```
                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

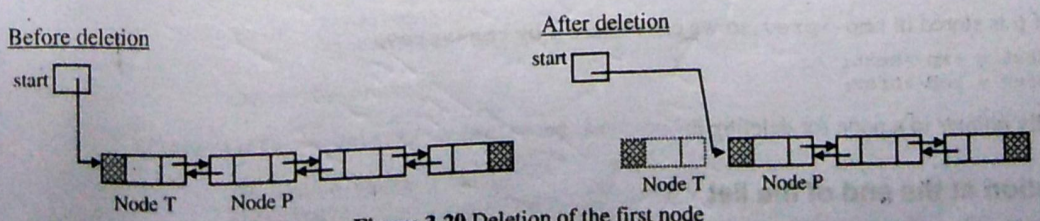


Figure 3.20 Deletion of the first node

`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

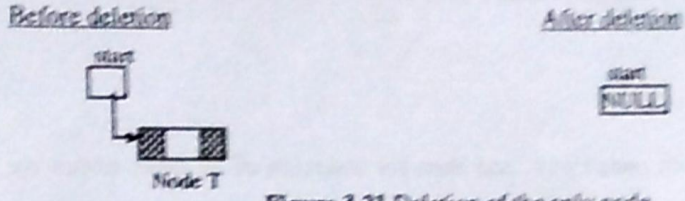


Figure 3.21 Deletion of the only node

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

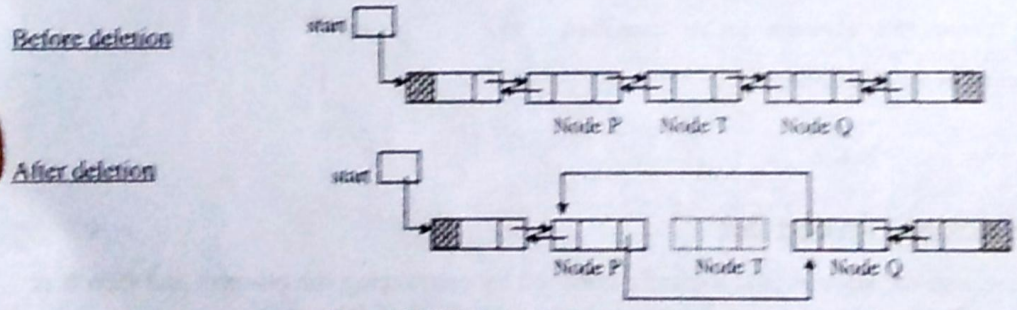


Figure 3.22 Deletion in between the nodes

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;
```

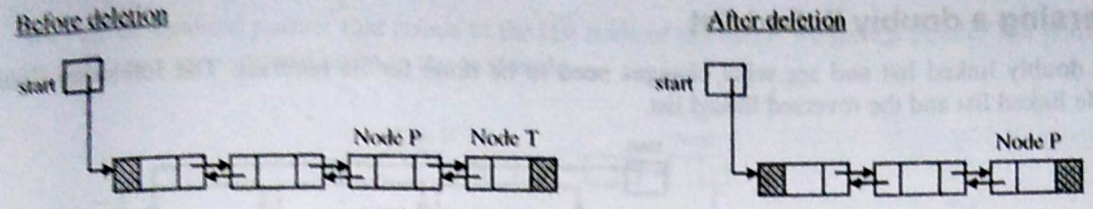


Figure 3.23 Deletion at the end of the list

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.

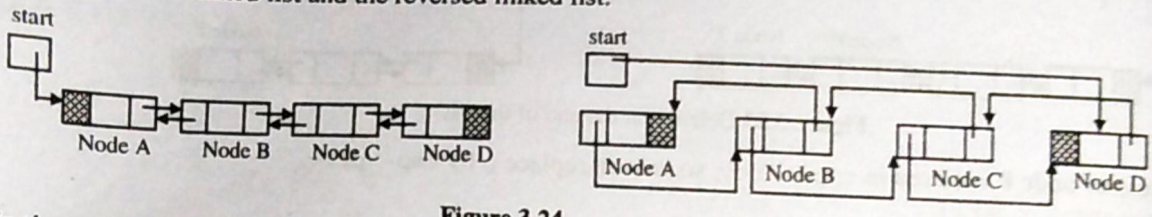


Figure 3.24

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.

