Lr is the start row index of matrix

Lc is the start column index of matrix

Complexity of array operations:

Algorithm	Average case	Worst case
search	O(n)	O(n)
insert	O(n)	O(n)
delete	O(n)	O(n)

Linked lists:

- Limitations of arrays
    - Fixed size – The size of the array is static, so wastage of memory.
    - One block allocation – Not always possible.
    - Complex position based insertion/deletion – To insert an element we need to shift the existing elements.
- Solution - Linked List

What is a linked list?

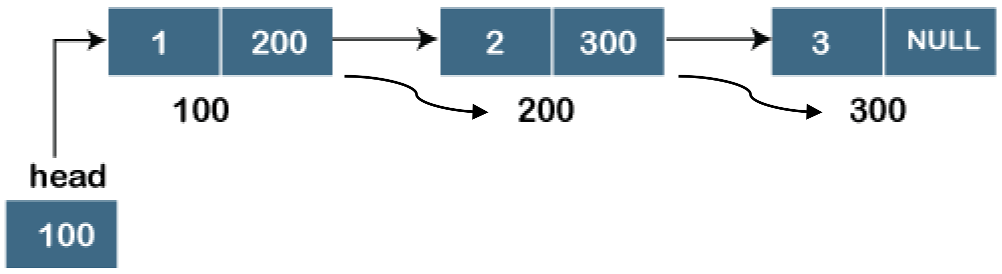
- A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node.
- Each node contains
    - **Data** will store the information part

- **Next** will store the address of the next node.

```
Struct node{

Int data;

Struct node * next;

}
```

In the above declaration, we have defined a structure named as ***a node*** consisting of two variables:

An integer variable (data), and the other one is the pointer (next), which contains the address of the next node.



We can observe in the above figure that the next pointer is storing the address of the next node.

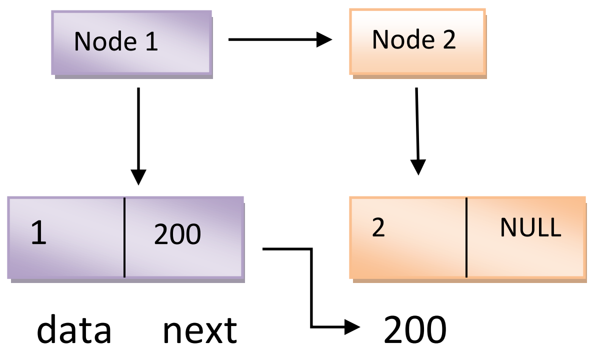
Node:

Step 1: data=1; Node 1

Next= NULL (as there is no another element)

Step 2: data1=2; Node 2

Node 1 next should store the address of the second node



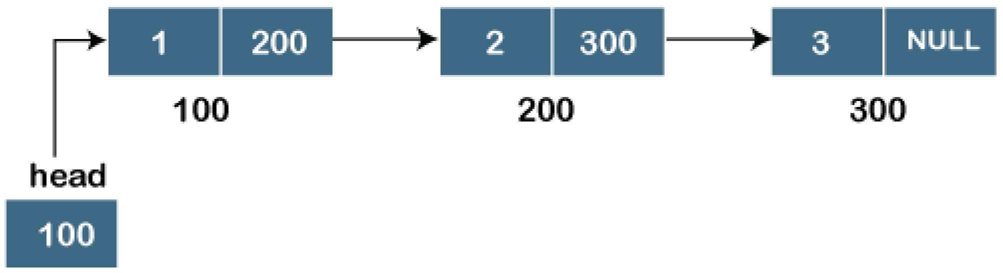
(Dynamic memory allocation)

Types of linked lists

1. Singly Linked List

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

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

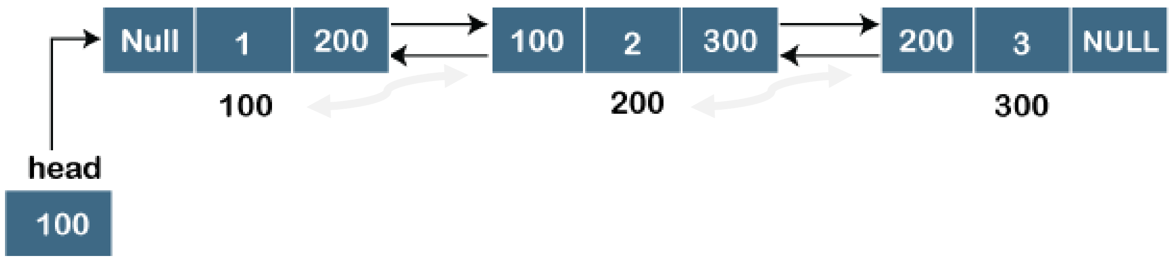


2. Doubly Linked List

Two-way linked list which contains a pointer to the next as well as the previous node in the sequence.

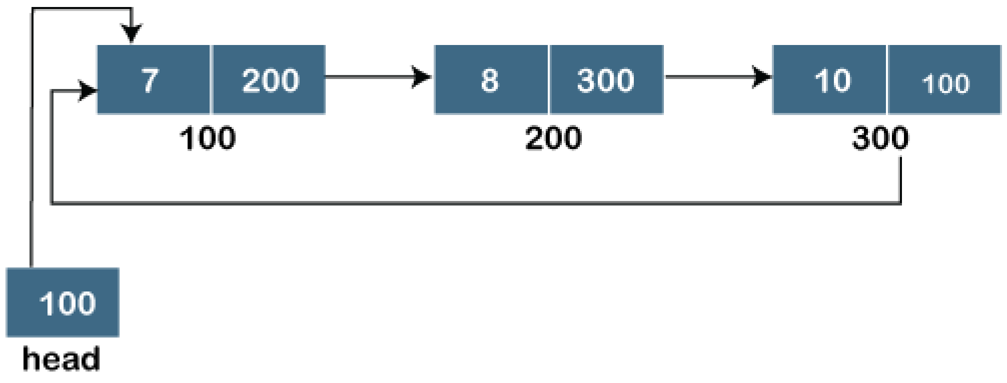
- ***prev*** pointer-points to the address of previous node
- ***next***-pointer points to the address of the next node

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



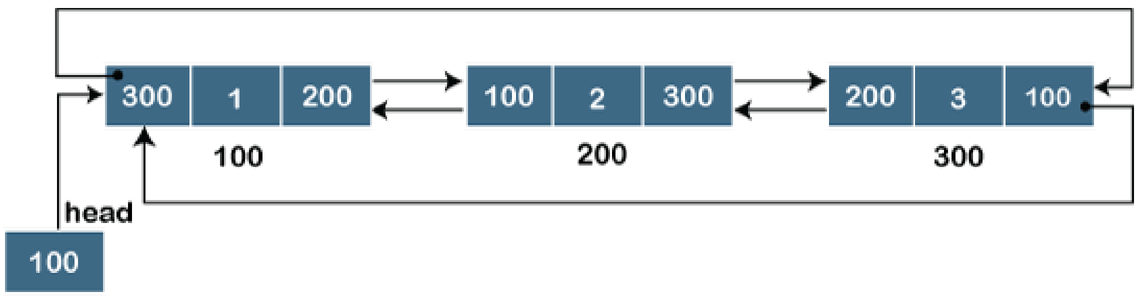
3. Circular Singly Linked List

A Singly linked list where the last node contains a pointer to the first node of the list.



4. Circular Doubly Linked List

A doubly linked list where the first node <prev pointer> is connected to the last node and the last node <next pointer> will connect to the first node.



Operations-Linked List

- Create
- Insert
    - Insert at the beginning
    - Insert at the end
    - Insert before a given node
    - Insert after a given node
- Delete
    - Delete a node from beginning
    - Delete a node from the end
    - Delete a given node
    - Delete after a given node
- Display/Traversing
- Sort

Singly linked list

Singly linked list can be defined as the collection of ordered set of elements.

A node in the singly linked list consists of two parts:

1. Data part (stores actual information that is to be represented by the node)

2. Link part or next part (stores the address of its next node)

Operations on Singly Linked list:

(i) Insertion at beginning

(ii) Insertion at any position

(iii) Insertion at End

(iv)Delete at beginning, any position, at end

(i) Insertion at beginning:

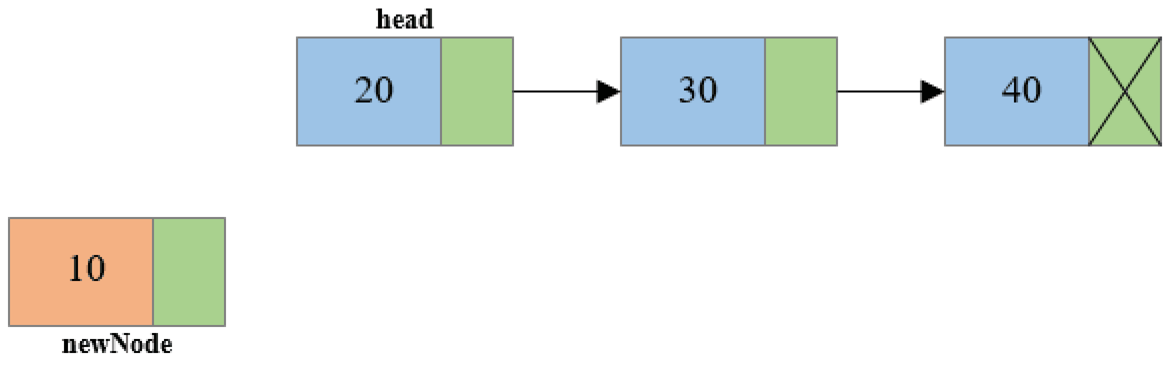
Whenever we are inserting a node at the beginning we have to just change the next pointer of the new node.

There are the following steps which need to be followed in order to insert a new node in the list at beginning.

