Linked list

Problem in Array

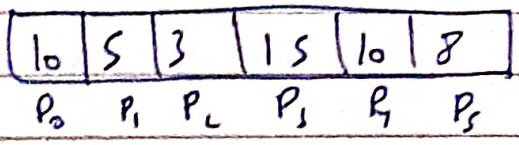
① Either size is fixed and pre-allocated (in both fixed & variable sized arrays), and the worst case insertion at the end is $O(n)$.
b/z when vector is full it double its size & copy the previous element before inserting new element.

② Insertion in the middle (or beginning) is costly.

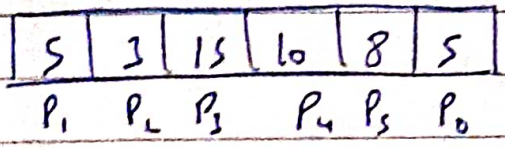
③ Deletion in the middle (or beginning) is costly.

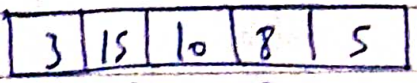
④ Implementation of DS like queue & deque is complex with arrays.

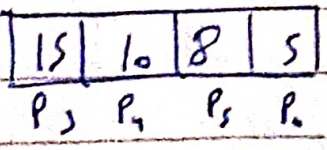
⑤ Implement Round Robin Scheduling is difficult



token time = 5







Difficult to implement using Array as size is changing & element is also moved at last.

Arrow Operator • allow to access elements in Structure & Union
• used with a pointer variable pointing to ——

(6) Given a sequence of items, whenever we see an item x in the sequence, we need to replace it with s instances of another item y.

I/P: d e a x g x x b y
O/P: d e a y y y y y y g x y y y y b y

Array require more traversal & extra array ~~with it we can~~

(7) We have multiple sorted sequences and we need merge them frequently.

$$\text{sequence}[] = \begin{cases} (5, 10, 15, 20) \\ (1, 12, 18) \\ (3, 30, 40) \\ (100, 200) \end{cases}$$

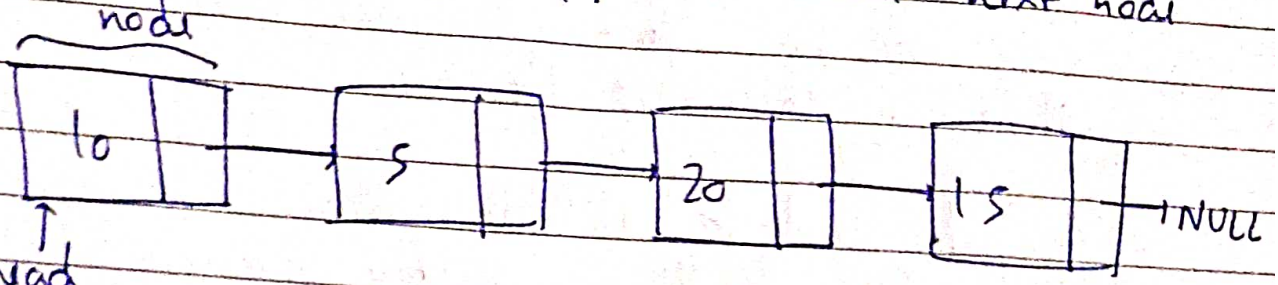
merge in Array require extra auxillary array while in LL no extra space required.

$$\text{merge}(0,1) \rightarrow \text{sequence}[] = \begin{cases} (1, 5, 10, 12, 15, 18, 20) \\ (3, 30, 40) \\ (100, 200) \end{cases}$$

* When we store these sequence using LL then merge operation don't require extra space.

LL

- linear DS
- elements are in sequence but at continuous location
- every node stores reference/pointer to the next node



Advantages of LL
- Insert in middle / beginning efficiently
- Delete

- If there are n elements we need n nodes
- No need to pre-allocate the space

LL implementation

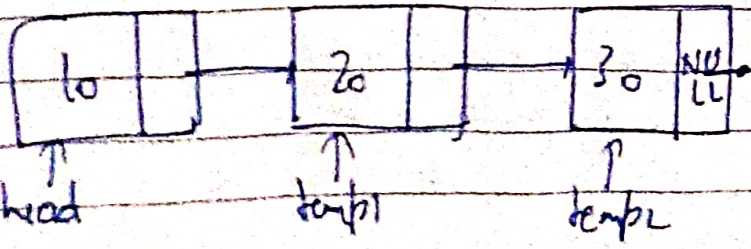
- every node takes 8 bytes in total (4 for data, 4 for address)
- change in size (depend on compiler)

```
struct Node
{   int data;
    Node *next;
    Node(int x)
    {   x = data; data = x;
        next = NULL;
    }
};
```

self referential structure

constructor

```
int main()
{
    Node *head = new Node(10);
    Node *temp1 = new Node(20);
    Node *temp2 = new Node(30);
    head->next = temp1;
    temp1->next = temp2;
}
```

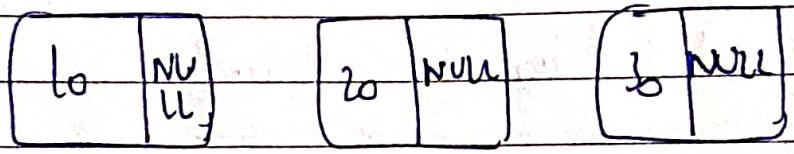


- we use structure b/z we need to store more than one datatype of values at one place.

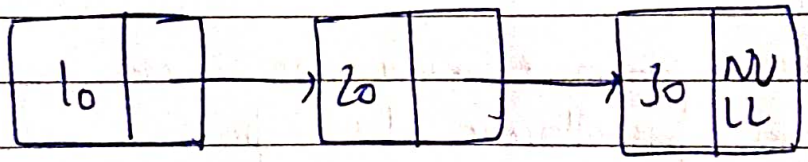
- Node *next → self referential structure.
  every node has reference of next node which is type Node ∴ *next is of type Node

- Constructor used to initialize a Node.
- With class length of structure increases.

