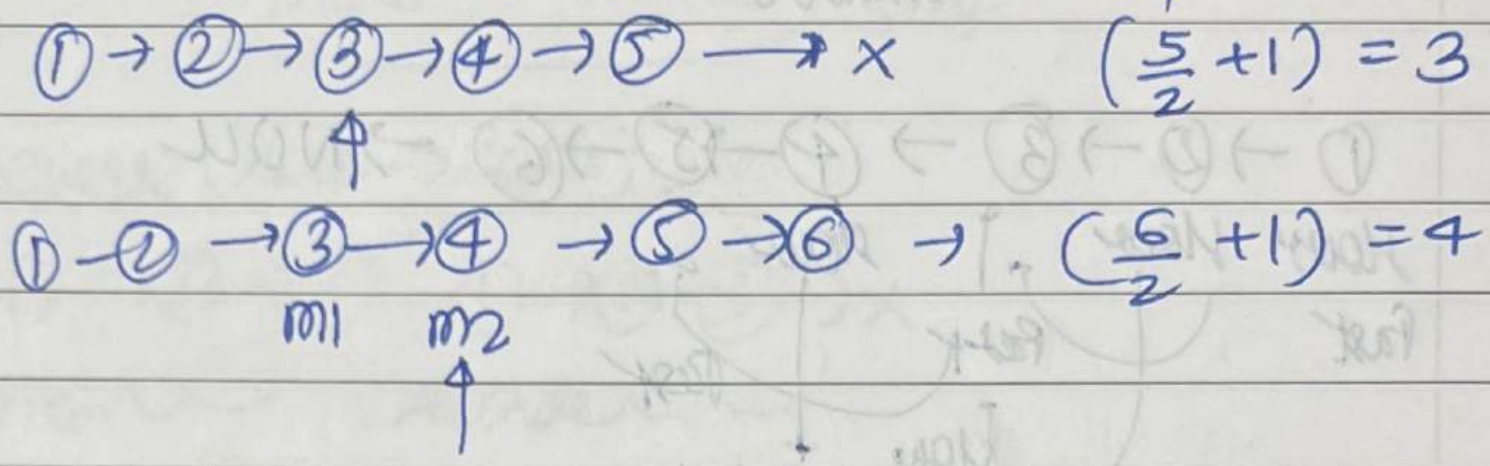
4 } Middle element of the linked list } $\{\frac{N}{2}+1\}$



@ Brute force :-

```
Node* callMiddle(head){
    int cnt=0;
    Node* temp=head;
    while(temp){
        temp=temp->next;
        cnt++;
    }
    int mid=cnt/2 + 1;
    temp=head;
    while(temp){
        mid--;
        if(!mid) return temp;
        temp=temp->next;
    }
    return head;
}
```

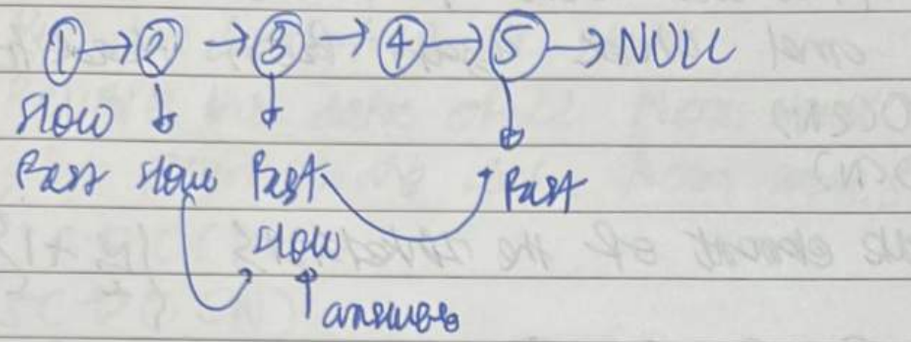
TC → O(2N - N/2)
SC → O(1)

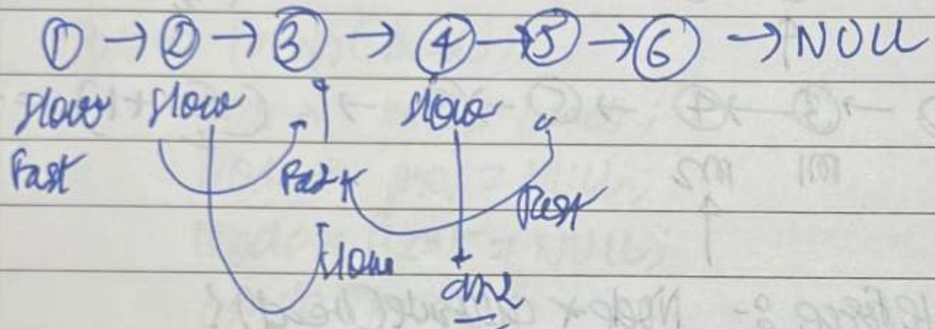
O(N) + O(N/2)

{ Tortoise & Hare }

__/__/___

Optimal :- { Tortoise and Hare }






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

TC → O(N/2)
SC → O(1)

