Linked Lists

6.1 INTRODUCTION

If the memory is allocated before the execution of a program, it is fixed cannot be changed. We have to adapt an alternative strategy to allocated memory only when it is required. There is a special data structure called **linked list** that provides a more flexible storage system and it does not require the use of arrays.

For stacks and queues, we employed arrays to represent them in computer memory. When we choose array representation (also called sequential representation) it is necessary to declare in advance the amount of memory to be utilized. We do this by declaring the maximum size of an array. When we really take up arrays, all the memory may not be unused even though allocated. This leads to unnecessary wastage of memory. The memory is very important resources. So, we should handle it efficiently.

6.2 LINKED LISTS

Linked lists are special list of some data elements linked to one another. The logical ordering is represented by having each element pointing to the next element. Each element is called a *node*, which has two parts. INFO part which stores the information and POINTER which points to the next element.

Following Figure 6.1 shows both types of lists (singly linked list and doubly linked lists).

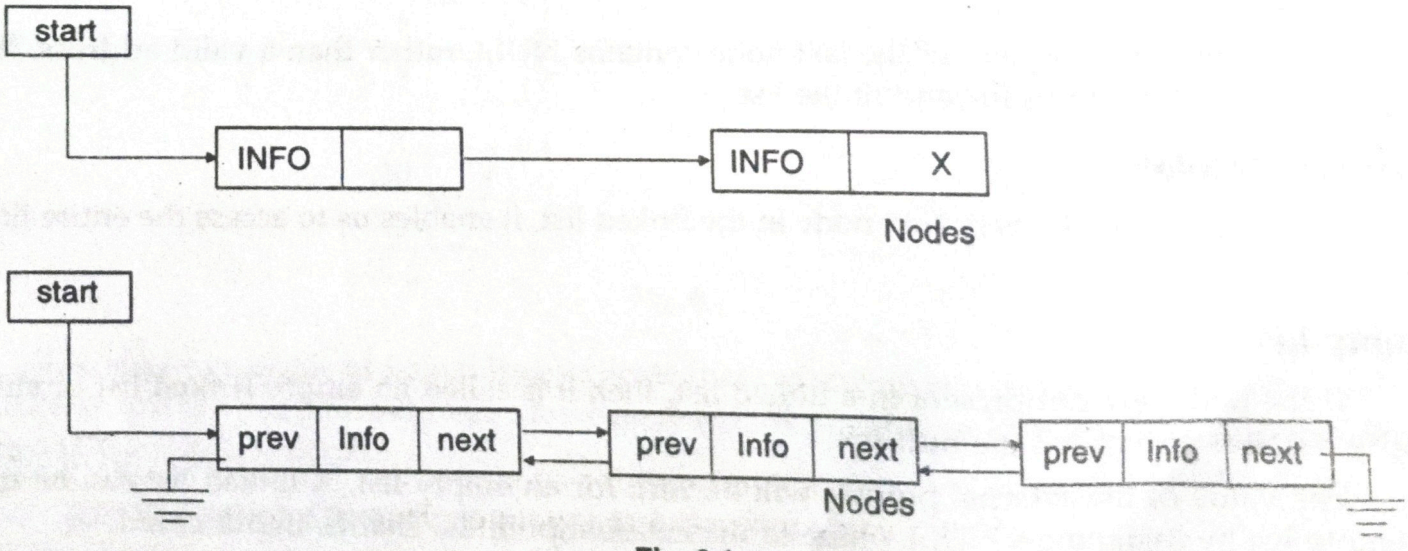


Fig. 6.1.

In singly linked list nodes have one pointer (Next) pointing to the next node, whereas nodes of doubly linked lists have two pointers (prev and next). *Prev* points to the previous node and *next* points to the next node in the list.

6.2.1 Advantages

Linked lists have many advantages. Some of the very important advantages are :

(*i*) ***Linked lists are dynamic data structures.*** That is, they can grow or shrink during the execution of a program.

(*ii*) ***Efficient memory utilization.*** Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (removed) when it is no longer needed.

(*iii*) ***Insertion and deletions are easier and efficient.*** Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

(*iv*) Many complex applications can be easily carried out with linked lists.

6.2.2 Disadvantages

(*i*) More memory : If the number of fields are more, then more memory space is needed.

(*ii*) Access to an arbitrary data item is little bit cumbersome and also time-consuming.

6.3 KEY TERMS

As you know, a linked list is a non-sequential collection of data items called **nodes**. These nodes in principle are **structures** containing fields. Each node in a linked list basically two fields.

1. Data field
2. Link field

The **Data** field contains an actual value to be stored and processed. And, the **link** field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer. Therefore, the elements in a linked list are ordered not by their physical placement in memory but their logical links stored as part of the data within the node itself. In other words, the logical and physical ordering of data items in a linked list need not be the same. But in sequential representation these ordering are the same.

