or 'O', depending on whose move is being computed (note that in tic-tac-toe the value of *player* could be computed from *brd*, so that this parameter is not strictly necessary), and *looklevel* is the look-ahead level used in constructing the tree. *newbrd* is an output parameter that represents the best board position that can be achieved by *player* from position *brd*.

nextmove uses two auxiliary routines, *buildtree* and *bestbranch*. The function *buildtree* builds the game tree and returns a pointer to its root. The function *bestbranch* computes the value of two output parameters: *best*, which is a pointer to the tree node representing the best move, and *value*, which is the evaluation of that move using the minimax technique.

```
void nextmove(char brd[][3], int looklevel, char player, char newbrd[][3])
{
   NODEPTR ptree, best;
   int i, j, value;
   ptree = buildtree(brd, looklevel);
   bestbranch(ptree, player, &best, &value);
   for (i=0; i < 3; ++i)
      for (j=0; j < 3; ++j)
         newbrd[i][j] = best->board[i][j];
} /* end nextmove */
```

The function *buildtree* returns a pointer to the root of a game tree. It uses the auxiliary function *getnode*, which allocates storage for a node and returns a pointer to it. It also uses a routine *expand(p, plevel, depth)*, in which *p* is a pointer to a node in a game tree, *plevel* is its level, and *depth* is the depth of the game tree that is to be constructed. *expand* produces the subtree rooted at *p* to the proper depth.

```
NODEPTR buildtree(char brd[][3], int looklevel)
{
   NODEPTR ptree;
   int i, j;

   /* create the root of the tree and initialize it */
   tree = getnode();
   for (i=0; i < 3; ++i)
      for (j=0; j < 3; ++j)
         ptree->board[i][j] = brd[i][j];
   /* the root is a plus node by definition */
   ptree->turn = 1;
   ptree->son = NULL;
   ptree->next = NULL;
   /* create the rest of the game tree */
   expand(ptree, 0, looklevel);
   return(ptree);
} /* end buildtree */
```

expand may be implemented by generating all board positions that may be obtained from the board position pointed to by *p* and establishing them as the sons of *p* in the game tree. *expand* then calls itself recursively using these sons as parameters until the desired depth is reached. *expand* uses an auxiliary function *generate*, which accepts a board position *brd* and returns a pointer to a list of nodes containing the board positions that can be obtained from *brd*. This list is linked together by the *next* field. We leave the coding of *generate* as an exercise for the reader.

```
void expand(NODEPTR p, int plevel, int depth)
{
   NODEPTR q;

   if (plevel < depth) {
      /* p is not at the maximum level */
      q = generate(p->board);
      p->son = q;
```

```
        while (q != NULL) {
           /* traverse the list of nodes */
           if (p->turn == 1)
              q->turn = -1;
           else
              q->turn = 1;
           q->son = NULL;
           expand(q, plevel+1, depth);
           q = q->next;
        } /* end while */
     } /* end if */
  } /* end expand */
```

Once the game tree has been created, *bestbranch* evaluates the nodes of the tree. When a pointer to a leaf is passed to *bestbranch*, it calls a function *evaluate* that statically evaluates the board position of that leaf for the player whose move we are determining. The coding of *evaluate* is left as an exercise. When a pointer to a nonleaf is passed to *bestbranch*, the routine calls itself recursively on each of its sons and then assighs the maximum of its sons' values to the nonleaf if it is a plus node, and the minimum if it is a minus node. *bestbranch* also keeps track of which son yielded this minimum or maximum value.

If $p->turn$ is -1, the node pointed to by p is a minus node and it is to be assigned the minimum of the values assigned to its sons. If, however, $p->turn$ is $+1$, the node pointed to by p is a plus node and its value should be the maximum of the values assigned to the sons of the node. If $min(x,y)$ is the minimum of x and y, and $max(x,y)$ is their maximum, $min(x,y) = -max(-x,-y)$ (you are invited to prove this as a trivial exercise). Thus, the correct maximum or minimum can be found as follows: in the case of a plus node, compute the maximum; in the case of a minus node, compute the maximum of the negatives of the values and then reverse the sign cf the result. These ideas are incorporated into *bestbranch*. The output parameters **pbest* and **pvalue* are, respectively, a pointer to that son of the tree's root that maximizes its value and the value of that son that has now been assigned to the root.

```
void bestbranch(NODEPTR pnd, char player, NODEPTR *pbest,
                                            int *pvalue)
{
   NODEPTR p, pbest2;
   int val;

   if (pnd->son == NULL) {
      /* pnd is a leaf */
      *pvalue = evaluate(pnd->board, player);
      *pbest = pnd;
   }
   else {
      /* the node is not a leaf, traverse the list of sons */
      p = pnd->son;
      bestbranch(p, player, pbest, pvalue);
      *pbest = p;
      if (pnd.turn == -1)
         *pvalue = -*pvalue;
      p = p->next;
      while (p != NULL) {
         bestbranch(p, player, &pbest2, &val);
         if (pnd->turn == -1)
            val = -val;
         if (val > *pvalue) {
            *pvalue = val;
            *pbest = p;
         } /* {end if */
         p = p->next;
      } /* end while */
      if (pnd->turn == -1)
         *pvalue = -*pvalue;
   } /* end if */
} /* end bestbranch */
```

EXERCISES

5.6.1. Examine the routines of this section and determine whether all the parameters are actually necessary. How would you revise the parameter lists?

5.6.2. Write the C routines *generate* and *evaluate* as described in the text.

5.6.3. Rewrite the programs of this and the preceding section under the implementation in which each tree node includes a field *father* containing a pointer to its father. Under which implementation are they more efficient?

5.6.4. Write nonrecursive versions of the routines *expand* and *bestbranch* given in the text.

5.6.5. Modify the routine *bestbranch* in the text so that the nodes of the tree are freed after they are no longer needed.

5.6.6. Combine the processes of building the game tree and evaluating its nodes into a single process so that the entire game tree need not exist at any one time and nodes are freed when no longer necessary.

5.6.7. Modify the program of the previous exercise so that if the evaluation of a minus node is greater than the minimum of the values of its father's older brothers, the program does not bother expanding that minus node's younger brothers, and if the evaluation of a plus node is less than the maximum of the values of its father's older brothers, the program does not bother expanding that plus node's younger brothers. This method is called the ***alpha-beta minimax*** method. Explain why it is correct.

5.6.8. The game of ***kalah*** is played as follows: Two players each have seven holes, six of which are called ***pits*** and the seventh a ***kalah***. These are arranged according to the following diagram.

```
Player 1

K P P P P P P
  P P P P P P K

Player 2
```

Initially there are six stones in each pit and no stones in either kalah, so that the opening position looks like this:

```
0666666
 6666660
```

The players alternate turns, each turn consisting of one or more moves. To make a move, a player chooses one of his or her nonempty pits. The stones are removed from that pit and are distributed counterclockwise into the pits and into that player's kalah (the opponent's kalah is skipped), one stone per hole, until there are no stones remaining. For example, if player 1 moves first, a possible opening move might result in the following board position:

```
1777770
 6666660
```

If a player's last stone lands in his or her own kalah, the player gets another move. If the last stone lands in one of the player's own pits that is empty, that stone and the stones in the opponent's pit directly opposite are removed and placed in the player's kalah. The game ends when either player has no stones remaining in his or her pits. At that point, all the stones in the opponent's pits are placed in the opponent's kalah and the game ends. The player with the most stones in his or her kalah is the winner. Write a program that accepts a kalah board position and an indication of whose turn it is and produces that player's best move.

5.6.9. How would you modify the ideas of the tic-tac-toe program to compute the best move in a game that contains an element of chance, such as backgammon?

5.6.10. Why have computers been programmed to play perfect tic-tac-toe but not perfect chess or checkers?

5.6.11. The game of ***nim*** is played as follows: Some number of sticks are placed in a pile. Two players alternate in removing either one or two sticks from the pile. The player to remove the last stick is the loser. Write a C function to determine the best move in nim.

6 Sorting

Sorting and searching are among the most common ingredients of programming systems. In the first section of this chapter we discuss some of the overall considerations involved in sorting. In the remainder of the chapter we discuss some of the more common sorting techniques and the advantages or disadvantages of one technique over another. In the next chapter we discuss searching and some applications.

6.1 GENERAL BACKGROUND

The concept of an ordered set of elements is one that has considerable impact on our daily lives. Consider, for example, the process of finding a telephone number in a telephone directory. This process, called a ***search***, is simplified considerably by the fact that the names in the directory are listed in alphabetical order. Consider the trouble you might have in attempting to locate a telephone number if the names were listed in the order in which the customers placed their phone orders with the telephone company. In such a case, the names might as well have been entered in random order. Since the entries are sorted in alphabetical rather than in chronological order, the process of searching is simplified. Or consider the case of someone searching for a book in a library. Because the books are shelved in a specific order (Library of Congress, Dewey System, and so forth), each book is assigned a specific position relative to the others and can be retrieved in a reasonable amount of time (if it is there). Or consider a set of numbers sorted sequentially in a computer's memory. As we shall see in the next chapter, it is usually easier to find a particular element of such a set if the numbers are maintained in sorted order. In general, a set of items is kept sorted in order to either produce a report (to simplify manual retrieval of information, as in a telephone book or a library shelf) or to make machine access to data more efficient.

We now present some basic terminology. A ***file*** of size n is a sequence of n items $r[0]$, $r[1]$, . . . , $r[n-1]$. Each item in the file is called a ***record***. (The terms file and record are not being used here as in C terminology to refer to specific data structures. Rather, they are being used in a more general sense.) A key, $k[i]$, is associated with each record $r[i]$. The key is usually (but not always) a subfield of the entire record. The file is said to be ***sorted on the key*** if $i < j$ implies that $k[i]$ precedes $k[j]$ in some ordering on the keys. In the example of the telephone book, the file consists of all the entries in the book. Each entry is a record. The key upon which the file is sorted is the name field of the record. Each record also contains fields for an address and a telephone number.

A sort can be classified as being ***internal*** if the records that it is sorting are in main memory, or ***external*** if some of the records that it is sorting are in auxiliary storage. We restrict our attention to internal sorts.

It is possible for two records in a file to have the same key. A sorting technique is called ***stable*** if for all records i and j such that $k[i]$ equals $k[j]$, if $r[i]$ precedes $r[j]$ in the original file, $r[i]$ precedes $r[j]$ in the sorted file. That is, a stable sort keeps records with the same key in the same relative order that they were in before the sort.

A sort takes place either on the records themselves or on an auxiliary table of pointers. For example, consider Figure 6.1.1a, in which a file of five records is shown. If the file is sorted in increasing order on the numeric key shown, the resulting file is as shown in Figure 6.1.1b. In this case the actual records themselves have been sorted.

Suppose, however, that the amount of data stored in each of the records in the file of Figure 6.1.1a is so large that the overhead involved in moving the actual data is prohibitive. In this case an auxiliary table of pointers may be used so that these pointers are moved instead of the actual data, as shown in Figure 6.1.2. (This is called ***sorting by address***.) The table in the center is the file, and the table at the left is the initial table of pointers. The entry in position j in the table of pointers points to record j. During

	Key	Other fields
Record 1	4	*DDD*
Record 2	2	*BBB*
Record 3	1	*AAA*
Record 4	5	*EEE*
Record 5	3	*CCC*

File

(a) Original file.

1	*AAA*
2	*BBB*
3	*CCC*
4	*DDD*
5	*EEE*

File

(b) Sorted file.

Figure 6.1.1 Sorting actual records.

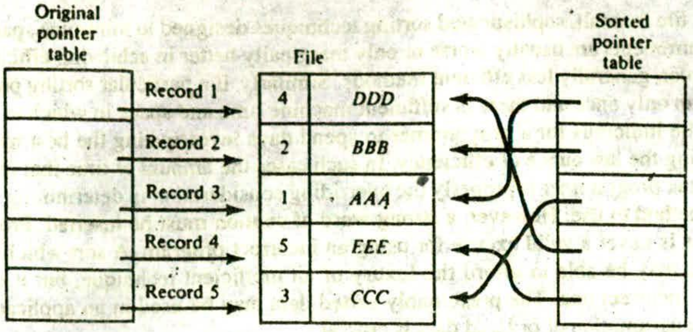


Figure 6.1.2 Sorting by using an auxiliary table of pointers.

the sorting process, the entries in the pointer table are adjusted sc that the final table is as shown at the right. Originally, the first pointer was to the first entry in the file; upon completion the first pointer is to the fourth entry in the table. Note that none of the original file entries are moved. In most of the programs in this chapter we illustrate techniques of sorting actual records. The extension of these techniques to sorting by address is straightforward and will be left as an exercise for the reader. (Actually, for the sake of simplicity, in the examples presented in this chapter we sort only the keys; we leave to the reader to modify the programs to sort full records.)

Because of the relationship between sorting and searching, the first question to ask in any application is whether or not a file should be sorted. Sometimes there is less work involved in searching a set of elements for a particular one than to first sort the entire set and to then extract the desired element. On the other hand, if frequent use of the file is required for the purpose of retrieving specific elements, it might be more efficient to sort the file. This is because the overhead of successive searches may far exceed the overhead involved in sorting the file once and subsequently retrieving elements from the sorted file. Thus it cannot be said that it is more efficient either to sort or not to sort. The programmer must make a decision based on individual circumstances. Once a decision to sort has been made, other decisions must be made, including what is to be sorted and what methods are to be used. There is no one sorting method that is universally superior to all others. The programmer must carefully examine the problem and the desired results before deciding these very important questions.

Efficiency Considerations

As we shall see in this chapter, there are a great number of methods that can be used to sort a file. The programmer must be aware of several interrelated and often conflicting efficiency considerations to make an intelligent choice about which sorting method is most appropriate to a particular problem. Three of the most important of these considerations include the length of time that must be spent by the programmer in coding a particular sorting program, the amount of machine time necessary for running the program, and the amount of space necessary for the program.

If a file is small, sophisticated sorting techniques designed to minimize space and time requirements are usually worse or only marginally better in achieving efficiencies than simpler, generally less efficient methods. Similarly, if a particular sorting program is to be run only once and there is sufficient machine time and space in which to run it, it would be ludicrous for a programmer to spend days investigating the best methods of obtaining the last ounce of efficiency. In such cases the amount of time that must be spent by the programmer is properly the overriding consideration in determining which sorting method to use. However, a strong word of caution must be inserted. Programming time is never a valid excuse for using an incorrect program. A sort which is run only once may be able to afford the luxury of an inefficient technique, but it cannot afford an incorrect one. The presumably sorted data may be used in an application in which the assumption of ordered data is crucial.

However, a programmer must be able to recognize the fact that a particular sort is inefficient and must be able to justify its use in a particular situation. Too often, programmers take the easy way out and code an inefficient sort, which is then incorporated into a larger system in which the sort is a key component. The designers and planners of the system are then surprised at the inadequacy of their creation. To maximize his or her own efficiency, a programmer must be knowledgeable of a wide range of sorting techniques and be cognizant of the advantages and disadvantages of each, so that when the need for a sort arises he or she can supply the one which is most appropriate for the particular situation.

This brings us to the other two efficiency considerations: time and space. As in most computer applications, the programmer must often optimize one of these at the expense of the other. In considering the time necessary to sort a file of size *n* we do not concern ourselves with actual time units, as these will vary from one machine to another, from one program to another, and from one set of data to another. Rather, we are interested in the corresponding change in the amount of time required to sort a file induced by a change in the file size *n*. Let us see if we can make this concept more precise. We say that y is ***proportional*** to x if the relation between y and x is such that multiplying x by a constant multiplies y by that same constant. Thus if y is proportional to x, doubling x will double y, and multiplying x by 10 will multiply y by 10. Similarly, if y is proportional to x^2, doubling x will multiply y by 4 and multiplying x by 10 will multiply y by 100.

Often we do not measure the time efficiency of a sort by the number of time units required but rather by the number of critical operations performed. Examples of such critical operations are key comparisons (that is, the comparisons of the keys of two records in the file to determine which is greater), movement of records or pointers to records, or interchanges of two records. The critical operations chosen are those that take the most time. For example, a key comparison may be a complex operation, especially if the keys themselves are long or the ordering among keys is nontrivial. Thus a key comparison requires much more time than say, a simple increment of an index variable in a ***for*** loop. Also, the number of simple operations required is usually proportional to the number of key comparisons. For this reason, the number of key comparisons is a useful measure of a sort's time efficiency.