__/__/____

⑤ { Reverse LL }

- Brute
  Put the data of LL in stack
  { Rewrite the data of LL from Head to tail }
  { By POPing the data from stack }
  T(C) → O(2N)
  S(C) → O(N)
- Optimal →

```
Node* Fun1(head){
    Node* temp = head;
    Node* prev = NULL;
    Node* front = NULL;
    while(temp){
        front = temp->next;
        temp->next = prev;
        prev = temp;
        temp = front;
    }
    return prev;
}
```

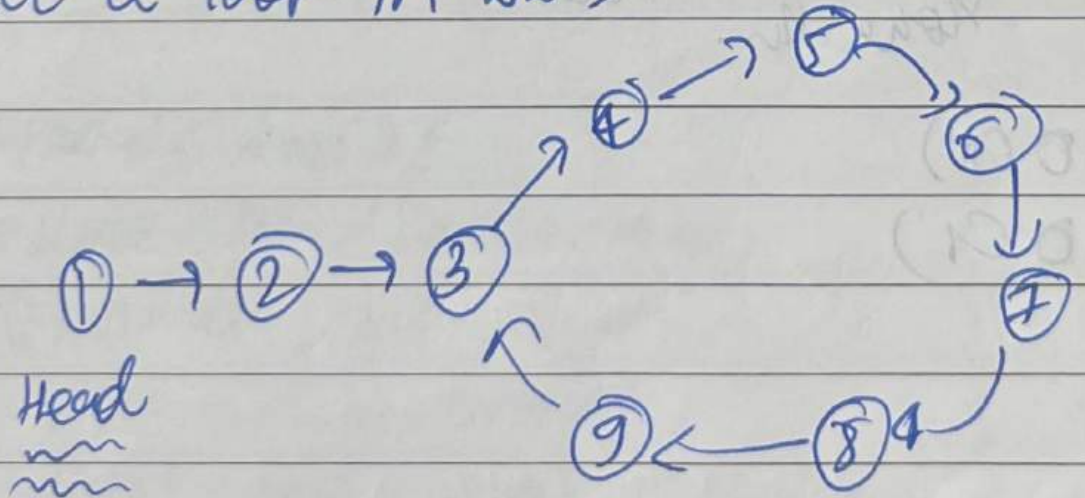
T(C) → O(N)
S(C) → O(1)

Using Recursion

```
Node* Recursion(Node* Head){
    if(!Head || !Head->next) return Head;
    Node* newHead = Recursion(Head->next);
    Node* front = Head->next;
    front->next = Head;
    Head->next = NULL;
    return newHead;
}
```

T(C) → O(N)
S(C) → O(N) → Recursive stack space

(10) { Detect a loop in LL 3

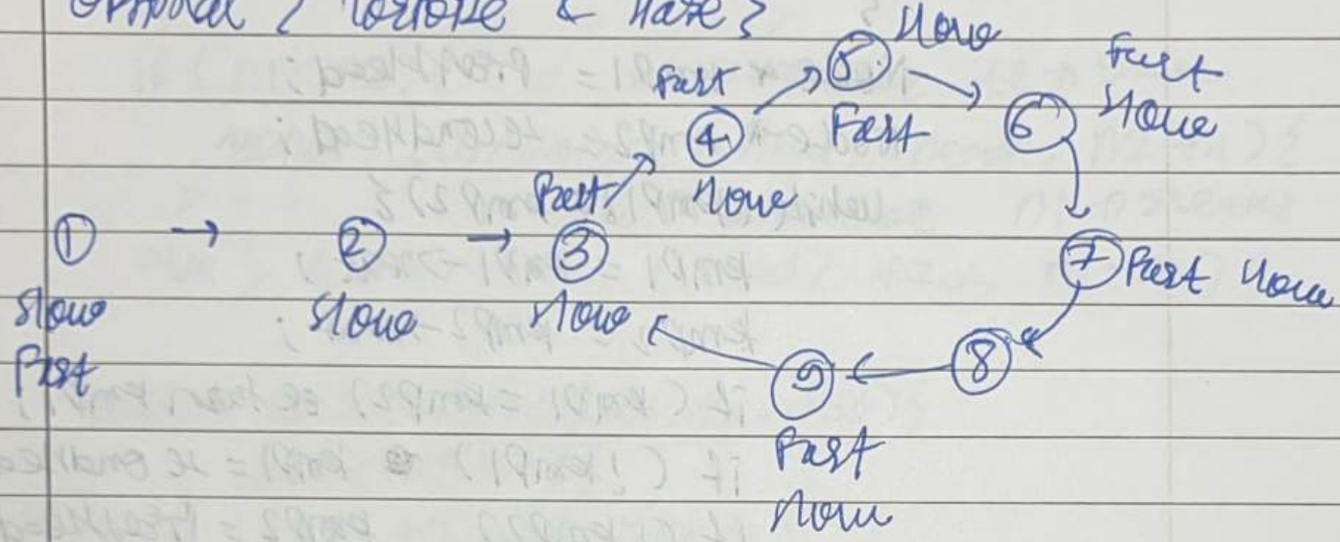


Logic :- store in map if it already in map {true}

TC → O(2N) for {unordered map}
SC → O(N)

