Section-11

Linked Lists

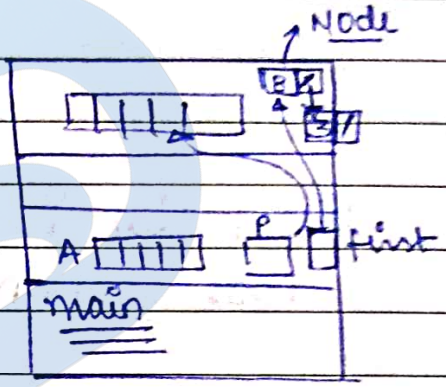
Reversed linked list:

Why we need Dynamic Data structure Linked list

1) Problem with Arrays

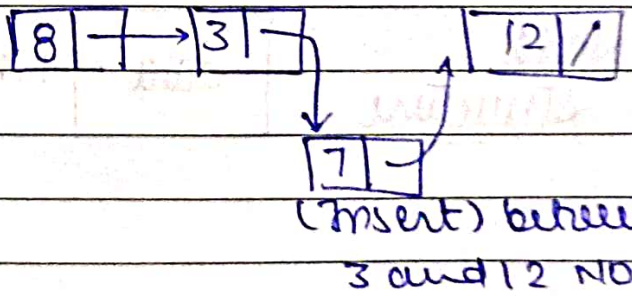
2) Difference between Array & Linked list

int A[10]; → Not available to store more 10 element in this array.



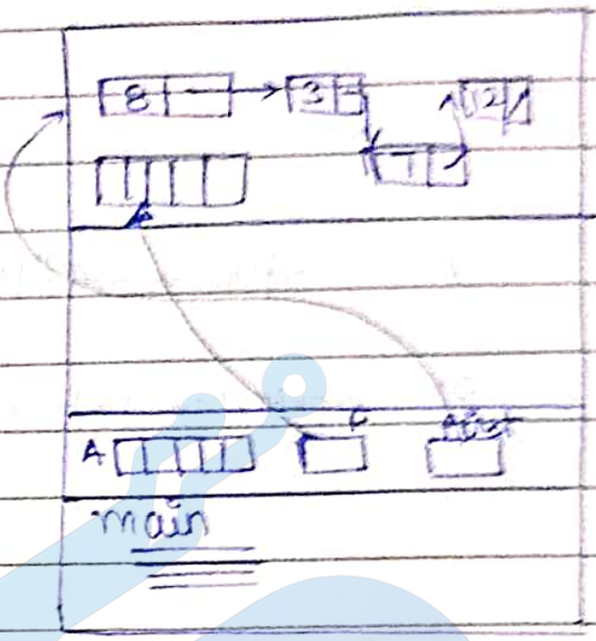

```
int A[5];
int *p=new int[5];
```

Together, element space and pointer space together, we call it as a Node.

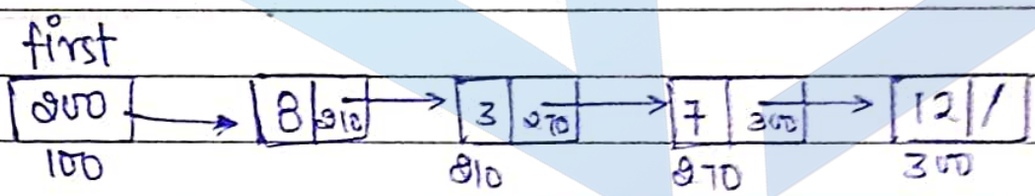


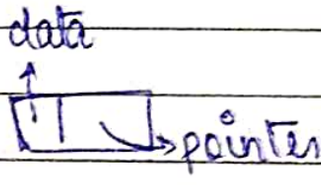
Linked List

- What is Linked list?
- What is Node?
- Node structure
- Create a Node
- Access Node

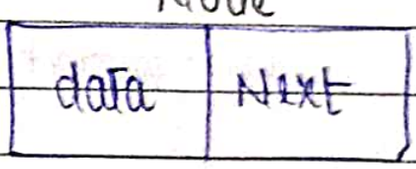


Linked list is a collection of nodes where each node contain data and pointer to next Node.





Node structure



Use of linked list:-

- Stacks and queue
- Image Gallery
- Music player
- Browsers
- Hashmap and Hash set.

class → by default → private

struct → by default → public

C Program / C++ Program

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

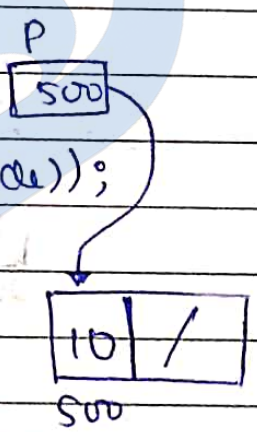
2 — int data;

2 — struct Node *next;

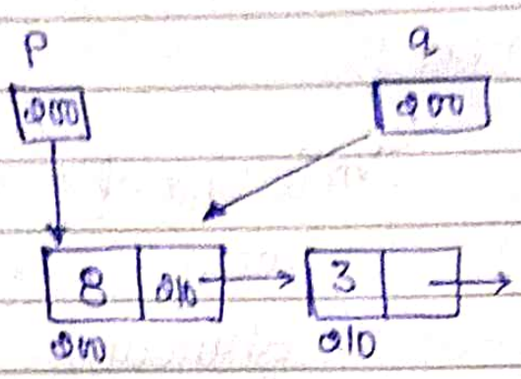
This type of structure is called as self-referential structure

4 bytes → size of Nodes

```
struct Node *p;
p = (struct Node*) malloc(sizeof(struct Node));
p = new Node;
p->data = 10;
p->next = 0;
```

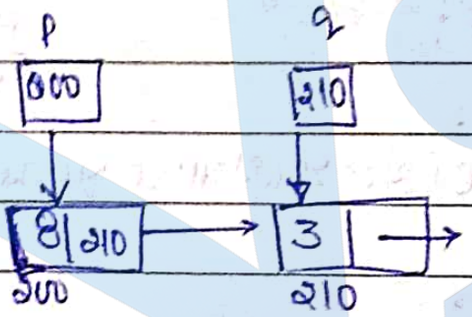


More about linked lists →

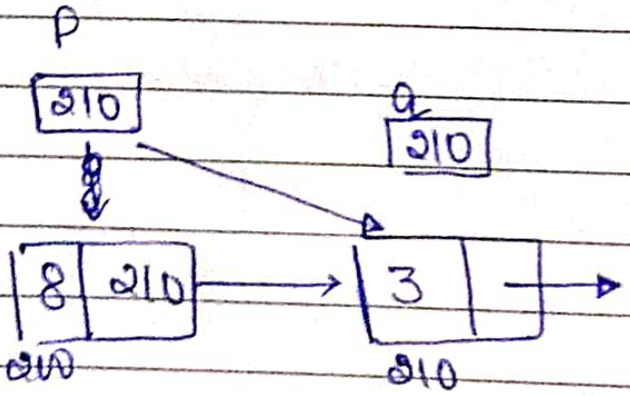


q = p

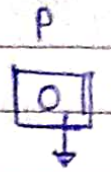
q = p → next



p = p → next

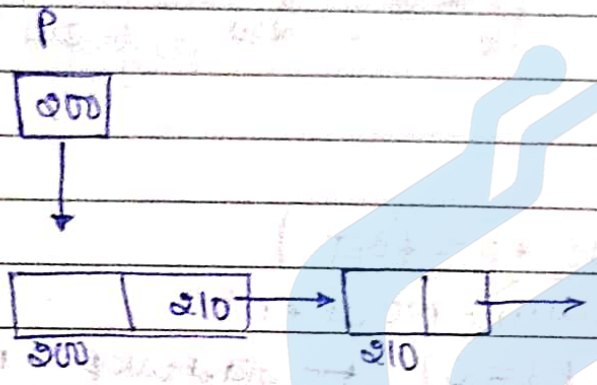


1.



→ struct Node *p=NULL;

2.



if (p == NULL) (condition 1)

if (p == 0) } condition 1)

if (!p) } condition 1)

if (p != NULL) (condition 2)

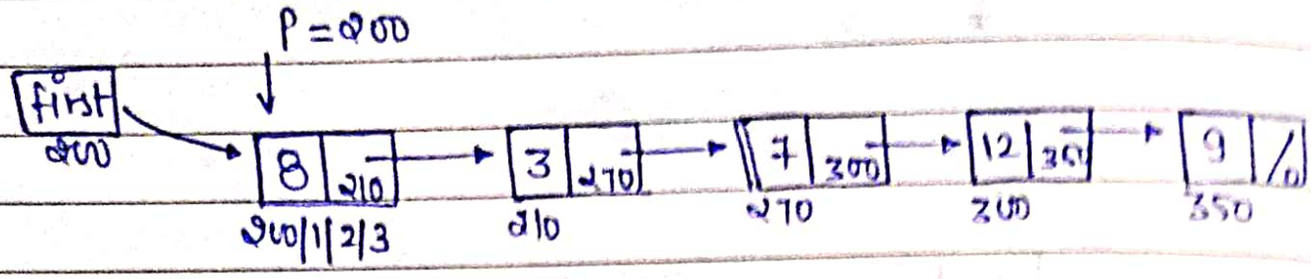
if (p != 0) (condition 2)

if (p) (condition 2)

if (p → next == NULL) } Means last Node of linked list }

if (p → next != NULL) } Means some node left in linked list }

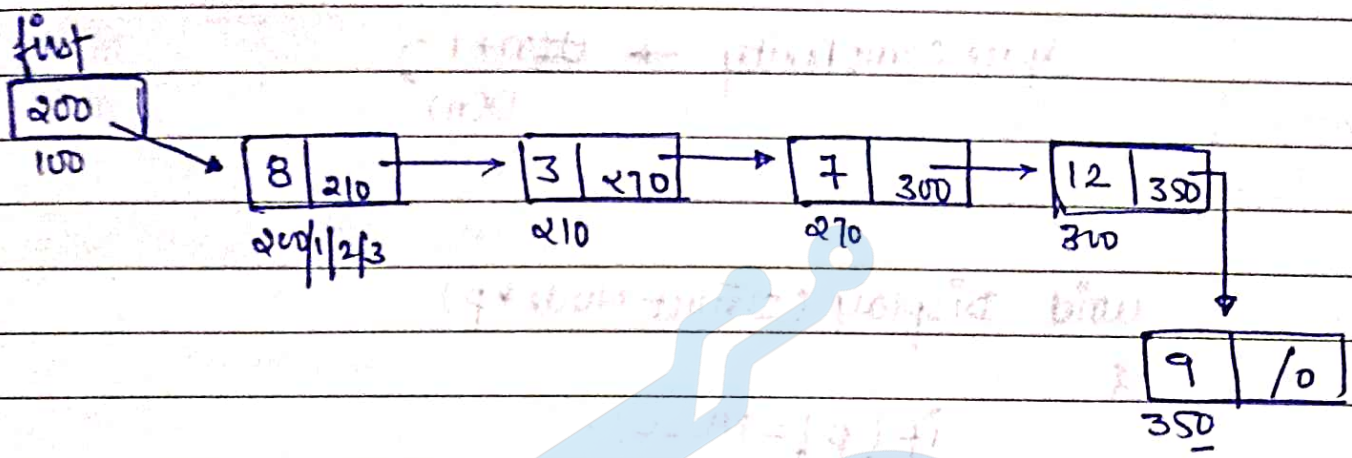
Display Linked List :-



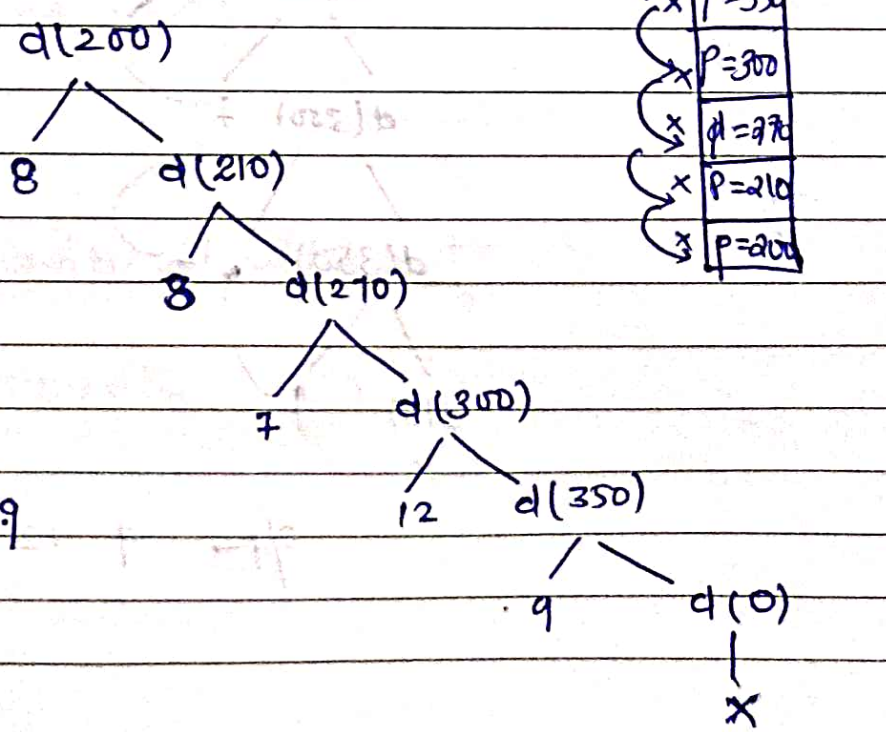

```
// struct Node *p = first;
display (struct Node *p) {
  while (p != 0)   ← जब तक p NULL नहीं हो जाता
  {
      cout<<p->data;
      p=p->next;

  }

}
```

```
display(first);
```

Recursive Display of linked list :-




```
void Display(struct Node *p)
{
    if (p!=NULL)
    {
        cout<< p->data;
        Display(p->next);

    }

}
```



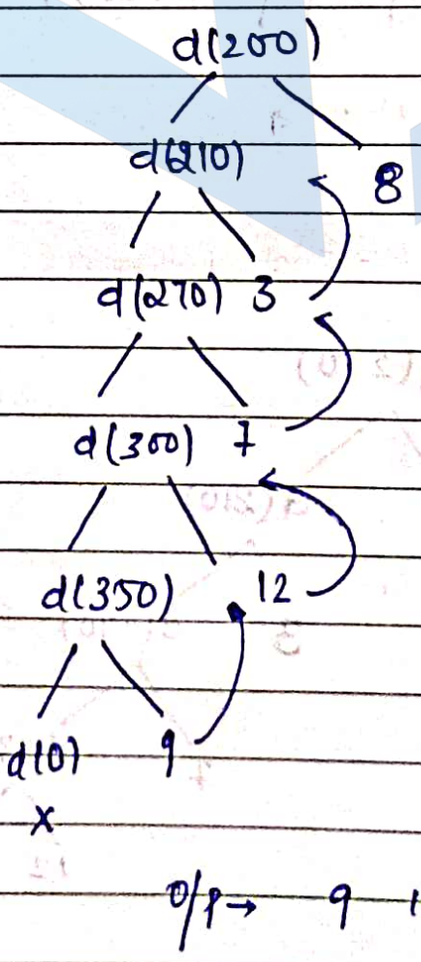
Q.12

O/P → 8 3 7 12 9

Time Complexity → O(n)

