Chapter 6

Static List and Linked List

Chapter Outline

6.1 INTRODUCTION

A list is a series of linearly arranged finite elements (numbers) of same type. The data elements are called nodes. The list can be of two types, i.e. basic data type or custom data type. The elements are positioned one after the other and their position numbers appear in

sequence. The first element of the list is known as head and the last element is known as tail.

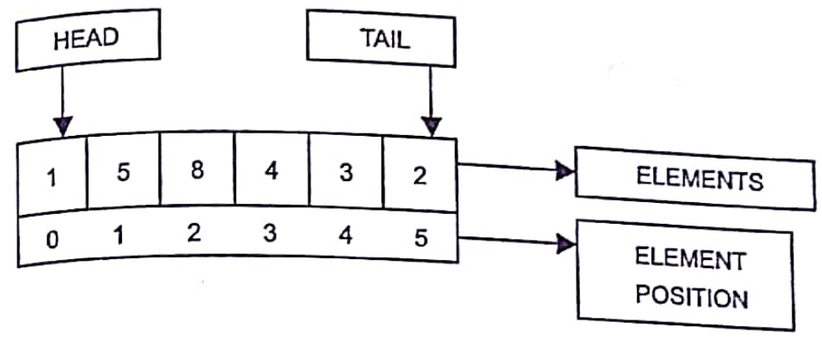


Figure 6.1 Static list

As shown in the above Fig. 6.1, the element 1 is at head position (0th) and element 2 is at tail position (5th). The element 5 is *predecessor* of element 8 and 4 is *successor*. Every element can act as *predecessor* excluding the first element because it does not have predecessor in the list. The list has following properties:

a. The list can be enlarged or reduced from both the ends.
b. The tail (ending) position of the list depends on how long the list is extended by the user.
c. Various operations such as transverse, insertion and deletion can be performed on the list.
d. The list can be implemented by applying static (array) or dynamic (pointer) implementation.

6.2 IMPLEMENTATION OF LIST

There are two methods of implementation of the list: they are static and dynamic.

6.2.1 Static Implementation

Static implementation can be implemented using arrays. It is a very simple method but it has few limitations. Once a size is declared, it cannot be changed during the program. It is also not efficient for memory. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupies the memory space. In both the cases, if we store less arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact number of elements are to be stored.

6.2.2 Dynamic Implementation

The linked list is a major application of the dynamic implementation and the pointers are used for the implementation. The limitations noticed in static implementation can be removed by using dynamic implementation. The memory is utilized efficiently in this method. Hence, it is superior than the static implementation. With pointers, link does not have any restriction on number of elements at run time. The list can be stretched like elastic. Memory is allocated only after node is added or element is accepted in the list. Memory is de-allocated whenever node or element is removed. Dynamic memory management policy is used in the implementation of

linked list and because of this memory is used resourcefully. In addition to that insertion and deletion of a node can be done easily.

Both the above representations are implemented with suitable examples in this chapter.

6.3 TRAVERSAL OF LIST

Traversing a list means to visit every element or node in the list beginning from first to last. The simple list can be created using an array. Its elements are stored in successive memory locations. Consider the following program for creating and displaying the elements of a list:

Example 6.1

Write a program to create a simple list of elements. Display the contents of list and reverse it.

Solution

```
# include <stdio.h>
# include <conio.h>

main()
{
   int sim[5],j;
   clrscr();
   printf ("\nEnter five elements: ");
   for (j=0;j<5;j++)
   scanf ("%d",&sim[j]);

   printf ("\n List: ");
   for (j=0;j<5;j++)
   printf (" %d ",sim[j]);

   printf ("\n Reverse List: ");
   for (j=4;j>=0;j--)
   printf (" %d ",sim[j]);
   getchar();
}
```

OUTPUT

```
Enter five elements: 1 5 9 7 3
List: 1 5 9 7 3
Reverse List: 3 7 9 5 1
```

Explanation This program is simple. Using simple declaration of an array a list can be implemented. Using *for loop* and *scanf()* statements, five integers are entered. The list can be displayed using *printf()* statement. Once a list is created, various operations such as sorting, searching can be applied. These operations are already explained in Chapters 4 and 5. The reader is advised to refer to Chapter 2 for more information on arrays.

6.4 SEARCHING AND RETRIEVING AN ELEMENT

Once a list is created, one can access and perform operations with the elements. In the last program, all the elements are displayed. One can also specify some conditions such as to display numbers after a number 10 or remove the duplicate numbers from the list, finding a specific number or concatenate two lists. If the list contains large elements, it may be difficult to find a particular element and its position. It consumes more time for locating that element. Consider the following program for searching a number in a list:

Example 6.2

Write a program to create a list of integer elements and search the entered number from the list.

Solution

```
# include <stdio.h>
# include <conio.h>
main()
{

int sim[7],j,n,flg;
clrscr();

printf ("\nEnter Seven Integers: ");
for (j=0;j<7;j++)
scanf ("%d",&sim[j]);

printf ("\n Enter Integer to Search: ");
scanf ("%d",&n);

for (j=0;j<7;j++)
{
if(sim[j]==n)
{
flg=1;
printf ("\n Found ! Position of integer %d is %d ", n,j+1 );
break;
}

}
if(flg=1)
printf ("\n Element not found !");

}
```

OUTPUT

```
Enter Seven Integers: 12 12 23 45 56 76 67
Enter Integer to Search: 67
Found ! Position of integer 67 is 7
Or
Enter Integers: 12 45 56 34 233 2345 56
Enter Integer to Search: 555
Element not found !
```

Explanation In this program an array *sim [7]* seven integers are entered. Followed by this, an element is entered to search in the list. This is done by the *for loop* and embedded in the *if statement*. The *if* statement checks the entered element with the list elements. When match is found the number and its positions are displayed.

6.5 PREDECESSOR AND SUCCESSOR

In the list of elements, for any location *n*, *(n–1)* is predecessor and *(n+1)* is successor. In other words, for any location n in the list, the left element is *predecessor* and the right element is *successor*. The first element does not have *predecessor* and last element does not have successor. The Fig. 6.2 shows it.

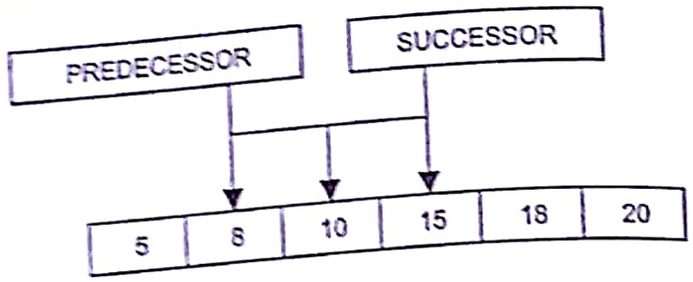


Figure 6.2 Predecessor and successor

The following program displays the predecessor and successor of the entered element from the list.

Example 6.3

Write a program to find predecessor and successor of the entered number in the list.

Solution

```
# include <stdio.h>
# include <conio.h>

main()
{
   int num[8],j,n,k=0;
   clrscr();
   printf ("\n Enter eight elements: ");
   for (j=0;j<8;j++)
   scanf ("%d",&num[j]);

printf ("\n Enter any element from the above: ");
scanf ("%d",&n);

for (j=0;j<8;j++)
{
if (n==num[j])
{
(j>0) ? printf ("\n The Predecessor of %d is %d ",num[j],num[j-1]):
printf (" No Predecessor");
```

Explanation In this program eight elements are entered. The user has to enter an element whose predecessor and successor elements are to be displayed. All the elements of the array are checked with the entered number. When the match is found, the next element is displayed as successor and the previous element is displayed as predecessor. If the element entered is the first element of the list then only successor is displayed. If the entered element is the last then only predecessor is displayed. The above conditions are checked using *conditional operator (?:)*.


```
(j==7) ? printf ("\n No Successor"):
printf ("\n The Successor of %d is %d", num[j],num[j+1]);
break;

}
j++;
}
}
```

OUTPUT

```
Enter eight elements: 1 2 5 8 7 4 4 6
Enter any element from the above: 5
The Predecessor of 5 is 2
The Successor of 5 is 8
```

6.6 INSERTION

Appending is a process in which new element is added at the end of the list. However, insertion of an element can also be done in the list. Insertion means an element is added in between two elements in the list. The insertion can be done at the beginning or at the end or anywhere in the list. Thus, the insertion of an element can be done at various positions in the list.

For successful insertion of an element, the array-implementing list should have at least one empty location. If the array is full, insertion is not possible. The target location where element is to be inserted is made empty by shifting elements downwards by one position and the newly entered element is placed at that location. Consider the following Fig. 6.3(a).

5	7	9	10	12		
0	1	2	3	4	5	6

Figure 6.3 (a)

As shown in Fig. 6.3(a), two empty spaces are available. Suppose, we want to insert 3 in between 7 and 9. All the elements after 7 must be shifted towards the end of the array. The resulting array would be as shown in Fig. 6.3(b).

5	7		9	10	12	
0	1	2	3	4	5	6

Figure 6.3 (b)

The entered number 3 can be assigned to that memory location and the array can be as shown in Fig. 6.3(c).

5	7	3	9	10	12	
0	1	2	3	4	5	6

Figure 6.3 (c)

A program for insertion of an element in a list is illustrated below.

Example 6.4

Write a program to create a list. Insert an element at the specific location.

Solution

```
# include <stdio.h>
# include <conio.h>
main ()
{
   int num[9]={0},j,p,n;
   clrscr();
   printf ("\n Enter eight elements: ");
   for (j=0;j<8;j++)
   scanf ("%d",&num[j]);
   printf ("\n Enter Position and element for insertion:-");
   scanf ("%d %d",&p,&n);
   --p;
   for (j=8;j>=p;j--)
   num[j]=num[j-1];
 num[p]=n;
 printf("\nList after insertion of element\n");
 for (j=0;j<=8;j++)
   printf (" %d ", num[j]);
 getch();
}
```

Explanation In this program an array *num [8]* is declared. The user has to enter eight elements continuously. Thus, the list contains one empty memory location, which is enough to carry insertion operation.

```
OUTPUT
Enter eight elements: 1 2 3 4 5 6 7 8
Enter Position and element for insertion:-6 12
List after insertion of element
1 2 3 4 5 12 6 7 8
```

The element and the position number where element is to be inserted, are to be entered. The user provides the values. Using *for loop* the position of vacant location is obtained. As the user specifies the position number, next all elements are shifted one memory location towards the end of the array. Due to shifting, space is generated and the entered element is placed at that location.

6.7 DELETION

Like insertion, deletion of an element can be done from the list. In deletion, the elements are moved upwards by one position.

5	7	3	9	10	12	
0	1	2	3	4	5	6

Figure 6.4 (a) Before deletion

The deletion of any element is very simple. Suppose, we want to remove element 3, then 3 is replaced with 9, 9 is replaced with 10, 10 is replaced with 12 and finally 12 is replaced with null. Thus, shifting of element is done. The Fig. 6.4 (a) and (b) shows status of static list before and after deletion of an element.

5	7	9	10	12		
0	1	2	3	4	5	6

Figure 6.4 (b) After deletion

Example 6.5

Write a program to create a list of integer elements. Delete the specific element from the list.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <process.h>
main()
{
int num[8]={0},j,k=0,p,n,count;
clrscr();
printf ("\n Enter elements ( 0 to exit ): " );
for (j=0;j<8;j++)
{
scanf ("%d",&enum[j]);
if ( num[j]==0)
  {
  count=j;
  break;
  }
}
```

Explanation In this program, elements are entered by the user. User can enter maximum eight elements. The element, which is to be removed, is also entered. The *while loop* calculates the position of the element in the list. The second *for loop* shifts all the elements next to the specified element one position up towards the beginning of array. The element, which is to be deleted, is replaced with successor element. The last element is replaced with zero. Thus, the empty spaces are generated at the end of the array.

2018-LOL

```
printf ("\n Enter an element to remove: ");
scanf ("%d",&n);
while (num[k]!=n)
k++;
for (j=k;j<count;j++)
num[j]=num[j+1];
num[j]=0;
for (j=0;j<count-1;j++)
printf (" %d ", num[j]);
}
```

```
OUTPUT
Enter elements ( 0 to exit ): 5 8 9 4 2 3 4 7
Enter an element to remove: 9
5 8 4 2 3 4 7
```

6.8 SORTING

Sorting and searching are two oft-used operations in data structures. In every walk of life one comes across these processes. Various methods would be applied for sorting and searching of records or elements. The following program explains sorting by exchange method. The reader can refer to chapters 10 and 11 where comprehensive information is provided about sorting and searching.

Example 6.6

Write a program to sort entered numbers by exchange method.

Solution

```
# include <stdio.h>
# include <conio.h>

main ()
{
   int num[8]={0};
   int k=0,h,a,n,tmp;
   clrscr();
   printf ("\n Enter Eight Numbers: ");
   for (a=0;a<8;++a)
   scanf ("%d",&num[a]);

   while (k<7)
   {
   for (h=k+1;h<8;++h)

  if (num[k]>num[h])
```

Explanation In this program eight numbers are entered. The *while loop* and *for loop* are used to access, compare and exchange the elements. Every element of an array is compared with all successive elements. When small element is found it is stored at the beginning location in the array using *tmp* variable.


```
  {
  tmp=num[k];
  num[k]=num[h];
  num[h]=tmp;

  }
  k++;
  }
  printf ("\n Sorted array: ");

  for (k=0;k<8;++k)
  printf (" %d ",num[k]);
  }
```

```
OUTPUT
Enter Eight Numbers: 4 5 8 7 9 3 2 1
Sorted array: 1 2 3 4 5 7 8 9
```

6.9 MERGING LIST

Merging is a process in which two or more lists are merged to form a new list. The procedure involved in merging is very straightforward. Consider the following program for merging of two lists:

Example 6.7

Write a program to create two-array list of integers. Sort and store elements of both the list in the third list.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <math.h>

main()
{
int m,n,p,sum=0;
int listA[5],listB[5],listC[10]={0};
clrscr();

printf ("\n Enter five elements of first list: ");

for (m=0;m<5;m++)
{
  scanf ("%d",&listA[m]);
  if (listA[m]==0) m--;
  sum=sum+abs(listA[m]);
```

Explanation In this program three arrays *listA []*, *listB []* and *listC []* are declared. Using *for loops* elements in *listA []* and *listB []* are declared. The sum of all ten elements entered in both the lists are taken in variable *sum*. The *while* and nested *for loop* checks corresponding elements of both the lists.

```
    }
    printf ("\n Enter five elements of second list: ");

    for (n=0,n<5;n++)
    {
    scanf ("%d",&listB[n]);
    if (listA[n]==0) n--;
    sum=sum+abs(listB[n]);
  }
  p=n=m=0;
 m=m-sum;

 while (m<sum)
 {
   for (n=0;n<5;n++)
 {
  if (m==listA[n] || m==listB[n])
  listC[p++]=m;

 if (m==listA[n] && m==listB[n])
 listC[p++]=m;
  }
  m++;
  }

  puts (" Merged sorted list: ");
  for (n=0;n<10;++n)
 printf (" %d ",listC[n]);
 }
```

OUTPUT
Enter five elements of first list: 1 5 4 -3 2
Enter five elements of second list: 9 5 1 2 10
Merged sorted list:
-3 1 1 2 2 4 5 5 9 10

a. If one of the corresponding elements is same, that element is stored in the *listC []* array.
b. If both the corresponding elements are same, they are stored in successive locations in the *listC []*.

The value of m is initially zero. The sum obtained is again subtracted from m. This is because if negative numbers are entered, they are to be considered in the sorting. For example, the value of *sum=20*. In real execution, the sum may be different depending on integers entered.

Value of *m* would be = –20 (as per *m=m*–20 when *m*=0).

Thus, in the *while loop* value of m varies from –20 to 20. In this range all the entered elements are covered. The value of *m* changes from –20 to 20, i.e. –20,–19 up to +20 in ascending order. Thus, the same order is applied while saving elements in the *arrayC[]*.

6.10 LINKED LIST

A linked list is a dynamic data structure. It is an ideal technique to store data when the user is not aware of the number of elements to be stored. The dynamic implementation of list using pointers is also known as *linked list*. Each element of the list is called as *node*. Each element points to the next element. In the linked list a node can be inserted or deleted at any position. Each node of linked list has two components. The first component contains the information or any data field and second part the address of the next node. In other words, the second part holds address of the next element. This pointer points to the next data item. The pointer variable member of the last record of the list is generally assigned a NULL value to indicate the end of the list. Fig. 6.5 indicates the linked list.

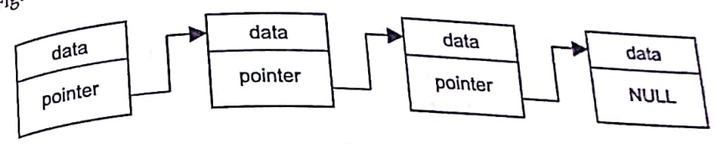


Figure 6.5 Linked list

The basic data type in the linked list can be *int*, *float* and user defined data types can be created by struct *cla*. The link structure as well as a pointer of structure type to the next object in the link is observed in it.

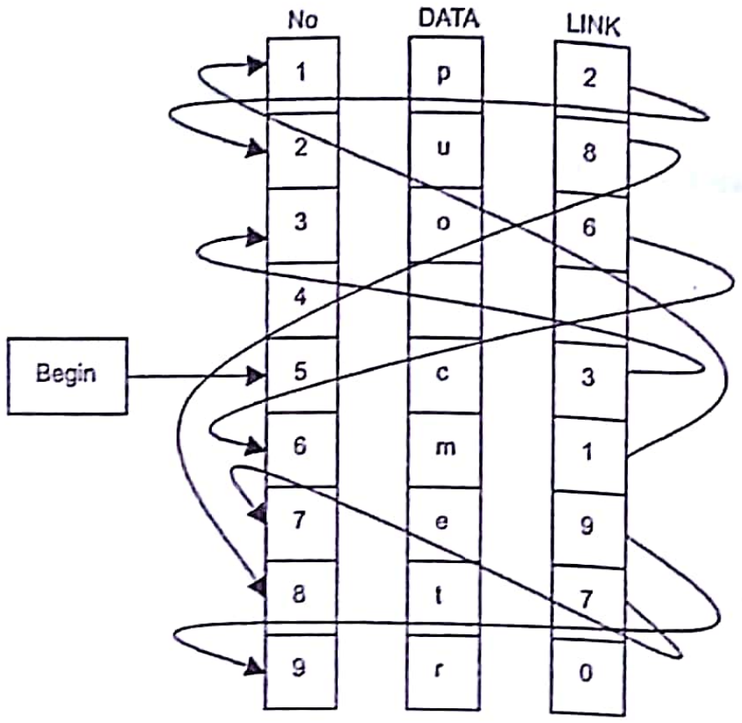
A simple example describing the operation of linked list is illustrated in the following section.

6.10.1 Illustration of Linked List for Storing a String

In this example, a linked list containing a string of characters to be stored in the memory, is described. The linked list has two field data, i.e. actual data stored and links, i.e. pointing to the next node of the linked list. These two fields are shown with two arrays such as data[j] and link[j] where 'j' is the position of an element. A variable begin is initiated which contains the beginning location of the list. The field '*No.*' is taken for numbering the nodes of the linked list. It is not the part of the linked list. It is given simply for understanding. In the link[j] '0' (zero) indicate the NULL pointer which is end of the string. In this type of lists the nodes or elements are not stored in the successive memory locations. Many links have been shown between data and link.

The nodes of the linked list are not ordered sequentially. The characters are stored dynamically with their link fields. The link field shows the link to the next node of the linked list. The information obtained from the above linked list is 'computer'. Its explanation is as follows:

a. Begin =5, data[5] is 'c' and link[5]= '3'.
b. Link[3]=6 and data[3]= 'o'.



c. Link[6]=1 and data[6]= 'm'.
d. Link[2]=8 and data[1]= 'p'.
e. Link[2]=8 and data[2]= 'u'.
f. Link[8]=7 and data[8]= 't'.
g. Link[7]=9 and data[7]= 'e'.
h. Link[9]=NULL (0) and data[9]= 'r'.