two operation {checking & storing in} map

optimal {tortoise & hare}



Head Slow Now Now Head Head If (they are standing at same pt)

edge cases { ① → ② → ③ → ④ → ⑤ → ×

① → ② → ③ → ④ → ⑤ → ⑥ → ×

{Not a loop}

Now Now Head Head Now Head Head

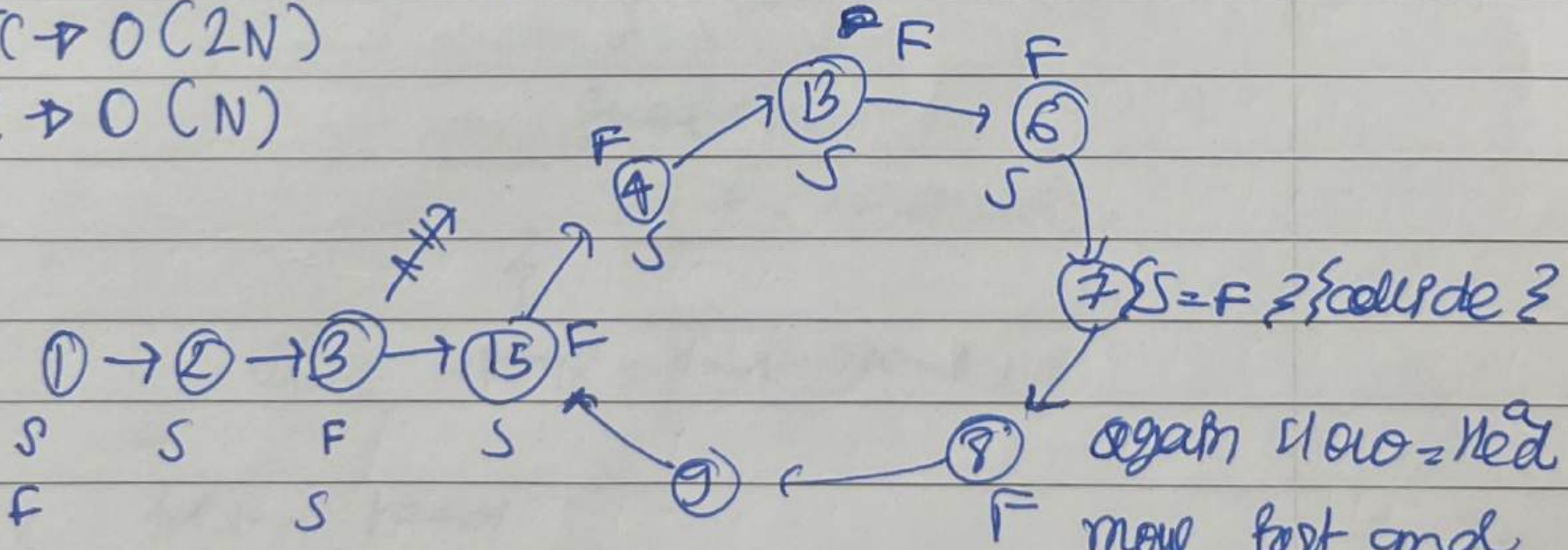
TC → O(N)
SC → O(1)

(13) { starting node in a loop of LL }

Logic + { put in the map }

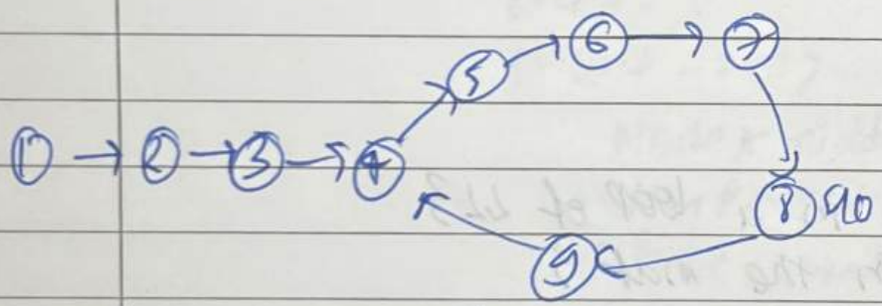
TC → O(2N)

SC → O(N)



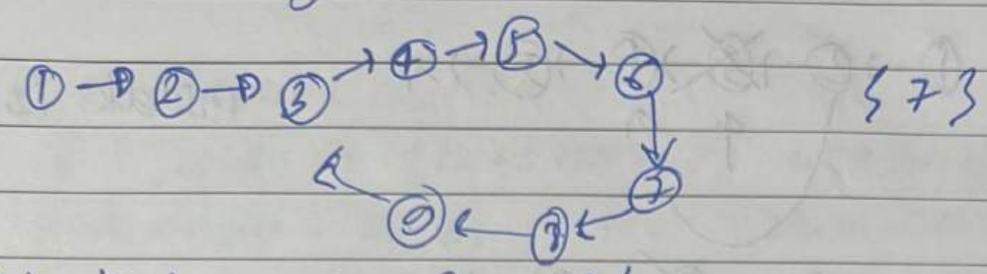
__/__/____

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



__/__/____

(11) { Find the length of a loop in the LL }



{ 7 }

```
int detectLengthLoop(head){
    Node *temp = head;
    int len = 0;
    unordered_map<Node*, int> mapp;
    while(temp){
        if(mapp.find(temp) != mapp.end()){
            return len - mapp[temp];
        }
        mapp[temp] = (len++);
        temp = temp->next;
    }
    return 0;
}
```

