UNIT-II

Linked Lists: Singly linked lists: representation and operations, doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

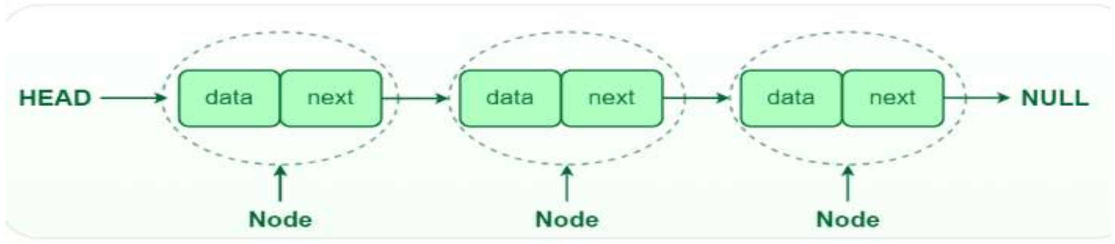
Linked List

Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

Linked List is a very commonly used linear data structure which consists of set of **nodes** in a sequence.

Each node has two fields known as data and link. **Data** field stores actual data and **link** Contains address of next node to point to the next node.

Linked List is a linear data structure, in which elements are not stored at a contiguous location; rather they are linked using pointers.



Node Structure: A node in a linked list typically consists of two components:
Data: It holds the actual value or data associated with the node.
Next Pointer: It stores the memory address (reference) of the next node in the sequence.
Head: The linked list is accessed through the head node, which points to the first node in the list.

Tail: The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List(Single or Double)

I. Singly Linked List

1. Introduction and Representation

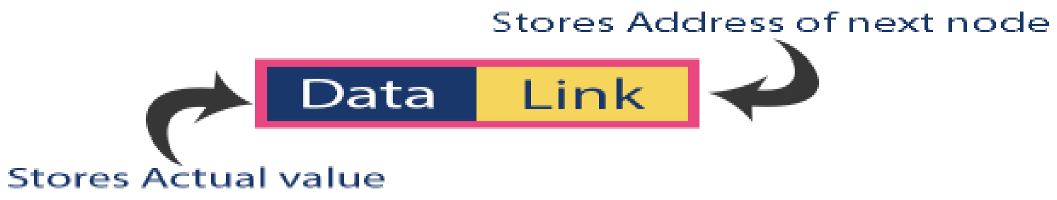
Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node".

Every "Node" containstwo fields, data and next.

The data field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...



In a single linked list, the address of the first node is always stored in a reference node known as "head"(Some times it is also known as "front").
Note:-- Always next part (reference part) of the last node must be NULL.

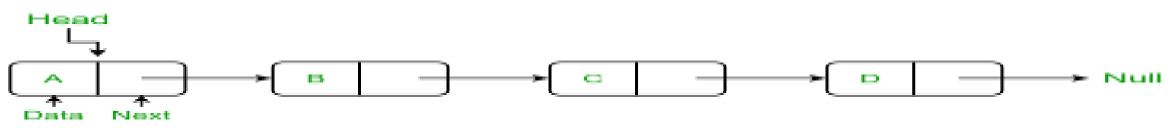
A node is represented as

```
struct node

{

        int data;

        struct node *link;

};
```

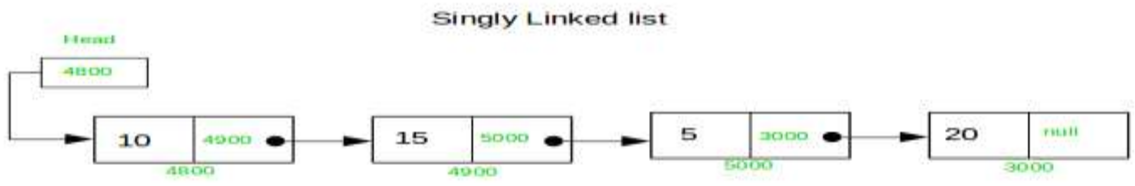
int data refers to the data part of a node and struct node *link is a pointer refers the next node in the list.

In a linked list, every node contains a pointer to another node which is of same type, called as “Self referential data type”, and structure is self referential structure.

Example 1:



Example 2:



Characteristics of a Singly Linked List:

- Each node holds a single value and a reference to the next node in the list.
- The list has a head, which is a reference to the first node in the list, and a tail, which is a reference to the last node in the list.
- The nodes are not stored in a contiguous block of memory, but instead, each node holds the address of the next node in the list.
- Accessing elements in a singly linked list requires traversing the list from the head to the desired node, as there is no direct access to a specific node in memory.

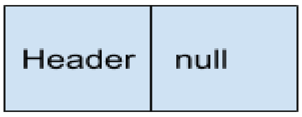
2. Operations of Singly Linked List

In a single linked list we perform the following operations...

- Insertion
- Deletion
- Display

Algorithm to create a node:

```
Createnode()

Begin
1.Allocate memory for header node.
   new_node = (struct node *) malloc(sizeof(struct node));
2. Verify the memory allocation.
         if(h==NULL) then
                  Printf("Memory not Allocated")
                  Return;
3.Allocate the next of header as NULL.
          h→link=NULL;
 4.Return h.
End.
```



2.1 Insertion operation

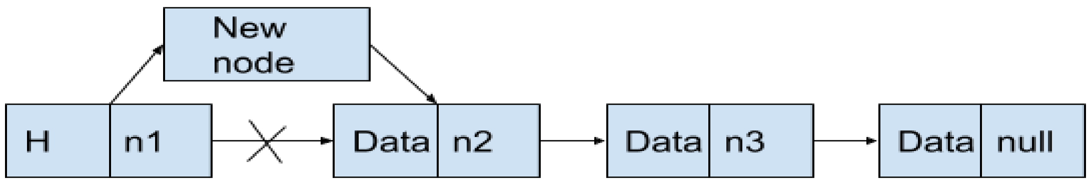
In a single linked list, the insertion operation can be performed in three ways. They are asfollows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

2.1.1 Insertion at Beginning of the list



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

Step 1: Create a new node with given value.

Step 2: Check whether list is Empty (head == NULL)

Step3: If it is Empty then, create head node.

```
new_node = (struct node *) malloc(sizeof(struct node));
 new_node →link  = NULL
 head = new_node.
```

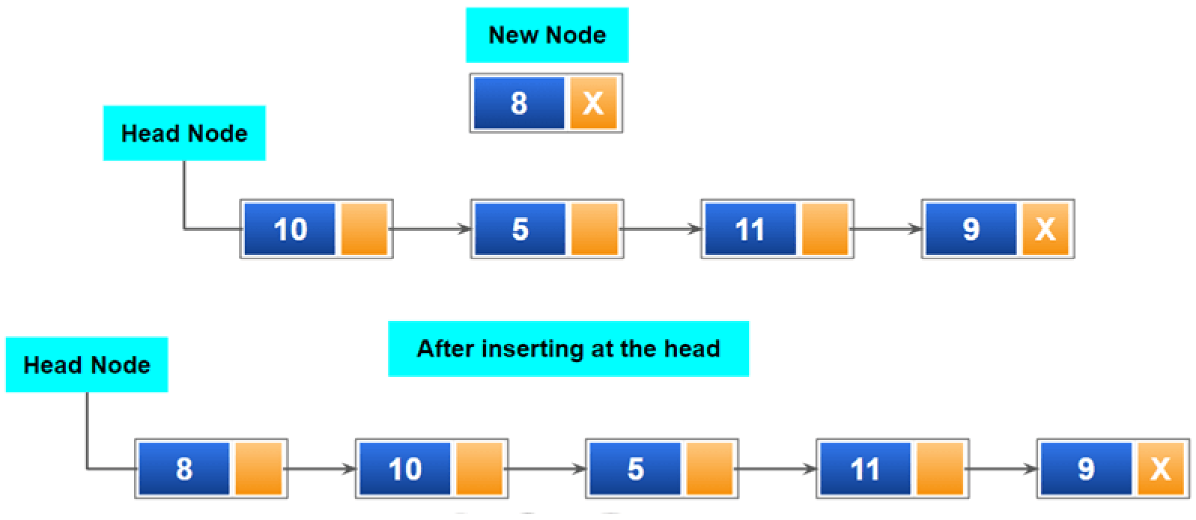
Step4: If it is Not Empty then,

Create new node.

