Module 2.0

Linear Data Structure:
Linked List

Memory Allocation & Deallocation for Linked List

Memory Allocation

There are two types of Memory Allocation

- **Compile Time or Static Allocation**
- **Runtime or Dynamic Allocation**

Compile Time or Static Allocation

- Memory allocated to the program element
  - **at the start of the program.**
- Memory allocated
  - **is fixed and determined by the compiler at compile time.**

- **Eg- float a[5]** ,

allocation of 20 bytes to the array, i.e. 5*4 bytes

Compile Time or Static Allocation

- **Inefficient Use of Memory-**
  - **Can cause under utilization of memory in case of over allocation**
    - That is if you store less number of elements than the number for elements which which you have declared memory.
    - Rest of the memory is wasted, as it is not available to other applications.

Compile Time or Static Allocation

- **Inefficient Use of Memory-**
  - **Can cause Overflow in case of under allocation**
    - For float a[5];
    - No bound checking in C for array boundaries, if you are entering more than fives values,
    - It will not give error but when these extra element will be accessed, the values will not be available.

Compile Time or Static Allocation

- No reusability of allocated memory
- Difficult to guess the exact size of the data at the time of writing the program

Runtime or Dynamic Allocation

- **All Linked Data Structures are preferably implemented through dynamic memory allocation.**
- Dynamic data structures provide flexibility in **adding, deleting or rearranging data objects at run time.**
- Managed in C through a set of library functions:
  - malloc()
  - Calloc()
  - free()
  - Realloc()

Runtime or Dynamic Allocation

- Memory space required by Variables is **calculated and allocated during execution**
- Get Required chunk of memory at Run time or As the need arises
- Best Suited –
  - **When we do not know the memory requirement in advance**, which is the case in most of the real life problems.

Runtime or Dynamic Allocation

- **Efficient use of Memory**
  - Additional Space can be allocated at run time.
  - Unwanted Space can be released at run time.

- **Reusability of Memory space**

The malloc() fn

- **Allocates a block of memory in bytes**
- The user should **explicitly give the block size** needed.
- **Request to the RAM of the system to allocate memory**,
  - If request is granted returns a pointer to the first block of the memory
  - If it fails, it returns NULL
- **The type of pointer is Void, i.e. we can assign it any type of pointer.**
- Available in header file alloc.h or stdlib.h

The mallloc() fn

- **Syntax-**

  ptr_var=(type_cast*)malloc(size)

- ptr_var = name of pointer that holds the starting address of allocated memory block
- type_cast* = is the data type into which the returned pointer is to be converted
- Size = size of allocated memory block in bytes

The mallloc() fn

Eg-

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

- Size of int=2bytes
- So 20 bytes are allocated,
- Void pointer is casted into int and assigned to ptr

Eg-

```
char *ptr;
ptr=(char *)malloc(10*sizeof(char));
```

The mallloc() fn- Usage in Linked List

Eg-

```
struct student;
{
        int roll_no;
        char name[30];
        float percentage;
};
struct student *st_ptr;
st_ptr=(struct student *)malloc(10*sizeof(struct student));
```

The callloc() fn

- Similar to malloc
  - It neds two arguments as against one argument in malloc() fn

Eg-

```
int *ptr;
ptr=(int *)calloc(10,2));
```

1st argument=no of elements

2nd argument=size of data type in bytes

- On initialization-
  - Malloc() contains garbage value
  - Calloc() contains all zeros
- Available in header file alloc.h or stdlib.h

Malloc() vs callloc() fn

Initialization:

- malloc() doesn't initialize the allocated memory.
  - If we try to access the content of memory block(before initializing) then we'll get segmentation fault error(or maybe garbage values).
- calloc() allocates the memory and also initializes the allocated memory block to zero.
  - If we try to access the content of these blocks then we'll get 0.

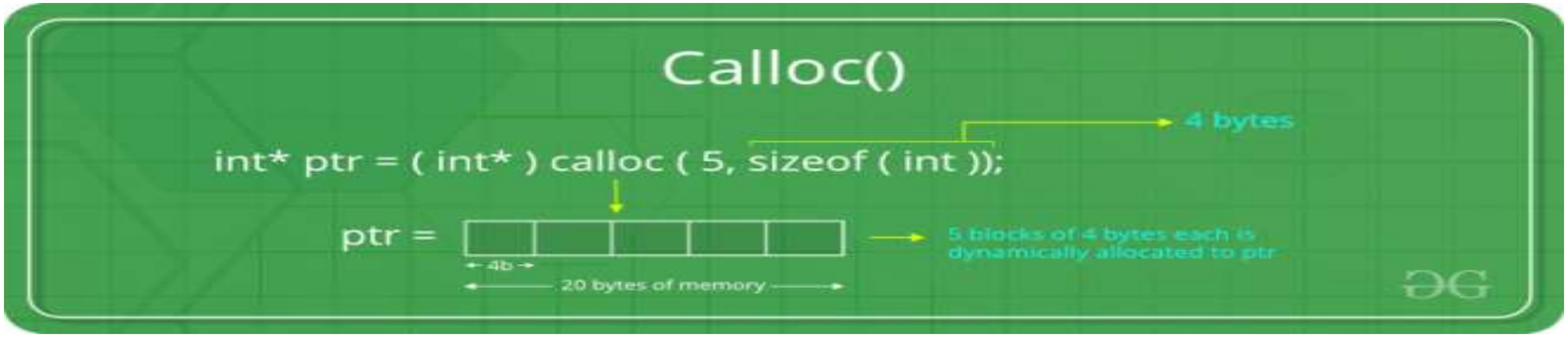
Malloc() vs callloc() fn

Malloc

- allocate a single large block of memory with the specified size.

Calloc

- allocate multiple blocks of memory
- dynamically allocate the specified number of blocks of memory of the specified type.



The free() fn

- Used to deallocate the previously allocated memory using malloc or calloc() fn

- **Syntax-**

  free(ptr_var);

- ptr_var is the pointer in which the address of the allocated memory block is assigned.

- Returns the allocated memory to the system RAM..

What happens when you don’t free memory after using malloc()

What happens when you don’t free memory after using malloc()

- The memory allocated using malloc() is not de-allocated on its own.
- So, “free()” method is used to de-allocate the memory.
- But the free() method is not compulsory to use.
- If free() is not used in a program
  - the memory allocated using malloc() will be de-allocated
  - after completion of the execution of the program
  - (included program execution time is relatively small and the program ends normally).

What happens when you don’t free memory after using malloc()

- Still, there are some important reasons to free() after using malloc():
  - Use of free after malloc is a good practice of programming.
  - There are some programs like a digital clock, a listener that runs in the background for a long time and there are also such programs which allocate memory periodically.

What happens when you don't free memory after using malloc()

- Still, there are some important reasons to free() after using malloc():

  - In these cases, even small chunks of storage add up and create a problem.
  - Thus our usable space decreases. This is also called "memory leak".
  - It may also happen that our system goes out of memory if the de-allocation of memory does not take place at the right time.

The realloc() fn

- To resize the size of memory block, which is already allocated using malloc.

- Used in two situations:
  - If the allocated memory is insufficient for current application
  - If the allocated memory is much more than what is required by the current application

The realloc() fn

- Syntax-

  ptr_var=realloc(ptr_var,new_size);

- ptr_var is the pointer holding the starting address of already allocated memory block.
- Available inn header file<stdlib.h>
- **Can resize memory allocated previously through malloc/calloc only.**

Eg of mallloc() fn

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *mem_allocation;
   /* memory is allocated dynamically */
   mem_allocation = malloc( 20 *
sizeof(char) );
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate
requested memory\n");
   }
   else
   {
     strcpy(
mem_allocation,"fresh2refresh.com");
   }
printf("Dynamically allocated memory
content  : " \
        "%s\n", mem_allocation );
   mem_allocation=realloc(mem_alloc
ation,100*sizeof(char));
   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate
requested memory\n");
   }
   else
   {
      strcpy( mem_allocation,"space is
extended upto " \
                         "100 characters");
   }
   printf("Resized memory : %s\n",
mem_allocation );
   free(mem_allocation);
}
```

Which part of the memory is allocated when static memory allocation is used, for int,char,float,arrays,struct,unions…..?

Which part of the memory is allocated when malloc and calloc are called for any variable?

- ??

Stack

- Basic type variables such as int, double, etc, and complex types such as arrays and structs. These variables are put on the **stack** in C.
- There is a limit (varies with OS) on the size of variables that can be stored on the stack.
- This is not the case for variables allocated on the **heap**

https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixprggd/genprogc/sys_mem_alloc.htm

Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- It is a more free-floating region of memory (and is larger).
- Dynamic memory allocation functions allocates memory from the heap.

Heap

- To allocate memory on the heap, you must use malloc() or calloc(), which are built-in C functions.
- Deallocate memory on heap using free
- If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes).

Stack vs Heap

Stack –

- Don't have to explicitly de-allocate variables
- The stack has size limits
- very fast access
- variables cannot be resized
- local variables only

Heap –

- You must manage memory (you're in charge of allocating and freeing variables)
- The heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Heap memory is slightly slower to be read from and written to, because **pointers** are used to access memory on the heap.
- Variables can be resized using realloc()
- variables can be accessed globally

https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

Stack

- Stack resident variables are:
  - created ("pushed on to") the stack when a basic block that contains them (i.e. function) is activated and
  - disappear from ("popped off") the stack when the function is done
    - (Those areas of memory that had been used by variables when a basic block was activated may then be overwritten after exit from the block.)
  - stack memory is more limited, i.e. there is less of it compared to heap memory

http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=13D3E255880D543426E53785A6A7895F?doi=10.1.1.705.51&rep=rep1&type=pdf

Heap

- Heap resident variables include:
  - variables declared outside all functions (globals)
  - variables declared inside basic blocks that are declared static
  - memory areas dynamically allocated at run time with malloc( )

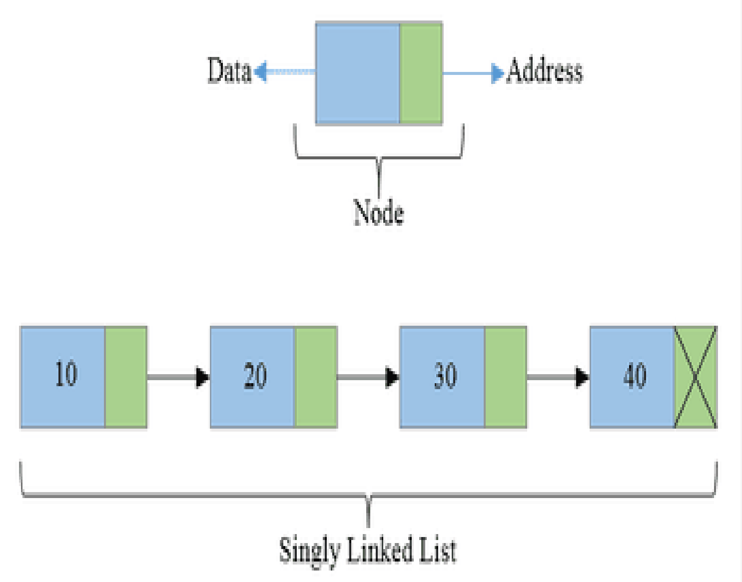
Storage for declared heap resident variables is:

- assigned at the time a program is loaded, and
- remains assigned for the life of a program

Linked List

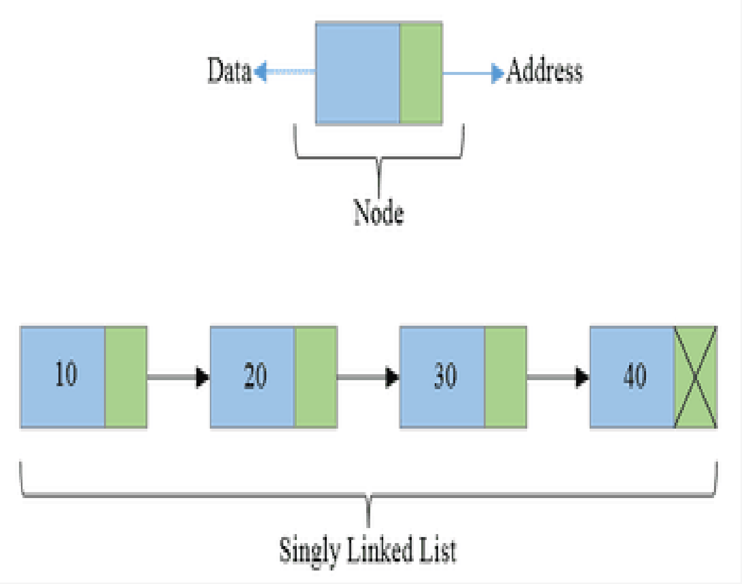
Linked Lists

- Linear Collection of data elements called Nodes
  - where the linear order is given by means of pointers.



Linked Lists

- Each node may be divided into atleast two fields for :
    - Storing Data
    - Storing Address of next element.
- The Last node's Address field contains Null rather than a valid address.
    - It's a NULL Pointer and indicates the end of the list.



Comparison between Array and Linked List

Advantages of Linked List

- **Linked are Dynamic Data Structures**
  - Grow and shrink during execution of the program
- **Efficient Memory Utilization**
  - As memory is not preallocated.
  - Memory can be allocated whenever required and deallocated when not needed.
- **Insertion and deletions are easier and efficient**
  - Provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position
- **Many complex applications can be easily carried out with linked lists**

Disadvantages of Linked List

- Access to an arbitary data item is little bit cumbersome and also time consuming

- **More memory**
  - If the number of fields are more, then more memory space is needed.

Advantages of Arrays

- Simple to use and define
- Supported by almost all programming languages
- **Constant access time**
  - Array element can be accessed a[i]
- Mapping by compiler
  - Compiler maps a[i] too its physical location in memory.
  - **This mapping is carried out in constant time, irrespective of which element is accessed**

Disadvantages of Arrays

Arrays suffer from some severe limitations

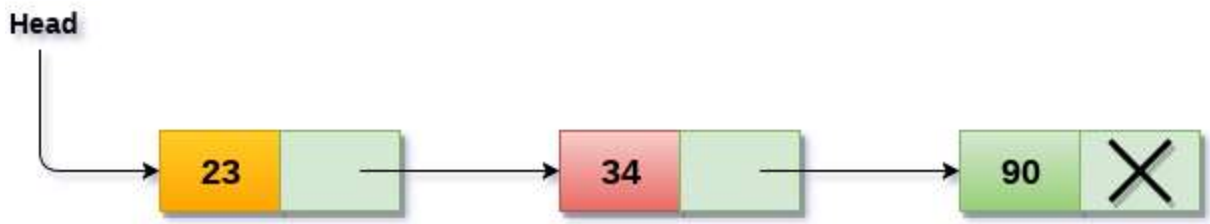
- **Static Data Structure-**
  - **Size of an array is defined at the time of programming**
- **Insertion and Deletion is time consuming**
- Requires Contiguous memory

Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Linked List

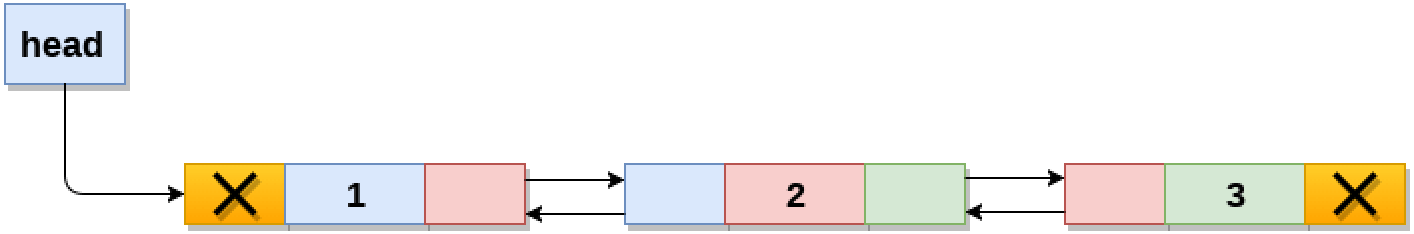
Singly Linked List

- All nodes are linked in sequential manner
- Linear Linked List
- One way chain
- It has beginning and end
- Problem-
  - The predecessor of a node cannot be accessed from the current node.
  - This can be overcome in doubly linked list.



Doubly Linked List

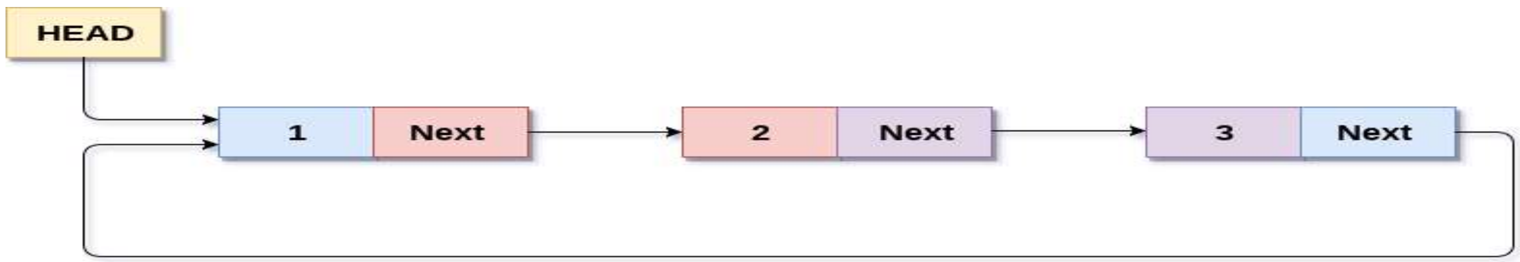
- Linked List holds two pointer fields
- Addresses of next as well as preceding elements are linked with current node.
- This helps to traverse in both Forward or Backward direction



Doubly Linked List

Circular Linked List

- The first and last elements are adjacent.
- A linked list can be made circular by
  - Storing the address of the first node in the link field of the last node.



Circular Singly Linked List

Linked List Operations

- Creation
- Insertion
- Deletion
- Traversal
- Searching

Implementation of Linked Lists

- Structures in C are used to define a node
- Address of a successor node can be stored in a pointer type variable

Linked Lists

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

Information field

Pointer that points to the structure itself,
Thus Linked List.

Singly Linked List

Creation of a new node

