CHAPTER 4

Linked Lists

4.1 SINGLY LINKED LISTS AND CHAINS

In the previous chapters, we studied the representation of simple data structures using an array and a sequential mapping. These representations had the property that successive nodes of the data object were stored a fixed distance apart. Thus, (1) if the element a_{ij} of a table was stored at location L_{ij}, then $a_{i,j+1}$ was at the location L_{ij} + 1; (2) if the ith element in a queue was at location L_i, then the $(i + 1)$th element was at location $(L_i + 1)$ % n for the circular representation; (3) if the topmost node of a stack was at location L_T, then the node beneath it was at location $L_T - 1$, and so on. These sequential storage schemes proved adequate for the tasks we wished to perform (accessing an arbitrary node in a table, insertion or deletion of stack and queue elements). However, when a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive. For example, consider the following list of three-letter English words ending in AT:

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)

To make this list more complete we naturally want to add the word GAT, which means

gun or revolver. If we are using an array and a sequential mapping to keep this list, then the insertion of GAT will require us to move elements already in the list either one location higher or lower. We must move either HAT, JAT, LAT, $\cdots$, WAT or BAT, CAT, EAT, and FAT. If we have to do many such insertions into the middle, neither alternative is attractive because of the amount of data movement. Excessive data movement also is required for deletions. Suppose we decide to remove the word LAT, which refers to the Latvian monetary unit. Then again, we have to move many elements so as to maintain the sequential representation of the list.

An elegant solution to this problem of data movement in *sequential* representations is achieved by using *linked* representations. Unlike a sequential representation, in which successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. In other words, in a sequential representation the order of elements is the same as in the ordered list, whereas in a linked representation these two sequences need not be the same. To access list elements in the correct order, with each element we store the address or location of the next element in that list. Thus, associated with each data item in a linked representation is a pointer or link to the next item. In general, a linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

Figure 4.1 shows how some of the elements in our list of three-letter words may be represented in memory by using pointers. The elements of the list are stored in a one-dimensional array called *data*, but the elements no longer occur in sequential order, BAT before CAT before EAT, and so on. Instead we relax this restriction and allow them to appear anywhere in the array and in any order. To remind us of the real order, a second array, *link*, is added. The values in this array are pointers to elements in the *data* array. For any i, *data* [i] and *link* [i] together comprise a node. Since the list starts at *data*[8] = BAT, let us set a variable *first* = 8. *link*[8] has the value 3, which means it points to *data*[3], which contains CAT. Since *link* [3] = 4, the next element, EAT, in the list is in *data* [4]. The element after EAT is in *data* [*link* [4]]. By continuing in this way we can list all the words in the proper order. We recognize that we have come to the end of our ordered list when *link* equals zero. To ensure that a *link* of zero always signifies the end of a list, we do not use position zero of *data* to store a list element.

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows, as in Figure 4.2. Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. The arrows reinforce in our own mind the facts that (1) the nodes do not actually reside in sequential locations and (2) the actual locations of nodes are immaterial. Therefore, when we write a program that works with lists, we do not look for a specific address except when we test for zero. The linked structures of Figures 4.1 and 4.2 are called singly linked lists or chains. In a *singly linked list*, each node has exactly one pointer field. A *chain* is a singly linked list that is comprised of zero or more nodes. When the number of nodes is zero, the chain is empty. The nodes of a non-empty chain are ordered so that the first node links to the second node; the second to the third; and so on. The last node of a chain has

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.
	.	.

Figure 4.1: Nonsequential list-representation

a 0 link.

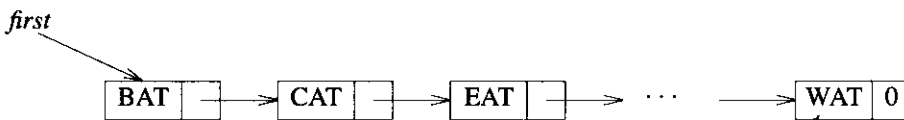


Figure 4.2: Usual way to draw a linked list

Let us now see why it is easier to make insertions and deletions at arbitrary positions using a linked list rather than a sequential list. To insert the data item GAT between FAT and HAT, the following steps are adequate:

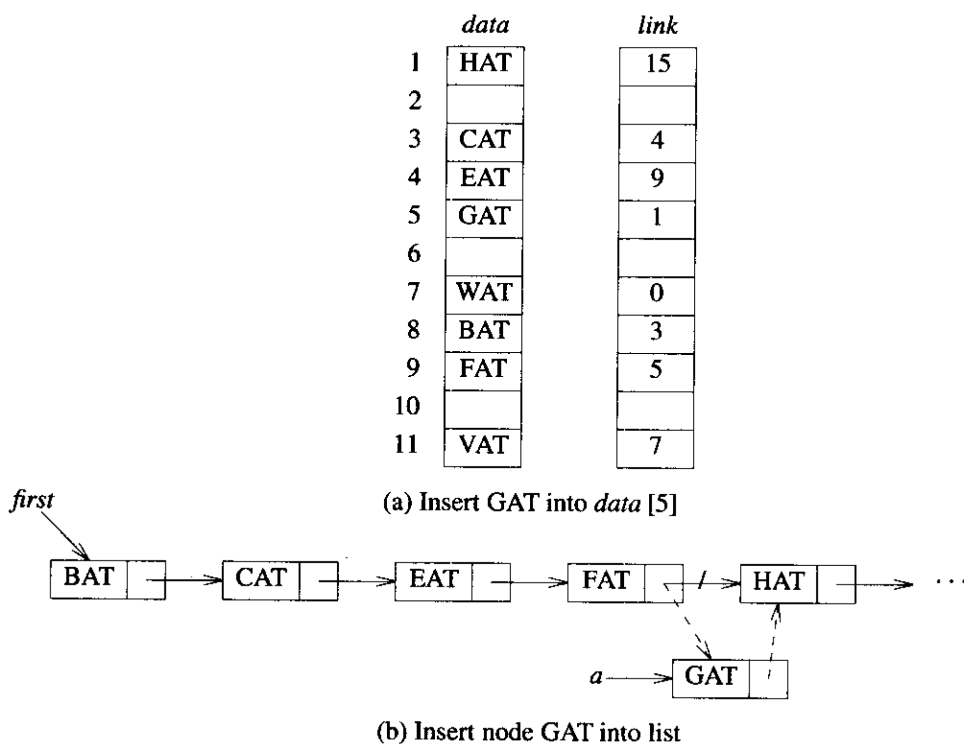
(1) Get a node *a* that is currently unused.

(2) Set the *data* field of *a* to GAT.

(3) Set the *link* field of *a* to point to the node after FAT, which contains HAT.

(4) Set the *link* field of the node containing FAT to *a*.

Figure 4.3(a) shows how the arrays *data* and *link* will be changed after we insert GAT. Figure 4.3(b) shows how we can draw the insertion using our arrow notation. Dashed arrows are new ones. The important thing to notice is that when we insert GAT, we do not have to move any elements that are already in the list. We have overcome the need to move data at the expense of the storage needed for the field *link*. Usually, this penalty is not too severe. When each list element is large, significant time is saved by not having to move elements during an insert or delete.



	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

(a) Insert GAT into *data* [5]

(b) Insert node GAT into list

Figure 4.3: Inserting into a linked list

Now suppose we want to delete GAT from the list. All we need to do is find the element that immediately precedes GAT, which is FAT, and set *link*[9] to the position of HAT which is 1. Again, there is no need to move the data around. Even though the link

of GAT still contains a pointer to HAT, GAT is no longer in the list as it cannot be reached by starting at the first element of list and following links (see Figure 4.4).

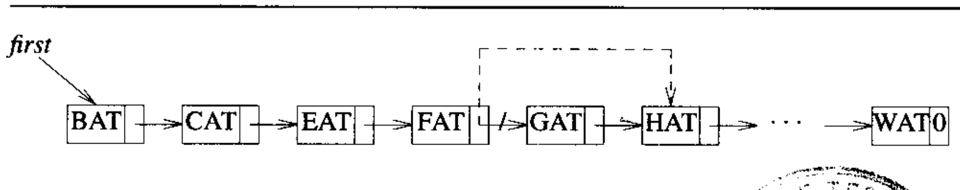


Figure 4.4: Delete GAT

4.2 REPRESENTING CHAINS IN C

We need the following capabilities to make linked representations possible:

(1) A mechanism for defining a node's structure, that is, the fields it contains. We use *self-referential structures*, discussed in Section 2.3.4, to do this.

(2) A way to create new nodes when we need them. The *MALLOC* macros defined in Section 1.2.2 handles this operation.

(3) A way to remove nodes that we no longer need. The *free* function handles this operation.

We will present several small examples to show how to create and use linked lists in C.

Example 4.1 [***List of words***]**:** To create a linked list of words, we first define a node structure for the list. This structure specifies the type of each of the fields. From our previous discussion we know that our structure must contain a character array and a pointer to the next node. The necessary declarations are:

```
typedef struct listNode *listPointer;
typedef struct {
        char data[4];
        listPointer link;
        } listNode;
```

These declarations contain an example of a *self-referential structure*. Notice that we have defined the pointer (*listPointer*) to the **struct** before we defined the **struct** (*listNode*). C allows us to create a pointer to a type that does not yet exist because otherwise we would face a paradox: we cannot define a pointer to a nonexistent type, but to define

the new type we must include a pointer to the type.

