CHAPTER 4

LISTS

4.1 Pointers

– Consider the following alphabetized list of three letter English words ending in *at*:

(*bat*, *cat*, *sat*, *vat*)

We would like to add the word *mat* to this list.

- If we store this list in an array, then we must move *sat* and *vat* one position to the right before we insert *mat*.
- Similarly, if we want to remove the word *cat* from the list, we must move *sat* and *vat* one position to the left to maintain our sequential representation.
- In general case, arbitrary insertion and deletion from arrays can be very time-consuming.

– We can attain an elegant solution to the problem of data movement in sequential representations by using lined representation.
  - In a sequential representation the order of elements is the same as in the ordered list, while in a linked representation these two sequences need not be the same.
  - To access elements of the list in the correct order with each element, we store the address, or location, of the next element in that list.
  - Thus, associated with each list element is a *node* which contains both a data component and a pointer to the next item in the list. The pointers are often called *links*.

– C provides extensive supports for pointers.

- The two most important operators used with the pointer type are:

  & the address operator

  * the dereferencing (or indirection) operator

  – **Example:**

  If we have the declaration:

  ```
  int i, *pi;
  ```

  then *i* is an integer variable and *pi* is a pointer to an integer.

  If we say:

  ```
  pi = &i;
  ```

  then &*i* returns the address of *i* and assigns it as the value of *pi*.

  To assign a value to *i* we can say:

  ```
  i = 10;
  ```

  or

  ```
  *pi = 10;
  ```

- Pointers can be dangerous
  - We can attain a high degree of flexibility and efficiency by using pointers. But pointers can be dangerous as well.
    - When programming in C, it is a wise practice to set all pointers to *NULL* when they are not actually pointing to an object.
    - Another wise programming tactic is to use explicit **type cast** when converting between pointer types.
      - **Example:**

```
pi = malloc(sizeof(int)); /*assign to pi a pointer to
  int*/
pf = (float *) pi; /*casts an int pointer to a float
  pointer*/
```

    - Another area of concern is that in many systems, pointers have the same size as type **int**.
      - Since **int** is the default type specifier, some programmers omit the return type when defining a function. The return type defaults to **int** which can later be interpreted as a pointer.

- Using dynamically allocated storage
  - When you write your program you may not know how much space you will need, nor do you wish to allocate some vary large area that may never be required.
    - To solve this program C provides a mechanism, called a *heap*, for allocating storage at run-time.
    - Whenever you need a new area of memory, you may call a function, *malloc*, and request the amount you need.
    - At a later time when you no longer need an area of memory, you may free it by calling another function, free, and return the area of memory to the system.
    - Example:

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Program 4.1: Allocation and deallocation of pointers

4.2 Singly linked lists

– Linked lists are drawn as an order sequence of nodes with links represented as arrows (Figure 4.1).
  - The name of the pointer to the first node in the list is the name of the list.
    – Thus, the list of Figure 4.1 is called *ptr*.
    – Notice that we do not explicitly put in the values of pointers, but simply draw allows to indicate that they are there.
  - We do this to reinforce the facts that:

    the nodes do not resident in sequential locations

    the locations of the nodes may change on different runs

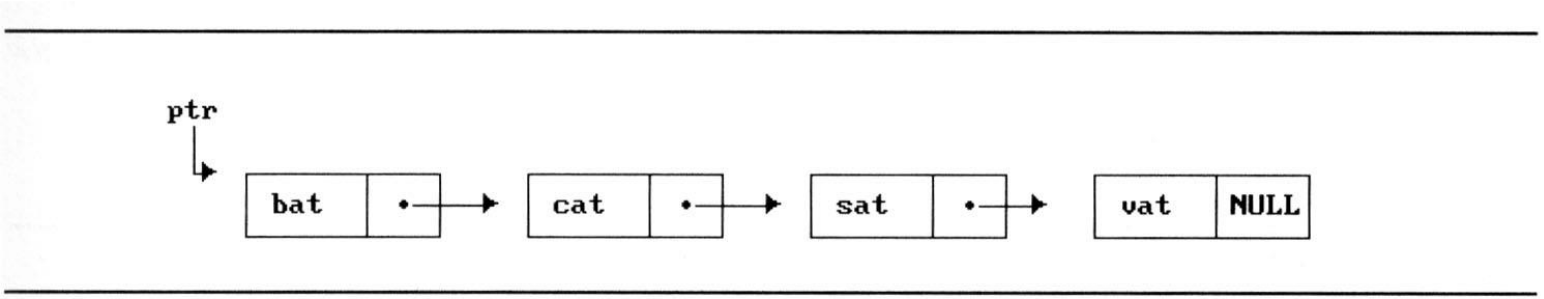


Figure 4.1: Usual way to draw a linked list

– Let us now see why it is easier to make arbitrary insertions and deletions using a linked list rather than a sequential list.

- To insert the word *mat* between *cat* can *sat*, we must:

  Get a node that is currently unused; let its address be *paddr*.

  Set the data field of this node to *mat*.

  Set *paddr*'s link field to point to the address found in the link field of the node containing *cat*.

  Set the link field of the node containing *cat* to point to *paddr*.

- Figure 4.2 show how the list changes after we insert *mat*.