```
struct node{
                type1 member1;
                type2 member2;
                ..................
                ..................
                struct node *link;
            };
```

Info part

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


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

Creation of a new node

New node=temp

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

Creating a Linked List

```
create_list(int data)
{
        struct node *q,*tmp;
        tmp= (struct node *) malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=NULL;

        if(start==NULL) /*If list is empty */
        {
                start=tmp;
        }
        else
        {    /*Element inserted at the end */
                q=start;
                while(q->link!=NULL)
                        q=q->link;
                q->link=tmp;
        }
}/*End of create_list()*/
```

Traversing a Linked List

- Assign the Value of start to another pointer say q

  struct node *q=start;

- Now q also points to the first element of linked list.

- For processing the next element, we assign the address of the next element to the pointer q as-

  q=q->link;

- Traverse each element of the Linked list through this assignment until pointer q has NULL address, which is link part of last element.

```
while(q!=NULL)
 {
         q=q->link;
 }
```

Searching a Linked List

- First traverse the linked list
- While traversing compare the info part of each element with the given element

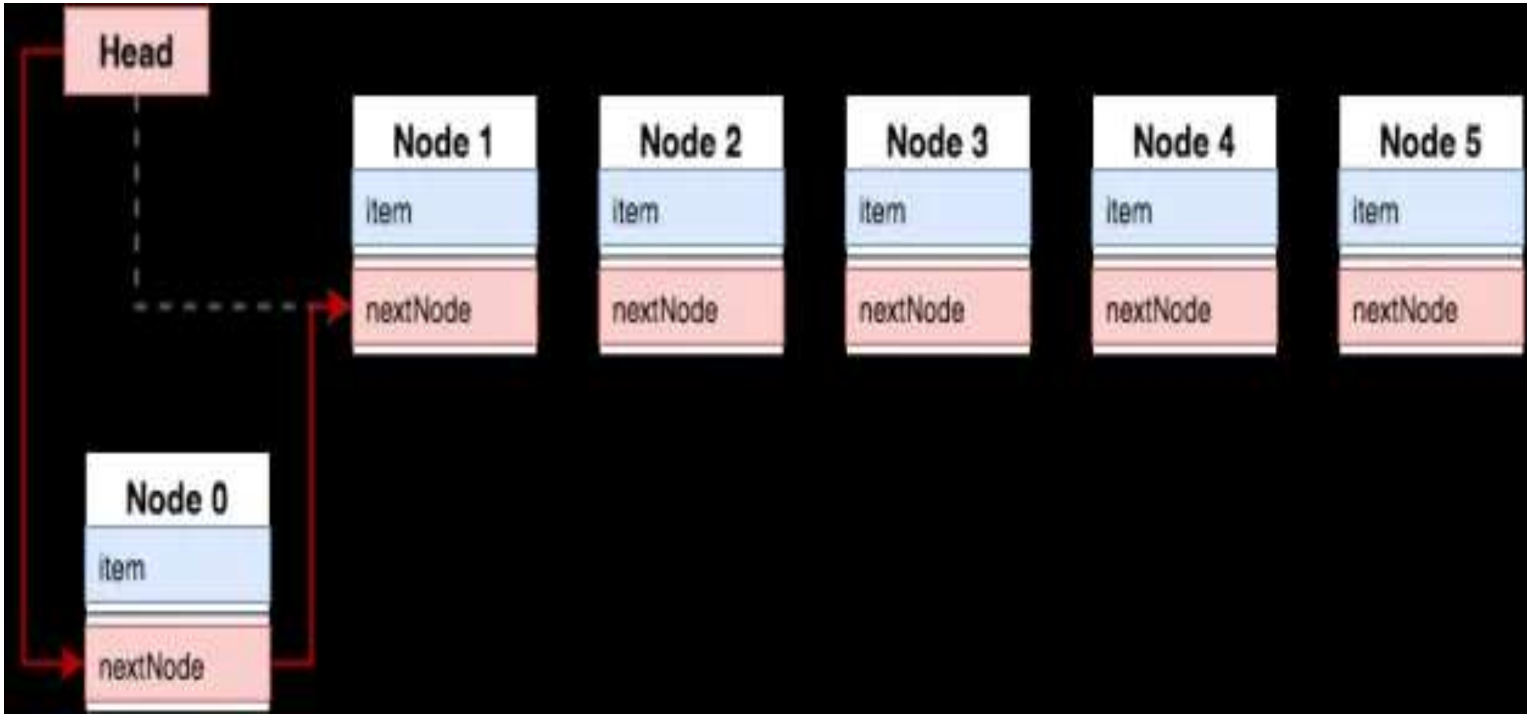
Searching a Linked List

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

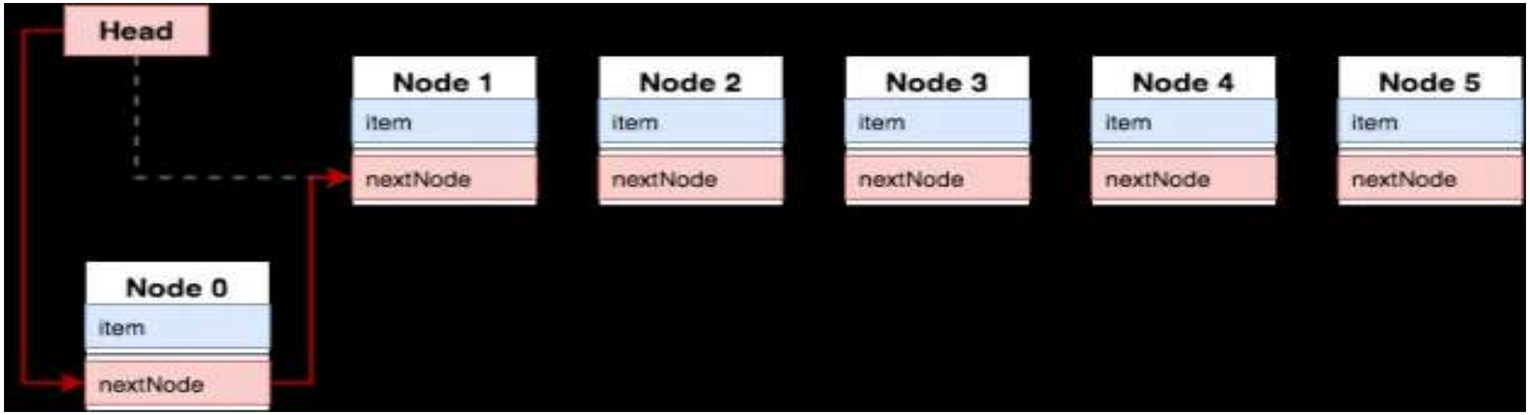
Insertion into a Linked List

- Insertion is possible in two ways:
  - Insertion at Beginning
  - Insertion in Between

Case 1- Insertion at Beginning



Case 1- Insertion at Beginning



CREATE THE NEW NODE, CONNECT NEW NODE TO THE OLD FIRST NODE CONNECT THE START POINTER TO THE NEW NODE,

……

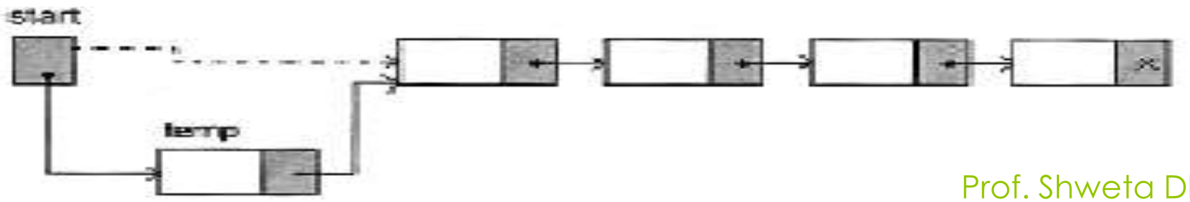
Case 1- Insertion at Beginning

- Lets say tmp is the pointer which points to the node that has to be inserted
- Assign data to the new node

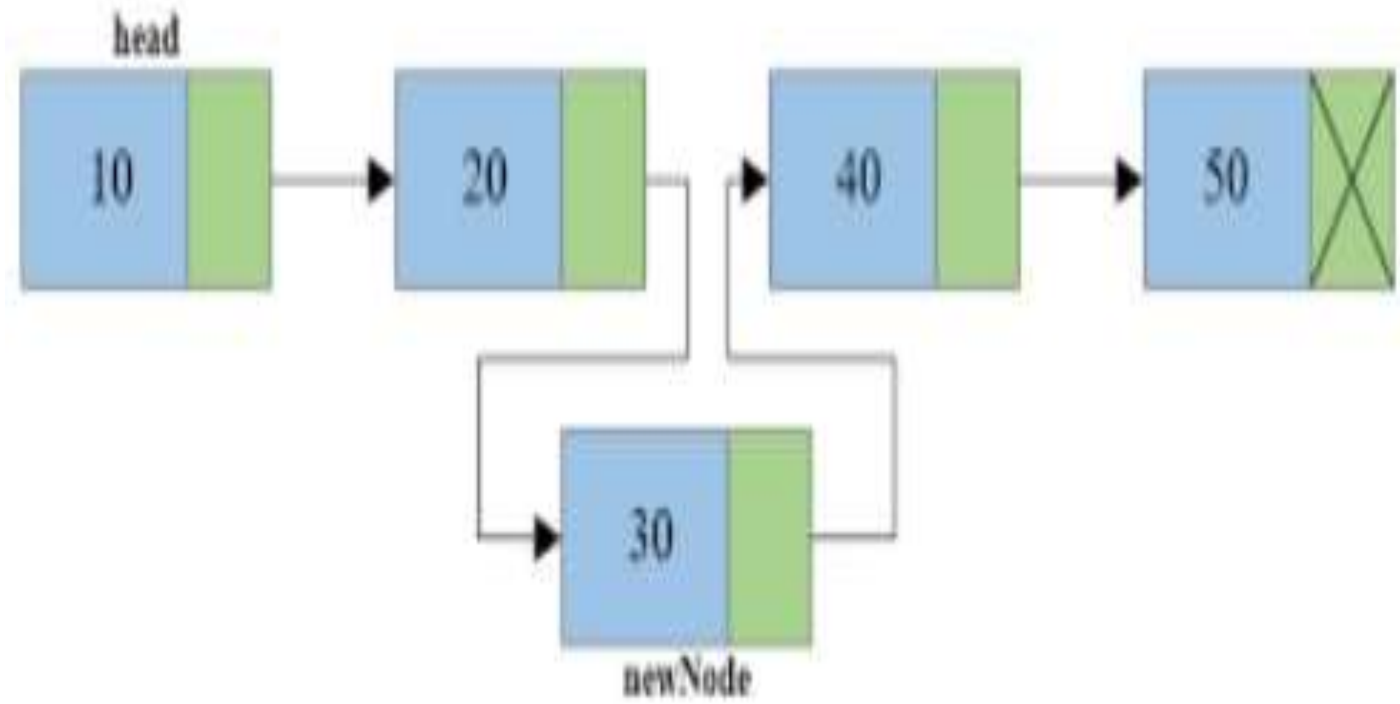
  tmp->info=data;
- Start points to the first element of linked list
- Assign the value of start to the link part of the inserted node as

  tmp->link=start;
- Now inserted node points beginning of the linked list.
- To make the newly inserted node the first node of the linked list:

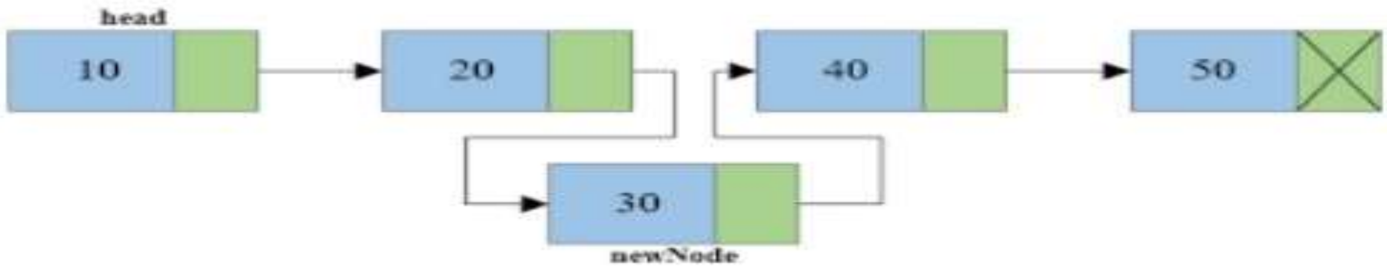
  start=temp



Case 2- Insertion in Between



Case 2- Insertion in Between



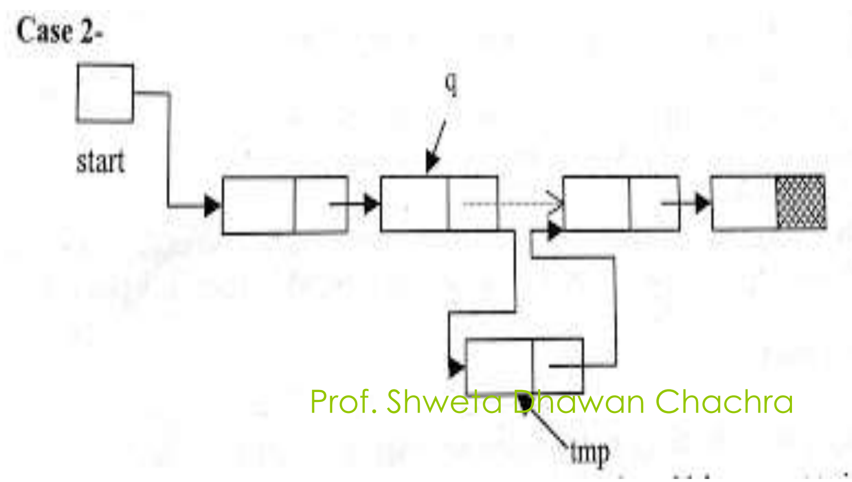
CREATE THE NEW NODE ,
CONNECT THE NEW NODE TO THE NEXT NODE
CONNECT THE PREVIOUS TO THE NEW NODE,

……

Case 2- Insertion in Between

- First we traverse the linked list for obtaining the node after which we want to insert the element

