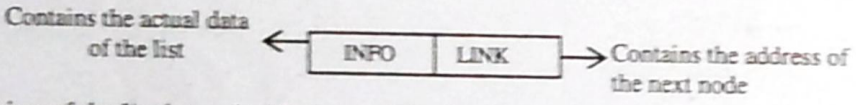
Linked Lists

3

List is a collection of similar type of elements. There are two ways of maintaining a list in memory. The f way is to store the elements of the list in an array, but arrays have some restrictions and disadvantages. T second way of maintaining a list in memory is through linked list. Now let us study what a linked list is a 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 conta 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 s 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 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 n node to point to, so it is made NULL. Hence if we reach a node whose link part has NULL value then we kn 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 represent it through linked list.

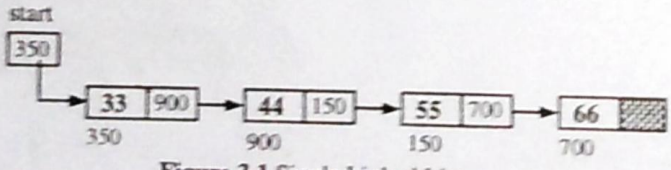


Figure 3.1 Single Linked List

From the figure it is clear that the info part contains the integer values and the link part contains the addr of the next node. The address of the first node is contained in the pointer `start` and the link part of last node 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 adjac to each other in the memory. The following figure shows the position of nodes of the above list in memory.

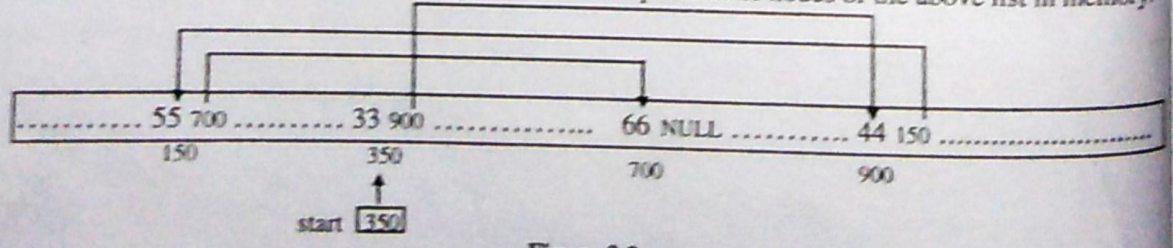


Figure 3.2

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.

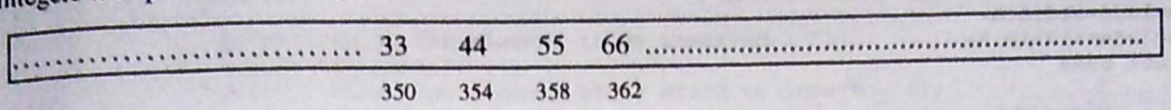


Figure 3.3

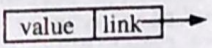
Here the elements are stored in consecutive memory locations in the same order as they appear in the list. So array is sequential representation of list while linked list is the linked representation of list. In a linked list, nodes are not stored contiguously as in array, but they are linked through 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-

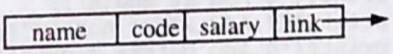
```
struct node{
            type1 member1;
            type2 member2;
            ..........
            ..........
            struct node *link;  /*Pointer to next node*/
        };
```