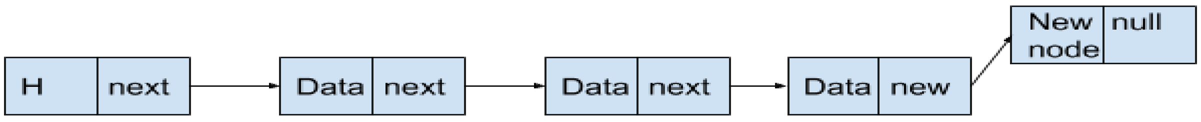
```
new_node = (struct node *) malloc(sizeof(struct node));
 new_node ->data=x;
 new_node →link  = head
 head = new_node.
```

Step5: Return h.

Example:



2.1.2 Inserting At End of the list



Step 1: Create a new node with given value.

Step 2: Check whether list is Empty (head == NULL)

Step3: If it is Empty then, create head node.

new_node = (struct node *) malloc(sizeof(struct node));

new_node →link = NULL
head = new_node.

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 → link is equal to NULL).

Step 6 –

Create new node.

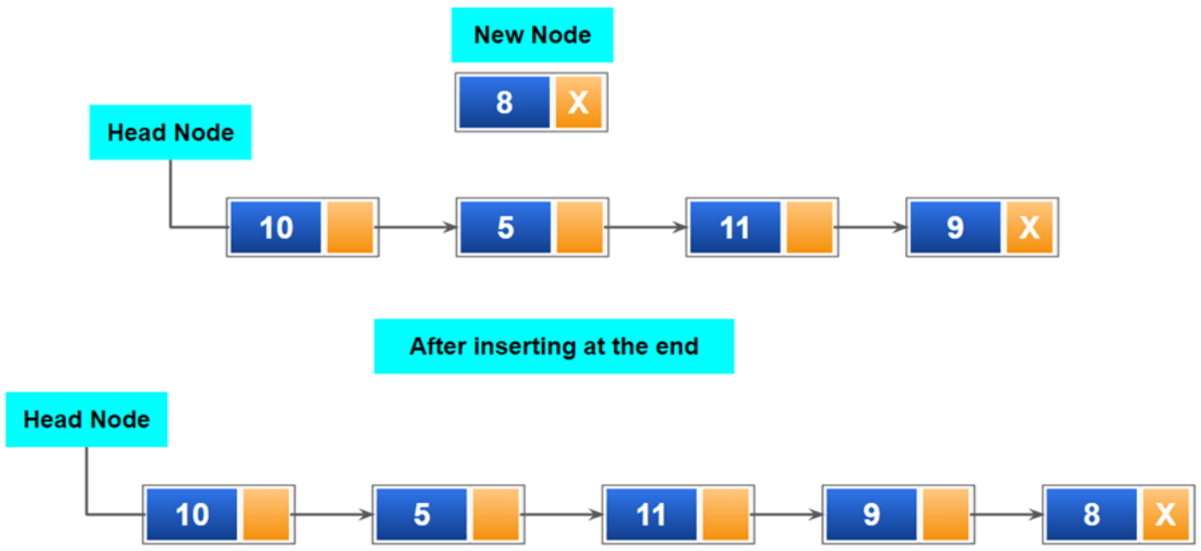
new_node = (struct node *) malloc(sizeof(struct node));

new_node ->data=x;

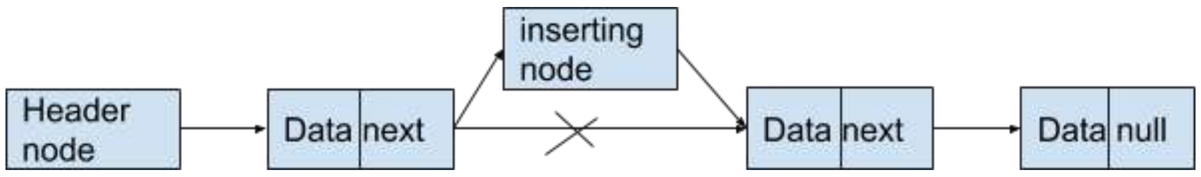
temp → link= new_node.

new_node →link = NULL

Example:



2.1.3 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 single 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, set new_node →link   = **NULL** and **head** = new_node.

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 node after which we want to insert the new_node (until **temp** → **data** is equal to **location**, here location is the node value after which we want to insert the new_node).

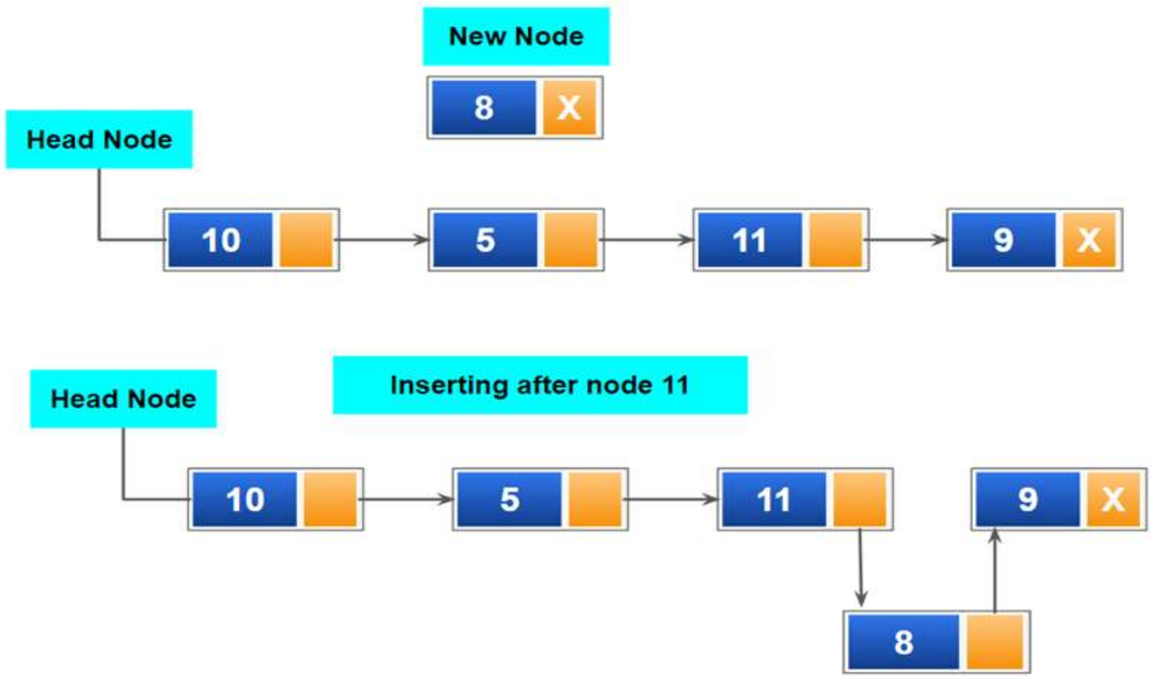
Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

Step 7 - Finally, Set

new_node →link = temp → next

temp → next = new_node

Example:



2.2 Deletion operation

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

2.2.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single 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.

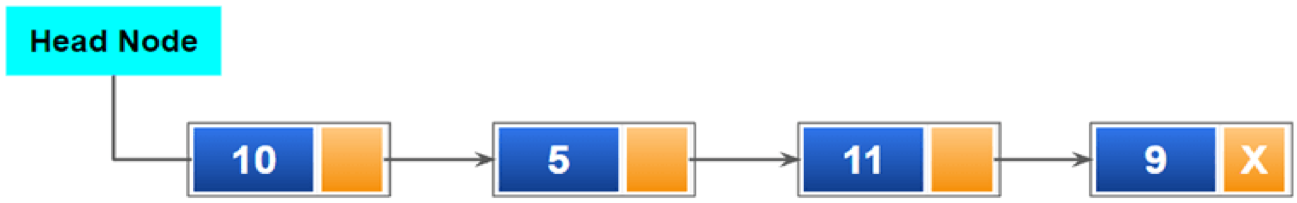
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list is having only one node (**temp** → **next** == **NULL**)

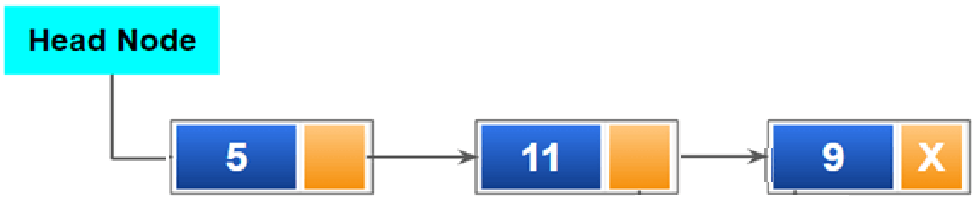
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

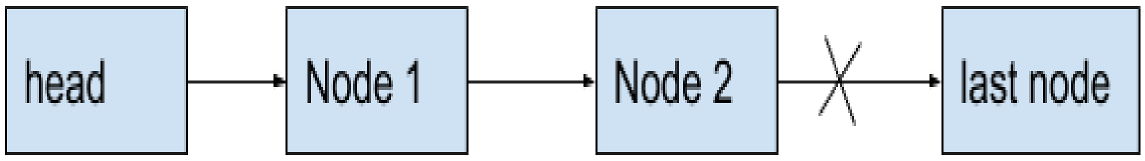
Example:



After deleting at head



2.2.2 Deleting from End of the list



We can use the following steps to delete a node from end of the single 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.

