3.3 Circular linked list

In a single linked list, for accessing any node of linked list, we start traversing from first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of single linked list. In a single linked list, link part of last node is NULL, if we utilize this link to point to the first node then we can have some advantages. The structure thus formed is called a circular linked list. The following figure shows a circular linked list.

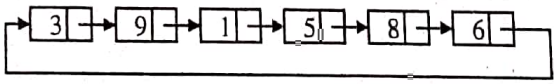

Figure 3.25

Each node has a successor and all the nodes form a ring. Now we can access any node of the linked list without going back and starting traversal again from first node because list is in the form of a circle and we can go from last node to first node.

We take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then `last->link` will point to the first node.

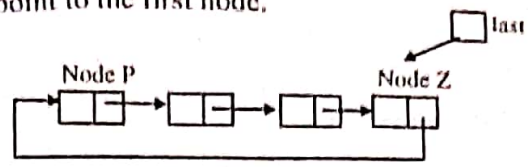

Figure 3.26

In the figure 3.26, the pointer `last` points to node Z and `last->link` points to node P. Let us see why we have taken a pointer that points to the last node instead of first node. Suppose we take a pointer `start` pointing to first node of circular linked list. Take the case of insertion of a node in the beginning.

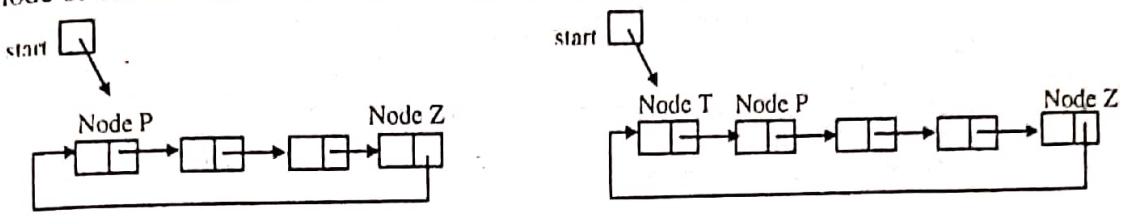

Figure 3.27

For insertion of node T in the beginning we need the address of node Z, because we have to change the link of node Z and make it point to node T. So we will have to traverse the whole list. For insertion at the end it is obvious that the whole list has to be traversed. If instead of pointer `start` we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So insertion in the beginning or at the end takes constant time irrespective of the length of the list.

If the circular list is empty the pointer last is `NULL`, and if the list contains only one element then the link of `last` points to `last`.

Now let us see different operations on the circular linked list. The algorithms are similar to that of single linked list but we have to make sure that after completing any operation the link of last node points to the first.

```
/*P3.3 Program of circular linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
      int info;
      struct node *link;
};
struct node *create_list(struct node *last);
void display(struct node *last);
struct node *addtoempty(struct node *last,int data);
struct node *addatbeg(struct node *last,int data);
struct node *addatend(struct node *last,int data);
struct node *addafter(struct node *last,int data,int item);
struct node *del(struct node *last,int data);

main()
{
      int choice,data,item;
      struct node *last=NULL;

      while(1)
      {
            printf("1.Create List\n");
            printf("2.Display\n");
            printf("3.Add to empty list\n");
            printf("4.Add at_beginning\n");
            printf("5.Add at end\n");
            printf("6.Add after \n");
            printf("7.Delete\n");
            printf("8.Quit\n");
```

```
            printf("Enter your choice : ");
            scanf("%d",&choice);

            switch(choice)
            {
             case 1:
                    last=create_list(last);
                    break;
             case 2:
                    display(last);
                    break;
             case 3:
                    printf("Enter the element to be inserted : ");
                    scanf("%d",&data);
                    last=addtoempty(last,data);
                    break;
            case 4:
                    printf("Enter the element to be inserted : ");
                    scanf("%d",&data);
                    last=addatbeg(last,data);
                    break;
            case 5:
                    printf("Enter the element to be inserted : ");
                    scanf("%d",&data);
                    last=addatend(last,data);
                    break;
            case 6:
                    printf("Enter the element to be inserted : ");
                    scanf("%d",&data);
                    printf("Enter the element after which to insert : ");
                    scanf("%d",&item);
                    last=addafter(last,data,item);
                    break;
            case 7:
                    printf("Enter the element to be deleted : ");
                    scanf("%d",&data);
                    last=del(last,data);
                    break;
            case 8:
                    exit(1);
            default:
                    printf("Wrong choice\n");
            }/*End of switch*/
        }/*End of while*/
}/*End of main()*/
```

3.3.1 Traversal in circular linked list

First of all we will check if the list is empty. After that we will take a pointer p and make it point to the first node.

```
p = last->link;
```