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.

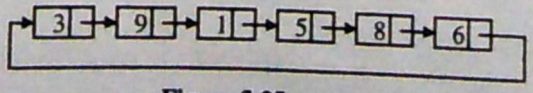


Figure 3.25

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.

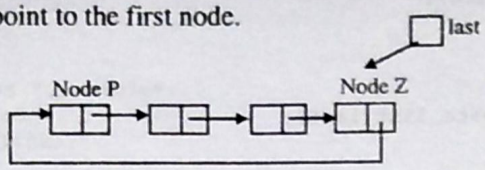


Figure 3.26

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.

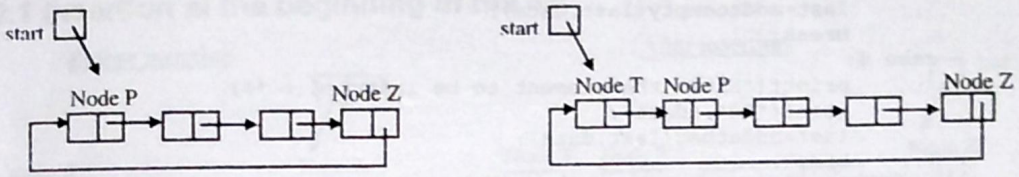


Figure 3.27

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

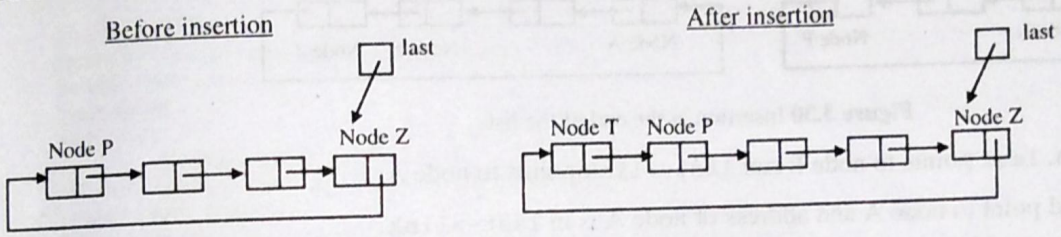


Figure 3.27 Insertion at the beginning of the list

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

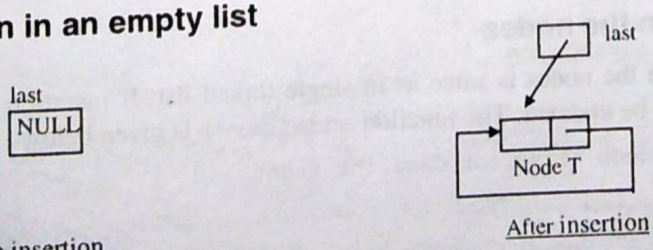


Figure 3.29 Insertion in an empty list

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

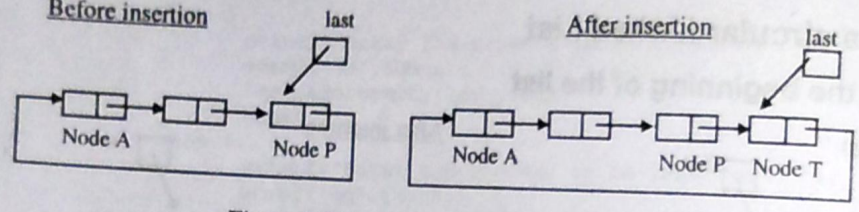


Figure 3.30 Insertion at the end of the list

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;
```

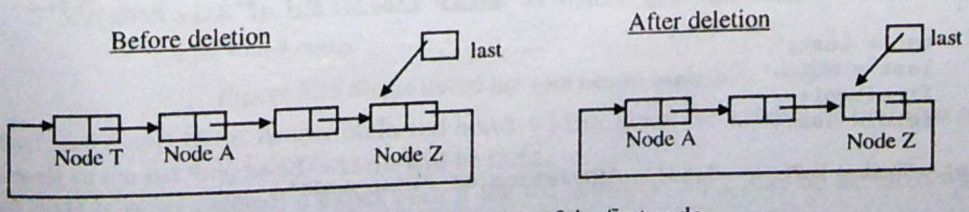


Figure 3.31 Deletion of the first node

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.

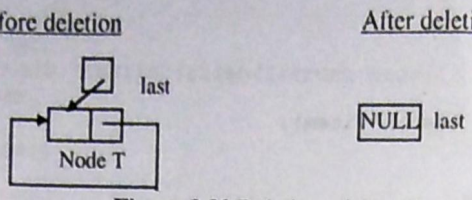


Figure 3.32 Deletion of the only node

There will be only one node in the list if link of last node points to itself. After deletion the list w 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

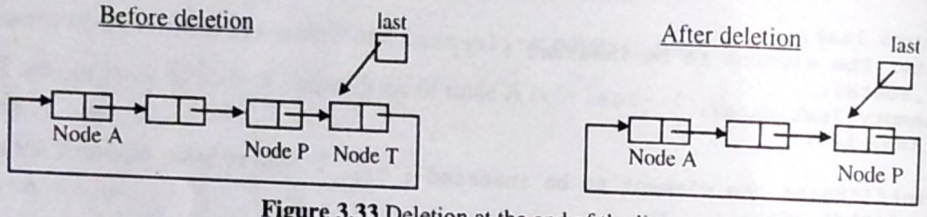


Figure 3.33 Deletion at the end of the list

Before deletion, `last` points to node T and `last->link` points to node A, `p` is a pointer to node 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.

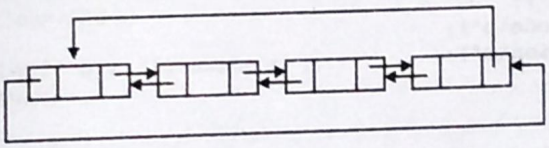


Figure 3.34 Circular double linked list

