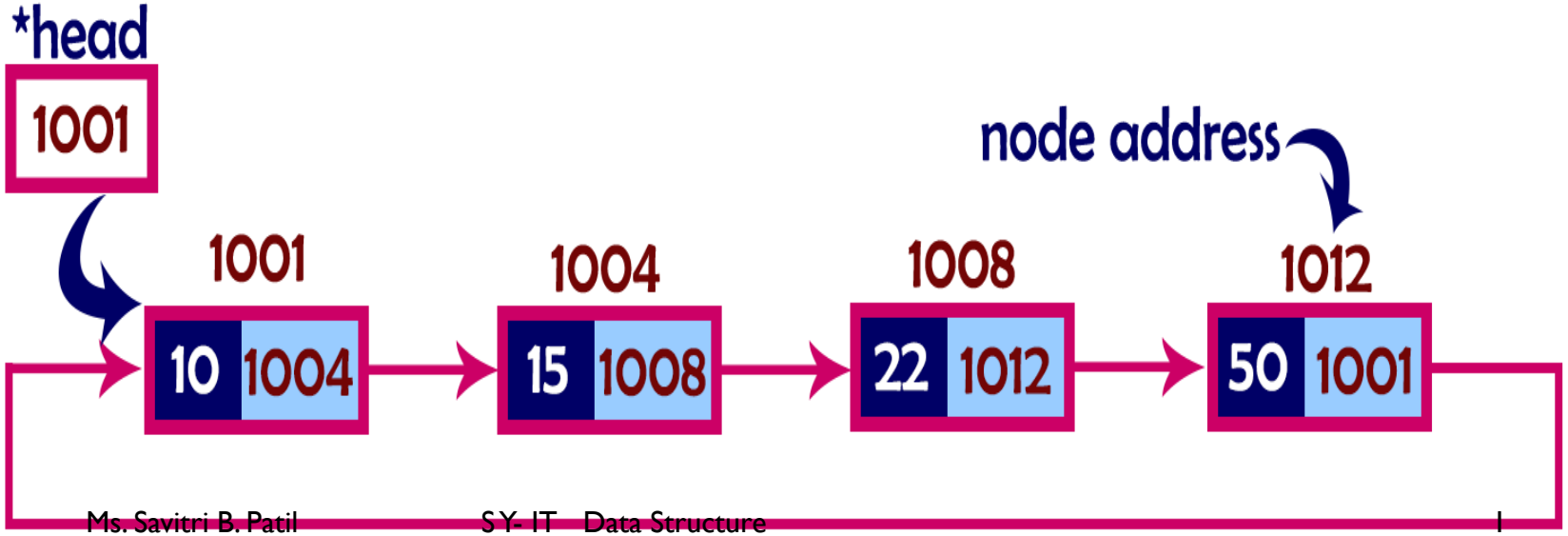
Circular Linked List

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



- Useful for playing videos and sound files in looping mode.
- Visit all node from any starting point

Drawbacks

- Code must avoid infinite loop
- No elements can be accesses randomly
- Reversing is difficult task

Operations

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

- Insertion
- Deletion
- Display

Insertion

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

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Insertion

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

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the 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** .

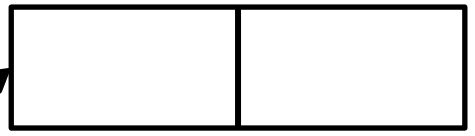
Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.

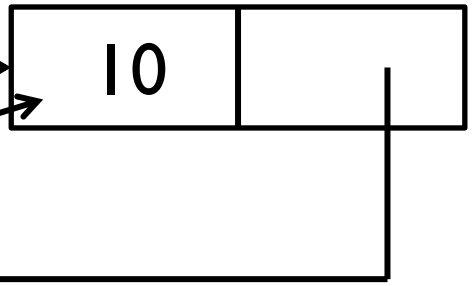
Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next == head**').

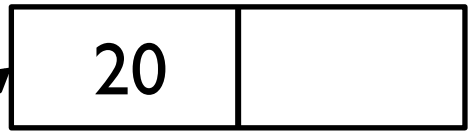
Step 6 - Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.

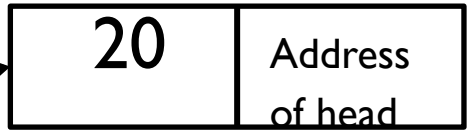
```
struct node {
    int data;
    struct node *next;
};
struct node *head = NULL;
```

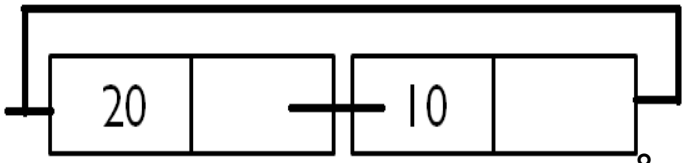
```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    if(head == NULL)
    {
       head = newNode;
       newNode -> next = head;
    }
    else    {       struct Node *temp = head;          newNode
       while(temp -> next != head)
          temp = temp -> next;
       newNode -> next = head;
       head = newNode;
       temp -> next = head;    }     printf("\nInsertio
```











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** →

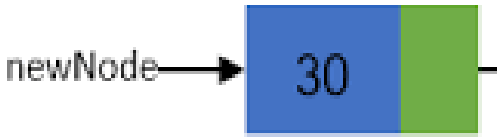
next == **head**).

