

Data Structures

IT 205

Dr. Manish Khare



Lecture – 5 & 6
12-Jan-2018



Linked List

Linked List

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.
 - **Link** – Each link of a linked list can store a data called an element.
 - **Next** – Each link of a linked list contains a link to the next link called Next.
 - **LinkedList** – A Linked List contains the connection link to the first link called First.

Why Linked List?

- Arrays can be used to store linear data of similar types, but arrays have following limitations.
 - 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
 - 2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to be shifted.
- For example, in a system if we maintain a sorted list of IDs in an array `id[]`.
 - `id[] = [1000, 1010, 1050, 2000, 2040]`.
- And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
- Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.

Advantages and Drawback over array

➤ Advantages over arrays

- 1) Dynamic size
- 2) Ease of insertion/deletion

➤ Drawbacks:

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
- 2) Extra memory space for a pointer is required with each element of the list.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.



- As per the above illustration, following are the important points to be considered.
- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Representation of Linked List in C/C++

- A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
- Each node in a list consists of at least two parts:
 - 1) data
 - 2) pointer to the next node
- In C/C++, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
// A linked list node
struct Node
{
    int data;
    struct Node *next;
};
```




➤ Program for Creation of Linked List

Types of Linked List

➤ Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Single Linked List – Basic Operations

- 
- Following are the basic operations supported by a list.
- **Insertion** – Adds an element at the beginning of the list.