There are two ways to determine the time requirements of a sort, neither of which yields results that are applicable to all cases. One method is to go through a sometimes

n	$a = 0.01n^2$	$b = 10n$	$a + b$	$\frac{(a + b)}{n^2}$
10	1	100	101	1.01
50	25	500	525	0.21
100	100	1,000	1,100	0.11
500	2,500	5,000	7,500	0.03
1,000	10,000	10,000	20,000	0.02
5,000	250,000	50,000	300,000	0.01
10,000	1,000,000	100,000	1,100,000	0.01
50,000	25,000,000	500,000	25,500,000	0.01
100,000	100,000,000	1,000,000	101,000,000	0.01
500,000	2,500,000,000	5,000,000	2,505,000,000	0.01

Figure 6.1.3

intricate and involved mathematical analysis of various cases (for example, best case, worst case, average case). The result of this analysis is often a formula giving the average time (or number of operations) required for a particular sort as a function of the file size n. (Actually the time requirements of a sort depend on factors other than the file size; however, we concern ourselves here only with the dependence on the file size.) Suppose that such a mathematical analysis on a particular sorting program results in the conclusion that the program takes $0.01n^2 + 10n$ time units to execute. The first and fourth columns of Figure 6.1.3 show the time needed by the sort for various values of n. You will notice that for small values of n, the quantity $10n$ (third column of Figure 6.1.3) overwhelms the quantity $0.01n^2$ (second column). This is because the difference between n^2 and n is small for small values of n and is more than compensated for by the difference between 10 and 0.01. Thus, for small values of n, an increase in n by a factor of 2 (for example, from 50 to 100) increases the time needed for sorting by approximately that same factor of 2 (from 525 to 1100). Similarly, an increase in n by a factor of 5 (for example, from 10 to 50) increases the sorting time by approximately 5 (from 101 to 525).

However, as n becomes larger, the difference between n^2 and n increases so quickly that it eventually more than compensates for the difference between 10 and 0.01. Thus when n equals 1000 the two terms contribute equally to the amount of time needed by the program. As n becomes even larger, the term $0.01n^2$ overwhelms the term $10n$ and the contribution of the term $10n$ becomes almost insignificant. Thus, for large values of n, an increase in n by a factor of 2 (for example, from 50,000 to 100,000) results in an increase in sorting time of approximately 4 (from 25.5 million to 101 million) and an increase in n by a factor of 5 (for example, from 10,000 to 50,000) increases the sorting time by approximately a factor of 25 (from 1.1 million to 25.5 million). Indeed, as n becomes larger and larger, the sorting time becomes more closely proportional to n^2, as is clearly illustrated by the last column of Figure 6.1.3. Thus for large n the time required by the sort is almost proportional to n^2. Of course, for small values of n, the sort may exhibit drastically different behavior (as in Figure 6.1.3), a situation that must be taken into account in analyzing its efficiency.

O Notation

To capture the concept of one function becoming proportional to another as it grows, we introduce some terminology and a new notation. In the previous example, the function $0.01n^2 + 10n$ is said to be "on the order of" the function n^2 because, as n becomes large, it becomes more nearly proportional to n^2.

To be precise, given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is ***on the order of*** $g(n)$ or that $f(n)$ is $O(g(n))$ if there exist positive integers a and b such that $f(n) \leq a * g(n)$ for all $n \geq b$. For example, if $f(n) = n^2 + 100n$ and $g(n) = n^2$, $f(n)$ is $O(g(n))$, since $n^2 + 100n$ is less than or equal to $2n^2$ for all n greater than or equal to 100. In this case a equals 2 and b equals 100. This same $f(n)$ is also $O(n^3)$, since $n^2 + 100n$ is less than or equal to $2n^3$ for all n greater than or equal to 8. Given a function $f(n)$, there may be many functions $g(n)$ such that $f(n)$ is $O(g(n))$.

If $f(n)$ is $O(g(n))$, "eventually" (that is, for $n \geq b$) $f(n)$ becomes permanently smaller or equal to some multiple of $g(n)$. In a sense we are saying that $f(n)$ is bounded by $g(n)$ from above, or that $f(n)$ is a "smaller" function than $g(n)$. Another formal way of saying this is that $f(n)$ is ***asymptotically bounded*** by $g(n)$. Yet another interpretation is that $f(n)$ grows more slowly than $g(n)$, since, proportionately (that is, up to a factor of a), $g(n)$ eventually becomes larger.

It is easy to show that if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, $f(n)$ is $O(h(n))$. For example, $n^2 + 100n$ is $O(n^2)$, and n^2 is $O(n^3)$ (to see this, set a and b both equal to 1): consequently $n^2 + 100n$ is $O(n^3)$. This is called the ***transitive property***.

Note that if $f(n)$ is a constant function [that is, $f(n) = c$ for all n], $f(n)$ is $O(1)$, since, setting a to c and b to 1, we have that $c \leq c * 1$ for all $n \geq 1$. (In fact, the value of b or n is irrelevant, since a constant function's value is independent of n.)

It is also easy to show that the function $c * n$ is $O(n^k)$ for any constants c and k. To see this, simply note that $c * n$ is less than or equal to $c * n^k$ for any $n \geq 1$ (that is, set $a = c$ and $b = 1$). It is also obvious that n^k is $O(n^{k+j})$ for any $j \geq 0$ (use $a = 1$, $b = 1$). We can also show that if $f(n)$ and $g(n)$ are both $O(h(n))$, the new function $f(n) + g(n)$ is also $O(h(n))$. All these facts together can be used to show that if $f(n)$ is any polynomial whose leading power is k [that is, $f(n) = c_1 * n^k + c_2 * n^{k-1} + \cdots + c_k * n + c_{k+1}$], $f(n)$ is $O(n^k)$. Indeed, $f(n)$ is $O(n^{k+j})$ for any $j \geq 0$.

Although a function may be asymptotically bounded by many other functions [as for example, $10n^2 + 37n + 153$ is $O(n^2)$, $O(10n^2)$, $O(37n^2 + 10n)$ and $O(0.05n^3)$], we usually look for an asymptotic bound that is a single term with a leading coefficient of 1 and that is as "close a fit" as possible. Thus we would say that $10n^2 + 37n + 153$ is $O(n^2)$, although it is also asymptotically bounded by many other functions. Ideally, we would like to find a function $g(n)$ such that $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. If $f(n)$ is a constant or a polynomial, this can always be done by using its highest term with a coefficient of 1. For more complex functions, however, it is not always possible to find such a tight fit.

An important function in the study of algorithm efficiency is the logarithm function. Recall that $\log_m n$ is the value x such that m^x equals n. m is called the ***base*** of the logarithm. Consider the functions $log_m\ n$ and $log_k\ n$. Let xm be $log_m\ n$ and xk be $log_k\ n$. Then

$$m^{xm} = n \quad \text{and} \quad k^{xk} = n$$

so that

$$m^{xm} = k^{xk}$$

Taking the $\log_m$ of both sides,

$$xm = \log_m(k^{xk})$$

Now it can easily be shown than $\log_z (x^y)$ equals $y * \log_z x$ for any x, y, and z, so that the last equation can be rewritten as (recall that $xm = \log_m n$)

$$\log_m n = xk * \log_m k$$

or as (recall that $xk = \log_k n$)

$$\log_m n = (\log_m k) * \log_k n$$

Thus $\log_m n_m$ and $\log_k n$ are constant multiples of each other.

It is easy to show that if $f(n) = c * g(n)$, where c is a constant, $f(n)$ is $O(g(n))$ [indeed, we have already shown that this is true for the function $f(n) = n^k$]. Thus $\log_m n$ is $O(\log_k n)$ and $\log_k n$ is $O(\log_m n)$ for any m and k. Since each logarithm function is on the order of any other, we usually omit the base when speaking of functions of logarithmic order and say that all such functions are $O(\log n)$.

The following facts establish an order hierarchy of functions:

c is $O(1)$ for any constant c.
c is $O(\log n)$, but $\log n$ is not $O(1)$.
$c*\log_k n$ is $O(\log n)$ for any constants c, k.
$c*\log_k n$ is $O(n)$, but n is not $O(\log n)$.
$c * n^k$ is $O(n^k)$ for any constants c, k.
$c * n^k$ is $O(n^{k+j})$, but n^{k+j} is not $O(n^k)$.
$c * n*\log_k n$ is $O(n \log n)$ for any constants c, k.
$c * n*\log_k n$ is $O(n^2)$, but n^2 is not $O(n \log n)$.
$c * n^j*\log_k n$ is $O(n^j \log n)$ for any constants c, j, k.
$c * n^j*\log_k n$ is $O(n^{j+1})$, but n^{j+1} is not $O(n^j \log n)$.
$c * n^j*(\log_k n)^l$ is $O(n^j (\log n)^l)$ for any constants $c, j, k; l$.
$c * n^j*(\log_k n)^l$ is $O(n^{j+1})$ but n^{j+1} is not $O(n^j(\log n)^l)$.
$c * n^j*(\log_k n)^l$ is $O(n^j(\log n)^{l+1})$ but $n^j (\log n)^{l+1}$ is not $O(n^j (\log n)^l)$.
$c * n^k$ is $O(d^n)$, but d^n is not $O(n^k)$ for any constants c and k, and $d > 1$.

The hierarchy of functions established by these facts, with each function of lower order than the next, is c, $\log n$, $(\log n)^k$, n, $n(\log n)^k$, n^k, $n^k(\log n)^l$, n^{k+1}, and d^n.

Functions that are $O(n^k)$ for some k are said to be of ***polynomial*** order, whereas functions that are $O(d^n)$ for some $d > 1$ but not $O(n^k)$ for any k are said to be of ***exponential order***.

The distinction between polynomial-order functions and exponential-order functions is extremely important. Even a small exponential-order function, such as 2^n, grows far larger than any polynomial-order function, such as n^k, regardless of the size of k. As an illustration of the rapidity with which exponential-order functions grow, consider that 2^{10} equals 1024 but that 2^{100} (that is, 1024^{10}) is greater than the number formed by a 1 followed by 30 zeros. The smallest k for which 10^k exceeds 2^{10} is 4, but the smallest k for which 100^k exceeds 2^{100} is 16. As n becomes larger, larger values of k are needed for n^k to keep up with 2^n. For any single k, 2^n eventually becomes permanently larger than n^k.

Because of the incredible rate of growth of exponential-order functions, problems that require exponential-time algorithms for solution are considered to be ***intractable*** on current computing equipment; that is, such problems cannot be solved precisely except in the simplest cases.

Efficiency of Sorting

Using this concept of the order of a sort, we can compare various sorting techniques and classify them as being "good" or "bad" in general terms. One might hope to discover the "optimal" sort that is $O(n)$ regardless of the contents or order of the input. Unfortunately, however, it can be shown that no such generally useful sort exists. Most of the classical sorts we shall consider have time requirements that range from $O(n \log n)$ to $O(n^2)$. In the former, multiplying the file size by 100 will multiply the sorting time by less than 200; in the latter, multiplying the file size by 100 multiplies the sorting time by a factor of 10,000. Figure 6.1.4 shows the comparison of $n \log n$ with n^2 for a range of values of n. It can be seen from the figure that for large n, as n increases, n^2 increases at a much more rapid rate than $n \log n$. However, a sort should not be selected simply because it is $O(n \log n)$. The relation of the file size n and the other terms constituting the actual sorting time must be known. In particular, terms which play an insignificant role for large values of n may play a very dominant role for small values of n. All these issues must be considered before an intelligent sort selection can be made.

A second method of determining time requirements of a sorting technique is to actually run the program and measure its efficiency (either by measuring absolute time units or the number of operations performed). To use such results in measuring the efficiency of a sort the test must be run on "many" sample files. Even when such statistics

n	$n \log_{10} n$	n^2
1×10^1	1.0×10^1	1.0×10^2
5×10^1	8.5×10^1	2.5×10^3
1×10^2	2.0×10^2	1.0×10^4
5×10^2	1.3×10^3	2.5×10^5
1×10^3	3.0×10^3	1.0×10^6
5×10^3	1.8×10^4	2.5×10^7
1×10^4	4.0×10^4	1.0×10^8
5×10^4	2.3×10^5	2.5×10^9
1×10^5	5.0×10^5	1.0×10^{10}
5×10^5	2.8×10^6	2.5×10^{11}
1×10^6	6.0×10^6	1.0×10^{12}
5×10^6	3.3×10^7	2.5×10^{13}
1×10^7	7.0×10^7	1.0×10^{14}

Figure 6.1.4 Comparison of $n \log n$ and n^2 for various values of n.

have been gathered, the application of that sort to a specific file may not yield results that follow the general pattern. Peculiar attributes of the file in question may make the sorting speed deviate significantly. In the sorts of the subsequent sections we shall give an intuitive explanation of why a particular sort is classified as $O(n^2)$ or $O(n \log n)$; we leave mathematical analysis and sophisticated testing of empirical data as exercises for the ambitious reader.

In most cases the time needed by a sort depends on the original sequence of the data. For some sorts, input data which is almost in sorted order can be completely sorted in time $O(n)$, whereas input data that is in reverse order needs time that is $O(n^2)$. For other sorts the time required is $O(n \log n)$ regardless of the original order of the data. Thus, if we have some knowledge about the original sequence of the data we can make a more intelligent decision about which sorting method to select. On the other hand, if we have no such knowledge we may wish to select a sort based on the worst possible case or based on the "average" case. In any event, the only general comment that can be made about sorting techniques is that there is no "best" general sorting technique. The choice of a sort must, of necessity, depend on the specific circumstances.

Once a particular sorting technique has been selected, the programmer should then proceed to make the program as efficient as possible. In many programming applications it is often necessary to sacrifice efficiency for the sake of clarity. With sorting, the situation is usually the opposite. Once a sorting program has been written and tested, the programmer's chief goal is to improve its speed, even if it becomes less readable. The reason for this is that a sort may account for the major part of a program's efficiency, so that any improvement in sorting time significantly affects overall efficiency. Another reason is that a sort is often used quite frequently, so that a small improvement in its execution speed saves a great deal of computer time. It is usually a good idea to remove function calls, especially from inner loops, and replace them with the code of the function in line, since the call-return mechanism of a language can be prohibitively expensive in terms of time. Also, a function call may involve the assignment of storage to local variables, an activity that sometimes requires a call to the operating system. In many of the programs we do not do this so as not to obfuscate the intent of the program with huge blocks of code.

Space constraints are usually less important than time considerations. One reason for this is that, for most sorting programs, the amount of space needed is closer to $O(n)$ than to $O(n^2)$. A second reason is that if more space is required it can almost always be found in auxiliary storage. An ideal sort is an ***in-place sort*** whose additional space requirements are $O(1)$. That is, an in-place sort manipulates the elements to be sorted within the array or list space that contained the original unsorted input. Additional space that is required is in the form of a constant number of locations (such as declared individual program variables) regardless of the size of the set to be sorted.

Usually, the expected relationship between time and space holds for sorting algorithms: those programs that require less time usually require more space, and vice versa. However, there are clever algorithms that utilize both minimum time and minimum space; that is, they are $O(n \log n)$ in-place sorts. These may, however, require more programmer time to develop and verify. They also have higher constants of proportionality than many sorts that do use more space or that have higher time-orders and so require more time to sort small sets.

In the remaining sections we investigate some of the more popular sorting techniques and indicate some of their advantages and disadvantages.

EXERCISES

6.1.1. Choose any sorting technique with which you are familiar.

(a) Write a program for the sort.

(b) Is the sort stable?

(c) Determine the time requirements of the sort as a function of the file size, both mathematically and empirically.

(d) What is the order of the sort?

(e) At what file size does the most dominant term begin to overshadow the others?

6.1.2. Show that the function $(\log_m n)^k$ is $O(n)$ for all m and k but that n is not $O((\log n)^k)$ for any k.

6.1.3. Suppose that a time requirement is given by the formula $a * n^2 + b * n * \log_2 n$, where a and b are constants. Answer the following questions by both proving your results mathematically and writing a program to validate the results empirically.

(a) For what values of n (expressed in terms of a and b) does the first term dominate the second?

(b) For what value of n (expressed in terms of a and b) are the two terms equal?

(c) For what values of n (expressed in terms of a and b) does the second term dominate the first?

6.1.4. Show that any process that sorts a file can be extended to find all duplicates in the file.

6.1.5. A ***sort decision tree*** is a binary tree that represents a sorting method based on comparisons. Figure 6.1.5 illustrates such a decision tree for a file of three elements. Each nonleaf of such a tree represents a comparison between two elements. Each leaf represents a completely sorted file. A left branch from a nonleaf indicates that the first key was smaller than the second; a right branch indicates that it was larger. (We assume that all the elements in the file have distinct keys.) For example, the tree of Figure 6.1.5 represents a sort on three elements $x[0]$, $x[1]$, $x[2]$ that proceeds as follows:
Compare $x[0]$ with $x[1]$. If $x[0] < x[1]$, compare $x[1]$ with $x[2]$, and if $x[1] < x[2]$, the sorted order of the file is $x[0]$, $x[1]$, $x[2]$; otherwise if $x[0] < x[2]$, the sorted order is

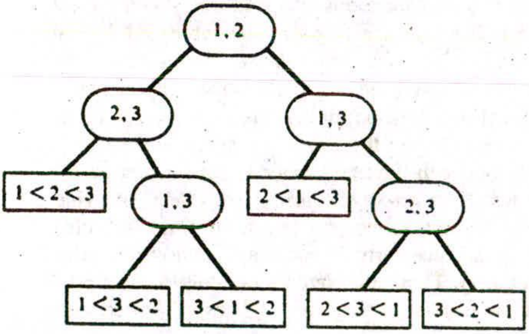


Figure 6.1.5 Decision tree for a file of three elements.

$x[0]$, $x[2]$, $x[1]$, and if $x[0] > x[2]$, the sorted order is $x[2]$, $x[0]$, $x[1]$. If $x[0] > x[1]$, proceed in a similar fashion down the right subtree.

