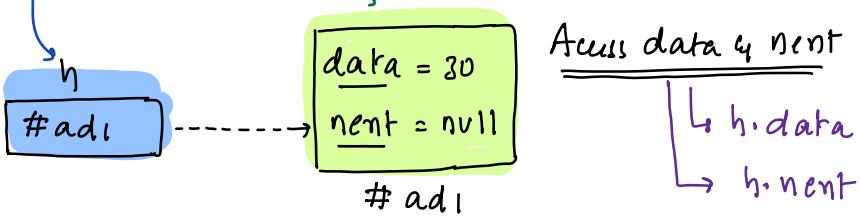
Todays Content:

- → Basic LinkedList ✓
- → Insert in a sorted linked list ✓
- → Delete k in a sorted linked list TODO
- → Reverse LinkedList ✓
  - a) Revern the first k nodes ✓
  - b) Revern all k groups ✓
- → Find the middle of linked list TODO

```
Class Node{ // in your language of choice
    int data
    Node next // object reference, can hold reference of Node object
    Node(int x){ // Constructor, used to initialize data members in a class
        data = x
        next = NULL
    }
}
```

```
Node h = new Node(30);
```

↳ // an object is created & h is object reference here



Access data & next

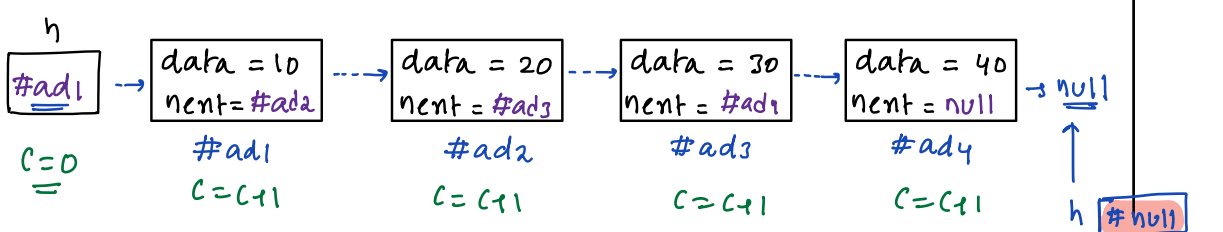
↳ h.data

↳ h.next

Note: h is just holding reference of object

print(h) : #ad1

Linked list:




```
int size(Node h){
    int c=0
    while(h != NULL){
        c=c+1
        h=h.next
    }
    return c;
}
```

Note: If we access h.data & h.next

a) h != NULL

: if we access h.next.data & h.next.next

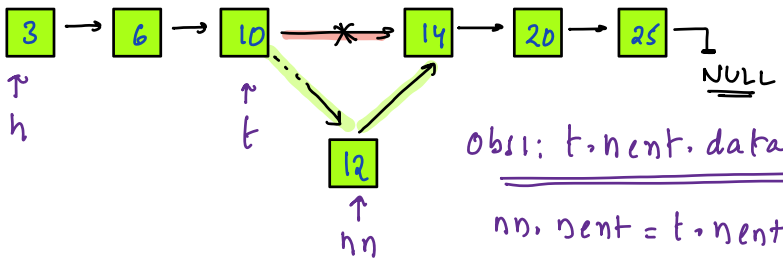
a) h != NULL

b) h.next != NULL

Problems:

1) Given head of a sorted linked, create & insert new node with x

Ex1: x = 12

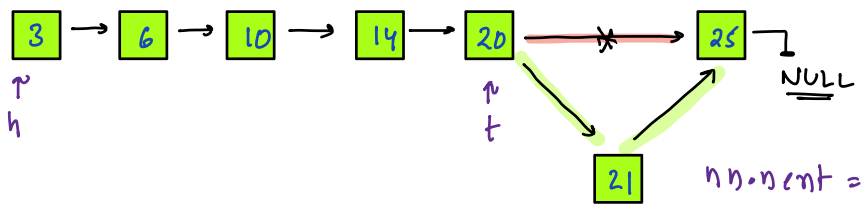


Obs1: t.next.data > 12

nn.next = t.next

t.next = nn

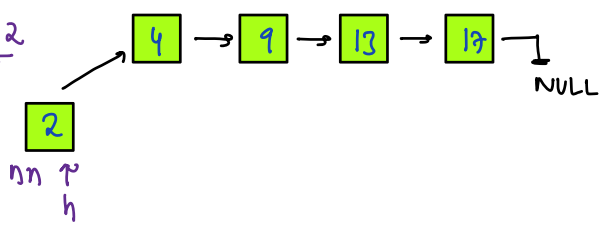
Ex2: x = 21



nn.next = t.next

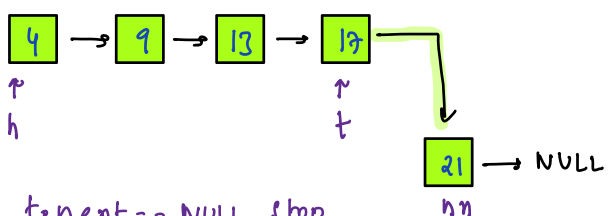
t.next = nn

Ex3: x = 2



#obs2: Insert at start: x < h.data : nn.next = h h = nn

Ex4: x = 21



#obs3: t.next == NULL stop

nn.next = t.next

t.next = nn

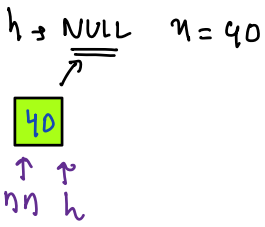
Since we might change head return head node of linked list

Node InsertSorted(Node h, int n){ TC: O(N) SC: O(1)

```
Node nn = new Node(n);
if (h==null || n < h.data){
    nn.next = h, h = nn; return h;
}
Node t = h
while (t.next != NULL && t.next.data < n){
    t = t.next
}
nn.next = t.next
t.next = nn
return h;
}
```

// if t.next == null we break & go to below line

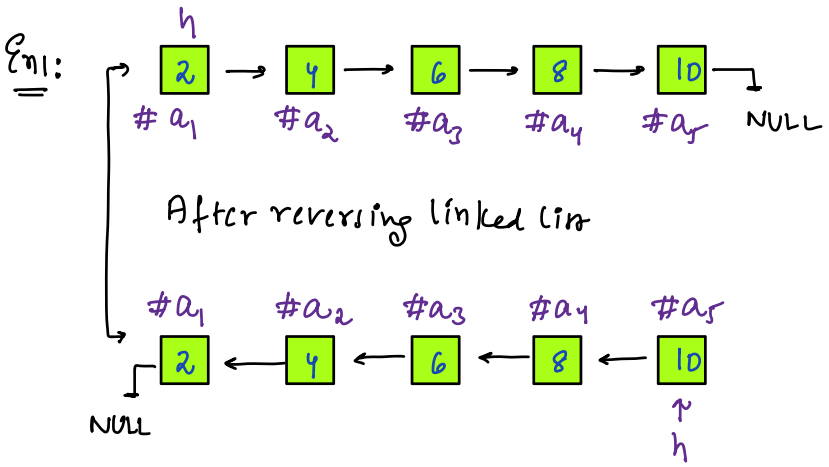
// insert node nn



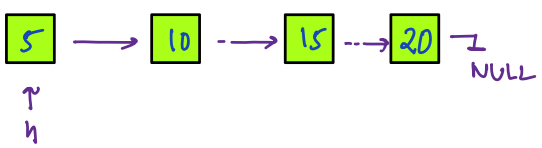
20] Given a head of linkedlist, revers linkedlist & return head

Note: No Extra Space & We cannot change val of node TC: O(N)

Eg1:

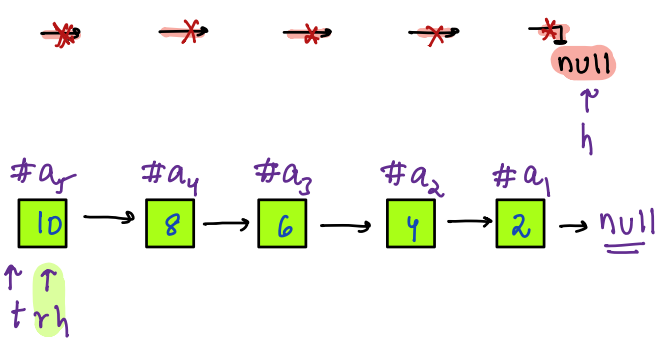


Idea: We inserted 20, 15, 10, 5 at start 1 by 1



// Obs: Inserting node at start will give revers

Eg:



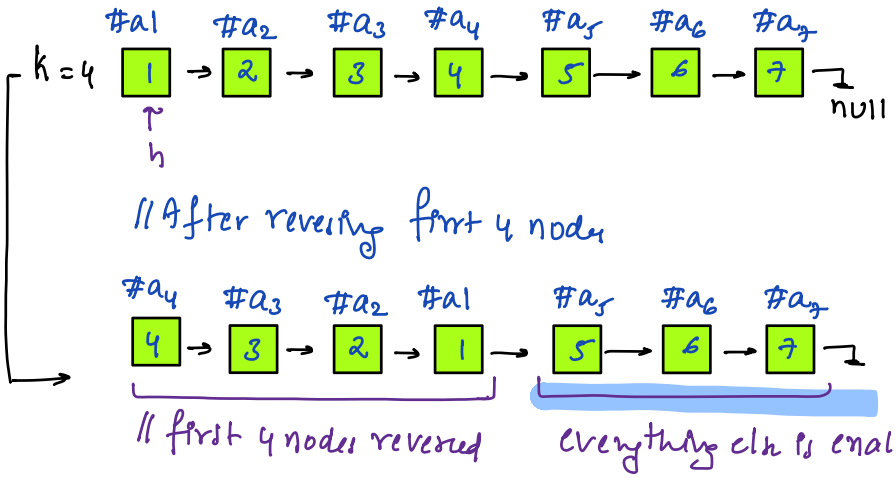
Idea: Isolate head node & add it at start of revers, repeat this process, unhill head == null

```
Node revers(Node h){ TC:O(N) SC:O(1)
    Node rh=null, t=h
    while(h!=NULL){
        t=h
        h=h.next
        t.next=null //not necessary
        t.next=rh
        rh=t
    }
    return rh;
    //rh is head node of reversed list
}
```

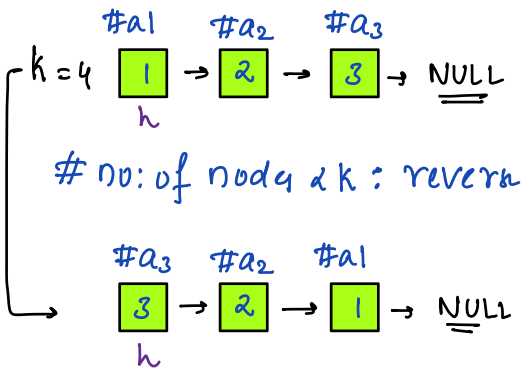
38) Given head of linked list, reverse first k nodes, given $k > 0$

Note: #no: of nodes $\geq k$, still reverse complete linked list

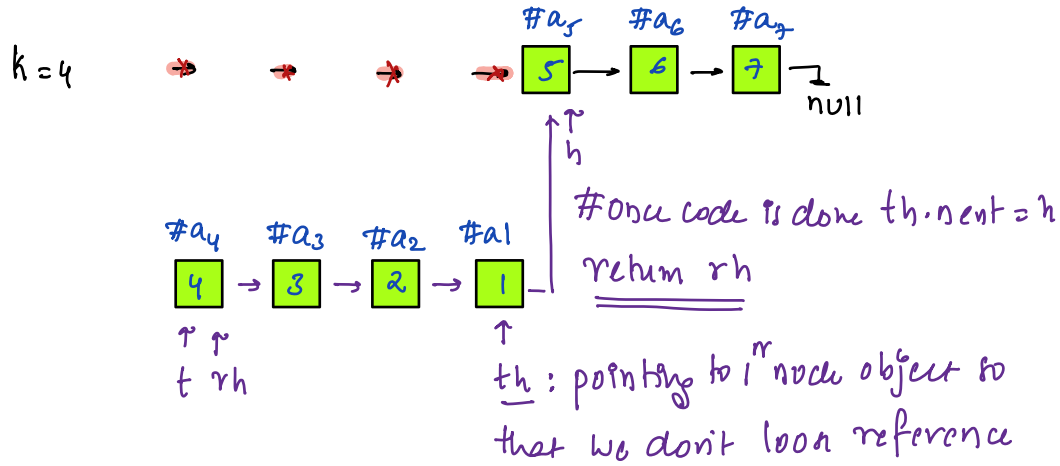
Ex1:



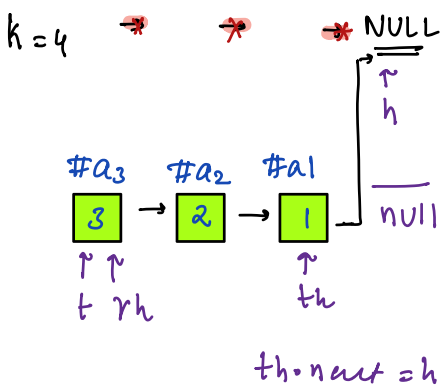
Ex2:



Trace:




```
Node reverse(Node h, int k){ TC: O(N)  SC: O(1)
    if(h == null){ return h;} // To avoid edge case th.next
    Node rh=null, t=h, th = h
    while(h != NULL && k>0){        → Even if # no: of node < k condition helps to resolve
        t=h
        h = h.next
        t.next=null //not necessary
        t.next=rh
        rh = t
        k = k-1
    }
    th.next = h // Edge Case if th = null
    return rh; //rh is head node of reversed list
}
```

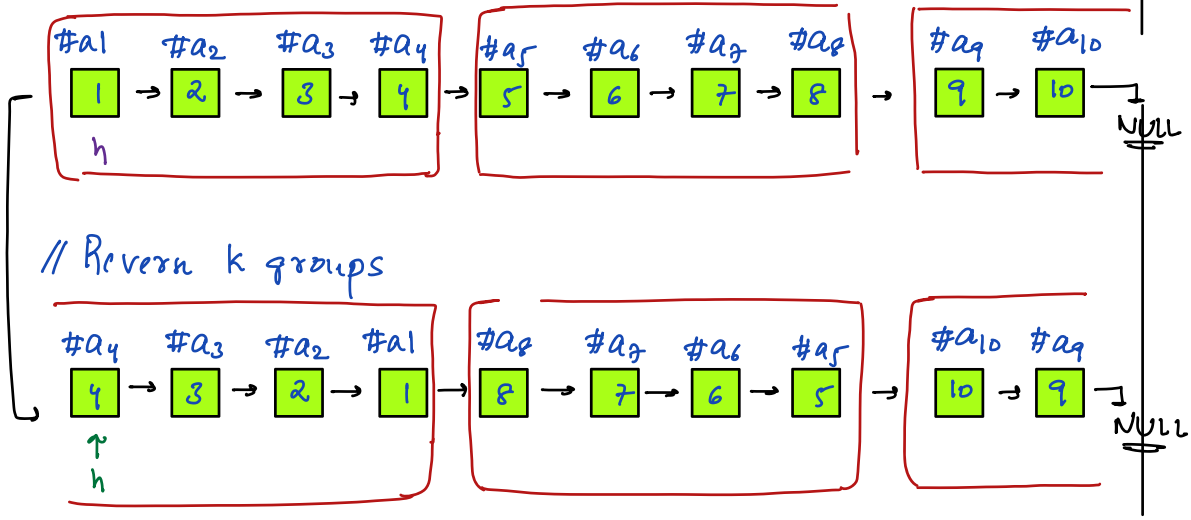


Q) Given head of linked list, reverse all groups of size: k > 0

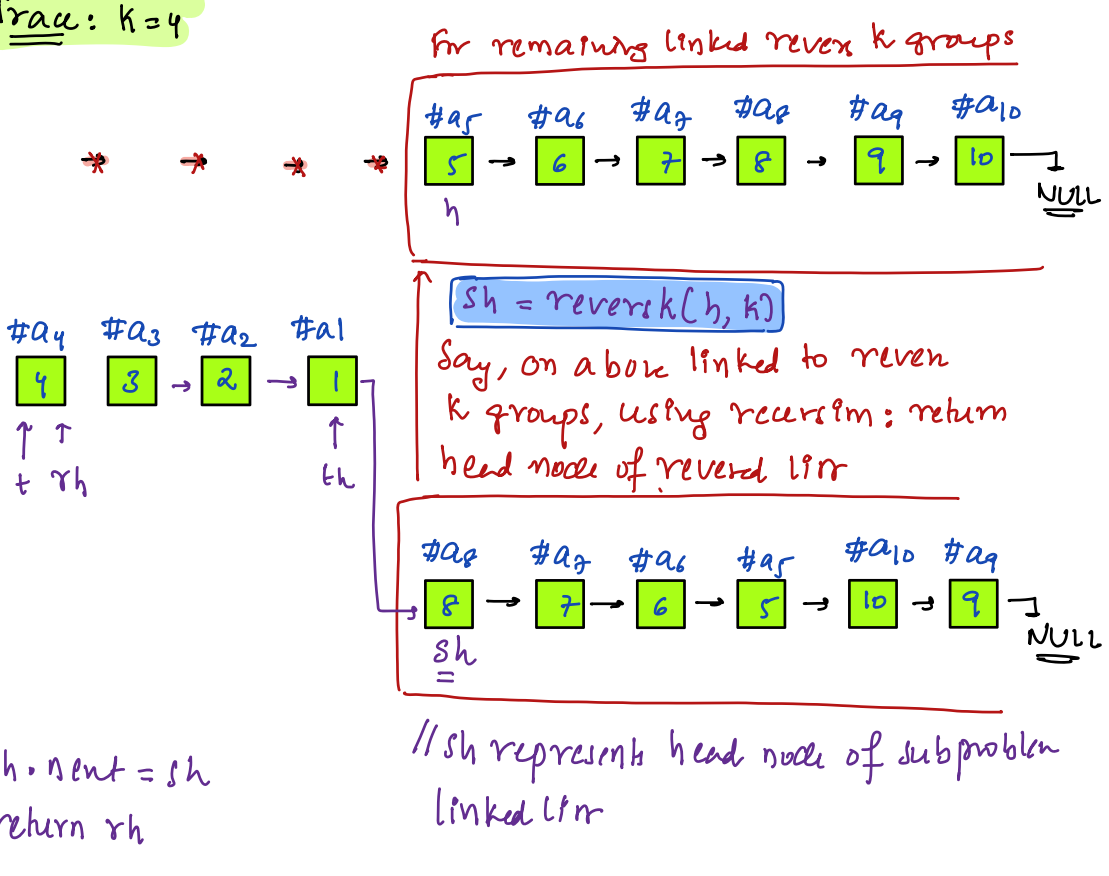
Note: If a group has less than k elements we still need reverse

Note: Recursion based extra space allowed

Ex: k = 4:



Trace: k = 4



Ass: Revern k groups for linked list & return head node

```
Node revenk(Node h, int k){ TC: O(N) SC: O(N/k)
    if(h== null){return h;}
    Node rh=null, t=h, th=h                    // reverse first k nodes
    int tk = k // storing temporary k          if no: of node are less
    while(h!=NULL && k>0){                     than k we will still
        t=h                                    revern
        h = h.nent
        t.nent=null //not necessary
        t.nent=rh
        rh=t
        k=k-1
    }
    Node sh = reversk(h, tk) // reven k groups in given list
                              // original k is getting updated
    th.nent = sh
    return rh; // head node for entire revern list
}
```

Note: reverse(h, k=2)

↳ // swapping every 2 adjacent nodes

// smallest prefix strings Doubts

A = abba
B = cdd

prefix A : { "a" "ab" "abb" "abba" }
prefix B : { "c" "cd" "cdd" }

"ac" = "acd" "acdd"

"abc" "abcd" "abcdd"

"abbc" "abbcd" "abbcdd"

abbac ablacd ablacdd