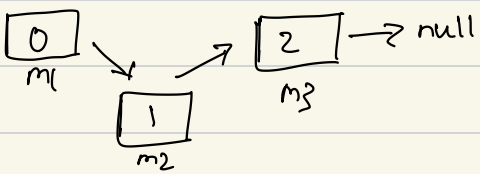
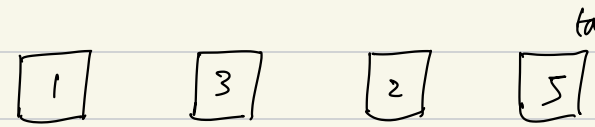
Linked Lists

(1) Introduction to LL:

Linked lists : Stored at a non-contiguous location



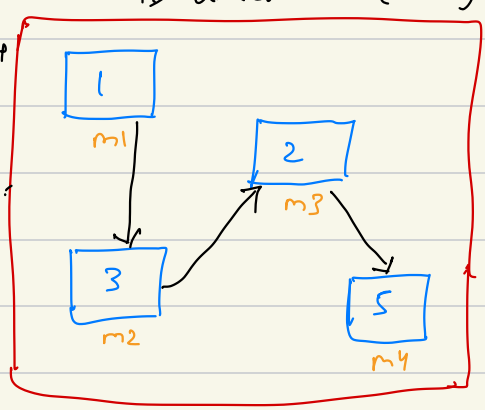
How do we store ?



→ JAVA

1D linked list (one-dir)

null.ptr (++



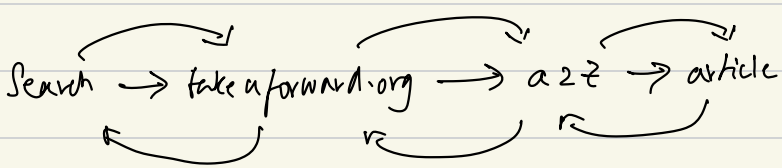
head = m1

tail = m4

It's used in **Stack / Queue**

In real life we use it on **Browser.**

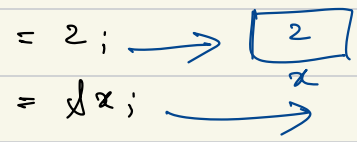
2D



D.S we are using is struct / class in C++ & Java

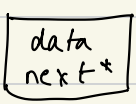
In C++

```
int x = 2;
int *y = &x;
cout << y; //pointer to the memory location.
```



What are we storing in linked list?

- data
- pointer to the next

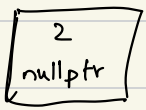


(self defined data type so struct)

```
Class
~~struct~~ Node {
    int data;
    Node * next;
    Node (data1, next1)      //constructor
    {
        data = data1;
        next = next1;
    }
}
```

How do we initialize?


```
Node x = Node (2, nullptr)
Node * y = &x ; //pointer to x.
```



easy way

```
Node* y = new Node (2, nullptr) ;
```

pointer to the memory location

```
vector<int> arr = {2,5,8,7}
```

```
cout << y ; // memory location 0x14aeo
cout << y->data ; // value (2)
```

Instead of struct use class

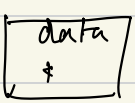
- using class you get access of OOPs.

In Java : (No ptr, only this keyword)

```
class Node {
    public:
        int data;
```

```
    Node next;
Node(int data1, Node next1){   // with two argument
    this.data = data1;
    this.next = next1;
}
Node(int data1){   // single argument
    this.data = data1;
    this.next = null;
}
}
cout << y.data  // 2
```

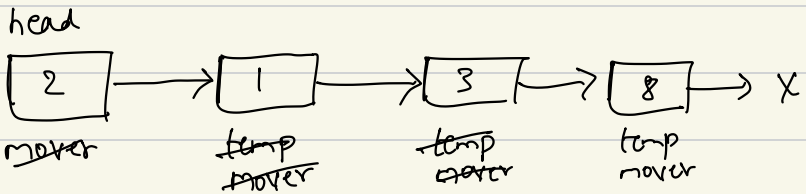
Memory Space :



32-bit	64-bit
int → 4 bytes	int → 4 bytes
* → 4 bytes	* → 8 bytes
8 bytes	12 bytes

Array → linked list

arr [] = [2, 1, 3, 8]



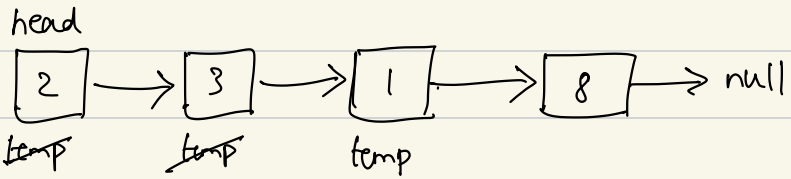
mover → next = temp

Code

```
Node * convertArr2LL (vector<int> & arr) {
    Node* head = new Node(arr[0]); // 1st node
    Node* mover = head;
    for(i=1; i<arr.size(); i++) {
        Node* temp = new Node(arr[i]);
        mover->next = temp;
        mover = temp; // mover = mover.next;
    }
    return head;
}
```

T.C ⇒ O(n)

Traversal in Linked list:



— Do not move head

```
int [] arr = {12, 5, 6, 8}
Node head = convertArr2LL(arr);

Node* temp = head
while (temp) {
    cout << temp -> data << " "; // 12, 5, 6, 8
    temp = temp -> next;
}
}
```

Length of LL

```
int lengthOfLL(Node* head) {
    int cnt = 0;
    Node* temp = head;
    while (temp) {
        cout << temp -> data << " ";
```

```
        temp = temp -> next;
        cnt++;
    }
    return cnt;   // 4
}
```

Search an Element :

```
int checkIfPresent(Node* head, int val){
    Node* temp = head;
    while(temp){
        if(temp -> data == val) return 1;
        temp = temp -> next
    }
    return -1;
}
```

T.C ⇒ O(N), O(1), O(N/2)

Code

Array to LL:

Introduction To Linked List

Easy • 0/40 • Average time to solve is 10m

Contributed by Prashant Thakur — 37 upvotes

Problem Statement — Send feedback

You are given an array *'Arr'* of size *'N'* consisting of positive integers.

Make a linked list from the array and return the head of the linked list.

The head of the linked list is the first element of the array, and the tail of the linked list is the last element.

Note:

In the output, you will see the elements of the linked list made by you.

Example:

Input: 'Arr' = [4, 2, 5, 1]

Output: 4 2 5 1

Explanation: Linked List for the array 'Arr' = [4, 2, 5, 1] is 4 -> 2 -> 5 -> 1.

```cpp
/**
 * Definition of linked list
 * class Node {
 *
 * public:
 *     int data;
 *     Node* next;
 *     Node() : data(0), next(nullptr) {}
 *     Node(int x) : data(x), next(nullptr) {}
 *     Node(int x, Node* next) : data(x), next
 * };
 */

Node* constructLL(vector<int>& arr) {
    Node* head = new Node(arr[0]);
    Node* mover = head;
    for(int i = 1; i < arr.size(); i++){
        Node* temp = new Node(arr[i]);
        mover->next = temp;
        mover = temp;
    }
    return head;
}
```

Count the length of LL:

codingninjas /studio — Interview Problem List

Topics | Problem | Submissions | Solutions | Discuss — C++ (g++ 5.4)

Count nodes of linked list

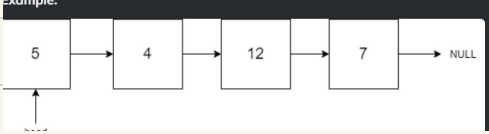
Easy • 40/40

Contributed by Ratnesh — 11 upvotes

Problem Statement — Send feedback

Given the head of a singly linked list of integers, find and return its length.

Example:




```cpp
int length(Node *head)
{
    int cnt = 0;
    Node* temp = head;
    while(temp) {
        temp = temp->next;
        cnt++;
    }
    return cnt;
}
```

Search in LL:

Topics | Problem | Submissions | Solutions | Discuss — C++ (g++ 5.4)

Search in a Linked List

40/40 • Average time to solve is 10m

Contributed by Arindam Majumder — 26 upvotes

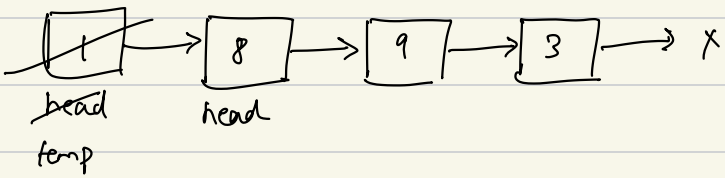
Problem Statement — Send feedback

You are given a Singly Linked List of integers with a head pointer. Every node of the Linked List has a value written on it.

```cpp
int searchInLinkedList(Node<int> *head, int k) {
    Node<int>* temp = head;
    while(temp){
        if(temp->data == k) return 1;
        temp = temp->next;
    }
    return 0;
}
```

L2) Deletion and Insertion in LL:

Delete the head of LL:



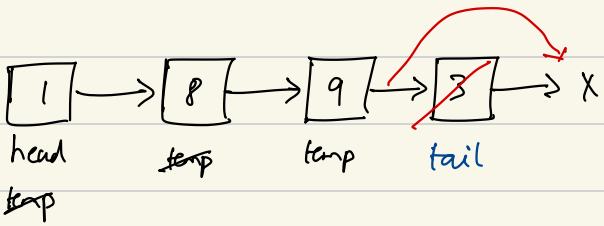
- Move the head to next
- You need the free the memory of prev head.

```
Node* deleteHead (Node * head)
{
    Node * temp = head;
        head = head -> next;
        free (temp); OR delete temp;
    return head;
}
```

→ For Java no need to write free or delete JAVA automatically does it.

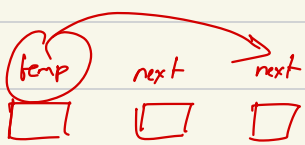
```
Node* removesHead(Node* head){
    if(head == NULL) return head;
    Node* temp = head;
    head = head->next;
    delete temp;
    return head;
}
```

Delete the tail of the LL:



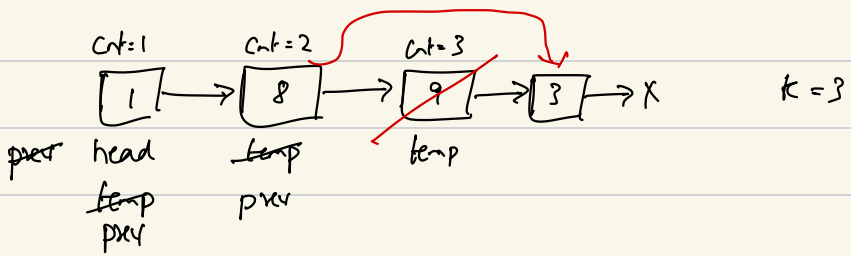
- we can point tail prev to the null.
- So you need to stop before the last element.
- For LL with 1 element we delete the 1 element

```
Node * deleteTail (Node * head)
{
    if (head == NULL ||
            head->next == null)
        return null.
    Node * temp = head;
    while (temp->next->next != null)
    {
        temp = temp->next;
    }
    free (temp->next);
    temp->next = nullptr;
    return temp;
}
```




```
Node *deleteLast(Node *head){
    if(head == NULL || head->next == NULL){
        return NULL;
    }
    Node* temp = head;
    while(temp->next->next != NULL){
        temp = temp->next;
    }
    delete temp->next;
    temp->next = nullptr;
    return head;
}
```

Delete the k^{th} element of the LL:




```
Node* deleteK(Node* head, int k)
{
    if(head == null) return head;
    if(k == 1) {
        Node* temp = head
        head = head->next;
        free(head)
        return head;
    }
k=3 cnt=0, Node* temp = head, prev = null
    while(temp != null)   // traverse
    {
        cnt++;
        if(cnt == k)
        {
            prev->next = prev->next->next
            free(temp);
        } break;
```

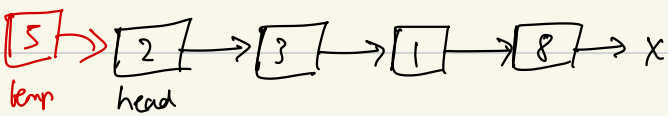
```
        prev = temp;
        temp = temp -> next;
    }
    return head
}
```

Remove the element from LL: (Same as above with few changes)

```
Node* removeEl (Node* head, int el) {
    if (head == null) return head;
    if (head -> data == el) {   // if head is the element
        Node* temp = head;
        head = head -> next;
        free(temp);
        return head;
    }
    Node* temp = head;
    Node* prev = null;
    while (temp != null) {
        if (temp -> data == el) {
            prev -> next = prev -> next -> next;
            free (temp),
            break;
        }
```

```
        prev = temp;
        temp = temp->next;
    }
    return head;
}
```

Insertion at the head:

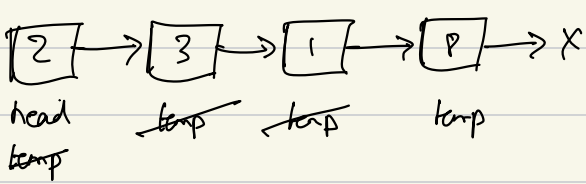


val = 5

→ Return the temp

```
Node* insertHead(Node* head, int val)
{
    Node* temp = new Node(val, head);
    return temp;
}
```

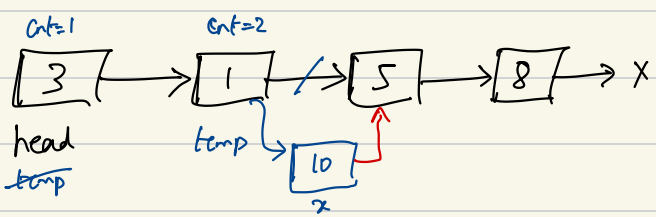
Insert at the last:



val = 5

```
Node* insertTail(Node* head, int val)
{
    if (head == null) return new Node(val);
    Node* temp = head;
    while (temp->next != null)
    {
        temp = temp->next  // should point at the last element
    }
    Node newNode = new Node(val);
    temp->next = newNode;
    return head;
}
```

Insert at the k^{th} element : k lies from (1 - N + 1)



el = 10, k

```
Node* insertAt(head, el, k)
{
    if (head == null) return new Node(el);
    if (k == 1) {   // insert at head
        Node* temp = new Node(el, val);
```

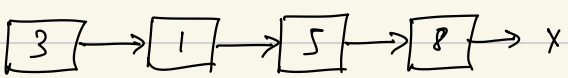
```
        return temp;
    }
```

if k=3

```
    cnt = 0 , temp = head
    while ( temp != null)
    {
        cnt ++;
        if (cnt == (k-1))
        {
            Node* x = new Node(val)
            x -> next = temp -> next // 1st connect 10 -> 5
            temp -> next = x // then connect 1 -> 10
            break;
        }
        temp = temp -> next
    }
    return head;
}
```

2 == 3-1 ✓

Insert element before the value x :



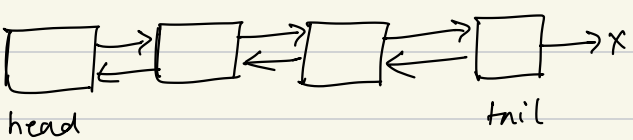
el = 10
val = 5

```
Node* insertBeforValue (Node* head, int el, int val) {
    if (head == null) return null;
    if (head -> data == val) return new Node (el, head);
```

```
Node* temp = head;
while (temp -> next != null) {
    if (temp -> next -> data == val) {
        Node* x = new Node (el, temp -> next);
        temp -> next = x;
        break;
    }
    temp = temp -> next;
}
return head;
}
```

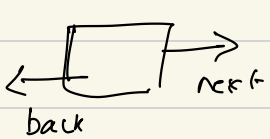
(3) Introduction to Double LL:

We can move in both the direction



Representation of DL

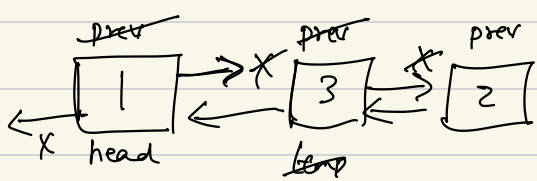
```
class Node {
public:
    int data;
    Node* next;
    Node* back;
```



```
public:
Node(int data1, Node* next1, Node* back1)
{
    data = data1;
    next = next1;
    back = back1;
}
public:
 Node (int data1)
 {
     data = data1;
     next = nullptr;
     back = nullptr
 }
};
```

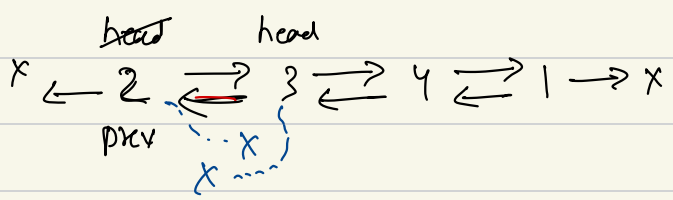
Arrays ⟶ DLL

arr[] = | 1 | 3 | 2 | 4 |



```
Node* convertArrToDLL(vector<int> &arr){
    Node* head = new Node(arr[0])
    Node* prev = head
    for(i=1 ⟶ n-1){
        Node* temp = new Node(arr[i], nullptr, prev)
        prev->next = temp
                prev = temp
    }
    return head;
}
```

Delete the head of DLL :



Edge case

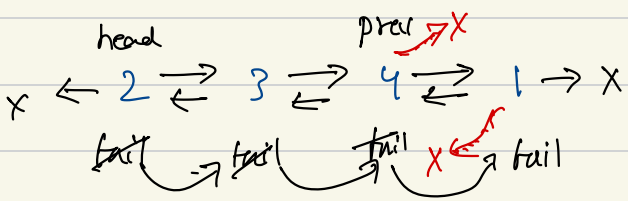
DLL → empty → return

DLL → 1 element → delete

```
prev = head
head = head->x
head->back = nullptr   // x ← [3]
prev->next = nullptr   // [2] → x
delete prev
```

```
Node* deleteHead(Node* head) {
    if (head == null || head->next == null) {
        return null;
    }
    Node* prev = head;
    head = head->next;
    head->back = nullptr;
    prev->next = nullptr;
    delete prev;
    return head;
}
```

Delete the tail of DLL:



Pseudo code

```
tail = head
while (tail->next != null) {
    tail = tail->next;
}
prev = tail->back
```

tail → back = nullptr // X ← [1]

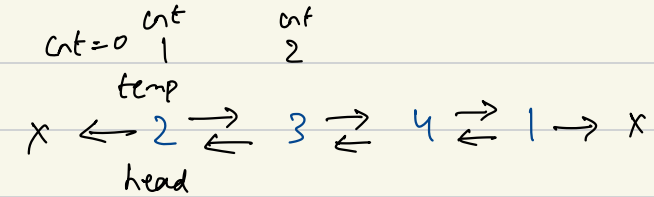
prev → next = nullptr // [4] → X

delete tail

Code

```
Node* deleteTail(Node* head){
    if(head == null || head->next == null){
        return null;
    }
    Node* tail = head;
    while(tail->next != null){
        tail = tail->next
    }
    Node* prev = tail->back;
    prev->next = nullptr;
    tail->back = nullptr;
    delete tail;
    return head
}
```

Delete the k^{th} node in DLL:



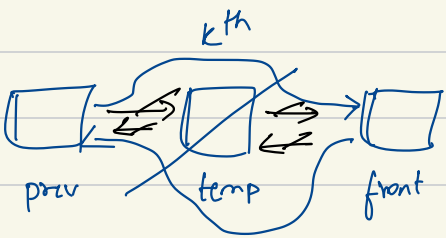
k = 1 to N

for k = N no next element
for k = 1
single element
⟹ three edge case

```
temp = head
cnt = 0
while(temp != null)
{
    cnt ++
    if(cnt == k) break;
    temp = temp -> next;
  }
prev = temp -> back;
front = temp -> next;
```


```
// edge case
if(prev == null && front == null) {
        delete temp;
  }     return null;
```

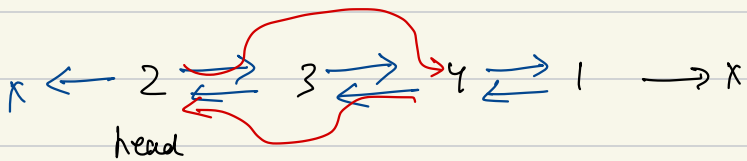
```
else if (prev == null)   // standing at head
{
    deleteHead(head)
    return
}
else if (front == null)  // standing at tail
{
    deleteTail(head)
    return
}
else {  // standing in b/w.
    prev -> next = front
    front -> back = prev
    temp -> next = nullptr
    temp -> back = nullptr
    delete (temp)
}
```

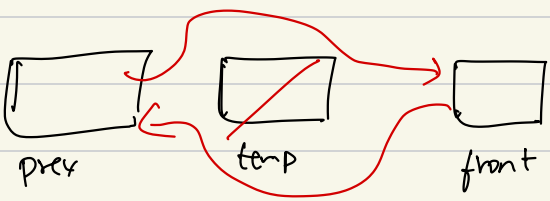


```
69  Node* removeKthElement(Node* head, int k) {
70      if(head == NULL) {
71          return NULL;
72      }
73      int cnt = 0;
74      Node* kNode = head;
75      while(kNode != NULL) {
76          cnt++;
77          if(cnt == k) break;
78          kNode = kNode->next;
79      }
80      Node* prev = kNode->back;
81      Node* front = kNode->next;
82
83      if(prev == NULL && front == NULL) {
84          return NULL;
85      }
86      else if(prev == NULL) {
87          return deleteHead(head);
88      }
89      else if(front == NULL) {
90          return deleteTail(head);
91      }
92      prev->next = front;
93      front->back = prev;
94
95      kNode->next = nullptr;
96      kNode->back = nullptr;
97      delete kNode;
98      return head;
```

Delete the Node of DLL:

Constraint:
Node is not head.

