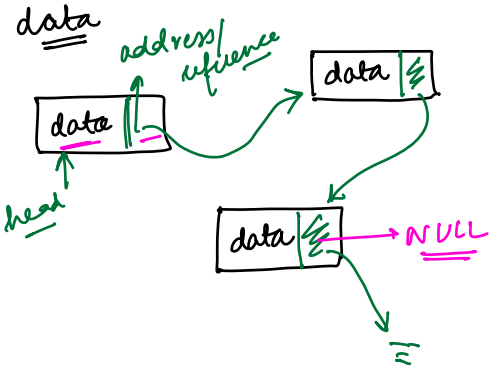
Linked list

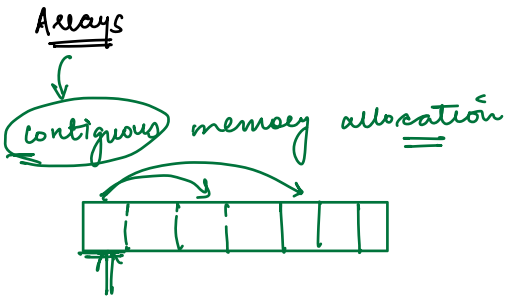
↓

linear DS

non-contiguous



Arrays

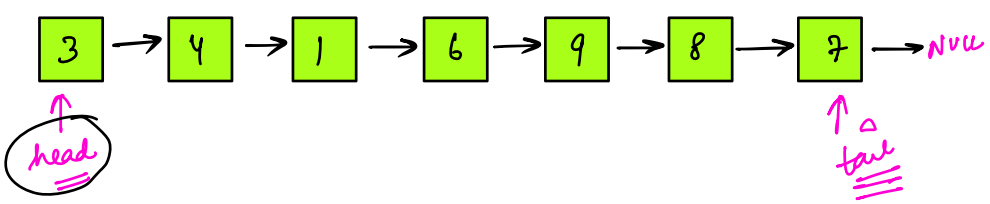


Random access - O(1)

insert ? deletion ? , size

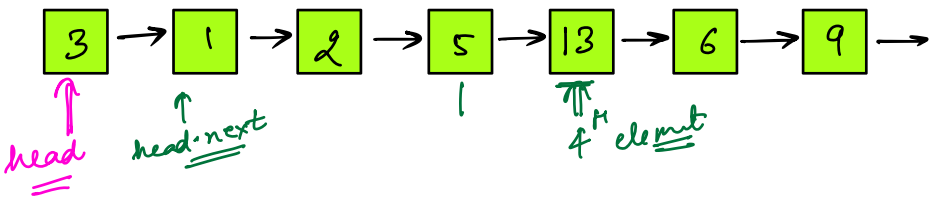
```
struct Node {
    int data;
    Node * next;
};
```

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



- LL (head), find K^{th} element in LL. arr[k]

0-based



y = 3

x = y

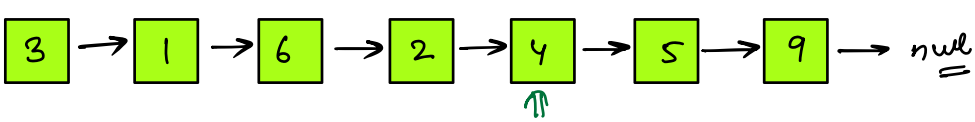
```
Node temp = head;
while (temp != NULL && K > 0)
{
    // head = head.next;
    temp = temp.next;
    k--;
}
if (temp != null)
    return temp.data;
return -1;
```

Never update your head until configuration changes

T.C: O(n)