The string stored in the memory as per above listing is 'computer'.

6.11 IMPORTANT TERMS

We have already discussed in previous sections that a linked list is a non-sequential collection of elements called nodes. These nodes are nothing but objects. These nodes are declared using structure or classes. Every node has two fields and they are:

1. Data field: In this field, the data or values are stored and processed.
2. Link field: This field holds address of the next data element of the list. This address is used to access the successive elements of the list.

In the linked list the ordering of elements is not done by their physical location in the memory but by logical links, which are stored in the link field.

Node The components or objects which form the list are called nodes.

Null Pointer The link field of the last record is assigned a NULL value instead of any address. It means NULL pointer not pointing to any element.

External Pointer It is a pointer to the starting node. The external pointer contains the base, i.e. address of first node. Once a base address is available, its next successive nodes can be accessed.

Empty List When there is no node in the list, it is called as empty list. If the external pointer were assigned a value NULL, the list would be empty.

6.11.1 Static List vs. Linked List

1. The static list is implemented using arrays. The linked list uses pointer in which memory can be allocated or de-allocated during program execution. The linked list is more efficient in memory management. The linked list can be expanded or shrunk as per requirement whereas in static list once an array is defined its size remains the same throughout the program.
2. Once an array is declared, its size remains the same. The memory allocated for array may be extra or insufficient. The user cannot determine the exact amount of memory required. Due to this, the program execution may not be safe. The pointer has intrinsic flexibility to enlarge or reduce the capacity of storage. We are acquainted with pointer arithmetic operations increment/decrement, which enables pointer to access any memory location. The use of pointer to point the next node in the linked list data structure indicates that elements which are logically contiguous due to linear organization need not be physically contiguous in the memory. This type of memory allocation is called linked allocation.
3. The elements are stored in order logically as well as physically. Array elements are stored in successive memory locations. This is not the case in the linked list. Insertion and deletion with static list are time consuming and shifting of elements is done to rearrange the numbers in order. In the linked list the insertion and deletion operations are done very easily. The element can be deleted by de-allocating the memory of that particular element. The insertion can be done by allocating space at the specified position.
4. A list has been created to hold an ordered set of elements without knowing the number of nodes. The simple method to create link is to enlarge each node and it should contain reference of the next node. This is called *singly linked linear list* or *one-way chain* as shown in Fig. 6.6. The arrow (first) indicates the address of the first node.

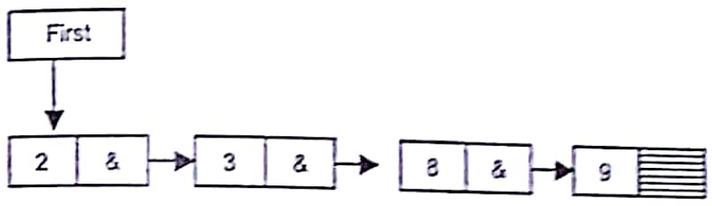


Figure 6.6 Singly linked linear list

5. By using linked list more complex data structures such as circular linked list, doubly linked list, stack and queues can be formed.
6. A pointer plays an important role in the dynamic implementation of linked list. The interpretation of pointer is an address, which is in-built, and the programmer need not take care of how the addresses are assigned in the memory. Mostly, all computer systems use addresses to know the instruction to be executed. Pointers of any data type always

are of same length and occupy equal space in the memory, i.e. 2 bytes only. The pointer is always of unsigned integer type. Consider the following program:

Example 6.8

Write a program to demonstrate the size of pointers of different data types.

Solution

```
# include <stdio.h>
# include <conio.h>

struct node { long int a; }*s;

main()
{
   int *p;
   float *f;
   clrscr();

   printf ("\n Integer Pointer Size=%d",sizeof(p));
   printf ("\n Float Pointer Size=%d",sizeof(f));
   printf ("\n Structure pointer Size=%d",sizeof(s));
}
```

Explanation In this program pointers of *integer*, *float* and *structure* type are declared. The *sizeof ()* operator returns the size of pointers. The size of any pointer is always two bytes. The address is always of unsigned integer having range 0 to 65535.

```
OUTPUT
Integer Pointer Size=2
Float Pointer Size=2
Structure pointer Size=2
```

7. In the above program we have demonstrated that pointers occupy only two bytes irrespective of the type of data. This property of pointers makes it possible for the pointer to operate in carrying out the referencing of structure in a homogeneous manner with easy allocation methods. The prototype of structure does not matter. In the sequential allocation, the address of any node in the list can be calculated if the storage allocation must be in same pattern.

Example 6.9

Write a program to create a simple linked list.

Solution

```
# include <stdio.h>
# include <conio.h>
struct list
{
```



```
int n;
list *p;
};
main()
{
struct list item0,item1, item2;
item0.n=3;
item0.p=NULL;
item1.n=5;
item1.p=&item2.n;
item0.n=7;
item0.p=&item1.n;
clrscr();
printf("\n Linked list elements are: ");
printf(" %d ",item0.n);
printf(" %d ",*item0.p);
printf(" %d ",*item1.p);
printf(" %d ",*item2.p);
}
```

Explanation In this program the structure list is declared with two elements *int n* and **p*. The pointer **p* recursively points to the same structure. The struct *item0*, *item1* and *item2* are three variables of type *list*. Consider the initialization.

```
OUTPUT
Linked list elements are: 7 5 3 0
```

```
item2.n=3;
item2.p=NULL;
```

The item2 is the third (last) node of the list. The pointer p is initialized with null because next to this no node is present and thus no need to point any address.

```
item1.n=5;
item1.p=&item2.n;
```

In this node, n is assigned with five. The pointer points to the data field of the next node i.e. item2.n (7).

```
item0.n=7;
item0.p=&item1.n;
```

This is the first node. The n is assigned 7 and pointer points to data field of *item1*, i.e. 5. Fig. 6.7 simulates the operation more clearly.

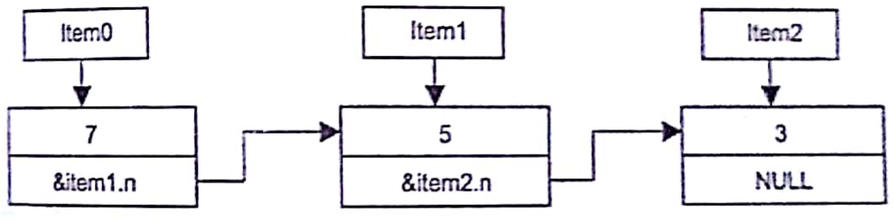


Figure 6.7 Linked list

Example 6.10

Write a program to display string using linked list.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>
struct link
{
char str[30];
struct link *next;
};

typedef struct link linked;
main ()
{
linked *begin, node1,node2,node3, node4;
int j;
clrscr();
printf("\nEnter four words in four lines:-\n ");
begin=(struct link*) malloc(sizeof(linked));
begin=&node1;
gets(node1.str);
node1.next=&node2;
gets(node2.str);
node2.next=&node3;
gets(node3.str);
node3.next=&node4;
gets(node4.str);
node4.next='\0';
printf ("Output: ");

printf ("%s ",begin->str);

do
{
begin=begin->next;
printf ("%s ",begin->str);
}
while (begin->next!='\0');
}

OUTPUT
Data
Structure
with
C
Output: Data Structure with C
```

Explanation In this program, structure link is declared. The *str []* char array is used to store string. The pointer *begin and objects *node1*, *node2*, *node3* and *node4* are declared. In the address field of object address of other object is stored. For example, in address field of *node1* address of *node2* is stored. In address field of *node2*, address of *node3* is stored and so on. Similarly, using *gets ()* in *str []* array string (entered by user) is stored. Using *while loop* successive addresses are accessed and complete string is displayed.

6.11.2 Types

The linked lists are classified in the following types:

Singly Linked List In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner. An example of a singly linked list can be cited as train in which two successive compartments are connected as shown in Fig. 6.8.

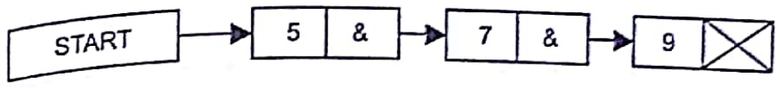


Figure 6.8 Singly linked list

Doubly Linked List In this type of linked list each node holds two-pointer field. In the singly linked list, address of only next element is pointed. Unlike that, in doubly linked list addresses of next as well as preceding elements are linked with current node. An example of doubly linked list is shown in Fig. 6.9.

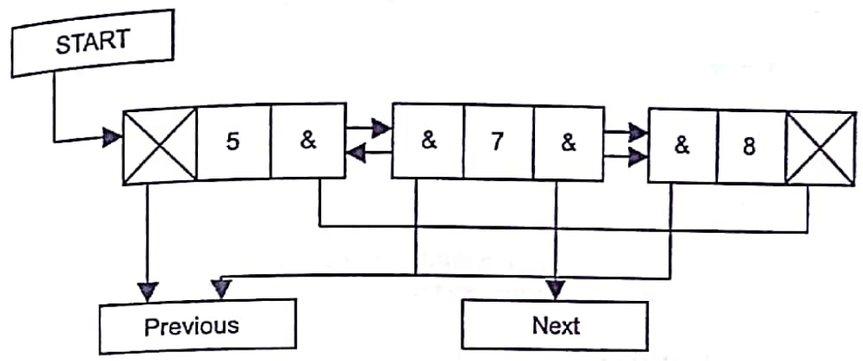


Figure 6.9 Doubly linked list

A Circular Linked List Recall that in circular list the first and last elements are adjacent. This type of list has neither end node *nor starting node*. A linear single linked list can be circled by storing the address of first node in the link field of last node as shown in Fig. 6.10.

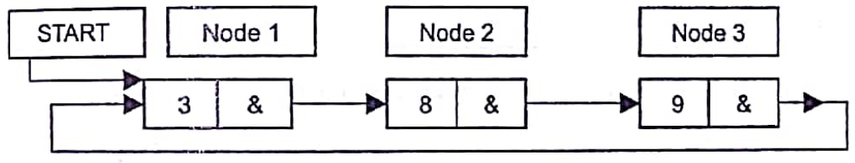


Figure 6.10 A circular linked list

A Circular Doubly Linked List In this type of linked list each node contains three fields: two link fields and one data field. The link field holds address of previous and

next elements. The address of end node is linked to first node and vice-verse. The Fig. 6.11 shows the circular double link list.

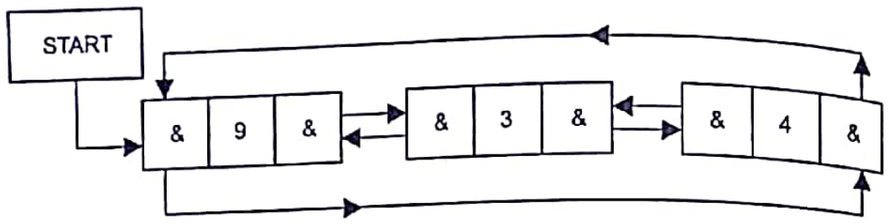


Figure 6.11 A circular doubly linked list

6.12 MEMORY ALLOCATION AND DE-ALLOCATION

Malloc (): This library function is used to allocate memory space in bytes to the variable. The function reserves bytes of requested size and returns the base address to pointer variable. The prototype of this function is declared in the header file *alloc.h* and *stdlib.h*. One of the header files must be included in the header. The syntax of the *malloc()* function is as follows:

Syntax: pnt=(data type*) malloc(size);

Here, pnt is a pointer.

Example: pnt =(int *) malloc(20);

In the above statement, 20 bytes are allocated to integer pointer *pnt*. In addition, there are other various memory allocation functions and its complete description is out of the scope of this book.

free(): This function is used to release the memory allocated by the *malloc ()* function.

Syntax: free(pointer variable)
Example: free(pnt)

The above statement releases the memory allocated to pointer *pnt*. Consider the following program based on this concept:

Example 6.11

Write a program to demonstrate the use of malloc () and free () functions.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <alloc.h>
main ()
{
char *str;
str=(char *) malloc(6);
clrscr();
printf("Enter string:-");
scanf("%s",str);
printf("\n str=%s",str);
free(str);
}
```

```
OUTPUT
Enter string:-India
str=India
```


Explanation In this program a character pointer variable *str* is declared. The *malloc ()* function allocates seven bytes to character pointer **str*. The string *"India"* is assigned to pointer *str* and it is displayed. The *free ()* function releases the memory from pointer *str*.

6.13 OPERATIONS ON LINKED LISTS

The following primitive operations can be performed with linked list:

1. Creation
2. Display
   a. Ascending
   b. Descending
3. Traversing
4. Insertion
   a. At beginning
   b. Before or after specified position
5. Searching
6. Concatenation
7. Merging

Creation The linked list creation operation involves allocation of structure size memory to pointer of the same structure. The structure must have a member which points recursively to the same structure. In this operation, constituent node is created and it is linked to the link field of preceding node.

Traversing It is the procedure of passing through (visiting) all the nodes of the linked list from starting to end. When any given record is to be searched in the linked list, traversing is applied. When the given element is found, the operation can be terminated or continued for next search. The traversing is a common procedure and it is necessary because operations like insertion, deletion, listing cannot be carried out without traversing linked list.

Display The operation in which data field of every node is accessed and displayed on the screen. In the display operation from beginning to end, link field of every node is accessed

which contains address of next node. The data of that field is displayed. When NULL is detected the operation ends. Each node points to the next node and this recursion fashion enables the pointer to reach the successive elements.

The above three operations are common and every program involves these operations. Hence, separate program is not given.

Searching The searching is a process, in which a given element is compared with all the linked list elements. The *if* statement is placed and it checks entire list elements with the given element. When an element is found it is displayed.

Besides the above operations additional operations such as concatenation and merging of list can be done.

6.14 SINGLY LINKED LIST

Recall that linked list is a dynamic data structure with ability to expand and shrink as per the program requirement. The singly linked list is easy and straightforward data structure as compared to other structures. By changing the link position other type of linked list such as circular, doubly linked list can be formed. For creating linked list the structure is defined as follows,

```
struct node
{
 int number;
 struct node *p;
};
```

The above structure is used to implement the linked list. In the *number*, variable entered numbers are stored. The second member is pointer to the same structure. The pointer **p* points to the same structure. Here, though the declaration of struct node has not been completed, the pointer declaration of the same structure type is permitted by the compiler. However, the variable declaration is not allowed. This is because, the pointers are dynamic in nature whereas variables are formed by early binding. The declaration of objects inside the struct leads to preparation of very complex data structure. This concept is called object composition and its detailed discussion is out of the scope of this book.

We are familiar with the array and we know the importance of base address. Once a base address is obtained, successive elements can also be accessed. In the linked list, list can be created with or without *header node*. The *head* holds the starting address.

6.15 LINKED LIST WITH HEADER

The following steps are used to create linked list with header:

1. Three pointers, i.e. header, first and rear are declared. The header pointer is initialized with *NULL*. For example, header=*NULL* where, header is a pointer to structure. If it remains *NULL*, it implies that the list has no element. Such list is known as *NULL* list or empty list, which is shown in Fig. 6.12(a).

Figure 6.12(a) Empty list

2. In the second step, memory is allocated for the first node of the linked list. For example, the address of first node is 1888. An integer, say 3, is stored in the variable num and value of header is assigned to pointer *next*.

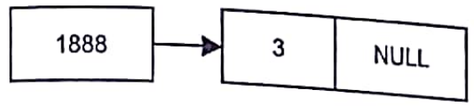


Figure 6.12(b) Link list

The address of first node is initialized by both the header and rear.
The statement would be,

```
header=first;
rear=first;
```

3. The address of pointer *first* is assigned to pointers *header* and *rear*. The *rear* pointer is used to identify the end of the list and to detect the NULL pointer.
4. Again, create a memory location suppose 1890 for the successive node.

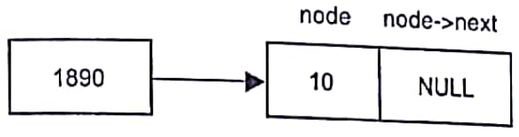


Figure 6.12(c)

5. Link the element of 1890 by assigning the value of node *rear->next*. Move the *rear* pointer to the last node.

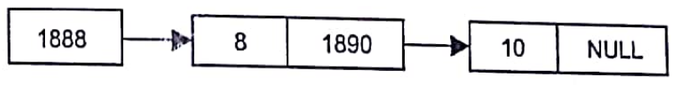


Figure 6.12(d)

The following program explains the above concept.

Example 6.12

Write a program to create linked list with header.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
   int num;
   struct num *next;
} *header,*first,*rear;

main ()
{
   void create ( void);
   clrscr();

   create ();

   printf ("\n Linked list elements are: ");

   while (header!=NULL)
 {
   printf (" %d ",header->num);
   header=header->next;
 }

}

void create ( void )
{
   struct num *node;
   printf ("\n Enter number ( 0 exit ): ");

 if (header==NULL)
 {
   first=(struct num*)malloc(sizeof(struct num));
   scanf ("%d",&first->num);
   first->next=header;
   header=first;
   rear=first;
}

while (1)
```

Explanation This program is an example of linked list with header. Three structure pointers **header*, **first*, and **rear* are declared after structure declaration. Initially, these pointers are NULL because they are declared as global. The create function is used to create linked list nodes. Inside the function *create ()* another structure pointer **node* is defined and its scope is local to the same function. The procedure for creating first and later successive nodes is different. The *if* statement checks the value of header. The value of header is NULL. The *malloc ()* function allocates memory to pointer first and the entered number is stored in variable num of the node. In the same *if* block, both the pointers header and rear are assigned the value of first. Once, the first node is created, next time the execution of *if* block is not required. The *while loop* iterated continuously and successive nodes are created by allocating memory to pointer node. Consider the following statements:


```
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf ("%d",&node->num);
  if(node->num==0) break;

  node->next=NULL;
  rear->next=node;
  rear=node;

}
}
```

```
OUTPUT
Enter number ( 0 exit ): 1 3 4 8 7 9 0
Linked list elements are: 1 3 4 8 7 9
```

1. `node->next=NULL;`

The above statement, assigns NULL to the pointer next of current node. The user can initiate this step if no further elements are desired in the linked list.

2. `rear->next=node;`

The rear pointer keeps track of last node; the address of current node (node) is assigned to link field of previous node.

3. `rear=node;`

Before creating next node, the address of last created node is assigned to pointer rear, which is used for statement (2). In function main (), using while loop the elements of linked list are displayed.

The pointer header is very useful in the formation of linked list. The address of *first* node (1888) is stored in pointer header. The value of header remains unchanged until it turns out to be NULL. The starting location of the list only can be determined by pointer header only.

```
while (header!=NULL)
{
printf (" %d ",header->num);
header=header->next;
}
```

6.15.1 Traverse a List with Header

We are familiar with traverse operation, which involves visiting all the elements of the queue. The above *while () loop* and associated statement are used to traverse and display the elements of the list.

6.16 LINKED LIST WITHOUT HEADER

In the previous topic, we discussed how linked list is created using header. The implementation of linked list without header is similar to the linked list with header. The only difference in manipulation is that in the header list pointer header contains address of first node. In the list without header list, pointer first itself is starting of the linked list. Consider the following program:

Example 6.13

Write a program to create a linked list without header.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
   int num;
   struct num *next;
} *first,*rear;

main()
{
   void create ( void);
   clrscr();

   create ();
   printf ("Double linked list elements: ");

   while (first!=NULL)
 {
 printf (" %d",first->num);
 first=first->next;
 }
}

void create ( void )
{
   struct num *node;
   printf ("\n Enter number (0 to exit): ");

if (first==NULL)
{
   first=(struct num*)malloc(sizeof(struct num));
```

Explanation In this program only two pointers, *first* and *rear*, are declared. They are global pointers; hence, their contents are NULL. The function *create ()* is invoked and the rest of the execution of the program is the same as the previous program.


```
scanf ("%d",&first->num);
first->next=first;
rear=first;
}

while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf ("%d",&node->num);
  if (node->num==0) break;
  node->next=NULL;
  rear->next=node;
  rear=node;

}
}
```

```
OUTPUT
Enter number (0 to exit): 1 5 9 8 7 4 2 0
Double linked list elements: 1 5 9 8 7 4 2
```

6.16.1 Traverse a List Without Header

In the list without header, the lists are traversed with the first node as shown in the above code.

