3.2 Doubly linked list

The linked list that we have studied contained only one link, this is why these lists are called single linked lists or one way lists. We could move only in one direction because each node has address of next node only. Suppose we are in the middle of linked list and we want the address of previous node then we have no way of doing this except repeating the traversal from the starting node. To overcome this drawback of single linked list we have another data structure called doubly linked list or two way list, in which each node has two pointers. One of these pointers points to the next node and the other points to the previous node. The structure for a node of doubly linked list can be declared as-

```
struct node{
                struct node *prev;
                int info;
                struct node *next;
        };
```

Here `prev` is a pointer that will contain the address of previous node and `next` will contain the address of next node in the list. So we can move in both directions at any time. The `next` pointer of last node and `prev` pointer of first node are NULL.

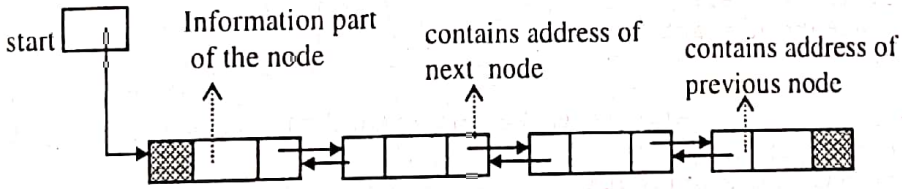

Figure 3.15 Doubly linked list

The basic logic for all operations is same as in single linked list, but here we have to do a little extra work because there is one more pointer that has to be updated each time. The `main()` function for the program of doubly linked list is-

```
/*P3.2 Program of doubly linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        struct node *prev;
        int info;
        struct node *next;
};
struct node *create_list(struct node *start);
void display(struct node *start);
struct node *addtoempty(struct node *start,int data);
struct node *addatbeg(struct node *start,int data);
struct node *addatend(struct node *start,int data);
struct node *addafter(struct node *start,int data,int item);
struct node *addbefore(struct node *start,int data,int item );
```

```
struct node *del(struct node *start,int data);
struct node *reverse(struct node *start);

main()
{
        int choice,data,item;
        struct node *start=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.Add before\n");
                printf("8.Delete\n");
                printf("9.Reverse\n");
                printf("10.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        start=create_list(start);
                        break;
                 case 2:
                        display(start);
                        break;
                 case 3:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addtoempty(start,data);
                        break;
                 case 4:
                       printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addatbeg(start,data);
                        break;
                 case 5:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start=addatend(start,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);
                        start=addafter(start,data,item);
                        break;
                 case 7:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element before which to insert : ");
                        scanf("%d",&item);
                        start=addbefore(start,data,item);
                        break;
                 case 8:
                        printf("Enter the element to be deleted : ");
                        scanf("%d",&data);
                        start=del(start,data);
                        break;
                 case 9:
```

```
                start=reverse(start);
                break;
        case 10:
                exit(1);
        default:
                printf("Wrong choice\n");
    }/*End of switch*/
  }/*End of while*/
}/*End of main ()*/
```

3.2.1 Traversing a doubly linked List

The function for traversal of doubly linked list is similar to that of single linked list.

```
void display(struct node *start)
{
    struct node *p;
    if(start==NULL)
    {
        printf("List is empty\n");
        return;
    }
    p = start;
    printf("List is :\n");
    while(p!=NULL)
    {
        printf("%d ",p->info);
        p = p->next;
    }
    printf("\n");
}/*End of display()*/
```

3.2.2 Insertion in a doubly linked List

We will study all the four cases of insertion in a doubly linked list.

1. Insertion at the beginning of the list.
2. Insertion in an empty list.
3. Insertion at the end of the list.
4. Insertion in between the nodes