3.4 Linked List with Header Node

Header node is a dummy node that is present at the beginning of the list and its link part is used to store the address of the first actual node of the list. The info part of this node may be empty or can be used to contain useful information about the list like count of elements currently present in the list. The figure 3.35(a) shows a single linked list with 4 actual nodes and a header node, and the figure (b) shows an empty list with a header node.

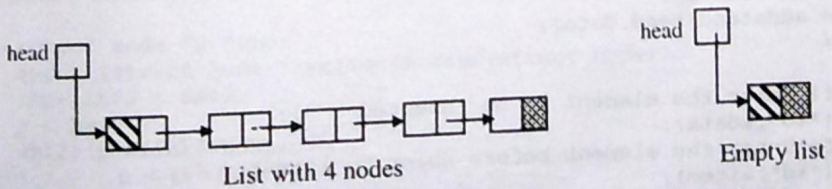


Figure 3.35 Single linked list with header node

The pointer `head` points to the header node and `head->link` gives the address of first true node of the list. If there is no node in the list then `head->link` will be `NULL`.

The header node is never deleted; it exists even if the list is empty. So it may be declared while writing the program instead of dynamically allocating memory for it.

The use of header nodes makes the program simple and faster. Since the list is never empty because header node is always there, we can avoid some special cases of empty list and that of insertion and deletion in the beginning. For example in the function `del()` and `addbefore()` that we had written for single linked list, we can drop the first two cases if we take a header node in our list.

The following program performs operations on a single linked list with header node. The logic of all operations is similar to that of single linked list without header but some cases have been removed.

