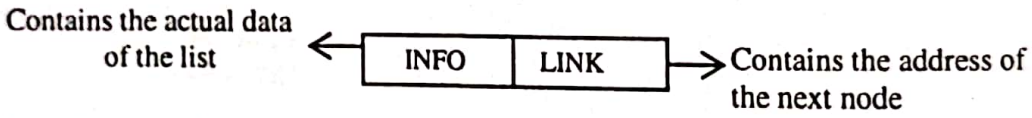
Linked Lists

3

List is a collection of similar type of elements. There are two ways of maintaining a list in memory. The first way is to store the elements of the list in an array, but arrays have some restrictions and disadvantages. The second way of maintaining a list in memory is through linked list. Now let us study what a linked list is and after that we will come to know how it overcomes the limitations of array.

3.1 Single Linked list

A single linked list is made up of nodes where each node has two parts, the first one is the info part that contains the actual data of the list and the second one is the link part that points to the next node of the list or we can say that it contains the address of the next node.



The beginning of the list is marked by a special pointer named `start`. This pointer points to the first node of the list. The link part of each node points to the next node in the list, but the link part of last node has no next node to point to, so it is made NULL. Hence if we reach a node whose link part has NULL value then we know that we are at the end of the list. Suppose we have a list of four integers 33, 44, 55, 66; let us see how we will represent it through linked list.

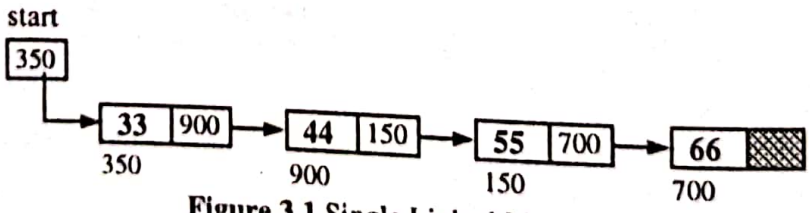


Figure 3.1 Single Linked List

From the figure it is clear that the info part contains the integer values and the link part contains the address of the next node. The address of the first node is contained in the pointer `start` and the link part of last node is NULL(represented by shaded part in the figure).

If we observe the memory addresses of the nodes, we find that the nodes are not necessarily located adjacent to each other in the memory. The following figure shows the position of nodes of the above list in memory.

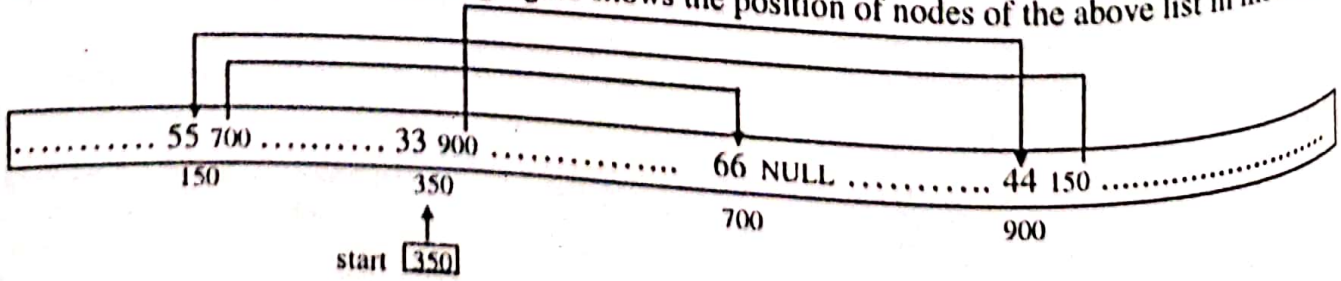


Figure 3.2

We can see that the nodes are scattered here and there in memory, but still they are connected to each other through the link part, which also maintains their linear order. Now let us see the picture of memory if the same list of integers is implemented through array.

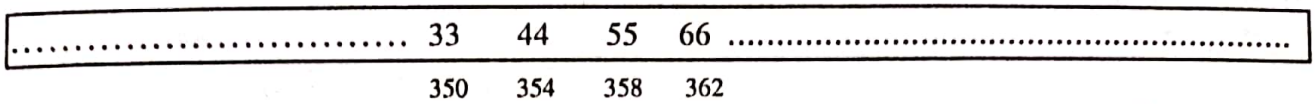


Figure 3.3

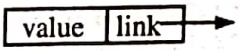
Here the elements are stored in consecutive memory locations in the same order as they appear in the list. So array is sequential representation of list while linked list is the linked representation of list. In a linked list, nodes are not stored contiguously as in array, but they are linked through pointers(links) and we can use these links to move through the list.

Now let us see how we can represent a node of a single linked list in C language. We will take a self referential structure for this purpose. Recall that a self referential structure is a structure which contains a pointer to a structure of the same type. The general form of a node of linked list is-

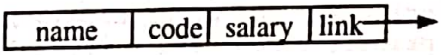
```
struct node{
            type1 member1;
            type2 member2;
            .....
            .....
            struct node *link;  /*Pointer to next node*/
        };
```

Let us see some examples of nodes-

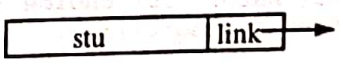
```
struct node{
            int value;
            struct node *link;
        };
```




```
struct node{
            char name[10];
            int code;
            float salary;
            struct node *link;
        };
```




```
struct node{
            struct student stu;
            struct node *link;
        };
```



In this chapter we will perform all operations on linked lists that contain only an integer value in the info part of their nodes.

In array we could perform all the operations using the array name and index. In the case of linked list, we will perform all the operations with the help of the pointer `start` because it is the only source through which we can access our linked list. The list will be considered empty if the pointer `start` contains NULL value. So our first job is to declare the pointer `start` and initialize it to NULL. This can be done as-

```
struct node *start;
start = NULL;
```

Now we will discuss the following operations on a single linked list.

(i) Traversal of a linked list
(ii) Searching an element
(iii) Insertion of an element
(iv) Deletion of an element
(v) Creation of a linked list
(vi) Reversal of a linked list

The main() function and declarations are given here, and the code of other functions are given with the explanation.

```
/*P3.1 Program of single linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *start);
void display(struct node *start);
void count(struct node *start);
void search(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 *addatpos(struct node *start,int data,int pos);
struct node *del(struct node *start,int data);
struct node *reverse(struct node *start);

main()
{
        struct node *start=NULL;
        int choice,data,item,pos;
        while(1)
        {
                printf("1.Create List\n");
                printf("2.Display\n");
                printf("3.Count\n");
                printf("4.Search\n");
                printf("5.Add to empty list / Add at beginning\n");
                printf("6.Add at end\n");
                printf("7.Add after node\n");
                printf("8.Add before node\n");
                printf("9.Add at position\n");
                printf("10.Delete\n");
                printf("11.Reverse\n");
                printf("12.Quit\n\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                 case 1:
                        start = create_list(start);
                        break;
                 case 2:
                        display(start);
                        break;
                 case 3:
                        count(start);
                        break;
                 case 4:
                        printf("Enter the element to be searched : ");
                        scanf("%d",&data);
                        search(start,data);
                        break;
                 case 5:
                        printf("Enter the element to be inserted : ");
                        scanf("%d",&data);
                        start = addatbeg(start,data);
                        break;
```

```
            case 6:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  start = addatend(start,data);
                  break;
            case 7:
                  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 8:
                  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 9:
                  printf("Enter the element to be inserted : ");
                  scanf("%d",&data);
                  printf("Enter the position at which to insert : ");
                  scanf("%d",&pos);
                  start = addatpos(start,data,pos);
                  break;
            case 10:
                  printf("Enter the element to be deleted : ");
                  scanf("%d",&data);
                  start = del(start, data);
                  break;
            case 11:
                  start = reverse(start);
                  break;
            case 12:
                  exit(1);
            default:
                  printf("Wrong choice\n");
          }/*End of switch*/
      }/*End of while*/
}/*End of main()*/
```

In the function `main()`, we have taken an infinite loop and inside the loop we have written a switch statement. In the different cases of this switch statement, we have implemented different operations of linked list. To come out of the infinite loop and exit the program we have used the function `exit()`.

The structure pointer `start` is declared in `main()` and initialized to `NULL`. We send this pointer to all functions because `start` is the only way of accessing linked list. Functions like those of insertion, deletion, reversal will change our linked list and value of `start` might change so these functions return the value of `start`. The functions like `display()`, `count()`, `search()` do not change the linked list so their return type is `void`.

3.1.1 Traversing a Single Linked List

Traversal means visiting each node, starting from the first node till we reach the last node. For this we will take a structure pointer `p` which will point to the node that is currently being visited. Initially we have to visit the first node so `p` is assigned the value of `start`.

```
p = start;
```

Now `p` points to the first node of linked list. We can access the info part of first node by writing `p->info`. Now we have to shift the pointer `p` forward so that it points to the next node. This can be done by assigning the address of the next node to `p` as-

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

Now p has address of the next node. Similarly we can visit each node of linked list through this assignment until p has NULL value, which is link part value of last element. So the linked list can be traversed as-

```
while(p!=NULL)
{
        printf("%d  ",p->info);
        p = p->link;
}
```

Let us take an example to understand how the assignment `p = p->link` makes the pointer p move forward. From now onwards we will not show the addresses, we will show only the info part of the list in the figures.

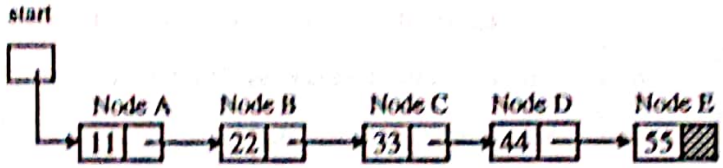


Figure 3.4

In figure 3.4, node A is the first node so start points to it, node E is the last node so its link is NULL.
Initially p points to node A, `p->info` gives 11 and `p->link` points to node B
After the statement `p = p->link;`
p points to node B, `p->info` gives 22 and `p->link` points to node C
After the statement `p = p->link;`
p points to node C, `p->info` gives 33 and `p->link` points to node D
After the statement `p = p->link;`
p points to node D, `p->info` gives 44 and `p->link` points to node E
After the statement `p = p->link;`
p points to node E, `p->info` gives 55 and `p->link` is NULL
After the statement `p = p->link;`
p becomes NULL, i.e. we have reached the end of the list so we come out of the loop.

The following function `display()` displays the contents of the 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->link;
        }
        printf("\n\n");
}/*End of display() */
```

Don't think of using start for moving forward. If we use `start = start->link`, instead of `p = p->link` then we will lose `start` and that is the only means of accessing our list. The following function `count()` finds out the number of elements of the linked list.

```
void count(struct node *start)
{
        struct node *p;
```

```
        int cnt = 0;
        p = start;
        while(p!=NULL)
        {
                p = p->link;
                cnt++;
        }
        printf("Number of elements are %d\n",cnt);
}/*End of count() */
```

3.1.2 Searching in a Single Linked List

For searching an element, we traverse the linked list and while traversing we compare the info part of each element with the given element to be searched. In the function given below, `item` is the element which we want to search.

```
void search(struct node *start,int item)
{
        struct node *p = start;
        int pos = 1;
        while(p!=NULL)
        {
                if(p->info == item)
                {
                        printf("Item %d found at position %d\n",item,pos);
                        return;
                }
                p = p->link;
                pos++;
        }
        printf("Item %d not found in list\n",item);
}/*End of search()*/
```

3.1.3 Insertion in a Single Linked List

There can be four cases while inserting a node in a linked list.

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

To insert a node, initially we will dynamically allocate space for that node using `malloc()`. Suppose `tmp` is a pointer that points to this dynamically allocated node. In the info part of the node we will put the data value.

```
tmp = (struct node *)malloc(sizeof(struct node));
tmp->info = data;
```

The link part of the node contains garbage value; we will assign address to it separately in the different cases. In our explanation we will refer to this new node as node T.

3.1.3.1 Insertion at the beginning of the list

We have to insert node T at the beginning of the list. Suppose the first node of list is node P, so the new node T should be inserted before it.

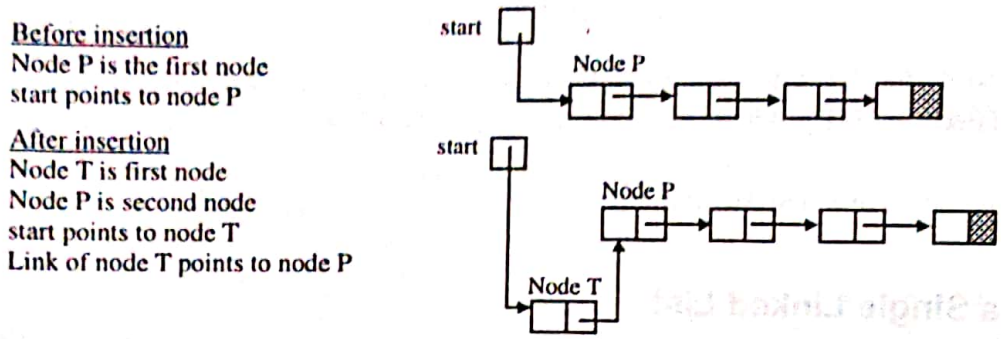


Figure 3.5 Insertion at the beginning of the list

(i) Link of node T should contain the address of node P, and we know that `start` has address of node P so we should write-

```
tmp->link = start;
```

After this statement, link of node T will point to node P.

(ii) We want to make node T the first node; hence we should update `start` so that now it points to node T.