```
q=start;
for(i=0;i<pos-1;i++)
    {
            q=q->link;
            if(q==NULL)
            {
                    printf("There are less than %d elements",pos);
                    return;
            }
    }/*End of for*/
```



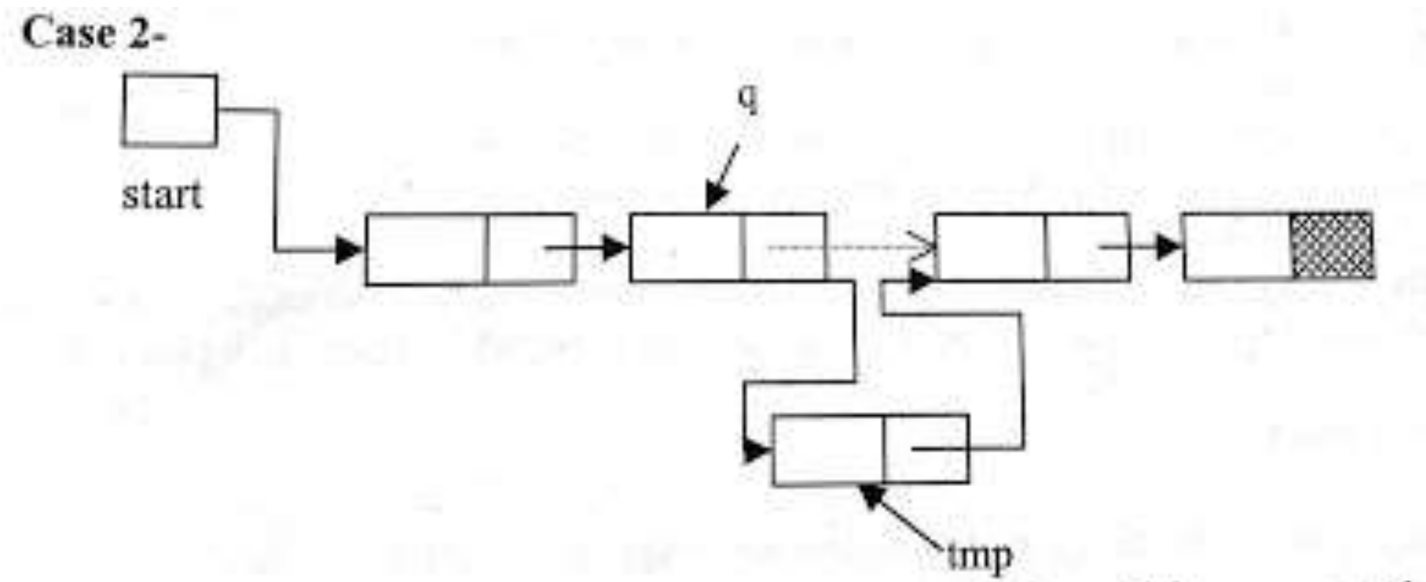
Case 2- Insertion in Between

- Then we add the new node by adjusting address fields

  tmp->info=data;

  tmp->link=q->link;

  q->link=tmp;

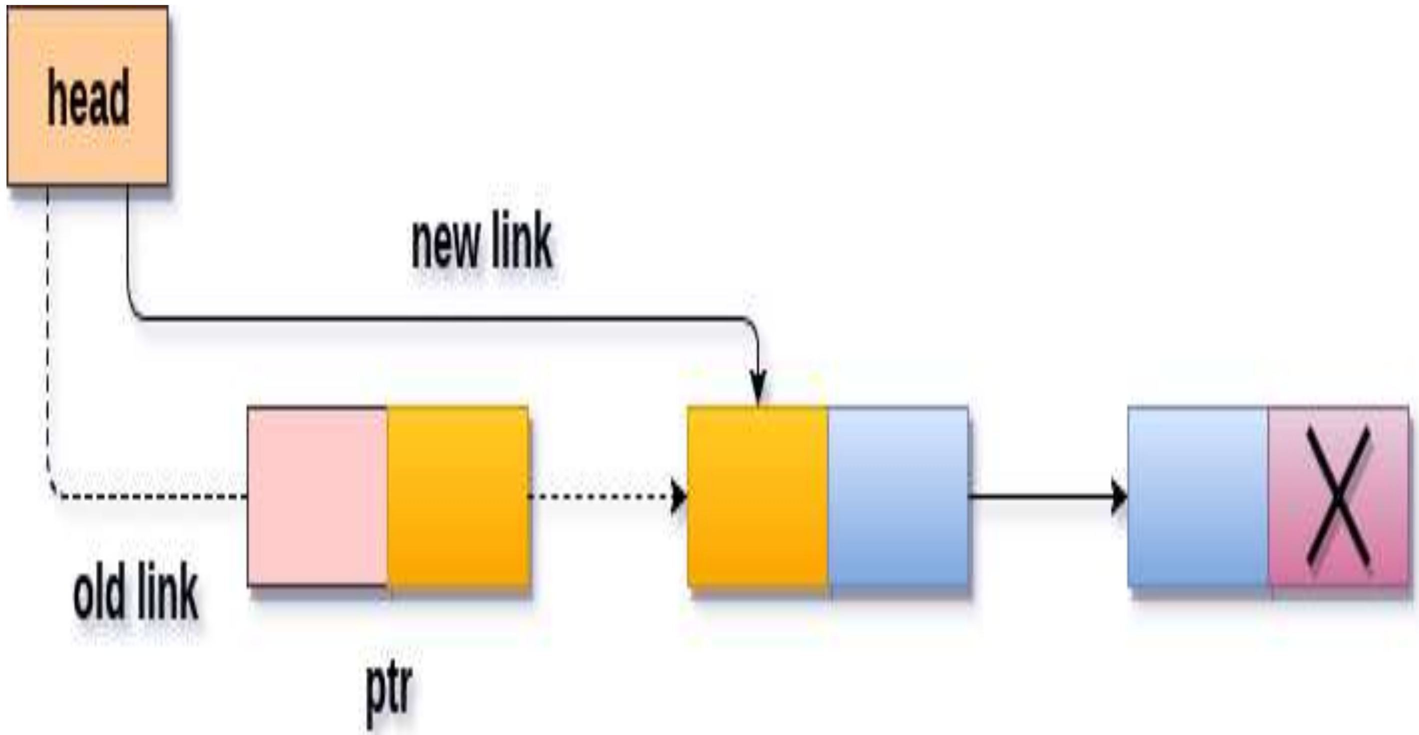


Deletion from a Linked List

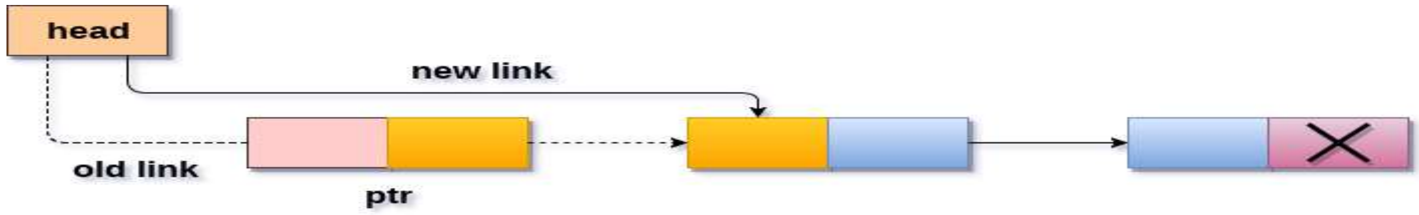
- For deleting the node from a linked list, first we traverse the linked list and compare with each element.

- After finding the element there may be two cases for deletion-
  - **Deletion in beginning**
  - **Deletion in between**

Deletion in beginning



Deletion in beginning



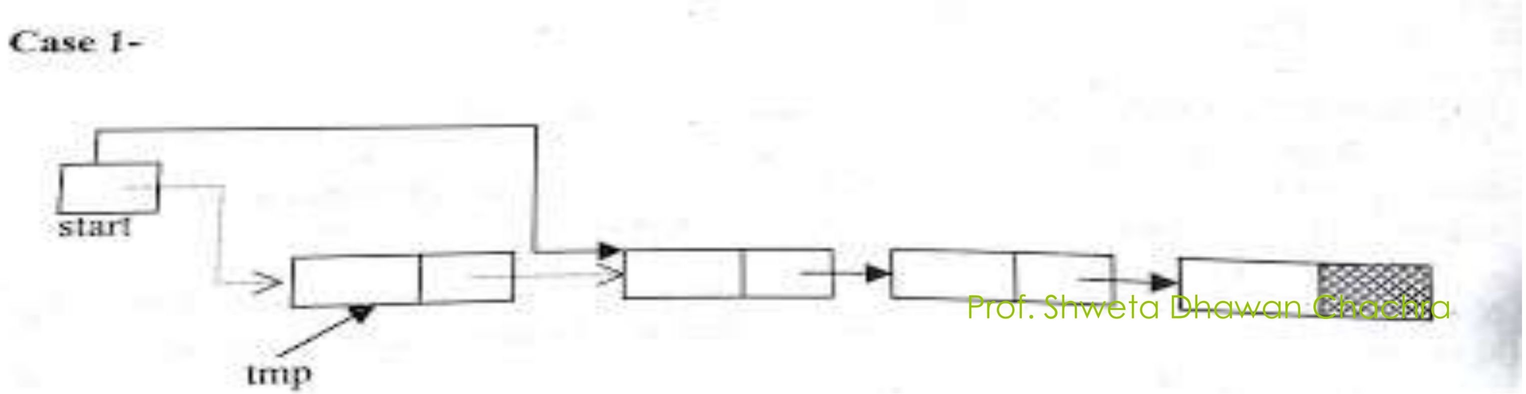
DELETE THE FIRST NODE AND CONNECT START POINTER TO THE SECOND NODE……

Deletion in beginning

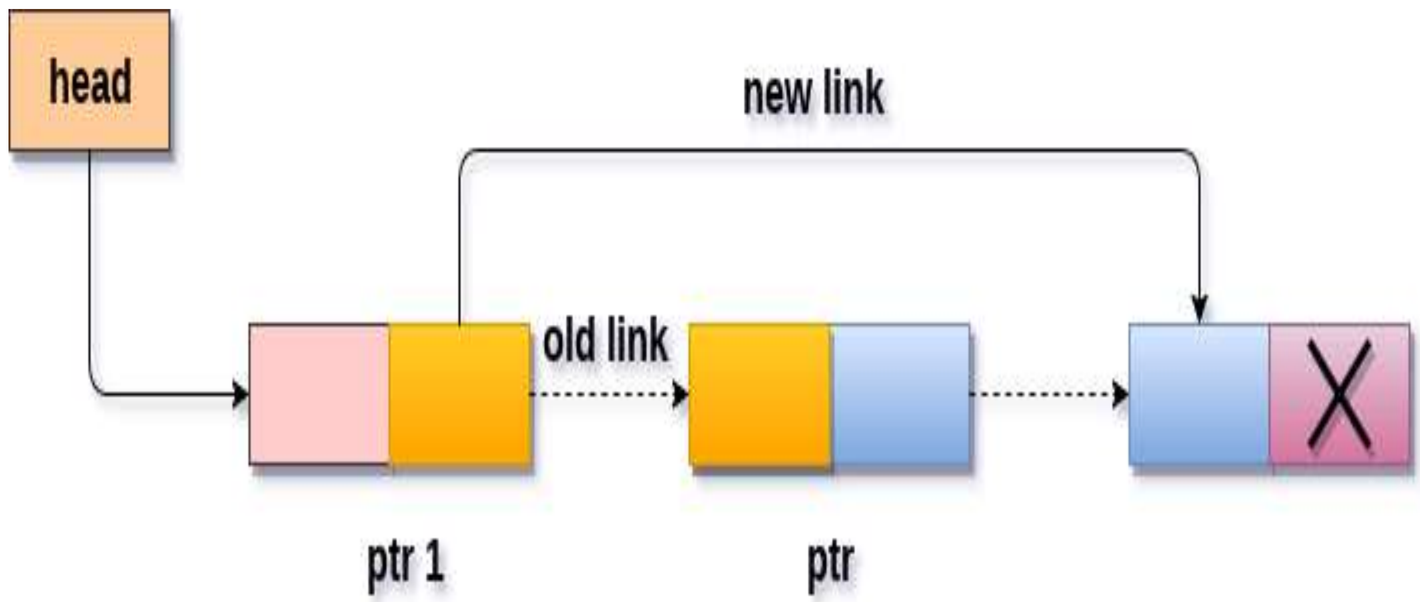
- Start points to the first element of linked list.
- If element to be deleted is the first element of linked list then we assign the value of start to tmp as-

  tmp = start;
- So tmp points to the first node which has to be deleted.
- Now assign the link part of the deleted node to start as-

  start=start->link;
- **Since start points to the first element of linked list, so start->link will point to the second element of linked list.**
- Now we should free the element to be deleted which is pointed by tmp. **free(tmp);**



Deletion in between



Deletion a node from specified position

Deletion in between

DELETE THE NODE AND CONNECT THE PREVIOUS AND THE NEXT NODE……

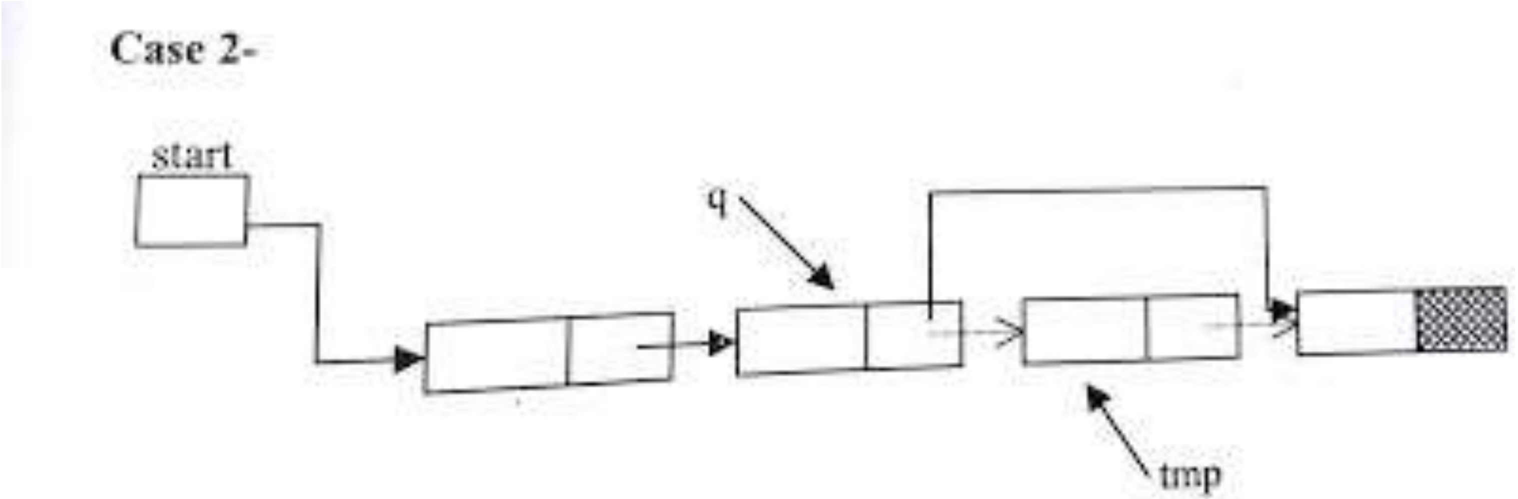
Deletion in between

- If the element is other than the first element of linked list then
  - **we give the link part of the deleted node to the link part of the previous node.**
  - This can be as-

tmp =q->link;

q->link = tmp->link;

free(tmp);



Deletion in between

- If node to be deleted is last node of linked list then statement 2 will be as-