Steps 3 - If it is Not **Empty then, define** two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.

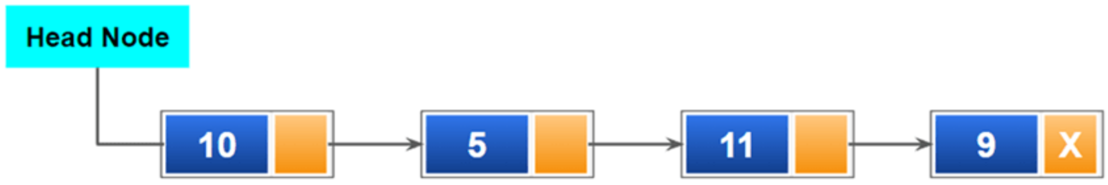
Step 4 - Check whether list has only one Node (**temp1 → next** == **NULL**)

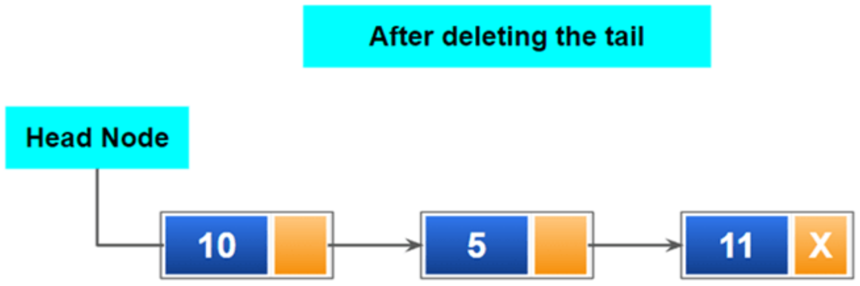
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function.

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Example:





2.2.3Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single 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 two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next 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 function.

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 to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

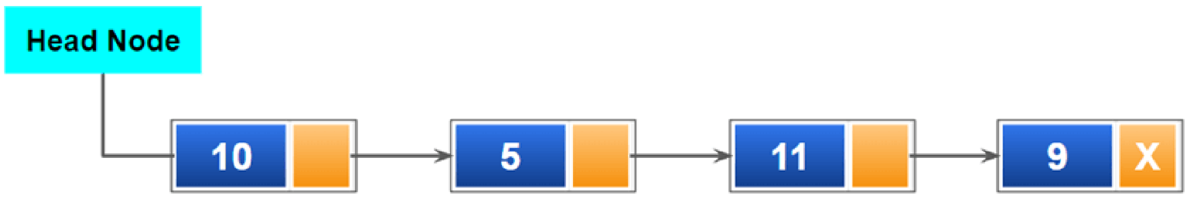
Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

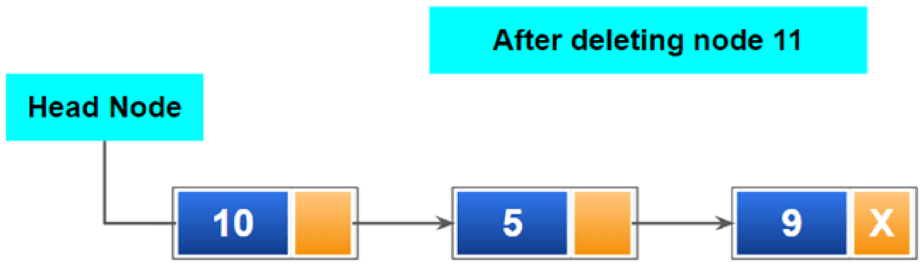
Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1** → **next** == NULL).

Step 11 - If **temp1** is last node then set **temp2** → **next** = **NULL** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2** → **next** = **temp1** → **next** and delete **temp1** (**free(temp1)**).

Example:





3.Displaying a Single Linked List

We can use the following steps to display the elements of a single 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: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5: Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Program

```
#include<stdlib.h>

#include<stdio.h>

struct node
```

```
{
        int data;
        struct node *link;
};
struct node *head,*last,*temp,*new_node;
void create();
void insert_begin();
void insert_end();
void insert_desired();
void delete_begin();
void delete_end();
void delete_desired();
void display();
void search();
void main()
{
        int choice;
        clrscr();
        while(1)
        {
                printf("\n          MAIN MENU          ");
                printf("\n1.Create 2.Insert at begin 3.insert end 4.insert desired \n5.delete
begin 6. delete end 7.del des 8.Display \n9.search 10.Exit\n");
                printf("*******************************************");
                printf("\nEnter your choice:");
                scanf("%d",&choice);
                switch(choice)
        {
            case 1: create();
                break;
            case 2: insert_begin();
                break;
            case 3: insert_end();
                break;
            case 4: insert_desired();
                break;
            case 5: delete_begin();
                break;
            case 6: delete_end();
                break;
            case 7: delete_desired();
                break;
            case 8: display();
                break;
            case 9: search();
                break;
            case 10: exit(1);
                break;
            defaulf:printf("Invalid option");
                break;
        }
    }

}
```

```
void create()
{
	int i,n;
	printf("Enter the no of nodes:");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		new_node = (struct node *) malloc(sizeof(struct node));
		printf("Enter the data:");
		scanf("%d",&new_node->data);
		new_node->link = NULL;
		if(head==NULL)
		{
			head = last = new_node;
		}
		else
		{
			 last->link = new_node;
			 last = new_node;
		}
	}
}
void insert_begin()
{
	new_node = (struct node*)malloc(sizeof(struct node));
	printf("Enter the data:");
	scanf("%d",&new_node->data);
```

```
        new_node->link=NULL;
        if(head==NULL)
        {
                head=last=new_node;
        }
        else
        {
                new_node->link = head;
                head = new_node;
        }
        printf("\nNODE INSERTED SUCCESSFULLY");
 }
void insert_end()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->link = NULL;
        if(head == NULL)
        {
                head=last=new_node;
        }
        else
        {
                last->link = new_node;
                last = new_node;
        }
```

```
}

void insert_desired()
{
        int pos,i;
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->link = NULL;
        printf("\nEnter the position to insert:");
        scanf("%d",&pos);
        if(head == NULL)
        {
                head = last = new_node;
        }
        else
        {      temp = head;
                for(i=1;i<pos-1;i++)
                {
                    temp = temp->link;
                }
                new_node->link=temp->link;
                temp->link = new_node;
                printf("\nSUCCESFULLY INSERTED\n");
        }
        getch();
}
```

```
void delete_begin()
{
        if(head==NULL)
        {
                printf("\nThere are no elements in the List.");
        }
        else
        {
                temp = head;
                head = temp->link;
                free(temp);
                printf("\nSUCCESSFULLY DELETED");
        }
        getch();
}
void delete_end()
{
        temp=head;

        if(head==NULL)
        {
                printf("\n Cannot delete node, because list is empty");
        }

        else if(head == last)
        {
                printf("\nDeleted node is %d", last->data);
                head=last=NULL;
                free(temp);
        }
        else
        {
            printf("\nDeleted node is %d",last->data);
            while(temp->link != last)
            {
                        temp=temp->link;
            }
            free(last);
            temp->link = NULL;
            last=temp;
        }
}
void delete_desired()
{
        int pos,i;
        struct node *temp1;
        temp = head;
        printf("\nEnter the position of the node to delete:");
        scanf("%d",&pos);
        if(head == NULL)
        {
                printf("\nThere are no nodes to delete:");
        }
```

```
else
        {
                temp = head;
                for(i=1;i<pos-1;i++)
                {
                        temp=temp->link;
                        if(temp->link == NULL)
                        {
                                printf("\nCannot delete");
                                return;
                        }
                }
                printf("\nDeleted node is %d", temp->link->data);
                temp1=temp->link;
                temp->link = temp->link->link;
                free(temp1);
        }
}
void search()
{
        int key;
        printf("\nEnter the search element:");
        scanf("%d",&key);
        temp = head;
        if(head==NULL)
        {
                printf("\nList is EMPTY");
        }
        else
        {
                while(temp != NULL)
                {
                        if(temp->data == key)
                        {
                                printf("\nSearch element is found");
                                return;
                        }
                        temp=temp->link;
                }
                printf("\nSearch element is not found");
        }
}
void display()
{
        if(head==NULL)
        {
                printf("There are no elements in the LL");
        }
        else
        {      printf("\nElements in the list are:\n");
                temp = head;
                do
                {
                        printf("%d ",temp->data);
                        temp=temp->link;
                }while(temp != NULL);
        }
}
```