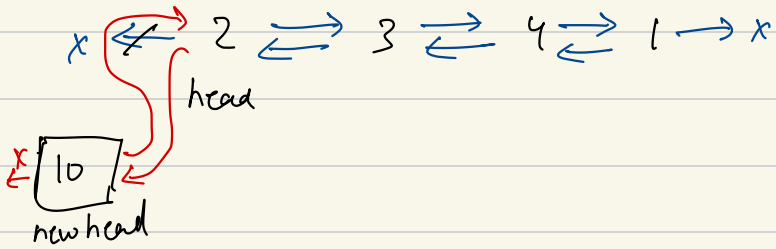





```
void deleteNode(Node * temp){
    Node* prev = temp -> back;
    Node * front = temp -> next;
//edge
    if(front == null){
        prev -> next = nullptr
        temp -> back = nullptr
        free(temp)
        return
    }
    prev -> next = front
    front -> back = prev

    temp -> next = temp -> back = nullptr;
```

```
free(temp)
}
```

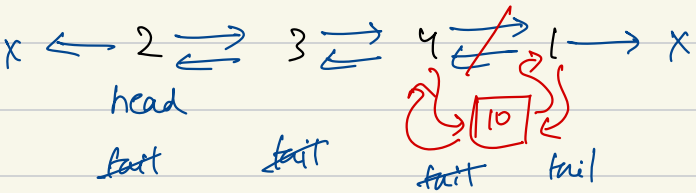
Insert before the head:



Val = 10

```
Node* newHead = new Node(val, head, nullptr)
head -> back = newHead
return newHead;
```

Insert before tail of DLL:



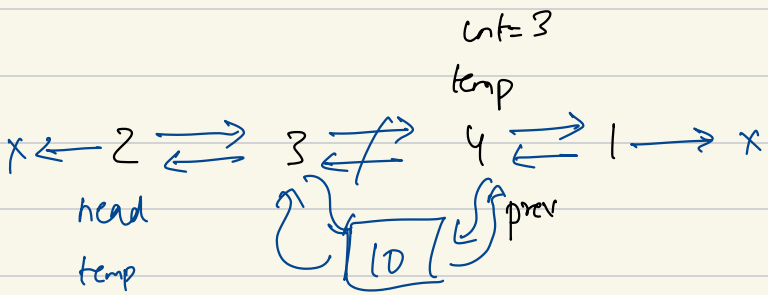
```
tail = head;
while(tail -> next != null){
        tail = tail -> next;
  }
  prev = tail -> back;
  newNode = newNode (10, tail, prev);
  prev -> next = newNode;
  tail -> back = newNode,
```

```
Node* insertBeforeTail(Node* head, int val) {
    if(head->next == NULL) {
        return insertBeforeHead(head, val);
    }
    Node* tail = head;
    while(tail->next != NULL) {
        tail = tail->next;
    }

    Node* prev = tail->back;
    Node* newNode = new Node(val, tail, prev);
    prev->next = newNode;
    tail->back = newNode;
    return head;
}
```

Insert before the k^{th} element

$k = 3$

$val = 10$



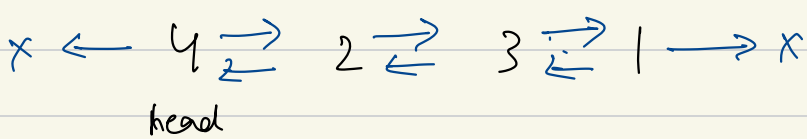
$k = 1$ // head of LL

$k = n$ // tail of LL

```
temp = head
cnt = 0
while (temp != null) {
    cnt = cnt + 1;
    if (cnt == k) break;
    temp = temp -> next;
}
prev = temp -> back;
```

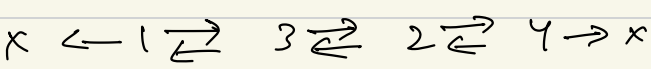
```
137  Node* insertBeforeKthElement(Node* head, int k, int val) {
138      if(k == 1) {
139          return insertBeforeHead(head, val);
140      }
141      Node* temp = head;
142      int cnt = 0;
143      while(temp != NULL) {
144          cnt++;
145          if(cnt == k) break;
146          temp = temp->next;
147      }
148      Node* prev = temp->back;
149      Node* newNode = new Node(val, temp, prev);
150      prev->next = newNode;
151      temp->back = newNode;
152      return head;
153  }
```

(4) Reverse a DLL



1st Approach (Stack)

- You can reverse the data

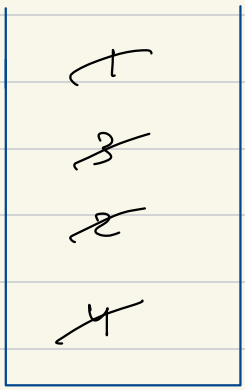


- You can use stack d.s (LIFO)

X ⇐ 4 ⇄ 2 ⇄ 3 ⇄ 1 → X

head temp temp temp temp

~~temp~~



(LIFO) Stack

- Now we will pop out the top element from the stack

So in 2nd iteration of temp we take the top element and replace the L.L.

X ⇐ 1 ⇄ 3 ⇄ 2 ⇄ 4 → X

head temp temp temp

~~temp~~

Pseudo Code

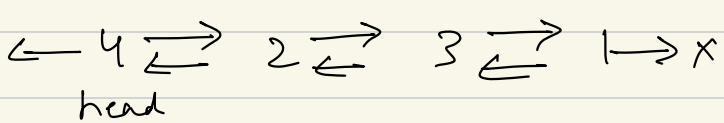
```
Node * temp = head;
Stack<int> st;
```

```
while (temp != null)
{
    st.push(temp -> data)
    temp = temp -> next   // we will add all the element to the stack
}

temp = head
while (temp != null)
{
    temp -> data = st.top();
    st.pop();
    temp = temp -> next
}
```

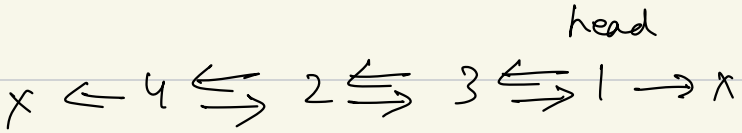
T.C ⇒ $O(N) + O(N)$ = $O(2N)$

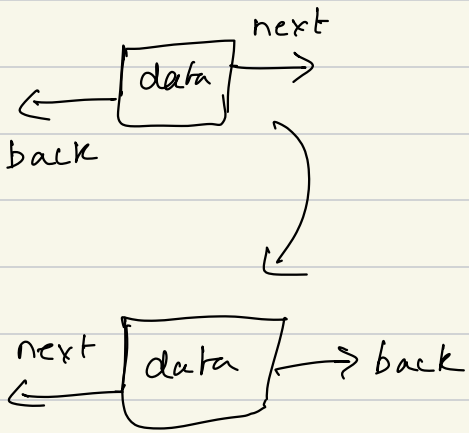
S.C ⇒ $O(N)$ // stack

2nd Approach



- We have to reverse the link





- we need to swap next & back pointer
- by using swap().

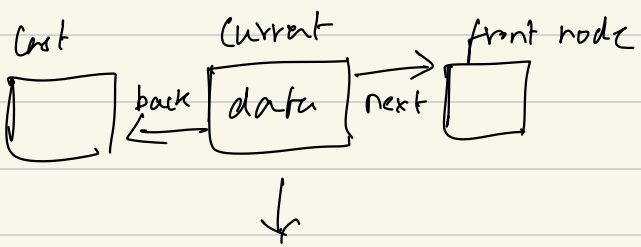
For e.g.

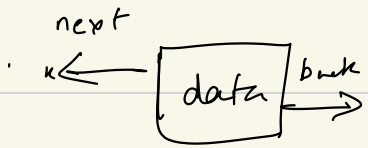
a = 2 b = 5

temp = a * We need to preserve a as a is overwritten.

a = b

b = temp





Current.next = last

current.back = front

(you're using two variable)

Can be done using one variable

last = null, curr = head

while (current != null)

{ last = current → back

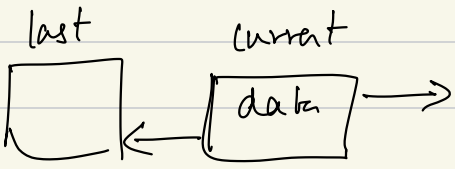
current → back = current → next

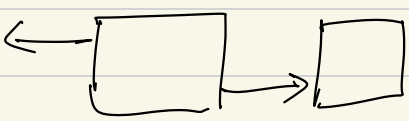
current → next = last

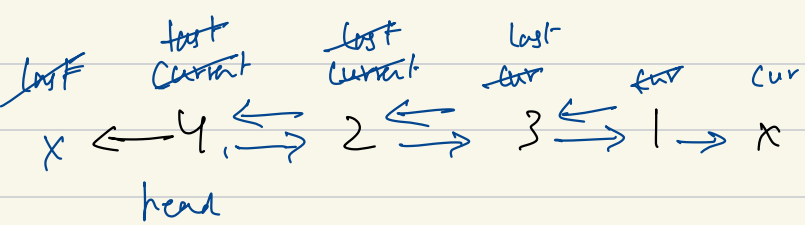
In order to go next element

} current = current → back

last → back = newHead







T.C ⇒ O(N) S.C ⇒ O(1)

```
31 Node *reverseDLL(Node *head) {
32   if (head == NULL || head->next == NULL) {
33     return head;
34   }
35   Node *prev = NULL;
36   Node *current = head;
37   while (current != NULL) {
38     prev = current->prev;
39     current->prev = current->next;
40     current->next = prev;
41     current = current->prev;
42   }
43   return prev->prev;
44 }
```

15) Adding two numbers in LL

head1

2 → 4 → 6 → X L1 = 6 4 2

head2

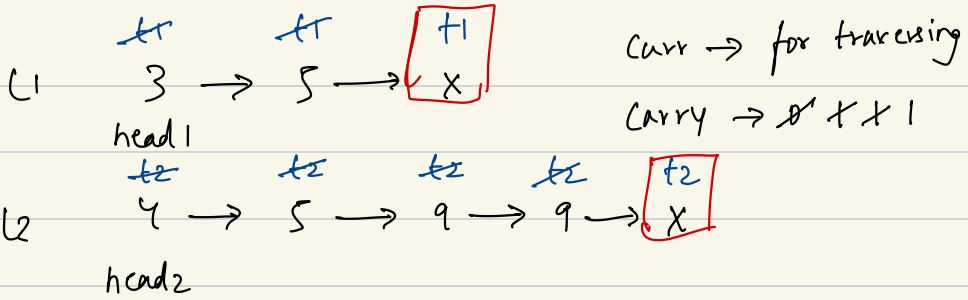
3 → 8 → 7 → X L2 = 7 8 3

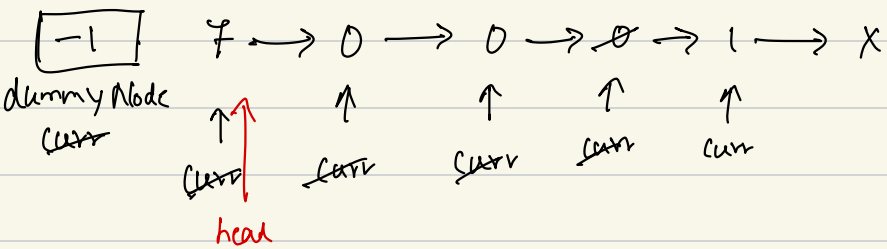
14 2 5

5 → 2 → 4 → 1 (sum list in reverse order)

head

- While addition you start from unit place, so your first head is unit place, if there is carry over you take it.





So dummyNode → next = head

Pseudo Code:

```
func(head1, head2)
{  t1 = head1 , t2 = head2
   dummyNode = new Node(-1).
   curr = dummyNode , carry = 0
   while(t1 != null || t2 != null)
   {
      sum = carry;
      if(t1)   sum = sum + t1 → data
      if(t2)   sum = sum + t2 → data
```

newNode = new Node (sum % 10)

carry = sum / 10;

curr → next = newNode

curr = curr → next

if (l1) l1 = l1 → next

if (l2) l2 = l2 → next

}

if (carry) { // if carry is left out at last

newNode = new Node(carry)

curr → next = newNode;

}

return dummyNode → next;

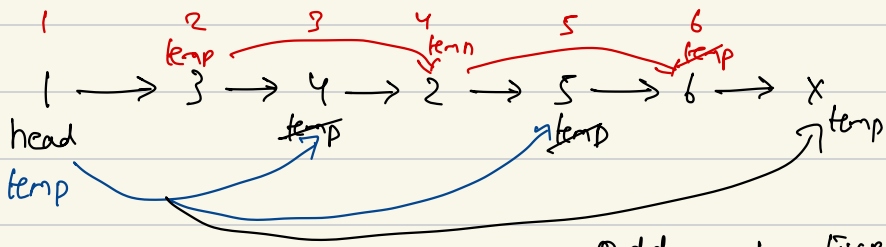
TC ⇒ O(max (n1, n2)) S.C ⇒ O(max(n1, n2))

```cpp
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummyNode = new ListNode(-1);
        ListNode* curr = dummyNode;
        ListNode* temp1 = l1;
        ListNode* temp2 = l2;
        int carry = 0;

        while(temp1 != NULL || temp2 != NULL){
            int sum = carry;
            if(temp1) sum += temp1->val;
            if(temp2) sum += temp2->val;
            ListNode* newNode = new ListNode(sum % 10);
            carry = sum / 10;

            curr->next = newNode;
            curr = curr->next;
            if(temp1) temp1 = temp1->next;
            if(temp2) temp2 = temp2->next;
        }
        if(carry){
            ListNode* newNode = new ListNode(carry);
            curr->next = newNode;
        }
        return dummyNode->next;
    }
};
```

```cpp
Node* addTwoNumbers(Node* num1, Node* num2) {
        Node* dummyNode = new Node(-1);
        Node* curr = dummyNode;
        Node* temp1 = num1;
        Node* temp2 = num2;
        int carry = 0;

        while(temp1 != NULL || temp2 != NULL){
            int sum = carry;
            if(temp1) sum += temp1->data;
            if(temp2) sum += temp2->data;
            Node* newNode = new Node(sum % 10);
            carry = sum / 10;

            curr->next = newNode;
            curr = curr->next;
            if(temp1) temp1 = temp1->next;
            if(temp2) temp2 = temp2->next;
        }
        if(carry){
            Node* newNode = new Node(carry);
            curr->next = newNode;
        }
        return dummyNode->next;
    }
```

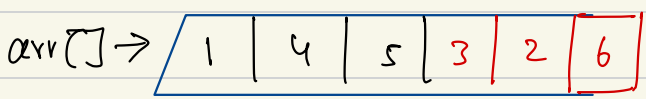
result

16) Odd Even Linked Lists :



O/P 1 → 4 → 5 | Odd → 3 → 2 → 6 | Even

i) Brute Force Data replacement



Lists

First iteration for the odd index

get the odd index first by jumping two times

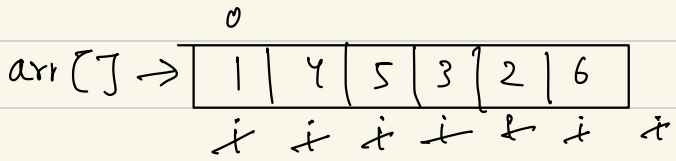
by next → next $O(n/2)$

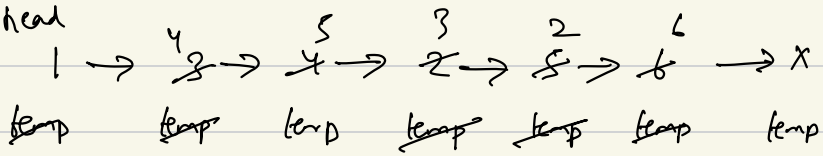
once it reach null we stop.

Second iteration for the even index

- keep the temp next to the head (temp → head.next)
- jump two times next → next.

- once it reach last node you stop.





Pseudo Code Edge case if (head == null || head → next == null)
return head;

```
arr → []   temp = head
while ( temp != null && temp → next != null)
{
    arr . add (temp → data)
    temp = temp → next → next
}
if (temp) arr.add (temp → data)   // for odd length L.L. we might skip the last element.
```

Odd
O(n/2)

```
temp = head → next
while (temp != null && temp.next != null)
{
    arr . add (temp → data)
    temp = temp → next → next
}
```

Even

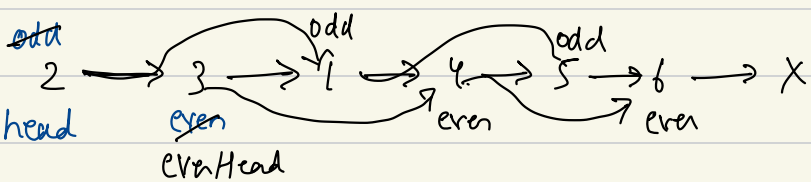
if (temp) . arr. add (temp → data) // in case of last node.

```
i = 0    temp = head
while( temp != null) {
    temp → data = arr [i]
        i++;
    temp = temp →next;
}
return head;
```

O(N)

T.C => O(2N) S.C => O(N)

2) Optimised Approach

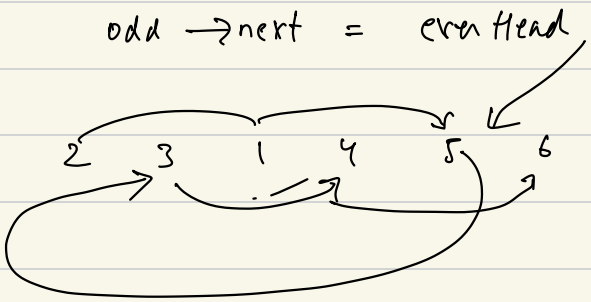


- changing the odd link
- changing the even link

odd → next = odd → next → next

event → next = event → next → next

The last odd should point to 1st even node, as we don't remember even, so memoize it before breaking the link. `head -> next = evenHead`



Pseudo Code

```
odd = Head      even = head -> next      evenHead = head -> next
while( even != null || even.next != null) // as even is bound to be at last.
{
    odd -> next = odd -> next -> next
    even -> next = even -> next -> next

    odd = odd -> next
    even = even -> next
}
odd.next = evenHead
return head
```

~~T.C => O(n²)~~ S.C => O(1)

```java
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if(head == null || head.next == null || head.next.next == null){
            return head;
        }
        ListNode odd = head;
        ListNode even = odd.next;
        ListNode evenhead = even;

        while(even != null && even.next != null){

            odd.next = odd.next.next;
            even.next = even.next.next;

            odd = odd.next;
            even = even.next;
        }
        odd.next = evenhead;
        return head;
    }
}
```

(7) Sort a L.L of 0's, 1's and 2's

Problem statement Send feedback

Given a linked list of *'N'* nodes, where each node has an integer value that can be 0, 1, or 2. You need to sort the linked list in non-decreasing order and the return the head of the sorted list.