Example 6.14

Write a program to create a single linked list and display the elements.

Solution

```
# include <stdio.h>
# include <conio.h>

struct node
{
  int number;
  struct node *p;
} *h,*f,*t;

main()
{
  char c;
  clrscr();
  f=NULL;
do
{
```

Explanation In this program, the structure node is declared. The *structure node* contains integer variable number and pointer to the same structure *p. The structure pointer variables *h, *f and *t are also declared. In function *main ()*, variable of character type c is defined. The pointer f is initialized to NULL.

```
h=(struct node *)malloc(sizeof(struct node));
printf ("Enter a number: ");
scanf ("%d",&h->number);

if (f!=NULL)
{
t->p=h;
t=h;
}
else f=t=h;
fflush(stdin);
printf ("Continue (y/n): ");
scanf ("%c",&c);

} while (c=='y');

t->p=NULL;
t=f;
printf ("\nThe list elements are: ");
while (t!=NULL)
{
printf (" %d ",t->number);
t=t->p;
}
}
```

```
OUTPUT
Enter a number: 5
Continue (y/n): y
Enter a number: 3
Continue (y/n): y
Enter a number: 8
Continue (y/n): y
Enter a number: 9
Continue (y/n): n
The list elements are: 5 3 8 9
```

In the *do–while loop*, using *malloc ()* function memory equal to size of structure node *(4 byte)* is allocated to pointer *h*. The entered number is stored in the number variable. The *if* statement checks the pointer *f*. If it is NULL, the *h* is assigned to other two pointers, i.e. *f* and *t*. Here, *f* is first node. *h* is header and *t* is used for temporary storage. The user has to enter '*y*' to enter next elements. First time, the value of *f* is NULL and hence, else block is executed. The purpose of assignment of *h* to *f* and *t* is as follows:

1. In *if ()* block *t* is used to assign address of newly created next node to the current, i.e. *h*.
2. Only for the first time else block is executed and in which address of *h* (when first node is created) is assigned to *f*. *f* contains address of first node.

When the user finishes entering number, *t->p=NULL;* this statement assigns NULL to pointer *t* and link ends here. The address of the first *node (f)* is assigned to *t*. Using *while loop*, the pointer *t* is accessed and elements are accessed. When NULL pointer is encountered, the *while loop* terminates.

In the following Fig. 6.13 the number contains the actual data of the node. The structure node pointer is used to point to the next node in the linked list hence it is called as the linked field. The link field refers to itself in recursive form.

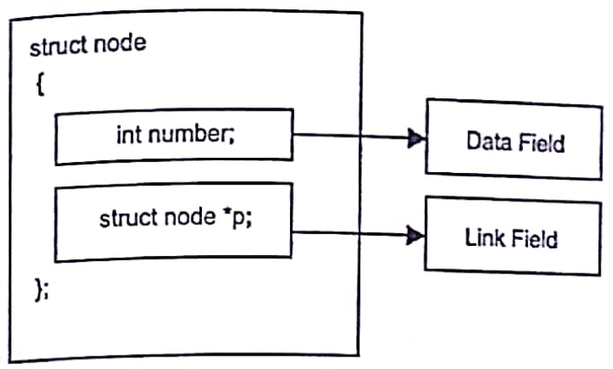


Figure 6.13 Structure for linked list creation

Example 6.15

Write a program to create a list. Display the list up to specified element.

Solution

```
#include <stdio.h>
#include <conio.h>

struct node
{
  int number;
  struct node *p;
} *h,*f,*t;

main ()
```

Explanation This program is based on searching operation. This is same as program 6.14. In this program, the specific element is compared with every element of the linked list. When the given element is found in the list, the *while loop* terminates.

```
{
   char c,n;
   clrscr();
   f=NULL;
   do
   {
   h=(struct node *)malloc(sizeof(struct node));
   printf ("Enter a number: ");
   scanf ("%d",&h->number);
   if (f!=NULL)
   {
   t->p=h;
   t=h;
}
   else f=t=h;
   fflush(stdin);
   printf ("Continue (y/n): ");
   scanf ("%c",&c);
} while (c=='y');
t->p=NULL;
t=f;
printf ("\n Enter a element to search: ");
scanf ("%d",&n);
printf ("\nThe list elements are: ");

while (t!=NULL)
{
printf (" %d ",t->number);
if (t->number==n)
break;
t=t->p;
}
}
```

```
OUTPUT
Enter a number: 1
Continue (y/n): y
Enter a number: 3
Continue (y/n): y
Enter a number: 5
Continue (y/n): y
Enter a number: 4
Continue (y/n): y
Enter a number: 9
Continue (y/n): n
Enter a element to search: 4
The list elements are: 1 3 5 4
```


6.17 INSERTION IN THE LINKED LIST

Insertion of an element in the linked list leads to several operations. The following steps are involved in inserting an element:

a. Creation of node: Before insertion, the node is created. Using *malloc()* function memory space for node is booked.
b. Assignment of data: Once the node is created, data values are assigned to members.
c. Adjusting pointers: The insertion operation changes the sequence. Hence, according to the sequence, the address of next element is assigned to inserted node. The address of current node (inserted) is assigned to previous node.

The insertion of node can be done in different positions in the list by the following ways:

a. Insertion of the node at the starting: The created node is inserted before the first element. After insertion, the newly inserted element will be the first element of the linked list. In this insertion only the contents of successive node's pointer are changed.
b. Insertion at the end of the list: A new element is appended at the end of the list. This operation is easy as compared to other two (a) and (c) operations. In this insertion only the contents of previous node's pointer is changed.
c. Insertion at the given position in the list: In this operation, the user has to enter the position number. The given position is counted and the element is inserted. In this insertion contents of both previous and next node's pointer are altered.

6.18 INSERTION OF NODE AT START

Inserting an element at the beginning involves updating links between link fields of two pointers. After insertion of new node, the previously existing nodes are moved to the front. Fig. 6.14 (a) and (b) and program 6.16 illustrates the insertion at the starting.

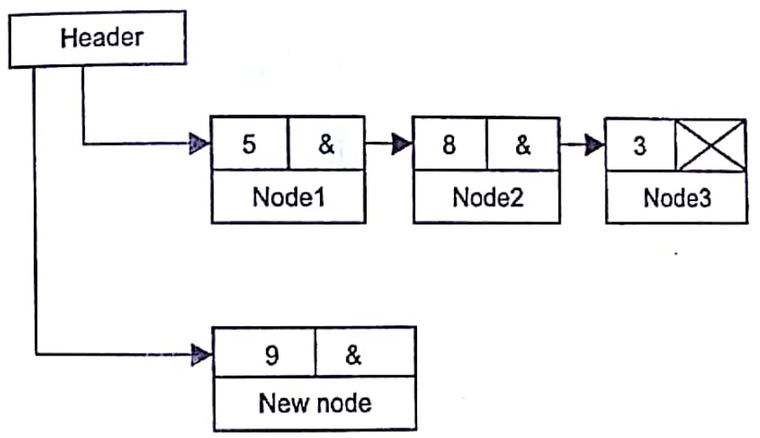


Figure 6.14 (a) Before insertion

The arrow indicates the location of insertion of new node.

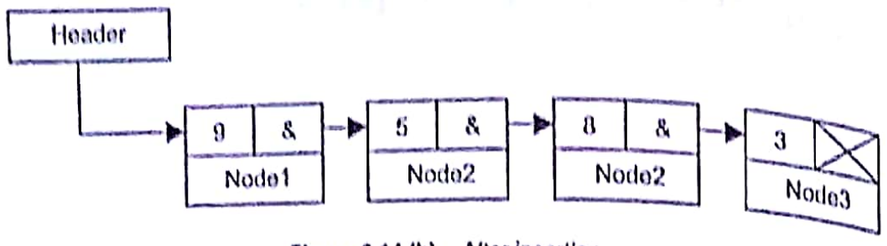


Figure 6.14 (b) After insertion

After insertion, the new node will be first node and its link field points to the second element, which was ex-first element.

The program given below explains the insertion of an element in the list.

Example 6.16

Write a program to insert an element at the beginning.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
    int num;
    struct num *next;
} *header,*first,*rear;

main()
{
    void atbeg(void);
    void create ( void);
    void show (void);
    clrscr();
    printf ("\n Operation creation:");
    create ();
    show();
    atbeg();
    printf ("\n The elements after insertion: ");
    show();
    atbeg();
    show();
    atbeg();
    show();
```

Explanation The creation of linked list is same as explained in program 6.15. Here, an element is inserted at the beginning. The function void *atbeg ()* performs this task. In this function two pointers **node* (local to this function) and **t* are declared. The *malloc()* function allocates memory to pointer *node*. The entered element is placed in the newly created node.


```
}
void create ( void )
{
  struct num *node;
  printf ("\n Enter numbers( 0 to exit): ");

if (header==NULL)
{
  first=(struct num*)malloc(sizeof(struct num));
  scanf("%d",&first->num);
  first->next=header;
  header=first;
  rear=first;
}

while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf("%d",&node->num);
  if(node->num==0) break;
  node->next=NULL;
  rear->next=node;
  rear=node;
}
}

void atbeg()
{
  struct num *node,*t;
  node=(struct num*) malloc(sizeof(struct num));
  printf ("\n Insert an element at starting: ");
  scanf("%d",&node->num);
  t=first;
  header=node;
  header->next=t;
  first=header;
}

void show()
{
  printf ("\n Linked list elements are: ");

  while (header!=NULL)
{
  printf (" %d ",header->num);
  header=header->next;
```

```
}
}
```

```
OUTPUT
Operation creation:
Enter numbers( 0 to exit): 1 2 3 4 0
Linked list elements are: 1 2 3 4
Insert an element at starting: 5
The elements after insertion:
Linked list elements are: 5 1 2 3 4
Insert an element at starting: 9
Linked list elements are: 9 5 1 2 3 4
Insert an element at starting: 3
Linked list elements are: 3 9 5 1 2 3 4
```

Consider the following statements:

```
t=first;
header=node;
header->next=t;
first=header;
```

The above four statements adjust the pointer links:

t=first This statement assigns contents of *first* to pointer *t*. This is essential because the newly created node will become first. Hence, before changing pointer *first*, it is assigned to pointer *t*.

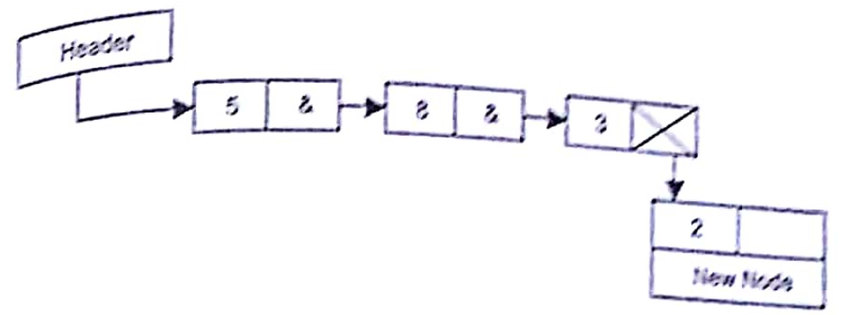
Header=node We know that header always points to the first node. The newly created node, that is to be inserted at starting is assigned to pointer header.

header->next=t The pointer *t* holds address of first element (before insertion). After insertion, it will be second. The address of second node is assigned to first. The pointer header holds address of first node and pointer *t* holds address of second element. Thus, these two successive elements are linked by this statement.

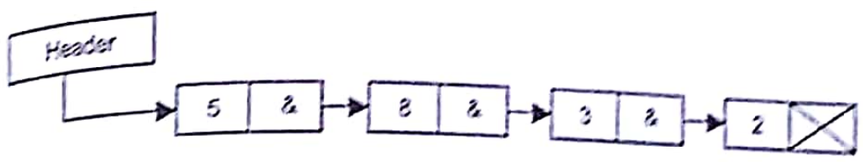
first=header Before execution of this statement the first points to second node. If we remove this statement, only one element can be inserted. If we try to insert second element, it will be replaced by the first element. After assignment of address of header to first, the insertion can be carried out successfully.

6.19 INSERTION OF NODE AT END

A new element is inserted or appended at the end of existing linked list. The address of newly created node is linked to previous node that is NULL. The new node link field is assigned a NULL value. The Fig. 6.15 shows the insertion operation at the end.



(a) Before insertion



(b) After insertion

Figure 6.15 Insertion of element at the end

Example 6.17

Write a program to insert an element at the end of the linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

struct node
{
  int num;
  struct node *next;
}*header,*first,*rear;

main()
{
  void atend(void);
  void create ( void);
  void show (void);
  clrscr();
  printf("\n Operation creation:");
  create();
```

Explanation In this program the function *atend ()* is used to insert element at the end of the list.

```
    show();
    atend();
    show();
    atend();
    show();
}
void create ( void )
{
    struct num *node;
    printf ("\n Enter numbers( 0 to exit): ");

 if (header==NULL)
 {
    first=(struct num*)malloc(sizeof(struct num));
    scanf ("%d",&first->num);
    first->next=header;
    header=first;
    rear=first;
 }

 while (1)
 {
    node=(struct num*) malloc(sizeof(struct num));
    scanf ("%d",&node->num);
    if (node->num==0) break;
    node->next=NULL;
    rear->next=node;
    rear=node;
 }
}

void atend()
{
    struct num *node,*t;
    t=rear;
    rear=(struct num*) malloc(sizeof(struct num));
    printf ("\n Insert an element at end: ");
    scanf ("%d",&rear->num);
    t->next=rear;
    rear->next=NULL;
}
void show()
{
    printf ("\n Linked list elements are: ");

    while (header!=NULL)
{
```



```
    printf (" %d ",header->num);
    header=header->next;
}

header=first;
}
```

```
OUTPUT
Operation creation:
Enter numbers( 0 to exit): 1 2 3 0
Linked list elements are: 1 2 3
Insert an element at end: 4
Linked list elements are: 1 2 3 4
Insert an element at end: 5
Linked list elements are: 1 2 3 4 5
```

Consider the following statements:

`t=rear` This statement assigns address of last node to pointer *t*.

`rear=(struct num*) malloc(sizeof(struct num))` After assigning address of *rear* pointer to *t*, new memory location is allocated to pointer *rear*. The new element entered is stored in the data field of pointer *rear*.

`t->next=rear` Now, the pointer *t* holds the last but one node of the linked list. The address of new inserted end node is assigned to link field of node through pointer *t*.

`rear->next=NULL` The rear pointer holds address of last node and its link field is assigned a NULL value.

6.20 INSERTION OF NODE AT A GIVEN POSITION

Insertion of node can be done at a specific position in the linked list. Fig. 6.16 and the program explain insertion of a node at the specific position in the linked list. If we want to insert a node at third position, then observe Figs 6.16(a) and 6.16(b).

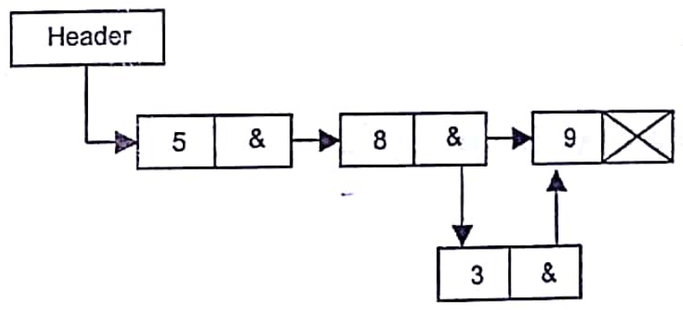


Figure 6.16a. Formed linked list

Node is inserted at third position and after insertion, the linked list would be

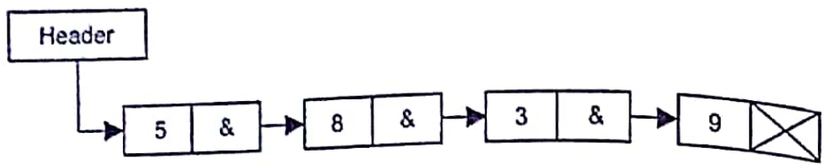


Fig. 16.16b. Linked list after insertion

Figure 6.16 Linked list after and before insertion

Example 6.18

Write a program to insert an element at a specific position.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
   int num;
   struct num *next;
} *header,*first,*rear;

int count;
main ()
{
   void atpo(int);
   void create ( void);
   void show (void);
   int p;
   clrscr();

   printf ("\n Operation creation:");
   create ();
   show();
   printf ("\n Enter position (upto %d )number:",count );
   scanf ("%d",&p);
   atpo(--p);
   show();
}

void create ( void )
{
```

Explanation In this program the operation creation and display of linked nodes is similar as explained in the last programs. Here, we are going to discuss insertion at a specified position carried out by a function *atpo()*.


```
struct num *node;
printf ("\n Enter numbers( 0 to exit): ");
if(header==NULL)
{
first=(struct num*)malloc(sizeof(struct num));
scanf("%d",&first->num);
first->next=header;
header=first;
rear=first;
}
while (1)
{
node=(struct num*) malloc(sizeof(struct num));
scanf("%d",&node->num);
if(node->num==0) break;
node->next=NULL;
rear->next=node;
rear=node;
}
}

void atpo(int k)
{
struct num *nw,*su,*pr;
int c=1;

while (header!=NULL)
{
c++;
header=header->next;

if(k==1) pr=first;
if(c==k) pr=header;
if(c==(k+1)) su=header;
}

nw=(struct num*) malloc(sizeof(struct num));
printf ("\n Enter an element: ");
scanf("%d", &nw->num);

pr->next=nw;
nw->next=su;
header=first;
}

void show()
{
```

```
    printf ("\n Linked list elements are: ");

    while (header!=NULL)
{
    printf (" %d ",header->num);
    count++;
    header=header->next;
}
header=first;
}

OUTPUT
Operation creation:
Enter numbers( 0 to exit): 1 2 3 4 5 0
Linked list elements are: 1 2 3 4 5
Enter position (upto 5 )number:3
Enter an element: 8
Linked list elements are: 1 2 8 3 4 5
```

The position number specified by the user after reducing value by one is passed to function *atpo ()*. The variable *k*, formal argument of *p*, gets this value. Another variable *c* is declared in the body of *atop ()*. The variable *c* is incremented and successive nodes of the linked list are accessed by the statement *header=header->next;*

Consider the statements

a. *if (k==1) pr=first;*
b. *if (c==k) pr=header;*
c. *if (c==(k+1)) su=header;*

The insertion of node can be done in between two nodes and hence the preceding and succeeding nodes must be stored in separate pointer variables. When the user specifies two, the entered value is reduced by one and becomes one and stored in variable *k*.

When value of *k* is one, i.e. the address of *first* node is stored in the pointer *pr*. When *k* is one the statement (b) has no reason to consider. The address of succeeding node is obtained in the statement (c). The address obtained is stored in the pointer variable *su*. Thus, by using following statements proper links are connected:

a. *pr->next=nw;*
b. *nw->next=su;*
c. *header=first;*

Before execution of the above statements, the new node is created using *nw* pointer. The *nw* is a node to be inserted.

The address of data field of node *nw* is linked to the previous node (*pr*) by the statement *pr->next=nw.*

The address of succeeding node is stored in the linked field of new node *nw* by the statement *nw->next=su.*

6.20.1 Counting Nodes

Sometimes, it may be essential to know how many nodes exist in the linked list. The linked list elements can be counted and displayed. The solution is very easy. In the last few programs, the procedure applied for displaying elements can be applied here. A counter variable can be inserted there to count the nodes. One can get the number of nodes simply by incrementing the counter variable. Consider the following program for counting the total number of nodes in the list.

Example 6.19

Write a program to count the total nodes of the linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct num
{
  int num;
  struct num *next;
}*header,*first,*rear;

main()
{
  void create ( void);
  int count=0;
  clrscr();

  create ();
  printf ("\n Linked list elements are: ");

  while (header!=NULL)
  {
  printf (" %d ",header->num);
  header=header->next;
  count++;
  }
  printf ("\n Total Nodes are: %d ",count);
}
void create ( void )
{
  struct num *node;
  printf ("\n Enter number: ");
  if (header==NULL)
  {
    first=(struct num*)malloc(sizeof(struct num));
    scanf ("%d",&first->num);
```

Explanation This program is same as the last few programs. The *count* variable is placed in the *while loop* and incremented. The *count* variable displays the number of nodes.

```
        first->next=header;
        header=first;
        rear=first;
    }
    while (1)
    {
        node=(struct num*) malloc(sizeof(struct num));
        scanf ("%d",&node->num);
        if (node->num==0) break;
        node->next=NULL;
        rear->next=node;
        rear=node;
    }
}
```

```
OUTPUT
Enter number: 1 2 3 4 5 6 7 8 0
Linked list elements are: 1 2 3 4 5 6 7 8
Total Nodes are: 8
```

6.20.2 Deletion

An element that is to be deleted from the linked list is to be searched first. For this, the traversing operation must be carried out thoroughly on the list. Deleting a node from the linked list means

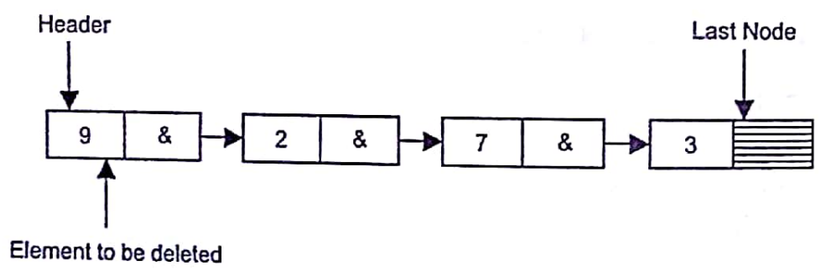
1. Deleting the first node
2. Deleting the last node
3. Deleting a specified node.

