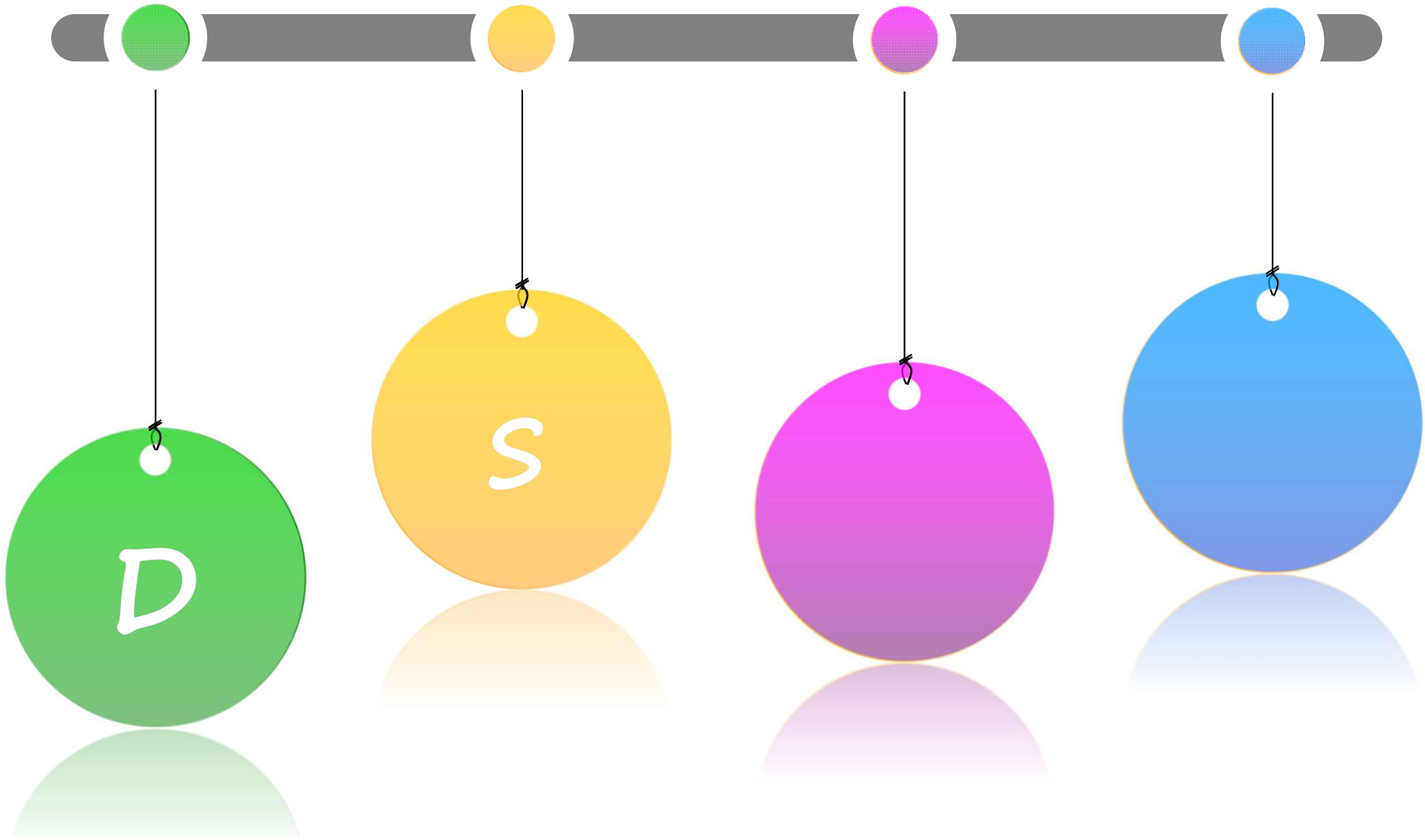

D
S

01 TOPIC — Dynamic Memory Allocation Functions

02 TOPIC — Self-Referential Structures

04 TOPIC — Basic Terminologies

06 TOPIC — Operations on Singly Linked Lists

01 TOPIC — malloc, calloc, free and realloc

03 TOPIC — Introduction to Linked List

05 TOPIC — Linked Lists versus Arrays

07 TOPIC — Doubly Linked Lists

Memory Allocation

- ✓ In **programming**, the term **memory allocation** plays **important role.**
- ✓ Basically, it is a **process** by which a particular computer **program is allocated memory space**.
- ✓ There are **two types of memory allocations.**
    - i. **Static**
    - ii. **Dynamic**

Difference between Static and Dynamic Memory Allocation:

Static memory allocation	Dynamic memory allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Dynamic Memory Allocation

- ✓ C **Dynamic Memory Allocation** can be defined as a procedure in which the **size of a memory block is changed** during the runtime.
- ✓ **C provides some functions** to achieve these tasks.
- ✓ There are **4 library functions provided** by **C** defined under **<stdlib.h>** header file or **<alloc.h>** to facilitate dynamic memory allocation in C programming.

Function Name	Description
malloc()	Allocates **single block** of memory
calloc()	Allocates **multiple block** of memory
free()	**Delete** the dynamically allocated memory
realloc()	**Reallocates the memory** (increase / decrease the size of block)

i.malloc( )

- ✓ It is used to dynamically allocate a **single large block of memory** with the **specified size.**

Syntax

ptr = (cast-type*) malloc(byte-size)

Example

p1 = (int*) malloc (10 * sizeof(int));

- ✓ Since the **size of int is 2 bytes**, this statement will **allocate 20 bytes of memory.**
- ✓ The **pointer ptr holds the address** of the **first byte** in the allocated memory.
- ✓ The malloc() function **allocates single block** of **requested memory**.
- ✓ On **failure, It returns NULL** i.e. memory is not sufficient.
- ✓ On **success, It returns base address** of memory block i.e. memory is sufficient

Example:

```
#include <stdio.h>
#include <stdlib.h>
int   main()
{
     int  *ptr;
     int n = 5, i;
   ptr = (int*)malloc(n * sizeof(int) );
   if (ptr == NULL)
  {
      printf("Memory not allocated.");
     exit(0);
  }
 else
 {
    printf("Memory successfully allocated using malloc.");
 for (i = 0; i < n;  i++)
    {
        ptr[ i ] = i + 10;
    }


 printf("The elements of the array are: ");
    for (i = 0; i < n; i++)
    {
        printf("%d ", ptr[i]);
    }
 }
  return 0;

}
```

Output:

```
successfully allocated using malloc.
The elements of the array are: 10 11 12 13 14
```

ii. calloc( )

- ✓ It is **used to** dynamically **allocate the specified number of blocks of memory** of the specified type.
- ✓ It initializes **each block with a default value '0'.**

Syntax

ptr = (cast-type*)calloc(n, element-size);

Example

ptr = (float*) calloc(25, sizeof(float));

- ✓ The above statement **allocates contiguous space in memory for 25 elements each** with the **size of the float.**
- ✓ It **returns NULL if memory is not sufficient**

iii.free( )

- ✓ It is used to **dynamically de-allocate the memory.**
- ✓ It **helps to reduce wastage of memory** by freeing it.