```
/*P3.4 Program of single linked list with header node*/
#include<stdio.h>
#include<stdlib.h>
```

```
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *head);
void display(struct node *head);
struct node *addatend(struct node *head,int data);
struct node *addbefore(struct node *head,int data,int item );
struct node *addatpos(struct node *head,int data,int pos);
struct node *del(struct node *head,int data);
struct node *reverse(struct node *head);

main()
{
        int choice,data,item,pos;
        struct node *head;
        head = (struct node *)malloc(sizeof(struct node));
        head->info = 0;
        head->link = NULL;
        head = create_list(head);
        while(1)
        {
                printf("1.Display\n");
                printf("2.Add at end\n");
                printf("3.Add before node\n");
                printf("4.Add at position\n");
                printf("5.Delete\n");
                printf("6.Reverse\n");
                printf("7.Quit\n\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        display(head);
                        break;
                 case 2:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        head = addatend(head,data);
                        break;
                 case 3:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the element before which to insert : ");
                        scanf("%d",&item);
                        head = addbefore(head,data,item);
                        break;
                 case 4:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        printf("Enter the position at which to insert : ");
                        scanf("%d",&pos);
                        head = addatpos(head,data,pos);
                        break;
                case 5:
                        printf("Enter the element to be deleted : ");
                        scanf("%d",&data);
                        head = del(head,data);
                        break;
                case 6:
                        head = reverse(head);
                        break;
```

```
                case 7:
                        exit(1);
                default:
                        printf("Wrong choice\n\n");
                }/*End of switch */
        }/*End of while */
}/*End of main()*/
struct node *create_list(struct node *head)
{
        int i,n,data;
        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                head = addatend(head,data);
        }
        return head;
}/*End of create_list()*/
void display(struct node *head)
{
        struct node *p;
        if(head->link==NULL)
        {
                printf("List is empty\n");
                return;
        }
        p = head->link;
        printf("List is :\n");
        while(p!=NULL)
        {
                printf("%d ",p->info);
                p=p->link;
        }
        printf("\n");
}/*End of display() */
struct node *addatend(struct node *head,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        p = head;
        while(p->link!=NULL)
                p = p->link;
        p->link = tmp;
        tmp->link = NULL;
        return head;
}/*End of addatend()*/
struct node *addbefore(struct node *head,int data,int item)
{
        struct node *tmp,*p;
        p = head;
        while(p->link!=NULL)
        {
                if(p->link->info==item)
                {
                        tmp = (struct node *)malloc(sizeof(struct node));
                        tmp->info = data;
                        tmp->link = p->link;
                        p->link = tmp;
                        return head;
```

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

struct node *addatpos(struct node *head,int data,int pos)
{
        struct node *tmp,*p;
        int i;
        tmp = (struct node *)malloc(sizeof(struct node) );
        tmp->info = data;
        p = head;
        for(i=1;i<=pos-1;i++)
        {
                p = p->link;
                if(p==NULL)
                {
                        printf("There are less than %d elements\n",pos);
                        return head;
                }
        }
        tmp->link = p->link;
        p->link = tmp;
        return head;
}/*End of addatpos()*/

struct node *del(struct node *head, int data)
{
        struct node *tmp,*p;
        p = head;
        while(p->link!=NULL)
        {
                if(p->link->info==data)
                {
                        tmp = p->link;
                        p->link = tmp->link;
                        free(tmp);
                        return head;
                }
                p = p->link;
        }
        printf("Element %d not found\n",data);
        return head;
}/*End of del()*/

struct node *reverse(struct node *head)
{
        struct node *prev, *ptr, *next;
        prev = NULL;
        ptr = head->link;
        while(ptr!=NULL)
        {
                next = ptr->link;
                ptr->link = prev;
                prev = ptr;
                ptr = next;
        }
        head->link = prev;
        return head;
}
```

The header node can be attached to circular linked lists and doubly linked lists also. The following fig shows circular single linked list with header node.