3.2.2.1 Insertion at the beginning of the list

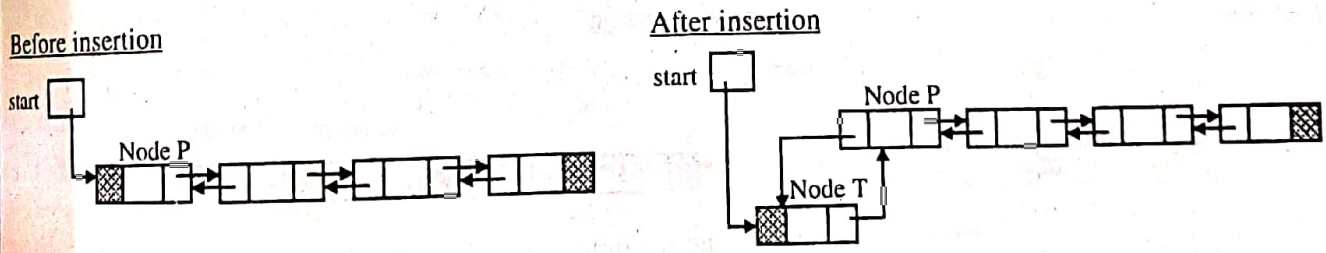

Figure 3.16 Insertion at the beginning of the list

Node T has become the first node so its `prev` should be `NULL`.

```
tmp->prev = NULL;
```

The `next` part of node T should point to node P, and address of node P is in `start` so we should write-

```
tmp->next = start;
```

Node T is inserted before node P so `prev` part of node P should now point to node T.

```
start->prev = tmp;
```

Now node T has become the first node so `start` should point to it.

```
start = tmp;
```

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

3.2.2.2 Insertion in an empty list

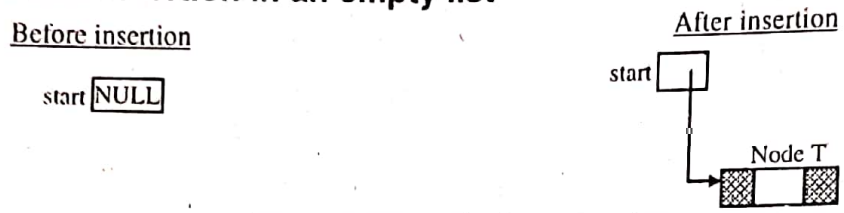

Figure 3.17 Insertion in an empty list

Node T is the first node so its `prev` part should be `NULL`, and it is also the last node so its `next` part should also be `NULL`. Node T is the first node so `start` should point to it.

```
tmp->prev = NULL;
tmp->next = NULL;
start = tmp;
```

In single linked list this case had reduced to the case of insertion at the beginning but here it is not so.

```
struct node *addtoempty(struct node *start,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        tmp->prev = NULL;
        tmp->next = NULL;
        start=tmp;
        return start;
}/*End of addtoempty()*/
```

3.2.2.3 Insertion at the end of the list

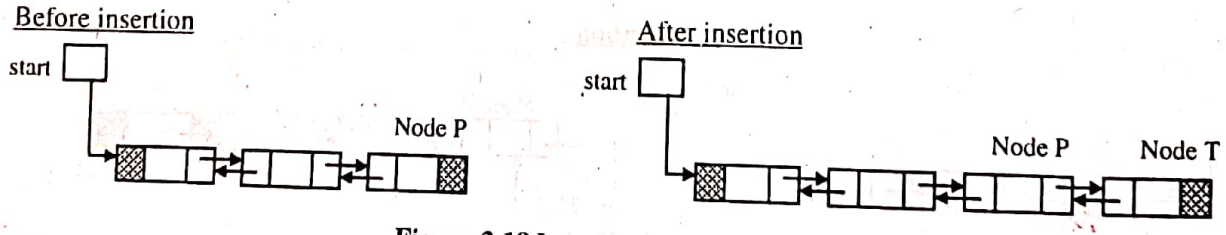

Figure 3.18 Insertion at the end of the list

Suppose p is a pointer pointing to the node P which is the last node of the list.

Node T becomes the last node so its `next` should be `NULL`

```
tmp->next = NULL;
```

`next` part of node P should point to node T

```
p->next = tmp;
```

`prev` part of node T should point to node P

```
tmp->prev = p;
```

```
struct node *addatend(struct node *start,int data)
{
```

```
    struct node *tmp,*p;
    tmp = (struct node *)malloc(sizeof(struct node));
    tmp->info = data;
    p = start;
    while(p->next!=NULL)
          p = p->next;
    p->next = tmp;
    tmp->next = NULL;
    tmp->prev = p;
    return start;
}/*End of addatend()*/
```