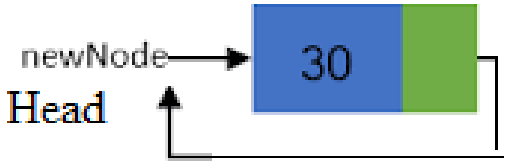
Step 6 - Set **temp** → **next** = **newNode** and **newNode**

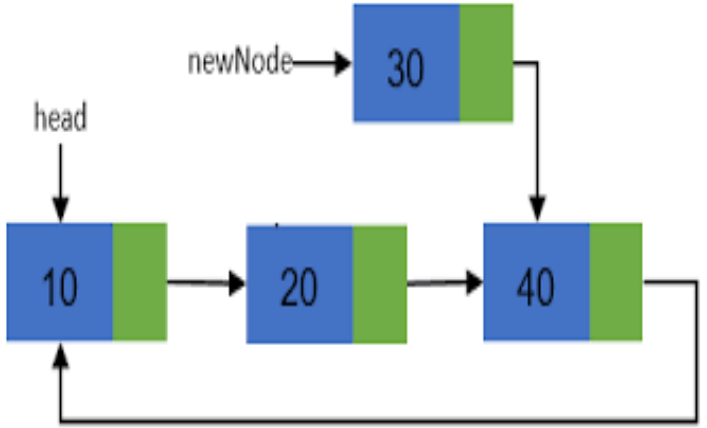
→ **next** = **head**.

```
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)   {
      head = newNode;
      newNode -> next = head;   }
   else
   {
      struct Node *temp = head;
      while(temp -> next != head)
         temp = temp -> next;
      temp -> next = newNode;
      newNode -> next = head;
head=new node;     }
```

```
void insertAtEnd(int value){
 struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)    {
      head = newNode;
      newNode -> next = head;    }
   else    {       struct Node *temp = h
      while(temp -> next != head)
         temp = temp -> next;
      temp -> next = newNode;       newNode -> next = head;
printf("\nInsertion success!!!");    }
```







Deletion

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

Deletion At Beginning of the list

Deletion At End of the list

Deletion of specific Value

Delete 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.

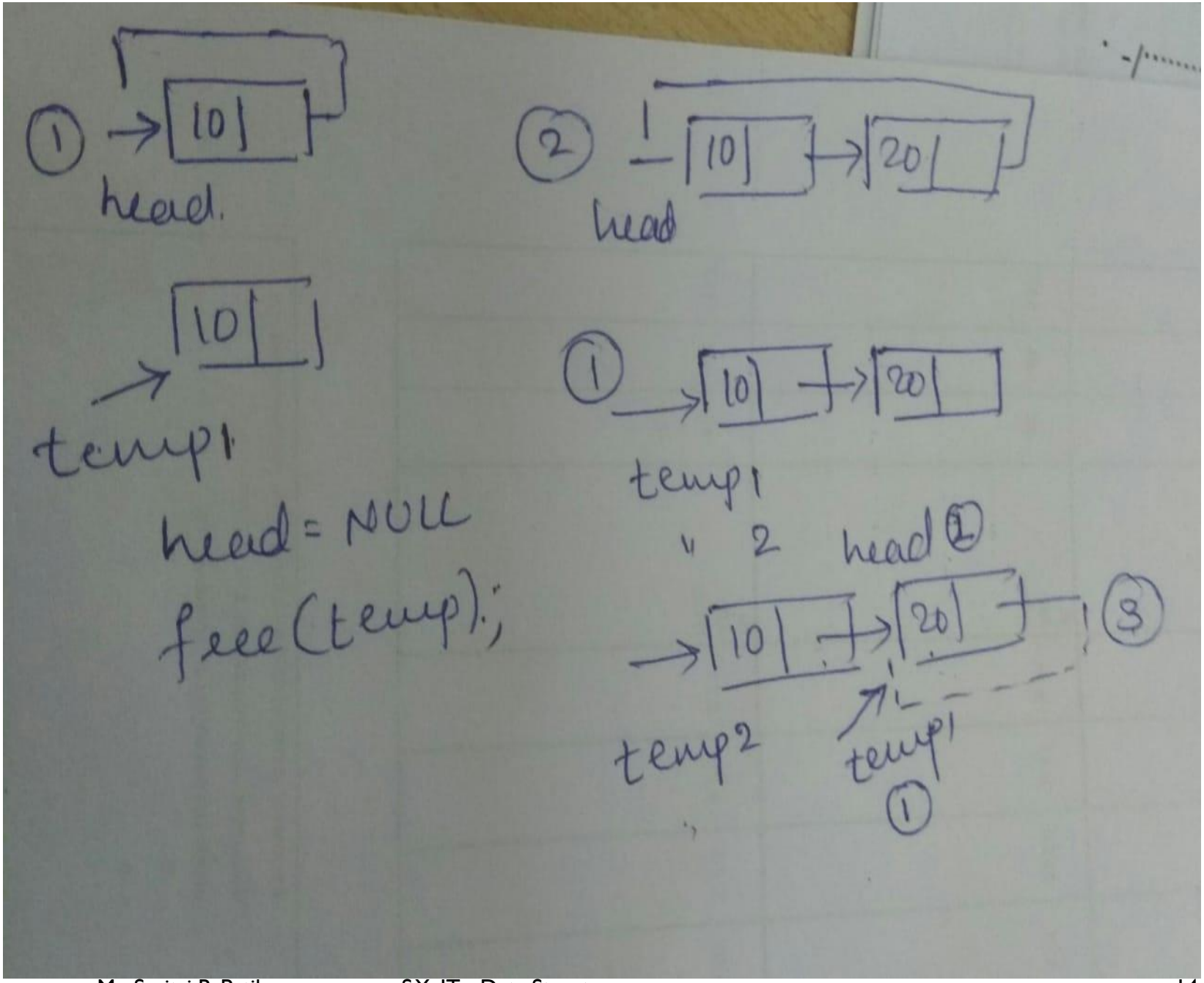
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.