Example:

Given linked list is 1 -> 0 -> 2 -> 1 -> 2.
The sorted list for the given linked list will be 0 -> 1 -> 1 -> 2 -> 2.

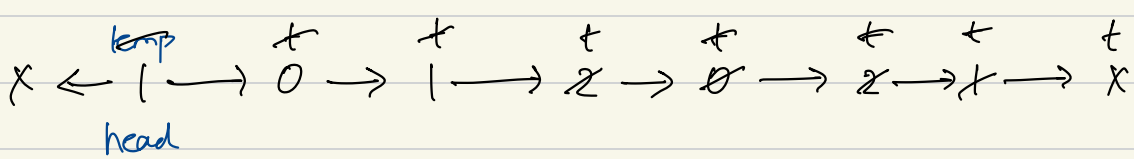
Detailed explanation (Input/output format, Notes, Images)

Sample Input 1:

7
1 0 2 1 0 2 1

Sample Output 1:

0 0 1 1 1 2 2



1) Brute Force Data Replacement

cnt 0 = 0x2 cnt 1 = 0x23 cnt 2 = 0x2

One more traversal for data replacement

cnt 0 = 2x0 cnt 1 = 3x0 cnt 2 = 2x0

temp t t t t t t t t

0 → 0 → 0 → 1 → 1 → 1 → 2 → 2 → X

head

Pseudo Code

temp = head cnt0 = 0 cnt1 = 0 cnt2 = 0

1st traversal while(temp != null)

{

if (temp → data == 0) cnt0 ++

else if (temp → data == 1) cnt1 ++

else (temp → data == 2) cnt2 ++

temp = temp → next

}

2nd traversal

temp = head

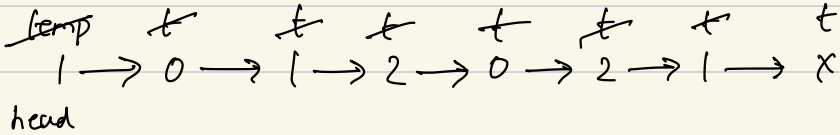
```
while (temp != null)
{
    if (cnt0)
        temp -> data = 0;
        cnt0--;
    else if (cnt1)
        temp -> data = 1;
        cnt1--;
    else
        temp -> data = 2;
        cnt2--;
    temp = temp -> next;
}
```

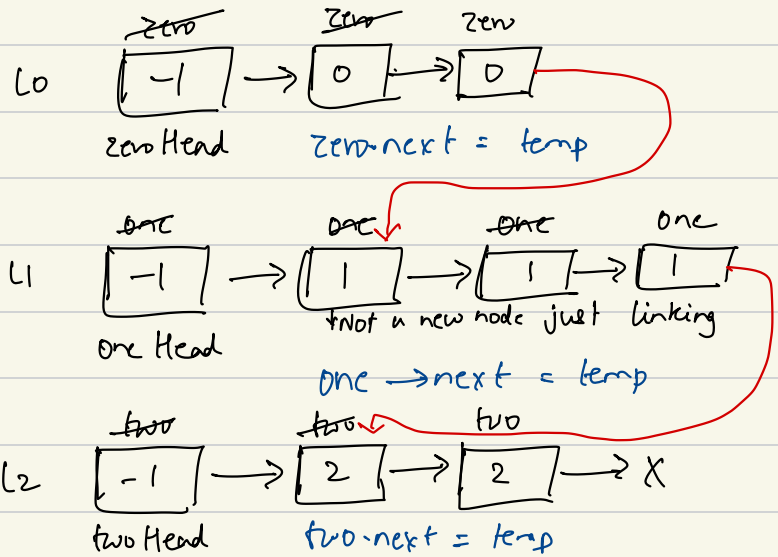
T.C ⇒ O(2N) S.C ⇒ O(1)

2 pass ⟶ 1 pass

↓

link change.





Pseudo Code

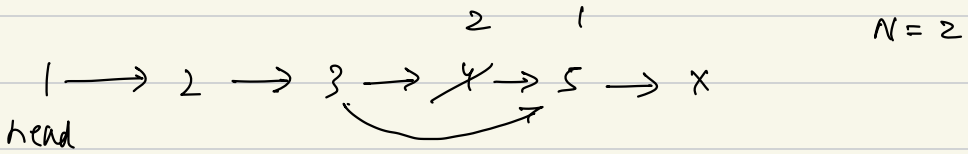
```
if (head == null || head → next == null) return head;
zeroHead = new Node(-1) , zero = zeroHead
oneHead = new Node(-1)  one = oneHead
twoHead = new Node(-1) ; two = twoHead

temp = Head
while (temp != NULL)
{
    if (temp → data == 0)
    {
```

```
            zero -> next = temp

            zero = temp;
        }
    else if (temp -> data == 1)
        {
            one -> next = temp

                one = temp
        }
        else {
                two -> next = temp

                    two = temp
            }
        temp = temp -> next;
    }
    //Now linking 0 -> 1 -> 2 if it's there,
    zero -> next = (one Head -> next) ? (one Head -> next) :
                                        (two Head -> next)

    one -> next = two Head -> next
    two -> next = null
    new Head = zero Head -> next
} delete (dummy Node All)
return new Head
```

T.C ⇒ O(n) S.C ⇒ O(1)

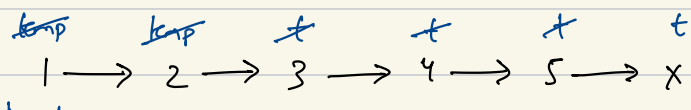
```cpp
Node* sortList(Node* head) {
    if (!head || !head->next) return head;
    Node* zeroHead = new Node(-1);
    Node* oneHead = new Node(-1);
    Node* twoHead = new Node(-1);
    Node* zero = zeroHead;
    Node* one = oneHead;
    Node* two = twoHead;
    Node* temp = head;

    while (temp) {
        if (temp->data == 0) {
            zero->next = temp;
            zero = zero->next;
        } else if (temp->data == 1) {
            one->next = temp;
            one = one->next;
        } else {
            two->next = temp;
            two = two->next;
        }
        temp = temp->next;
    }
    zero->next = (oneHead->next) ? (oneHead->next) : (twoHead->next);
    one->next = twoHead->next;
    two->next = NULL;

    Node* newHead = zeroHead->next;
    delete zeroHead;
    delete oneHead;
    delete twoHead;
    return newHead;
}
```

18) Remove n^{th} Node from end of the L.L :



1) Brute Force

cnt = 0 (count the length of L.L)



cnt = 0 1 2 3 4 5

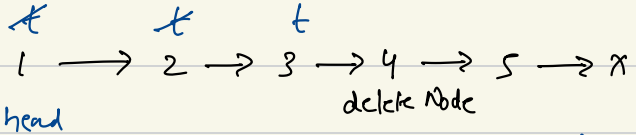
To reach the N^{th} node to delete = cnt − n

= 5 − 2

= 3

res = 3

- Start iterating again



res = 3 2 1 0 // once it reach '0' you stop the temp

- and now `delckNode = temp -> next`
- Linking `temp -> next = temp -> next -> next`
- `free (delckNode)`
- Now if the N is at head then simple you do `head = head -> next`

Pseudo Code

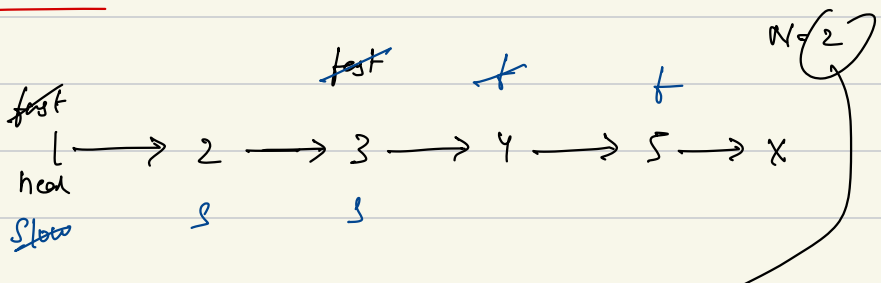
```
cnt=0        temp = head
while (temp != NULL)
{
    cnt++
    temp = temp->next
}
if (cnt == N) // the head
{
    newHead = head -> next;
    free (newHead);
    return newHead;
}
res = cnt - N      temp = head
while (temp != null) {
    res--;
```

```
        if (res == 0):
            break;
        temp = temp -> next;
    }
    deleteNode = temp -> next;
    temp -> next = temp -> next -> next;
    free (deleNode)
    return head;
}
```

T.C $\Rightarrow$ $O(len) + O(len - N)$ S.C $\Rightarrow O(1)$

$\approx O(2\,len)$

Optimial Solution



- Initialize move fast = N times. So now f $\Leftrightarrow$ f.next
- Now slow = head, now move slow, head simentaously one by one until f -> next = null.
- Now you can see slow -> next = deleteNode

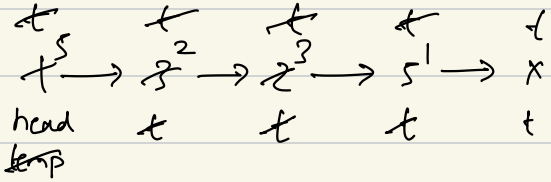
Pseudo Code

```
fast = head   // n steps we have to move
for (i = 0 -> n)  fast = fast -> next
slow = head
while (fast -> next != null)
{
    slow = slow . next    fast = fast . next
}  ---> if (fast == null) { return head = head -> next)
                            // edge case.
delNode = slow . next
slow . next = slow . next . next
delete (delNode)
return head
```

T.C => O(n) S.C => O(1)

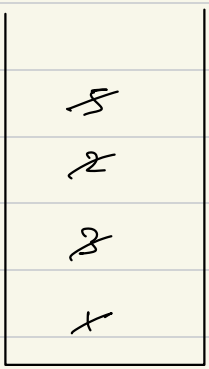
```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int k) {
        ListNode* fast = head;
        ListNode* slow = head;
        for(int i = 0; i < k; i++) fast = fast->next;
        if(fast == NULL) return head->next; // edge case
        while(fast->next != NULL){
            fast = fast->next;
            slow = slow->next;
        }
        ListNode* delNode = slow->next;
        slow->next = slow->next->next;
        delete(delNode);
        return head;
    }
};
```

L4) Reverse a L.L.



1) Brute force Data replacement

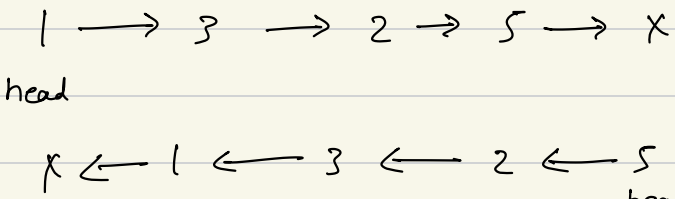
- Push all the element to stack
- Again take "temp" pop out the top element to the L.L.

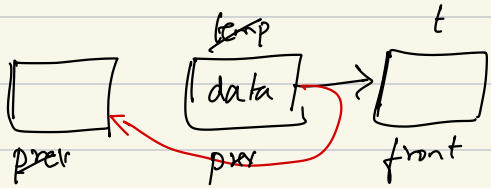


Pseudo Code

```
temp = head, stack st,
while (temp ! null)
{
    st.push(temp -> data)      } S-1
    temp = temp -> next
}
temp = head
while (temp != null)
{
    temp -> data = st.top()    } S-2
    st.pop()                   T.C => O(2N)
    temp = temp -> next        S.C => O(N)
}
```

2) Optimal Approach Link change :



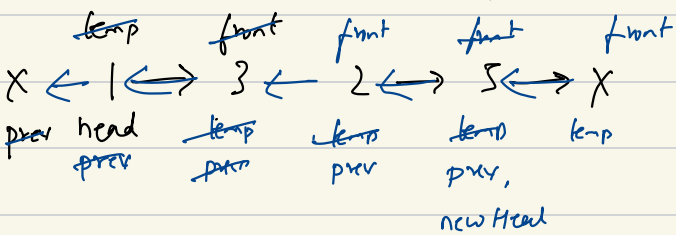


front = temp → next // Store the front

temp → next = prev

prev = temp

temp = front

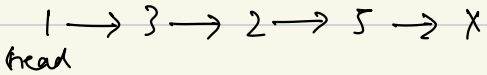



```
while(temp != null)
{
    front = temp → next
    temp → next = prev
    prev = temp
    temp = front
}
return prev (newHead)
```

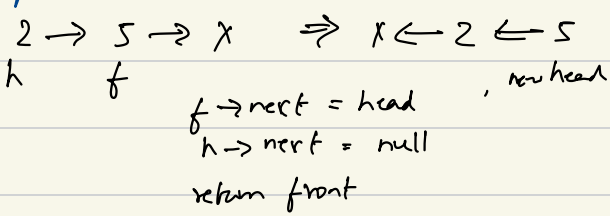
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* temp = head;
        ListNode* prev = NULL;
        while(temp != NULL){
            ListNode* front = temp->next;
            temp->next = prev;
            prev = temp;
            temp = front;
        }
        return prev;
    }
};
```

T.C => O(n) S.C => O(1)

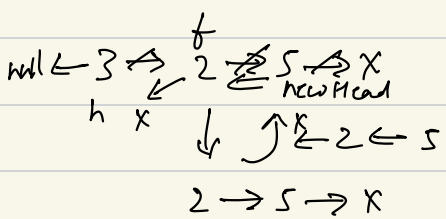
3) Recursion Solution



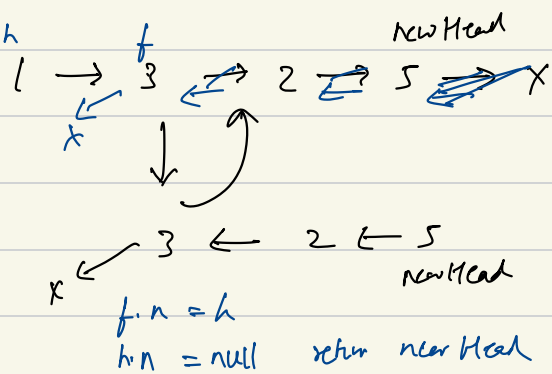
for 2 nodes

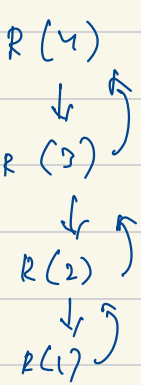


for 3 nodes



for 4 nodes



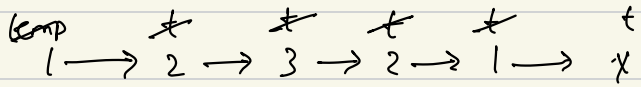


```
reverse (head)
{
    //base case
    if (head == null || head->next == null) return head;

    Node* newHead = reverse (head -> next)
    Node* front   = head -> next;
          front->next = head;
          head ->next = null;
          return newHead
}
```

```cpp
*/
//  Recursive Solution
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        // base case
        if(head == NULL || head->next == NULL){
            return head;
        }
        // recursive case
        ListNode* newHead = reverseList(head->next);
        ListNode* front = head->next;
        front->next = head;
        head->next = NULL;
        return newHead;
    }
};
```

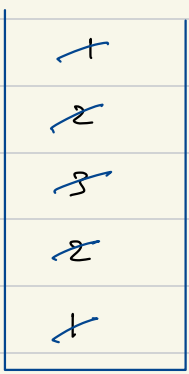
(10) Check if a LL is palindrome:

i) Brute force



1 2 3 2 1 ⇒ True

- Now compare the top element with the L.L.



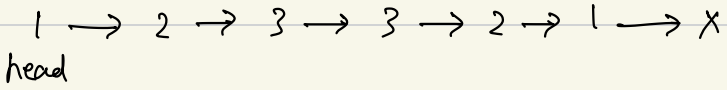

```
Stack st, temp = head
while (temp != NULL)
{
    st.push(temp -> data)
    temp = temp -> next
}
temp = head
while (temp != null)
{
    if (temp -> data != st.top()) return false;
    temp = temp -> next;
    st.pop();
}
```

S-1

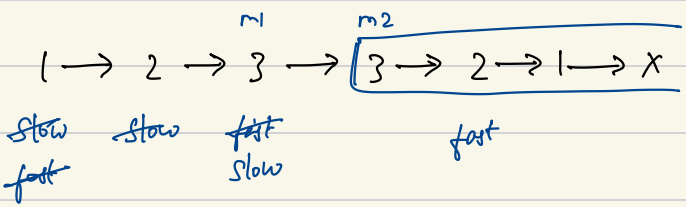
return true;

T.C ⇒ O(2N) S.C ⇒ O(N)

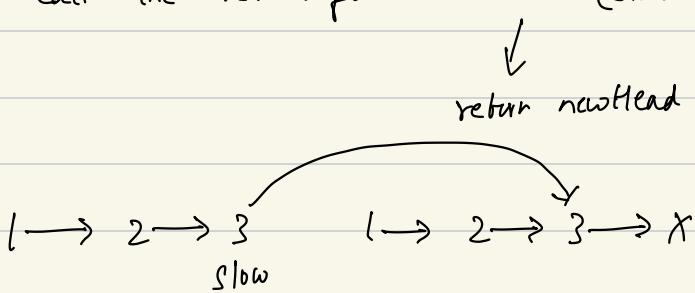
2) Optimial Solution :



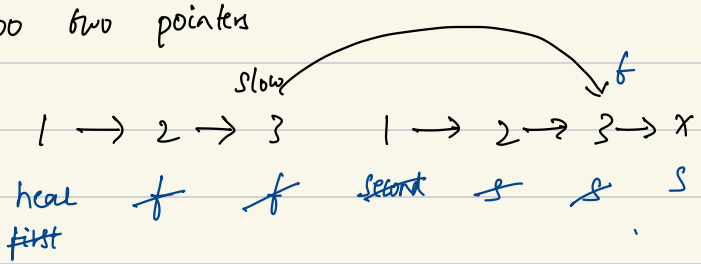
- In case of even linked LL, you can compare 1st half & 2nd half.
- Go to the middle of L.L (using tortoise & Hare Algo)



- Now you can reverse the link of 2nd half
- Call the reverse function reverse (slow → next)



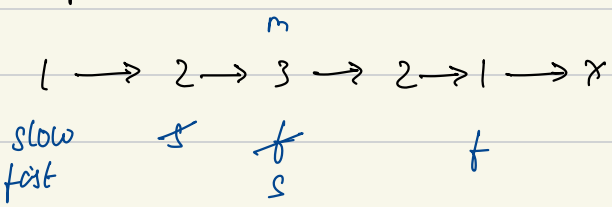
- Two two pointers



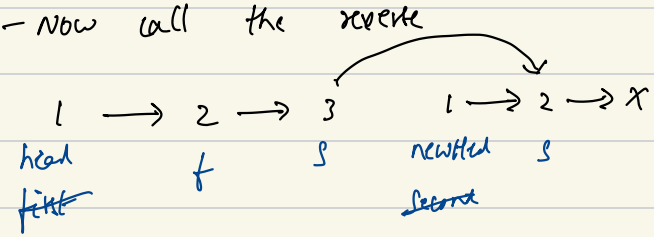
- Now keep compare first & second and move.
- after compare reverse back to (reverse (newHead)) original L.L.

1 → 2 → 3 → 3 → 2 → 1 → x

Now for odd L.L:



- Now call the reverse



- Compare & reverse back to original L.L

// finding the middle

```
slow = head , fast = head
while (fast.next != null && fast.next.next != null)
{
    slow = slow -> next
    fast = fast -> next -> next
}
```

// Reverse the 2nd half.