Null Pointer

Note that, the link field of the last node contains NULL rather than a valid address. It is a **null pointer** and indicates the end of the list.

External Pointer

It is a pointer to the very first node in the linked list, it enables us to access the entire linked list.

Empty List

If the nodes are not present in a linked list, then it is called an **empty linked** list or simply **empty list**. It is also called the **null list**.

The value of the external pointer will be zero for an empty list. A linked list can be made an empty list by assigning a NULL value to the external pointer. That is, in our case.

```
Start = NULL ;
```

Notations

Let P be s pointer to node in a linked list. Then, we can form the following notations. To do something on linked lists.

- Node (p) ; a node pointed to by the pointer p.
- Data (p) ; data of the node pointed to by p.
- Link (p) ; address of the next node that follows the node pointed to by the pointer p.

Suppose, if the value P = 2000, then

Node (p) = Node (2000)

Is the second node, and

Data (p) = Data (2000) = 20
Link (p) = Link (2000) = 3000

Which is the address of the third node.

EXAMPLE 6.1. *Consider a linked list in memory where each node of the list contains single character.*

SOLUTION. Start = 4, Data [4] = 'M' and Link [4] (next pointer) = 2

Data [2] = 'L' Link [2] = 5
Data [5] = 'P' Link [5] = 3
Data [3] = 'Q' Link [3] = 1
Data [1] = 'S' Link [1] = 8
Data [8] = 'N' Link [8] = 0 *i.e.*, NULL

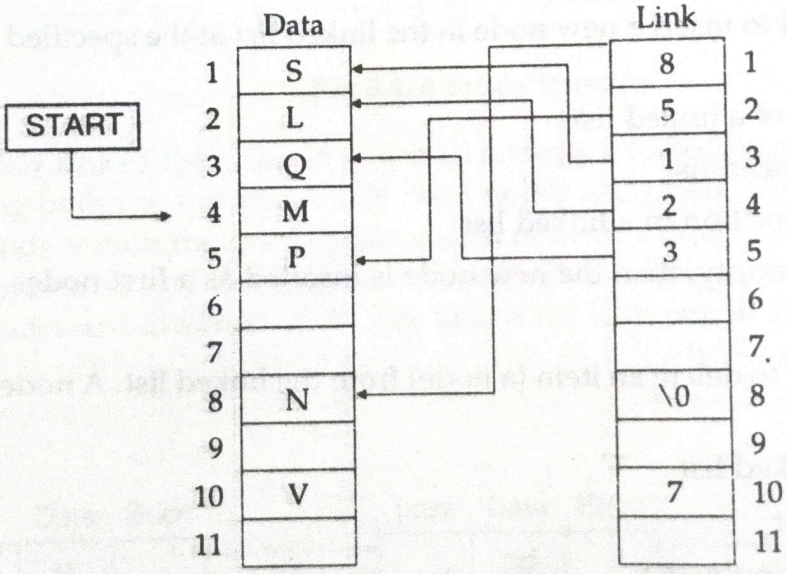


Fig. 6.2.

The above example of linked lists indicate that two nodes of a list need not occupy adjacent elements in the arrays Data and Link. However, each list must have its own pointer variable giving the location of its first node.

6.4 REPRESENTATION OF LINEAR LINKED LIST

Suppose we want to store list of integer numbers, then the linear linked list can be represented in memory with the following declarations.

```
struct node
{
    int a ;
```

```
        struct node *next ;
    } ;
    typedef struct node NODE ;
    NODE *start ;
```

The above declaration defines a new data type, whose each element is of type *node_type* and gives it a name node.

6.5 OPERATION ON LINKED LIST

The basic operations to be performed on the linked lists are as follows :

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Concatenation
7. Display

Let us define these terminology before applying them to any of the four types of linked lists.

Creation

This operation is used to create a linked list. Here, the constituent node is created as and when it is required and linked to the list to preserve the integrity of the list.

Insertion

This operation is used to insert a new node in the linked list at the specified position. A new node may be inserted

- At the beginning of a linked list
- At the end of a linked list
- At the specified position in a linked list
- If the list itself is empty, then the new node is inserted as a first node.

Deletion

This operation is used to delete an item (a node) from the linked list. A node may be deleted from the

- Beginning of a linked list
- End of a linked list
- Specified position in the list.

Traversing

It is a process of going through all the nodes of a linked list from one end to the other end. If we start traversing from the very first node towards the last node, it is called forward traversing. If the desired element is found, we signal operation "SUCCESSFULL". Otherwise, we signal it as "UNSUCCESSFULL".

Concatenation

It is the process of appending (joining) the second list to the end of the first list consisting of m nodes. When we concatenate two lists, the second list has n nodes, then the concatenated list will be having $(m + n)$ nodes. The last node of the list is modified so that it is now pointing to the first node in the second list.

