Linked List

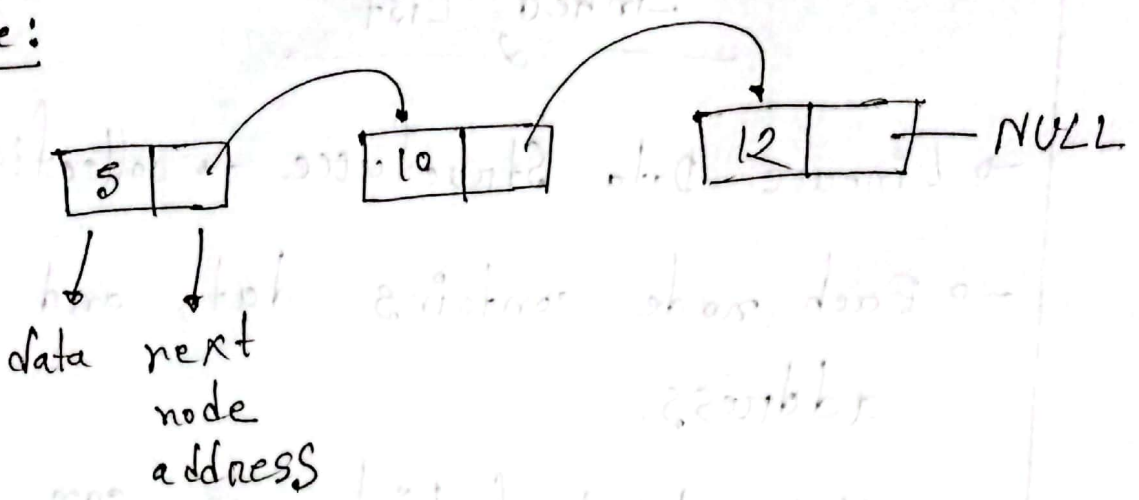
→ Linear Data Structure → collection of nodes

→ Each node contains data and next node address.

→ Using Linked List we can insert or remove any elements of any position in the linked list.

	Linked List	Array
1. Data Structure	Non-contiguous	Contiguous
2. Memory Allocation	individual elements	whole array
3. Insertion/Deletion	Efficient	Inefficient
4. Access	Sequential	Random

Example:




```
class Node {
public:
    int data;
    Node* next;
//constructor
Node (int data){
    this -> data = data;
    this -> next = NULL;
}
```

encate a Node;

```
Node* newnode = new Node(val);
```

Access:

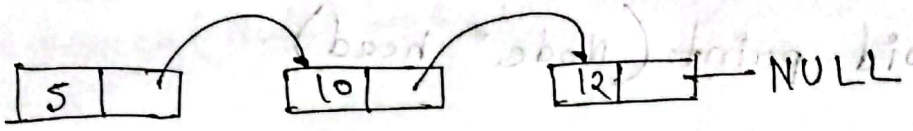
```
newnode -> data
newnode -> next
```

Types of Linked List

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

Singly Linked List

Ex:



Operations on Singly Linked List:

- Traversal
- Searching
- Length
- Insertion:
  - at the beginning (head)
  - at the end (tail)
  - at specific position.
- Deletion:
  - from the beginning (Head)
  - from the end (tail)
  - at specific node

Traversal:

```
void print (Node* head)
{
    // start from the beginning
    Node* temp = head;

    // traverse untill null pointer
    while (temp != NULL)
    {
        cout << temp -> data << " ";
        temp = temp -> next;
    }
    cout << endl;
}
```

Time Complexity : O(N)

Space Complexity : O(1)

call function:

```
print (head);
```

Searching in Singly Linked List:

```
bool search(Node* head, int target)
{
    Node* temp = head;
    while(temp != NULL)
    {
        if(temp->data == target)
        {
            return true;
        }
        temp = temp->next;
    }
    return false;
}
```

Time Complexity : O(N)

Space Complexity : O(1)

call function:

```
search(head, 5);
```

Finding Length in Singly Linked List:

```
int Length(Node* head)
{
    int cnt = 0;
    Node* temp = head;
    while (temp != NULL)
    {
        cnt++;
        temp = temp -> next;
    }
    return cnt;
}
```