```
start = tmp;
```

The order of the above two statements is important. First we should make link of T equal to `start` and after that only we should update `start`. Let's see what happens if the order of these two statements is reversed.

```
start = tmp;
```

`start` points to `tmp` and we lost the address of node P.

```
tmp->link = start;
```

Link of `tmp` will point to itself because `start` has address of `tmp`. So if we reverse the order, then link of node T will point to itself and we will be stuck in an infinite loop when the list is processed.

The following function `addatbeg()` adds a node at the beginning of the list.

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

3.1.3.2 Insertion in an empty list

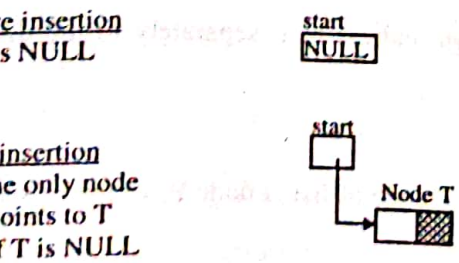


Figure 3.6 Insertion in an empty list

When the list is empty, value of `start` will be `NULL`. The new node that we are adding will be the only node in the list. Since it is the first node, `start` should point to this node and it is also the last node so its link should be `NULL`.

```
tmp->link = NULL;
start = tmp;
```

Since initially `start` was `NULL`, we can write `start` instead of `NULL` in the first statement, so now these two statements can be written as-

```
tmp->link = start;
start = tmp;
```

These two statements are same as in the previous case (3.1.3.1), so we can see that this case reduces to the previous case of insertion in the beginning and the same code can be written for both the cases.

3.1.3.3 Insertion at the end of the list

We have to insert a new node T at the end of the list. Suppose the last node of list is node P, so node T should be inserted after node P.

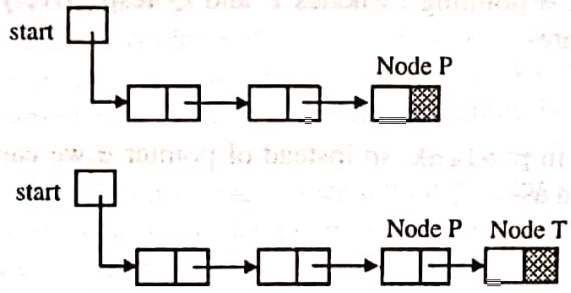


Figure 3.7 Insertion at the end of the list

Suppose we have a pointer `p` pointing to the node P. These are the two statements that should be written for this insertion-

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

So in this case we should have a pointer `p` pointing to the last node of the list. The only information about the linked list that we have is the pointer `start`. So we will traverse the list till the end to get the pointer `p` and then do the insertion. This is how we can obtain the pointer `p`.

```
p = start;
while(p->link!=NULL)
        p = p->link;
```

In traversal of list(3.1.1) our terminating condition was `(p!=NULL)`, because there we wanted the loop to terminate when `p` becomes `NULL`. Here we want the loop to terminate when `p` is pointing to the last node so the terminating condition is `(p->link!=NULL)`.

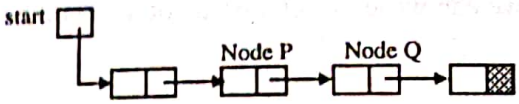
The following function `addatend()` inserts a node at the end of the list.

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

3.1.3.4 Insertion in between the list nodes

We have to insert a node T between nodes P and Q.

Before insertion
Node Q is after node P
Link of P points to node Q



After insertion
Node T is between nodes P and Q
Link of node T points to node Q
Link of node P points to node T

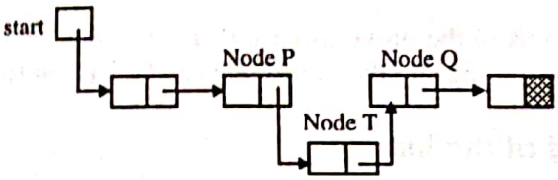


Figure 3.8 Insertion in between the list nodes

Suppose we have two pointers p and q pointing to nodes P and Q respectively. The two statements that should be written for insertion of node T are-

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

Before insertion address of node Q is in p->link, so instead of pointer q we can write p->link. Now the two statements for insertion can be written as-

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

Note : The order of these two statements is important, if you write them in the reverse order then you will lose your links. The address of node Q is in p->link, suppose we write the statement (p->link = tmp;) first then we will lose the address of node Q, there is no way to reach node Q and our list is broken. So first we should assign the address of node Q to the link of node T by writing (tmp->link = p->link). Now we have stored the address of node Q, so we are free to change p->link.

Now we will see three cases of insertion in between the nodes-
1. Insertion after a node
2. Insertion before a node
3. Insertion at a given position

The two statements of insertion(tmp->link = p->link; p->link = tmp;) will be written in all the three cases, but the way of finding the pointer p will be different.

3.1.3.4.1 Insertion after a node

In this case we are given a value from the list, and we have to insert the new node after the node that contains this value. Suppose the node P contains the given value, and node Q is its successor. We have to insert the new node T after node P, i.e. T is to be inserted between nodes P and Q. In the function given below, data is the new value to be inserted and item is the value contained in node P. For writing the two statements of insertion (tmp->link = p->link; p->link = tmp;), we need to find the pointer p which points to the node that contains item. The procedure is same as we have seen in searching of an element in the linked list.

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

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

Let us see what happens if `item` is present in the last node and we have to insert after the last node. In this case p points to last node and its link is NULL, so `tmp->link` is automatically assigned NULL and we don't have any need for a special case of insertion at the end. This function will work correctly even if we insert after the last node.

3.1.3.4.2 Insertion before a node

In this case we are given a value from the list, and we have to insert the new node before the node that contains this value. Suppose the node Q contains the given value, and node P is its predecessor. We have to insert the new node T before node Q, i.e. T is to be inserted between nodes P and Q. In the function given below, `data` is the new value to be inserted and `item` is the value contained in node Q. For writing the two statements of insertion (`tmp->link = p->link; p->link = tmp;`) we need to find the pointer `p` which points to the predecessor of the node that contains `item`. Since item is present in Q and we have to find pointer to node P, so here the condition for searching would be `if(p->link->info == item)` and terminating condition of the loop would be `(p->link! = NULL)`

```
struct node *addbefore(struct node *start,int data,int item)
{
        struct node *tmp,*p;
        if(start == NULL )
        {
                printf("List is empty\n");
                return start;
        }
        /*If data to be inserted before first node*/
        if(item == start->info)
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info = data;
                tmp->link = start;
                start = tmp;
                return start;
        }
        p = start;
        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 start;
                }
                p = p->link;
        }
        printf("%d not present in the list\n",item);
        return start;
}/*End of addbefore()*/
```

If the node is to be inserted before the first node, then that case has to be handled separately because we have to update `start` in this case. If the list is empty, start will be `NULL` and the term `start->info` will create problems so before checking the condition `if(item == start->info)` we should check for empty list.

3.1.3.4.3 Insertion at a given position

In this case we have to insert the new node at a given position. The two insertion statements are same as in the previous cases (`tmp->link = p->link; p->link = tmp;`).

The way of finding pointer `p` is different. If we have to insert at the first position we will have to update `start` so that case is handled separately.

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

3.1.4 Creation of a Single Linked List

A list can be created using the insertion operations. First time we will have to insert into an empty list, and then we will keep on inserting nodes at the end of the list. The case of insertion in an empty list reduces to the case of insertion in the beginning, so for inserting the first node we will call `addatbeg()`, 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 = addatbeg(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()*/
```

3.1.5 Deletion in a Single Linked List

For deletion of any node, the pointers are rearranged so that this node is logically removed from the list. To physically remove the node and return the memory occupied by it to the pool of available memory we will use the function `free()`. We will take a pointer variable `tmp` which will point to the node being deleted so that after the pointers have been altered we will still have address of that node in `tmp` to free it. There can be four cases while deleting an element from a linked list.

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

In all the cases, at the end we should call `free(tmp)` to physically remove node T from the memory.

3.1.5.1 Deletion of first node

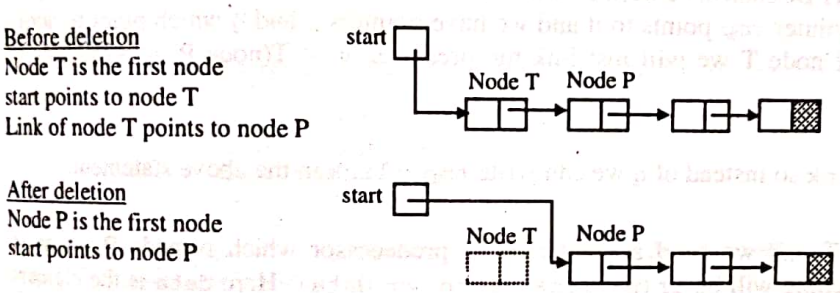


Figure 3.9 Deletion of first node

Since the node to be deleted is the first node, `tmp` will be assigned the address of first node.

```
tmp = start;
```

So now `tmp` points to the first node, which has to be deleted. Since `start` points to the first node of linked list, `start->link` will point to the second node of linked list. After deletion of first node, the second node(node P) would become the first one, so `start` should be assigned the address of the node P as-

```
start = start->link;
```

After this statement, `start` points to node P so now it is the first node of the list.

3.1.5.2 Deletion of the only node

If there is only one node in the list and we have to delete it, then after deletion the list would become empty and `start` would have `NULL` value.

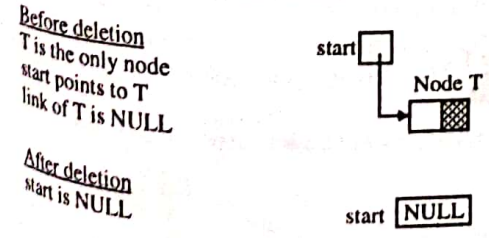


Figure 3.10 Deletion of the only node

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

In the second statement, we can write `start->link` instead of `NULL`. So this case reduces to the first one(3.1.5.1).

3.1.5.3 Deletion in between the list nodes

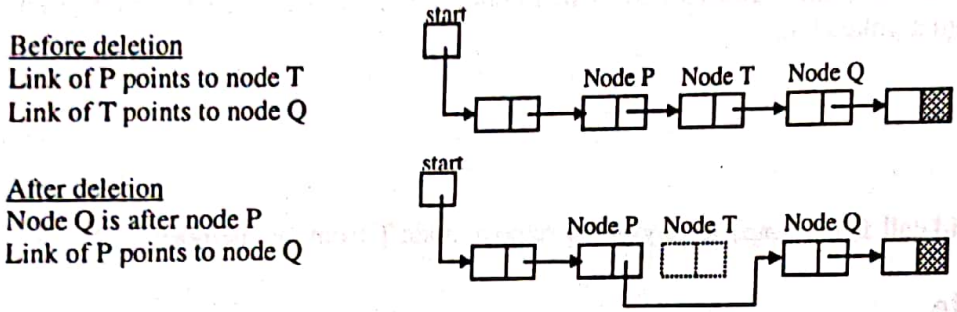


Figure 3.11 Deletion in between the list nodes

Suppose node T is to be deleted and pointer `tmp` points to it and we have pointers `p` and `q` which point to nodes P and Q respectively. For deletion of node T we will just link the predecessor of T(node P) to successor of T(node Q).

```
p->link = q;
```

The address of `q` is stored in `tmp->link` so instead of `q` we can write `tmp->link` in the above statement.

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

The value to be deleted is in node T and we need a pointer to its predecessor which is node P, so as in `addbefore()` our condition for searching will be `if(p->link->info == data)`. Here `data` is the element to be deleted.

```
p = start;
while(p->link!= NULL)
{
        if(p->link->info == data)
        {
                tmp = p->link;
                p->link = tmp->link;
                free(tmp);
                return start;
        }
        p = p->link;
}
```

3.1.5.4 Deletion at the end of the list

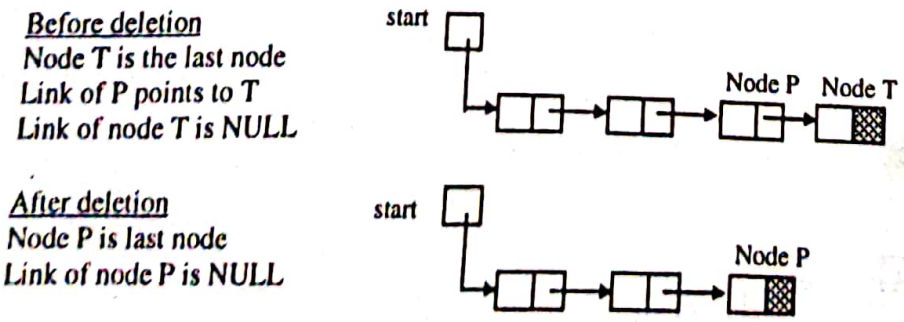


Figure 3.12 Deletion at the end of the list

If `p` is a pointer to node P, then node T can be deleted by writing the following statement.

```
p->link = NULL;
```

Since link of node T is NULL, in the above statement instead of NULL we can write tmp->link. Hence we can write this statement as-

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

So we can see that this case reduces to the previous case(3.1.5.3).

