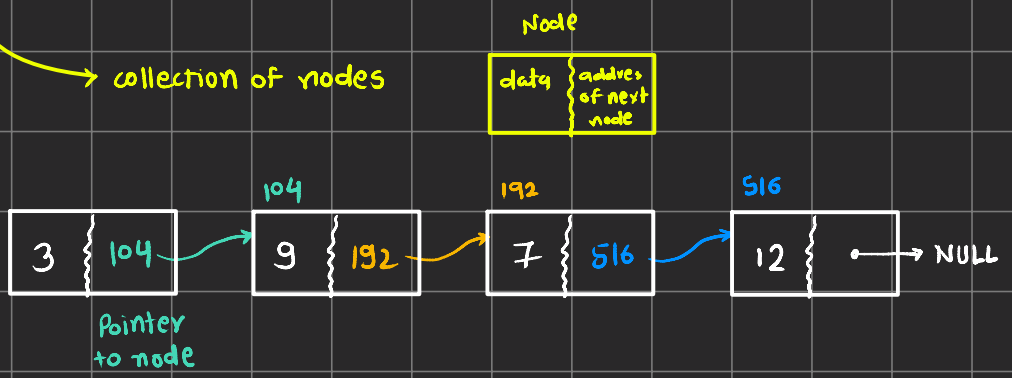
Linked List

- array → Linear DS • allocate continious memory allocation
- LL → • can work on non continious memory allocation

→ collection of nodes



(like in vector for ele out of size creates almost doubled size)

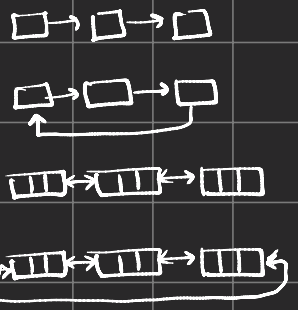
- memory can be waste
- every time new node so no waste (dynamically grow/shrink)

- ek insert k liye TC. O(n)
- ek insert k liye TC. O(1) can be very

```
class Node {
public:
    int data;    // To store the value
    Node* next;  // Pointer to the next node
};
```

Types

- → singly ll
- → circular singly ll
- → doubly ll
- → circular doubly ll



temp → next means 10 wala node

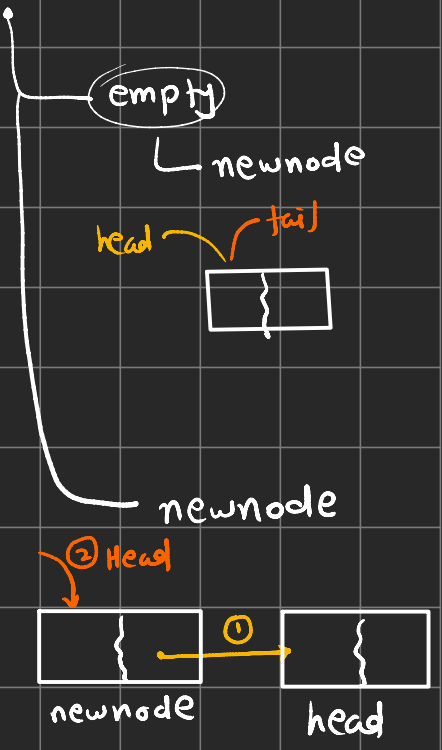
temp → next → next means 15 wala node

```cpp
class Node{

    public:
        int data;
        Node* next;

        Node(){
            this->data = 0;
            this->next = NULL;
        }

        Node(int data){
            this->data = data;
            this->next = NULL;
        }

        ~Node(){
            int value = this -> data;
            //memory free
            if(this->next != NULL) {
                delete next;
                this->next = NULL;
            }
            cout << " memory is free for node with data " << value << endl;
        }

};
```

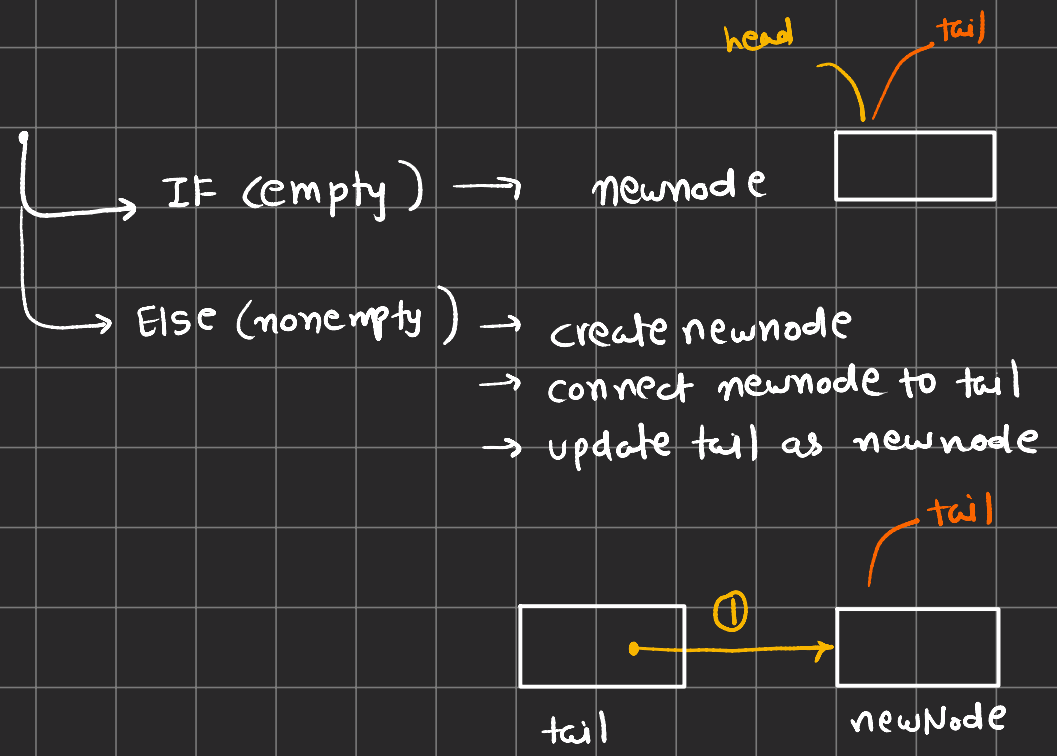
```cpp
void print(Node* &head){
    Node*temp = head;
    while( temp != NULL ){
        cout << temp->data << endl;
        temp = temp->next;
    }
}
```

Add condition that if empty or not

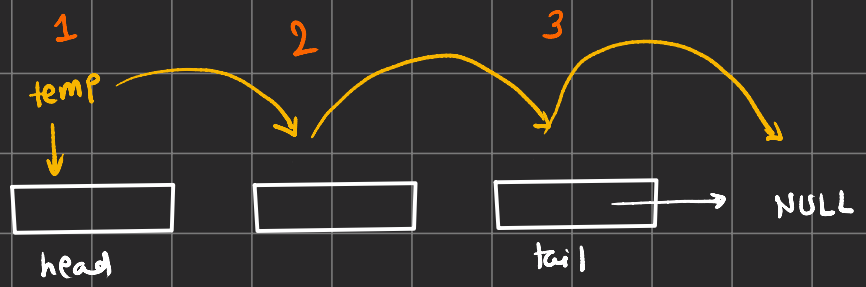
```cpp
// i want to insert a node right at the head of linked list
// (here head passed as referenace so can change in orignal)
void insertAtHead(Node* &head, Node* &tail,int data){

    if(head == NULL){
        //empty ll

        Node* newNode = new Node(data);

        // means newNode is going to be first node
        head = newNode;
        tail = newNode;

    }
    else{
        //1)create a newNode
        Node* newNode = new Node(data);

        //2) attach newNode->next to head
        newNode->next = head;

        //3) update head as newNode
        head = newNode;
    }
}
```



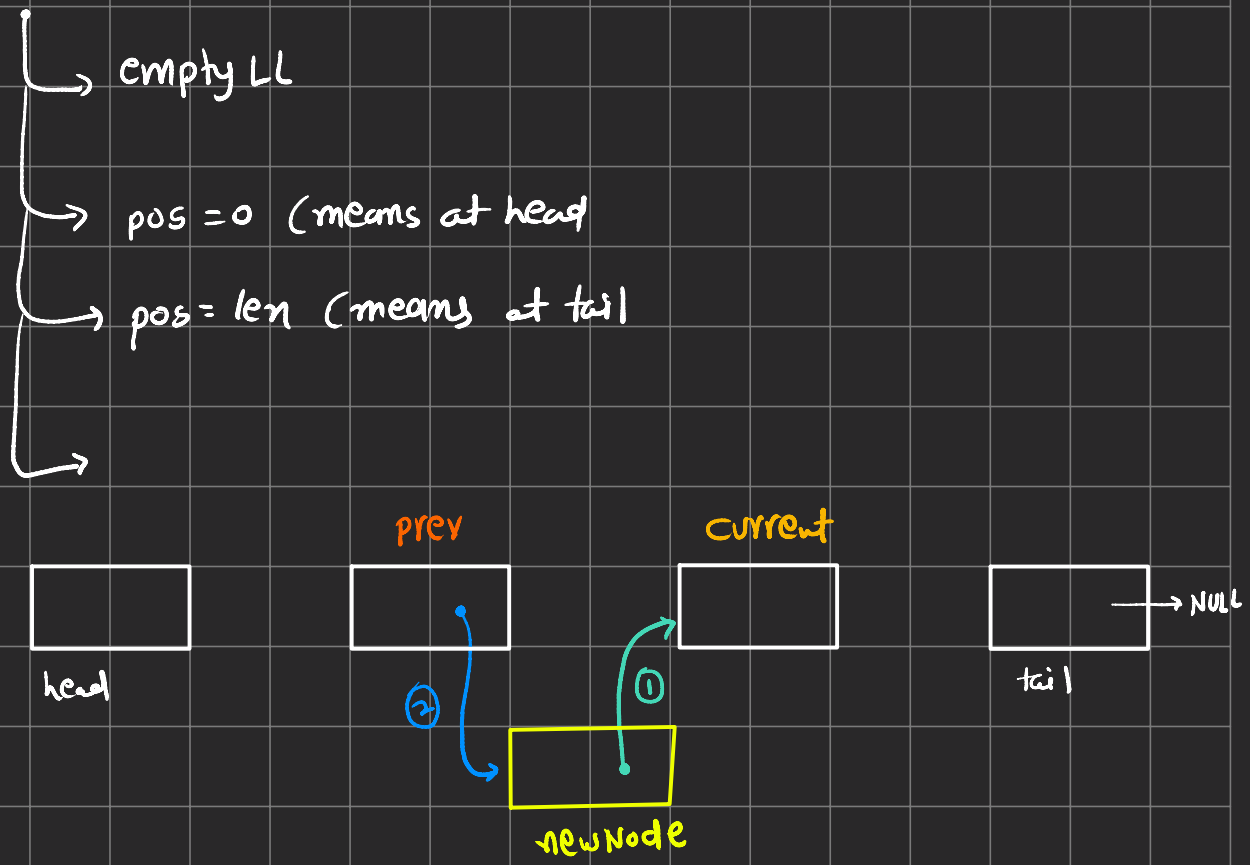
```cpp
// i want to insert a new node right at the end of linkedlist
void insertAtTail(Node* &head, Node* &tail, int data){

    if(head == NULL){
        //empty ll

        Node* newNode = new Node(data);

        // means newNode is going to be first node
        head = newNode;
        tail = newNode;

        return;

    }

        //1)create a newNode
        Node* newNode = new Node(data);

        //2) attach tail->next to newNode
        tail->next = newNode;

        //3) update tail as newNode
        tail = newNode;

}
```