If first node is to be deleted, then the second node is assigned to header.

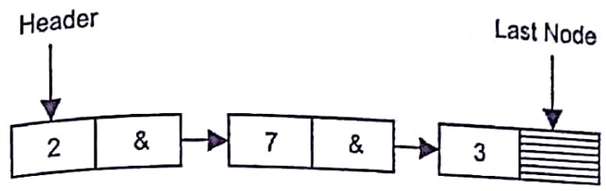
If the last node is to be deleted, NULL value is assigned to link field of the last but one node.

If the node which is to be deleted is in between the linked list then access the predecessor and successor node of the specified node and link them. Fig. 6.17 shows the deletion of first node.

1. Deleting the first node



(a) Before deletion



(b) After deletion

Figure 6.17 Deletion operation with linked list

Example 6.20

Write a program to remove elements from the beginning of the linked list.

Solution

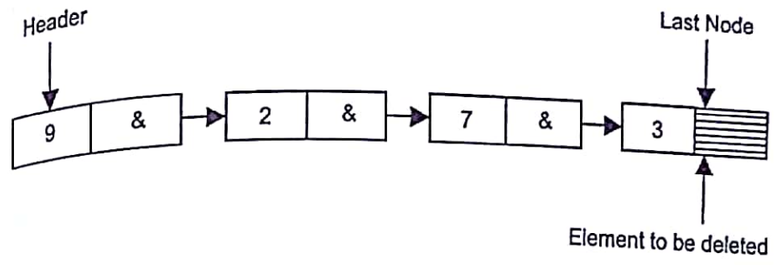
```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

struct num
 {int num;
struct num *next;
}*header,*first,*rear,*k;

main()
{
 void erase(void);
 void create ( void);
 void show (void);
 clrscr();

printf ("\n Operation creation:");
create ();
show();
printf ("\n After deletion: ");
erase();
show();
erase();
show();
erase();
show();
}

void create ( void )
{
```

Explanation In this program *erase ()* function is used to remove the beginning node of the linked list. The creation of linked list is similar as discussed in the last few program. In the *erase ()* function another object **s* is declared. The pointer header is assigned to pointer **s*. The pointer **s* occupies same amount of memory like header. The *free ()* function is used to release the memory. The memory allocated with pointer **s* is released. The address of second node is assigned to header and first node. Thus, whenever the function *erase ()* is invoked, the beginning element is removed and the updated list is displayed by the function *show ()*.

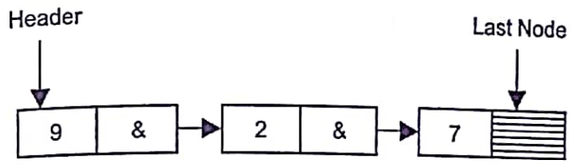
```
struct num *node;
printf ("\n Enter numbers( 0 to exit): ");
if (header==NULL)
{
first=(struct num*)malloc(sizeof(struct num));
scanf ("%d",&first->num);
first->next=header;
header=first;
rear=first;
}
while (1)
{
node=(struct num*) malloc(sizeof(struct num));
scanf ("%d",&node->num);
if (node->num==0) break;
node->next=NULL;
rear->next=node;
rear=node;
}
}
void erase()
{
   struct num *s;
   s=header;
   free(s);
   header=first->next;
   first=first->next;
}
void show()
{
printf ("\n Linked list elements are: ");
while (header!=NULL)
{
printf (" %d ",header->num);
header=header->next;
}

}
```

```
OUTPUT
Operation creation:
Enter numbers( 0 to exit): 1 2 3 4 5 0
Linked list elements are: 1 2 3 4 5
After deletion:
Linked list elements are: 2 3 4 5
Linked list elements are: 3 4 5
Linked list elements are: 4 5
```


2. Deleting the last node
This operation is shown in Fig. 6. 18.



(a) Before deletion



(b) After deletion

Figure 6.18 Deletion operation with linked list

Example 6.21

Write a program to remove last elements of the linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

int c;
struct num
{
  int num;
  struct num *next;
}*header,*first,*rear,*k;

main()
{
void create ( void);
```

Explanation This program demonstrates the deletion from the end of the list. Here, the *show ()* function performs the deletion function and after the function is executed, the rear pointer of the list is freed and an address of previous node is assigned to it. The *show ()* function also displays the elements.

```
void show (void);
clrscr();
printf ("\n Operation creation:");
create ();
show();
printf ("\n After deletion: ");
show();
show();
show();
}

void create ( void )
{
struct num *node;
printf ("\n Enter numbers( 0 to exit): ");
if (header==NULL)
{
first=(struct num*)malloc(sizeof(struct num));
scanf ("%d",&first->num);
first->next=header;
header=first;
rear=first;
c++;
}
while (1)
{
node=(struct num*) malloc(sizeof(struct num));
scanf ("%d",&node->num);
if (node->num==0) break;
c++;
node->next=NULL;
rear->next=node;
rear=node;
}
}
void show()
{
struct num *pr;
int k=c;
printf ("\n Linked list elements are: ");
while (header!=NULL)
{
printf (" %d ",header->num);
header=header->next;
k--;
if (k==1)
{
```



```
free(rear);
rear=header;
header=first;
break;
}
}
c--;
}
```

```
OUTPUT
Operation creation:
Enter numbers( 0 to exit): 1 5 9 7 5 3 2 0
Linked list elements are: 1 5 9 7 5 3
After deletion:
Linked list elements are: 1 5 9 7 5
Linked list elements are: 1 5 9 7
Linked list elements are: 1 5 9
```

3. Deleting the specific node

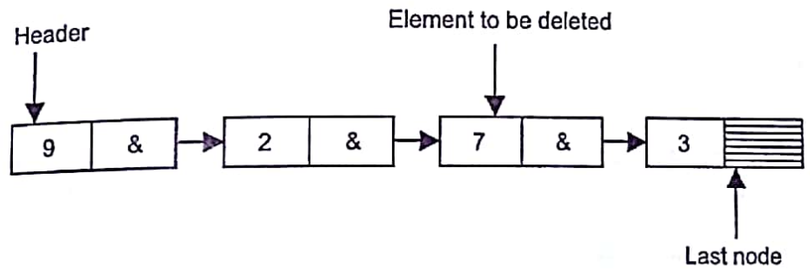


Figure 6.19 Deleting a specific node

So far, we have studied how to remove the first and last node of the list. Now, we want to remove the node randomly from the list. This case is different from the last two cases. In this deletion operation, when a specific node is removed, the previous and next nodes are linked. The memory of the given node is released. The removal of node is shown in the Fig. 6.19.

Example 6.22

Write a program to remove a specific element from the list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
```

```
# include <process.h>
struct num
{
   int num;
   struct num *next;
} *header,*first,*rear,*k;
int count;

main ()
{
   void remove1(int);
   void create ( void);
   void show (void);
   int n;
   clrscr();
   printf ("\n Operation creation:");
   create ();
   show();
   printf ("\n Enter position of number for removal < %d ",count);
   scanf ("%d",&n);
   remove1(n);
   getch();
}
void create ( void )
{
   struct num *node;
   printf ("\n Enter numbers( 0 to exit): ");
   if (header==NULL)
   {
   first=(struct num*)malloc(sizeof(struct num));
   scanf ("%d",&first->num);
   first->next=header;
   header=first;
   rear=first;
   count =1;
}
while (1)

 node=(struct num*) malloc(sizeof(struct num));
 scanf ("%d",&node->num);
 if (node->num==0) break;
 count++;
 node->next=NULL;
rear->next=node;
rear=node;
```

Explanation In this program the element specified by the user is removed. After entering the number, the list of number is displayed. User has to specify only element number. Using *for loop* the specified location is searched. The address of next element is assigned to previous element by skipping the element to be deleted. Thus, the link between previous and next element is joined.


```
void remove1(int j)

{
int p,h;
void show(void);
j--;

p=j-1;
if (j==0)
{
header=header->next;
show();
}
for ( h=0;h<count;h++)
{
if (h==p)
{
header->next=header->next->next;
header=first;
show();
}
header=header->next;
}
}

void show()
{
printf ("\n Linked list elements are: ");
while (header!=NULL)
{
printf (" %d ",header->num);
header=header->next;
}
header=first;
}
```

OUTPUT

```
Operation creation:
Enter numbers( 0 to exit): 4 8 6 5 9 7 4 0
Linked list elements are: 4 8 6 5 9 7 4
Enter position of number for removal < 7 4
Linked list elements are: 4 8 6 9 7 4
```

6.21 REVERSING THE SINGLY LINKED LIST

Conceptually, the singly linked list can be traversed in only forward direction. The structure of singly linked list contains only one pointer, pointing to the next node. The same list can be displayed in reverse fashion using the following program.

Example 6.23

Write a program to reverse the singly linked list.

Solution

```
# include <stdio.h>
# include <conio.h>
struct node
{
 int value;
 struct node *next;
 };
 void create (void);
 void reverse(void);
 void show (void);
 struct node *rear;
 int nodes;
 struct node *head;

main ()
{
    clrscr();
    create();
    reverse();
    show ();
}
void create ( )
{
    struct node *item;
    printf ("Enter numbers( 0 to exit): ");
 if (head==NULL)
 {
    head=(struct node*)malloc(sizeof(struct node));
    scanf ("%d",&head->value);
    head->next=NULL;
    rear=head;
 }
 while (1)
 {
    item=(struct node *)malloc(sizeof(struct node));
    scanf ("%d",&item->value);
    if (item->value==0) break;
    item->next=NULL;
    rear->next=item;
    rear=item;
 }
}
```

Explanation In this program the pointer **first* is global and hence has an initial value NULL. The logic of the program is very simple. The three pointers **prv, *cur, *next;* are used for swapping of the addresses. The end of the list is obtained and the address of previous is assigned to first. Thus, the link list is reversed. The pointer head now points to the last node and every node contains an address of the previous node instead of next node.


```
void reverse()
{
struct node *prv, *cur, *next;
if (head==NULL)
{
printf ("List is empty");
return;
}
cur=head;
prv =NULL;
while (cur !=NULL)
{
next=cur->next;
cur->next=prv;
prv=cur;
cur=next;
}
head=prv;
}
void show ( )
{
printf ("\n Reverse list is: ");
while (head!=NULL)
{
nodes++;
printf (" %d ",head->value);
head=head->next;
}
nodes=0;
}
```

```
OUTPUT
Enter numbers( 0 to exit): 1 2 3 4 5 6 7 0
Reverse list is: 7 6 5 4 3 2 1
```

6.22 CONCATENATION OF TWO LISTS

Concatenation of two lists means adding a list at the end of another list as shown in Fig. 6.20. Consider the following program.

Example 6.24

Write a program to concatenate two linked lists and display the elements.

Solution

```
#include <stdio.h>
```

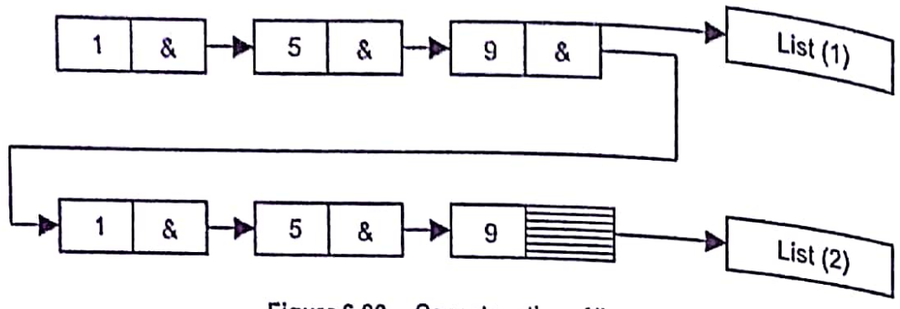


Figure 6.20 Concatenation of lists

```
# include <conio.h>
# include <malloc.h>

struct node
{
 int value;
 struct node *next;
};
struct node *rear,*start,*f;
struct node *create (void);
void concat(void);
void print (void );
int count=1;
struct node *one,*two;
main ()
{
 clrscr();
 one=create();
 two=create();
 concat();
 print();
}
struct node *create ()
 {
    struct node *item,*temp;
    temp=(struct node *)malloc(sizeof(struct node));
    printf ("\nEnter numbers of list (%d) (0) to exit: ",count);
    scanf ("%d",&temp->value);
    temp->next=NULL;
    rear=temp;
    while (1)
    {
    item=(struct node*)malloc(sizeof(struct node));
    scanf ("%d",&item->value);
    if (item->value==0)
```

Explanation The *creation ()* function is used to create linked lists. In this program, two lists are created. The starting address of the *list (2)* is assigned to the rear of the first list. Here, the same function is used to create both the lists. Hence, the contents of the local variables used in *create ()* function alters. In order to obtain the address of last node of the first list, *if* condition is inserted and the address of the rear node is stored in the pointer *'f.* All this process is carried out by the function *concat ()*. In the function *concat ()*, the starting address of the *list (2)* is assigned to rear of the *list (1)*. The pointer start is assigned the address of the *list (1)*. Using the same starting address the function *print ()* prints the element.


```
    {
    break;
    }
    item->next=NULL;
    rear->next=item;
    rear=item;
    if (count==1)
    f=rear;
    }
    count++;
    return (temp);
    }
    void concat ( )
    {
    f->next=two;
    start=one;
}
void print ( )
   {
   printf ("Merged list: ");
      while (start!=NULL)
   {
      printf (" %d ",start->value);
      start=start->next;
   }
}
```

```
OUTPUT
Enter numbers of list (1) (0) to exit: 1 5 9 0
Enter numbers of list (2) (0) to exit: 7 3 4 0
Merged list: 1 5 9 7 3 4
```

6.23 SPLITTING OF A LINKED LIST

Once a singly linked list is formed, it can split many sub-lists. The technique involved in splitting is very simple. Put the NULL value in the node where you want to finish the first sub-list in the main list. Later, the next node will act as first node of the second sub-list. Thus, two or more sub-lists can be created from a single list (see Fig. 6.21).

Example 6.25

Write a program to split a single linked list in two separate lists and display the elements.

Solution

```
# include <stdio.h>
# include <conio.h>
```

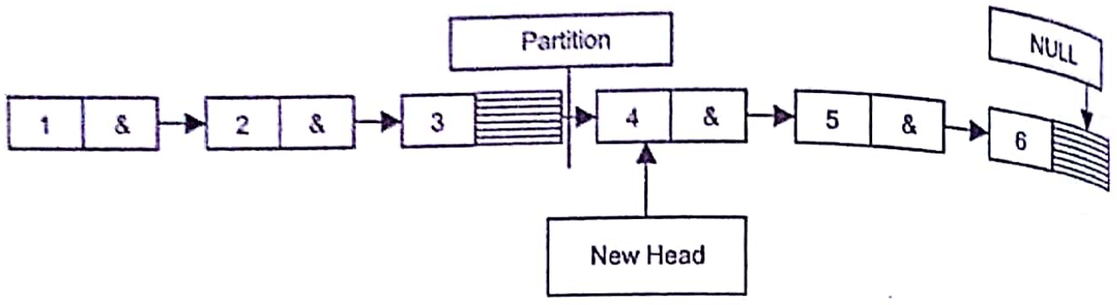


Figure 6.21 Partition of a single list

```
# include <malloc.h>
struct node
{
 int value;
 struct node *next;
};
 struct node *rear,*start,*f;
 struct node *create (void);
 struct node * split(int);
 void print (struct node * );
 int count=0;
 struct node *one,*two;

main ()
{
 clrscr();
 one=create();
 two=split(count/2);
 printf("The two separated list are:-");
 print(one);
 print(two);
}
struct node *create ()
 {
    struct node *item,*temp;
    temp=(struct node *)malloc(sizeof(struct node));
    printf ("\nEnter numbers of list( 0 to exit): ");
    scanf ("%d",&temp->value);
    count++;
    temp->next=NULL;
    rear=temp;
    if (count==1) start=temp;
    while (1)
    {
```

Explanation In the *create ()* function the counter variable *c* is incremented and keep an eye on the number of nodes created. In the *split ()*, function *c* is divided by two, i.e. to divide the list elements. In case, odd number of nodes are created then, the first sub-list gets more nodes. In the *split ()* function, exactly half of the nodes are traversed and the node in the middle is assigned a value NULL. Thus, the first sub-list ends here. The next node is considered as a head node for the second sub-list. The address of head node of the second sub-list is assigned to pointer two. Thus, using pointers one and two with function *print ()* two sub-lists are displayed.


```
item=(struct node *)malloc(sizeof(struct node));
scanf("%d",&item->value);
if(item->value==0)
break;
count++;
item->next=NULL;
rear->next=item;
rear=item;
}
count++;
return(temp);
}
struct node * split (int j )
{
int c=0;
while (one!=NULL)
{
  c++;
if(c==j)
{
 two=one->next;
 one->next=NULL;
 one=start;
 break;
 }
one=one->next;
}
return two;
}
  void print (struct node *s )
  {
  static c;
  c++;
  printf ("\n list (%d): ",c);

  while(s!=NULL)
{
  printf (" %d ",s->value);
  s=s->next;
}
}

OUTPUT
Enter numbers of list( 0 to exit): 1 2 3 4 5 6 7 8 0
The two separated list are:-
list (1): 1 2 3 4
list (2): 5 6 7 8
```

6.24 CIRCULAR LINKED LIST

In circular linked list the last node points to the header node. The linear link list can be converted to circular linked list by linking the last node to the first node. The last node of the linear linked list holds *NULL* pointer, which indicates the end of linked list but performance of linked list can be advanced with minor adjustment in the linear linked list. Instead of placing the NULL pointer, the address of the first node can be given to the last node, such a list is called *circular linked list* (as shown in Fig. 6.22).