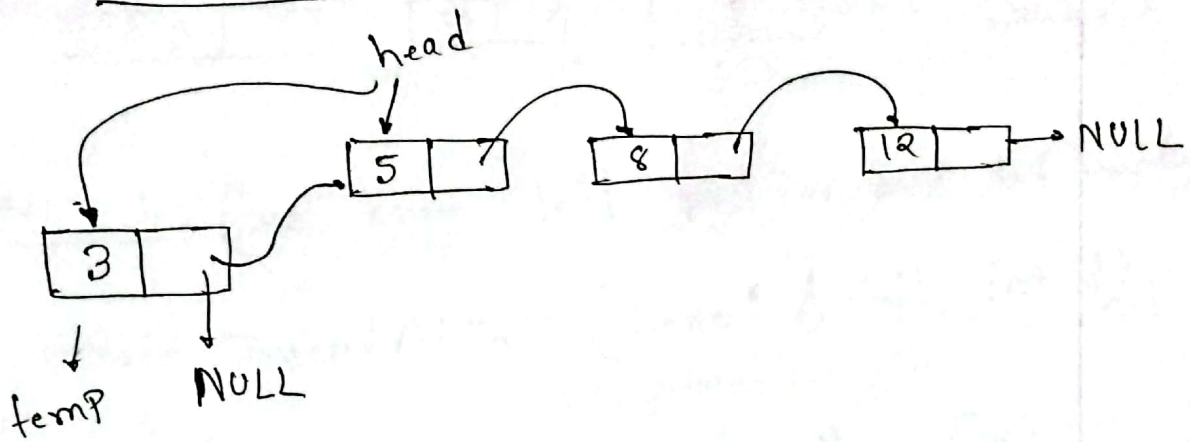
Time Complexity: O(N)
Space Complexity: O(1)

//call functions

```
Length(head);
```

Insertion in Singly Linked List:

1. Insert at the beginning:




```
void InsertAthead (Node* &head, int d)
{
    Node* temp = new Node (d);   → Create new node
    temp → next = head;
    head = temp;
}
```

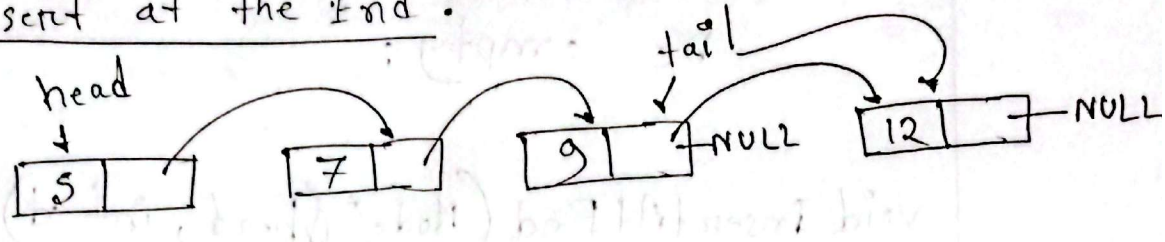
Time Complexity: O(1)

Space Complexity: O(1)

call function:

```
InsertAthead (head, 3);
```

2. Insert at the End :



Case - 1 : (we know tail position and Not empty)

```
void InsertAtTail (Node* &tail, int d)
{
    Node* temp = new Node (d);
    tail -> next = temp;
    tail = temp;
}
```