Let us see some examples of nodes-

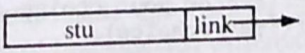
```
struct node{
            int value;
            struct node *link;
        };
```




```
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.

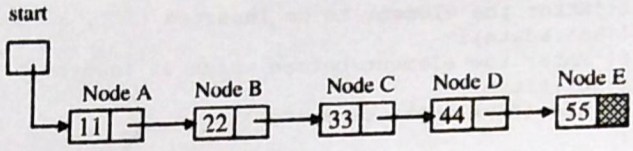


Figure 3.4

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.

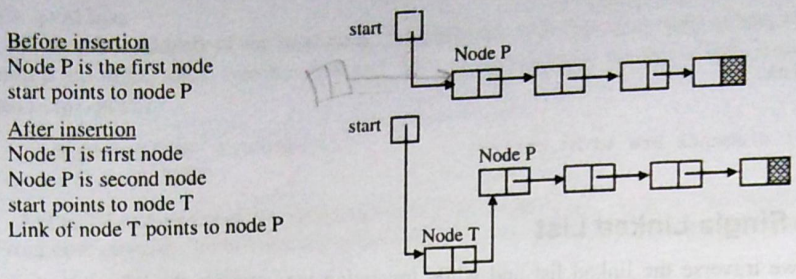


Figure 3.5 Insertion at the beginning of the list

(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

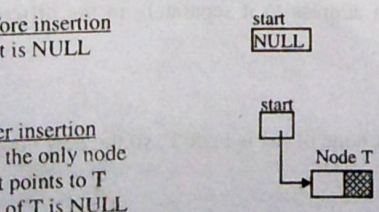


Figure 3.6 Insertion in an empty list

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.

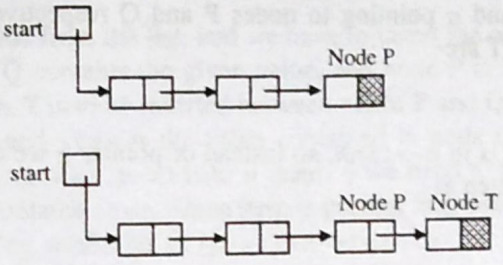


Figure 3.7 Insertion at the end of the list

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.

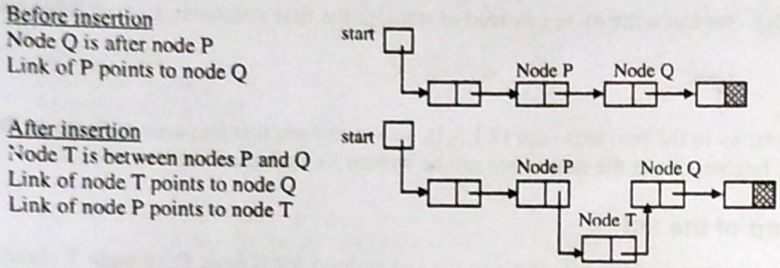


Figure 3.8 Insertion in between the list nodes

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

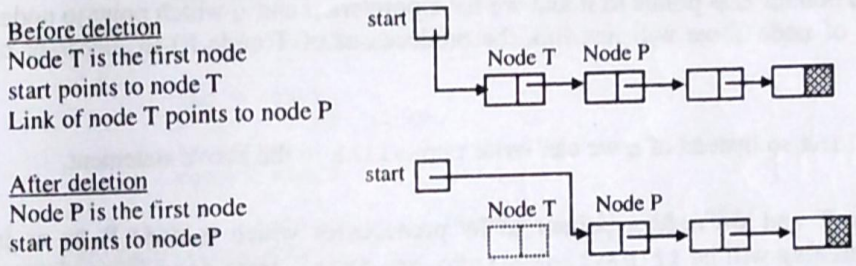


Figure 3.9 Deletion of first node

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.

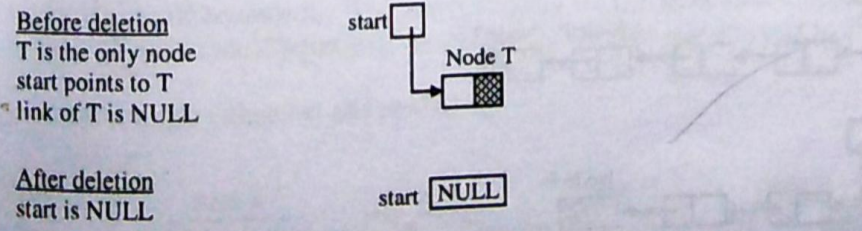


Figure 3.10 Deletion of the only node