TC → O(2N) approach
S → O(N)

{ TC → O(N × length) }
{ SC → O(1) }

Optimal →

```
int optimal(head){
    Node *slow = head, *fast = head;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(fast == slow){
            return length(slow, fast);
        }
    }
    return 0;
}
```

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

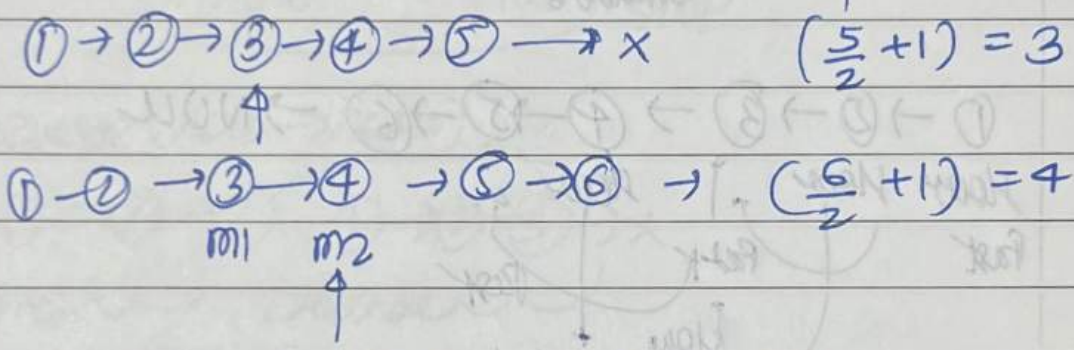
__/__/____

⑦ {Check if LL is Palindrome}

✓ brute {Put all data in stack and check again from Head to tail}

TC → O(2N)
SC → O(N)

✓ {Middle element of the LinkedList} $\left\{\frac{N}{2}+1\right\}$



@ Brute force :-

```
Node* callBrute(head){
    int cnt=0;
    Node* temp=head;
    while(temp){
        temp=temp->next;
        cnt++;
    }
    int mid=cnt/2 + 1;
    temp=head;
    while(temp){
        mid--;
        if(!mid) return temp;
        temp=temp->next;
    }
    return head;
}
```

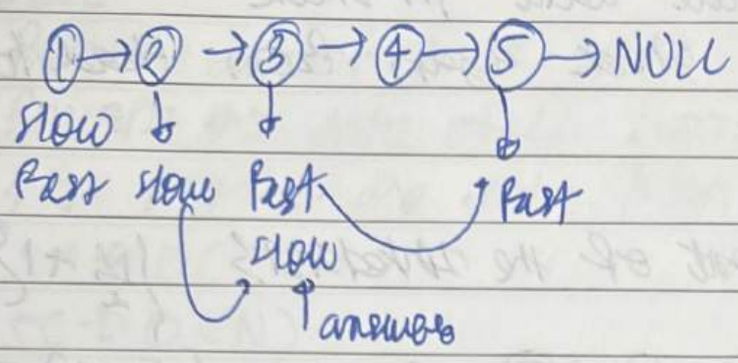
TC → O(2N+N/2) } O(N)
SC → O(1)

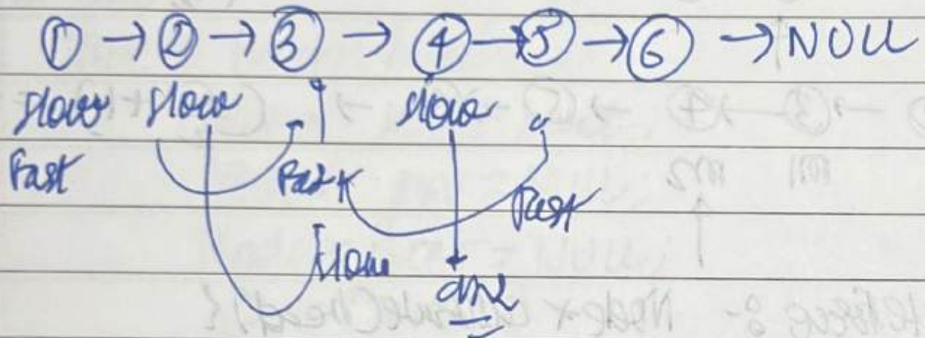
} O(N/2)