Syntax

```
free(ptr);
```

iv. realloc()

- ✓ If the **dynamically allocated memory** is **insufficient** or **more than required.**
- ✓ It is used to dynamically **change the memory allocation** of a **previously allocated memory.**
- ✓ In other words, if the **memory previously allocated** with **the help of malloc or calloc** is **insufficient, realloc can be used** to dynamically **re-allocate memory.**

Syntax

```
ptr = realloc(ptr, newSize);
```

Example

```
p1 = (int*) malloc (10 * sizeof(int));
```

Example

```
p1 = realloc(p1, 40);
```

Self Referential Structure

- ✓ The self-referential structure is a **structure that points to the same type of structure**.
- ✓ It **contains one** or **more pointers** that **ultimately point to the same structure**.
- ✓ Self-referential structure **plays a very important role** in **creating other data structures** like **Linked list.**

Syntax

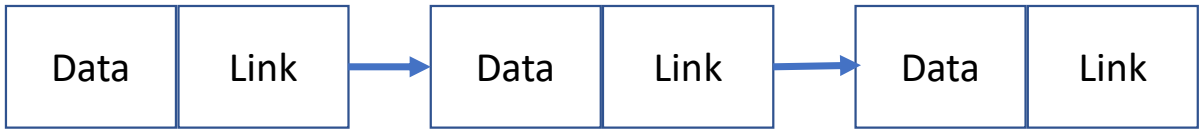
```
struct tagname
{
        Data Member_1;
        Data Member_2;
        ……….
        ……….
        struct  tagname *PointerName;
}Structure_variable;
```

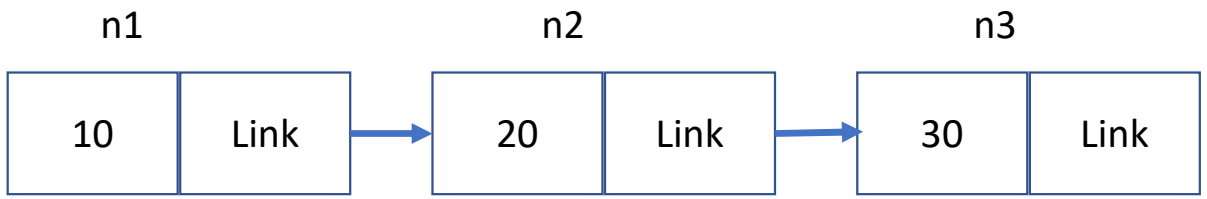
- ✓ Note: Here, **tagname** means **structure name.**

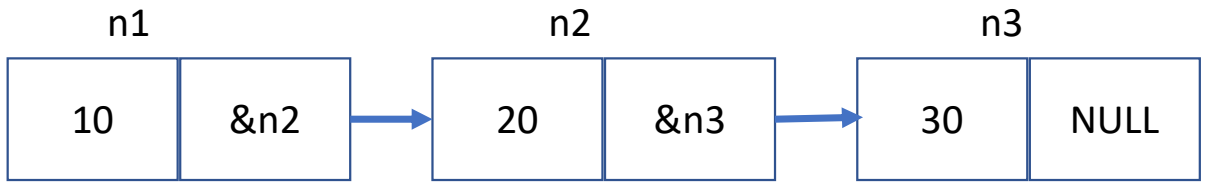
Example

```
struct Node
{
        int data;
        struct  Node *Link;
}n1;
```

```c
//program for self referential structure
#include <stdio.h>
struct node
 {
        int data;
        struct node* link;
}n1,n2,n3;
void main()
{
        n1.data = 10;
        n2.data = 20;
        n3.data = 30;
        n1.link = &n2;
        n2.link = &n3;
        n3.link = NULL;
        printf("%4d", n1.data);
        printf("%4d", n1.link->data);
        printf("%4d", n1.link->link->data);
}
```







Linked List

- A linked list data structure includes a series of connected nodes.
- Here, **each node store** the data and the address of the next node.
- In linked list elements are **not stored at contiguous memory locations.**
- The **entry point** into a linked list is called the **head** of the list.
- Each **node in a list consists** of at least **two parts:**
  - ✓ **Data**
  - ✓ **Pointer or Address of next node**

Uses of Linked List:

- The list is not required to be contiguously present in the memory.
- The node can reside any where in the memory and linked together to make a list. This achieves **optimized utilization of space.**
- **list size** is **doesn't need to be declared in advance.**

Difference between Array and Linked list

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Types of Linked List

- There are **two different types** of Linked Lists. They are :

    1. **Singly Linked List**

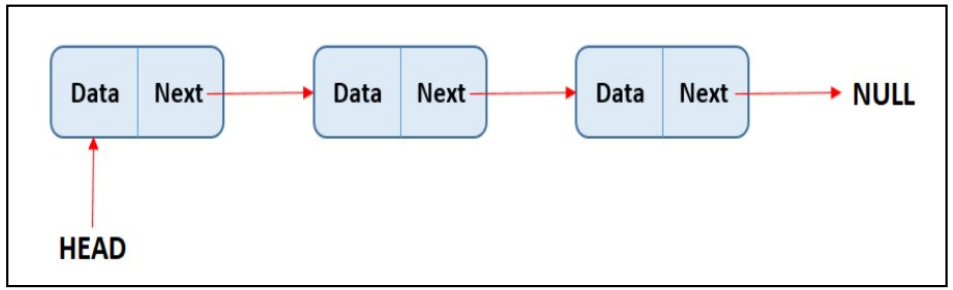
    2. **Doubly Linked List**

Singly Linked List (SLL)

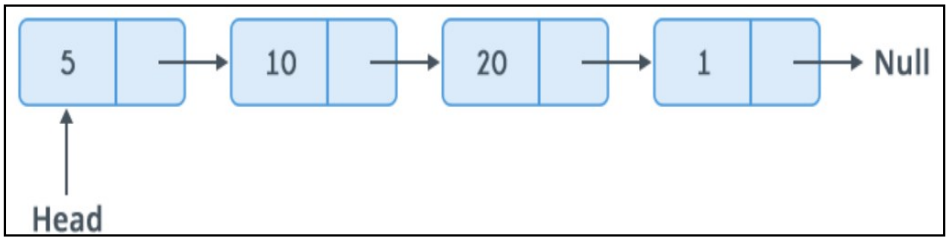
- It is the simplest type of **linked list in which every node contains some data** and a **pointer to the next node** of the same type.