```
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

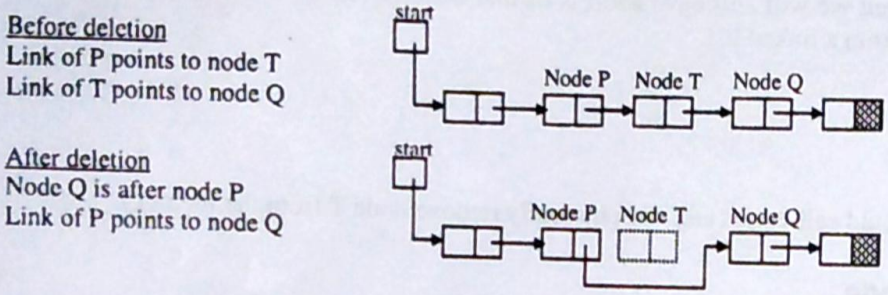


Figure 3.11 Deletion in between the list nodes

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

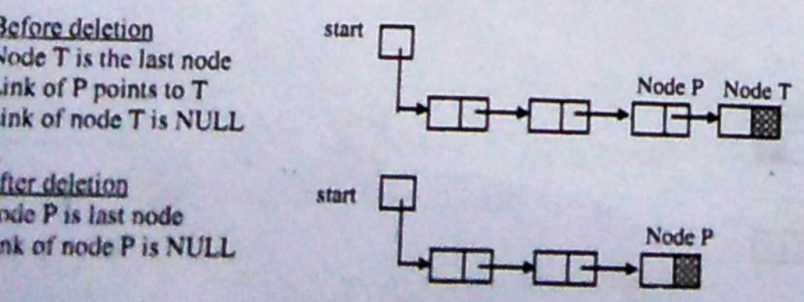


Figure 3.12 Deletion at the end of the list

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

(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-

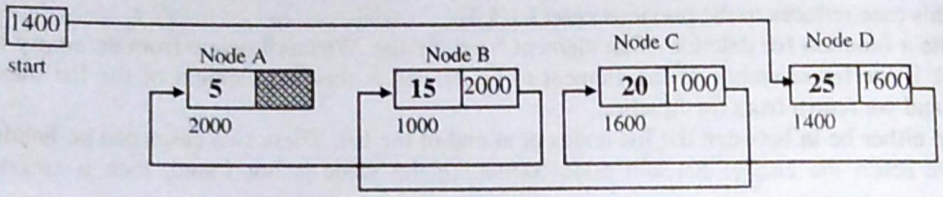


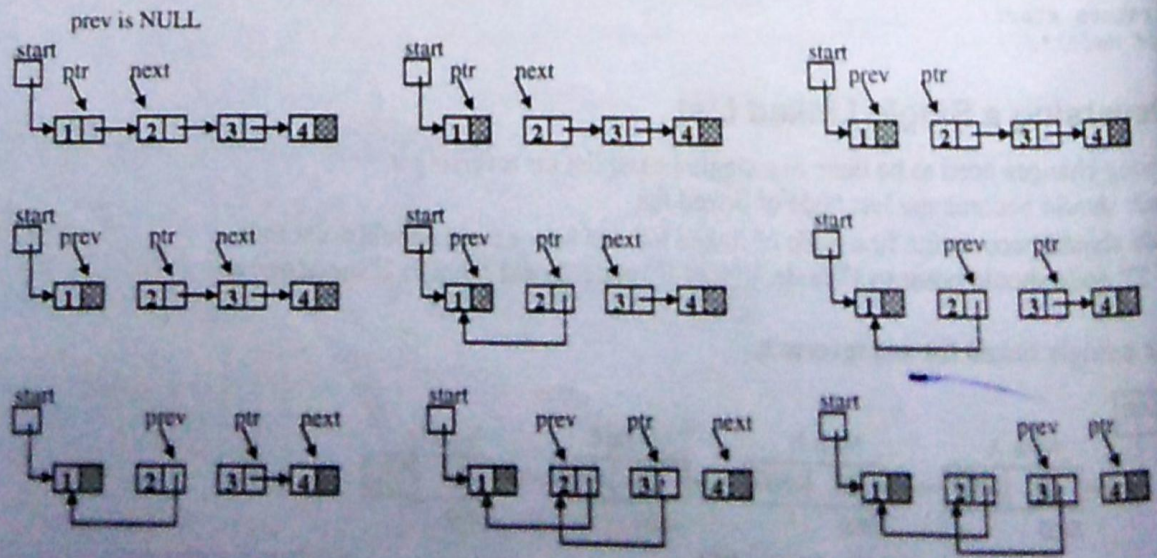
Figure 3.14

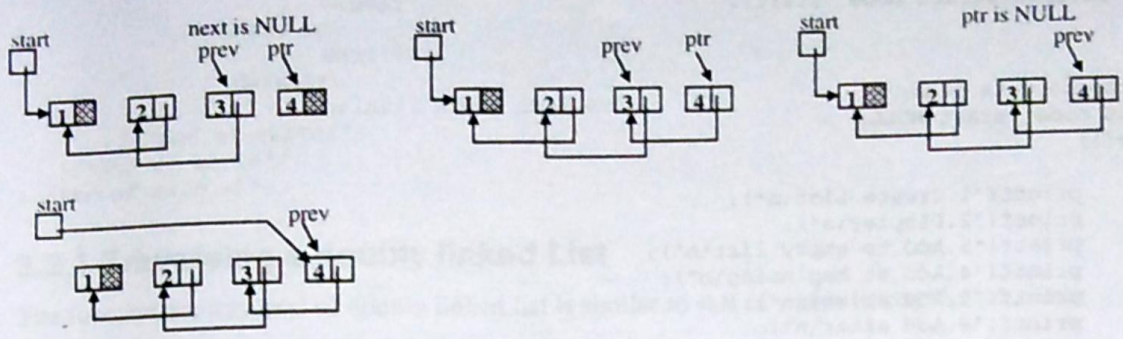
(i) Node D is the first node so `start` points to it.
(ii) Node A is the last node so its link is `NULL`.
(iii) Link of D points to C, link of C points to B and link of B points to A.