Step 1: Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.

newnode = (struct node *) malloc(sizeof(struct node *));
//allocating memory

newnode → data = item //in this step the value of the data is stored

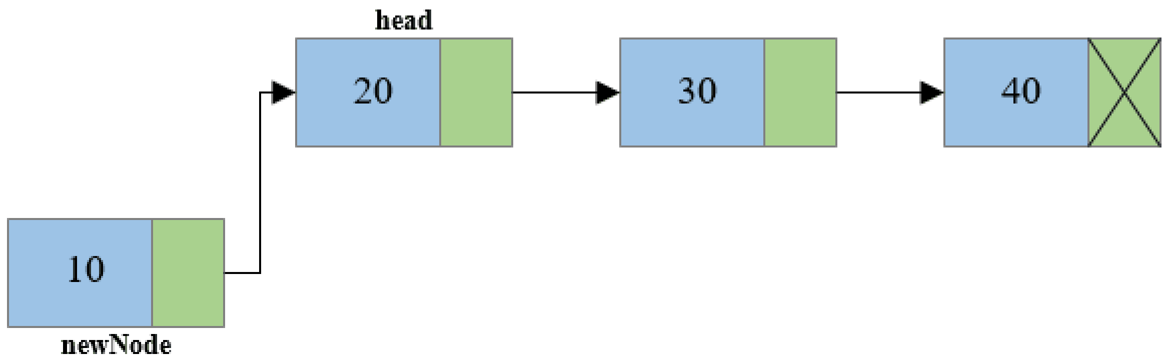


Step 2: Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.

newnode->next= head;

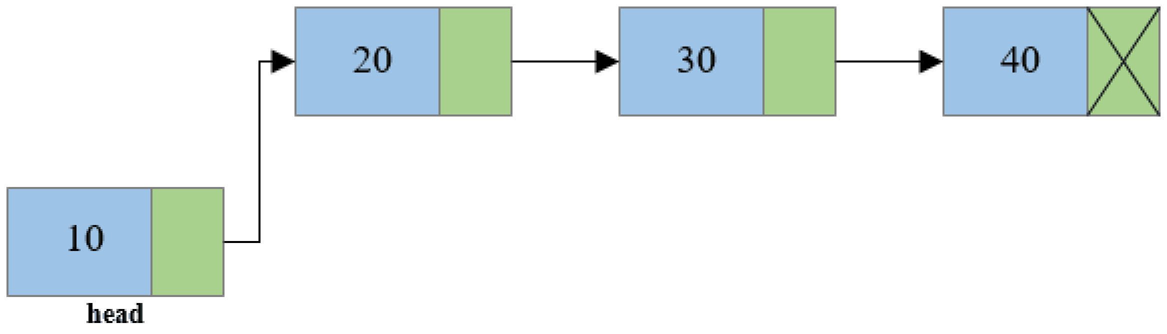
//in this newnode next pointer points to the first node

//here head represents the first node of the list



Step 3: here we need to make our newnode as head.

head=newnode;



Source code:

Insert_Begin(ele, head) // ele is the element to add and head is the LL header

*//**new node creation***

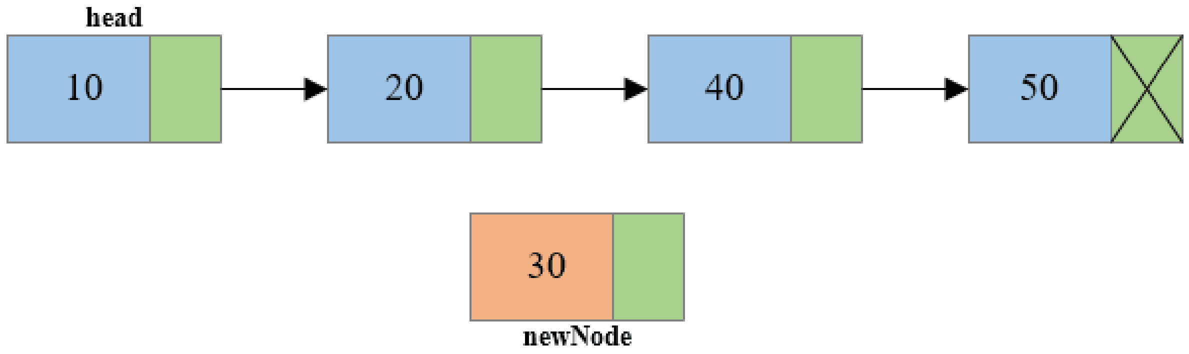
```
Struct Node *newnode, *t;

newnode=(struct Node *) malloc(sizeof(struct Node *));

        if (newnode == NULL)  then, print (“NO memory”) exit(0);

                    //new node assignment

newnode→data=ele;

newnode→next=NULL;

      // check if list is empty then make newnode as first node

if( head==NULL)

          head=newnode;

else

// if list is not empty, then update the new_node next and update head

newnode→next=head;

head=newnode;
```

(ii) Insertion at any position:

In order to insert an element at any position into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. This will be done by using the following statements.

```
    temp=head;
        for (i=0;i<location; i++)
      {
       temp = temp->next;
        }
```

//here we have to move the temp pointer

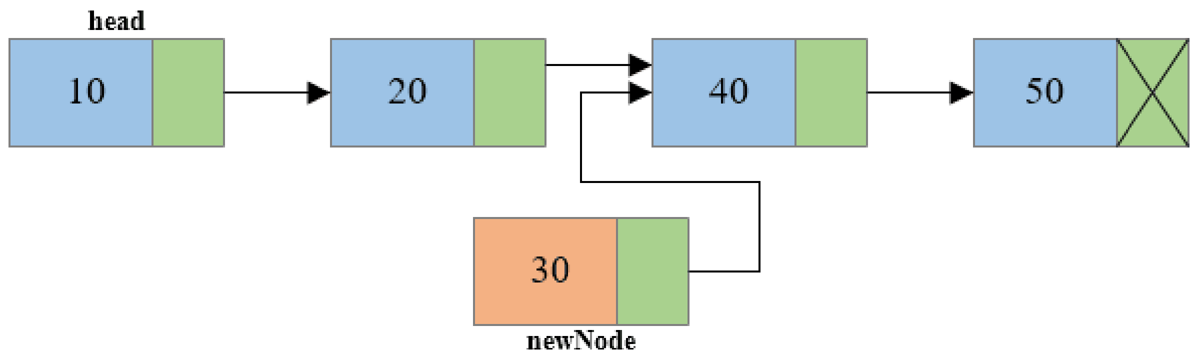


Step 2: Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements

```
ptr = (struct node *) malloc (sizeof(struct node));

 ptr->data = item;
```

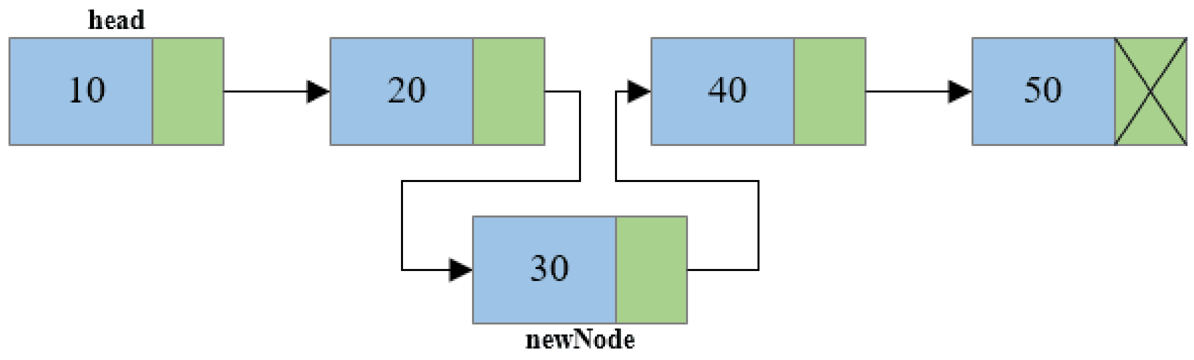
Step 3: Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). This will be done by using the following statements.

```
ptr→ next = temp → next
```



Step 4: we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

temp ->next = ptr;



Source code:

Insert(*ele, aele, head*)

// ele-to insert, aele-after element, head is the LL header

//create a new node

Struct Node *new_node, *t;

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

```
if (new_node == NULL)  then, print ("NO memory") exit(0);
      // Assign data to new_node and update the address pointer
new_node→data=ele;
new_node→next=NULL;
          // if list is empty, assign new_node address in head
if( head==NULL)
                    print(" Insertion not possible");
         // if  list is not empty, then mover the desired location
else
temp=head;
while(temp→data!=aele && temp!= NULL)
       temp=temp→next;
If(temp != NULL)
           //changing new_node next value
new→next=temp→next;
           // changing temp->next to new_node
temp→next=new_node;
else print("Element Not Found");
```

(iii) Insertion at end:

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list
2. The node is being added to the end of the linked list

Case 1:

1) The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.

```
ptr->data = item;   ptr -> next = NULL;
```

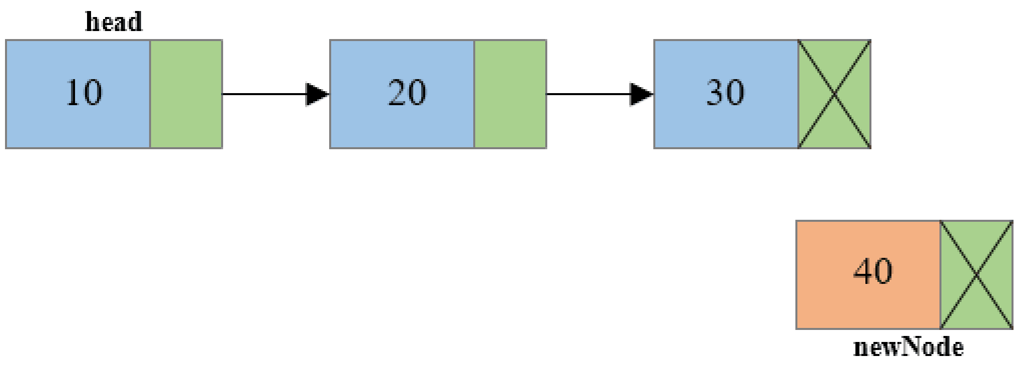
2) Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.

```
Head = ptr
```

Case 2:

Step 1: The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

```
Temp = head
```

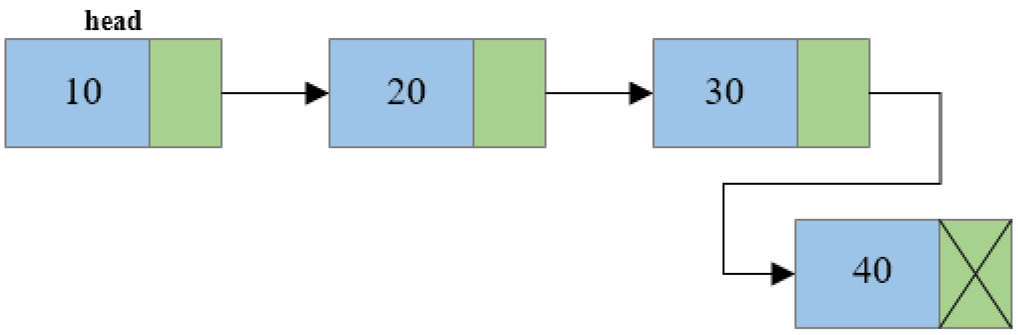


Step 2: Then, traverse through the entire linked list using the statements:

```
while (temp→ next != NULL)
temp = temp → next;
```