3.2.2.4 Insertion in between the nodes

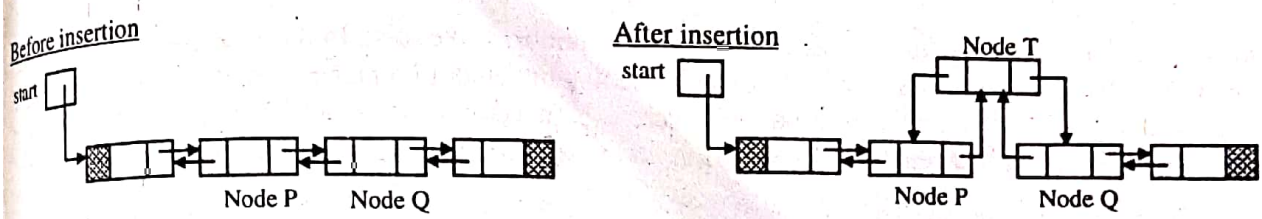

Figure 3.19 Insertion in between the nodes

Suppose pointers p and q point to nodes P and Q respectively.
Node P is before node T so prev of node T should point to node P

```
tmp->prev = p;
```

Node Q is after node T so next part of node T should point to node Q

```
tmp->next = q;
```

Node T is before node Q so prev part of node Q should point to node T

```
q->prev = tmp;
```

Node T is after node P so next part of node P should point to node T

```
p->next = tmp;
```

Now we will see how we can write the function addafter() for doubly linked list. We are given a value and the new node is to be inserted after the node containing this value.
Suppose node P contains this value so we have to add new node after node P. As in single linked list here also we can traverse the list and find a pointer p pointing to node P. Now in the four insertion statements we can replace q by p->next.

```
tmp->prev = p;      ->      tmp->prev = p;
tmp->next = q;      ->      tmp->next = p->next;
q->prev = tmp;      ->      p->next->prev = tmp;
p->next = tmp;      ->      p->next = tmp;
```

Note that p->next should be changed at the end because we are using it in previous statements.
In single linked list we had seen that the case of inserting after the last node was handled automatically. But here when we insert after the last node the third statement (p->next->prev = tmp;) will create problems. The pointer p points to last node so its next is NULL hence the term p->next->prev is meaningless here. To avoid this problem we can put a check like this-

```
if(p->next!=NULL)
      p->next->prev = tmp;
```

```
struct node *addafter(struct node *start,int data,int item)
{
      struct node *tmp,*p;
      tmp = (struct node *)malloc(sizeof(struct node));
      tmp->info = data;
      p = start;
      while(p!=NULL,
```

```
        {
                if(p->info == item)
                {
                        tmp->prev = p;
                        tmp->next = p->next;
                        if(p->next!=NULL)
                                p->next->prev = tmp;
                        p->next = tmp;
                        return start;
                }
                p = p->next;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addafter()*/
```

Now we will see how to write function `addbefore()` for doubly linked list. In this case suppose we have to insert the new node before node Q, so we will traverse the list and find a pointer `q` to node Q. In single linked list we had to find the pointer to predecessor but here there is no need to do so because we can get the address of predecessor by `q->prev`. So just replace `p` by `q->prev` in the four insertion statements.

```
tmp->prev = p;      ->      tmp->prev = q->prev;
tmp->next = q;      ->      tmp->next = q;
q->prev = tmp;      ->      q->prev = tmp;
p->next = tmp;      ->      q->prev->next = tmp;
```

`q->prev` should be changed at the end because it being used in other statements. Thus third statement should be the last one.

```
tmp->prev = q->prev;
tmp->next = q;
q->prev->next = tmp;
q->prev = tmp;
```

As in single linked list, here also we will have to handle the case of insertion before the first node separately.

```
struct node *addbefore(struct node *start,int data,int item)
{
        struct node *tmp,*q;
        if(start==NULL)
        {
                printf("List is empty\n");
                return start;
        }
        if(start->info == item)
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info = data;
                tmp->prev = NULL;
                tmp->next = start;
                start->prev = tmp;
                start = tmp;
                return start;
        }
        q = start;
        while(q!=NULL)
        {
                if(q->info == item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->prev = q->prev;
                        tmp->next = q;
                        q->prev->next = tmp;
                        q->prev = tmp;
```

```
                return start;
            }
            q = q->next;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addbefore()*/
```

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

3.2.4 Deletion from doubly linked list

As in single linked list, here also the node is first logically removed by rearranging the pointers and then it is physically removed by calling the function `free()`. Let us study the four cases of deletion-.

1. Deletion of first node.
2. Deletion of the only node.
3. Deletion in between the nodes.
4. Deletion at the end.

In all the cases we will take a pointer variable `tmp` which will point to the node being deleted.