- **(a)** Show that a sort decision tree that never makes a redundant comparison (that is, never compares $x[i]$ and $x[j]$ if the relationship between $x[i]$ and $x[j]$ is known) has $n!$ leaves.
- **(b)** Show that the depth of such a decision tree is at least $\log_2 (n!)$.
- **(c)** Show that $n! \geq (n / 2)^{n\ 2}$, so that the depth of such a tree is $O(n \log n)$.
- **(d)** Explain why this proves that any sorting method that uses comparisons on a file of size n must make at least $O(n \log n)$ comparisons.

6.1.6. Given a sort decision tree for a file as in the previous exercise, show that if the file contains some equal elements, the result of applying the tree to the file (where either a left or right branch is taken whenever two elements are equal) is a sorted file.

6.1.7. Extend the concept of the binary decision tree of the previous exercises to a ternary tree that includes the possibility of equality. It is desired to determine which elements of the file are equal, in addition to the order of the distinct elements of the file. How many comparisons are necessary?

6.1.8. Show that if k is the smallest integer greater than or equal to $n + \log_2 n - 2$, k comparisons are necessary and sufficient to find the 'argest and second largest elements of a set of n distinct elements.

6.1.9. How many comparisons are necessary to find the largest and smallest of a set of n distinct elements?

6.1.10. Show that the function $f(n)$ defined by

$$f(1) = 1$$
$$f(n) = f(n - 1) + 1\ n \text{ for } n > 1$$

is $O(\log n)$.

6.2 EXCHANGE SORTS

Bubble Sort

The first sort we present is probably the most widely known among beginning students of programming: the ***bubble sort***. One of the characteristics of this sort is it is easy to understand and program. Yet, of all the sorts we shall consider, it is pr the least efficient.

In each of the subsequent examples, x is an array of integers of which the first n are to be sorted so that $x[i] \leq x[j]$ for $0 \leq i < j < n$. It is straightforward to extend this simple format to one which is used in sorting n records, each with a subfield key k.

The basic idea underlying the bubble sort is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor ($x[i]$ with $x[i + 1]$) and interchanging the two elements if they are not in proper order. Consider the following file:

25 57 48 37 12 92 86 33

The following comparisons are made on the first pass:

$x[0]$	with	$x[1]$	(25 with 57)	No interchange
$x[1]$	with	$x[2]$	(57 with 48)	Interchange
$x[2]$	with	$x[3]$	(57 with 37)	Interchange
$x[3]$	with	$x[4]$	(57 with 12)	Interchange
$x[4]$	with	$x[5]$	(57 with 92)	No interchange
$x[5]$	with	$x[6]$	(92 with 86)	Interchange
$x[6]$	with	$x[7]$	(92 with 33)	Interchange

Thus, after the first pass, the file is in the order

25 48 37 12 57 86 33 92

Notice that after this first pass, the largest element (in this case 92) is in its proper position within the array. In general, $x[n - i]$ will be in its proper position after iteration i. The method is called the bubble sort because each number slowly "bubbles" up to its proper position. After the second pass the file is

25 37 12 48 57 33 86 92

Notice that 86 has now found its way to the second highest position. Since each iteration places a new element into its proper position, a file of n elements requires no more than $n - 1$ iterations.

The complete set of iterations is the following:

Iteration 0 (original file)	25	57	48	37	12	92	86	33
Iteration 1	25	48	37	12	57	86	33	92
Iteration 2	25	37	12	48	57	33	86	92
Iteration 3	25	12	37	48	33	57	86	92
Iteration 4	12	25	37	33	48	57	86	92
Iteration 5	12	25	33	37	48	57	86	92
Iteration 6	12	25	33	37	48	57	86	92
Iteration 7	12	25	33	37	48	57	86	92

On the basis of the foregoing discussion we could proceed to code the bubble sort. However, there are some obvious improvements to the foregoing method. First, since all the elements in positions greater than or equal to $n - i$ are already in proper position after iteration i, they need not be considered in succeeding iterations. Thus on the first pass $n - 1$ comparisons are made, on the second pass $n - 2$ comparisons, and on the $(n - 1)$th pass only one comparison is made (between $x[0]$ and $x[1]$). Therefore the process speeds up as it proceeds through successive passes.

We have shown that $n - 1$ passes are sufficient to sort a file of size n. However, in the preceding sample file of eight elements, the file was sorted after five iterations, making the last two iterations unnecessary. To eliminate unnecessary passes we must be able to detect the fact that the file is already sorted. But this is a simple task, since in a sorted file no interchanges are made on any pass. By keeping a record of whether or not any interchanges are made in a given pass it can be determined whether any further passes are necessary. Under this method, if the file can be sorted in fewer than $n - 1$ passes, the final pass makes no interchanges.

Using these improvements, we present a routine *bubble* that accepts two variables x and n. x is an array of numbers, and n is an integer representing the number of elements to be sorted. (n may be less than the number of elements in x.)

```
void bubble(int x[], int n)
{
   int hold, j, pass;
   int switched = TRUE;

   for (pass = 0; pass < n-1 && switched == TRUE; pass++) {
      /*     outer loop controls the number of passes         */
      switched = FALSE;          /* initially no interchanges have */
                                 /*     been made on this pass.    */
      for (j = 0; j < n-pass-1; j++)
         /*   inner loop governs each individual pass          */
         if (x[j] > x[j+1]) {
            /*           elements out of order                 */
            /*       an interchange is necessary               */
            switched = TRUE;
            hold = x[j];
            x[j] = x[j+1];
            x[j+1] = hold;
         } /* end if */
   } /* end for */
} /* end bubble */
```

What can be said about the efficiency of the bubble sort? In the case of a sort that does not include the two improvements outlined previously, the analysis is simple. There are $n - 1$ passes and $n - 1$ comparisons on each pass. Thus the total number of comparisons is $(n - 1) * (n - 1) = n^2 - 2n + 1$, which is $O(n^2)$. Of course, the number of interchanges depends on the original order of the file. However, the number of interchanges cannot be greater than the number of comparisons. It is likely that it is the number of interchanges rather than the number of comparisons that takes up the most time in the program's execution.

Let us see how the improvements that we introduced affect the speed of the bubble sort. The number of comparisons on iteration i is $n - i$. Thus, if there are k iterations the total number of comparisons is $(n - 1) + (n - 2) + (n - 3) + \cdots + (n - k)$, which equals $(2kn - k^2 - k)/2$. It can be shown that the average number of iterations, k, is $O(n)$, so that the entire formula is still $O(n^2)$, although the constant multiplier is smaller than before. However, there is additional overhead involved in testing and initializing the variable *switched* (once per pass) and setting it to *TRUE* (once for every interchange).

The only redeeming features of the bubble sort are that it requires little additional space (one additional record to hold the temporary value for interchanging and several simple integer variables) and that it is $O(n)$ in the case that the file is completely sorted (or almost completely sorted). This follows from the observation that only one pass of $n - 1$ comparisons (and no interchanges) is necessary to establish that a sorted file is sorted.

There are some other ways to improve the bubble sort. One of these is to observe that the number of passes necessary to sort the file is the largest distance by which a number must move "down" in the array. In our example, for instance, 33, which starts at position 7 in the array, ultimately finds its way to position 2 after five iterations. The bubble sort can be speeded up by having successive passes go in opposite directions so that the small elements move quickly to the front of the file in the same way that the large ones move to the rear. This reduces the required number of passes. This version is left as an exercise.

Quicksort

The next sort we consider is the ***partition exchange sort*** (or ***quicksort***). Let x be an array, and n the number of elements in the array to be sorted. Choose an element a from a specific position within the array (for example, a can be chosen as the first element so that $a = x[0]$). Suppose that the elements of x are partitioned so that a is placed into position j and the following conditions hold:

1. Each of the elements in positions 0 through $j - 1$ is less than or equal to a.
2. Each of the elements in positions $j + 1$ through $n - 1$ is greater than or equal to a.

Notice that if these two conditions hold for a particular a and j, a is the jth smallest element of x, so that a remains in position j when the array is completely sorted. (You are asked to prove this fact as an exercise.) If the foregoing process is repeated with the subarrays $x[0]$ through $x[j - 1]$ and $x[j + 1]$ through $x[n - 1]$ and any subarrays created by the process in successive iterations, the final result is a sorted file.

Let us illustrate the quicksort with an example: If an initial array is given as

25 57 48 37 12 92 86 33

and the first element (25) is placed in its proper position, the resulting array is

12 25 57 48 37 92 86 33

At this point, 25 is in its proper position in the array ($x[1]$), each element below that position (12) is less than or equal to 25, and each element above that position (57, 48, 37, 92, 86, and 33) is greater than or equal to 25. Since 25 is in its final position the original problem has been decomposed into the problem of sorting the two subarrays

(12) and (57 48 37 92 86 33)

Nothing need be done to sort the first of these subarrays; a file of one element is already sorted. To sort the second subarray the process is repeated and the subarray is further subdivided. The entire array may now be viewed as

12 25 (57 48 37 92 86 33)

where parentheses enclose the subarrays that are yet to be sorted. Repeating the process on the subarray $x[2]$ through $x[7]$ yields

12 25 (48 37 33) 57 (92 86)

and further repetitions yield

```
12  25  (37   33)  48  57  (92  86)
12  25  (33)  37   48  57  (92  86)
12  25   33   37   48  57  (92  86)
12  25   33   37   48  57  (86)  92
12  25   33   37   48  57   86   92
```

Note that the final array is sorted.

By this time you should have noticed that the quicksort may be defined most conveniently as a recursive procedure. We may outline an algorithm *quick(lb,ub)* to sort all elements in an array *x* between positions *lb* and *ub* (*lb* is the lower bound, *ub* the upper bound) as follows:

```
if (lb >= ub)
    return;                /*        array is sorted          */

partition(x,lb,ub,j);      /* partition the elements of the   */
                           /* subarray such that one of the   */
                           /*  elements (possibly x[lb]) is   */
                           /*  now at x[j] (j is an output    */
                           /*        parameter) and:          */
                           /* 1. x[i] <= x[j] for lb <= i < j */
                           /* 2. x[i] >= x[j] for j < i <= ub */
                           /*     x[j] is now at its final    */
                           /*            position             */

quick(x,lb,j - 1);         /* recursively sort the subarray   */
                           /* between positions lb and j - 1  */

quick(x,j + 1,ub);         /* recursively sort the subarray   */
                           /* between positions j + 1 and ub  */
```

There are now two problems. We must produce a mechanism to implement *partition* and produce a method to implement the entire process nonrecursively.

The object of *partition* is to allow a specific element to find its proper position with respect to the others in the subarray. Note that the manner in which this partition is performed is irrelevant to the sorting method. All that is required by the sort is that the elements be partitioned properly. In the preceding example, the elements in each of the two subfiles remain in the same relative order as they appear in the original file. However, such a partition method is relatively inefficient to implement.

One way to effect a partition efficiently is the following: Let $a = x[lb]$ be the element whose final position is sought. (There is no appreciable efficiency gained by selecting the first element of the subarray as the one which is inserted into its proper position; it merely makes some of the programs easier to code.) Two pointers, *up* and *down*, are initialized to the upper and lower bounds of the subarray, respectively. At any point during execution, each element in a position above *up* is greater than or equal to *a*, and each element in a position below *down* is less than or equal to *a*. The two pointers *up* and *down* are moved towards each other in the following fashion.

Step 1: Repeatedly increase the pointer *down* by one position until *x*[*down*] > *a*.
Step 2: Repeatedly decrease the pointer *up* by one position until *x*[*up*] <= *a*.
Step 3: If *up* > *down*, interchange *x*[*down*] with *x*[*up*].

The process is repeated until the condition in step 3 fails (*up* <= *down*), at which point *x*[*up*] is interchanged with *x*[*lb*] (which equals *a*), whose final position was sought, and *j* is set to *up*.

We illustrate this process on the sample file, showing the positions of *up* and *down* as they are adjusted. The direction of the scan is indicated by an arrow at the pointer being moved. Three asterisks on a line indicates that an interchange is being made.

```
a = x[lb] = 25

down-->                                                   up
 25      57      48      37      12      92      86      33

        down                                              up
 25      57      48      37      12      92      86      33

        down                                          <--up
 25      57      48      37      12      92      86      33

        down                                  <--up
 25      57      48      37      12      92      86      33

        down                          <--up
 25      57      48      37      12      92      86      33

        down                     up
 25      57      48      37      12      92      86      33

        down                     up
 25      12      48      37      57      92      86      33      ***

        down-->                  up
 25      12      48      37      57      92      86      33

                down             up
 25      12      48      37      57      92      86      33

                down         <--up
 25      12      48      37      57      92      86      33

                down <--up
 25      12      48      37      57      92      86      33
```

```
             <--up,down
25      12      48      37      57      92      86      33

        up      down
25      12      48      37      57      92      86      33

        up      down
12      25      48      37      57      92      86      33     ***
```

At this point 25 is in its proper position (position 1), and every element to its left is less than or equal to 25, and every element to its right is greater than or equal to 25. We could now proceed to sort the two subarrays (12) and (48 37 57 92 86 33) by applying the same method.

This particular algorithm can be implemented by the following procedure.

```
void partition (int x[], int lb, int ub, int *pj)
{
   int a, down, temp, up;

   a = x[lb];              /*  a is the element whose final */
                           /*       position is sought      */
   up = ub;
   down = lb;
   while (down < up)  {
      while (x[down] <= a && down < ub)
         down++;           /*       move up the array       */
      while (x[up] > a)
         up--;             /*      move down the array      */
      if (down < up) {
         /*                  interchange x[down] and x[up] */
         temp = x[down];
         x[down] = x[up];
         x[up] = temp;
      }  /* end if */
   } /* end while */
   x[lb] = x[up];
   x[up] = a;
   *pj = up;
} /* end partition */
```

Note that if k equals $ub - lb + 1$, so that we are rearranging a subarray of size k, the routine uses k key comparisons (of $x[down]$ with a and $x[up]$ with a) to perform the partition.

The routine can be made slightly more efficient by eliminating some of the redundant tests. You are asked to do this as an exercise.

Although the recursive quicksort algorithm is relatively clear in terms of what it accomplishes and how, it is desirable to avoid the overhead of routine calls in programs such as sorts in which execution efficiency is a significant consideration. The recursive calls to *quick* can easily be eliminated by using a stack as in Section 3.4. Once *partition* has been executed, the current parameters to *quick* are no longer needed, except in computing the arguments to the two subsequent recursive calls. Thus instead of stacking the current parameters upon each recursive call, we can compute and stack the new parameters for each of the two recursive calls. Under this approach, the stack at any point contains the lower and upper bounds of all subarrays that must yet be sorted. Furthermore, since the second recursive call immediately precedes the return to the calling program (as in the Towers of Hanoi problem), it may be eliminated entirely and replaced with a branch. Finally, since the order in which the two recursive calls are made does not affect the correctness of the algorithm, we elect in each case to stack the larger subarray and process the smaller subarray immediately. As we explain shortly, this technique keeps the size of the stack to a minimum.

We may now code a function to implement the quicksort. As in the case of *bubble*, the parameters are the array *x* and the number of elements of *x* that we wish to sort, *n*. The routine *push* pushes *lb* and *ub* onto the stack, *popsub* pops them from the stack, and *empty* determines if the stack is empty.

```
#define MAXSTACK ...                    /* maximum stack size */
void quicksort(intx[], int n)
{
   int i, j;
   struct bndtype {
      int lb;
      int ub;
   } newbnds;
   /* stack is used by the pop, push and empty functions */
   struct {
      int top;
      struct bndtype bounds[MAXSTACK];
   } stack;

   stack.top = -1;
   newbnds.lb = 0
   newbnds.ub = n-1;
   push(&stack, &newbnds);
   /* repeat as long as there are any */
   /* unsorted subarrays on the stack */
   while (!empty(&stack)) {
      popsub(&stack, &newbnds);
      while (newbnds.ub > newbnds.lb)  {
         /* process next subarray */
         partition(x, newbnds.lb, newbnds.ub, &j);
         /* stack the larger subarray */
         if (j-newbnds.lb > newbnds.ub-j) {
            /* stack lower subarray */
            i = newbnds.ub;
            newbnds.ub = j-1;
```

```
                push(&stack, &newbnds);
                /* process upper subarray */
                newbnds.lb = j+1;
                newbnds.ub = i;
            }
            else {
                /*  stack upper subarray  */
                i = newbnds.lb;
                newbnds.lb = j+1;
                push(&stack, &newbnds);
                /* process lower subarray */
                newbnds.lb = i;
                newbnds.ub = j-1;
            } /* end if */
        } /* end while */
    } /* end while */
} /* end quicksort */
```

The routines *partition*, *empty*, *popsub*, and *push* should be inserted in line for maximum efficiency. Trace the action of *quicksort* on the sample file.

Note that we have chosen to use *x*[*lb*] as the element around which to partition each subfile because of programming convenience in the proc·dure *partition*, but any other element could have been chosen as well. The element around which a file is partitioned is called a *pivot*. It is not even necessary that the pivot be an element of the subfile; *partition* can be written with the header

```
    partition(lb, ub, x, j, pivot)
```

to partition *x*[*lb*] through *x*[*ub*] so that all elements between *x*[*lb*] and *x*[*j* − 1] are less than *pivot* and all elements between *x*[*j*] and *x*[*ub*] are greater than or equal to *pivot*. In that case the element *x*[*j*] is itself included in the second subfile (since it is not necessarily in its proper position), so that the second recursive call to *quick* is *quick*(*x*, *j*, *ub*) rather than *quick*(*x*, *j* + 1, *ub*).