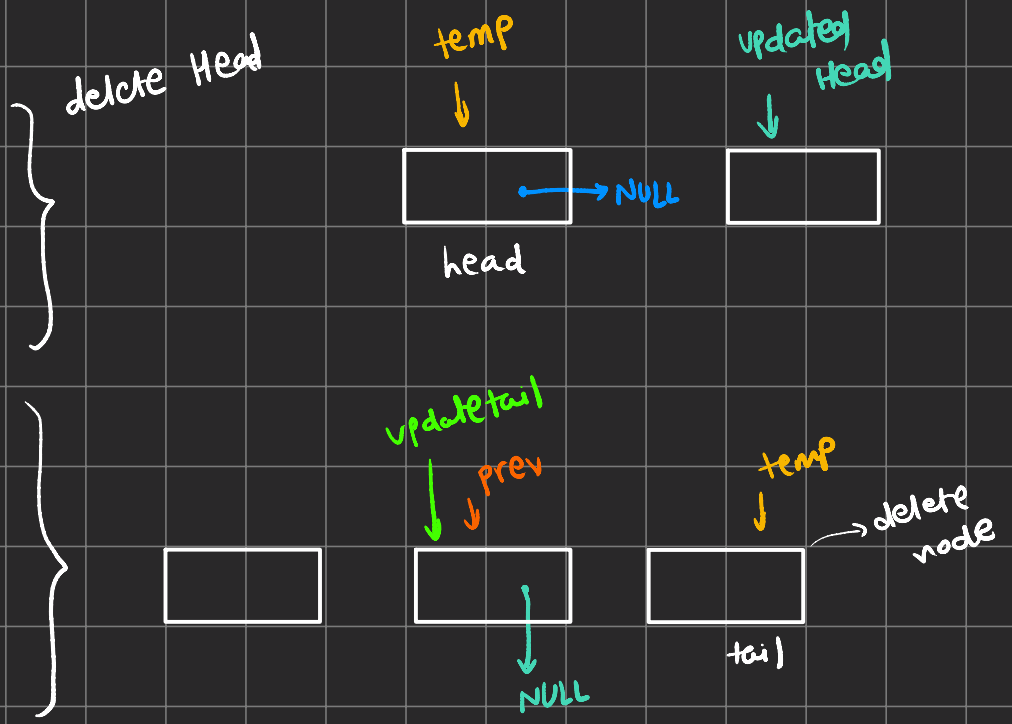

```cpp
int findLength(Node* &head){
    Node* temp = head;

    int len = 0;
    while(temp!=NULL){
        temp=temp->next;
        len++;
    }

    return len;
}
```



```cpp
void insertAtPosition(Node* &head, Node* &tail, int position, int data){

    if(head == NULL){
        //empty ll

        Node* newNode = new Node(data);

        // means newNode is going to be first node
        head = newNode;
        tail = newNode;

    }
    else if(position==0){
        insertAtHead(head,tail,data);
    }
    else if(position>=findLength(head)){
        insertAtTail(head,tail,data);
    }
    else{
        //1) find position: prev & curr
        int i=1;
        Node* prev = head;

        while(i<position){
            prev = prev->next;
            i++;
        }

        Node* curr = prev->next;

        //2) create newNode
        Node* newNode = new Node(data);

        //3) newNode-> next = current
        newNode-> next = curr;

        //4) prev->next = current
        prev->next = newNode;

    }
```

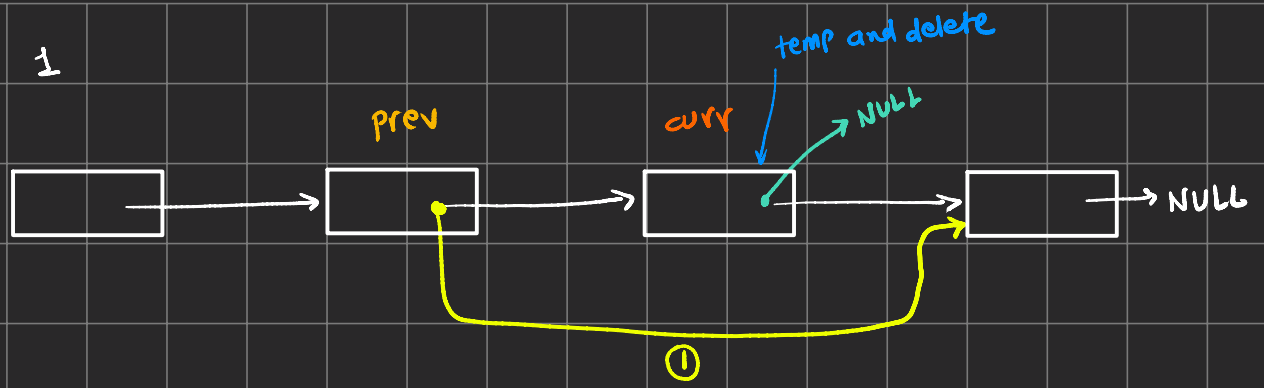


```cpp
void deleteNode(Node* &head, Node* &tail, int position){
    if(head == NULL){
        //empty ll
        cout << "cannot delte Linkedlist is empty" << endl;
        return;
    }

    int len = findLength(head);

    if (position < 1 || position > len) {
        // Invalid position
        cout << "Invalid position" << endl;
        return;
    }


    // delete at head (first node)
    if(position==1){
        Node* temp = head;
        head = head->next;
        temp->next = NULL;
        delete temp;
        return;
    }

    // delete at tail (last node)
    if(position == findLength(head)){
        //1)find previous
        int i=1;
        Node* prev = head;

        while(i<position-1){
            prev = prev->next;
            i++;
        }


        //2) prev->next ko null
        prev->next = NULL;

        //3) got temp as tail
        Node* temp = tail;

        //4) arange temp as tail
        tail = prev;

        //5)
        delete temp;
```

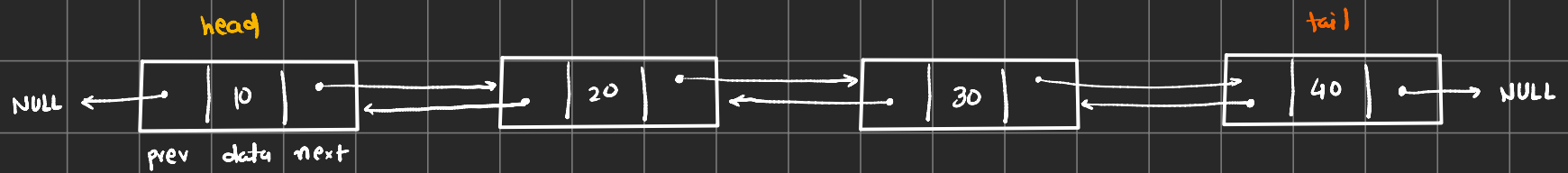


making head->next = NULL is imp if we forgot then it can delete whole LL while delete temp;

```
    // deleting middle node
    //1) find position: prev & curr
        int i=1;
        Node* prev = head;

        while(i<position-1){
            prev = prev->next;
            i++;
        }

        Node* curr = prev->next;

        //2)
        prev->next = curr->next;

        //3)
        curr->next = NULL;

        //4)
        delete curr;

}
```



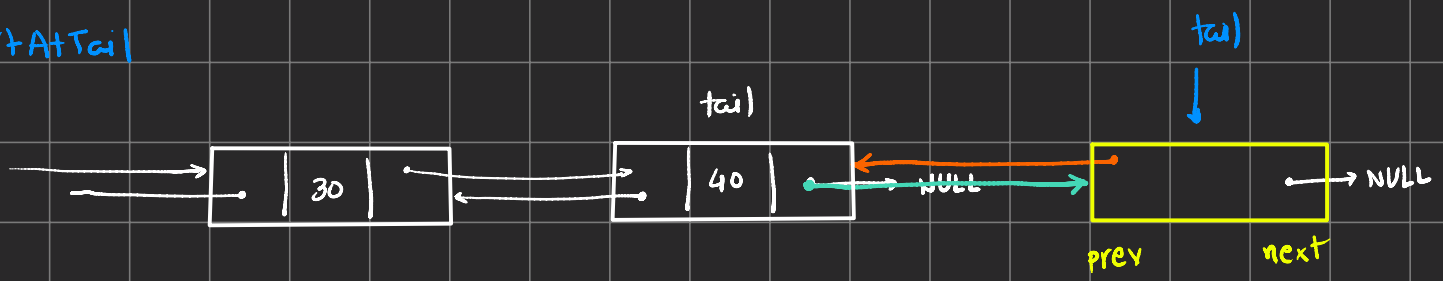
Doubly Linked List



insert at Head

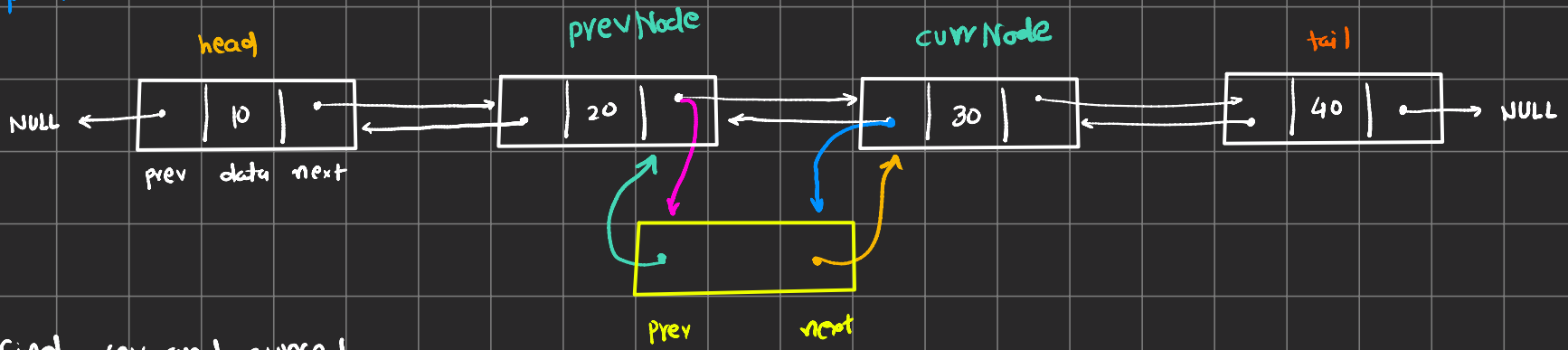
- create new node
- newnode → next ko head par point
- head → prev ko newnode se connect (newNode → prev = NULL automatic constructor se hoga)
- head ko update karo

insertAtTail



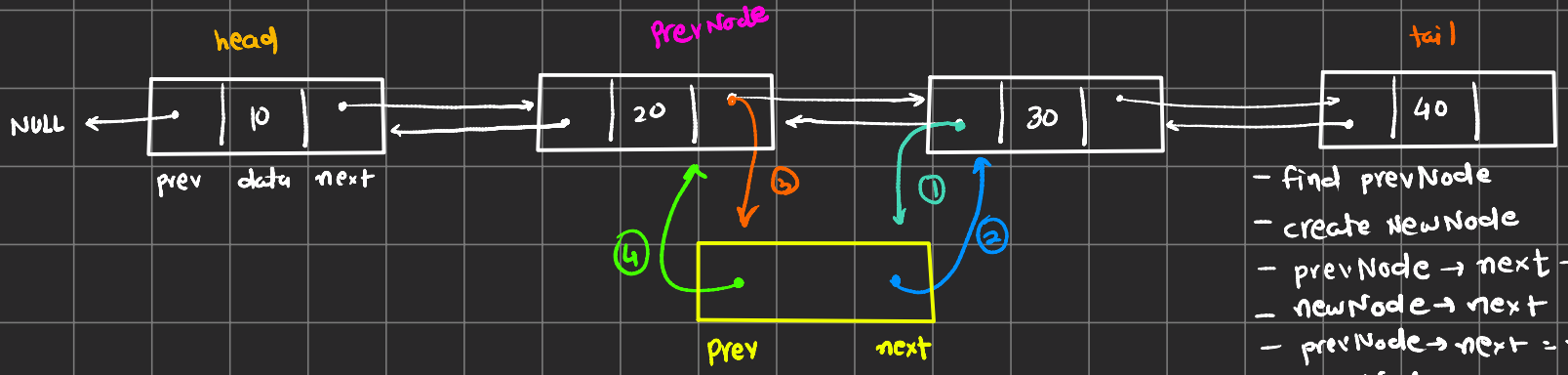
- create new node
- tail → next = newNode
- newNod → prev = tail
- Update tail

