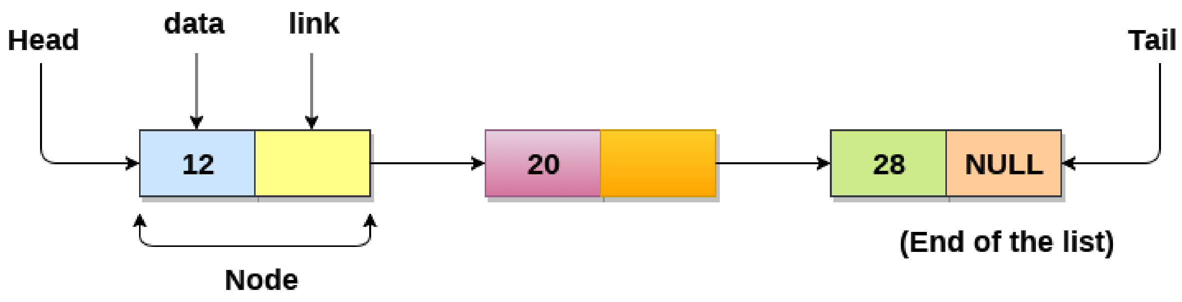
LINKED LISTS

A linked list is a linear data structure that includes a series of connected nodes. Here, each node store the **data** and the **address** of the next node

In linked list Each node is separated into two different parts:

- The first part holds the information of the element
- The second piece contains the address of the next node (link / next-pointer field)



Uses of Linked List

- The list is not required to be contiguously present in the memory.
- The node can reside anywhere in the memory and linked together to make a list.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

- Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory.
- However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains the following limitations:

- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process.

- It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory.
- Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array.

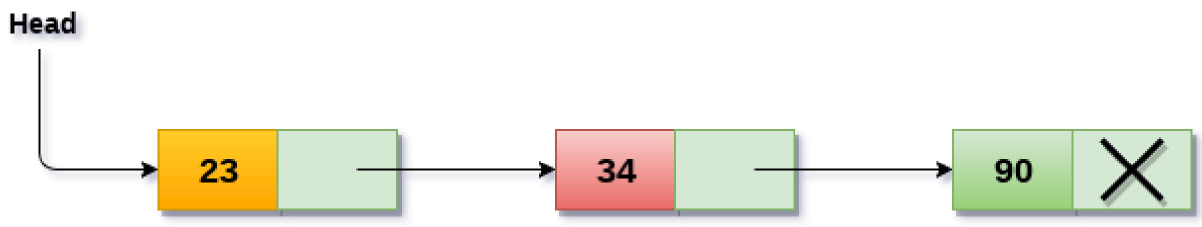
Advantages of linked list

- It allocates the memory dynamically.
- All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- Sizing is no longer a problem since we do not need to define its size at the time of declaration.
- List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

- Singly linked list can be defined as the collection of ordered set of elements.
- The number of elements may vary according to need of the program.
- A node in the singly linked list consist of two parts: data part and link part.
- Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction.
- In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links.

- The data part of every node contains the marks obtained by the student in the different subject.
- The last node in the list is identified by the null pointer which is present in the address part of the last node.
- We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list.

A list of all such operations is given below.

Insertion

- The insertion into a singly linked list can be performed at different positions.
- Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new the Head of the list.
2	Insertion at end list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the can be inserted as the last one.
3	Insertion after specified node	It involves insertion after the specified node of the linke

		We need to skip the desired number of nodes in order t the node after which the new node will be inserted. .

Deletion and Traversing

- The Deletion of a node from a singly linked list can be performed at different positions.
- Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of t
2	Deletion at end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different l implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node list. we need to skip the desired number of nodes to rea node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list a once in order to perform some specific operation on example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list w given element. If the element is found on any of the lo then location of that element is returned otherwise returned. .