Step 3: At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr).

```
temp = head;
 while (temp -> next != NULL)
      {
        temp = temp -> next;
    }
   temp->next = ptr;
  ptr->next = NULL;
```



Source code:

Insert_End(ele, head) // ele is the element to be added, head is the SLL header

//new node creation

- struct Node *new_node, *t;
- new_node = (struct Node *) malloc(sizeof(struct Node *));
- ***if (new_node == NULL) then, print ("NO memory") exit(0);***

//new node assignment

- new_node→data = x;
- new_node→next=NULL;

// if list is empty then make new_node as first node

- if(head==NULL)
    - head=new_node;

// if list is not empty, then reach the last node and then add the address of new_node in last node next

- else
    - t=head;

//move to the end of list

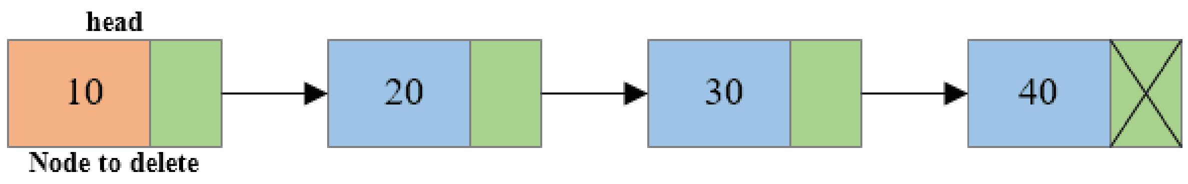
    - while(t→next!=NULL)
        - t=t→next;

//adding node at the end

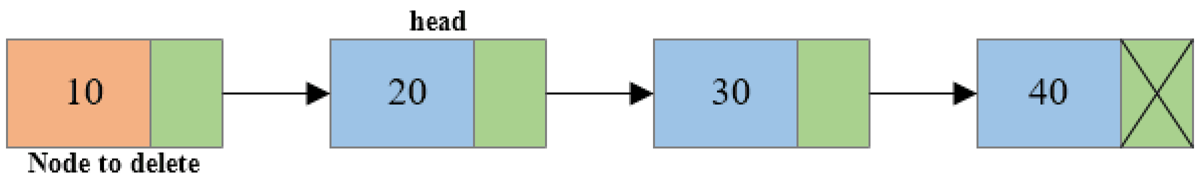
    - t→next=new_node;

Deletion of first node:

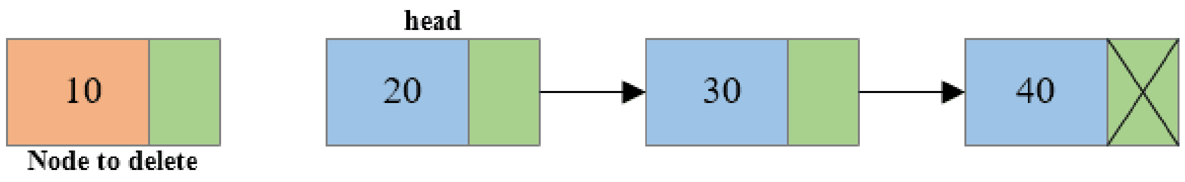
1) Copy the address of first node i.e. head node to some temp variable



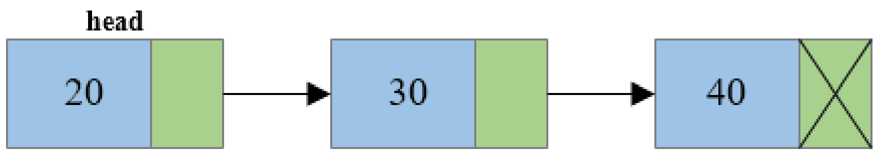
2) Move the head to the second node of the linked list i.e. head = head->next



3) Disconnect the connection of first node to second node (temp->next=NULL)

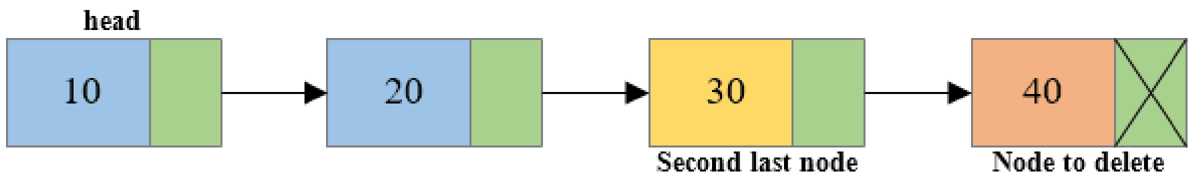


4) Free the memory occupied by the first node. (free(temp))

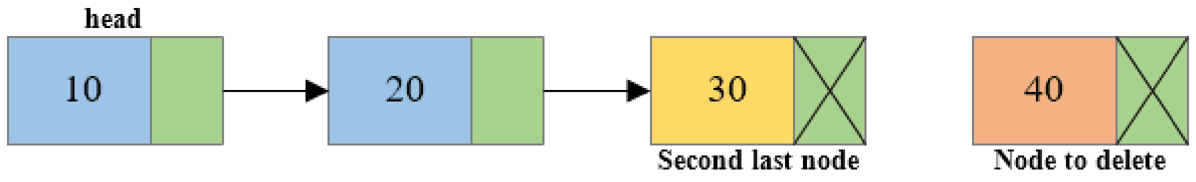


Deletion of end node:

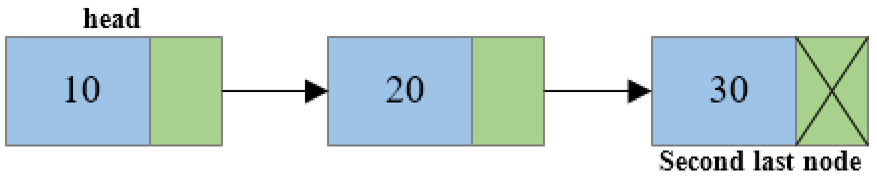
Step 1: Traverse to the last node of the linked list keeping track of the second last node in some temp (`secondLastNode`)



Step 2: If the last node is the head node then make the head node as NULL else disconnect the second last node with the last node i.e. secondLastNode->next = NULL

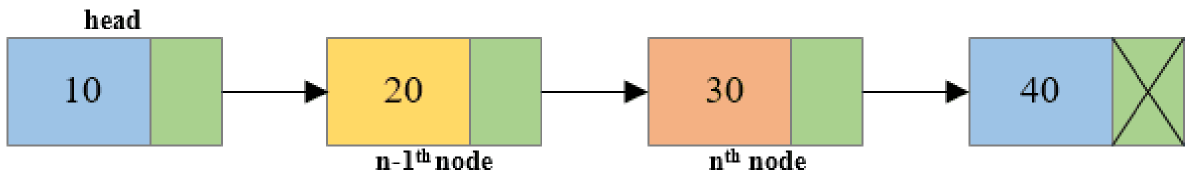


Step 3: Free the memory occupied by the last node.(free(temp))

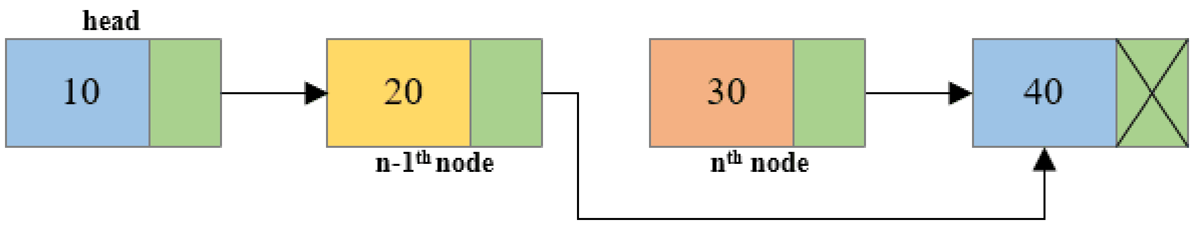


Deletion at any position:

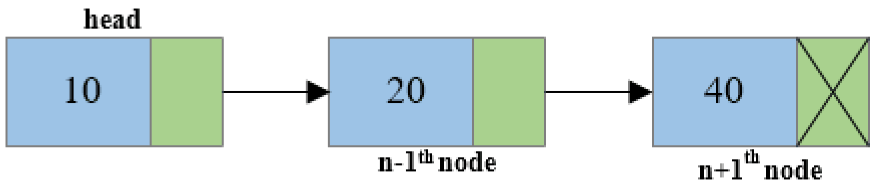
Step 1: Traverse to the nth node of the singly linked list and also keep reference of n-1th node in some temp variable say prevnode.



Step 2: Reconnect the n-1th node with the n+1th node i.e. prevNode->next = toDelete->next (Where prevNode is n-1th node and toDelete node is the nth node and toDelete->next is the n+1th node).



Step 3: Free the memory occupied by the nth node i.e. toDelete node



Doubly linked list

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains
  a pointer to the next as well as the previous node in the sequence.
- It is a sequence of elements/nodes and each element consists of three components,
  - Data: data / value of an element
  - Next: pointer points to the next node in a list
  - Prev: pointer points to the previous node in a list

Operations:

(i) Insertion at beginning

(ii) Insertion at any position

(iii) Insertion at End

(iv)Delete at beginning, any position, at end

Insertion at beginning:

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL.

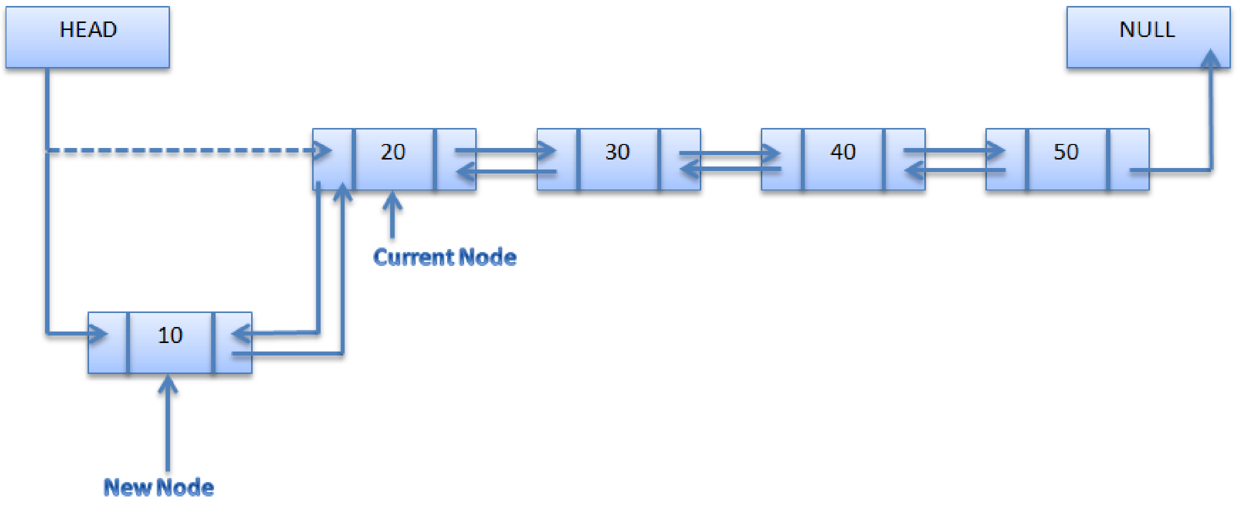
Source code:

Step 1 : Create a **new_node** with given value and **new_node → prev** as **NULL** and **new_node→next** as **NULL.**

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

Step 3 : If list is **Empty** then, assign **new_node** to **head,** and **return.**

Step 4 : If list is **not Empty** then, assign **head** to **new_node → next, new_node to head→prev**, and **new_node** to **head**.