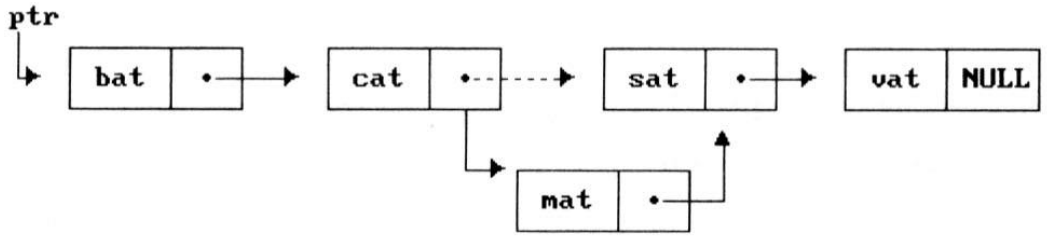


Figure 4.2: Insert *mat* after *cat*

- Now suppose that want to delete mat from the list.
  - We only need to find the element that immediately precedes mat, which is cat, and set its link field to point to mat's link (Figure 4.3).
  - We have not moved any data, and although the link field of mat still points to sat, mat is no longer in the list.

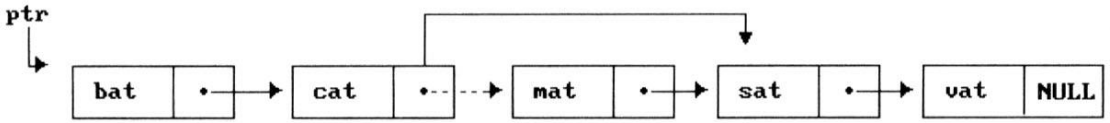
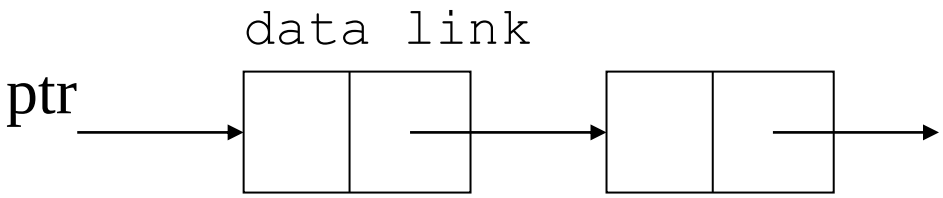


Figure 4.3: Delete *mat* from list

- From this brief discussion of linked lists, we see that we need the following capabilities to make linked representations possible:
  - A mechanism for defining a node's structure, that is, the fields it contains. We use *self-referential structures*, discussed in Section 2.2 to do this.
  - A way to create new nodes when we need them. The *malloc* function handles this operation.
  - A way to remove nodes that we no longer need. The *free* function handles this operation.

- A *self-referential structure* is one in which one or more of it component is a pointer to itself.
  - Self-referential structures usually require dynamic storage management routines (*malloc* and *free*) to explicitly obtain and release memory.
  - **Example:**

```
typedef struct list_node *list_pointer;
typedef struct list_node {
                char data[4];
                list_pointer link;
                };
list_pointer ptr=NULL;
```



4.3 Dynamically linked stacks and queues

– 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 direction of links for both 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 form the top of the stack.
    – In the case of Figure 4.10(b), we can easily add a node to the rear of the queue and add or delete a node at the front of a queue.

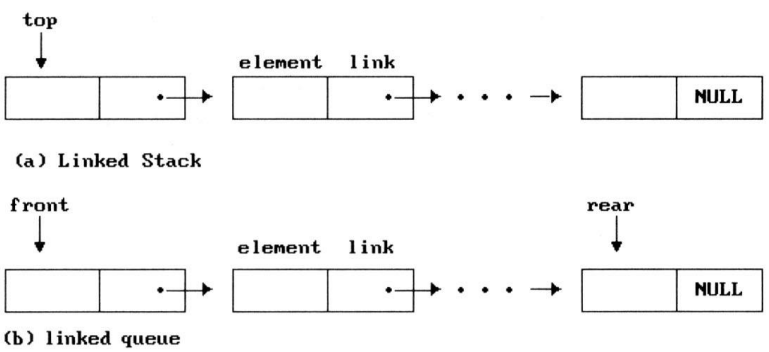


Figure 4.10: Linked stack and queue

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

```
void add(stack_pointer *top, element item)
{
/* add an element to the top of the stack */
   stack_pointer temp =
                  (stack_pointer) malloc(sizeof (stack));
   if (IS_FULL(temp)) {
      fprintf(stderr, "The memory is full\n");
      exit(1);
   }
   temp->item = item;
   temp->link = *top;
   *top = temp;
}
```

Program 4.6: Add to a linked stack

```
element delete(stack_pointer *top) {
/* delete an element from the stack */
   stack_pointer temp = *top;
   element item;
   if (IS_EMPTY(temp)) {
      fprintf(stderr, "The stack is empty\n");
      exit(1);
   }
   item = temp->item;
   *top = temp->link;
   free(temp);
   return item;
}
```

Program 4.7: Delete from a linked stack

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

```
void addq(queue_pointer *front, queue_pointer *rear,
                                element item)
{
/* add an element to the rear of the queue */
   queue_pointer temp =
                    (queue_pointer) malloc(sizeof(queue));
   if (IS_FULL(temp)) {
      fprintf(stderr, "The memory is full\n");
      exit(1);
   }
   temp->item = item;
   temp->link = NULL;
   if (*front) (*rear)->link = temp;
   else *front = temp;
   *rear = temp;
}
```

Program 4.8: Add to the rear of a linked queue

```
element deleteq(queue_pointer *front)
{
/* delete an element from the queue */
   queue_pointer temp = *front;
   element item;
   if (IS_EMPTY(*front)) {
      fprintf(stderr, "The queue is empty\n");
      exit(1);
   }
   item = temp->item;
   *front= temp->link;
   free(temp);
   return item;
}
```

Program 4.9: Delete from the front 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.