Insert At position



- find prev and current
- newnode create
- prevNode → next = newNode
- newNode → prev = prevNode
- currNode → prev = newNode
- newNode → next = currNode

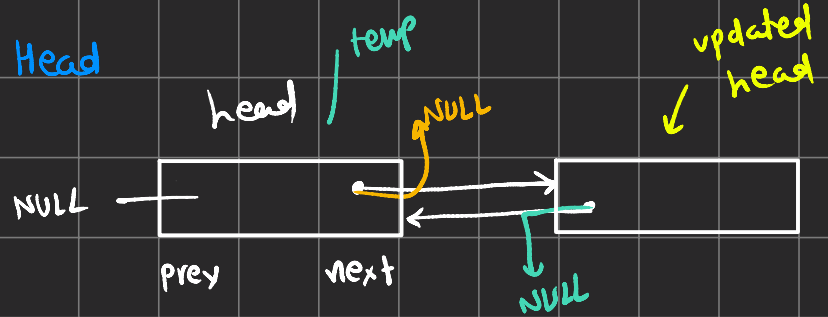
Without currentNode



Order Matters

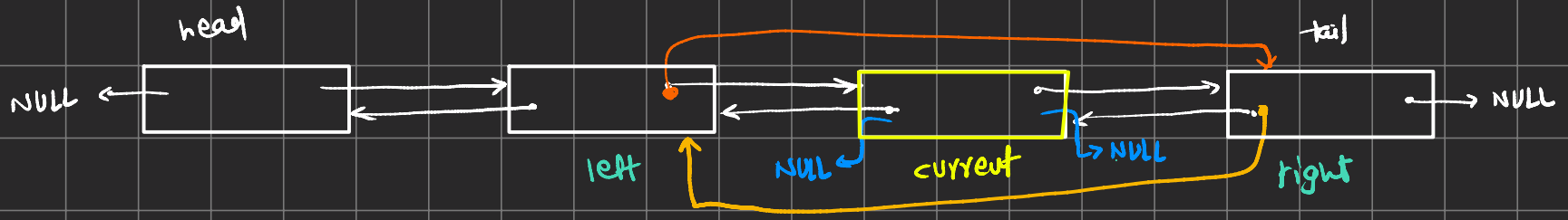
- find prevNode
- create newNode
- prevNode → next → prev = newNode
- newNode → next = prevNode → next
- prevNode → next = newNode
- newNode → prev = prevNode

deletion at Head



- temp = head
- head = head → next
- head → prev = NULL
- temp → next = NULL
- delete temp

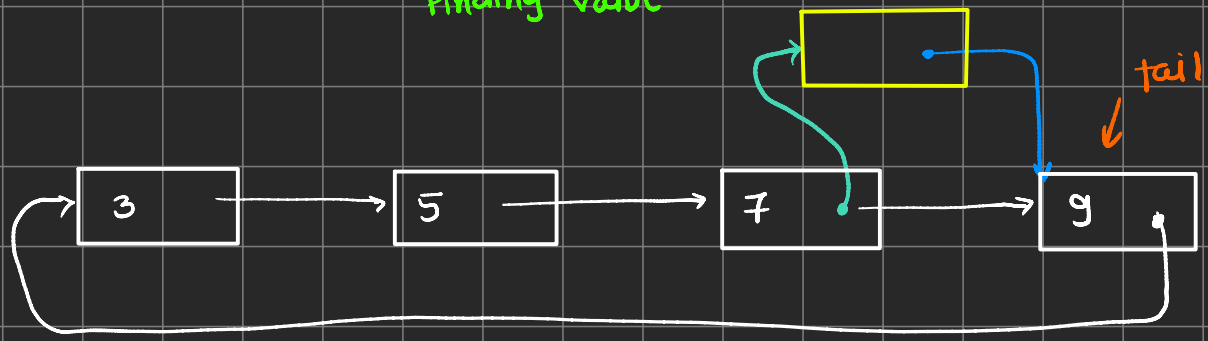
deletion at middle



- find left, right, current
- left → next = right
- right → prev = left
- Current → prev = NULL
- current → next = NULL
- delete current

circular singly ll

on basis of value
Insert using finding value



(no head
no tail in circular
but for pointing
adding one pointer
as tail

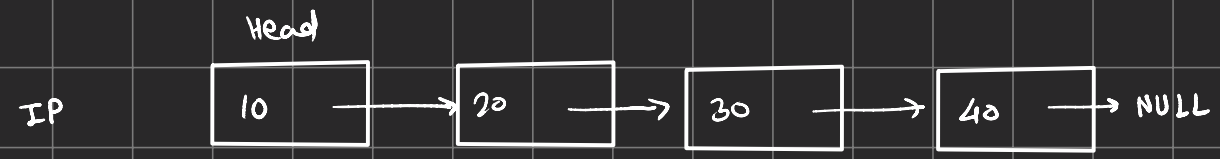
```cpp
class Node {
    public:
    int data;
    Node* next;

    // Constructor
    Node(int d) {
        this->data = d;
        this->next = NULL;
    }

    // Destructor
    ~Node() {
        int value = this->data;
        if(this->next != NULL) {
            delete next;
            next = NULL;
        }
        cout << "Memory is free for node with data " << value << endl;
    }
};

void insertNode(Node* &tail, int element, int d) {
    // Empty list
    if(tail == NULL) {
        Node* newNode = new Node(d);
        tail = newNode;
        newNode->next = newNode;
    }
    else {
        // Non-empty list, assuming the element is present in the list
        Node* curr = tail;

        while(curr->data != element) {
            curr = curr->next;
        }

        // Element found, insert new node
        Node* temp = new Node(d);
        temp->next = curr->next;
        curr->next = temp;
    }
}
```

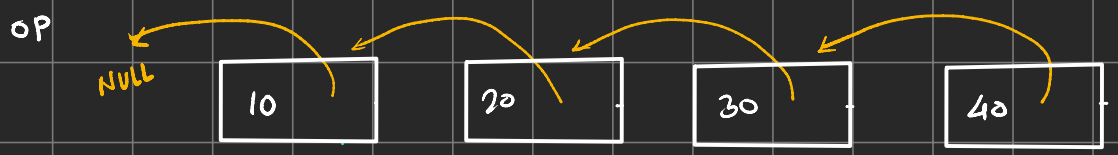
```cpp
void print(Node* tail) {
    // Empty list
    if(tail == NULL) {
        cout << "List is Empty" << endl;
        return;
    }

    Node* temp = tail;
    do {
        cout << tail->data << " ";
        tail = tail->next;
    } while(tail != temp);

    cout << endl;
}

void deleteNode(Node* &tail, int value) {
    // Empty list
    if(tail == NULL) {
        cout << "List is empty, please check again" << endl;
        return;
    }
    else {
        // Non-empty list, assuming "value" is present in the Linked List
        Node* prev = tail;
        Node* curr = prev->next;

        while(curr->data != value) {
            prev = curr;
            curr = curr->next;
        }

        prev->next = curr->next;

        // Case: 1 Node Linked List
        if(curr == prev) {
            tail = NULL;
        }
        // Case: >=2 Node linked list
        else if(tail == curr) {
            tail = prev;
        }

        curr->next = NULL;
        delete curr;
    }
}
```

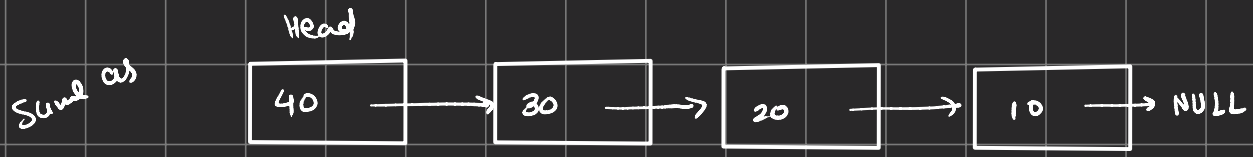
```cpp
bool isCircularList(Node* head) {
    // Empty list
    if(head == NULL) {
        return true;
    }

    Node* temp = head->next;
    while(temp != NULL && temp != head) {
        temp = temp->next;
    }

    return temp == head;
}

bool detectLoop(Node* head) {
    if(head == NULL) {
        return false;
    }

    map<Node*, bool> visited;
    Node* temp = head;

    while(temp != NULL) {
        // Cycle is present
        if(visited[temp] == true) {
            return true;
        }

        visited[temp] = true;
        temp = temp->next;
    }
    return false;
}
```

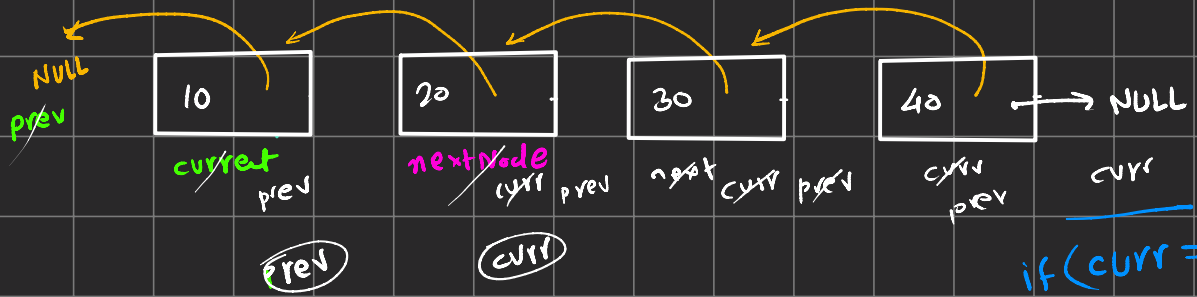
```cpp
int main() {
    Node* tail = NULL;

    insertNode(tail, 5, 3);
    print(tail);

    insertNode(tail, 3, 5);
    print(tail);

    /* Additional operations can be uncommented for testing

    insertNode(tail, 5, 7);
    print(tail);

    insertNode(tail, 7, 9);
    print(tail);

    insertNode(tail, 5, 6);
    print(tail);

    insertNode(tail, 9, 10);
    print(tail);

    insertNode(tail, 3, 4);
    print(tail);

    deleteNode(tail, 5);
    print(tail);
    */

    if(isCircularList(tail)) {
        cout << "Linked List is Circular in nature" << endl;
    }
    else {
        cout << "Linked List is not Circular" << endl;
    }

    return 0;
}
```

☆ Reverse a linked list









current → next = prev

if (curr = NULL)
return prev

Base case

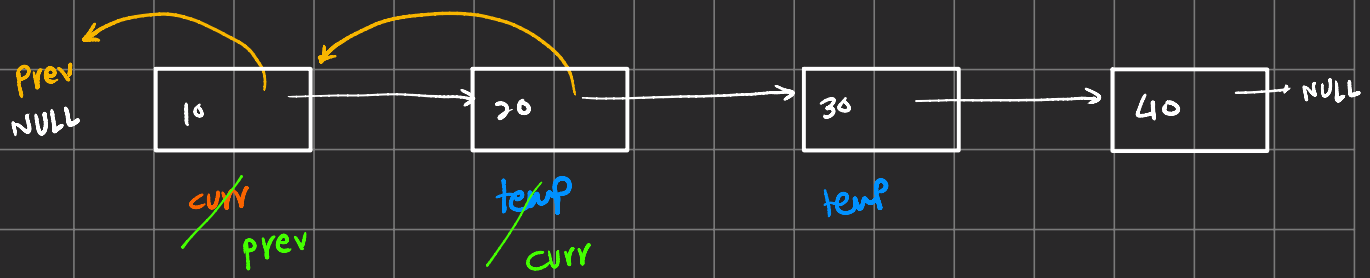
returning pointer of new Head

```cpp
Node* reverse(Node* &prev, Node* &curr){

    // base case
    if(curr == NULL){
        return prev; // new head
    }

    // rr
    Node* forward = curr->next;
    curr->next = prev;

    reverse(curr,forward);
}
```

prev

just for pointing next

```cpp
Node* reverseUsingLoop(Node* &head){

    Node* prev = NULL;
    Node* curr = head;

    while(curr != NULL){
        Node* temp = curr->next;
        curr->next = prev;

        prev = curr;
        curr = temp;
    }

    return prev;


}

Node* reverseUsingRecursion(Node* &prev, Node* &curr){

    // base case
    if(curr == NULL){
        return prev; // new head
    }

    // rr
    Node* temp = curr->next;
    curr->next = prev;

    prev = curr;
    curr = temp;

    reverseUsingRecursion(prev,temp);

}
```