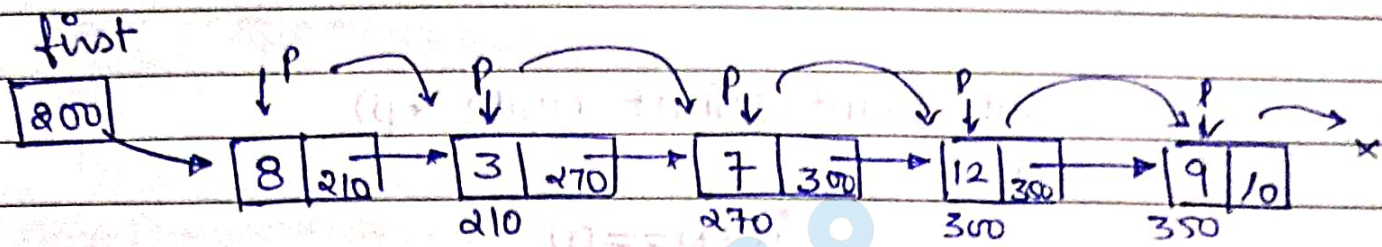
Space complexity → ~~O(n+1)~~
O(n)

```
void Display (struct Node *p)
{
    if(p!=NULL)
    {
        Display(p->next);
        cout<<p->data;
    }
}
```



o/p → 9 12 7 3 8

Counting Nodes in a Linked List →



count = 0;

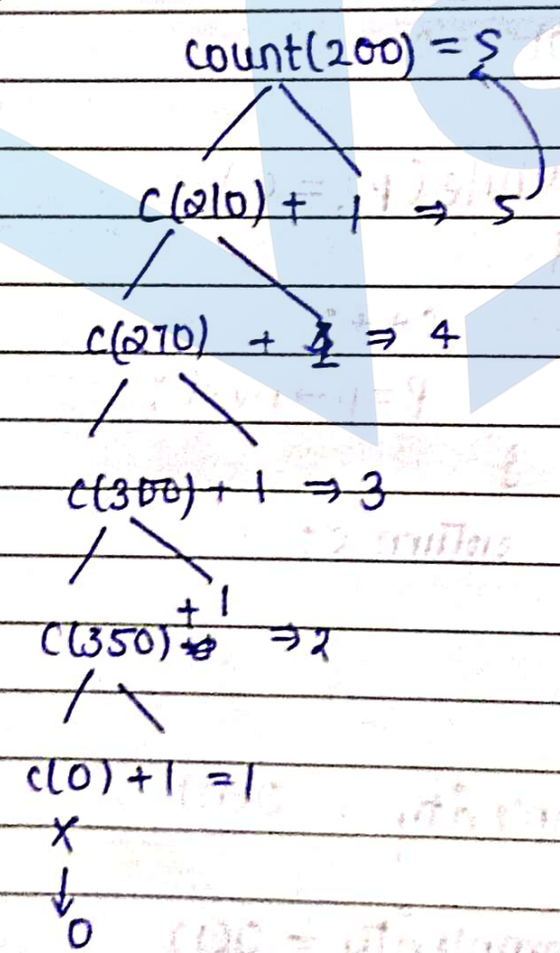
```
int count (struct Node *p)
{
    int c=0;

    while(p!=0)
    {
        c++;
        p=p->next;
    }
    return c;
}
```

Time complexity → O(n)

Space complexity → O(1)

Recursive function for count

```
int count(struct Node *p)
{
    if(p==0)
        return 0;
    else
        return count(p->next)+1;
}
```



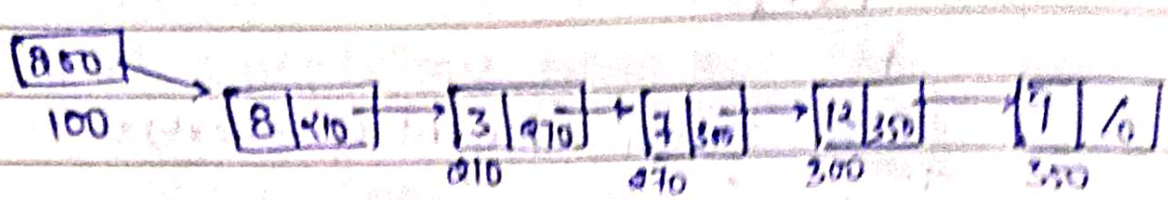
Time → O(n)

Space → O(n)

```
int count (struct Node *p)
{
    if (p != NULL)
    {
        return 1+count (p->next);
    }
    else
    {
        return 0;
    }
}
```

```
int count (struct Node *p)
{
    int x=0;
    if (p)
    {
        x = count (p->next);
        return x+1;
    }
    else
    {
        return 0;
    }
}
```

Sum of All Element in a linked list:-




```
int Add(struct Node *p)
{
    int sum=0;
    while(p)
    {
        sum=sum+p->data;
        p=p->next;
    }
    return sum;
}
```

Time complexity → O(n)

Space → O(1)

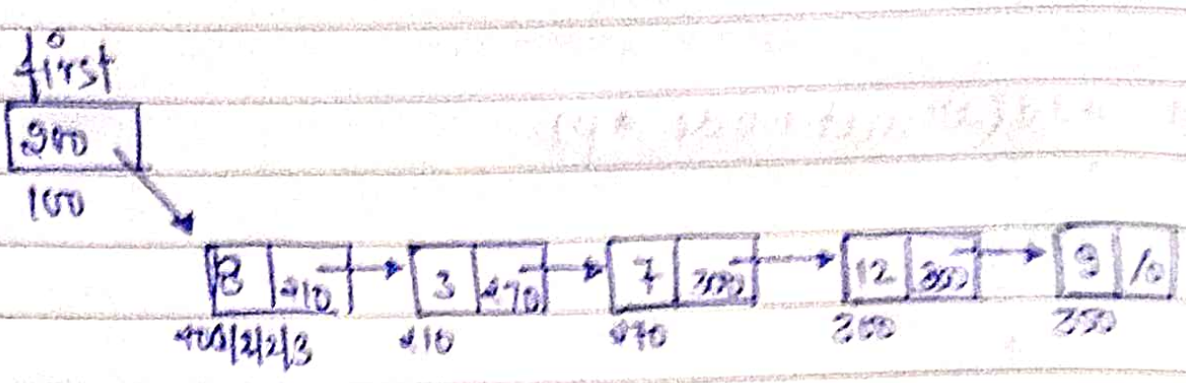
Recursive function

```
int Add(struct Node *p)
{
    if(p==0)
    {
        return 0;
    }
    else
    {
        return Add(p->next) + p->data;
    }
}
```

Time → O(n)

Space → O(n)

Maximum element in a Linked list →



max = MIN_INT;

max = -32768

12

```
int max(struct Node *p)
{
    int max = INT_MIN;
    while(p)
    {
        if(p->data > max)
        {
            m = p->data;
        }
        p = p->next
    }
    return max;
}
```

T → O(n)

S → O(1)

Recursive function

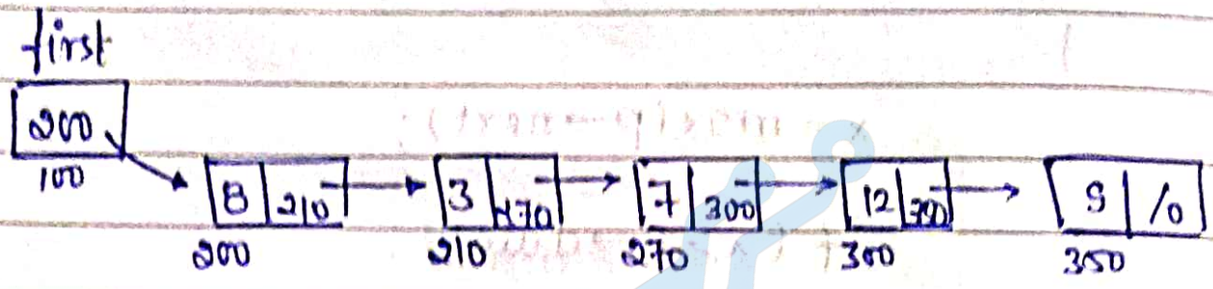
```
int max(Node *p){
    if(p==0)
        return INT_MIN;
    int x=0;
    else
    {
        x = max(p->next);
        if (x>p->data)
        {
            return x;
        }
        else
        {
            return p->data;
        }
    }
```

```
int maxIsIter Node *p)
{
    int x=0;
    if (p==0)
        return MIN_INT;
    x=max(p->next);
    return x>p->data? x : p->data;
}
```

Searching in a linked lists :-

1- Linear search

2.- Binary search (Not suitable for linked lists)

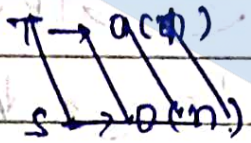


key = 12

```
Node *  int search (Struct Node *p, int key)
{
    while(p!=NULL)
    {
        if(p->data == key)
        {
            return(p);
        }
        p=p->next;
    }
    return 0;
}
```

Recursive function →

```
Node *Search (struct Node *p, int key)
{
    if (p == NULL)
    {
        return NULL;
    }
    if (key == p->data)
    {
        return p;
    }
    return Search(p->next, key);
}
```



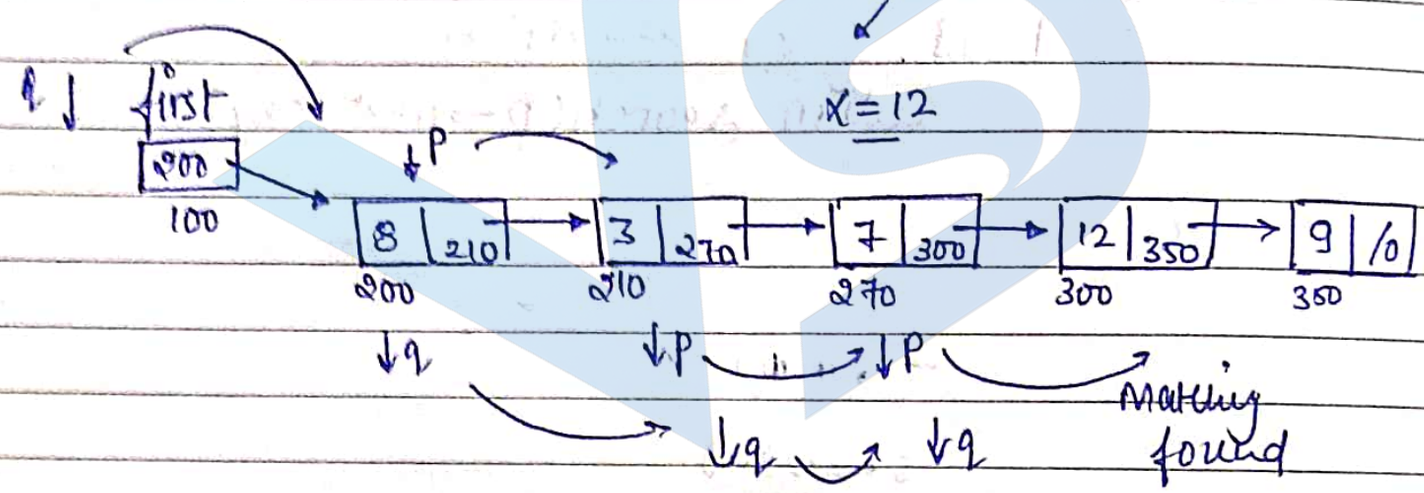
Improving Searching in Linked List :-

Improving the linear search

Two methods

Transportation / Transposition

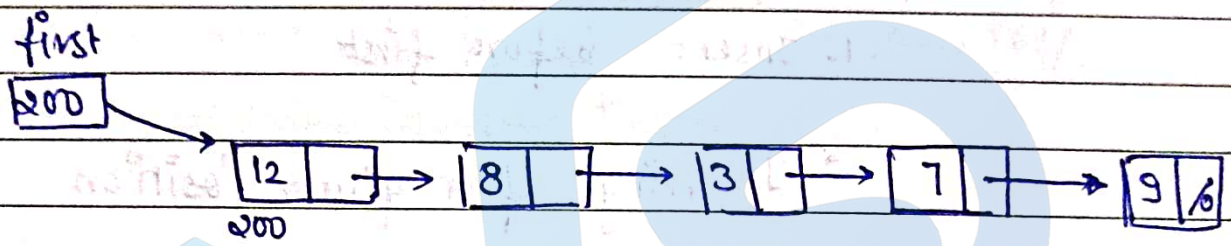
move to head



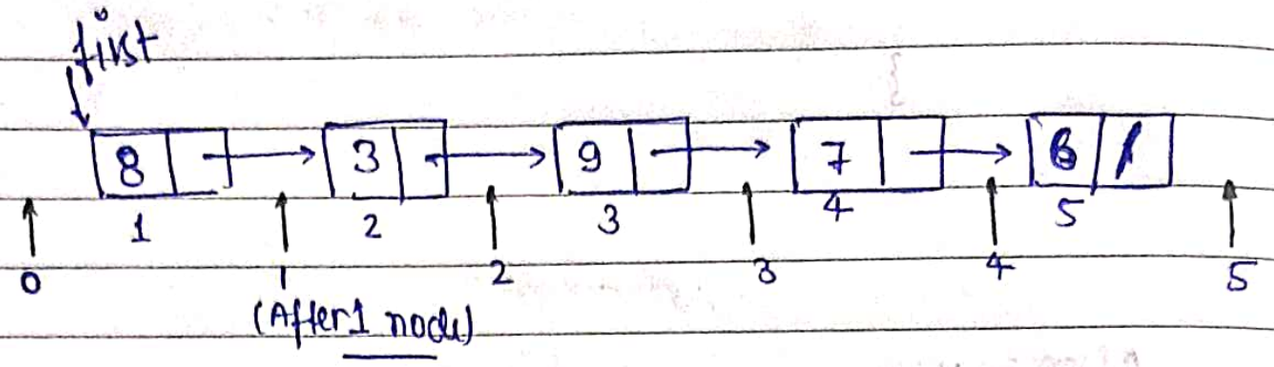

```
Node* search (Node *p, int key)
{
    Node *q = NULL;
    while (p != NULL)
    {
        if (key == p->data)
        {
            q->next = p->next;
            p->next = first;
            first = p;
        }
    }
```

```
        q=p;
        p=p->next;

    }
}
```

After search



Inserting in a linked list :-



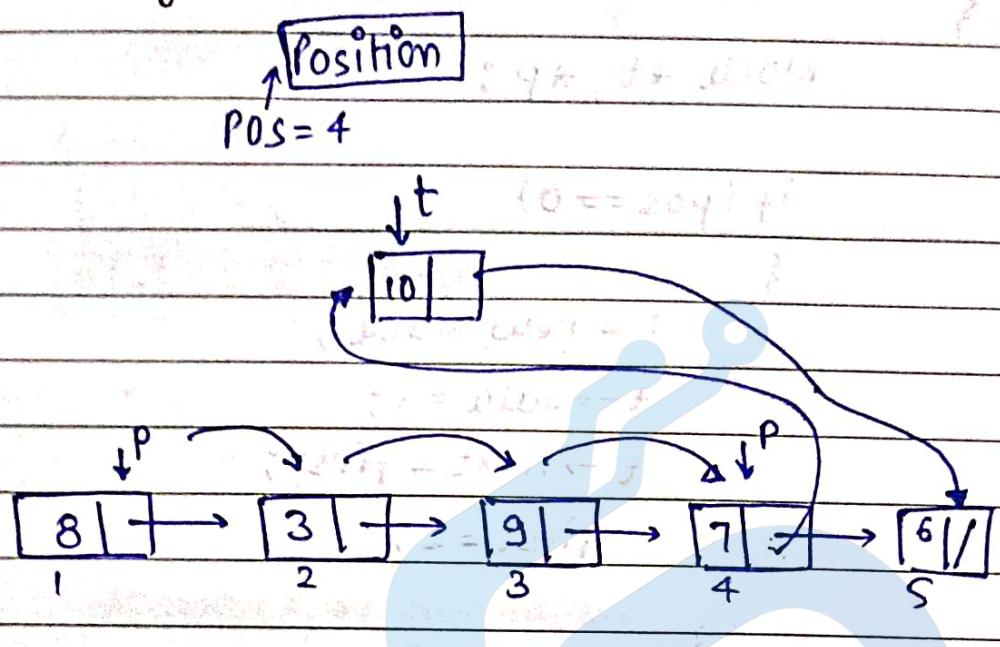
1. Insert before first
2. Inserting after given Position