{ Tortoise & Hare }

__/__/____

Optimal :- { Tortoise and Hare }





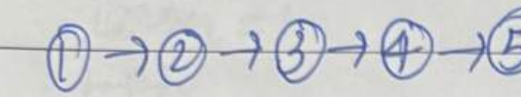

```
Node* callOptHead (head){
    Node* slow = head, fast = head;
    while(fast != NULL && fast->next != NULL){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

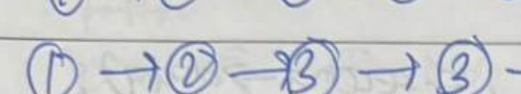
TC → O(N/2)

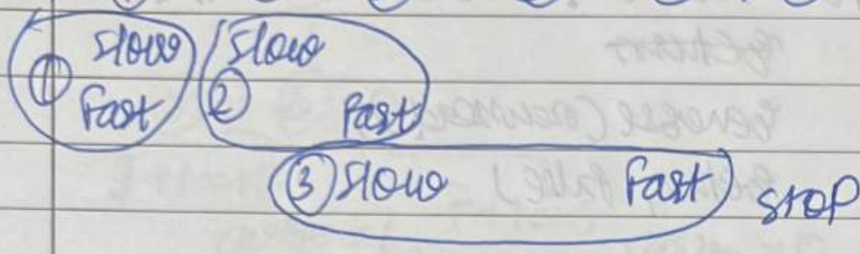
SC → O(1)

__/__/___

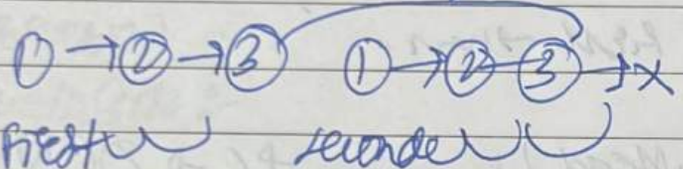
{Palindrome}



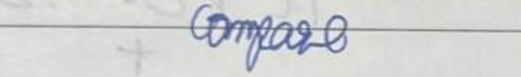


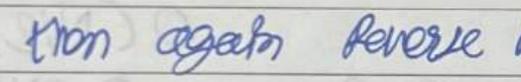


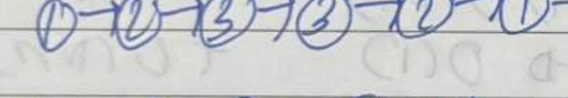
reverse (slow->next);



then again reverse newHead

 } even length

 } odd length



Pseudo Code :-

slow=head, fast=head

while(fast->next != NULL and fast.next.next != NULL){

slow = slow->next;

fast = fast->next->next;

}

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

__/__/____

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

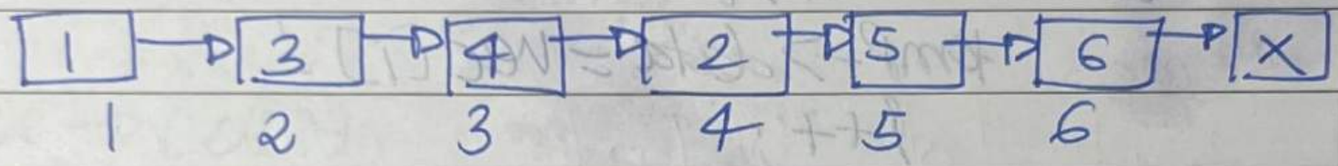
T(n) → O(N/2) + O(N/2) + O(N/2) + O(N/2)

T(n) → O(2N) + O(N/2) + O(N/2)

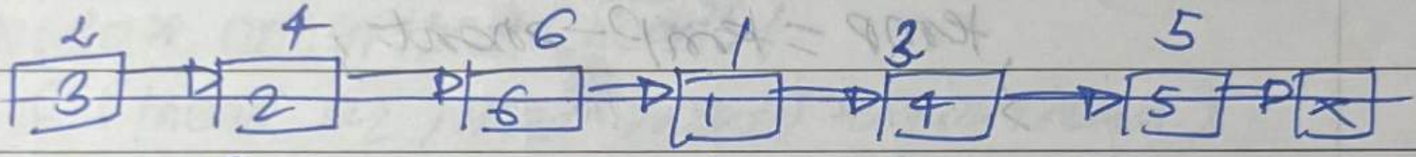
T(n) → O(1) + O(N/2) + O(N/2)

③ {Segregate Even and Odd}

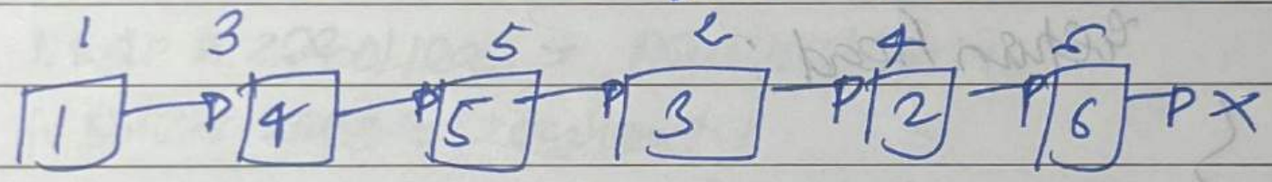
Input {



output {



output {



⊕ brute → store in vector and then change the value in each node

TC → O(2N)
SC → O(N)

{Pseudo code}

```
Node*  callbrute(Node *head){
vector<int> vec;
Node* temp= head;
while(temp!=NULL and temp->next!=NULL)
{  vec.push_back(temp->data);
   temp = temp->next->next;
}
if(temp){ vec.push_back(temp->data) }
```

__/__/____