- The node contains **a pointer to the next node** means that the **node stores the address** of the **next node** in the **sequence.**

- A single linked list allows **traversal of data only in one way.**



Representation of Single Linked List



Example of Single Linked List

Operations of Singly Linked List

- **Creation**
- **Insertion of Node**
  - ✓ Insertion at the beginning
  - ✓ Insertion at the end
  - ✓ Insertion At a given position
- **Deletion of Node**
  - ✓ Deletion At the beginning
  - ✓ Deletion At the end
  - ✓ Deletion At a given position
- **Display**
- **Count number of Nodes**
- **Reverse of Linked list**

Creation of Singly Linked List

```
//program for self referential structure
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void create();
void display();
struct Node
{
   int data;
   struct Node *next;
}*head=NULL,*rear=NULL,*temp=NULL;

int main()
{
create();
display();
return 0;
}
```

```
void display()
{
   printf("\n Linked List is: \n");
   temp=head;
   while(temp!=NULL)
   {
      printf("%3d",temp->data);
      temp=temp->next;
   }

}
```

Creation of Singly Linked List

```
void create()
{
        struct Node *newnode=NULL;
        newnode=(struct Node*)malloc(sizeof(struct Node));
        printf("Enter any Value:");
        scanf("%d",&newnode->data);
        newNode->next=NULL;
        if(head==NULL)
        {
                head=rear=newnode;
                count++;
        }

        while(1)
        {
                struct Node *temp=NULL;
                temp=(struct Node*)malloc(sizeof(struct Node));
                printf("Enter any value Press Zero to exit:");
                scanf("%d",&temp->data);
                if(temp->data==0)
                        break;
                else
                {
                        temp->next=NULL;
                        rear->next=temp;
                        rear=temp;
                        count++;
                }
        }
}
```

```
Enter any Value:10
Enter any value Press Zero to exit:20
Enter any value Press Zero to exit:30
Enter any value Press Zero to exit:0

 Linked List is:
 10 20 30
--------------------------------
```

i. Insertion at the beginning(SLL)

- In this case ,If the **list is empty**, we **make the new node** as the **head of the list.**
- **Otherwise**, we have to connect the new node to the current head of the list and **make the new node** as **head** of the list.

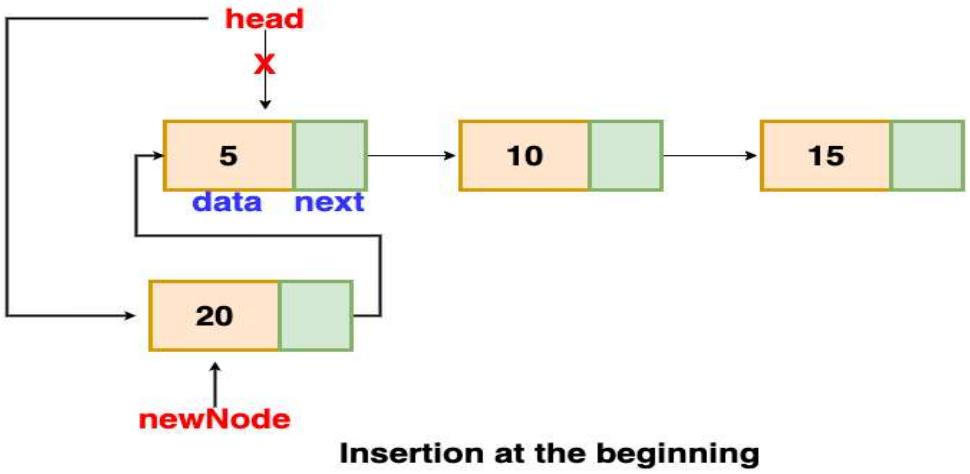
Steps:

i. **Allocate memory for new node**

ii. **Store data**

iii. **Change next of new node** to point to **head**

iv. **Change head** to point to **recently created node**



Insertion at the beginning

i. Insertion at the beginning(SLL)

```
//Insertion of a node at begining
void insert_begining()
{
        struct Node *temp=NULL;
        temp=(struct Node *)malloc(sizeof(struct Node));
        printf("\n Enter any Value to insert at begining:");
        scanf("%d",&temp->data);
        if(head==NULL)
        {       temp->next=NULL;
                head=rear=temp;
        }
        else
        {
                temp->next=head;
                head=temp;

        }
}
```

ii. Insertion at end(SLL)

- In this case **we will traverse** the list **until we find the last node**. Then we **insert the new node to** the **end of the list.**
- **In case** of a **list** being **empty**, the **inserted node** is **the first** as well as the **last node** of the linked list.

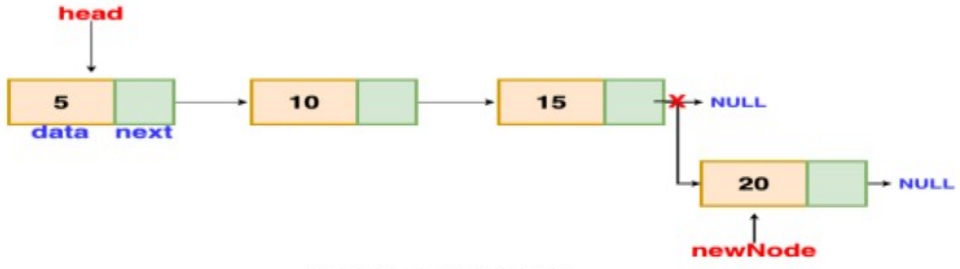
Steps:

i. **Allocate memory** for **new node**

ii. **Store data**

iii. **Traverse to last node**

iv. **Change next of last node** to **recently created node**



Insertion at the end

```
//Insertion of a node at ending
void insert_ending()
{
        struct Node *temp=NULL;
        temp=(struct Node *)malloc(sizeof(struct Node));
        printf("\n Enter any Value to insert at begining:");
        scanf("%d",&temp->data);
        temp->next=NULL;
        rear->next=temp;
        temp=rear;
}
```