```
Node* newHead = reverse (slow -> next);
```

// comparing

```
first = head        second = newHead
while (second != null)
{
    if (first -> data != second -> data)
    {
        reverse (newHead)
        return false
    }
    first = first -> next;
    second = second -> next;
}
reverse (newHead); return true.
```

T.C => $O(n/2) + O(n/2) + O(n/2) + O(n/2)$

=> $O(2n)$

S.C => $O(1)$

C++ | Auto

```cpp

class Solution
{

private:
    ListNode *reverseList(ListNode *head)
    {
        // base case
        if (head == NULL || head->next == NULL)
        {
            return head;
        }
        // recursive case
        ListNode *newHead = reverseList(head->next);
        ListNode *front = head->next;
        front->next = head;
        head->next = NULL;
        return newHead;
    }

public:
    bool isPalindrome(ListNode *head)
    {
        if (head == NULL || head->next == NULL)
            return true;

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

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

        slow->next = reverseList(slow->next);
        slow = slow->next;
        ListNode *dummy = head;

        while (slow != NULL)
        {
            if (dummy->val != slow->val)
                return false;
            dummy = dummy->next;
            slow = slow->next;
        }
        return true;
    }
};
```

(ii) Add 1 to a number

head

1 → 5 → 9

o/p 1 → 6 → 0

head

159
1
160

1 → 5 → 9

1
1 5 9
1
1 6 0

- Backward traversal ⇒ But it's a single linked lists.
- So you reverse a L.L.

head

~~9~~0 → ~~5~~6 → 1 → X

temp

carry = ~~1~~ + 0

if carry = 0, break out

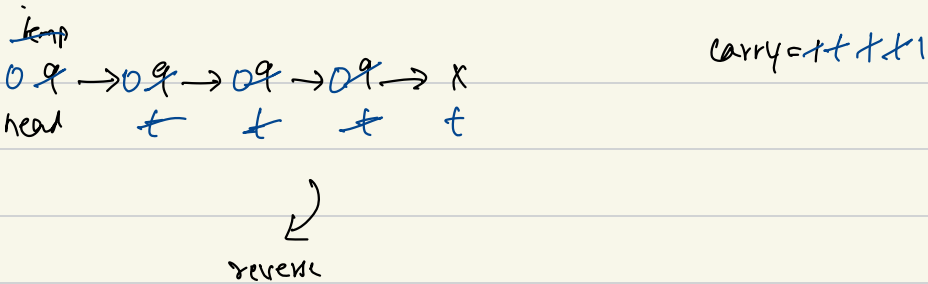
- Again call reverse

1 → 6 → 0 → X

one more e.g

9 → 9 → 9 → 9 → X

reverse



- Still carry is present, so creak a new node and point it to the head.

```
func (head)
{
    head = reverse (head)
    temp = head , carry = 1
    while (temp != null)
    {
        temp.data = temp.data + carry
        if (temp -> data < 10)
        {
            carry = 0
            break;
        }
```

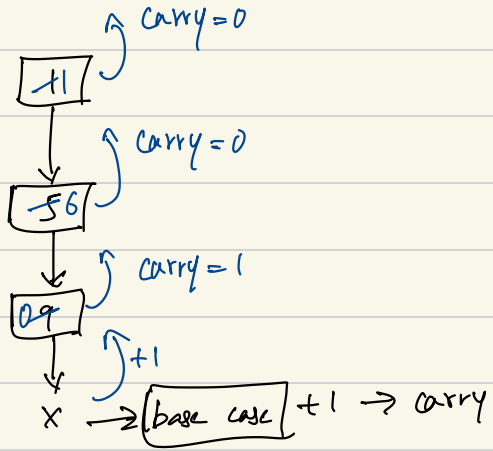
```
        else {
            temp.data = 0
            carry = 1
        }
        temp = temp.next
    }
    if (carry == 1)
    {
        Node newNode = new Node (1)
        head = reverse (head)
        newNode -> next = head
        return newNode
    }
    head = reverse (head) ; // if carry is '0'.
    return head ;
}
```

T.C ⇒ O(3N) S.C ⇒ O(1)

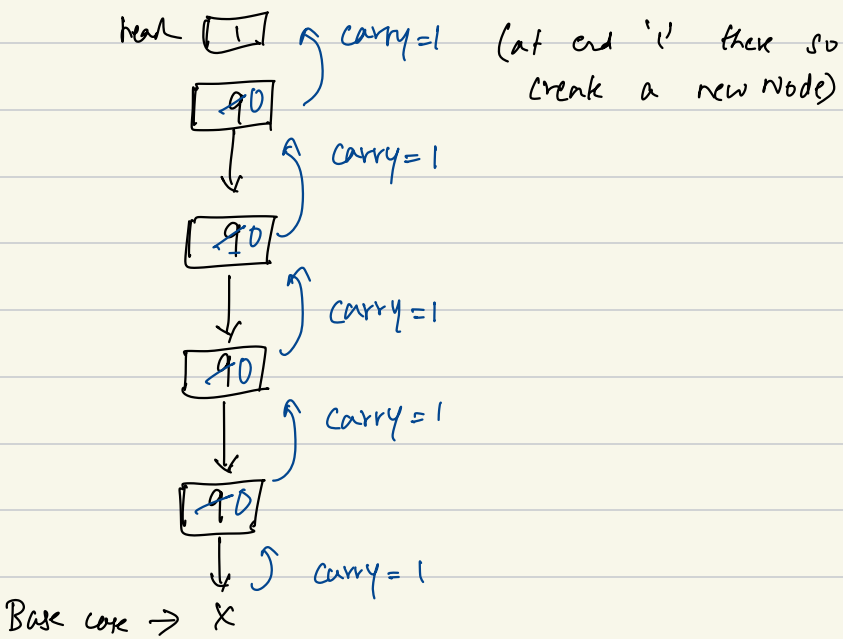
Recursive Solution without reversing :

1 → 5 → 9 → X

head


carry = 0
11
carry = 0
56
carry = 1
09
+1
X
base case
+1 → carry

9 → 9 → 9 → 9 → X


head
1
carry = 1
(at end '1' there so create a new node)
90
carry = 1
90
carry = 1
90
carry = 1
90
carry = 1
Base case → X

```
func (head)
{
    carry = helper (head)
    if (carry == 1) {
        newNode = newNode (1)
        newNode.next = head
        return newNode
    }
    return head
}
helper (temp)
{
    if (temp == null)
            return 1;
    carry = helper (temp -> next)
    temp.data = temp.data + carry
    if (temp -> data < 10)
            return 0;
    temp -> data = 0;
    return 1;
}
```

12) Find the intersection point of 4 LL:

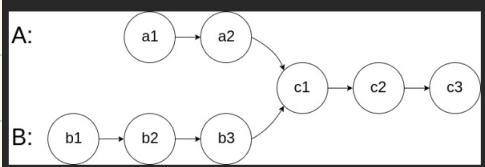
160. Intersection of Two Linked Lists

Easy 14.3K 1.3K

Companies

Given the heads of two singly linked-lists `headA` and `headB`, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return `null`.

For example, the following two linked lists begin to intersect at node `c1`:



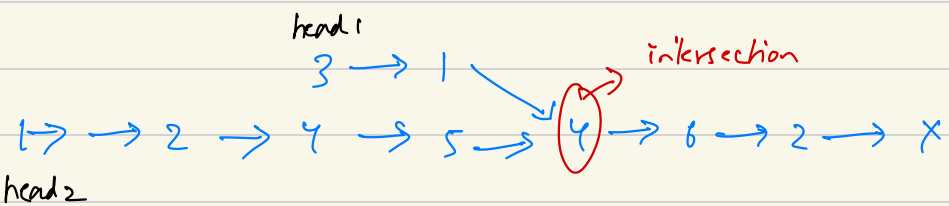
The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must **retain their original structure** after the function returns.

Custom Judge:

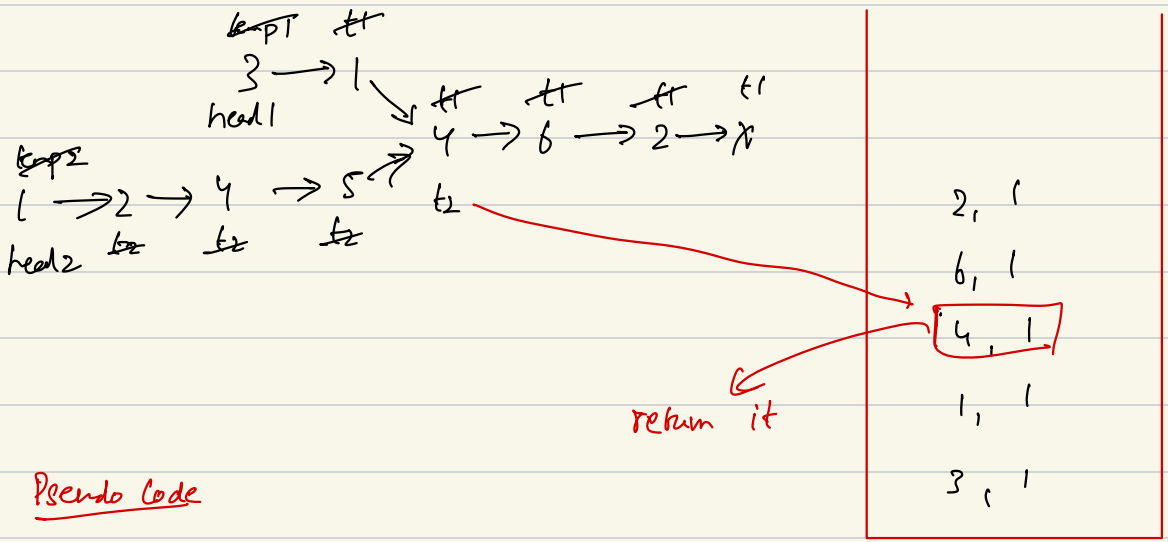
The inputs to the **judge** are given as follows (your program is **not** given these inputs):

- `intersectVal` - The value of the node where the intersection occurs. This is `0` if there is no intersected node.



1) Brute Force

- Remember the LL in the memory & again traverse the 2nd LL if you see any node which you memorized has been visited again. (Intersection Point)
- How can you memorized? You can use hashing concept.



Pseudo code

```
map<Node, int> mpp
    temp = head1
while (temp != NULL)
{
    mpp[temp] = 1; // Store entire node
        temp = temp -> next;
}
temp = head2
while (temp != null)
{
    if (mpp.find(temp) == ✓)
            return temp;
    temp = temp -> next;
}
```

Store the entire node, not only the value.
(Node, int)

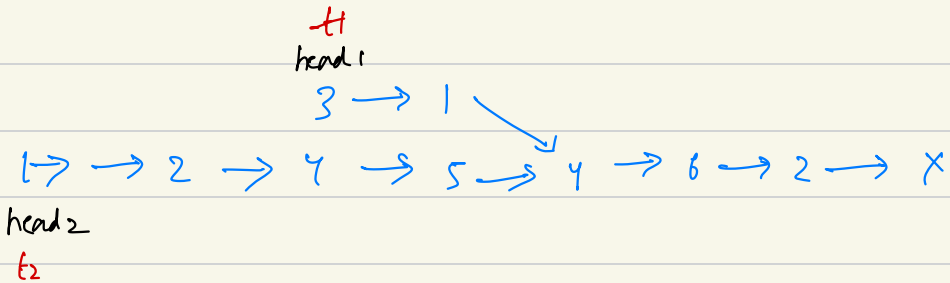
return null;

}

T.C ⇒ $O(N1 * mpp) + O(N2 * mpp)$

S.C ⇒ $O(N1)$

2) Optimised Approach:



- Which ever is longer LL, will make sure if goes in the same level as the shorter LL.

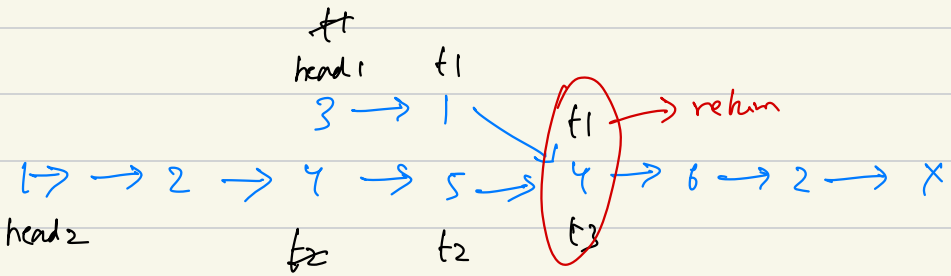
- Let's compute the length.

  N1 = 5 N2 = 7 (traverse)

  - if length is equal start, if not

    N2 - N1 = 7 - 5 = 2

  - if there is difference make sure the longer LL to move (N2 - N1) steps.
  - So here T2 will move '2' steps



- Now move simultaneous until we reach same node

```
t1 = head   N1 = 0                t2 = head   N2 = 0
// traverse to compute the length
while (t1 != null)                while (t2 != null)
{                                 {
    N1 ++;                            N2 ++;
    t1 = t1.next;                     t2 = t2.next
}                                 }
if (N1 < N2) // if L2 is longer then move
   return collision (head1, head2, N2 - N1)
else                    ↳ smaller  ↳ greater  ↳ distance
   return collision (head2, head1, N1 - N2)
```

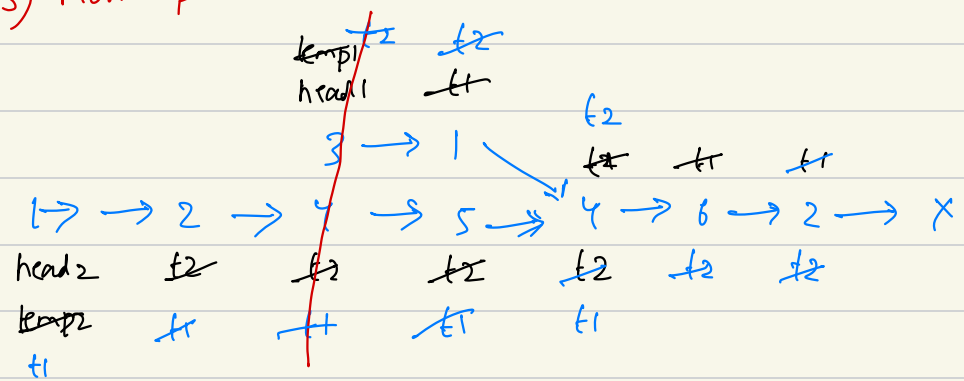
```
collisionPoint(t1, t2, d)
{
    while (d)
    {
        d--;
        t2 = t2.next;   // at same level
    }
    while(t1 != t2)
    {
        t1 = t1.next
        t2 = t2.next
    }
    return T1,
```

T.C ⇒ $O(N1) + O(N2) + O(N2 - N1) + O(N1)$

⇒ $O(N1 + 2N2)$

S.C ⇒ $O(1)$

3) More Optimised Approach

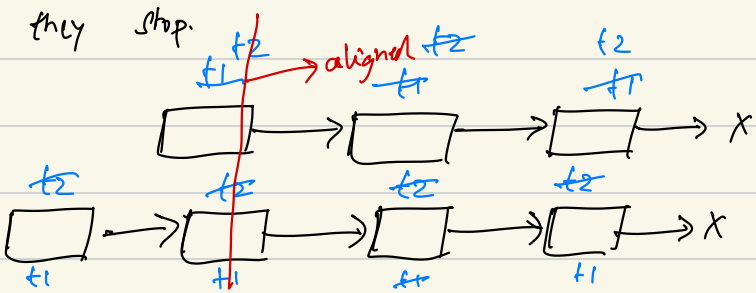


- You move t1, t2 one step simultaneously.

- Once t1.next = null, move the t1 to the head2, and move the t2 as well once t2.next = null move it to the head1.
- Now if you see t1 & t2 is aligned
- Now keep moving until they meet the same node.

The above implementation for unequally length of L.L.

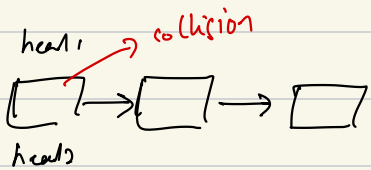
- Edge cases if there is same length of L.L. no need to change the t1 & t2 to the next L.L.
- One more edge case if there is no collision point how do they stop.



- Again t1, t2 is reached null so there is no collision point.

Pseudo Code

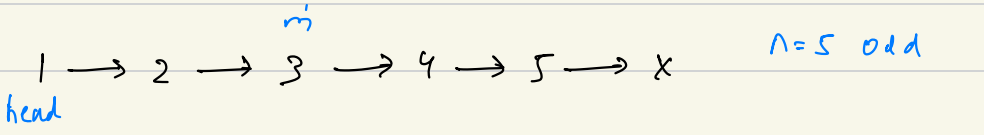
```
if (head1 == null || head2 == null) return null
    t1 = head1        t2 = head2
    while (t1 != t2)
    {
      t1 = t1.next
      t2 = t2.next
      if (t1 == t2) return t1 or t2;
      if (t1 == null)  t1 = head2
      if (t2 == null)  t2 = head1
    }
    return t1 or t2;
```

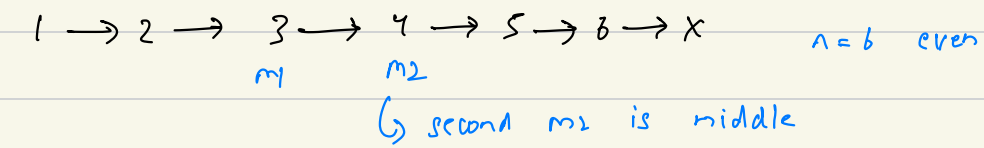


T.C ⇒ O(N1 + N2)

S.C ⇒ O(1)

L13) Middle of the linked list :





i) Brute force

- Need to find the relations of N and m

when N = 5 , m = 3rd node

N = 6 , m = 4th node

$$m = \frac{N}{2} + 1$$

when N = 5

$$m = \frac{5}{2} + 1 = 3^{rd} \text{ node}$$

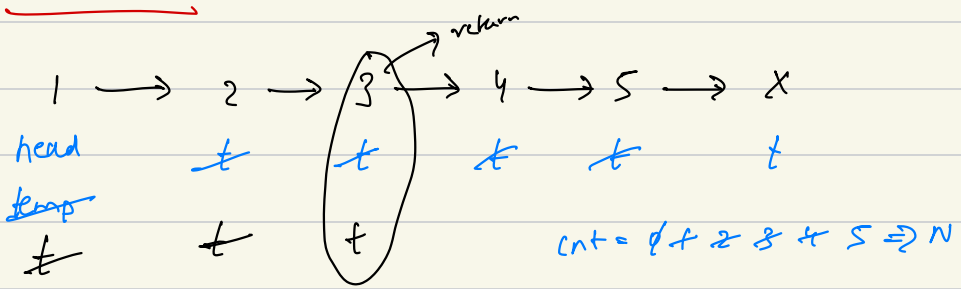
N = 6

$$m = \frac{6}{2} + 1 = 4^{th} \text{ node}$$

So m $\Rightarrow \left(\frac{N}{2}+1\right)^{th}$ node

To find N $\Rightarrow$ we do traversal & find the length.

Pseudo Code



middleNode = $\frac{5}{2}+1$ = 3rd

— So again keep temp and traversal

middleNode = 3 → X