- Insert before first →

O(1)

```
Node *t = new Node;
t->data = x;
t->next = first;
first = t;
```

- Inserting after given Position



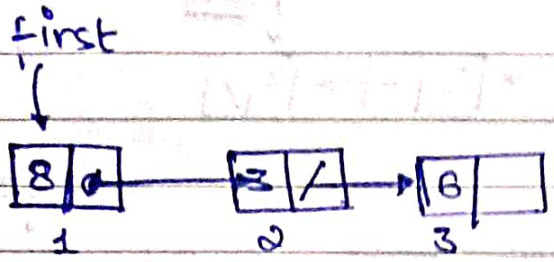

```
Node *t = new Node;
t->data = x;
p = first;

for(int i=0 ; i<pos-1; i++)
{
    p = p->next;
}
t->next = p->next;
p->next = t;
```

O(n)

min → O(1)
Max → O(n)

```
void Insert (int pos, int x)
{
    Node *t, *p;

    if (pos == 0)
    {
        t = new Node;
        t->data = x;
        t->next = first;
        first = t;
    }
    else if (pos > 0)
    {
        p = first;
        for (i=0; i<pos-1 && p; i++)
        {
            p = p->next;
        }
        if (p)
        {
            t = new Node;
            t->data = x;
            t->next = p->next;
            p->next = t;
        }
    }
}
```

Creating a linked list using insert →

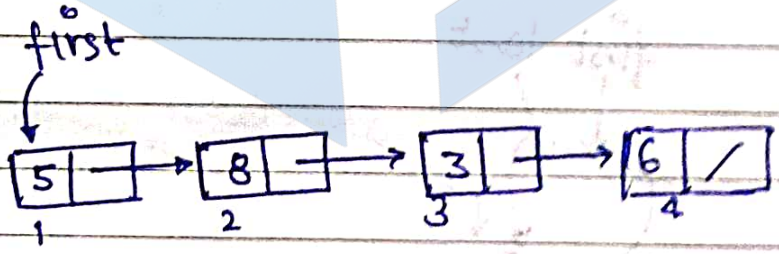


Insert(0,8)

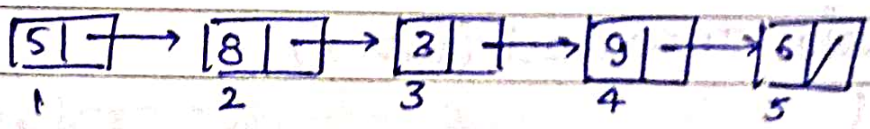
Insert(1,3)

Insert(2,6)

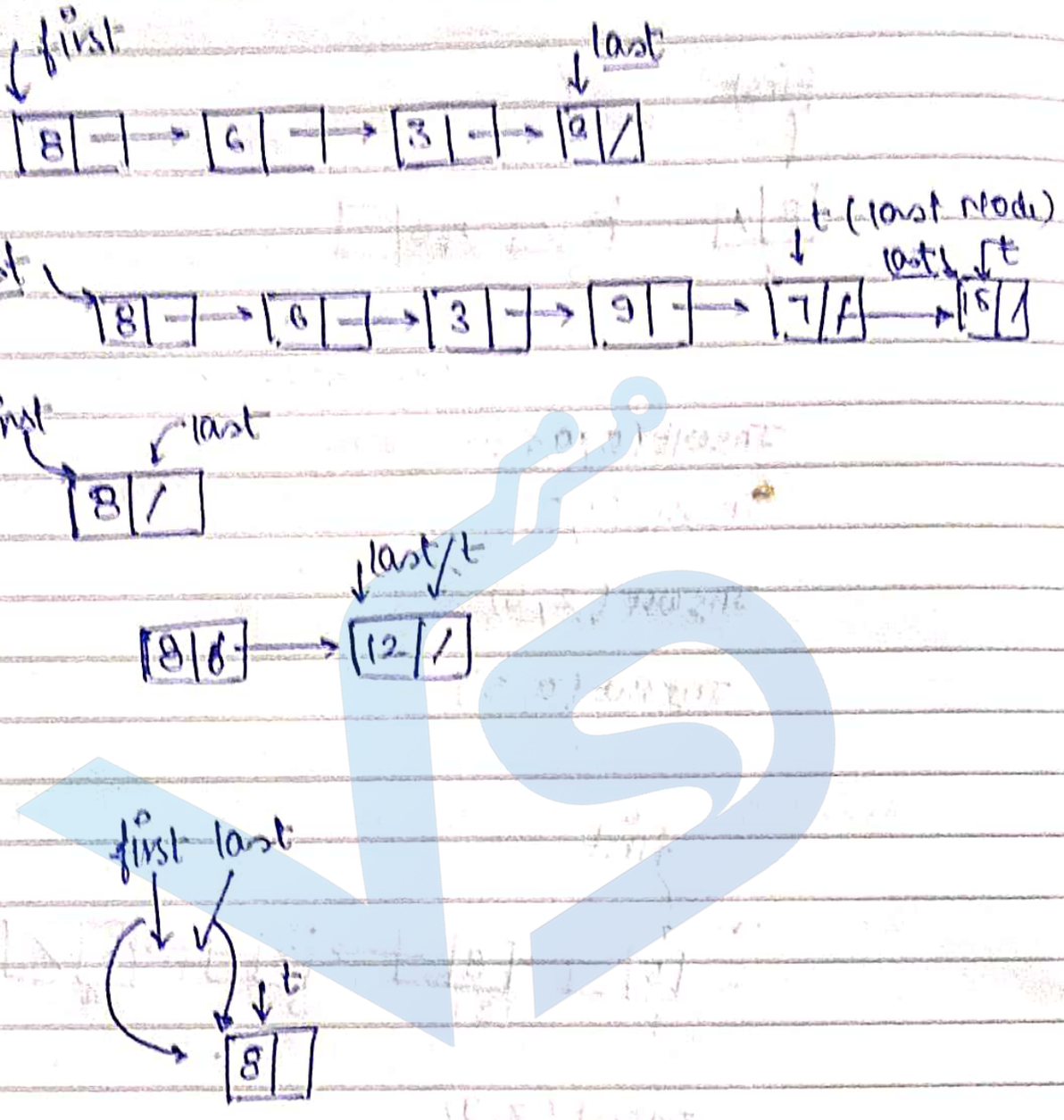
Insert(0,5)



Insert(3,9)



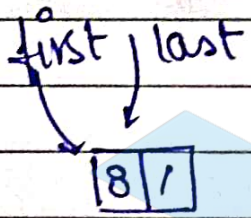
Creating a linked list by inserting at last →



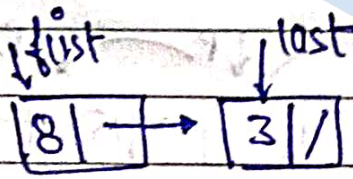

```
Void InsertLast(int x)
{
    Node *t = new Node;
    t->data = x;
    t->next = NULL;
    if(first==NULL)
    {
        first=last=t;
    }
```

```
else
{
    last->next=t;
    last=t;
}

}
```

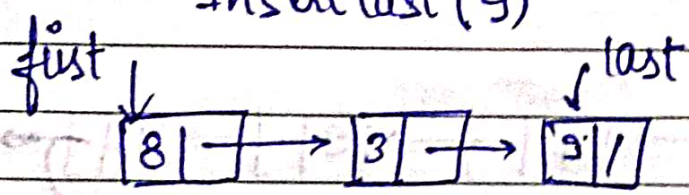
InsertLast(8)



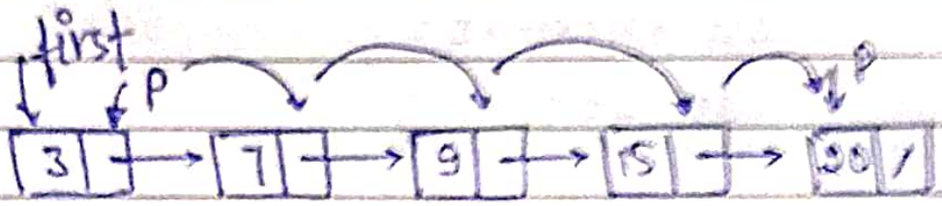
InsertLast(3)



InsertLast(9)



Inserting in a sorted linked lists:-



x = 18 → (Insert)

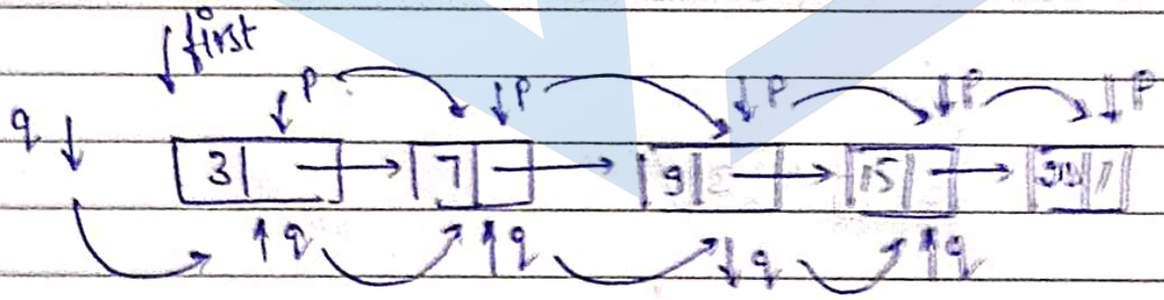
3 < 18 (move P)

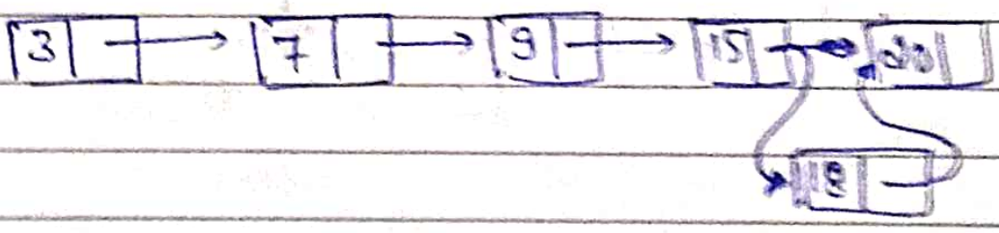
7 < 18 (move P)

9 < 18 (move P)

15 < 18 (move P)

20 > 18 (Stop P)



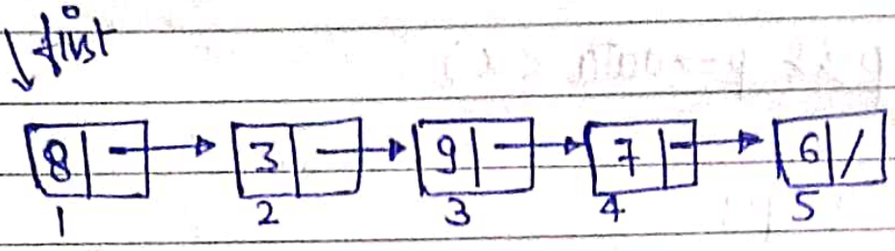


```
p = first;
q = NULL;

while( p && p->data < x)
{
    q = p;
    p = p->next;
}
t = new Node;
t->data = x;

t->next = q->next;
q->next = t;
```

min → O(1)

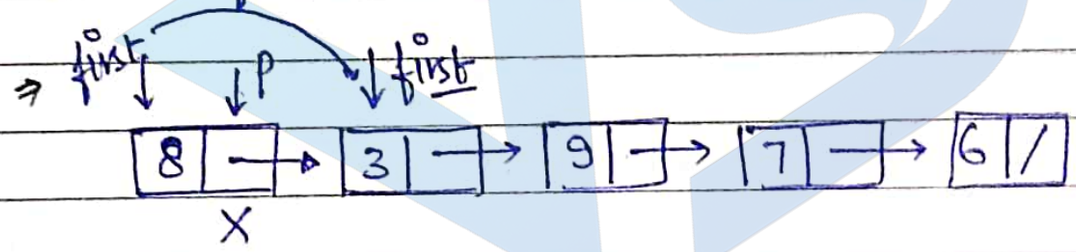
max → O(n)

Deleting from Linked Lists →



1. Delete first Node
2. Delete a Node at given position

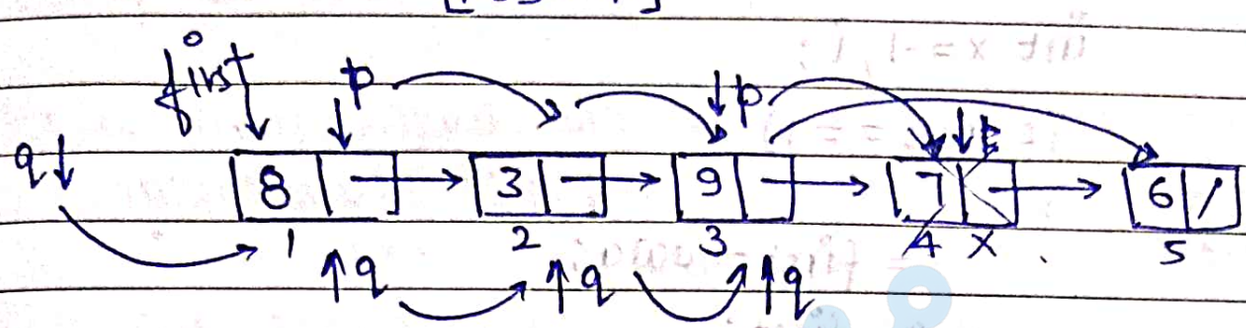
- Delete first Node →




```
Node *p = first;
first = first → next;
x = p→data;
delete p;
```

Time → O(1)

- Delete a Node at given position



Max → O(n)
Min → O(1)

```
Node *p = first;
Node *q = NULL;

for(i=0; i<pos-1; i++)
{
    q = p;
    p = p->next;
}
q->next = p->next;
x = p->data;
delete p;
```

```
int Delete(int pos)
{
    Node *p, *q;
    int x=-1, i;
    if(pos==1)
    {
        x=first->data;
        p=first;
        first=first->next;
        delete p;
    }
    else
    {
        p=first;
        q=NULL;
        for(i=0;i<pos-1 && p;i++)
        {
            q=p;
            p=p->next;
        }
        if(p)
        {
            q->next=p->next;
            x=p->next;
            delete p;
        }
    }
    return x;
}
```

Important !!!!

Why we need linked list?

→ we need linked list when we want to store and manipulate data that has an unknown or variable size. Linked list is a dynamic data structure that can grow and shrink at runtime by allocating and deallocating memory. This means that we don't need to specify the initial size of linked list or waste memory for unused elements.