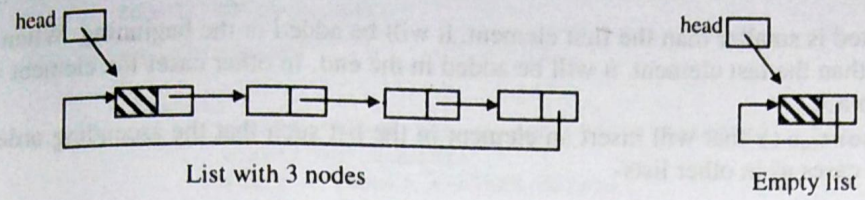
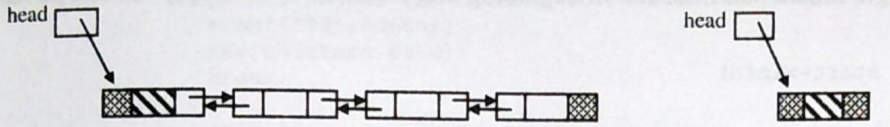


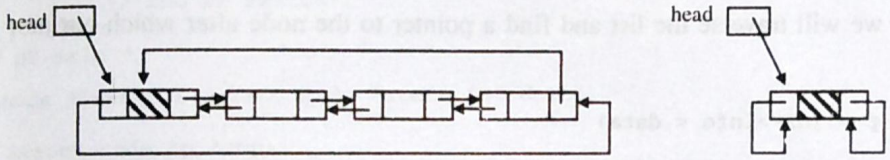
Figure 3.36 Circular single linked list with header

Here the external pointer points to the header node rather than at the end.

The following figures show doubly linked list and doubly linked circular list with header nodes.



(a) Doubly linked list with header



(b) Doubly linked circular list with header

Figure 3.37

3.5 Sorted linked list

In some applications, it is better if the elements in the list are kept in sorted order. For maintaining a list in sorted order we have to insert the nodes in proper place. Let us take an ascending order linked list and insert some elements in it.

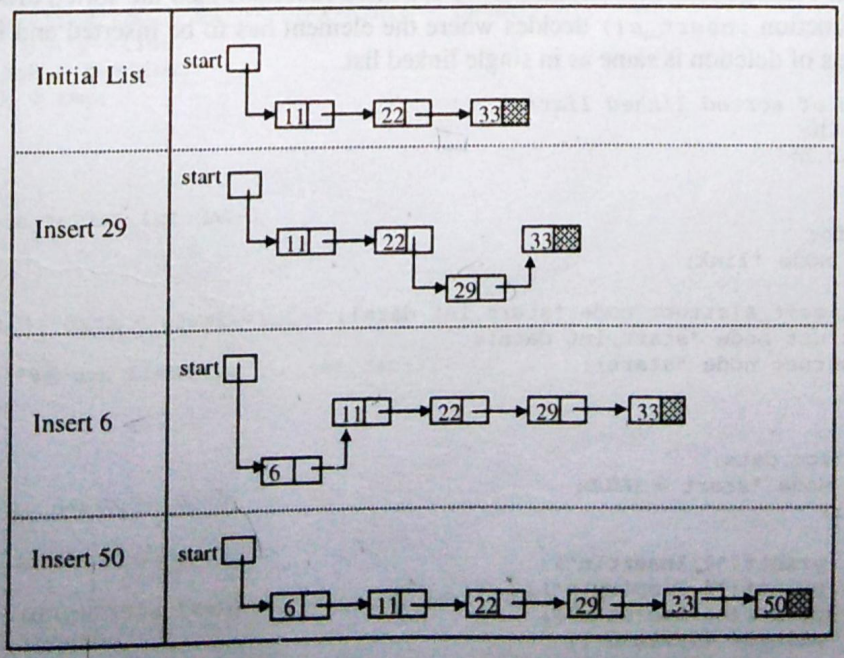


Figure 3.38 Insertion in a sorted linked list

When the element to be inserted is smaller than the first element, it will be added in the beginning. When the element to be inserted is greater than the last element, it will be added in the end. In other cases the element will be added in the list at its proper place.