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 the link of pointer `ptr` is stored in pointer `next` and after that the link of `ptr` is changed so that it points to 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.

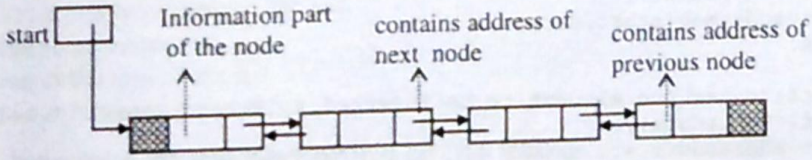


Figure 3.15 Doubly linked list

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

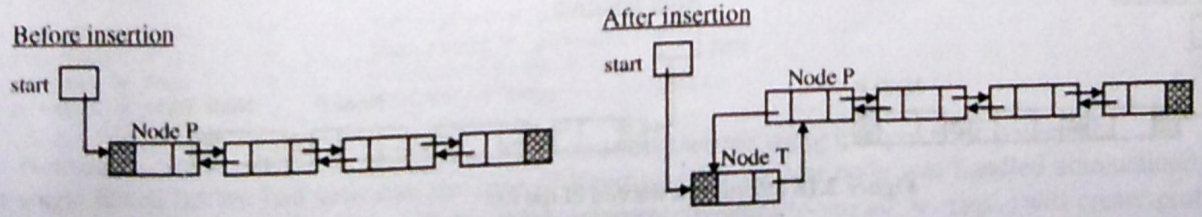


Figure 3.16 Insertion at the beginning of the list

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

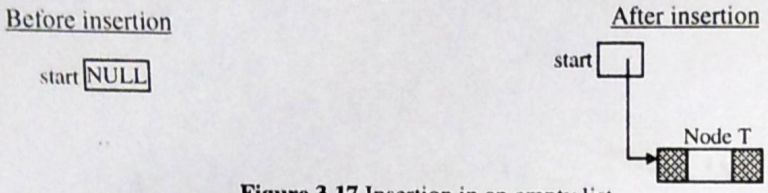


Figure 3.17 Insertion in an empty list

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

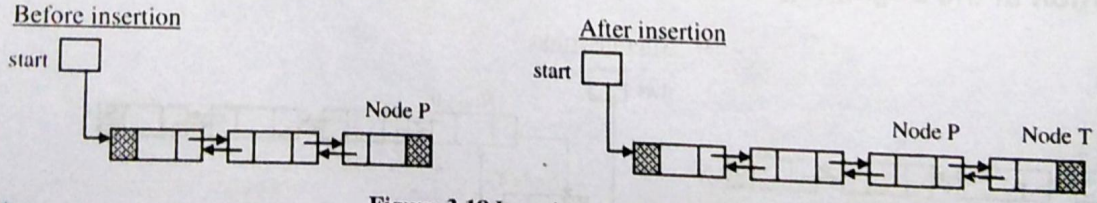


Figure 3.18 Insertion at the end of the list

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

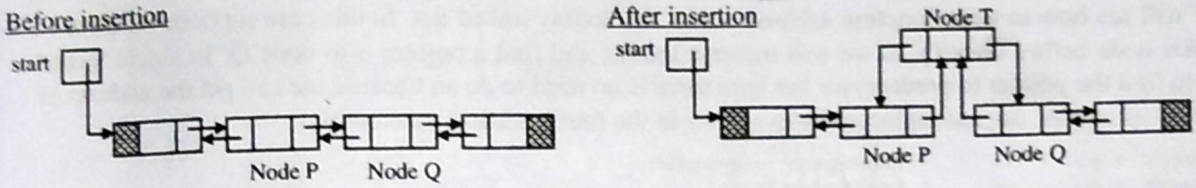


Figure 3.19 Insertion in between the nodes

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;
```