```
tmp =q->link;
q->link = NULL;
free(tmp);
```

Complexity of operations in Linked List:

- **Access/Traverse**
  - It is not possible to have a constant access time in linked list operations.
  - The data required may be at the other end of the list and the worst case may be to traverse the whole list to get it.
  - The time complexity is hence **O(n)**.
- **Arrays-**
  - **It's just O(1) for array because of constant access time using a[i]**

Complexity of operations in Linked List:

Insertion

- Insertion in a linked list involves only manipulating the pointers of the previous node and the new node, provided we know the location where the node is to be inserted.
- Thus, the insertion of an element is **O(1)**.

Deletion

- Similar to insertion, deletion in a linked list involves only manipulating the pointers of the previous node and freeing the new node, provided we know the location where the node is to be deleted.
- Thus, the deletion of an element is **O(1)**.

- In order to delete a node and connect the previous and the next node together, you need to know their pointers.

Complexity of operations in Linked List:

- **Insertion and deletion at a known position is O(1).**
- However, **finding that position is O(n), unless it is the head (Singly linked list) or tail of the list(Circular Linked list).**
- **In a doubly-linked list, both next and previous pointers are available in the node that is to be deleted. The time complexity is constant in this case, i.e., O(1)**
- **When we talk about insertion and deletion complexity, we generally assume we already know where that's going to occur.**

Time complexity for Search Operation in Singly Linked List

- **Best Case-**
  - Element found in the first node, while loop executes only once so **O(1)**
- **Worst Case-**
  - Element present in the last node so while node will work n times so **O(n)**

Courtesy of https://www.youtube.com/watch?v=lWEBpaVPoJA

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

Courtesy of https://www.bigocheatsheet.com/

Circular Linked List

Why Circular?

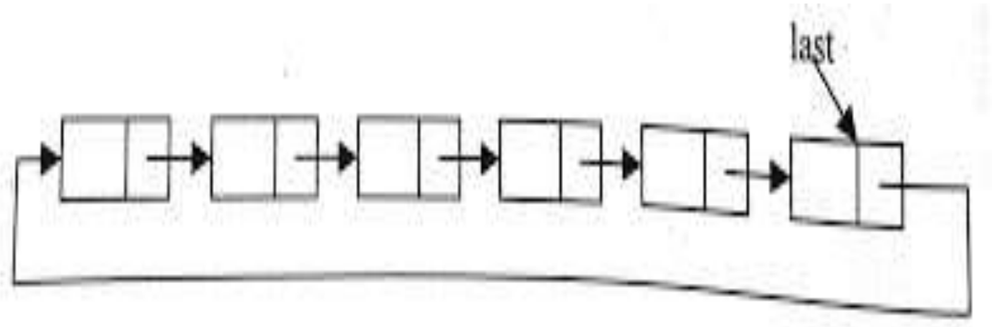
- In a singly linked list,
  - **If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node.**
  - This problem can be solved by slightly altering the structure of singly linked list.

How?

- In a singly linked list, next part (pointer to next node) of the last node is NULL,
  - if we utilize this link to point to the first node then we can reach preceding nodes.

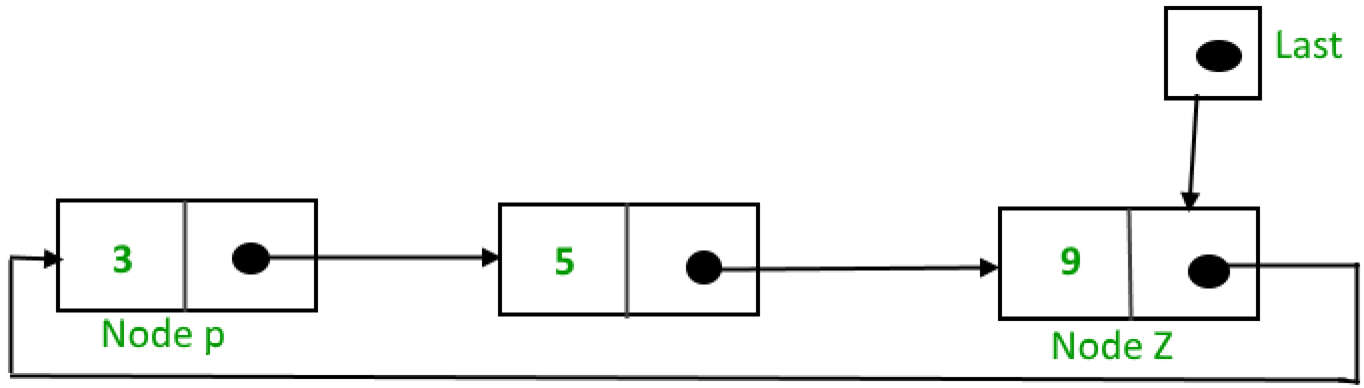
Implementation of circular linked list

- Creation of circular linked list is same as single linked list.
- **Last node will always point to first node instead of NULL.**



Implementation of circular linked list

- One pointer last,
  - which points to last node of list and link part of this node points to the first node of list.



Advantages of a Circular linked list

- In circular linked list, **we can easily traverse to its previous node**, which is not possible in singly linked list.
- **Entire list can be traversed from any node.**
  - If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node.

Advantages of a Circular linked list

- In Single Linked List, for insertion at the end, the whole list has to be traversed.
- In Circular Linked list,
  - **with pointer to the last node there won't be any need to traverse the whole list.**
  - So insertion in the begging or at the end takes constant time irrespective of the length of the list i.e O(1).
  - **It saves time when we have to go to the first node from the last node.**
    - **It can be done in single step because there is no need to traverse the in between nodes**

Disadvantages of Circular linked list

- Circular list are complex
  - as compared to singly linked lists.
- **Reversing of circular list is a complex**
  - **as compared to singly or doubly lists.**
- If not traversed carefully,
  - then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also **doesn't supports direct accessing of elements.**

Insertion into a circular linked list :-

Insertion in a circular linked list may be possible in two ways-

- ***Insertion in an empty list***
- **Insertion at the end of the list**
- ***Insertion at beginning***
- ***Insertion in between***

Insertion in an empty list

New element can be added as-

- *If linked list is empty:*

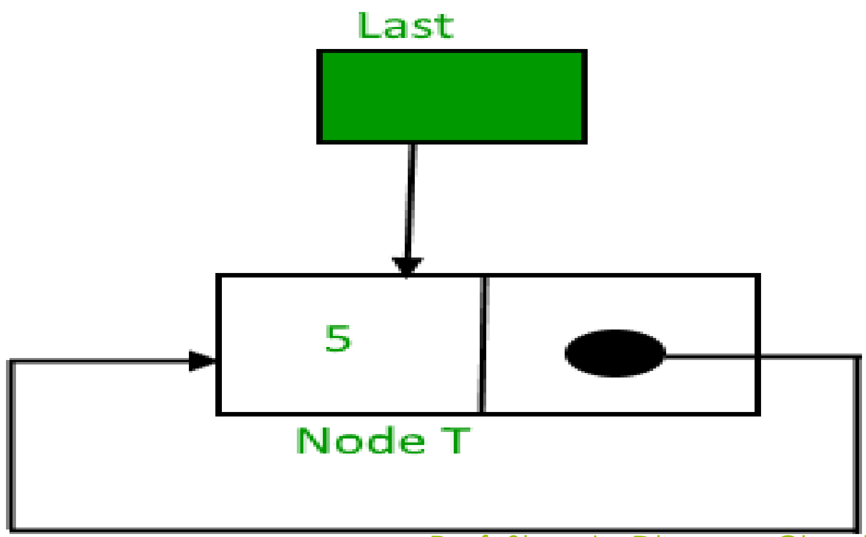
If (last==NULL)

{

last=tmp;

tmp->link=last

}



Insertion at the end of circular linked list

- *If linked list is not empty:*

Insertion at the end of the list

{

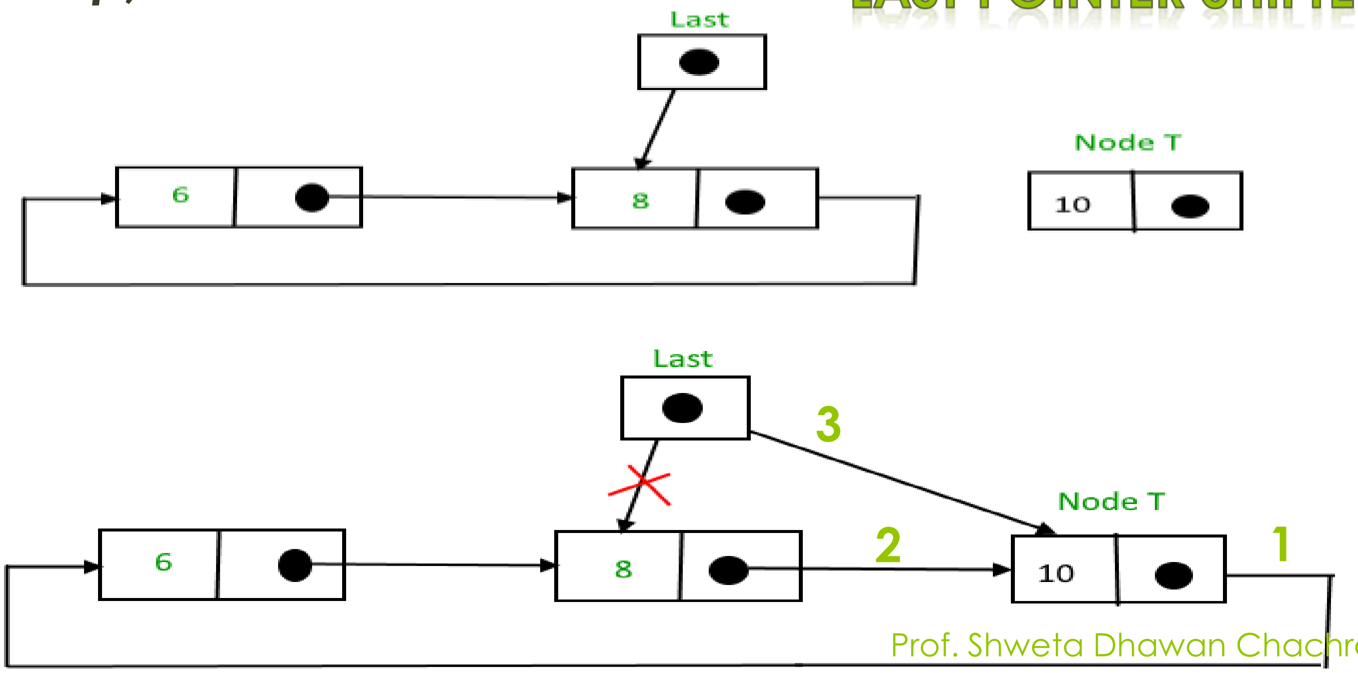
tmp->link = last->link; /* added at the end of list*/

last->link = tmp;

last = tmp;

}

LAST POINTER SHIFTED



Insertion at the beginning of circular linked list

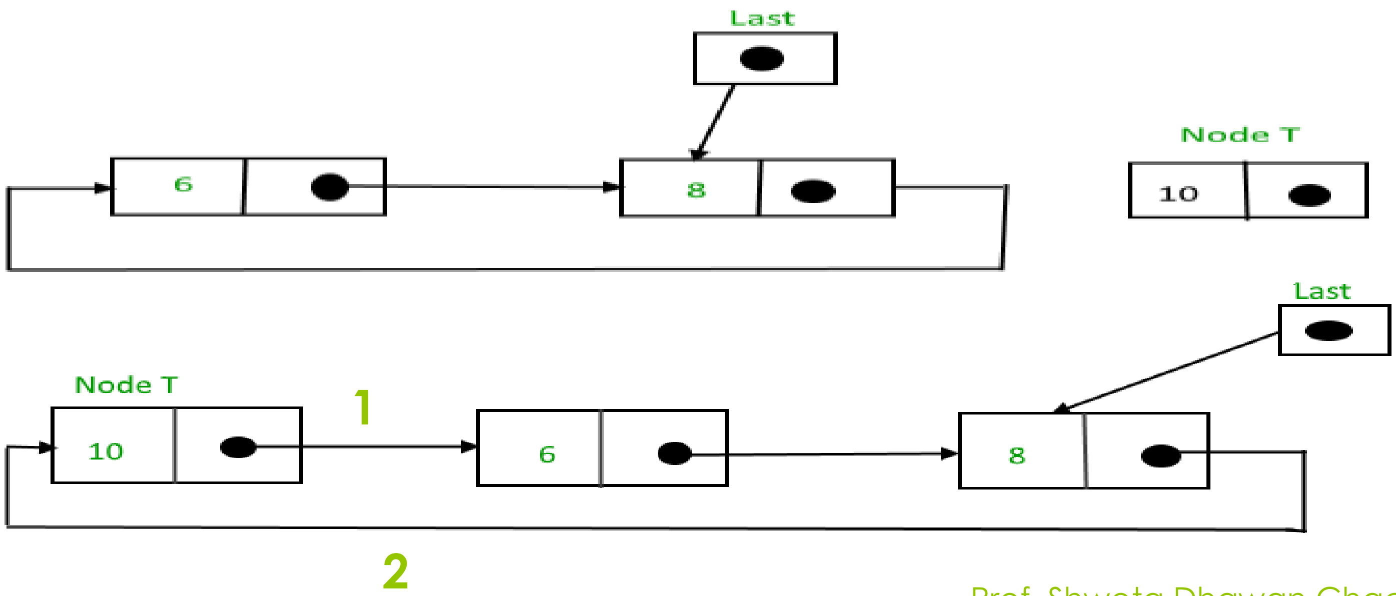
- *If linked list is not empty:*

Insertion at the beginning of the list

Follow these step:

1. Create a node, say tmp.
2. Make tmp-> next = last -> next.
3. last -> next = tmp.

LAST POINTER NOT SHIFTED !!



Insertion in between of circular linked list

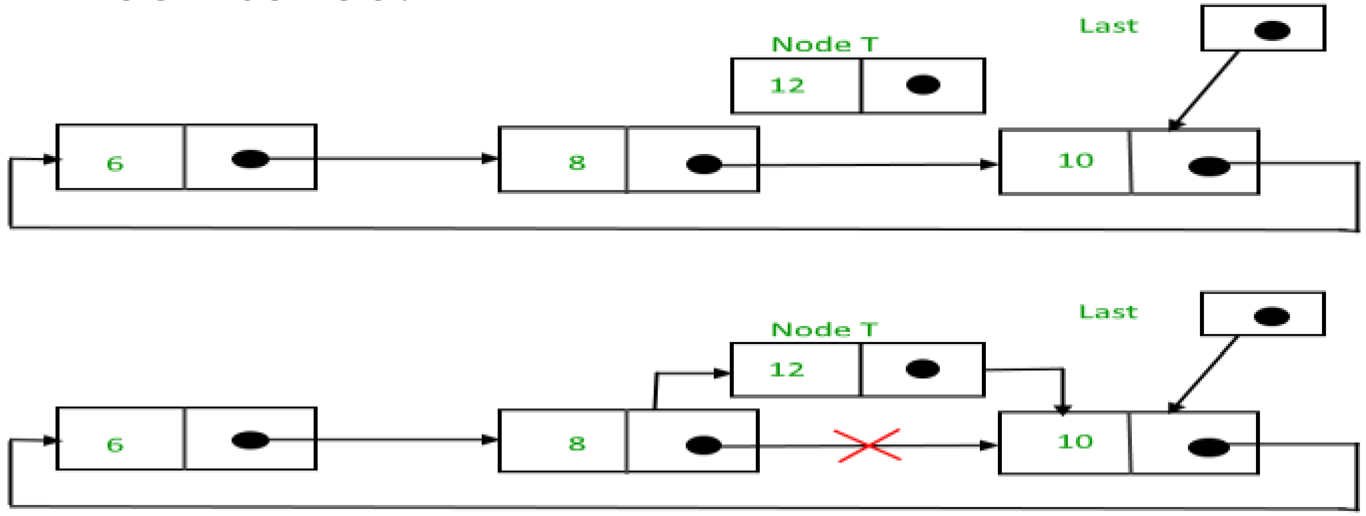
- Insertion in between is same as in single linked list. This can be as-

tmp->link = q->link;

tmp->info = num;

q->link =tmp;

- Here q points to the node after which new node will be inserted.



Traversal of circular linked list

- *Pointer variable which points to first node of list is needed.*
  - *Here we maintain the pointer last which points to last node.*
  - ***But link part of this last pointer points to first node of list,***
  - ***So we can assign this value to pointer variable which will point to first node.***
- *Now we can traverse the list until the last node of list comes-*