- search on element K in LL

array → linear search O(n)
array → Binary search

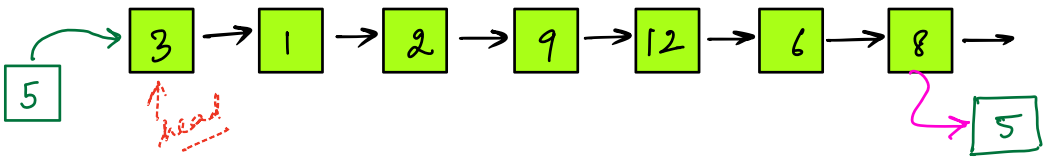


search 4

```
temp = head;
while (temp != NULL) {
    if (temp.data == k)
        return temp;
    temp = temp.next;
}
return NULL;
```

T.C: O(N)

- Insertion in a LL:



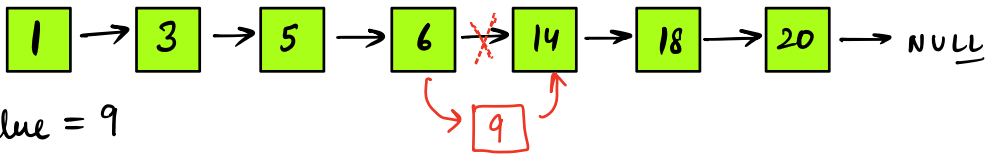
- At head (value K) k=5

```
Node x = new Node(k);          // x->next = NULL
x.next = head;
head = x
```

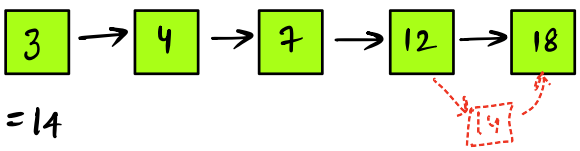
- At tail (K) Node x = new Node(k); head = NULL

```
if( head==NULL) { head=x; return x; }

temp= head;
while ( temp.next !=null )
{
      temp = temp->next;
}
  temp.next = x;
```

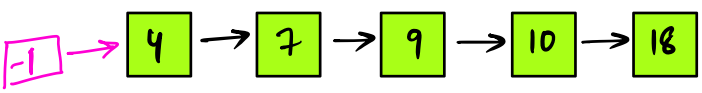


value = 9



value = 14

Insert such that the LL remains sorted in C

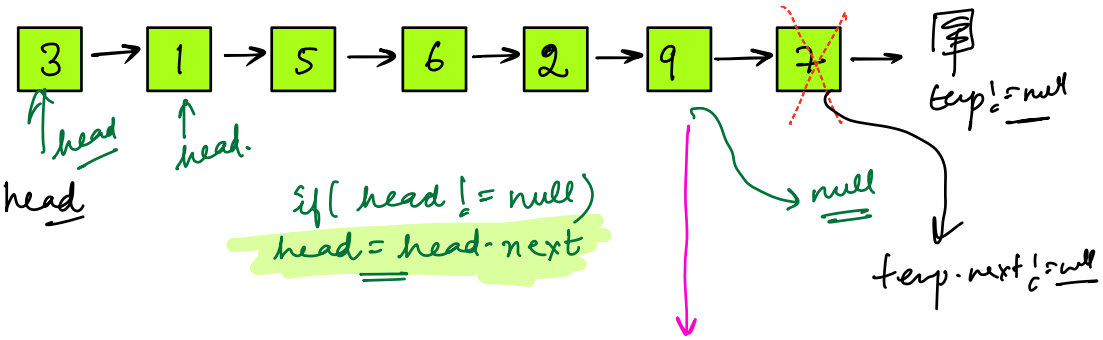


value = -1

```
Node x = new Node(k);
if( head == NULL) { return x;}

if( head.data >= k) { // insert at the head }
     temp = head;
 while ( temp.next != NULL && temp.next.data < k)
{
          temp = temp.next;
}
          x.next = temp.next
          temp.next = x;
```

- Delete a LL node.