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**.


1
10
head.
2
10
20
head
10
temp1
head = NULL
free(temp);
1
10
20
temp1
2
head
10
20
3
temp2
temp1
1

Deletion from End

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**)

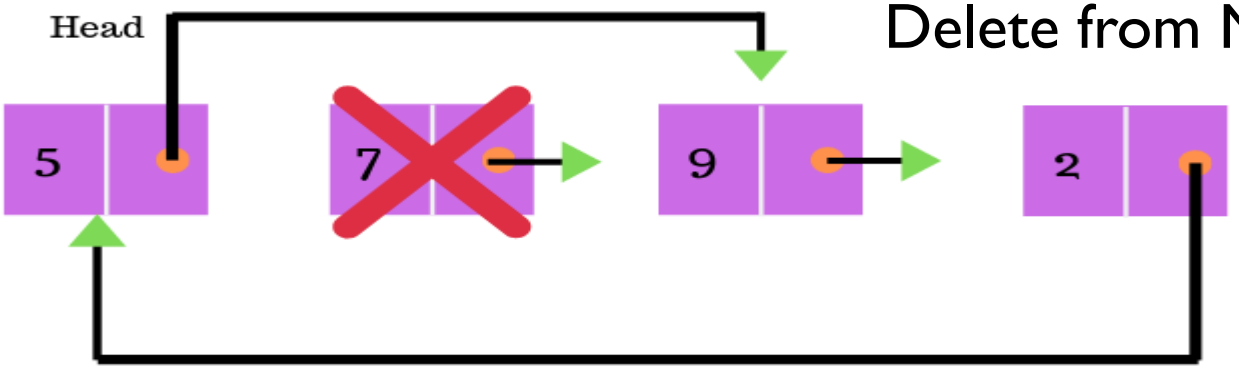
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

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**)

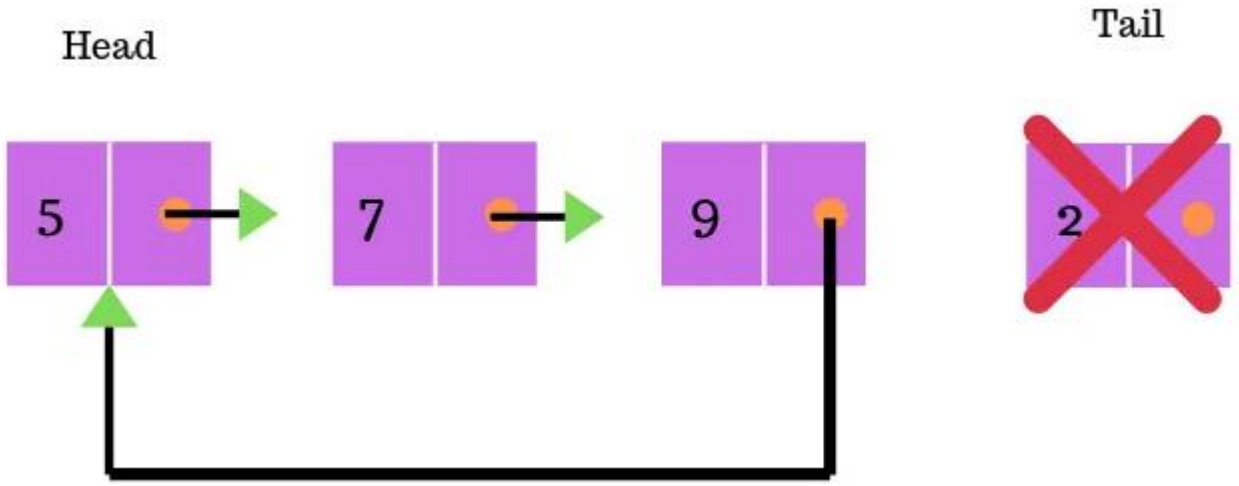
Step 7 - Set **temp2** → **next** = **head** and delete **temp1**.

```
void deleteEnd()
{
   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
      struct Node *temp1 = head, temp2;
      if(temp1 -> next == head)
      {
         head = NULL;
         free(temp1);
      }
      else{
         while(temp1 -> next != head){
            temp2 = temp1;
            temp1 = temp1 -> next;
         }
         temp2 -> next = head;
         free(temp1);
      }
      printf("\nDeletion success!!!");
   }
}
```