```
q = last->link;
while(q != last)
{
        ..........
        q = q->link;
}
```

Deletion from a circular linked list :-

Deletion in a circular linked list may be possible in four ways-

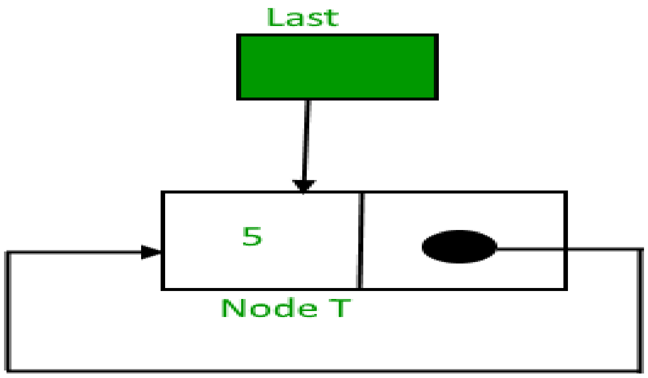
- ***If list has only one element***
- ***Node to be deleted is the first node of list***
- ***Deletion in between***
- ***Node to be deleted is last node of list***

Deletion from a circular linked list :-

Case 1:- ***If list has only one element***

- Here we check the condition for only one element of list
- then assign NULL value to last pointer because after deletion no node will be in list.

```
if(last->link == last && last->info == num) /* Only one element */
{
     tmp = last;
    last = NULL;
    free(tmp);
}
```

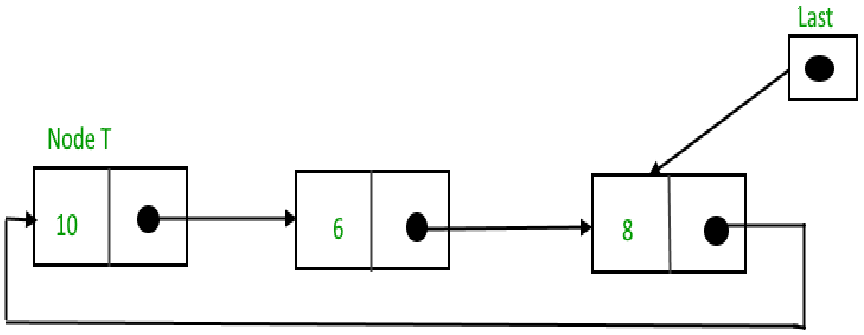


Deletion from a circular linked list :-

Case 2:- ***Node to be deleted is the first node of list***

- Assign the link part of deleted node to the link part of pointer last.
- So now link part of last pointer will point to the next node which is now first node of list after deletion.

