(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 −1 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 placed 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.

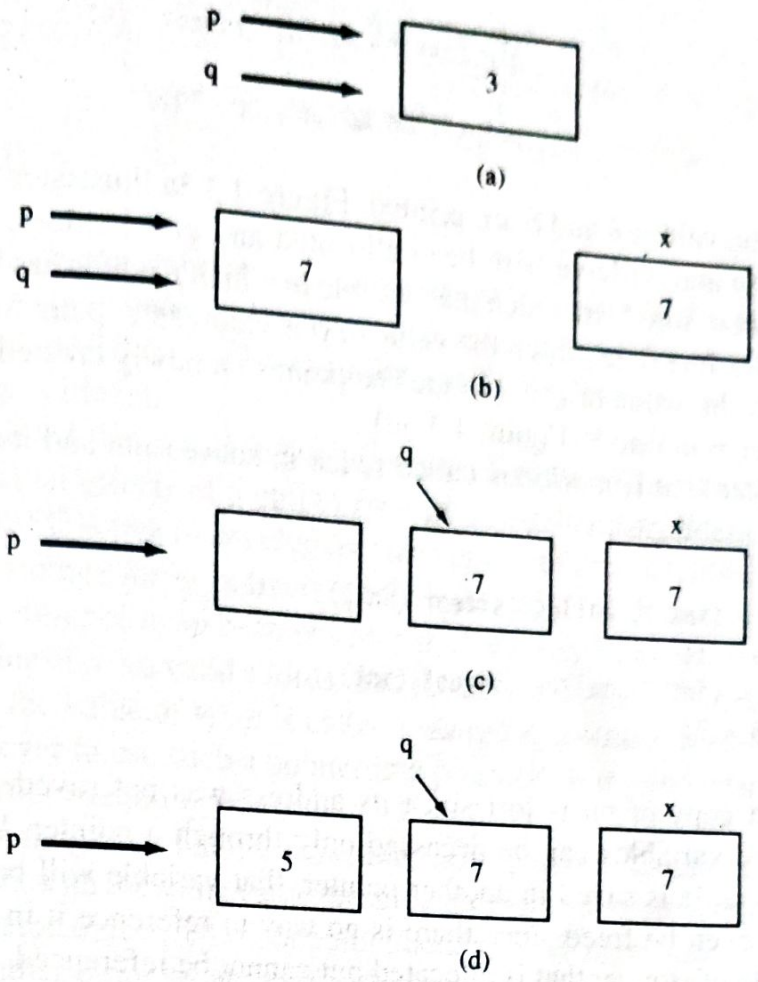


Figure 4.3.2

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:

```
1        p = (int *) malloc (sizeof (int));
2        *p = 5;
3        q = (int *) malloc (sizeof (int));
4        *q = 8;
5        free(p);
6        p = q;
```

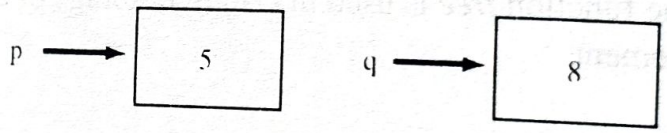
```
                q = (int *) malloc (sizeof (int));
7               *q = 6;
8               printf("%d %d \n", *p  *q);
9
```

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).

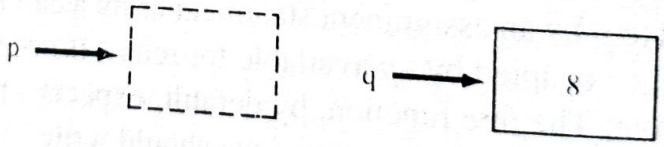
Note that if *malloc* is called twice in succession and its value is assigned to the same variable, as in:

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

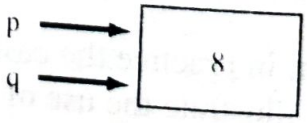
the first copy of $*p$ is lost since its address was not saved. The space allocated for dynamic variables can be accessed only through a pointer. Unless the pointer to the first variable is saved in another pointer, that variable will be lost. In fact, its storage cannot even be freed since there is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



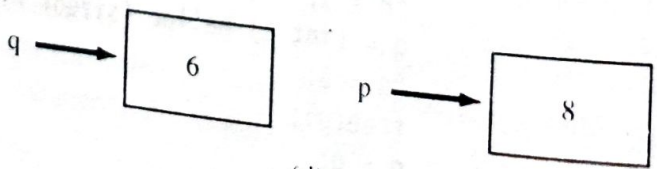
(a)



(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

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

Dynamic Implementation

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

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

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

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

```
void insert(struct queue *pq, int x)
{
   NODEPTR p;

   p = getnode();
   p->info = x;
   p->next = NULL;
   if (pq->rear == NULL)
      pq->front = p;
   else
      (pq->rear)->next = p;
   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 p, x;

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

```
int remove(struct queue *pq)
{
   NODEPTR p;
   int x;
   if (empty(pq)) {
      printf("queue underflow\n");
      exit(1);
   }
   p = pq->front;
   x = p->info;
   pq->front = p->next;
   if (pq->front == NULL)
      pq->rear = NULL;
   freenode(p);
   return(x);
} /* 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 into a 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 imple-

mentation 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 *s1* and *s2* 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, *s1* and *s2* 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.