```
temp = head        cnt = 0
while (temp != null)
{
    cnt++;
    temp = temp.next;
}
```

→ S-1

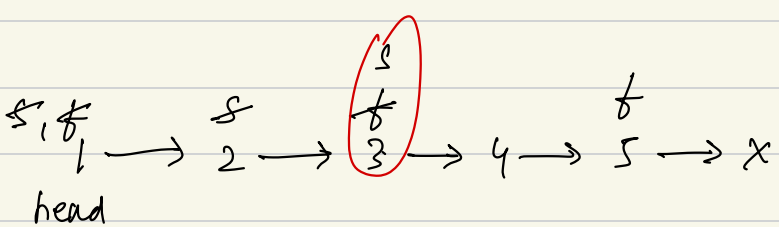
middleNode = $\left(\frac{cnt}{2} + 1\right)$

```
temp = head
while (temp != null)
{
    middleNode = middleNode - 1;
    if (middleNode == 0)
        break;
    temp = temp.next
}
return temp
```

→ S-2

T.C ⇒ $O(N + N/2)$ S.C ⇒ $O(1)$

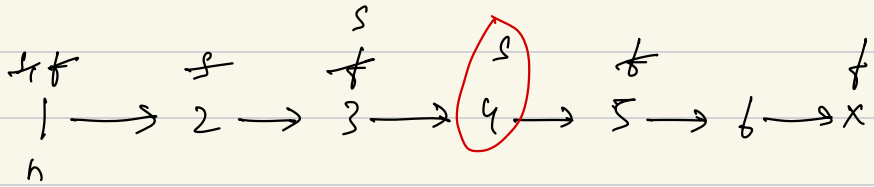
2) Optimised Approach — (Tortoise & Hare)



slow, fast = head

- slow → one step fast → two step

move this simultaneously, once fast reaches to the last node. then return slow.



If length is odd, f → last node

" " is even, f → reaches null

Pseudo Code

```
slow, fast = head
while (fast != null && fast.next != null)
{
    slow = slow.next;
    fast = fast.next.next;
}
return slow
```

T.C ⇒ O(n/2) S.C ⇒ O(1)

14) Linked List Cycle

141. Linked List Cycle

Easy 14.5K 1.2K

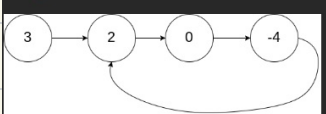
Companies

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

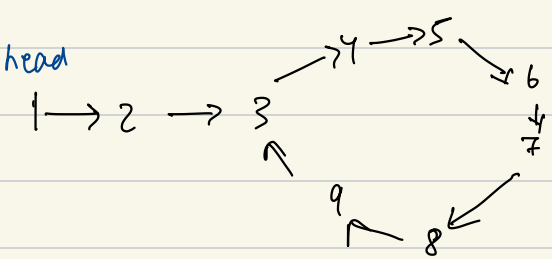
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. **Note that `pos` is not passed as a parameter.**

Return `true` *if there is a cycle in the linked list.* Otherwise, return `false`.

Example 1:



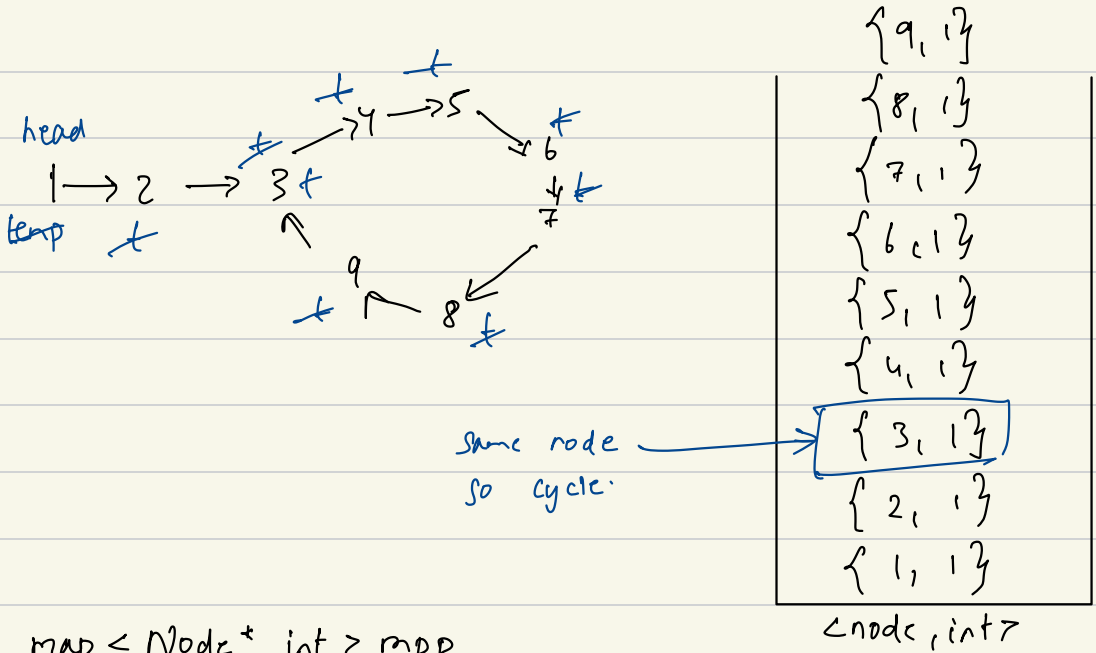

```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail
connects to the 1st node (0-indexed).
```



True

1) Brute force – Concept of Hashing

- if we take any one node and reaches back to it then there is a cycle

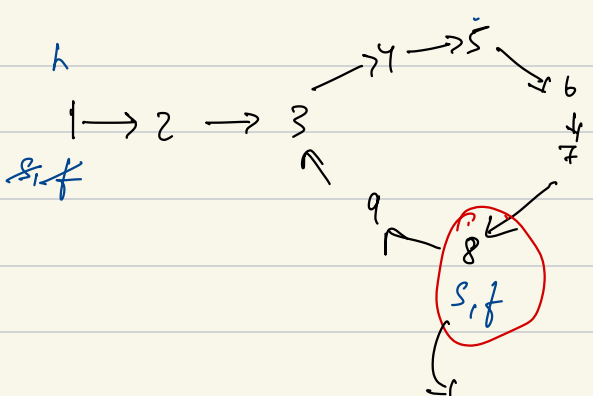



```
map < Node*, int > mpp
 · temp = head
  while ( temp != null)
  {
     if (mpp.find (temp) == ✓)
                return true;
     mpp [temp] = 1;
     temp = temp → next;
   }
  return false
```

T.C ⇒ $O(N \times 2)$

S.C ⇒ $O(N)$

2) Optimised — Tortoise & Hare



slow → 1

fast → 2

When s, f stands at the same node i.e. cycle is present.

Stop condition `fast.next != null && fast != null`

Pseudo Code

```
slow, fast = head
while (fast != null && fast.next != null)
{
    slow = slow.next
    fast = fast.next.next
    if (slow == fast) return true
```

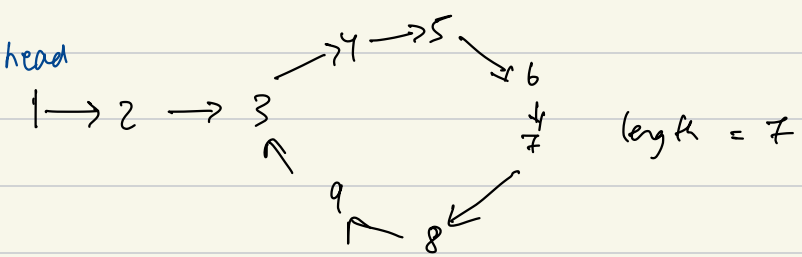
```
}
return false;
}
```

T.C ⇒ O(n)

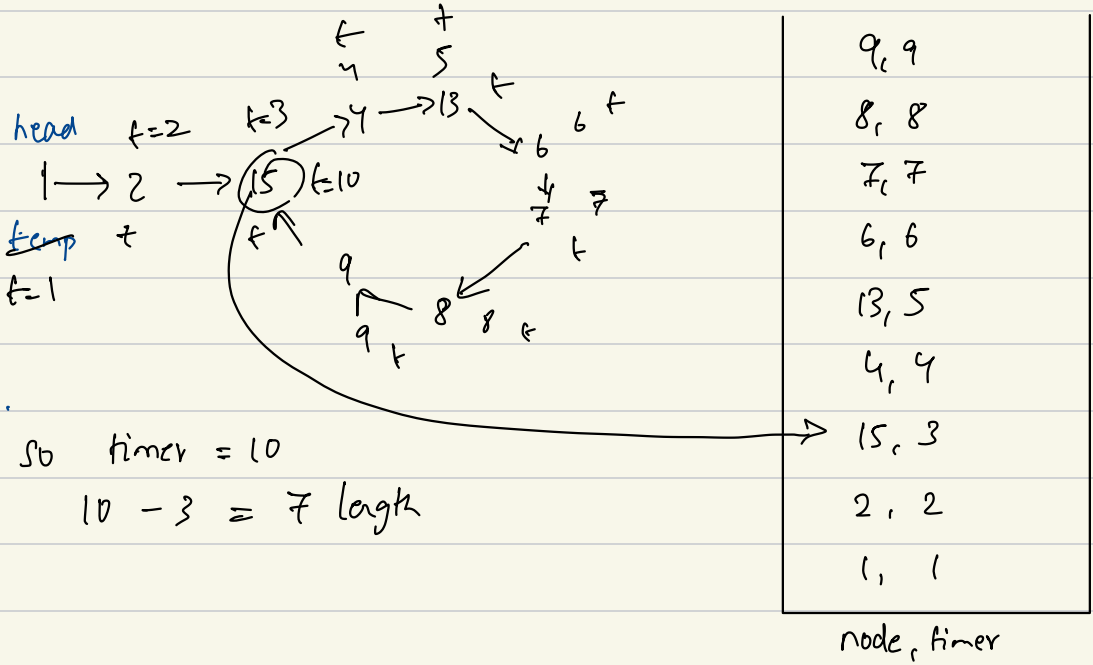
S.C ⇒ O(1)

```cpp
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while(fast != NULL && fast->next){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) return true;
        }
        return false;
    }
};
```

15) Find the length of the loop in L.L.



Brute force — Hashing



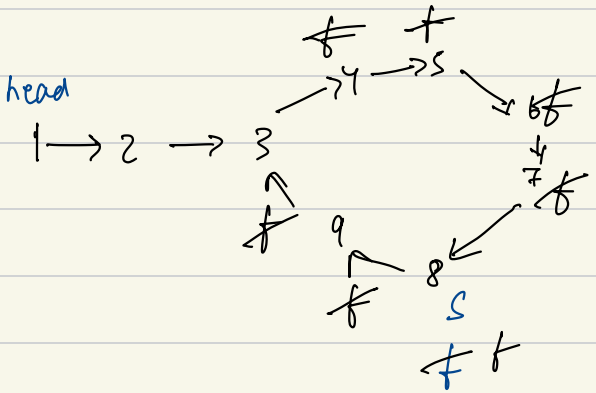
Pseudo Code

```
map < node, int > mpp      temp = head      timer = 1
while (temp != null)
{
    if (mpp.find (temp) == ✓)
    {
        value = mpp [temp];
        return = timer − value
    }
    mpp [temp] = timer
    timer ++;
```

```
        temp = temp->next
    }
    return 0;
}
```

TC ⇒ $O(W*2*(\log n))$ SC ⇒ $O(n)$

Optimised Approach - T & H



- Once S, f collide, keep a cnt = 0 & move S or f by 1 step and increase the cnt ++ once you reach again return cnt.

Pseudo Code

```
Slow, fast = head
while (fast != null && fast.next != null)
{
    slow = slow.next
    fast = fast.next.next
    if (slow == fast)
        return findlength(slow, fast)
}
return 0;
```

```
findlength (slow, fast)
{
    cnt = 1, fast = fast->next
    while (slow != fast)
    {
        cnt++
        fast = fast->next
    }
    return cnt
```

```c
int findLength(Node *slow, Node *fast) {
  int cnt = 1;
  fast = fast->next;
  while (slow != fast) {
    cnt++;
    fast = fast->next;
  }
  return cnt;
}
int lengthOfLoop(Node *head) {
  // Write your code here
  Node *slow = head;
  Node *fast = head;
  while (fast != NULL && fast->next) {
    slow = slow->next;
    fast = fast->next->next;
    if (slow == fast)
      return findLength(slow, fast);
  }
  return 0;
}
```

(16) Delete the middle Node of a Linked List

2095. Delete the Middle Node of a Linked List

Hint

Medium 3.8K 69

Companies

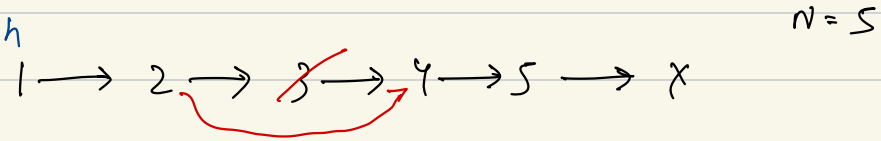
You are given the `head` of a linked list. **Delete** the **middle node**, and return *the* `head` *of the modified linked list.*

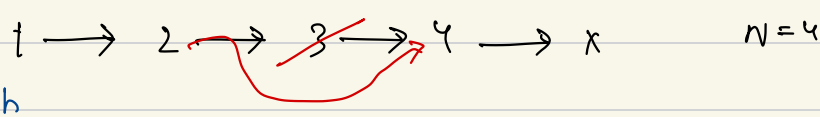
The **middle node** of a linked list of size `n` is the `⌊n / 2⌋`th node from the **start** using **0-based indexing**, where `⌊x⌋` denotes the largest integer less than or equal to `x`.

- For `n` = `1`, `2`, `3`, `4`, and `5`, the middle nodes are `0`, `1`, `1`, `2`, and `2`, respectively.

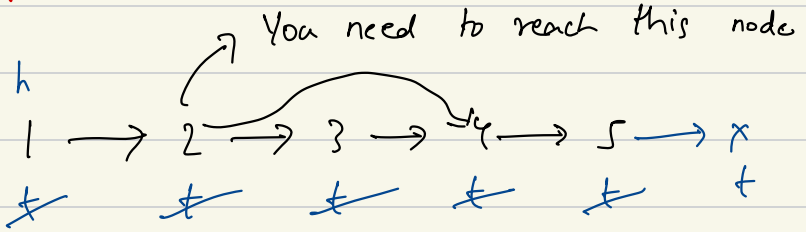
Example 1:


```
Input: head = [1,3,4,7,1,2,6]
Output: [1,3,4,1,2,6]
Explanation:
The above figure represents the given linked list. The indices
of the nodes are written below.
Since n = 7, node 3 with value 7 is the middle node, which is
marked in red.
```





Brute force



node.next = node.next.next

N → traverse the LL

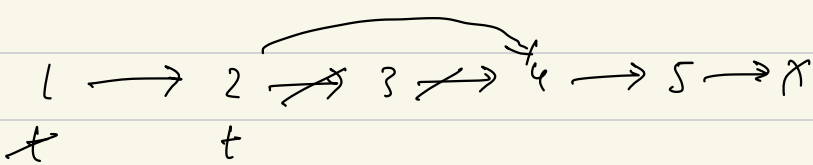
$\frac{N}{2}$ => (You need to reach this node and link

N = 5

$val = \frac{N}{2} = 2$

- Again traverse and reduce the val

val = 2 + 0



middle = temp.next

t.next = temp.next.next

free (middle)

Pseudo Code

```
temp = head      n = 0
while (temp != null)
{
    n++
    temp = temp.next
}
int res = n/2
temp = head
while (temp != null)
{
    res--;
    if (res == 0)
    {
        middle = temp.next
        temp.next = temp.next.next
        free (middle)
        break
    }
    temp = temp -> next
return head
```

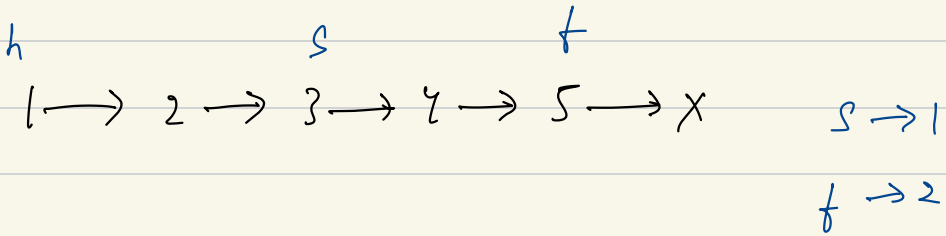
→ O(n)

→ O(n/2)

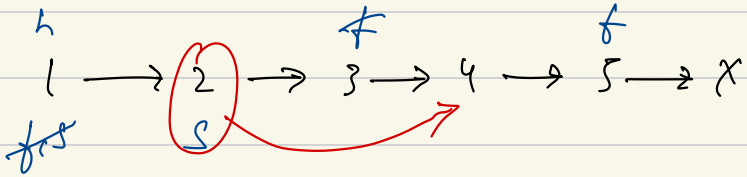
T.C ⇒ O(n + n/2)

S.C ⇒ O(1)

Optimised Approach (Tortise & Hare)



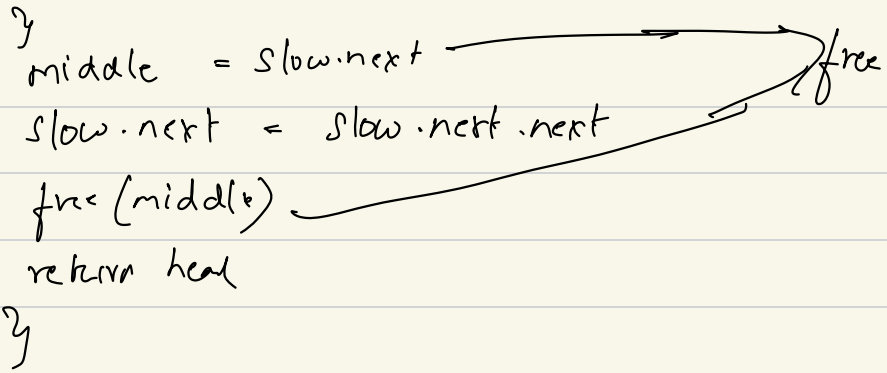
- You reached middle node but you need to go middle previous to link.
- So slight change so skip one step of slow



skip 1 step of slow

Pseudo Code

```
slow = head    fast = head
fast = fast.next.next
while (fast != null && fast.next != null)
{
        slow = slow.next
        fast = fast.next.next
```