Delete from Middle



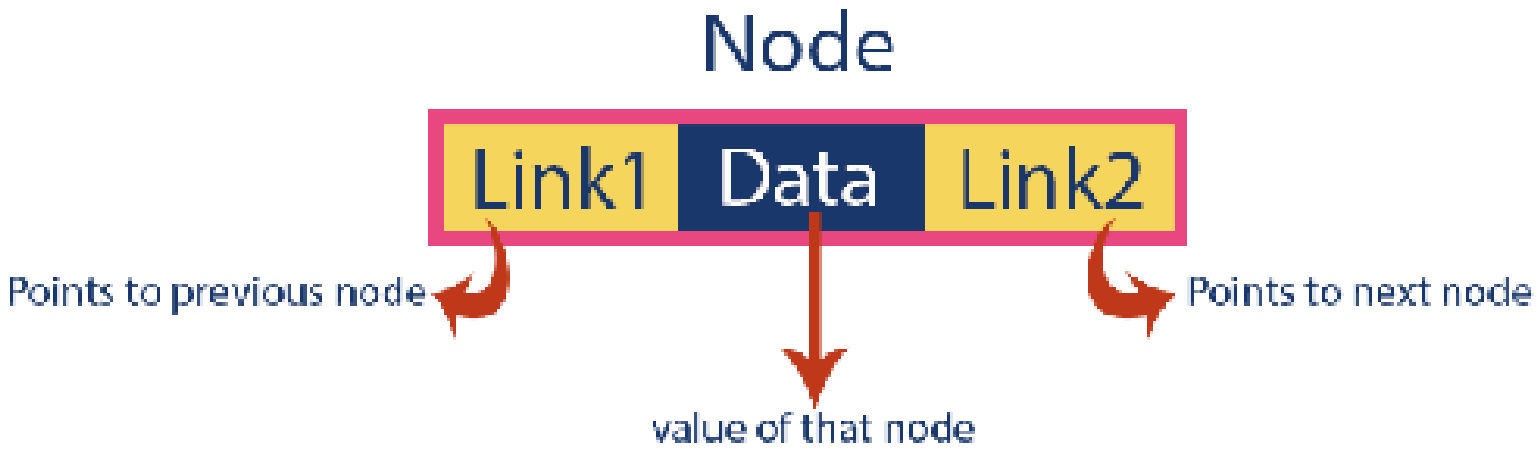
Delete from End

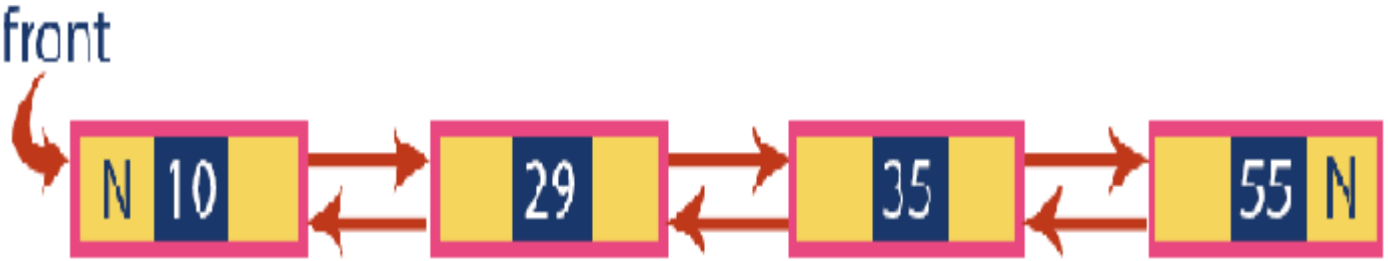


Doubly Linked List

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

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





Important Points to be Remembered

- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.

Operations on Double Linked List

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

Insertion and Deletion

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

Inserting/Deleting At Beginning of the list
Inserting /Deleting At End of the list
Inserting/Deleting At Specific location in the list

Insert at Beginning

Step 1 - Create a **newNode** with given value and **newNode** →

previous as **NULL**.

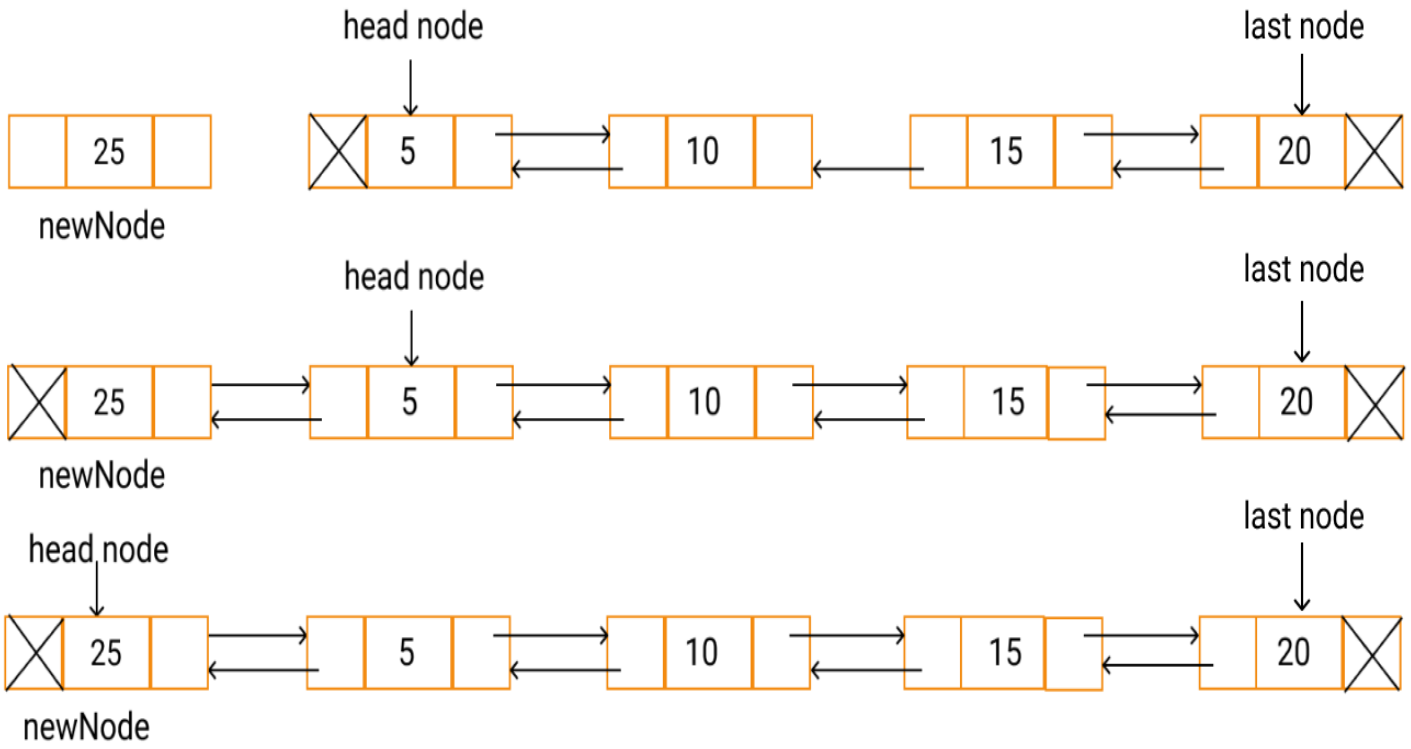
Step 2 - Check whether list is **Empty** (**head** == **NULL**)