Example:

```
#include<stdlib.h>
#include <stdio.h>
void create();
void display();
void insert_begin();
void insert_end();
void insert_pos();
void delete_begin();
void delete_end();
void delete_pos();
struct node
{
        int info;
        struct node *next;
};
struct node *start=NULL;
int main()
{
        int choice;
        while(1)
          {
              printf("\n                MENU  ");
              printf("\n 1.Create ");
              printf("\n 2.Display");
              printf("\n 3.Insert at the beginning ");
              printf("\n 4.Insert at the end");
              printf("\n 5.Insert at specified position ");
              printf("\n 6.Delete from beginning ");
              printf("\n 7.Delete from the end");
              printf("\n 8.Delete from specified position");
              printf("\n 9.Exit ");
              printf("\n--------------------------------------");
              printf("\n Enter your choice:");
              scanf("%d",&choice);
              switch(choice)
              {
                      case 1:
                        create();
                        break;
                case 2:
                        display();
                        break;
                case 3:
                        insert_begin();
                        break;
                case 4:
                        insert_end();
                        break;
                case 5:
                        insert_pos();
                        break;
                case 6:
                        delete_begin();
                        break;
                case 7:
                        delete_end();
                        break;
                case 8:
                        delete_pos();
                        break;

                case 9:
                        exit(0);
                        break;

                default:
                        printf("n Wrong Choice:");
                        break;
            }
        }
        return 0;
}
void create()
{
        struct node *temp,*ptr;
        temp=(struct node *)malloc(sizeof(struct node));
        if(temp==NULL)
        {
                printf("\n Out of Memory Space:");
                exit(0);
        }
        printf("\n Enter the data value for the node:");
        scanf("%d",&temp->info);
        temp->next=NULL;
        if(start==NULL)
        {
                start=temp;
        }
        else
        {
                ptr=start;
                while(ptr->next!=NULL)
                {
                        ptr=ptr->next;
                }
                ptr->next=temp;
        }
}
void display()
{
        struct node *ptr;
        if(start==NULL)
        {
                printf("\n List is empty:");
                return;
        }
        else
        {
                ptr=start;
                printf("\n The List elements are:");
                while(ptr!=NULL)
                {
                        printf("%d\t",ptr->info );
                        ptr=ptr->next ;
                }
        }
}
```

```
void insert_begin()
{
        struct node *temp;
        temp=(struct node *)malloc(sizeof(struct node));
        if(temp==NULL)
        {
                printf("\n Out of Memory Space:n");
                return;
        }
        printf("\n Enter the data value for the node:" );
        scanf("%d",&temp->info);
        temp->next =NULL;
        if(start==NULL)
        {
                start=temp;
        }
        else
        {
                temp->next=start;
                start=temp;
        }
}
void insert_end()
{
        struct node *temp,*ptr;
        temp=(struct node *)malloc(sizeof(struct node));
        if(temp==NULL)
        {
                printf("\n Out of Memory Space:");
                return;
        }
        printf("\n Enter the data value for the node:" );
        scanf("%d",&temp->info );
        temp->next =NULL;
        if(start==NULL)
        {
                start=temp;
        }
        else
        {
            ptr=start;
            while(ptr->next !=NULL)
            {
                  ptr=ptr->next ;
            }
            ptr->next =temp;
      }
}
void insert_pos()
{
      struct node *ptr,*temp;
      int i,pos;
      temp=(struct node *)malloc(sizeof(struct node));
      if(temp==NULL)
      {
            printf("\n Out of Memory Space:");
            return;
      }
      printf("\n the position for the new node to be inserted:");
      scanf("%d",&pos);
      printf("\n Enter the data value of the node:");
      scanf("%d",&temp->info) ;

      temp->next=NULL;
      if(pos==0)
      {
            temp->next=start;
            start=temp;
      }
      else
      {
            for(i=0,ptr=start;i<pos-1;i++)
              {
                  ptr=ptr->next;
                  if(ptr==NULL)
                  {
                        printf("\n Position not found:[Handle with care]");
                        return;
                  }
            }
```

```
		temp->next =ptr->next ;
		ptr->next=temp;
	}
}
void delete_begin()
{
	struct node *ptr;
	if(start==NULL)
	{
		printf("\n List is Empty:");
		return;
	}
	else
	{
		ptr=start;
		start=start->next ;
		printf("\n The deleted element is :%d",ptr->info);
		free(ptr);
	}
}
void delete_end()
{
	struct node *temp,*ptr;
	if(start==NULL)
	{
		printf("\n List is Empty:");
		exit(0);
	}
	else if(start->next ==NULL)
	{
		ptr=start;
		start=NULL;
		printf("\n The deleted element is:%d",ptr->info);
		free(ptr);
	}
	else
	{
		ptr=start;
		while(ptr->next!=NULL)
		{
```

```
                        temp=ptr;
                        ptr=ptr->next;
                }
                temp->next=NULL;
                printf("\n The deleted element is:%d",ptr->info);
                free(ptr);
        }
}
void delete_pos()
{
        int i,pos;
        struct node *temp,*ptr;
        if(start==NULL)
        {
                printf("\n The List is Empty:");
                exit(0);
        }
        else
        {
                printf("\n Enter the position of the node to be deleted:");
                scanf("%d",&pos);
                if(pos==0)
                {
                        ptr=start;
                        start=start->next ;
                        printf("\n The deleted element is:%d",ptr->info  );
                        free(ptr);
                }
                else
                {
                        ptr=start;
                        for(i=0;i<pos;i++) { temp=ptr; ptr=ptr->next ;
                                if(ptr==NULL)
                                {
                                        printf("\n Position not Found:");
                                        return;
                                }
                        }
                        temp->next =ptr->next ;
                        printf("\n The deleted element is:%d",ptr->info );
```

```
                    free(ptr);
                }
        }
}
```