```
temp = head->next;
while( temp != NULL and temp->next != NULL){
    vec.push_back(temp->data);
    temp = temp->next->next;
}
if (temp){
    vec.push_back(temp->data);
}
temp = head;
int i=0;
while (temp){
    temp->data = vec[i];
    i++;
    temp = temp->next;
}
return head;
}
```

optimal $\overset{\text{TC}}{\approx} O(N)$

TC → O(N/2) ≈ O(N)

SC → O(1)

```
Node* callOptimal (head){
    Node* odd = head;
    Node* even = head->next;
    Node* memoiz = head->next;
    while(even != NULL and even->next != NULL)
    {
        odd->next = odd->next->next;
        even->next = even->next->next;
        odd = odd->next;
        even = even->next;
    }
    odd->next = memoiz;
} return head;
```

⑤ { Remove k^{th} Node from end in 22 }

} Brute {

```
Node* removeKth(head, k){
    int cnt=0;
    Node* temp=head;
    while(temp){cnt++; temp=temp->next;}
    if(cnt==k){ return head->next;}
    int one = cnt-k;
    temp=head;
    while(
```

} edge } →

__/__/__

```
while(temp){
    ans--;
    if(ans==0){
        Node* del = temp->next;
        temp->next = temp->next->next;
        free(del);
        break;
    }
    temp = temp->next;
}
return head;
}
```

TC → O(len) + O(len − N)
SC → O(1)

→ Optimal

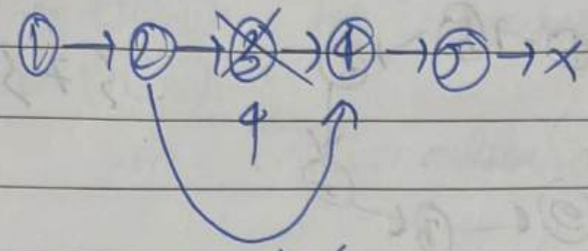
```
Node* removeNth(head, K){
    Node* fast = head;
    Node* slow = head;
    for(int i=0 to K){ fast = fast.next; }
    while(fast){
        fast = fast.next;
        slow = slow.next;
    }
    if(fast==NULL){ return head->next; }
    If Node* del = slow->next;
        slow->next = slow->next->next;
        free(del);
        return head;
}
```

TC → O(len)
SC → O(1)

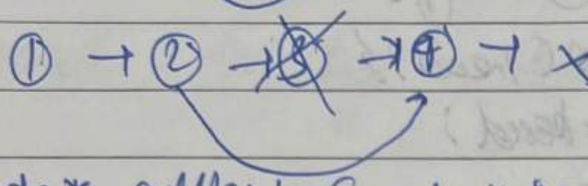
① → ② → ✕

__/__/__

(12) { Delete the middleNode of LL }



middle is { $\frac{N}{2}+1$ }



```
Node* callRuse(Node* head){
  if(!head or !head->next){
        return NULL;
  }
  int cnt=0;
O(N){ temp=head while(temp){cnt++, temp=temp->next;}
    cnt = (cnt/2)
  temp=head while(temp){
        cnt--;
        if(cnt==0){
            Node* middle = temp->next;
            temp->next = temp->next->next;
            free(middle);}
            break;
        }
        temp=temp->next;
    }
    return head;
}
```

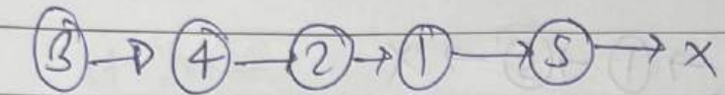
T(n) → O(N) + O(N/2)

S(n) → O(1)

__/__/____

(20) Last 12

→



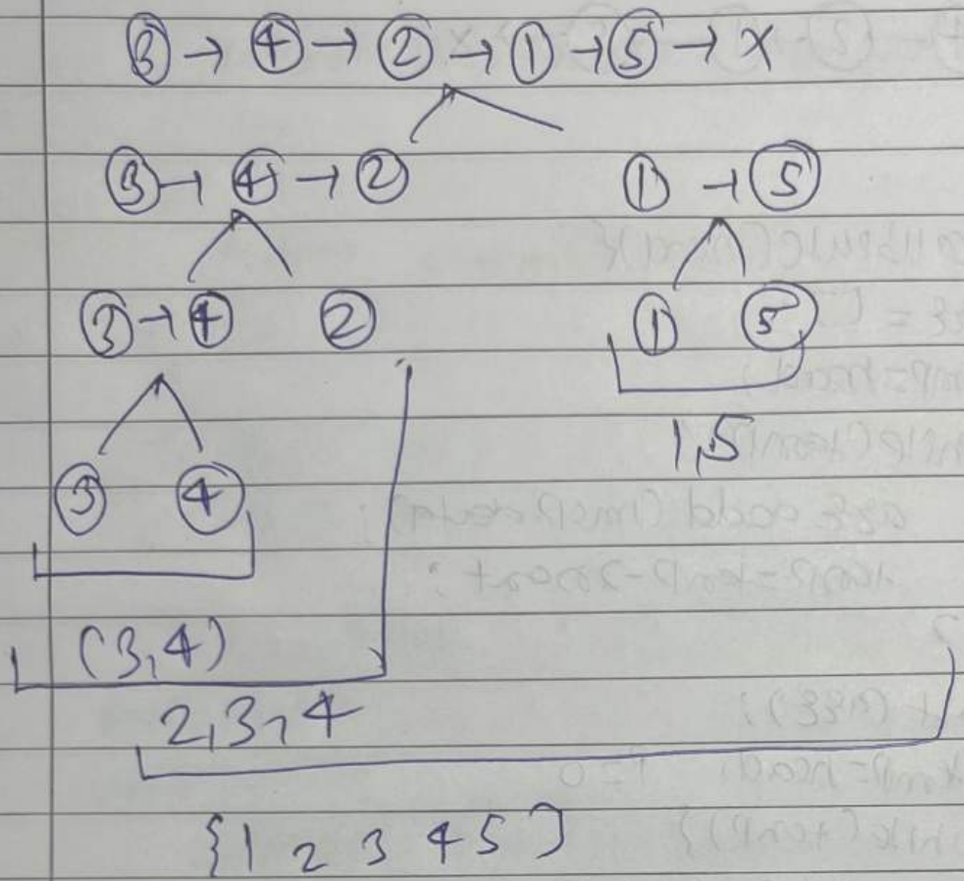