 - **Deletion** – Deletes an element at the beginning of the list.
 - **Display** – Displays the complete list.
 - **Search** – Searches an element using the given key.
 - **Delete** – Deletes an element using the given key.

Insertion In Linked list



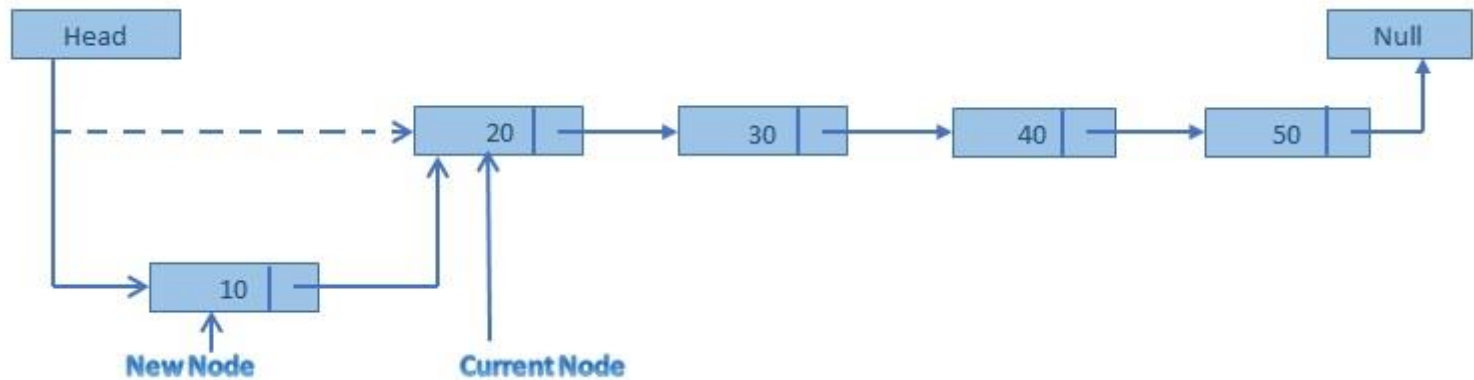
➤ There are three situation for inserting element in list.

- Insertion at the front of list.
- Insertion in the middle of the list.
- Insertion at the end of the list.

Insertion at the front of list

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then,
set **newNode**→**next** = **NULL** and **head** = **newNode**.
- **Step 4:** If it is **Not Empty** then,
set **newNode**→**next** = **head** and **head** = **newNode**.

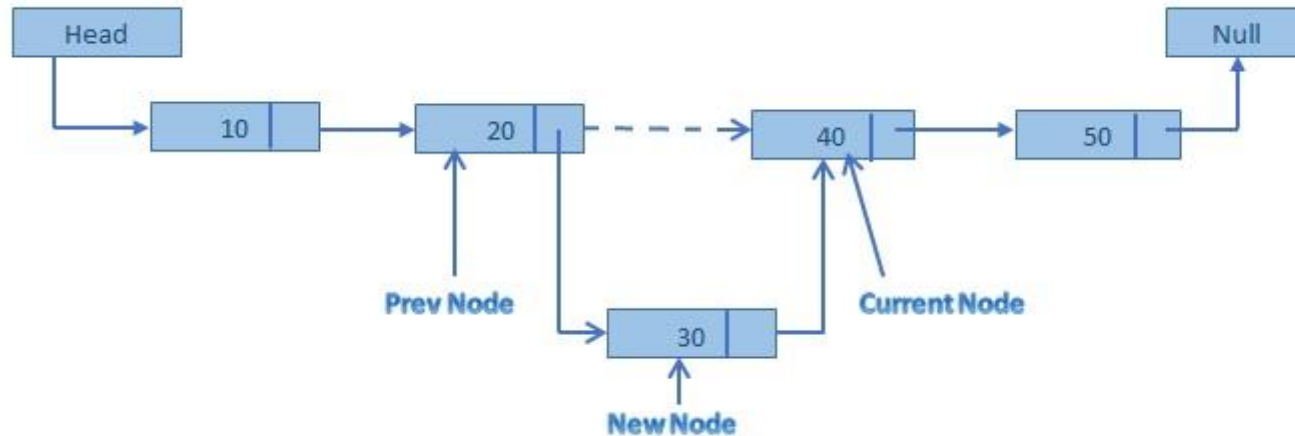
Insertion at the front of list



Insertion Node in given location Linked List

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set **newNode** \rightarrow **next = NULL** and **head = newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** \rightarrow **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6:** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** Finally, Set '**newNode** \rightarrow **next = temp** \rightarrow **next**' and '**temp** \rightarrow **next = newNode**'

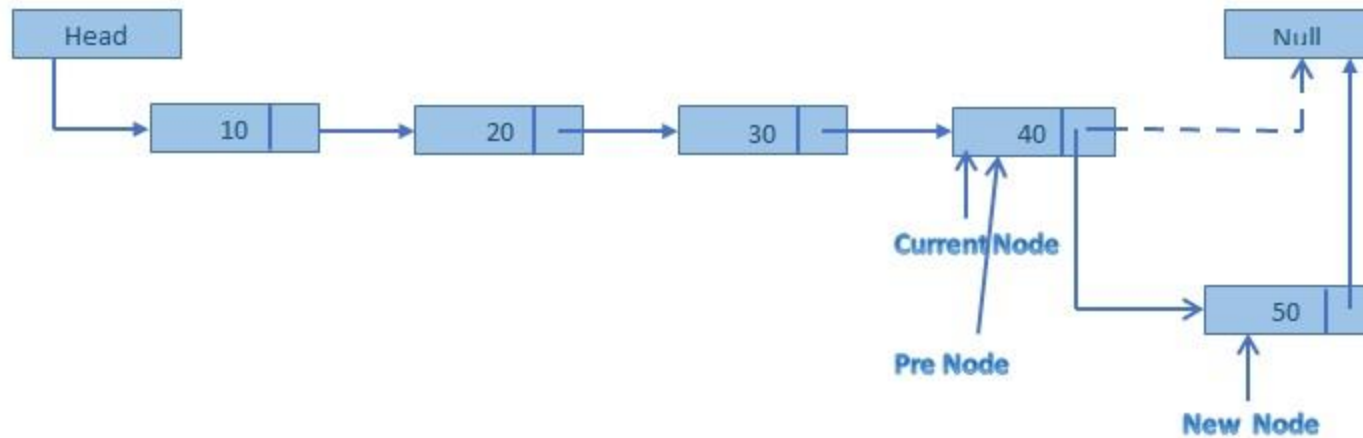
Insertion Node in given location Linked List



Insertion at the end of the list.

- **Step 1:** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head == NULL**).
- **Step 3:** If it is **Empty** then, set **head = newNode**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6:** Set **temp** → **next = newNode**.

Insertion at the end of the list.



Deletion In Singly linked list



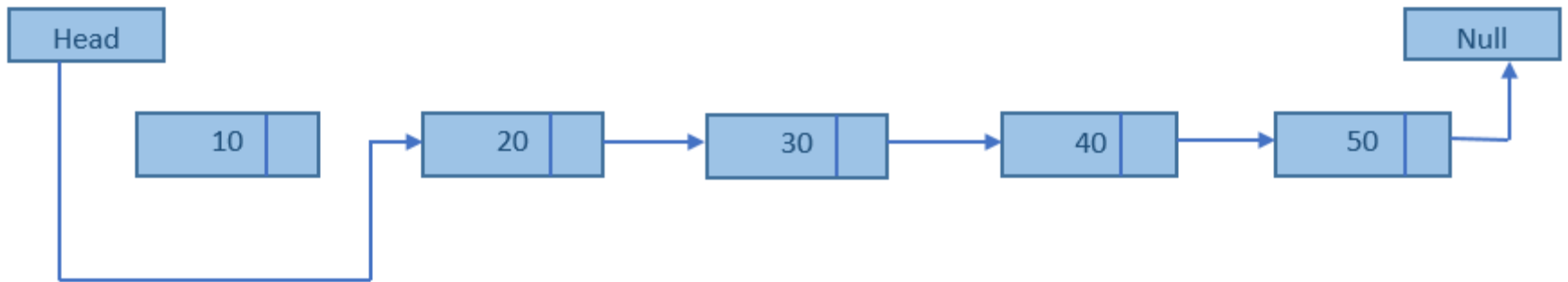
➤ There are three situation for Deleting element in list.

1. Deletion at beginning of the list.
2. Deletion at the middle of the list.
3. Deletion at the end of the list.

Deletion at beginning of the list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4:** Check whether list is having only one node (**temp → next == NULL**)