Initial



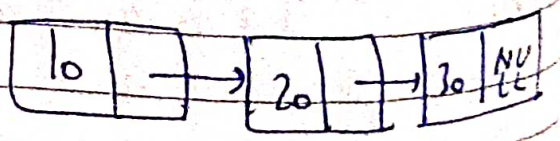
after linking




```
int main()
{
    Node *head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
}
```

→ shorter implementation

Traversing a Singly LL:




```
void f(Node *head)
{
    Node *curr = head;
    while(curr != NULL)
    {   cout << (curr->data) << " ";
        curr = curr->next;
```

o/p → 10 20 30

Node Creation

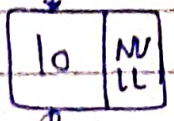
Node *curr;
Simply make a pointer
Node *curr = head;
Assign a node to pointer

Node *curr = new Node(10);
create a pointer & assign a new node with data 10

```
int main()
{
    Node *head = new Node(10);
    printList(head);
    printList(head);
}
```

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

main head



printList head

both are different head

0|p → 10 10

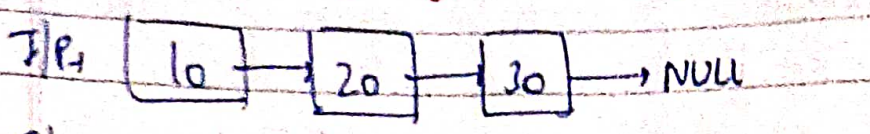
यहाँ head में change होगा तो main के head में कोई change नहीं होगा।

- In C++ when we pass pointer to a function then both head variable are different but points to same memory location.

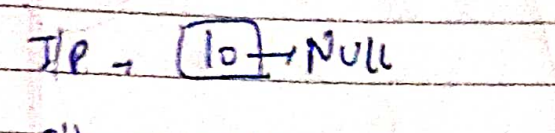
Recursive Traversal of LL

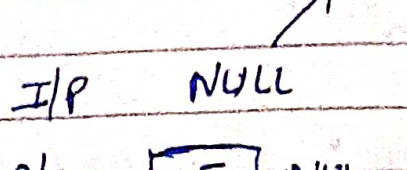
```
void f(Node *head)
{   if(head == NULL)
        return;
    cout << head->data << " ";
    f(head->next);
}
```

Insert at Beginning of Singly Linked list : O(1)



head points to NULL initially

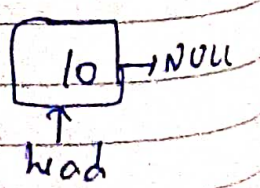


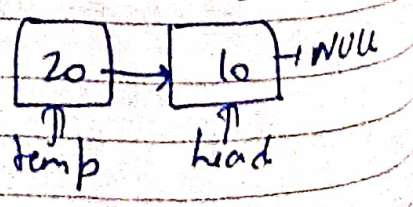


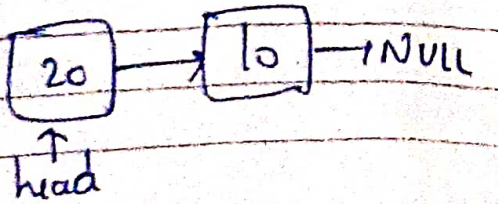

```
struct Node
{   int data;
    Node *next;
    Node(int x)
    {   data=x;
        next = NULL;
    }
};
int main()
{     Node *head = NULL;
      head = insertBegin(head, 10);
      head = insertBegin(head, 20);
}
Node *insertBegin (Node *head, int x)
{
      Node *temp = new Node(x);
      temp->next = head;
      return temp;
}
```

head1 भी लिख के कोई problem नहीं

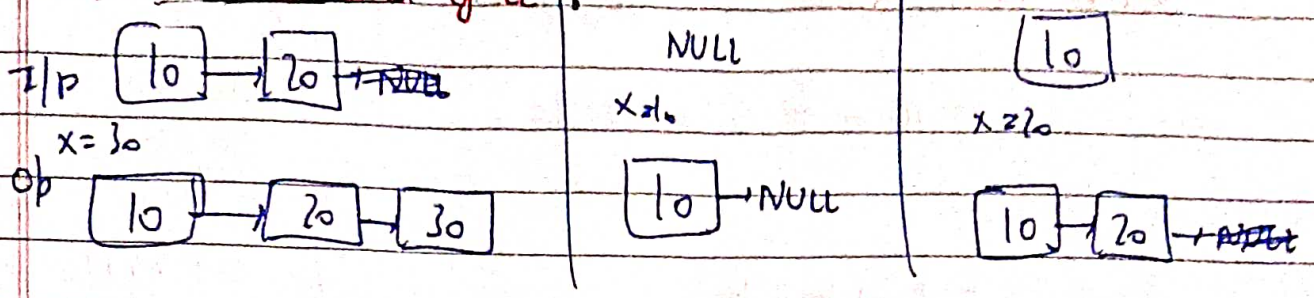
head = NULL





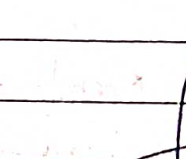


Insertion at End of LL :

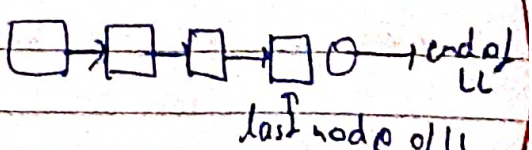



```
int main()
{
    Node *head = NULL;
    head = insertEnd(head, 10);
    head = insertEnd(head, 20);
    head = insertEnd(head, 30);
}
Node *insertEnd(Node *head, int x)
{
    Node *temp = new Node(x);
    if(head == NULL)
        return temp;
    Node *curr = head;
    while(curr->next != NULL)
        curr = curr->next;
    curr->next = temp;
    return head;
}
```