Several choices for the pivot value have been found to improve the efficiency of quicksort by guaranteeing more nearly balanced subfiles. The first technique uses the median of the first, last, and middle elements of the subfile to be sorted (that is, the median of *x*[*lb*], *x*[*ub*], and *x*[(*lb* + *ub*)/2]) as the pivot value. This median-of-three value is closer to the median of the subfile being partitioned than *x*[*lb*], so that the two partitions of the subfile are more nearly equal in size. In this method the pivot value is an element of the file, so that *quick* (*x*, *j* + 1, *ub*) can be used as the second recursive call.

A second method, called ***meansort***, utilizes *x*[*lb*] or the median-of-three as pivot when partitioning the original file but adds code in *partition* to compute the means (averages) of the two subfiles being created. In subsequent partitions the mean of each subfile, calculated when the subfile was created, is used as a pivot value. Again, this mean is closer to the median of the subfile than *x*[*lb*] and results in more nearly balanced files. The mean is not necessarily an element of the file, so that *quick* (x, j, ub) must be used as the second recursive call. The code to find the mean does not require any additional key comparisons but does add some extra overhead.

Another technique, called ***Bsort***, uses the middle element of a subfile as the pivot. During partition, whenever the pointer *up* is deceased, $x[up]$ is interchanged with $x[up + 1]$ if $x[up] > x[up + 1]$. Whenever the pointer *down* is increased, $x[down]$ is interchanged with $x[down - 1]$ if $x[down] < x[down - 1]$. Whenever $x[up]$ and $x[down]$ are interchanged $x[up]$ is interchanged with $x[up + 1]$ if $x[up] > x[up + 1]$, and $x[down]$ is interchanged with $x[down - 1]$ if $x[down] < x[down - 1]$. This guarantees that $x[up]$ is always the smallest element in the right subfile (from $x[up]$ to $x[ub]$) and that $x[down]$ is always the largest element in the left subfile (from $x[lb]$ to $x[down]$).

This allows two optimizations: If no interchanges between $x[up]$ and $x[up + 1]$ were required during the partition, the right subfile is known to be sorted and need not be stacked, and if no interchanges between $x[down]$ and $x[down - 1]$ were required, the left subfile is known to be sorted and need not be stacked. This is similar to the technique of keeping a flag in bubblesort that detects that no interchanges have taken place during an entire pass so that no additional passes are necessary. Second, a subfile of size 2 is known to be sorted and need not be stacked. A subfile of size 3 can be directly sorted with just a single comparison and possible interchange (between the first two elements in a left subfile and between the last two in a right subfile). Both optimizations in Bsort reduce the number of subfiles that must be processed.

Efficiency of Quicksort

How efficient is the quicksort? Assume that the file size n is a power of 2, say $n = 2^m$, so that $m = \log_2 n$. Assume also that the proper position for the pivot always turns out to be the exact middle of the subarray. In that case there will be approximately n comparisons (actually $n - 1$) on the first pass, after which the file is split into two subfiles each of size $n/2$, approximately. For each of these two files there are approximately $n/2$ comparisons, and a total of four files each of size $n/4$ are formed. Each of these files requires $n/4$ comparisons yielding a total of $n/8$ subfiles. After halving the subfiles m times, there are n files of size 1. Thus the total number of comparisons for the entire sort is approximately

$$n + 2 * (n/2) + 4 * (n/4) + 8 * (n/8) + \cdots + n * (n/n)$$

or

$$n + n + n + n + \cdots + n (m \text{ terms})$$

comparisons. There are m terms because the file is subdivided m times. Thus the total number of comparisons is $O(n * m)$ or $O(n \log n)$ (recall that $m = \log_2 n$). Thus if the foregoing properties describe the file, the quicksort is $O(n \log n)$, which is relatively efficient.

For the unmodified quicksort in which $x[lb]$ is used as the pivot value, this analysis assumes that the original array and all the resulting subarrays are unsorted, so that the pivot value $x[lb]$ always finds its proper position at the middle of the subarray.

Suppose that the preceding conditions do not hold and the original array is sorted (or almost sorted). If, for example, $x[lb]$ is in its correct position, the original file is split into subfiles of sizes 0 and $n - 1$. if this process continues, a total of $n - 1$ subfiles are sorted, the first of size n, the second of size $n - 1$, the third of size $n - 2$, and so on. Assuming k comparisons to rearrange a file of size k, the total number of comparisons to sort the entire file is

$$n + (n - 1) + (n - 2) + \cdots + 2$$

which is $O(n^2)$. Similarly, if the original file is sorted in descending order the final position of $x[lb]$ is ub and the file is again split into two subfiles that are heavily unbalanced (sizes $n - 1$ and 0). Thus the unmodified quicksort has the seemingly absurd property that it works best for files that are "completely unsorted" and worst for files that are completely sorted. The situation is precisely the opposite for the bubble sort, which works best for sorted files and worst for unsorted files.

It is possible to speed up quicksort for sorted files by choosing a ***random*** element of each subfile as the pivot value. If a file is known to be nearly sorted, this might be a good strategy (although, in that case choosing the middle element as a pivot would be even better). However, if nothing is known about the file, such a strategy does not improve the worst case behavior, since it is possible (although improbable) that the random element chosen each time might consistently be the smallest element of each subfile. As a practical matter, sorted files are more common than a good random number generator happening to choose the smallest element repeatedly.

The analysis for the case in which the file size is not an integral power of 2 is similar but slightly more complex; the results, however, remain the same. It can be shown, however, that on the average (over all files of size n), the quicksort makes approximately $1.386n \log_2 n$ comparisons even in its unmodified version. In practical situations, quicksort is often the fastest available because of its low overhead and its average $O(n \log n)$ behavior.

If the median-of-three technique is used, quicksort can be $O(n \log n)$ even if the file is sorted (assuming that *partition* leaves the subfiles sorted). However, there are pathological files in which the first, last, and middle elements of each subfile are always the three smallest or largest elements. In such cases, quicksort remains $O(n^2)$. Fortunately, these are rare.

Meansort is $O(n \log n)$ as long as the elements of the file are uniformly distributed between the largest and smallest. Again, some rare distributions may make it $O(n^2)$, but this is less likely than the worst case of the other methods. For random files, meansort does not offer any significant reductions in comparisons or interchanges over standard quicksort. Its significant overhead for computing the mean requires far more CPU time than standard quicksort. For a file known to be almost sorted, meansort does provide significant reduction in comparisons and interchanges. However, the mean-finding overhead makes it slower than quicksort unless the file is very close to being completely sorted.

Bsort requires far less time than quicksort or meansort on sorted or nearly sorted input, although it does require more comparisons and interchanges than meansort for nearly sorted input (but meansort has significant overhead in finding the mean). It requires fewer comparisons but more interchanges than meansort and more of both than quicksort for randomly sorted input. However, its CPU requirements are far lower than meansort's, although somewhat greater than quicksort for random input.

Thus Bsort can be recommended if the input is known to be nearly sorted or if we are willing to forgo moderate increases in average sorting time to avoid very large increases in worst-case sorting time. Meansort can be recommended only for input known to be very nearly sorted and standard quicksort for input likely to be random or if average sorting time must be as fast as possible. In Section 6.5, we present a technique that is faster than either Bsort or meansort on nearly sorted files.

The space requirements for the quicksort depend on the number of nested recursive calls or on the size of the stack. Clearly, the stack can never grow larger than the number of elements in the original file. How much smaller than *n* the stack grows depends on the number of subfiles generated and on their sizes. The size of the stack is somewhat contained by always stacking the larger of the two subarrays and applying the routine to the smaller of the two. This guarantees that all smaller subarrays are subdivided before larger subarrays, giving the net effect of having fewer elements on the stack at any given time. The reason for this is that a smaller subarray will be divided fewer times than a larger subarray. Of course, the larger subarray will ultimately be processed and subdivided, but this will occur after the smaller subarrays have already been sorted and therefore removed from the stack.

Another advantage of quicksort is locality of reference. That is, over a short period of time all array accesses are to one or two relatively small portions of the array (a subfile or portion thereof). This insures efficiency in the virtual memory environment, where pages of data are constantly being swapped back and forth between external and internal storage. Locality of reference results in fewer page swaps being required for a particular program. A simulation study has shown that in such an environment, quicksort uses less space-time resources than any other sort considered.

EXERCISES

6.2.1. Prove that the number of passes necessary in the bubble sort of the text before the file is in sorted order (not including the last pass, which detects the fact that the file is sorted) equals the largest distance by which an element must move from a larger index to a smaller index.

6.2.2. Rewrite the routine *bubble* so that successive passes go in opposite directions.

6.2.3. Prove that, in the sort of the previous exercise, if two elements are not interchanged during two consecutive passes in opposite directions, they are in their final position.

6.2.4. A sort by ***counting*** is performed as follows. Declare an array *count* and set *count*[*i*] to the number of elements that are less than *x*[*i*]. Then place *x*[*i*] in position *count*[*i*] of an output array. (However, beware of the possibility of equal elements.) Write a routine to sort an array *x* of size *n* using this method.

6.2.5. Assume that a file contains integers between *a* and *b*, with many numbers repeated several times. A ***distribution sort*** proceeds as follows. Declare an array *number* of size $b - a + 1$, and set *number*[$i - a$] to the number of times that integer *i* appears in the file, and then reset the values in the file accordingly. Write a routine to sort an array *x* of size *n* containing integers between *a* and *b* by this method.

6.2.6. The ***odd-even transposition sort*** proceeds as follows. Pass through the file several times. On the first pass, compare $x[i]$ with $x[i + 1]$ for all odd i. On the second pass, compare $x[i]$ with $x[i + 1]$ for all even i. Each time that $x[i] > x[i + 1]$, interchange the two. Conti::ue alternating in this fashion until the file is sorted.

- **(a)** What is the condition for the termination of the sort?
- **(b)** Write a C routine to implement the sort.
- **(c)** On the average what is the efficiency of this sort?

6.2.7. Rewrite the program for the quicksort by starting with the recursive algorithm and applying the methods of Chapter 3 to produce a nonrecursive version.

6.2.8. Modify the quicksort program of the text so that if a subarray is small, the bubble sort is used. Determine, by actual computer runs, how small the subarray should be so that this mixed strategy will be more efficient that an ordinary quicksort.

6.2.9. Modify *partition* so that the middle value of $x[lb]$, $x[ub]$, and $x[ind]$ (where $ind = (ub + lb)/2$) is used to partition the array. In what cases is the quicksort using this method more efficient than the version of the text? In what cases is it less efficient?

6.2.10. Implement the meansort technique. *partition* should use the mean of the subfile being partitioned, computed when the subfile was created, as the pivot value and should compute the mean of each of the two subfiles that it creates. When the upper and lower bounds of a subfile are stacked, its mean should be stacked as well.

6.2.11. Implement the Bsort technique. The middle element of each file should be used as the pivot, the last element of the left subfile being created should be maintained as the largest in the left subfile, and the first element of the right subfile should be maintained as the smallest in the right subfile. Two bits should be used to keep track of whether the two subfiles are sorted at the end of the partition. A sorted subfile need not be processed further. If a subfile has three or fewer elements, sort it directly by a single interchange, at most.

6.2.12. **(a)** Rewrite the routines for the bubble sort and the quicksort as presented in the text and the sorts of the exercises so that a record is kept of the actual number of comparisons and the actual number of interchanges made.

- **(b)** Write a random-number generator (or use an existing one if your installation has one) that generates integers between 0 and 999.
- **(c)** Using the generator of part (b), generate several files of size 10, size 100, and size 1000. Apply the sorting routines of part (a) to measure the time requirements for each of the sorts on each of the files.
- **(d)** Measure the results of part (c) against the theoretical values presented in this section. Do they agree? If not, explain. In particular, rearrange the files so that they are completely sorted and in reverse order and see how the sorts behave with these inputs.

6.3 SELECTION AND TREE SORTING

A ***selection sort*** is one in which successive elements are selected in order and placed into their proper sorted positions. The elements of the input may have to be preprocessed to make the ordered selection possible. Any selection sort can be conceptualized as the following general algorithm that uses a descending priority queue (recall that *pqinsert* inserts into a priority queue and *pqmaxdelete* retrieves the largest element of a priority queue).

```
set dpq to the empty descending priority queue;
/* preprocess the elements of the input array */
/* by inserting them into the priority queue  */
for (i = 0; i < n; i++)
   pqinsert(dpq, x[i]);
/* select each successive element in order    */
for (i = n-1; i >= 0; i--)
   x[i] = pqmaxdelete(dpq);
```

This algorithm is called the ***general selection sort.***

We now examine several different selection sorts. Two features distinguish a specific selection sort. One feature is the data structure used to implement the priority queue. The second feature is the method used to implement the general algorithm. A particular data structure may allow significant optimization of the general selection sort algorithm.

Note also that the general algorithm can be modified to use an ascending priority queue *apq* rather than *dpq*. The second loop that implements the selection phase would be modified to

```
for (i = 0; i < n; i++)
   x[i] = pqmindelete(apq);
```

Straight Selection Sort

The ***straight selection sort***, or ***push-down sort***, implements the descending priority queue as an unordered array. The input array *x* is used to hold the priority queue, thus eliminating the need for additional space. The straight selection sort is, therefore, an in-place sort. Moreover, because the input array *x* is itself the unordered array that will represent the descending priority, the input is already in appropriate format and the preprocessing phase is unnecessary.

Therefore the straight selection sort consists entirely of a selection phase in which the largest of the remaining elements, *large*, is repeatedly placed in its proper position, *i*, at the end of the array. To do so, *large* is interchanged with the element *x*[*i*]. The initial *n*-element priority queue is reduced by one element after each selection. After $n - 1$ selections the entire array is sorted. Thus the selection process need be done only from $n - 1$ down to 1 rather than down to 0. The following C function implements straight selection:

```
void selectsort(int x[], int n)
{
   int i, indx, j, large;

   for (i = n-1; i > 0; i--) {
      /* place the largest number of x[0] through */
      /* x[i] into large and its index into indx  */
      large = x[0];
      indx = 0;
```

```
        for (j = 1; j <= i; j++)
           if (x[j] > large) {
              large = x[j];
              indx = j;
           } /* end for ... if */
        x[indx] = x[i];
        x[i] = large;
     } /* end for */
  } /* end selectsort */
```

Analysis of the straight selection sort is straightforward. The first pass makes $n - 1$ comparisons, the second pass makes $n - 2$, and so on. Therefore, there is a total of

$$(n - 1) + (n - 2) + (n - 3) + \cdots + 1 = n * (n - 1)/2$$

comparisons, which is $O(n^2)$. The number of interchanges is always $n - 1$ (unless a test is added to prevent the interchanging of an element with itself). There is little additional storage required (except to hold a few temporary variables). The sort may therefore be categorized as $O(n^2)$, although it is faster than the bubble sort. There is no improvement if the input file is completely sorted or unsorted, since the testing proceeds to completion without regard to the makeup of the file. Despite the fact that it is simple to code, it is unlikely that the straight selection sort would be used on any files but those for which n is small.

It is also possible to implement a sort by representing the descending priority queue by an ordered array. Interestingly, this leads to a sort consisting of a preprocessing phase that forms a sorted array of n elements. The selection phase is, therefore, superfluous. This sort is presented in Section 6.4 as the ***simple insertion sort***; it is not a selection sort, since no selection is required.

Binary Tree Sorts

In the remainder of this section we illustrate several selection sorts that represent a priority queue by a binary tree. The first method is the ***binary tree sort*** of Section 5.1, which uses a binary search tree. The reader is advised to review that sort before proceeding.

The method involves scanning each element of the input file and placing it into its proper position in a binary tree. To find the proper position of an element, y, a left or right branch is taken at each node, depending on whether y is less than the element in the node or greater than or equal to it. Once each input element is in its proper position in the tree, the sorted file can be retrieved by an inorder traversal of the tree. We present the algorithm for this sort, modifying it to accommodate the input as a preexisting array. Translating the algorithm to a C routine is straightforward.

```
/* establish the first element as root */
tree = maketree(x[0]);
/* repeat for each successive element */
```

```
for (i = 1; i < n; i++) {
   y = x[i];
   q = tree;
   p = q;
   /* travel down the tree until a leaf is reached */
   while (p != null) {
      q = p;
      if (y < info(p))
         p = left(p);
      else
         p = right(p);
   } /* end while */
   if (y < info(q))
      setleft(q,y);
   else
      setright(q,y);
} /* end for */
/* the tree is built, traverse it in inorder */
intrav(tree);
```

To convert the algorithm into a routine to sort an array, it is necessary to revise *intrav* so that visiting a node involves placing the contents of the node into the next position of the original array.

Actually, the binary search tree represents an ascending priority queue, as described in Exercises 5.1.13 and 5.2.13. Constructing the tree represents the preprocessing phase, and the traversal represents the selection phase of the general selection sort algorithm.