After defining the node's structure, we create a new empty list. This is accomplished by the statement:

```
listPointer first = NULL;
```

This statement indicates that we have a new list called *first*. Remember that *first* contains the address of the start of the list. Since the new list is initially empty, its starting address is zero. Therefore, we use the reserved word *NULL* to signify this condition. We also can use an *IS_EMPTY* macro to test for an empty list:

```
#define IS_EMPTY(first) (!(first))
```

To create new nodes for our list we use the *MALLOC* macro of Section 1.2.2. We would apply this macro as follows to obtain a new node for our list:

```
MALLOC(first, sizeof(*first));
```

We are now ready to assign values to the fields of the node. This introduces a new operator, →. If *e* is a pointer to a structure that contains the field *name*, then *e→name* is a shorthand way of writing the expression (**e*).*name*. The → operator is referred to as the *structure member* operator, and its use is preferred when one has a pointer to a **struct** rather than the * and dot notation.

To place the word BAT into our list we use the statements:

```
strcpy(first→data,"BAT");
first→link = NULL;
```

These statements create the list illustrated in Figure 4.5. Notice that the node has a null link field because there is no next node in the list. □

Example 4.2 [***Two-node linked list***]: We want to create a linked list of integers. The node structure is defined as:

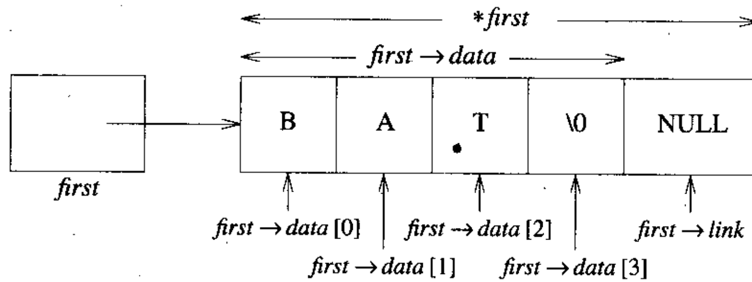


Figure 4.5: Referencing the fields of a node

```
typedef struct listNode *listPointer;
typedef struct {
        int data;
        listPointer link;
        } listNode;
```

A linked list with two nodes is created by function *create2* (Program 4.1). We set the data field of the first node to 10 and that of the second to 20. The variable *first* is a pointer to the first node; *second* is a pointer to the second node. Notice that the link field of the first node is set to point to the second node, while the link field of the second node is *NULL*. The variable *first*, which is the pointer to the start of the list, is returned by *create2*. Figure 4.6 shows the resulting list structure. □

Example 4.3 [***List insertion***]: Let *first* be a pointer to a linked list as in Example 4.2. Assume that we want to insert a node with a data field of 50 after some arbitrary node *x*. Function *insert* (Program 4.2) accomplishes this task. In this function, we pass in two pointer variables. The variable, *first*, is the pointer to the first node in the list. If this variable contains a null address (i.e., there are no nodes in the list), we want to change *first* so that it points to the node with 50 in its data field. This means that we must pass in the address of *first*. This is why we use the declaration *listPointer *first*. Since the value of the second pointer, *x*, does not change, we do not need to pass in its address as a parameter. A typical function call would be *insert* (*&first*, *x*); where *first* points to the start of the list and *x* points to the node after which the insertion is to take place.

The function *insert* uses an **if** ··· **else** statement to distinguish between empty and nonempty lists. For an empty list, we set *temp*'s link field to *NULL* and change the value of *first* to the address of *temp*. For a nonempty list, we insert the *temp* node between *x*

```
listPointer create2()
{/* create a linked list with two nodes */
   listPointer first, second;
   MALLOC(first, sizeof(*first));
   MALLOC(second, sizeof(*second));
   second→link = NULL;
   second→data = 20;
   first→data = 10;
   first→link = second;
   return first;
}
```

Program 4.1: Create a two-node list

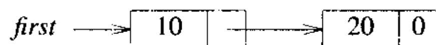


Figure 4.6: A two-node list

and the node pointed to by its link field. Figure 4.7 shows the two cases.

Example 4.4 [*List deletion*]: Deleting an arbitrary node from a list is slightly more complicated than insertion because deletion depends on the location of the node. Assume that we have three pointers: *first* points to the start of the list, *x* points to the node that we wish to delete, and *trail* points to the node that precedes *x*. Figures 4.8 and 4.9 show two examples. In Figure 4.8, the node to be deleted is the first node in the list. This means that we must change the value of *first*. In Figure 4.9, since *x* is not the first node, we simply change the link field in *trail* to point to the link field in *x*.

An arbitrary node is deleted from a linked list by function *delete* (Program 4.3). In addition to changing the link fields, or the value of **first*, *delete* also returns the space that was allocated to the deleted node to the system memory. To accomplish this task, we use *free*. □

Example 4.5 [*Printing out a list*]: Program 4.4 prints the data fields of the nodes in a list. To do this we first print out the contents of *first*'s data field, then we replace *first* with the address in its *link* field. We continue printing out the *data* field and moving to

```
void insert(listPointer *first, listPointer x)
{/* insert a new node with data = 50 into the chain
    first after node x */
   listPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = 50;
   if (*first) {
      temp→link = x→link;
      x→link = temp;
   }
   else {
      temp→link = NULL;
      *first = temp;
   }
}
```

Program 4.2: Simple insert into front of list

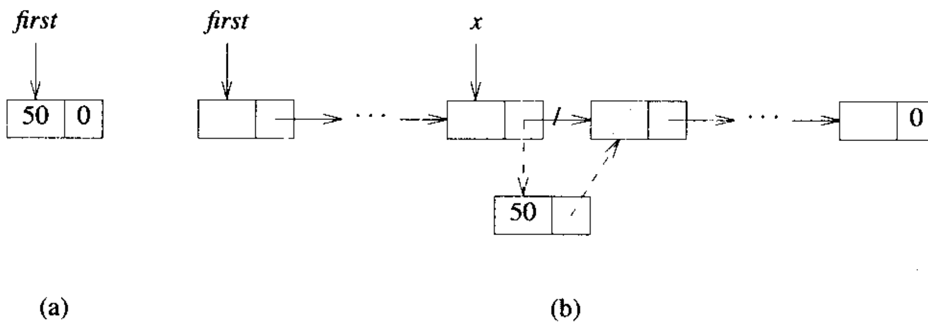


Figure 4.7: Inserting into an empty and nonempty list

the next node until we reach the end of the list. □

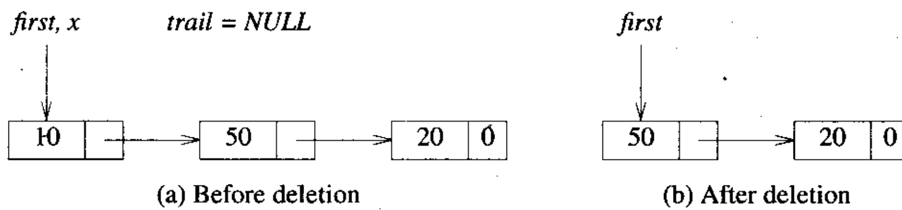


Figure 4.8: List before and after the function call *delete(&first, NULL, first);*

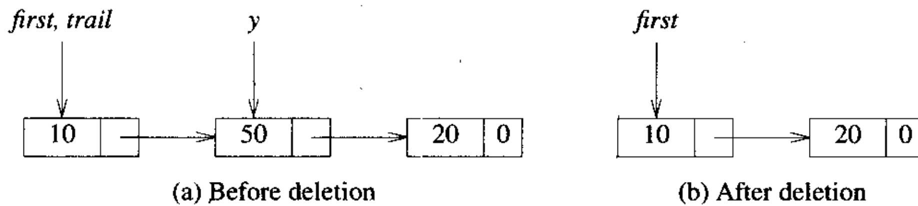


Figure 4.9: List after the function call *delete(&first, y, y→link);*

EXERCISES

1. Rewrite *delete* (Program 4.3) so that it uses only two pointers, *first* and *trail*.
2. Assume that we have a list of integers as in Example 4.2. Create a function that searches for an integer, *num*. If *num* is in the list, the function should return a pointer to the node that contains *num*. Otherwise it should return *NULL*.
3. Write a function that deletes a node containing a number, *num*, from a list. Use the search function (Exercise 2) to determine if *num* is in the list.
4. Write a function, *length*, that returns the number of nodes in a list.
5. Let p be a pointer to the first node in a singly linked list. Write a procedure to delete every other node beginning with node p (i.e., the first, third, fifth, etc. nodes of the list are deleted). What is the time complexity of your algorithm?
6. Let $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_m)$ be two linked lists. Assume that in each list, the nodes are in nondecreasing order of their data field values. Write an algorithm to merge the two lists together to obtain a new linked list z in which

```
void delete(listPointer *first, listPointer trail,
                                 listPointer x)
{/* delete x from the list, trail is the preceding node
    and *first is the front of the list */
   if (trail)
      trail->link = x->link;
   else
      *first = (*first)->link;
   free(x);
}
```

Program 4.3: Deletion from a list

```
void printList(listPointer first)
{
   printf("The list contains: ");
   for (; first; first = first->link)
      printf("%4d",first->data);
   printf("\n");
}
```

Program 4.4: Printing a list

the nodes are also in this order. Following the merge, x and y do not exist as individual lists. Each node initially in x or y is now in z. No additional nodes may be used. What is the time complexity of your algorithm?

7. Let $list_1 = (x_1, x_2, \cdots, x_n)$ and $list_2 = (y_1, y_2, \cdots, y_m)$. Write a function to merge the two lists together to obtain the linked list, $list_3 = (x_1, y_1, x_2, y_2, \cdots, x_m, y_m, x_{m+1}, \cdots, x_n)$ if $m \leq n$; and $list_3 = (x_1, y_1, x_2, y_2, \cdots, x_n, y_n, x_{n+1}, \cdots, x_m)$ if $m > n$.