(ii) Insert at end:

The new node is always added after the last node of the given Linked List. We have to traverse the list till end and then change the next of last node to new node.

Source code:

Step 1: Create a **new_node** with given value and **new_node → prev** and **new_node→next** as **NULL**.

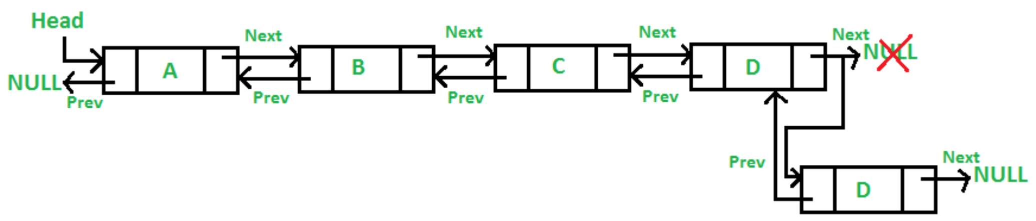
Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If list is **Empty** then, assign **new_node** to **head and return**.

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

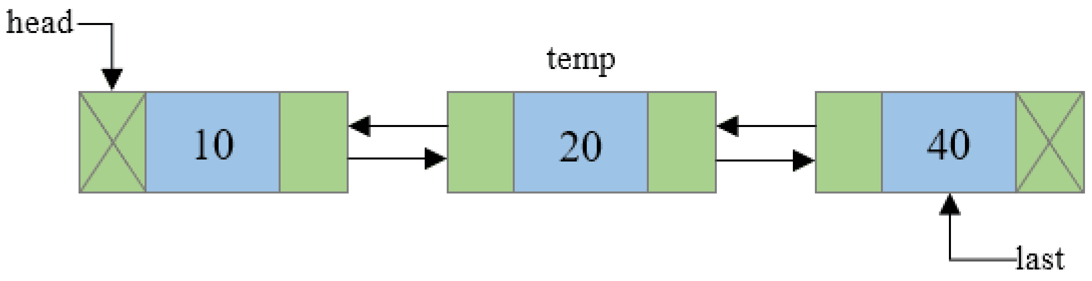
Step 5: Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is not equal to **NULL**).

Step 6 : **set temp → next =new_node** and **new_node → prev=temp**.

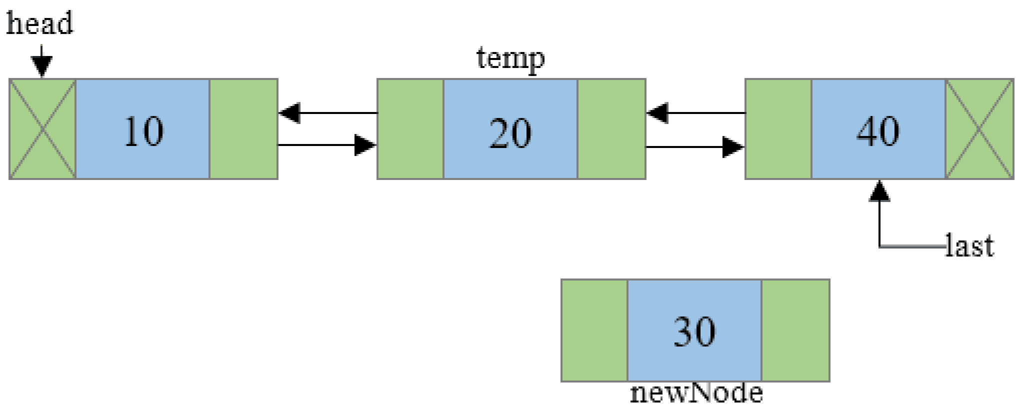


(iii) Insert after a given node:

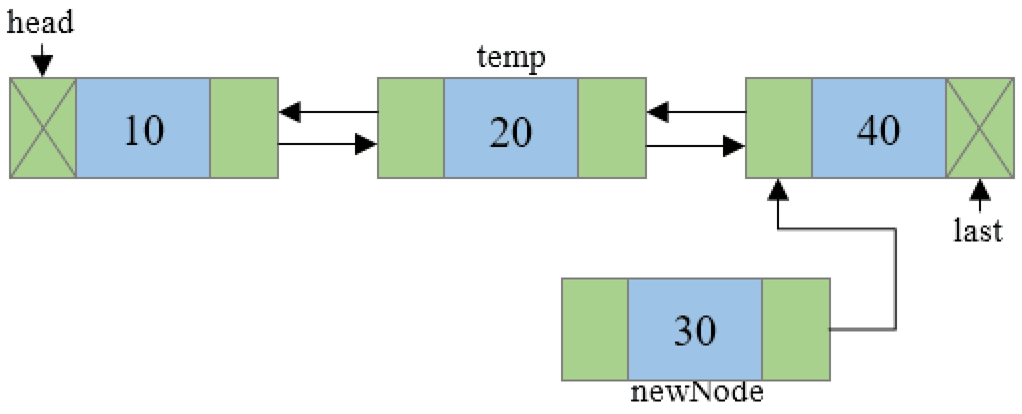
Step 1: Traverse to N-1 node in the list. Where N is the position to insert. Say temp now points to N-1th node.



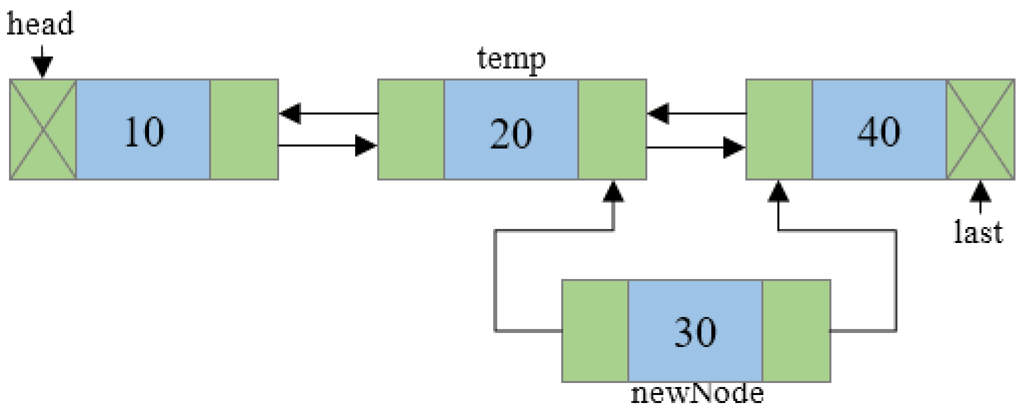
Step 2: Create a newNode that is to be inserted and assign some data to its data field



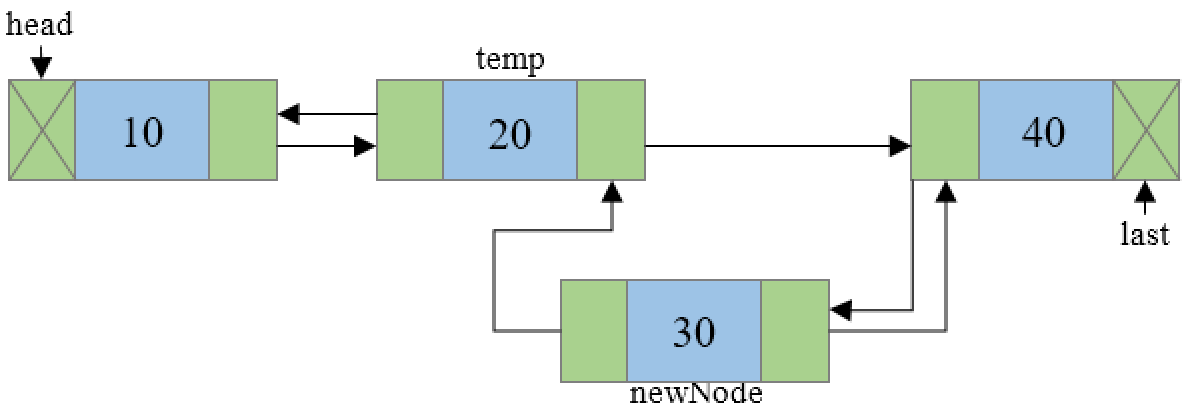
Step 3: Connect the next address field of newNode with the node pointed by next address field of temp node.



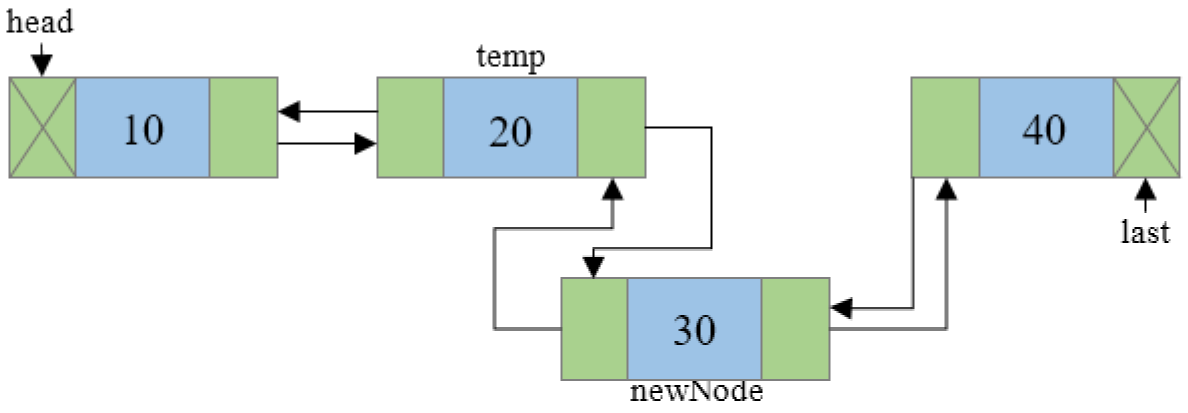
Step 4: Connect the previous address field of newNode with the temp node.