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**.



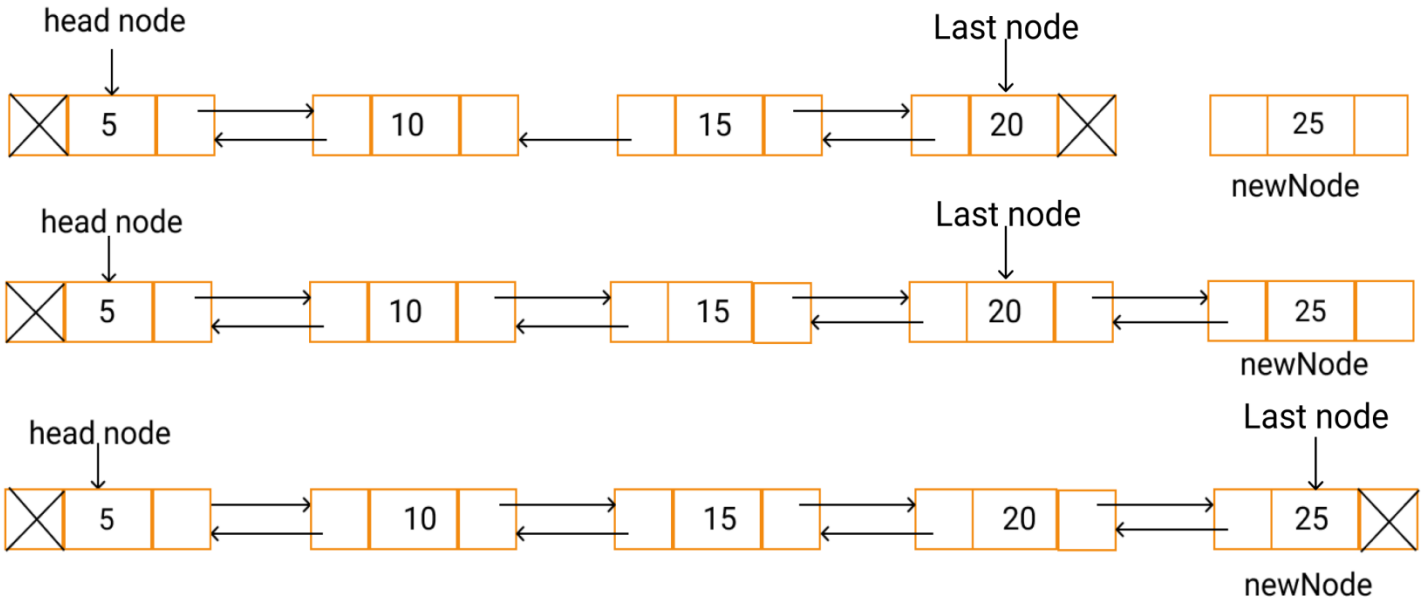
Let us assume a newNode as shown above. The newNode with data = 25 has to be inserted at the beginning of the list.

The **next** pointer of the **newNode** is referenced to the head node and its **previous** pointer is referenced to **NULL**.

The **previous** pointer of the head node is referenced to the **newNode**.

The **newNode** is then made as the head node.

```
void insertAtBeginning(int value)
{
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode -> data = value;
    newNode -> previous = NULL;
    if(head == NULL)     {
       newNode -> next = NULL;
       head = newNode;     }
    else     {
       newNode -> next = head;
       head = newNode;
    }   }
```



Now, let us assume a newNode as shown above.

The **newNode** with data = 25 has to be inserted at the end of the linked list.

Make the **next** pointer of the **last** node to point to the **newNode** .