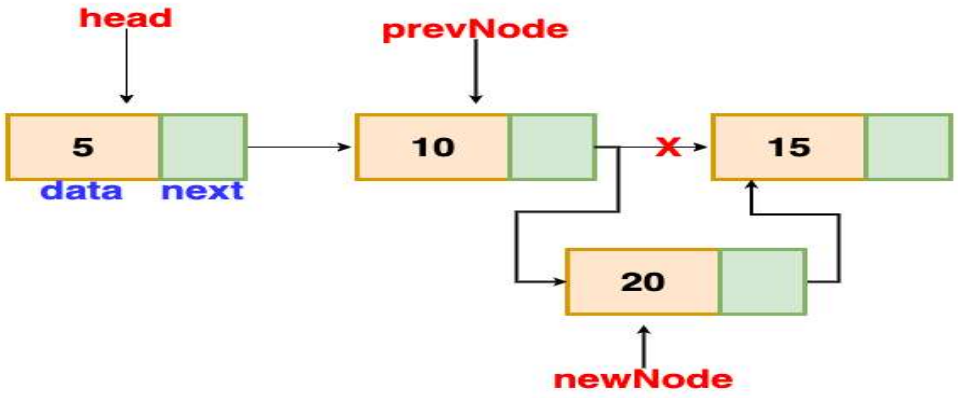
iii. Insertion after a given node(SLL)

Here, the **new node** is **inserted after** the **given node.**

i. **Allocate memory** and **store data** for **new node**

ii. **Traverse** to **node just before** the **required position** of new node

iii. **Change next pointers** to include **new node in between.**



Insertion after a given node

iii. Insertion after a given node(SLL)

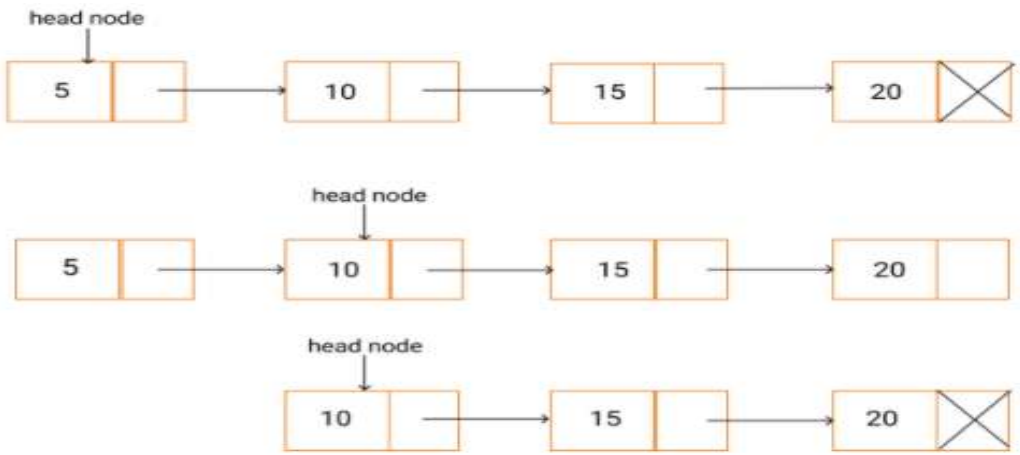
```
//Insertion of a node at ending
void insert_at_anyposition()
{
        int pos,i;
        struct Node*temp=NULL,*prev=NULL,*succ=NULL;
        printf("\n Enter the any position numer:");
        scanf("%d",&pos);
        temp=head;
                for(i=2;i<pos;i++)
                {
                    if(temp->next!=NULL)
                    {
                     temp=temp->next;
                    }
                }
                struct Node *newnode=NULL;
                newnode=(struct Node*)malloc(sizeof(struct  Node));
                printf("\n**INSERT AT ANY GIVEN POSITION**\n");
                printf("\n Enter any value");
                scanf("%d",&newnode->data);
                newnode->next = temp->next;
                temp->next = newnode;
                count++;
}
```

i. Deletion at the beginning(SLL)

i. To delete the first node (head node), **copy the head node** in a **temp node**.

ii. **Make the second node** as the **head node.**

iii. Now, **delete the temporary node.**



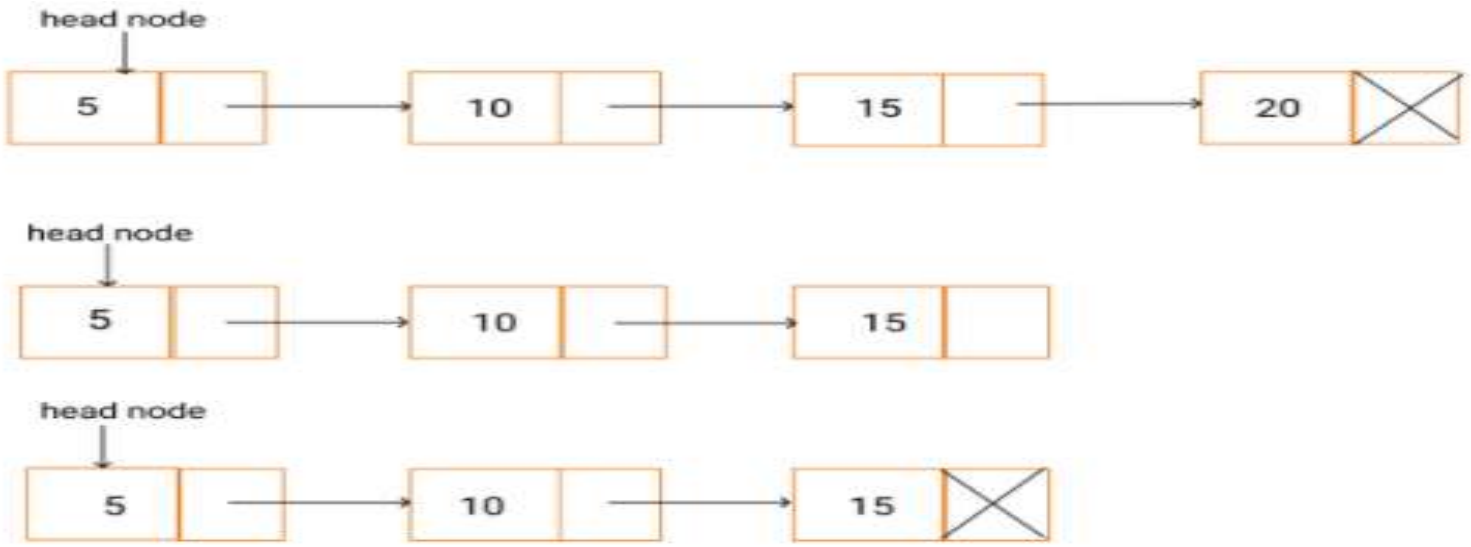

```
//Deletion of a node at begining
void del_at_begining()
{
 struct Node *temp;
 temp=head;
 if (head==NULL)
  {
   printf ("Linked List Empty, ");
   return;
  }
 head=head->next;
 free(temp);
}
```

ii.Deletion at the end(SLL)

i. **To delete the last node**, start traversing the list from the head node and continue traversing until the address part of the node is **NULL**.