The link of last node does not contain NULL but contains the address of first node so here the terminating condition of our loop becomes (p!=last->link). We have used a do-while loop in the function display() because if we take a while loop then the terminating condition will be satisfied in the first time only and the loop will not execute at all.

```
void display(struct node *last)
{
        struct node *p;

        if(last == NULL)
        {
                printf("List is empty\n");
```

```
                return;
        }
        p = last->link;
        do
        {
                printf("%d ",p->info);
                p = p->link;
        }while(p!=last->link);
        printf("\n");
}/*End of display()*/
```

3.3.2 Insertion in a circular Linked List

3.3.2.1 Insertion at the beginning of the list

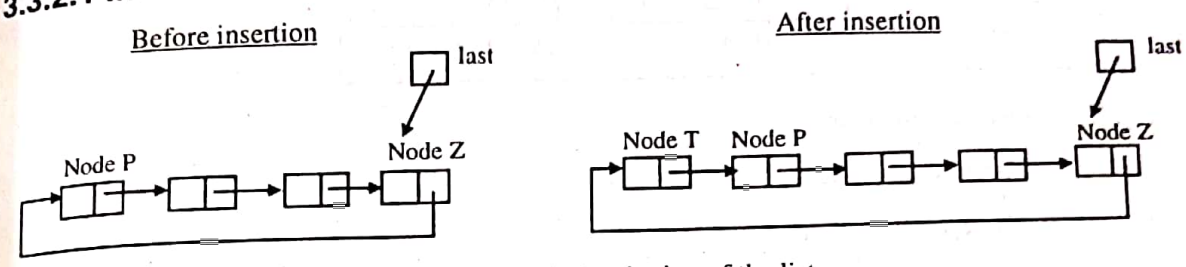

Figure 3.27 Insertion at the beginning of the list

Before insertion, P is the first node so `last->link` points to node P.

After insertion, link of node T should point to node P and address of node P is in `last->link`

```
tmp->link = last->link;
```

Link of last node should point to node T.

```
last->link = tmp;
```

```
struct node *addatbeg(struct node *last,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = last->link;
        last->link = tmp;
        return last;
}/*End of addatbeg()*/
```

3.3.2.2 Insertion in an empty list

Figure 3.29 Insertion in an empty list

After insertion, T is the last node so pointer `last` points to node T.

```
last = tmp;
```

We know that `last->link` always points to the first node, here T is the first node so `last->link` points to node T (or `last`).

```
last->link = last;
```

```
struct node *addtoempty(struct node *last,int data)
```

```
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        last = tmp;
        last->link = last;
        return last;
}/*End of addtoempty()*/
```

3.3.2.3 Insertion at the end of the list

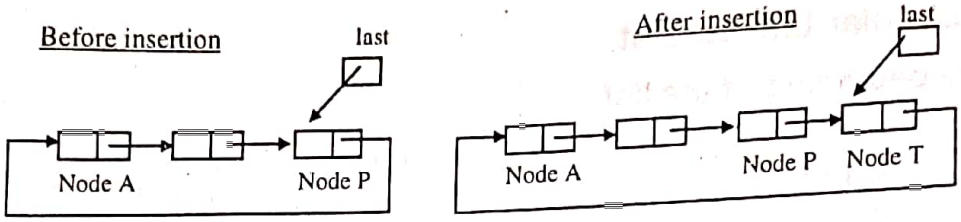

Figure 3.30 Insertion at the end of the list

Before insertion, `last` points to node P and `last->link` points to node A

Link of T should point to node A and address of node A is in `last->link`.

```
tmp->link = last->link;
```

Link of node P should point to node T

```
last->link = tmp;
```

`last` should point to node T

```
last = tmp;
```

The order of the above three statements is important.

```
struct node *addatend(struct node *last,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->link = last->link;
        last->link = tmp;
        last = tmp;
        return last;
}/*End of addatend()*/
```

3.3.2.4 Insertion in between the nodes

The logic for insertion in between the nodes is same as in single linked list. If insertion is done after the last node then the pointer `last` should be updated. The function `addafter()` is given below-

```
struct node *addafter(struct node *last,int data,int item)
{
        struct node *tmp,*p;
        p = last->link;
        do
        {
                if(p->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
                        p->link = tmp;
                        if(p==last)
                                last = tmp;
```

```
                return last;
            }
            p = p->link;
        }while(p!=last->link);
        printf("%d not present in the list\n",item);
        return last;
}/*End of addafter()*/
```

3.3.3 Creation of circular linked list

For inserting the first node we will call `addtoempty()`, and then for insertion of all other nodes we will call `addatend()`.

```
struct node *create_list(struct node *last)
{
    int i,n,data;
    printf("Enter the number of nodes : ");
    scanf("%d",&n);
    last=NULL;
    if(n==0)
        return last;
    printf("Enter the element to be inserted : ");
    scanf("%d",&data);
    last=addtoempty(last,data);
    for(i=2; i<=n; i++)
    {
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        last=addatend(last,data);
    }
    return last;
}/*End of create_list()*/
```