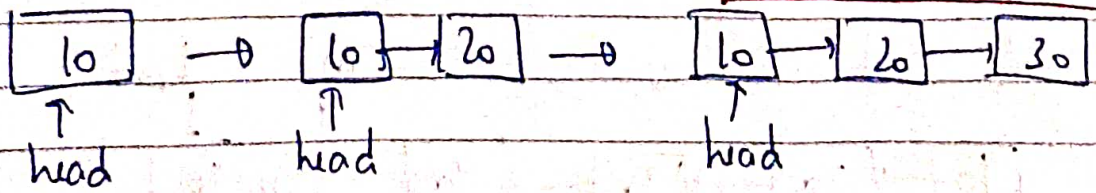
NULL का अर्थ है कि last node of LL पहुंचने के लिए



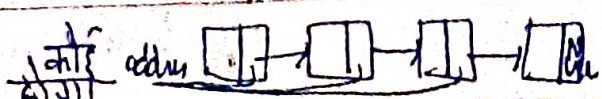
while(curr->next != NULL)
LL के end पर आ रहे



head = NULL

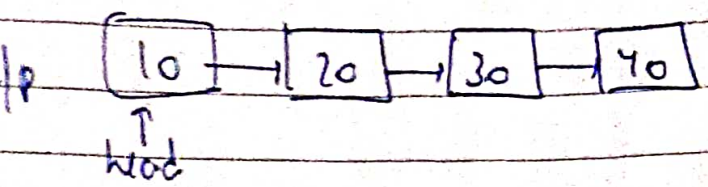


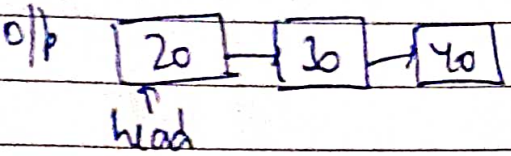
आखरी वाले Node का next होगा NULL को Point करता है, और तब NULL exist नहीं करता.

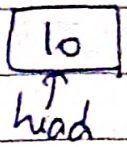


Delete First Node in Singly LL

O(1)







I/p head = NULL

O/p head = NULL

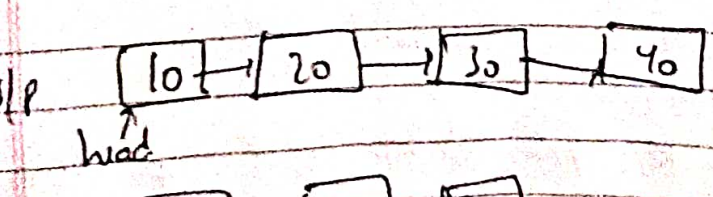
```
Node * deleteFirst(Node *head)
{
    if(head == NULL)
        return NULL;

    Node *temp = head;
    head = head->next;

    delete temp;
    return head;
}
```

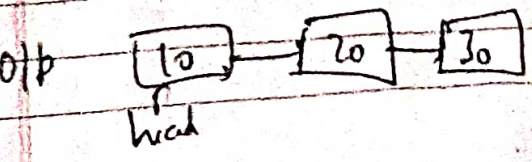
OR

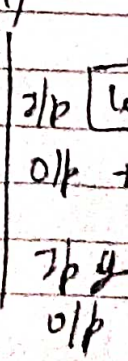
```
Node * temp = head->next;
delete head;
return temp;
```

** In C++ we have to deallocate the memory of unwanted node.

Delete last Node of Singly LL O(n)





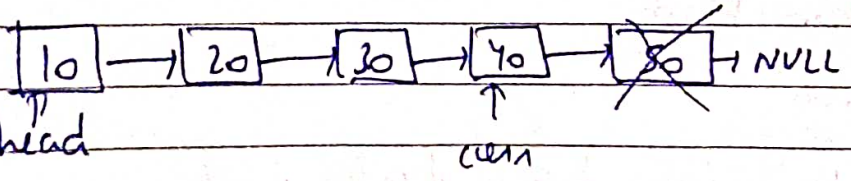


NULL → next

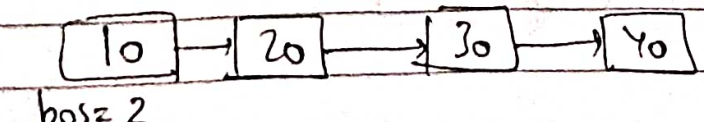
↓

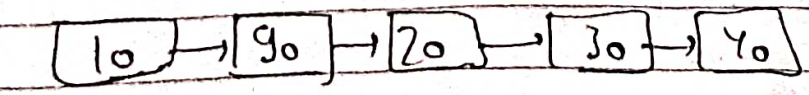
Segmentation Fault

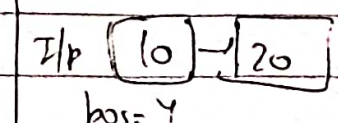
```
Node *delNode(Node *head)
{
    if(head==NULL) return NULL;
    if(head->next==NULL)
    {
        delete head;
        return NULL;
    }
    Node *curr = head;
    while(curr->next->next!=NULL)
        curr = curr->next;

    delete (curr->next);    // first delete then assign NULL
    curr->next = NULL;
    return head;
}
```

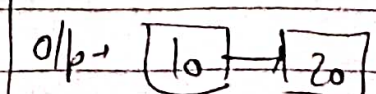


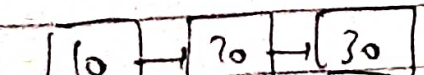
→ Insert at Given Position in LL :