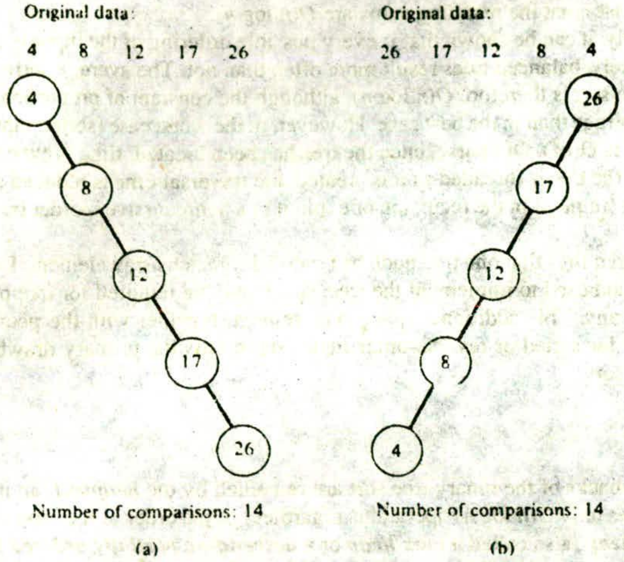
Ordinarily, extracting the minimum element (*pqmindelete*) of a priority queue represented by a binary search tree involves traveling down the left side of the tree from the root. Indeed, that is the first step of the inorder traversal process. However, since no new elements are inserted into the tree once the tree is constructed and the minimum element does not actually have to be deleted, the inorder traversal efficiently implements the successive selection process.

The relative efficiency of this method depends on the original order of the data. If the original array is completely sorted (or sorted in reverse order), the resulting tree appears as a sequence of only right (or left) links, as in Figure 6.3.1. In this case the insertion of the first node requires no comparisons, the second node requires two comparisons, the third node three comparisons, and so on. Thus the total number of comparisons is

$$2 + 3 + \cdots + n = n * (n + 1)/2 - 1$$

which is $O(n^2)$.

On the other hand, if the data in the original array is organized so that approximately half the numbers following any given number a in the array are less than a and half are greater than a, balanced trees such as those in Figure 6.3.2 result. In such a case the depth of the resulting binary tree is the smallest integer d greater than or equal to $\log_2 (n + 1) - 1$. The number of nodes at any level l (except possibly for the last) is



Number of comparisons: 14

(a)

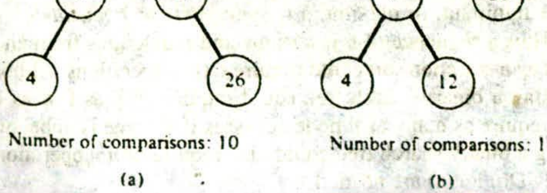
Number of comparisons: 14

(b)

Figure 6.3.1

2^l and the number of comparisons necessary to place a node at level l (except when $l = 0$) is $l + 1$. Thus the total number of comparisons is between

$$d + \sum_{l=1}^{d-1} 2^l * (l + 1) \text{ and } \sum_{l=1}^{d} 2^l * (l + 1)$$



Number of comparisons: 10

(a)

Number of comparisons: 10

(b)

Figure 6.3.2

It can be shown (mathematically inclined readers might be interested in proving this fact as an exercise) that the resulting sums are $O(n \log n)$.

Fortunately, it can be shown that if every possible ordering of the input is considered equally likely, balanced trees result more often than not. The average sorting time for a binary tree sort is therefore $O(n \log n)$, although the constant of proportionality is larger on the average than in the best case. However, in the worst case (sorted input), the binary tree sort is $O(n^2)$. Of course, once the tree has been created, time is expended in traversing it. If the tree is threaded as it is created, the traversal time is reduced and the need for a stack (implicit in the recursion or explicit in a nonrecursive inorder traversal) is eliminated.

This sort requires that one tree node be reserved for each array element. Depending on the method used to implement the tree, space may be required for tree pointers and threads, if any. This additional space requirement, together with the poor $O(n^2)$ time efficiency for sorted or reverse-order input, represents the primary drawback of the binary tree sort.

Heapsort

The drawbacks of the binary tree sort are remedied by the ***heapsort***, an in-place sort that requires only $O(n \log n)$ operations regardless of the order of the input. Define a ***descending heap*** (also called a ***max heap*** or a ***descending partially ordered tree***) of size n as an almost complete binary tree of n nodes such that the content of each node is less than or equal to the content of its father. If the sequential representation of an almost complete binary tree is used, this condition reduces to the inequality

$$info[j] \leq info[(j-1)/2] \quad \text{for} \quad 0 \leq ((j-1)/2) < j \leq n-1$$

It is clear from this definition of a descending heap that the root of the tree (or the first element of the array) contains the largest element in the heap. Also note that any path from the root to a leaf (or indeed, any path in the tree that includes no more than one node at any level) is an ordered list in descending order. It is also possible to define an ***ascending heap*** (or a ***min heap***) as an almost complete binary tree such that the content of each node is greater than or equal to the content of its father. In an ascending heap, the root contains the smallest element of the heap, and any path from the root to a leaf is an ascending ordered list.

A heap allows a very efficient implementation of a priority queue. Recall from Section 4.2 that an ordered list containing n elements allows priority queue insertion (*pqinsert*) to be implemented using an average of approximately $n/2$ node accesses, and deletion of the minimum or maximum (*pqmindelete* or *pqmaxdelete*) using only one node access. Thus a sequence of n insertions and n deletions from an ordered list such as is required by a selection sort could require $O(n^2)$ operations. Although priority queue insertion using a binary search tree could require only as few as $\log_2 n$ node accesses, it could require as many as n node accesses if the tree is unbalanced. Thus a selection sort using a binary search tree could also require $O(n^2)$ operations, although on the average only $O(n \log n)$ are needed.

As we shall see, a heap allows both insertion and deletion to be implemented in $O(\log n)$ operations. Thus a selection sort consisting of n insertions and n deletions can be implemented using a heap in $O(n \log n)$ operations, even in the worst case. An additional bonus is that the heap itself can be implemented within the input array x using the sequential implementation of an almost complete binary tree. The only additional space required is for program variables. The heapscrt is, therefore, an $O(n \log n)$ inplace sort.

Heap as a Priority Queue

Let us now implement a descending priority queue using a descending heap. Suppose that *dpq* is an array that implicitly represents a descending heap of size k. Because the priority queue is contained in array elements 0 to $k - 1$, we add k as a parameter of the insertion and deletion operations. Then the operation *pqinsert(dpq, k, elt)* can be implemented by simply inserting *elt* into its proper position in the descending list formed by the path from the root of the heap (*dpq*[0]) to the leaf *dpq*[*k*]. Once *pqinsert(dpq, k, elt)* has been executed, *dpq* becomes a heap of size $k + 1$.

The insertion is done by traversing the path from the empty position k to position 0 (the root), seeking the first element greater than or equal to *elt*. When that element is found, *elt* is inserted immediately preceding it in the path (that is, *elt* is inserted as its son). As each element less than *elt* is passed during the traversal, it is shifted down one level in the tree to make room for *elt*. (This shifting is necessary because we are using the sequential representation rather than a linked representation of the tree. A new element cannot be inserted between two existing elements without shifting some existing elements.)

This heap insertion operation is also called the *siftup* operation because *elt* sifts its way up the tree. The following algorithm implements *pqinsert (dpq, k, elt)*:

```
s = k;
f = (s - 1)/2;  /*  f is the father of s  */
while (s > 0 && dpq[f] < elt) {
   dpq[s] = dpq[f];
   s = f; /* advance up the tree */
   f = (s - 1)/2;
} /* end while */
dpq[s] = elt;
```

Insertion is clearly $O(\log n)$, since an almost complete binary tree with n nodes has $\log_2 n + 1$ levels, and at most, one node per level is accessed.

We now examine how to implement *pqmaxdelete (dpq,k)* for a descending heap of size k. First we define *subtree* (p,m), where m is greater than p, as the subtree (of the descending heap) rooted at position p within the elements *dpq*[*p*] through *dpq*[*m*]. For example, *subtree*(3,10) consists of the root *dpq*[3] and its two children *dpq*[7] and *dpq*[8]. *subtree*(3,17) consists of *dpq*[3], *dpq*[7], *dpq*[8], *dpq*[15], *dpq*[16], and *dpq*[17]. If *dpq*[*i*] is included in *subtree*(*p,m*), *dpq*[$2 * i + 1$] is included if and only if $2 * i + 1 <= m$, and $dpq[2 * i + 2]$ is included if and only if $2 * i + 2 <= m$. If m is less than p, *subtree(p,m)* is defined as the empty tree.

To implement *pqmaxdelete(dpq,k)*, we note that the maximum element is always at the root of a k-element descending heap. When that element is deleted, the remaining $k - 1$ elements in positions 1 through $k - 1$ must be redistributed into positions 0 through $k - 2$ so that the resulting array segment from $dpq[0]$ through $dpq[k - 2]$ remains a descending heap. Let *adjustheap(root,k)* be the operation of rearranging the elements $dpq[root + 1]$ through $dpq[k]$ into $dpq[root]$ through $dpq[k - 1]$ so that *subtree(root,k − 1)* forms a descending heap. Then *pqmaxdelete(dpq,k)* for a k-element descending heap can be implemented by

```
p = dpq[0];
adjustheap(0,k - 1);
return(p);
```

In a descending heap, not only is the root element the largest element in the tree, but an element in *any* position p must be the largest element in *subtree(p,k)*. Now, *subtree(p,k)* consists of three groups of elements: its root, $dpq[p]$; its left subtree, *subtree*$(2 * p + 1, k)$; and its right subtree, *subtree*$(2 * p + 2, k)$. $dpq[2 * p + 1]$, the left son of the root, is the largest element of the left subtree, and $dpq[2 * p + 2]$, the right son of the root, is the largest element of the right subtree. When the root $dpq[p]$ is deleted, the larger of these two sons must move up to take its place as the new largest element of *subtree(p,k)*. Then the subtree rooted at the position of the larger element moved up must be readjusted in turn.

Let us define *largeson(p, m)* as the larger son of $dpq[p]$ within *subtree(p, m)*. It may be implemented as

```
s = 2 * p + 1;
if (s + 1 <= m && x[s] < x[s + 1])
   s = s + 1;
/* check if out of bounds */
if (s > m)
   return(-1);
else
   return(s);
```

Then *adjustheap(root, k)* may be implemented recursively by

```
f = root;
s = largeson(f, k - 1);
if (s >= 0 && dpq[k] < dpq[s]) {
   dpq[f] = dpq[s];
   adjustheap(s, k);
}
else
   dpq[f] = dpq[k];
```

The following is an iterative version of *adjustheap*. The algorithm uses a temporary variable *kvalue* to hold the value of *dpq*[*k*]:

```
f = root;
kvalue = dpq[k];
s = largeson(f, k - 1);
while (s >= 0 && kvalue < dpq[s])  {
   dpq[f] = dpq[s];
   f = s;
   s = largeson(f, k - 1);
}
dpq[f] = kvalue;
```

Note that we traverse a path of the tree from the root toward a leaf, shifting up by one position all elements in the path greater than *dpq*[*k*] and inserting *dpq*[*k*] in its proper position in the path. Again, the shifting is necessary because we are using the sequential representation rather than a linked implementation of the tree. The adjustment procedure is often called the *siftdown* operation because *dpq*[*k*] sifts its way from the root down the tree.

This heap deletion algorithm is also $O(\log n)$, since there are $\log_2 n + 1$ levels in the tree and at most two nodes are accessed at each level. However, the overhead of shifting and computing *largeson* is significant.

Sorting Using a Heap

Heapsort is simply an implementation of the general selection sort using the input array *x* as a heap representing a descending priority queue. The preprocessing phase creates a heap of size *n* using the siftup operation, and the selection phase redistributes the elements of the heap in order as it deletes elements from the priority queue using the siftdown operation. In both phases the loops need not include the case where *i* equals 0, since *x*[0] is already a one-element priority queue and the array is sorted once *x*[1] through *x*[$n - 1$] are in proper position.

```
/* Create the priority queue; before each loop iteration */
/* the priority queue consists of elements x[0] through  */
/* x[i - 1]. Each iteration adds x[i] to the queue.      */
for (i = 1; i < n; i++)
   pqinsert(x, i, x[i]);
/* select each successive element in order */
for (i = n - 1; i > 0; i--)
   x[i] = pqmaxdelete(x, i + 1);
```

Figure 6.3.3 illustrates the creation of a heap of size 8 from the original file

25 57 48 37 12 92 86 33

The dotted lines in that figure indicate an element being shifted down the tree.

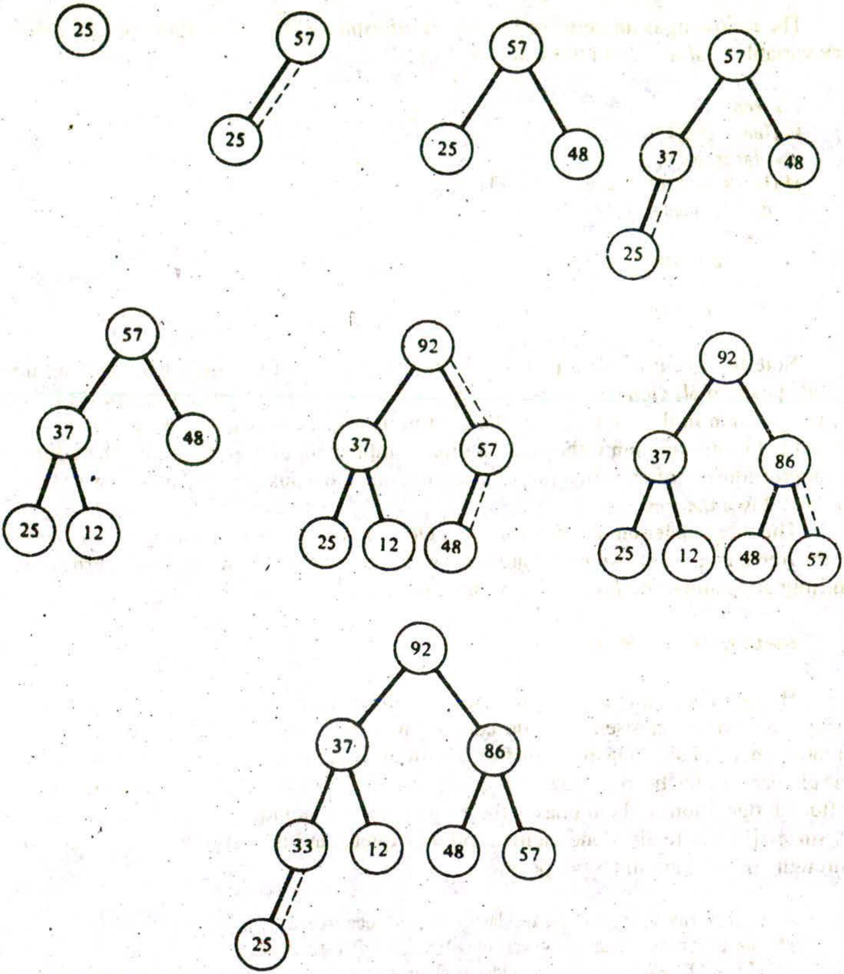
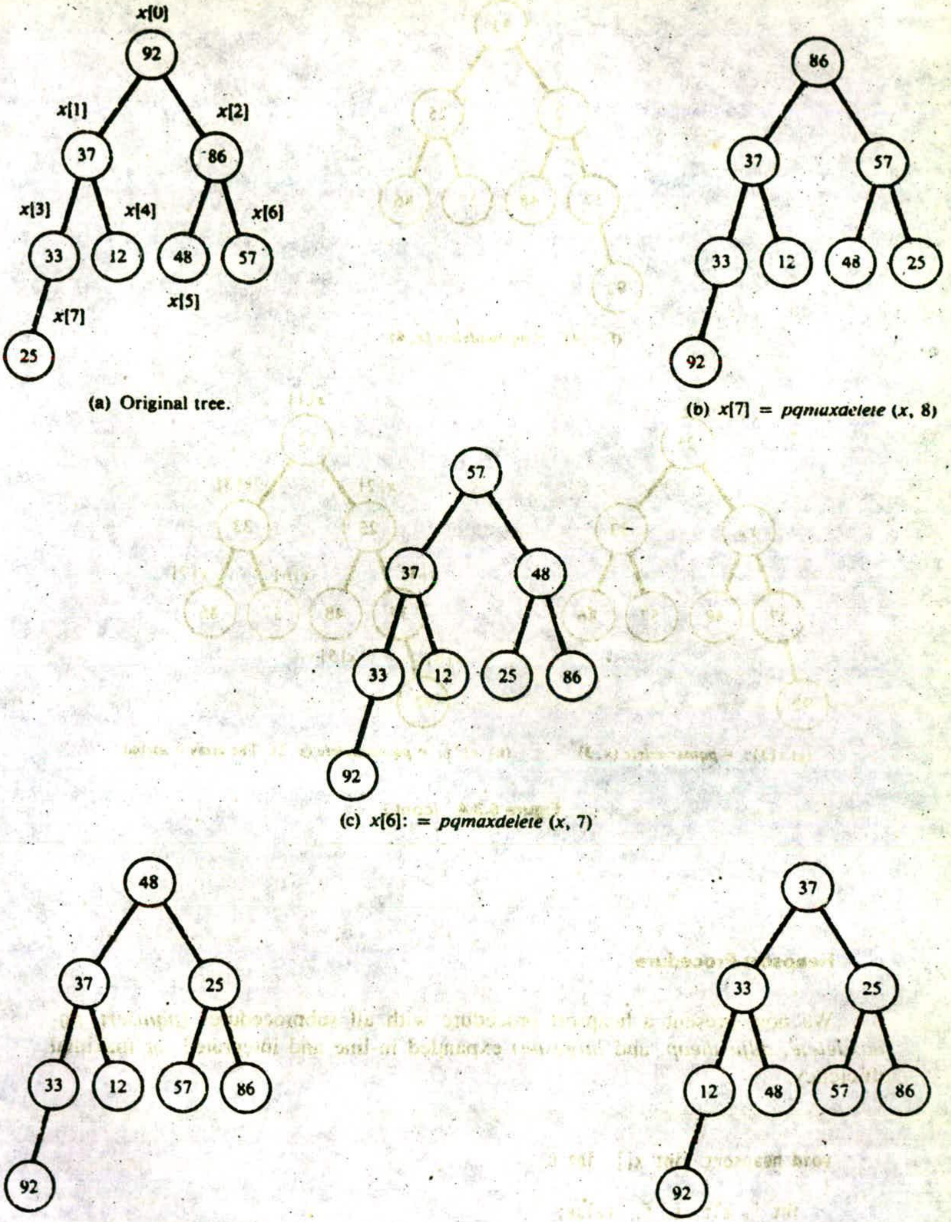


Figure 6.3.3 Creating a heap of size 8.

