## Linked List → Linear Data Structure < Arrays / Vectors

↓ • Complete Tutorial

Data structure (Unique)

"Nodes are connected with each other in the form of chain"

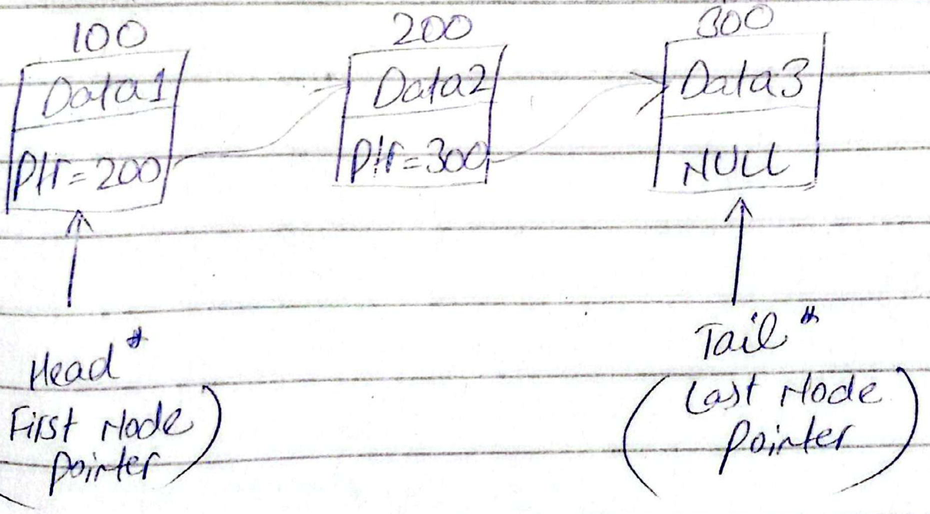
Difference

| Linked List | Arrays | Vectors |
|---|---|---|
| Not contiguous | Contigious | Contiguous |
| Dynamic | Not Dynamic | Dynamic |

Linked List

↓

Node < Data (Int, float etc) / Next Node# / Ptr (stores address of next node)

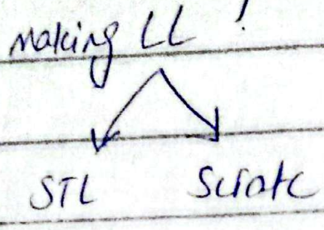


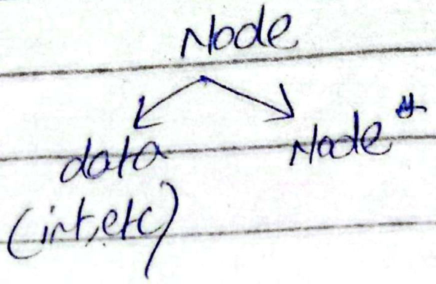
We cannot access linked list elements like Linked_List[ ], but it can be accessed using the Head pointer.

→ T.C (1) [Not possible in the Linked List]

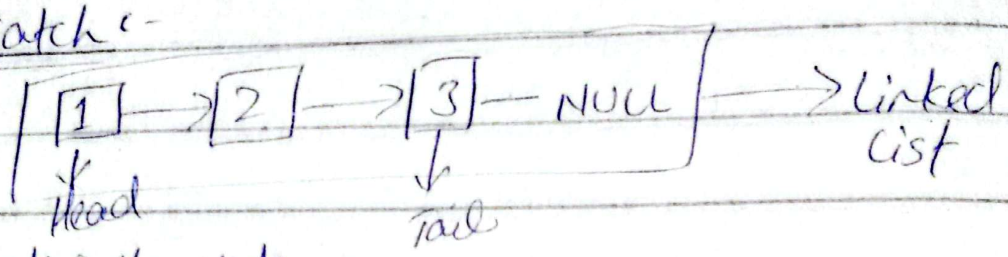
→ T.C (n) [Possible in the Linked List]

## Implementation of LinkedList





Scratch :-



Creating the Node

```
class Node {
    public:
      int data;
     Node* next;
    Node(int val){
      data = val;
      next = NULL;
          }
           };
```

This is the code for the specific node creation, so every node is the object of this class (In technical term).

Here we need another class to cover all these objects Nods which is Linked List.

Creating the List(Linked)

```
class List {
   Node* Head;
   Node* Tail;
     public
     List(){
   Head, Tail = NULL;
          }};
```

Now, this class List will cover all the objects/nodes collectivly.