- **Step 5:** If it is **TRUE** then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE** then set **head = temp → next**, and delete **temp**.

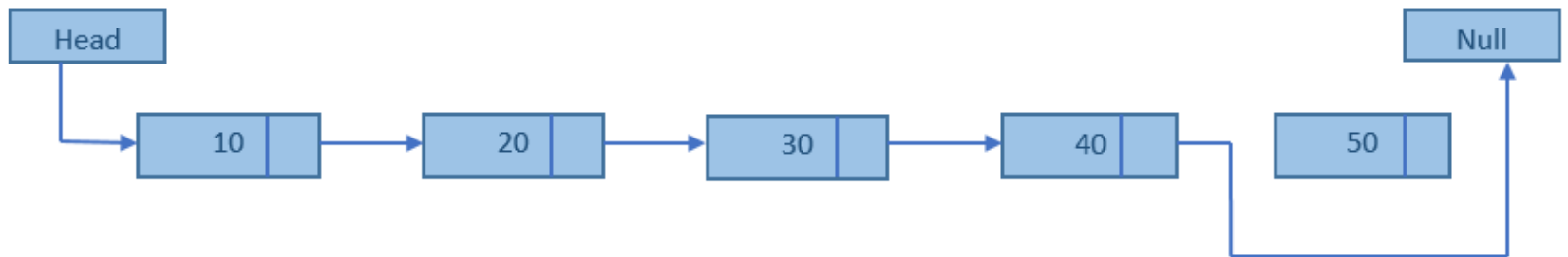
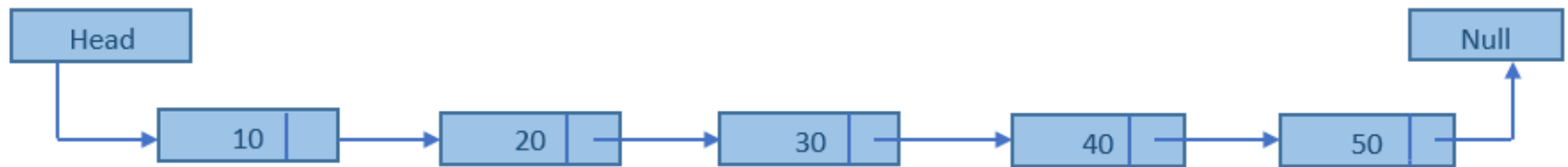
Deletion at beginning of the list



Deletion at Last Node of the Linked List

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1 → next == NULL**)
- **Step 5:** If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next == NULL**)
- **Step 7:** Finally, Set **temp2 → next = NULL** and delete **temp1**.

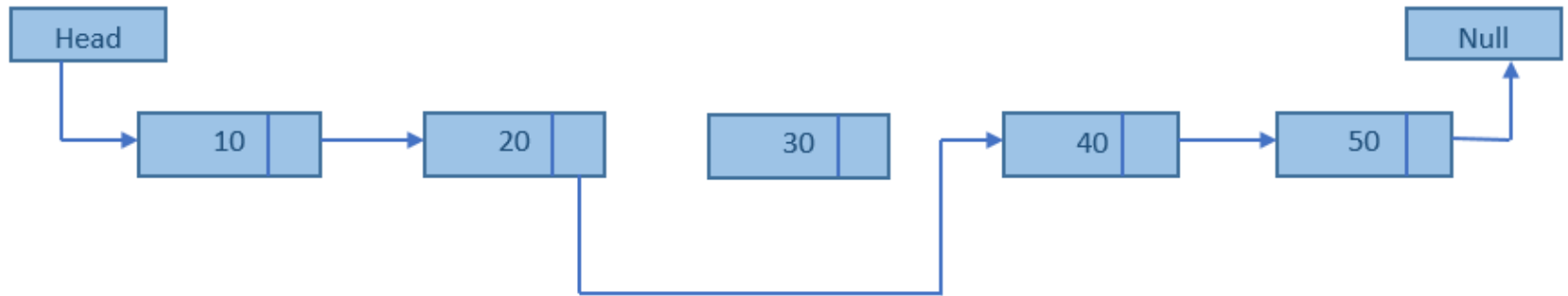
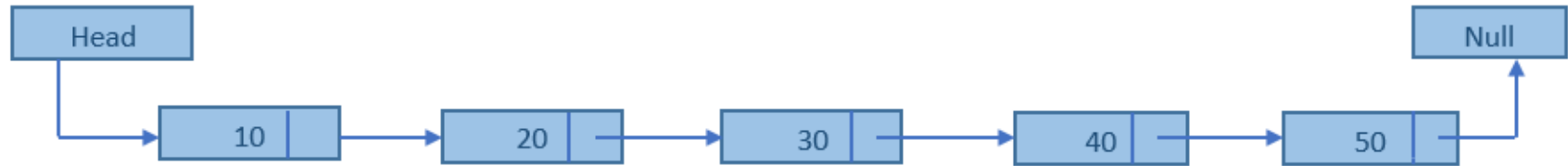
Deletion at Last Node of the Linked List



Deletion of Node in given location of Linked List

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5:** If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7:** If list has only one node and that is the node to be deleted, then set **head = NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11:** If **temp1** is last node then set **temp2 → next = NULL** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).


Deletion of Node in given location of Linked List



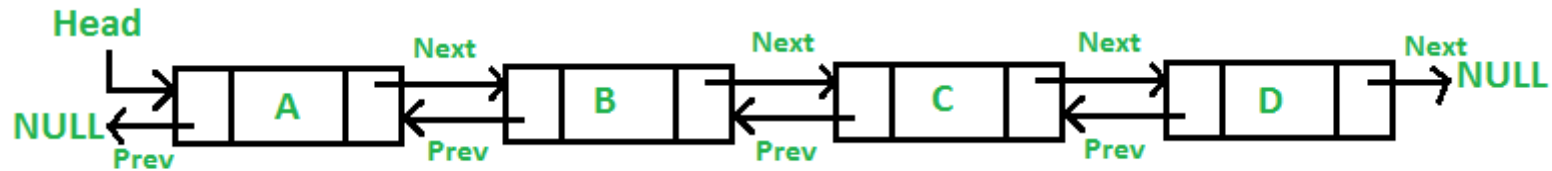
Displaying a Single Linked List

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

Doubly Linked List

- 
- Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.
 - **Link** – Each link of a linked list can store a data called an element.
 - **Next** – Each link of a linked list contains a link to the next link called Next.
 - **Prev** – Each link of a linked list contains a link to the previous link called Prev.
 - **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List



- As per the above illustration, following are the important points to be considered.
- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Doubly Linked List

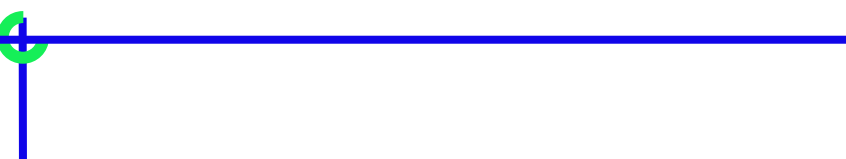
➤ Advantages over singly linked list