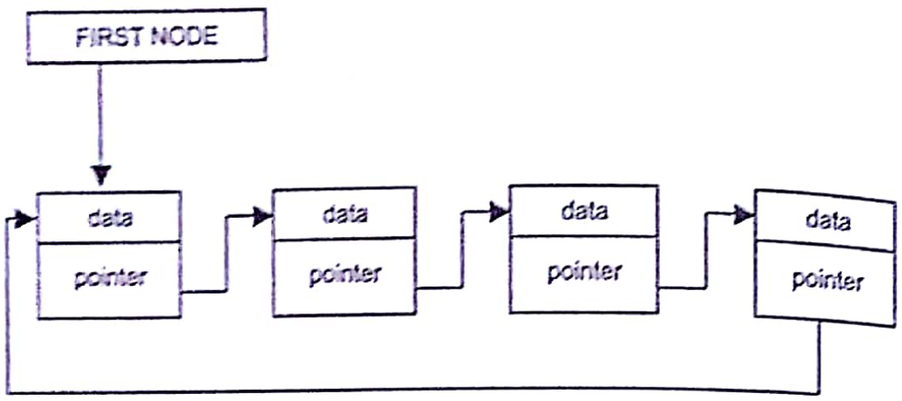


Figure 6.22 Circular linked list

The circular linked list is more helpful as compared to singly linked list. In the circular linked list, all the nodes of the list are accessible from the given node. Once a node is accessed, by traversing all the nodes can be accessed in succession.

In this type of list, the deletion operation is very easy. To delete an element from the singly linked list, it is essential to obtain the address of the first node of the list. For example, we want to delete the element, say 5, which exists in the middle of the list. To remove the element five, we need to find predecessor of five. Obviously, a particular element can be searched using searching process in which all elements are visited and compared. In the circular linked list, no such process is needed. The address of predecessor can be found from the given element itself. In addition, the operation splitting and concatenation of the list (discussed later) are easier.

Though the circular linked list has advantages over linear linked list, it also has some limitations. This list does not have first and last node. While traversing the linked list, due to lack of NULL pointer, there may be a possibility to get into an infinite *loop*. Thus, in the circular linked list it is necessary to identify the end of the list. This can be done by setting up the first and last node by convention. We detect the first node by creating the *list head*, which holds the address of the first node. The list head is also called as external pointer. We can also keep a counter, which is incremented when nodes are created and end of the node can be detected. The Fig. 6.23 gives an example of circular linked list with header.

6.25 METHOD FOR DETECTING END

There are two methods for finding or detecting the end of the circular linked list. They are:

1. Detecting end with counter
2. Detecting end with pointer.

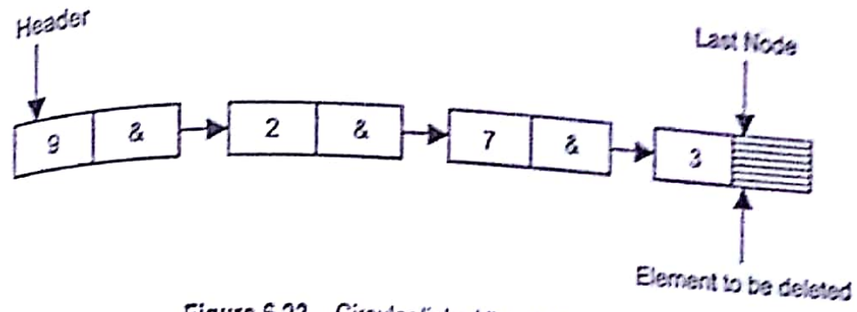


Figure 6.23 Circular linked list with header

6.25.1 Detecting End with Counter

This is the first method for detecting the end point of the circular linked list. The program 6.26 is similar to the program used for the creation of circular linked list. But only one change is done as compared to previous program, i.e. last node is pointing to the first node. Instead of assigning NULL pointer, the address of first node is assigned. Fig. 6.24 and program 6.26 explain the method for detecting end with counter.

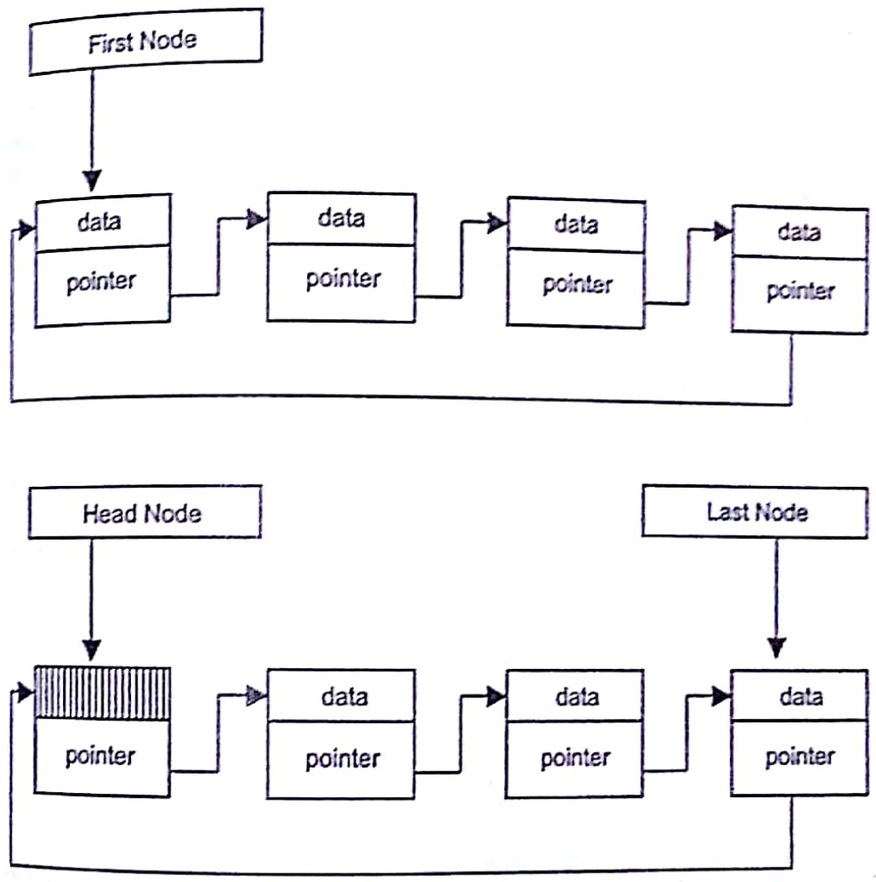


Figure 6.24 Detecting end with counter

Example 6.26

Write a program to create a circular linked list and display the elements.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>
struct num
{
    int num;
    struct num *next;
} *header,*first,*rear;
int count=0;

main ()
{

   void create ( void);
   int j,k=0;
   clrscr();
   create ();
    printf ("\n Linked list elements are: \n");
    printf("The circular linked list traversing\n");
    for (j=0;j<5;j++)
   {
    printf("\n In (%d ) round: "j+1);
    while (k<count)
   {
      printf (" %d ",header->num);
      header=header->next;
      k++;
 }
  k=0;
}
}
void create ( void )
{
   struct num *node;
   printf ("\n Enter numbers( 0 to exit): ");

  if (header==NULL)
  {
 first=(struct num*)malloc(sizeof(struct num));
 scanf ("%d",&first->num);
 first->next=header;
header=first;
  rear=first;
  count++;

 }
while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf("%d",&node->num);
  if (node->num==0) break;
  count++;
  node->next=header;
  rear->next=node;
  rear=node;

 }
}
```

Explanation Consider the following statement *node->next=header;*. The address of the first node (header node which points to the first node) is assigned to last. If you see the same statement in the creation of singly linked list, the statement is *node->next=NULL;*. Instead of NULL, header is assigned.


```
OUTPUT
Enter numbers( 0 to exit): 1 5 9 8 7 0
Linked list elements are:
The circular linked list traversing
In ( 1 ) round: 1 5 9 8 7
In ( 2 ) round: 1 5 9 8 7
In ( 3 ) round: 1 5 9 8 7
In ( 4 ) round: 1 5 9 8 7
In ( 6 ) round: 1 5 9 8 7
```

With creation operation, the counter variable is incremented. The counter variable is used to count the total number of nodes in the linked list. The *for loop* iterates from zero to 5 and the linked list elements are displayed five times. The *while loop* takes loops and it contains the condition *k*<counter. The variable *k* is incremented and when it is one less than counter it means end of the circular linked lists is reached. Again, in second iteration of *for loop*, *k* is initialized to zero. The value of *k* is nothing but the position of the node in the list. The value 0 means first node, 1 means second node and so on. Thus, first and last node of the list can be obtained.

If we remove the counter and the condition statements, which terminate the loop, we get into the infinite loop. After the last node, immediately first node is accessed. After removing the statement *printf ("\nRound (%d): ", j+1);* (after *for loop*) the output of the program would be

```
OUTPUT
Enter number: 1 2 3 0
Linked list elements are:
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

From the output, you can observe that sequence is preserved after last element (3). First, element 1 appears and displays the remaining elements. This process would be continued.

6.25.2 Detecting End with Pointer

This is the second method to detect the end of the circular linked list. In the last few programs, a rear pointer is declared which points to the last node and useful to find the end of the singly linked list. The same pointer is used here to detect the end of the circular linked list. Recall that in the circular linked lists the last node points to the first node, i.e. the pointer field of the rear pointer points to the data field of the first node as per the Fig. 6.25.

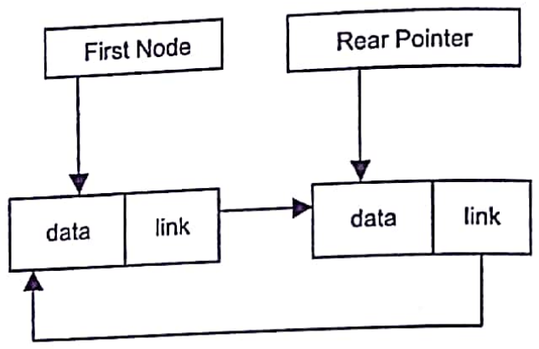


Figure 6.25 Detecting end with pointer

While scanning the circular linked list in order to get the end of the list, we have to check contents of all the pointer fields of the nodes. A link field of a node in a circular linked list pointing to the data field of the first node is the last node of the linked list. To implement this idea into a practical program we have to follow the following theory.

1. Obtain the address of the data field of the first node.
2. While traversing, compare the obtained address with the link field.
3. If the address obtained and the address pointing link field is same, i.e. that node is the last node.

Consider the following program,

Example 6.27

Write a program to detect the last node of the list using external pointer.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
   int num;
   struct num *next;
```

Explanation This program is similar to the previous program.


```
}*header,*first,*rear;
main()
{
void create ( void);
int j;
clrscr();
create ();
printf("\n Elements of circular linked list are:\n");
for (j=0;j<5;j++)
{
printf("\nRound (%d ): "j+1);

do
{
printf(" %d ",header->num);
header=header->next;
if(rear->next==(struct num *)&(header->num))
break;
}while(1);
}
printf("\nAddress of rear link field = %u",rear->next);
printf("\nAddress of first data field = %u",(struct num *)&(header->num));
}
void create ( void )
{
struct num *node;
printf ("\n Enter numbers( 0 to exit): ");
if (header==NULL)
{
  first=(struct num*)malloc(sizeof(struct num));
  scanf("%d",&first->num);
  first->next=header;
  header=first;
  rear=first;
}
while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf("%d",&node->num);
  if(node->num==0) break;
  node->next=header;
  rear->next=node;
  rear=node;
}
}
```

```
OUTPUT
Enter numbers( 0 to exit): 5 4 8 7 0
Elements of circular linked list are:
Round (1 ): 5 4 8 7
Round (2 ): 5 4 8 7
Round (3 ): 5 4 8 7
Round (4 ): 5 4 8 7
Round (5 ): 5 4 8 7
Address of rear link field = 1970
Address of first data field = 1970
```

Consider the statement,

```
if (rear->next==(struct num *)&(header->num))
break;
```

As soon as the last node is detected, the above statement terminates the *while loop*. The *if* statement compares the address of data field of first node with pointer field of each node. When it is found that both are same, the end of linked list is spotted. Here, the address of the data field is obtained by the typecasting statement (*struct num* *)&(*header->num*).

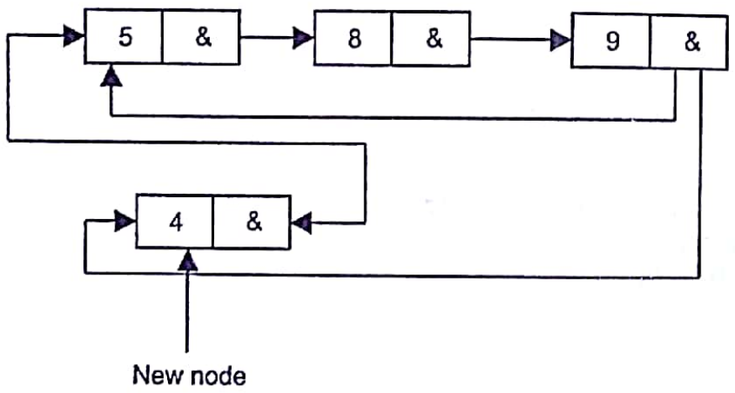
Here, typecasting is essential because an address of integer cannot be assigned to pointer of structure type. The syntax *(struct num *)* converts the address of integer *header->num* to pointer *rear->next*. In the output, addresses are also displayed.

6.25.3 Insertion in Circular Linked List

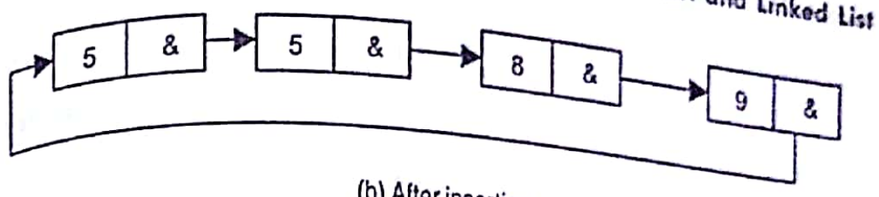
Insertions in the circular linked list are possible in three different ways:

1. Insertion at the beginning
2. Insertion at the end
3. Insertion at the specified position.

1. Insertion at the beginning



(a) A circular linked list before insertion



(b) After insertion

Figure 6.26 Insertion operation with circular list

Insertion of new node at the beginning involves memory allocation. Moreover, the value is assigned to the data field of the new node. When a new node is created for insertion, the address of data field of this new node is assigned to pointer of last node. Similarly, the address of data field of previously first node is assigned to new node. Thus, the new node becomes first node and the previous first node is moved to second position. Fig. 6.26 illustrates the insertion operation in circular linked list.

2. Insertion at the end

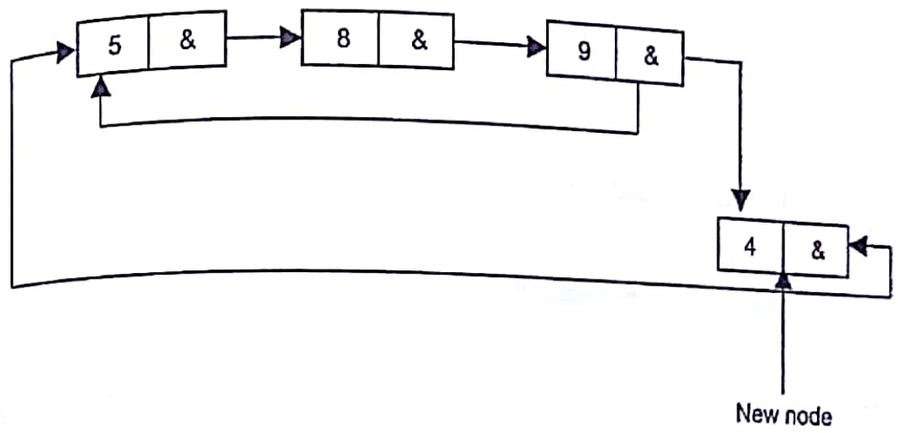


Figure 6.27 Insertion at end

When a new node is inserted at the end, the address of new node is given to the last node. The address of first node is given to the new node. Fig. 6.26 shows the insertion operation at end.

Example 6.28

Write a program to demonstrate insertion of element at the beginning and at end of the circular linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
```

```
# include <malloc.h>
struct num
{
    int num;
    struct num *next;
} *header,*first,*rear;
main ()
{
    void create ( void);
    void iatend(void);
    void iatbeg(void);
    void show(void);
    clrscr();
    create (); /* creation of linked list*/
    show(); /* display of linked list*/
    iatbeg(); /* insertion at beginning*/
    show(); /* display of elements*/
    iatend(); /* insertion at the end*/
    show();
}
void create ( void )
{
    struct num *node;
    printf ("\n Enter numbers(0 to exit): ");
    if (header==NULL)
    {
    first=(struct num*)malloc(sizeof(struct num));
    scanf ("%d",&first->num);
    first->next=header;
    header=first;
    rear=first;
 }
while (1)
{
    node=(struct num*) malloc(sizeof(struct num));
    scanf ("%d",&node->num);
    if (node->num==0) break;
    node->next=header;
    rear->next=node;
    rear=node;
}
}
void iatend()
{
    struct num *node;
    node=(struct num*) malloc(sizeof(struct num));
    printf ("\nEnter a number to be inserted at end: ");
    scanf ("%d",&node->num);
    node->next=header;
    rear->next=node;
    rear=node;
}
void iatbeg()
{
    struct num *node;
    node=(struct num*) malloc(sizeof(struct num));
    printf ("\nEnter a number to be inserted at beginning: ");
    scanf ("%d",&node->num);
    node->next=header;
    header=node;
    rear->next=header;
}
void show ()
{
    printf ("Linked list elements are: ");
do
{printf (" %d ",header->num);
  header=header->next;
  if(rear->next==(struct num *)&(header->num))
  break;
}while(1);
}
```

Explanation The reader is already aware of the creation and display of linked list. The function *iatbeg ()* inserts the element at the beginning of the list and the function *iatend ()* inserts the element at the end.


```
OUPUT
Enter numbers( 0 to exit): 1 2 3 0
Linked list elements are: 1 2 3
Enter a number to be inserted at beginning: 4
Linked list elements are: 4 1 2 3
Enter a number to be inserted at end: 5
Linked list elements are: 4 1 2 3 5
```

We know that the statement *node=(struct num*) malloc(sizeof(struct num));* is used to create a new node. After creation of new node, only pointer adjustment is done. When we want to insert an element at the beginning the pointer adjustment is done as follows:

```
node->next=header;
header=node;
rear->next=header;
```

Remember the following points:

1. The pointer node is a new node.
2. The header points to the first node.
3. The pointer rear points to the last node.

Since we are going to insert at the beginning, the new node created will be the first node of the list. Then to accomplish this the link field of new node is assigned the address of first node, i.e. header. After this, the first node will be the second node and the new node will be the first node. The pointer header is always used to point first node. Hence, after insertion its contents must be updated. Thus, the statement *header=node;* changes the contents of the pointer header. The last pointer adjustment to be done is, to assign the address of the first node data field to last node. The statement *rear->next=header;* serves the same purpose.

The following program explains insertion at the specified position.

Example 6.29

Write a program to insert an element at a given position in the circular linked list.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct num
{
   int num;
   struct num *next;
} *header,*first,*rear;

main ()
{
   void create ( void);
   void show(void);
   void atgiven(int);
   int n;
   clrscr();
   create ();
   printf ("\n Enter position number: ");
   scanf ("%d",&n);
   n--;
   atgiven(n);
   show();
}
void create ( void )
{
   struct num *node;
   printf ("\n Enter numbers( 0 to exit): ");

  if (header==NULL)
  {
  first=(struct num*)malloc(sizeof(struct num));
```

Explanation In this program, the function *atgiven()* is used to insert the element at the specified position. The user specifies the position number where number is to be inserted. The position number is passed to the function *atgiven()*. In this function, using *while loop* the linked list is traversed and the previous and next node of the given element are stored in the pointers *prv* and *header*.