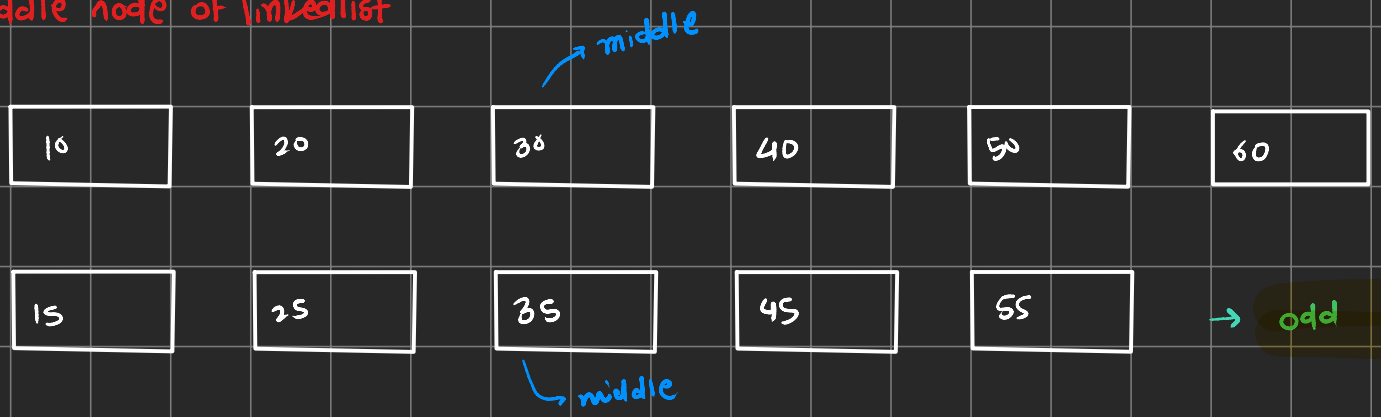
temp = curr → next

curr → next = prev

prev = curr

curr = temp

* find middle node of linkedlist

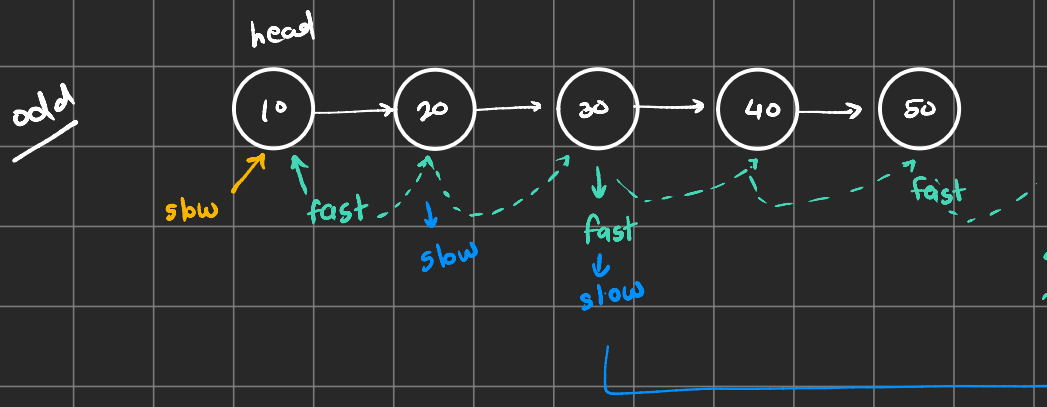


n = length → find it.

→ even $n/2$

→ odd $\frac{n}{2}+1$

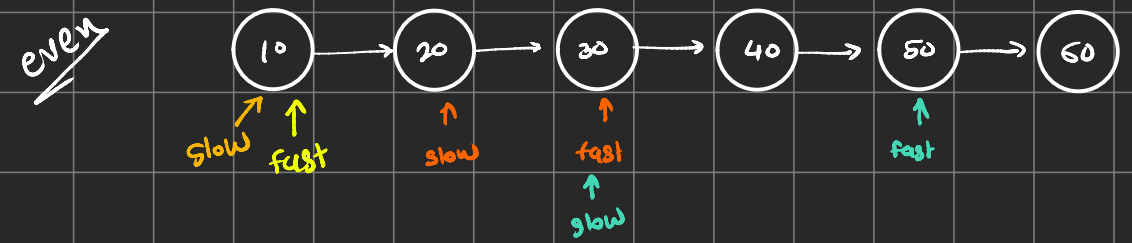
Another algo



slow → 1 step

fast → 2 step

fast aage badh nahi payega us time slow ko change nahi karna



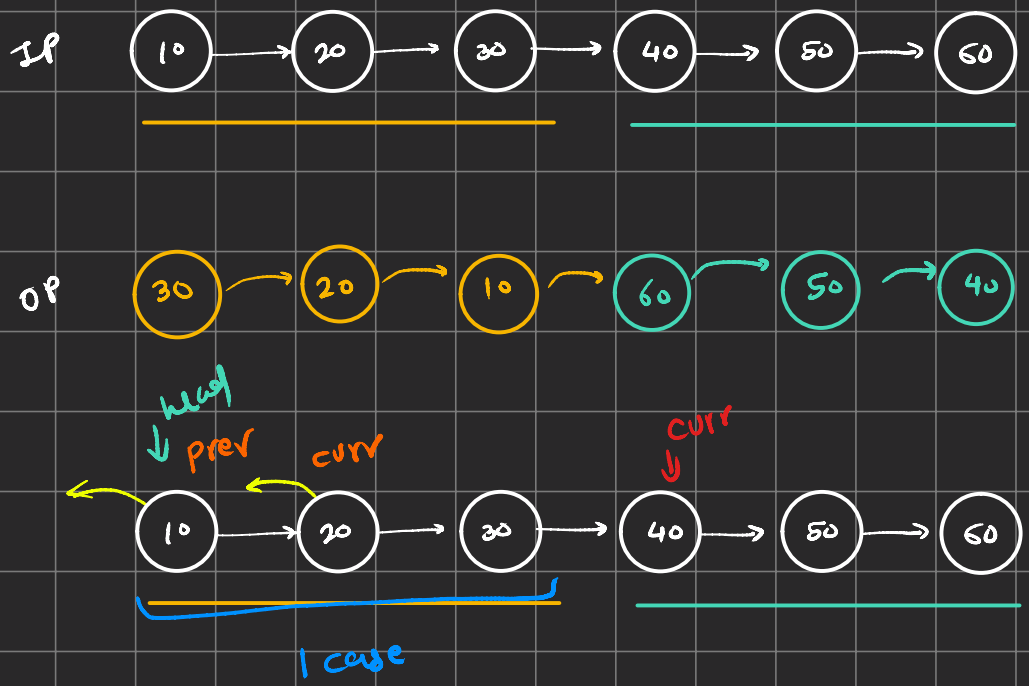
fast is out of bond = NULL
so do not move slow so here slow is ans.

Tc = $O(n/2)$
$\approx O(n)$

```
ListNode* middleNode(ListNode* head) {

    if(head == NULL){
        // cout << "ll is empty";
        return head;
    }

    if(head->next == NULL){
        return head;
    }

    // ll have > than 1 node
    ListNode* slow = head;
    ListNode* fast = head;


    while(slow != NULL && fast!=NULL){
        fast = fast->next;
        if(fast != NULL){
            fast = fast->next;
            slow = slow->next;
        }
    }

    // cout << "middle node is "<< slow->val << cout;
    return slow;
}
```

* K groups reverse

k=3



prev = NULL
curr = head

for (i=0, i<K, i++)

next = curr->next

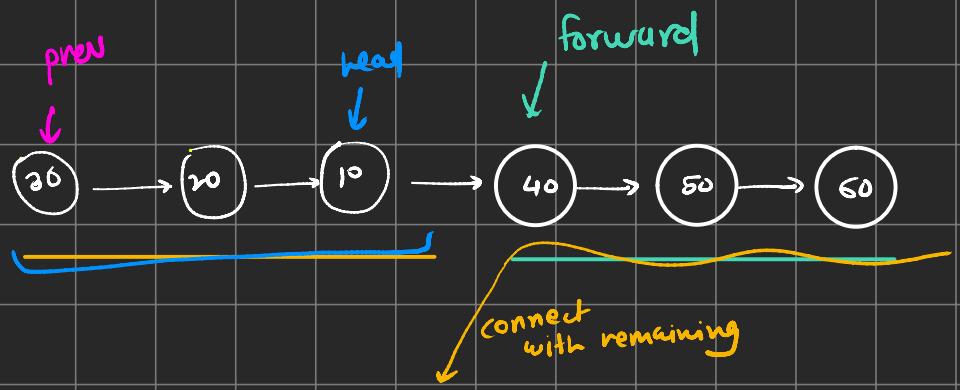
curr-> next = prev

prev = curr

curr = next

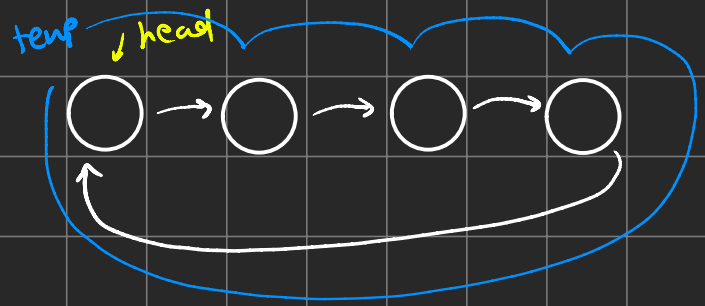
} updated head = forward

```
ListNode* reverseKGroup(ListNode* head, int k) {
    if (head == NULL) {
        // Linked list is empty
        return head;
    }

    if (k > getLen(head)) {
        // Invalid k value
        return head;
    }

    // Reverse logic
    ListNode* prev = NULL;
    ListNode* curr = head;
    ListNode* next = curr->next;
    int count = 0;
    ListNode* forward = NULL;

    while (count < k && curr != NULL) {
        forward = curr->next;
        curr->next = prev;
        prev = curr;
        curr = forward;
        count++;
    }

    // Recursive reversal for remaining nodes
    if (forward != NULL) {
        head->next = reverseKGroup(forward, k);
    }

    // Return the new head of the modified linked list
    return prev;
}
```




```
head->next = reverseKNode(forward, k)
```