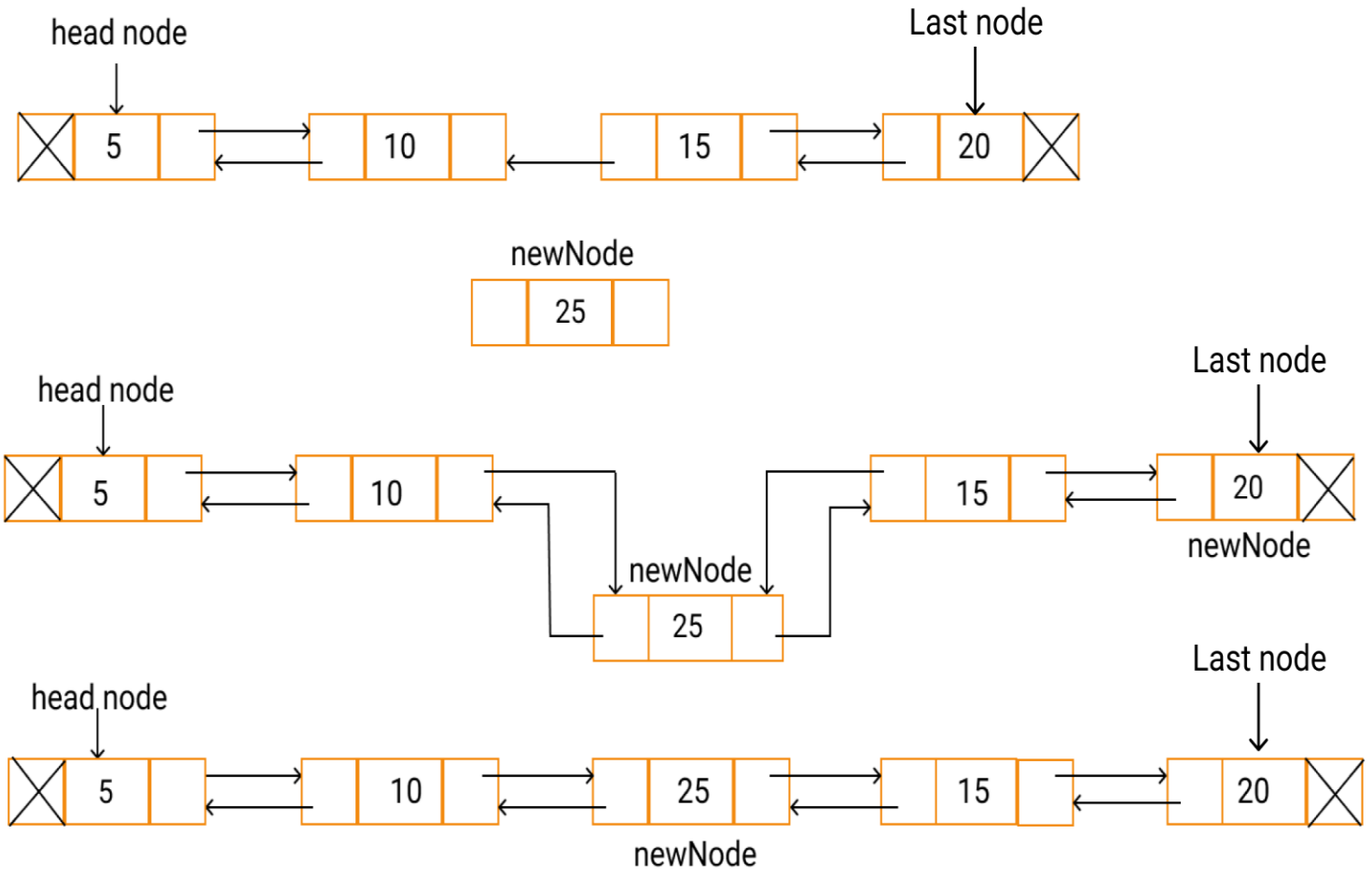
The **next** pointer of the **newNode** is referenced to NULL and its **prev** pointer is made to point to the last node.

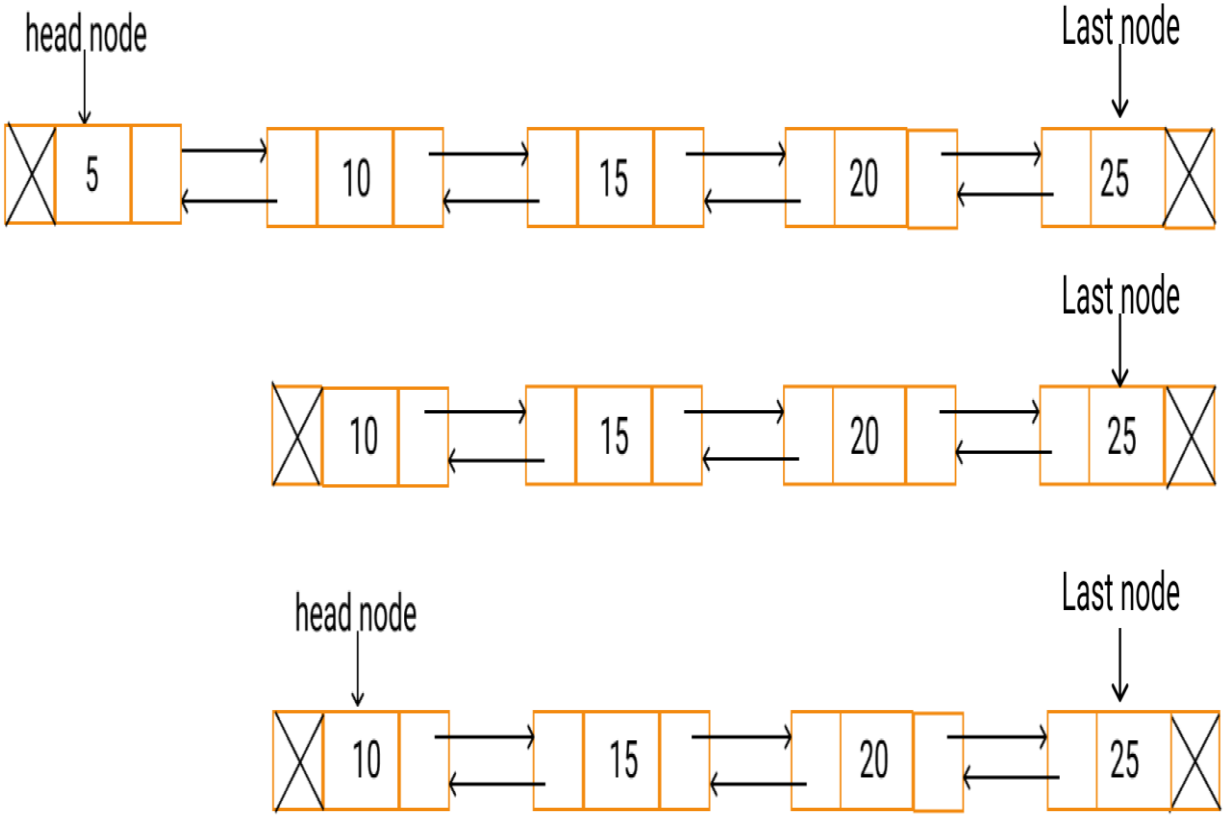
Then, the **newNode** is made as the **last node**

```
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   newNode -> next = NULL;

   if(head == NULL)
   {
      newNode -> previous = NULL;
      head = newNode;
   }
   else
   {
      struct Node *temp = head;
      while(temp -> next != NULL)
         temp = temp -> next;
      temp -> next = newNode;
      newNode -> previous = temp;
   }

}
```