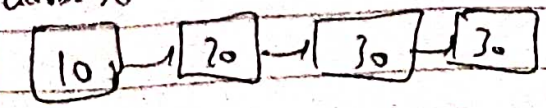








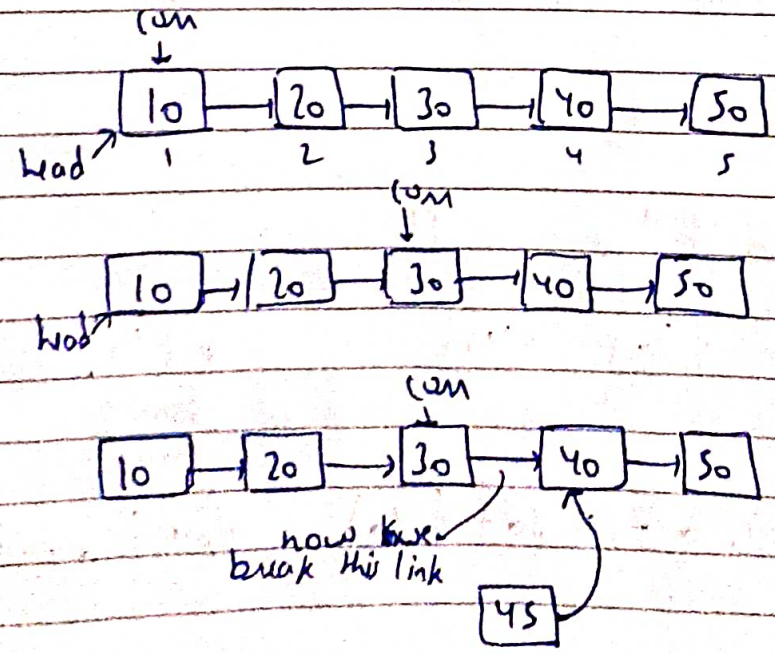




NULL → next : Segmentation fault

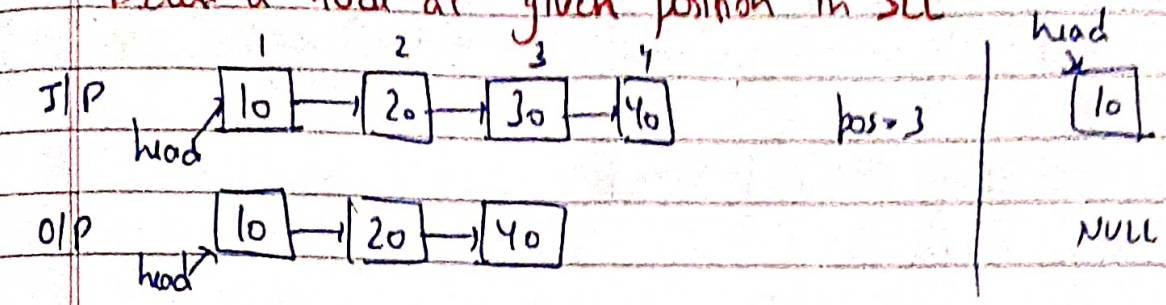
```
Node *insertPos (Node *head, int pos, int data)
{
    Node *temp = new Node(data);
    if (pos==1)
    {
        temp->next = head;
        return temp;
    }
    Node *curr = head;
    for(int i=1; i<=(pos-2) && curr!=NULL ; i++)
        curr = curr->next;
*   if(curr==NULL)
        return head;
    temp->next = curr->next;
    curr->next = temp;
    return head;
}
```

* when pos > (size+1) then we simply return head.



We can also use free() to return memory of deleted Node. eg; free(temp);

Delete a node at given position in SLL




```
Node *deleteNode(Node *head, int x)
{
    Node *temp = new Node(10);
    if(x==1)
    {   temp=head;
        head = head->next;
        delete temp; // don't waste to write
        return head;
    }
    Node *curr = head;
    for(int i=1; i<=(x-2) && curr!=NULL; i++)
        curr = curr->next;
    if(curr==NULL)
        return head;
    temp->next = curr->next->next;
    delete curr->next; //
    curr->next = temp->next;
    delete temp; //
    return head;
}
```

→ any random value.

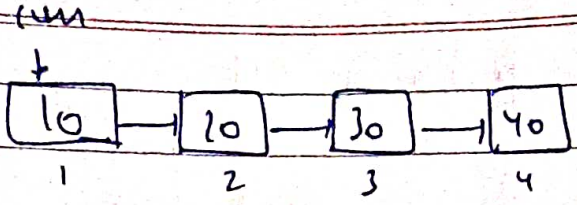
```
if(x==1)
{   head=head->next;
    return head;
}
Node *curr=head;
for(int i=1; i<=(x-2); i++)
    curr = curr->next;
if(curr->next==NULL)
    return head;
temp->next =(curr->next->next);
curr->next = temp->next;
return head;
```

if we delete x > size. Segmentation fault

अगर position > size है तो curr - last node पे रुक जाएगा जिसपे (curr->next==NULL) valid होगा।

for(i=1 to 3)

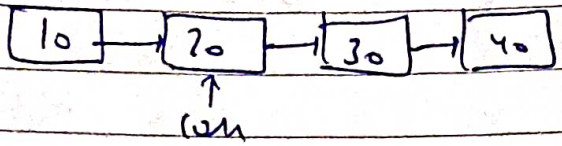
Initial LL



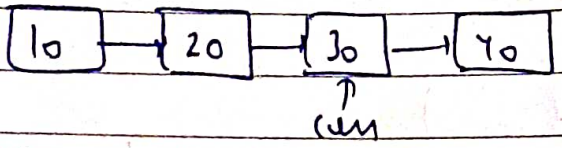
x = 5
delete 5th position which is invalid

i=0

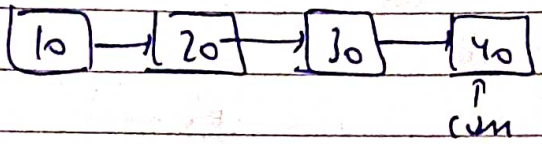
i=1



i=2



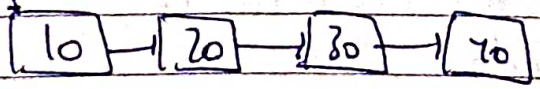
i=3



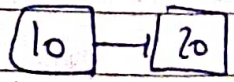
Now (curr->next==NULL)
return head;