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given. In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

➤ Disadvantages over singly linked list

- 1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though.
- 2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Doubly Linked List - Basic Operations

- 
- Following are the basic operations supported by a list.
 - **Insertion** – Adds an element at the beginning of the list.
 - **Deletion** – Deletes an element at the beginning of the list.
 - **Insert Last** – Adds an element at the end of the list.
 - **Delete Last** – Deletes an element from the end of the list.
 - **Insert After** – Adds an element after an item of the list.
 - **Delete** – Deletes an element from the list using the key.
 - **Display forward** – Displays the complete list in a forward manner.
 - **Display backward** – Displays the complete list in a backward manner.

Insertion in doubly linked list



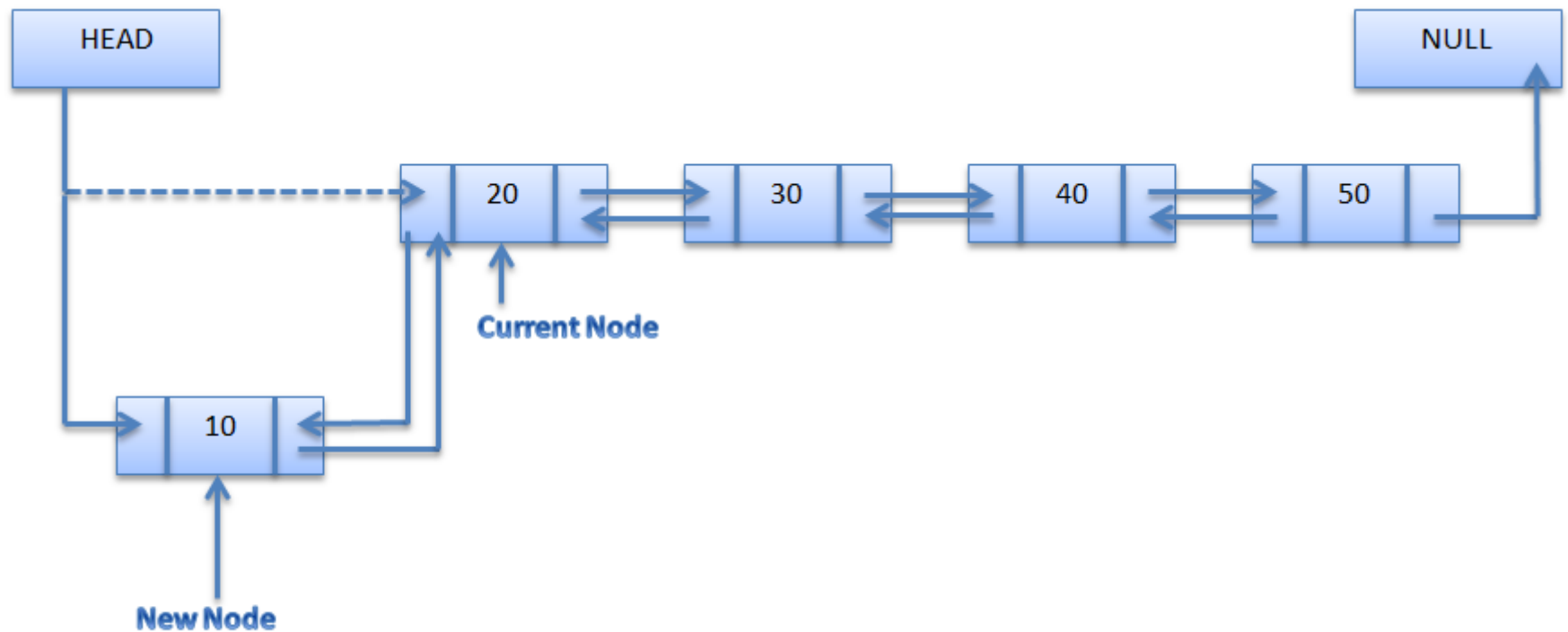
➤ There are three situation for inserting element in list.

- 1.Insertion at the front of list.
- 2.Insertion in the middle of the list.
- 3.Insertion at the end of the list.

Insertion at the beginning of doubly linked list

- **Step 1:** Create a **newNode** with given value and **newNode** → **previous** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, assign **NULL** to **newNode** → **next** and **newNode** to **head**.
- **Step 4:** If it is **not Empty** then, assign **head** to **newNode** → **next** and **newNode** to **head**.

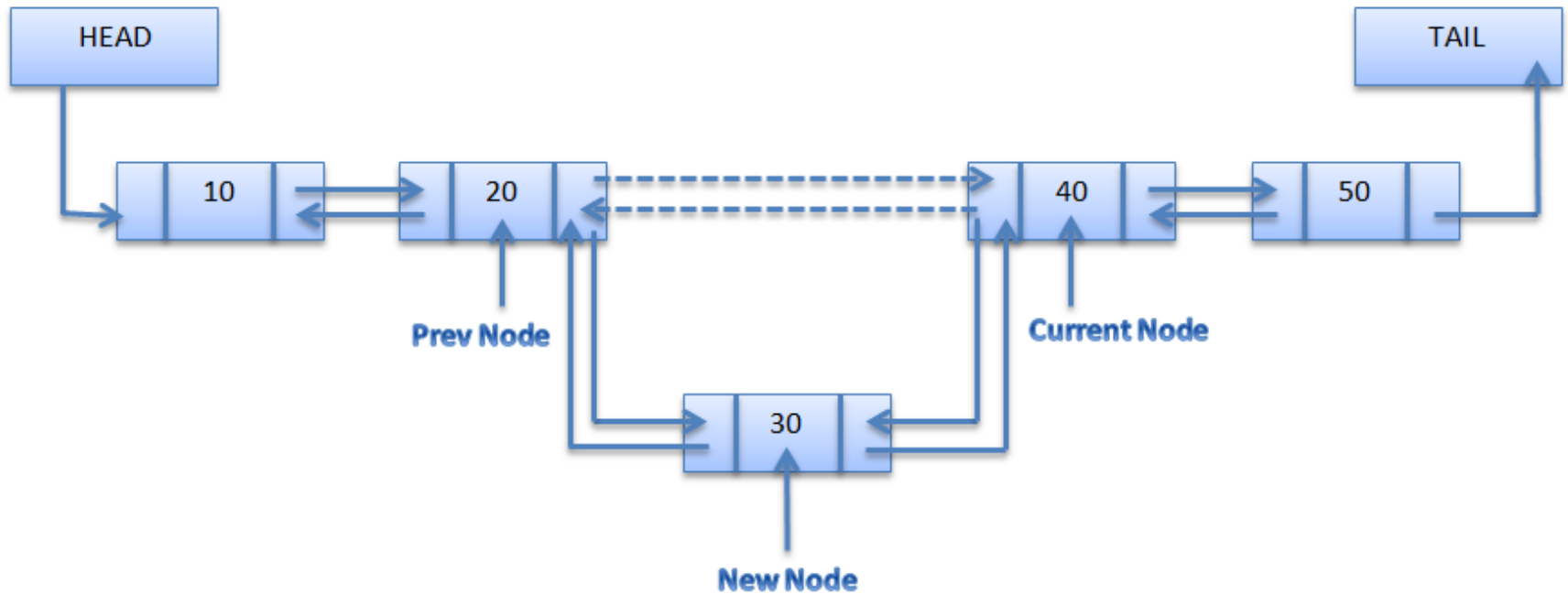
Insertion at the beginning of doubly linked list



Insertion in the location of doubly linked list

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, assign **NULL** to **newNode** → **previous** & **newNode** → **next** and **newNode** to **head**.
- **Step 4:** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5:** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the **newNode** (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the **newNode**).