8. § It is possible to traverse a linked list in both directions (i.e., left-to right and restricted right-to-left) by reversing the links during the left-to-right traversal. A possible configuration for the list under this scheme is given in Figure 4.10. The variable r points to the node currently being examined and l to the node on its left. Note that all nodes to the left of r have their links reversed.

   (a) Write a function to move r to the right n nodes from a given position (l, r).

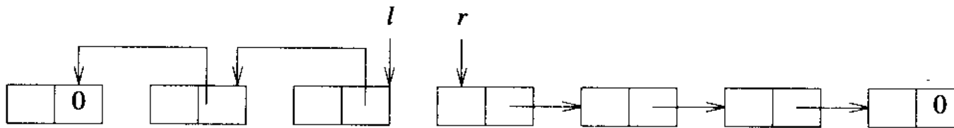


Figure 4.10: Possible configuration for a chain traversed in both directions

(b) Write a function to move *r* to the left *n* nodes from any given position *(l, r)*.

4.3 LINKED STACKS AND QUEUES

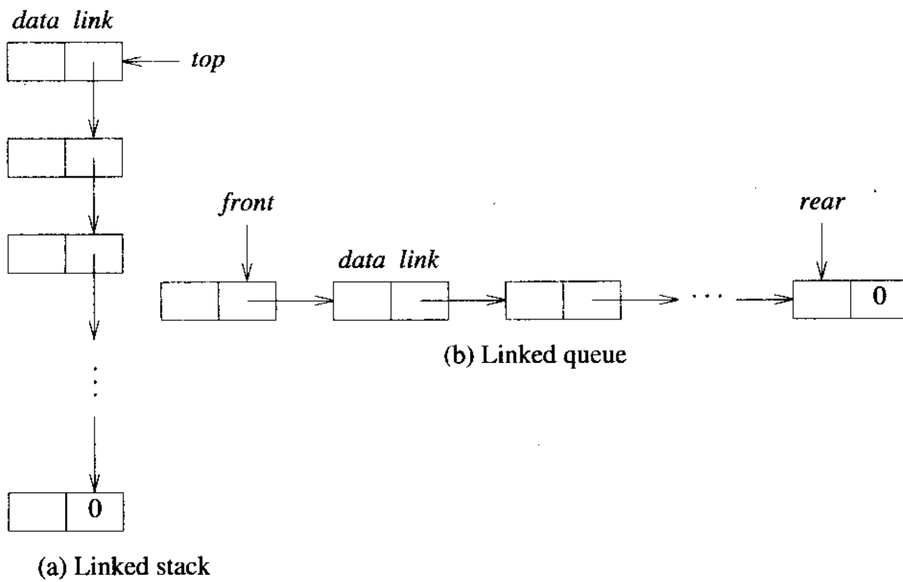
Previously we represented stacks and queues sequentially. Such a representation proved efficient if we had only one stack or one queue. However, when several stacks and queues coexisted, there was no efficient way to represent them sequentially. Figure 4.10 shows a linked stack and a linked queue. Notice that the direction of links for both the stack and the queue facilitate easy insertion and deletion of nodes. In the case of Figure 4.10(a), we can easily add or delete a node from the top of the stack. In the case of Figure 4.11(b), we can easily add a node to the rear of the queue and add or delete a node at the front, although we normally will not add items to the front of a queue.

If we wish to represent $n \leq MAX_STACKS$ stacks simultaneously, we begin with the declarations:

```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
        int key;
        /* other fields */
        } element;
typedef struct stack *stackPointer;
typedef struct {
        element data;
        stackPointer link;
        } stack;
stackPointer top[MAX_STACKS];
```

We assume that the initial condition for the stacks is:

$$top[i] = NULL,\ 0 \leq i < MAX_STACKS$$



(a) Linked stack

(b) Linked queue

Figure 4.11: Linked stack and queue

and the boundary condition is:

$$top[i] = NULL \text{ iff the } i\text{th stack is empty}$$

Functions *push* (Program 4.5) and *pop* (Program 4.6) add and delete items to/from a stack. The code for each is straightforward. Function *push* creates a new node, *temp*, and places *item* in the data field and *top* in the link field. The variable *top* is then changed to point to *temp*. A typical function call to add an element to the *i*th stack would be *push*(*i*,*item*). Function *pop* returns the top element and changes *top* to point to the address contained in its link field. The removed node is then returned to system memory. A typical function call to delete an element from the *i*th stack would be *item* = *pop*(*i*);

To represent $m \leq MAX_QUEUES$ queues simultaneously, we begin with the declarations:

```
void push(int i, element item)
{/* add item to the ith stack */
   stackPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = item;
   temp→link = top[i];
   top[i] = temp;
}
```

Program 4.5: Add to a linked stack

```
element pop(int i)
{/* remove top element from the ith stack */
   stackPointer temp = top[i];
   element item;
   if (!temp)
     return stackEmpty();
   item = temp→data;
   top[i] = temp→link;
   free(temp);
   return item;
}
```

Program 4.6: Delete from a linked stack

```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queuePointer;
typedef struct {
        element data;
        queuePointer link;
        } queue;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

We assume that the initial condition for the queues is:

$$front[i] = NULL, 0 \leq i < MAX_QUEUES$$

and the boundary condition is:

front[*i*] = NULL iff the *i*th queue is empty

Functions *addq* (Program 4.7) and *deleteq* (Program 4.8) implement the add and delete operations for multiple queues. Function *addq* is more complex than *push* because we must check for an empty queue. If the queue is empty, we change *front* to point to the new node; otherwise we change *rear*'s link field to point to the new node. In either case, we then change *rear* to point to the new node. Function *deleteq* is similar to *pop* since we are removing the node that is currently at the start of the list. Typical function calls would be *addq* (*i*,*item*); and *item* = *deleteq* (*i*);.

```
void addq(i, item)
{/* add item to the rear of queue i */
   queuePointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→data = item;
   temp→link = NULL;
   if (front[i])
      rear[i]→link = temp;
   else
      front[i] = temp;
   rear[i] = temp;
}
```

Program 4.7: Add to the rear of a linked queue

The solution presented above to the *n*-stack, *m*-queue problem is both computationally and conceptually simple. We no longer need to shift stacks or queues to make space. Computation can proceed as long as there is memory available. Although we need additional space for the link field, the use of linked lists makes sense because the overhead incurred by the storage of the links is overridden by (1) the ability to represent lists in a simple way, and (2) the reduced computing time required by linked representations.

EXERCISES

1. A palindrome is a word or phrase that is the same when spelled from the front or the back. For example, "reviver" and "Able was I ere I saw Elba" are both palindromes. We can determine if a word or phrase is a palindrome by using a stack. Write a C function that returns *TRUE* if a word or phrase is a palindrome and

```
element deleteq(int i)
{/* delete an element from queue i */
   queuePointer temp = front[i];
   element item;
   if (!temp)
      return queueEmpty();
   item = temp->data;
   front[i]= temp->link;
   free(temp);
   return item;
}
```

Program 4.8: Delete from the front of a linked queue

FALSE if it is not.

2. We can use a stack to determine if the parentheses in an expression are properly nested. Write a C function that does this.
3. Consider the hypothetical data type $X2$. $X2$ is a linear list with the restriction that while additions to the list may be made at either end, deletions can be made at one end only. Design a linked list representation for $X2$. Write addition and deletion functions for $X2$. Specify initial and boundary conditions for your representation.

4.4 POLYNOMIALS

4.4.1 Polynomial Representation

Let us tackle a reasonably complex problem using linked lists. This problem, the manipulation of symbolic polynomials, has become a classic example of list processing. As in Chapter 2, we wish to be able to represent any number of different polynomials as long as memory is available. In general, we want to represent the polynomial:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0x^{e_0}$$

where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that $e_{m-1} > e_{m-2} > \cdots > e_1 > e_0 \geq 0$. We represent each term as a node containing coefficient and exponent fields, as well as a pointer to the next term. Assuming that the coefficients are integers, the type declarations are:

```
typedef struct polyNode *polyPointer;
typedef struct {
        int coef;
        int expon;
        polyPointer link;
        } polyNode;
polyPointer a,b;
```

We draw *polyNode*s as:

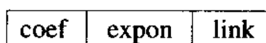


Figure 4.12 shows how we would store the polynomials

$$a = 3x^{14} + 2x^8 + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^6$$

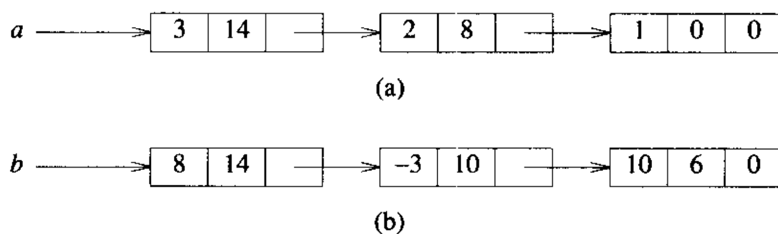


Figure 4.12: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

4.4.2 Adding Polynomials

To add two polynomials, we examine their terms starting at the nodes pointed to by a and b. If the exponents of the two terms are equal, we add the two coefficients and create a new term for the result. We also move the pointers to the next nodes in a and b. If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, attach this term to the result, called c, and advance the pointer to the next term in b. We take a similar action on a if $a \rightarrow expon > b \rightarrow expon$. Figure 4.13 illustrates this process for the polynomials represented in Figure 4.12.

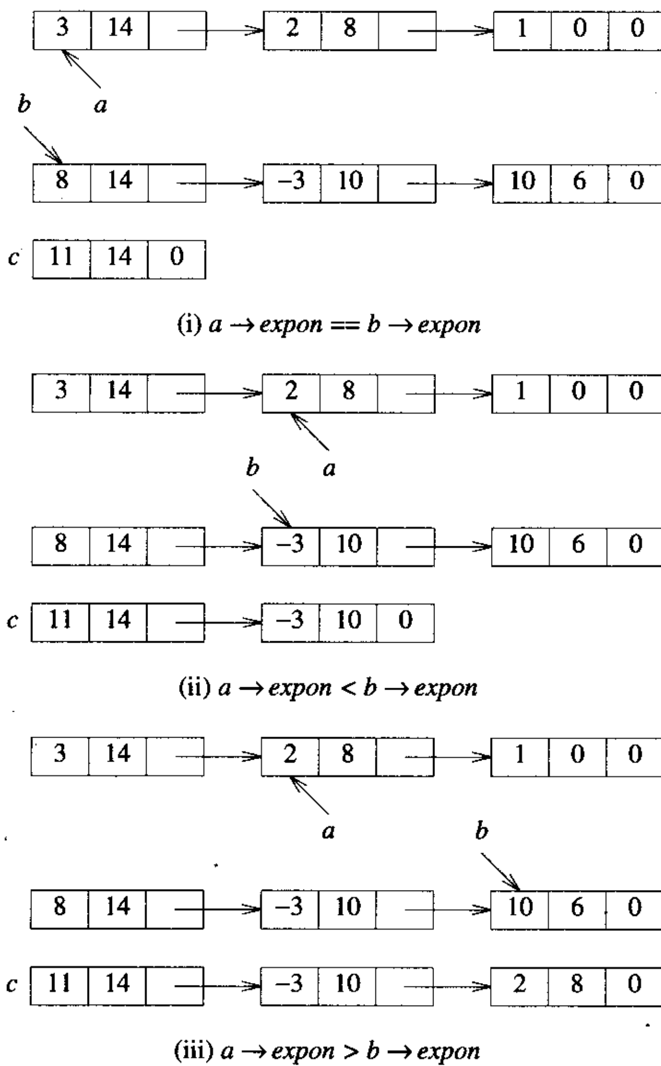


Figure 4.13: Generating the first three terms of $c = a + b$

Each time we generate a new node, we set its *coef* and *expon* fields and append it to the end of *c*. To avoid having to search for the last node in *c* each time we add a new node, we keep a pointer, *rear*, which points to the current last node in *c*. The complete addition algorithm is specified by *padd* (Program 4.9). To create a new node and append

it to the end of *c*, *padd* uses *attach* (Program 4.10). To make things work out neatly, initially we give *c* a single node with no values, which we delete at the end of the function. Although this is somewhat inelegant, it avoids more computation.

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
      switch (COMPARE(a→expon,b→expon)) {
         case -1: /* a→expon < b→expon */
               attach(b→coef,b→expon,&rear);
               b = b→link;
               break;
         case 0: /* a→expon = b→expon */
               sum = a→coef + b→coef;
               if (sum) attach(sum,a→expon,&rear);
               a = a→link;  b = b→link; break;
         case 1: /* a→expon > b→expon */
               attach(a→coef,a→expon,&rear);
               a = a→link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a→link) attach(a→coef,a→expon,&rear);
   for (; b; b = b→link) attach(b→coef,b→expon,&rear);
   rear→link = NULL;
   /* delete extra initial node */
   temp = c; c = c→link;  free(temp);
   return c;
}
```

Program 4.9: Add two polynomials

This is our first complete example of list processing, so you should study it carefully. The basic algorithm is straightforward, using a streaming process that moves along the two polynomials, either copying terms or adding them to the result. Thus, the **while** loop has three cases depending on whether the next pair of exponents are =, <, or >. Notice that there are five places where we create a new term, justifying our use of function *attach*.

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
  polyPointer temp;
  MALLOC(temp, sizeof(*temp));
  temp→coef = coefficient;
  temp→expon = exponent;
  (*ptr)→link = temp;
  *ptr = temp;
}
```

Program 4.10: Attach a node to the end of a list

Analysis of *padd*: To determine the computing time of *padd*, we first determine which operations contribute to the cost. For this algorithm, there are three cost measures:

(1) coefficient additions

(2) exponent comparisons

(3) creation of new nodes for c

If we assume that each of these operations takes a single unit of time if done once, then the number of times that we perform these operations determines the total time taken by *padd*. This number clearly depends on how many terms are present in the polynomials a and b. Assume that a and b have m and n terms, respectively:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

$$B(x) = b_{n-1}x^{f_{n-1}} + \cdots + b_0 x^{f_0}$$

where $a_i, b_i \neq 0$ and $e_{m-1} > \cdots > e_0 \geq 0, f_{n-1} > \cdots > f_0 \geq 0$. Then clearly the number of coefficient additions varies as:

$$0 \leq \text{number of coefficient additions} \leq \min\{m,n\}$$

The lower bound is achieved when none of the exponents are equal, while the upper is achieved when the exponents of one polynomial are a subset of the exponents of the other.

As for the exponent comparisons, we make one comparison on each iteration of the **while** loop. On each iteration, either a or b or both move to the next term. Since the total number of terms is $m + n$, the number of iterations and hence the number of exponent comparisons is bounded by $m + n$. You can easily construct a case when $m + n - 1$ comparisons will be necessary, for example, $m = n$ and

$$e_{m-1} > f_{m-1} > e_{m-2} > f_{m-2} > \cdots > e_1 > f_1 > e_0 > f_0$$

The maximum number of terms in c is $m + n$, and so no more than $m + n$ new terms are created (this excludes the additional node that is attached to the front of d and later removed).

In summary, the maximum number of executions of any statement in *padd* is bounded above by $m + n$. Therefore, the computing time is $O(m + n)$. This means that if we implement and run the algorithm on a computer, the time it takes will be $c_1 m + c_2 n + c_3$, where c_1, c_2, c_3 are constants. Since any algorithm that adds two polynomials must look at each nonzero term at least once, *padd* is optimal to within a constant factor. □

4.4.3 Erasing Polynomials

The use of linked lists is well suited to polynomial operations. We can easily imagine writing a collection of functions for input, output, addition, subtraction, and multiplication of polynomials using linked lists as the means of representation. A hypothetical user who wishes to read in polynomials $a(x)$, $b(x)$, and $d(x)$ and then compute $e(x) = a(x) * b(x) + d(x)$ would write his or her main function as:

```
polyPointer a, b, d, e
.
.
.
a = readPoly();
b = readPoly();
d = readPoly();
temp = pmult(a,b);
e = padd(temp,d);
printPoly(e);
```

If our user wishes to compute more polynomials, it would be useful to reclaim the nodes that are being used to represent *temp*(x) since we created *temp*(x) only to hold a partial result for $d(x)$. By returning the nodes of *temp*(x), we may use them to hold other polynomials. One by one, *erase* (Program 4.11) frees the nodes in *temp*.

```
void erase(polyPointer *ptr)
{/* erase the polynomial pointed to by ptr */
   polyPointer temp;
   while (*ptr) {
      temp = *ptr;
      *ptr = (*ptr)->link;
      free(temp);
   }
}
```

Program 4.11: Erasing a polynomial

4.4.4 Circular List Representation of Polynomials

We can free all the nodes of a polynomial more efficiently if we modify our list structure so that the link field of the last node points to the first node in the list (see Figure 4.14). We call this a *circular list*. A singly linked list in which the last node has a null link is called a *chain*.

Figure 4.14: Circular representation of $3x^{14} + 2x^8 + 1$

As we indicated earlier, we free nodes that are no longer in use so that we may reuse these nodes later. We can meet this objective, and obtain an efficient erase algorithm for circular lists, by maintaining our own list (as a chain) of nodes that have been "freed." When we need a new node, we examine this list. If the list is not empty, then we may use one of its nodes. Only when the list is empty do we need to use *malloc* to create a new node.

Let *avail* be a variable of type *polyPointer* that points to the first node in our list of freed nodes. Henceforth, we call this list the available space list or *avail* list. Initially, we set *avail* to *NULL*. Instead of using *malloc* and *free*, we now use *getNode* (Program 4.12) and *retNode* (Program 4.13).

We may erase a circular list in a fixed amount of time independent of the number

```
polyPointer getNode(void)
{/* provide a node for use */
   polyPointer node;
   if (avail) {
      node = avail;
      avail = avail->link;
   }
   else
      MALLOC(node, sizeof(*node));
   return node;
}
```

Program 4.12: *getNode* function

```
void retNode(polyPointer node)
{/* return a node to the available list */
   node->link = avail;
   avail = node;
}
```

Program 4.13: *retNode* function

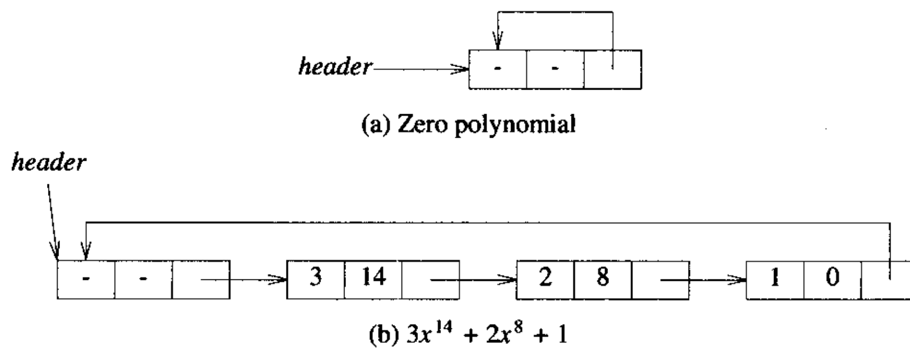
of nodes in the list using *cerase* (Program 4.14).

A direct changeover to the structure of Figure 4.14 creates problems when we implement the other polynomial operations since we must handle the zero polynomial as a special case. To avoid this special case, we introduce a *header node* into each polynomial, that is, each polynomial, zero or nonzero, contains one additional node. The *expon* and *coef* fields of this node are irrelevant. Thus, the zero polynomial has the representation of Figure 4.15(a), while $a(x) = 3x^{14} + 2x^8 + 1$ has the representation of Figure 4.15(b).

To simplify the addition algorithm for polynomials represented as circular lists, we set the *expon* field of the header node to -1. Program 4.15 gives the function to add polynomials represented in this way.