```cpp
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if(head == NULL || head->next == NULL) return NULL;
        ListNode* slow = head;
        ListNode* fast = head;
        fast = fast->next->next;

        while(fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        }
        slow->next = slow->next->next;
        return head;
    }
};
```

(17) Find the starting point of the loop/cycle in LL:

142. Linked List Cycle II

Medium 13K 901

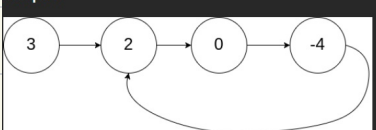
Companies

Given the `head` of a linked list, return *the node where the cycle begins. If there is no cycle, return* `null`.

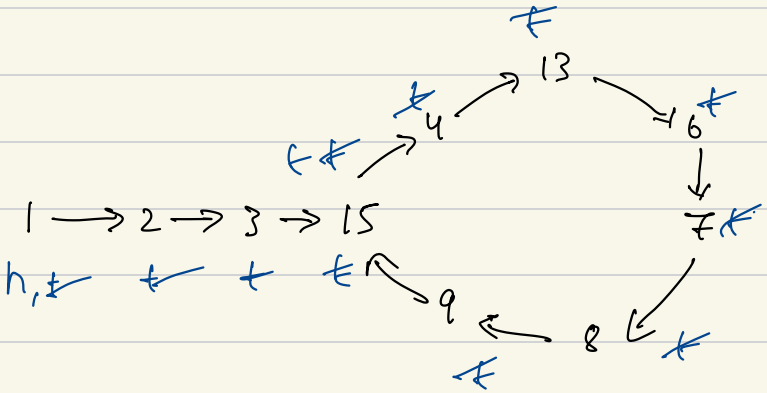
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to (**0-indexed**). It is `-1` if there is no cycle. **Note that** `pos` **is not passed as a parameter**.

Do not modify the linked list.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: tail connects to node index 1
Explanation: There is a cycle in the linked list, where tail connects to the second node.
```

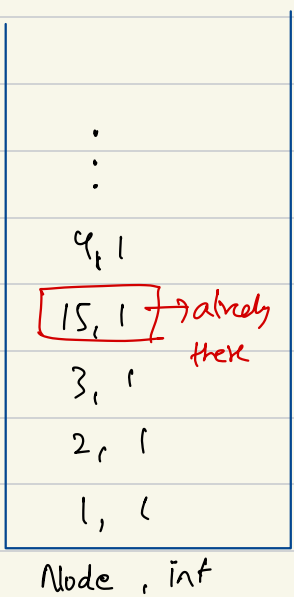


Brute force Using Map

- Any moment I reach the node twice then it's the starting point.
- Before putting into map check if the node already present or not

Pseudo code

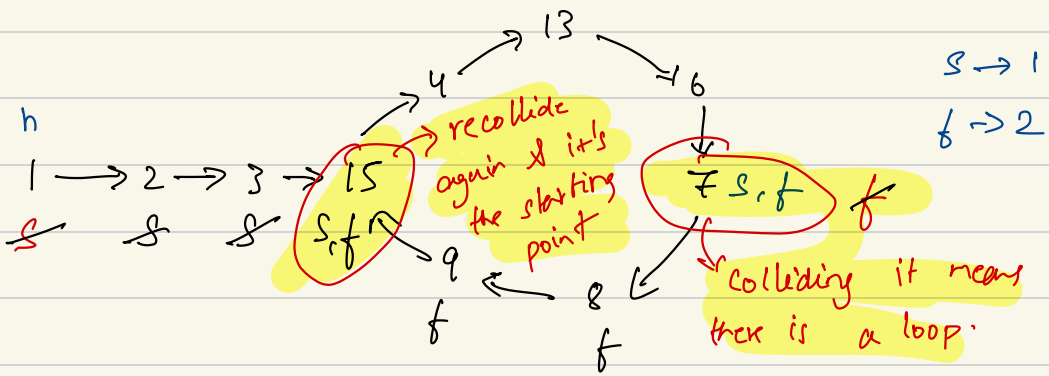
```
map<Node*, int> mpp      temp = head
while (temp != NULL)
{
    if (map.find(temp) == v)
    {
        return temp;
    }
```



```
mpp.put(temp, 1)   OR   mpp[temp] = 1

temp = temp.next
}
return null
```

T.C ⇒ $O(N \times 2 \times \log N)$ S.C ⇒ $O(N)$

Optimised Tortoise & Hare



① → deleting the loop

② → finding the starting ⇒ S → head

f → collision point

move S, f move them simultaneously by "1".

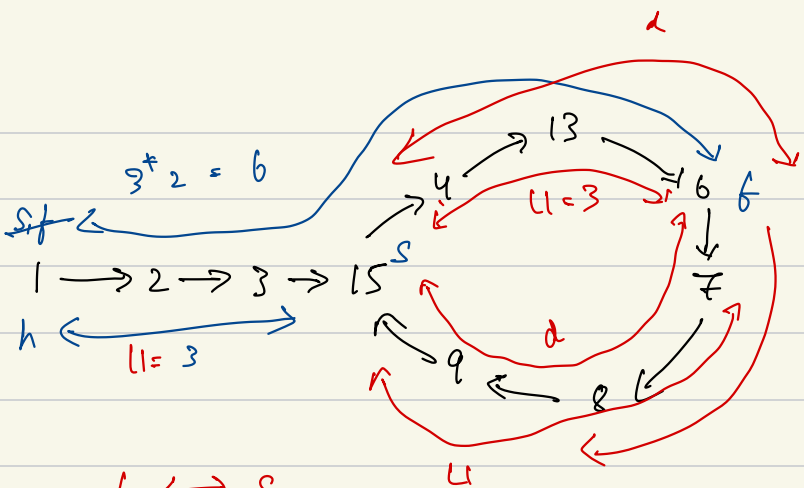
Pseudo Code

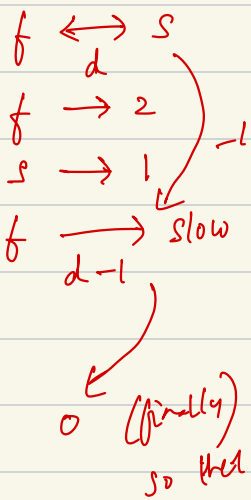
```
slow, fast = head
while (fast != null && fast.next != null)
{
```

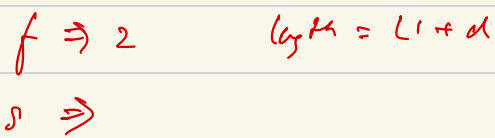
```
slow = slow.next
fast = fast.next.next
if (slow == fast)  // Colliding
 {
    // Step - 2
     slow = head
     while ( slow != fast)
      {
         slow = slow.next
         fast = fast.next
      }
      return slow | fast;
   }
 }
return null
```

Why ? 1) How are we sure that they will collide

2) How are we sure that collision point will be start

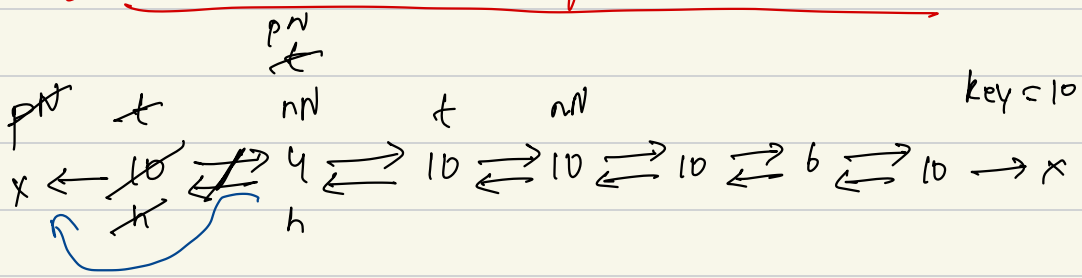







```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while(fast != NULL && fast->next != NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) {
                // Step - 2
                slow = head;
                while(slow != fast){
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
        return NULL;
    }
};
```

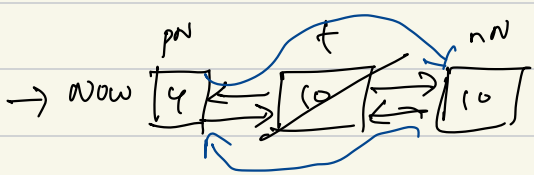
(18) Delete all occurence of a key in DLL:



- if the temp value is equal to key we have to delete.
- Here we delete the head as t == h so we update the head

pN → prevNode, nN → nextNode

→ nextNode.prev = prevNode

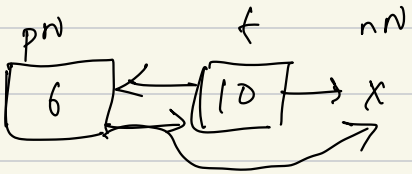


pN.next = nN

nN.prev = pN

The process continue.

Now



pN . next = null

There is no nN so free the temp

Pseudo code

```
temp = head
while (temp != null)
{
   if ( temp.val == k)
    {
        if ( temp == head)
        {
            head = head.next
        }
        nextNode = temp.next
        prevNode = temp.prev
        if (nextNode)
            nextNode.prev = prevNode
        if (prevNode)
            prevNode.next = nextNode
```

free(temp)

temp = nextNode;

}

else

temp = temp.next

}

return head;

```
Node * deleteAllOccurrences(Node* head, int k) {
    Node* temp = head;
    while(temp != NULL){
        if(temp->data == k){
            if(temp == head){
                head = temp->next;
            }

            Node* nextNode = temp->next;
            Node* prevNode = temp->prev;
            if(nextNode){
                nextNode->prev = prevNode;
            }
            if(prevNode){
                prevNode->next = nextNode;
            }

            free(temp);
            temp = temp->next;
        }
        else {
            temp = temp->next;
        }
    }
    return head;
}

```

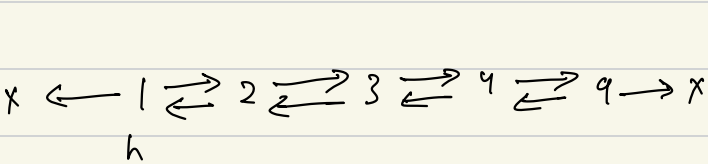
L(4) Find pairs with given sum in Sorted DLL

Sum = 5

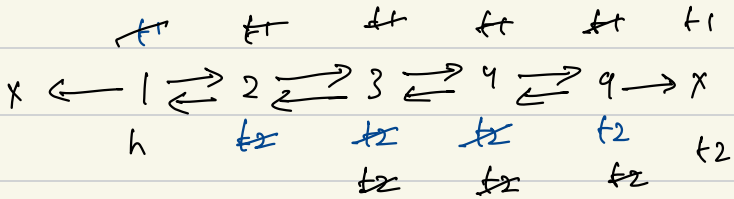
O/P

(1, 4)

(2, 3)



Brute force



- check if t1 + t2 = sum or not, if not then move t2.
- since it's sorted 1 + 9 > sum so right side it won't be there

(1, 4)

(2, 3)

- 2 + 4 > sum so no need to move t2.

So on.

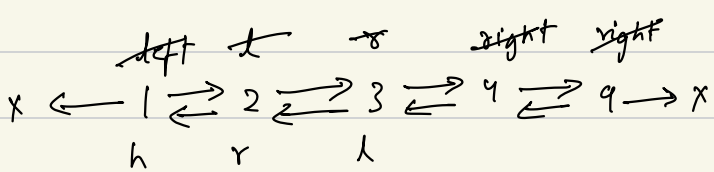
Pseudo Code

```
temp1 = head        ds = []
while ( temp != null)
{
```

```
temp2 = temp1.next
while (temp2 != null && temp1.val + temp2.val <= sum)
{
    if (temp1.val + temp2.val == sum)
    {
        ds.add ({temp1.val, temp2.val})
    }
    temp2 = temp2->next
}
temp1 = temp1->next
}
```

T.C $\Rightarrow \simeq O(n^2)$ S.C $\Rightarrow O(1)$

Optimised



if left + right > sum

So right --

(1, 4) = 5

when left, right == sum

then left -- right --

(2+3) = 5

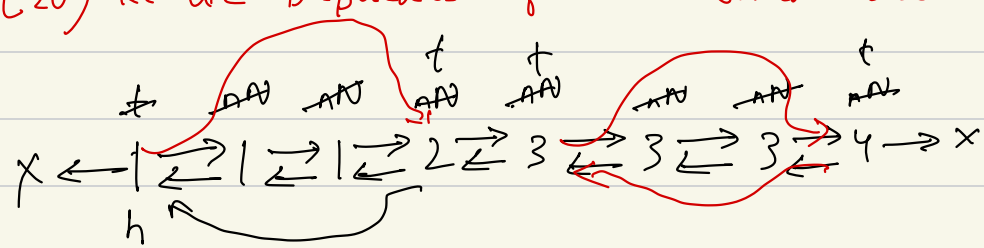
once right cross left you stop.

Pseudo Code

```
left = head      right = findTail(head)
while (left.val < right.val)
{
    if (left.val + right.val == sum)
    {
        ds.add(l, r)
        left = left → next
        right = right → prev
    }
    else if (left.val + right.val < sum)
    {
        left = left → next
    }
    else {
        right = right → prev
    }
    return ds.
```

```cpp
Node* findTail(Node* head){
    Node* tail = head;
    while(tail->next != NULL) tail = tail->next;
    return tail;
}

vector<pair<int, int>> findPairs(Node* head, int k)
{
    vector<pair<int, int>> ans;
    if(head == NULL) return ans;
    Node* left = head;
    Node* right = findTail(head);

    while(left->data < right->data){
        if(left->data + right->data == k){
            ans.push_back({left->data, right->data});
            left = left->next;
            right = right->prev;
        }
        else if(left->data + right->data < k){
            left = left->next;
        }
        else {
            right = right->prev;
        }
    }
    return ans;
}
```

(20) Remove Duplicates from a Sorted DLL



temp.next = nextNode (~N)

nextNode → prev = temp

Pseudo Code

```
temp = head
while (temp != NULL && temp.next != NULL)
{
    nextNode = temp.next
    while (nextNode != null &&
            nextNode -> val == temp -> val)
    {
        nextNode = nextNode -> next
    }
    temp -> next = nextNode
    if (nextNode) {
        nextNode -> prev = temp
    }
    temp = temp -> next;
}
return head
```

T.C ⇒ O(N) S.C ⇒ O(1)

```
Node * removeDuplicates(Node *head)
{
    Node* temp = head;
    while(temp != NULL && temp->next != NULL){
        Node* nextNode = temp->next;
        while(nextNode != NULL && nextNode->data == temp->data){
            Node* duplicate = nextNode;
            nextNode = nextNode->next;
            free(duplicate);
        }
        temp->next = nextNode;
        if(nextNode != NULL) nextNode->prev = temp;
        temp = temp->next;
    }
    return head;

}
```

(21) Reverse Node in k Group Size of LL:

k = 3

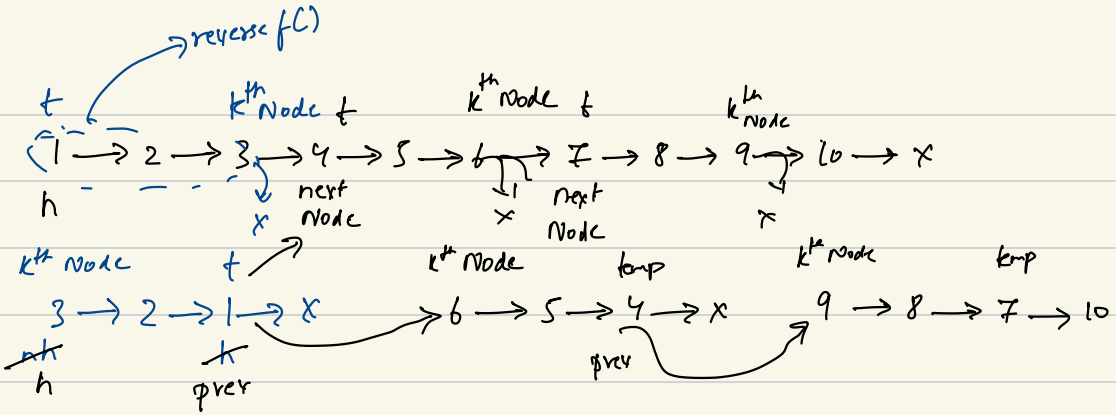
1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → X

h

O/P

3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7 → 10 → X

h



if no k^{th} Node then you just attach

Pseudo Code

```
temp = head    nextNode    prevNode = NULL
while( temp != NULL)
{
   kth Node = findkthNode(temp, k);
   if (kth Node == NULL)
    { if (prevNode) prevNode->next = temp.

      }  break;
   nextNode = kth Node -> next
   kth Node.next = null
          reverse(temp)
   //1st kth group
    if (temp == head)
```

```
{
        head = kth Node;
}
else {
        prevNode -> next = kth Node
}
prevNode = temp;
temp = next Node;
}
```

(22) Rotate List

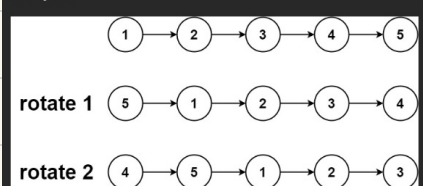
61. Rotate List

Medium 9.1K 1.4K

Companies

Given the `head` of a linked list, rotate the list to the right by `k` places.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]
```

k = 2

1 → 2 → 3 → 4 → 5 → X

h

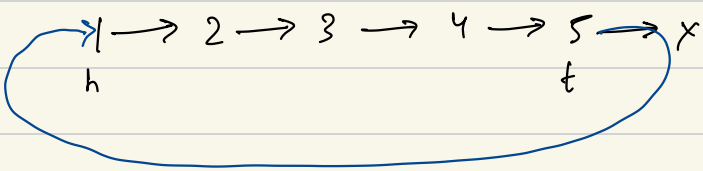
↓ 1 rotate

5 → 1 → 2 → 3 → 4 → X

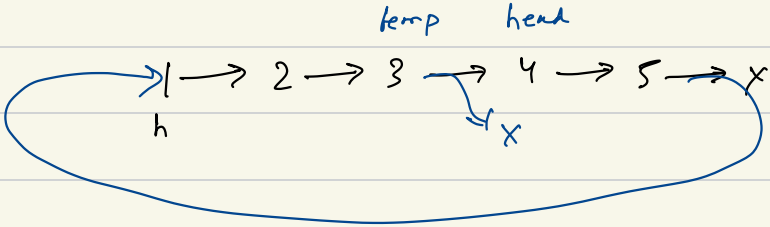
↓ 1 rotate

4 → 5 → 1 → 2 → 3 → X

- Traverse to the L.L & reach the tail
- → After that join the tail to the head



- So K = 2, L = 5 (L - K) = 5 - 2 = 3rd Node
- Reach the 3rd node and point it to null
- before that you need the update Head
- So traverse in L.L till (L - K) = 3 and stop there and point the next to the update Head and then point the 3rd Node to null



O/P ⇒ 4 → 5 → 3 → 2 → 1 → X

Now if the k is the length of L.L.

k = 5

1 → 2 → 3 → 4 → 5 → X

h

- if you try to rotate then you will reach the original

So if k == len of L.L. then just return the head

Now if k > len of L.L. k = 10

1 → 2 → 3 → 4 → 5 → X

h

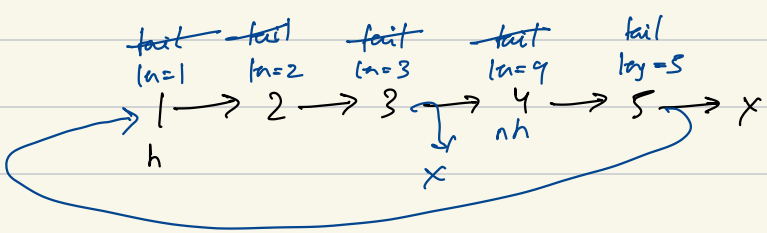
- if the multiple of length then don't do anything just return the head.

if k = 13 ⇒ 5 + 5 + 3

↓ ↓ ↳ 3 times (just like before)

original original

if (k % len == 0) return head



if k = 14

14 % 5 = 4

length

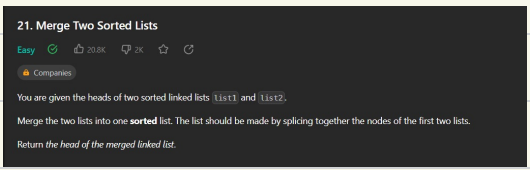
Pseudo Code

