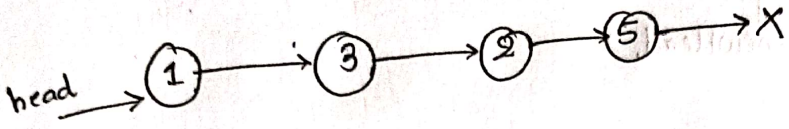
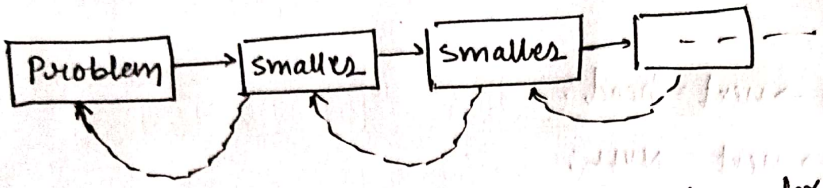
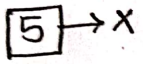
REVERSE A LINKED LIST (RECURSIVE SOLUTION).





first we try to solve it for 3 nodes → than for 2 nodes and than for one nodes.

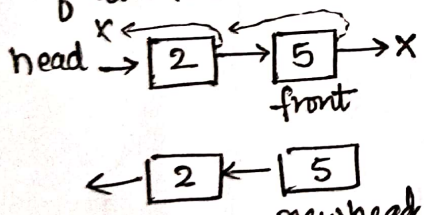
(a) if linked list has one node.



no need to reverse, just return the node, thus it can become my base case.

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

(b) if linked list has two nodes.

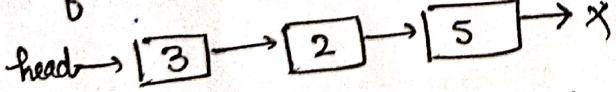


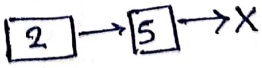

```
front -> next = head
head -> next = null.
return front;
```

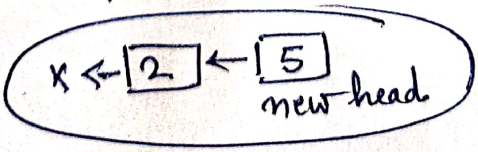
→ this front will be my newhead.

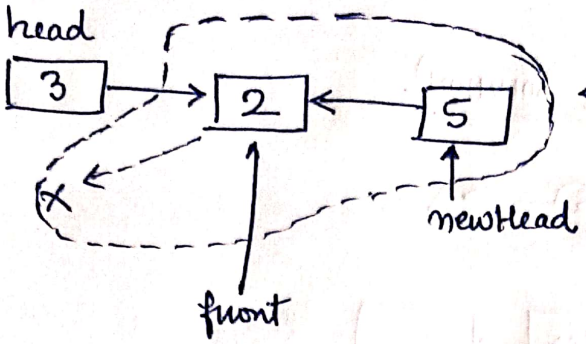
↑ and we return this newHead as an answer.

(c) if linked list has three nodes.



↳ so we call the sub-problem 

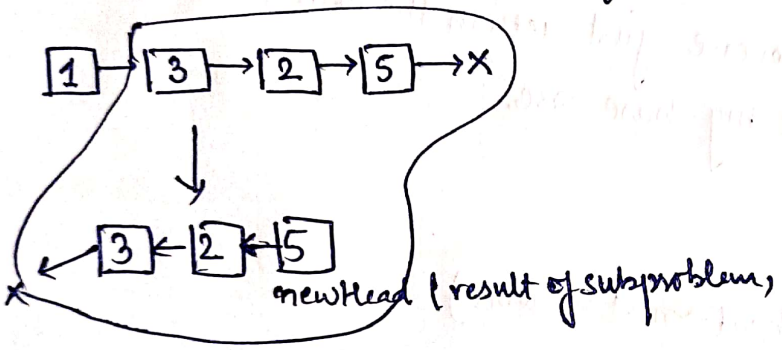
it will return 

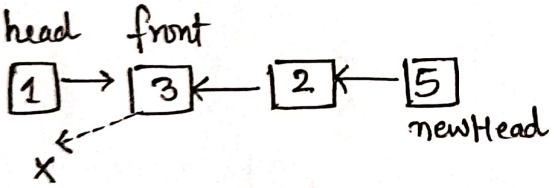


← This is value return from the previous subproblem.