```
q = last->link; /*q is pointing to the first node of list*/
if(q->info == num)
{
        tmp = q;
        last->link = q->link;
        free(tmp);
}
```



Deletion from a circular linked list :-

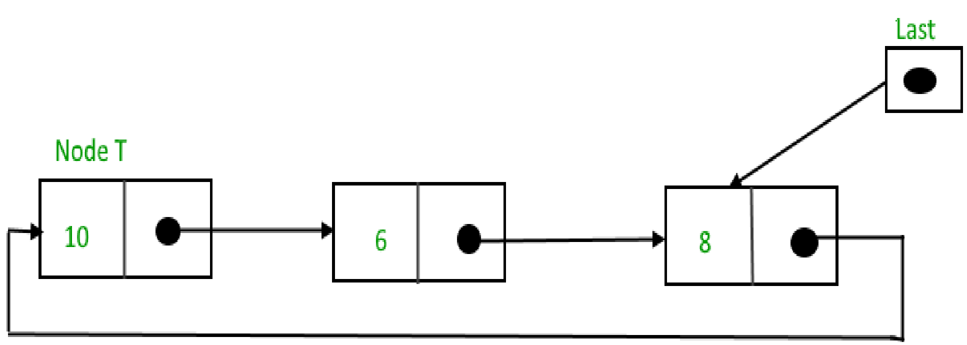
Case 3:- **Deletion in between is same as in single linked list.**

Deletion of node in between will be as-

```
q = last->link;
while(q->link != last)
{
        if(q->link->info == num) /*
Element deleted in between */
        {
                tmp = q->link;
                q->link = tmp->link;
                free(tmp);
        }
        q = q->link;
} /* End of while */
```

- First we are traversing the list, when we find the element to be deleted,
- then q points to the previous node.
- We assign the link part of node to be deleted to the link part of previous node and
- then we free the address of node to be deleted from memory.

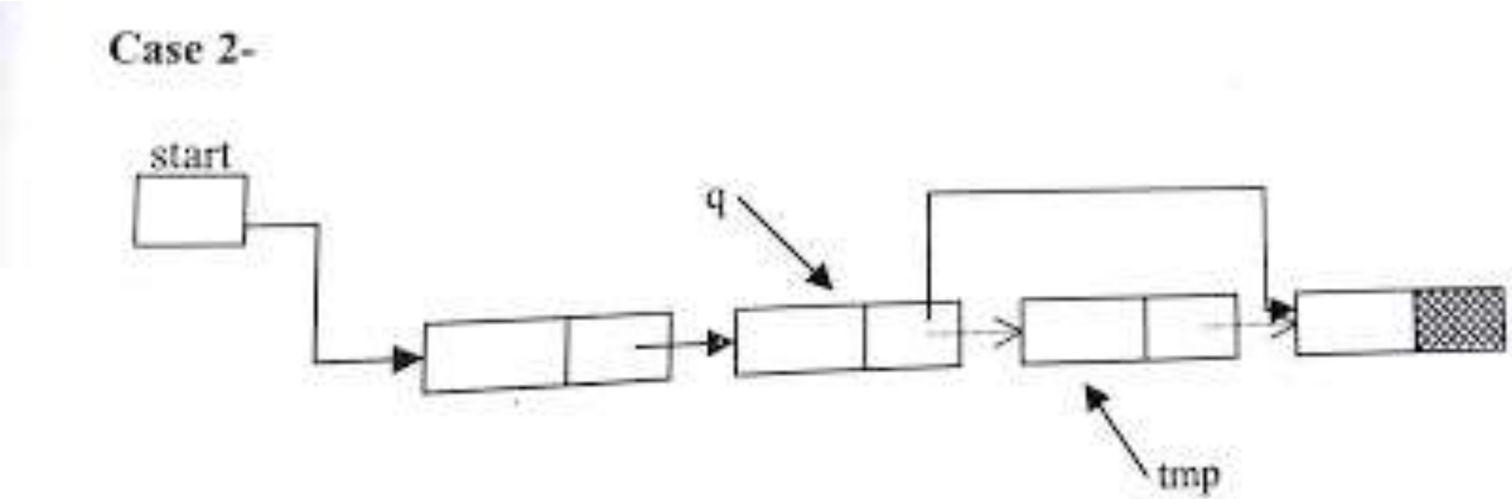
As we need to stop at the previous node for deletion



Deletion in between in Singly Linked List

- If the element is other than the first element of linked list then
  - **we give the link part of the deleted node to the link part of the previous node.**
  - This can be as-

```
tmp =q->link;
q->link = tmp->link;
free(tmp);
```

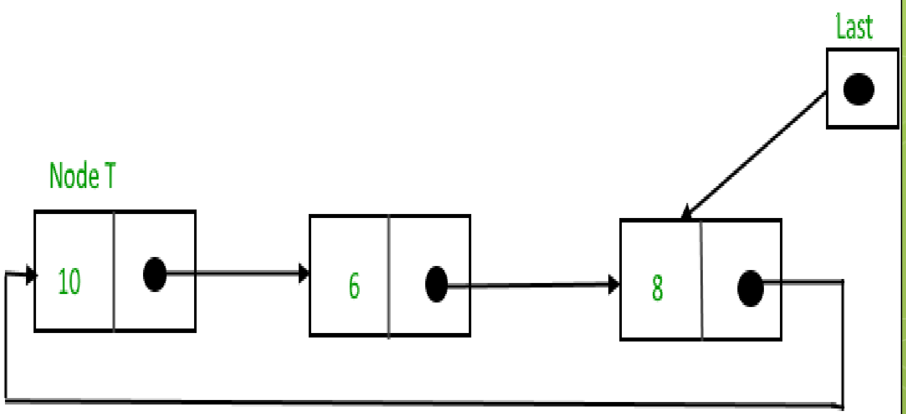


Deletion from a circular linked list :-

Case 4:- **Deletion of last node**

- Assign the link part of last node to the link part of previous node.
- So link part of previous node will point to the first node of list.
- Then assign the value of previous node to the pointer variable last **because after deletion of last node , pointer variable last should point to the previous node.**

```
tmp = q->link;
q->link = last->link;
free(tmp);
last =q;
```



05-06-2020

Doubly Linked List

Doubly Linked List

Drawback of single linked list-

- In single linked list,
  - **we can traverse only in one direction because each node has address of next node only.**
- Suppose we are in the middle of singly linked list and
  - **To do operation with just previous node then we have no way to go on previous node,**
  - We will again traverse from starting node.

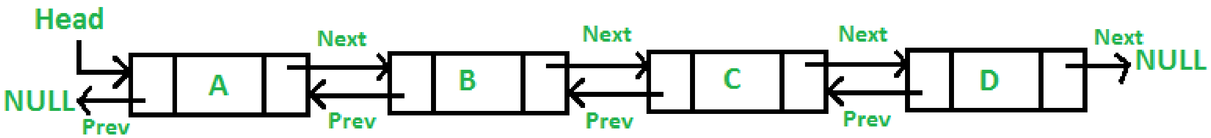
Doubly Linked List

Drawback of single linked list-

Solution-

- Doubly linked list, in this each node has address of previous and next node also.

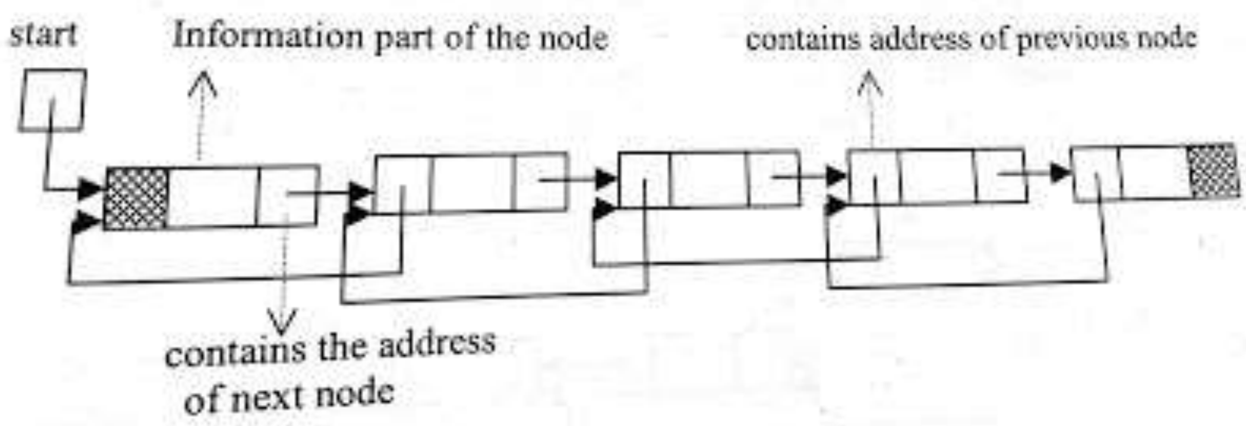
Doubly Linked List



Doubly Linked List

The data structure for doubly linked list will be as-

```
struct node
{
        struct node *prev;
        int info;
        struct node *next;
}*start;
```



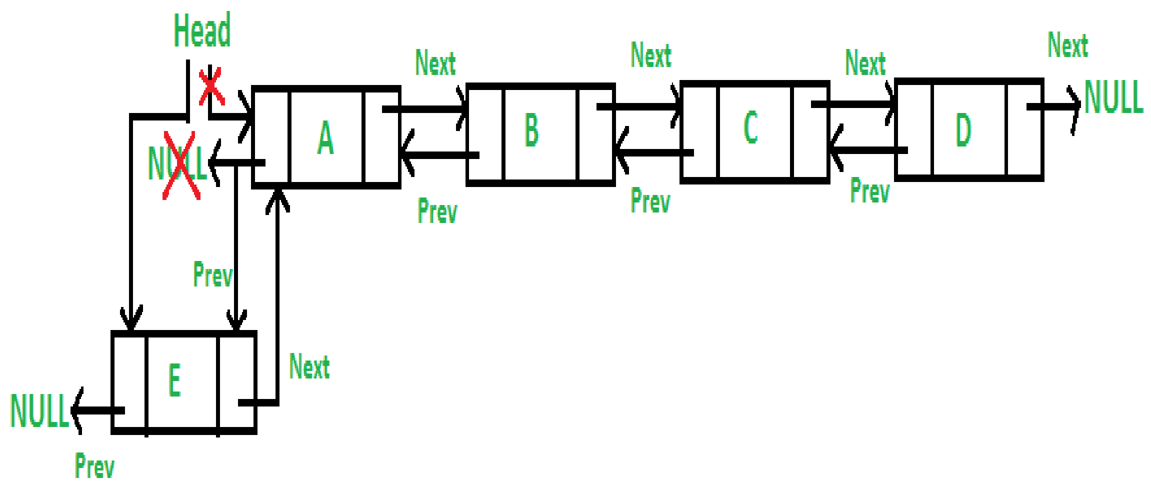
Doubly Linked List