Now we will write a function for deletion of an element from the list. We can't delete from an empty list so firstly we will check if the list is empty. If the element to be deleted is the first element of the list then it is deleted accordingly and we return from the function.

Now the element can either be in between the list nodes or at end of the list. These two cases can be handled in the same way. If we reach the end of list and node containing the value is not found, then a message is displayed.

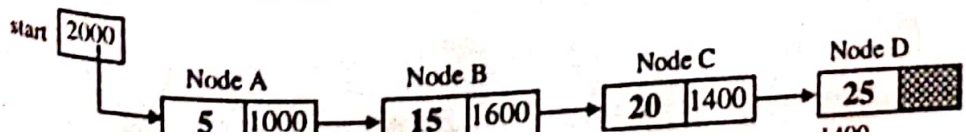
```
struct node *del(struct node *start,int data)
{
        struct node *tmp,*p;
        if(start == NULL)
        {
                printf("List is empty\n");
                return start;
        }
        if(start->info == data)  /*Deletion of first node*/
        {
                tmp = start;
 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

3.1.6 Reversing a Single Linked List

The following changes need to be done in a single linked list for reversing it.

1. First node should become the last node of linked list.
2. Last node should become the first node of linked list and now start should point to it.
3. Link of 2nd node should point to 1st node, link of 3rd node should point to 2nd node and so on.

Let us take a single linked list and reverse it.



(i) Node A is the first node so `start` points to it.
(ii) Node D is the last node so its link is `NULL`.
(iii) Link of A points to B, link of B points to C and link of C points to D.

After reversing, the linked list would be-

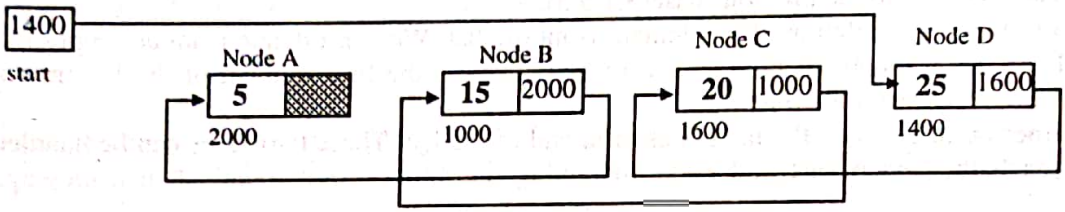


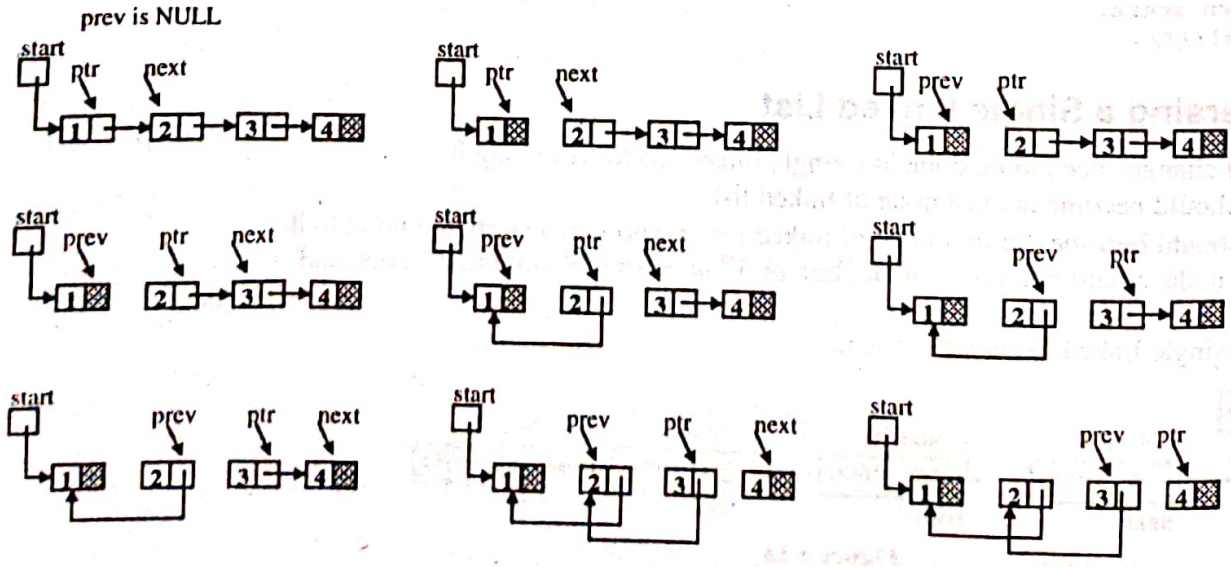
Figure 3.14

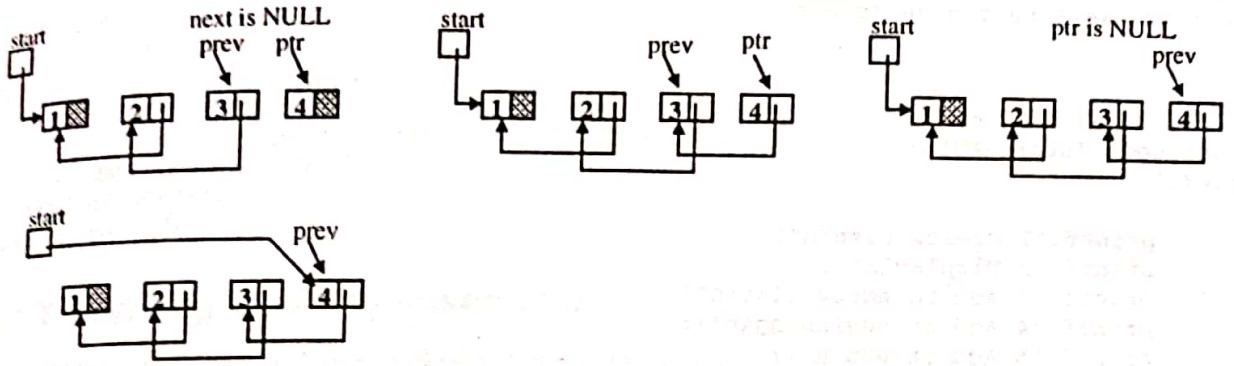
(i) Node D is the first node so `start` points to it.
(ii) Node A is the last node so its link is `NULL`.
(iii) Link of D points to C, link of C points to B and link of B points to A.

Now let us see how we can make a function for the reversal of a linked list. We will take three pointers `prev`, `ptr` and `next`. Initially the pointer `ptr` will point to `start` and `prev` will be `NULL`. In each pass, first the link of pointer `ptr` is stored in pointer `next` and after that the link of `ptr` is changed so that it points to its previous node. The pointers `prev` and `ptr` are moved forward. Here is the function `reverse()` that reverses a linked list.

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

The following example shows all the steps of reversing a single linked list.





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.

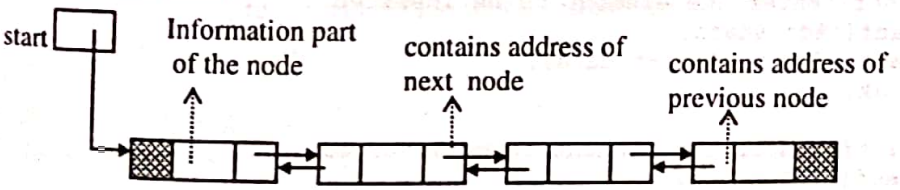


Figure 3.15 Doubly linked list

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

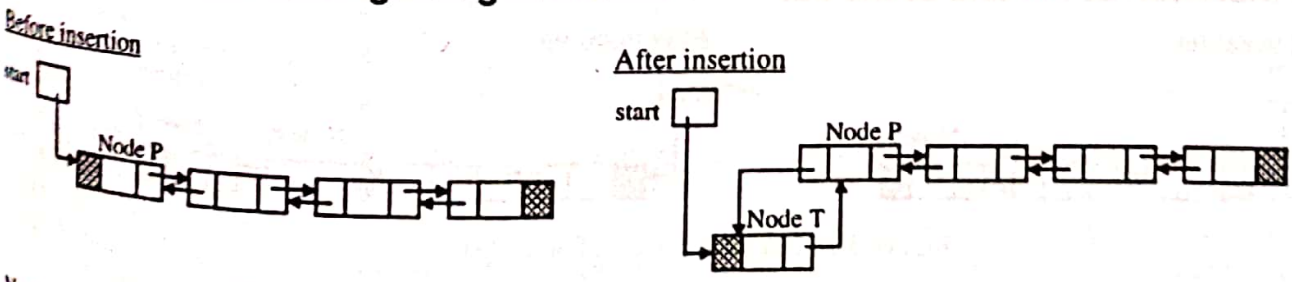


Figure 3.16 Insertion at the beginning of the list

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

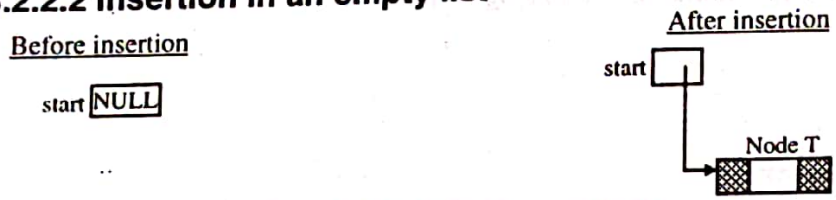


Figure 3.17 Insertion in an empty list

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

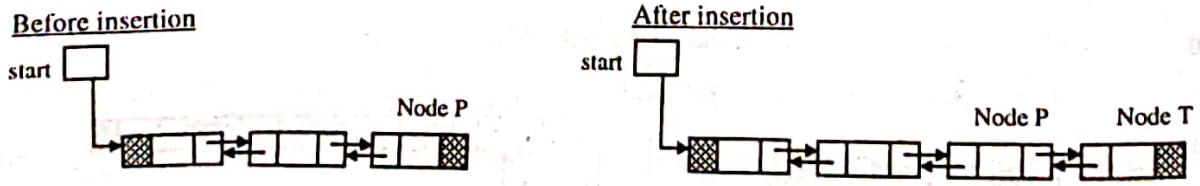


Figure 3.18 Insertion at the end of the list

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

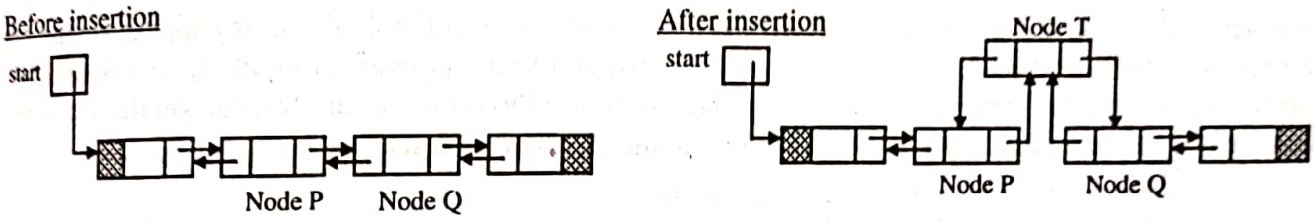


Figure 3.19 Insertion in between the nodes

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

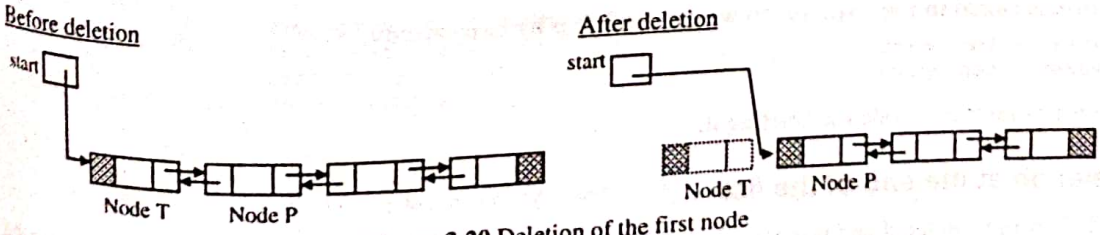


Figure 3.20 Deletion of the first node

`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

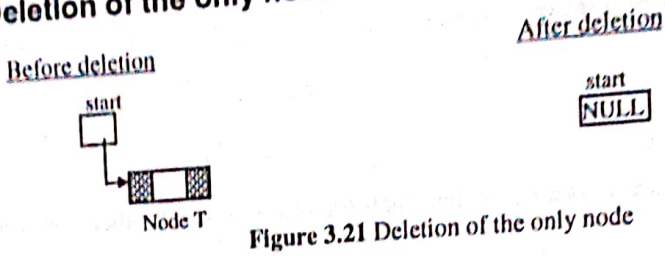


Figure 3.21 Deletion of the only node

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

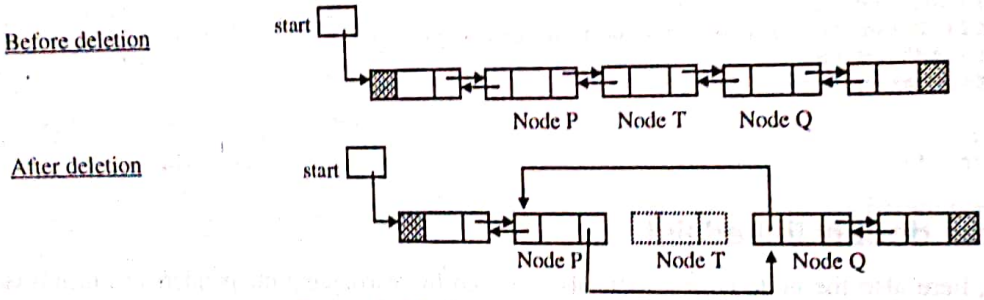


Figure 3.22 Deletion in between the nodes

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;
```