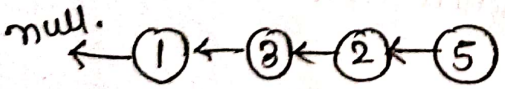
```
front -> next = head;
head -> next = NULL;
return newhead.
```

(d) Now consider the linked list of 4 nodes.






```
front -> next = head
head -> next = null
return newHead.
```

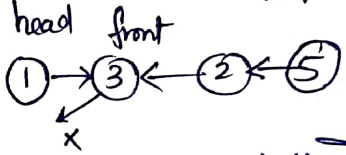


Code.

```
Reverse (head)
{
  if (head==NULL || head->next==NULL)        // for One Node or Zero Node.
      return head;
}
Node * newHead = Reverse (head->next)
Node *front= head->next
front->next= head
head->next= null
return newhead
```

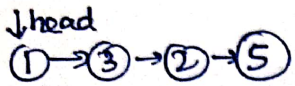
this head of reversed link.

Reverse(4)
↓
Reverse(3)
↓
Reverse(2)
↓
Reverse(1).

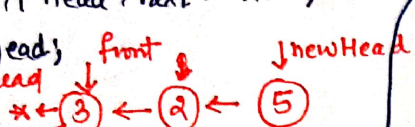


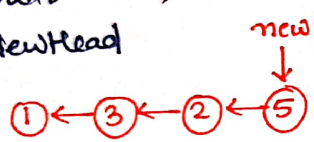
When you got the answer.

RECURSION OF REVERSING A LINKED LIST.

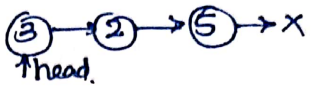


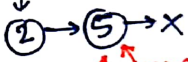

```
reverse (head)
{
if (head==NULL || head->next==NULL)
    return head;
}
Node * newHead = reverse (head->next);
Node * front = head->next
front->next = head;
head->next = NULL;
return NewHead
}
```






```
reverse (head)
{
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
}
```

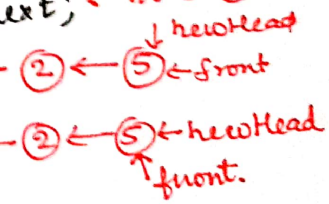





```
reverse (head)
{
if (head==NULL || head->next==NULL)
    return head;
}
Node * newHead = reverse (head->next)
Node * front = head->next;
front->next = head;
head->next = NULL;
return NewHead;
}
```

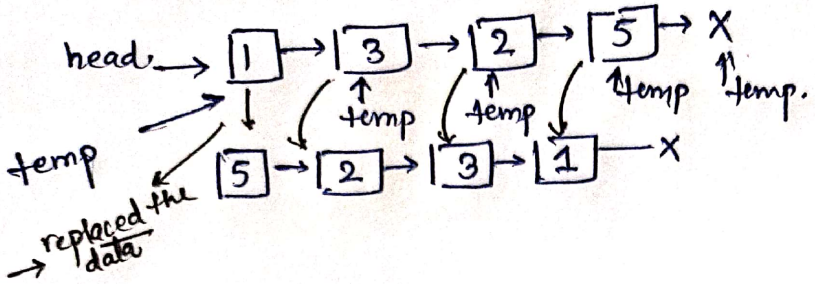
newHead
5

here 2 is act head.

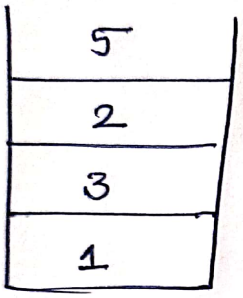



```
reverse (head)
{
if (head==NULL || head->next==NULL)
    return head;
}
Node * newHead = reverse (head->next)
Node * front = head->next;
front->next = head;
head->next = NULL;
return NewHead
}.
```

REVERSE A LINKED LIST.



Brute force.



Look at the stack

Step1 :-

```
temp = head          stack st

while (temp! =NULL)
  {
    st.push (temp -> data)
      temp = temp -> next;
  }
```

step1 → here we are putting all the data into the stack.