4.4 Polynomials

- Representing Polynomials As Singly Linked Lists
  - The manipulation of symbolic polynomials,has a classic example of list processing.In general, we want to represent the polynomial:

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

    - Where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that

      $e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0$.
  - We will 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 poly_node *poly_pointer;
typedef struct poly_node {
        int coef;
        int expon;
        poly_pointer link;
};
poly_pointer a,b,d;
```

Draw poly_nodes:

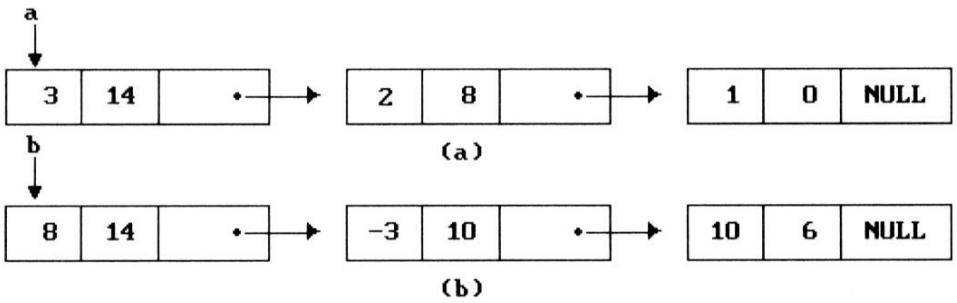


Figure 4.11: Polynomial representation

- 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.
  - 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 *d*,and advance the pointer to the next term in *b*.
  - We take a similar action on *a* if *a->expon* > *b->expon*.

  •Figure 4.12 Generating the first three term of $d = a+b$.

  (next page)

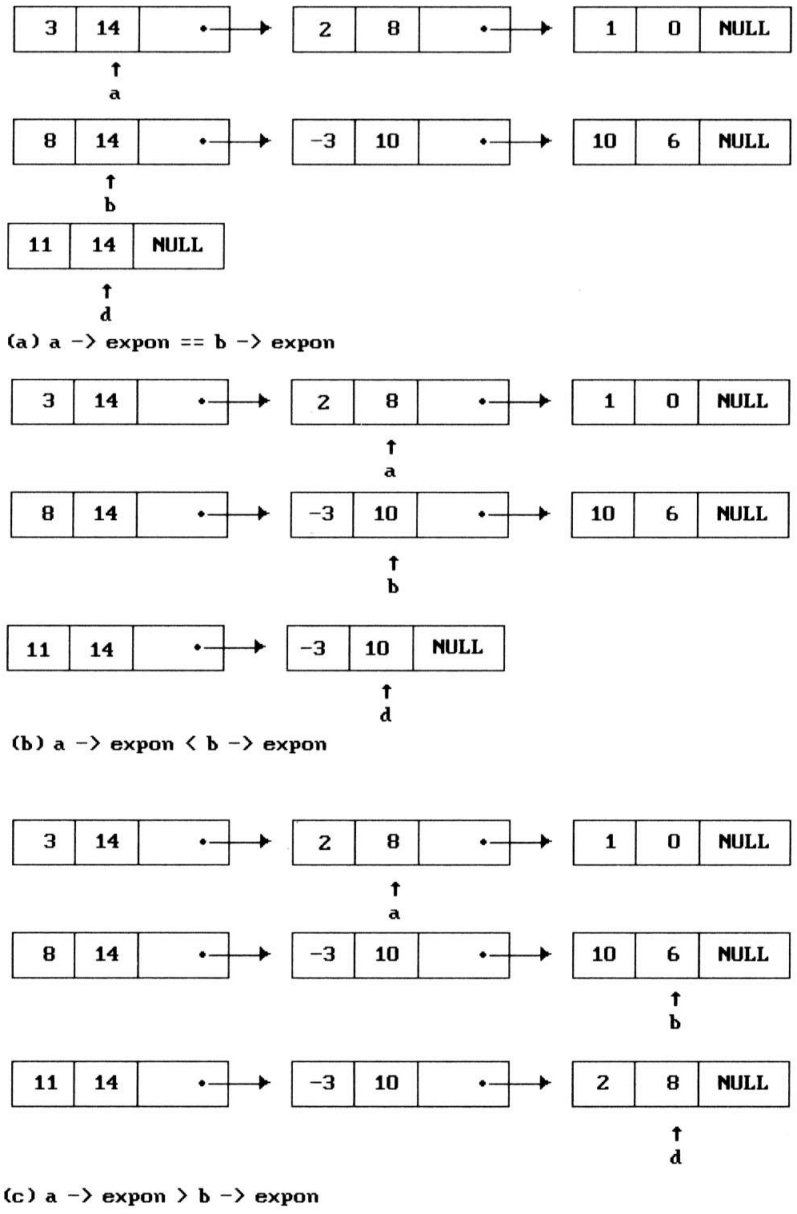


Figure 4.12: Generating the first three terms of $d = a + b$

- 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:
    - coefficient additions
    - exponent comparisons
    - creation of new nodes for *d*
  - 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:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0} \qquad B(x) = b_{n-1}x^{f_{n-1}} + \ldots + b_0 x^{f_0}$$

where $a_i, b_i \neq 0$ and $e_{m-1} > \ldots > e_0 \geqq 0$, $f_{n-1} > \ldots > f_0 \geqq 0$

- The number of coefficient additions varies as:

  $0 \leqq$ number of coefficient addition $\leqq \min\{m,n\}$

  – 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.

  (next page: program4.10)
- Since the total number of term is m+n,the number of iterations and hence the number of exponent comparisons is bounded by m+n.

– For example,m=n:

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

The maximum number of terms in *d* is *m*+*n*,and so no more than *m*+*n* new terms are created.

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{
/* return a polynomial which is the sum of a and b */
   poly_pointer front, rear, temp;
   int sum;
   rear = (poly_pointer)malloc(sizeof(poly_node));
   if (IS_FULL(rear)) {
      fprintf(stderr, "The memory is full\n");
      exit(1);
   }
   front = 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 = front; front = front->link;  free(temp);
   return front;
}
```

Program 4.10: Add two polynomials

- Summary:
  - The maximum number of executions of any statement in ***padd*** is bounded above by m+n.
  - The computing time is O(m+n).
  - If we implement and run the algorithm on a computer,the time it takes will be $c_1m+c_2n+c_3$, where c_1,c_2,$c_3$are constant.
  - Since any algorithm that adds two polynomials must look at each nonzero term at least once,***padd*** is optimal to within a constant factor.