head node
Last node
5
10
15
20
newNode
25
head node
Last node
5
10
15
20
newNode
newNode
25
head node
Last node
5
10
25
15
20
newNode

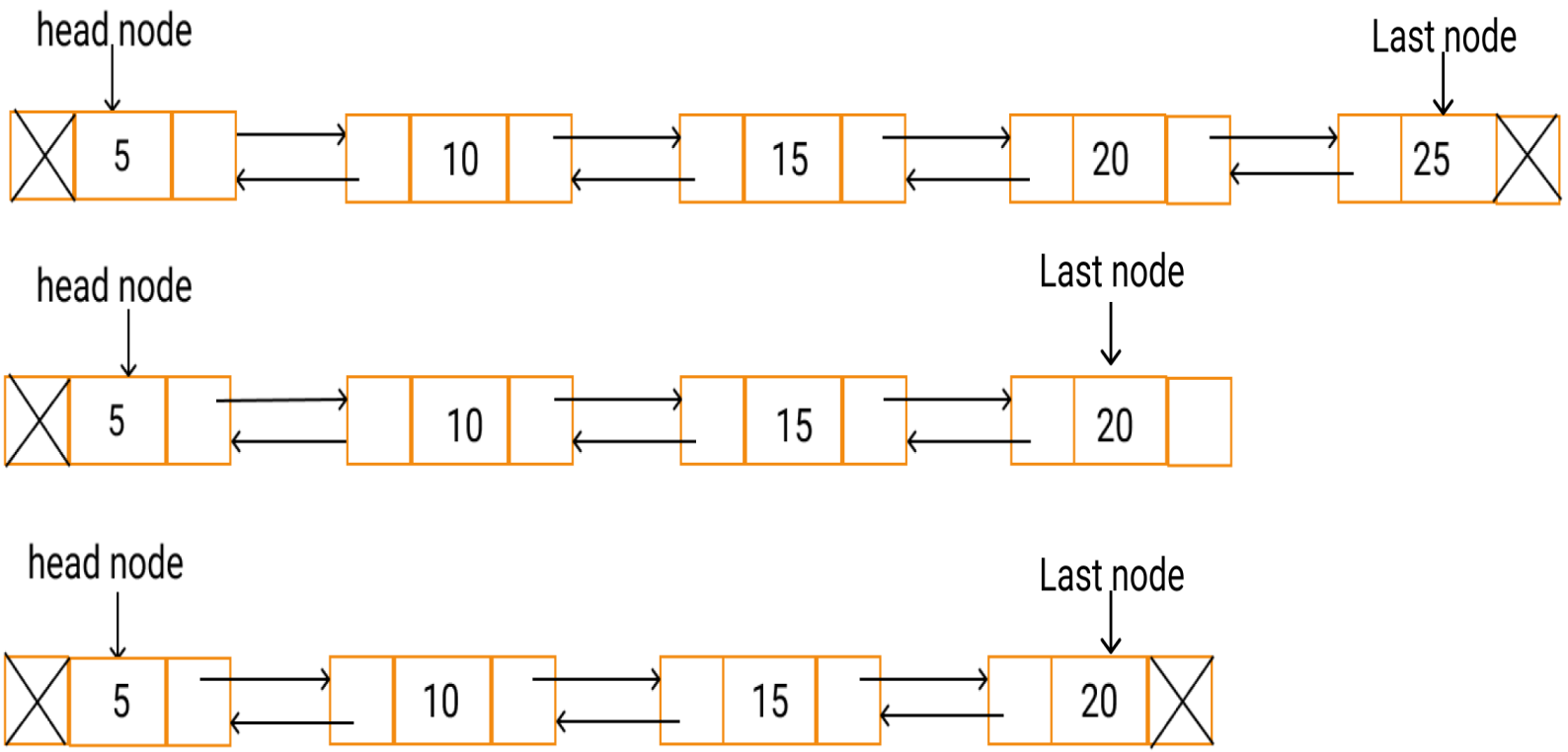


Copy the head node in some temporary node.

Make the second node as the head node.

The **prev** pointer of the head node is referenced to NULL.

Delete the temporary node.