```
  scanf("%d",&first->num);
  first->next=header;
  header=first;
  rear=first;
  }
while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
  scanf("%d",&node->num);
  if (node->num==0) break;
  node->next=header;
  rear->next=node;
  rear=node;
}
}
void show ()
{
 printf ("Linked list elements are: ");
do
{printf (" %d ",header->num);
 header=header->next;
 if(rear->next==(struct num *)&(header->num))
 break;
 }while(1);
}
void atgiven(int J)
 {
  int c=0,k=J-2;
  struct num *node,*prv;
  printf ("\nLinked list upto %d element are: ",J);
do
  {printf (" %d ",header->num);
  header=header->next;
  if(k==c) prv=header;
  c++;
} while(c<J);
node=(struct num*) malloc(sizeof(struct num));
printf ("\n Enter a element: ");
scanf("%d",&node->num);
 node->next=header;
 prv->next=node;
 header=first;
 }

 OUTPUT
 Enter numbers( 0 to exit): 1 2 3 5 6 7 0
 Enter position number: 3
```

```
Linked list up to 3 element are: 1 2 3
Enter a element: 4
Linked list elements are: 1 2 3 4 5 6 7
```

```
node->next=header;
prv->next=node;
header=first;
```

The new node created is given the address of header. At this point *header* is not pointing to the first node. Due to traversing in the *while loop*, it is pointing to the nth element, see the statement (*header=header->next;*). The new node is assigned the address of next node. The prv is assigned the address of new node. Again, the header is assigned the address of first node.

Some authors describe the insertion before and after. I think there is no need to classify it differently. The above program inserts the element after the specified position. In order to insert an element before the given position, only decrease the value of variable *n* before passing to function *atgiven ()*. By changing the value of *n*, the user can accomplish the operation insertion before, after and exactly at given position.

6.25.4 Deletion in Circular Linked List

Like insertion, deletion of element in the circular linked list can be done at various positions and they are as follows:

1. Deletion at beginning
2. Deletion at the end
3. Deletion at the specified position.

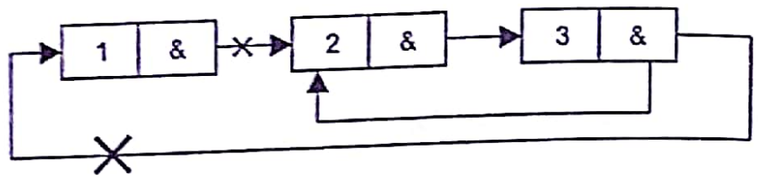


Figure 6.28 Deletion at beginning

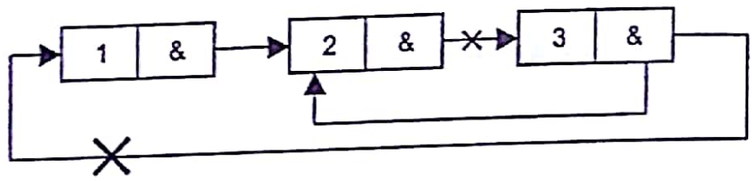


Figure 6.29 Deletion at end

The pointer adjustment is same in insertion and deletion. However, in insertion, an element is inserted and memory is allocated. In deletion, a node from the list is removed and memory is released. Consider the following program. Refer the Fig. 6.28 and 6.29.

Example 6.30

Write a program to delete an element from end and beginning of the linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct num
{
int num;
struct num *next;
}*header,*first,*rear;
static int c;

main()
{
void create (void);
void datend(void);
void datbeg(void);
void show(void);
clrscr();
create ();
show();
printf ("\nAfter deletion of beginning element");
datbeg();
show();
printf ("\nAfter deletion of end element");
datend();
show();
}
void create ()
{
struct num *node;
printf ("\n Enter numbers( 0 to exit): ");
if(header==NULL)
{
first=(struct num*)malloc(sizeof(struct num));
scanf("%d",&first->num);
first->next=header;
header=first;
rear=first;
c++;
}
while (1)
{
node=(struct num*) malloc(sizeof(struct num));
```

Explanation In this program to erase the element from beginning and end, help of counter variable is taken. The variable *c* is incremented in the function *create ()* when a new node is created. Similarly, when an element is removed the counter *c* is decremented. The counter variable is needful only in the function *datend ()*, because to erase the last element, the last but one element is obtained and it is connected to first node.

```
    scanf ("%d",&node->num);
    if (node->num==0) break;
    node->next=header;
    rear->next=node;
    rear=node;
    c++;
}
}
void datend()
{
    int k=0;
    while (k++<(c-2))
    header=header->next;
    free(rear);
    rear=header;
    rear->next=first;
    header=first;
    c--;
}
void datbeg()
{
    struct num *node;
    node=first;
    first=header=header->next;
    free(node);
    rear->next=header;
    c--;
}
void show ()
{
printf ("\nLinked list elements are: ");
do
{
printf (" %d ",header->num);
header=header->next;
if (rear->next==header) break;
} while(1);
header=first;
}

OUTPUT
Enter numbers( 0 to exit): 1 2 3 4 5 6 7 0
Linked list elements are: 1 2 3 4 5 6 7
After deletion of beginning element
Linked list elements are: 2 3 4 5 6 7
After deletion of end element
Linked list elements are: 2 3 4 5 6
```


Example 6.31

Write a program to delete the specified element from the linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct num
{
  int num;
  struct num *next;
}*header, *first, *rear;
static int c;

main()
{
  void create (void);
  void atgiven(int);
  void show(void);
  int n;
  clrscr();
  create ();
  show();
  printf ("\n Enter position number: ");
  scanf ("%d",&n);
  atgiven(n);
  show();
  getch();
}
void create ()
{
  struct num *node;
  printf ("\n Enter numbers( 0 to exit): ");
  if(header==NULL)
  {
  first=(struct num*)malloc(sizeof(struct num));
  scanf("%d",&first->num);
  first->next=header;
  header=first;
  rear=first;
  c++;
}
while (1)
{
  node=(struct num*) malloc(sizeof(struct num));
```

Explanation In this program the function *atgiven ()* deletes the specified element from the circular linked list. The user enters the position number and it is passed to the function *atgiven ()*. In this function using *while loop* entire list is scanned and predecessor and successor of the given element are stored in the pointer **prv* and *header*. The node, which is to be removed, is stored in the pointer *temp*. Using *free ()* function the memory allocated is released. Then, a link between pointer *prv* and *header* is established by the following statements.

```
    scanf("%d",&node->num);
    if (node->num==0) break;
    node->next=header;
    rear->next=node;
    rear=node;
    c++;
  }
 }
    void atgiven(int g)
    {
    struct num *prv,*temp;
    int k=0,p;
    p=g-2;
    while (k++<g )
    {
    header=header->next;
    if (k==p) prv=header;
    if (k==(p+1)) temp=header;
  }
    c--;
    prv->next=header;
    header=first;
    free(temp);
}
void show ()
{
 printf ("\nLinked list elements are: ");
 do
 {
    printf (" %d ",header->num);
    header=header->next;
    if (rear->next==header) break;
 } while(1);
 header=first;
 }
```

OUTPUT

```
Enter numbers( 0 to exit: 5 8 7 4 5 3 2 0
Linked list elements are: 5 8 7 4 5 3 2
Enter position number: 3
Linked list elements are: 5 8 4 5 3 2
```


6.26 DOUBLY LINKED LIST

The singly linked list and circular linked list contain only one pointer field. Every node holds an address of next node. Thus, the singly linked list can traverse only in one direction, i.e. forward. This limitation can be overcome by doubly linked list. Each node of the doubly linked list has two pointer fields and holds the address of predecessor and successor elements. These pointers enable bi-directional traversing, i.e. traversing the list in backward and forward direction. In several applications, it is very essential to traverse the list in backward direction. The pointer pointing to the predecessor node is called left link and pointer pointing to successor is called right link. A list having such type of node is called doubly linked list. The pointer field of the first and last node holds NULL value, i.e. the beginning and end of the list can be identified by NULL value. The structure of the node is as shown in Fig. 6.30.

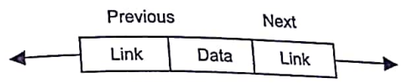


Figure 6.30 Structure of node

The structure of node would be as follows:

```
struct node
{
  int number;
  struct node *llink
  struct node *rlink;
}
```

the above structure can be represented by using the Fig. 6.31.

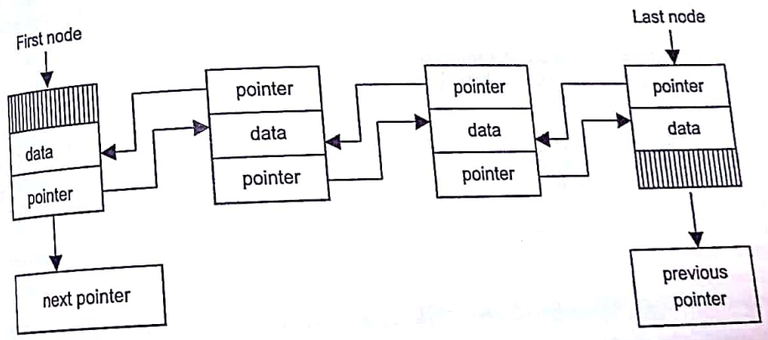


Figure 6.31 Doubly linked list

Example 6.32

Write a program to create the doubly linked list. Display the elements in original and reverse sequence.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>

struct dlist
{
   struct dlist *prev;
   int number;
   struct dlist *next;
};
struct dlist *end,*start;
struct dlist *node=NULL;
void create (void);
void showl(void);
void showr(void);

main ()
{

  clrscr();
  create();
  showl();
  showr();
  getch();
}
void create ()
{
   node=(struct dlist*)malloc(sizeof(struct dlist));
  printf ("\n Enter Numbers (0 to stop): ");
  while (1)
  {
  scanf ("%d",&node->number);
  if (node->number==0) break;
  else
  {
  node->prev =end;
  end=node;
  node=(struct dlist*) malloc(sizeof(struct dlist));
  }
  }
node=end;
start=NULL;
while (node!=NULL)
{
node->next=start;
start=node;
node=node->prev;
}
}
void showl()
{
printf ("\n Original List: ");
while(start!=NULL)
{
 printf (" %d ",start->number);
 start=start->next;
}
}
void showr()
{
printf ("\n Reverse List: ");
while (end!=NULL)
{
 printf (" %d ",end->number);
 end=end->prev;
}
}
```

Explanation The *create ()* function is used to create the doubly linked list. The method of creation of node is same as in the previously discussed methods of the linked list. Here, an additional one pointer field is present and we will discuss that in detail in next section.


```
OUTPUT
Enter Numbers (0 to stop): 1 2 4 7 5 0
Original List: 1 2 4 7 5
Reverse List: 5 7 4 2 1
```

We know the use of next pointer and how we have used it previously to access the next node. Every time a new node is inserted, the address of new node is assigned to the next pointer of previous node. This is required for details of manipulating the next pointer. The pointer **prev* (pointing to data field of previous node) is not different case. The pointer next is right hand and **prev* is the left hand of the node. Every time a new node is created, the address of data field of previous node is assigned to the previous pointer. The rest of the operations are same for accessing the elements. Go through the functions *showl()* and *showr()*.

6.26.1 Insertion and Deletion with Doubly Linked List

We are aware of insertion process in which an element can be inserted at beginning, at end and at the specified position. Fig. 6.32 shows the insertion in the doubly linked list.

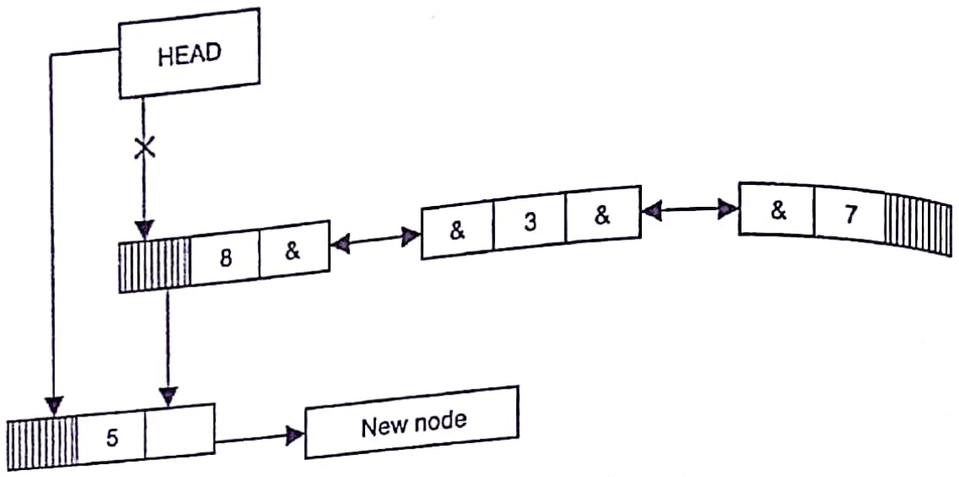


Figure 6.32 Inserting a node at the beginning

We know that the head node of the doubly linked list contains NULL value. When a new node is to be inserted at the beginning, the address of the head node is assigned to the new node. The previous pointer of the node is assigned a NULL value. The arrow ↔ indicates that the node has both previous and next pointer.

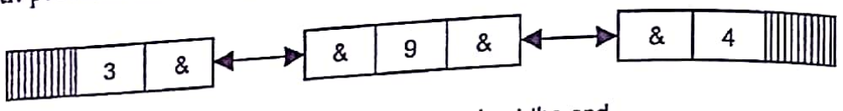


Figure 6.33 Inserting a node at the end

When a node is inserted at the end, the next pointer of the new node is assigned a NULL value and the previous pointer of the node is assigned the address of last node. The Fig. 6.33 describes the insertion at the end.

In the deletion operation as shown in the Fig. 6.34 when a node is deleted, the memory allocated to that node is released and the previous and next nodes of that node are linked.

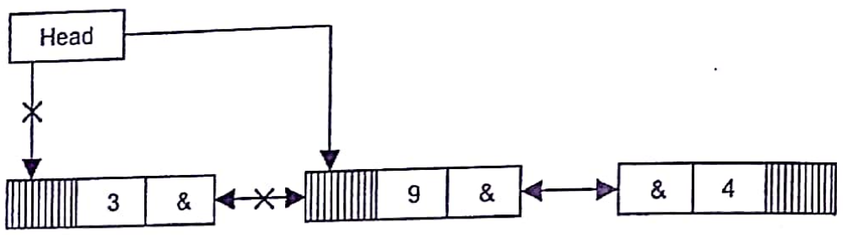


Figure 6.34 Deleting a node from beginning

When a node is to be deleted from the beginning of the node, the head pointer points to the second node. Because after deletion of first node, the second node becomes the first. The symbol × indicates the link will be destroyed. This is shown in the Fig. 6.35.

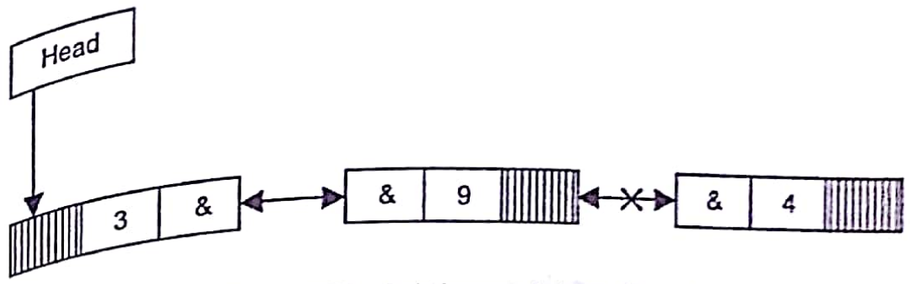


Figure 6.35 Deleting a node from the end

Example 6.33

Write a program to perform insertion and deletion operation on the doubly linked list.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

struct dlist
{
  struct dlist *prev;
  int number;
  struct dlist *next;
};
struct dlist *end,*start;
struct dllst *node=NULL;
void create (void);
void show(void);
void insert (void);
void re_move(void);

main()
{
  clrscr();
  create();
  insert();
  show();
  re_move();
  show();
  getch();
```

Explanation In this program both insertion and deletion operations are performed. The method of adjusting pointer is same as explained in the programs of singly and circular linked list operations. The user can insert or delete the element of any position including beginning and end.

```
}
void create ()
{
   node=(struct dlist*)malloc(sizeof(struct dlist));
   printf ("\n Enter Numbers (0 to stop): ");
   while (1)
 {
   scanf ("%d",&node->number);
   if (node->number==0) break;
   else
 {
   node->prev =end;
   end=node;
   node=(struct dlist*) malloc(sizeof(struct dlist));
 }
 }
 node=end;
start=NULL;
while (node!=NULL)
{
node->next=start;
start=node;
node=node->prev;
   }
}
void show( )
{
   struct dlist *x;
printf ("\n Origional List: ");

x=start;

while (start!=NULL)
{
printf (" %d ",start->number);
start=start->next;
}
start=x;
}
void insert ()
{
struct dlist *q,*r,*p;
int number,k=1;
p=start;
printf (" (Insertion )Enter position of node: ");
scanf ("%d",&number);
if (number==1)
```



```
{
q=(struct dlist*)malloc(sizeof(struct dlist));
printf ("\n (Insertion) Enter the Number: ");
scanf ("%d",&q->number);
q->prev=NULL;
q->next=start;
start=q;
}
while(start!=NULL)
{
if(++k==number)
{
q=(struct dlist*)malloc(sizeof(struct dlist));
printf ("\n Enter the Number: ");
scanf ("%d",&q->number);
r=start->next;
r->prev=q;
q->next=r;
q->prev=start;
start->next=q;
break;
}
else
start=start->next;
}
if(number==1) start=q;
else start=p;
}
void re_move ()
{
struct dlist *q,*r,*p,*n;
int number,c=0;
printf ("\n (Deletion) Enter position of node: ");
scanf ("%d",&number);
r=start;
if(number==0)
{
p=start;
free(p);
start=start->next;
start->prev=NULL;
}
else
while (r!=NULL)
{
c++;
if(c==(number-1)) p=r;
```

```
if (c==number)
{
q=r;
free(q);
}
if (c==(number+1)) n=r;
r=r->next;
}
p->next=n;
}