isCircular LL or not

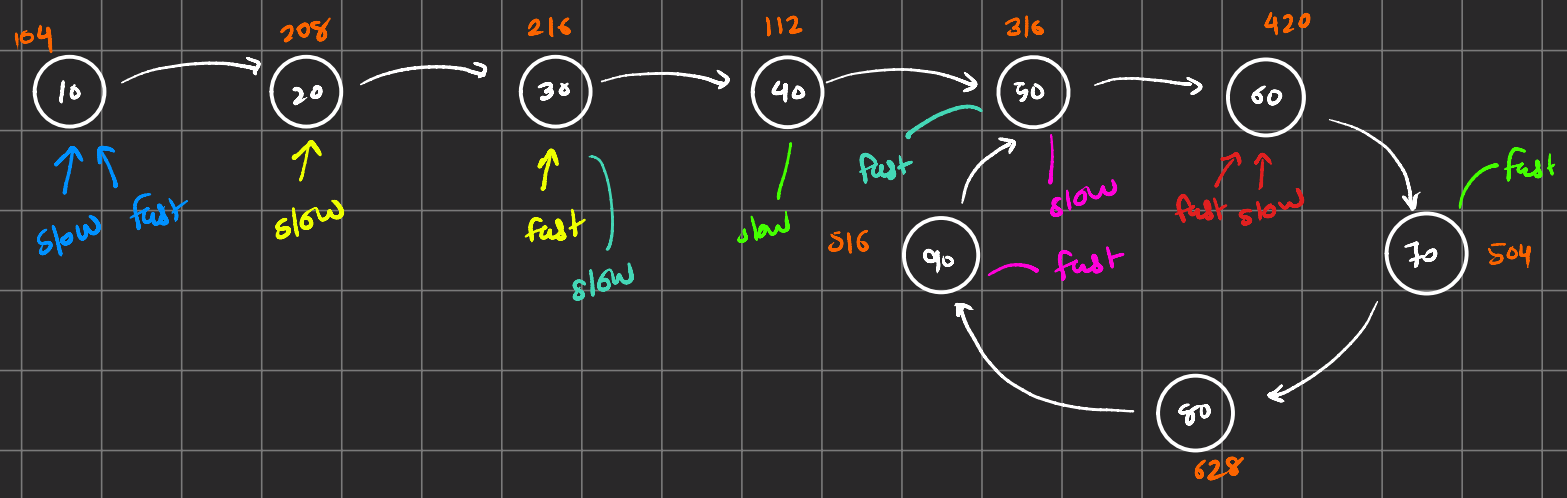


temp = head

```
bool isCircularList(Node* head) {
    // Empty list
    if(head == NULL) {
        return true;
    }

    Node* temp = head->next;
    while(temp != NULL && temp != head) {
        temp = temp->next;
    }

    return temp == head;
}

bool detectLoop(Node* head) {
    if(head == NULL) {
        return false;
    }

    map<Node*, bool> visited;
    Node* temp = head;

    while(temp != NULL) {
        // Cycle is present
        if(visited[temp] == true) {
            return true;
        }

        visited[temp] = true;
        temp = temp->next;
    }
    return false;
}
```

detect and delete loop

- check loop present in LL or not
- starting point of loop
- remove loop



map

key (address)	value (T/F)
104	T
208	T
216	T
112	T
316	T
420	T
524	T
628	T
516	T
316	

circular loop present

Floyed cycle Detection

(slow/ Fast algo)

when (fast == slow) → loop present

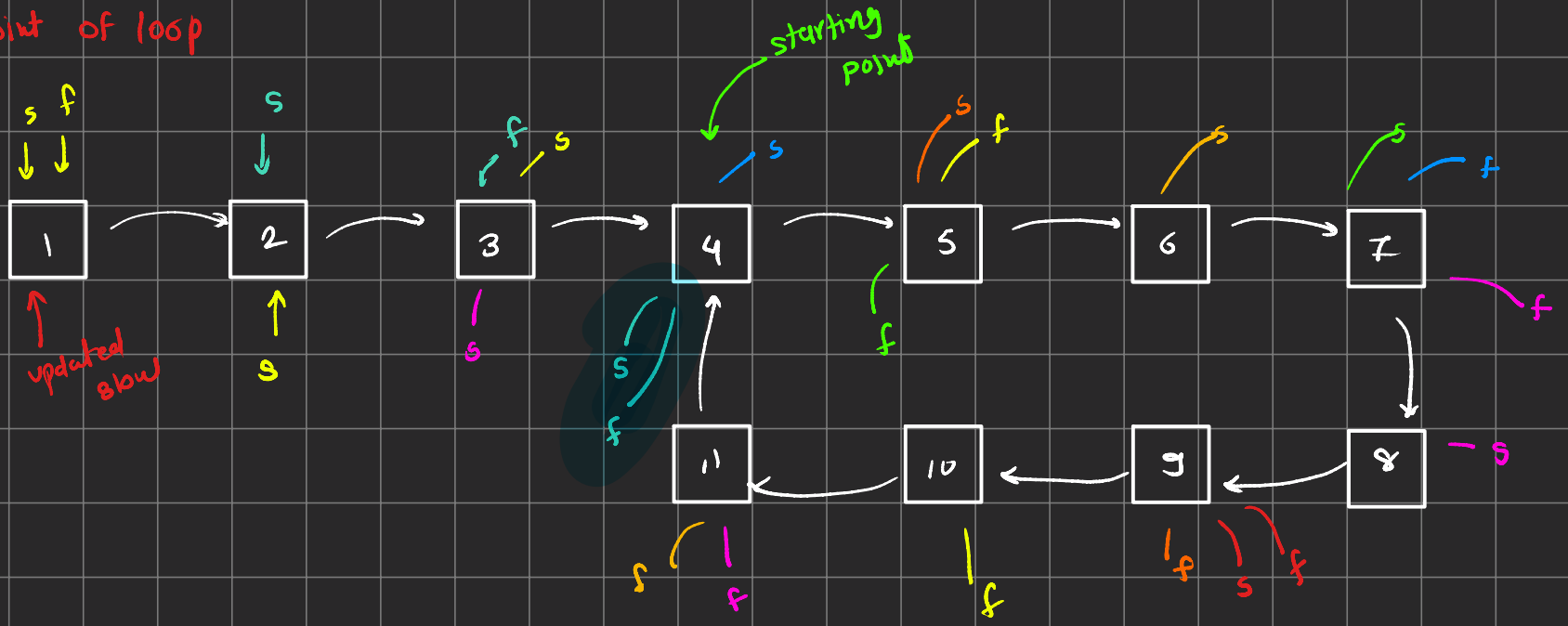
both pointer same

when (fast == NULL) → loop ends means LL done

```
bool hasCycle(ListNode *head) {

    ListNode* slow = head;
    ListNode* fast = head;

    while(slow != NULL && fast != NULL){
                no need
        fast = fast->next;
        if(fast!=NULL){
            fast = fast->next;
            slow = slow->next;
        }

        if(fast == slow){
            return true;
        }

    }

    return false;
}
```

* starting point of loop

slow = 1
fast = 2 step

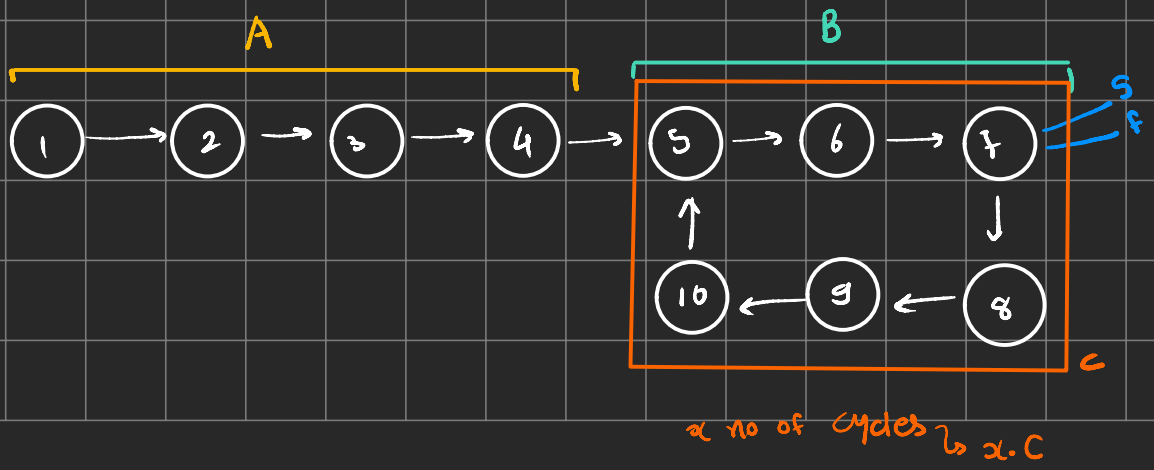


① fast and slow meet karwa do

② slow = head

③ slow and fast ko ek ek step chalo ⇝ Jab meet to go starting point

Maths



x no of cycles ↳ x.C

distance travelled by Fast pointer = 2 * distance travelled by slow pointer

$$A + xC + B = 2 * (A + yC + B)$$

$$A + xC + B = 2A + 2yC + 2B$$

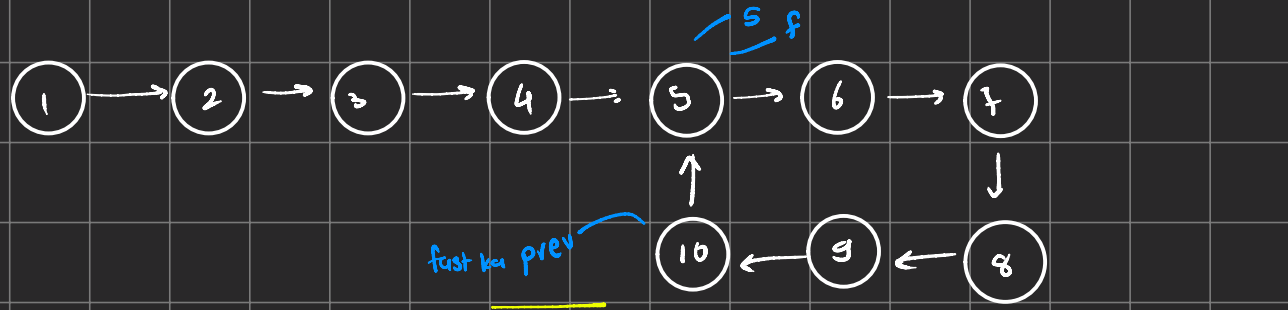
$$(x - 2y)C = A + B$$

let $x - 2y = k$

$$k \cdot C = A + B$$

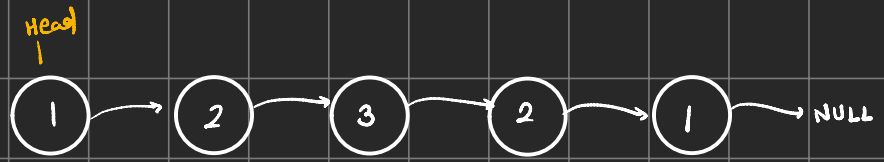
★ Remove cycle / loop

- while finding starting point of loop, during 3rd step



save prev pointer always with fast during 3rd step and then make it NULL

★ LL is palindrome or not



ip = 12321
op = 12321

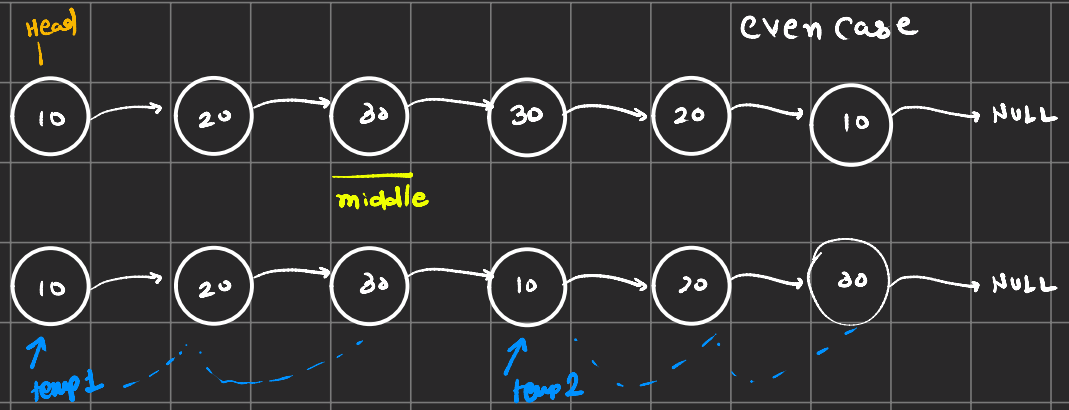
① normal i/p, reversed LL } compare both