Figure 6.3.4 illustrates the adjustment of the heap as $x[0]$ is repeatedly selected and placed into its proper position in the array and the heap is readjusted, until all the heap elements are processed. Note that after an element has been "deleted" from the heap, it remains in the array; it is merely ignored in subsequent processing.



(a) Original tree.

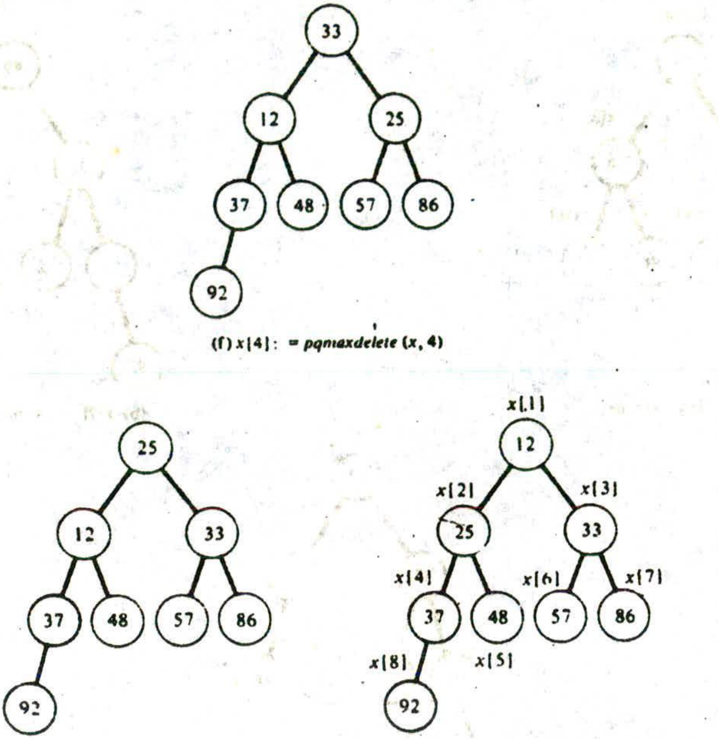
(b) *x*[7] = *pqmaxdelete* (*x*, 8)

(c) *x*[6]: = *pqmaxdelete* (*x*, 7)

(d) *x*[5]: = *pqmaxdelete* (*x*, 6)

(e) *x*[4]: = *pqmaxdelete* (*x*, 5)

Figure 6.3.4 Adjusting a heap.



(f) *x*[4]: = *pqmaxdelete* (*x*, 4)

(g) *x*[3]: = *pqmaxdelete* (*x*, 3)

(h) *x*[2]: = *pqmaxdelete* (*x*, 2). The array is sorted.

Figure 6.3.4 (cont.)

Heapsort Procedure

We now present a heapsort procedure with all subprocedures (*pqinsert*, *pqmaxdelete*, *adjustheap*, and *largeson*) expanded in-line and integrated for maximal efficiency.

```
void heapsort (int x[], int n)
{
   int i, elt, s, f, ivalue;
   /* preprocessing phase; create initial heap */
   for (i = 1; i < n; i++) {
      elt = x[i];
      /* pqinsert(x, i, elt) */
      s = i;
      f = (s-1)/2;
      while (s > 0 && x[f] < elt)  {
         x[s] = x[f];
         s = f;
         f = (s-1)/2;
      } /* end while */
      x[s] = elt;
   }  /* end for */
   /* selection phase; repeatedly remove x[0], insert it */
   /*     in its proper position and adjust the heap      */
   for (i = n-1; i > 0; i--)  {
      /* pqmaxdelete(x, i+1) */
      ivalue = x[i];
      x[i] = x[0];
      f = 0;
      /* s = largeson (0, i-1) */
      if (i == 1)
         s = -1;
      else
         s = 1;
      if (i > 2 && x[2] > x[1])
         s = 2;
      while (s >= 0 && ivalue < x[s]) {
         x[f] = x[s];
         f = s;
         /* s = largeson(f, i-1) */
         s = 2*f+1;
         if (s+1 <= i-1 && x[s] < x[s+1])
            s = s+1;
         if (s > i-1)
            s = -1;
      } /* end while */
      x[f] = ivalue;
   } /* end for */
} /* end heapsort */
```

To analyze the heapsort, note that a complete binary tree with n nodes (where n is one less than a power of two) has log $(n + 1)$ levels. Thus if each element in the array were a leaf, requiring it to be filtered through the entire tree both while creating and adjusting the heap, the sort would still be $O(n \log n)$.

In the average case the heapsort is not as efficient as the quicksort. Experiments indicate that heapsort requires twice as much time as quicksort for randomly sorted input. However, heapsort is far superior to quicksort in the worst case. In fact, heapsort remains $O(n \log n)$ in the worst case. Heapsort is also not very efficient for small n because of the overhead of initial heap creation and computation of the location of fathers and sons.

The space requirement for the heapsort (aside from array indices) is only one additional record to hold the temporary for switching, provided the array implementation of an almost complete binary tree is used.

EXERCISES

6.3.1. Explain why the straight selection sort is more efficient than the bubble sort.

6.3.2. Consider the following ***quadratic selection sort:*** Divide the n elements of the file into $\sqrt{n}$ groups of $\sqrt{n}$ elements each. Find the largest element of each group and insert it into an auxiliary array. Find the largest of the elements in this auxiliary array. This is the largest element of the file. Then replace this element in the array by the next largest element of the group from which it came. Again find the largest element of the auxiliary array. This is the second largest element of the file. Repeat the process until the file has been sorted. Write a C routine to implement a quadratic selection sort as efficiently as possible.

6.3.3. A ***tournament*** is an almost complete strictly binary tree in which each nonleaf contains the larger of the two elements in its two sons. Thus the contents of a tournament's leaves completely determine the contents of all its nodes. A tournament with n leaves represents a set of n elements.

- **(a)** Develop an algorithm *pqinsert*(t, n, *elt*) to add a new element *elt* to a tournament containing n leaves represented implicitly by an array t.
- **(b)** Develop an algorithm *pqmaxdelete*(t,n) to delete the maximum element from a tournament with n elements by replacing the leaf containing the maximum element with a dummy value smaller than any possible element (for example, -1 in a tournament of nonnegative integers) and then readjusting all values in the path from that leaf to the root.
- **(c)** Show how to simplify *pqmaxdelete* by maintaining a pointer to a leaf in each nonleaf *info* field, rather than an actual element value.
- **(d)** Write a C program to implement a selection sort using a tournament. The preprocessing phase builds the initial tournament from the array x and the selection phase applies *pqmaxdelete* repeatedly. Such a sort is called a ***tournament sort***.
- **(e)** How does the efficiency of the tournament sort compare with that of the heapsort?
- **(f)** Prove that the tournament sort is $O(n \log n)$ for all input.

6.3.4. Define an ***almost complete ternary tree*** as a tree in which every node has at most three sons, and in which the nodes can be numbered from 0 to $n - 1$, so that the sons of *node*[i] are *node*[$3 * i + 1$], *node*[$3 * i + 2$], and *node*[$3 * i + 3$]. Define a ***ternary heap*** as an almost complete ternary tree in which the content of each node is greater than or equal to the contents of all its descendants. Write a sorting routine similar to the heapsort using a ternary heap.

6.3.5. Write a routine *combine*(x) that accepts an array x in which the subtrees rooted at $x[1]$ and $x[2]$ are heaps and that modifies the array x so that it represents a single heap.

6.3.6. Rewrite the program of Section 5.3 that implements the Huffman algorithm so that the set of root nodes forms a priority queue implemented by an ascending heap.

6.3.7. Write a C program that uses an ascending heap to merge n input files, each sorted in ascending order, into a single output file. Each node of the heap contains a file number and a value. The value serves as the key by which the heap is organized. Initially, one value is read from each file, and the n values are formed into an ascending heap, with the file number from which each value came kept together with that value in a node. The smallest value is then in the root of the heap and it is the output, with the next value of its associated file input to take its place. That value, together with its associated file number, is sifted down to find its proper place in the heap, and the new root value is output. This process of output/input/siftdown is repeated until no input remains.

6.3.8. Develop an algorithm using a heap of k elements to find the largest k numbers in a large, unsorted file of n numbers.

6.4 INSERTION SORTS

Simple Insertion

An ***insertion sort*** is one that sorts a set of records by inserting records into an existing sorted file. An example of a simple insertion sort is the following procedure:

```
void insertsort(int x[], int n)
{
   int i, k, y;

   /* initially x[0] may be thought of as a sorted file of */
   /* one element.  After each repetition of the following */
   /* loop, the elements x[0] through x[k] are in order.   */
   for (k = 1; k < n; k++)  {
      /* Insert x[k] into the sorted file */
      y = x[k];
      /* Move down 1 position all elements greater than y  */
      for (i = k-1; i >= 0 && y < x[i]; i--)
         x[i+1] = x[i];
      /* Insert y at proper position */
      x[i+1] = y;
   } /* end for */
} /* end insertsort */
```

As we noted at the beginning of Section 6.3, the simple insertion sort may be viewed as a general selection sort in which the priority queue is implemented as an ordered array. Only the preprocessing phase of inserting the elements into the priority queue is necessary; once the elements have been inserted, they are already sorted, so that no selection is necessary.

If the initial file is sorted, only one comparison is made on each pass, so that the sort is $O(n)$. If the file is initially sorted in the reverse order, the sort is $O(n^2)$, since the total number of comparisons is

$$(n-1) + (n-2) + \cdots + 3 + 2 + 1 = (n-1) * n/2$$

which is $O(n^2)$. However, the simple insertion sort is still usually better than the bubble sort. The closer the file is to sorted order, the more efficient the simple insertion sort becomes. The average number of comparisons in the simple insertion sort (by considering all possible permutations of the input array) is also $O(n^2)$. The space requirements for the sort consist of only one temporary variable, *y*.

The speed of the sort can be improved somewhat by using a binary search (see Sections 3.1, 3.2, and 7.1) to find the proper position for $x[k]$ in the sorted file $x[0], \ldots, x[k-1]$. This reduces the total number of comparisons from $O(n^2)$ to $O(n \log n)$. However, even if the correct position *i* for $x[k]$ is found in $O(\log n)$ steps, each of the elements $x[i+1], \ldots, x[k-1]$ must be moved one position. This latter operation performed *n* times requires $O(n^2)$ replacements. Unfortunately, the binary search technique does not, therefore, significantly improve the overall time requirements of the sort.

Another improvement to the simple insertion sort can be made by using ***list insertion***. In this method there is an array *link* of pointers, one for each of the original array elements. Initially $link[i] = i + 1$ for $0 <= i < n - 1$ and $link[n-1] = -1$. Thus the array may be thought of as a linear list pointed to by an external pointer *first* initialized to 0. To insert the *k*th element the linked list is traversed until the proper position for $x[k]$ is found, or until the end of the list is reached. At that point $x[k]$ can be inserted into the list by merely adjusting the list pointers without shifting any elements in the array. This reduces the time required for insertion but not the time required for searching for the proper position. The space requirements are also increased because of the extra *link* array. The number of comparisons is still $O(n^2)$, although the number of replacements in the *link* array is $O(n)$. The list insertion sort may be viewed as a general selection sort in which the priority queue is represented by an ordered list. Again, no selection is needed because the elements are sorted as soon as the preprocessing, insertion phase is complete. You are asked to code both the binary insertion sort and the list insertion sort as exercises.

Both the straight selection sort and the simple insertion sort are more efficient than bubble sort. Selection sort requires fewer assignments than insertion sort but more comparisons. Thus selection sort is recommended for small files when records are large, so the assignment is inexpensive, but keys are simple, so that comparison is cheap. If the reverse situation holds, insertion sort is recommended. If the input is initially in a linked list, list insertion is recommended even if the records are large, since no data movement (as opposed to pointer modification) is required.

Of course, heapsort and quicksort are both more efficient than insertion or selection for large *n*. The break even point is approximately 20–30 for quicksort; for fewer than 30 elements use insertion sort; for more than 30 use quicksort. A useful speedup of quicksort uses insertion sort on any subfile of size less than 20. For heapsort, the break even point with insertion sort is approximately 60–70.

Shell Sort

More significant improvement on simple insertion sort than binary or list insertion can be achieved by using the ***Shell sort*** (or ***diminishing increment sort***), named after its discoverer. This method sorts separate subfiles of the original file. These subfiles contain every kth element of the original file. The value of k is called an ***increment***. For example, if k is 5, the subfile consisting of $x[0]$, $x[5]$, $x[10]$, ... is first sorted. Five subfiles, each containing one fifth of the elements of the original file are sorted in this manner. These are (reading across)

Subfile 1	->	$x[0]$	$x[5]$	$x[10]$	...
Subfile 2	->	$x[1]$	$x[6]$	$x[11]$	...
Subfile 3	->	$x[2]$	$x[7]$	$x[12]$	...
Subfile 4	->	$x[3]$	$x[8]$	$x[13]$	...
Subfile 5	->	$x[4]$	$x[9]$	$x[14]$	...

The ith element of the jth subfile is $x[(i - 1) * 5 + j - 1]$. If a different increment k is chosen, the k subfiles are divided so that tne ith element of the jth subfile is $x[(i - 1) * k + j - 1]$.

After the first k subfiles are sorted (usually by simple insertion), a new smaller value of k is chosen and the file is again partitioned into a new set of subfiles. Each of these larger subfiles is sorted and the process is repeated yet again with an even smaller value of k. Eventually, the value of k is set to 1 so that the subfile consisting of the entire file is sorted. A decreasing sequence of increments is fixed at the start of the entire process. The last value in this sequence must be 1.

For example, if the original file is

25 57 48 37 12 92 86 33

and the sequence (5,3,1) is chosen, the following subfiles are sorted on each iteration:

First iteration (increment = 5)

$(x[0], x[5])$
$(x[1], x[6])$
$(x[2], x[7])$
$(x[3])$
$(x[4])$

Second iteration (increment = 3)

$(x[0], x[3], x[6])$
$(x[1], x[4], x[7])$
$(x[2], x[5])$

Third iteration (increment = 1)

$(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])$

Figure 6.4.1 illustrates the Shell sort on this sample file. The lines underneath each array join individual elements of the separate subfiles. Each of the subfiles is sorted using the simple insertion sort.

We present below a routine to implement the Shell sort. In addition to the standard parameters x and n, it requires an array *incrmnts*, containing the diminishing increments of the sort, and *numinc*, the number of elements in the array *incrmnts*.