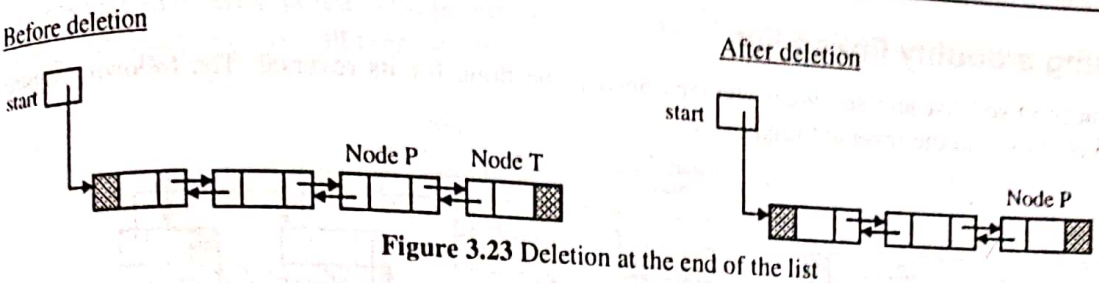


Figure 3.23 Deletion at the end of the list

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.

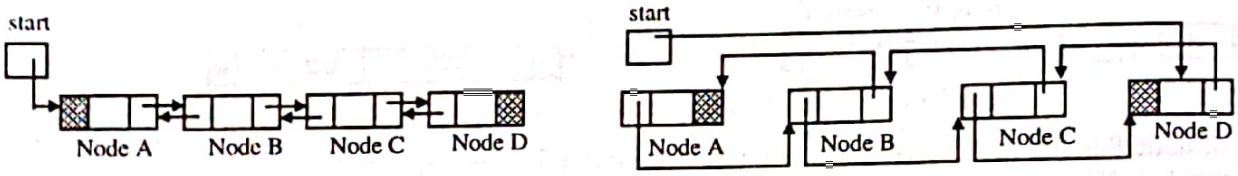


Figure 3.24

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.

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.

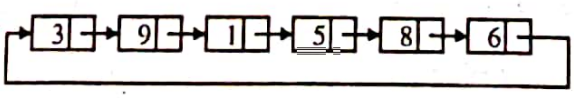


Figure 3.25

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.

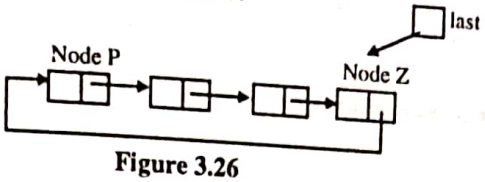


Figure 3.26

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.