- At head

```
if( head != null)
head = head.next
```

- At tail

// return head of LL

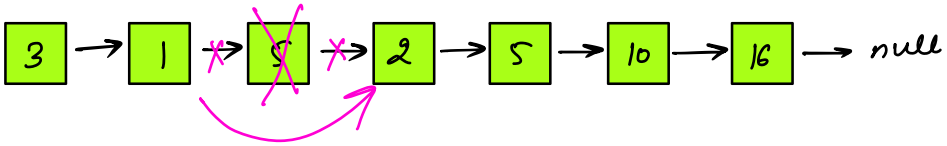
temp.next.next = null

↓

Last 2nd node

```
if ( head == null || head.next == NULL) return NULL;
    temp = head
while ( temp.next.next != null)
{
        temp = temp.next;
}
        temp.next = NULL;
```

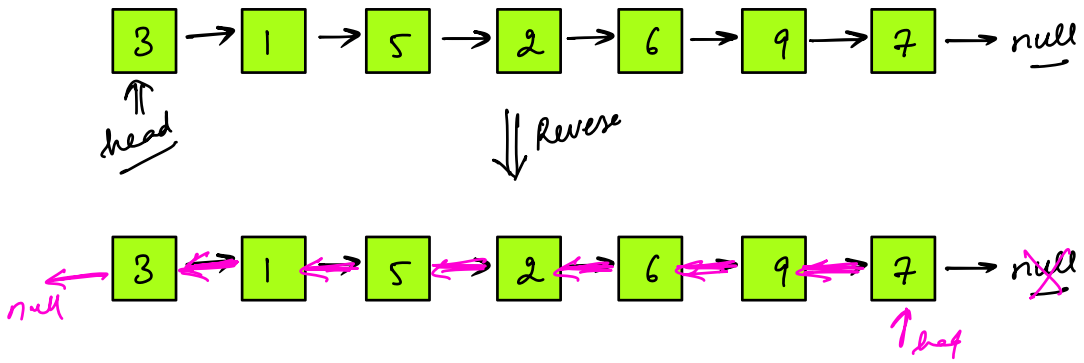
- delete first occurrence of element K & return final head of LL

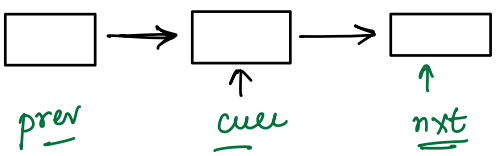


K = 5

```
if( head == NULL) return NULL;
if( head.data == k) { // delete head } head = head.next
temp = head   while ( temp . next != NULL)
{       if ( temp.next.data == k)
            {     temp.next = temp.next.next; return head; }
        temp = temp->next
}
```

Q Reverse the LL without using extra space.