II.Double Linked List

1. Introduction and Representation

What is Double Linked List?

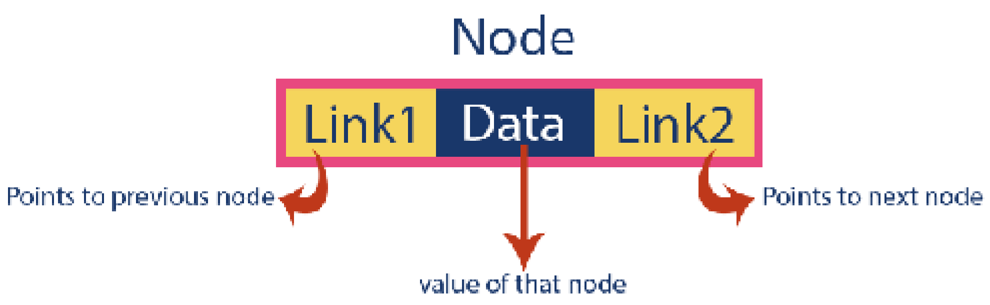
In a single linked list, every node has link to its next node in the sequence. So, we can traverse fromone node to other node only in one direction and we can not traverse back. We can solve this kind of problem by using double linked list.

Double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverseforward by using next field and can traverse backward by using previous field.

Every node in a double linked list contains three fields and they are shown in the following figure...



Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

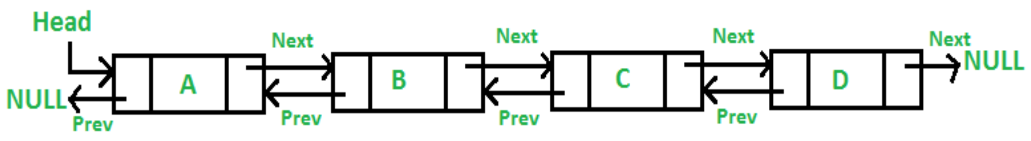
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link calledPrev.

Representation of Doubly Linked list

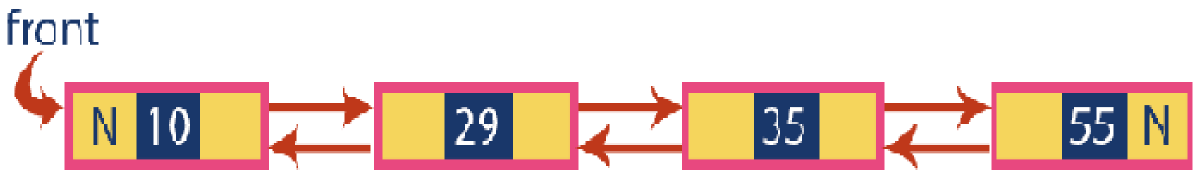
A node is represented as

```
struct node

{

        int data;

        struct node *prev;

        struct node *next;

};
```

Example 1:



Example 2:



2. Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

2.1 Insertion operation

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

2.1.1 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**.

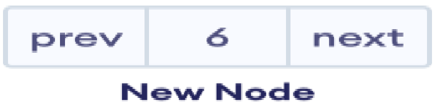
Step 2 **-** Check whether list is **Empty** (**head** == **NULL**)

Step 3 **-** If it is **Empty** then, assign **NULL** to **newNode** → **next** and **newNode** to **head**.

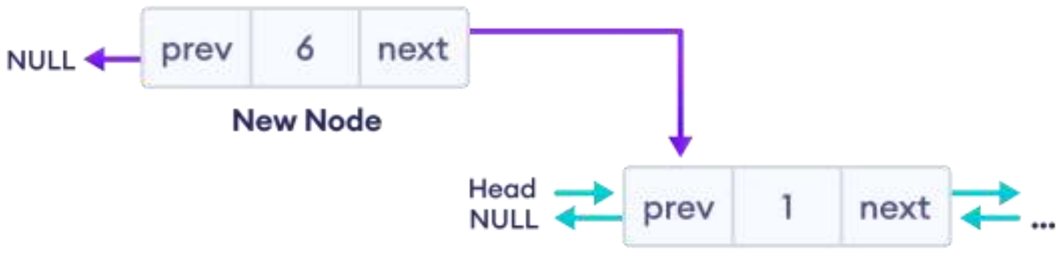
Step 4 **-** If it is **not Empty** then, assign **head** to **newNode** → **next** and **newNode** to **head**.

Example:

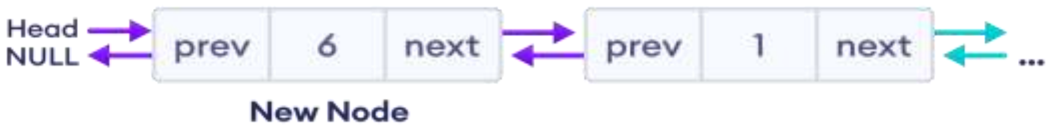
1.Create a new node



2. **Set prev and next pointers of new node**



3. Make new node as head node



2.1.2 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**)

Step 3 - If it is **Empty**, then assign **NULL to newNode** → **previous** and **newNode to head**.

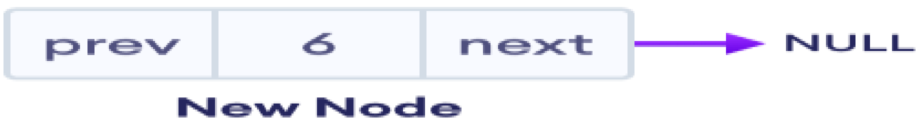
Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.

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**).

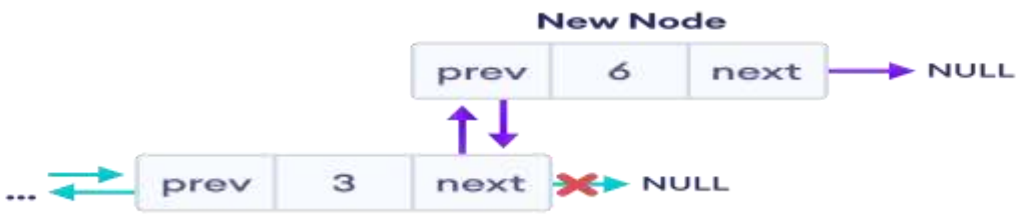
Step 6 - Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

Example:

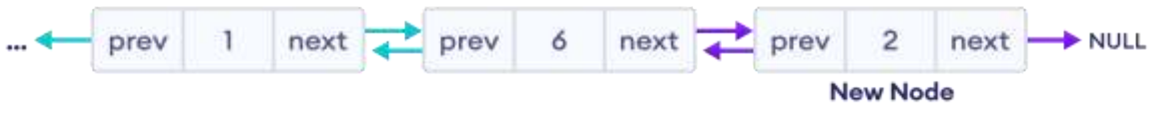
1.Create a new node



2.Set prev and next pointers of new node and the previous node



3. Make new node as tail node



2.1.3 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**.

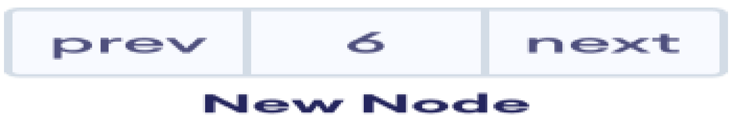
Step 4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.