- **Step 6:** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7:** Assign **temp1** → **next** to **temp2**, **newNode** to **temp1** → **next**, **temp1** to **newNode** → **previous**, **temp2** to **newNode** → **next** and **newNode** to **temp2** → **previous**.

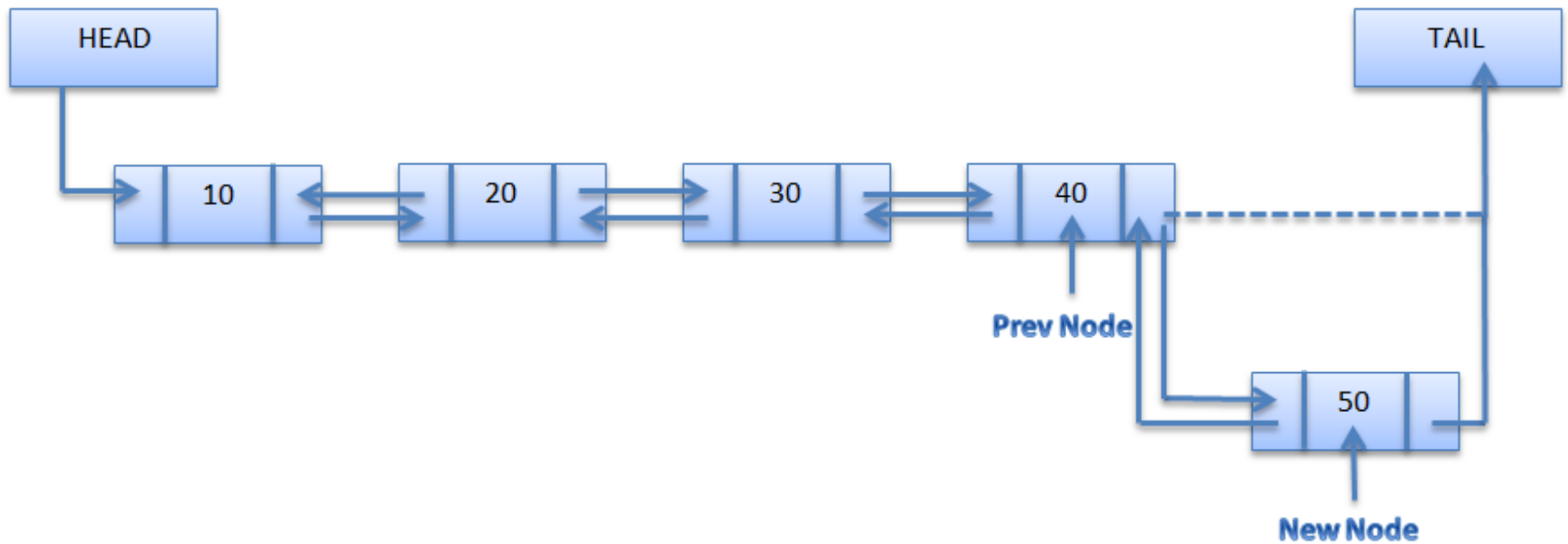
Insertion in the location of doubly linked list




Insertion at last of doubly linked list

- **Step 1:** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty**, then assign **NULL** to **newNode** → **previous** and **newNode** to **head**.
- **Step 4:** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6:** Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

Insertion at last of doubly linked list



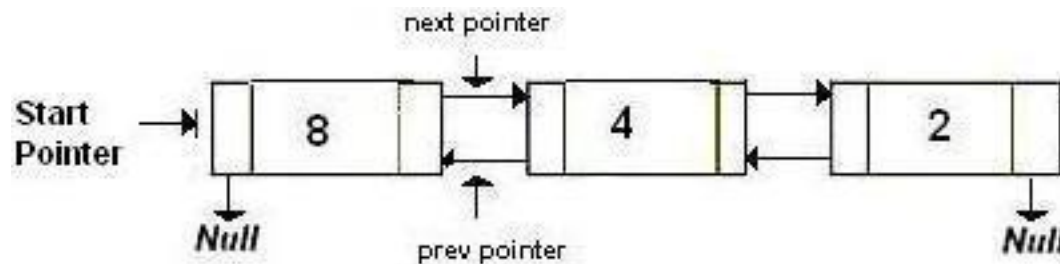
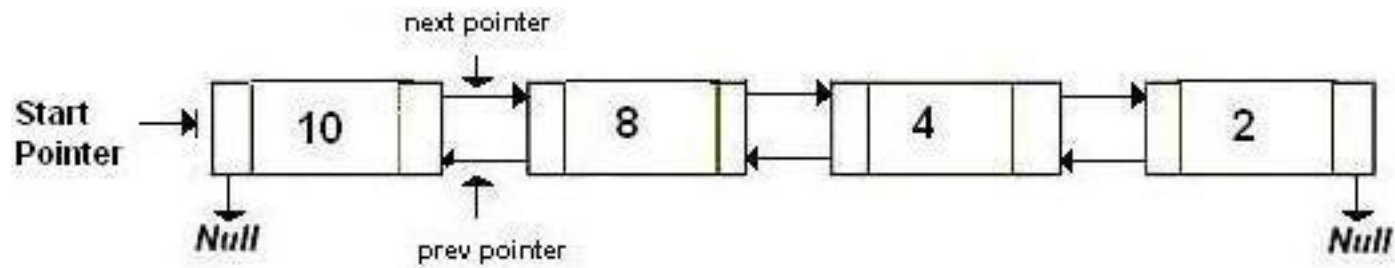
Deletion in doubly linked list

- 
- In a double linked list, the deletion operation can be performed in three ways as follows...
 - Deleting from Beginning of the list
 - Deleting from End of the list
 - Deleting a Specific Node

Deletion from the beginning of doubly linked list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is not Empty then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4:** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5:** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

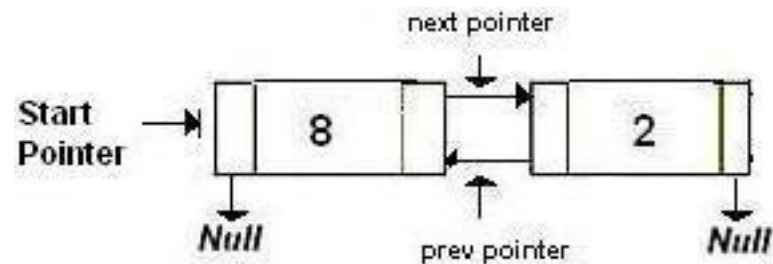
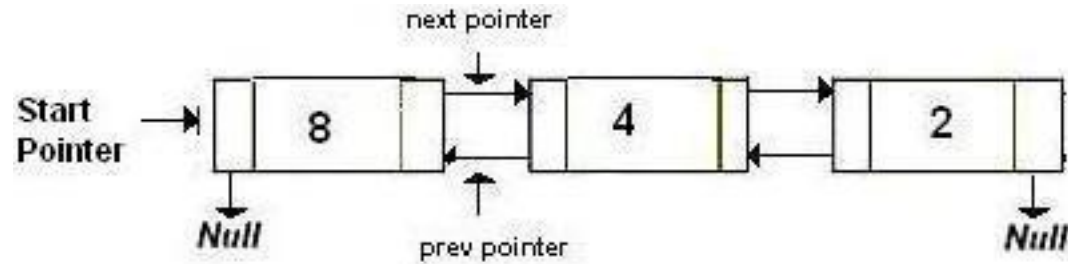
Deletion from the beginning of doubly linked list



Deletion in the location of doubly linked list

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

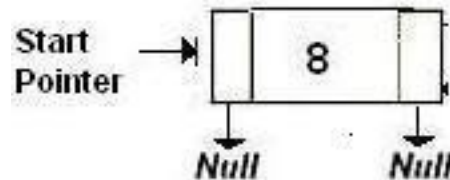
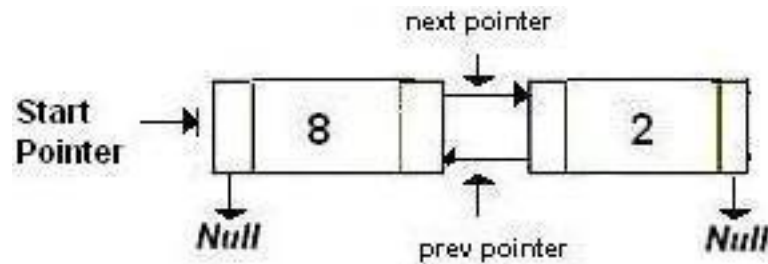
Deletion in the location of doubly linked list



Deletion at last of doubly linked list

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

Deletion at last of doubly linked list

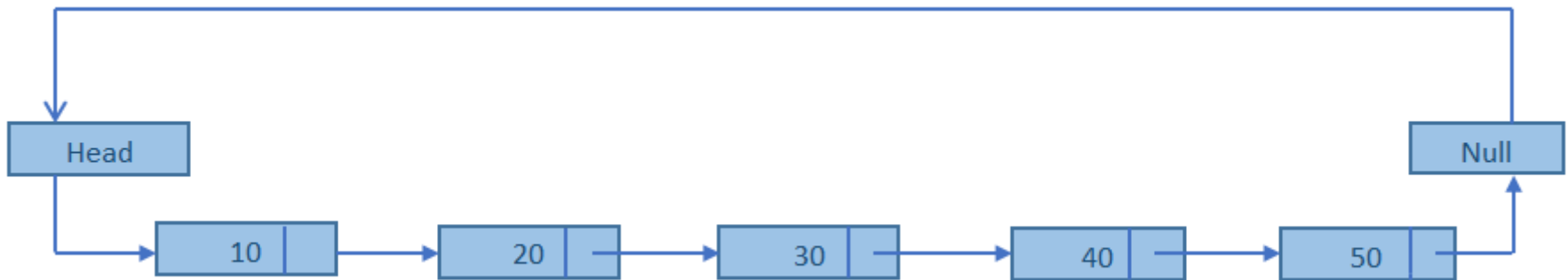


Traversal of Double Linked List

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

Circular Linked List

- Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element.