```
Node* callmerge(head){
    arr = [];
    temp=head;
    while(temp){
        arr.add(temp.data);
        temp=temp->next;
    }
    sort(arr);
    temp=head;  i=0
    while(temp){
        temp=arr[i];
            i++
        temp=temp->next;
    }
    return head;
}
```

TC → O(N log N + N + N)

SC → O(N).

__/__/____

{ applying merge sort }




```
mergesort( arr, low, high)
    if( low == high){
        return;
    }
    mid = (low + high) /2;
    mergesort(arr, low, mid);
    mergesort(arr mid+1, high);

    merge (arr, low , mid high)
}
```

__/__/___

```
if(head){

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

    lefthead = head
    righthead = middle->next
    middle->next = NULL;

    lefthead = f(lefthead);
    righthead = f(righthead);

    return merge2(lefthead, righthead);
}
```

↳ See the Platform U Code

⑧ Add 1 to LinkedLists

Input: ① → ⑤ → ⑨ → x 159

output: ① → ⑥ → ⓪ → x 1 / 160

Input: ⑨ → ⑨ → ⑨ → x

output: ① → ⓪ → ⓪ → ⓪ → x

__/__/___

① → ⑤ → ⑤ → x

carry = 1

~~++~~.

⑤ → ⑤ → ① → *

⑨ → ⑥ →

9+1=10

carry = 1 5+1=0

carry = 0

break;

~~(crossed out)~~

Pseudo Code :-

```
Node* calculate (head){
    newhead = recursion(head)
    int carry = 1
    dummyNode = new Node(-1).
    temp = dummyNode;
    while(newhead)
        sum = carry;
        if(newhead){
            sum += newhead->data;
            newhead = newhead->next;
        }
        dummyNode->next = new Node(sum%10);
        dummyNode = dummyNode->next;
        sum/=10;
        carry = sum;
    }
```

__/__/___

```
if (carry) {
    dummyNode->next = new Node(carry);
}

return recursion(temp->next);
}
```

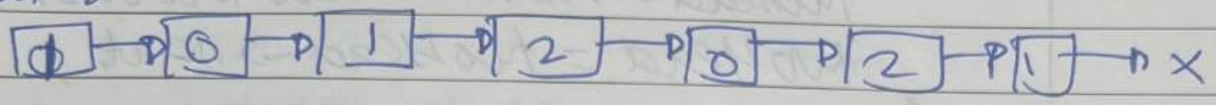
~~~~~~~~~~

```
int recursionOptimize(temp){
    if (!temp) return 1;
    int carry = recursionOptimize(temp->next);
    temp->data = temp->data + carry;
    if (temp->data < 10) return 0;
    temp->data = 0;
    return 1;
}

Node* callOptimal(head){
    carry = recursionOptimize(head);
    if (carry){
        Node* newNode = new Node(carry);
        newNode->next = head;
        return newNode;
    }
    return head;
}
```
~~~~~~~~~~

④ {Sort 0, 1, 2}

Brute:



cnt0 = 2
cnt1 = 3
cnt2 = 2
} { Replace data of LL as per the count }

TC → O(2N)
SC → O(1)

Optimal :-

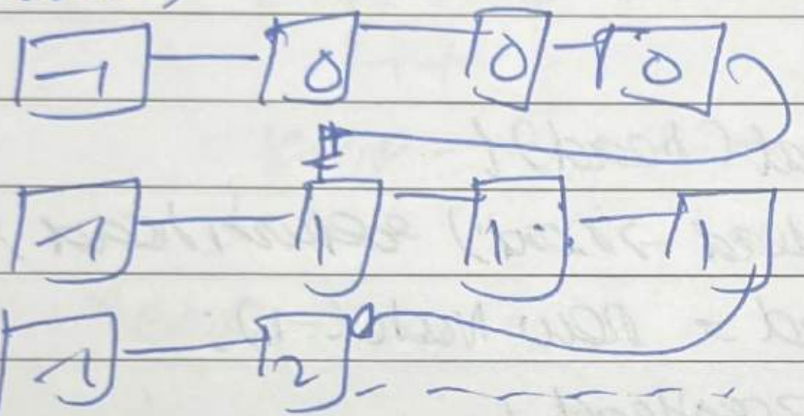
TC → O(O(N))
SC → O(1)