Display

This operation is used to print each and every node's information. We access each node from the beginning (or the specified position) of the list and output the data housed there.

You will be studying the algorithms for each of these basic operations in the successive sections.

6.6 TYPES OF LINKED LIST

Basically, we can put linked lists into the following four types :

- Singly-linked list
- Doubly linked list
- Circular linked list
- Circular doubly linked list

1. A **singly linked list** is one in which all nodes are linked together in some sequential manner. Hence, it is also called **linear linked list**. Clearly it has the beginning and the end. The problem with this list is that we cannot access the predecessor of node from the current node. This can be overcome in doubly linked lists.

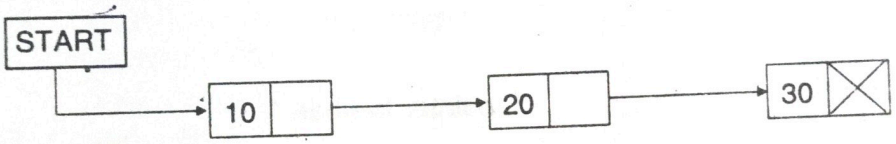


Fig. 6.3. A circular linked list.

2. A **doubly linked list** is one in which all nodes are linked together by multiple links which help in accessing both the successor node (next node) and predecessor node (previous node) for any arbitrary node within the list. Therefore each node in a doubly linked list fields (pointers) to the left node (previous) and the right node (next). This helps to traverse the list in the forward direction and backward direction. A doubly linked list is shown in Figure 6.3Ai.

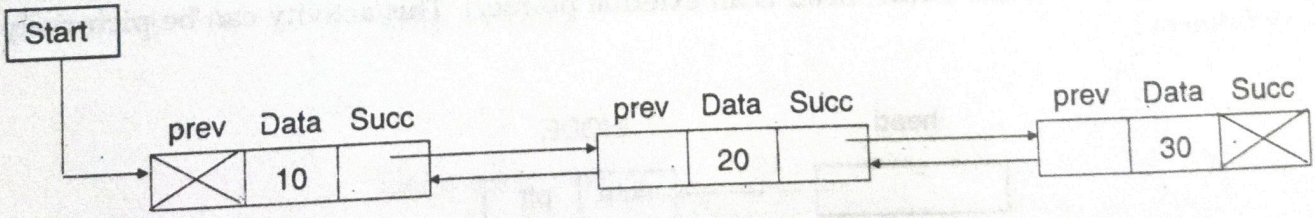


Fig. 6.3A. A doubly linked list.

3. **A circular linked list** is one which has no **beginning** and **no end**. A singly linked list can be made a circular linked list by simply sorting the address of the very first node in the link field of the last node. A circular linked list is shown in Figure 6.4.

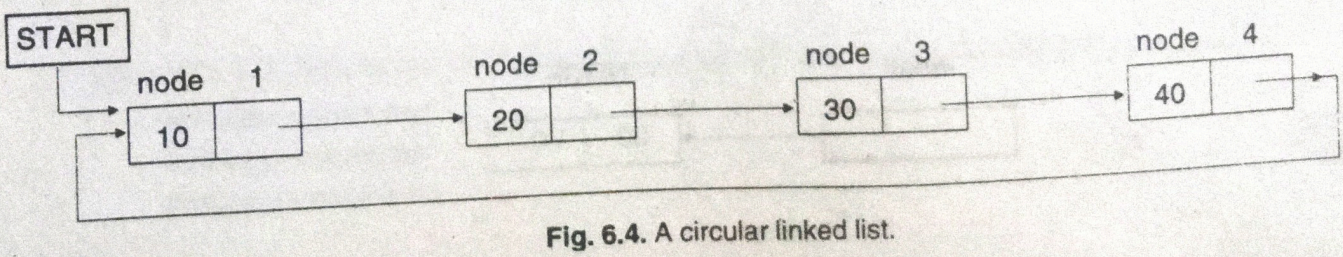


Fig. 6.4. A circular linked list.

4. A **circular doubly linked list** is one which has both the successor pointer and predecessor pointer in circular manner. It is shown in Figure 6.5.

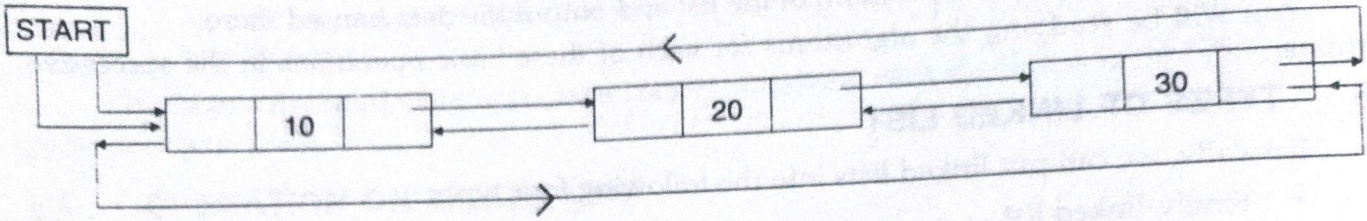


Fig. 6.5. A circular doubly linked list.