- Both Singly Linked List and Doubly Linked List can be made into a circular linked list.



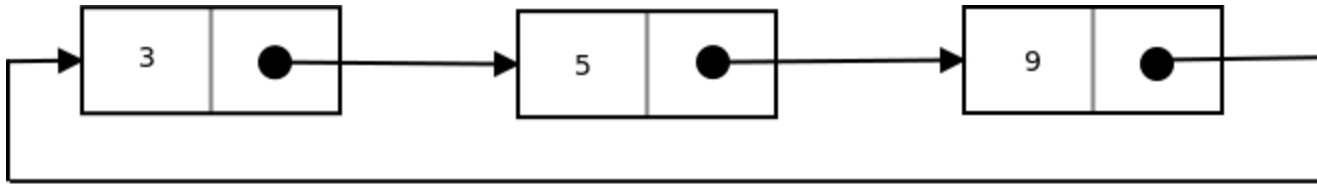
Advantages of Circular Linked Lists

- 1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- 2) Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- 3) Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

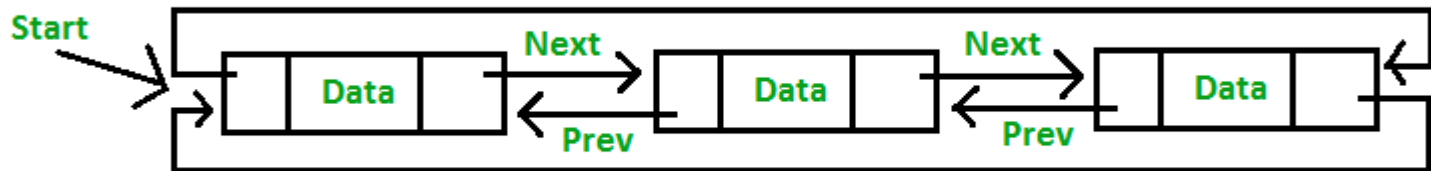
Why Circular Linked List

- **Why Circular?** In a singly linked list, for accessing any node of linked list, we start traversing from the first node.
- 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.
- In a singly linked list, next part (pointer to next node) is NULL, if we utilize this link to point to the first node then we can reach preceding nodes.

➤ Singly linked List as Circular Linked List



➤ Doubly Linked List as Circular Linked List



Insertion In Circular Linked List



➤ There are three situation for inserting element in Circular linked list.

1. Insertion at the front of Circular linked list.

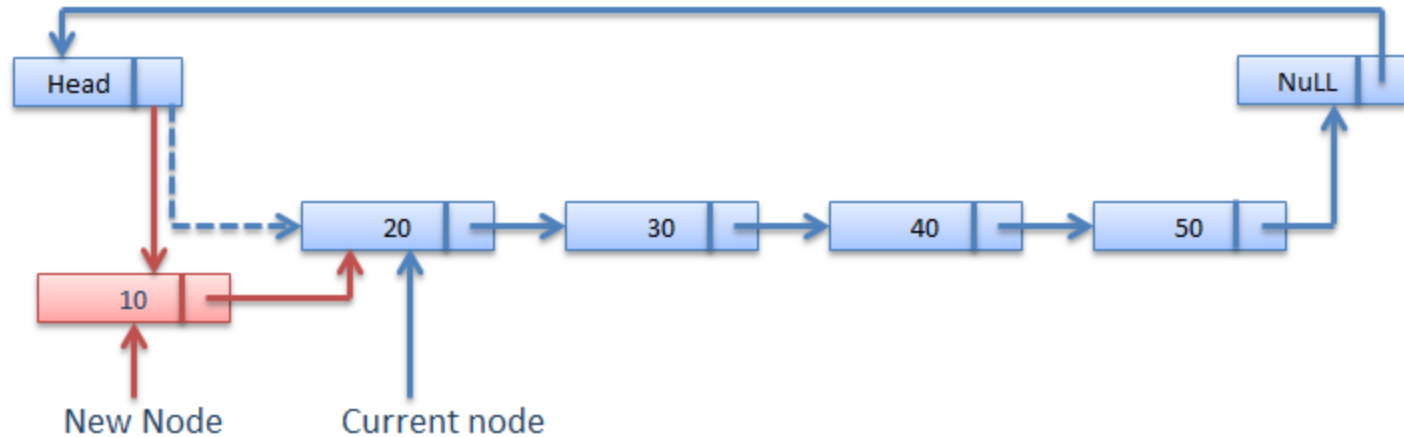
2. Insertion in the middle of the Circular linked list.

3. Insertion at the end of the Circular linked list.

Insertion at the front of Circular linked list

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then,
set **head = newNode** and **newNode → next = head** .
- **Step 4:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5:** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').
- **Step 6:** Set '**newNode → next = head**', '**head = newNode**' and '**temp → next = head**'.

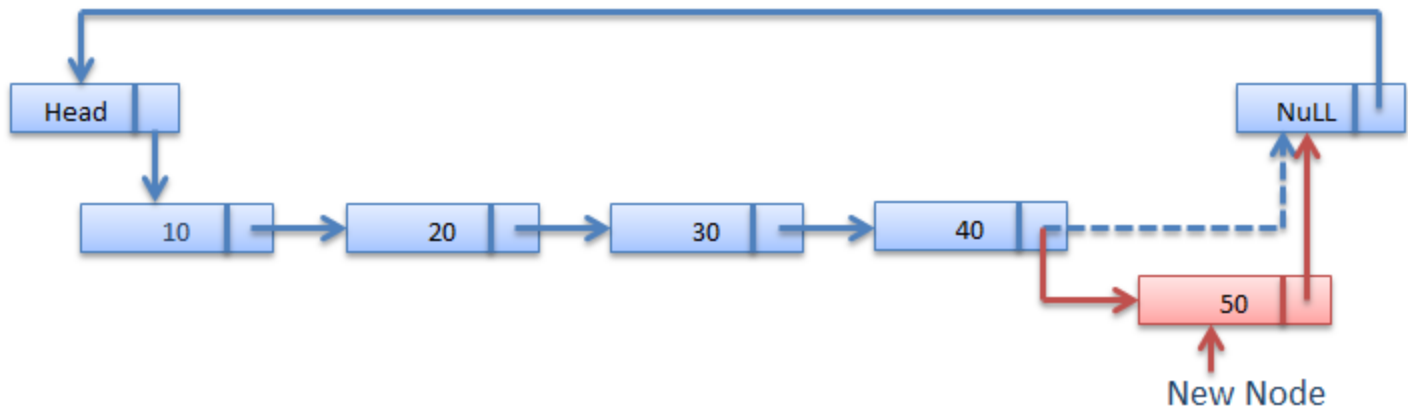
Insertion at the front of Circular linked list



Insertion at the end of Circular linked list

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**).
- **Step 3:** If it is **Empty** then, set **head = newNode** and **newNode** → **next = head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next == head**).
- **Step 6:** Set **temp** → **next = newNode** and **newNode** → **next = head**.