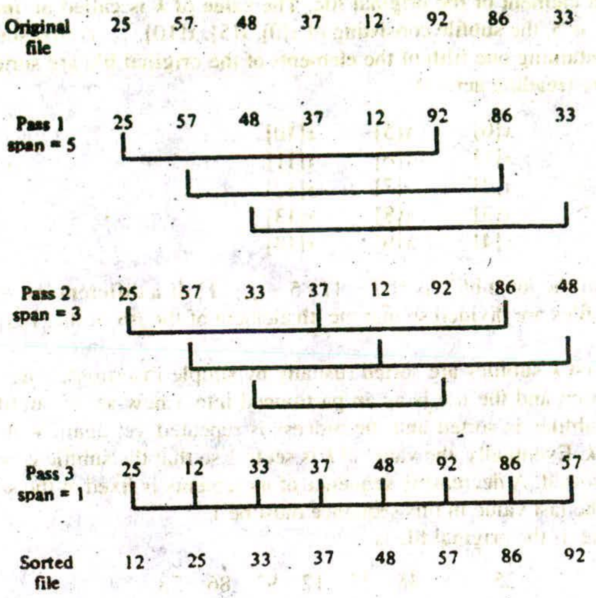


Figure 6.4.1

```
void shellsort(int x[], int n, int incrmnts[], int numinc)
{
   int incr, j, k, span, y;

   for (incr = 0; incr < numinc; incr++) {
      /*  span is the size of the increment     */
      span = incrmnts[incr];
      for (j = span; j < n; j++)  {
         /* Insert element x[j] into its proper */
         /* position within its subfile         */
         y = x[j];
         for (k = j-span; k >= 0 && y < x[k]; k -= span)
            x[k+span] = x[k];
         x[k+span] = y;
      }  /* end for */
   } /* end for */
}  /* end shellsort */
```

Be sure that you can trace the actions of this program on the sample file of Figure 6.4.1. Notice that on the last iteration, where *span* equals 1, the sort reduces to a simple insertion.

The idea behind the Shell sort is a simple one. We have already noted that the simple insertion sort is highly efficient on a file that is in almost sorted order. It is also important to realize that when the file size n is small, an $O(n^2)$ sort is often more efficient than an $O(n \log n)$ sort. The reason for this is that $O(n^2)$ sorts are generally quite simple to program and involve very few actions other than comparisons and replacements on each pass. Because of this low overhead, the constant of proportionality is rather small. An $O(n \log n)$ sort is generally quite complex and employs a large number of extra operations on each pass in order to reduce the work of subsequent passes. Thus its constant of proportionality is larger. When n is large, n^2 overwhelms $n*\log(n)$, so that the constants of proportionality do not play a major role in determining the faster sort. However, when n is small, n^2 is not much larger than $n*\log(n)$, so that a large difference in those constants often causes an $O(n^2)$ sort to be faster.

Since the first increment used by the Shell sort is large, the individual subfiles are quite small, so that the simple insert on sorts on those subfiles are fairly fast. Each sort of a subfile causes the entire file to be more nearly sorted. Thus, although successive passes of the Shell sort use smaller increments and therefore deal with larger subfiles, those subfiles are almost sorted due to the actions of previous pa ᵉs. Thus, the insertion sorts on those subfiles are also quite efficient. In this connection, it is significant to note that if a file is partially sorted using an increment k and is subsequently partially sorted using an increment j, the file remains partially sorted on the increment k. That is, subsequent partial sorts do not disturb earlier ones.

The efficiency analysis of the Shell sort is mathematically involved and beyond the scope of this book. The actual time requirements for a specific sort depend on the number of elements in the array *incrmnts* and on their actual values. One requirement that is intuitively clear is that the elements of *incrmnts* should be relatively prime (that is, have no common divisors other than 1). This guarantees that successive iterations intermingle subfiles so that the entire file is indeed almost sorted when *span* equals 1 on the last iteration.

It has been shown that the order of the Shell sort can be approximated by $O(n(\log n)^2)$ if an appropriate sequence of increments is used. For other series of increments, the running time can be proven to be $O(n^{1.5})$. Empirical data indicates that the running time is of the form $a * n^b$, where a is between 1.1 and 1.7 and b is approximately 1.26, or of the form $c * n * (ln(n))^2 - d * n * ln(n)$, where c is approximately 0.3 and d is between 1.2 and 1.75. In general the Shell sort is recommended for moderately sized files of several hundred elements.

Knuth recommends choosing increments as follows: define a function h recursively so that $h(1) = 1$ and $h(i + 1) = 3 * h(i) + 1$. Let x be the smallest integer such that $h(x) \geq n$, and set *numinc*, the number of increments, to $x - 2$ and *incrmnts*$[i]$ to $h(numinc - i + 1)$ for i from 1 to *numinc*.

A technique similar to the Shell sort can also be used to improve the bubble sort. In practice, a major source of the bubble sort's inefficiency is not the number of comparisons but the number of interchanges. If a series of increments are used to define subfiles to be bubble sorted individually, as in the case of the Shell sort, the initial bubble sorts are on small files and the later ones are on more nearly sorted files in which few interchanges are necessary. This modified bubble sort, which requires very little overhead, works well in practical situations.

Address Calculation Sort

As a final example of sorting by insertion, consider the following technique called sorting by ***address calculation*** (sometimes called sorting by ***hashing***). In this method a function *f* is applied to each key. The result of this function determines into which of several subfiles the record is to be placed. The function should have the property that if $x \leq y, f(x) \leq f(y)$. Such a function is called ***order-preserving***. Thus all of the records in one subfile will have keys that are less than or equal to the keys of the records in another subfile. An item is placed into a subfile in correct sequence by using any sorting method; simple insertion is often used. After all the items of the original file have been placed into subfiles, the subfiles may be concatenated to produce the sorted result.

For example, consider again the sample file

25 57 48 37 12 92 86 33

Let us create ten subfiles, one for each of the ten possible first digits. Initially, each of these subfiles is empty. An array of pointers *f*[10] is declared, where *f*[*i*] points to the first element in the file whose first digit is *i*. After scanning the first element (25) it is placed into the file headed by *f*[2]. Each of the subfiles is maintained as a sorted linked list of the original array elements. After processing each of the elements in the original file, the subfiles appear as in Figure 6.4.2.

We present a routine to implement the address calculation sort. The routine assumes an array of two-digit numbers and uses the first digit of each number to assign that number to a subfile.

```
#define NUMELTS ...

addr(int x[], int n)
{
    int f[10], first, i, j, p, y;
    struct {
       int info;
       int next;
    } node[NUMELTS];

    /* Initialize available linked list */
    int avail = 0;
    for (i = 0; i < n-1; i++)
       node[i].next = i+1;
    node[n-1].next = -1;
    /* Initialize pointers */
    for (i = 0; i< 10; i++)
       f[i] = -1;

    for (i = 0; i < n; i++) {
       /* We successively insert each element into its */
       /* respective subfile using list insertion.     */
```

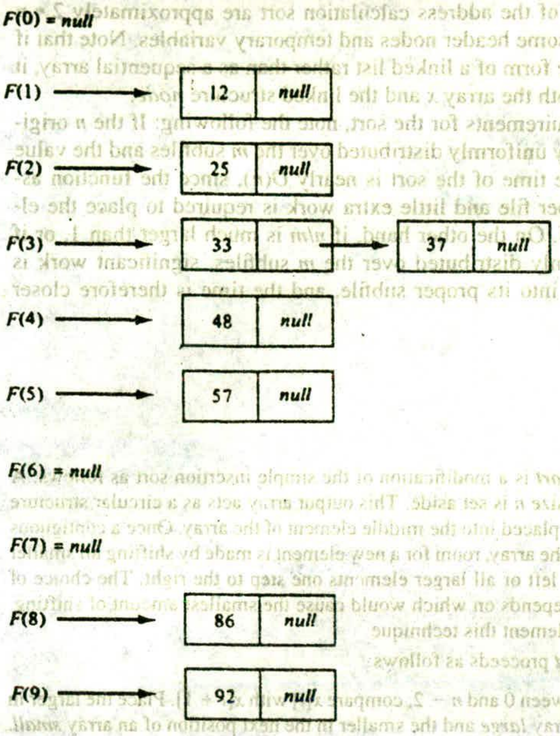


Figure 6.4.2 Address calculation sort.

```
          y = x[i];
          first = y/10;  /* Find the 1st digit of a two digit number */
          /* Search the linked list */
          place (&f[first], y);
          /* place inserts y into its proper position  */
          /* in the linked list pointed to by f[first] */
     } /* end for */
     /* Copy numbers back into the array x          */
     i = 0;
     for (j = 0; j < 10; j++) {
          p = f[j];
          while (p != -1) {
               x[i++] = node[p].info;
               p = node[p].next;
          } /* end while */
     } /* end for */
} /* end addr */
```

The space requirements of the address calculation sort are approximately $2 * n$ (used by the array *node*) plus some header nodes and temporary variables. Note that if the original data is given in the form of a linked list rather than as a sequential array, it is not necessary to maintain both the array *x* and the linked structure *node*.

To evaluate the time requirements for the sort, note the following: If the *n* original elements are approximately uniformly distributed over the *m* subfiles and the value of *n/m* is approximately 1, the time of the sort is nearly $O(n)$, since the function assigns each element to its proper file and little extra work is required to place the element within the subfile itself. On the other hand, if *n/m* is much larger than 1, or if the original file is not uniformly distributed over the *m* subfiles, significant work is required to insert an element into its proper subfile, and the time is therefore closer to $O(n^2)$.

EXERCISES

6.4.1. The ***two-way insertion sort*** is a modification of the simple insertion sort as follows: A separate output array of size *n* is set aside. This output array acts as a circular structure as in Section 4.1. *x*[0] is placed into the middle element of the array. Once a contiguous group of elements are in the array, room for a new element is made by shifting all smaller elements one step to the left or all larger elements one step to the right. The choice of which shift to perform depends on which would cause the smallest amount of shifting. Write a C routine to implement this technique.

6.4.2. The ***merge insertion sort*** proceeds as follows:

Step 1: For all even *i* between 0 and $n - 2$, compare *x*[*i*] with *x*[*i* + 1]. Place the larger in the next position of an array *large* and the smaller in the next position of an array *small*. If *n* is odd, place *x*[*n* − 1] in the last position of the array *small*. (*Large* is of size *ind*, where *ind* = (*n* − 1)/2; *small* is of size *ind* or *ind* + 1, depending on whether *n* is even or odd.)

Step 2: Sort the array *large* using merge insertion recursively. Whenever an element *large*[*j*] is moved to *large*[*k*], *small*[*j*] is also moved to *small*[*k*]. (At the end of this step, *large*[*i*] <= *large*[*i* + 1] for all *i* less than *ind*, and *small*[*i*] <= *large*[*i*] for all *i* less than or equal to *ind*.

Step 3: Copy *small*[0] and all the elements of *large* into *x*[0] through *x*[*ind*].

Step 4: Define the integer *num*[*i*] as $(2^{i+1} + (-1)^i)/3$. Beginning with *i* = 0 and proceeding by 1 while *num*[*i*] <= (*n*/2) + 1, insert the elements *small*[*num*[*i* + 1]] down to *small*[*num*[*i*] + 1] into *x* in turn, using binary insertion. (For example, if *n* = 20, the successive values of *num* are *num*[0] = 1, *num*[1] = 1, *num*[2] = 3, *num*[3] = 5, and *num*[4] = 11, which equals (*n*/2) + 1. Thus the elements of *small* are inserted in the following order: *small*[2], *small*[1]; then *small*[4], *small*[3]; then *small*[9], *small*[8], *small*[7], *small*[6], *small*[5]. In this example, there is no *small*[10].)

Write a C routine to implement this technique.

6.4.3. Modify the quicksort of Section 6.2 so that it uses a simple insertion sort when a subfile is below some size *s*. Determine by experiments what value of *s* should be used for maximum efficiency.

6.4.4. Prove that if a file is partially sorted using an increment j in the Shell sort, it remains partially sorted on that increment even after it is partially sorted on another increment, k.

6.4.5. Explain why it is desirable to choose all the increments of the Shell sort so that they are relatively prime.

6.4.6. What is the number of comparisons and interchanges (in terms of file size n) performed by each of the following sorting methods (a–j) for the following files:

1. A sorted file
2. A file that is sorted in reverse order (that is, from largest to smallest)
3. A file in which the elements $x[0], x[2], x[4], \ldots$ are the smallest elements and are in sorted order, and in which the elements $x[1], x[3], x[5], \ldots$ are the largest elements and are in reverse sorted order (that is, $x[0]$ is the smallest, $x[1]$ is the largest, $x[2]$ is next to smallest, $x[3]$ is the next to the largest, and so on)
4. A file in which $x[0]$ through $x[ind]$ (where $ind = (n - 1)/2$) are the smallest elements and are sorted, and in which $x[ind + 1]$ through $x[n - 1]$ are the largest elements and are in reverse sorted order
5. A file in which $x[0], x[2], x[4], \ldots$ are the smallest elements in sorted order, and in which $x[1], x[3], x[5], \ldots$ are the largest elements in sorted order

(a) Simple insertion sort
(b) Insertion sort using a binary search
(c) List insertion sort
(d) Two-way insertion sort of Exercise 6.4.1
(e) Merge insertion sort of Exercise 6.4.2
(f) Shell sort using increments 2 and 1
(g) Shell sort using increments 3, 2, and 1
(h) Shell sort using increments 8, 4, 2, and 1
(i) Shell sort using increments 7, 5, 3, and 1
(j) Address calculation sort presented in the text

6.4.7. Under what circumstances would you recommend the use of each of the following sorts over the others?

(a) Shell sort of this section
(b) Heapsort of Section 6.3
(c) Quicksort of Section 6.2

6.4.8. Determine which of the following sorts is most efficient.

(a) Simple insertion sort of this section
(b) Straight selection sort of Section 6.3
(c) Bubble sort of Section 6.2

6.5 MERGE AND RADIX SORTS

Merge Sorts

Merging is the process of combining two or more sorted files into a third sorted file. An example of a routine that accepts two sorted arrays a and b of $n1$ and $n2$ elements, respectively, and merges them into a third array c containing $n3$ elements is the following:

```
void mergearr(int a[], int_b[], int c[], int n1, int n2, int n3)
{
   int apoint, bpoint, cpoint;
   int alimit, blimit, climit;

   alimit = n1-1;
   blimit = n2-1;
   climit = n3-1;
   if (n1 + n2 != n3) {
      printf("array bounds incompatible/n");
      exit(1);
   } /* end if */
   /* apoint and bpoint are indicators of how far */
   /* we are in arrays a and b respectively.      */
   apoint = 0;
   bpoint = 0;
   for (cpoint = 0; apoint <= alimit && bpoint <= blimit; cpoint++)
      if (a[apoint] < b[bpoint])
         c[cpoint] = a[apoint++];
      else
         c[cpoint] = b[bpoint++];
   while (apoint <= alimit)
      c[cpoint++] = a[apoint++];
   while (bpoint <= blimit)
      c[cpoint++] = b[bpoint++];
} /* end mergearr */
```

We can use this technique to sort a file in the following way. Divide the file into *n* subfiles of size 1 and merge adjacent (disjoint) pairs of files. We then have approximately *n*/2 files of size 2. Repeat this process until there is only one file remaining of size *n*. Figure 6.5.1 illustrates how this process operates on a sample file. Each individual file is contained in brackets.

We present a routine to implement the foregoing description of a ***straight merge sort***. An auxiliary array *aux* of size *n* is required to hold the results of merging two subarrays of *x*. The variable *size* contains the size of the subarrays being merged. Since at any time the two files being merged are both subarrays of *x*, lower and upper bounds are required to indicate the subfiles of *x* being merged. *l*1 and *u*1 represent the lower and upper bounds of the first file, and *l*2 and *u*2 represent the lower and upper bounds of the second file, respectively. *i* and *j* are used to reference elements of the source files being merged, and *k* indexes the destination file *aux*. The routine follows:

```
#define NUMELTS ...

void mergesort(int x[], int n)
{
   int aux[NUMELTS], i, j, k, l1, l2, size, u1, u2;
```

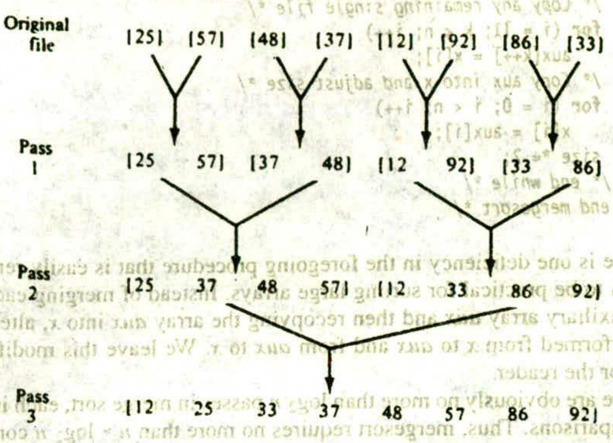


Figure 6.5.1 Successive passes of the merge sort.

```
size = 1;  /* Merge files of size 1 */
while (size < n) {
   l1 = 0;      /* Initialize lower bounds of first file */
   k = 0;         /* k is index for auxiliary array. */
   while (l1+size < n) {      /* Check to see if there */
                              /* are two files to merge */
      /* Compute remaining indices */
      l2 = l1+size;
      u1 = l2-1;
      u2 = (l2+size-1 < n) ? l2+size-1 : n-1;
      /* Proceed through the two subfiles */
      for (i = l1, j = l2; i <= u1 && j <= u2; k++)
         /* Enter smaller into the array aux */
         if (x[i] <= x[j])
            aux[k] = x[i++];
         else
            aux[k] = x[j++];
      /* At this point, one of the subfiles   */
      /* has been exhausted. Insert any       */
      /* remaining portions of the other file */
      for (; i <= u1; k++)
         aux[k] = x[i++];
      for (; j <= u2; k++)
         aux[k] = x[j++];
      /* Advance l1 to the start of the next pair of files. */
      l1 = u2+1;
   } /* end while */
```

```
      /* Copy any remaining single file */
      for (i = l1; k < n; i++)
         aux[k++] = x[i];
      /* Copy aux into x and adjust size */
      for (i = 0; i < n; i++)
         x[i] = aux[i];
      size *= 2;
   } /* end while */
} /* end mergesort */
```

There is one deficiency in the foregoing procedure that is easily remedied if the program is to be practical for sorting large arrays. Instead of merging each set of files into the auxiliary array *aux* and then recopying the array *aux* into *x*, alternate merges can be performed from *x* to *aux* and from *aux* to *x*. We leave this modification as an exercise for the reader.