- Erasing polynomials
  - 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:

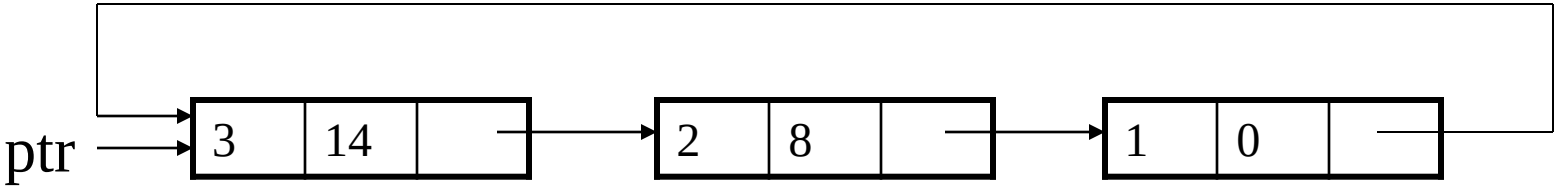
```
Poly_pointer a, b, c, e
:
a = read_poly();
b = read_poly();
c = read_poly();
temp = pmult(a,b);
e = padd(temp,d);
print_poly(e);
```

    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*** frees the nodes in *temp*.

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

- Representing polynomials As Circularly Linked Lists
  - Circular list:
    - The link field of the last node points to the first node in the list.
    - Example:circular representation of *ptr* = $3x^{14}+2x^{8}+1$



  - Chain:
    - A singly linked list in which the last node has a null link .
  - 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 poly_pointer 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 get_node(program4.13) and ret_node (program4.14).

```
poly_pointer get_node(void)
/* provide a node for use */
{
   poly_pointer node;
   if (avail) {
      node = avail;
      avail = avail->link;
   }
   else {
      node = (poly_pointer) malloc(sizeof(poly_node));
      if (IS_FULL(node)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
   }
   return node;
}
```

Program 4.13: *get_node* function

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

Program 4.14: *ret_node* function

– Erase a circular list in a fixed amount of time independent of the number of nodes in the list using ***cerase***(program 4.15).

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

Program 4.15: Erasing a circular list

– Figure4.14 shows the changes involved in erasing a circular list.

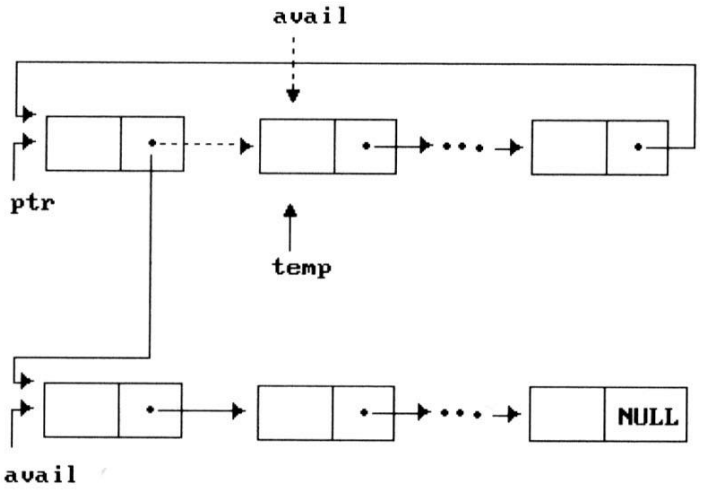
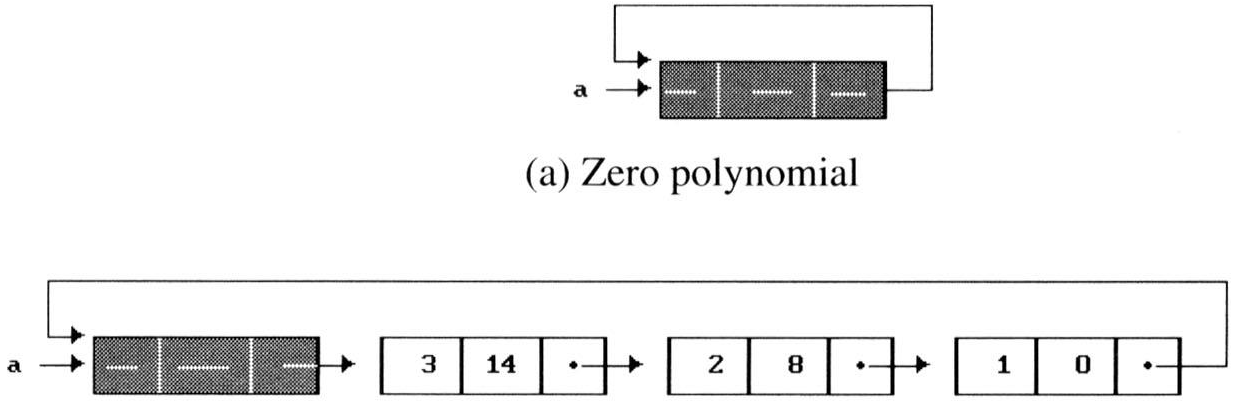


Figure 4.14: Returning a circular list to the avail list

– 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 ***head node*** into each polynomial,that is,each polynomial,zero or nonzero,contains one additional node.
– The *expo*n 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).