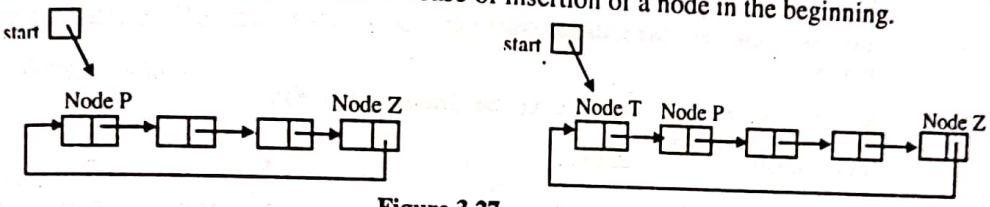


Figure 3.27

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

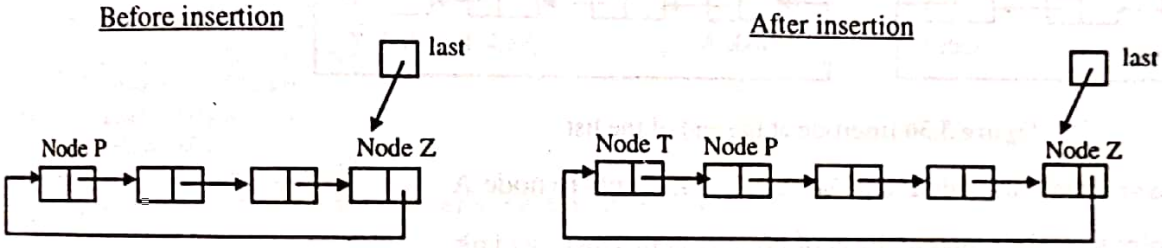


Figure 3.27 Insertion at the beginning of the list

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

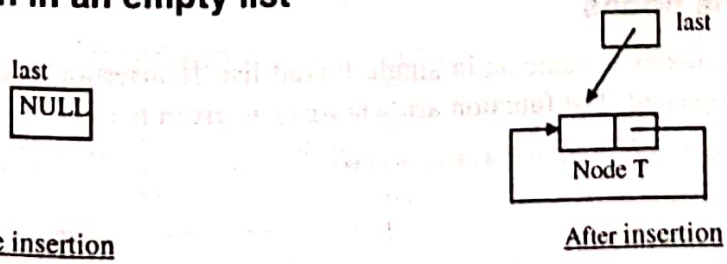


Figure 3.29 Insertion in an empty list

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

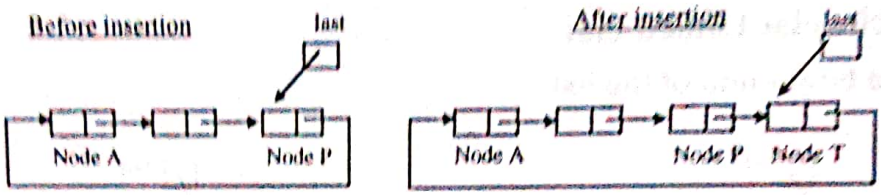


Figure 3.30 Insertion at the end of the list

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;
```

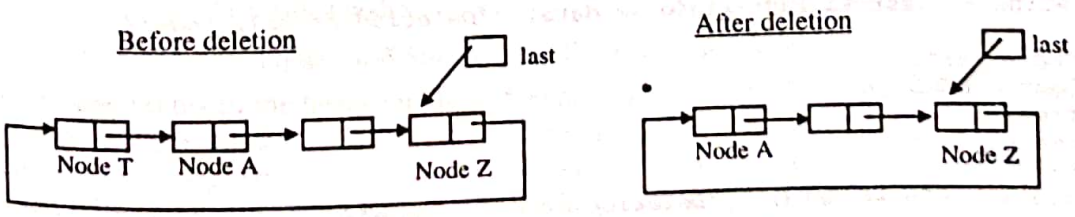


Figure 3.31 Deletion of the first node

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.

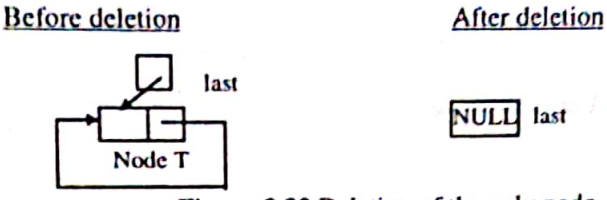


Figure 3.32 Deletion of the only node

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

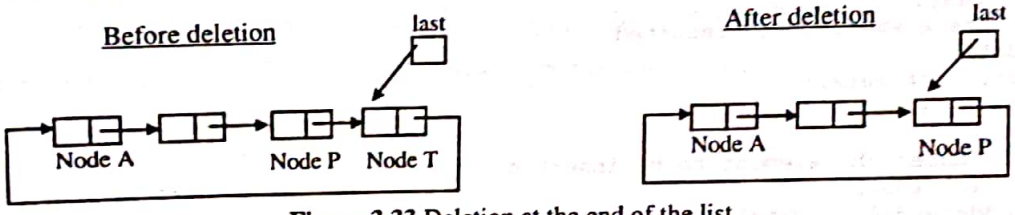


Figure 3.33 Deletion at the end of the list

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.

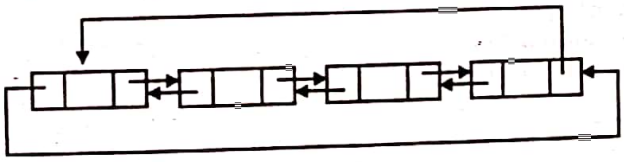


Figure 3.34 Circular double linked list

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.

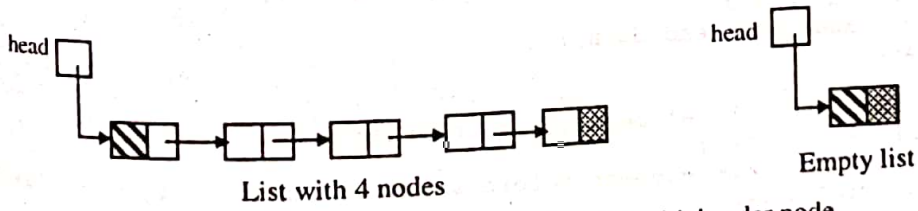


Figure 3.35 Single linked list with header node

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 figure shows circular single linked list with header node.

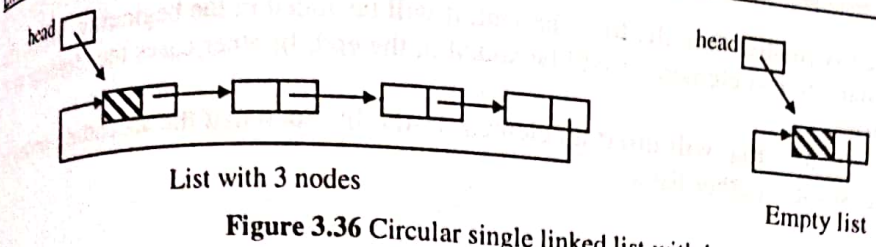
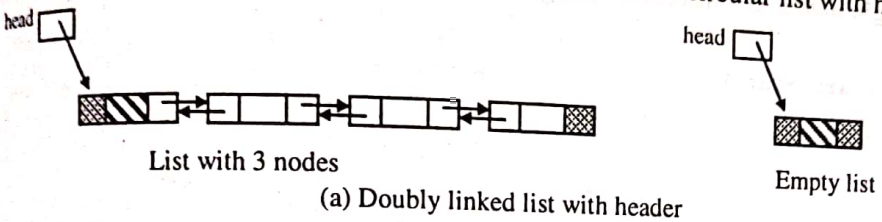


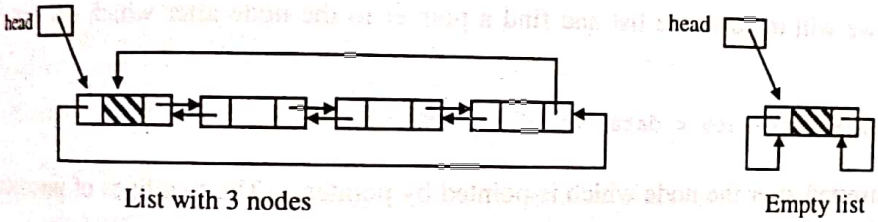
Figure 3.36 Circular single linked list with header

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.

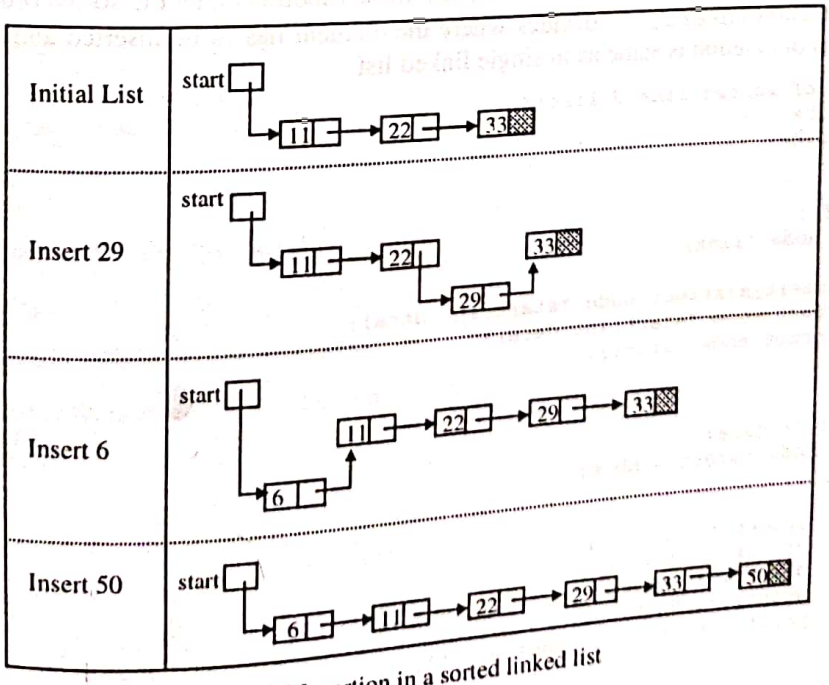


Figure 3.38 Insertion in a sorted linked list

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.

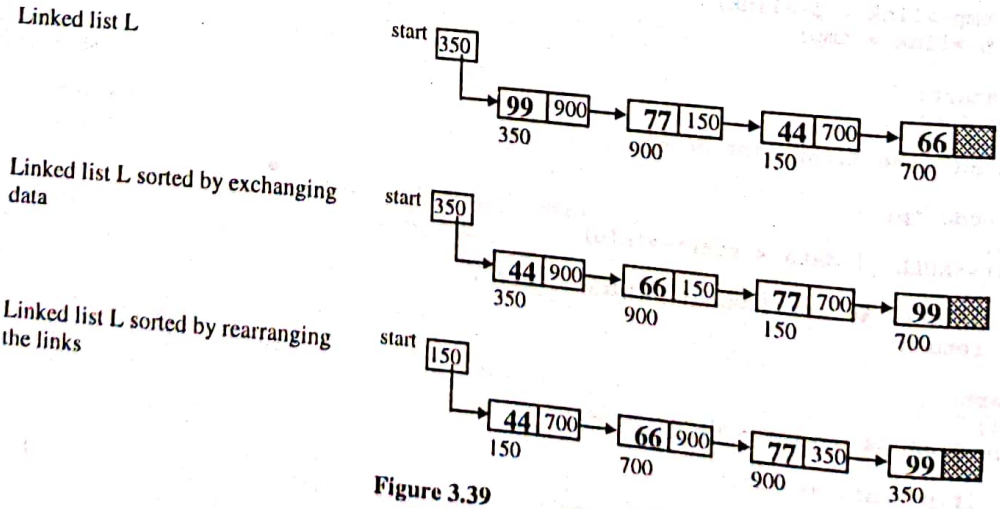
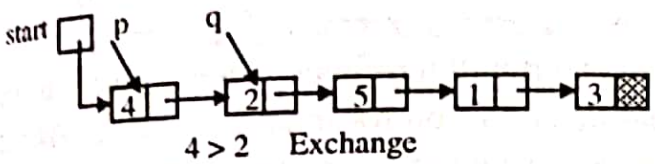


Figure 3.39

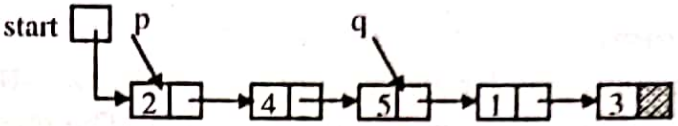
3.6.1 Selection Sort by exchanging data

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

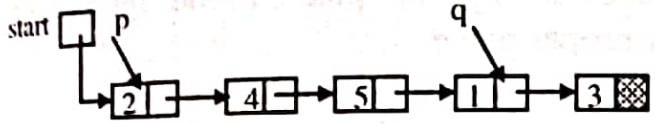
First pass



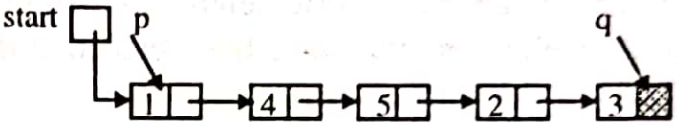
4 > 2 Exchange



2 < 5 No Exchange

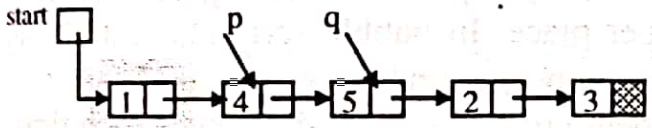


2 > 1 Exchange

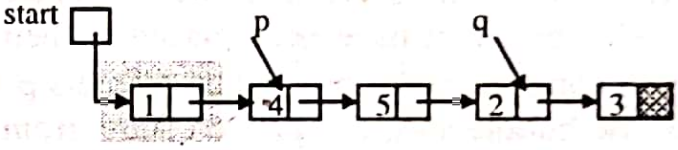


1 < 3 No Exchange

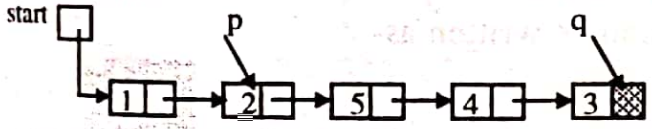
Second pass



4 < 5 No Exchange

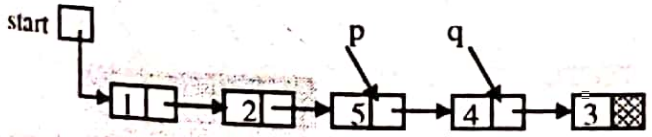


4 > 2 Exchange

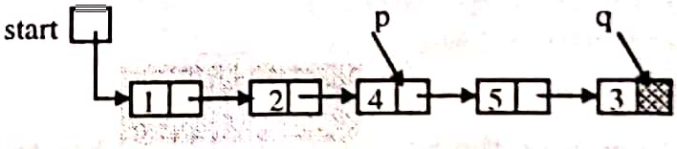


2 < 3 No Exchange

Third pass

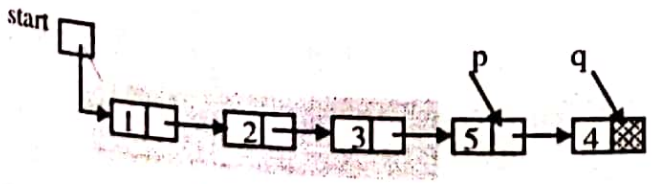


5 > 4 Exchange



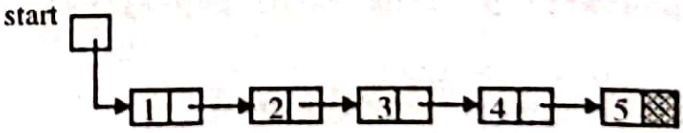
4 > 3 Exchange

Fourth pass



5 > 4 Exchange

Sorted List



```
                    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;
                }
        }
}
```

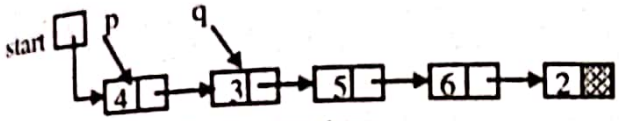
The pointer variable `end` is NULL in the first iteration of outer loop, so inner loop will terminate when p points to the last node i.e. the inner loop will work only till p reaches second last node. After first iteration, value of `end` is updated and is made equal to q. So now `end` points to the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

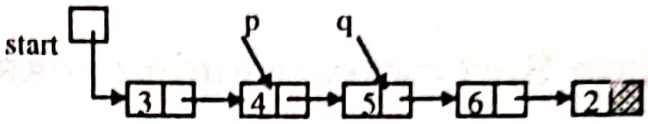
so on. The terminating condition for outer loop is taken as `(end!=start->link)`, so the outer terminate when end points to second node, i.e. the outer loop will work only till end reaches the third

First pass

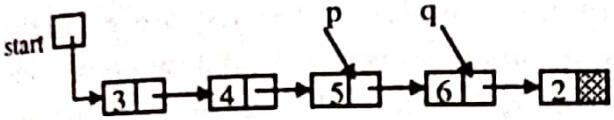
Pointer end is NULL



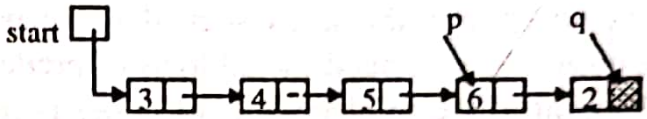
4 > 3 Exchange



4 < 5 No Exchange

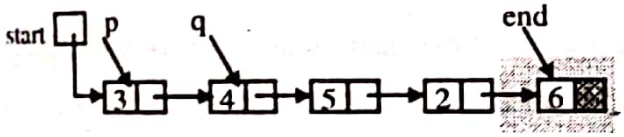


5 < 6 No Exchange

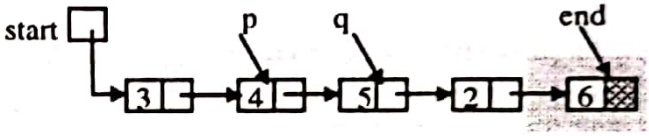


6 > 2 Exchange

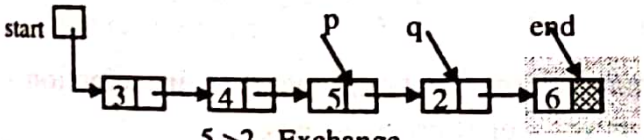
Second pass



3 < 4 No Exchange

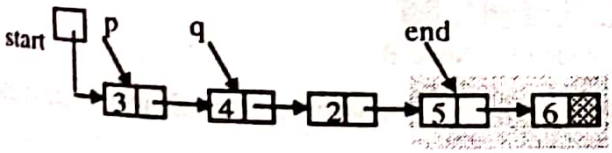


4 < 5 No Exchange

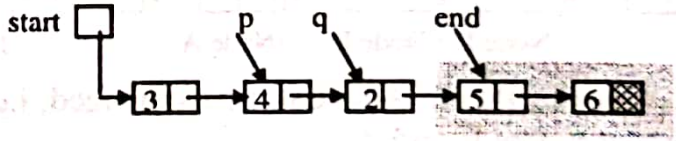


5 >2 Exchange

Third pass



3 < 4 No Exchange



4 > 2 Exchange

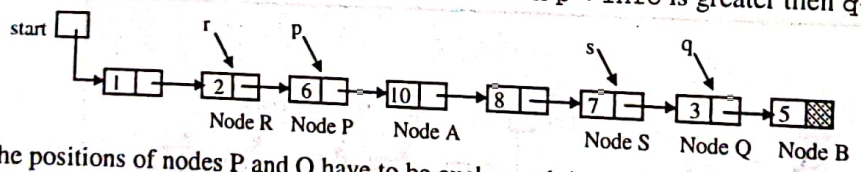
```
                        p->info = q->info;
                        q->info = tmp;
                    }
                }
            }
}/*End of bubble()*/
```

3.6.3 Selection Sort by rearranging links

The pointers p and q will move in the same manner as in selection sort by exchanging data. We will compare nodes pointed by pointers p and q. If the value in node pointed by p is more than the value in node pointed by q then we will have to change the links such that the positions of these nodes in the list are exchanged. For changing the positions we will need the address of predecessor nodes also. So we will take two more pointers r and s which will point to the predecessors of nodes pointed by p and q respectively. In this case the two loops can be written as-

```
for(r=p=start; p->link!=NULL; r=p,p=p->link)
{
        for(s=q=p->link; q!=NULL; s=q,q=q->link)
        {
                if(p->info > q->info)
                {
                        ..................
                }
        }
}
```

In both the loops, pointers p and q are initialized and changed in the same way as in selection sort by exchanging data. Now let us see what is to be done if `p->info` is greater then `q->info`.



The positions of nodes P and Q have to be exchanged, i.e. node P should be between nodes S and B and node Q should be between nodes R and A

(i) Node P should be before node B, so link of node P should point to node B
`p->link = q->link;`
(ii) Node Q should be before node A, so link of node Q should point to node A
`q->link = p->link;`
(iii) Node Q should be after node R, so link of node R should point to node Q
`r->link = q;`
(iv) Node P should be after node S, so link of node S should point to node P
`s->link = p;`

For writing the first two statements we will need a temporary pointer, since we are exchanging `p->link` and `q->link`.

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

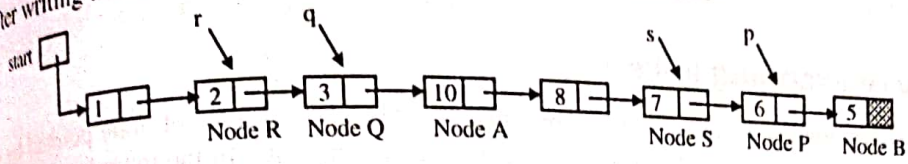
If p points to the first node, then r also points to the first node i.e. nodes R and P both are same, so in this case there is no need of writing the third statement `(r->link = q;)`
We need the third statement only if the pointer p is not equal to `start`. So it can be written as-

```
if(p!=start)
        r->link = q;
```

If `start` points to node P, then `start` needs to be updated and now it should point to node Q.

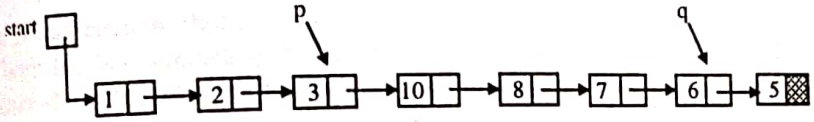
```
if(p==start)
    start = q;
```

After writing the above statements the linked list will look like this-



The positions of nodes P and Q have changed and this is what we want because the value in node P was more than value in node Q. Now we will bring the pointers p and q back to their positions to continue with our sorting process. For this we will exchange the pointers p and q with the help of a temporary pointer.

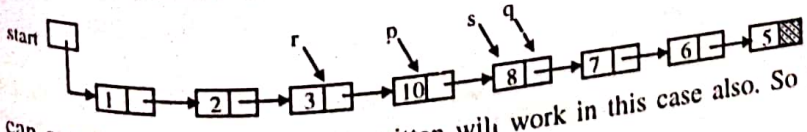
```
tmp = p;   p = q;   q = tmp;
```



Here is the function for selection sort by rearranging links.