→ Search in LL

a) Iterative



x=20
o/p → 2



x = 30
o/p → -1

```
int search(Node *head, int x)
{   int pos=1;
    Node *curr = head;
    while(curr!=NULL)
    {   if(curr->data==x)
            return pos;
        else
        {   pos++;
            curr = curr->next;
        }
    }
    return -1;
}
```

★★

Video देखो

b) Recursive

```
int search(Node *head, int x)
{
    if(head->data == x)
        return 1;
    if (head->next == NULL)
        return -1;
int res = search(head->next, x);
if(res == -1)
    return -1;
else
    return (res+1);
}
```

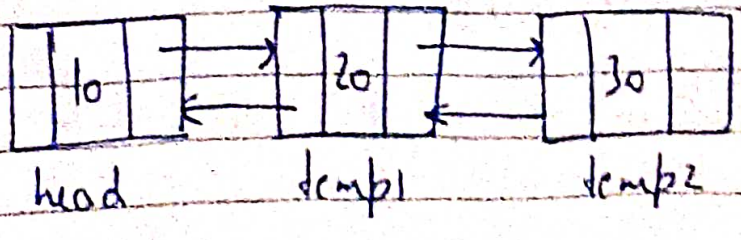
By me →

```
if (head == NULL) return -1;
if (head->data == x)
    return 1;
else
{
int res = search(head->next, x);
if(res==-1) return -1;
else return res+1;
}
```

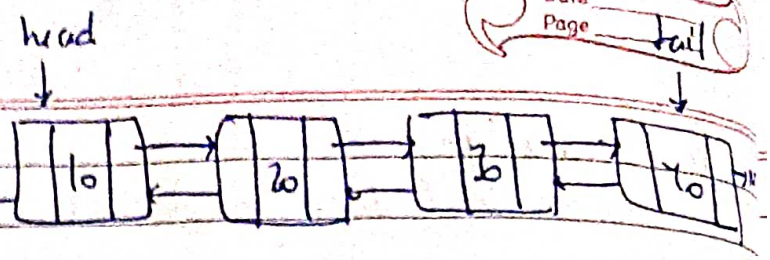
→ **Doubly linked list:**

```
struct Node
{   int data;
    Node * prev;
    Node * next;
    Node(int x)
    {   data = x;
        prev = NULL;
        next = NULL;
    }
};
```

```
int main()
{   Node *head = new Node(10);
    Node *temp1 = new Node(20);
    Node *temp2 = new Node(30);
    head->next = temp1;
    temp1->prev = head;
    temp1->next = temp2;
    temp2->prev = temp1;
}
```



Singly vs Doubly LL :



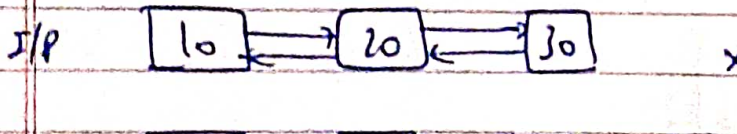
Advantages

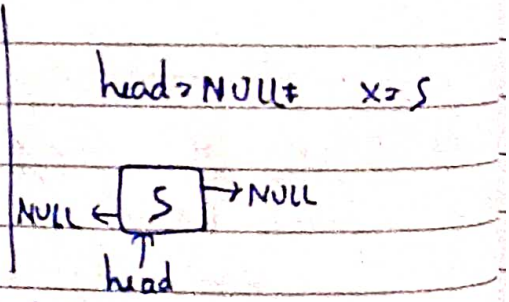
- Can be traversed in both directions
- Delete a node in O(1) if pointer to it is given.
- Insert/Delete before a given node
- _____ from both ends in O(1) time by maintaining tail.

Disadvantages

- Extra space for prev
- Code becomes more complex.

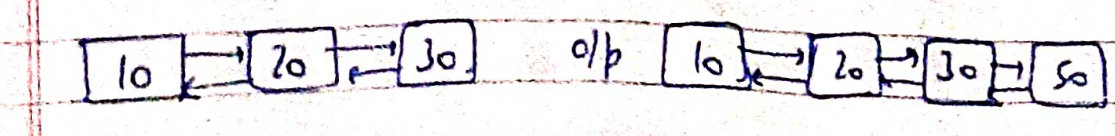
→ Insert at Beginning of DLL

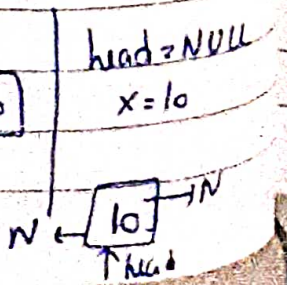





```
Node *insertBegin(Node *head, int data)
{
    Node *temp = new Node(data);
    temp->next = head;
    if(head!=NULL)
        head->prev = temp;
    return temp;
}
```

→ Insert at End of DLL :





```
Node *insertEnd(Node *head, int data)
{
    Node *temp = new Node(data);
    if(head == NULL)
        return temp;
    Node *curr = head;
    while(curr->next != NULL)
        curr = curr->next;
    curr->next = temp;
    temp->prev = curr;
    return head;
}
```