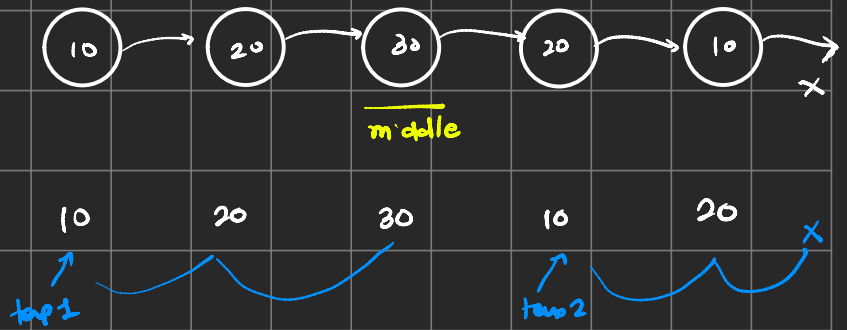
SC = O(n)
TC = O(n)

② LL data copy to Array then use two pointers method

SC. O(n)
TC. O(n)

③ Even case





① Find middle

② Reverse LL after middle node

③ start comparing both half

Tc. $O(n/2) \simeq O(n)$

Sc. $O(1)$

do not care odd/even

```
ListNode* findMiddle(ListNode* head) {

    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            fast = fast->next;
            slow = slow->next;
        }
    }

    return slow;
}
```

```
ListNode* reverse(ListNode*& middle) {

    ListNode* prev = NULL;
    ListNode* curr = middle;
    ListNode* forward = curr->next;

    while (curr != NULL) {
        forward = curr->next;
        curr->next = prev;

        prev = curr;
        curr = forward;
    }

    return prev;
}
```

```
bool isPalindrome(ListNode* head) {

    if (head == nullptr || head->next == nullptr)
        return true;

    // 1)find middle
    ListNode* middle = findMiddle(head);

    // 2)reverse ll after middle
    ListNode* reverseKaHead = reverse(middle);

    // middle->next = reverseKaHead;

    // 3) compare
    ListNode* temp1 = head;
    ListNode* temp2 = reverseKaHead;

    bool isPalin = true;

    while (temp2 != nullptr) {
        if (temp1->val != temp2->val) {
            isPalin = false;
            break;
        }
        temp1 = temp1->next;
        temp2 = temp2->next;
    }

    return isPalin;
}
```

connecting is not imp (giving error in LC)

★ Remove duplicates from sorted LL

curr → data == curr → next → data → equal → curr → next = curr → next → next

→ not equal → curr = curr → next

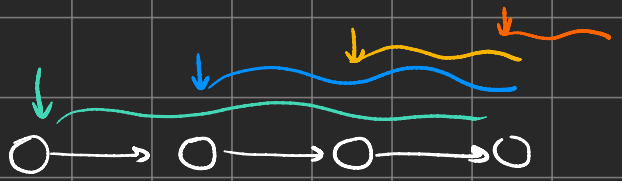
```
ListNode* deleteDuplicates(ListNode* head) {

    ListNode* temp = head;

    while (temp != NULL) {

        if(temp->next != NULL && (temp->val == temp->next->val)){
            temp->next = temp->next->next;
        }
        else {
            temp = temp->next;
        }
    }

    return head;
}
```

```
ListNode* deleteDuplicates(ListNode* head) {

    ListNode* curr = head;

    while (curr != NULL) {

        if(curr->next != NULL && (curr->val == curr->next->val)){
            ListNode* temp = curr->next;
            curr->next = curr->next->next;

            temp->next = NULL;
            delete temp;
        }
        else {
            curr = curr->next;
        }
    }

    return head;
}
```

→ with deletion of that node

✱ remove duplicates from unsorted LL

① nested loop



if that ele available after then delete it

② map — make tracking map

value → T/F availability

if already presented in map then remove that node

③ sort LL + prev logic (upper wala)

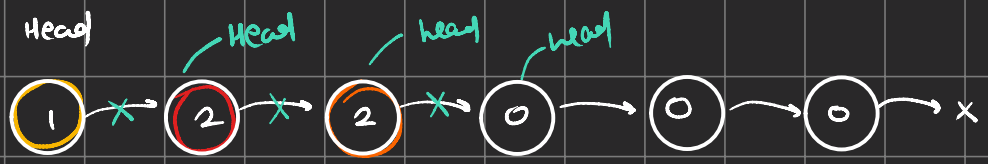
✱ sort 0s, 1s and 2s

i/p: 2 → 1 → 1 → 0 → 0 → 2 → 1 → x

o/p: 0 → 0 → 1 → 1 → 1 → 2 → 2 → x

zerocount = ~~0~~ 2
onecount = ~~0~~ 3
twocount = ~~0~~ 2

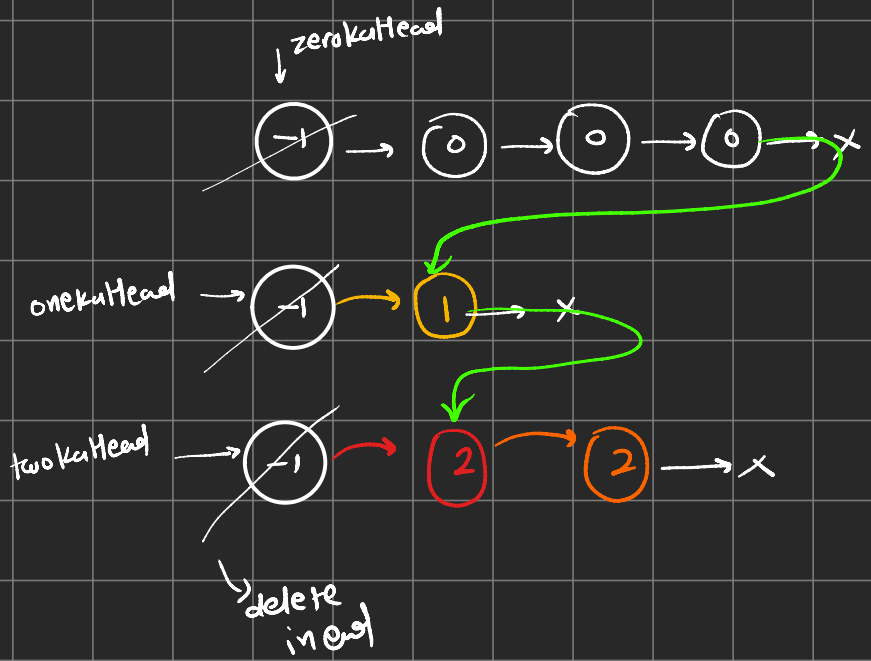
issue: data replaced



① temp = head

② head = head → next

③ temp → next = null



```
Node* sortList(Node* head) {

    // dummy nodes
    Node* zeroHead = new Node(-1);
    Node* zeroTail = zeroHead;
    Node* oneHead = new Node(-1);
    Node* oneTail = oneHead;
    Node* twoHead = new Node(-1);
    Node* twoTail = twoHead;

    Node* curr = head;

    // Create separate lists for 0s, 1s, and 2s
    while (curr != nullptr) {

        int value = curr->data;

        if (value == 0) {
            Node* temp = curr;
            curr = curr->next;
            temp->next = NULL;

            zeroTail->next = temp;
            zeroTail = temp;

        } else if (value == 1) {
            Node* temp = curr;
            curr = curr->next;
            temp->next = NULL;

            oneTail->next = temp;
            oneTail = temp;
        } else if (value == 2) {
            Node* temp = curr;
            curr = curr->next;
            temp->next = NULL;

            twoTail->next = temp;
            twoTail = temp;
        }

        curr = curr->next;
    }

    // modify one wali list
    Node* temp = oneHead;
    oneHead = oneHead->next;
    temp->next = NULL;
    delete temp;

    // modify two wali list
    temp = twoHead;
    twoHead = twoHead->next;
    temp->next = NULL;
    delete temp;

    // join list
    if(oneHead!=NULL){
        // one wali list non empty
        zeroTail->next = oneHead;
        if(twoHead!=NULL){
            oneTail->next = twoHead;
        }
    }
    else{
        // one wali list empty
        if(twoHead != NULL){
            zeroTail->next = twoHead;
        }
    }

    // remove zerohead dummy node
    temp = zeroHead;
    zeroHead = zeroHead->next;
    temp->next = NULL;
    delete temp;

    return zeroHead;
}
```

delete dummy node from one wali LL

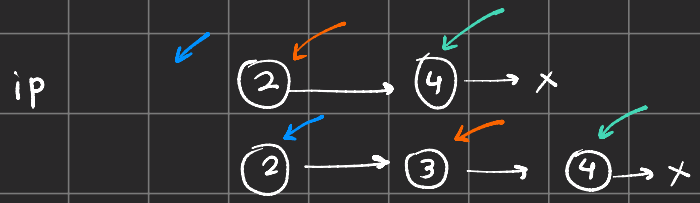
zeroList ko one se attach

if two exist then onelist ko two se attach

one list exist nahi karti

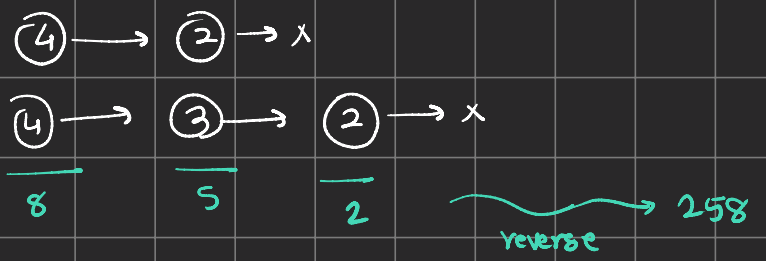
two wali list exist karti hai so zero wali ko two se attach karo

* Add two Numbers




```
  24
+ 234
-----
 258
```

reverse them



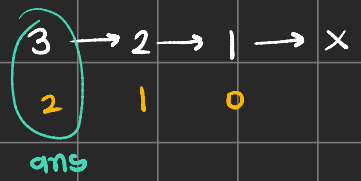

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    // reverse
    l1 = reverse(l1);
    l2 = reverse(l2);

    // calculate sum
    ListNode* ansHead = NULL;
    ListNode* ansTail = NULL;

    int carry = 0;
    while (l1 != NULL && l2 != NULL) {
        int sum = l1->val + l2->val + carry;
        int digit = sum % 10;
        carry = sum / 10;

        ListNode* newNode = new ListNode(digit);
        if (ansHead == NULL) {
            // first node
            ansHead = newNode;
            ansTail = newNode;
        } else {
            ansTail->next = newNode;
            ansTail = newNode;
        }
        l1 = l1->next;
        l2 = l2->next;
    }
```

```
    // head1 list ki length head2 list se jyada hogi
    while (l1 != NULL) {
        int sum = l1->val + carry;
        int digit = sum % 10;
        carry = sum / 10;

        ListNode* newNode = new ListNode(digit);
        ansTail->next = newNode;
        ansTail = newNode;
        l1 = l1->next;
    }

    // head2 list ki length head1 se jyada he
    while (l2 != NULL) {
        int sum = l2->val + carry;
        int digit = sum % 10;
        carry = sum / 10;

        ListNode* newNode = new ListNode(digit);
        ansTail->next = newNode;
        ansTail = newNode;
        l2 = l2->next;
    }

    // if still carry is there means notZero
    while (carry != 0) {
        int sum = carry;
        int digit = sum % 10;
        carry = sum / 10;