3.2.4.1 Deletion of the first node

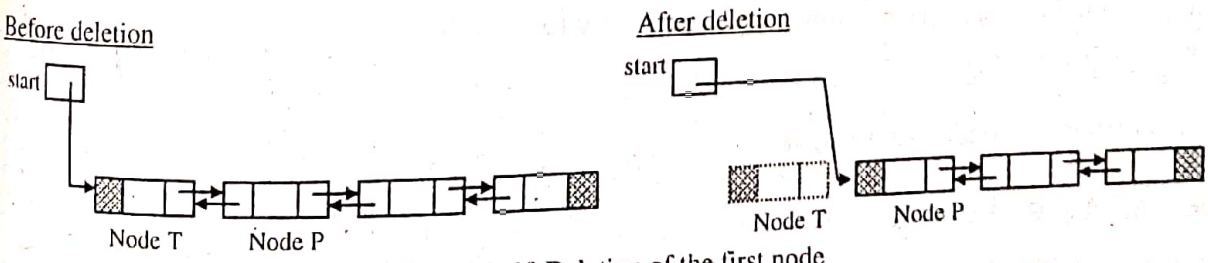

Figure 3.20 Deletion of the first node

`tmp` will be assigned the address of first node.

```
tmp = start;
```

`start` will be updated so that now it points to node P

```
start = start->next;
```

Now node P is the first node so its `prev` part should contain `NULL`.

```
start->prev = NULL
```

3.2.4.2 Deletion of the only node

Figure 3.21 Deletion of the only node

The two statements for deletion will be -

```
tmp = start;
start = NULL;
```

In single linked list we had seen that this case reduced to the previous one. Let us see what happens here. We can write `start->next` instead of `NULL` in the second statement, but then also this case does not reduce to the previous one. This is because of the third statement in the previous case, since start becomes `NULL`, the term `start->prev` is meaningless.

3.2.4.3 Deletion in between the nodes

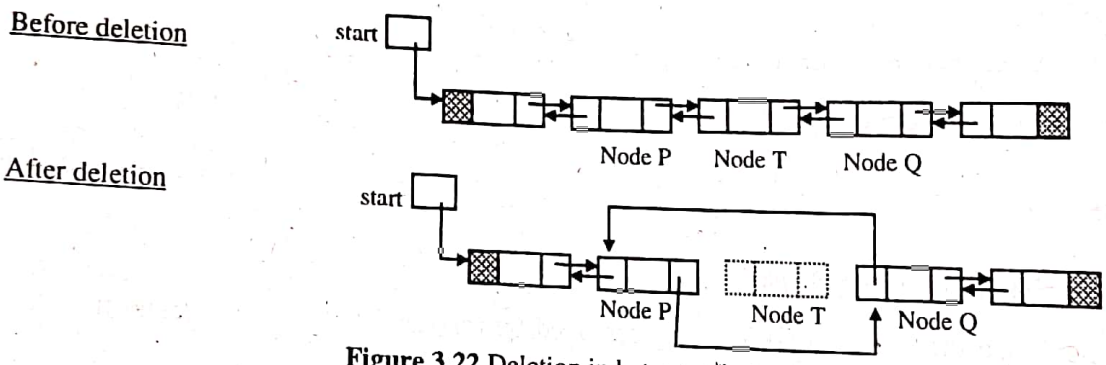

Figure 3.22 Deletion in between the nodes

Suppose we have to delete node T, and let pointers `p`, `tmp` and `q` point to nodes P, T and Q respectively. The two statements for deleting node T can be written as-

```
p->next = q;
q->prev = p;
```

The address of `q` is in `tmp->next` so we can replace `q` by `tmp->next`.

```
p->next = tmp->next;
tmp->next->prev = p;
```

The address of `p` is stored in `tmp->prev` so we can replace `p` by `tmp->prev`.

```
tmp->prev->next = tmp->next;
tmp->next->prev = tmp->prev;
```

So we need only pointer to a node for deleting it.