```
void cerase(polyPointer *ptr)
{/* erase the circular list pointed to by ptr */
   polyPointer temp;
   if (*ptr) {
      temp = (*ptr)->link;
      (*ptr)->link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```

Program 4.14: Erasing a circular list



(b) $3x^{14} + 2x^8 + 1$

Figure 4.15: Example polynomials with header nodes

4.4.5 Summary

Let us review what we have done so far. We have introduced the concepts of a singly linked list, a chain, and a singly linked circular list. Each node on one of these lists consists of exactly one link field and at least one other field.

In dealing with polynomials, we found it convenient to use circular lists. Another concept we introduced was an available space list. This list consisted of all nodes that had been used at least once and were not currently in use. By using the available space list and *getNode*, *retNode*, and *cerase*, it became possible to erase circular lists in

```
polyPointer cpadd(polyPointer a, polyPointer b)
{/* polynomials a and b are singly linked circular lists
    with a header node. Return a polynomial which is
    the sum of a and b */
  polyPointer startA, c, lastC;
  int sum, done = FALSE;
  startA = a;           /* record start of a */
  a = a→link;           /* skip header node for a and b*/
  b = b→link;
  c = getNode();        /* get a header node for sum */
  c→expon = -1; lastC = c;
  do {
    switch (COMPARE(a→expon, b→expon)) {
      case -1: /* a→expon < b→expon */
            attach(b→coef,b→expon,&lastC);
            b = b→link;
            break;
      case 0:  /* a→expon = b→expon */
            if (startA == a)  done = TRUE;
            else {
              sum = a→coef + b→coef;
              if (sum) attach(sum,a→expon,&lastC);
              a = a→link; b = b→link;
            }
            break;
      case 1:  /* a→expon > b→expon */
            attach(a→coef,a→expon,&lastC);
            a = a→link;
    }
  } while (!done);
  lastC→link = c;
  return c;
}
```

Program 4.15: Adding two polynomials represented as circular lists with header nodes

constant time, and also to reuse all nodes not currently in use. As we continue, we shall see more problems that call for variations in node structure and list representation because of the operations we wish to perform.

EXERCISES

1. Write a function, *pread*, that reads in n pairs of coefficients and exponents, ($coef_i$, $expon_i$), $0 \le i < n$ of a polynomial, x. Assume that $expon_{i+1} > expon_i$, $0 \le i < n-2$, and that $coef_i \neq 0$, $0 \le i < n$. Show that this operation can be performed in $O(n)$ time.
2. Let a and b be pointers to two polynomials. Write a function to compute the product polynomial $d = a*b$. Your function should leave a and b unaltered and create d as a new list. Show that if n and m are the number of terms in a and b, respectively, then this multiplication can be carried out in $O(nm^2)$ or $O(n^2m)$ time.
3. Let a be a pointer to a polynomial. Write a function, *peval*, to evaluate the polynomial a at point x, where x is some floating point number.
4. Rewrite Exercise 1 using a circular representation for the polynomial.
5. Rewrite Exercise 2 using a circular representation for the polynomial.
6. Rewrite Exercise 3 using a circular representation for the polynomial.
7. § [***Programming project***] Design and build a linked allocation system to represent and manipulate polynomials. You should use circularly linked lists with header nodes. Each term of the polynomial will be represented as a node, using the following structure:

coef	expon	link

In order to erase polynomials efficiently, use the available space list and associated functions discussed in this section.

Write and test the following functions:

(a) *pread*. Read in a polynomial and convert it to its circular representation. Return a pointer to the header node of this polynomial.

(b) *pwrite*. Output the polynomial using a form that clearly displays it.

(c) *padd*. Compute $c = a + b$. Do not change either a or b.

(d) *psub*. Compute $c = a - b$. Do not change either a or b.

(e) *pmult*. Compute $c = a*b$. Do not change either a or b.

(f) *eval*. Evaluate a polynomial at some point, a, where a is a floating point constant. Return the result as a floating point.

(g) *perase*. Return the polynomial represented as a circular list to the available space list.

4.5 ADDITIONAL LIST OPERATIONS

4.5.1 Operations For Chains

It is often necessary, and desirable, to build a variety of functions for manipulating singly linked lists. Some that we have seen already are *getNode* and *retNode*, which get and return nodes to the available space list. Inverting (or reversing) a chain (Program 4.16) is another useful operation. This routine is especially interesting because we can do it "in place" if we use three pointers. We use the following declarations:

```
typedef struct listNode *listPointer;
typedef struct {
        char data;
        listPointer link;
        } listNode;
```

Try out this function with at least three examples, an empty list and lists of one and two nodes, so that you understand how it works. For a list of $length \geq 1$ nodes, the **while** loop is executed *length* times and so the computing time is linear or O(*length*).

Another useful function is one that concatenates two chains, *ptr*1 and *ptr*2 (Program 4.17). The complexity of this function is O(length of list *ptr*1). Since this function does not allocate additional storage for the new list, *ptr*1 also contains the concatenated list. (The exercises explore a concatenation function that does not alter *ptr*1.)

```
listPointer invert(listPointer lead)
{/* invert the list pointed to by lead */
  listPointer middle,trail;
  middle = NULL;
  while (lead) {
    trail = middle;
    middle = lead;
    lead = lead→link;
    middle→link = trail;
  }
  return middle;
}
```

Program 4.16: Inverting a singly linked list

```
listPointer concatenate(listPointer ptr1, listPointer ptr2)
{/* produce a new list that contains the list
    ptr1 followed by the list ptr2. The
    list pointed to by ptr1 is changed permanently */
   listPointer temp;
   /* check for empty lists */
   if (!ptr1) return ptr2;
   if (!ptr2) return ptr1;

   /* neither list is empty, find end of first list */
   for (temp = ptr1; temp->link; temp = temp->link) ;

   /* link end of first to start of second */
   temp->link = ptr2;
}
```

Program 4.17: Concatenating singly linked lists

4.5.2 Operations For Circularly Linked Lists

Now let us take another look at circular lists like the one in Figure 4.14. By keeping a pointer *last* to the last node in the list rather than to the first, we are able to insert an element at both the front and end with ease. Had we kept a pointer to the first node instead of the last node, inserting at the front would require us to must move down the entire length of the list until we find the last node so that we can change the pointer in the last node to point to the new first node. Program 4.18 gives the code at insert a node at the front of a circular list. To insert at the rear, we only need to add the additional statement **last = node* to the **else** clause of *insertFront* (Program 4.18).

As another example of a simple function for circular lists, we write a function (Program 4.19) that determines the length of such a list.

EXERCISES

1. Create a function that searches for an integer, *num*, in a circularly linked list. The function should return a pointer to the node that contains *num* if *num* is in the list and *NULL* otherwise.
2. Write a function that deletes a node containing a number, *num*, from a circularly linked list. Your function should first search for *num*.

```
void insertFront(listPointer *last, listPointer node)
{/* insert node at the front of the circular list whose
    last node is last */
   if (!(*last)) {
   /* list is empty, change last to point to new entry */
      *last = node;
      node→link = node;
   }
   else {
   /* list is not empty, add new entry at front */
      node→link = (*last)→link;
      (*last)→link = node;
   }
}
```

Program 4.18: Inserting at the front of a list

```
int length(listPointer last)
{/* find the length of the circular list last */
   listPointer temp;
   int count = 0;
   if (last) {
      temp = last;
      do {
         count++;
         temp = temp→link;
      } while (temp != last);
   }
   return count;
}
```

Program 4.19: Finding the length of a circular list

3. Write a function to concatenate two circular lists together. Assume that the pointer to each such list points to the last node. Your function should return a pointer to the last node of the concatenated circular list. Following the concatenation, the input lists do not exist independently. What is the time complexity of

your function?

4. Write a function to reverse the direction of pointers in a circular list.

4.6 EQUIVALENCE CLASSES

Let us put together some of the concepts on linked and sequential representations to solve a problem that arises in the design and manufacture of very large-scale integrated (VLSI) circuits. One of the steps in the manufacture of a VLSI circuit involves exposing a silicon wafer using a series of masks. Each mask consists of several polygons. Polygons that overlap electrically are equivalent and electrical equivalence specifies a relationship among mask polygons. This relation has several properties that it shares with other equivalence relations, such as the standard mathematical equals. Suppose that we denote an arbitrary equivalence relation by the symbol $\equiv$ and that:

(1) For any polygon x, $x \equiv x$, that is, x is electrically equivalent to itself. Thus, $\equiv$ is reflexive.

(2) For any two polygons, x and y, if $x \equiv y$ then $y \equiv x$. Thus, the relation $\equiv$ is symmetric.

(3) For any three polygons, x, y, and z, if $x \equiv y$ and $y \equiv z$ then $x \equiv z$. For example, if x and y are electrically equivalent and y and z are also equivalent, then x and z are also electrically equivalent. Thus, the relation $\equiv$ is transitive.