## Functions of Linked List

① push-front (Adds the Node on the front)

② push-back (Adds the Node on the back)

③ pop-front ( Removes the Node on the front)

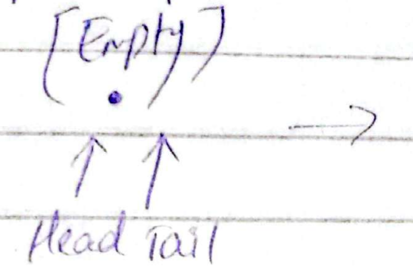
④ pop-back (Removes the Node on the back)

## Explaination:

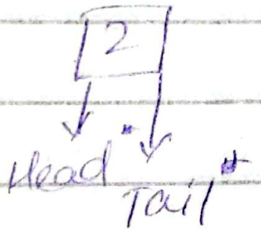
→ push-front function — two scenarios

head

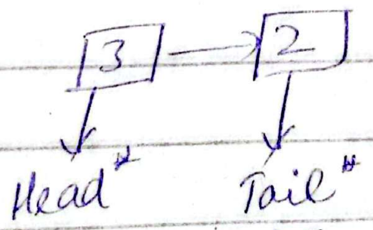
| NULL | Not NULL |
|---|---|
| ① Create new Node | Creat new Node ① |
| ② Head, Tail → New Node | newNode → next = head ② |
| | head = newnode ③ |



pushfront(2) →



pushfront(3) →

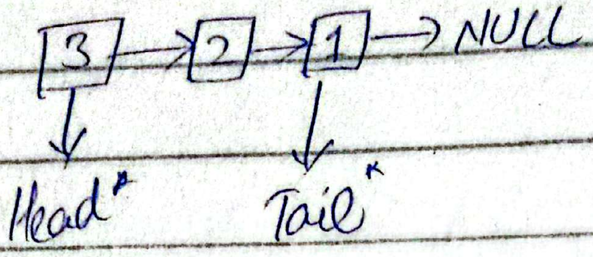


```
void push-front(int val){
  Node* newNode = new Node(val);
    if(head==NULL){
      head=tail=newNode;
        return;
  }
  else{
    newNode->next =head;
       head = newNode;
  }
}
```

→ this make the code dynamic, val will also be available in the main funct as new key-word is used

T.C=O(1)

## → Print a Linked List

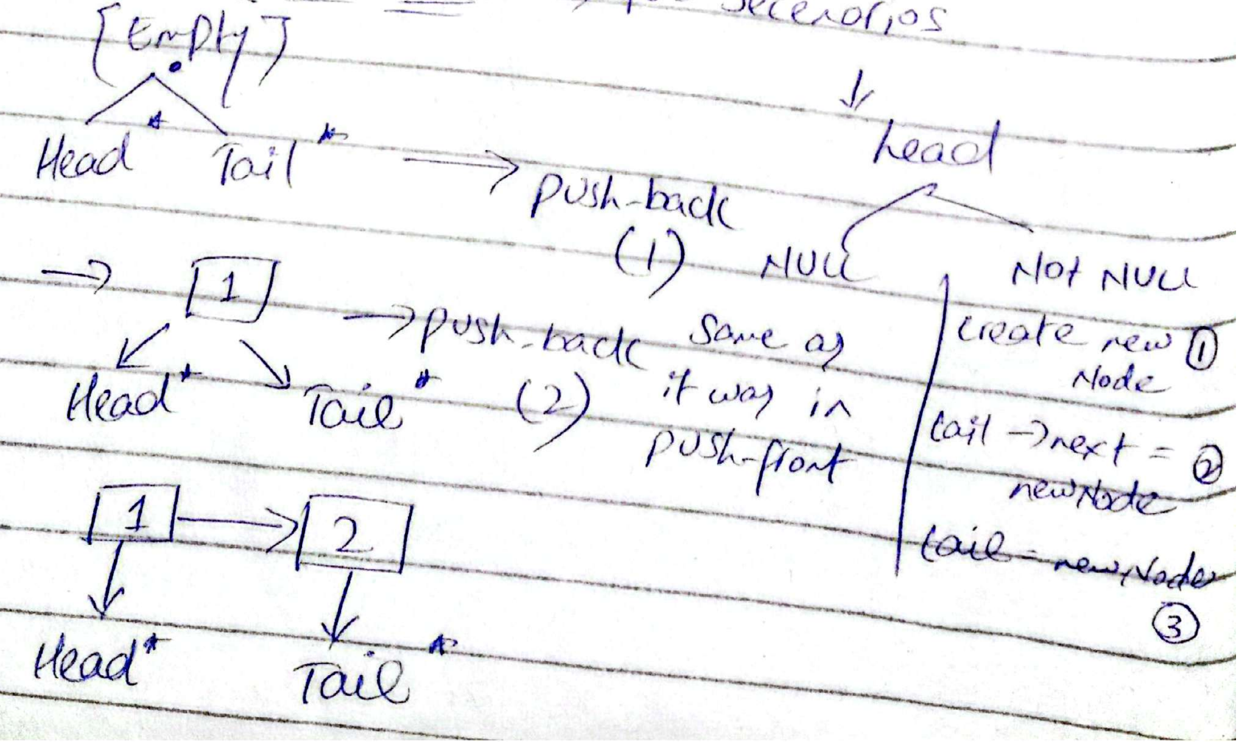


we will use the temp variable (cannot use the Head* for this role as we have to preserve it for its specific role (handles the first node) because a temp variable moves it cannot come back (reverse) as the LL moves sequentially.