We will make a function `insert_s()` that will insert an element in the list such that the ascending order is maintained. We have the same 4 cases as in other lists-

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

Since we have taken a single linked list, insertion in beginning and insertion in an empty list can be handled in the same way.

```
if(start==NULL || data < start->info)
{
        tmp->link = start;
        start = tmp;
        return start;
}
```

For insertion in between, we will traverse the list and find a pointer to the node after which our new node should be inserted.

```
p = start;
while(p->link != NULL && p->link->info < data)
        p = p->link;
```

The new node has to be inserted after the node which is pointed by pointer `p`. The two lines of insertion are same as in single linked list.

```
tmp->link = p->link;
p->link = tmp;
```

If the insertion is to be done in the end, then also the above statements will work.

Other functions like `display()`, `count()` etc will remain same. The functions `search()` will be altered a little because here we can stop our search as soon as we find an element with value larger than the given element to be searched. The functions like `addatbeg()`, `addatend()`, `addafter()`, `addbefore()`, `addatpos()` don't make sense here because if we use these functions then the sorted order of the list might get disturbed. The function `insert_s()` decides where the element has to be inserted and inserts it in the proper place. The process of deletion is same as in single linked list.

```
/*P3.5 Program of sorted linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *insert_s(struct node *start,int data);
void search(struct node *start,int data);
void display(struct node *start);

main()
{
        int choice,data;
        struct node *start = NULL;
        while(1)
        {
                printf("1.Insert\n");
                printf("2.Display\n");
                printf("3.Search\n");
                printf("4.Exit\n");
                printf("Enter your choice : ");
```

```
            scanf("%d",&choice);
            switch(choice)
            {
             case 1:
                    printf("Enter the element to be inserted : ");
                    scanf("%d",&data);
                    start = insert_s(start,data);
                    break;
             case 2:
                    display(start);
                    break;
             case 3:
                    printf("Enter the element to be searched : ");
                    scanf("%d",&data);
                    search(start,data);
                    break;
             case 4:
                    exit(1);
             default:
                    printf("Wrong choice\n");
            }/*End of switch*/
      }/*End of while*/
} /*end of main */

struct node *insert_s(struct node *start,int data)
{
      struct node *p,*tmp;
      tmp = (struct node *)malloc(sizeof(struct node));
      tmp->info = data;
      /*list empty or new node to be added before first node*/
      if(start==NULL || data<start->info)
      {
            tmp->link = start;
            start = tmp;
            return start;
      }
      else
      {
            p = start;
            while(p->link!=NULL && p->link->info < data)
                  p = p->link;
            tmp->link = p->link;
            p->link = tmp;
      }
      return start;
}/*End of insert()*/

void search(struct node *start,int data)
{
      struct node *p;
      int pos;
      if(start==NULL || data < start->info)
      {
            printf("%d not found in list\n",data);
            return;
      }
      p = start;
      pos = 1;
      while(p!=NULL && p->info<=data)
      {
            if(p->info == data)
            {
                  printf("%d found at position %d\n",data,pos);
                  return;
            }
```

```
                p = p->link;
                pos++;
        }
        printf("%d not found in list\n",data);
}/*End of search()*/
void display(struct node *start)
{
        struct node *q;
        if(start == NULL)
        {
                printf("List is empty\n");
                return;
        }
        q = start;
        printf("List is :\n");
        while(q!=NULL)
        {
                printf("%d ",q->info);
                q = q->link;
        }
        printf("\n");
}/*End of display() */
```

3.6 Sorting a Linked List

If we have a linked list in unsorted order and we want to sort it, we can apply any sorting algorithm. We will use selection sort and bubble sort techniques (procedure given in chapter 8). In that chapter the elements to be sorted are stored in an array and here elements to be sorted are stored in a linked list. So here we can carry out the sorting in two ways-

(1) By exchanging the data
(2) By rearranging the links

Sorting by exchanging the data is similar to sorting carried out in arrays. If we have large records then this method is inefficient since the movement of records will take more time. In linked list, we can perform the sorting by one more method i.e. by rearranging the links. In this case there will be no movement of data, only the links will be changed.