Difference between Array and Linked list

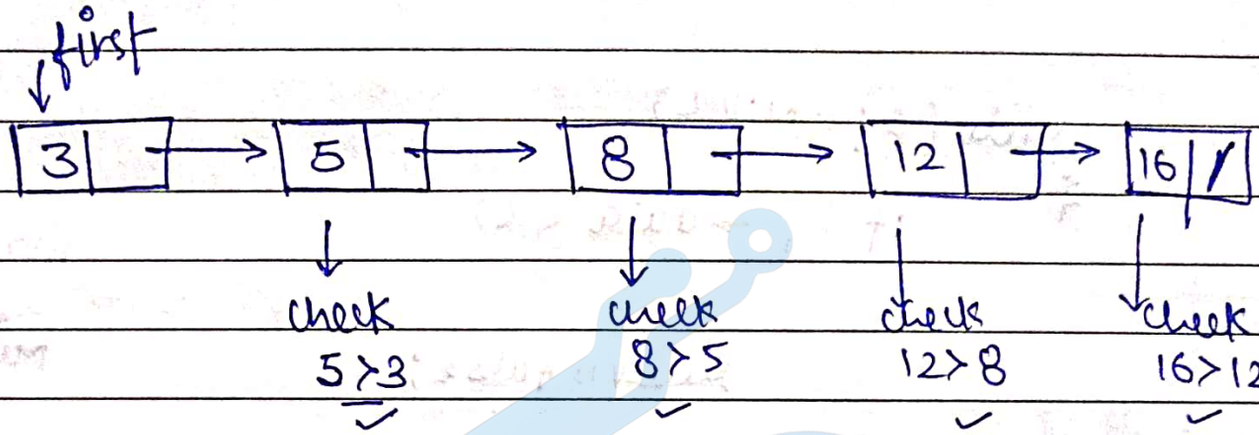
Array	Linked lists
• Arrays are stored in contiguous location.	• Linked list are not stored in contiguous location.
• fixed in size.	• Dynamic size.
• Memory is allocated at compile time.	• Memory is allocated at run time.
• Use less memory than linked lists.	• Use more memory because it stores both data and the address of next node.

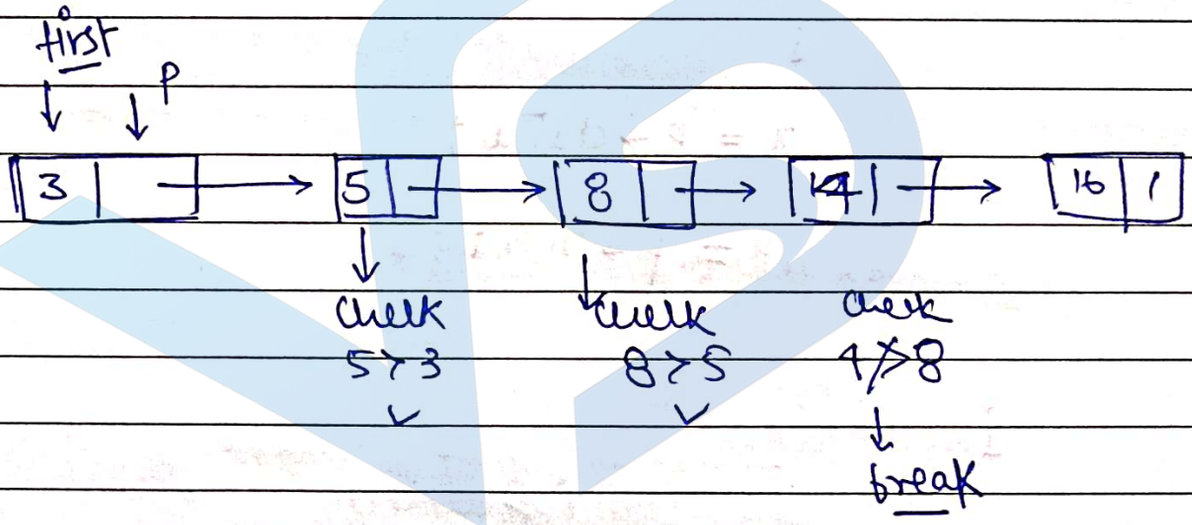
• Elements can be accessed easily.	• Element accessing requires the traversal of whole linked list.
• Insertion and deletion operation takes time.	• Insertion and deletion operation is faster.

Why we are not using vector array because when a vector is resized, it may be necessary to copy all of the elements to a new memory location. This can be time consuming, espically for large vectors.

~~Check if~~

Check if a linked list is sorted





x = -32768; ~~3 5 8 12~~ 16

3 > -32768

5 > 3

8 > 5

12 > 8

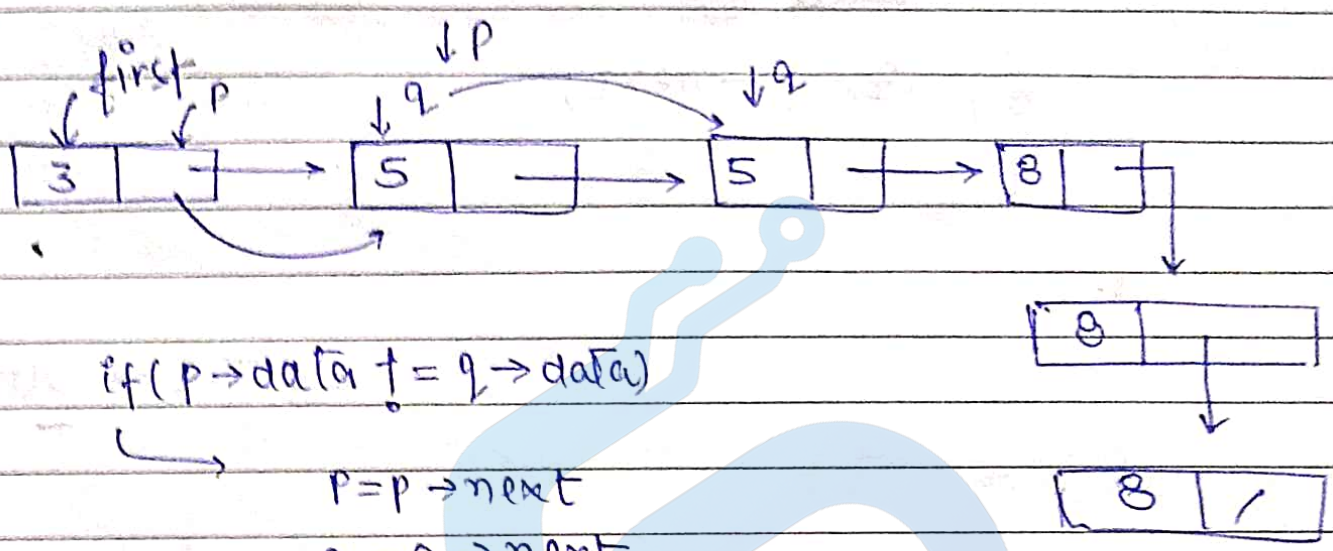
16 > 12

```
int x = INT_MIN;
Node *p = first;

while (p != NULL)
{
    if (p->data < x)
    {
        return false;
    }
    x = p->data;
    p = p->next;
}
```

max → O(n)

min → O(1)

Remove Duplicates from Sorted Linked List




```
if(p->data != q->data)
    p=p->next
    q=q->next
```

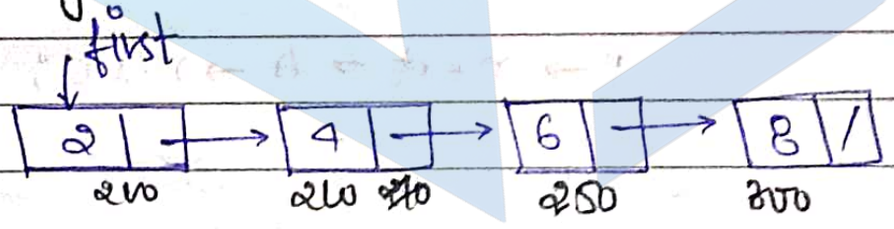
```
if (p->data == q->data)
    p->next = q->next;
    delete q;
```

```
Node *p = first
Node *q = first->next;
while(q!=NULL)
{ if (p->data != q->data)
  {
        p=q;
        q=q->next;
  }
  else
  {
```

```
    p->next = q->next;
    delete q;
    q = p->next;
  }
}
```

T.C → O(n)

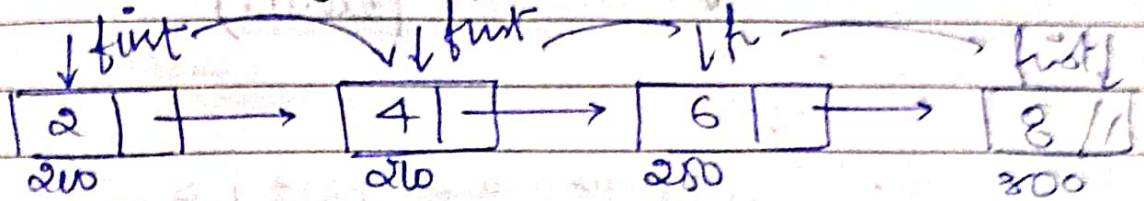
Reversing a linked list :-

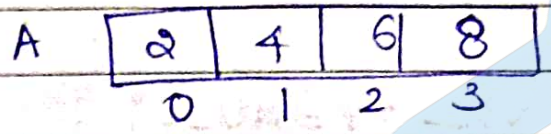


Methods

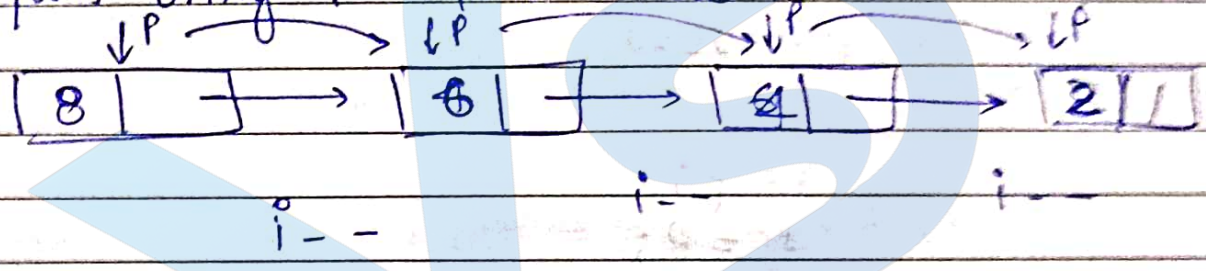
1. Reversing element (swapping)
2. Reversing links
3. Recursive Reverse

Method 1:- Reversing element (swapping)



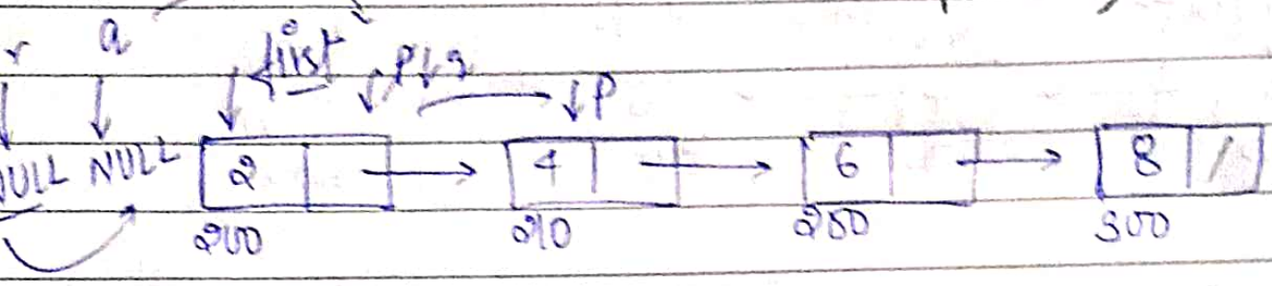


Again bring P on first node

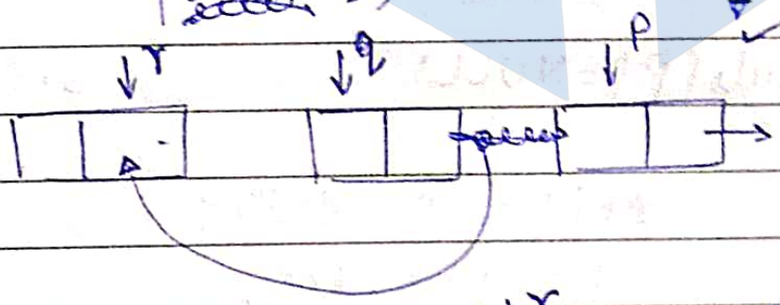


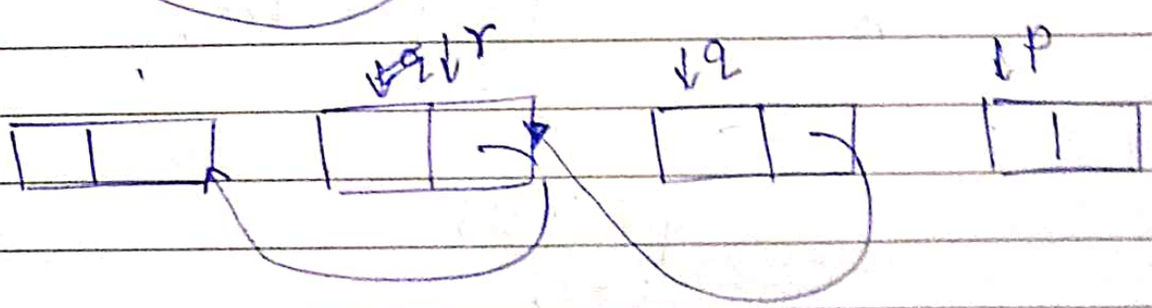

```
Node * p = first ;                          T.C → O(n)
i = 0;
While (p != NULL)                           S.C → O(n)
{
    A[i] = p → data;
    p = p → next;
    i++;
}
p = first;
i--;
While (p != NULL)
{
    p → data = A[i];
    i--;
    p = p → next;
}
```

Method 2 :- Reversing links (Reversing using sliding pointer)

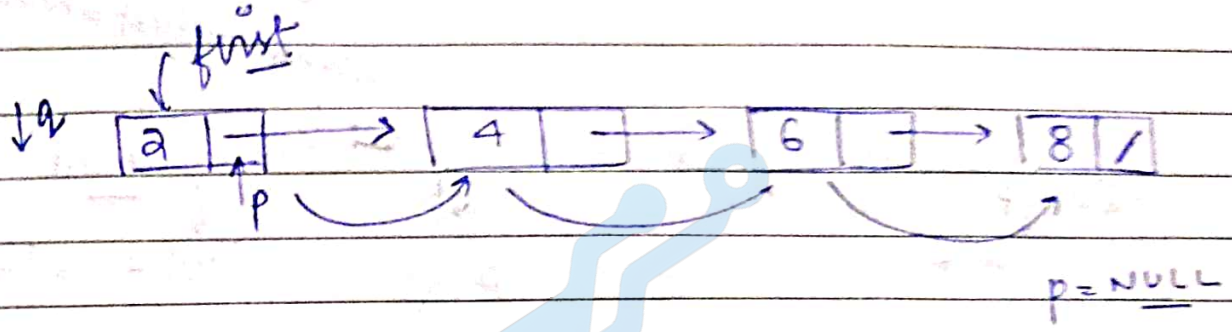



```
Node *p = first, *q = NULL, *r = NULL;

while(p != NULL)
{
    r = p;
    q = p;
    p = p->next;
    q->next = r;
}
first = q;
```