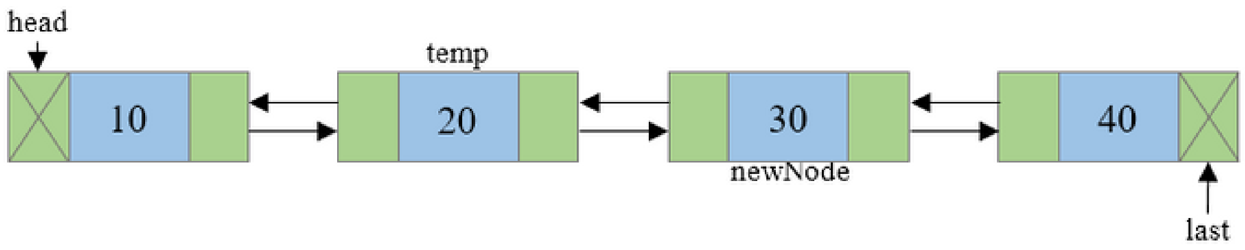
Step 5: Check if temp->next is not NULL then, connect the previous address field of node pointed by temp.next to newNode.



Step 6: Connect the next address field of temp node to newNode i.e. temp->next will now point to newNode.



Step 7: Final list



Deleting a node from the beginning:

Step 1 - Check whether list is Empty (**head == NULL**)

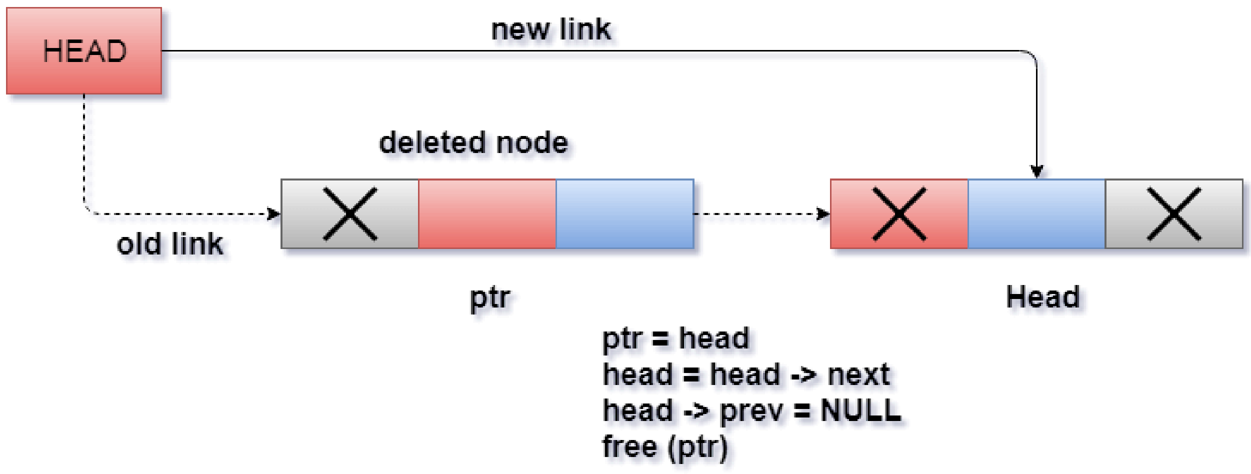
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If list 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 4.1 - If list is having only one node, then set head to NULL and delete temp (Setting Empty list conditions)

Step 4.2 - If list is with more than one node, then assign **temp → next to head**, **NULL** to **head → prev** and delete temp.



Deletion in doubly linked list from beginning

Deleting a node from the end:

Step 1 - Check whether list is Empty (**head == NULL**)

Step 2 - If list is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

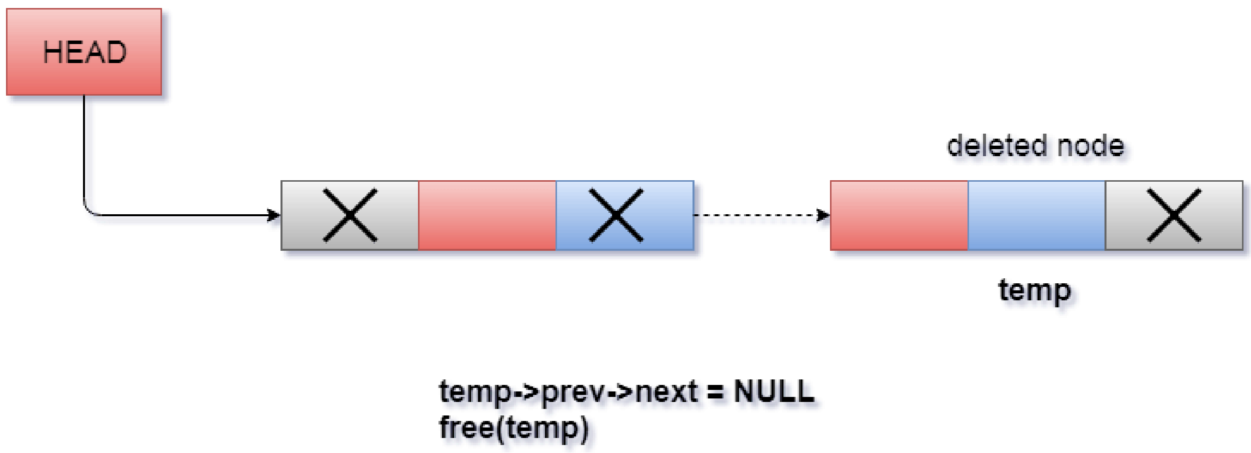
Step 3 - If list is not Empty then, define a Node pointer 'temp' and initialize with head.

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

Step 5 - If list has only one node, then assign **NULL to head** and delete temp. Terminate the function. (Setting Empty list condition)

Step 6 - If list has more than one node, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)

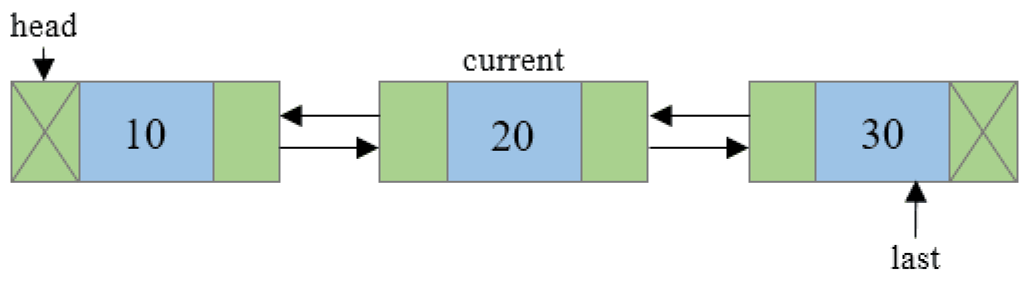
Step 7 - Assign NULL to temp → previous → next and delete temp.



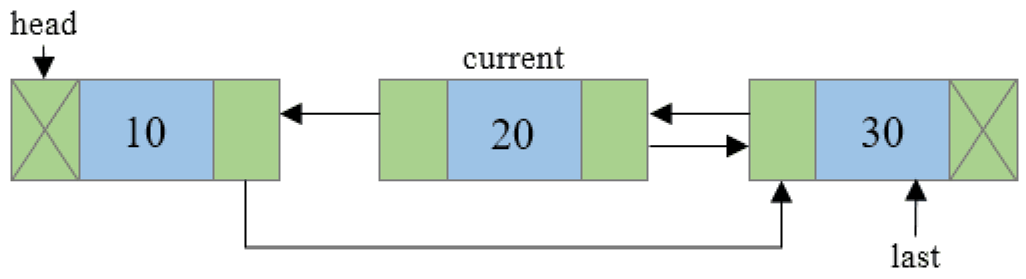
Deletion in doubly linked list at the end

Deleting a node from any position:

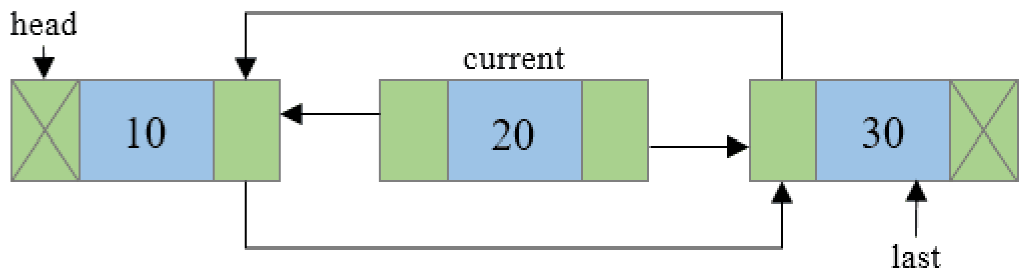
Step 1: Traverse to Nth node of the linked list, lets say a pointer current points to Nth node in our case 2 node.



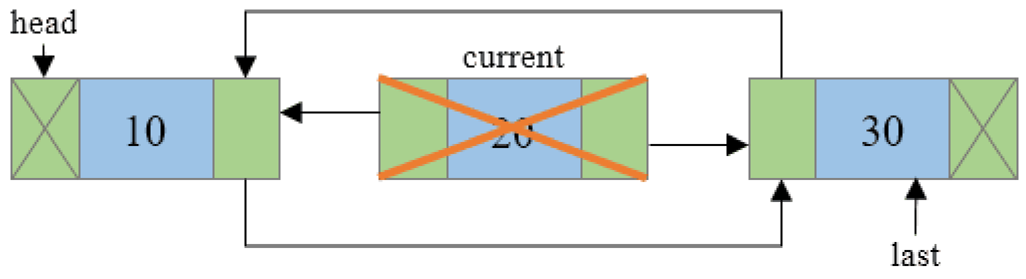
Step 2: Link the node behind current node with the node ahead of current node, which means now the N-1th node will point to N+1th node of the list. Which can be implemented as current->prev->next = current->next.

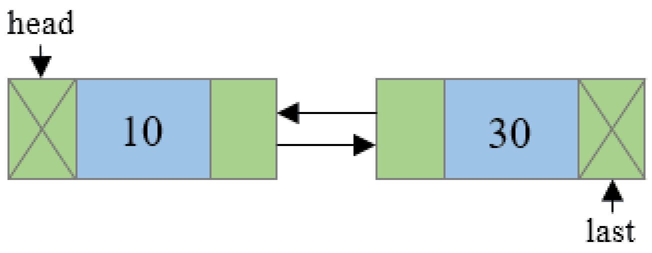


Step 3: If N+1th node is not NULL then link the N+1th node with N-1th node i.e. now the previous address field of N+1th node will point to N-1th node. Which can be implemented as current->next->prev = current->prev.