OUTPUT
Enter Numbers (0 to stop): 1 2 3 4 5 6 7 0
(Insertion )Enter position of node: 3
Enter the Number: 8
Original List: 1 2 8 3 4 5 6 7
(Deletion) Enter position of node: 6
Original List: 1 2 8 3 4 6 7
```

6.26.2 Advantages of Doubly Linked List

1. The doubly linked list is bi-directional, i.e. it can be traversed in both backward and forward direction.
2. The operations such as insertion, deletion and searching can be done from both ends.
3. Predecessor and successor of any element can be searched quickly.
4. It is very helpful to implement arithmetic operations on large integer numbers.

6.26.3 Disadvantages of Doubly Linked List

1. It consumes more memory space.
2. There is large pointer adjustment during insertion and deletion of element.
3. It consumes more time for few basic list operations.

6.27 CIRCULAR DOUBLY LINKED LIST

A circular doubly linked list has both successor and predecessor pointers. Using the circular fashioned doubly linked list the insertion and deletion operation, which are little complicated in the previous types of linked list are easily performed. Consider the following Fig. 6.36:

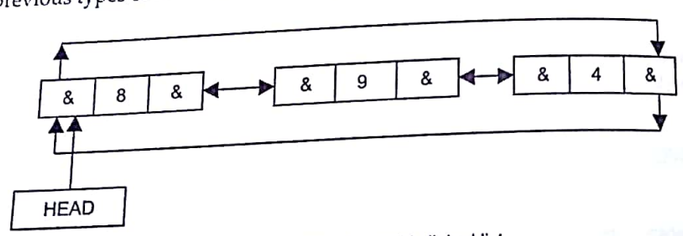


Figure 6.36 Circular doubly linked list

6.27.1 Insertion and Deletion Operation

The insertion operation is similar to what we have already learnt in previous types. The only difference is the way we link the pointer fields. Consider Fig. 6.37.

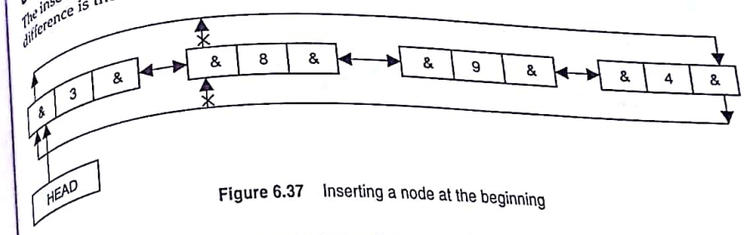


Figure 6.37 Inserting a node at the beginning

The × indicates that the previous links are destroyed. The pointer links from *ex-first node* are removed and linked to new inserted node at the beginning. The element 8 was previous first node and 3 is the new node inserted and becomes first node now after inserting it at the beginning.

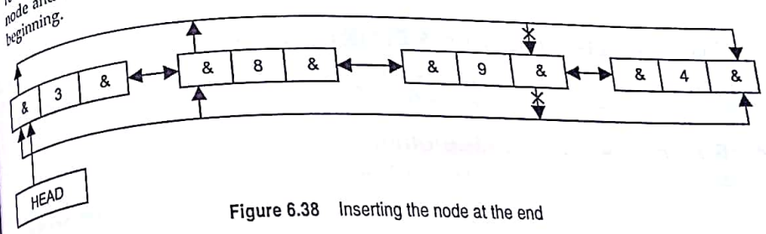


Figure 6.38 Inserting the node at the end

The × indicates, that the previous links are destroyed. The pointer links from *ex-last node* are removed and linked to new inserted node at the end. The address of last node is given to first node to form circular list. The node 9 was previously the last node but after insertion of the node at the end, newly inserted node is the last node. Fig. 6.39 shows the deletion of the node at beginning.

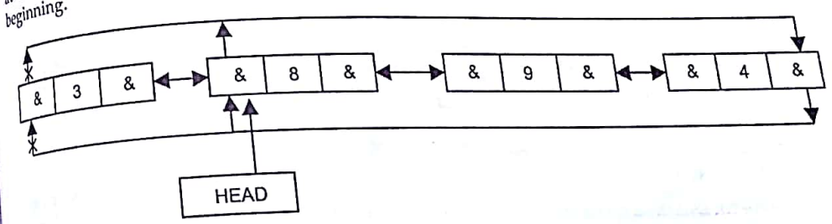


Figure 6.39 Deleting a node from the beginning

The × indicates that the previous links are destroyed. After deletion of the first node, the second node becomes first node. The pointer head also points to the newly appeared first node. Thus, when a node from the beginning is removed, the node followed by it will become the head node (first node). Accordingly, pointer adjustment is performed in the real application.

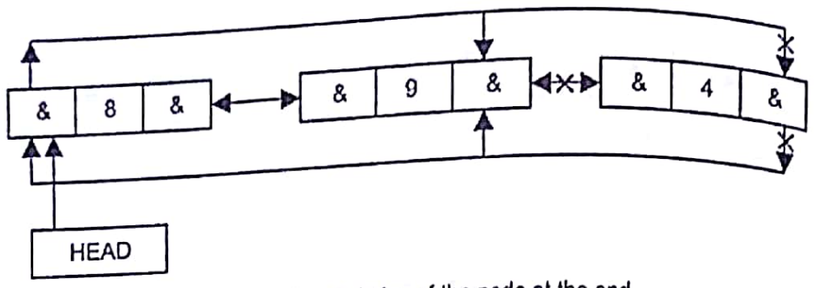


Figure 6.40 Deletion of the node at the end

The × as usual is the symbol destroying previous links. When the last node is removed, one node will be the last node and its links are established with the first node. The removal operation is shown in Fig. 6.40.

6.28 APPLICATIONS OF LINKED LIST

The most useful linear data structure is linked list. This section introduces you to a few applications of linked lists which are useful in computer science.

6.28.1 Polynomial Manipulation

A polynomial can be represented and manipulated using linear link list. Various applications on polynomials can be implemented with linked lists. We perform various operations such as addition, multiplication with polynomials. To get better proficiency in processing, each polynomial is stored in decreasing order. These arrangements of polynomial in series allow easy operation on them. Actually, two polynomials can be added by checking each term. The prior comparison can be easily done to add corresponding terms of two polynomials.

A polynomial is represented with various terms containing coefficients and exponents. In other words, a polynomial can be expressed with different terms, each of which comprises of coefficients and exponents. The structure of a node of linked list for polynomial implementation will be as follows. Its pictorial representation is shown in Fig. 6.41.

Figure 6.41 A term of a polynomial

The coefficient field contains the value of coefficient of the term. Similarly, the exponent field contains the value of exponent. As usual, the link field points to the term (next node).

The structure for the above node would be as follows:

```
struct poly
{
  double coeff;
  int exp;
  struct poly *next;
};
```

Consider a polynomial

$P = P^6 + 5P^4 - 7P^2 + 6P$

In this equation 1,5,7 and 6 are coefficients and exponents are 8,4,2 and 1. The number of nodes required would be the same as the number of terms in the polynomial. There are four terms in this polynomial hence it is represented with four nodes.

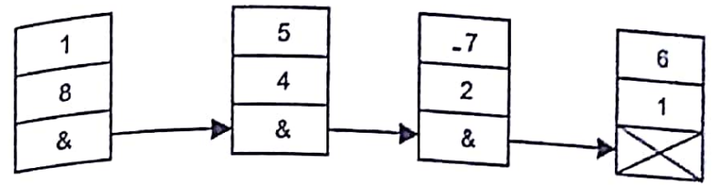


Figure 6.42 Polynomial representation

The top of every node represents coefficients of the polynomial, exponents are at the centre and the pointers are (*next*) at the bottom. The terms are stored in order of descending exponent in the linked list. It is assumed that no two terms have the similar exponents. Fig. 6.42 shows the polynomial.

Consider the following equations,

$P = P^6 + 5P^4 - 7P^2 + 6P$
$Q = 2P^9 + 7P^6 - 3P^4 + 2P^3$

The representation of the above two polynomials can be shown in Fig. 6.43.

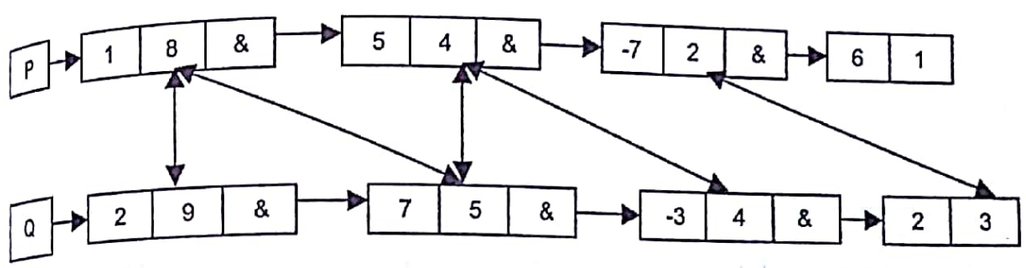


Figure 6.43 Addition of two polynomials

If the term is shown without coefficient then the coefficient is assumed to be 1. In case the term is without variable, the coefficient is zero. Such terms are not required to be stored in the memory. The arrow in the figure indicates that the two exponents are compared. Where the link of arrow is disconnected, exponent of both the terms is same. The term, which has no touch of any arrow, means it is the last node and inserted in the resulting list at the end. The arrow indicates two terms, which are compared systematically from left to right.

Table 6.1 Exponent

Exponent comparison from list P and Q	*List R (Inserted exponent)*	*Smaller exponent*
8<9	9	8 is carried forward
8>5	8	5 is carried forward
5>4	5	4 is carried forward
4= =4	4 (Sum of coefficient is taken)	No term is carried
2<3	3	2 is carried forward
1 is taken (last node of P)	1	End of linked list

Traverse the list *P* and *Q*. Compare the corresponding terms of list *P* and *Q*. In case one node has larger exponent value than the other then insert the larger exponent node in the third list and forward the pointer to next node of the list whose current term is inserted in the third list. The pointer of the list whose exponent is smaller will not be forwarded. The pointer of the lists forwarded only when the current nodes from the lists are inserted into the third list. Table 6.1 shows these operations.

If exponents are equal, add the coefficients and insert their addition in the third list. In this step, exponents from both the expressions are same, move the pointer to next node in both the list *P* and *Q*. Repeat the same process until one list is scanned completely.

expo(*P*) = expo(*Q*).

The possible conditions can be stated as follows:

1. If exponent of list *P* is greater than corresponding exponent of list *Q*, insert the term of *P* into the list *R* and forward the pointer in list *P* to access the next term. In this case, the pointer in the list *Q* will point to the same term (will not be forwarded).
2. If exponent of list *Q* is greater than exponent *P*, the term from *Q* is inserted in the list *R*. The pointer of list *Q* will be forwarded to point to next node. The pointer in the list *P* will remain at the same node.
3. If exponents of both nodes are equal, addition of coefficients is taken and inserted in the list *R*. In this case, pointers in both the lists are forwarded to access the next term.

The steps involved,

1. Traverse the two lists (*P*) and (*Q*) and inspect all the elements.
2. Compare corresponding exponents *P* and *Q* of two polynomials. The first terms of the two polynomials contain exponents 8 and 9, respectively. Exponent of first term of first

polynomial is smaller than the second one. Hence 8<9. Here, the first exponent of list *Q* is greater than the first exponent of list *P*. Hence, the term having larger exponent will be inserted in the list *R*. The list *R* initially looks like as shown in Fig. 6.44(a).

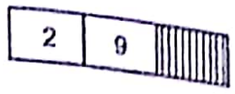


Figure 6.44 (a)

Next, check the next term of the list *Q*. Compare it with the present term (exponent) of list *P*. The next node from *P* will be taken when current node is inserted in the list *R* because 8>5. Here, the exponent of current node of list *P* is greater that list *Q*. Hence, current term (node) of list *P* will be inserted in the list *R* and the list *R* becomes as Fig. 6.44 (b).

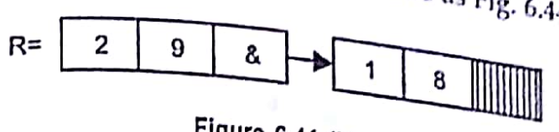


Figure 6.44 (b)

After moving to next node in list *P*, the exponent is 4, and exponent of *Q* is 5. Compare 4 with 5. Of course 4<5. Here, the term of list *Q* is greater than term of *P*. Therefore, the term of list, *Q* (7,5) will be inserted to list *R*. The list *R* becomes Fig. 6.44 (c).

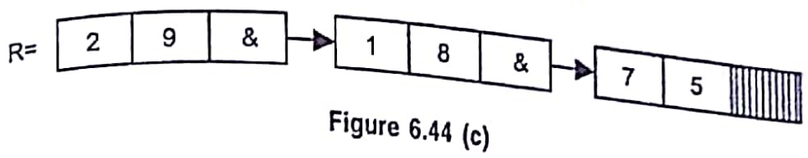


Figure 6.44 (c)

In this step a node from list *Q* is inserted and therefore, the pointer in the list *Q* will be forwarded and point to the term (–3,4) and from list *P* we have the current node (5,4). Compare exponent of these two terms. The condition observed here is 4==4. Here, exponents of both the terms are equal. Therefore, addition of coefficients is taken and result is inserted in the list R. The addition is 2 (5–3). The list *R* becomes as the Fig. 6.44 (d).

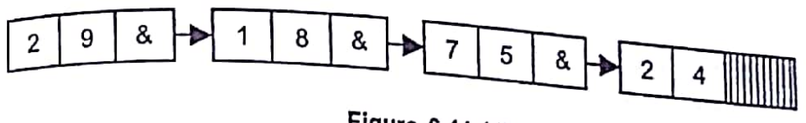


Figure 6.44 (d)

Move forward the pointers to next nodes in both the lists, since, the previous terms were having same exponents. The next comparison is 2<3. Here, the exponent of current node of list *Q* is greater than of *P*. The node from *Q* will be inserted to list *R*. The list *R* will be shown as Fig. 6.44(e).

If the term is shown without coefficient then the coefficient is assumed to be 1. In case the term is without variable, the coefficient is zero. Such terms are not required to be stored in the memory. The arrow in the figure indicates that the two exponents are compared. Where the link of arrow is disconnected, exponent of both the terms is same. The term, which has no touch of any arrow, means it is the last node and inserted in the resulting list at the end. The arrow indicates two terms, which are compared systematically from left to right.

Table 6.1 Exponent

Exponent comparison from list P and Q	*List R (Inserted exponent)*	*Smaller exponent*
8<9	9	8 is carried forward
8>5	8	5 is carried forward
5>4	5	4 is carried forward
4= =4	4	No term is carried
	(Sum of coefficient is taken)	
2<3	3	2 is carried forward
1 is taken (last node of P)	1	End of linked list

Traverse the list *P* and *Q*. Compare the corresponding terms of list *P* and *Q*. In case one node has larger exponent value than the other then insert the larger exponent node in the third list and forward the pointer to next node of the list whose current term is inserted in the third list. The pointer of the list whose exponent is smaller will not be forwarded. The pointer of the lists forwarded only when the current nodes from the lists are inserted into the third list. Table 6.1 shows these operations.

If exponents are equal, add the coefficients and insert their addition in the third list. In this step, exponents from both the expressions are same, move the pointer to next node in both the list *P* and *Q*. Repeat the same process until one list is scanned completely.

expo(*P*) = expo(*Q*).

The possible conditions can be stated as follows:

1. If exponent of list *P* is greater than corresponding exponent of list *Q*, insert the term of *P* into the list *R* and forward the pointer in list *P* to access the next term. In this case, the pointer in the list *Q* will point to the same term (will not be forwarded).
2. If exponent of list *Q* is greater than exponent *P*, the term from *Q* is inserted in the list *R*. The pointer of list *Q* will be forwarded to point to next node. The pointer in the list *P* will remain at the same node.
3. If exponents of both nodes are equal, addition of coefficients is taken and inserted in the list *R*. In this case, pointers in both the lists are forwarded to access the next term.

The steps involved,

1. Traverse the two lists (*P*) and (*Q*) and inspect all the elements.
2. Compare corresponding exponents *P* and *Q* of two polynomials. The first terms of the two polynomials contain exponents 8 and 9, respectively. Exponent of first term of first

polynomial is smaller than the second one. Hence 8<9. Here, the first exponent of list *Q* is greater than the first exponent of list *P*. Hence, the term having larger exponent will be inserted in the list *R*. The list *R* initially looks like as shown in Fig. 6.44(a).

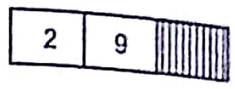


Figure 6.44 (a)

Next, check the next term of the list *Q*. Compare it with the present term (exponent) of list *P*. The next node from *P* will be taken when current node is inserted in the list *R* because 5>5. Here, the exponent of current node of list *P* is greater that list *Q*. Hence, current term (node) of list *P* will be inserted in the list *R* and the list *R* becomes as Fig. 6.44 (b).

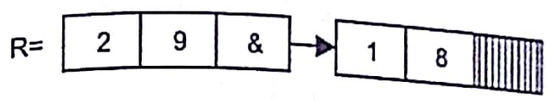


Figure 6.44 (b)

After moving to next node in list *P*, the exponent is 4, and exponent of *Q* is 5. Compare 4 with 5. Of course 4<5. Here, the term of list *Q* is greater than term of *P*. Therefore, the term of list, *Q* (7,5) will be inserted to list *R*. The list *R* becomes Fig. 6.44 (c).

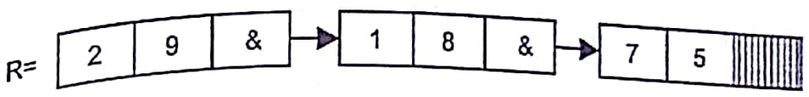


Figure 6.44 (c)

In this step a node from list *Q* is inserted and therefore, the pointer in the list *Q* will be forwarded and point to the term (–3,4) and from list *P* we have the current node (5,4). Compare exponent of these two terms. The condition observed here is 4==4. Here, exponents of both the terms are equal. Therefore, addition of coefficients is taken and result is inserted in the list R. The addition is 2 (5–3). The list *R* becomes as the Fig. 6.44 (d).

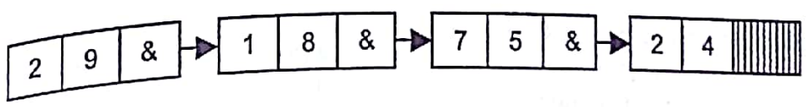


Figure 6.44 (d)

Move forward the pointers to next nodes in both the lists, since, the previous terms were having same exponents. The next comparison is 2<3. Here, the exponent of current node of list *Q* is greater than of *P*. The node from *Q* will be inserted to list *R*. The list *R* will be shown as Fig. 6.44(e).

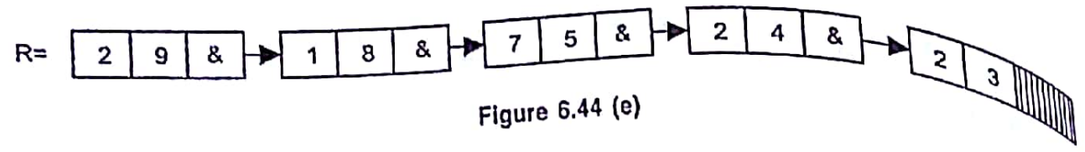


Figure 6.44 (e)

The list Q is completely scanned and reached to end. The remaining node from the list p will be inserted to list *R*. The list *R* is as Fig. 6.44(f).

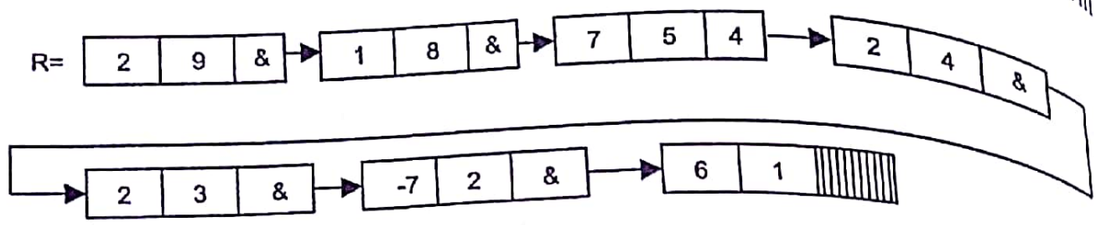


Figure 6.44 (f)

A program on the above is provided for understanding.

Example 6.35

Write a program to perform addition of two polynomials

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>
struct term
{
    float coef;
    int expn;
    struct term *next;
};
struct term *ployadd(struct term *, struct term *);
struct term *poly_in (struct term *);
struct term *insert(struct term *,float,int);
struct term *show (struct term *);

main ()
{
   struct term *t1,*t2,*t3;
   clrscr();
   t1=t2=t3=0;
   puts ("Polynomial (A): ");
   t1=poly_in(t1);
   puts ("Polynomial (B):");
   t2=poly_in(t2);
   t3=ployadd(t1,t2);
   puts ("Polynomial (A): ");
   show(t1);
   puts ("Polynomial (B): ");
   show(t2);
   puts ("Added Polynomial (C): ");
   show (t3);
}
struct term *poly_in(struct term *begin)
{
 int j,ex,n;
 float cof;
 printf ("\n Enter number of terms: ");
 scanf ("%d",&n);
 for (j=1;j<=n;j++)
 {
 printf ("Enter coefficient for term %d: " j);
 scanf ("%f",&cof);
 printf ("Enter exponent for term %d:" j);
 scanf ("%d",&ex);
 begin=insert(begin,cof,ex);
 begin=insert(begin,cof,ex);

 }
return begin;
}
struct term *insert(struct term *begin,float co, int e_x)
{
 struct term *pt,*temp;
 temp=(struct term*)malloc(sizeof(struct term));
 temp->coef=co;
 temp->expn=e_x;
 if (begin==0 || e_x>begin->expn)
 {
 temp->next=begin;
 begin=temp;
 }
 else
 {
  pt=begin;
  while (pt->next!=0 && pt->next->expn>e_x)
  pt=pt->next;
  temp->next=pt->next;
  pt->next=temp;
  if (pt->next==0) temp->next=0;
```