(a) Zero polynomial

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

Figure 4.15: Polynomial representations

- For the circular list with head node representation,we may remove the test for (*ptr) from cerase.The only changes that we need to make to *padd*(program4.10) are:
  - (1) Add two variables,starta = a and startb = b.
  - (2) Prior to the **while** loop,assign a = a -> link and b = b-> link.
  - (3) Change the **while** loop to **while**(a != starta && b!= startb).
  - (4) Change the first **for** loop to for(; a != starta ; a = a -> link).
  - (5) Change the second **for** loop to for(; b != startb ; b = b-> link).
  - (6) Delete the lines:

    rear -> link = NULL; /* delete extra initial node */
  - (7) Change the lines:

    temp = front; front = front -> link; free(temp);

    to

    rear -> link = front;

  Thus,the algorithm stays essentially the same,and now handle zero polynomials in the same way as nonzero polynomials.

- We may further simplify the addition algorithm if we set the *expon* field of the head node to –1.
  - Now after we have examined all the nodes of a, starta = a and starta -> expon = -1.
  - Since –1 $\leqq$ b -> expon,we can copy the remaining terms of b by further executions of the **switch** statement.The same is true if we examine all the nodes of b before those of a.This means that we no longer need the additional code to copy the remaining terms.
  - The final algorithm,*cpadd*,takes the simple form given in Program4.16 (next page).

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{
/* polynomials a and b are singly linked circular lists
with a head node. Return a polynomial which is the sum
of a and b */
  poly_pointer starta, d, lastd;
  int sum, done = FALSE;
  starta = a;           /* record start of a */
  a = a->link;          /* skip head node for a and b*/
  b = b->link;
  d = get_node();        /* get a head node for sum */
  d->expon = -1; lastd = d;
  do {
    switch (COMPARE(a->expon, b->expon)) {
      case -1: /* a->expon < b->expon */
            attach(b->coef,b->expon,&lastd);
            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,&lastd);
              a = a->link; b = b->link;
            }
            break;
      case 1:  /* a->expon > b->expon */
            attach(a->coef,a->expon,&lastd);
            a = a->link;
    }
  } while (!done);
  lastd->link = d;
  return d;
}
```

Program 4.16: Adding circularly represented polynomials

- Summary:
  - 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.
  - 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.

4.5 Additional list operations

- Operations for chains
  - Inverting a chain
    - Try out this function with at least three example,an empty list and lists of one and two nodes.
    - For a list of *length* $\geqq 1$ nodes,the **while** loop is executed *length* times and so the computing time is linear or O(*length*).
  - Concatenates two chains
    - Concatenates two chains,ptr1 and ptr2.Assign the list ptr1 followed by the list ptr2.
    - The complexity of this function is O(length of list *ptr1*)

– Inverting a singly linked list:(program4.17)

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

Program 4.17: Inverting a singly linked list

– Concatenating singly linked lists:(program 4.18)

```
list_pointer concatenate(list_pointer ptr1,
                         list_pointer ptr2)
{
/* produce a new list that contains the list ptr1 followed
by the list ptr2. The list pointed to by ptr1 is changed
permanently */
   list_pointer temp;
   if (IS_EMPTY(ptr1)) return ptr2;
   else {
      if (!IS_EMPTY(ptr2)) {
         for (temp = ptr1; temp->link; temp = temp->link)
            ;
         temp->link = ptr2;
      }
      return ptr1;
   }
}
```

Program 4.18: Concatenating singly linked lists

- Operations for circularly linked lists
  - Insert a new node at the front of a circular list.
    - void insert_front(list_pointer *ptr,list_pointer node)

```
/* insert node at the front of the circular list ptr,where ptr is the
 last node in the list */
{
   if(IS_EMPTY(*ptr))   {
   /* list is empty,change ptr to point to new entry */
      *ptr = node;
      node -> link = node;
   }
   else   {
   /* list is not empty,add new entry at front */
       node -> link = (*ptr) -> link;
       (*ptr) -> link = node;
   }
```

– Insert a new node at the rear of a circular list.

- To insert *node* at the rear,we only need to add the additional statement **ptr* = *node* to the else clause of *insert_front* (program4.19).

```
void insert_front(list_pointer *ptr, list_pointer node)
/* insert node at the front of the circular list ptr,
where ptr is the last node in the list */
{
   if (IS_EMPTY(*ptr)) {
   /* list is empty, change ptr to point to new entry */
      *ptr = node;
      node->link = node;
   }
   else {
   /* list is not empty, add new entry at front */
      node->link = (*ptr)->link;
      (*ptr)->link = node;
   }
}
```

Program 4.19: Inserting at the front of a list

- Finding the length of a circular list.