```
struct node *selection_1(struct node *start)
{
    struct node *p,*q,*r,*s,*tmp;
    for(r=p=start; p->link!=NULL; r=p,p=p->link)
    {
        for(s=q=p->link; q!=NULL; s=q,q=q->link)
        {
            if(p->info > q->info)
            {
                tmp = p->link;
                p->link = q->link;
                q->link = tmp;
                if(p!=start)
                r->link = q;
                s->link = p;
                if(p == start)
                    start = q;
                tmp = p;
                p = q;
                q = tmp;
            }
        }
    }
    return start;
}/*End of selection_1()*/
```

In the previous figures, we have taken the case when p and q point to non adjacent nodes, and we have written our code according to this case only. Now let us see whether this code will work when p and q point to adjacent nodes. The pointers p and q will point to adjacent nodes only in the first iteration of inner loop. In that case s and q point to same node. The following figure shows this situation-



You can see that the code we have written will work in this case also. So there is no need to consider a separate case when nodes p and q are adjacent.

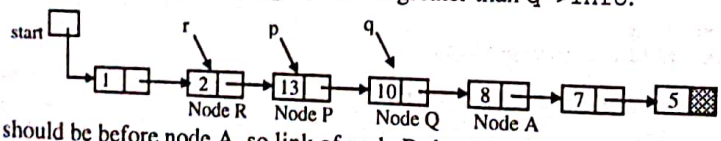
When we sort a linked list by exchanging data, links are not disturbed so `start` remains same. But when sorting is done by rearranging links, the value of `start` might change so it is necessary to return the value of `start` at the end of the function.

3.6.4 Bubble sort by rearranging links

In bubble sort, since p and q are always adjacent, there is no need of taking predecessor of node pointed by q. Both the loops are written in the same way as in bubble sorting by exchanging data. In the inner loop we have taken a pointer r that will point to the predecessor of node pointed by p.

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

Now let us see what is to be done if `p->info` is greater than `q->info`.



Node P should be before node A, so link of node P should point to node A
`p->link = q->link;`
Node Q should be before node P, so link of node Q should point to node P
`q->link = p;`
Node Q should be after node R, so link of node R should point to node Q
`r->link = q;`

Here is the function for bubble sort by rearranging links.

```
struct node *bubble_l(struct node *start)
{
        struct node *end,*r,*p,*q,*tmp;
        for(end=NULL; end!=start->link; end=q)
        {
                for(r=p=start; p->link!=end; r=p,p=p->link)
                {
                        q = p->link;
                        if(p->info > q->info )
                        {
                                p->link = q->link;
                                q->link = p;
                                if(p!=start)
                                  r->link = q;
                                else
                                    start = q;
                                tmp = p;
                                p = q;
                                q = tmp;
                        }
                }
        }
        return start;
}/*End of bubble_l()*/
```

3.7 Merging

If there are two sorted linked lists, then the process of combining these sorted lists into another list of sorted order is called merging. The following figure shows two sorted lists and a third list obtained by merging them.

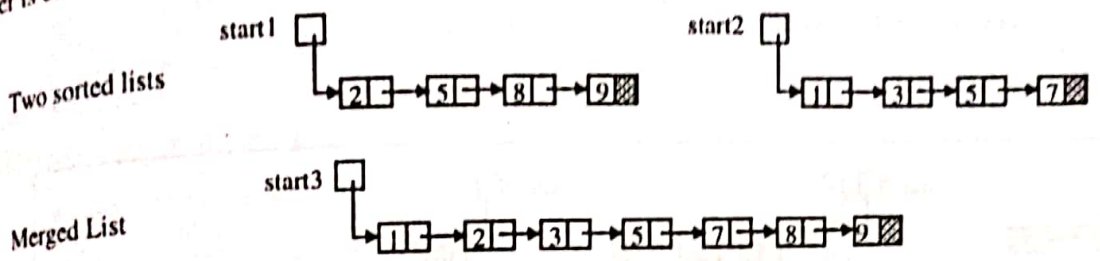


Figure 3.42

If there is any element that is common to both the lists, it will be inserted only once in the third list. For merging, both the lists are scanned from left to right. We will take one element from each list, compare them and then take the smaller one in third list. This process will continue until the elements of one list are finished. Then we will take the remaining elements of unfinished list in third list. The whole process for merging is shown in the figure 3.43. We've taken two pointers `p1` and `p2` that will point to the nodes that are being compared. There can be three cases while comparing `p1->info` and `p2->info`

1. If `(p1->info) < (p2->info)`
The new node that is added to the resultant list has info equal to `p1->info`. After this we will make `p1` point to the next node of first list.

2. If `(p2->info) < (p1->info)`
The new node that is added to the resultant list has info equal to `p2->info`. After this we will make `p2` point to the next node of second list.

3. If `(p1->info) == (p2->info)`
The new node that is added to the resultant list has info equal to `p1->info`(or `p2->info`). After this we will make `p1` and `p2` point to the next nodes of first list and second list respectively. The procedure of merging is shown in figure 3.43.

```
while(p1!=NULL && p2!=NULL)
{
      if(p1->info < p2->info)
      {
            start3 = insert(start3,p1->info);
            p1 = p1->link;
      }
      else if(p2->info < p1->info)
      {
            start3 = insert(start3,p2->info);
            p2 = p2->link;
      }
      else if(p1->info == p2->info)
      {
            start3 = insert(start3,p1->info);
            p1 = p1->link;
            p2 = p2->link;
}     }
```

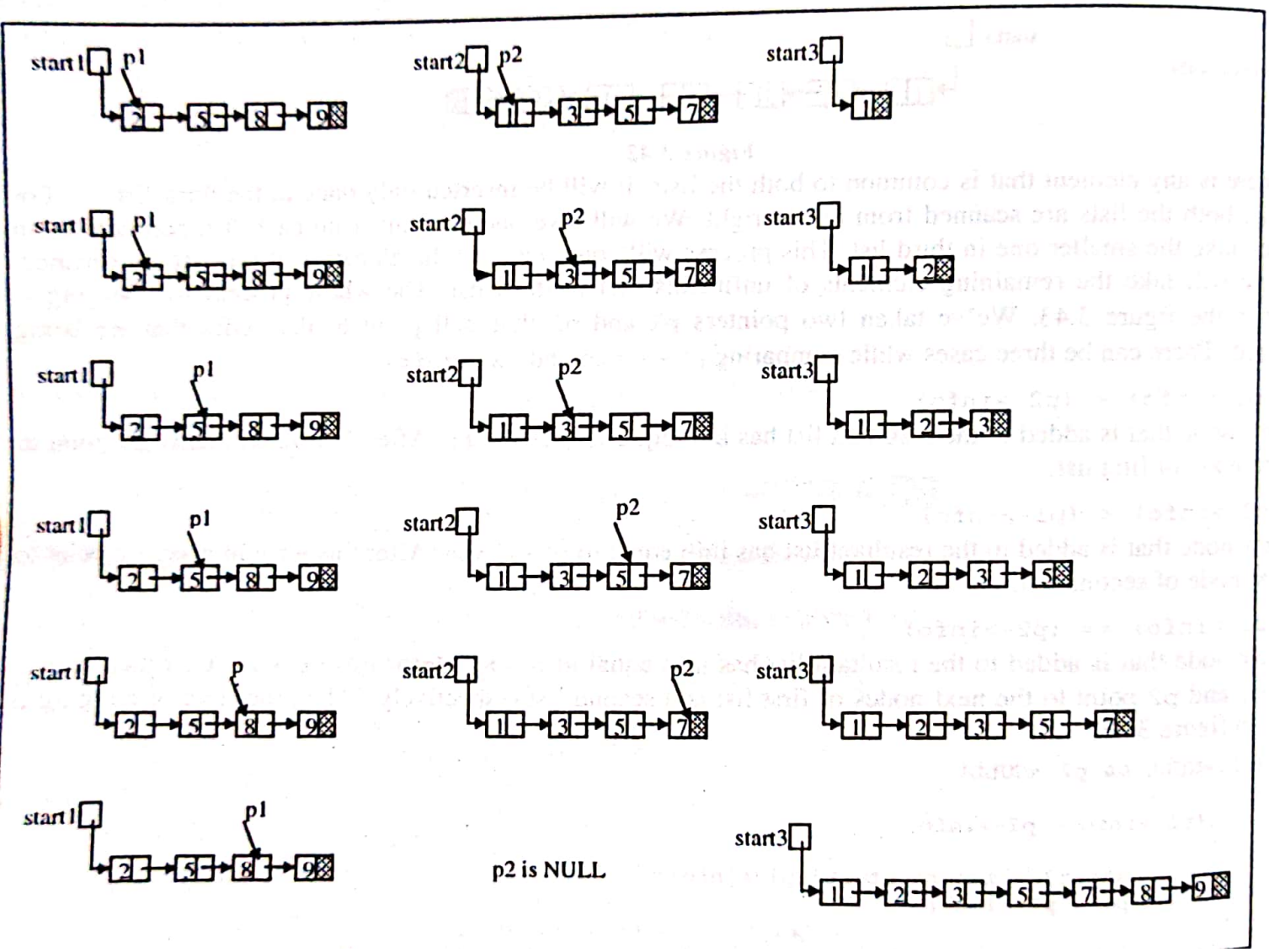
The above loop will terminate when any of the list will finish. Now we have to add the remaining nodes of the unfinished list to the resultant list. If second list has finished, then we will insert all the nodes of first list in the resultant list as-

```
while(p1!=NULL)
{
      start3 = insert(start3,p1->info);
```

```
        p1 = p1->link;
}
```

If first list has finished, then we will insert all the nodes of second list in the resultant list as-

```
while(p2!=NULL)
{
        start3 = insert(start3,p2->info);
        p2 = p2->link;
}
```



```
main()
{
        struct node *start1 = NULL,*start2 = NULL;
        start1 = create(start1);
        start2 = create(start2);
        printf("List1 : "); display(start1);
        printf("List2 : "); display(start2);
        merge(start1, start2);
}/*End of main()*/
void merge(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info < p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                }
                else if(p2->info < p1->info)
                {
                        start3 = insert(start3,p2->info);
                        p2 = p2->link;
                }
                else if(p1->info==p2->info)
                {
                        start3 = insert(start3,p1->info);
                        p1 = p1->link;
                        p2 = p2->link;
                }
        }
        /*If second list has finished and elements left in first list*/
        while(p1!=NULL)
        {
                start3 = insert(start3,p1->info);
                p1 = p1->link;
        }
        /*If first list has finished and elements left in second list*/
        while(p2!=NULL)
        {
                start3 = insert(start3,p2->info);
                p2 = p2->link;
        }
        printf("Merged list is : ");
}       display(start3);

struct node *create(struct node *start)
{
        int i,n,data;

        printf("Enter the number of nodes : ");
        scanf("%d",&n);
        start = NULL;
        for(i=1;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start = insert_s(start, data);
        }
        return start;
}/*End of create_slist()*/
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 data to be added in beginning */
        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_s()*/

struct node *insert(struct node *start,int data)
{
        struct node *p,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info = data;
        if(start == NULL)  /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
                return start;
        }
        else    /*Insert at the end of the list*/
        {
                p = start;
                while(p->link!=NULL)
                        p = p->link;
                tmp->link = p->link;
                p->link = tmp;
        }
        return start;
}/*End of insert()*/

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

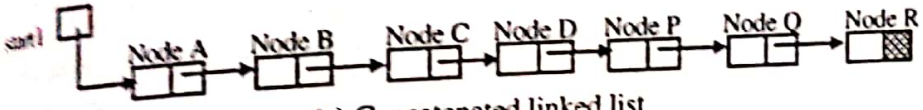
The function insert_s() is the same that we have made in sorted linked list and it inserts nodes in ascending order. We have used this function to create the two sorted lists that are to be merged. The function

insert() is a simple function that inserts nodes in a linked list at the end. We have used this function to insert nodes in the third list.