call function :

```
InsertAtTail (tail, d);
```

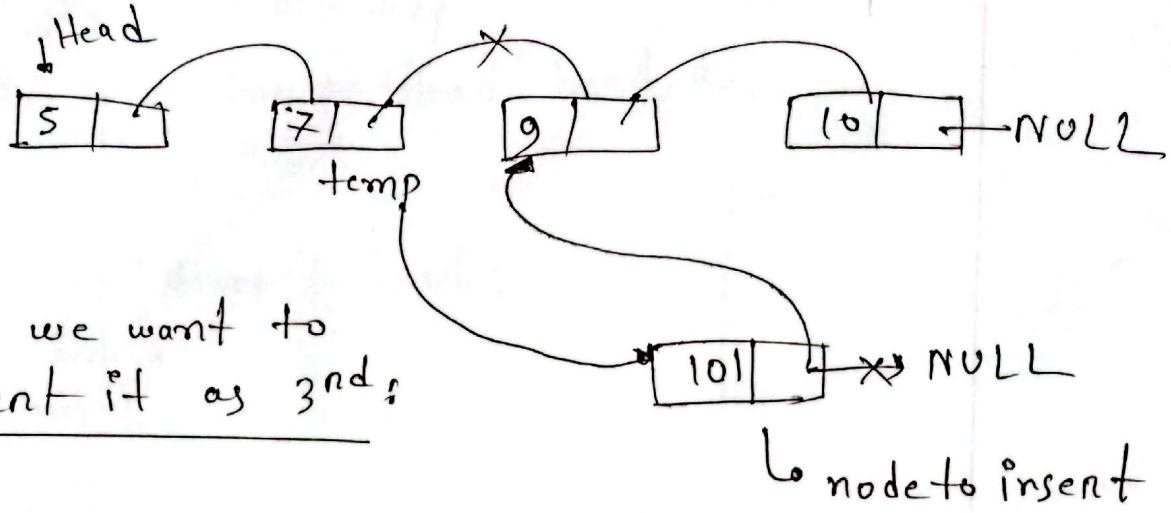
Case-2: we don't know tail and it can be empty:

```
void InsertAtEnd (Node* &head, int d)
{
    Node* temp = new Node(d);

    //empty list
    if (head == NULL)
    {
        ~~return temp;~~
        head = temp;
        return;
    }

    Node* curr = head;
    while (curr->next != NULL)
    {
        curr = curr->next;
    }
    curr->next = temp;
}
```

T.C = O(N)

S.C = O(1)

Insert at a Specific Position of the Singly Linked List:



Suppose we want to insert it as 3rd:

n → position

traverse (n-1)

```
void InsertAtPosition(Node* &head, Node* &tail, int pos, int d)
{
    if (pos == 1){
        InsertAtHead(head, d);
        return;
    }

    Node* temp = head;
    int cnt = 1
    while (cnt < pos-1)
    {
        temp = temp -> next
        cnt++;
    }
    if (temp -> next == NULL) {
        InsertAtTail(tail, d);
        return;
    }

    Node* nodetoinsert = new Node(d);
    nodetoinsert -> next = temp -> next;
    temp -> next = nodetoinsert;
}
```

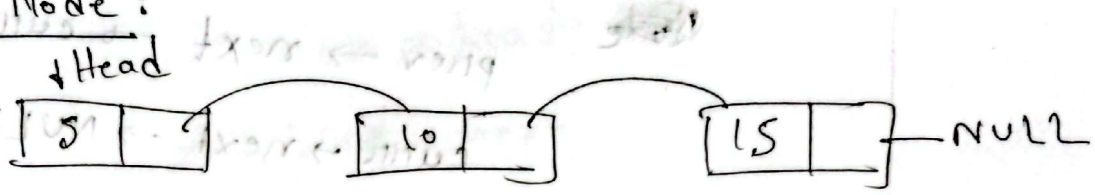
T.C = O(N)

S.C = O(1)

call function: InsertAtPosition(head, tail, 5, 7);

Deletion in Singly Linked List

First Node :

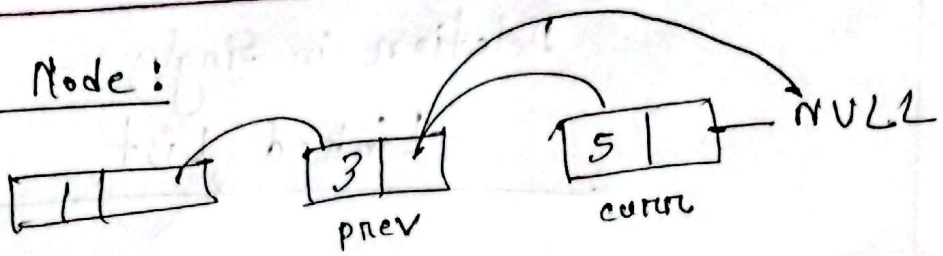



```
Node* temp = head;
head = head->next;
temp->next = NULL;
delete temp;
```

Time Complexity → O(1)