ii. **Keep track of the second last node** in some **temporary node.**

iii. Set the **address part** of the temporary **node as NULL** and then **delete the last node.**

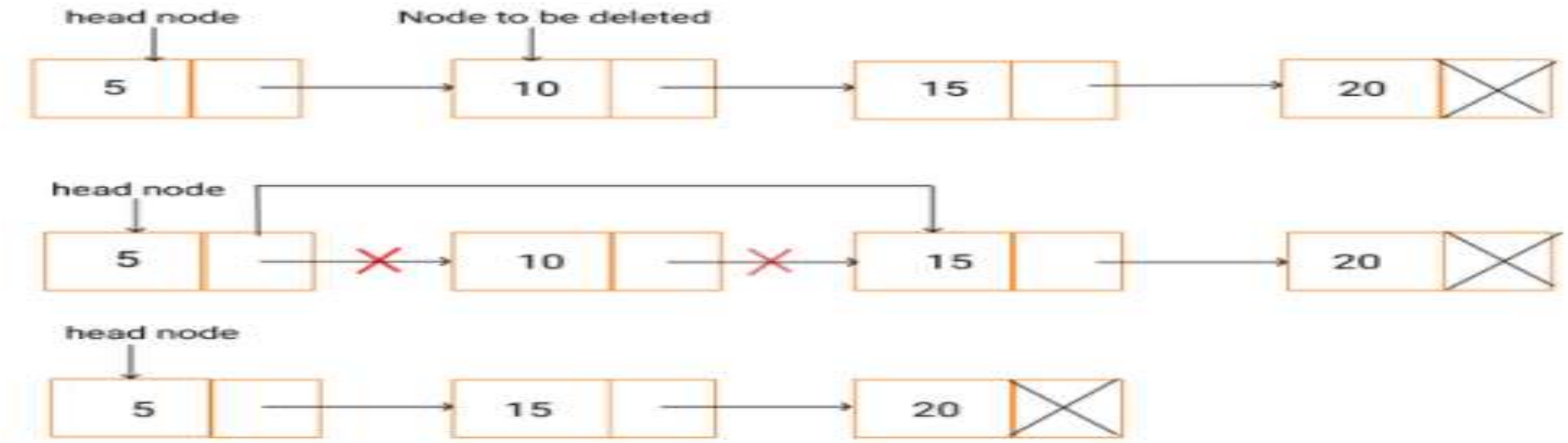


ii.Deletion at the end(SLL)

```
void del_End()
{
        if(head==NULL)
        {
        printf("\nList is Empty!!!\n");
        }
        else if(head->next==NULL)
        {
                free(head);
        }
        else
        {
        struct Node *temp,*prev;
        temp=head;
         while(temp->next!=NULL)
         {
                prev=temp;
                temp=temp->next;
         }
         prev->next=NULL;
        free(rear);
        rear=prev;
        printf("\nOne node deleted!!!\n\n");
   }
}
```

iii. Deletion at a given position(SLL)

- Now let us assume that the **node at position 2 has to be deleted**.
- **Start traversing** the list **from the head node** and **move up to that position.**
- **While traversing**, **keep track** of the **previous node** to the node to be deleted.
- In this case, since **we want to delete the second node**, you **need to traverse till node 2**, **storing node 1** in **some temporary variable**.
- Now, the address **part of node 2** is assigned to the **address part of node 1** and then **node 2 is deleted.**



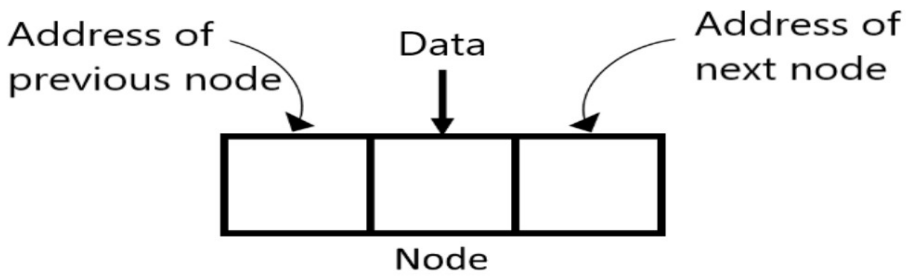
iii. Deletion at a given position(SLL)

```
void deletion_atposition()
{
        int pos,i;
        struct Node *temp,*n;
        printf("\n Enter position");
        scanf("%d",&pos);
        temp=head;
        for(i=2;i<pos;i++)
        {
                if(temp->next!=NULL)
                {

                        temp-=temp->next;
                }

        }
      n=temp->next;
      temp->next= temp->next->next;
      free(n);
}
```

Doubly Linked List(DLL)

✓ Doubly Linked List(DLL) 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.**

✓ A **sample node in a doubly linked list** is :



✓ Therefore, in a doubly linked list, a **node consists of three parts:**

node data,

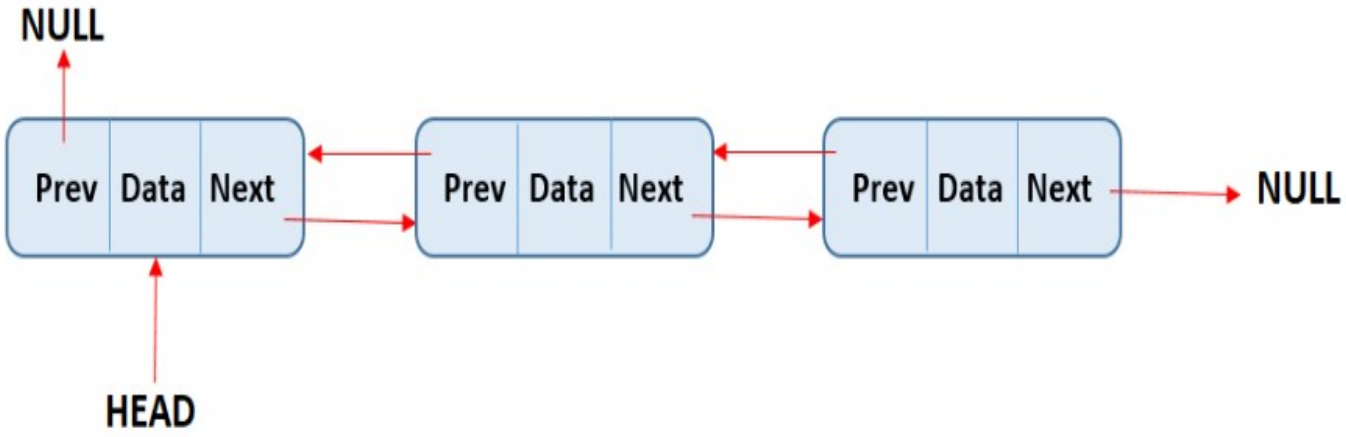
-**pointer to the next node** in sequence (next pointer)