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

T.C = O(n)

## Push_back in LL → two secenorios

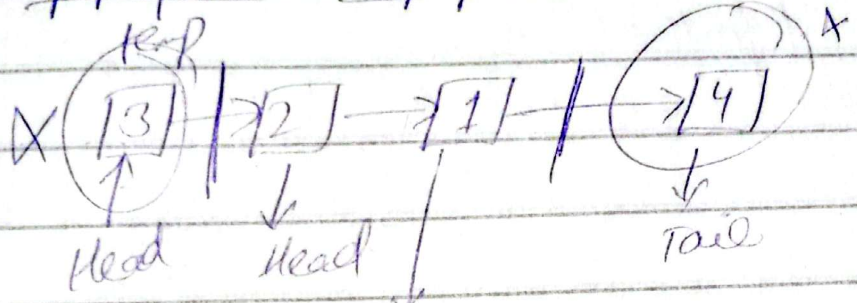


NULL: Same as it was in push_front

Not NULL:
1. create new Node
2. tail->next = newNode
3. tail = newNode

```
void push_back(int val){
    Node* newNode = new Node(val);
    if(head==NULL){
        head = tail = newNode; }
    else{
        tail->next = newNode;      T.C=O(1)
        tail= newNode;
    }}
```

→ pop_front & pop_back



pop_front

```
if(head == NULL){
    return }
Node* temp = head
  head = head->next
    temp->next = NULL
    delete temp

TC = O(1)
```

pop_back

(temp->next == tail)

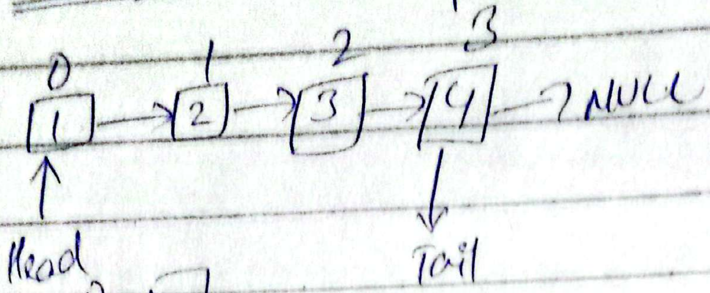
(temp->next->next == NULL)

→ If tail is given — finding previous Node

→ If tail is not given

```
if(head == NULL){
        return;
Node* temp = ~~NULL~~head
while(temp->next!=tail){
temp = temp->next }
  temp->next = NULL
    delete tail;
    tail = temp;     TC = O(n)
```

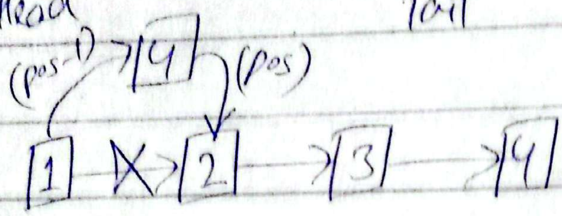
# Insert In Middle of LL



insert(val, pos)

(1) Create a Node

insert(4,1)



Case 01

```
if(pos<0)
    return
```

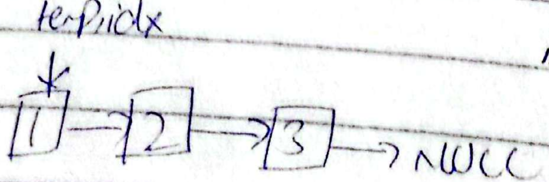
Case 02

```
if(pos==0)
push-front(val)
```

Case 03 TC = O(n)

```
Create a NewNode
Node* temp = head
for(i=0; i<pos-1; i++){
    temp = temp->next
newNode->next = temp->next
temp->next = newNode
→ Extracheck
  if(temp==NULL){
    cout<<"Invalid pos"
    return
  }
```

## Search in Linked List



```
int Search(key)
Node* temp = head
int idx = 0
while(temp!=NULL){
if(temp->data == key){
    return idx;
}
temp = temp->next
idx++
}
return -
```

TC = O(n)