        ListNode* newNode = new ListNode(digit);
        ansTail->next = newNode;
        ansTail = newNode;
    }

    ansHead = reverse(ansHead);
    return ansHead;
  }
};
```

```
ListNode* reverse(ListNode*& head) {

    // Reverse the linked list starting from the given head
    ListNode* prev = nullptr;
    ListNode* curr = head;
    ListNode* forward = nullptr;

    while (curr != nullptr) {
        forward = curr->next;
        curr->next = prev;
        prev = curr;
        curr = forward;
    }

    return prev;
}
```

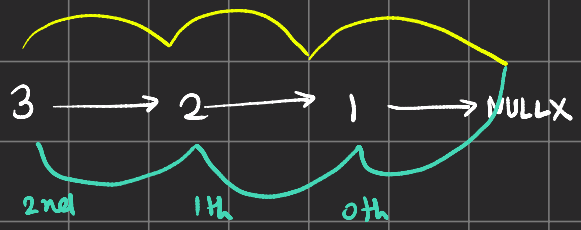
☆ print kth node from the end

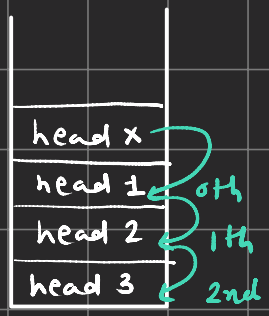
k = 2



0 th indexing

= len - k
= 3 - 2
= 1st Element From head

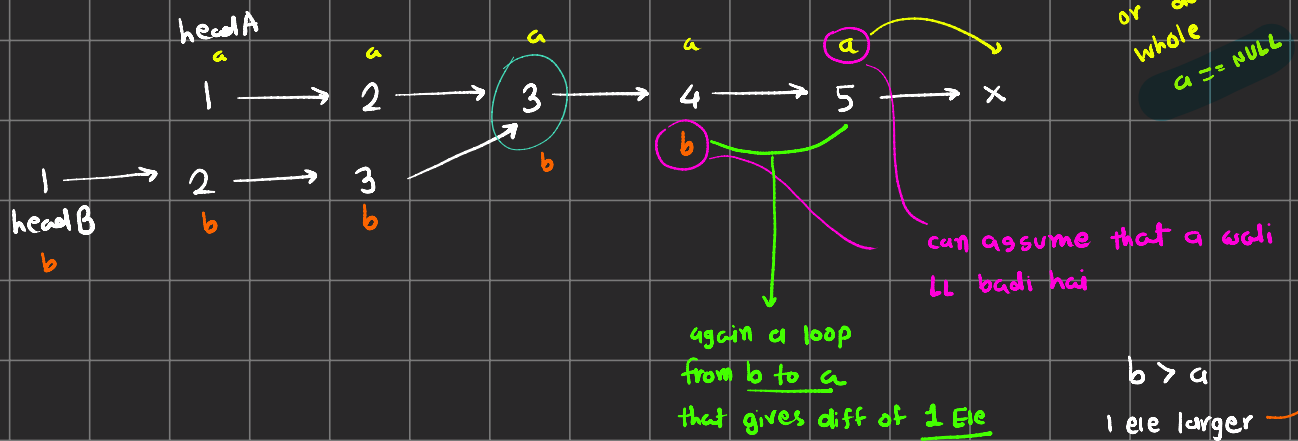


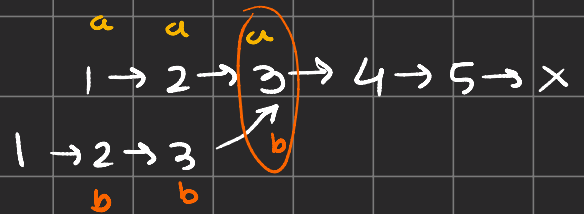


Recursion

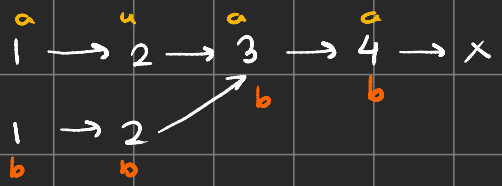
```
void func(SinglyLinkedListNode* head, int& pos, int& ans){

    // base
    if(head == NULL){
        return;
    }

    // head = head->next; // giving error
    func(head->next, pos, ans);

    if(pos == 0){
        ans = head->data;
    }
    pos--;

}
```

☆ Intersection of 2 LL





Same ele type of case



or we can compure in initial loop

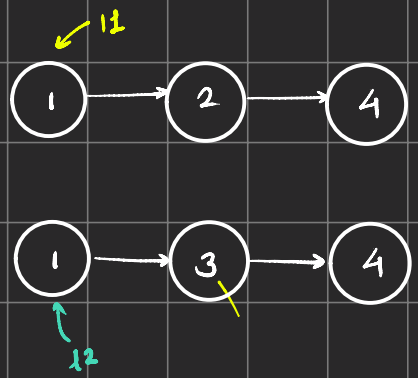
```cpp
ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {

    ListNode* a = headA;
    ListNode* b = headB;

    // Fix: Check for NULL before accessing next
    while (a != NULL && b != NULL) {

        // If both lengths of both linked lists are same
        if (a == b) {
            return a;
        }

        a = a->next;
        b = b->next;
    }

    // Fix: Ensure a and b are not NULL before accessing next
    if (a == NULL) {
        // b is longer
        int blen = 0;
        while (b != NULL) {  // Fix: Traverse b safely
            blen++;
            b = b->next;
        }
        while (blen--) {
            headB = headB->next;
        }
    } else {
        // a is longer
        int alen = 0;
        while (a != NULL) {  // Fix: Traverse a safely
            alen++;
            a = a->next;
        }
        while (alen--) {
            headA = headA->next;
        }
    }

    // Traverse both lists together
    while (headA != NULL && headB != NULL && headA != headB) {
        headA = headA->next;
        headB = headB->next;
    }

    return headA;  // Return intersection node or NULL
}
};
```

avoid to use like this

```cpp
while (a->next || b->next ) {

    // if both length of both ll is same
    if (a == b) {
        return a;
    }

    a = a->next;
    b = b->next;
}

if(a->next == NULL){
    // b is bigger
```

It gives error of accessing NULL→next so never use

✱ Merge two sorted LL



do with address not data

if (l1 <= l2) → attach l1 to ansTail then
l1 = l1 → next

else → attach l2 to ansTail then
l2 = l2 → next

ansHead
-1
ansTail

✱ sort List / Merge sort

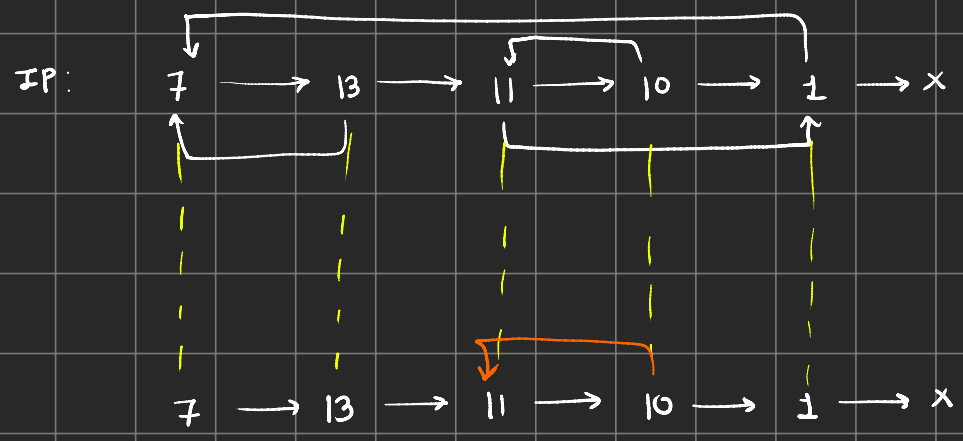
```
ListNode* findMid(ListNode* head){
    ListNode* slow = head;
    ListNode* fast = head->next;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

```
ListNode* sortList(ListNode* head) {

    if( head == NULL || head->next == NULL ){
        return head;
    }

    // break ll into two part
    ListNode* mid = findMid(head);
    ListNode* left = head;
    ListNode* right = mid->next;

    mid->next = NULL;

    // sort RE
    left = sortList(left);
    right = sortList(right);

    // megre left-right
    ListNode* mergeLL = merge(left, right);
    return mergeLL;
}
```

```
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {

    if (list1 == NULL)
        return list2;
    if (list2 == NULL)
        return list1;

    // Dummy node to simplify list construction
    ListNode* ansHead = new ListNode(-1);
    ListNode* ansTail = ansHead;

    while (list1 != NULL && list2 != NULL) {

        if (list1->val <= list2->val) {
            ansTail->next = list1;
            ansTail = list1;
            list1 = list1->next;
        } else {
            ansTail->next = list2;
            ansTail = list2;
            list2 = list2->next;
        }
    }

    // Attach the remaining part of the list
    if (list1 != NULL) {
        ansTail->next = list1;
    }
    if (list2 != NULL) {
        ansTail->next = list2;
    }

    // Return the merged list (skip the dummy node)
    return ansHead->next;
}
```

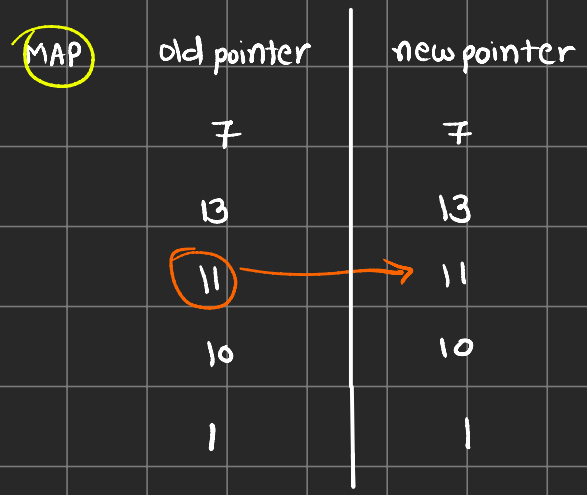
while (list1 != NULL)

clone LL with Random Pointer



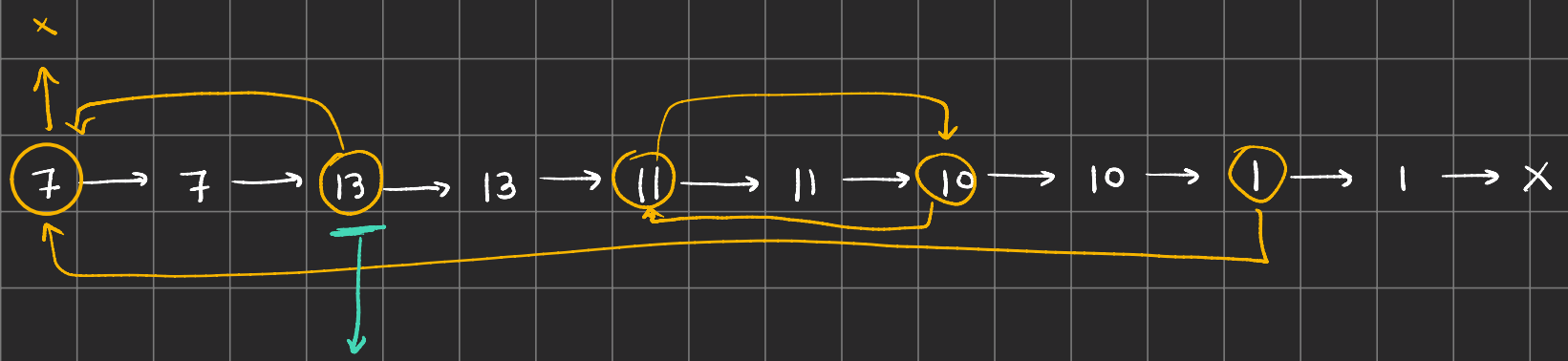
Purana 10 Purane 11 par point kar raha tha

so new 10 ko Bhi new 11 par point karakrna hai




```cpp
Node* helper(Node* head, unordered_map<Node*, Node*> &mp){

    if(head==NULL) return NULL;

    Node* newHead = new Node(head->val);
    mp[head] = newHead;
    newHead->next = helper(head->next, mp);

    if(head->random){
        // agar puranewala head kahi random point kar rha he
        newHead->random = mp[head->random];
    }

    return newHead;
}

Node* copyRandomList(Node* head) {

    unordered_map<Node*, Node*> mp;
    return helper(head, mp);
```

Purani head ka



old

new

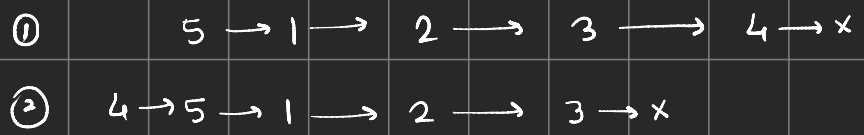
purane wale 13 ka
random pointer purane 7 par point