```
int length(list_pointer ptr)
{
/* find the length of the circular list ptr */
   list_pointer temp;
   int count = 0;
   if (ptr) {
      temp = ptr;
      do {
         count++;
         temp = temp->link;
      } while (temp != ptr);
   }
   return count;
}
```

Program 4.20: Finding the length of a circular list

4.6 Equivalence relations

- Definition:
  - A relation,≡,over a set,S is said to be an *equivalence relation* over S *iff* it is symmetric, reflexive, and transitive over S.

- Example: "equal to" is an equivalence relation because
  1) $x = x$
  2) $x = y$ implies $y = x$
  3) $x = y$ and $y = z$ implies $x = z$

應用

- 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 patterns numbered 0 to 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 patterns into the following equivalence classes: {0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

Algorithm

- The algorithm to determine equivalence works in two phases.
  - **The first phase:**

    we read in and store the equivalence pairs <i, j>.
  - **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.

– Use a one-dimensional array, *seq*[n], to hold the head nodes of the *n* lists. The variable *seq*[i] points to the list of nodes that contains every number that is directly equivalent to *i* by an input relation.

– We use the array *out*[n] and the constants TRUE and FALSE to tell us whether or not the object, *i* , has been printed.

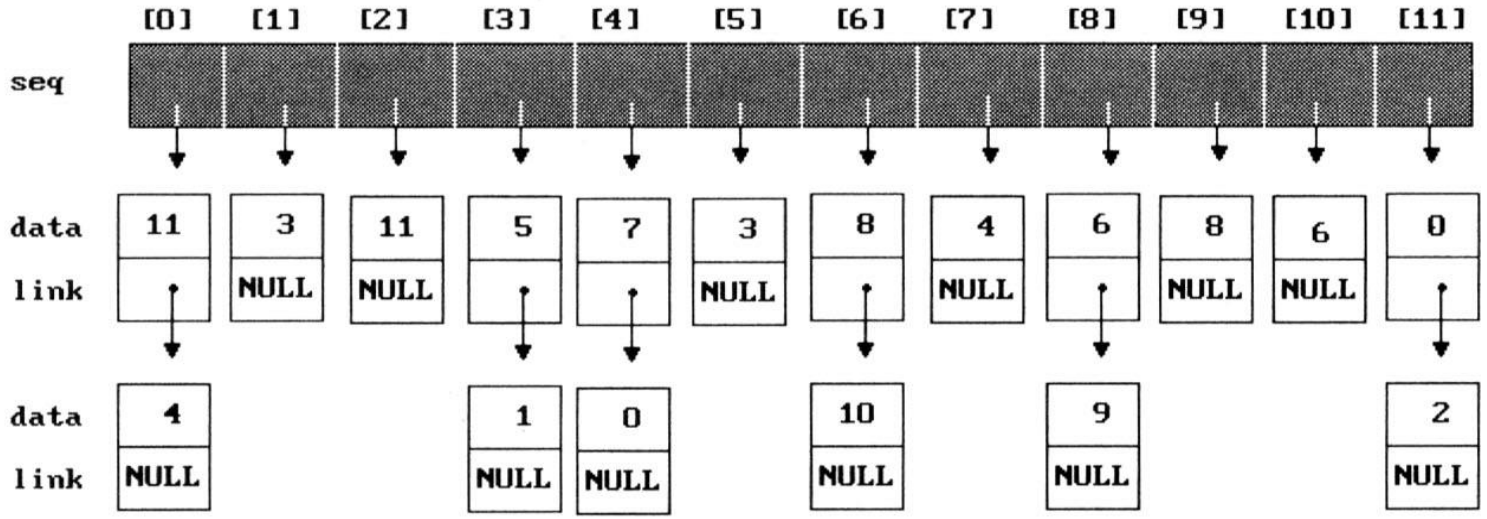


Figure 4.18: Lists after pairs are input

```
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 4.22: A more detailed version of the equivalence algorithm

- Program 4.23

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define IS_FULL(ptr) (!(ptr))
#define FALSE 0
#define TRUE 1
typedef struct node *node_pointer;
typedef struct node {
        int data;
        node_pointer link;
        };
void main(void)
{
   short int out[MAX_SIZE];
   node_pointer seq[MAX_SIZE];
   node_pointer 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);
```

- (continue on next page)

```
    while (i >= 0) {
       x = (node_pointer)malloc(sizeof(node));
       if (IS_FULL(x)) {
          fprintf(stderr,"The memory is full\n");
          exit(1);
       }
       x->data = j;  x->link = seq[i];  seq[i] = x;
       x = (node_pointer)malloc(sizeof(node));
       if (IS_FULL(x)) {
          fprintf(stderr, "The memory is full\n");
          exit(1);
       }
       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 4.23: Program to find equivalence classes

- Analysis of the equivalence program:
  - The initialization of *seq* and *out* takes O(n) time.
  - In phase1 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 pair input.
  - In phase2,we put each node onto the linked stack at most once. Since there are only 2*m* 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 all *n* patterns at least once. Thus, there is no algorithm with a computing time less than O(*m*+*n*).

- The space required by the algorithm is also O(*m*+*n*).

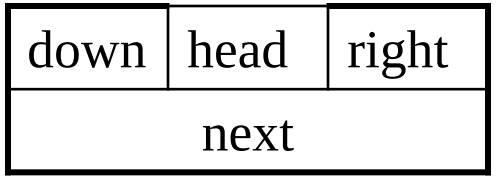
4.7 Sparse matrices

- In this section,we study a linked list representation for sparse matrices.
- In our data representation,we represent each column of a sparse matrix as a circularly linked list with a head node.
  - Each node has a tag field,which we use to distinguish between head nodes and entry nodes.
  - Each head node has three additional fields:*down*,*right*,and *next*(Figure4.19(a)).
    - We use the *down* field to link into a column list.
    - The *right* field to link into a row list.
    - The *next* field links the head nodes together.
  - The head node for row *i* is also the head node for column *i*, and the total number of head nodes is max {number of rows, number of columns}.