There are obviously no more than $\log_2 n$ passes in merge sort, each involving n or fewer comparisons. Thus, mergesort requires no more than $n * \log_2 n$ comparisons. In fact, it can be shown that mergesort requires fewer than $n * \log_2 n - n + 1$ comparisons, on the average, compared with $1.386 * n * \log_2 n$ average comparisons for quicksort. In addition, quicksort can require $O(n^2)$ comparisons in the worst case, whereas mergesort never requires more than $n * \log_2 n$. However, mergesort does require approximately twice as many assignments as quicksort on the average, even if alternating merges go from *x* to *aux* and from *aux* to *x*.

Mergesort also requires $O(n)$ additional space for the auxiliary array, whereas quicksort requires only $O(\log n)$ additional space for the stack. An algorithm has been developed for an in-place merge of two sorted subarrays in $O(n)$ time. This algorithm would allow mergesort to become an in-place $O(n \log n)$ sort. However, that technique does require a great deal many more assignments and would thus not be as practical as finding the $O(n)$ extra space.

There are two modifications of the foregoing procedure that can result in more efficient sorting. The first of these is the ***natural merge***. In the straight merge, the files are all the same size (except perhaps for the last file). We can, however, exploit any order that may already exist among the elements and let the subfiles be defined as the longest subarrays of increasing elements. You are asked to code such a routine as an exercise.

The second modification uses linked allocation instead of sequential allocation. By adding a single pointer field to each record, the need for the second array *aux* can be eliminated. This can be done by explicitly linking together each input and output subfile. The modification can be applied to both the straight merge and the natural merge. You are asked to implement these in the exercises.

Note that using mergesort on a linked list eliminates both of its drawbacks relative to quicksort: It no longer requires significant additional space and does not require significant data element movement. Generally, data elements can be large and complex, so that assignment of data elements requires more work than the reassignment of pointers that is still required by a list-based mergesort.

Mergesort can also be presented quite naturally as a recursive process in which the two halves of the array are first recursively sorted using mergesort and, once sorted, are joined by merging. For details, see Exercises 6.5.1 and 6.5.2. Both mergesort and quicksort are methods that involve splitting the file into two parts, sorting the two parts separately, and then joining the two sorted halves together. In mergesort, the splitting is trivial (simply taking two halves) and the joining is hard (merging the two sorted files). In quicksort, the splitting is hard (partitioning) and the joining is trivial (the two halves and the pivot automatically form a sorted array).

Insertion sort may be considered a special case of mergesort in which the two halves consist of a single element and the remainder of the array. Selection sort may be considered a special case of quicksort in which the file is partitioned into one half consisting of the largest element alone and a second half consisting of the remainder of the array.

The Cook–Kim Algorithm

Frequently, it is known that a file is almost sorted with only a few elements out of order. Or it may be known that an input file is likely to be sorted. For small files that are very nearly sorted or for sorted files, simple insertion is the fastest sort (considering both comparisons and assignments) that we have encountered. For large files or files that are slightly less sorted, quicksort using the middle element as pivot is fastest. (Considering only comparisons, mergesort is fastest.) However, another hybrid algorithm discovered by Cook and Kim is faster than both insertion sort and middle-element quicksort for nearly sorted input.

The Cook–Kim algorithm operates as follows: The input is examined for unordered pairs of elements (for example, $x[k] > x[k + 1]$). The two elements in an unordered pair are removed and added to the end of a new array. The next pair examined after an unordered pair is removed consists of the predecessor and successor of the removed pair. The original array, with the unordered pairs removed, is now in sorted order. The array of unordered pairs is then sorted using middle-element quicksort if it contains more than 30 elements, and simple insertion otherwise. The two arrays are then merged.

The Cook–Kim algorithm takes more advantage of the sortedness of the input than any other sorts and is significantly better than middle-element quicksort, insertion sort, merge sort, or Bsort on nearly sorted input. However, for randomly ordered input, Cook–Kim is less efficient than Bsort (and certainly than quicksort or merge sort). Middle-element quicksort, merge sort, or Bsort is therefore preferable when large sorted input files are likely but good random-input behavior is also required.

Radix Sort

The next sorting method that we consider is called the ***radix sort***. This sort is based on the values of the actual digits in the positional representations of the numbers being sorted. For example, the number 235 in decimal notation is written with a 2 in the hundreds position, a 3 in the tens position, and a 5 in the units position. The larger of two such integers of equal length can be determined as follows: Start at the most-significant digit and advance through the least-significant digits as long as the corresponding digits in the two numbers match. The number with the larger digit in the first position in which the digits of the two numbers do not match is the larger of the two numbers. Of course, if all the digits of both numbers match, the numbers are equal.

We can write a sorting routine based on the foregoing method. Using the decimal base, for example, the numbers can be partitioned into ten groups based on their most-significant digit. (For simplicity, we assume that all the numbers have the same number of digits, by padding with leading zeros, if necessary.) Thus every element in the "0" group is less than every element in the "1" group, all of whose elements are less than every element in the "2" group, and so on. We can then sort within the individual groups based on the next significant digit. We repeat this process until each subgroup has been subdivided so that the least-significant digits are sorted. At this point the original file has been sorted. (Note that the division of a subfile into groups with the same digit in a given position is similar to the *partition* operation in the quicksort, in which a subfile is divided into two groups based on comparison with a particular element.) This method is sometimes called the ***radix-exchange sort***; its coding is left as an exercise for the reader.

Let us now consider an alternative to the foregoing method. It is apparent from the foregoing discussion that considerable bookkeeping is involved in constantly subdividing files and distributing their contents into subfiles based on particular digits. It would certainly be easier if we could process the entire file as a whole rather than deal with many individual files.

Suppose that we perform the following actions on the file for each digit, beginning with the least-significant digit and ending with the most-significant digit. Take each number in the order in which it appears in the file and place it into one of ten queues, depending on the value of the digit currently being processed. Then restore each queue to the original file starting with the queue of numbers with a 0 digit and ending with the queue of numbers with a 9 digit. When these actions have been performed for each digit, starting with the least significant and ending with the most significant, the file is sorted. This sorting method is called the ***radix sort***.

Notice that this scheme sorts on the less-significant digits first. Thus when all the numbers are sorted on a more significant digit, numbers that have the same digit in that position but different digits in a less-significant position are already sorted on the less-significant position. This allows processing of the entire file without subdividing the files and keeping track of where each subfile begins and ends. Figure 6.5.2 illustrates this sort on the sample file

25 57 48 37 12 92 86 33

Be sure that you can follow the actions depicted in the two passes of Figure 6.5.2.

We can therefore outline an algorithm to sort in the foregoing fashion as follows:

```
for (k = least significant digit; k <= most significant digit; k++) {
   for (i = 0; i < n; i++) {
      y = x[i];
      j = kth digit of y;
      place y at rear of queue[j];
   } /* end for */
```

Original file

25 57 48 37 12 92 86 33

Queues based on least significant digit.

	Front	Rear
queue [0]		
queue [1]		
queue [2]	12	92
queue [3]	33	
queue [4]		
queue [5]	25	
queue [6]	86	
queue [7]	57	37
queue [8]	48	
queue [9]		

After first pass:

12 92 33 25 86 57 37 48

Queues based on most significant digit.

	Front	Rear
queue [0]		
queue [1]	12	
queue [2]	25	
queue [3]	33	37
queue [4]	48	
queue [5]	57	
queue [6]		
queue [7]		
queue [8]	86	
queue [9]	92	

Sorted file: 12 25 33 37 48 57 86 92

Figure 6.5.2 Illustration of the radix sort.

```
    for (qu = 0; qu < 10; qu++)
       place elements of queue[qu] in next sequential position of x;
} /* end for */
```

We now present a program to implement the foregoing sort on *m*-digit numbers. In order to save a considerable amount of work in processing the queues (especially in the step where we return the queue elements to the original file) we write the program using linked allocation. If the initial input to the routine is an array, that input is first converted into a linear linked list; if the original input is already in linked format, this step is not necessary and, in fact, space is saved. This is the same situation as in the routine *addr* (address calculation sort) of Section 6.4. As in previous programs, we do not make any internal calls to routines but rather perform their actions in place.

```
#define NUMELTS ...

void radixsort(int x[], int n)
{
   int front[10], rear[10];
   struct {
      int info;
      int next;
   } node[NUMELTS];
   int exp, first, i, j, k, p, q, y;

   /* Initialize linked list */
   for (i = 0; i < n-1; i++) {
      node[i].info = x[i];
      node[i].next = i+1;
   } /* end for */
   node[n-1].info = x[n-1];
   node[n-1].next = -1;
   first = 0;   /* first is the head of tne linked list */
   for (k = 1; k < 5; k++) {
      /* Assume we have four-digit numbers */
      for (i = 0; i < 10; i++) {
         /* Initialize queues */
         rear[i] = -1;
         front[i] = -1;
      } /* end for */
      /* Process each element on the list */
      while (first != -1) {
         p = first;
         first = node[first].next;
         y = node[p].info;
         /* Extract the kth digit */
         exp = power(10,k-1);          /* raise 10 to (k-1)th power */
         j = (y/exp)%10;
         /* Insert y into queue[j] */
         q = rear[j];
         if (q == -1)
            front[j] = p;
         else
            node[q].next = p;
         rear[j] = p;
      } /* end while */
      /* At this point each record is in its proper queue based on digit k. We now */
      /* form a single list from all the queue elements. Find the first element.   */
      for (j = 0; j < 10 && front[j] == -1; j++)
         ;
      first = front[j];
      /* Link up remaining queues */
      while (j <= 9) {   /* Check if finished */
         /* Find the next element */
         for (i = j+1; i < 10 && front[i] == -1; i++)
            ;
         if (i <= 9) {
            p = i;
            node[rear[j]].next = front[i];
         } /* end if */
         j = i;
      } /* end while */
      node[rear[p]].next = -1;
   } /* end for */
   /* Copy back to original array */
   for (i = 0; i < n; i++)  {
      x[i] = node[first].info;
      first = node[first].next;
   } /* end for */
} /* end radixsort */
```

The time requirements for the radix sorting method clearly depend on the number of digits (m) and the number of elements in the file (n). Since the outer loop *for* $(k = 1; k <= m; k++)$ is traversed m times (once for each digit) and the inner loop n times (once for each element in the file), the sort is approximately $O(m * n)$. Thus the sort is reasonably efficient if the number of digits in the keys is not too large. It should be noted, however that many machines have the hardware facilities to order digits of a number (particularly if they are in binary) much more rapidly than they can execute a compare of two full keys. Therefore it is not reasonable to compare the $O(m * n)$ estimate with some of the other results we arrived at in this chapter. Note also that if the keys are dense (that is, if almost every number that can possibly be a key is actually a key), m approximates log n, so that $O(m * n)$ approximates $O(n \log n)$. The sort does require space to store pointers to the fronts and rears of the queues in addition to an extra field in each record to be used as a pointer in the linked lists. If the number of digits is large it is sometimes more efficient to sort the file by first applying the radix sort to the most-significant digits and then using straight insertion on the rearranged file. In cases where most of the records in the file have differing most-significant digits, this process eliminates wasteful passes on the least-significant digits.

EXERCISES

6.5.1. Write an algorithm for a routine *merge*(x, $lb1$, $ub1$, $ub2$) that assumes that $x[lb1]$ through $x[ub1]$ and $x[ub1 + 1]$ through $x[ub2]$ are sorted and merges the two into $x[lb1]$ through $x[ub2]$.

6.5.2. Consider the following recursive version of the merge sort that uses the routine *merge* of Exercise 6.5.1. It is initially called by *msort2*(x, 0, $n - 1$). Rewrite the routine by eliminating recursion and simplifying. How does the resulting routine differ from the one in the text?

```
void msort2(int x[], int lb, int ub)
{
   if (lb != ub) {
      mid = (ub+lb)/2;
      msort2(x, lb, mid);
      msort2(x, mid+1, ub);
      merge(x, lb, mid, ub);
   } /* end if */
} /* end msort2 */
```

6.5.3. Let $a(l1, l2)$ be the average number of comparisons necessary to merge two sorted arrays of length $l1$ and $l2$, respectively, where the elements of the arrays are chosen at random from among $l1 + l2$ elements.

(a) What are the values of $a(l1,0)$ and $a(0,l2)$?

(b) Show that for $l1 > 0$ and $l2 > 0$, $a(l1,l2)$ is equal to $(l1/(l1 + l2))*(1 + a(l1 - 1, l2)) + (l2/(l1 + l2))*(1 + a(l1,l2 - 1))$. (*Hint:* Express the average number of comparisons in terms of the average number of comparisons after the first comparison.)

(c) Show that $a(l1,l2)$ equals $(l1*l2*(l1 + l2 + 2))/((l1 + 1)*(l2 + 1))$.

(d) Verify the formula in part c for two arrays, one of size 2 and one of size 1.

6.5.4. Consider the following method of merging two arrays *a* and *b* into *c*: Perform a binary search for *b*[0] in the array *a*. If *b*[0] is between *a*[*i*] and *a*[*i* + 1], output *a*[1] through *a*[*i*] to the array *c*, then output *b*[0] to the array *c*. Next perform a binary search for *b*[1] in the subarray *a*[*i* + 1] to *a*[*la*] (where *la* is the number of elements in the array *a*) and repeat the output process. Repeat this procedure for every element of the array *b*.

(a) Write a C routine to implement this method.

(b) In which cases is this method more efficient than the method of the text? In which cases is it less efficient?

6.5.5. Consider the following method (called ***binary merging***) of merging two sorted arrays *a* and *b* into *c*: Let *la* and *lb* be the number of elements of *a* and *b*, respectively, and assume that *la* >= *lb*. Divide *a* into *lb* + 1 approximately equal subarrays. Compare *b*[0] with the smallest element of the second subarray of *a*. If *b*[0] is smaller, find *a*[*i*] such that *a*[*i*] <= *b*[0] <= *a*[*i* + 1] by a binary search in the first subarray. Output all elements of the first subarray up to and including *a*[*i*] into *c*, and then output *b*[0] into *c*. Repeat this process with *b*[1], *b*[2], . . . , *b*[*j*], where *b*[*j*] is found to be larger than the smallest element of the second subarray. Output all remaining elements of the first subarray and the first element of the second subarray into *c*. Then compare *b*[*j*] with the smallest element of the third subarray of *a*, and so on.

(a) Write a program to implement the binary merge.

(b) Show that if *la* = *lb*, the binary merge acts like the merge described in the text.

(c) Show that if *lb* = 1, the binary merge acts like the merge of the previous exercise.

6.5.6. Determine the number of comparisons (as a function of *n* and *m*) that are performed in merging two ordered files *a* and *b* of sizes *n* and *m*, respectively, by each of the following merge methods, on each of the following sets of ordered files:

Merge Methods:

(a) the merge method presented in the text

(b) the merge of Exercise 6.5.4

(c) the binary merge of Exercise 6.5.5

Sets of Files:

(a) $m = n$ and $a[i] < b[i] < a[i + 1]$ for all i

(b) $m = n$ and $a[n] < b[1]$

(c) $m = n$ and $a[n/2] < b[1] < b[m] < a[(n/2) + 1]$

(d) $n = 2 * m$ and $a[i] < b[i] < a[i + 1]$ for all i between 0 and $m - 1$

(e) $n = 2 * m$ and $a[m + i] < b[i] < a[m + i + 1]$ for all i between 0 and $m - 1$

(f) $n = 2 * m$ and $a[2 * i] < b[i] < a[2 * i + 1]$ for all i between 0 and $m - 1$

(g) $m = 1$ and $b[0] = a[n/2]$

(h) $m = 1$ and $b[0] < a[0]$

(i) $m = 1$ and $a[n] < b[0]$

6.5.7. Generate two random sorted files of size 100 and merge them by each of the methods of the previous exercise, keeping track of the number of comparisons made. Do the same for two files of size 10 and two files of size 1000. Repeat the experiment ten times. What do the results indicate about the average efficiency of the merge methods?

6.5.8. Write a routine that sorts a file by first applying the radix sort to the most significant r digits (where r is a given constant) and then uses straight insertion to sort the entire file. This eliminates excessive passes on low-order digits that may not be necessary.

6.5.9. Write a program that prints all sets of six positive integers $a1$, $a2$, $a3$, $a4$, $a5$, and $a6$ such that

$$a1 <= a2 <= a3 <= 20$$

$$a1 < a4 <= a5 <= a6 <= 20$$

and the sum of the squares of $a1$, $a2$, and $a3$ equals the sum of the squares of $a4$, $a5$, and $a6$. (*Hint:* Generate all possible sums of three squares, and use a sorting procedure to find duplicates.)