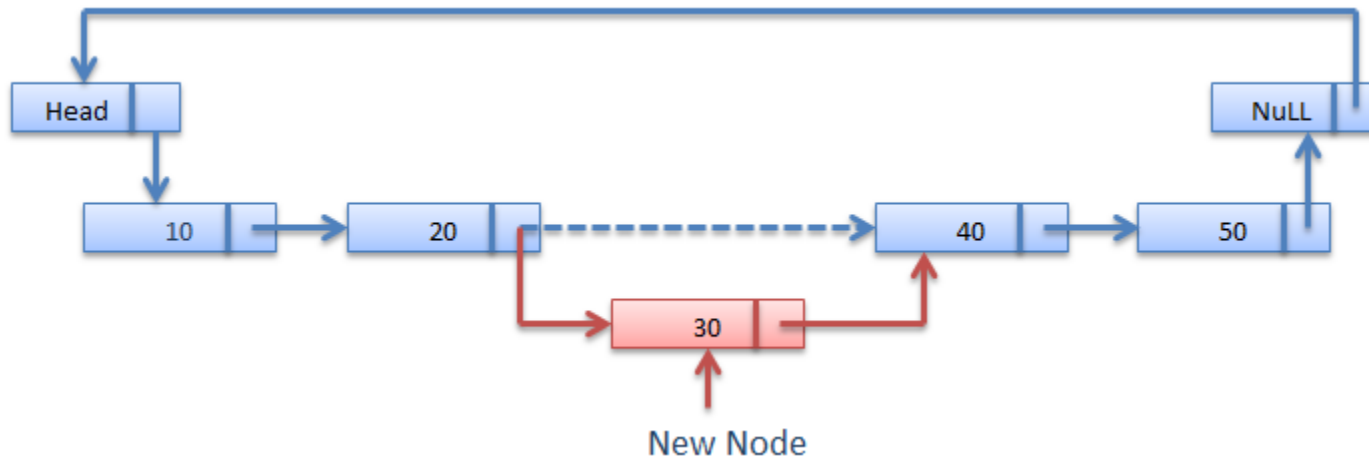
Insertion at the end of Circular linked list




Insertion At Location in Circular linked list

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set **head = newNode** and **newNode → next = head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5:** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6:** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp** to next node.
- **Step 7:** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (**temp → next == head**).
- **Step 8:** If **temp** is last node then set **temp → next = newNode** and **newNode → next = head**.
- **Step 8:** If **temp** is not last node then set **newNode → next = temp → next** and **temp → next = newNode**.

Insertion At Location in Circular linked list



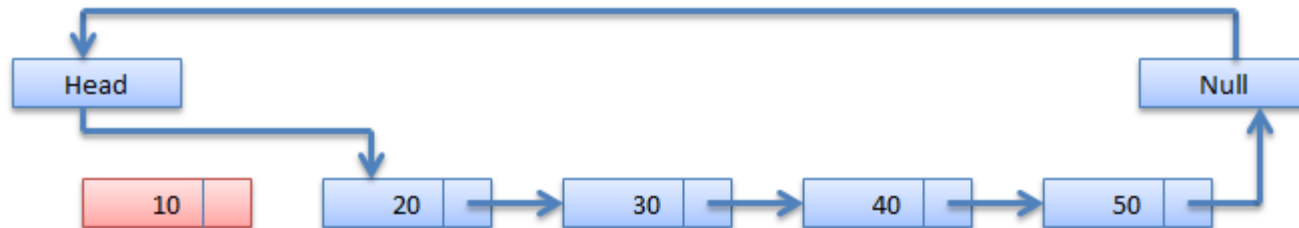
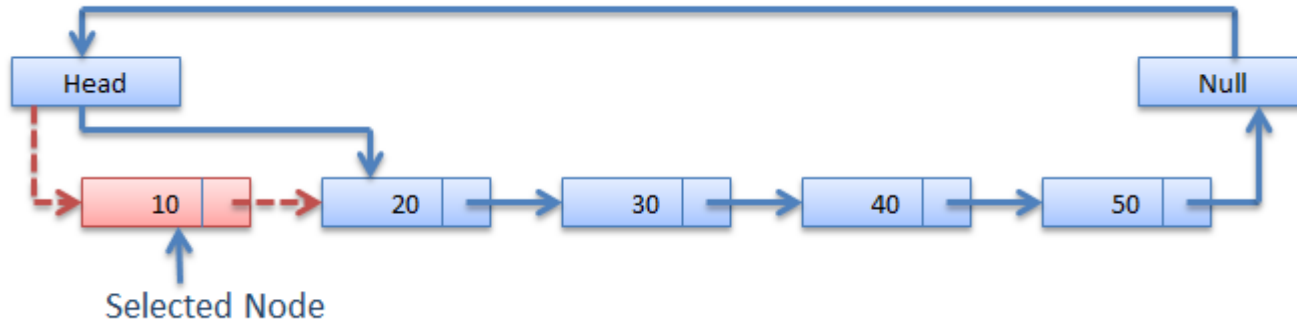
Deletion In Circular Linked List

- 
- In a circular linked list, the deletion operation can be performed in three ways those are as follows...
 - Deleting from Beginning of the list
 - Deleting from End of the list
 - Deleting a Specific Node

Deletion at beginning of the Circular linked list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4:** Check whether list is having only one node (**temp1 → next == head**)
- **Step 5:** If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6:** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)
- **Step 7:** Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**.

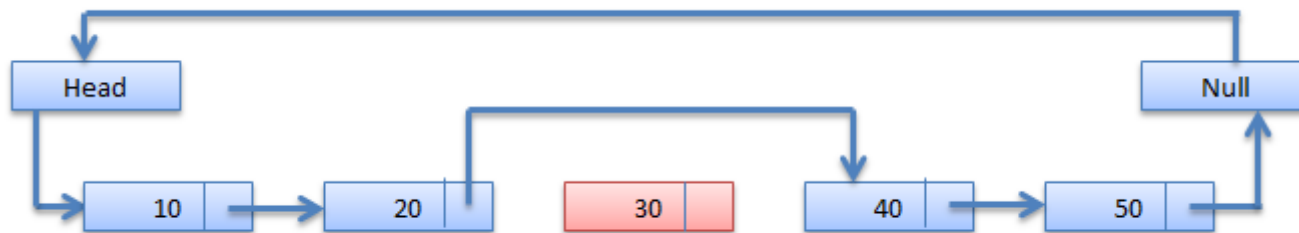
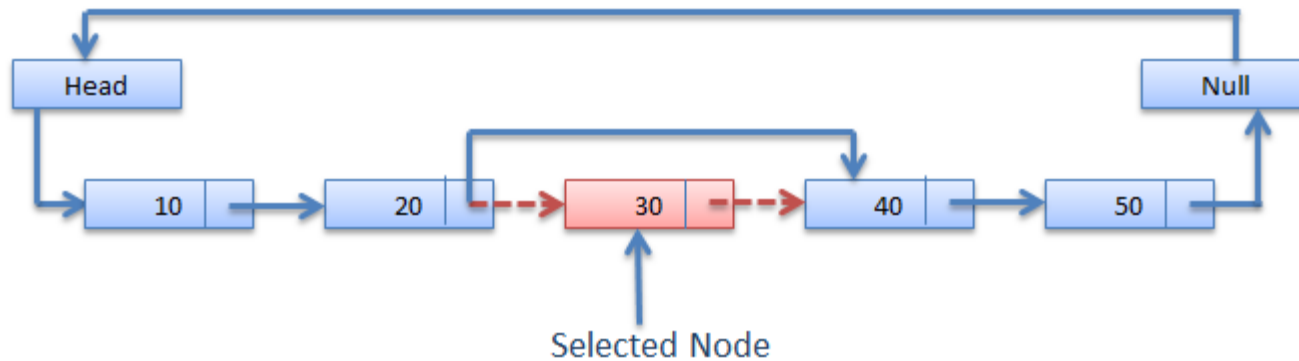
Deletion at beginning of the Circular linked list



Deletion at the middle of the Circular linked list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5:** If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- **Step 6:** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next == head**)
- **Step 7:** If list has only one node and that is the node to be deleted then set **head = NULL** and delete **temp1** (**free(temp1)**).