Space Complexity → O(1)

Last Node:



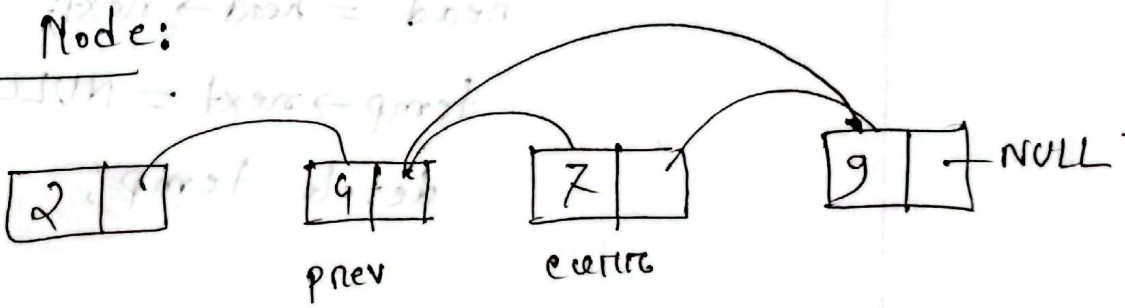
prev → next → curr → next

curr → next → NULL

delete curr;

T.C → O(N)
S.C → O(1)

Middle Node:



prev → next → curr → next

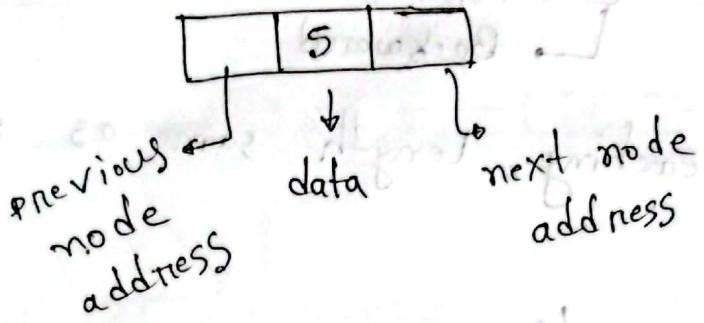
curr → next → NULL

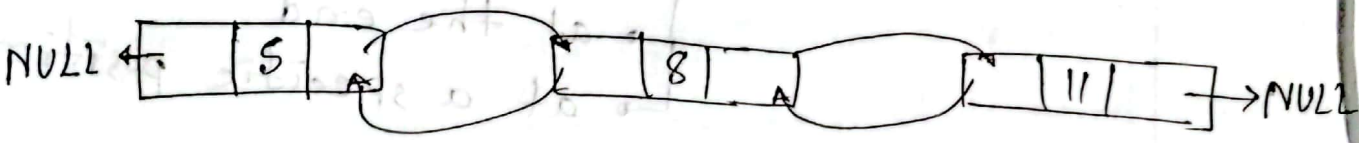
delete curr;

T.C → O(N)
S.C → O(1)

```
void DeleteNode(Node* &head, int pos)
{
    if(pos == 1) {
        Node* temp = head;
        head = head->next;
        temp->next = NULL;
        delete temp;
    }
    else {
        Node* curr = head;
        Node* prev = NULL;
        int cnt = 1;
        while(cnt < pos) {
            prev = curr;
            curr = curr->next;
            cnt++;
        }
        prev->next = curr->next;
        curr->next = NULL;
        delete curr;
    }
```