```
struct node
{
        struct node *prev;
        int info;
        struct node *next;
}*start;
```

- ***struct node *previous is a pointer to structure, which will contain the address of previous node***
- ***struct node * next will contain the address of next node in the list.***
- ***Traversal in both directions at any time.***

Doubly Linked List-Insertion at beginning

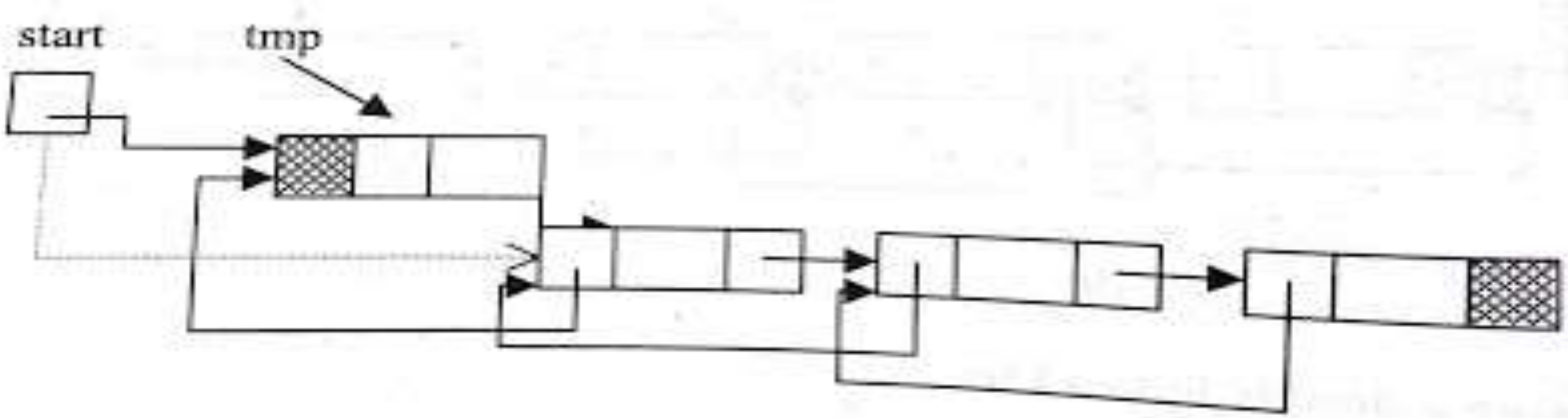
A 5 steps process



Doubly Linked List-Insertion at beginning

- Start points to the first node of doubly linked list.
- Assign the value of start to the next part of inserted node and address of inserted node to the prev part of start as-
  1) **tmp->next=start;**
  2) **tmp->info=data**
  3) **start->prev = tmp;**

1) Now inserted node points to the next node, which was beginning node of the doubly linked list and
3) prev part of second node will point to the new inserted node. Now inserted node is the first node of the doubly linked list.



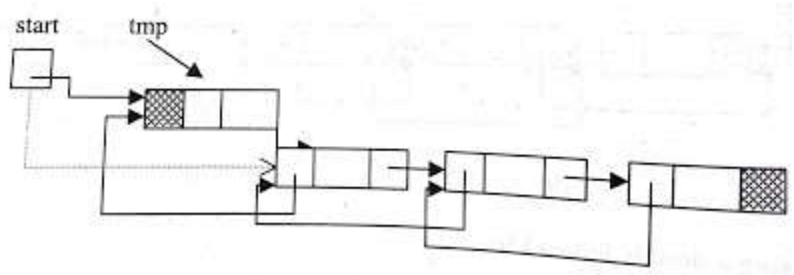
Doubly Linked List-Insertion at beginning

- So start will be reassigned as-

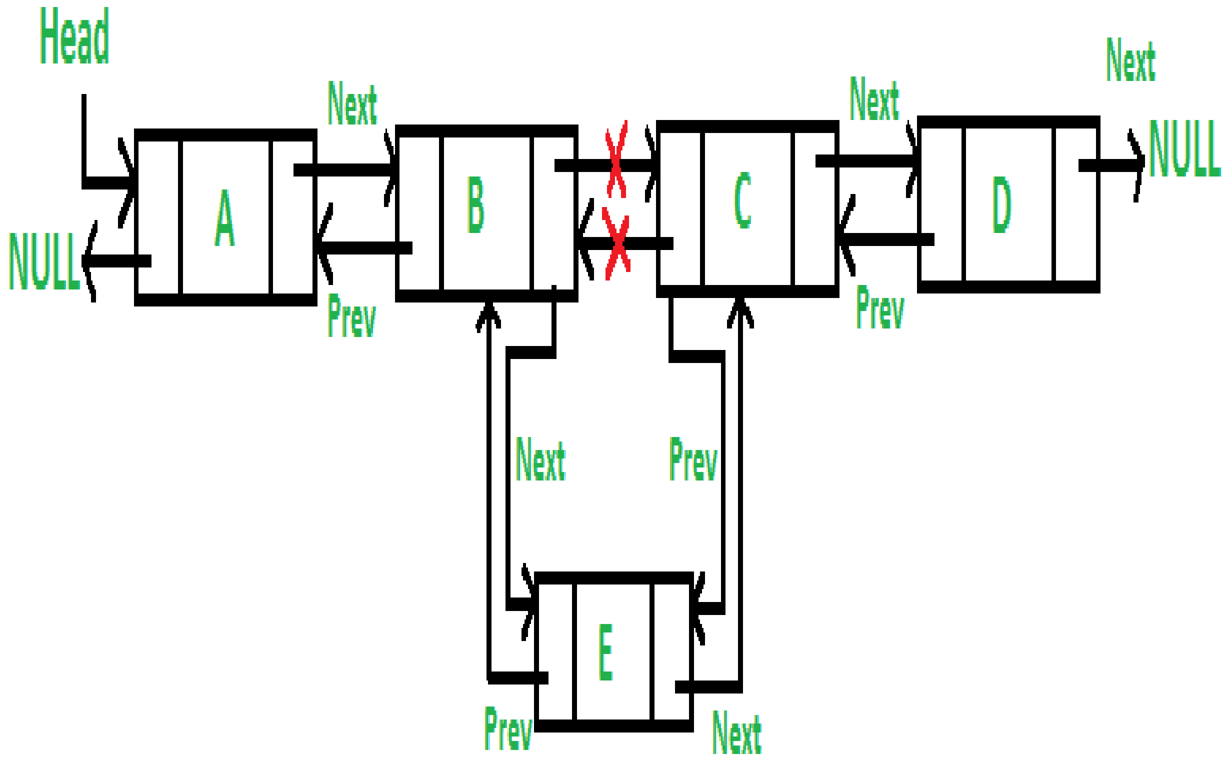
start= tmp;

- Now start will point to the inserted node which is first node of the doubly linked list.
- Assign NULL to prev part of inserted node since now it will become the first node and prev part of first node is NULL-

tmp->prev=NULL;

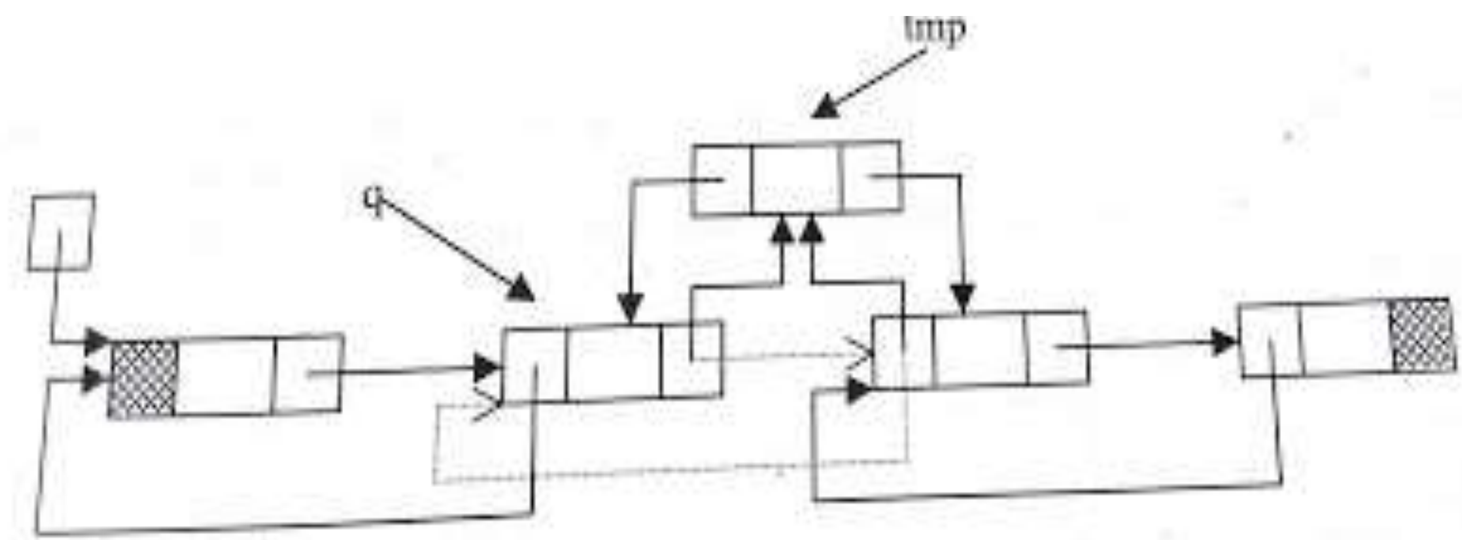


Doubly Linked List-Insertion in between



Doubly Linked List-Insertion in between

- *Traverse to obtain the **node (q) after which we want to insert the element.***
- *Assign the address of **inserted node(tmp**) to the prev part of next node.*
  q->next->prev=tmp;
- *Assign the next part of previous node to the next part of inserted node.*
  tmp->next=q->next;
- *Address of previous node will be assigned to prev part of inserted node*
  tmp->prev=q;
- *Address of inserted node will be assigned to next part of previous node.*
  q->next=tmp;

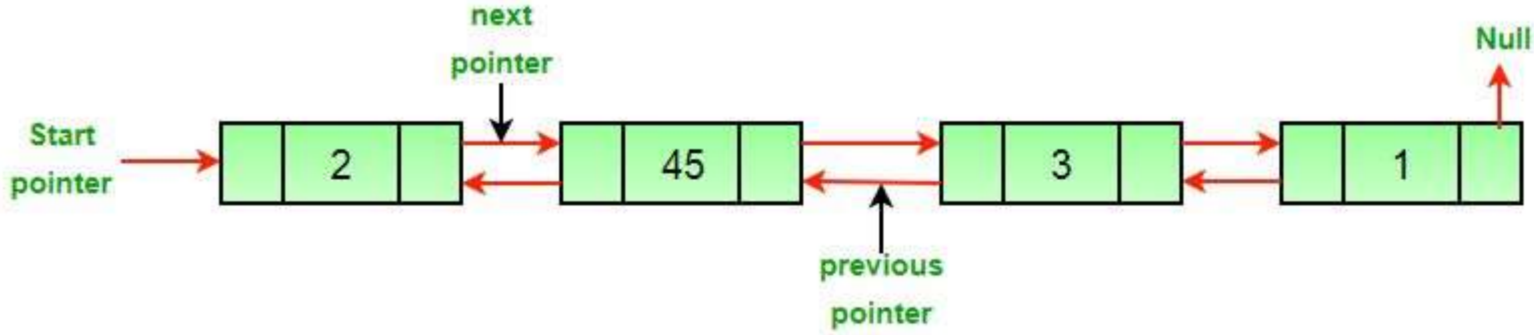


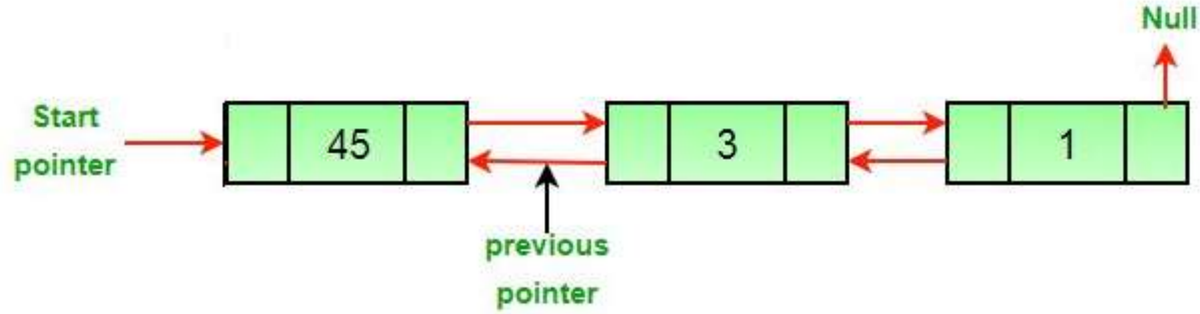
Deletion from doubly linked list

Traverse the linked list and compare with each element. After finding the element there may be three cases for deletion-

- **Deletion at beginning**
- **Deletion in between**
- **Deletion of last node**

Deletion from doubly linked list-Deletion at beginning





Doubly linked list-Deletion at beginning

- *Assign the value of start to tmp as-*

  tmp = start;

- *Now we assign the next part of deleted node to start as-*

  start=start->next;

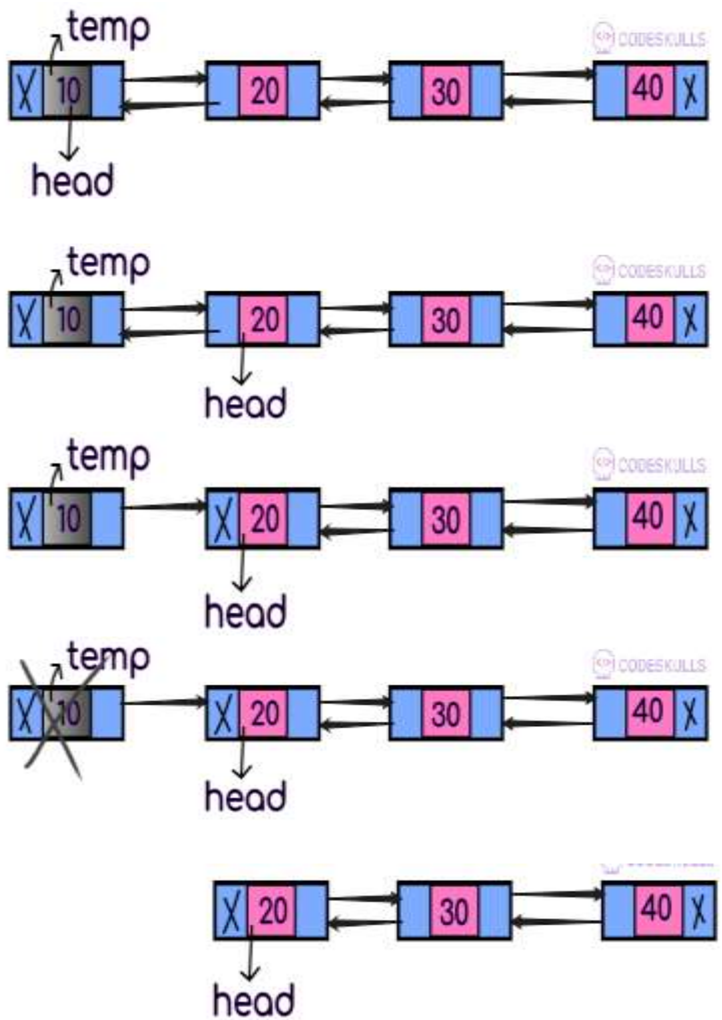
- *Since start points to the first node of linked list, so start->next will point to the second node of list.*
- *Then NULL will be assigned to start->prev.*

  start->prev = NULL;

- *Now we should free the node to be deleted which is pointed by tmp.*

  free(tmp);

Doubly linked list-Deletion at beginning



- *Assign the value of start to tmp as-*

  tmp = start;

- *Now we assign the next part of deleted node to start as-*

  start=start->next;

- *Since start points to the first node of linked list, so start->next will point to the second node of list.*
- *Then NULL will be assigned to start->prev.*

  start->prev = NULL;

- *Now we should free the node to be deleted which is pointed by tmp.*

  free(tmp);

Deletion from doubly linked list-Deletion in between

- *If the element is other than the first element of linked list*
  - *then we assign the next part of the deleted node to the next page of the previous node*
  - *address of the previous node to prev part of next node. This can be as-*

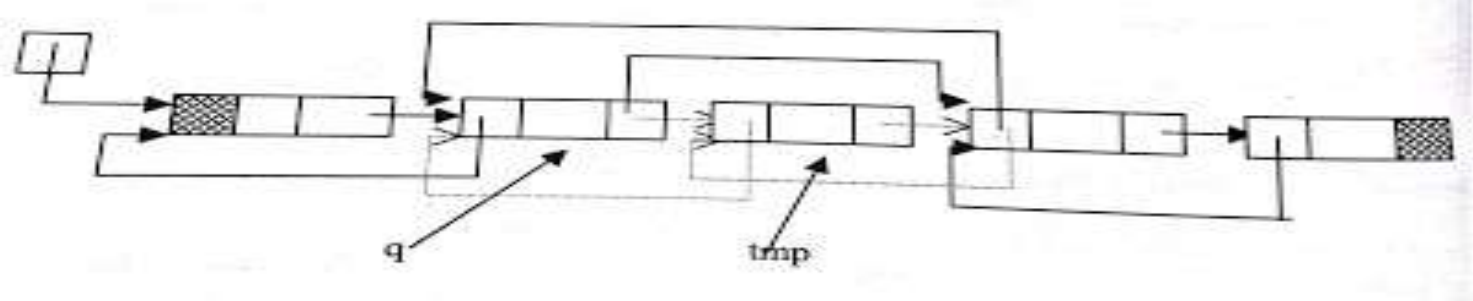
tmp=q->next;

q->next=tmp->next;

tmp->next->prev=q;

free(tmp);

- *Here q is pointing to the previous node of node to be deleted.*
  - *After statement 1 tmp will point to the node to be deleted.*
  - *After statement 2 next part of previous node will point to next node of the node to be deleted*
  - *After statement 3 prev part of next node will point to previous node.*



Doubly linked list-Deletion of last node

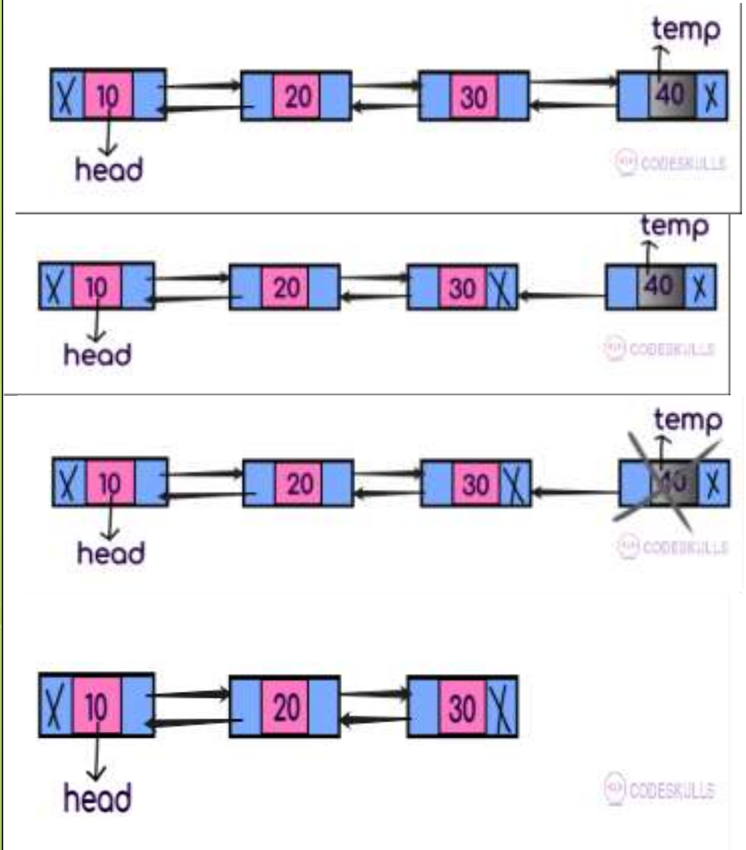
- *If node to be deleted is last node of doubly linked list then*
  - *we will just free the last node and*

    tmp=q->next;

    free(tmp);
  - *next part of second last node will be NULL.*

    q->next=NULL;
- *Here q is second last node*
- *After statement 1, tmp will point to last node*
- *After statement 2 last node will be deleted and after statement 3 second last node will become the last node of list.*

Doubly linked list-Deletion of last node



- *If node to be deleted is last node of doubly linked list then*
  - *we will just free the last node and*

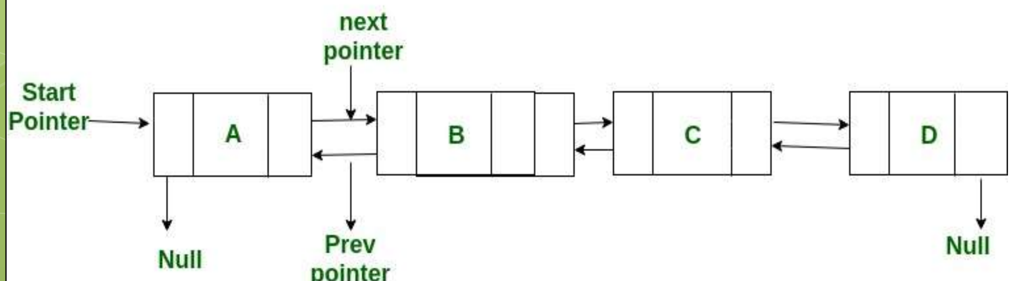
    tmp=q->next;

    free(tmp);
  - *next part of second last node will be NULL.*

    q->next=NULL;
- *Here q is second last node*
- *After statement 1, tmp will point to last node*
- *After statement 2 last node will be deleted and after statement 3 second last node will become the last node of list.*

Ordinary Representation:

- Node A:
  prev = NULL, next = add(B) // previous is NULL and next is address of B
- Node B:
  prev = add(A), next = add(C) // previous is address of A and next is address of C
- Node C:
  prev = add(B), next = add(D) // previous is address of B and next is address of D
- Node D:
  prev = add(C), next = NULL // previous is address of C and next is NULL



XOR List Representation:

- Let us call the address variable in XOR representation **as npx (XOR of next and previous)**
- Node A:
  npx = 0 XOR add(B) // bitwise XOR of zero and address of B
- Node B:
  npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C
- Node C:
  npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D
- Node D:
  npx = add(C) XOR 0 // bitwise XOR of address of C and 0

Polynomial arithmetic with linked list

- ***One useful application of linear linked list***
- ***Linked list can be used***
  - ***to represent polynomial expression***
  - ***for arithmetic operations also.***

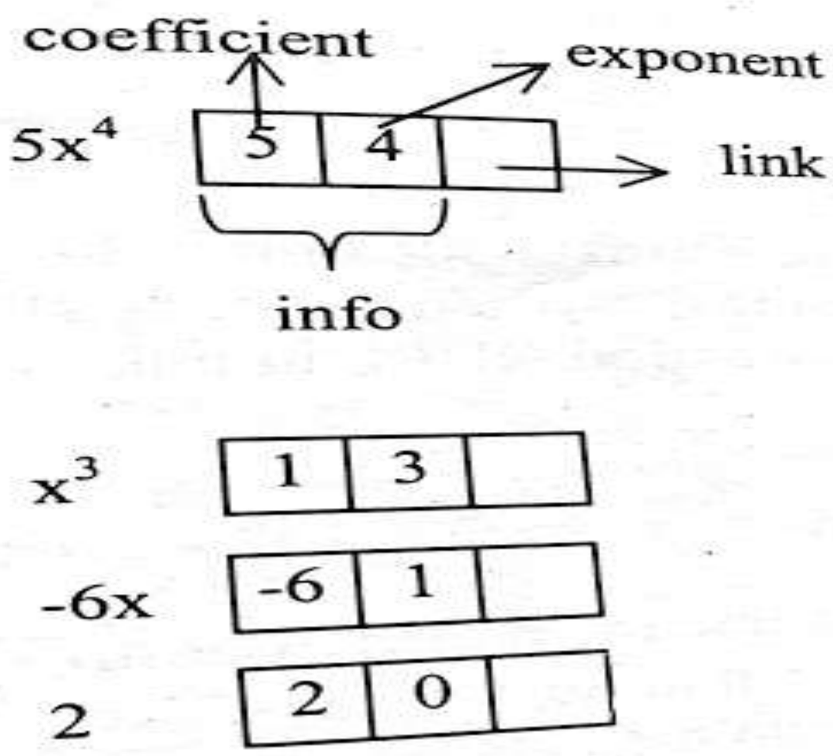
Polynomial arithmetic with linked list

- *Let us take a polynomial expression-*

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

- *Here we can see every symbol x is attached with two things,*
  - *coefficient*
  - *exponent.*
  - *As in $5x^4$, coefficient is 5 and exponent is 4.*
- *So we can represent polynomial expression in single linked list*
  - ***where each node of list will contain the coefficient and exponent of each term of polynomial expression.***

Polynomial arithmetic with linked list

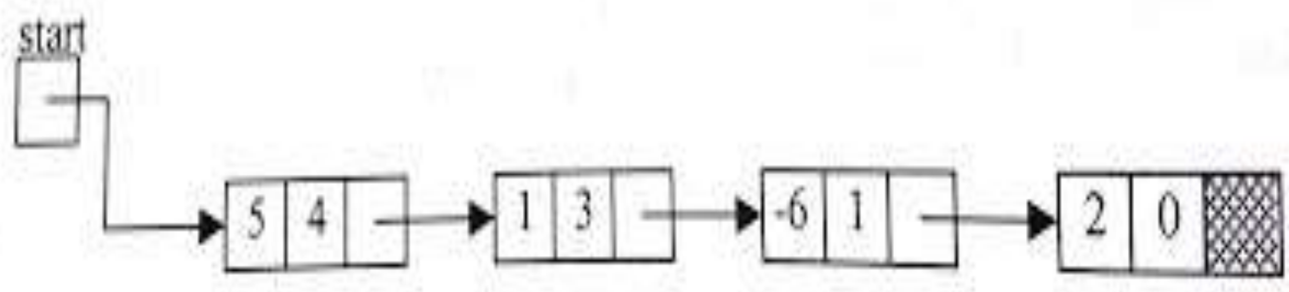


Polynomial arithmetic with linked list

- *So the data structure for polynomial expression will be-*

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

- ***Descending sorted linked list is used based on the exponent because it will be easier for arithmetic operation with polynomial linked list.***
- *Otherwise, we have a need to traverse the full list for every arithmetic operation.*
- *Now we can represent polynomial expression* $\mathbf{5x^4 + x^3 - 6x + 2}$ *as-*



Creation of polynomial linked list

- Creation of polynomial linked list will be same as
  - creation of sorted linked list but
  - ***it be in descending order and based on exponent of symbol.***

Creation of polynomial linked list

- *In if condition we are checking for the node to be added will be first node or not.*