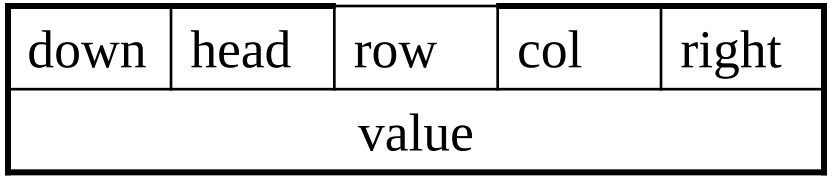
- Each entry node has five fields in addition to the tag field:*row*,*col*,*down*,*right*,*value*(figure4.19(b)).
  - We use the *down* field to link to the next nonzero term in the same column
  - The *right* field to link to the next nonzero term in the same row.
  - Example:if $a_{i,j} \neq 0$,there is a node with tag field = entry ,value = $a_{i,j}$,row = i , and col = j (Figure4.19(c))
- We link this node into the circular linked list for row I and column j.Hence, it is simultaneously linked into two different list.

Figure 4.19 :

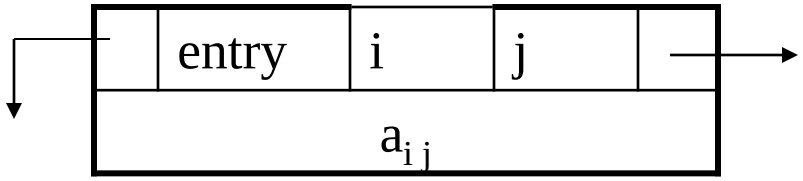
(a) head node:



(b)entry node



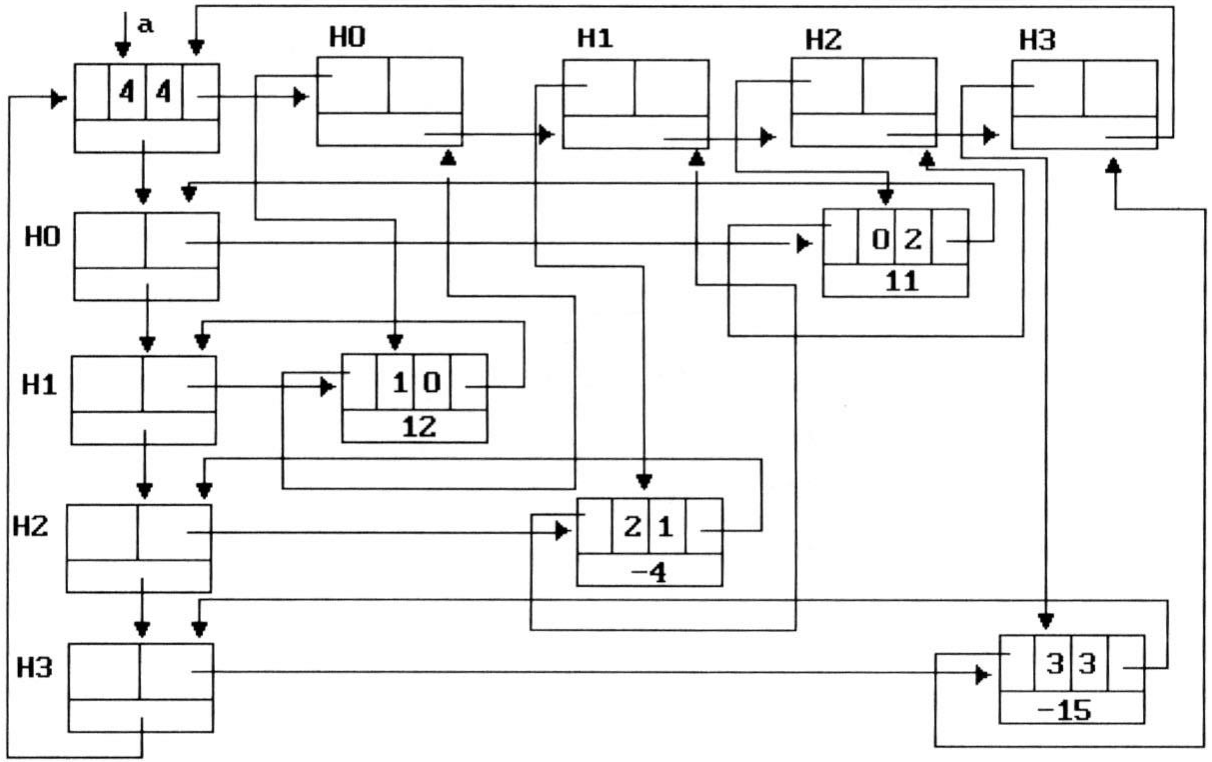
(c)set up for $a_{i\ j}$



- Each head node is in three lists:
  - a list of rows
  - a list of columns
  - a list of head nodes(the list nodes also has a head node that has the same structure as an entry node(Figure 4.19(b))).
- Example:(Figure 4.20 & Figure 4.21(next page))

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$

Figure 4.20: 4×4 sparse matrix a



NOTE: The tag field of a node is not shown; its value for each node should be clear from the node structure.

Figure 4.21: Linked representation of the sparse matrix *a*

- If we wish to represent a *num_rows*∗*num_cols* matrix *num_terms* nonzero terms,then we need max {*num_rows*,*num_cols*}+*num_terms*+1 nodes.
- While each node may require several words of memory,the total storage will be less than *num_rows*·*num_cols* when *num_terms* is sufficiently small.
- In our representation,we use ***union*** to create the appropriate data structure.