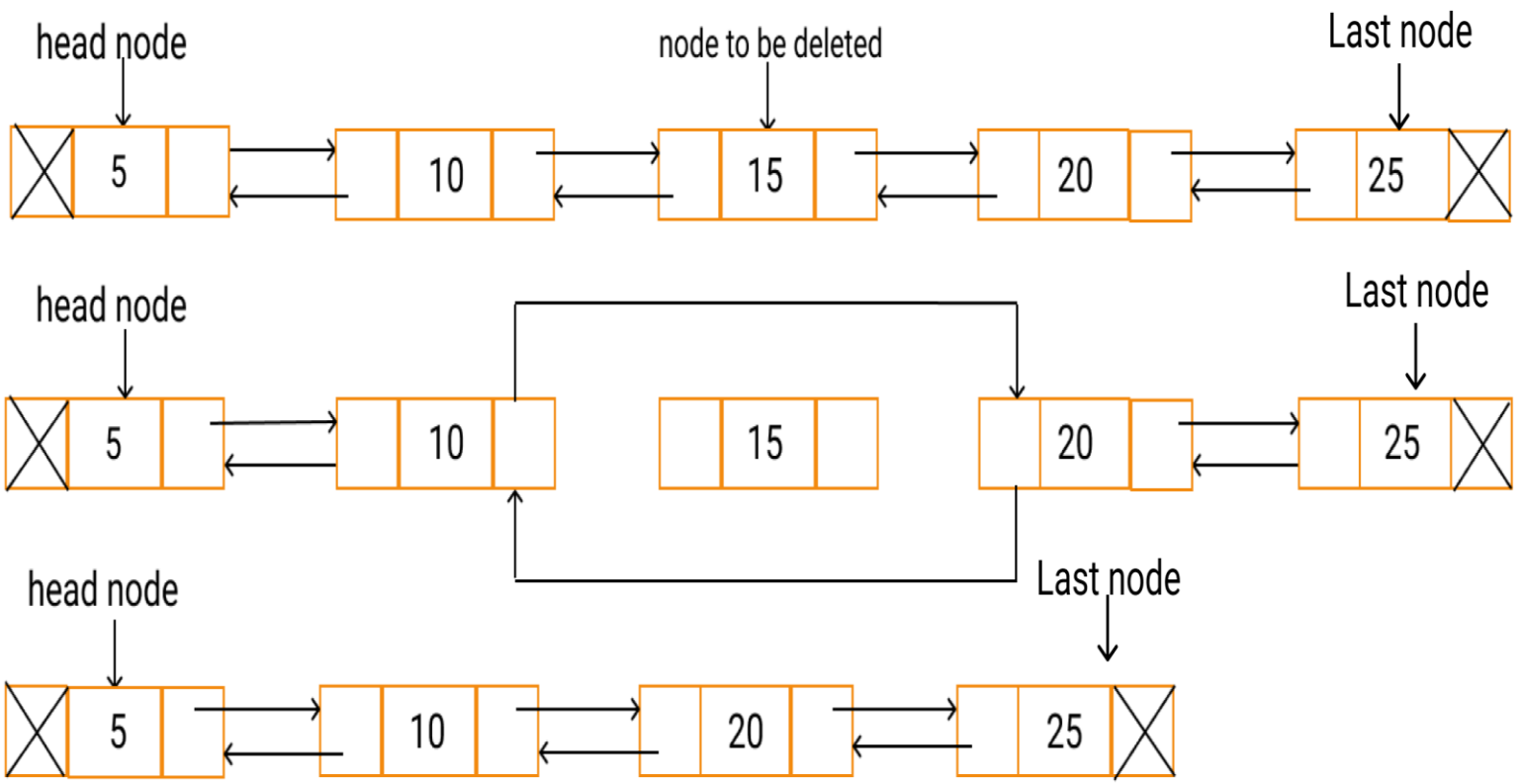


Copy the **last** node to a temporary node.

Shift the **last** node to the second last position.

Make the **last** node's **next** pointer as NULL.

Delete the temporary node.



Suppose you want to delete the third node from the list.

Start traversing the linked list from the head until the position = 2 of the node to be deleted.

Let the node at the position 2 of the list be **temp**.

```
temp->prev->next = temp->next;
temp->next->prev = temp->prev
```

Assign the **next** pointer of temp to temp's previous node's **next** pointer.

Assign the temp's **prev** pointer to temp's next node's **prev** pointer.

Delete the **temp** node.

Garbage Collection

- Garbage collection is a term used in computer programming to describe the process of finding and deleting objects which are no longer being referenced by other objects. In other words, garbage collection is the process of removing any objects which are not being used by any other objects.

- Garbage collection is the process of managing memory, automatically. It finds the unused objects (that are no longer used by the program) and delete or remove them to free up the memory. The garbage collection mechanism uses several GC algorithms

- The purpose of garbage collection is to identify and discard those objects that are no longer needed by the application, in order for the resources to be reclaimed and reused.

- Suppose you have references A -> B -> C -> D . When you delete the reference to B from A , you're left with an orphaned chain of Objects B -> C -> D .

- In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit.

- Define 10 queues each representing a bucket for each digit from 0 to 9.
- Step 2 - Consider the least significant digit of each number in the list which is to be sorted.
- Step 3 - Insert each number into their respective queue based on the least significant digit.
- Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- Step 5 - Repeat from step 3 based on the next least significant digit.
- Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

Pass - 1

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1<u>0</u>0, 1<u>5</u>0, 9<u>0</u>1, <u>8</u>2, <u>1</u>2, <u>2</u>3, <u>5</u>5 & <u>7</u>7

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

<u>1</u>00, <u>9</u>01, 12, 23, <u>1</u>50, 55, 77 & 82

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.