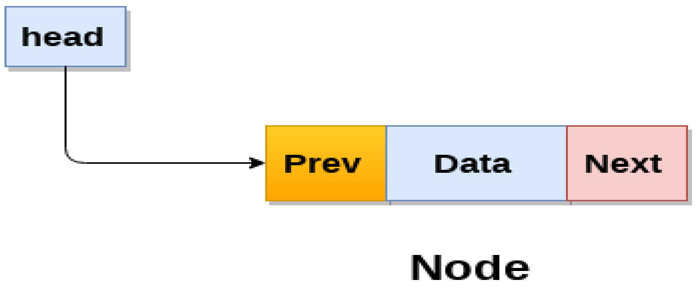
Output:

```
     MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:1

 Enter the data value for the node:100

                MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:1

 Enter the data value for the node:200
```

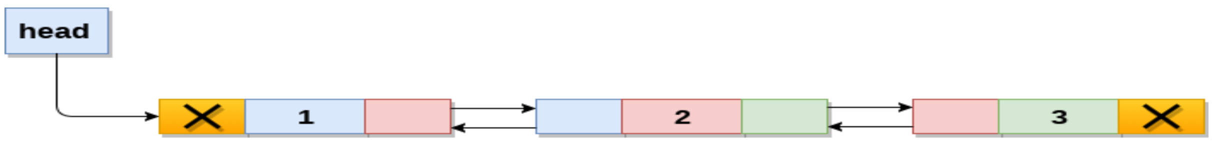
```
                MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:1

 Enter the data value for the node:300

                MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:2

 The List elements are:100      200     300
                MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:3

 Enter the data value for the node:50

             MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:2

 The List elements are:50      100     200     300
             MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:4
 Enter the data value for the node:500

              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:2

 The List elements are:50      100     200     300     500
              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:5

 the position for the new node to be inserted:4

 Enter the data value of the node:400

              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:2

 The List elements are:50       100     200     300     400     500
              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:7

 The deleted element is:500
              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:2

 The List elements are:50       100     200     300     400
              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:8

 Enter the position of the node to be deleted:3

 The deleted element is:300
              MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
-------------------------------------
 Enter your choice:2
 The List elements are:50      100     200     400
                        MENU
 1.Create
 2.Display
 3.Insert at the beginning
 4.Insert at the end
 5.Insert at specified position
 6.Delete from beginning
 7.Delete from the end
 8.Delete from specified position
 9.Exit
--------------------------------------
 Enter your choice:Killed


...Program finished with exit code 9
Press ENTER to exit console.
```

Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).
- A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

Structure of a node in doubly linked list can be given as :