3.2.4.4 Deletion at the end of the list

Suppose node T is to be deleted and pointers `tmp` and `p` point to nodes T and P respectively. The deletion can be performed by writing the following statement.

```
p->next = NULL;
```

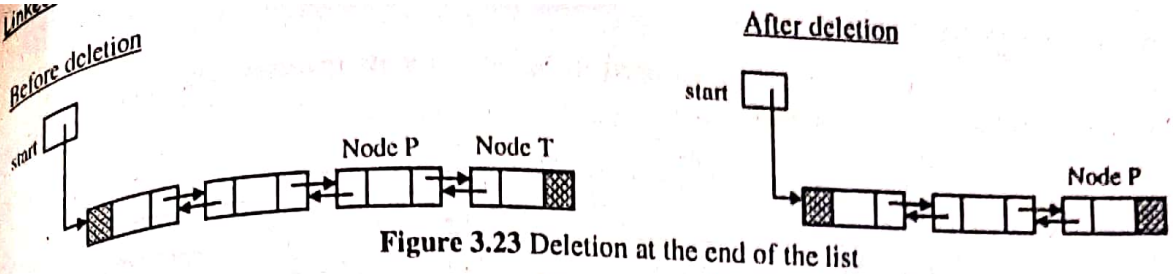

Figure 3.23 Deletion at the end of the list

The address of node P is stored in tmp->prev, so we can replace p by tmp->prev.

```
tmp->prev->next = NULL;
```

In single linked list, this case reduced to the previous case but here it won't.

```
struct node *del(struct node *start,int data)
{
    struct node *tmp;
    if(start == NULL)
    {
        printf("List is empty\n");
        return start;
    }
    if(start->next == NULL) /*Deletion of only node*/
        if(start->info == data)
        {
            tmp = start;
            start = NULL;
            free(tmp);
            return start;
        }
        else
        {
            printf("Element %d not found\n",data);
            return start;
        }
    if(start->info == data) /*Deletion of first node*/
    {
        tmp = start;
        start = start->next;
        start->prev = NULL;
        free(tmp);
        return start;
    }
    tmp = start->next; /*Deletion in between*/
    while(tmp->next!=NULL )
    {
        if(tmp->info == data)
        {
            tmp->prev->next = tmp->next;
            tmp->next->prev = tmp->prev;
            free(tmp);
            return start;
        }
        tmp = tmp->next;
    }
    if(tmp->info == data) /*Deletion of last node*/
    {
        tmp->prev->next = NULL;
        free(tmp);
        return start;
    }
    printf("Element %d not found\n",data);
    return start;
}/*End of del()*/
```

3.2.5 Reversing a doubly linked list

Let us take a doubly linked list and see what changes need to be done for its reversal. The following figure shows a double linked list and the reversed linked list.

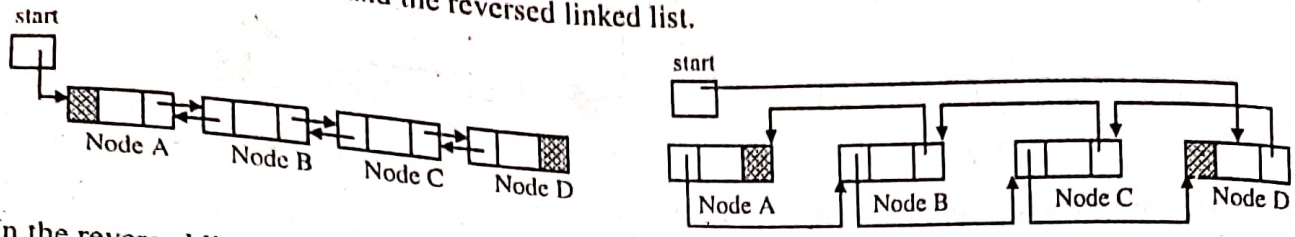

Figure 3.24

In the reversed list-
(i) start points to Node D.
(ii) Node D is the first node so its prev is NULL.
(iii) Node A is the last node so its next is NULL.
(iv) next of D points to C, next of C points to B and next of B points to A.
(v) prev of A points to B, prev of B points to C, prev of C points to D.

For making the function of reversal of doubly linked list we will need only two pointers.

```
struct node *reverse(struct node *start)
{
        struct node *p1,*p2;
        p1 = start;
        p2 = p1->next;
        p1->next = NULL;
        p1->prev=p2;
        while(p2!=NULL)
        {
                p2->prev = p2->next;
                p2->next = p1;
                p1 = p2;
                p2 = p2->prev;
        }
        start = p1;
        printf("List reversed\n");
        return start;
}/*End of reverse()*/
```

In a doubly linked list we have an extra pointer which consumes extra space, and maintenance of this pointer makes operations lengthy and time consuming. So doubly linked lists are beneficial only when we frequently need the predecessor of a node.