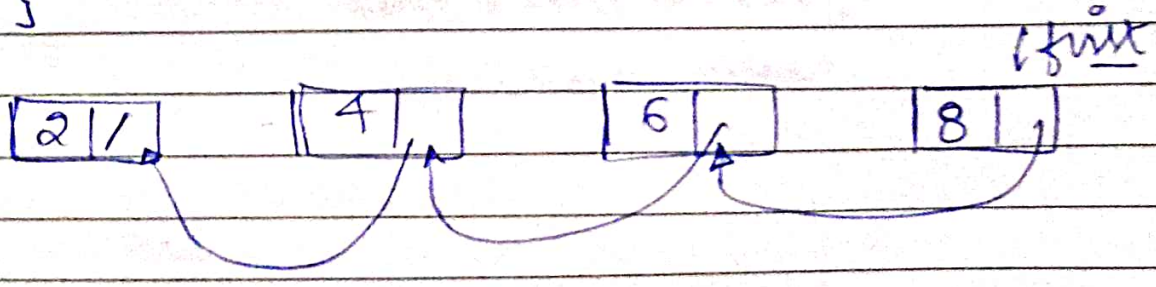




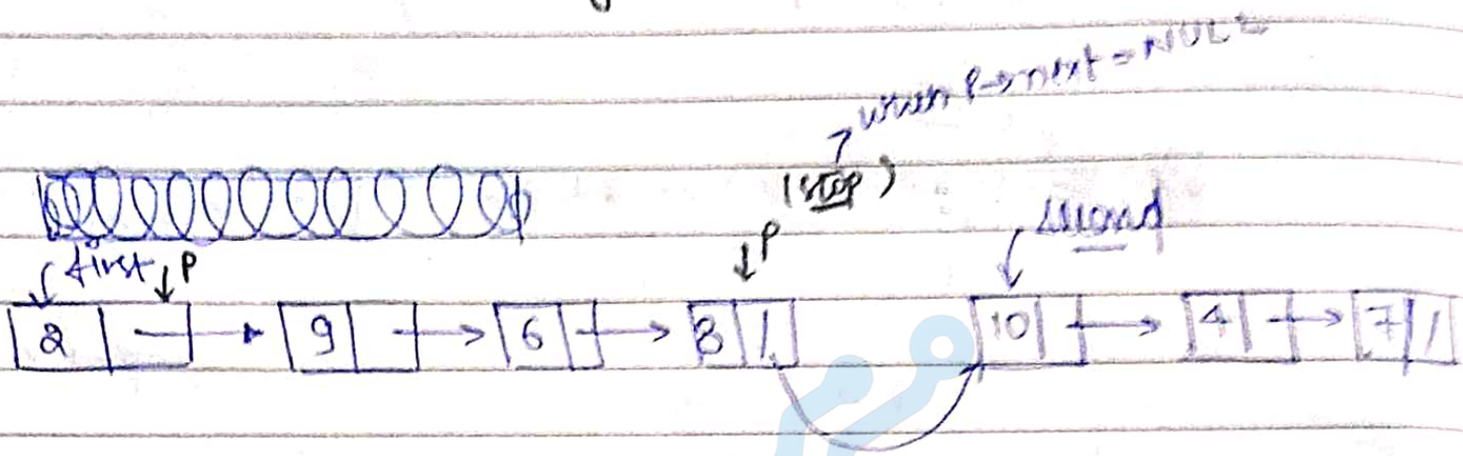
Method 3:- Recursive Reverse for linked list




```
void reverse(Node *q, Node *p)
{
    if(p!=NULL)
    {
        reverse(p, p->next);
        Now in reversing
        p->next=q;
    }
    else
    {
        first=q;
    }
}
```



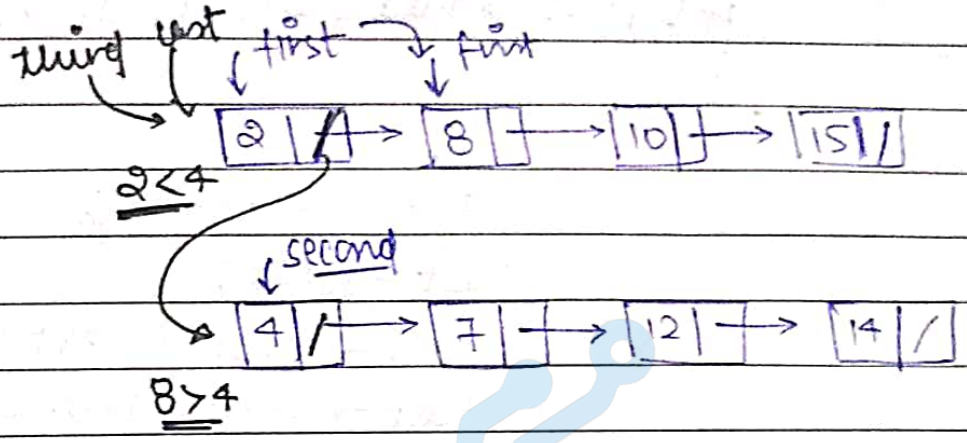
Concatenating 2 linked lists :-




```
Node *p = first
while (p->next != NULL)
{
    p = p->next;
}
p->next = second;
second = NULL;
```

O(n)

Merging 2 Linked lists :-



Merging → combining two sorted lists into single sorted linked lists

third (Point to first)

last (last node of merge linked list)

code:-

```
Node * third, *last;

if (first->data < second->data)
{
        third = last = first;
        first = first->next;
        last->next = NULL;

}
else
{
        third = last = second;
        second = second->next;
        last->next = NULL;
```

S.C → O(1)

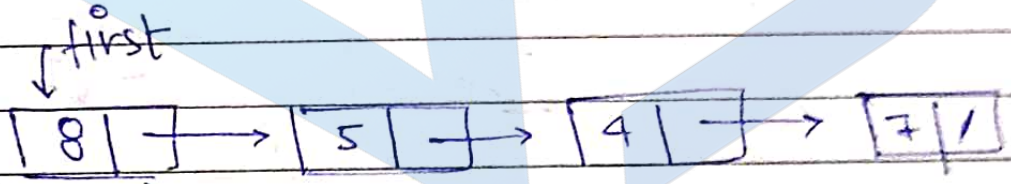
T.C → O(m+n)

```
while ( first != NULL && second != NULL)
{
    if ( first -> data < second -> data)
    {

        last -> next = first;
        last = first;
        first = first -> next;
        last -> next = NULL;

    }
    else
    {
        last -> next = second;
        last = second;
        second = second -> next;
        last -> next = NULL;

    }
}
if (first != NULL)
{
    last -> next = first;
}
else
{
    last -> next = second;

}
```

Check for loop in linked list :-

last node of linked list pointing same node of the linked list.

→ Loop linked list



(linear)

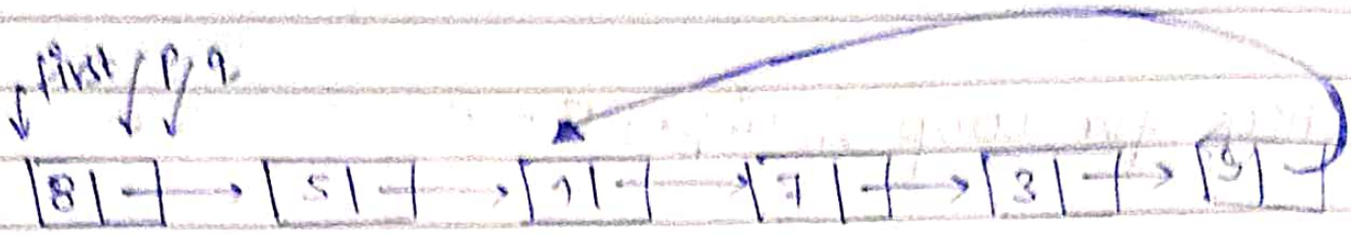
checking linkedlist is linear →

↳ if pointer become NULL

checking linked list is loop →

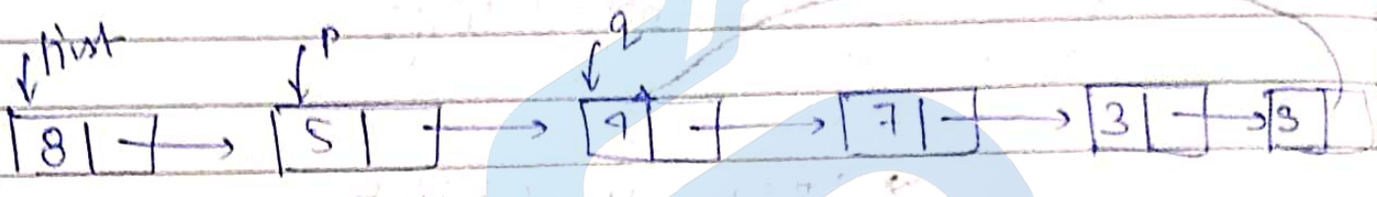
↳ if pointer

↓

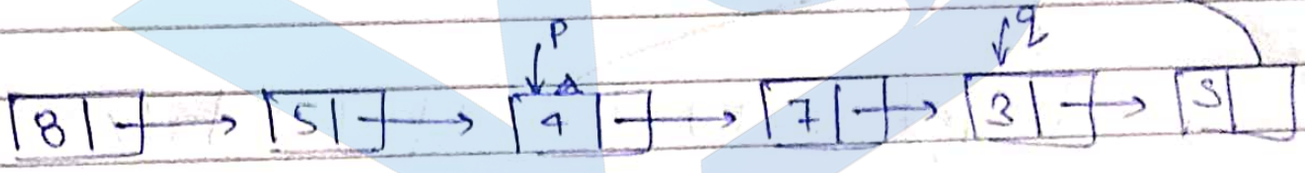


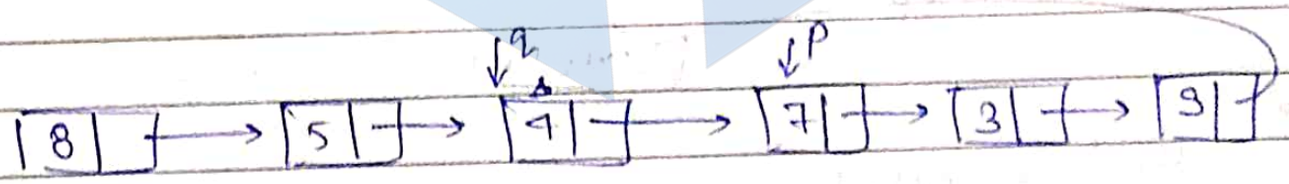
p → move one step

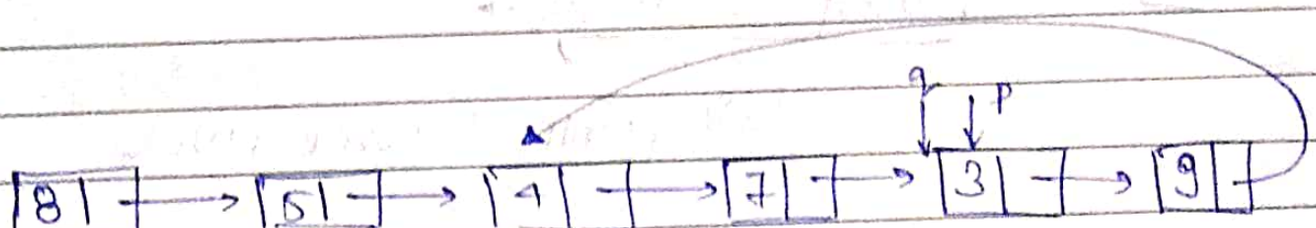
q → move two step



Again move





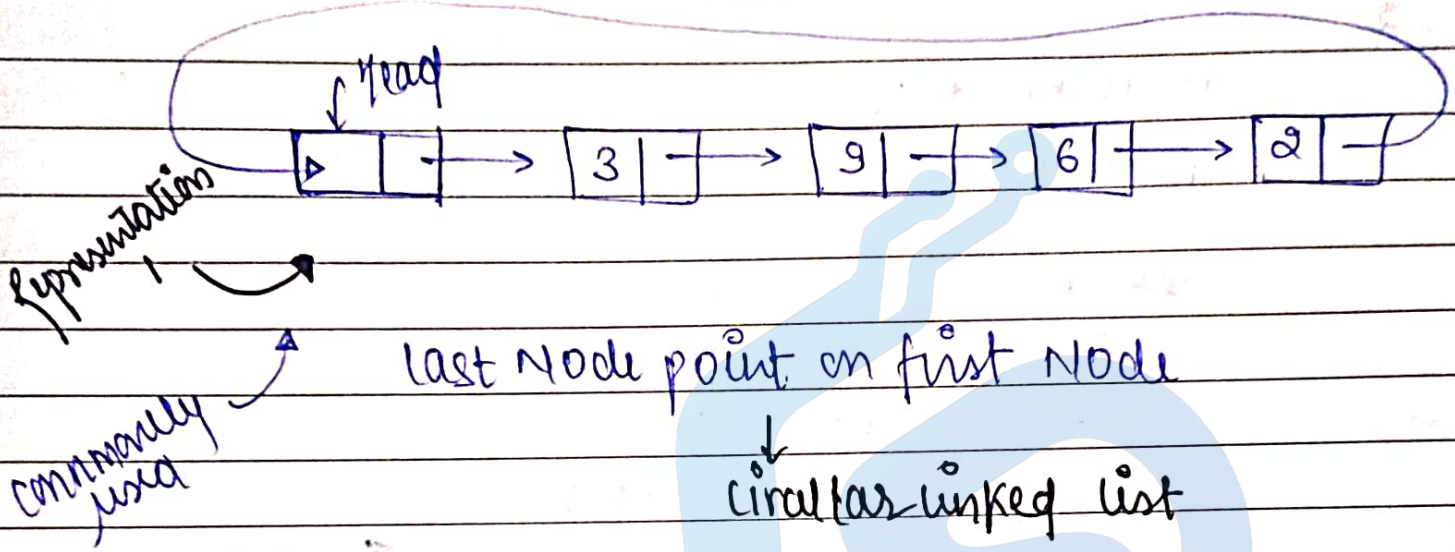


Meet again

O(n) → T.C

```
int isLoop(Node *f)
{
    Node *p, *q;
    p = q = f;

    do
    {
        p = p->next;
        q = q->next;
        q = q != NULL ? q->next : NULL;
    } while(p && q);

    if (p == q)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Circular linked list:-



last Node point on first Node

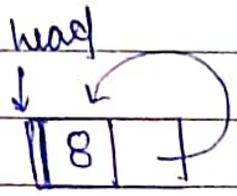
↓

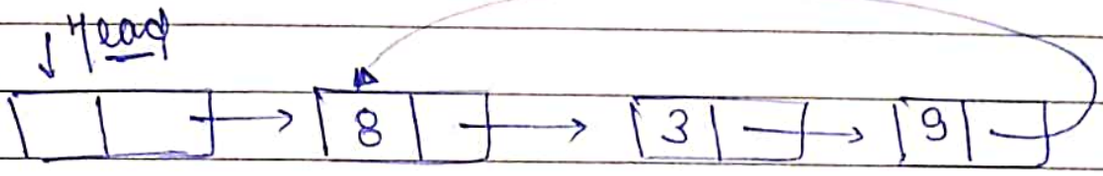
circular linked list

Collection of nodes which are circularly connected

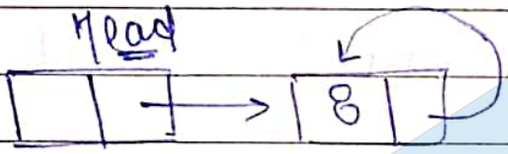
~~head is one first and last node.~~

Benefits :-

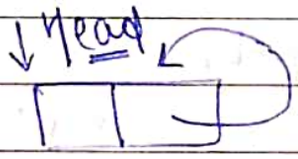




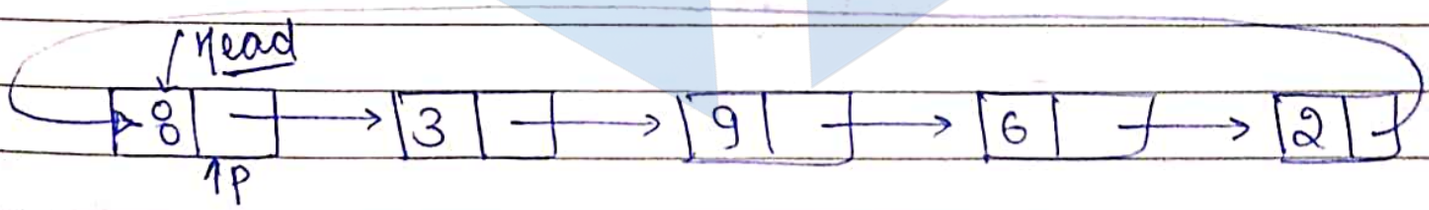
Representation 2



Overcome of empty linked list is Omlea



Display a Circular linked list :-



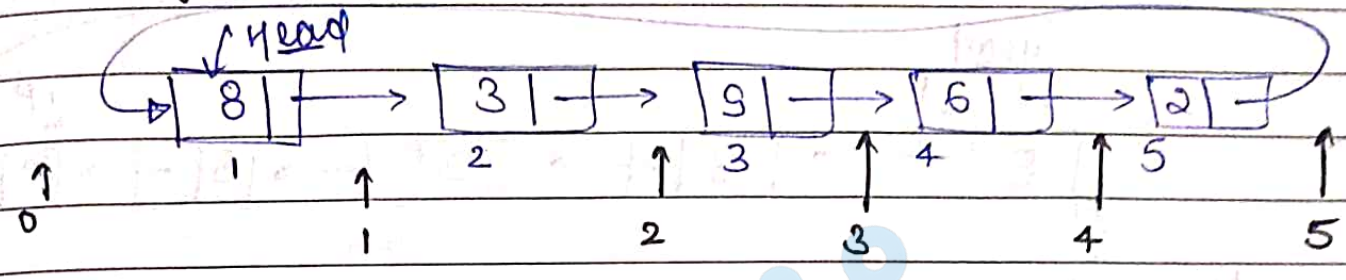