→ Delete head of DLL

I/P: 10 ⇄ 20 ⇄ 30

O/P: 20 ⇄ 30

I/P: 10	head = NULL
O/P: head = NULL	head = NULL

same

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

this 2 line only needed b/c C++ don't automatically return memory

→ Delete last node of DLL

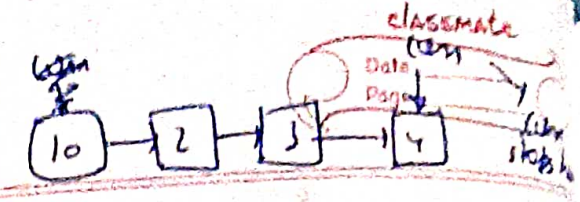
I/P: 10 ⇄ 20 ⇄ 30

I/P: 10 ⇄ 20

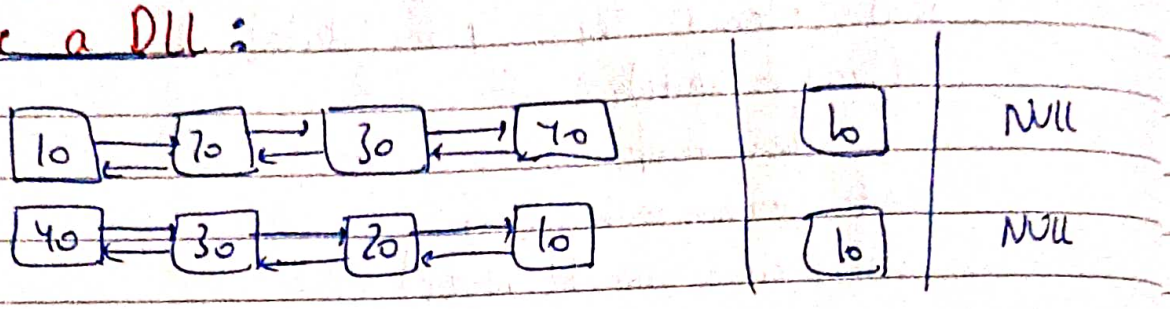
same

```
Node *delLast(Node *head)
{
    Node *curr = head;
    while(curr->next != NULL)
        curr = curr->next;
    curr->prev->next = NULL;
    delete curr;
    return head;
}
```

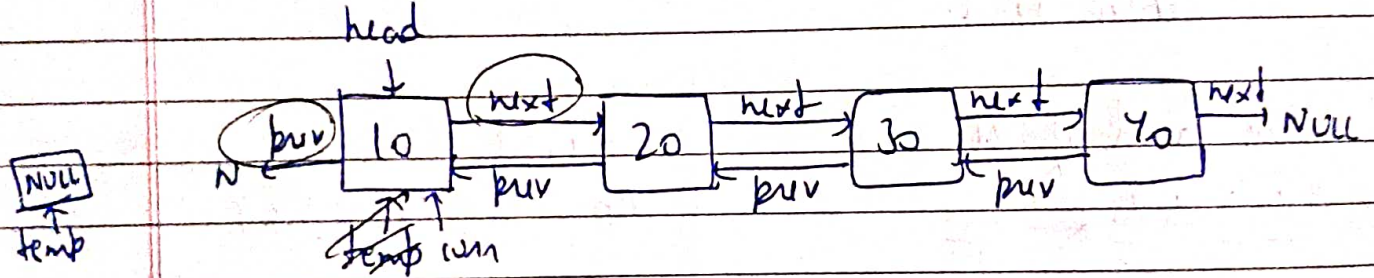
last node को पहुंचने के लिए
while(curr->next!=NULL)
curr = curr->next;



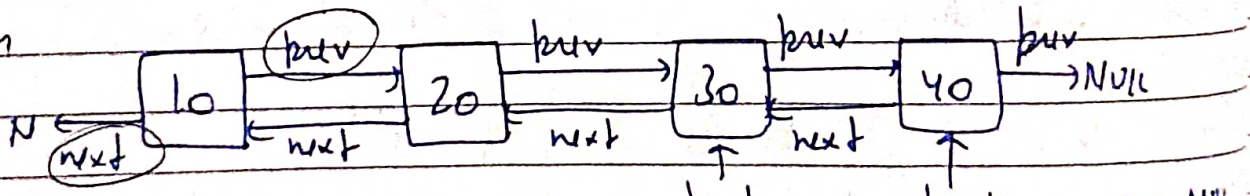
→ Reverse a DLL :



Logic : we swap the prev & next of every node
- And last node becomes new head



after situation is complete



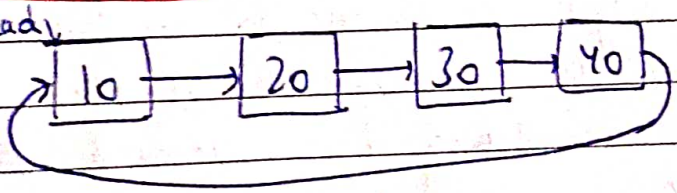
★

```
Node *reverseDLL(Node *head)
{   if(head==NULL || head->next==NULL) return head;
    Node *temp=NULL, *curr=head;
    while(curr!=NULL)
    {
        temp = curr->prev;
        curr->prev = curr->next;
        curr->next = temp;
        curr = curr->prev;    // since after swap next & prev are
    }                         swaped :. curr = curr->prev
                              curr = curr->next.
    return temp->prev;
}
```

Recursive

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

Circular LL :




```
struct Node
{
    int data;
    Node *next;
    Node(int x)
    {
        data = x;
        next = NULL;
    }
};
```

```
int main()
{
    Node *head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
head->next->next->next->next = head;
}
```

Circular LL

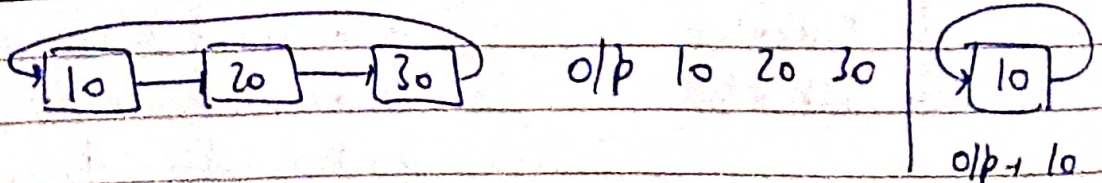
→ Advantage:
- We can traverse the whole list from any node.
- Implement of algorithm like Round Robin (it is CPU scheduling algo) is easy.
- We can insert at the beginning & end by just maintaining one tail reference/pointer.

→ Disadvantage
- Implementation of operations become complex.