3.8 Concatenation

Suppose we have two single linked lists and we want to append one at the end of another. For this the link of last node of first list should point to the first node of the second list. Let us take two single linked lists and concatenate them.

(a) First linked list (b) Second linked list



(a) Concatenated linked list

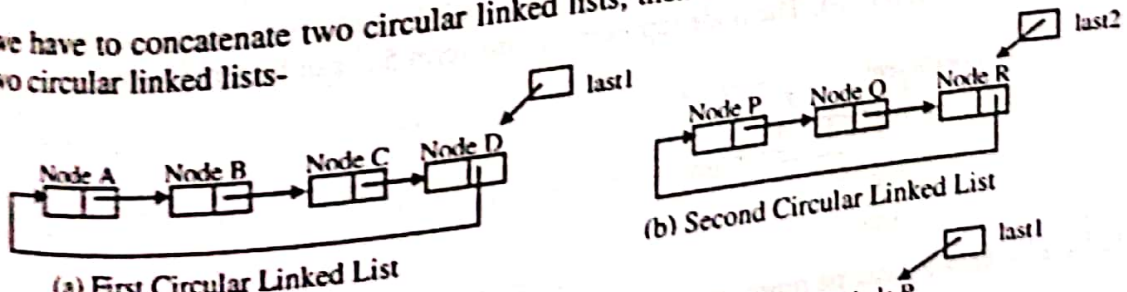
Figure 3.44

For concatenation, link of node D should point to node P. To get the address of node D, we have to traverse the first list till the end. Suppose ptr points to node D, then ptr->link should be made equal to start2.

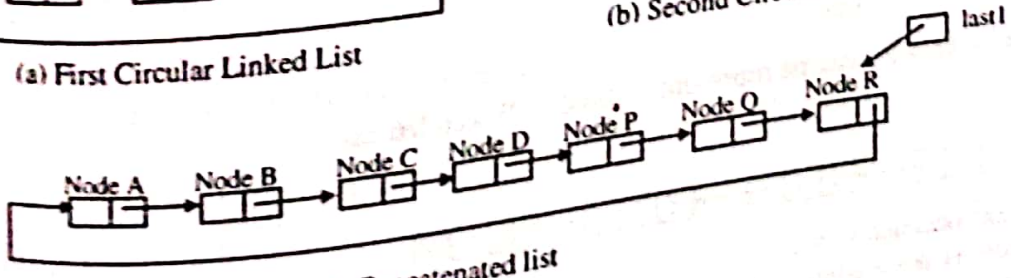
```
ptr->link = start2;
```

```
struct node *concat(struct node *start1, struct node *start2)
{
      struct node *ptr;
      if(start1 == NULL)
      {
            start1 = start2;
            return start1;
      }
      if(start2 == NULL)
            return start1;
      ptr = start1;
      while(ptr->link != NULL)
            ptr = ptr->link;
      ptr->link = start2;
      return start1;
}
```

If we have to concatenate two circular linked lists, then there is no need to traverse any of the lists. Let us take two circular linked lists-



(a) First Circular Linked List (b) Second Circular Linked List



(c) Concatenated list

Figure 3.45

Link of node D should point to node P

```
last1->link = last2->link;
```

We will lose the address of node A, so before writing this statement we should save `last1->link`.

```
ptr = last1->link;
```

Link of node R should point to node A

```
last2->link = ptr;
```

The pointer `last1` should point to node R

```
last1 = last2;
```

```
struct node *concat(struct node *last1, struct node *last2)
{
        struct node *ptr;
        if(last1 == NULL)
        {
                last1 = last2;
                return last1;
        }
        if(last2 == NULL)
                return last1;
        ptr = last1->link;
        last1->link = last2->link;
        last2->link = ptr;
        last1 = last2;
        return last1;
}
```

3.9 Polynomial arithmetic with linked list

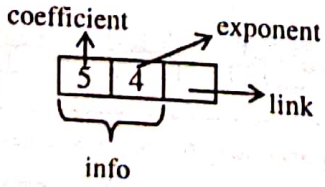
A useful application of linked list is the representation of polynomial expressions. Let us take a polynomial expression with single variable-

$5x^4 + x^3 - 6x + 2$

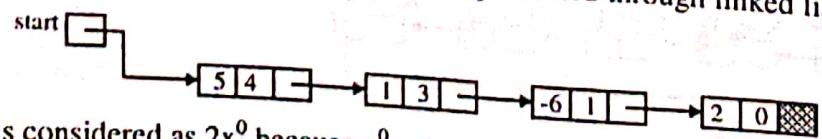
In each term we have a coefficient and an exponent. For example in the term $5x^4$, coefficient is 5 and exponent is 4. The whole polynomial can be represented through linked list where each node will represent a term of the expression. The structure for each node will be-

```
struct node{
                float coefficient;
                int exponent;
                struct node *link;
            }
```

Here the info part of the node contains coefficient and exponent and the link part is same as before and will be used to point to the next node of the list. The node representing the term $5x^4$ can be represented as-



The polynomial $(5x^4 + x^3 - 6x + 2)$ can be represented through linked list as-



Here 2 is considered as $2x^0$ because $x^0 = 1$.

The arithmetic operations are easier if the terms are arranged in descending order of their exponents. For example it would be better if the polynomial expression $(5x + 6x^3 + x^2 - 9 + 2x^6)$ is stored as $(2x^6 + 6x^3 + x^2 + 5x - 9)$. So for representing the polynomial expression, we will use sorted

descending order based on the exponent. An empty list will represent zero polynomial. The following program shows creation of polynomial linked lists and their addition and multiplication.

```
/*P3.10 Program of polynomial addition and multiplication using linked list*/
#include<stdio.h>
#include<stdlib.h>
struct node
{
      float coef;
      int expo;
      struct node *link;
};
struct node *create(struct node *);
struct node *insert_s(struct node *,float,int);
struct node *insert(struct node *,float,int);
void display(struct node *ptr);
void poly_add(struct node *,struct node *);
void poly_mult(struct node *,struct node *);

main()
{
      struct node *start1 = NULL,*start2 = NULL;
      printf("Enter polynomial 1 :\n");   start1 = create(start1);
      printf("Enter polynomial 2 :\n");   start2 = create(start2);
      printf("Polynomial 1 is :  ");  display(start1);
      printf("Polynomial 2 is :  ");  display(start2);
      poly_add(start1, start2);
      poly_mult(start1, start2);
}/*End of main()*/

struct node *create(struct node *start)
{
      int i,n,ex;
      float co;
      printf("Enter the number of terms : ");
      scanf("%d",&n);
      for(i=1;i<=n;i++)
      {
            printf("Enter coeficient for term %d : ",i);
            scanf("%f",&co);
            printf("Enter exponent for term %d : ",i);
            scanf("%d",&ex);
            start = insert_s(start,co,ex);
      }
      return start;
}/*End of create()*/

struct node *insert_s(struct node *start,float co,int ex)
{
      struct node *ptr,*tmp;
      tmp = (struct node *)malloc(sizeof(struct node));
      tmp->coef = co;
      tmp->expo = ex;
      /*list empty or exp greater than first one*/
      if(start == NULL || ex > start->expo)
      {
            tmp->link = start;
            start = tmp;
      }
      else
      {
            ptr = start;
            while(ptr->link!=NULL && ptr->link->expo >= ex)
                  ptr = ptr->link;
```

```
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/

struct node *insert(struct node *start,float co,int ex)
{
        struct node *ptr,*tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->coef = co;
        tmp->expo = ex;
        if(start == NULL)       /*If list is empty*/
        {
                tmp->link = start;
                start = tmp;
        }
        else    /*Insert at the end of the list*/
        {
                ptr = start;
                while(ptr->link!=NULL)
                        ptr = ptr->link;
                tmp->link = ptr->link;
                ptr->link = tmp;
        }
        return start;
}/*End of insert()*/

void display(struct node *ptr)
{
        if(ptr == NULL)
        {
                printf("Zero polynomial\n");
                return;
        }
        while(ptr!=NULL)
        {
                printf("(%.1fx^%d)", ptr->coef,ptr->expo);
                ptr = ptr->link;
                if(ptr!=NULL)
                        printf(" + ");
                else
                        printf("\n");
        }
}/*End of display()*/

void poly_add(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3 = NULL;
        while(p1!=NULL && p2!=NULL)
        {
                if(p1->expo > p2->expo)
                {
                        start3 = insert(start3,p1->coef,p1->expo);
                        p1 = p1->link;
                }
                else if(p2->expo > p1->expo)
                {
                        start3 = insert(start3,p2->coef,p2->expo);
                        p2 = p2->link;
                }
                else if(p1->expo == p2->expo)
                {
                        start3 = insert(start3,p1->coef+p2->coef,p1->expo);
```

```
                p1 = p1->link;
                p2 = p2->link;
            }
        }
        /*if poly2 has finished and elements left in poly1*/
        while(p1!=NULL)
        {
            start3 = insert(start3,p1->coef,p1->expo);
            p1 = p1->link;
        }
        /*if poly1 has finished and elements left in poly2*/
        while(p2!=NULL)
        {
            start3 = insert(start3,p2->coef,p2->expo);
            p2 = p2->link;
        }
        printf("Added polynomial is : ");
        display(start3);
}/*End of poly_add() */

void poly_mult(struct node *p1, struct node *p2)
{
        struct node *start3;
        struct node *p2_beg = p2;

        start3 = NULL;
        if(p1 == NULL || p2 == NULL)
        {
            printf("Multiplied polynomial is zero polynomial\n");
            return;
        }
        while(p1!=NULL)
        {
           p2 = p2_beg;
           while(p2!=NULL)
           {
             start3 = insert_s(start3,p1->coef*p2->coef,p1->expo+p2->expo);
             p2 = p2->link;
           }
           p1 = p1->link;
        }
       printf("Multiplied polynomial is : ");
       display(start3);
}/*End of poly_mult()*/
```

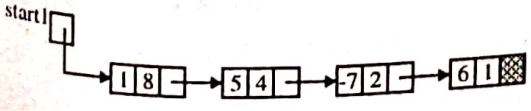
3.9.1 Creation of polynomial linked list

The function `create()` is very simple and calls another function `insert_s()` that inserts a node in polynomial linked list. The function `insert_s()` is similar to that of sorted linked lists. The only difference is that here our list is in descending order based on the exponent.

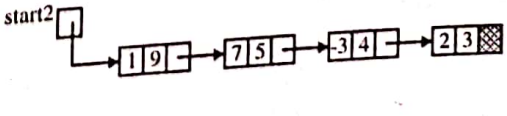
3.9.2 Addition of 2 polynomials

The procedure for addition of 2 polynomials represented by linked lists is somewhat similar to that of merging. Let us take two polynomial expression lists and make a third list by adding them.

Poly1- $x^8 + 5x^4 - 7x^2 + 6x$



Poly2- $x^9 + 7x^5 - 3x^4 + 2x^3$



The pointers p1 and p2 will point to the current nodes in the polynomials which will be added. The process of addition is shown in the figure 3.46. Both polynomials are traversed until one polynomial finishes. We can have three cases-

1. If (p1->expo) > (p2->expo)
The new node that is added to the resultant list has coefficient equal to p1->coef and exponent equal to p1->expo. After this we will make p1 point to the next node of polynomial1.

2. If (p2->expo) > (p1->expo)
The new node that is added to the resultant list has coefficient equal to p2->coef and exponent equal to p2->expo. After this we will make p2 point to the next node of polynomial2.