Definition: A relation, $\equiv$, over a set, S, is said to be an *equivalence relation* over S *iff* it is symmetric, reflexive, and transitive over S. □

Examples of equivalence relations are numerous. For example, the "equal to" (=) relationship is an equivalence relation since

(1) $x = x$

(2) $x = y$ implies $y = x$

(3) $x = y$ and $y = z$ implies that $x = z$

We can use an equivalence relation to partition a set S into equivalence classes such that two members x and y of S are in the same equivalence class *iff* $x \equiv y$. For example, if we have twelve polygons numbered 0 through 11 and the following pairs overlap:

$$0 \equiv 4,\ 3 \equiv 1,\ 6 \equiv 10,\ 8 \equiv 9,\ 7 \equiv 4,\ 6 \equiv 8,\ 3 \equiv 5,\ 2 \equiv 11,\ 11 \equiv 0$$

then, as a result of the reflexivity, symmetry, and transitivity of the relation $\equiv$, we can partition the twelve polygons into the following equivalence classes:

$$\{0, 2, 4, 7, 11\};\ \{1, 3, 5\};\ \{6, 8, 9, 10\}$$

These equivalence classes are important because they define a signal net that we can use to verify the correctness of the masks.

The algorithm to determine equivalence works in two phases. In the first phase, we read in and store the equivalence pairs $<i, j>$. In the second phase we begin at 0 and find all pairs of the form $<0, j>$, where 0 and j are in the same equivalence class. By transitivity, all pairs of the form $<j, k>$ imply that k is in the same equivalence class as 0. We continue in this way until we have found, marked, and printed the entire equivalence class containing 0. Then we continue on.

Our first design attempt appears in Program 4.20. Let m and n represent the number of related pairs and the number of objects, respectively. We first must figure out which data structure we should use to hold these pairs. To determine this, we examine the operations that are required. The pair $<i, j>$ is essentially two random integers in the range 0 to $n-1$. Easy random access would dictate an array, say *pairs*[n][m]. The ith row would contain the elements, j, that are paired directly to i in the input. However, this could waste a lot of space since very few of the array elements would be used. It also might require considerable time to insert a new pair, $<i, k>$, into row i since we would have to scan the row for the next free location or use more storage.

```
void equivalence()
{
   initialize;
   while (there are more pairs) {
      read the next pair <i,j>;
      process this pair;
   }
   initialize the output;
   do
      output a new equivalence class;
   while (not done);
}
```

Program 4.20: First pass at equivalence algorithm

These considerations lead us to a linked list representation for each row. Our node structure requires only a data and a link field. However, since we still need random access to the ith row, we use a one-dimensional array, *seq* [n], to hold the header nodes of the n lists. For the second phase of the algorithm, we need a mechanism that tells us whether or not the object, i, has been printed. We use the array *out* [n] and the constants *TRUE* and *FALSE* for this purpose. Our next refinement appears in Program 4.21.

```
void equivalence()
{
   initialize seq to NULL and out to TRUE;
   while (there are more pairs) {
      read the next pair, <i,j>;
      put j on the seq[i] list;
      put i on the seq[j] list;
   }
   for (i = 0; i < n; i++)
      if (out[i]) {
         out[i] = FALSE;
         output this equivalence class;
      }
}
```

Program 4.21: A more detailed version of the equivalence algorithm

Let us simulate this algorithm, as we have developed it thus far, using the previous data set. After the **while** loop is completed the lists resemble those appearing in Figure 4.16. For each relation $i \equiv j$, we use two nodes. The variable *seq* [*i*] points to the list of nodes that contains every number that is directly equivalent to *i* by an input relation.

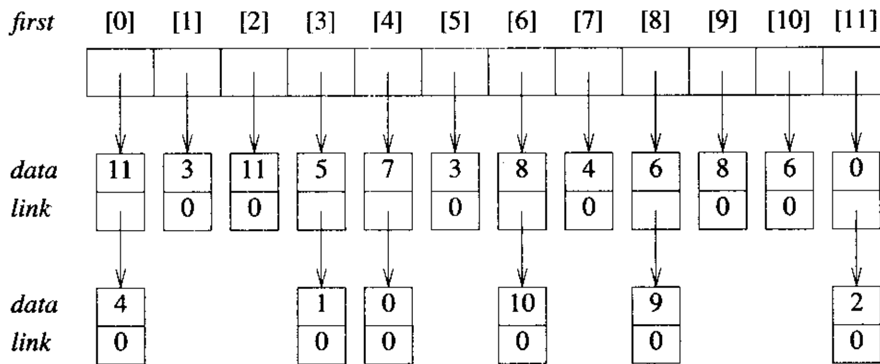


Figure 4.16: Lists after pairs have been input

In phase two, we scan the *seq* array for the first i, $0 \le i < n$, such that $out[i]$ = *TRUE*. Each element in the list $seq[i]$ is printed. To process the remaining lists which, by transitivity, belong in the same class as i, we create a stack of their nodes. We do this by changing the link fields so that they point in the reverse direction. Program 4.22 contains the complete equivalence algorithm.

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define FALSE 0
#define TRUE 1
typedef struct node *nodePointer;
typedef struct {
        int data;
        nodePointer link;
        } node;
void main(void)
{
   short int out[MAX_SIZE];
   nodePointer seq[MAX_SIZE];
   nodePointer x,y,top;
   int i,j,n;

   printf("Enter the size (<= %d) ",MAX_SIZE);
   scanf("%d",&n);
   for (i = 0; i < n; i++) {
   /* initialize seq and out */
      out[i] = TRUE;   seq[i] = NULL;
   }

   /* Phase 1: Input the equivalence pairs: */
   printf("Enter a pair of numbers (-1 -1 to quit): ");
   scanf("%d%d",&i,&j);
   while (i >= 0) {
      MALLOC(x, sizeof(*x));
      x->data = j;  x->link = seq[i];  seq[i] = x;
      MALLOC(x, sizeof(*x));
      x->data = i;  x->link = seq[j];  seq[j] = x;
      printf("Enter a pair of numbers (-1 -1 to quit): ");
      scanf("%d%d",&i,&j);
   }
```

```
    /* Phase 2: output the equivalence classes */
    for (i = 0; i < n; i++)
      if (out[i]) {
        printf("\nNew class: %5d",i);
        out[i] = FALSE;    /* set class to false */
        x = seq[i];  top = NULL; /* initialize stack */
        for (;;) {      /* find rest of class */
          while (x) {  /* process list */
            j = x→data;
            if (out[j]) {
              printf("%5d",j);  out[j] = FALSE;
              y = x→link; x→link = top; top = x; x = y;
            }
            else x = x→link;
          }
          if (!top) break;
          x = seq[top→data]; top = top→link;
                                      /* unstack */
        }
      }
}
```

Program 4.22: Program to find equivalence classes

Analysis of the equivalence program: The initialization of *seq* and *out* takes $O(n)$ time. Inputting the equivalence pairs in phase 1 takes a constant amount of time per pair. Hence, the total time for this phase is $O(m+n)$ where m is the number of pairs input. In phase 2, we put each node onto the linked stack at most once. Since there are only $2m$ nodes, and we execute the **for** loop n times, the time for this phase is $O(m + n)$. Thus, the overall computing time is $O(m + n)$. Any algorithm that processes equivalence relations must look at all m equivalence pairs and at all n polygons at least once. Thus, there is no algorithm with a computing time less than $O(m+n)$. This means that the equivalence algorithm is optimal to within a constant factor. Unfortunately, the space required by the algorithm is also $O(m+n)$. In Chapter 5, we look at an alternative solution to this problem that requires only $O(n)$ space. □

4.7 SPARSE MATRICES

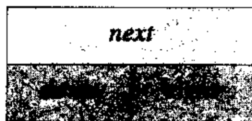
4.7.1 Sparse Matrix Representation

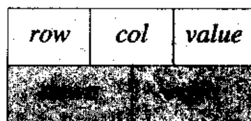
In Chapter 2, we saw that we could save space and computing time by retaining only the nonzero terms of sparse matrices. When the nonzero terms did not form a "nice" pattern,

such as a triangle or a band, we devised a sequential scheme in which we represented each nonzero term by a node with three fields: *row*, *column*, and *value*. We organized these nodes sequentially. However, we found that when we performed matrix operations such as addition, subtraction, or multiplication, the number of nonzero terms varied. Matrices representing partial computations, as in the case of polynomials, were created and later destroyed to make space for further matrices. Thus, the sequential representation of sparse matrices suffered from the same inadequacies as the similar representation of polynomials. In this section, we study a linked list representation for sparse matrices. As we have seen previously, linked lists allow us to efficiently represent structures that vary in size, a benefit that also applies to sparse matrices.

In our data representation, we represent each column of a sparse matrix as a circularly linked list with a header node. We use a similar representation for each row of a sparse matrix. Each node has a tag field, which we use to distinguish between header nodes and entry nodes. Each header node has three additional fields: *down*, *right*, and *next* (Figure 4.17(a)). We use the *down* field to link into a column list and the *right* field to link into a row list. The *next* field links the header nodes together. The header node for row i is also the header node for column i, and the total number of header nodes is max {number of rows, number of columns}.

Each entry node has five fields in addition to the tag field: *row, col, down, right, value* (Figure 4.17(b)). We use the *down* field to link to the next nonzero term in the same column and the *right* field to link to the next nonzero term in the same row. Thus, if $a_{ij} \neq 0$, there is a node with tag field = *entry*, *value* = a_{ij}, *row* = i, and *col* = j (Figure 4.17(c)). We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.





(a) header node (b) element node

head field is not shown

Figure 4.17: Node structure for sparse matrices

As noted earlier, each header node is in three lists: a list of rows, a list of columns, and a list of header nodes. The list of header nodes also has a header node that has the same structure as an entry node (Figure 4.17(b)). We use the *row* and *col* fields of this node to store the matrix dimensions.

Consider the sparse matrix, a, shown in Figure 4.18. Figure 4.19 shows the linked representation of this matrix. Although we have not shown the value of the tag fields, we can easily determine these values from the node structure. For each nonzero term of a, we have one entry node that is in exactly one row list and one column list. The header nodes are marked $H0$-$H3$. As the figure shows, we use the *right* field of the header node list header to link into the list of header nodes. Notice also that we may reference the entire matrix through the header node, a, of the list of header nodes.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

Figure 4.18: 4×4 sparse matrix a

If we wish to represent a *numRows* × *numCols* matrix with *numTerms* nonzero terms, then we need max {*numRows, numCols*} + *numTerms* + 1 nodes. While each node may require several words of memory, the total storage will be less than *numRows* × *numCols* when *numTerms* is sufficiently small.

Since we have two different types of nodes in our representation, we use a **union** to create the appropriate data structure. The necessary C declarations are as follows:

```
#define MAX_SIZE 50 /*size of largest matrix*/
typedef enum {head,entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct {
        int row;
        int col;
        int value;
        } entryNode;
typedef struct {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union {
             matrixPointer next;
             entryNode entry;
             } u;
        } matrixNode;
matrixPointer hdnode[MAX_SIZE];
```

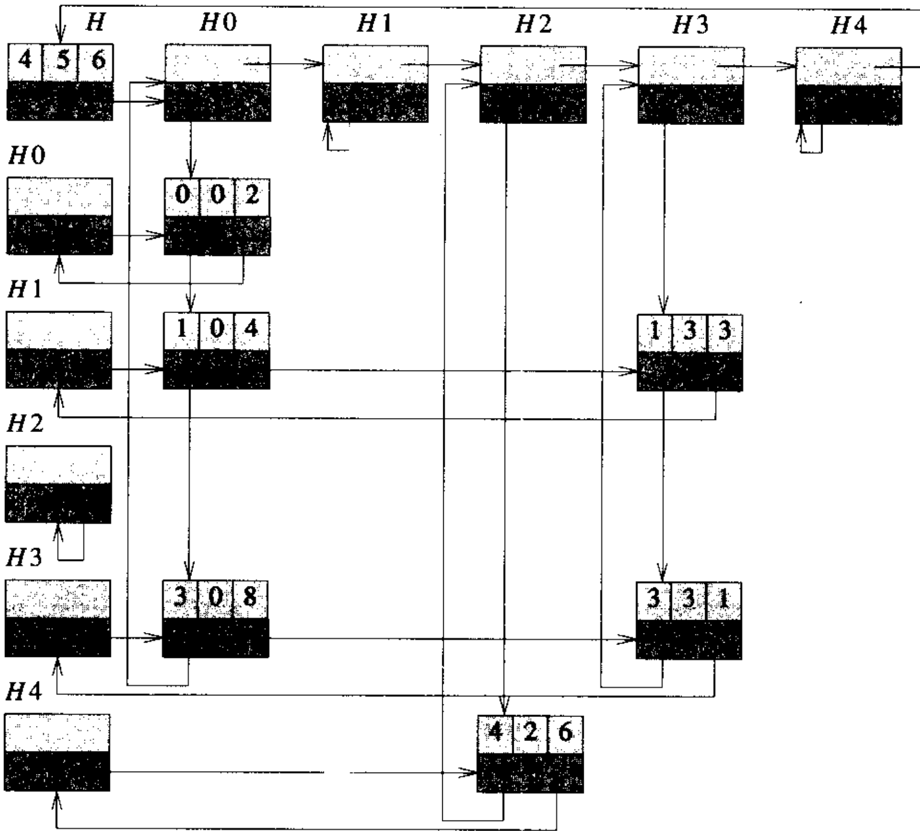


Figure 4.19: Linked representation of the sparse matrix of Figure 4.18 (the *head* field of a node is not shown)

4.7.2 Sparse Matrix Input

The first operation we implement is that of reading in a sparse matrix and obtaining its linked representation. We assume that the first input line consists of the number of rows (*numRows*), the number of columns (*numCols*), and the number of nonzero terms (*numTerms*). This line is followed by *numTerms* lines of input, each of which is of the form: *row*, *col*, *value*. We assume that these lines are ordered by rows and within rows by columns. For example, Figure 4.20 shows the input for the 4×4 matrix of Figure 4.18.

	[0]	[1]	[2]
[0]	4	4	4
[1]	0	2	11
[2]	1	0	12
[3]	2	1	−4
[4]	3	3	−15

Figure 4.20: Sample input for sparse matrix

We use an auxiliary array, *hdnode*, which we assume is at least as large as the largest-dimensioned matrix to be input. *hdnode*[*i*], which is a pointer to the header node for column *i* and row *i*, allows us to access efficiently columns at random, while we are setting up the input matrix. The function *mread* (Program 4.23) first sets up the header nodes and then sets up each row list while simultaneously building the column lists. The *next* field of header node, *i*, is initially used to keep track of the last node in column *i*. The last **for** loop of *mread* links the header nodes together through this field.

```
matrixPointer mread(void)
{/* read in a matrix and set up its linked representation.
    An auxiliary global array hdnode is used */
  int numRows, numCols,  numTerms, numHeads, i;
  int row, col, value, currentRow;
  matrixPointer temp,last,node;

  printf("Enter the number of rows, columns
                         and number of nonzero terms: ");
  scanf("%d%d%d",&numRows, &numCols, &numTerms);
  numHeads = (numCols > numRows) ? numCols : numRows;
  /* set up header node for the list of header nodes */
  node = newNode(); node→tag = entry;
  node→u.entry.row = numRows;
  node→u.entry.col = numCols;

  if (!numHeads) node→right = node;
  else { /* initialize the header nodes */
    for (i = 0; i < numHeads; i++) {
      temp = newNode;
```

```
         hdnode[i] = temp; hdnode[i]→tag = head;
         hdnode[i]→right = temp;  hdnode[i]→u.next = temp;
      }
      currentRow = 0;
      last = hdnode[0]; /* last node in current row */
      for (i = 0; i < numTerms; i++) {
         printf("Enter row, column and value: ");
         scanf("%d%d%d",&row,&col,&value);
         if (row > currentRow) {/* close current row */
            last→right = hdnode[currentRow];
            currentRow = row; last = hdnode[row];
         }
         MALLOC(temp, sizeof(*temp));
         temp→tag = entry;  temp→u.entry.row = row;
         temp→u.entry.col = col;
         temp→u.entry.value = value;
         last→right = temp; /* link into row list */
         last = temp;
         /* link into column list */
         hdnode[col]→u.next→down = temp;
         hdnode[col]→u.next = temp;
      }
      /*close last row */
      last→right = hdnode[currentRow];
      /* close all column lists */
      for (i = 0; i < numCols; i++)
         hdnode[i]→u.next→down = hdnode[i];
      /* link all header nodes together */
      for (i = 0; i < numHeads-1; i++)
         hdnode[i]→u.next = hdnode[i+1];
      hdnode[numHeads-1]→u.next =  node;
      node→right = hdnode[0];
   }
   return node;
}
```

Program 4.23: Read in a sparse matrix

Analysis of *mread*: Since *MALLOC* works in a constant amount of time, we can set up all of the header nodes in O(max {*numRows,numlCols*}) time. We can also set up each nonzero term in a constant amount of time because we use the variable *last* to keep track of the current row, while *next* keeps track of the current column. Thus, the **for** loop that

inputs and links the entry nodes requires only $O(numTerms)$ time. The remainder of the function takes $O(\max\{numRows, numCols\})$ time. Therefore, the total time is:

$$O(\max\{numRows, numCols\} + numTerms)$$

$$= O(numRows + numCols + numTerms).$$

Notice that this is asymptotically better than the input time of $O(numRows \times numCols)$ for a $numRows \times numCols$ matrix using a two-dimensional array. However it is slightly worse than the sequential method used in Section 2.5. □

4.7.3 Sparse Matrix Output

We would now like to print out the contents of a sparse matrix in a form that resembles that found in Figure 4.20. The function *mwrite* (Program 4.24) implements this operation.

```
void mwrite(matrixPointer node)
{/* print out the matrix in row major form */
   int i;
   matrixPointer temp, head = node→right;
   /* matrix dimensions */
   printf(" \n numRows = %d, numCols = %d \n",
                node→u.entry.row, node→u.entry.col);
   printf(" The matrix by row, column, and value: \n\n");
   for (i = 0; i < node→u.entry.row; i++) {
   /* print out the entries in each row */
      for (temp = head→right; temp != head;
                                    temp = temp→right)
         printf("%5d%5d%5d \n",temp→u.entry.row,
              temp→u.entry.col, temp→u.entry.value);
      head = head→u.next; /* next row */
   }
}
```

Program 4.24: Write out a sparse matrix

Analysis of *mwrite*: The function *mwrite* uses two **for** loops. The number of iterations of the outer **for** loop is *numRows*. For any row, *i*, the number of iterations of the inner **for** loop is equal to the number of entries for row *i*. Therefore, the computing time of the *mwrite* function is $O(numRows + numTerms)$. □

4.7.4 Erasing a Sparse Matrix

Before closing this section we want to look at an algorithm that returns all nodes of a sparse matrix to the system memory. We return the nodes one at a time using *free*, although we could develop a faster algorithm using an available space list (see Section 4.4). The function *merase* (Program 4.25) implements the erase operation.