```
len = 1        tail = head
while ( tail.next != null)
{
        len ++ ;  tail = tail.next
}
if ( k % len == 0)   return head
        k = k % len;
tail -> next = head;        S-1
newLastNode = findNthNode (head, len - k)
head = newLastNode -> next;
newLastNode -> next = NULL;
return head;
```

C++ | Auto

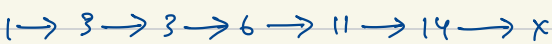
```cpp
class Solution {
private:
    ListNode* findNthNode(ListNode* temp, int k){
        int cnt = 1;
        while(temp != NULL){
            if(cnt == k) return temp;
            cnt++;
            temp = temp->next;
        }
        return temp;
    }
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(head == NULL || k == 0) return head;
        ListNode* tail = head;
        int len = 1;
        while(tail->next != NULL){
            tail = tail->next;
            len += 1;
        }
        if(k % len == 0) return head;
        k = k % len;

        // attach the tail to the head;
        tail->next = head;
        ListNode* newLastNode = findNthNode(head, len - k);
        head = newLastNode->next;
        newLastNode->next = NULL;
        return head;
    }
};
```

(23) Merge two sorted L.L. :



L1

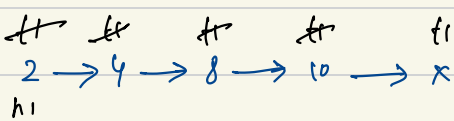
L2



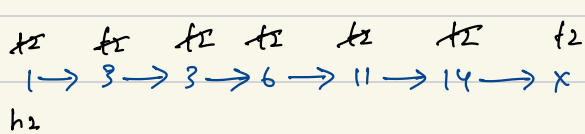
L3

Brute force :

L1



L2



— Take an array & traverse the L1 until t1 reaches the null

2, 4, 8, 10

same for L2

1 3 3 6 11 14

Array 2 4 8 10 1 3 3 6 11 14

↓ Sort

1 2, 3 3 4 6 8 10 11 14

↓ convert an array to list.

Pseudo Code

```
arr = []   temp1 = head1   temp2 = head2
while (temp1 != null)
{
    arr.add (temp1 -> data)
    temp1 = temp1 -> next
}
sort (arr)
```

```
while (temp2 != null)
{
    arr.add (temp2 -> data)
    temp2 = temp2 -> next
}
```

// Call the convert fn()

head = convertToLL(arr)

return head;

T.C ⇒ $O(N1 + N2 + N \log N + O(N))$

S.C ⇒ $O(N) + O(N)$

Two Pointer Approach :

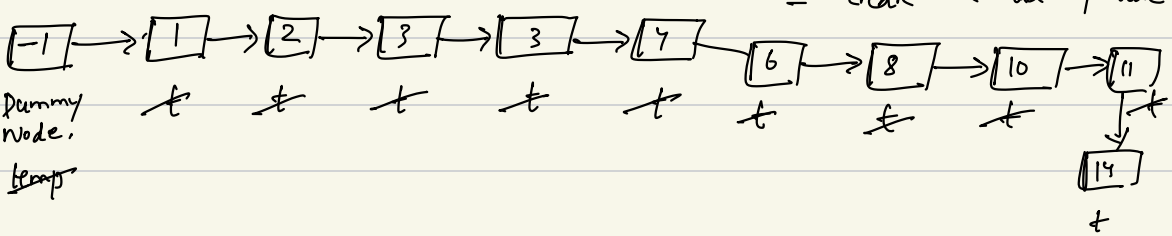
L1 2 → 4 → 8 → 10 → X

h1

L2 1 → 3 → 3 → 6 → 11 → 14 → X

h2

- Compare the Node
- Reassign the list as no need to create new list
- Create a dummy Node



Pseudo Code

t1 = head1 , t2 = head2

dNode = new Node(-1) , temp = dNode

```
while (t1 != NULL && t2 != NULL)
{
    // Pick up the small element
    if (t1.data < t2.data)
    {
        temp -> next = t1
        temp = t1
        t1 = t1 -> next
    }
    else {
        temp -> next = t2
        temp = t2
        t2 = t2 -> next
    }
}
// t1 / t2 is left out
if (t1)  temp -> next = t1
else     temp -> next = t2
return dNode -> next
```

T.C ⇒ O(n1 + n2)

S.C ⇒ O(1)

Code :

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* temp1 = list1;
        ListNode* temp2 = list2;
        ListNode* dummyNode = new ListNode(-1);
        ListNode* temp = dummyNode;
        while(temp1 != NULL && temp2 != NULL) {
            if(temp1->val < temp2->val){
                temp->next = temp1;
                temp = temp1;
                temp1 = temp1->next;
            } else {
                temp->next = temp2;
                temp = temp2;
                temp2 = temp2->next;
            }
        }
        if(temp1) temp->next = temp1;
        else temp->next = temp2;
        return dummyNode->next;
    }
};
```

(24) Flattening a linked list :

430. Flatten a Multilevel Doubly Linked List

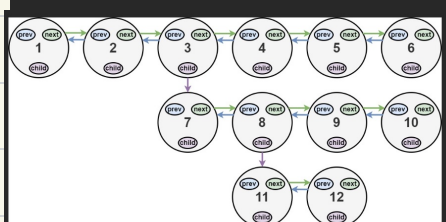
Medium 4.9K 313

Companies

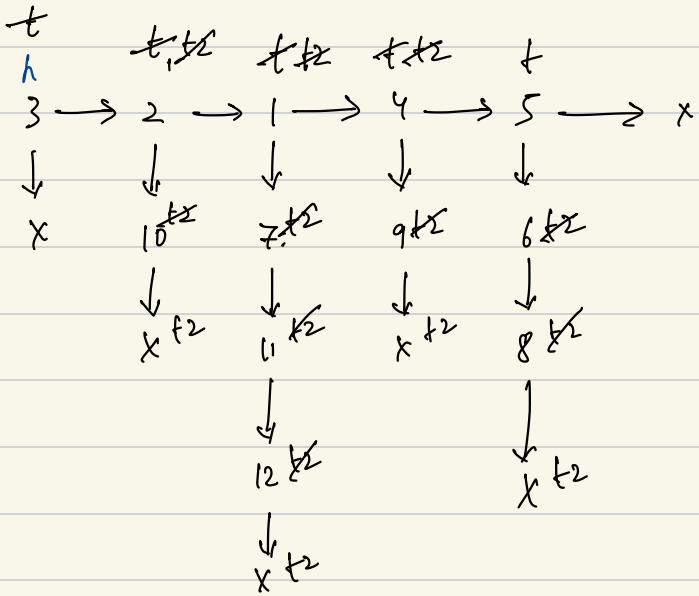
You are given a doubly linked list, which contains nodes that have a next pointer, a previous pointer, and an additional **child pointer**. This child pointer may or may not point to a separate doubly linked list, also containing these special nodes. These child lists may have one or more children of their own, and so on, to produce a **multilevel data structure** as shown in the example below.

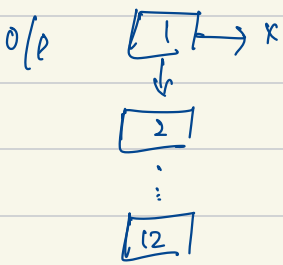
Given the `head` of the first level of the list, **flatten** the list so that all the nodes appear in a single-level, doubly linked list. Let `curr` be a node with a child list. The nodes in the child list should appear **after** `curr` and **before** `curr.next` in the flattened list.

Return the `head` *of the flattened list. The nodes in the list must have* ***all*** *of their child pointers set to* `null`.



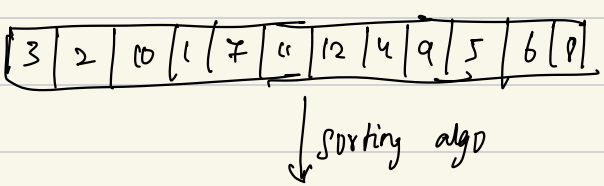
Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
Output: [1,2,3,7,8,11,12,9,10,4,5,6]
Explanation: The multilevel linked list in the input is shown.
After flattening the multilevel linked list it becomes:

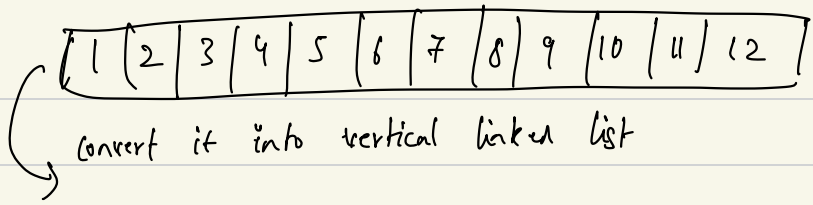




Brute force

— Take all the integers but then into the array.





Psendo Code

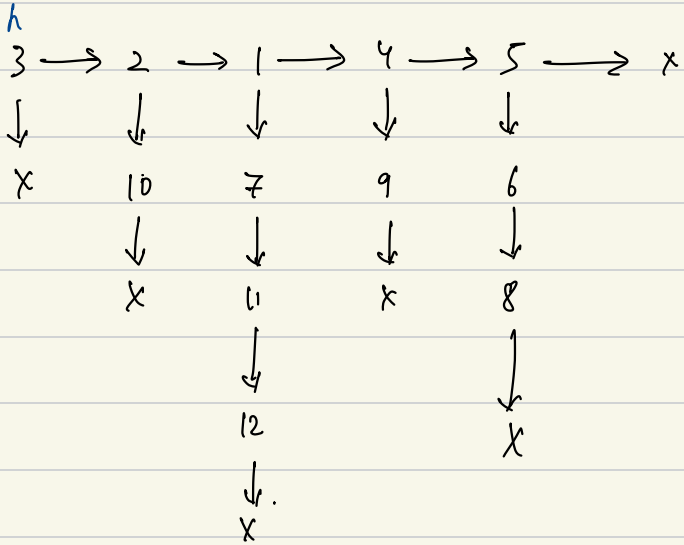
```
arr = []      temp = head
while (temp != null)
{
    t2 = temp
    while (t2 != null)
    {
        arr.add (t2 → val)
        t2 = t2 → child;
    }
    temp = temp → next
}
sort (arr)
head = convert (arr)
```

```
convert (arr)
{
    if (arr.size == 0) return null
    head = new Node (arr [0]), temp = head
    for (i = 1 ⟶ n-1)
    {
        newNode = newNode (arr [i])
        temp ⟶ child = newNode;
        temp = temp ⟶ child;
    }
    return head;
}
```

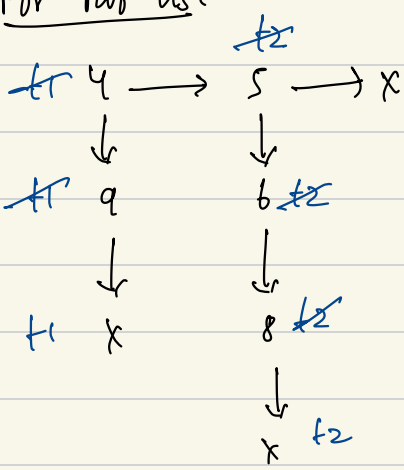
T.C ⇒ $O(N \times M) + O(N \times M \log N \times M) + O(N \times M)$

S.C ⇒ $O(N \times M) \times 2$

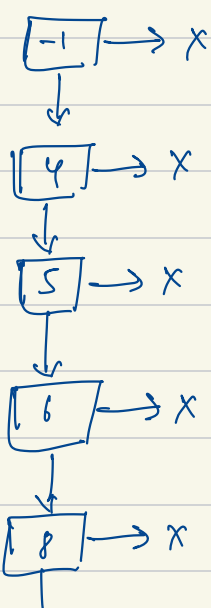
Better Approach

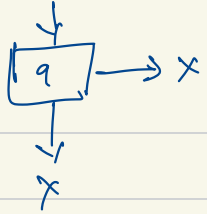


For two list



dummyNode





Pseudo Code

```
mergeTwoLists (list1, list2)
{
    dummyNode = new Node(-1);
    res = dummyNode;
    while (list1 != null && list2 != null)
    {
        if (list1.val < list2.val)
        {
            res -> child = list1;
            res = list1;
            list1 = list1.next;
        }
        else {
            res -> child = list2
            res = list2
            list2 = list2.next
        }
        res -> next = null
    }
```

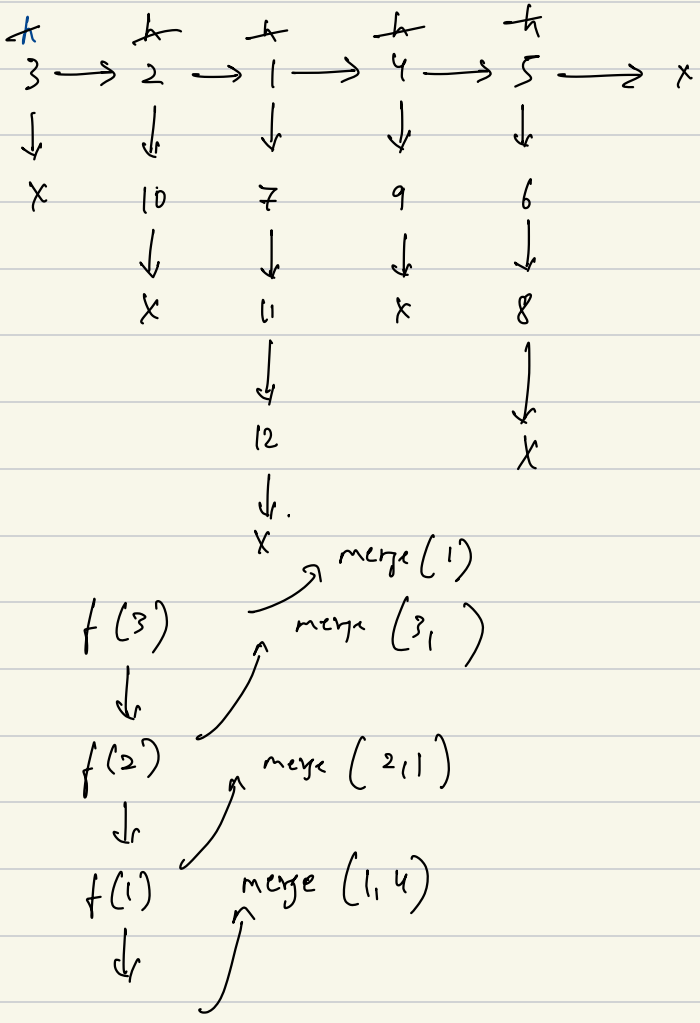
if (list 1) res → child = l1

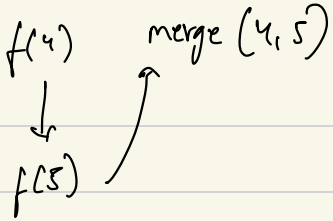
else res → child = l2

return dummyNode → child;

O(N1 + N2)

For multiple list





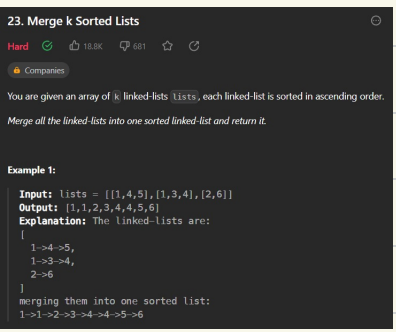

```
f (head)
{
    if (head == null || head -> next == null)
            return head;
    mergedhead = f (head -> next)
    return merge2list(head, mergedHead)
}
```

T.C ⇒ O(N) x O(2M)

S.C ⇒ O(N) (Recursive Stack Memory)

```
*/
Node* merge(Node* list1, Node* list2) {
    Node* dummyNode = new Node(-1);
    Node* res = dummyNode;
    while(list1 != NULL && list2 != NULL) {
        if(list1->data < list2->data) {
            res->child = list1;
            res = list1;
            list1 = list1->child;
        }
        else {
            res->child = list2;
            res = list2;
            list2 = list2->child;
        }
        res->next = nullptr;
    }
    if(list1) res->child = list1;
    else res->child = list2;
    if(dummyNode->child) dummyNode->child->next = nullptr;
    return dummyNode->child;
}
Node* flattenLinkedList(Node* head)
{
    if (head == NULL || head->next == NULL) return head;

    Node* mergedHead = flattenLinkedList(head->next);
    head = merge(head, mergedHead);
```

125) Merge k Sorted Lists:



23. Merge k Sorted Lists

Hard 18.8K 681

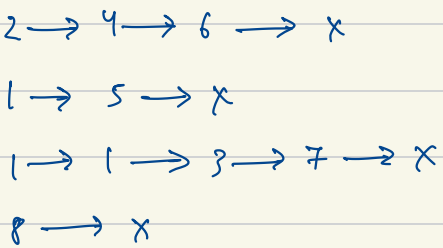
Companies

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
1->1->2->3->4->4->5->6
```



Brute force

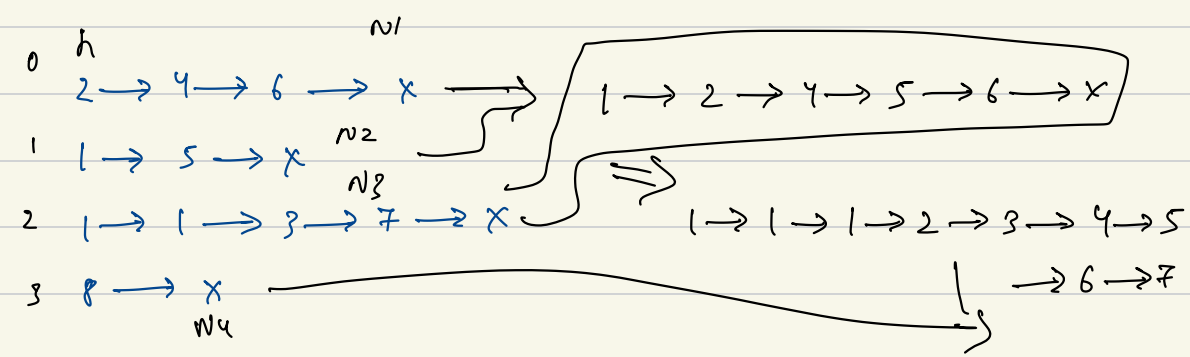
```
f( list <Node> lists)
{
    arr = [ ]
    for( i=0 ; i< lists.size ; i++)
    {
```

```
Node temp = lists[i];
while (temp != null)
{
    arr.add(temp.val)
    temp = temp.next
}
}
sort(arr)
head = convertToList(arr)
return head
}
```

T.C ⇒ $O(k \times n) + O(M \log M) + O(M)$

S.C ⇒ $O(M) + O(M)$

Better Approach:



1 → 1 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → X

Pseudo Code