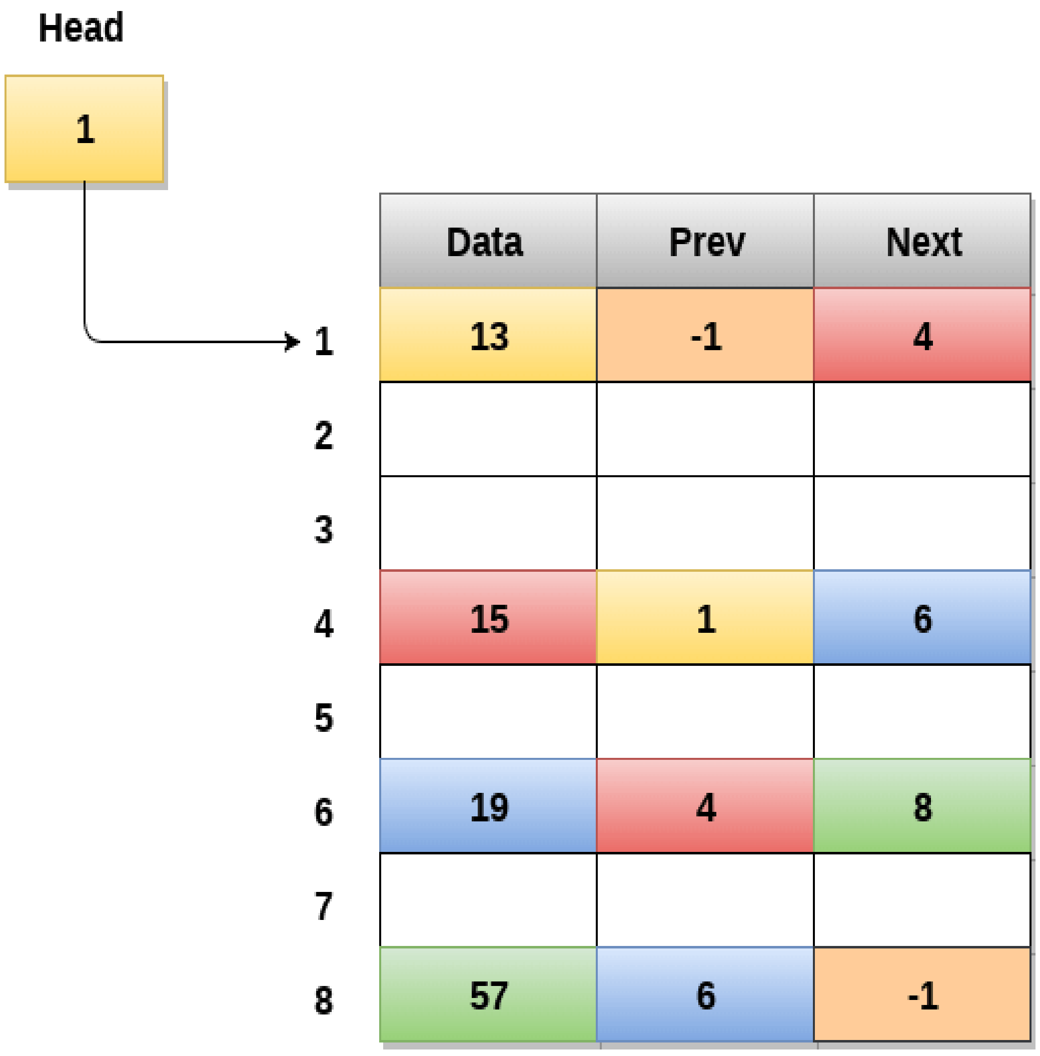
```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

- The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.
- In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes.
- However, doubly linked list overcome this limitation of singly linked list.
- Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

- Memory Representation of a doubly linked list is shown in the following image.
- Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion.
- However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
- In the following image, the first element of the list that is i.e. 13 stored at address 1.
- The head pointer points to the starting address 1.
- Since this is the first element being added to the list therefore the **prev** of the list **contains** null.

- The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
- We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Operations on doubly linked list

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the spe node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node at given data	Removing the node which is present just after the containing the given data.
7	Searching	Comparing each node data with the item searched and return the location of the item in t if the item found else return null.
8	Traversing	Visiting each node of the list at least once in or perform some specific operation like searching, so display, etc.

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1** - Create a **newNode** with given value and **newNode** → **previous** as **NULL**.

- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- Step 4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step3 - If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- Step 6 - Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- Step4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- Step 5 - Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.

- Step7 - Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- Step5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- Step6 - If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- Step 7 - Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- Step 5 - If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- Step 10 - If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- Step 11 - If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- Step 12 - If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Display **'NULL <--- '**.
- Step 5 - Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the last node
- Step 6 - Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Types of Linked List

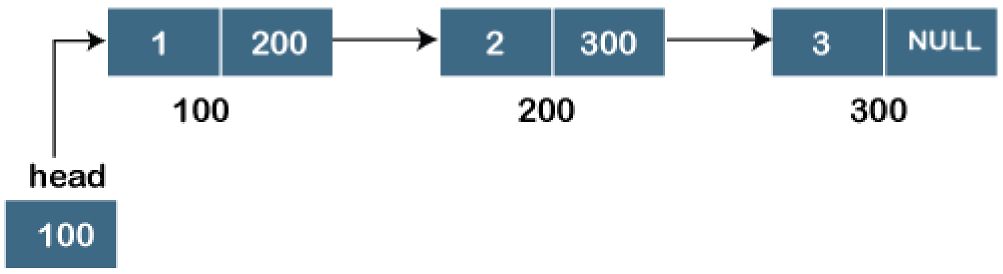
The following are the types of linked lists

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

Singly Linked list

- It is the commonly used linked list in programs.
- If we are talking about the linked list, it means it is a singly linked list.
- The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node.
- The address part in a node is also known as a **pointer**.
- Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively.

The representation of three nodes as a linked list is shown in the below figure:



- We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively.
- The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node.
- The pointer that holds the address of the initial node is known as a ***head pointer***.
- The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link.
- In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

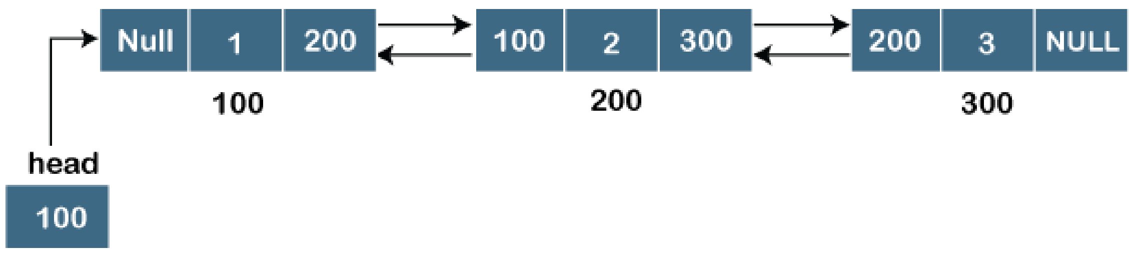
```
struct node
{
  int data;
  struct node *next;
}
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