```
void Display(Node *p)
{
    do
    {
        cout<< p->data;
        p=p->next;
    }while(p!=head);
}

Display(head)
```

Recursion Display function

```
void Display( Node *p)
{

    static int flag=0;
        if(p!=head || flag==0)
        {

                flag=1;
                cout<< p->data;
                Display(p->next);

        }
        flag=0;
}
```

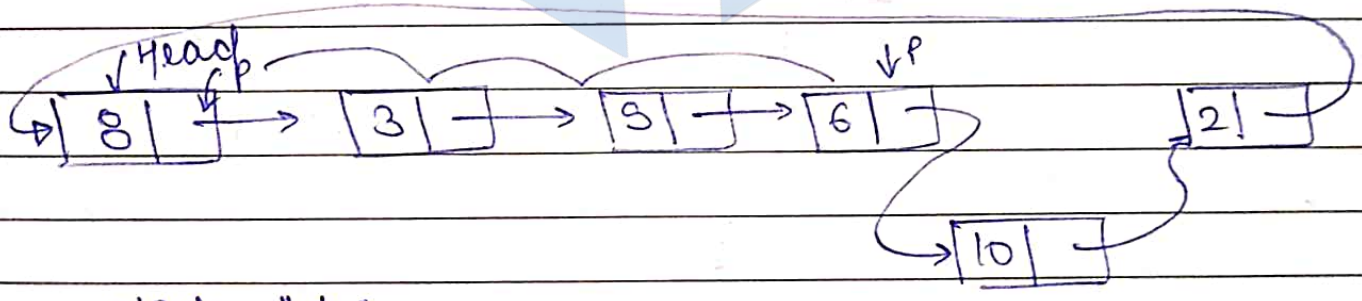
```
Display(head);
```

Inserting in a circular linked list :-



- Insert before head
- Insert at any other position

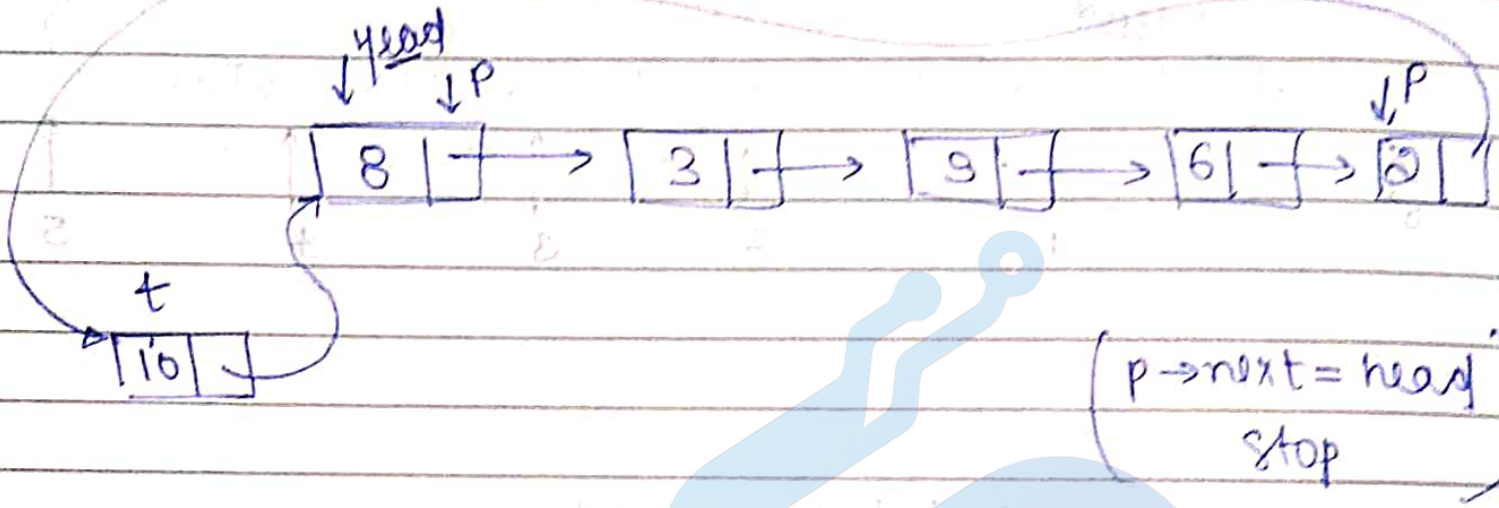
- ~~Insert before head~~ • Insert at any other position




```
Node *t;
Node *p = head;
for(int i=0; i<pos-1; i++)
    p = p->next;
t = new Node;
t->data = x;
t->next = p->next;
p->next = t;
```

```
p->next = t
t->next = p->next;
p->next = t;
```

- Insert before head Node :-



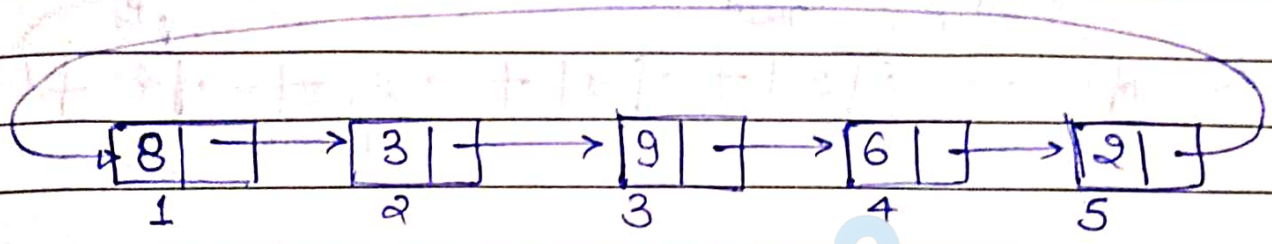

```
Node *p = head;
Node *t = newNode;

t->data = x;
t->next = head;

while(p->next != head)
    {
        p = p->next;
    }
p->next = t;
head = t;
```

Combine

```
void Insert (int pos, int x)
{
    Node *t, *p;
    int i;
    if (pos==0)
    {
        t=new Node;
        t->data=x;
        if (Head==NULL)
        {
            Head=t;
            Head->next=head;
        }
        else
        {
            P=head;
            while(p->next!=Head)
            {
                P=p->next;
            }
            p->next=t;
            t->next=head;
            Head=t;
        }
    }
```

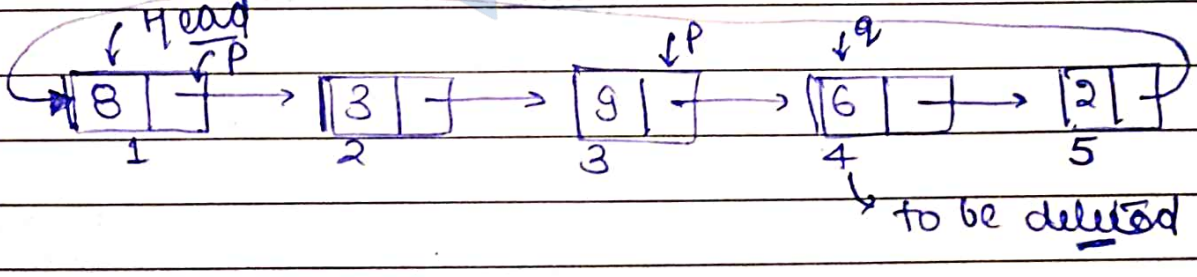
```
else
{
    p=head;
    for(int i=0; i<pos-1; i++)
    {
        p=p->next;
    }
    t=newNode;
    t->data=x;
    t->next=p->next;
    p->next=t;

  }

}
```

Deleting from Circular linked list :-



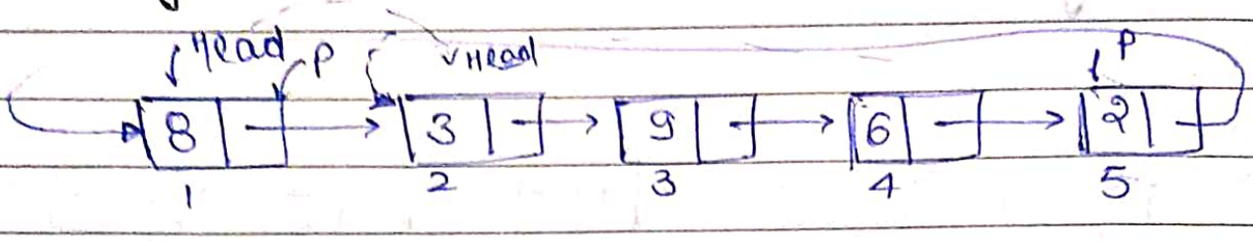
1. Deleting Head Node
2. Deleting a Node from given position.

- Deleting a Node from given position :-




```
P= Head;
for (i=0; i<pos-2; i++)
{
    P=p->next;
}
q=p->next;
p->next = q->next;
x=q->data;
delete q;
```

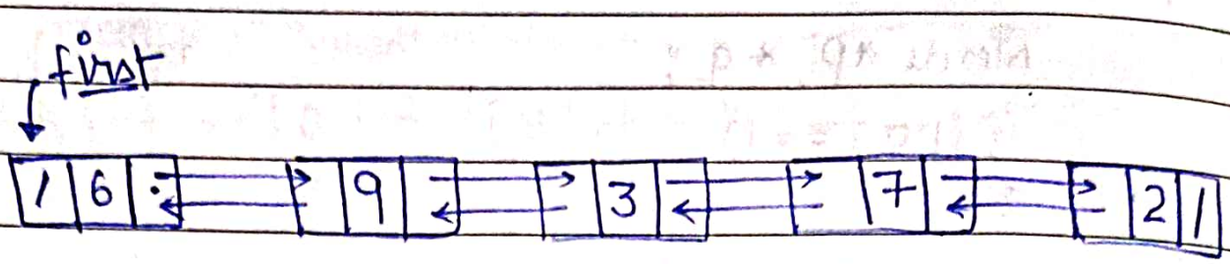
- Deleting Head Node :-




```
P = Head;
while (p->next != Head)
{
    p = p->next;
}
p->next = head->next;
x = head->data;
~~Head = p->next;~~
delete Head;
Head = p->next;
```

```
int Delete(int pos)
{
    Node *p, *q;
    if(pos==1)
    {
        p=Head;
        while(p->next!=Head)
            p=p->next;

        x=Head->data;
        if(p==Head)
        {
            delete Head;
            Head=NULL;
        }
        else
        {
            p->next=Head->next;
            delete Head;
            Head=p->next;

        }

    }
    else
    {
        p=Head;
        for(i=0;i<pos-2;i++)
            p=p->next;

        q=p->next;
        x=p->next;
        delete q
} return x; }
```

Doubly linked list:-

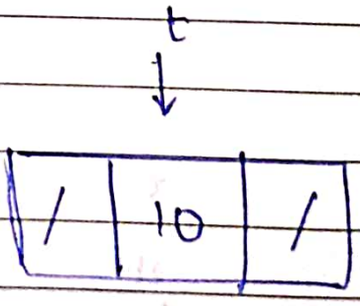


Node

prev	data	next

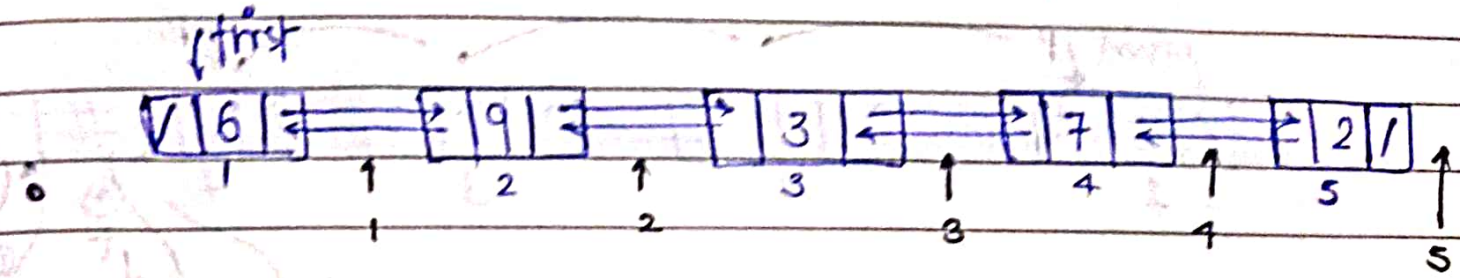
```
Struct Node
{
    Struct Node *prev;
    int data;
    Struct Node *next;
}
```

Struct Node *t;



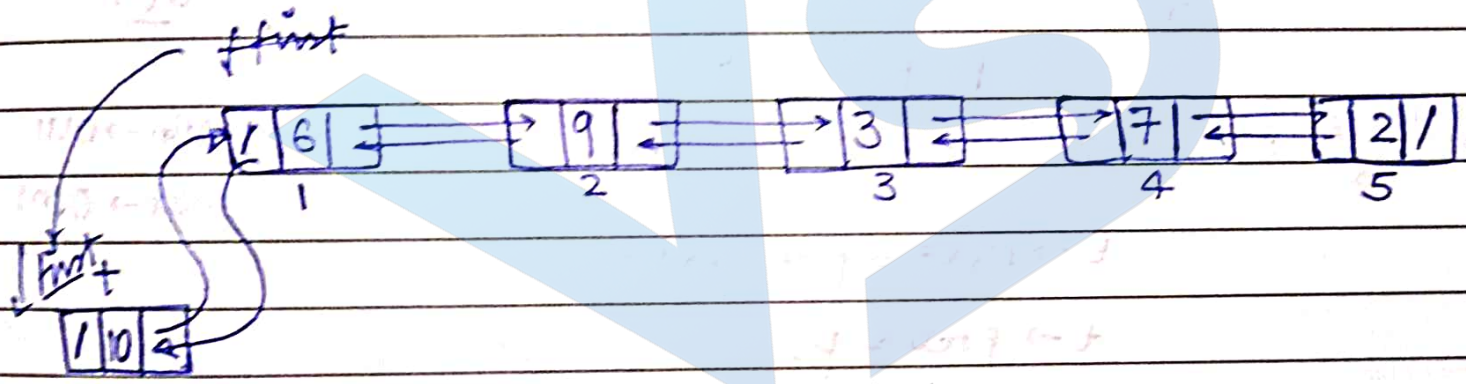