Step 4: Finally delete the current node from memory and you are done.





Circular singly linked list

- A circular singly linked list is list of elements, where each element consists of data and next pointer as values and last node points to first node.
- These Elements are connected through a link(address).

Insertion

A node can be added in three ways:

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between the nodes

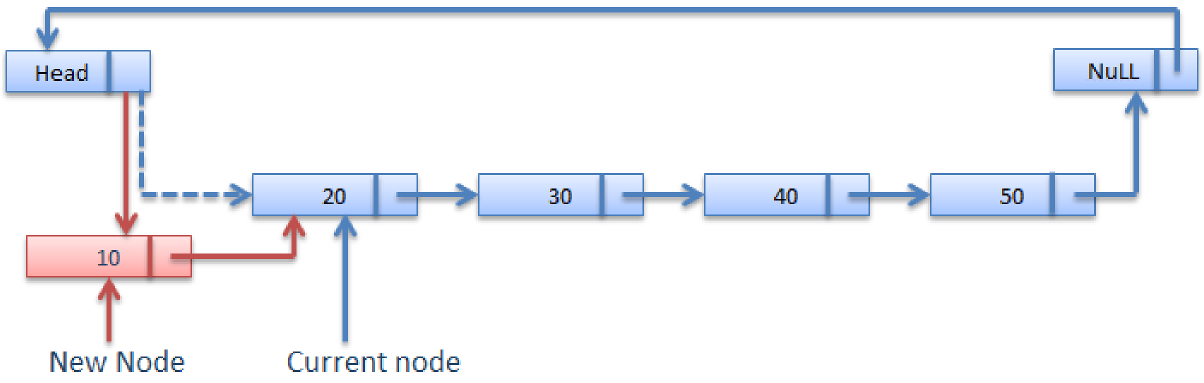
Insertion at the beginning of the list:

Step 1: Create a **new_node** with a given value and **new_node → next** as **NULL**.
Step 2: Check whether list is **Empty** (**head** == **NULL**).
Step 3: If list is **Empty** then, set **head** = **new_node**

and new_node→next = new_node;

Step 4 : If list is **Not Empty** then,

- *Declare a node pointers **temp, temp1** and initialize with **head***
- Set **new_node → next** = **head**.
- Set **head=new_node**.
- *Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is not equal to temp1).*
- *Set **temp → next** = **head.***



Insertion at the end of the list:

Insert_End(ele, head) // ele is the element to be added, head is the SLL header

//new node creation

- struct Node *new_node, *t;
- new_node = (struct Node *) malloc(sizeof(struct Node));
- ***if (new_node == NULL) then, print ("NO memory") exit(0);***

//new node assignment

- new_node→data = x;
- new_node→next=NULL;

// if list is empty then make new_node as first node

- if(head==NULL)
    - head=new_node and new_node→next = new_node;

// if list is not empty, then reach the last node and then add the address of new_node in last node next

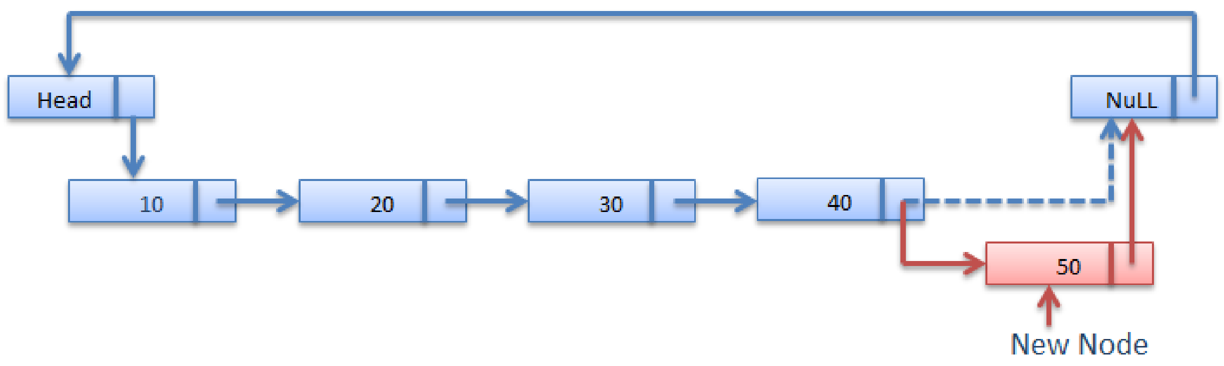
- else
    - t=head;

//move to the end of list

- while(t→next!=head)
    - t=t→next;

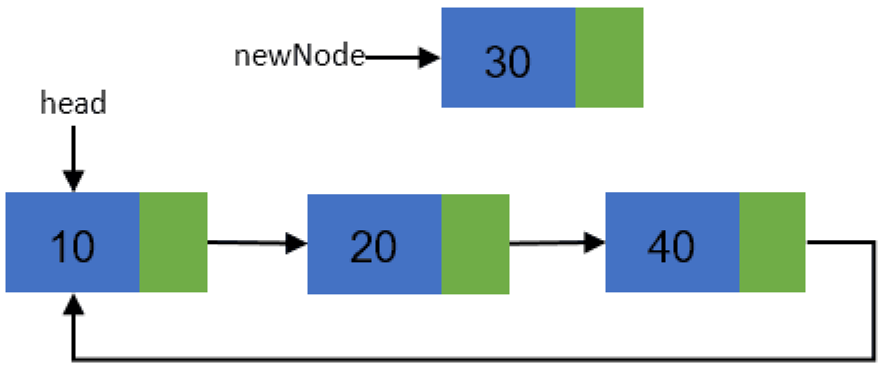
//adding node at the end

- t→next=new_node and new_node→next = head

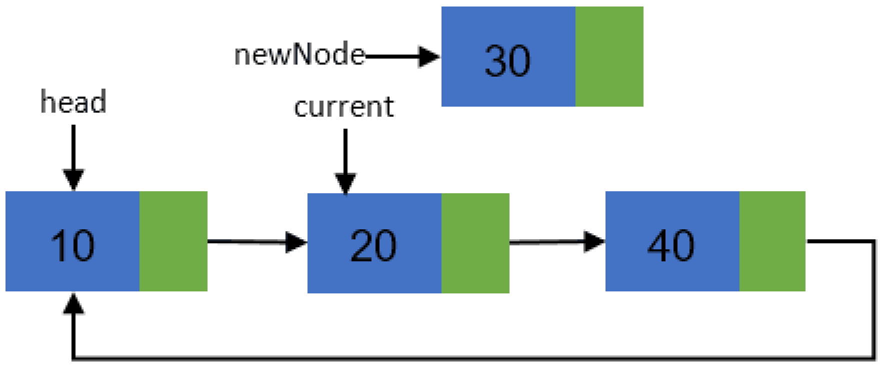


Insertion at the any position of the list:

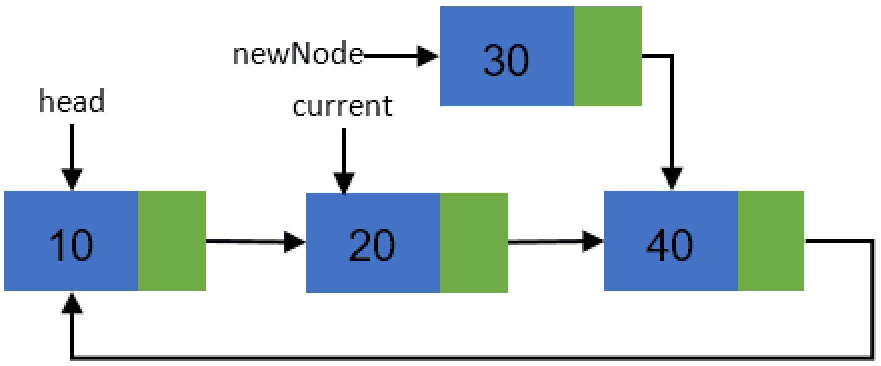
Step 1: Create a newNode and assign some data to its data field.



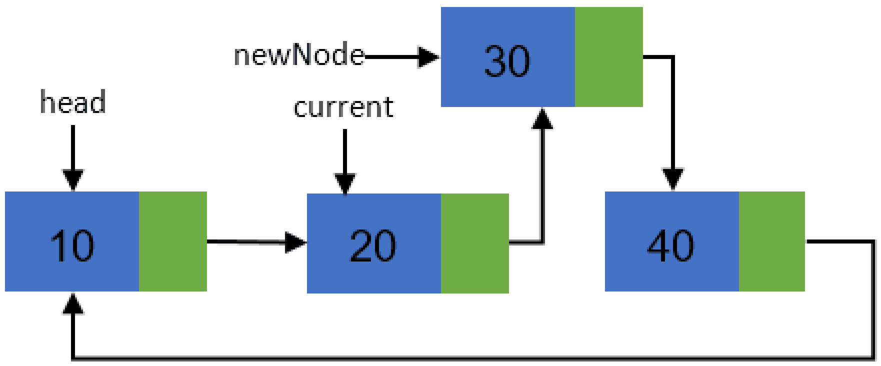
Step 2: Traverse to N-1 position in the list, in our case since we want to insert node at 3rd position therefore we would traverse to 3-1 = 2nd position in the list. Say **current** pointer points to N-1th node.



Step 3: Link the next pointer field of newNode with the node pointed by the next pointer field of current(N-1) node. Which means newNode.next = current.next.

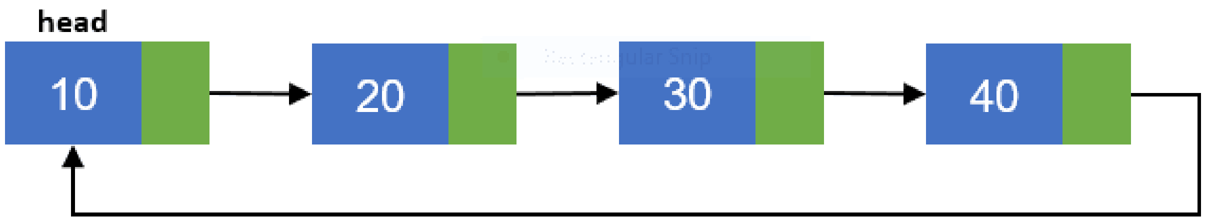


Step 4: Connect the next pointer field of current node with the newly created node which means now next pointer field of current node will point to newNode and you are done.