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).

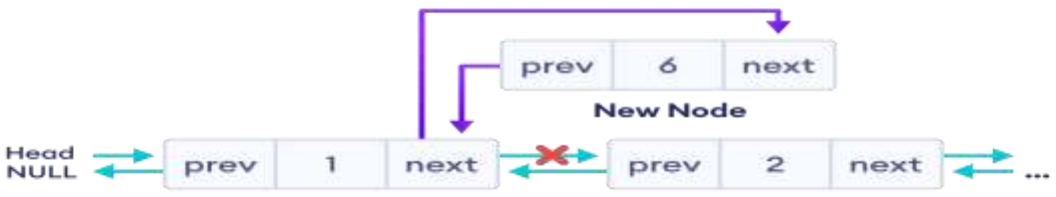
Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.

Step 7: Assign temp1 →next to temp2, newNode to temp1 →next, temp1 to newNode → previous, temp2 to newNode →nextand newNode to temp2 →previous

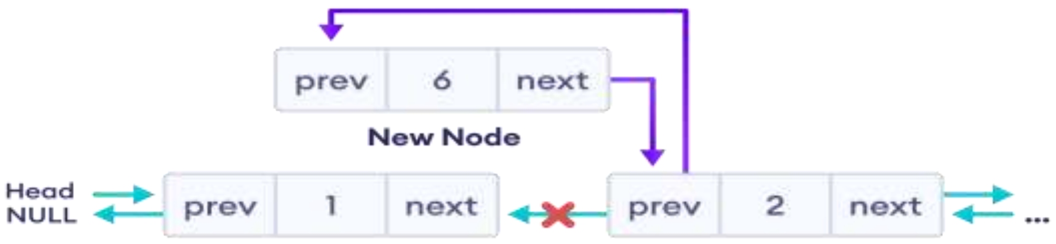
1. Create a new node



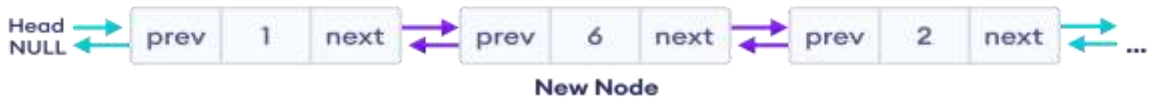
2. Set the next pointer of new node and previous node



3. **Set the prev pointer of new node and the next node**



4. The final doubly linked list is after this insertion is:



2.2 Deletion operation

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

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

2.2.1 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**.

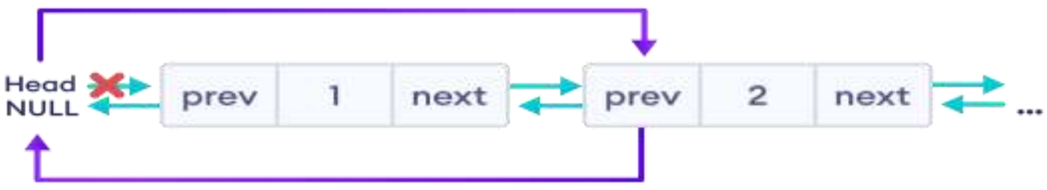
Step 4 - Check whether list is having only one node (**temp** → **previous** is equal to **temp** → **next**)

Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)

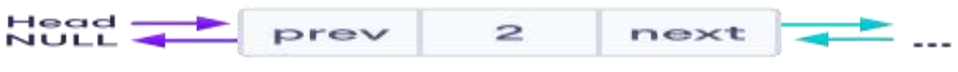
Step 6 - If it is **FALSE**, then assign **temp** → **next** to **head**, **NULL** to **head** → **previous** and delete **temp**.

Example:

If the node to be deleted (i.e. del_node) is at the beginning



Finally, free the memory of del_node. And, the linked will look like this



Free the space of the first node

2.2.2 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**)

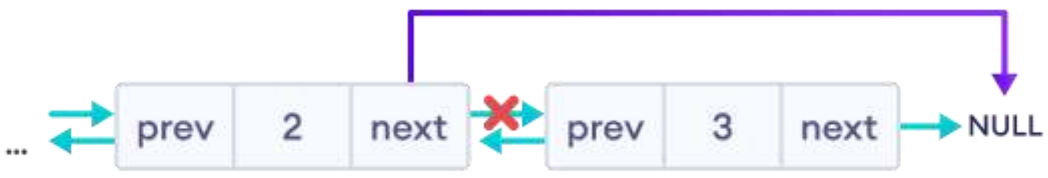
Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)

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**)

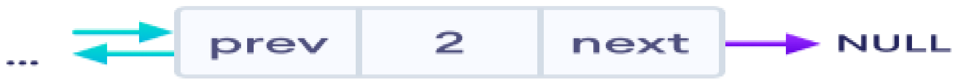
Step 7 - Assign **NULL** to **temp** → **previous** → **next** and delete **temp**.

Example:

In this case, we are deleting the last node with value **3** of the doubly linked list.



The final doubly linked list looks like this.



2.2.3 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**).

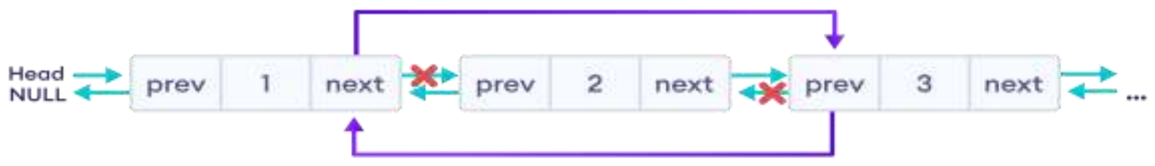
Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.

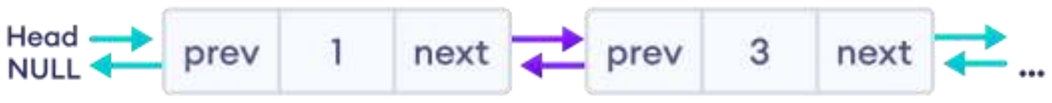
Step 10 - If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).

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)**).

Example:





3.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**).