```
/* list empty or exp greater than first one */
if(start==NULL || ex> start->expo)
{
        tmp->link=start;
        start=tmp;
        return start;
}
```

Creation of polynomial linked list

```
else
{
        ptr=start;
        while(ptr->link!=NULL && ptr->link->expo > ex)
                {
                        ptr=ptr->link;
                }
        tmp->link=ptr->link;
        ptr->link=tmp;

        if(ptr->link==NULL) /* item to be added in the end */
                tmp->link=NULL;
}
```

- *In else part*
  - *we traverse the list and*
  - *check the condition for exponent then*
  - *we add the node at proper place in list.*
- ***If node will be added at the end of list then***
  - *we assign NULL in link part of added node.*

Addition with polynomial linked list

Input :

$p1 = 13x^8 + 7x^5 + 32x^2 + 54$

$p2 = 3x^{12} + 17x^5 + 3x^3 + 98$

Output : $3x^{12} + 13x^8 + 24x^5 + 3x3 + 32x^2 + 152$

Addition with polynomial linked list

- For addition of two polynomial linked list, we have a need to traverse the nodes of both the lists.
  - **If the node has exponent value higher than another, then**
    - that node will be added to the resultant list or
    - **<u>we can say that nodes which are unique to both the lists will be added in the resultant list.</u>**

Addition with polynomial linked list

- If the nodes have **same exponent** value then
  - first the coefficient of both nodes will be added
  - then the result will be added to the resultant list.
- Suppose in traversing one list is finished then
  - **remaining node of the another list will be added to the resultant list**.