```
Node* callOptimal(head){
if (!head or (Head->next) return head;
 Node *zeroHead = new Node(-1);
 Node* zero = zeroHead;
      oneHead = new(Node(-1);
     one = oneHead.
     twoHead = new Node(-1);
     two = twoHead;
     Node* temp = Head;
     while(temp){
        if(temp->data ==0){
           zeroHead->next = new Node(0);
           zeroHead = = zeroHead->next;
         }
      else if(temp->data ==1){
           oneHead ->next =new Node(1);
           oneHead = oneHead -> next;
```

__/__/____

```
else {
    twoHead->next = new Node(2);
    twoHead = twoHead->next;
}
temp = temp->next;
zeroHead->next = (one->next) ? one->next :
                                two->next;
oneHead->next = two->next;
return zero->next;
}
```

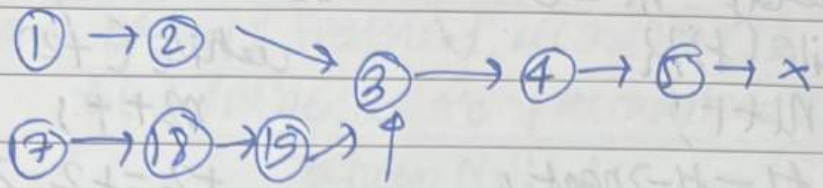
* approach



connect all 0

__/__/___

⑨ { Find the Intersection Point of 4 Linked-List



bruteForce :-

```
Node* callBrute(firstHead, secondHead){
    Node* ans = NULL;
    unordered_map<Node*, int> mapp;
    while(firstHead){
        mapp[firstHead]=1;
        firstHead = firstHead->next;
    }
    while(secondHead){
        if(mapp.find(secondHead) != mapp.end()){
            ans = secondHead;
            break;
        }
        secondHead = secondHead->next;
    }
    return ans;
}
```

↑ unordered map

TC → O(N1 * 1) + O(N2 * 1)

SC → O(N1)

↳ store first Head

better approach

```
t1=head1  n1=0                    t2=head2  n2=0
while(t1){                        while(t2){
  n1++;                             n2++;
  t1=t1->next;                      t2=t2->next;
}                                 }
O(N1)                             O(N2)

if(n1<n2){                        t1 -> smaller  t2 -> greater
  return collisionPoint(head1, head2, n2-n1);}
else{ return collisionPoint(head1, head2, n1-n2);}

Node* collisionPoint(t1, t2, dist){
  while(dist--){                  O(N2-N1)
    t2=t2->next;
  }
  while(t1 && t2){                O(N1)
    if(t1==t2) return t1;
    else if(!t1){ return NULL; }
    else if(!t2){ return NULL;}
    t1 = t1->next;
    t2 = t2->next;
  }
  return NULL;
}
```

TC → O(N1) + O(N2) + O(N2-N1) + O(N1)

→ O(N1 + O(2N2))

→ O(N1 + 2N2)

SC → O(1)

__/__/____

{ Optimal Approach }

```
Node * Optimal ( FirstHead, SecondHead ){
    if( !FirstHead or !SecondHead ){
        return NULL;
    }
    Node * temp1 = FirstHead;
    Node * temp2 = SecondHead;
    while (temp1 != temp2){
        temp1 = temp1->next;
        temp2 = temp2->next;
        if (temp1 == temp2) return temp1;
        if (!temp1) temp1 = SecondHead;
        if (!temp2) temp2 = FirstHead;
    }
    return temp1;
}
```

TC → O(N+M)

SC → O(1)

__/__/____

② { Add two numbers in LL }

Input { 2 → 4 → 6 → X (6 4 2)
3 → 8 → 7 → X (7 8 3)
14 2 5

output { 5 → 2 → 4 → 1 → X

{ No need to reverse linked list }

2 4 6 X
3 8 7 X

-1 → 5 → 2 → 4 → 1 → X

Return dummy->next

carry=0 | carry=1 | carry=1 | carry=0
12+0 | 13+1
=12 | =14

Pseudo code:-

```
Node* func(Node* n1, Node* n2){
    int carry=0, sum=0;
    Node* dummyNode = new Node(-1);
    Node* current = dummyNode;
    while(num1!=NULL or num2!=NULL){
        sum=carry;
        if(num1){ sum+=num1->data; num1=num1->next; }
        if(num2){ sum+=num2->data; num2=num2->next; }
        Node* temp = new Node(sum%10);
        current->next = temp;
        current = current->next;
        carry /= 10;
    }
```

__/__/____

```
if(carry){
    current->next = new Node(carry%10);
    return dummyNode->next;
  }
current->next = NULL;
return dummyNode->next;
}
```