Doubly Linked List





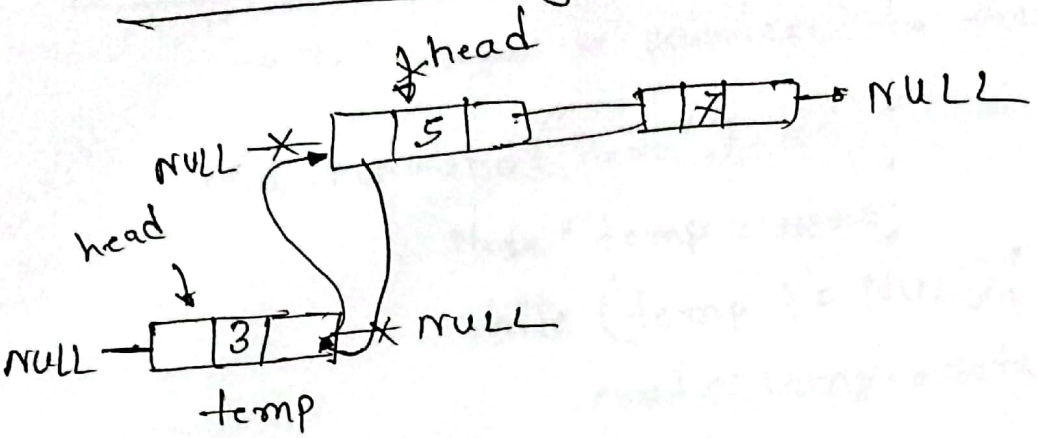

```
class Node {
    public:
    Node* prev;
     int data;
    Node* next;
  Node (int d) {
      this -> data = d;
      this -> prev = NULL;
       this -> next = NULL;
   }
}
```

Operations:

- Traversal
  - Forward
  - Backward
- Searching, length same as singly Linked List
- Insertion
  - at the beginning
  - at the end
  - at a specific position
- Deletion
  - From the beginning
  - From the end
  - at a specific position

Insertion in Doubly Linked List:

At the beginning:



→ Node * temp = new Node (d)

→ temp → next = head

→ head → prev = temp

→ head = temp

If head is NULL, then,

```
if (head != NULL) {
    head → prev = temp
}
```

Traversal in Doubly Linked List:

Forward Traversal:

→ Initialize a pointer to the head.

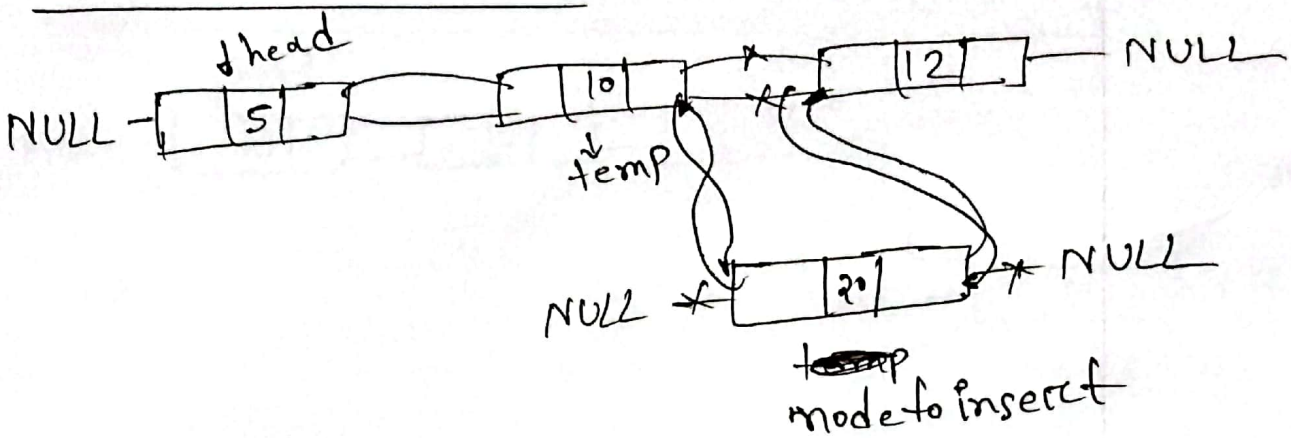
```
void Forward (Node *&head){
    Node * temp = head;
    while (temp != NULL){
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
```

Backward Traversal:

→ Initialize a pointer to the tail

```
void Backward (Node * &tail){
    Node * temp =tail;
    while(temp != NULL){
        cout << temp->data<<" ";
        temp = temp->prev;
    }
    cout<<endl;
}
```

Insert at middle:



→ Node * node to insert = new Node(d)

node to insert → next = temp → next

temp → next → prev = node to insert

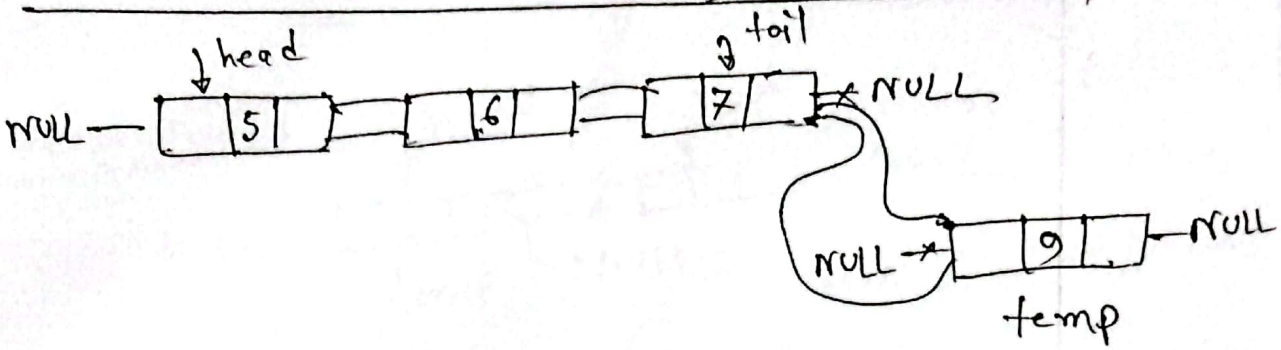
temp → next = node to insert

node insert → prev = temp

If pos = 1, Insert At head

pos = Last, Insert At tail

Insert At the End of Doubly Linked List



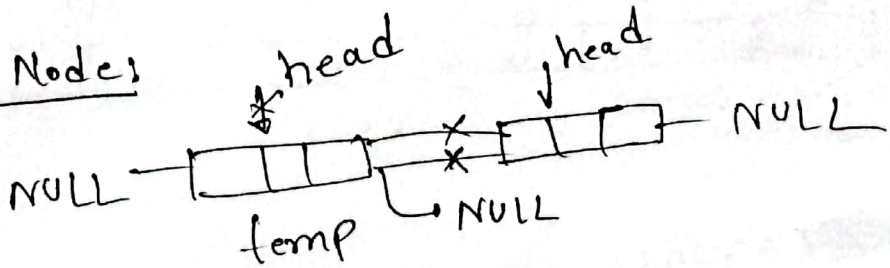
- `Node *temp = new Node(d);`
- `tail->next = temp`
- `temp->prev = tail`
- `tail = temp`

For empty list,

```
if (tail == NULL) {
        tail = temp;
        return;
}
```

Delete

First Node:



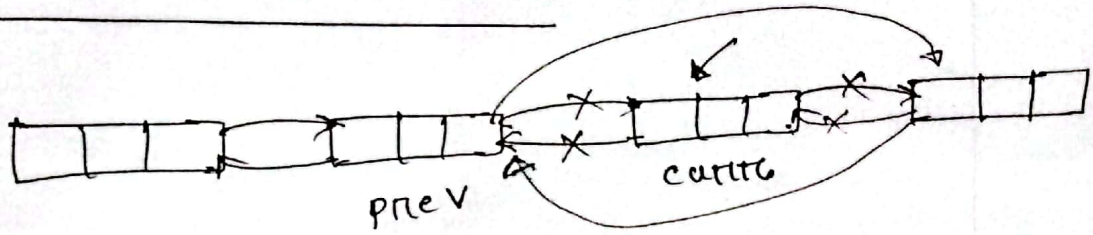
→ temp → next → prev = NULL

→ head = temp → next

→ temp → next = NULL

delete temp;

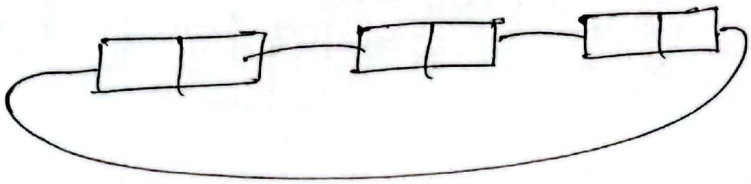
Delete the Middle or Last:



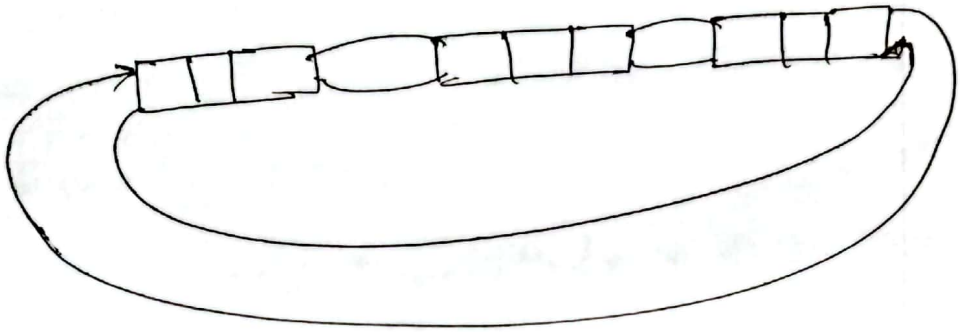

```
curr->next->prev = prev
curr->prev = NULL
prev->next = curr->next
curr->next = NULL
delete curr;
```

Circular Linked List

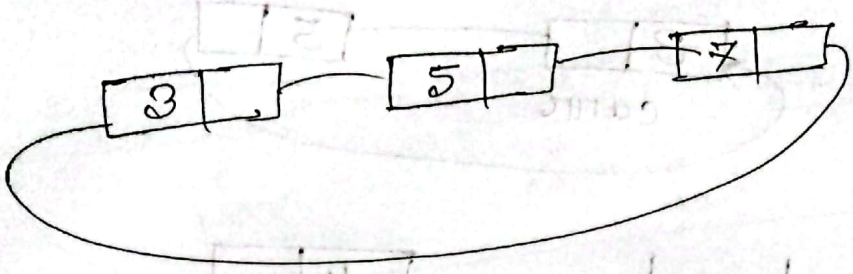
Singly :



Doubly :



Circulare Singly Linked List

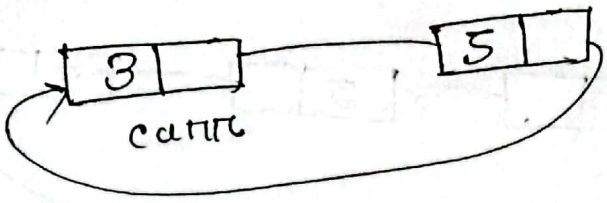


Insert :

Empty list :

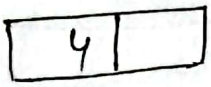
```
Node* newNode = new Node(d)
tail = newNode;
tail -> next = newNode
```

Non - empty:



i/p → element

here = 3

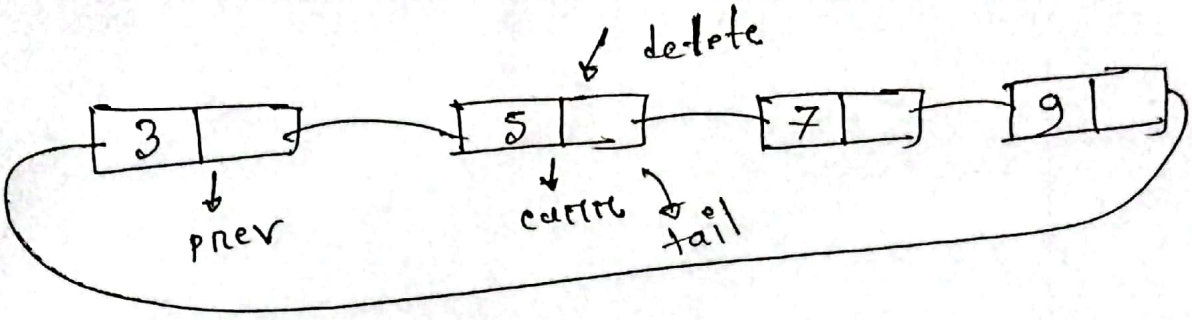


→ Node * temp = new Node(d);

→ temp → next = curr → next

→ curr → next = temp

Delete



i/p element = 5, prev->next = curr->next

1 Node

prev == curr

tail = NULL

>/ 2 Node

tail == curr

tail = prev

curr->next = NULL;

delete curr