-**pointer to the previous node** (previous pointer)

-**Data** of the node

Representation of Doubly Linked List

✓**Each node** of **doubly linked list** consists of **three components**, **namely, previous pointer**, **data,** and **next pointer .**

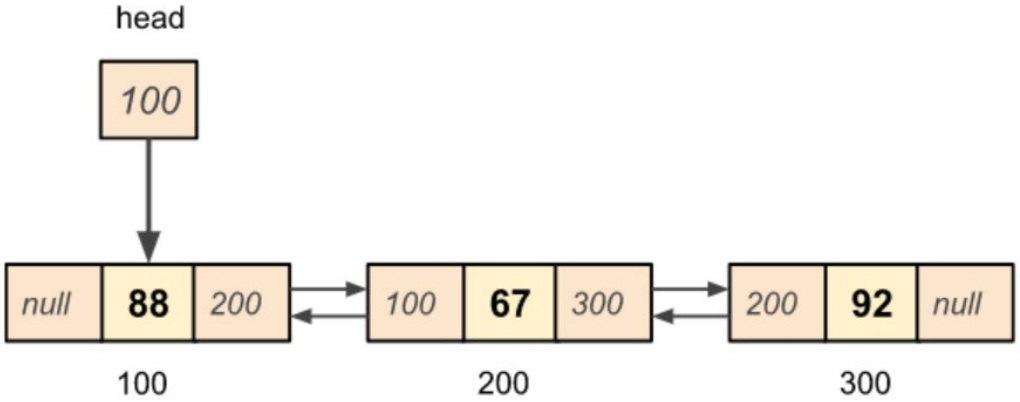


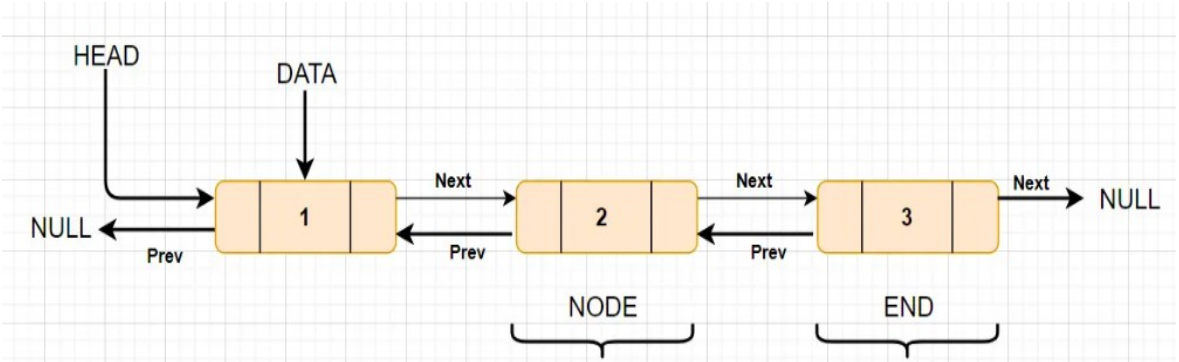

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

✓The **first node** also **known as HEAD** and **last node** is known as **REAR.**

✓The **previous of HEAD** node and **next of REAR** node points **to NULL.**

Example of Doubly Linked List





Operations of Doubly Linked List(DLL)

- **Creation**
- **Insertion of Node**
  - ✓ Insertion at the beginning
  - ✓ Insertion at the end
  - ✓ Insertion At a given position
- **Deletion of Node**
  - ✓ Deletion At the beginning
  - ✓ Deletion At the end
  - ✓ Deletion At a given position
- **Display**
- **Count No of Nodes**
- **Reverse**

Creation of Doubly Linked List

```
//program for double linked list creation and display
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void create();
void display();
struct Node
{
        struct Node *prev;
        int data;
        struct Node *next;
}*head=NULL,*rear=NULL;
int main()
{
create();
display();
return 0;
}
```

```
void display()
{
   printf("\n Linked List is: \n");
   temp=head;
   while(temp!=NULL)
   {
      printf("%3d",temp->data);
      temp=temp->next;
   }

}
```

Creation of Doubly Linked List

```
void create()
{
        struct Node *temp=NULL;
        temp=(struct Node*)malloc(sizeof(struct Node));
        printf("Enter any Value:");
        scanf("%d",&temp->data);
        temp->prev=NULL;
        temp->next=NULL;
        if(head==NULL)
        {
                head=rear=temp;
                count++;
        }
```

```
while(1)
{
        struct Node *temp=NULL;
        temp=(struct Node*)malloc(sizeof(struct Node));
        printf("Enter any value Press Zero to exit:");
        scanf("%d",&temp->data);
        if(temp->data==0)
                break;
        else
        {
                temp->next=NULL;
                temp->prev=rear;
                rear->next=temp;
                rear=temp;
                count++;
        }
}
}
```

i. Insertion at the beginning(DLL)

- While **inserting** a node at the **begining position**, **two situation may arise**:

i. Insertion when Doubly Linked List is empty:

Step-1 :**Create a new node** and **insert the value** in **data part**, **null** into the **next part** and **previous part**

Step-2 :**Make new node** as **head** and **rear.**

ii. Insertion when Doubly Linked List is not empty:

Step-1 : **Create a new node**, **add value** into the **data part.**

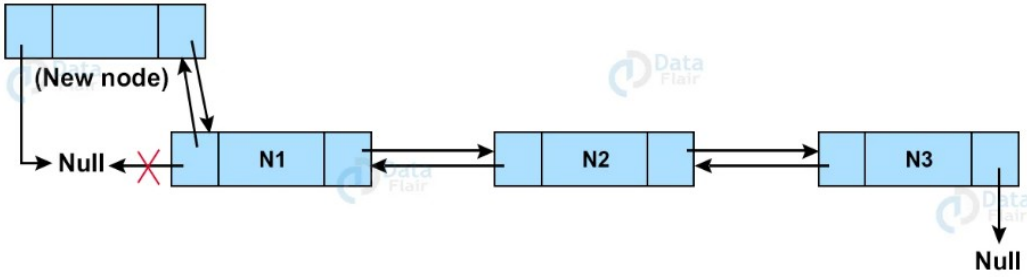
Step-2 : Place **null** into the **previous part.**

Step-3 : **next part** of **new node** assign to **the head** (existing list starting node address).