T.C = O(N)

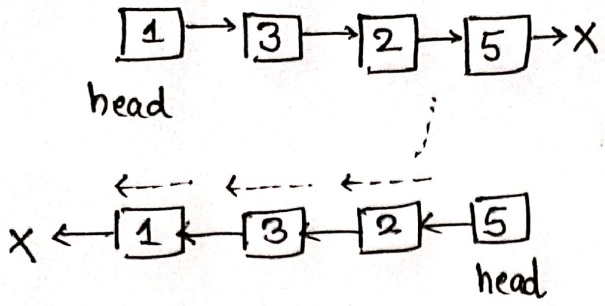
```
temp = head
while(temp! = null)
  {
    temp -> data = st.top();
       st.pop();
     temp = temp -> next;
  }
```

step2 → Taking back from stack and putting it back on the linked list.

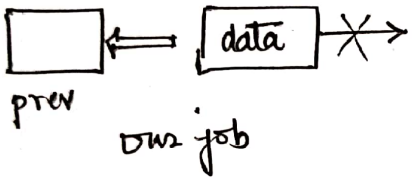
T.C = O(N).

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

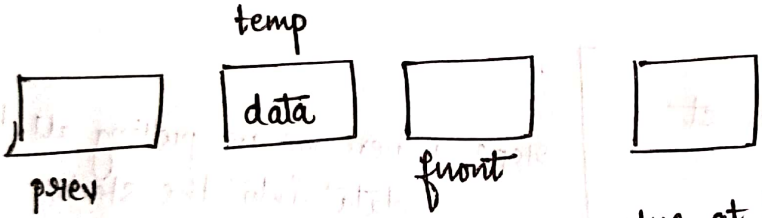
→ OPTIMISED CODE :→



→ Our main concern → is to set up next.



Our job



front = temp→next (store the value at front).

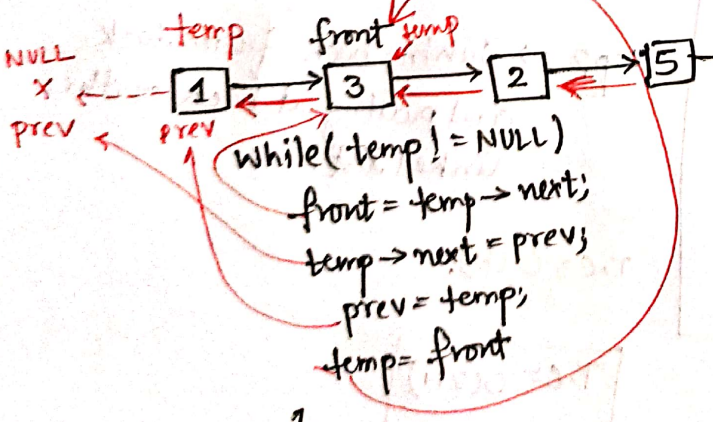
temp → next = prev

prev = temp.

~~front~~ temp = front (as you already stored the value of front = temp→next).

(and as we again have to perform the job).

~~prev~~ =

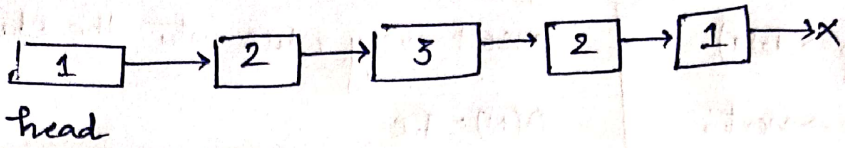


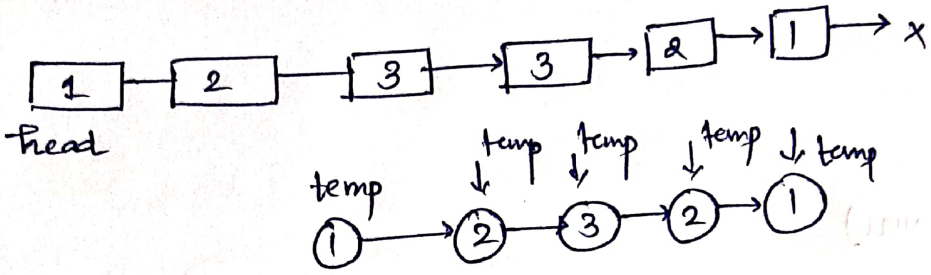

```
while(temp != NULL)
    front = temp->next;
    temp->next = prev;
    prev = temp;
    temp = front
}
return prev.
```

T.C = O(N)
S.C = O(1)

Changing the links only.