3. If (p1->expo) == (p2->expo)
The new node that is added to the resultant list has coefficient equal to (p1->coef + p2->coef) and exponent equal to p1->expo(or p2->expo). After this we will make p1 and p2 point to the next nodes of polynomial1 and polynomial 2 respectively. The procedure of polynomial addition is shown in figure 3.46.

```
while(p1!=NULL && p2!=NULL)
{
        if(p1->expo > p2->expo)
        {
                p3_start = insert(p3_start,p1->coef,p1->expo);
                p1 = p1->link;
        }
        else if(p2->expo > p1->expo)
        {
                p3_start = insert(p3_start,p2->coef,p2->expo);
                p2 = p2->link;
        }
        else if(p1->expo == p2->expo)
        {
                p3_start = insert (p3_start,p1->coef+p2->coef,p1->expo);
                p1 = p1->link;
                p2 = p2->link;
        }
}
```

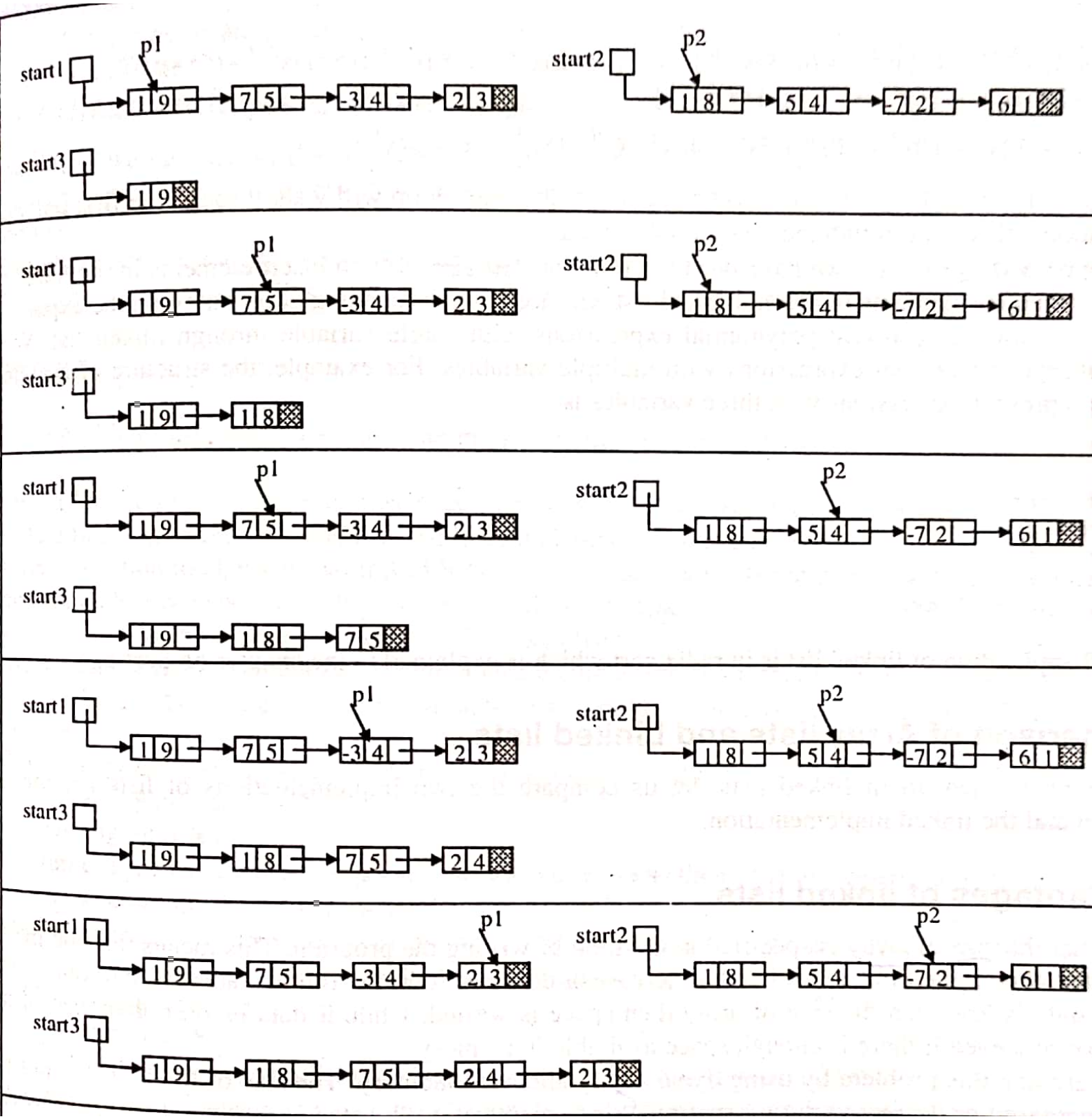
The above loop will terminate when any of the polynomial will finish. Now we have to add the remaining nodes of the unfinished polynomial to the resultant list. If polynomial 2 has finished, then we will put all the terms of polynomial 1 in the resultant list as-

```
while(p1!=NULL)
{
        p3_start = insert(p3_start,p1->coef,p1->expo);
        p1 = p1->link;
}
```

If polynomial 1 has finished, then we will put all the terms of polynomial 2 in the resultant list as-

```
while(p2!=NULL)
{
        p3_start = insert(p3_start,p2->coef,p2->expo);
        p2 = p2->link;
}
```

We can see the advantage of storing the terms in descending order of their exponents. If it was not so then we would have to scan both the lists many times.



For this we have to multiply each term of the first polynomial with each term of second polynomial. When we multiply two terms, their coefficients are multiplied and exponents are added. The product of the above two polynomials is-

$[\ (4*2)x^{3+5} + (4*6)x^{3+4} + (4*1)x^{3+2} + (4*8)x^{3+0}\ ] + [\ (5*2)x^{2+5}\ (5*6)x^{2+4} + (5*1)x^{2+2} + (5*8)x^{2+0}\ ] +$
$[\ (-3*2)x^{1+5} + (-3*6)x^{1+4} + (-3*1)x^{1+2} + (-3*8)x^{1+0}\ ]$

$8x^8 + 24x^7 + 4x^5 + 32x^3 + 10x^7 + 30x^6 + 5x^4 + 40x^2 - 6x^6 - 18x^5 - 3x^3 - 24x^1$

It is obvious that we will have to use two nested loops, the outer loop will walk through the first polynomial and the inner loop will walk through the second polynomial.

In the function `poly_mult()`, we have used the function `insert_s()` to insert elements in the third list. If we don't do so, then the elements in the multiplied list will not be in descending order based on the exponent.

We have seen how to represent polynomial expressions with single variable through linked list. We can extend this concept to represent expressions with multiple variables. For example, the structure of a node that can be used to represent expressions with three variables is-

```
struct node
{
        float coef;
        int expx;
        int expy;
        int expz;
        struct node *link;
};
```

Another useful application of linked list is in radix sort which is explained in the chapter on Sorting.

3.10 Comparison of Array lists and Linked lists

Now after having studied about linked lists, let us compare the two implementations of lists i.e. the array implementation and the linked implementation.

3.10.1 Advantages of linked lists

(1) We know that the size of array is specified at the time of writing the program. This means that the memory space is allocated at compile time, and we can't increase or decrease it during runtime according to our needs. If the amount of data is less than the size of array then space is wasted, while if data is more than size of array then overflow occurs even if there is enough space available in memory.
Linked lists overcome this problem by using dynamically allocated memory. The size of linked list is not fixed and it can be increased or decreased during runtime. Since memory is allocated at runtime, there is no wastage of memory and we can keep on inserting the elements till memory is available. Whenever we need a new node we dynamically allocate it i.e. we get it from the free storage space. When we don't need the node, we can return the memory occupied by it to the free storage space so that it can be used by other programs. We have done these two operations by using `malloc()` and `free()`.

(2) Insertion and deletion inside arrays is not efficient since it requires shifting of elements. For example if we want to insert an element at the 0^{th} position then we have to shift all the elements of the array to the right, and if we want to delete the element present at the 0^{th} position then we have to shift all the elements to the left. These are the worst cases of insertion and deletion and the efficiency is O(n). If the array consists of big records then this shifting is even more time consuming.
In linked lists, insertion and deletion requires only change in pointers. There is no physical movement of data, only the links are altered.

(3) We know that in arrays all the elements are always stored in contiguous locations. Sometimes arrays can't be used because the amount of memory needed by them is not available in contiguous locations i.e. the total

memory required by the array is available but it is dispersed. So in this case we can't create an array in spite of available memory.
In linked list, elements are not stored in contiguous locations. So in the above case, there will be no problem in creation of linked list.

3.10.2 Disadvantages of linked lists

(1) In arrays we can access the n^{th} element directly, but in linked lists we have to pass through the first n-1 elements to reach the n^{th} element. So when it comes to random access, array lists are definitely better than linked lists.
(2) In linked lists the pointer fields take extra space that could have been used for storing some more data.
(3) Writing of programs for linked list is more difficult than that for arrays.

Exercise

In all the problems assume that we have an integer in the info part of nodes.

1. Write a function to count the number of occurrences of an element in a single linked list.
2. Write a function to find the smallest and largest element of a single linked list.
3. Write a function to check if two linked lists are identical. Two lists are identical if they have same number of elements and the corresponding elements in both lists are same
4. Write a function to create a copy of a single linked list.
5. Given a linked list L, write a function to create a single linked list that is reverse of the list L. For example if the list L is `1->2->3->4->5` then the new list should be `5->4->3->2->1`. The list L should remain unchanged.
6. Write a program to swap adjacent elements of a single linked list
(i) by exchanging info part
(ii) by rearranging links.
For example if a linked list is `1->2->3->4->5->6->7->8`, then after swapping adjacent elements it should become `2->1->4->3->6->5->8->7`.
7. Write a program to swap adjacent elements of a double linked list by rearranging links.
8. Write a program to swap the first and last elements of a single linked list
(i) by exchanging info part.
(ii) by rearranging links.
9. Write a function to move the largest element to the end of a single linked list.
10. Write a function to move the smallest element to the beginning of a single linked list.
11. Write a function for deleting all the nodes from a single linked list which have a value N.
12. Given a single linked list L1 which is sorted in ascending order, and another single linked list L2 which is not sorted, write a function to print the elements of second list according to the first list. For example if the first list is `1->2->5->7->8`, then the function should print the 1^{st}, 2^{nd}, 5^{th}, 7^{th}, 8^{th} elements of second list.
13. Write a program to remove first node of the list and insert it at the end, without changing info part of any node.
14. Write a program to remove the last node of the list and insert it in the beginning, without changing info part of any node.
15. Write a program to move a node n positions forward in a single linked list.
16. Write a function to delete a node from a single linked list. The only information we have is a pointer to the node that has to be deleted.
17. Write functions to insert a node just before and just after a node pointed to by a pointer p, without using the pointer start.

18. What is wrong in the following code that attempts to free all the nodes of a single linked list.

```
p=start;
while(p!=NULL)
{
        free(p);
        p=p->link;
}
```

Write a function `Destroy()` that frees all the nodes of a single linked list.

19. Write a function to remove duplicates from a sorted single linked list.

20. Write a function to remove duplicates from an unsorted single linked list.

21. Write a function to create a linked list that is intersection of two single linked lists, i.e. it contains only the elements which are common to both the lists.

22. Write a function to create a linked list that is union of two single linked lists, i.e. it contains all elements of both lists and if an element is repeated in both lists, then it is included only once.

23. Given a list L1, delete all the nodes having negative numbers in info part and insert them into list L2 and all the nodes having positive numbers into list L3. No new nodes should be allocated.

24. Given a linked list L1, create two linked lists one having the even numbers of L1 and the other having the odd numbers of L1. Don't change the list L1.

25. Write a function to delete alternate nodes(even numbered nodes) from a single linked list. For example if the list is `1->2->3->4->5->6->7` then the resulting list should be `1->3->5->7`.

26. Write a function to get the n^{th} node from the end of a single linked list, without counting the elements or reversing the list.

27. Write a function to find out whether a single linked is NULL terminated or contains a cycle/loop. If the list contains a cycle, find the length of the cycle and the length of the whole list. Find the node that causes the cycle i.e. the node at which the cycle starts. This node is pointed by two different nodes of the list. Remove the cycle from the list and make it NULL terminated.

28. Write a function to find out the middle node of a single linked list without counting all the elements of the list.

29. Write a function to split a single linked list into two halves.

30. Write a function to split a single linked list into two lists at a node containing the given information.

31. Write a function to split a single linked list into two lists such that the alternate nodes(even numbered nodes) go to a new list.

32. Write a function to combine the alternate nodes of two null terminated single linked lists. For example if the first list is `1->2->3->4` and the second list is `5->7->8->9` then after combining them the first list should be `1->5->2->7->3->8->4->9` and second list should be empty. If both lists are not of the same length, then the remaining nodes of the longer list are taken in the combined list. For example if the first list is `1->2->3->4` and the second list is `5->7` then the combined list should be `1->5->2->7->3->4`.

33. Suppose there are two null terminated single linked lists which merge at a given point and share all the nodes after that merge point (Y shaped lists). Write a function to find the merge point (intersection point).

34. Create a double linked list in which info part of each node contains a digit of a given number. The digits should be stored in reverse order, i.e. the least significant digit should be stored in the first node and the most significant digit in the last node. If the number is 5468132 then the linked list should be `2->3->1->8->6->4->5`. Write a function to add two numbers represented by linked lists.

35. Modify the program in the previous problem so that now in each node of the list we can store 4 digits of the given number. For example if the number is 23156782913287 then the linked list would be `3287->8291->1567->23`.

36. Write a function to find whether a linked list is palindrome or not.

37. Construct a linked list in which each node has the following information about a student - rollno, name, marks in 3 subjects. Enter records of different students in list. Traverse this list and calculate the total marks, percentage of each student. Count the number of students who scored passing marks(above 40 percent).

38. Modify the previous program so that now the names are inserted in alphabetical order in the list. Make this program menu driven with the following menus-

(i) Create list (ii) Insert (iii) Delete (iv) Modify (v) Display record (vi) Display result
Delete menu should have the facility of entering name of a student and the record of that student should be deleted. Display record menu should ask for the roll no of a student and display all information. Display result should display the number of students who have passed. Modify menu has the facility of modifying a record given the roll number.