6.7 SINGLY LINKED LIST

A singly linked list is a dynamic data structure. It may grow or shrink. Growing or shrinking depends on the operations made. Let us start the study of the singly list by first creating it.

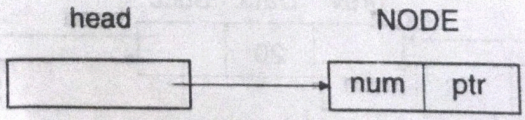
In C, a linked list is created using structures, pointers and dynamic memory allocation function **malloc**. We consider **head** as an external pointer. This helps in creating and accessing other nodes in the linked list. Consider the following structure definition and head creation.

```
struct node
{
    int num ;
    struct node *ptr ;                    /* pointer to node */
} ;
typedef struct node NODE ;                /* type Definition making it abstract data type */
NODE *head ;                              /* pointer to the  node of linked list */
head = (NODE *) malloc (sizeof (NODE)) ;  /*  dynamic memory allocation */
```

When the statement

```
head = (NODE *) malloc (sizeof (NODE)) ;
```

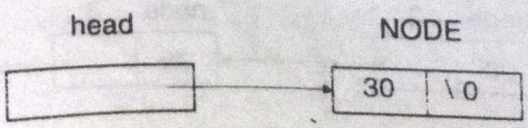
is executed, a block of memory sufficient to store the NODE is allocated and assigns **head** as the starting address of the NODE (Now, head is an external pointer). This activity can be pictorially shown as follows :



Now, we can assign values to the respective fields of NODE.

```
head -> number = 30 ;      /* Data fields contains value 30 */
head -> ptr = ' \0' ;      /* Null pointer assignment */
```

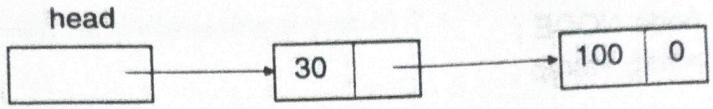
Thus, new vision of the NODE would be like this :



Any number of nodes can be created and linked to the existing node. Suppose we want to add another node to the above list, then the following statements are required.

```
head -> ptr = (NODE *)  malloc (sizeof (NODE)) ;
head -> ptr.num = 100 ;
head -> ptr.ptr = '\0' ;
```

Thus, the intened linked list would have the following view




```
struct node
{
    int num ;
    struct node *ptr ;
} ;
void create( )
{
    typedef struct node NODE ;
    NODE *head, *first, *temp ;
    int count = 0 ;
    char choice ;
    first = NULL ;
    do
    {
        head = (NODE *) malloc (sizeof (NODE)) ;
        printf ("Enter the data item\n") ;
        scanf ("%d", &head->num) ;
        if (first != NULL)
        {
            temp -> ptr = head ;
            temp = head ;
        }
        else
        {
            first = temp = head ;
        }
        fflush(stdin) ;
        printf ("Do you want to continue (type y or n) ? \n") ;
        scanf ("%c", &choice) ;
    }
    while ((choice = = 'y') || (choice = = 'Y')) ;
}
```

EXAMPLE 6.2. *Simply create element in the link list and display the link list's element.*

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

```
void main( )
{
    struct node
    {
        int num ;
        struct node *ptr ;
    } ;
    typedef struct node NODE ;
    NODE *head, *first, *temp ;
    int count = 0 ;
    char choice ;
    clrscr( ) ;
    first = NULL ;
    do
    {
        head = (NODE *) malloc (sizeof (NODE)) ;
        printf ("Enter the data item\n") ;
        scanf ("%d", &head -> num) ;
        if (first != NULL)
        {
            temp -> ptr = head ;
            temp = head ;
        }
        else
        {
            first = temp = head ;
        }
        fflush(stdin) ;
        printf ("Do you want to continue (type y or n) ? \n") ;
        scanf ("%c", &choice) ;
    }
    while ((choice = = 'y') || (choice = = 'Y')) ;
    temp -> ptr = NULL ;
    temp = first ;
    printf ("Status of the linked list is \n") ;
    while (temp != NULL)
    {
        printf ("%d \n", temp -> num) ;
        count + + ;
        temp = temp -> ptr ;
    }
    printf ("NO of nodes in the list = %d \n", count) ;
    getch( ) ;
}
```