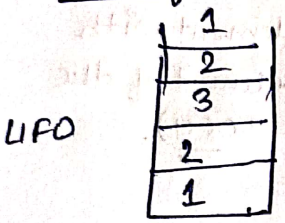
PALINDROME OF A LINKED LIST



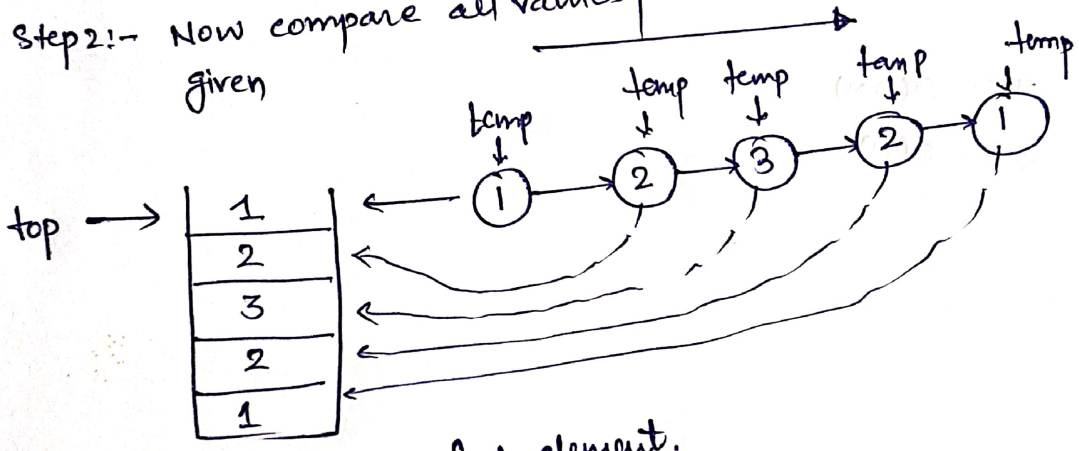


Brute force.

Step1 → put all values into the stack.



Step2:- Now compare all values present in the stack with linked list given



top element is the last element.

top match with temp

```
Stack st          temp = head
while(temp != NULL)
  {
   st.push(temp -> data)
     temp = temp -> next;
  }
```

Step 1.
Insert every value into the stack.
O(N) = T.C

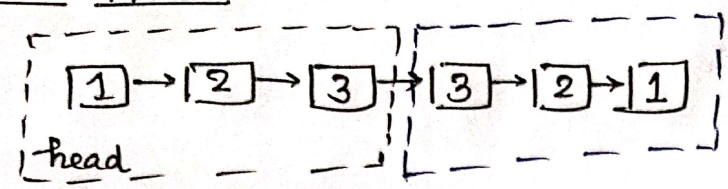
```
temp = head
While (temp != NULL)
  {
    if(temp -> data != st.top()) return false;
     temp = temp -> next;
      st.pop();
  }
return true.
```

→ Step 2 → comparing top element with every element of the stack. → O(N).

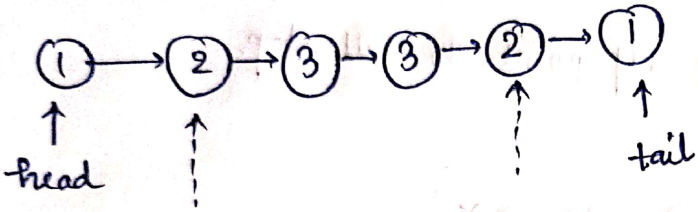
T.C = O(2N)

S.C = O(N)

optimal approach.