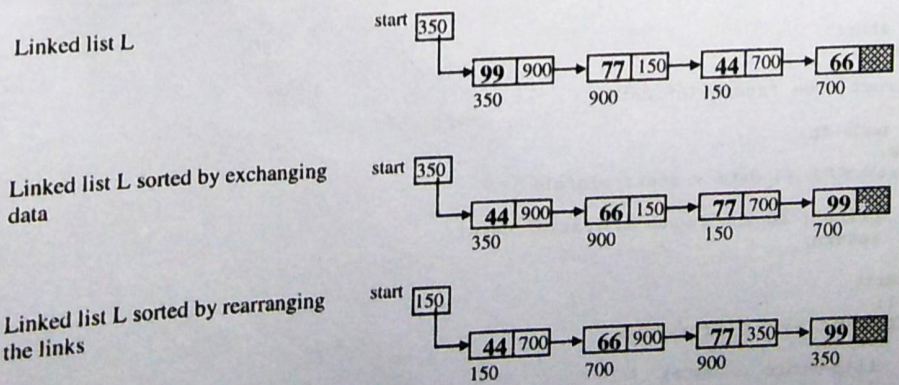


Figure 3.39

3.6.1 Selection Sort by exchanging data

The procedure of sorting a linked list through selection sort is shown in the following figure.