```
t = new Node;
t->prev = NULL;
t->data = 10;
t->next = NULL;
```

Insert in a Doubly Linked List :-



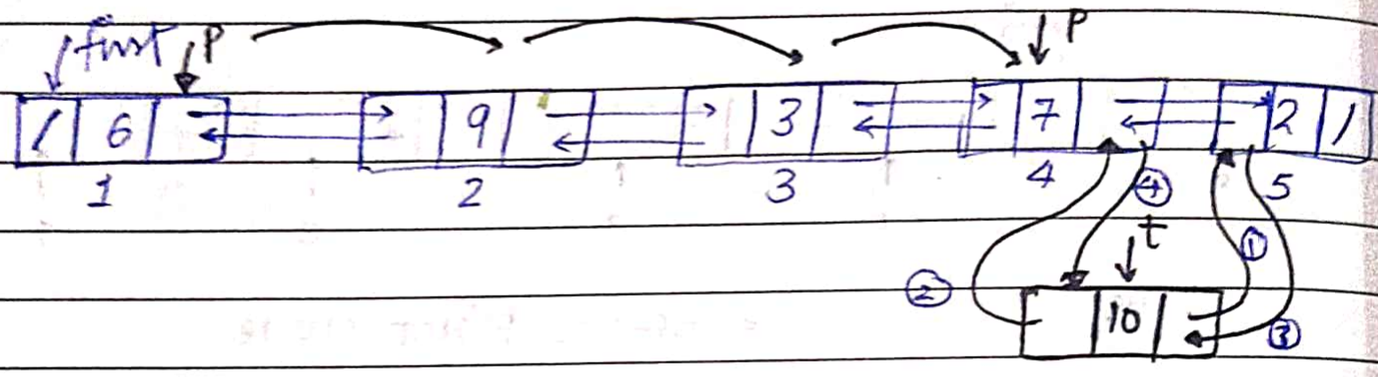
- Before First Node
- At any given Index

- Before first Node :-




```
Node *t = new Node
t -> data = x;
t -> prev = NULL;
t -> next = first;
first -> prev = t;
first = t;
```

- At any given Index :- (Pos = 4)



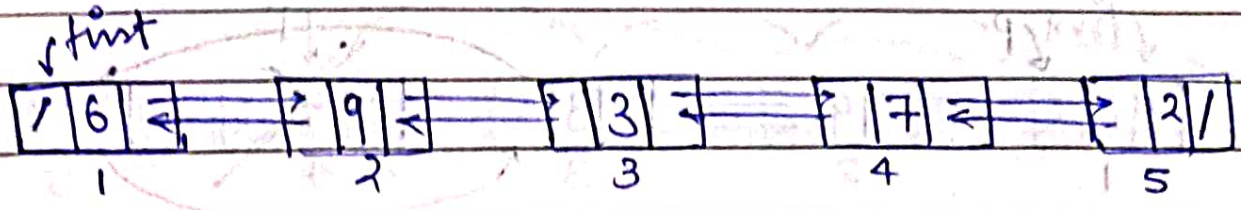

```
Node *t= new Node;
t→data = x;
for(int i=0; i<pos-1; i++)
{
        p=p→next;
}
t→next = p→next;
t→prev = p;
if(p→next!=NULL)
{
        p→next→prev = t;
}
p→next = t;
```

T.C. → O(n)

Min → O(1)

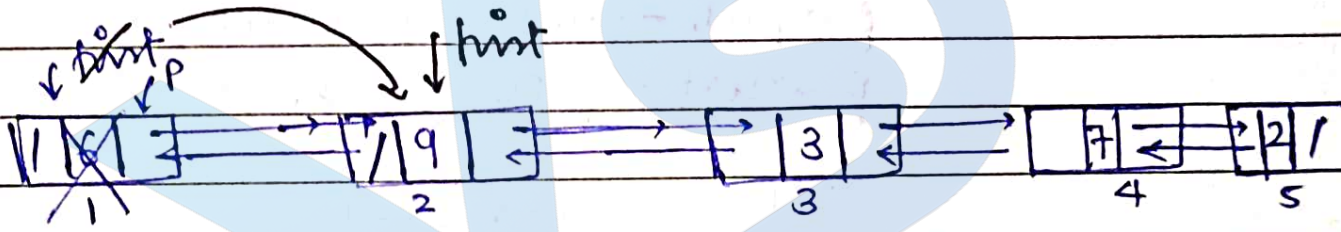
Max → O(n)

Deleting from Doubly linked list:-



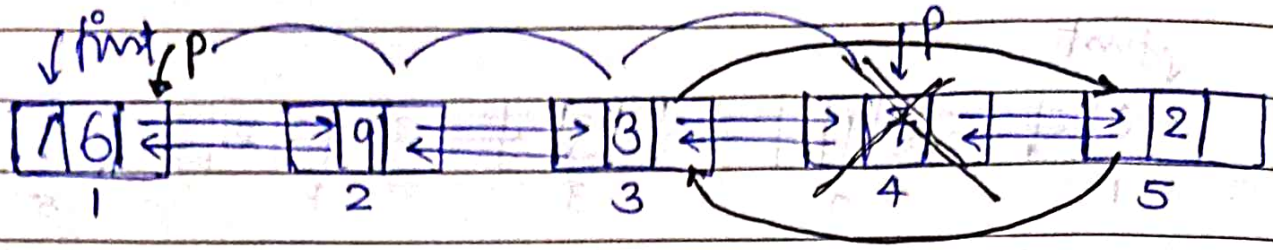
1. Delete first Node
2. Delete from given Index.

- Delete first Node :-




```
p = first;
first = first → next;
x = p → data;                    O(1)
delete p;
if (first != NULL)
{
        first → prev = NULL;
}
```

- Delete from given index :- [POS = 4]



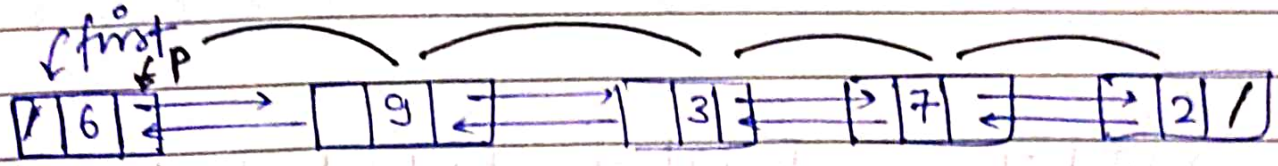

```
p = first;
for (i=0; i<pos-1; i++)
{
        p = p->next;
}
p->prev->next = p->next;
p->next->prev = p->prev;
if(p->next != NULL)
{
        p->next->prev = p->prev;
}
delete p;
x = p->data;
delete p;
```

T.C

↳ Min → O(1)

Max → O(n)

Reverse a Doubly linked list :-

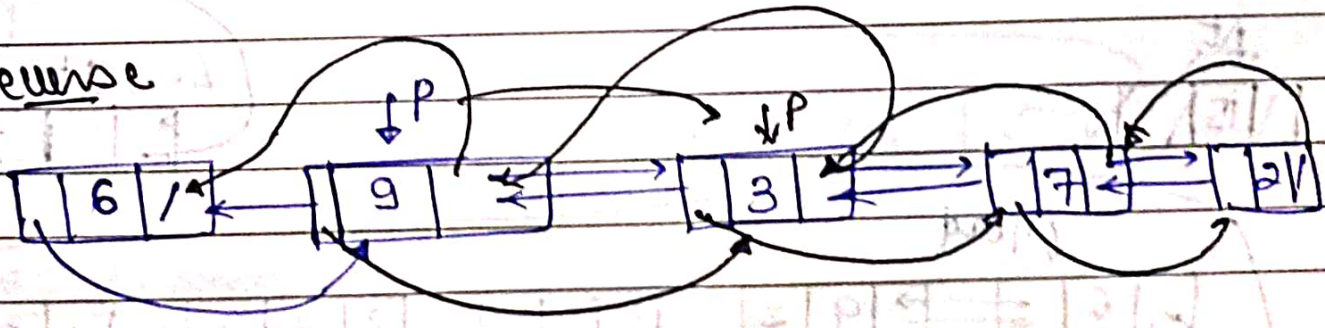


1. Display.
2. Reverse.

- **Display**

```
p=first;

while( p!=NULL    )
{

    cout<<p->data;
    p=p->next;

}
```

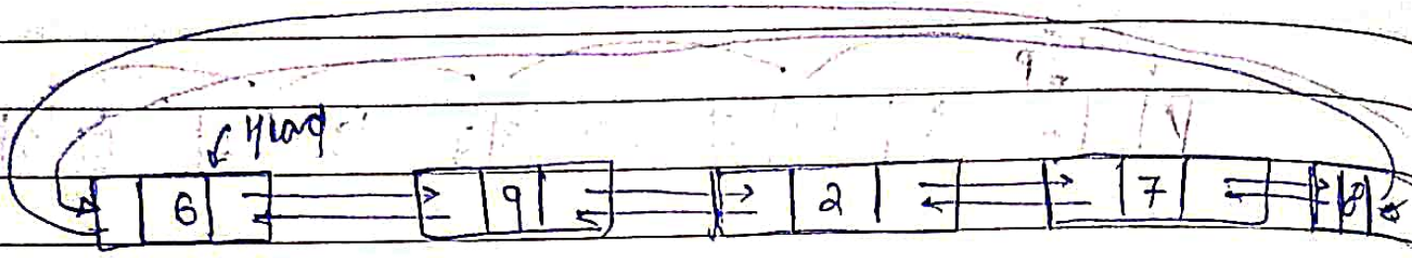
- **Reverse**




```
p=first;
while(p!=NULL)
{
    temp=p->next;
    p->next=p->prev;
    p->prev=temp;
    p=p->prev;
}
```

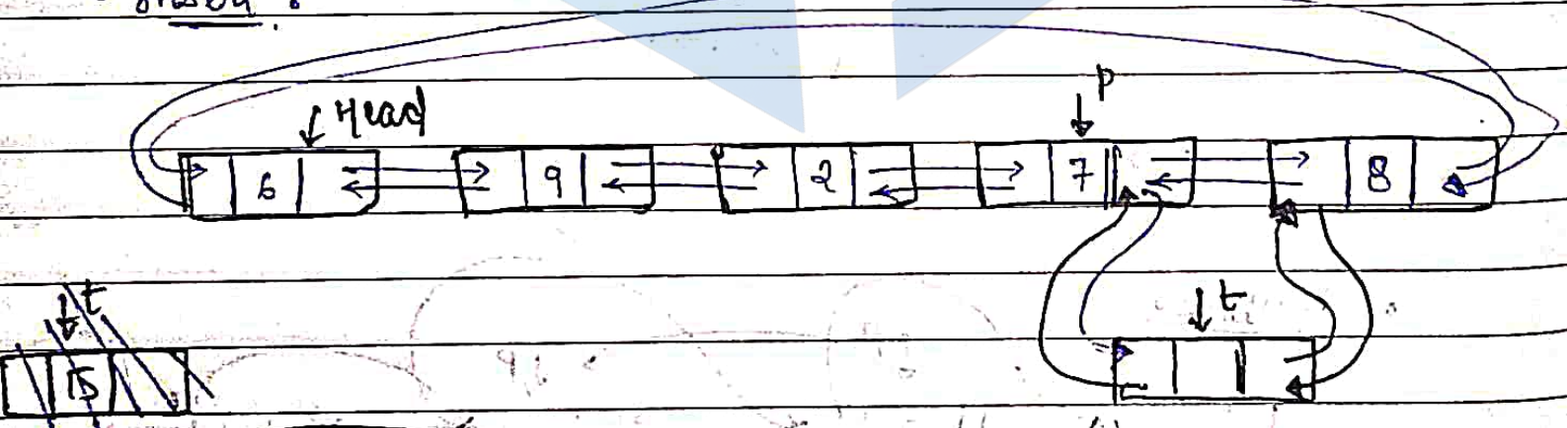
```
if(p->next==NU
{ first=p;
}
}
```

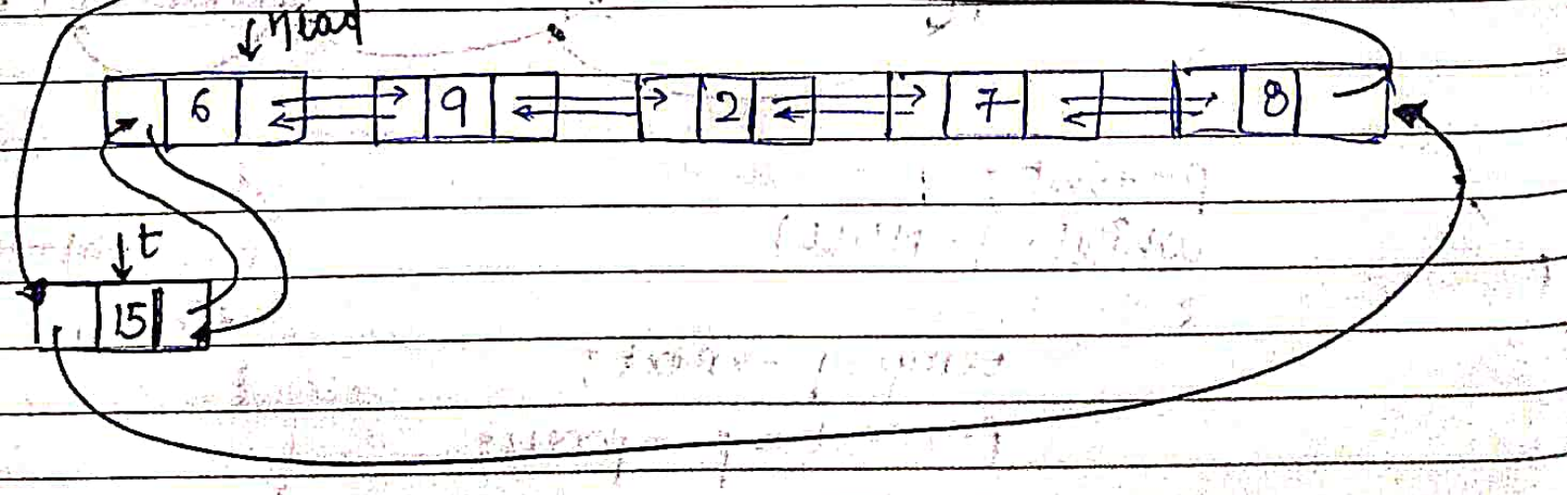
Circular Doubly Linked list :-

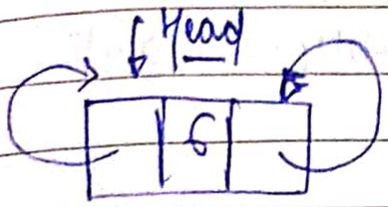


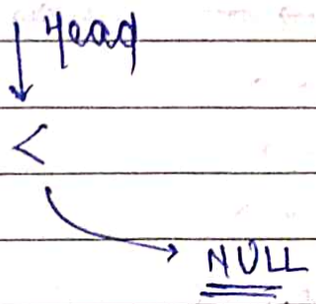