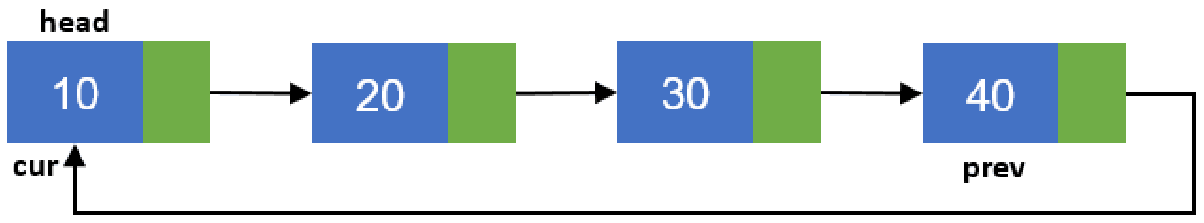
Deleting an element from a circular singly linked list:

Step 1: Create a circular linked list and assign reference of first node to `head`.

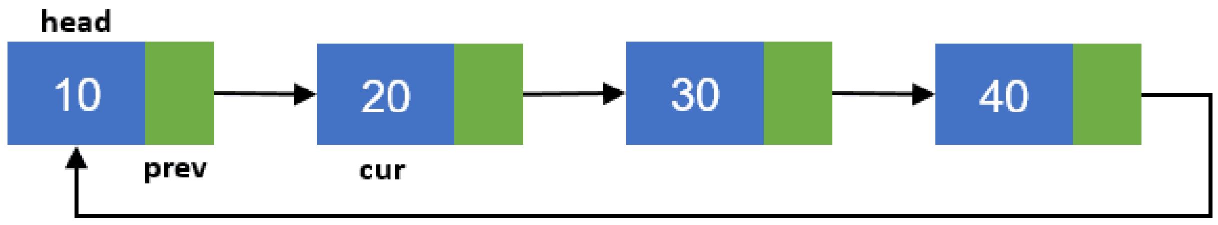


Step 2: Input key to delete from user. Store it in some variable say key. Say key to delete is 20.

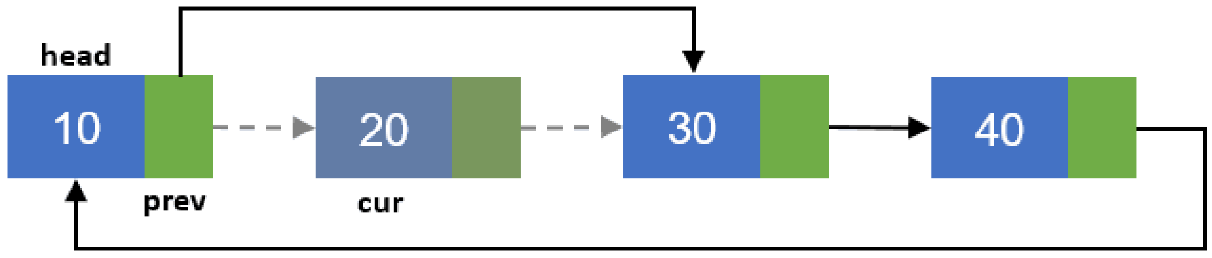
Step 3: To keep track of previous node and node to delete, declare two variables of node type. Say cur = head and prev. Make sure prev points to last node.



Step 4: If current node contains key, i.e. if (cur->data == key). Then you got node to delete.

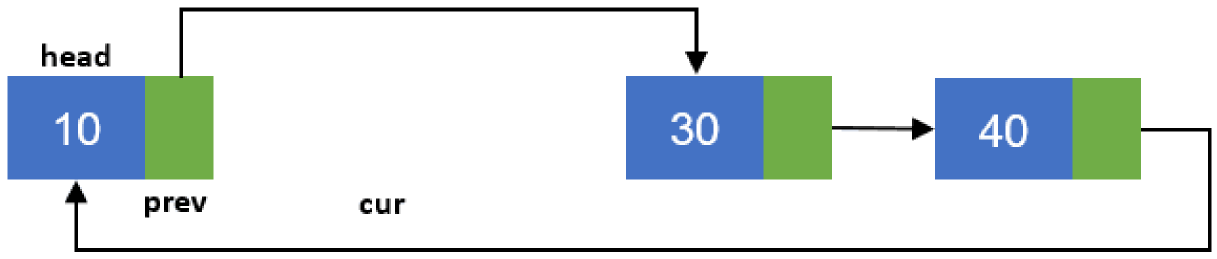


Before deleting a node, you must first adjust previous node link. Link prev->next with cur->next if there are more than one nodes i.e. cur->next != NULL. Otherwise assign prev->next = NULL.

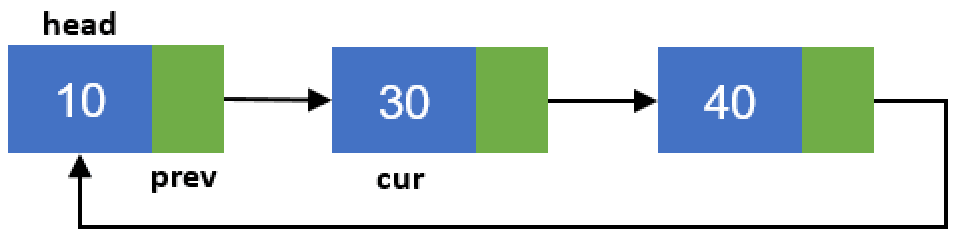


Adjust head node if needed. Means assign head = prev->next if cur == head.

Delete the node using free(cur);.



Update current node, i.e. assign cur = prev->next if prev != NULL. Otherwise assign NULL.



Step 5: If current node does not contain key to delete, then simply update previous and current node. Say prev = cur and cur = cur->next.

Step 6: Repeat step 3-4 till last node.

Circular Doubly Linked list

- Any linked list is called circular doubly linked list if it is a doubly linked list and the first node of the list must be connect to the last node and last node must be connected to the first node.
- It is a sequence of elements/nodes and each element consists of three components,
    - Data: data / value of an element
    - Next: pointer points to the next node in a list

Prev: pointer points to the previous node in a list

Circular doubly linked list insertion at beginning:

Step 1: Create a **new_node** with given value and **new_node → prev** as **NULL** and **new_node→next** as **NULL.**

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

Step 3: If list is **Empty** then, assign **new_node** to **head, new_node→prev,** and to **new_node→next; terminate the function.**

Step 4: If list is **not Empty** then, assign **head** to

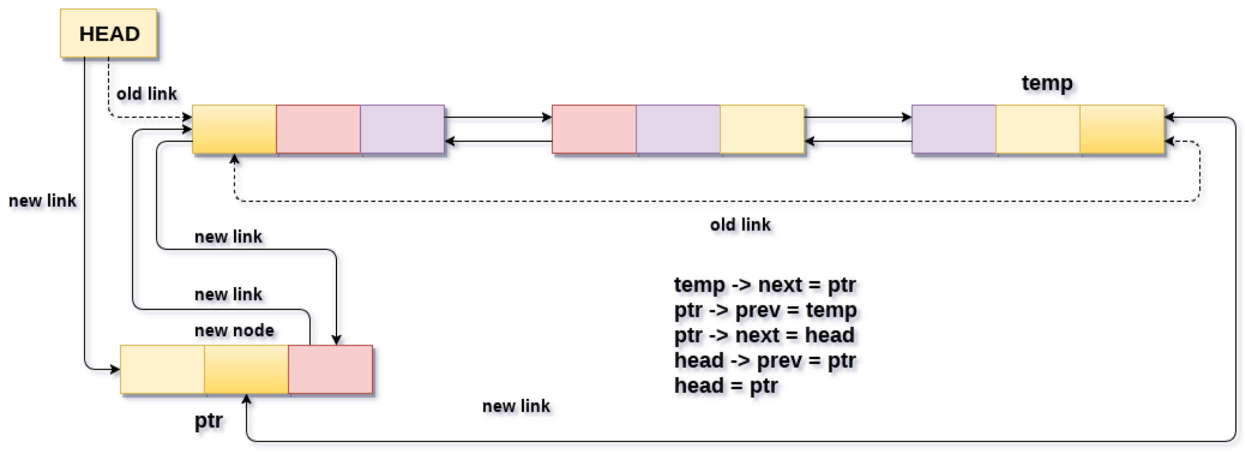
new_node → next,

head→prev to **new_node→prev**,

new_node to **head→prev→next**,

new_node to **head→prev→next,**

new_node to **head**.



Insertion into circular doubly linked list at beginning

Circular doubly linked list insertion at end:

Step 1: Create a **new_node** with given value and **new_node → prev** and **new_node→next** as **NULL**.

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

Step 3: If list is **Empty** then, assign **new_node** to **head, new_node→prev,** and to **new_node→next; terminate the function.**

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

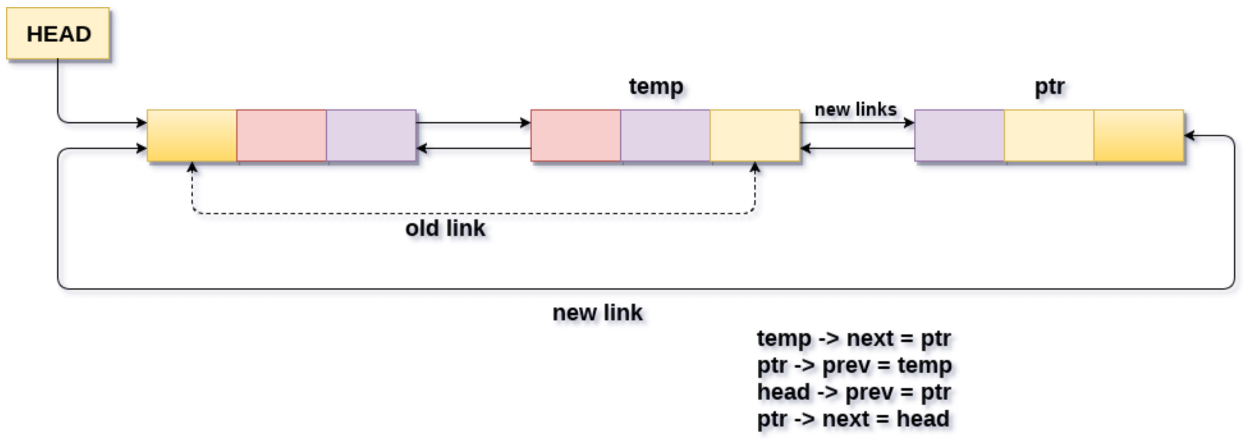
Step 5 : **If the list is not empty, then**