Round Robin:
- We traverse through all the process, we maintain the Queue of the process & traverse in circular manner.
- CPU is allocated to these process one by one.
- When process is completed it is removed from Queue for which we do delete operation
- Also we update the time-slot.

Achieved by Circular LL

→ Circular LL Traversal

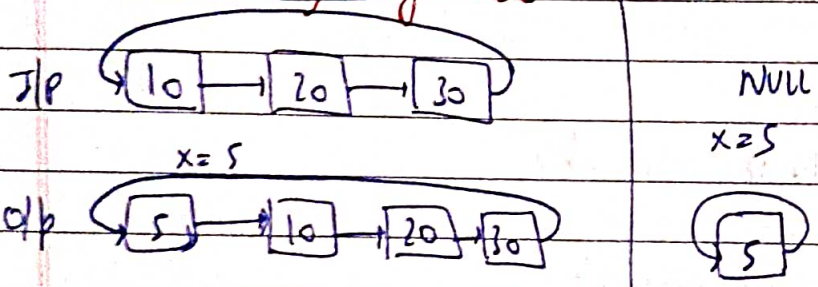


(I)
```
void print(Node *head)
{  if(head==NULL) return;
   cout<<head->data<<" ";
   (for(Node *p=head->next; p!=head; p=p->next)
        cout<<(p->data)<<" ";
```
When only one node is there, this loop is not execute }

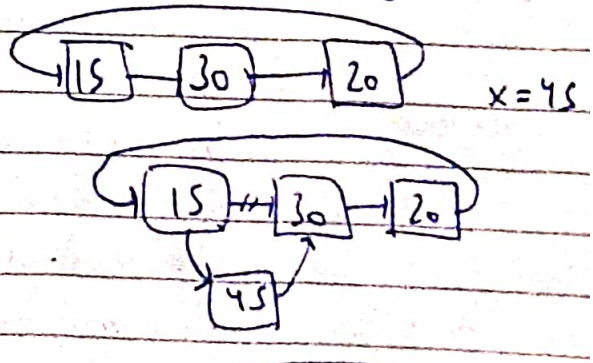
(II) do-while

```
void print(Node *head)
{   if(head==NULL) return;
    Node *p = head;
    do
    {   cout<<(p->data)<<" ";
        p = p->next;
    }while(p!=head);
}
```

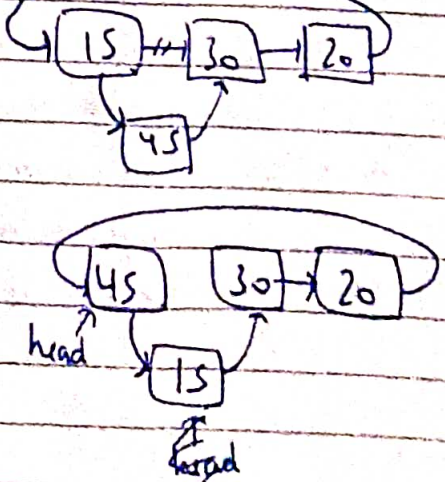
→ Insert at Begin of CLL



a) Naive O(n) iterate whole CLL & reach last node (say curr)
& change curr->next = temp;
temp->next = head;
return temp;



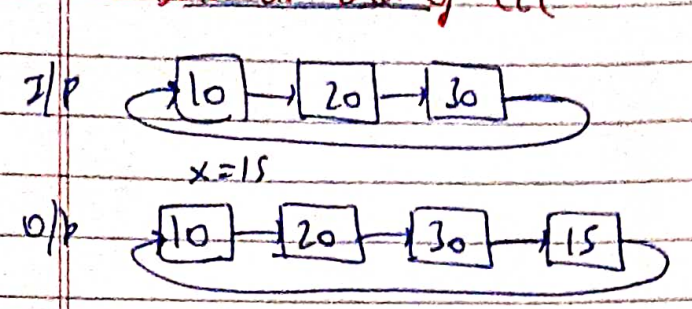
b) Node *insertBegin(Node *head, int x)
{ Node *temp = new Node(x);
if(head==NULL)
{ temp->next = temp; return temp; }

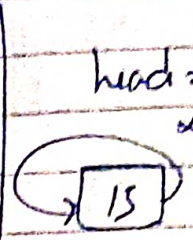



```
    temp->next = head->next;
    head->next = temp;
    int t = head->data;
    head->data = temp->data;
    temp->data = t;
    return head;
}
```

→ does swaping value of head & temp

→ Insert at end of CLL





a) Naive O(n): Iterate till curr to the end (here 30) & make

```
curr->next = temp;
temp->next = head;
return head;
```

Remember to take curr & edge case when head == NULL

b) O(1) swap

```
Node *insertEnd(Node *head, int x)
{
    Node *temp = new Node(x);
    if(head == NULL)
    {   temp->next = temp;
        return temp;
    }
```

Insert temp after head:

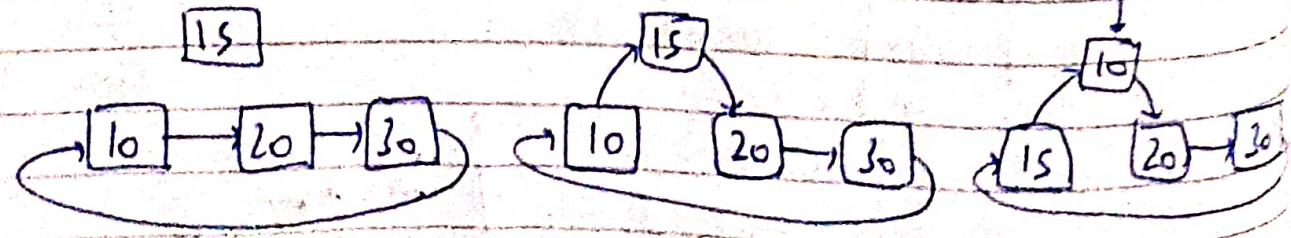
```
    temp->next = head->next;
    head->next = temp;
```

Swapping:

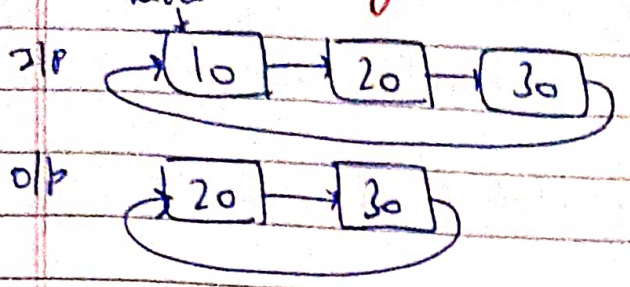
```
    int t = head->data;
    head->data = temp->data;
    temp->data = t;
```

new head:

```
    return temp;
}
```



Delete head of CLL :



head = NULL

NULL

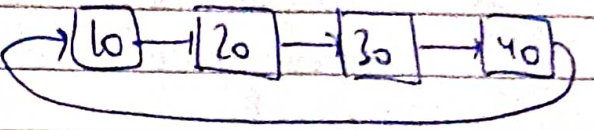
head == NULL

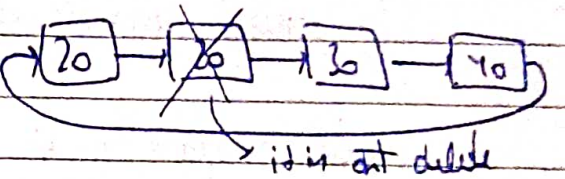
a) Naive O(n)

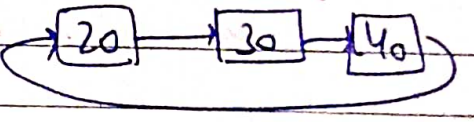
```
Node *f(Node *head)
{
    if(head==NULL)
        return NULL;
    if(head->next==head)
    {   delete head;
        return NULL;
    }
    Node *curr = head;
    while(curr->next!=NULL)
      curr = curr->next;
    curr->next = head->next;
    delete head;
    return (curr->next);
}
```

this is not required in Java also if you don't write in C++ code works but space is not released.

b) efficient O(1)



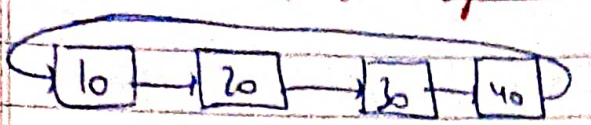





```
Node *f(Node *head)
{   if(head==NULL) return NULL;
    if(head->next==head)
    {   delete head; return NULL;
    }
    head->data = head->next->data;
    Node *temp = head->next;
    head->next = head->next->next;
    delete temp;
    return head;
}
```

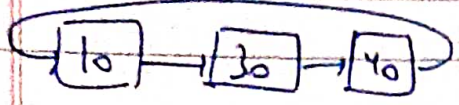
→ Delete K^{th} Node from a CLL

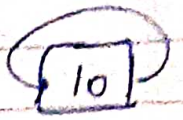
I/P



k = 2

O/P





K=1

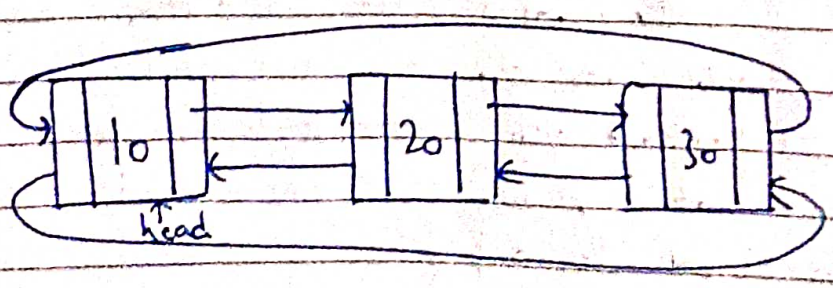
head = NULL

```
Node *deleteKth(Node *head, int k)
{   if(head==NULL) return NULL;
    if(k==1)
        return deleteHead(head);
    Node *curr=head;
    for(int i=0;i<k-2;i++)
        curr=curr->next;

    Node *temp = curr->next;
    curr->next = curr->next->next;
    delete temp;
    return head;
}
```

→ when a first node is deleted

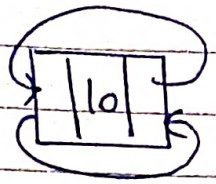
→ previous code

→ Circular Doubly LL



- Previous of head is last node
- Next of last node is first —

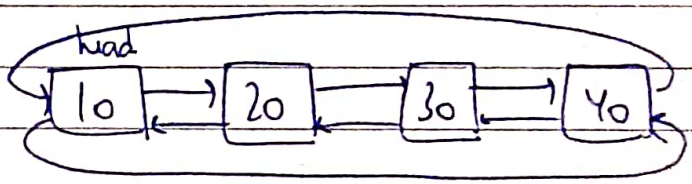
- empty circular DLL → head = NULL
- Sing node



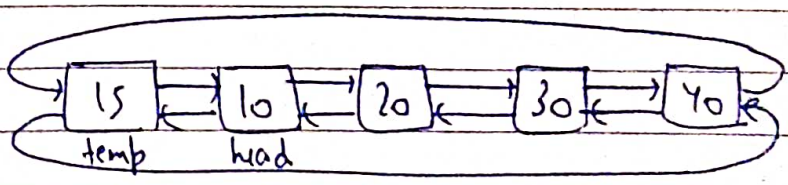
- All advantages of circular & DLL valid
- Access last node in O(1) time without maintaining tail.

→ Insert at beginning of CDLL

```
Node *temp = new Node(x);
if(head == NULL)
{   temp -> next = temp;
    temp -> prev = temp;
    return temp;
}
temp -> prev = head -> prev;
temp -> next = head;
head -> prev -> next = temp;
head -> prev = temp;
return temp;
```



x = 15



→ Insertion at end of CDLL

Same code as above
Only change is return head;