Program

```
#include<stdlib.h>

#include<stdio.h>

struct node

{

        int data;

        struct node *prev;

        struct node *next;

};

struct node *head=NULL,*last=NULL,*temp,*new_node;

void create();

void insert_begin();

void insert_end();

void insert_desired();
void delete_begin();

void delete_end();

void delete_desired();

void display();

void search();

void main()

{

        int choice;

        clrscr();


        while(1)

        {

                printf("\n          MAIN MENU          ");

                printf("\n1.Create 2.Insert at begin 3.insert end 4.insert desired \n5.delete
begin 6. delete end 7.del des 8.Display \n9.search 10.Exit\n");

                printf("*********************************************");

                printf("\nEnter your choice:");

                scanf("%d",&choice);

                switch(choice)

                {

                        case 1: create();

                                break;

                        case 2: insert_begin();

                                break;

                        case 3: insert_end();

                                break;

                        case 4: insert_desired();
```

```
                    break;
                case 5: delete_begin();
                    break;
                case 6: delete_end();
                    break;
                case 7: delete_desired();
                    break;
                case 8: display();
                    break;
                case 9:search();
                    break;
                case 10: exit(1);
                    break;
                defaulf:printf("Invalid option");
                    break;
            }
        }
    }
    void create()
    {
        inti,n;
        printf("Enter the no of nodes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            new_node = (struct node *) malloc(sizeof(struct node));
            printf("Enter the data:");
```

```
                scanf("%d",&new_node->data);
                new_node->prev = NULL;
                new_node->next = NULL;
                if(head==NULL)
                {
                        head = last = new_node;
                }
                else
                {
                        last->next = new_node;
                        new_node->prev = last;
                        last = new_node;
                }
        }
}

void insert_begin()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("Enter the data:");
        scanf("%d",&new_node->data);
        new_node->prev=NULL;
        new_node->next=NULL;
        if(head==NULL)
        {
                head=last=new_node;
        }
        else
        {
                new_node->next = head;
                head->prev = new_node;
                head = new_node;
        }
        printf("\nNODE INSERTED SUCCESSFULLY");
 }
void insert_end()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->prev = NULL;
        new_node->next= NULL;

        if(head == NULL)
        {
                head=last=new_node;
        }
        else
        {
                last->next = new_node;
                new_node->prev = last;
                last = new_node;
        }
}
```

```
void insert_desired()
{
        intpos,i;
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->prev = NULL;
        new_node->next= NULL;
        printf("\nEnter the position to insert:");
        scanf("%d",&pos);
        if(head == NULL)
        {
                head = last = new_node;
        }
        else
        {    temp = head;
                for(i=1;i<pos-1;i++)
                {
                    temp = temp->next;
                }
                temp->next->prev = new_node;
                new_node->next=temp->next;
                new_node->prev=temp;
                temp->next = new_node;


                printf("\nSUCCESFULLY INSERTED\n");
        }
        getch();
}
void delete_begin()
{
        if(head==NULL)
        {
                printf("\nThere are no elements in the List.");
        }
        else
        {
                temp = head;
                head = temp->next;
                free(temp);
                printf("\nSUCCESSFULLY DELETED");
        }
        getch();
}
void delete_end()
{
        temp=head;

        if(head==NULL)
        {
                printf("\n Cannot delete node, because list is empty");
        }
        else if(head == last)
        {
                printf("\nDeleted node is %d", last->data);
                head=last=NULL;
                free(temp);
        }
        else
        {
        printf("\nDeleted node is %d",last->data);
             while(temp->next != last)
             {
                        temp=temp->next;
             }
             free(last);
             temp->next = NULL;
             last=temp;
        }
}
void delete_desired()
{
        intpos,i;
        struct node *temp1;
        printf("\nEnter the position of the node to delete:");
        scanf("%d",&pos);
        if(head == NULL)
        {
                printf("\nThere are no nodes to delete:");
        }
        else
        {
                temp = head;
                if(temp->next->next == NULL)
                {
                        printf("\nCannot delete because there are only 2 nodes in the list, so
use delete_begin or delete_end functions ");
                        return;
                }
                for(i=1;i<pos-1;i++)
                {
                        temp=temp->next;
                }
                printf("\nDeleted node is %d", temp->next->data);
                temp1=temp->next;
                temp->next = temp->next->next;
                temp->next->prev = temp;
                free(temp1);
        }
}
void search()
{
        int key;
        printf("\nEnter the search element:");
        scanf("%d",&key);
        temp = head;
        if(head==NULL)
        {
                printf("\nList is EMPTY");
        }
        else
        {
                while(temp != NULL)
                {
                        if(temp->data == key)
                        {
                                printf("\nSearch element is found");
                                return;
                        }
                        temp=temp->next;
                }
                printf("\nSearch element is not found");
        }
}
void display()
{
        if(head==NULL)
        {
                printf("There are no elements in the DLL");
        }
        else
        {  printf("\nElements in the DLL list are:\n");
                temp = head;
                do
                {
```

```
                printf("%d ",temp->data);

                temp=temp->next;

            }while(temp != NULL);

        }

}
```

III Circular Linked List

1. Introduction and Representation

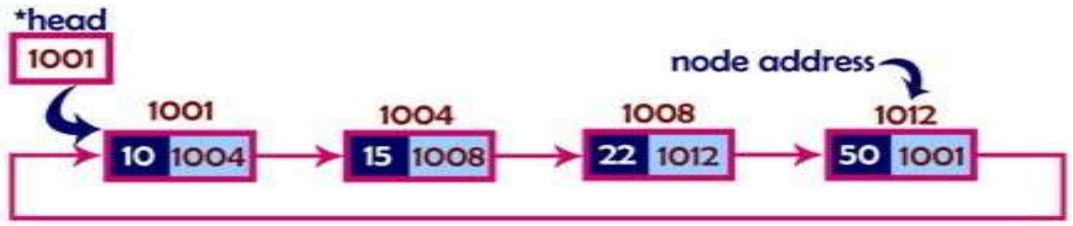
What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

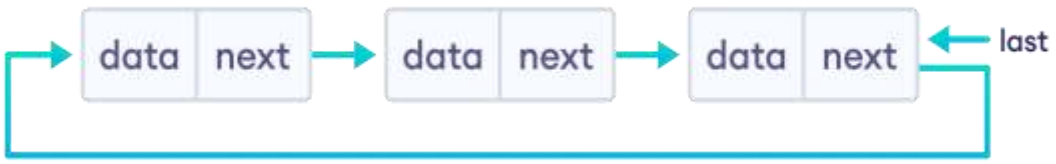
Example:



There are basically two types of circular linked list:

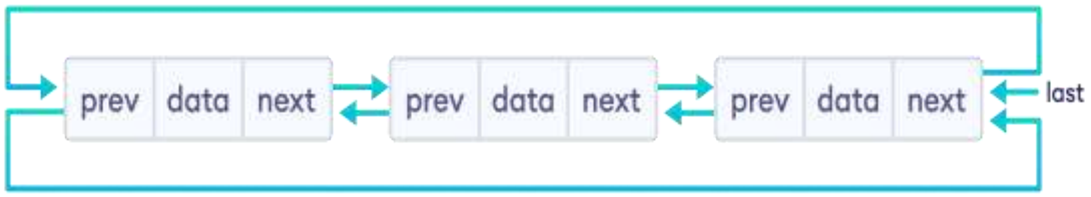
1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



2. Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



2. Operations of Circular Linked List

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

2.1 Insertion operation

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

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

2.1.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular 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, set **head** = **newNode** and **newNode→next** = **head** .

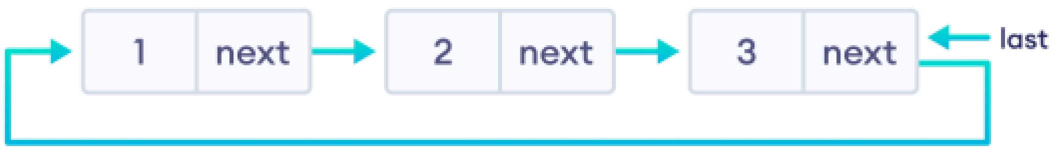
Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').

Step 6 - Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Example:

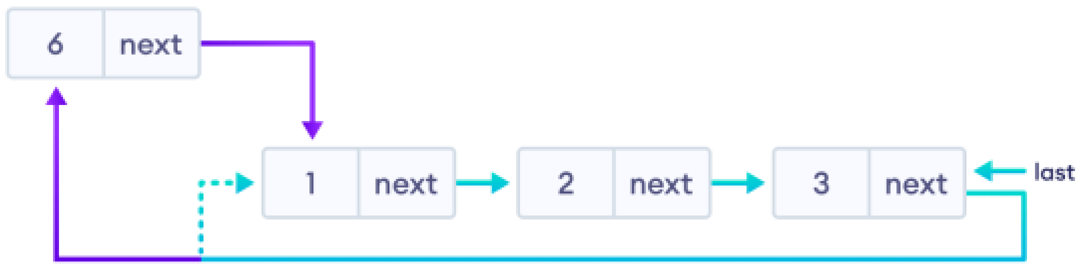
Suppose we have a circular linked list with elements 1, 2, and 3.



The first step is to create a new node.

6
next
New Node

The next step is to insert at beginning of list



2.1.2 Inserting At End of the list

We can use the following steps to insert a new node at end of the circular 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, set **head** = **newNode** and **newNode** → **next** = **head**.

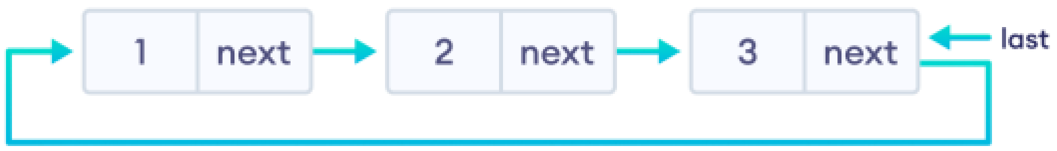
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).

Step 6 - Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

Example:

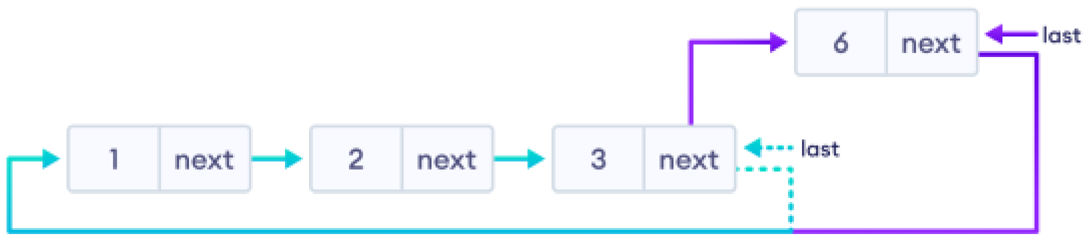
Suppose we have a circular linked list with elements 1, 2, and 3.