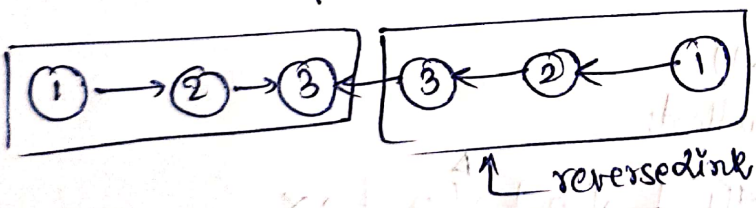


- if first half compared with second half.

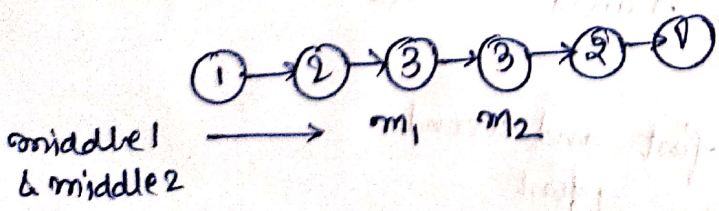


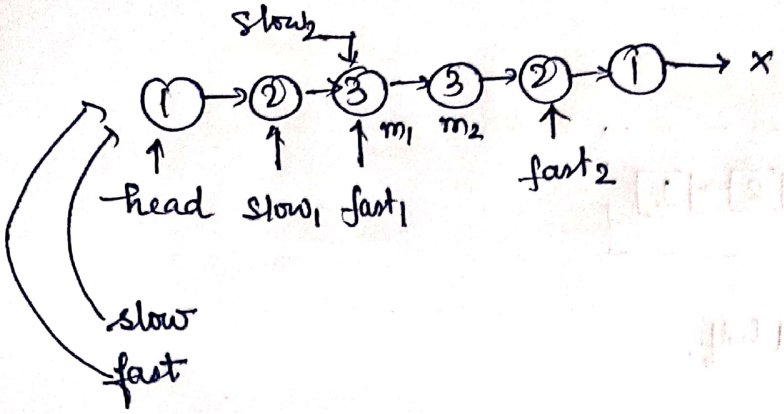
now match head with the tail. But there is a issue that it is a singly linked list. Tail pointer can't move backward, we have to reverse the linked list. (Second half of the past).

- First discuss the case of EVEN LENGTH LINKED LIST.
  we have to compare first half with second half.



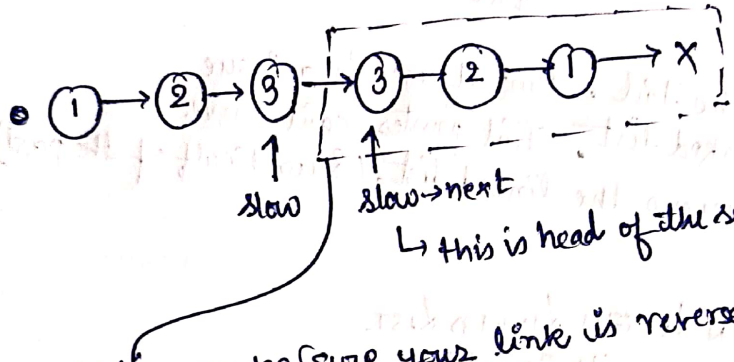
- So first we have to recognise, what will be the second half.
  Use TORTOISE & HARE ALGORITHM.





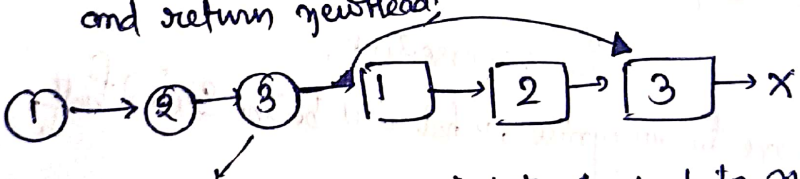
- here in this example, we are at m_1 and slow pointer is present here and fast-pointer is present at second last node.

  ↳ fast → next → next if points to null stop.