- **Step 8:** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9:** If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next = head** and delete **temp1**.
- **Step 10:** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 11:** If **temp1** is last node then set **temp2 → next = head** and delete **temp1** (**free(temp1)**).
- **Step 12:** If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).

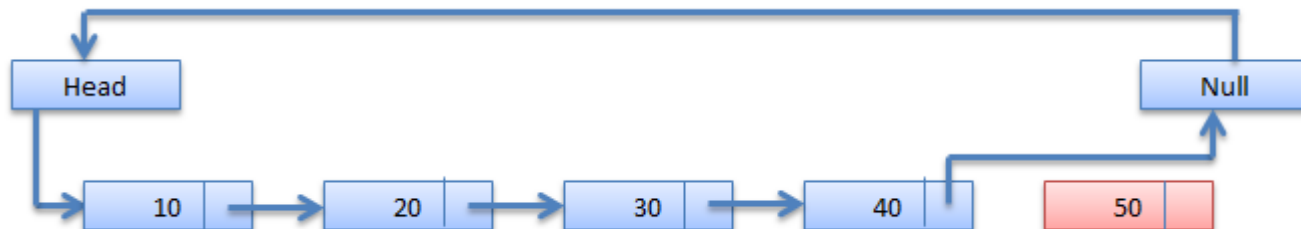
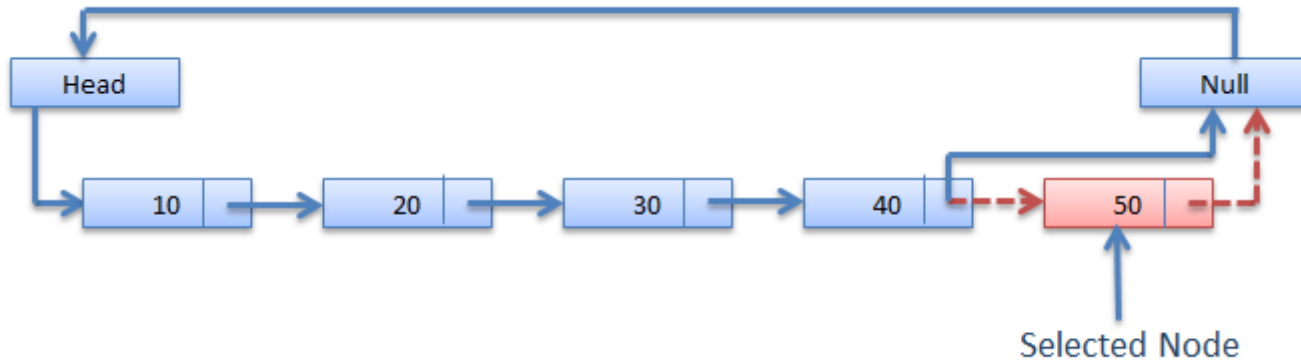
Deletion at the middle of the Circular linked list



Deletion at the end of the Circular linked list

- **Step 1:** Check whether list is **Empty** (**head == NULL**)
- **Step 2:** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4:** Check whether list has only one Node (**temp1 → next == head**)
- **Step 5:** If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6:** If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
- **Step 7:** Set **temp2 → next = head** and delete **temp1**.

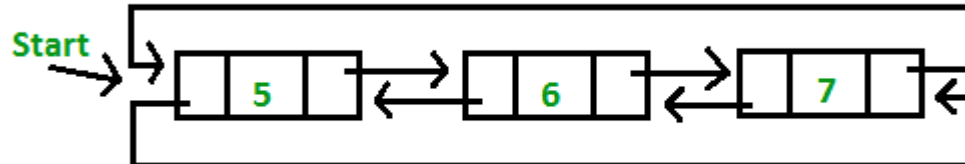
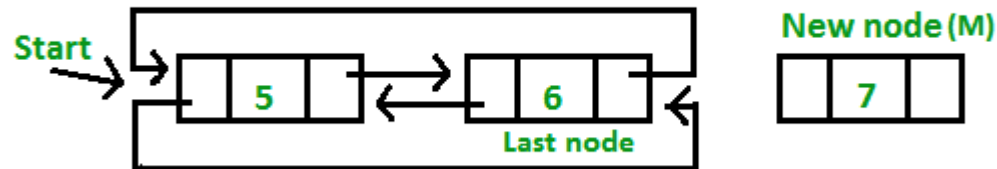
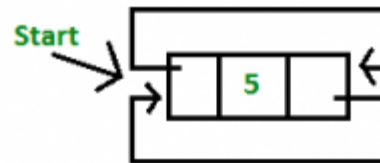
Deletion at the end of the Circular linked list



Insertion in Doubly Linked List

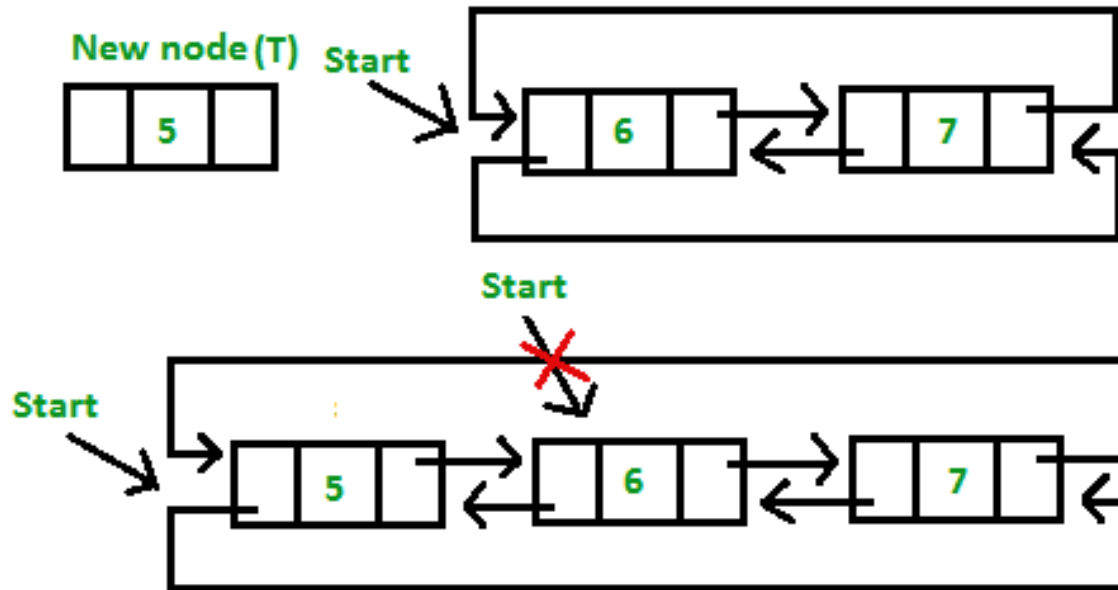
➤ Insertion at the end of list or in an empty list

Start → NULL



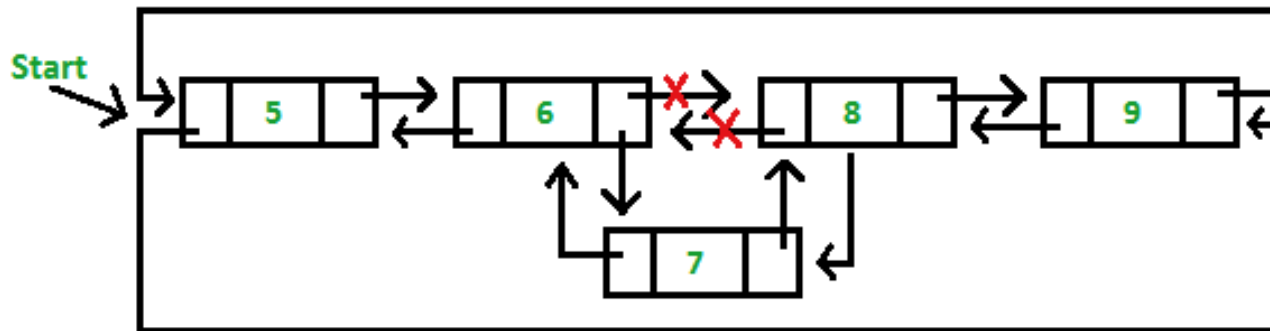
Insertion in Doubly Linked List

➤ Insertion at the beginning of the list



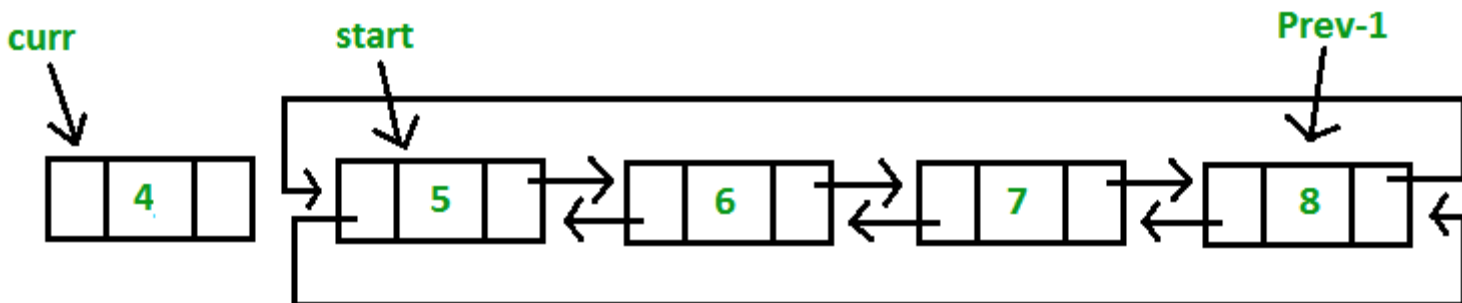
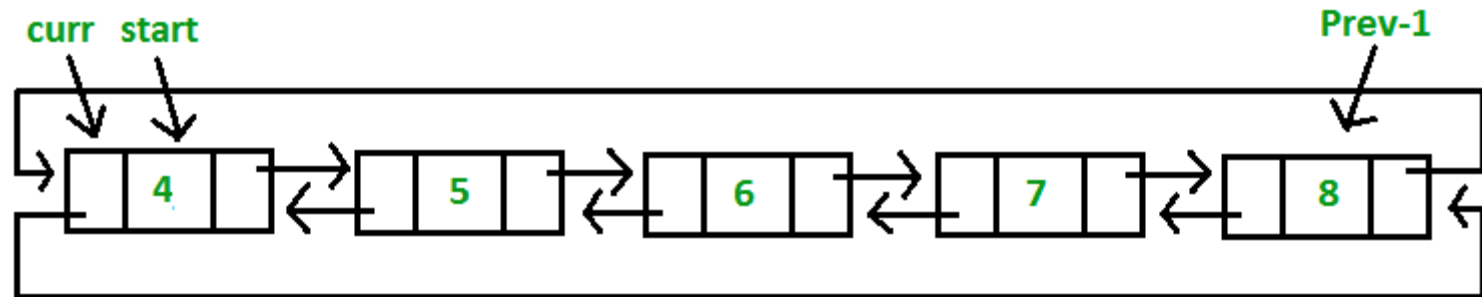
Insertion in Doubly Linked List

- Insertion in between the nodes of the list



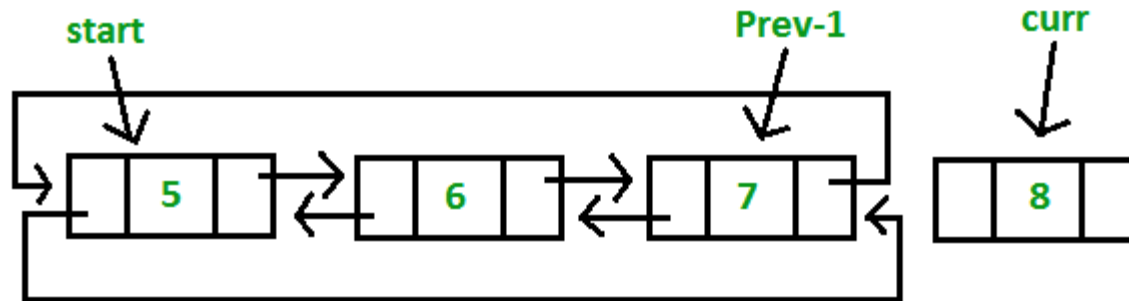
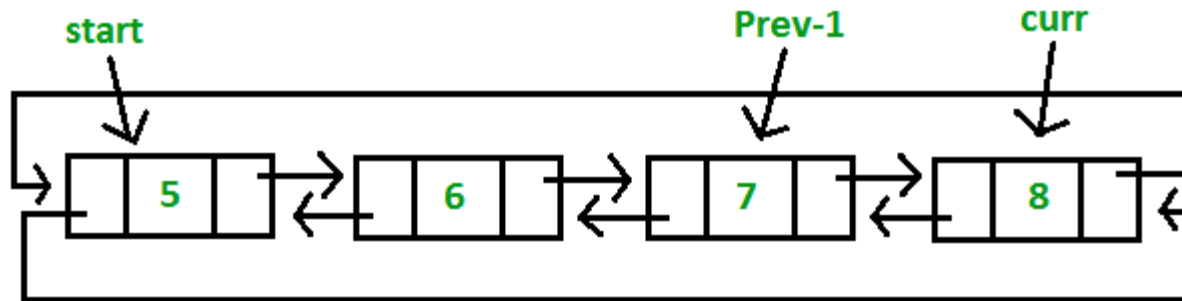
Deletion in Doubly Linked List

➤ Deletion form the front



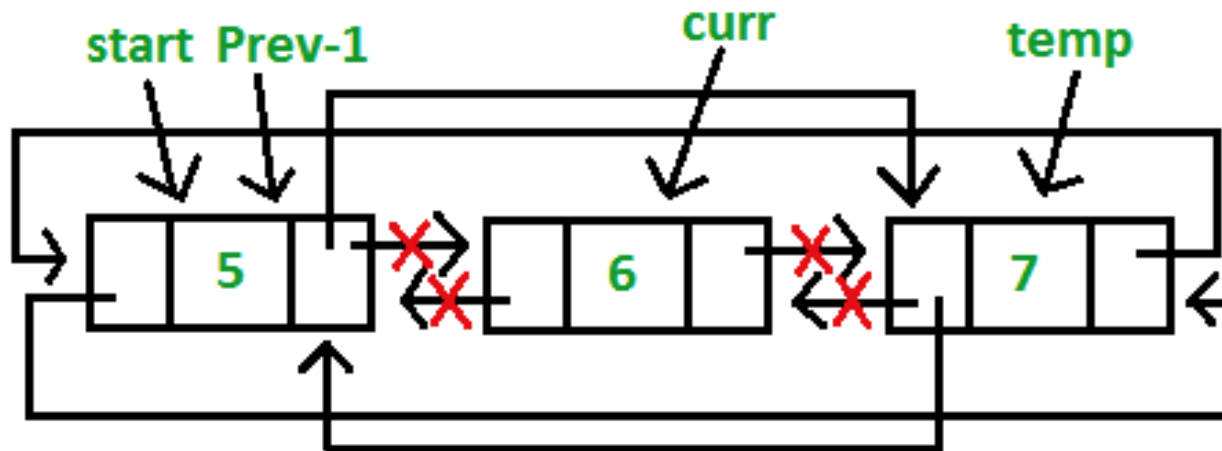
Deletion in Doubly Linked List

➤ Deletion at the end of the list



Deletion in Doubly Linked List

- Deletion in between the nodes of the list



Displaying a circular Linked List

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