The first step is to create a new node.

6
next
New Node

The next step is to insert at beginning of list



2.1.3 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 circular 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, set **head** = **newNode** and **newNode** → **next** = **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 node after which we want to insert the newNode (until **temp** → **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 **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

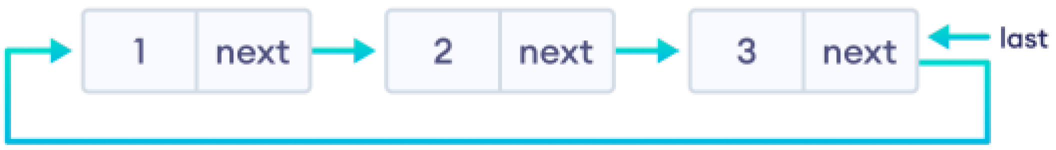
Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8 - If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.

Step 8 - If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Example:

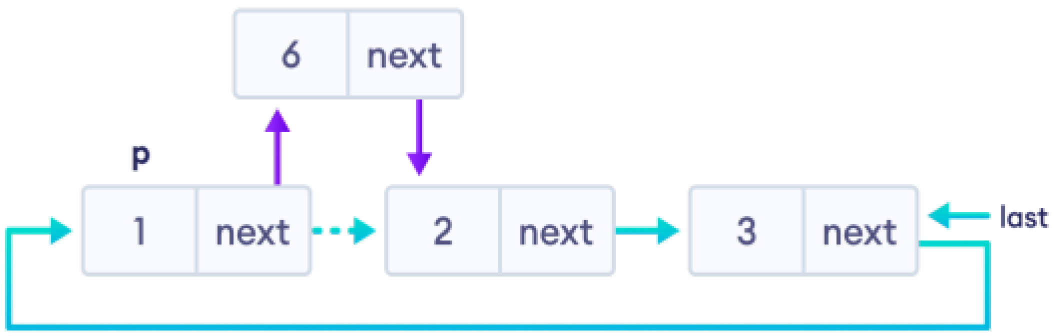
Suppose we have a circular linked list with elements 1, 2, and 3.



The first step is to create a new node.

6
next
New Node

The next step is to insert at specific position of list



2.2. Deletion operation

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

2.2.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular 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 two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

Step 4 - Check whether list is having only one node (**temp1 → next** == **head**)

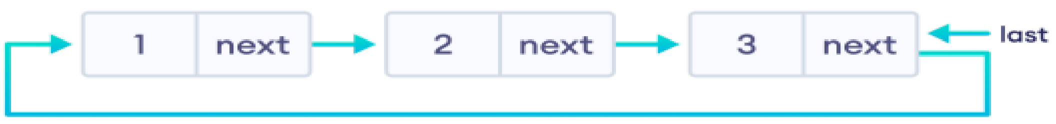
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)

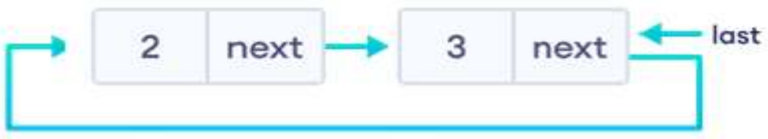
Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Example:

Suppose we have a double-linked list with elements 1, 2, and 3.



After deleting first node from the list



2.2.2 Deleting from End of the list

We can use the following steps to delete a node from end of the circular 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 two Node pointers **'temp1'** and **'temp2'** and initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1** → **next** == **head**)

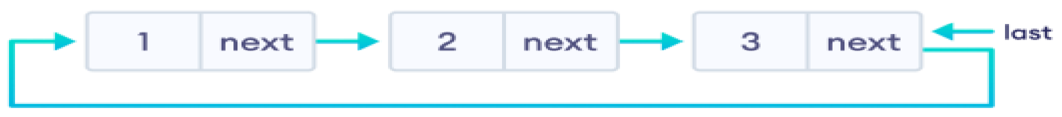
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2** = **temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1** → **next** == **head**)

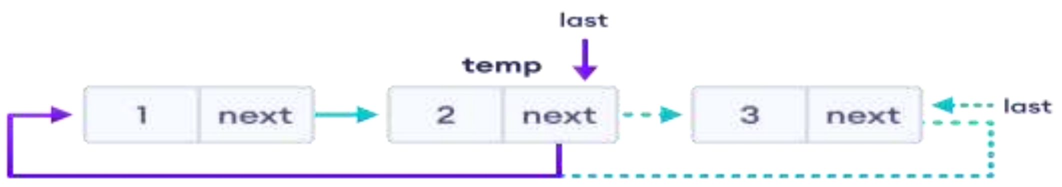
Step 7 - Set **temp2** → **next** = **head** and delete **temp1**.

Example :

Suppose we have a double-linked list with elements 1, 2, and 3.



After deleting last node from the list



2.2.3 Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular 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 two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next 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 function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)

Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.

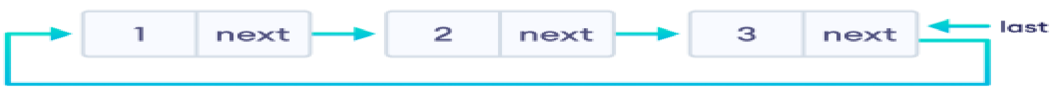
Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).

Step11- If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).

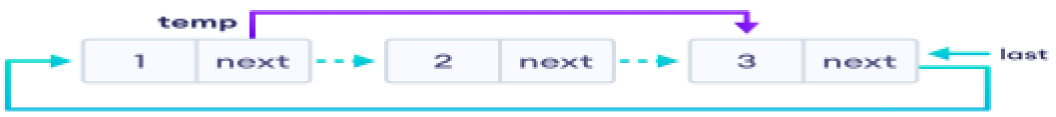
Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Example:

Suppose we have a double-linked list with elements 1, 2, and 3.



After deleting middle of node from the list



3.Displaying a circular Linked List

We can use the following steps to display the elements of a circular 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 - Keep displaying **temp** → **data** with an arrow (**--->**) until **temp** reaches to the last node

Step 5 - Finally display **temp** → **data** with arrow pointing to **head** → **data**.

Program

```
#include<stdlib.h>

#include<stdio.h>

struct node

{

        int data;

        struct node *link;

};

struct node *head,*last,*temp,*new_node;

void create();

void insert_begin();

void insert_end();

void insert_desired();

void delete_begin();

void delete_end();

void delete_desired();

void display();

void search();

void main()
```