Set **head** to **new_node→next**,

head→prev to **new_node→prev,**

new_node to **head→prev**,

new_node to **head→prev→next.**



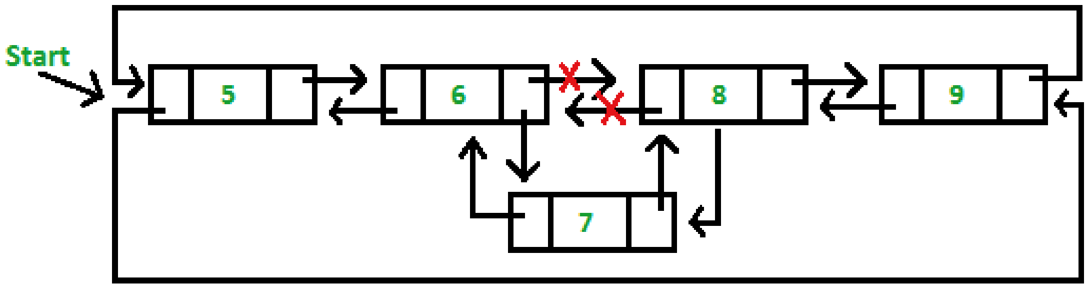
Insertion into circular doubly linked list at end

Circular doubly linked list insertion after a given node:

- **Step 1**
  - Create a **new_node** with given value and initialize its pointers with NULL.
- **Step 2**
  - Check whether list is **Empty** (**head** == **NULL**)
- **Step 3**
  - If list is **Empty then display insertion is not possible**, and **return**.
- **Step 4**
  - If list is **not Empty** then, define a pointer **temp** and initialize **temp** with **head**.
- **Step 5**
  - Keep moving the **temp** until it reaches to the node after which we want to insert the new_node or the list exhausted (while((**temp→data!= desired) && temp!=head**)).
- **Step 6**

    - If temp is equal to head then Display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function.
- **Step 7**
    - **new_node→prev=temp,**
      new_node → next=temp→next; temp→next=new_node;

new_node->next->prev=new_node;



Deletion from Beginning:

Step 1 - Check whether list is Empty (**head == NULL**)

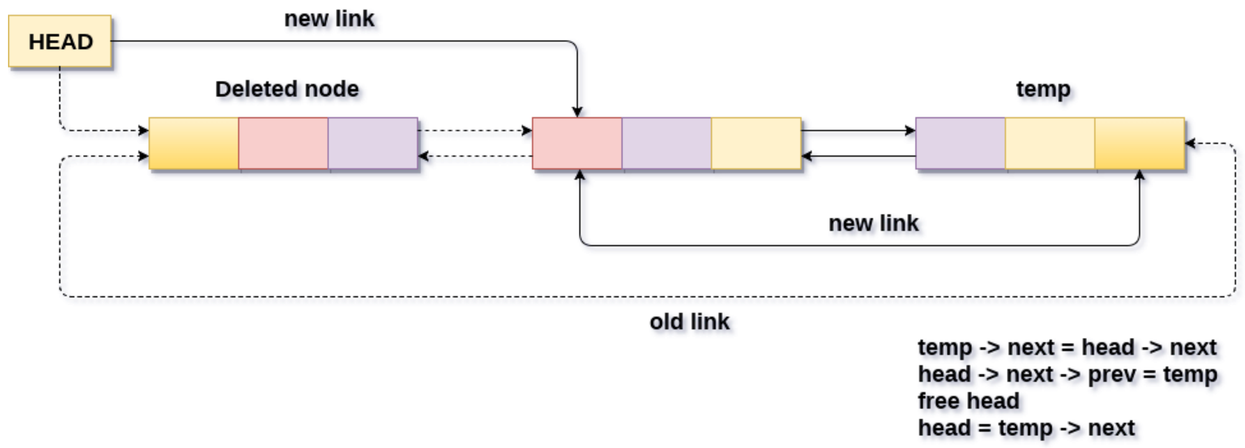
Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If list 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 == temp && temp→prev = temp**)

Step 4.1 - If list is having only one node, then set head to NULL and delete temp (Setting Empty list conditions)

Step 4.2 - If list is with more than one node, then assign **temp → next** to **head**, **temp→next** to **temp→prev→next, temp→prev** to **temp→next→prev,** and delete temp.



Deletion in circular doubly linked list at beginning

Deletion from End in doubly circular linked list:

Step 1 - Check whether list is Empty (**head == NULL**)

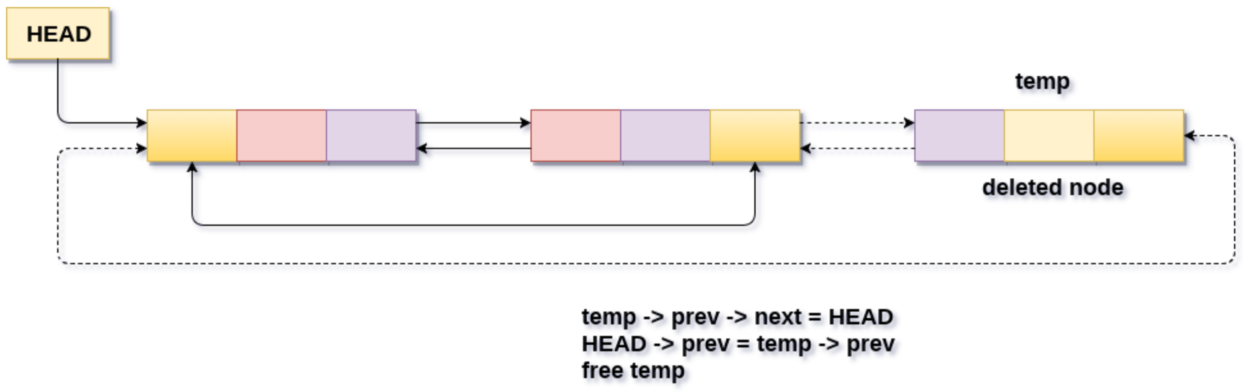
Step 2 - If list is Empty, then display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If list is not Empty then, define a Node pointer 'temp' and initialize with **head→prev**.

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

Step 5 - If list has only one node, then assign **NULL to head** and delete temp. Terminate the function. (Setting Empty list condition)

Step 6 - If list has more than one node, Assign head to temp → previous → next, temp→prev to head→prev and delete temp.



Deletion in circular doubly linked list at beginning

Delete a node after a specific node in the list:

Step 1 - Check whether list is Empty (**head == NULL**)

Step 2 - If list is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If list is not Empty, then define a Node pointer 'temp' and initialize with head also define a Node pointer ptr.

Step 4 - Keep moving the temp until it reaches to the location after a node to be deleted or to the last node.(**while (temp→data != desired && temp→next != head))**.

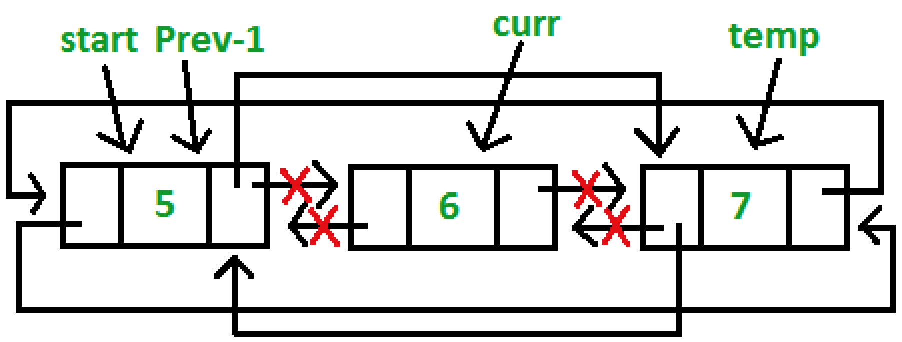
Step 5 – If temp points to last node(**temp→next== head)**, then display 'Deletion is not possible!' and **terminate the function**.

Step 6 – Set ptr to temp→next;

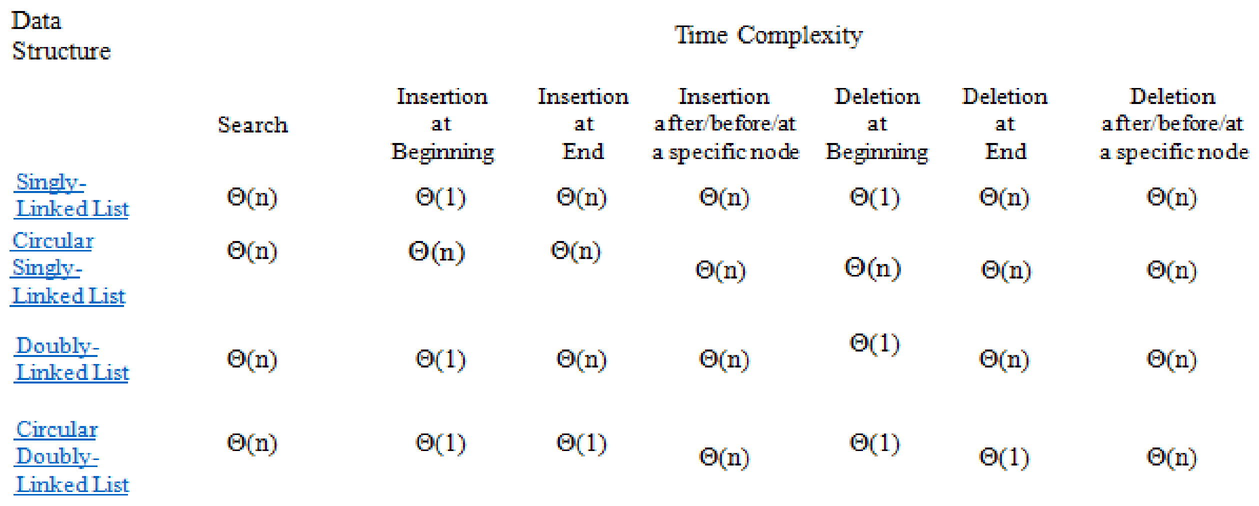
Set temp→next=ptr→next;

ptr→next→prev=temp;

free(ptr);



Complexity analysis



Data Structure	Time Complexity						
	Search	Insertion at Beginning	Insertion at End	Insertion after/before/at a specific node	Deletion at Beginning	Deletion at End	Deletion after/before/at a specific node
Singly-Linked List	Θ(n)	Θ(1)	Θ(n)	Θ(n)	Θ(1)	Θ(n)	Θ(n)
Circular Singly-Linked List	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(n)
Doubly-Linked List	Θ(n)	Θ(1)	Θ(n)	Θ(n)	Θ(1)	Θ(n)	Θ(n)
Circular Doubly-Linked List	Θ(n)	Θ(1)	Θ(1)	Θ(n)	Θ(1)	Θ(1)	Θ(n)