```
void merase(matrixPointer *node)
{/* erase the matrix, return the nodes to the heap */
   matrixPointer x,y, head = (*node)→right;
   int i;
   /* free the entry and header nodes by row */
   for (i = 0; i < (*node)→u.entry.row; i++) {
      y = head→right;
      while (y != head) {
         x = y; y = y→right; free(x);
      }
      x = head; head = head→u.next; free(x);
   }
   /* free remaining header nodes*/
   y = head;
   while (y != *node) {
      x = y; y = y→u.next; free(x);
   }
   free(*node); *node = NULL;
}
```

Program 4.25: Erase a sparse matrix

Analysis of *merase*: First, *merase* returns the entry nodes and the row header nodes to the system memory using a nested loop structure that resembles the structure found in *mwrite*. Thus, the computing time for the nested loops is O(*numRows* + *numTerms*). The time to erase the remaining header nodes is O(*numRows* + *numCols*). Hence, the computing time for *merase* is O(*numRows* + *numCols* + *numTerms*). □

EXERCISES

1. Let a and b be two sparse matrices. Write a function, *madd*, to create the matrix $d = a + b$. Your function should leave matrices a and b unchanged, and set up d as a new matrix. Show that if a and b are $numRows \times numCols$ matrices with $numterms_a$ and $numTerms_b$ nonzero terms, then we can perform this addition in

$O(numRows + numCols + numTerms_a + numTerms_b)$ time.

2. Let a and b be two sparse matrices. Write a function, *mmult*, to create the matrix $d = a*b$. Show that if a is a $numRows_a \times numCols_a$ matrix with $numTerms_a$ nonzero terms and b is a $numCols_a \times numCols_b$ matrix with $numTerms_b$ nonzero terms, then we can compute d in $O(numCols_b \times numTerms_a + numRows_a \times numTerms_b)$ time. Can you think of a way to compute d in $O(\min \{numCols_b \times numTerms_a, numRows_a \times numTerms_b\})$ time?

3. (a) Rewrite *merase* so that it places the erased list into an available space list rather than returning it to system memory.

   (b) Rewrite *mread* so that it first attempts to obtain a new node from the available space list rather than the system memory.

4. Write a function, *mtranspose*, to compute the matrix $b = a^T$, the transpose of the sparse matrix a. What is the computing time of your function?

5. Design a function that copies a sparse matrix. What is the computing time of your function?

6. § [***Programming project***] We want to implement a complete linked list system to perform arithmetic on sparse matrices using our linked list representation. Create a user-friendly, menu-driven system that performs the following operations. (The matrix names are used only for illustrative purposes. The functions are specified as templates to which you must add the appropriate parameters.)

   (a) *mread*. Read in a sparse matrix.

   (b) *mwrite*. Write out the contents of a sparse matrix.

   (c) *merase*. Erase a sparse matrix.

   (d) *madd*. Create the sparse matrix $d = a + b$

   (e) *mmult*. Create the sparse matrix $d = a*b$.

   (f) *mtranspose*. Create the sparse matrix $b = a^T$.

4.8 DOUBLY LINKED LISTS

So far we have been working chiefly with chains and singly linked circular lists. For some problems these would be too restrictive. One difficulty with these lists is that if we are pointing to a specific node, say p, then we can move only in the direction of the links. The only way to find the node that precedes p is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. As can be seen from Example 4.4, easy deletion of an arbitrary node requires knowing the preceding node. If we have a problem in which it is necessary to move in either direction or in which we must delete arbitrary nodes, then it is useful to have doubly linked lists. Each node now has two link fields, one linking in the forward direction and the other linking in the backward direction.

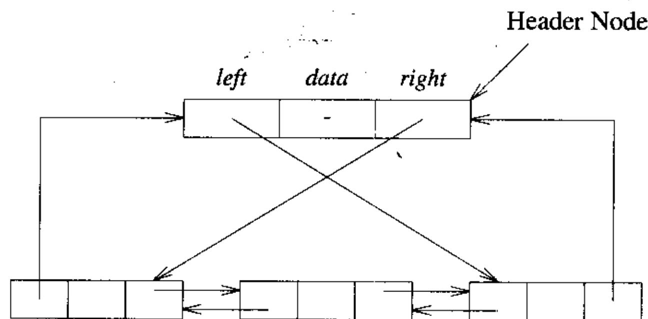


Figure 4.21: Doubly linked circular list with header node

A node in a doubly linked list has at least three fields, a left link field (*llink*), a data field (*data*), and a right link field (*rlink*). The necessary declarations are:

```
typedef struct node *nodePointer;
typedef struct {
        nodePointer llink;
        element data;
        nodePointer rlink;
        } node;
```

A doubly linked list may or may not be circular. A sample doubly linked circular list with three nodes is given in Figure 4.21. Besides these three nodes, we have added a header node. As was true in previous sections, a header node allows us to implement our operations more easily. The data field of the header node usually contains no information. If *ptr* points to any node in a doubly linked list, then:

$$ptr = ptr \rightarrow llink \rightarrow rlink = ptr \rightarrow rlink \rightarrow llink$$

This formula reflects the essential virtue of this structure, namely, that we can go back and forth with equal ease. An empty list is not really empty since it always has a header node whose structure is illustrated in Figure 4.22.

To use doubly linked lists we must be able to insert and delete nodes. Insertion into a doubly linked list is fairly easy. Assume that we have two nodes, *node* and *newnode*, *node* may be either a header node or an interior node in a list. The function *dinsert* (Program 4.26) performs the insertion operation in constant time.

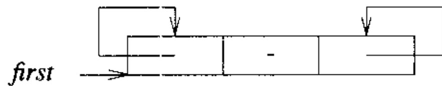


Figure 4.22: Empty doubly linked circular list with header node

```
void dinsert(nodePointer node, nodePointer newnode)
{/* insert newnode to the right of node */
   newnode->llink = node;
   newnode->rlink = node->rlink;
   node->rlink->llink = newnode;
   node->rlink = newnode;
}
```

Program 4.26: Insertion into a doubly linked circular list

Deletion from a doubly linked list is equally easy. The function *ddelete* (Program 4.27) deletes the node *deleted* from the list pointed to by node. To accomplish this deletion, we only need to change the link fields of the nodes that precede (*deleted→llink→rlink*) and follow (*deleted→rlink→llink*) the node we want to delete. Figure 4.23 shows the deletion in a doubly linked list with a single node.

EXERCISES

1. Assume that we have a doubly linked list, as represented in Figure 4.21, and that we want to add a new node between the second and third nodes in the list. Redraw the figure so that it shows the insertion. Label the fields of the affected nodes so that you show how each statement in the *dinsert* function is executed. For example, label *newnode→llink*, *newnode→rlink*, and *node→rlink→llink*.
2. Repeat Exercise 1, but delete the second node from the list.
3. Devise a linked representation for a list in which insertions and deletions can be made at either end in O(1) time. Such a structure is called a *deque*. Write functions to insert and delete at either end.

```
void ddelete(nodePointer node, nodePointer deleted)
{/* delete from the doubly linked list */
   if (node == deleted)
      printf("Deletion of header node not permitted.\n");
   else {
      deleted→llink→rlink = deleted→rlink;
      deleted→rlink→llink = deleted→llink;
      free(deleted);
   }
}
```

Program 4.27: Deletion from a doubly linked circular list

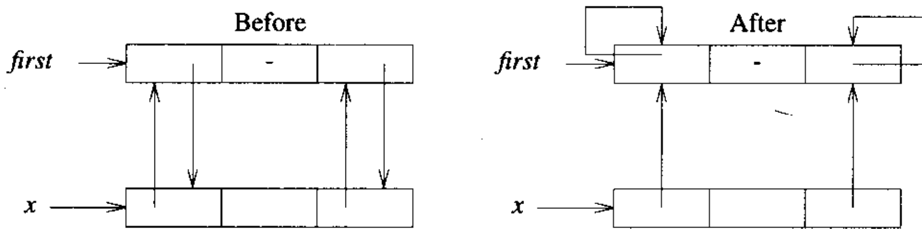


Figure 4.23: Deletion from a doubly linked circular list

4. Consider the operation XOR (exclusive OR, also written as $\oplus$) defined as follows (for i, j binary):

$$i \oplus j = \begin{cases} 0 & \text{if } i \text{ and } j \text{ are identical} \\ 1 & \text{otherwise} \end{cases}$$

This definition differs from the usual OR of logic, which is defined as

$$i \text{ OR } j = \begin{cases} 0 & \text{if } i = j = 0 \\ 1 & \text{otherwise} \end{cases}$$

The definition can be extended to the case in which i and j are binary strings (i.e.,

take the XOR of corresponding bits of i and j). So, for example, if i = 10110 and j = 01100, then i XOR j = $i \oplus j$ = 11010. Note that

$$a \oplus (a \oplus b) = (a \oplus a) \oplus b = b$$

and

$$(a \oplus b) \oplus b = a \oplus (b \oplus b) = a$$

This notation gives us a space-saving device for storing the right and left links of a doubly linked list. The nodes will now have only two data members: *data* and *link*. If l is to the left of node x and r to its right, then $x \rightarrow link = l \oplus r$ If x is the leftmost node of a non-circular list, $l = 0$, and if x is the rightmost node, $r = 0$. For a new doubly linked list class in which the link field of each node is the exclusive or of the addresses of the nodes to its left and right, do the following.

(a) Write a function to traverse the doubly linked list from left to right, printing out the contents of the *data* field of each node.

(b) Write a function to traverse the list from right to left, printing out the contents of the *data* field of each node.