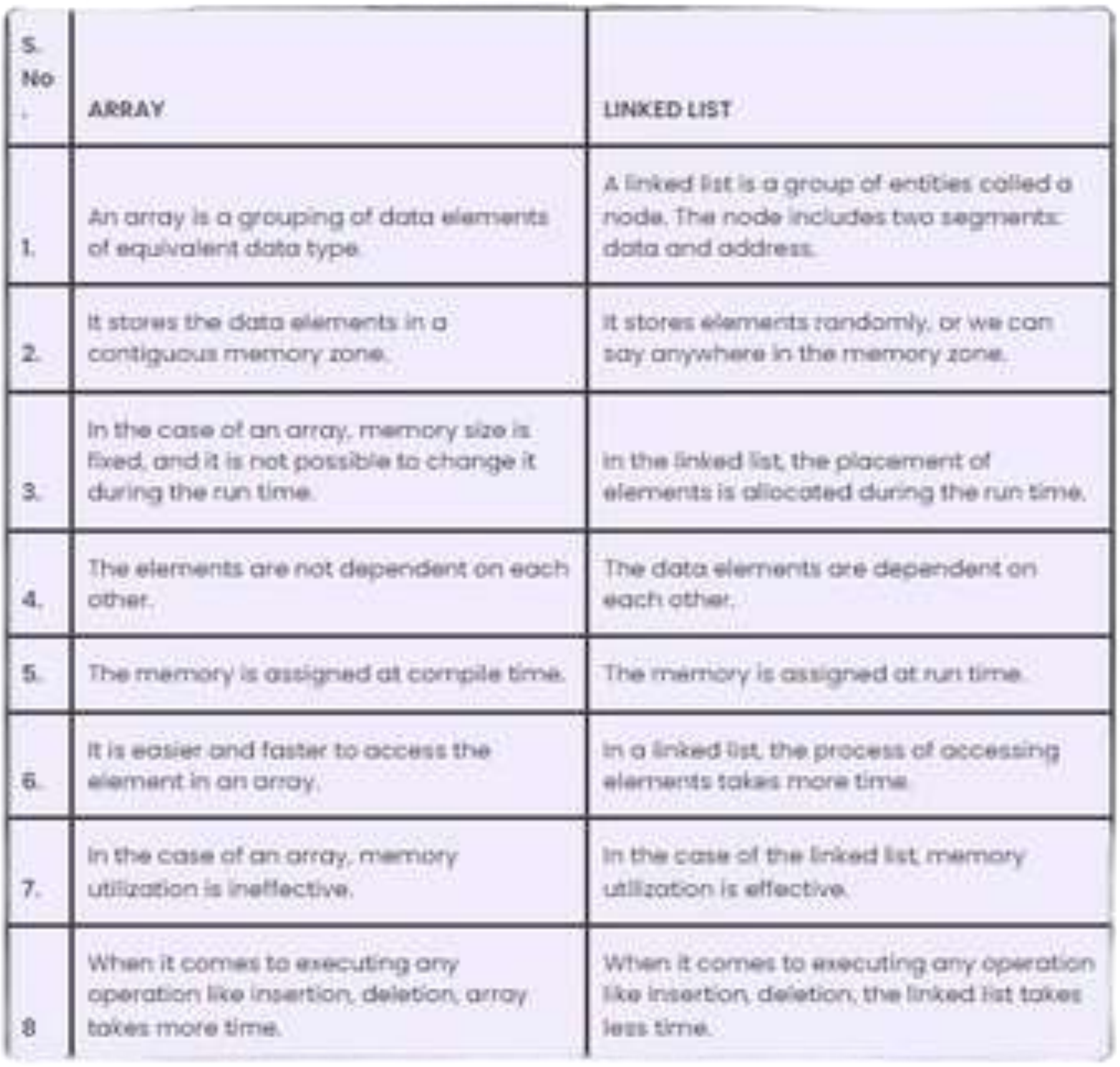
```
{
        int choice;
        clrscr();
                while(1)
        {
                printf("\n          MAIN MENU          ");
                printf("\n1.Create  2.Insert  at  begin  3.insert  end  4.insert  desired  \n5.delete
begin 6. delete end 7.del des 8.Display \n9.search 10.Exit\n");
                printf("*******************************************");
                printf("\nEnter your choice:");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: create();
                                break;
                        case 2: insert_begin();
                                break;
                        case 3: insert_end();
                                break;
                        case 4: insert_desired();
                                break;
                        case 5: delete_begin();
                                break;
                        case 6: delete_end();
                                break;
                        case 7: delete_desired();
                                break;
                        case 8: display();
                                break;
                        case 9: search();
                                break;
                        case 10: exit(1);
                                break;
                        defaulf:printf("Invalid option");
                                break;
                }
        }
}
 void create()
 {
        inti,n;
        printf("Enter the no of nodes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                new_node = (struct node *) malloc(sizeof(struct node));
                printf("Enter the data:");
                scanf("%d",&new_node->data);
                new_node->link = NULL;
                if(head==NULL)
                {
                        head = last = new_node;
                        last->link=head;
                }
                else
                {
                         last->link = new_node;
                        new_node->link = head;
                         last = new_node;
                }
        }
}
void insert_begin()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("Enter the data:");
        scanf("%d",&new_node->data);
        new_node->link=NULL;

        if(head==NULL)
        {
                head = last = new_node;
                last->link=head;
        }
        else
        {
                new_node->link = head;
                last->link = new_node;
                head = new_node;
        }
        printf("\nNODE INSERTED SUCCESSFULLY");
```

```
 }
void insert_end()
{
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->link = NULL;

        if(head == NULL)
        {
                head=last=new_node;
                last->link=head;
        }
        else
        {
                last->link = new_node;
                new_node->link = head;
                last = new_node;
        }
}
void insert_desired()
{
        intpos,i;
        new_node = (struct node*)malloc(sizeof(struct node));
        printf("\nEnter the data:");
        scanf("%d",&new_node->data);
        new_node->link = NULL;
        printf("\nEnter the position to insert:");
        scanf("%d",&pos);
        if(head == NULL)
        {
                head = last = new_node;
                last->link=head;
        }

else
        {   temp = head;
                for(i=1;i<pos-1;i++)
                {
                    temp = temp->link;
                }
                new_node->link=temp->link;
                temp->link = new_node;

                if(new_node->link == head)
                        last=new_node;
                printf("\nSUCCESFULLY INSERTED\n");
        }
        getch();
}
void delete_begin()
{
        if(head==NULL)
        {
                printf("\nThere are no elements in the List.");
        }
        else
        {
                temp = head;
                head = temp->link;
                last->link=head;
                free(temp);
                printf("\nSUCCESSFULLY DELETED");
        }
        getch();
}
void delete_end()
{
        temp=head;
        if(head==NULL)
        {
                printf("\n Cannot delete node, because list is empty");
        }
        else if(head == last)
        {
                printf("\nDeleted node is %d", last->data);
                head=last=NULL;
                free(temp);
        }
        else
        {
          printf("\nDeleted node is %d",last->data);
                while(temp->link != last)
                {
                              temp=temp->link;
                }
                free(last);
                temp->link = head;
                last=temp;
          }
}
void delete_desired()
{
          intpos,i;
          struct node *temp1;
          temp = head;
          printf("\nEnter the position of the node to delete:");
          scanf("%d",&pos);
          if(head == NULL)
          {
                    printf("\nThere are no nodes to delete:");
          }
          else
          {
                    temp = head;
                    if(temp->link->link == last)
                    {
```

```
                                        printf("\nCannot delete because there are only 2 nodes, so use
either delete_begin or delete_end functions");
                                        return;
                    }
                    for(i=1;i<pos-1;i++)
                    {
                              temp=temp->link;
                    }
                    printf("\nDeleted node is %d", temp->link->data);
                    temp1=temp->link;
                    temp->link = temp->link->link;
                    free(temp1);
          }
}
void search()
{
          int key;
          printf("\nEnter the search element:");
          scanf("%d",&key);
          temp = head;
          if(head==NULL)
          {
                    printf("\nList is EMPTY");
          }
          else
          {
                    do
```

```
                {
                        if(temp->data == key)
                        {
                                printf("\nSearch element is found");
                                return;
                        }
                        temp=temp->link;
                }while(temp != head);
                printf("\nSearch element is not found");
        }
}
void display()
{
        if(head==NULL)
        {
                printf("There are no elements in the LL");
        }
        else
        {      printf("\nElements in the list are:\n");
                temp = head;
                do
                {
                        printf("%d ",temp->data);
                        temp=temp->link;
                }while(temp != head);
        }
}
```

IV Comparing Arrays and Linked list



S. No .	ARRAY	LINKED LIST
1.	An array is a grouping of data elements of equivalent data type.	A linked list is a group of entities called a node. The node includes two segments: data and address.
2.	It stores the data elements in a contiguous memory zone.	It stores elements randomly, or we can say anywhere in the memory zone.
3.	In the case of an array, memory size is fixed, and it is not possible to change it during the run time.	In the linked list, the placement of elements is allocated during the run time.
4.	The elements are not dependent on each other.	The data elements are dependent on each other.
5.	The memory is assigned at compile time.	The memory is assigned at run time.
6.	It is easier and faster to access the element in an array.	In a linked list, the process of accessing elements takes more time.
7.	In the case of an array, memory utilization is ineffective.	In the case of the linked list, memory utilization is effective.
8	When it comes to executing any operation like insertion, deletion, array takes more time.	When it comes to executing any operation like insertion, deletion, the linked list takes less time.

V.Applications of linked list

1. Implementation of stacks and queues

2. Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.

3. Dynamic memory allocation: We use a linked list of free blocks.

4. Maintaining a directory of names

5. Performing arithmetic operations on long integers

6. Manipulation of polynomials by storing constants in the node of the linked list

7. Representing sparse matrices.