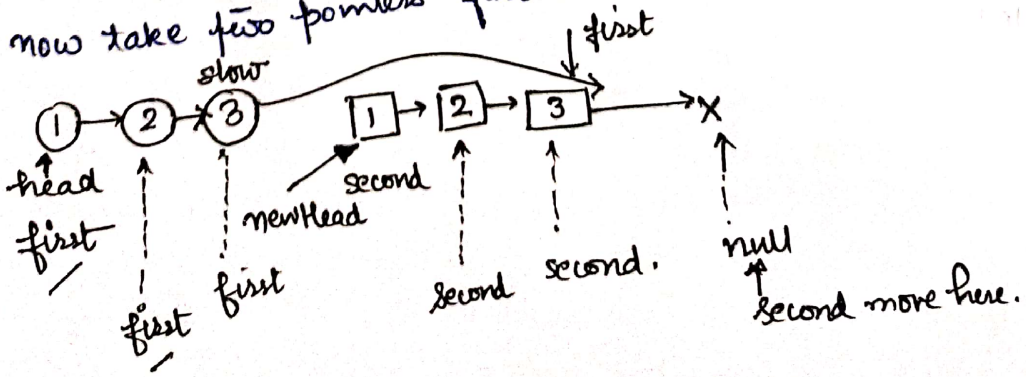
→ Step 2: make sure your link is reversed. of second half of the linked list.

reverse (slow→next)
and return newHead;



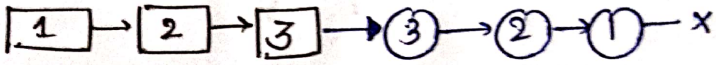
this (3) node not point to 1, point to node (3) because you never changed the link.

→ now take two pointers first and second

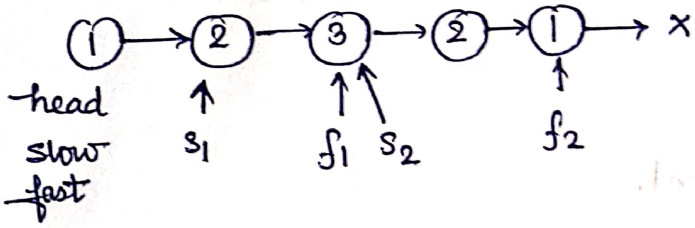


As second reaches the Null you should stop.

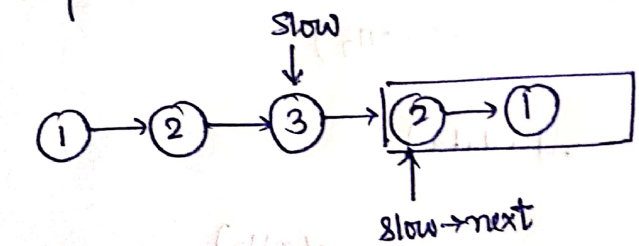
→ make sure that before returning true, reverse (newHead) again. to make the linked list as it was before.
reverse the second half of the linked list again.

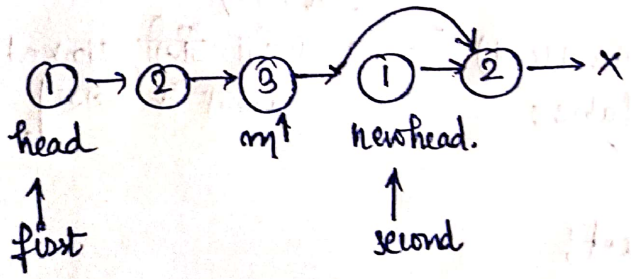


Case 2. When the linked list is odd.



- stop at the last node. means you stop where fast → next = NULL.





Step 1: find the middle of the linked list.

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

$O(N/2)$ → go half way to find out the middle.

Step 2 reverse the second half of the linked list.

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

$O(N/2)$ → because reversing the half part of the linked list.

Step 3. comparing first and second.

```
first = head    second = newhead.
while( second != Null)  ——————  O(N/2)
    {
    if ( first -> data != second -> data)
        {
            reverse (newHead)  ——  O(N/2)
            return false;
        }
    first = first -> next;
    second = second -> next;
    }
reverse( newHead);
return true;
}
```

पहले जैसी Linked List करने के लिए।

T.C = $O(N/2) + O(N/2) + O(N/2) + O(N/2)$

= $O(2N)$.

S.C = $O(1)$ using pointers.