```
f( list <Node> lists)
{
    head = lists [0]
    for (i= 1; i < lists.size() ; i + +)
    {
      head =  merge 2 lists (head, lists [i])
    }
     return head
}
```

T.C ⇒ $O(N1 + N2) + O(N1 + N2 + N3) + O(N1 + N2 + N3 + N4)$

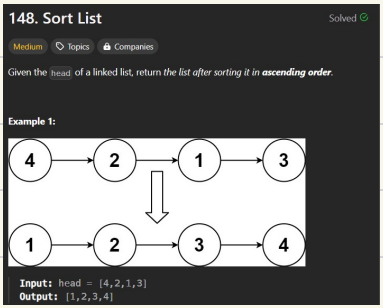
$\simeq O(2N) + O(3N) + O(4N)$

$\simeq N(1+2+3+\ldots k)$

$\simeq N \times \frac{k \times k+1}{2}$

S.C = O(1)

(26) Sort a Linked Lists :



Brute Force :

temp

3 → 4 → 2 → 1 → 5 → X

head

Create an array and move the element to it until the temp reach null

arr ⇒ | 3 | 4 | 2 | 1 | 5 |

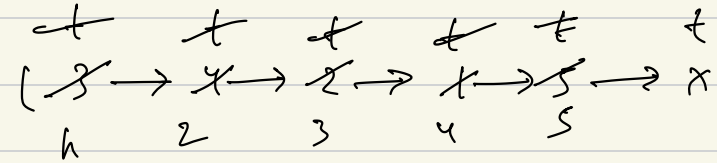
↓ Sort any algo

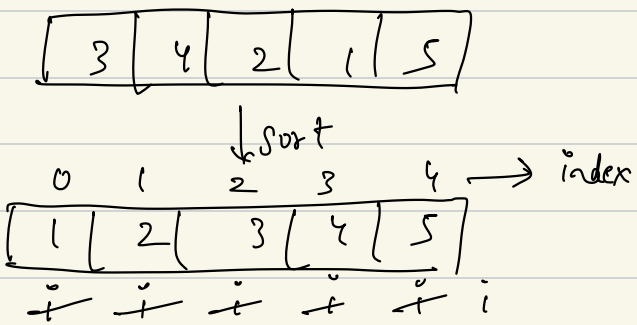
| 1 | 2 | 3 | 4 | 5 |

↓ convert array to LL

1 → 2 → 3 → 4 → 5

OR instead of calling the convert function we can reuse a li.





Now the value at 'i' will be moved to the temp, so keep moving i and temp.

1 → 2 → 3 → 4 → 5

Pseudo Code

```
f(head)
{
    arr = [], temp = head
```

```
while ( temp != NULL)
{
    arr.add( temp -> val)
    temp = temp -> next
}
```

→ O(N)

```
sort ( arr)
```

→ N log N

```
i = 0   temp = head
while (temp != NULL)
{
    temp -> data = arr [i]
    i = i+1 , temp = temp -> next
}
```

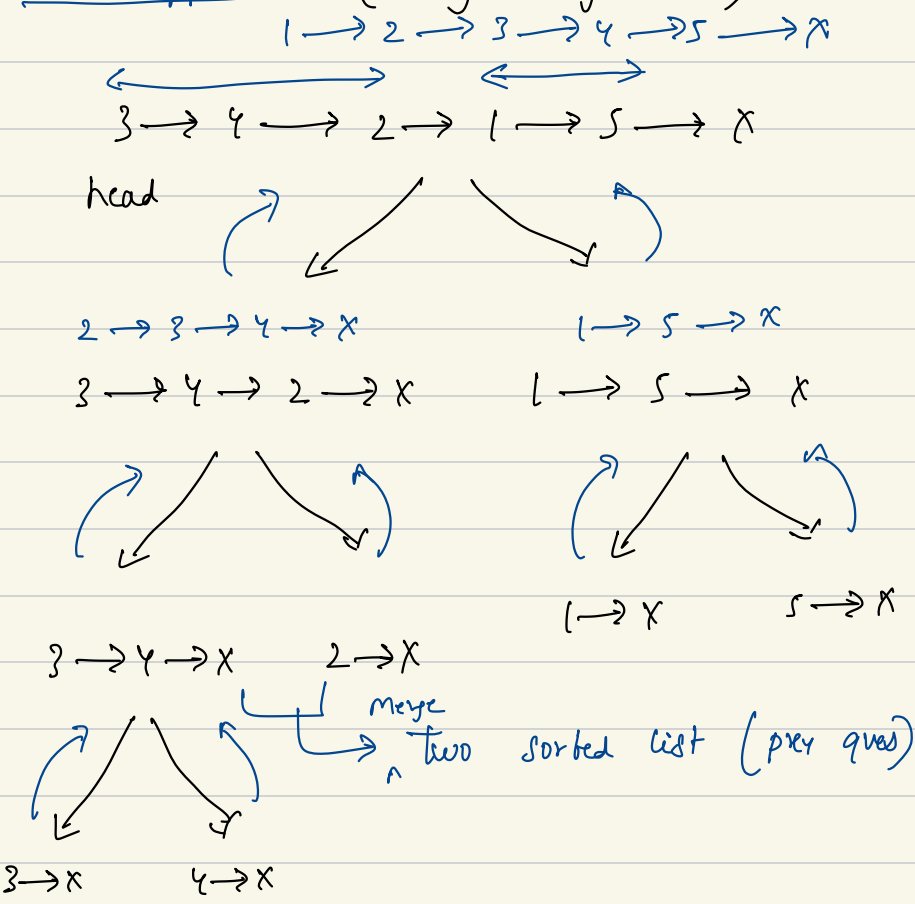
→ O(N)

```
return head;
}
```

T.C ⇒ N + NlogN + N

S.C ⇒ O(N)

Better Approach : (using merge sort)



Pseudo Code :

```
MS (arr, low, high)
{
    if (low == high) return
```

```
    mid = (low + high) / 2
    ms (arr, low, mid)
    ms (arr, mid+1, high)
    merge (arr, low, mid, high)
}
```

In form of C.C.

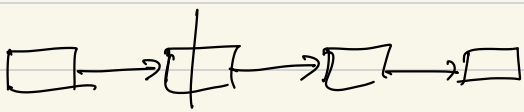
```
ms (head)
{
  // base
  if (head == null || head -> next == null)
    return head;

  middle = findMiddle (head);   // tortoise method
  leftHead = head
  rightHead = middle -> next
  middle -> next = null

  leftHead = ms (leftHead)
  rightHead = ms (rightHead)
```

return merge2(leftHead, rightHead)

↳ already done

```cpp
class Solution {
private:
    ListNode* findMiddle(ListNode* head){
        ListNode* slow = head;
        ListNode* fast = head->next;
        while(fast != NULL && fast -> next != NULL) {
            slow = slow -> next;
            fast = fast -> next -> next;
        }
        return slow;
    }
private:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2){
        ListNode* dummyNode = new ListNode(-1);
        ListNode* temp = dummyNode;
        while(list1 != NULL && list2 != NULL) {
            if(list1 -> val < list2 -> val){
                temp -> next = list1;
                temp = list1;
                list1 = list1 -> next;
            }
            else {
                temp -> next = list2;
                temp = list2;
                list2 = list2 -> next;
            }
        }
        if(list1) temp-> next = list1;
        else temp -> next = list2;
        return dummyNode->next;
    }
public:
    ListNode* sortList(ListNode* head) {
        if(head == NULL || head->next == NULL) return head;
        ListNode* middle = findMiddle(head);
        ListNode* right = middle->next;
        middle->next = nullptr;
        ListNode* left = head;

        left = sortList(left);
        right = sortList(right);
        return mergeTwoLists(left, right);
    }
};
```

→ only one change because we need the first middle not the second middle



T.C ⇒ $O(\log N) \times O(N + N/2)$

S.C ⇒ $O(N)$ // Recursive Space

L27) Clone a linked list

138. Copy List with Random Pointer

Solved

Medium · Topics · Companies · Hint

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

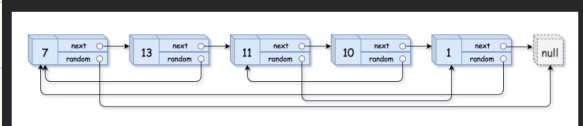
Return *the head of the copied linked list.*

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

- `val`: an integer representing `Node.val`
- `random_index`: the index of the node (range from `0` to `n-1`) that the `random` pointer points to, or `null` if it does not point to any node.

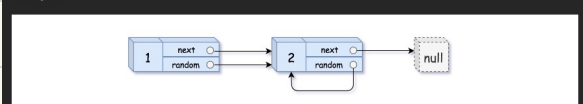
Your code will **only** be given the `head` of the original linked list.

Example 1:

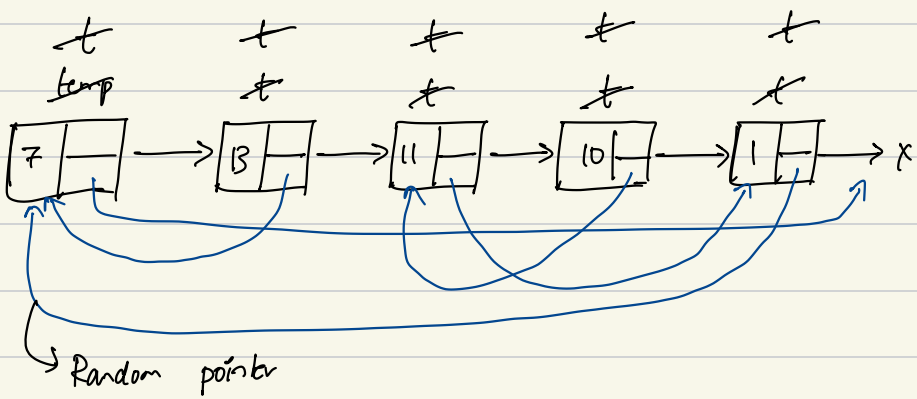



```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

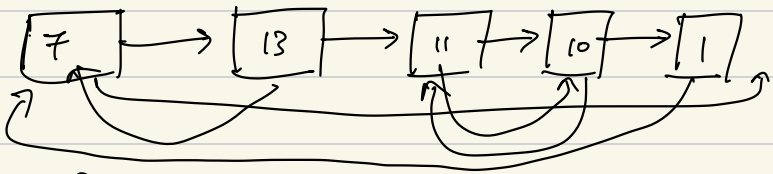
Example 2:



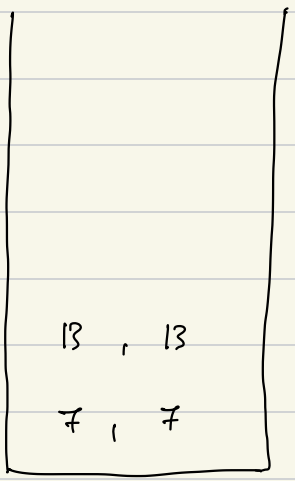

```
Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]
```



- Create each node, with the help of Hashmap you can store the address of them.



- Point the next pointer and the random pointer. and keep moving the temp.



<Node, Node>
original copy
(Storing the node, not the val)

Pseudo Code:

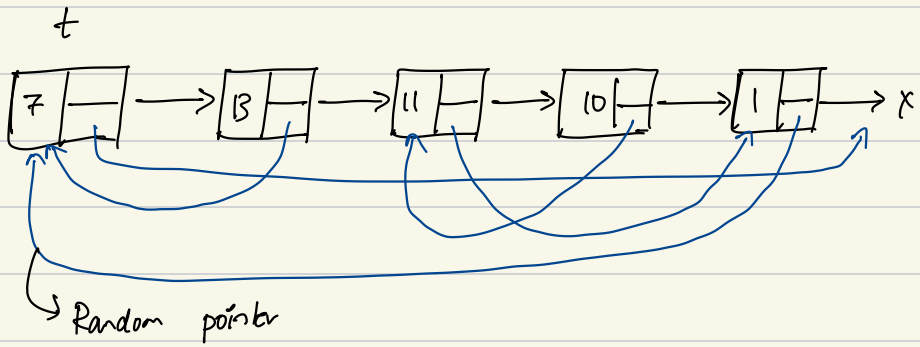
```
f(head)
{
    temp = head,   map<Node, Node> mpp
```

```
while (temp != null)
{
    Node newNode = new Node(temp -> val);
        mpp[temp] = new Node;
            temp = temp -> next;
}
    temp = head
    while (temp != null)
    {
        copyNode = mpp[temp]
        copNode -> next = mpp[temp -> next]
        copyNode -> random = mpp[temp -> random]
        temp = temp -> next
    }
    return mpp[head];
}
```

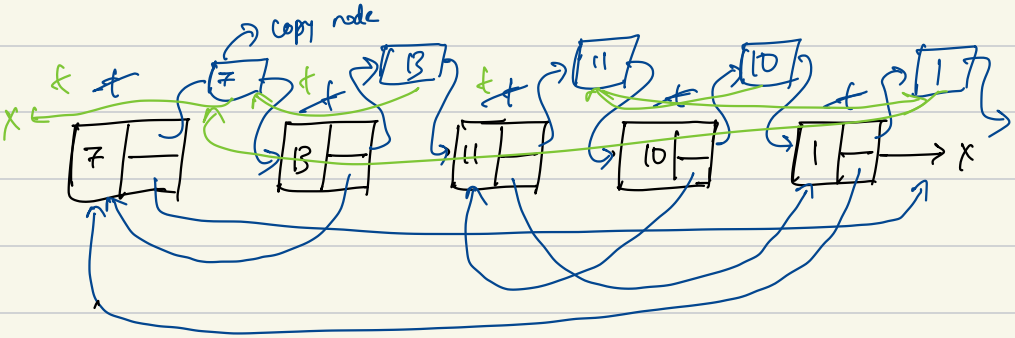
T.C ⇒ O(N) + O(N)

S.C ⇒ O(N) + O(N)

Optimised Approach :



1) Insert copy nodes in between



2) Connect Random pointer

copyNode is temp.next

3) Connect the next pointer with the help of dummy Node.

Pseudo Code

① temp = head

while (temp != NULL)

{

copyNode = new Node(temp → val)

copyNode → next = temp → next

temp → next = copyNode

temp = temp → next → next

}

② temp = head

while (temp != null)

{

copyNode = temp.next

if (random != null) copyNode → random = temp → random → next // connecting the random pointer

temp = temp → next → next

③ dNode = newNode(-1) res = dNode temp = head

while (temp != null)

{

res.next = temp.next

temp.next = temp.next.next

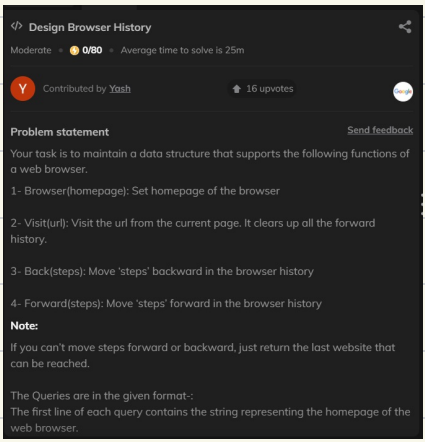
```
res = res.next
temp = temp.next
}
return dNode -> next;
```

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        Node* temp = head;
        //step 1
        while(temp != NULL) {
            Node* newNode = new Node(temp->val);
            newNode->next = temp->next;
            temp->next = newNode;
            temp = temp->next->next;
        }
        //step 2
        Node* itr = head;
        while(itr != NULL) {
            if(itr->random != NULL)
                itr->next->random = itr->random->next;
            itr = itr->next->next;
        }
        //step 3
        Node* dummy = new Node(0);
        itr = head;
        temp = dummy;
        Node* fast;
        while(itr != NULL) {
            fast = itr->next->next;
            temp->next = itr->next;
            itr->next = fast;
            temp = temp->next;
            itr = fast;
        }
        return dummy->next;

    }
};
```

T.C ⇒ O(3N)

S.C ⇒ O(N) // to create a new list

1472) Design a Browser History:



Design Browser History

Moderate • 0/80 • Average time to solve is 25m

Contributed by Yash — 16 upvotes

Problem statement — Send feedback

Your task is to maintain a data structure that supports the following functions of a web browser.

1- Browser(homepage): Set homepage of the browser

2- Visit(url): Visit the url from the current page. It clears up all the forward history.

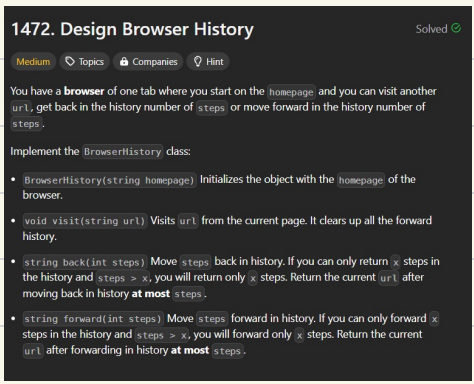
3- Back(steps): Move 'steps' backward in the browser history

4- Forward(steps): Move 'steps' forward in the browser history

Note:

If you can't move steps forward or backward, just return the last website that can be reached.

The Queries are in the given format-:
The first line of each query contains the string representing the homepage of the web browser.



1472. Design Browser History — Solved

Medium · Topics · Companies · Hint

You have a **browser** of one tab where you start on the `homepage` and you can visit another `url`, get back in the history number of `steps` or move forward in the history number of `steps`.

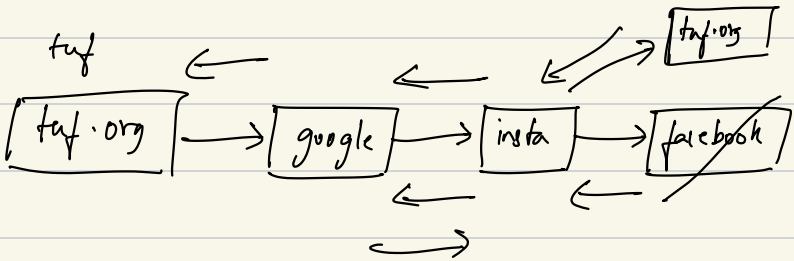
Implement the `BrowserHistory` class:

- `BrowserHistory(string homepage)` Initializes the object with the `homepage` of the browser.
- `void visit(string url)` Visits `url` from the current page. It clears up all the forward history.
- `string back(int steps)` Move `steps` back in history. If you can only return `x` steps in the history and `steps > x`, you will return only `x` steps. Return the current `url` after moving back in history **at most** `steps`.
- `string forward(int steps)` Move `steps` forward in history. If you can only forward `x` steps in the history and `steps > x`, you will forward only `x` steps. Return the current `url` after forwarding in history **at most** `steps`.

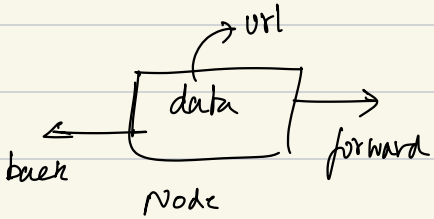
Queries

DLL (ds)

homePage (fb.org)
visit (google.com)
visit (instagram.com)
visit (facebook.com)
back (1) // insta
back (1) // google
forward (1) // insta
visit (fb.org)
forward (2) after fb then is no forward so // fb
back (2) // google
back (7) //



DLL (ds)




```
Class Browser
{
    Node* curr;
    Browser (homepage)
    {
        current = new Node (homepage)
    }
    visit (url)
    {
        newNode = newNode (url)
        curr.next = newNode
        curr = newNode
```

```
}

back (steps)
{
    while (steps)
    {   if (curr -> back)
            curr = curr -> back
            steps--
    }
   return curr -> data;
}
forward (steps)
{
   while (steps)
   {
      if (curr -> front)
          curr = curr -> next
      else
          break
      steps--
   }
    return curr -> data
}
```

Code

C++ Auto

```cpp
class BrowserHistory {
public:
    list<string> history;
    list<string>::iterator it;

    BrowserHistory(string homepage) {
        history.push_back(homepage);
        it = history.begin();
    }

    void visit(string url) {
        auto del = it;
        del++;
        history.erase(del, history.end());
        history.push_back(url);
        it++;
    }

    string back(int steps) {
        while (it != history.begin() && steps--) {
            it--;
        }
        return *it;
    }

    string forward(int steps) {
        while (it != --history.end() && steps--) {
            it++;
        }
        return *it;
    }
};
```