Step-4: **Make new node as head .**

i. Insertion at the beginning

```
//Doubly Linked list insert at beginning operation
void insert_at_begining()
{
        struct Node *temp=NULL;
        temp=(struct Node*)malloc(sizeof(struct Node));
        printf("\n***INSERT AT BEGINING***\n");
        printf("\n Enter any value:");
        scanf("%d",&temp->data);
        if(head==NULL)
        {
                temp->next=NULL;
                temp->prev=NULL;
                head=rear=temp;

        }
        else
        {
                temp->prev=NULL;
                temp->next=head;
                head->prev=temp;
                head=temp;

        }
        count++;
}
```



ii. Insertion at end(DLL)

- **While inserting** a **node at the end** position, **two situation may arise:**

i. Insertion when Doubly Linked List is empty:

Step-1:**Create a new node** and **insert the value** in **data part**, **null** into the **next part** and **previous part.**

Step-2:**Make new node** as **head** and **rear.**

ii. Insertion when Doubly Linked List is not empty:

Step-1:**Create a new node** and **store the value.**

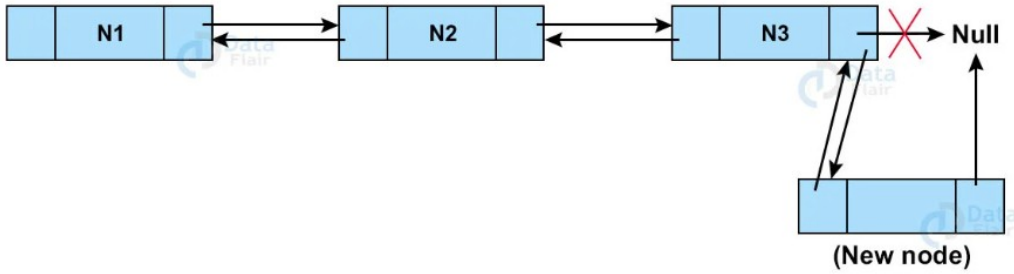
Step-2: Assign the **next part** of **new node to NULL** and **previous part of new node as rear.**

Step-3:**Assign** the **next part of rear node** to **new node.**

Step-4: **Make new node as rear**.

ii. Insertion at end(DLL)

```
//Doubly Linked list insert at end operation
void insert_at_end()
{
        struct Node *temp=NULL;
        temp=(struct Node*)malloc(sizeof(struct Node));
        printf("\n***INSERT AT ENDING***\n");
        printf("\n Enter any value:");
        scanf("%d",&temp->data);
        if(head==NULL)
        {
                temp->next=NULL;
                temp->prev=NULL;
                head=rear=temp;
        }
        else
        {
                temp->next=NULL;
                temp->prev=rear;
                rear->next=temp;
                rear=temp;
        }
        count++;
}
```

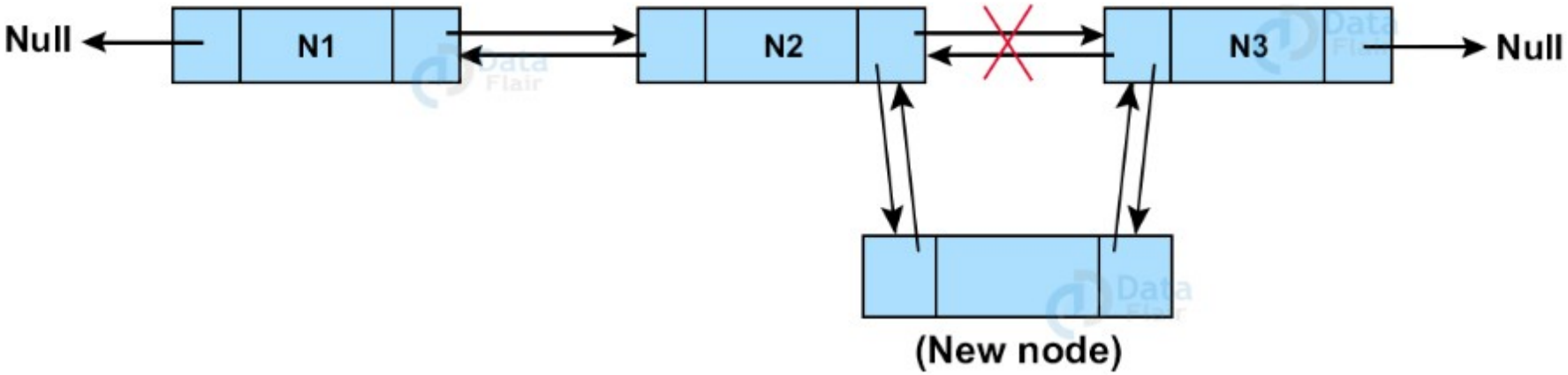


iii. Insertion after a given node(DLL)

Step-1: In this case, **use a temp variable** and **initialize** with the **head**.

Step-2: **Traverse till the given position - 1**

Step-3: **Insert the new node** by **adjusting** the **previous and next pointer**.



iii. Insertion after a given node (DLL)

```
//Doubly Linked list insert at end operation
void insert_at_anyposition()
{
        int pos,c=1;
        struct Node *temp=NULL,*p=NULL, *newnode=NULL;
        newnode=(struct Node*)malloc(sizeof(struct  Node));
         printf("\n Enter any value");
        scanf("%d",&newnode->data);
        printf("\n Enter the any position numer:");
        scanf("%d",&pos);
        temp=head;
        for(i=2;i<pos;i++)
        {
                if(temp->next!=NULL)
                {
                        p=temp;
                        temp-=temp->next;
                }

        }
```

```
p->next=newnode;
newnode->prev=p;
newnode->next=temp;
temp->prev=newnode;
count++;
}
```

i. Deletion at the beginning(DLL)

While deleting a node at the **from begining**, **three situation may arise:**

i. Linked List is empty:

Step-1:**Display Linked List is empty.**

ii. If only one node is present:

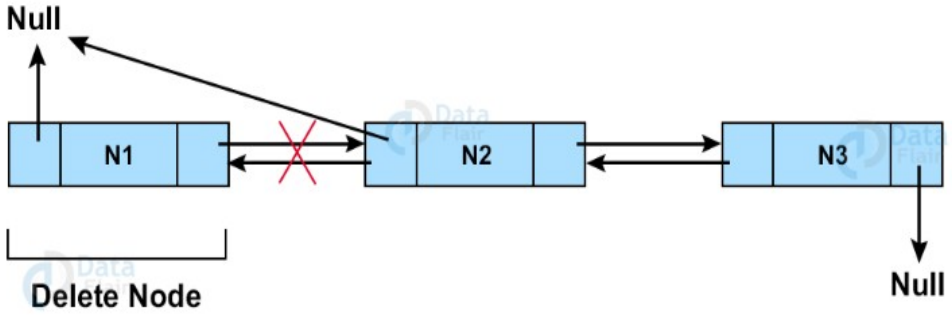
Step-1:In this case, **delete that one node** and **set head and rear** as **NULL.**

iii. If multiple nodes are present:

Step-1: Assume **present head as temp.**
Step-2: Then **shift the head to next node**(Second node).
Step-3: **Set the previous part** of **new head node** as **NULL.**
Step-4: Then **delete the temp node.**

i. Deletion at the beginning (DLL)

```
void del_at_begin()
{
 struct Node *temp;
 temp=head;
            if (head==NULL)
  {
    printf ("Linked List Empty");
    return;
  }
  else if(head->next==NULL)
            {
                        free(head);
                        count--;
            }
else
   {
            head=head->next;
            head->prev=NULL;
            free(temp);
            count--;
            printf("\nOne node deleted from Begining!!!\n");
   }
}
```



ii.Deletion at the end (DLL)

- While **deleting a node at the from the end, three situation may arise**:

i. Deletion when Linked List is empty:

Step-1 :In this case, **display list is empty.**

ii. If only one node is present:

Step-1 :**In this case, delete that one node** and **set head and rear** as **NULL**

iii. If multiple nodes are present:

Step-1:**Assume present(current) rear as temp.**

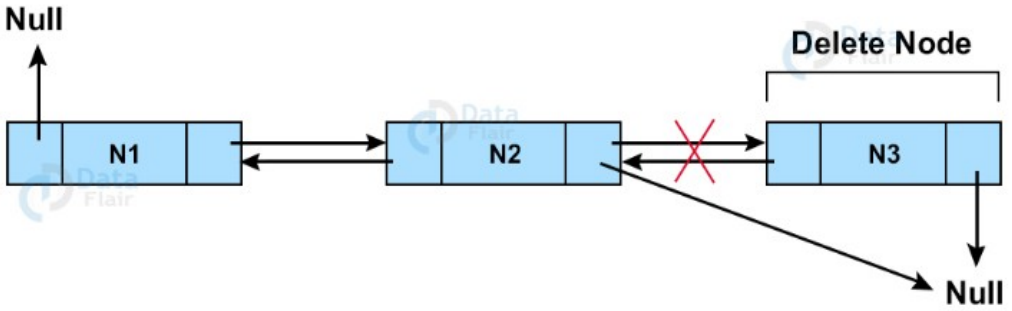
Step-2:Then **shift the rear node** to the **previous node**.

Step-3: **Set NULL** to the **next part of new rear node.**

Step-4: **Delete temp node.**

ii. Deletion at the end (DLL)

```
void del_at_end()
{
        if(head==NULL)
        {
        printf("\nList is Empty!!!\n");
        }
        else if(head->next==NULL)
        {
                free(head);
                count--;
        }
        else
        {
        struct Node *temp,*prev;
        temp=head;
        while(temp->next!=NULL)
         {
                prev=temp;
                temp=temp->next;
         }
         prev->next=NULL;
        free(rear);
        rear=temp1;
        count--;
        printf("\nOne node deleted from end!!!\n");
   }
}
```

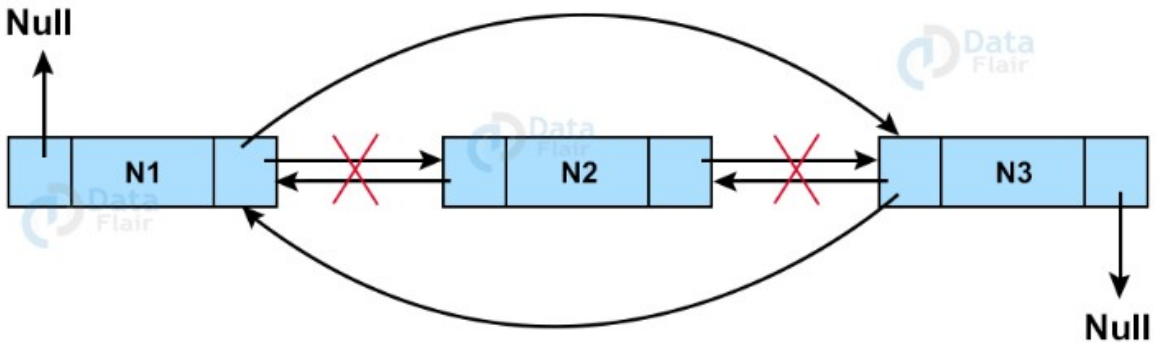


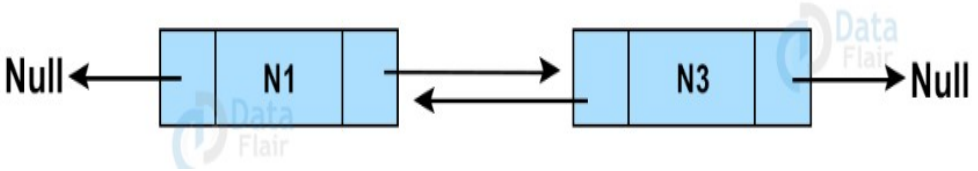
iii. Deletion at a given position (DLL)

Step-1:In this case, use a **temp variable** and **initialize with the head.**

Step-2:**Traverse till the given position.**

Step-3: **Adjust the next** and **previous pointers** of the **nodes situated after and before this node.**





iii. Deletion at a given position (DLL)

```
void del_at_anyposition()
{
        int pos,pr,i;
        struct Node *temp,*n,*succ=NULL;
        printf("\n Enter position");
        scanf("%d",&pos);
        pos=pos-1;
        temp=head;
         for(i=1;i<=count;i++)
        {
                if(i==pos)
                {
                        n=temp->next;
                        succ=temp->next->next;
                        temp->next=temp->next->next;
                        succ->prev=succ->prev->prev;
                        free(n);
                        count--;
                }
                else
                {
                        temp=temp->next;
                }
        }
}
```

Doubly linked list

Advantages of Doubly linked list:

i. It is the **easiest** data structures to **implement.**

ii. **Allows traversal** of nodes in **both direction**.

iii. **Deletion of nodes** is **easy** when **compared** to **singly linked list**

Disadvantages of Doubly linked list:

i. It **uses extra memory** when **compared to singly linked list**.

Difference Between Single Linked List and Double Linked List

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is O(n).	In a doubly-linked list, the time complexity for inserting and deleting an element is O(1).


T
H
A
N
'Q'