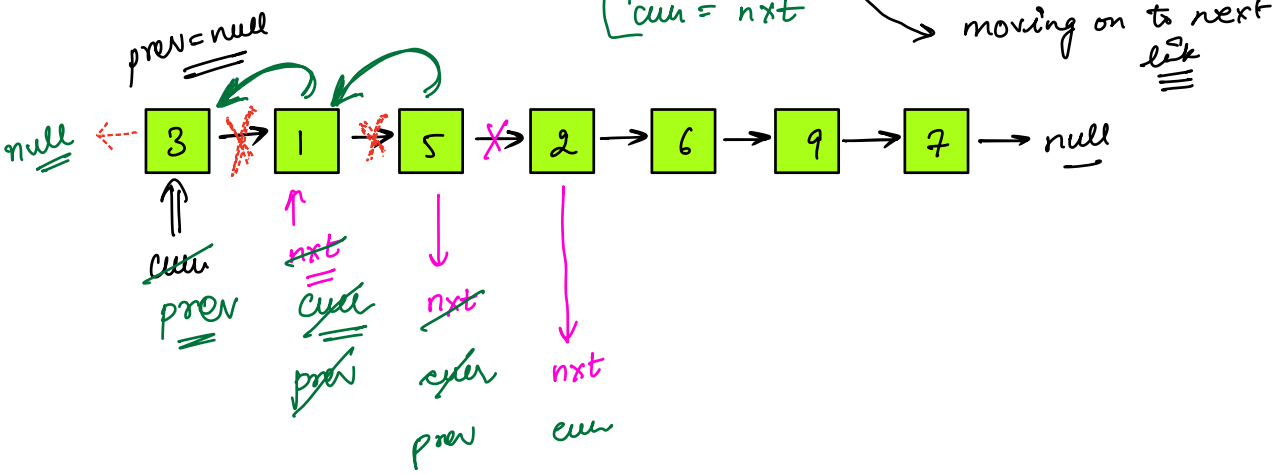


nxt = curr.next // store our next node so that we don't loose it

curr.next = prev — update the link to go backwards

prev = curr

curr = nxt — moving on to next link

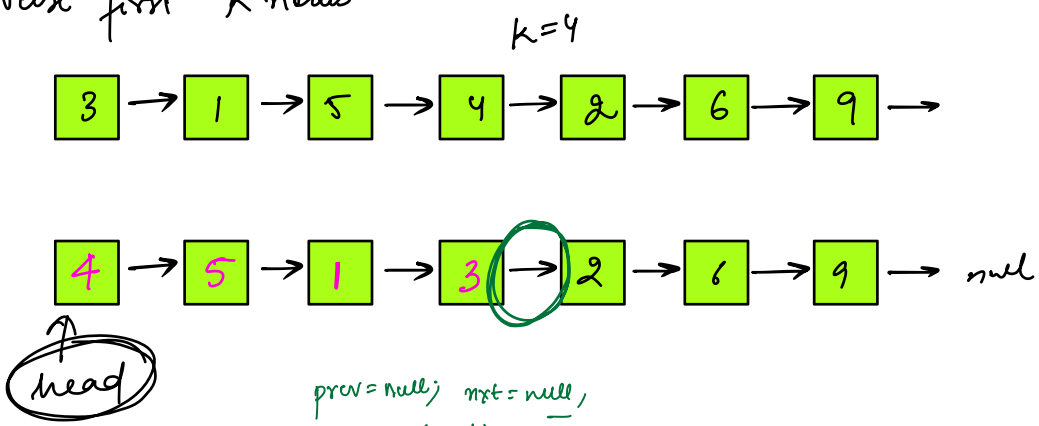


```
prev = null;  nxt = null;
    curr = head;

while( curr != null)
{
      nxt  =  curr.next
         curr.next = prev;
         prev = curr;
          curr = nxt;
}

return prev;        // prev is pointing to head
```

- reverse first k nodes



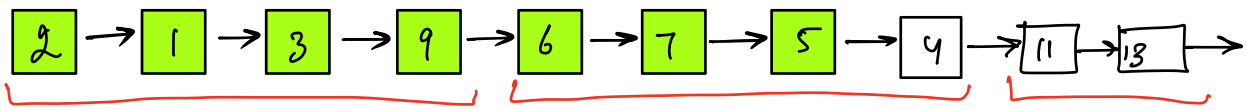

```
prev = null;  nxt = null;
    curr = head;

while( curr != null && K>0)
{
      nxt  =  curr.next
         curr.next = prev;
         prev = curr;
          curr = nxt;   K--;
}
head->next = curr;
return prev;        // prev is pointing to head
```

old head

Q Reverse every group of k size.

K = 4



9 → 3 → 1 → 2 → 4 → 5 → 7 → 6 → 13 → 11 → null

```
Node revk( Node head, int k)
{   if (head == NULL) return NULL;
    prev = null; nxt = null;
    curr = head;

    while( curr != null && k > 0)
    {
        nxt  = curr.next
        curr.next = prev;
        prev = curr;
        curr = nxt;   k--;
    }
    head -> next = revk( curr, k);
    return prev;      // prev is pointing to head
}
```

old head