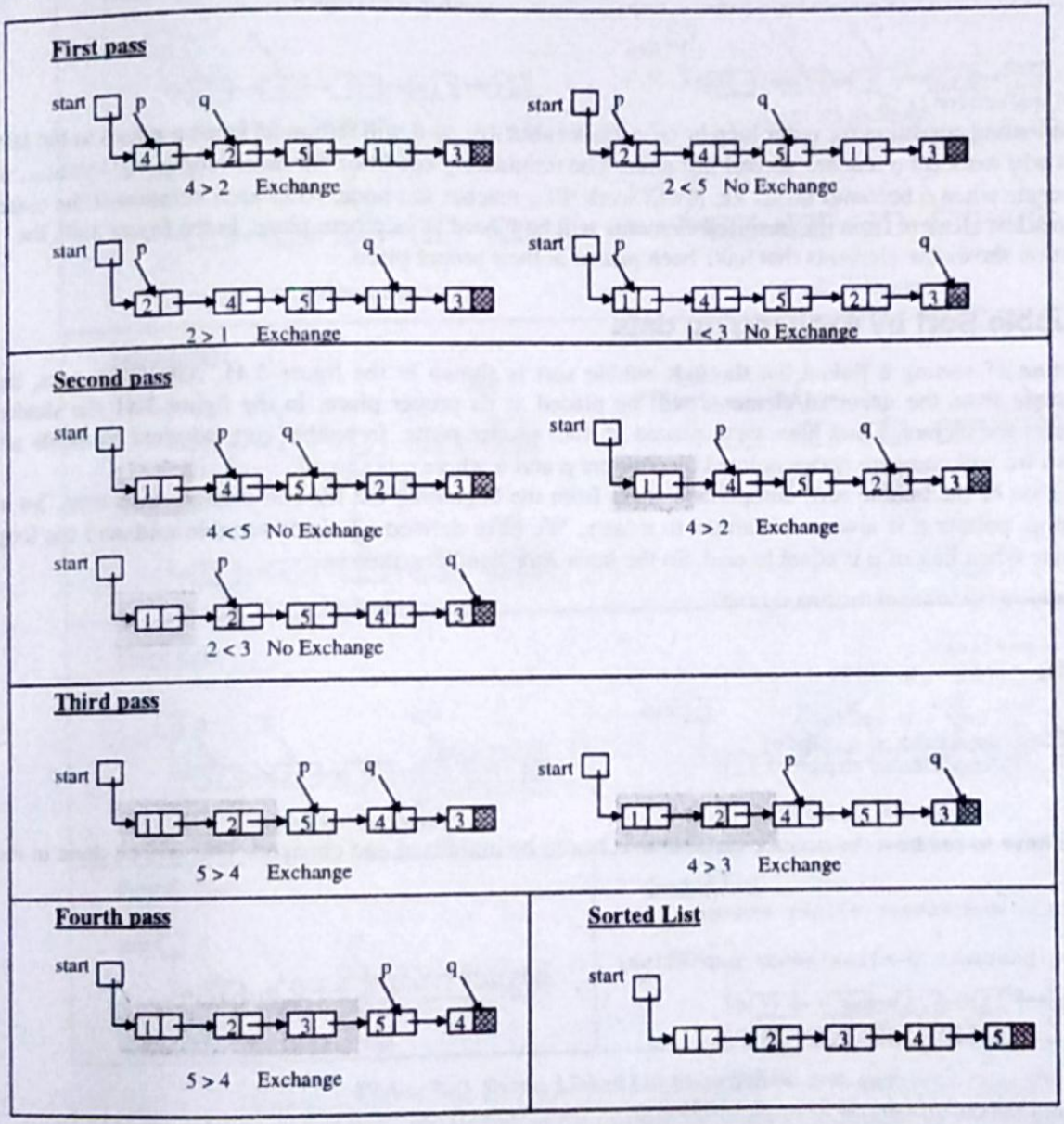


Figure 3.40 Sorting a Linked List using Selection sort

The following function will sort a single linked list through selection sort technique by exchanging data.

```
void selection(struct node *start)
{
        struct node *p,*q;
        int tmp;
        p = start;

        for(p=start; p->link!=NULL; p=p->link)
        {
                for(q=p->link; q!=NULL; q=q->link)
                {
```

```
                if(p->info > q->info)
                {
                        tmp = p->info;
                        p->info = q->info;
                        q->info = tmp;
                }
            }
        }
)/*End of selection()*/
```

The terminating condition for outer loop is `(p->link!=NULL)`, so it will terminate when p points to the last node, i.e. it will work till p reaches second last node. The terminating condition for inner loop is `(q!=NULL)`, so it will terminate when q becomes NULL, i.e. it will work till q reaches last node. After each iteration of the outer loop, the smallest element from the unsorted elements will be placed at its proper place. In the figure 3.40, the shaded portion shows the elements that have been placed at their proper place.

3.6.2 Bubble Sort by exchanging data

The procedure of sorting a linked list through bubble sort is shown in the figure 3.41. After each pass, the largest element from the unsorted elements will be placed at its proper place. In the figure 3.41 the shaded portion shows the elements that have been placed at their proper place. In bubble sort, adjacent elements are compared so we will compare nodes pointed by pointers p and q where q is equal to `p->link`.

In each pass of the bubble sort, comparison starts from the beginning but the end changes each time. So in the inner loop, pointer p is always initialized to `start`. We have defined a pointer variable `end`, and the loop will terminate when link of p is equal to end. So the inner loop can be written as-

```
for(p=start; p->link!=end; p=p->link)
{
        q = p->link;
        if(p->info > q->info )
        {
                tmp = p->info;
                p->info = q->info;
                q->info = tmp;
        }
}
```

Now we have to see how the pointer variable end has to be initialized and changed. This will be done in the outer loop.

```
for(end=NULL; end!=start->link; end=q)
{
        for(p=start; p->link!=end; p=p->link)
        {
                q = p->link;
                if(p->info > q->info)
                {
                        tmp = p->info;
                        p->info = q->info;
                        q->info = tmp;
                }
        }
}
```

The pointer variable end is NULL in the first iteration of outer loop, so inner loop will terminate when p points to the last node i.e. the inner loop will work only till p reaches second last node. After first iteration value of end is updated and is made equal to q. So now end points to the last node. This time the inner loop will terminate when p points to the second last node i.e. the inner loop will work only till p reaches third last node.

After each iteration of outer loop, the pointer end moves one node back towards the beginning. Initially end is NULL, after first iteration it points to the last node, after second iteration it points to the second last node and