- As the name suggests, the doubly linked list contains two pointers.
- We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part.
- In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

- Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively.
- The representation of these nodes in a doubly-linked list is shown below:



- As we can observe in the above figure, the node in a doubly-linked list has two address parts;
- one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***.
- The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

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

In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type.

The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next and prev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

Circular linked list

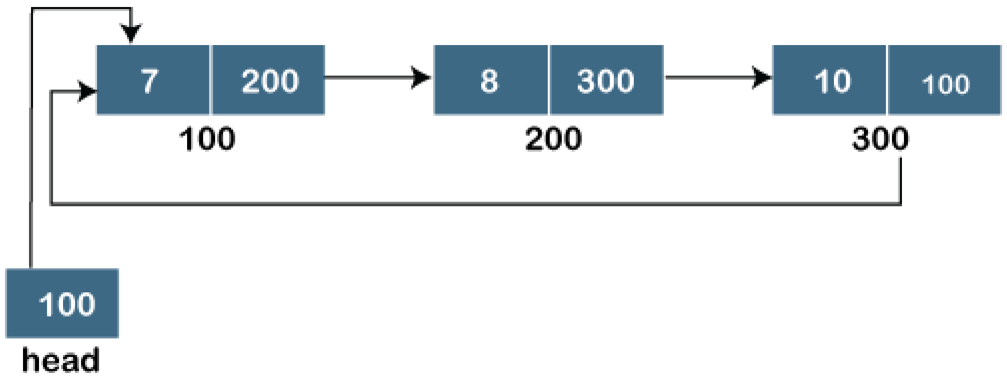
- A circular linked list is a variation of a singly linked list.
- The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value.
- On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address.
- The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward.

The diagrammatic representation of the circular linked list is shown below:

```
struct node
{
  int data;
  struct node *next;
}
```

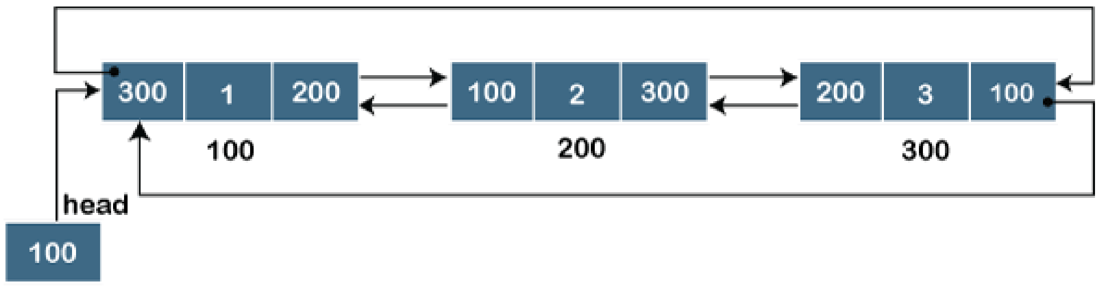
A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node.

The representation of the circular linked list will be similar to the singly linked list, as shown below:



Doubly Circular linked list

The doubly circular linked list has the features of both the ***circular linked list*** and ***doubly linked list***.



- The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle.
- It is a doubly linked list also because each node holds the address of the previous node also.
- The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node.
- As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.