3.3.4 Deletion in circular linked list

3.3.4.1 Deletion of the first node

Before deletion, `last` points to node Z and `last->link` points to node T

After deletion, link of node Z should point to node A, so `last->link` should point to node A. Address of node A is in link of node T.

```
last->link = tmp->link;
```

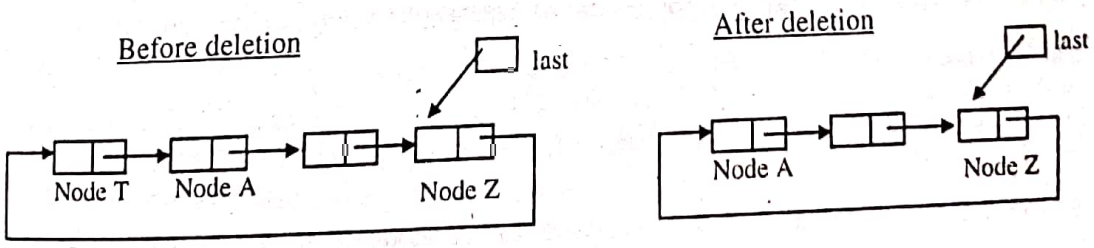

Figure 3.31 Deletion of the first node

3.3.4.2 Deletion of the only node

If there is only one element in the list then we assign NULL value to `last` pointer because after deletion there will be no node in the list.

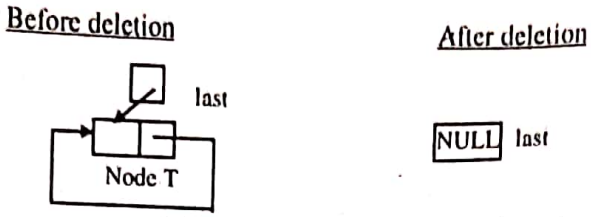

Figure 3.32 Deletion of the only node

There will be only one node in the list if link of last node points to itself. After deletion the list will become empty so NULL is assigned to last.

```
last = NULL;
```

3.3.4.3 Deletion in between the nodes

Deletion in between is same as in single linked list

3.3.4.4 Deletion at the end of the list

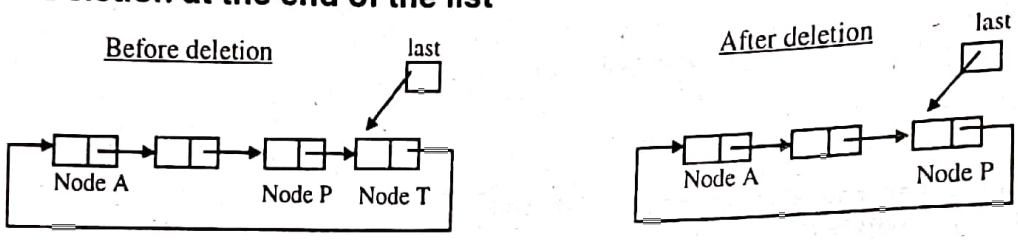

Figure 3.33 Deletion at the end of the list

Before deletion, `last` points to node T and `last->link` points to node A, `p` is a pointer to node P. After deletion, link of node P should point to node A. Address of node A is in `last->link`.

```
p->link = last->link;
```

Now P is the last node so `last` should point to node P.

```
last = p;
```

```
struct node *del(struct node *last,int data)
{
        struct node *tmp,*p;
        if(last == NULL)
        {
                printf("List is empty\n");
                return last;
        }
        if(last->link == last && last->info == data) /*Deletion of only node*/
        {
                tmp = last;
                last = NULL;
                free(tmp);
                return last;
        }
        if(last->link->info == data)  /*Deletion of first node*/
        {
                tmp = last->link;
                last->link = tmp->link;
                free(tmp);
                return last;
        }
        p = last->link;          /*Deletion in between*/
        while(p->link!=last)
        {
```

```
            if(p->link->info == data)
            {
                    tmp = p->link;
                    p->link = tmp->link;
                    free(tmp);
                    return last;
            }
            p = p->link;
    }
    if(last->info == data) /*Deletion of last node*/
    {
            tmp = last;
            p->link = last->link;
            last = p;
            free(tmp);
            return last;
    }
    printf("Element %d not found\n",data);
    return last;
}/*End of del()*/
```

We have studied circular lists which are singly linked. Double linked lists can also be made circular. In this case the next pointer of last node points to first node, and the `prev` pointer of first node points to the last node.

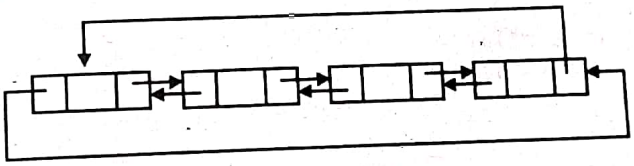

Figure 3.34 Circular double linked list