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

4.8 Doubly linked lists

- Singly linked lists pose problems because we can move easily only in the direction of the links.
  - For example:suppose that we are pointing to a specific node, say *ptr*,and we want to find the node that precedes *ptr*.We obviously cannot perform this operation efficiently with singly linked list.
- Doubly linked list has at least three fields,a left link field(*llink*),a data field(*item*),and a right link field(*rlink*).
  - The declaration are:

```
typedef struct node *node_pointer;
typedef dtruct node   {
    node_pointer llink;
    element iteml
    node_pointer rlink;
 }
```

- Sample:
  - doubly linked circular with head node:(Figure 4.23)

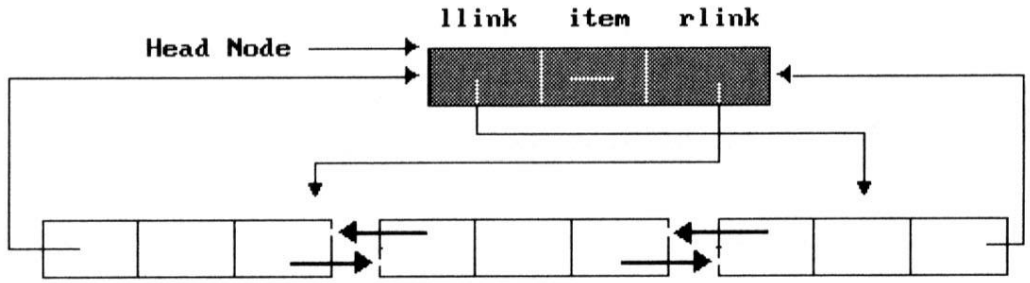


Figure 4.23: Doubly linked circular list with head node

  - Empty double linked circular list with head node(Figure 4.24)

Figure 4.24: Empty doubly linked circular list with head node

- Now suppose that *ptr* points to any node in a doubly linked list.Then:

  ptr = ptr -> llink -> rlink = ptr -> rlink -> llink

- Insert and delete nodes:
  - Insert node:(program 4.28 & Figure 4.25)

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

Program 4.28: Insertion into a doubly linked circular list

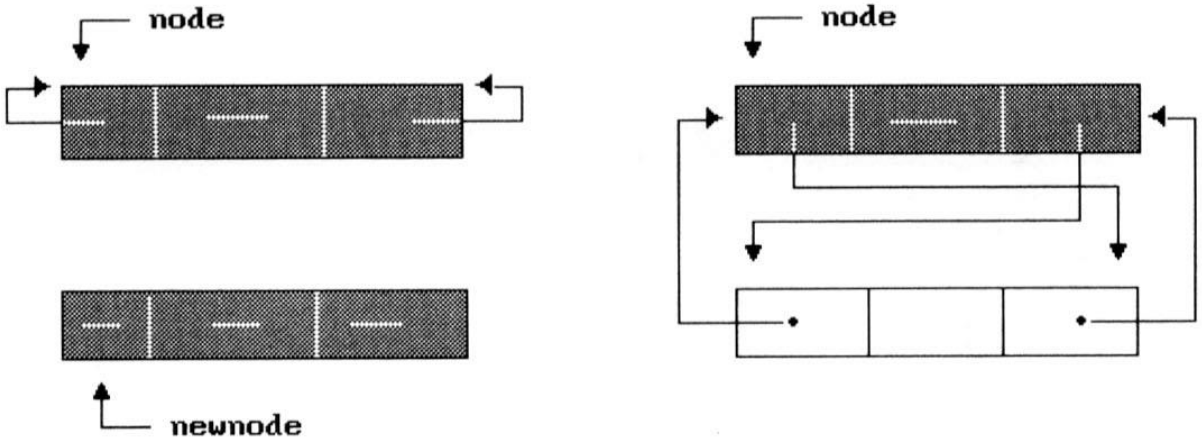


Figure 4.25: Insertion into an empty doubly linked circular list

– Delete node:(program 4.29 & Figure 4.26)

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

Program 4.29: Deletion from a doubly linked circular list

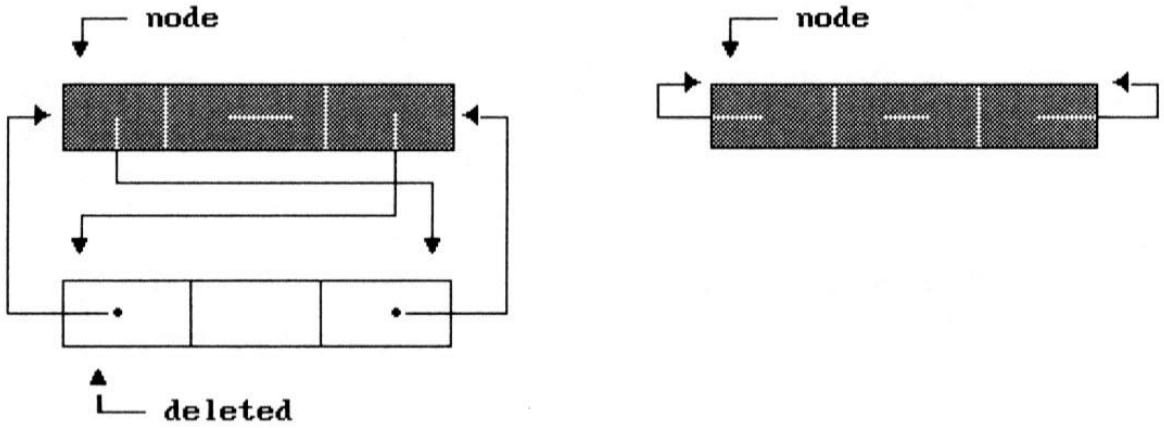


Figure 4.26: Deletion from a doubly linked circular list