```
P = Head;
do
{
    cout << p -> data;
    p = p -> next;
} while( p != Head)
```

- Insert :-





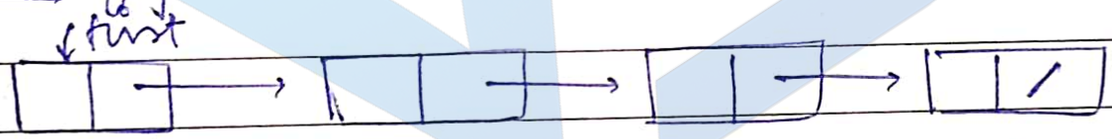




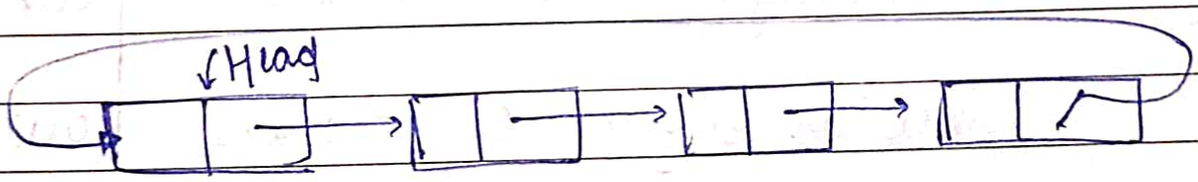
- Insert
- Delete

- Comparison of linked list

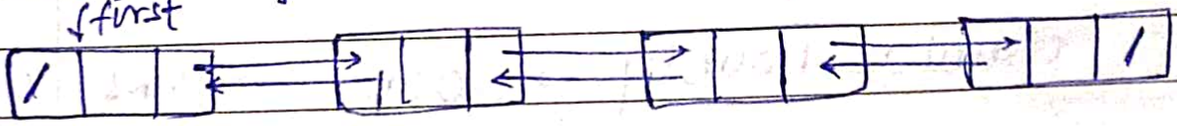
Linear Singly



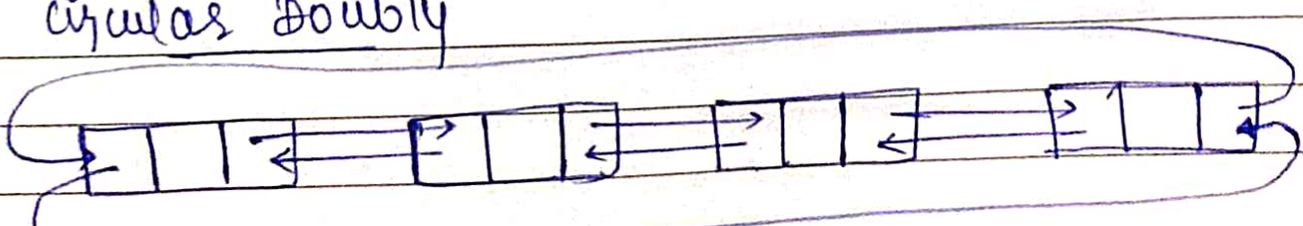
Circular Singly



Linear Doubly



Circular Doubly



space → linear and circular singly → 2 pointers
→ linear and circular Doubly → 3 pointers

Insert →

Delete →

Traverse →

↳ Singly → linear direction

↳ Circular singly → linear and circular

↳ linear Doubly →
- forward and backward

→ Circular Doubly →
forward, Backward and circular

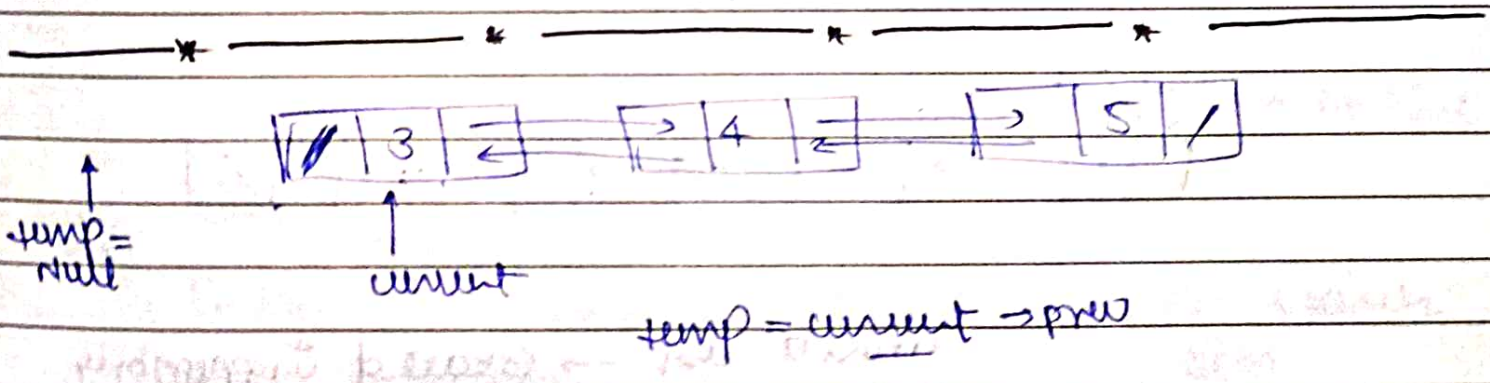
Insert	Before head (Modified)	Given At position Min	Max
Linear Singly →	O(1) → 1 links	O(1)	O(n) 2 links
Circular singly →	O(n) → 2 links	O(1)	O(n) 4 links
linear Doubly →	O(1) → 3 links	O(1)	O(n) 3 links
Circular Doubly →	O(n) → 4 links	O(1)	O(n) 4 links

Delete

	Head		At Given position		
		Modified			
Linear Singly	O(1)	NO links	O(1)	O(n)	1 link
Circular Singly	O(n)	1 link	O(1)	O(n)	1 link
~~Ckd~~ Doubly Singly	O(1)	1 link	O(1)	O(n)	2 link
Circular Doubly	O(1)	2 link	O(1)	O(n)	2 link

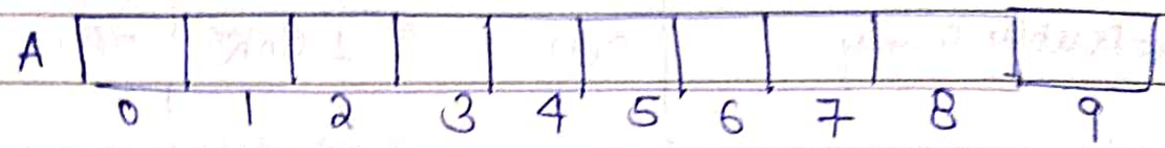
Linear Singly → Queue

Doubly Circular → Best



Comparison of Array with linked list :-

Array vs linked list

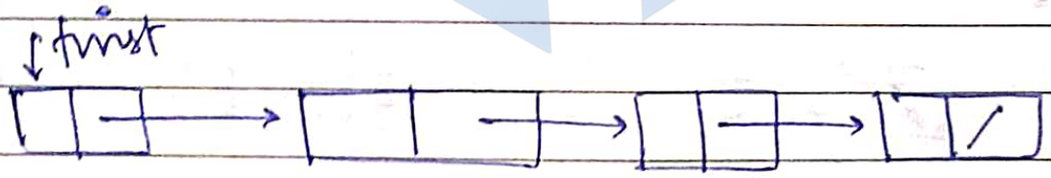


Array → Heap
→ Stack

exists in memory

size → fixed

fixed size, so the chance of getting perfectly utilizes is less
↳ space left or not sufficient to store no. of element



linked list → created in memory

↓

heap

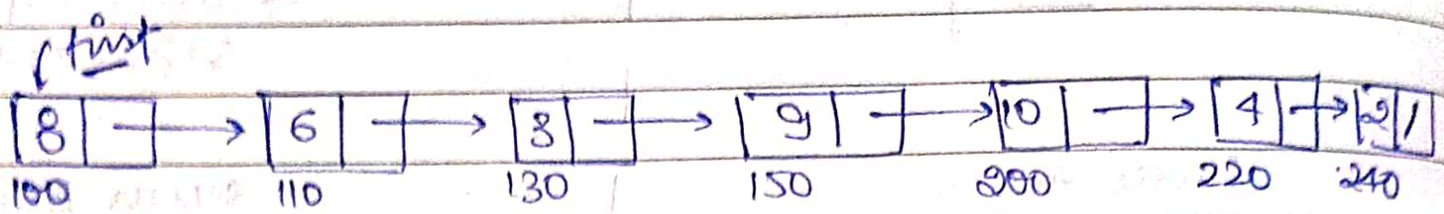
size → variable size

When not sure about how many element is used, then used linked list.

Take extra space as compared to Array

Array	Linked list
Access Randomly	Not Accessed sequentially.
faster Access	slower Access
Insert → O(1) (At last) ↓ O(n) At first	Insert → O(1) (At last) ↓ O(1) At first
It is costly (to move data)	It is not costly
Delete → O(1) → last ↓ O(n) At first	Delete → O(n) → last ↓ O(1) → At first
Linear / binary search O(n) O(log n)	linear search O(n) Binary search (nlogn) (NOT perform)
Insertion	Insertion and Merge sort

Student Challenge :- Finding Middle Element of a linked list

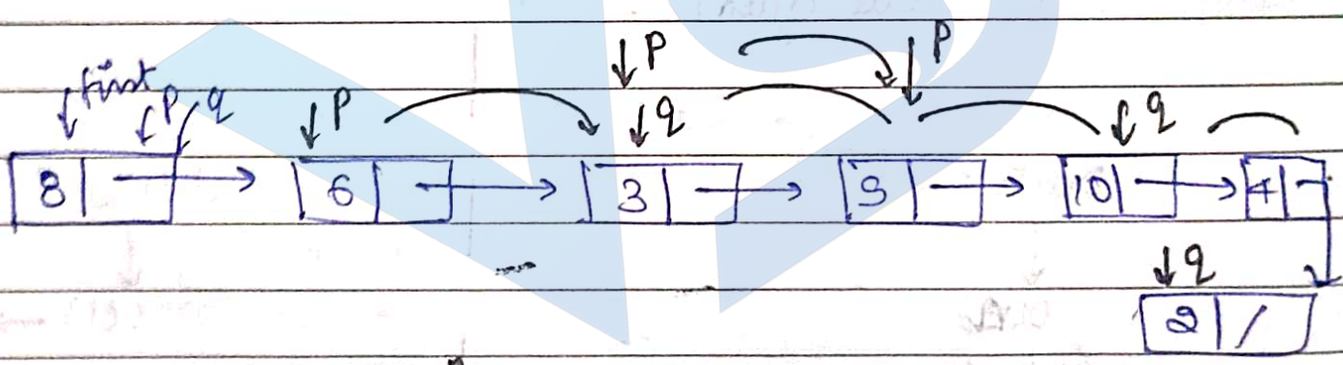


O(n) — 1. Find length → 7

O(n) — 2. Reached to Middle → 7/2 → 3.5 → (3.5) = 4

first

O(2n)



p → one times

q → two times

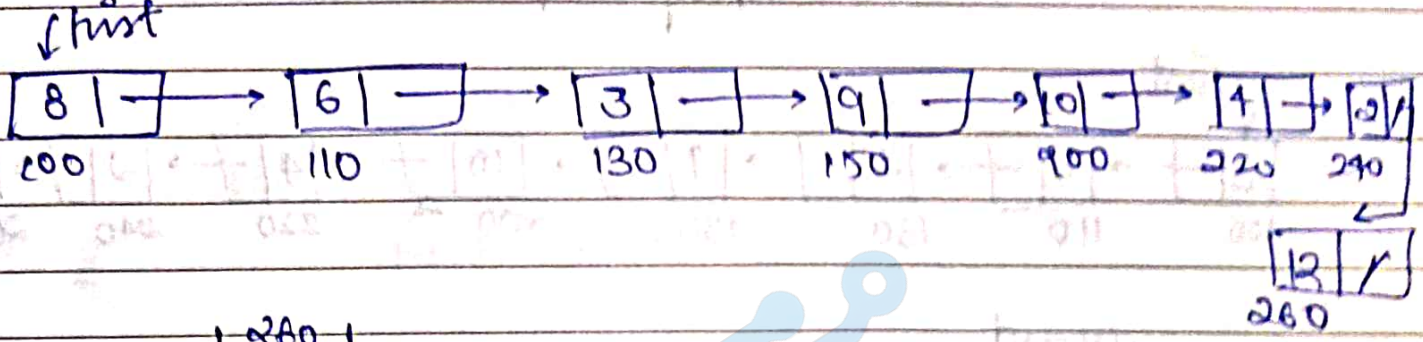
→ move

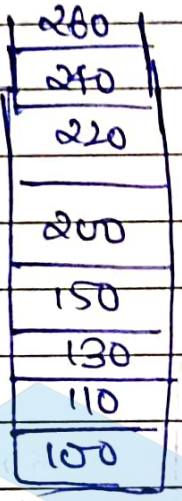
q→next

```
while(q!=NULL)
{
    q=q->next;
    if(q!=NULL)
    {
        q=q->next;
    }
    if(q!=NULL)
    {
        p=p->next;
    }
}
```

cout<<p->data;

Using stack :-





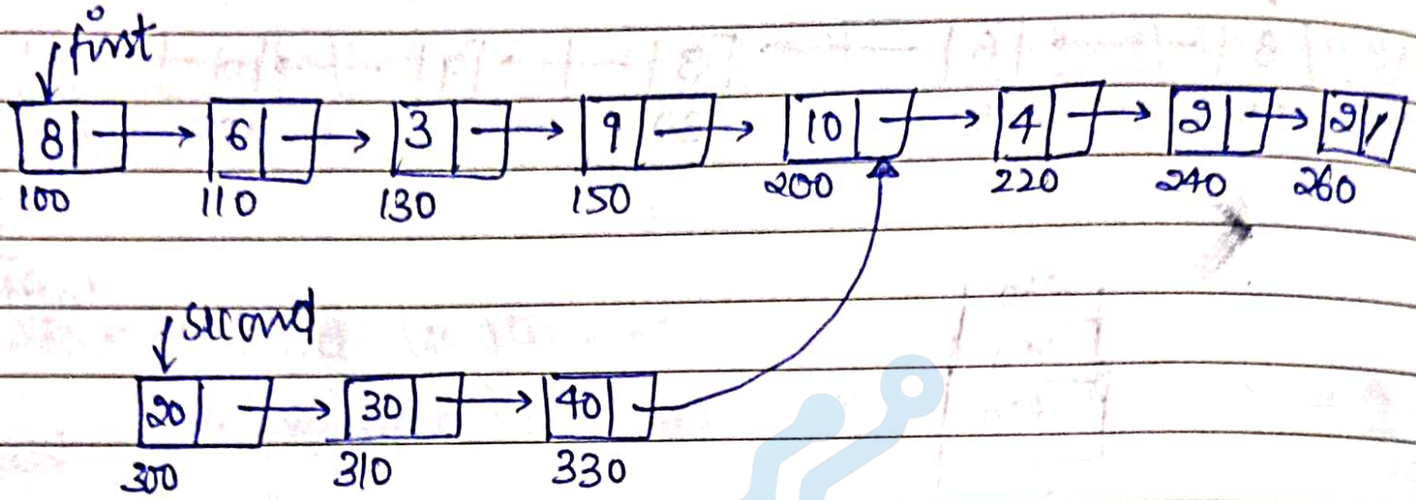
→ 8/2 → 4

Addr → 150

T.C
↓
O(n)

S.C
↓
O(n)

Student challenge: Finding intersecting point of Two Linked list :-



200

S1
~~2 260~~
~~2 240~~
~~4 220~~
~~10 200~~
~~9 150~~
3 130
6 110
8 100

S2
~~260~~
~~240~~
~~220~~
~~200~~
330
310
300

Method 1: Using stack

T.C → $O(n+n)+n$

$O(3n)$

S.C → $O(n)$

p = first;
q = second;

```
while (p != NULL)
{
        push (&stk1, p);
}
p = second;
while (p != NULL)
{
        push (&stk2, p)
}
while (stackTop(stk1) == stackTop(stk2))
{
        p = pop(&stk1);
        pop(&stk2);
}
cout << p->data;
```

intersecting point