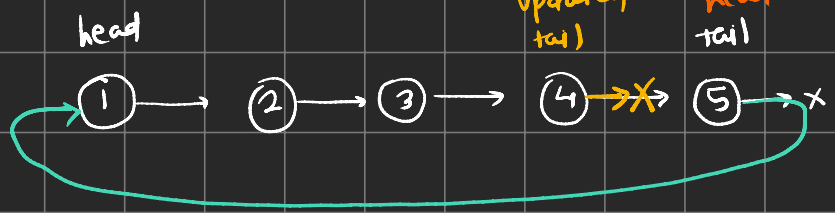
so purane 13 → next means new wala 13

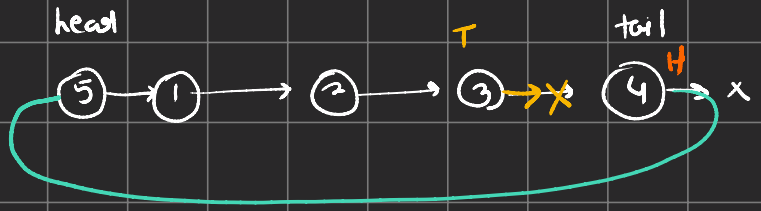
then purane 7 → next means new wala 7

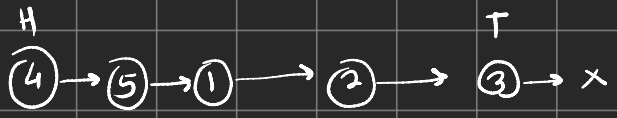
```
Node* copyRandomList(Node* head) {

    if(!head) return 0;

    // 1)clone A(old) -> A'(new)
    Node* it = head;

    while(it!=NULL){
        Node* clonedNode = new Node(it->val);
        clonedNode->next = it->next;
        it->next = clonedNode;
        it = it->next->next;
    }

    // 2) assign random links of A' with the help of A
    it = head;
    while(it){
        Node* clonedNode = it->next;
        clonedNode->random = it->random ? it->random->next : NULL;
        it = it->next->next;
    }

    // 3) detach both old and new
    it = head;
    Node* clonedHead = it->next;

    while(it){
        Node* clonedNode = it->next;
        it->next = it->next->next;
        if(clonedNode->next){
            clonedNode->next = clonedNode->next->next;
        }
        it = it->next;
    }

    return clonedHead;
```

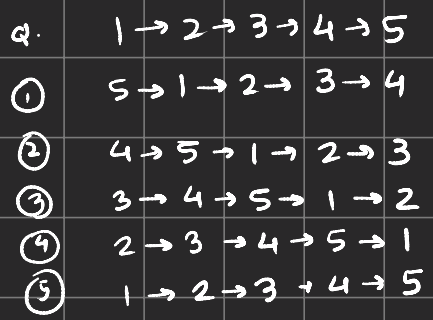
Rotate List

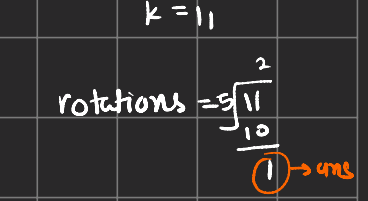












rotations = k % len

Remove TLE

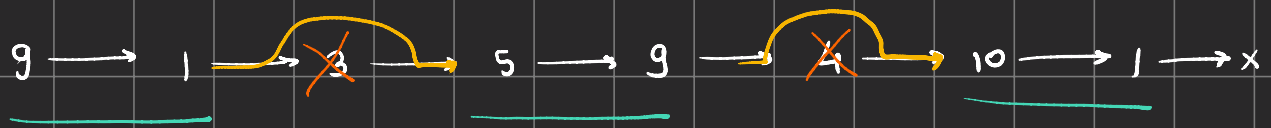
```
ListNode* getTail(ListNode* head){
    ListNode* temp = head;
    while(temp->next != NULL){
        temp = temp->next;
    }
    return temp;
}

int getLen(ListNode* head){
    ListNode* temp = head;
    int count = 0;
    while(temp != NULL){
        temp = temp->next;
        count++;
    }
    return count;
}

ListNode* getPrev(ListNode* head, ListNode* tail){
    ListNode* temp = head;
    while(temp->next != tail){
        temp = temp->next;
    }
    return temp;
}
```

```
ListNode* rotateRight(ListNode* head, int k) {

    if(head == NULL) return NULL;
    if(head->next == NULL) return head;

    // ListNode* head = head;
    ListNode* tail = getTail(head);

    int updatedK = k % getLen(head);   // Remove → TLE
    while(updatedK--){
        ListNode* prev = getPrev(head,tail);
        prev->next = NULL;

        ListNode* tailNode = tail;
        tailNode->next = head;

        tail = prev;
        head = tailNode;
    }

    return head;

}
```

✱ delete n nodes after m nodes

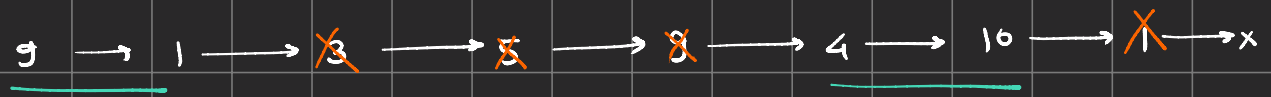
m = 2 ⇝ skip

n = 1 ⇝ delete



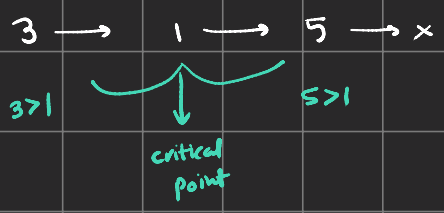
m = 2 ⇝ skip

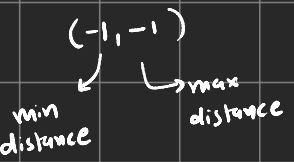
n = 3 ⇝ delete

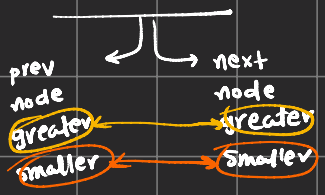


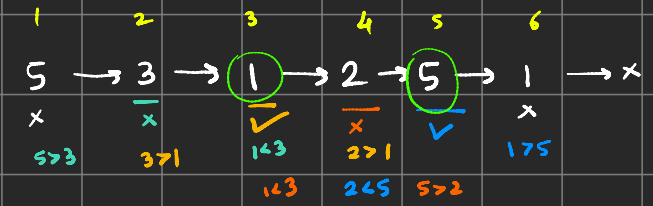

```
void linkdelete(struct Node  *head, int M, int N)
{
    if(!head) return;
    Node*it = head;
    for(int i=0;i<M-1;++i){
        // if M nodes are N.A.
        if(!it) return;
        it=it->next;
    }

    // it -> would be at Mth node;
    if(!it) return;

    Node*MthNode = it;
    it = MthNode->next;
    for(int i=0;i<N;++i){
        if(!it) break;

        Node*temp = it->next;
        delete it;
        it = temp;
    }
    MthNode->next = it;
    linkdelete(it, M, N);
}
```

★ Find the minimum and Maximum Number of Nodes Between Critical Points

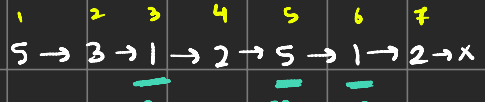








min = 5 - 3 = 2
max = 5 - 3 = 2



max = first CP - last CP
= 6 - 3 = 3

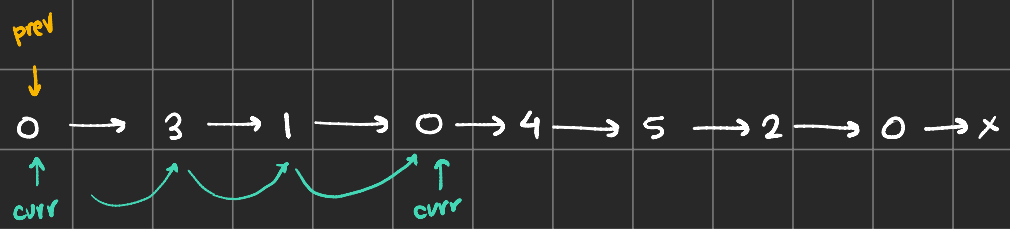
min = initial when we have CP1 that time
= after we got CP2 so
CP2 - CP1 = 5 - 3 = 2

= 2, min distance after we got CP3 ⇒ CP3 - CP2
⇒ 6 - 5
⇒ 1

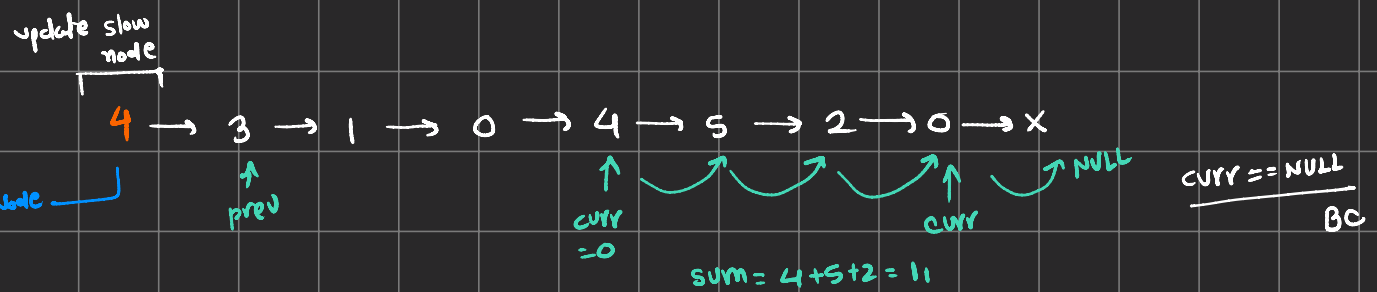
= min (2, 1)
= 1

```cpp
vector<int> nodesBetweenCriticalPoints(ListNode* head) {
    vector<int> ans = {-1, -1}; // minDist, maxDist

    ListNode* prev = head;
    if (prev == NULL) return ans;

    ListNode* curr = head->next;
    if (curr == NULL) return ans;

    ListNode* nextNode = head->next->next;
    if (nextNode == NULL) return ans;

    int firstCp = -1;
    int lastCp = -1;

    int minDist = INT_MAX;
    int i = 1;

    while (nextNode != NULL) {
        bool isCp = ((curr->val > prev->val && curr->val > nextNode->val) ||
                     (curr->val < prev->val && curr->val < nextNode->val)) ? true : false;

        if (isCp && firstCp == -1) {
            // first cp got
            firstCp = i;
            lastCp = i;
        } else if (isCp) {
            // means two cp mile he means one se jyada (two,three...)
            minDist = min(minDist, i - lastCp);
            lastCp = i;
        }
        i++;
        prev = prev->next;
        curr = curr->next;
        nextNode = nextNode->next;
    }

    if (firstCp == lastCp) {
        // only one cp found
        return ans;
    } else {
        ans[0] = minDist;
        int maxDist = lastCp - firstCp;
        ans[1] = maxDist;
        return ans;
    }
}
```

☆ Merge Nodes in between Zeros



sum with curr
= 0+3+1
= 4



curr == NULL
BC

and delete from prev pointer to that LL.

```cpp
ListNode* mergeNodes(ListNode* head) {

    ListNode* prev = head;
    ListNode* curr = head;

    ListNode* lastNode = NULL;

    while(curr!=NULL && curr->next != NULL){

        curr=curr->next;

        int sum = 0;
        for(int i=0; curr&&curr->val!=0; i++){
            sum = sum+curr->val;
            curr = curr->next;
        }

        lastNode = prev;
        lastNode->val = sum;
        prev = lastNode->next;

    }

    lastNode->next = NULL;
    cout << head->val << endl;

    return head;
}
```

memory leaks not handled here