Explanation In this program at the beginning three pointers *t1*, *t2*, and *t3* are declared and initialized to NULL (0). The value zero means NULL. The pointer *t1* and *t2* denotes address of polynomial *t1* and *t2* respectively. The pointer *t3* points to the added polynomial. The pointer *t3* always points to the term added. In case the polynomials are empty, the resulting polynomial will be empty. That is why we are returning NULL value through the pointer *t3*. The NULL value of *t3* indicates that the polynomial is empty.

```
}
return begin;
}
struct term *ployadd(struct term *g1,struct term *g2)
{
struct term *t3=0,*p3, *temp;
if (g1==0 && g2==0) return t3;
while (g1!=0 && g2!=0)
{
   temp=(struct term*)malloc(sizeof(struct term));
      if (t3==0)
       {
       t3=temp;
       p3=t3;
       }
    else
       {
       p3->next=temp;
       p3=p3->next;
       }

       if (g1->expn>g2->expn)
       {
       temp->coef=g1->coef;
       temp->expn=g1->expn;
       g1=g1->next;
       }
       else  if (g2->expn>g1->expn)
       {
          temp->coef=g2->coef;
          temp->expn=g2->expn;
          g2=g2->next;
       }
       else if (g1->expn==g2->expn)
       {
          temp->coef=g1->coef+g2->coef;
          temp->expn=g1->expn;
          g1=g1->next;
          g2=g2->next;
       }
}
while (g1!=0)
{
   temp=(struct term*)malloc(sizeof(struct term));
   temp->coef=g1->coef;
```



```
   t3=temp;
   p3=t3;
   }
   else
   {
   p3->next=temp;
   p3=p3->next;
   }
   g1=g1->next;
}
while (g2!=0)
{
   temp=(struct term*)malloc(sizeof(struct term));
   temp->coef=g2->coef;
   temp->expn=g2->expn;
   if (t3==0)
    {
    t3=temp;
    p3=t3;
    }
    else
    {
    p3->next=temp;
    p3=p3->next;
    }
    g2=g2->next;
}
   p3->next=0;
   return t3;
}
struct term *show (struct term *pt)
{
   if (pt==0)
    {
    printf("Empty\n");
    return 0;
    }
    while (pt!=0)
    {
    printf("%.1fx^%d)+",pt->coef,pt->expn);
    pt=pt->next;
    }
    printf("\b\b\bn");
    return 0;
}
```

```
OUTPUT
***
Polynomial (A):
Enter number of terms: 1
Enter coefficient for term 1: 2
Enter exponent for term 1:3
Polynomial (B):
Enter number of terms: 1
Enter coefficient for term 1: 2
Enter exponent for term 1:3
Polynomial (A):
2.0x^3)+
Polynomial (B):
2.0x^3)+
Added Polynomial (C):
4.0x^3)+
```

Both polynomials are traversed, i.e. each and every node is visited. The new node is inserted to the third polynomial and followed by it coefficient and exponent are assigned.
We have written a ladder of *if-else* conditions.

1. if *t1->expo* > *t2->expo*: In this case, the value of current node of *t1* is assigned to current node of *t3*. The polynomial *linked list* (*A*) is traversed and pointer *t1* will point to the next node.
2. If *t2->expo*>*t1->expo*: In this case, the value of current node of polynomial (*B*) is assigned to current node of *t3*. The node from polynomial is added hence, the linked list (*B*) is traversed and pointer *t2* will point to next node.
3. If (*t1->expo*==*t2->expo*): In this case, where terms of both polynomials are same, the coefficient of both terms are added and the resulting value is assigned to node of third list. Here, both the lists are traversed and pointer *t1* and *t2* will point to next node of polynomial (*A*) and (*B*).

The same addition of two polynomials can be done from the following example.

Example 6.36

Write a program to perform addition of two polynomials.

Explanation The logic of this program is same as program 6.35.

Solution

```
# include <stdio.h>
# include <conio.h>
# include <malloc.h>
```



```
int expo;
struct poly *nterm;
};
int j,num;
struct poly begin;
void create_poly (struct poly *node)
{
char h;
begin.nterm=NULL;
node=&begin;
j=0;
printf ("\n Input n for end: ");
h=getchar();
while (h!='n')
{
node->nterm=(struct poly*) malloc(sizeof(struct poly));
node=node->nterm;
printf ("\n Enter coefficient value %d: ",j+1);
scanf ("%g",&node->coef);
printf ("\n Enter exponent value %d: "j+1);
scanf ("%d",&node->expo);
node->nterm=NULL;
fflush(stdin);
printf ("\n Input n for end: ");
h=getchar ();
j++;
}
printf ("\n Total Nodes = %d" J);
}
void show ( struct poly *node )
{
node=&begin;
node=node->nterm;
while (node)
{
printf ("\n + %g", node->coef);
printf ("X^%d", node->expo);
node=node->nterm;
}
}
void main ()
{
struct poly *node=(struct poly*)malloc(sizeof(struct poly));
clrscr();
create_poly(node);
show(node);
getch();
```

```
}

OUTPUT
Input n for end:
Enter coefficient value 1: 4
Enter exponent value 1: 5
Input n for end:
Enter coefficient value 2: 9
Enter exponent value 2: 8
Input n for end: n
 Total Nodes = 2
+ 4X^5 + 9X^8
```

6.28.2 Linked Dictionary

In compiler, creation of the linked dictionary has an important role. In compiler, the organization and maintenance of dictionary holding names and their corresponding values are maintained in the linked dictionary. This linked directory is called *symbol table*. While designing compiler, two factors, i.e. time and memory space are considered.

The process of compiler designing has various steps. The memory space and speed of symbol table algorithm have opposite relationship. The construction and referencing is an important factor of linked dictionary. Construction means insertion of symbols with their values available. Referencing means, obtaining values from the table.

The ratio of estimated number of insertion to reference is significant factor. In symbol-tables, access retrieve time and insertion time are very much associated. The quick symbol table can be simply constructed if large memory space is available. When a reference is obtained from arithmetic value of character (forming name), each name is stored in a unique memory location.

The easiest way to access the symbol table is linear search method. Using simple linked list the symbols are organized in a sequence. The insertion can be easily done by adding symbols at the rear of the list. If any particular symbol is to be searched, by traversing each node the required element can be searched.

The symbol table can also be searched using binary search. All the entries in the symbol table are sorted in ascending order. A middle entry is spotted and its value is determined. A separate topic is given for search techniques in this book. However, this method has some drawbacks.

6.28.3 Addition of Long Positive Integers

The application of adding long positive integers can be solved using circular list. There is always a restriction or limitation in the programming language that every data type has a lower and upper limit. The variables of a particular data type can have the values as per the range. Some time, it is very essential if any problem to be solved is having maximum range, which exceeds the limit of the range. In order to add two long integers, below mentioned steps are followed:

1. The digits of the given numbers are individually traversed from left to right.
2. In parallel, i.e. corresponding digits and a carry from prior digits sums, are added.

6.28.4 Searching in a Linked List

Finding some specific element in a given set of elements is called searching. Assume, we have a linked list of 10 numbers. In order to search a particular number in the list, all the elements of list are visited and compared with the expected one. When the given number matches with some number, search process completes. Occurrence of a particular number in a list for number of times can also be found. In other words, we can find out for how many times a particular number exist in the list. A program is illustrated below on this point.

Example 6.37

Write a program to create a linked list of integers and search for a given element.

Solution

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>

struct node
{
  int data;
  struct node *next;
};
int j;
struct node begin;
void search(struct node*);
void create_list(struct node*);
void show(struct node *);
void create_list(struct node *node)
{
 int n=1;
 begin.next=NULL;
 node=&begin;
 j=0;
 printf ("\n Input integers (0 to stop):");
 while (n)
 {
 scanf ("%d", &n);
 if (n==0) break;
 else node->next=(struct node*) malloc(sizeof(struct node));
 node->data=n;
 node=node->next;
 node->next=NULL;
 fflush(stdin);
 }
 }
 void search(struct node *node)
```

Explanation In this program a list is created with *create* () function. The number, which is to be searched, is entered. In the *search* () function the entered number is compared with the linked list elements. When equal number is found the number along with its position in the linked list is displayed. The *show* () function is used to display all the linked list elements. Thus, through traversing and comparing elements search process is done.

```
{
 int node_num=0;
 int s_node;
 int flag=0;
 node=&begin;
 printf ("\n Enter number to be searched: ");
 scanf ("%d",&s_node);
 if (node==NULL)
 printf ("\n List is empty ");
 while (node)
 {
 if (s_node==node->data)
 {
 printf ("\n search is successful");
 printf ("\n Position of %d from beginning of the list: %d",s_node,node_num+1);
 node=node->next;
 flag=1;7
 }
 else
 node=node->next;
 node_num++;
 }
 if (!flag)
 {
 printf ("\n Search is unsuccessful");
 printf ("\n %d does not found in the list", s_node);
 }
 }
 void show (struct node * node)
 {
 node=&begin;
 while(node->next)
 {
 printf(" %d",node->data);
 node=node->next;
 }
 }
 void main ()
 {
 struct node *node=(struct node *) malloc (sizeof(struct node));
 clrscr();
 create_list(node);
 printf ("\n List: ");
 show(node);
 search(node);
 }
```



```
OUTPUT
Input integers (0 to stop):
1
2
3
4
5
9
7
5
0
List: 1 2 3 4 5 9 7 5
Enter number to be searched: 3
search is successful
Position of 3 from beginning of the list: 3
```

6.28.5 Sorting a Linked List

Example 6.38

Write a program to create linked list and sort the list in ascending order.

Solution

```
# include <stdio.h>
# include <conio.h>

struct node
{
 int value;
 struct node *next;
};
  void create (void);
  void sort(void);
  void show (void);
  struct node *rear;
  int nodes;
  struct node *head;
  main ()
{
  clrscr();
  create();
  sort();
  show ();
}
void create ( )
```

Explanation In this program the *create()* function is used for inputting the elements into the linked list. The *sort()* is used to sort the linked list elements in ascending order. The *struct num* is used to store the linked list. The *show()* displays the sorted elements of linked list.

```
{
   struct node *item;
   printf ("Enter numbers( 0 to exit): ");
   if (head==NULL)
 {
   head=(struct node*)malloc(sizeof(struct node));
   scanf ("%d",&head->value);
   head->next=NULL;
   rear=head;
 }
 while (1)
 {
   item=(struct node *)malloc(sizeof(struct node));
   scanf ("%d",&item->value);

   if (item->value==0) break;
   item->next=NULL;
   rear->next=item;
   rear=item;
 }
}
void sort()
{
 int temp;
 struct node *nlink, *count;
 if (head==NULL)
 {
 printf ("List is empty");
 }
 nlink=head;
    for(;nlink->next!=NULL;nlink=nlink->next)
for(count=nlink->next;count!=NULL;count=count->next)
if(nlink->value>count->value)
{
temp=nlink->value;
nlink->value=count->value;
count->value=temp;
}
}
 void show ( )
 {
   printf ("\n Sorted list is: ");
   while (head!=NULL)
{
   nodes++;
   printf (" %d ",head->value);
   head=head->next;
}
nodes=0;
}
```

OUTPUT

```
Enter numbers( 0 to exit): 12 1 2 34 5 65 4 0
Sorted list is: 1 2 4 5 12 34 65
```


Summary

This chapter presents the following points:

1. Series of linearly arranged numbers is a list. The list can be of basic data type or custom data type. The first element of the list is called HEAD and the last element TAIL.
2. Static implementation of list can be implemented using arrays. Examples have been illustrated on this point.
3. Pointers are used for implementation of linked list. The linked list is a major application of the dynamic implementation.
4. In the list of elements, for any location *n*, *(n–1)* is *predecessor* and *(n+1)* is *successor*. In other words, for any location *n* in the list, the left element is *predecessor* and the right element is *successor*.
5. The merging is a procedure in which two or more lists can be combined and third list is created.
6. A linked list is a dynamic data structure. It is an ideal technique to store data when user does not know in advance how many elements are to be stored. The dynamic implementation of list using pointers is also known as *linked list*.
7. *Singly linked list*: In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner.
8. *Doubly linked list*: In this type of linked list the data structure holds two-pointer fields.
9. *Circular list*: In this list the first and last elements are adjacent. This type of list has neither end nor starting node.
10. *A circular doubly linked list*: In this type of linked list the structure field contains three fields. Two link fields and one data field.
11. *Creation*: The linked list creation operation involves allocation of structure size memory to pointer of the same structure.
12. *Traversing*: It is the procedure of passing through (visiting) all the nodes of the linked list from starting to end.
13. *Display*: The operation in which data field of every node is accessed and are displayed on the screen.
14. *Splitting of a linked list*: Once a singly linked list is created, it can be divided into many sub-lists. The technique is very simple. Put the NULL value in the node where you want to finish the first sub-list in the main list. Later, the next node will act as first node for the second sub-list.
15. *Doubly linked list*: Each node of the doubly linked list has two pointer fields and holds the address of predecessor and successor elements. These pointers enable bi-directional traversing, i.e. traversing the list in backward and forward direction.

2018-10-25

16. *Circular doubly linked list*: A circular doubly linked list has both successor and predecessor pointers. In this type insertion and deletion operations are easily performed as compared to other type of linked lists.
17. *Polynomial manipulation*: The linked list is used for implementation of polynomial. The operations such as addition, multiplication etc., are performed. To get better proficiency in processing each polynomial is stored in decreasing order.
18. *Linked dictionary*: While compiling a program the linked dictionary has an important role. In compiler, the organization and maintenance of dictionary holding names and their corresponding values are maintained in the linked dictionary. This linked directory is called *symbol table*. While designing compiler, two factors, i.e. time and memory space are considered.

Exercises

A. Answer the following questions:
1. What is the static list? How is it implemented?
2. Explain traversal of static list.
3. What is predecessor and successor of an element in a list?
4. The first element does not have predecessor and the last element does not have successor in a list. Explain.
5. Explain insertion and deletion operation with the static list.
6. Explain merging operation of two lists.
7. Differentiate between static and dynamic list.
8. Explain the use of pointer head in the linked list.
9. Mention types of linked lists with explanation. On which basis are the linked lists classified?
10. Explain applications of linked list.

B. Select the appropriate options from the following:
1. The elements of static lists are stored in
   a. successive memory locations
   b. random memory locations
   c. alternate memory locations
   d. all of the above.
2. The elements of linked lists are stored in
   a. successive memory locations
   b. random memory locations
   c. alternate memory locations
   d. all of the above.
3. Name the function that is used for memory allocation in implementation of linked list
   1. malloc ()
   2. realloc()
   3. free()
   4. both (a) and (c).
4. If it is decided to create the linked list in C++, function that allocates memory is
   a. malloc ()
   b. new
   c. delete
   d. both (a) and (b).
5. The free() function is used to
   a. release the memory
   b. to unlink the node
   c. to unlink the fist and last node
   d. none of the above.
6. The pointer head points to the
   a. first node
   b. last node
   c. either first or last node
   d. none of the above.

7. This type of linked list does not have null value in the last node
   a. circular linked list
   b. singly linked list
   c. static list
   d. none of the above.
8. The first node of this type of linked list has NULL value
   a. doubly linked list
   b. doubly circular linked list
   c. singly linked list
   d. all of the above.
9. This type of linked list does not have first and last node
   a. circular linked list
   b. singly linked list
   c. doubly linked list
   d. static list.
10. When the malloc () function returns NULL value it means
    a. memory is not allocated
    b. memory is allocated but no data entered
    c. memory is allocated
    d. none of the above.
11. If memory is not allocated, what further action will you suggest?
    a. try again
    b. exit from the program
    c. both (a) & (b)
    d. continue the program.

C. Attempt the following programs:
1. Write a program to remove first and last node of the linked list.
2. Write a program to duplicate a linked list from other linked list. Remove all odd numbers from the new list and display both of the linked lists.
3. Write a program to create the linked list and remove all the duplicate elements of the list. The list should have unique elements.
4. Write a program to find successor of the given element in a linked list.
5. Write a program to create and display the elements of a circular linked list.
6. Write a program to terminate the program when malloc () returns to NULL.
7. Write a program to merge two linked list and display elements.
8. Write a program to swap first and last nodes of the linked list.
9. Write a program to create a linked list with integer elements and count the occurrence of given number in linked list.
10. Write a program to find the number of even and odd elements in a linked list.
11. Write a program to find the average value of elements of a linked list.
12. Write a menu driven program to perform the following operations on linked list:
    1. Create
    2. Insert
    3. Delete
    4. Display
    5. Exit.
13. Write a program to multiply two polynomials.
14. Write a program to create and display circular linked list containing five elements.
15. Write a program to find the minimum and maximum value from the linked list.

D. What will be the output of the following programs?

1.
```
main ()
{
  int sim[5]={0,1,2,3,4},j;
  for (j=4;j>0;j—)
  printf (" %d ",sim[j]);
}
```

2.

```
main ()
{
    int sim[]={0,1,2,3,4}j=3;
    clrscr();
    printf ("\n Predecessor of %d = %d",sim[j],
    sim[j-1]);
    printf ("\n Successor of %d = %d", sim[j],
    sim[j+1]);
}
```

3.

```
main ()
{
    int num[8]={1,2,3,4,5,6,7,8}j=0;
    clrscr();
    while (j<=7)
{
    if (j>=5)
    num[j]=num[j+1];
    j++;
}
j=num[j]=0;
while (j<8)
printf (" %d ",num[j++]);
}
```

4.

```
main ()
{
int num[8]={1,2,3,4,5,6,7,8}j=0,n=-5;
clrscr();
while (j<8)
{
    if (num[j++]==abs(n))
{
printf ("Elemenent found");
exit(0); } }
printf ("Element not found");

}
```

5.

```
# include <stdio.h>
# include <conio.h>
main ()

{
int array[6]={0}j,no1,no2;
clrscr();

printf ("\n Enter elements: " );
for (j=0;j<5;j++)
scanf ("%d",&array[j]);

printf ("\n Enter elements for insertion:");
scanf ("%d %d",&no1,&no2);
for (j=5;j>=1;j—)
array[j]=array[j-1];

array[0]=no1;
array[6]=no2;

printf("\nList after insertion of elements\n");
for (j=0;j<=6;j++)
printf (" %d ", array[j]);
getch();
}
```

Chapter 7

Storage Management

Chapter Outline

7.1 Introduction
7.2 Allocation Techniques
7.3 Memory Representation
7.4 Boundary Tag System
7.5 Storage Allocations
7.6 Storage Release
7.7 Buddy System
7.8 Binary Buddy System
7.9 Compaction
7.10 Garbage Collections

7.1 INTRODUCTION

The fundamental purpose of any program is to manipulate data and its storage in the computer memory. Storage management consists of techniques that are used to manage the heap. Two memory management techniques are used for this purpose. They are:

1. Static storage management
2. Dynamic storage management.

Static Storage Management It is necessary to load the program into the memory before execution of a program. Before execution of a program, it is essential that the system should have enough memory. When program execution starts, it takes memory through operating system. Once the memory is allocated to the program, the memory allocated cannot be increased or decreased during program execution. The same memory cannot be used by other programs. All these are taken care of by the operating system. For example, an array is best example of static implementation in which the memory allocated is fixed in size.

Dynamic Storage Management The dynamic storage management allows the user to manage the memory during program execution. According to the request made by the program,