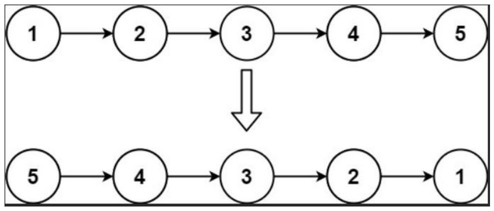
Day 5: Linked List

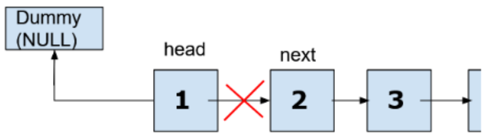
1. Reverse Linked List (Leetcode 206)

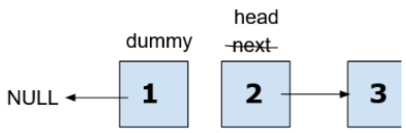
Problem statement: Given the head of a singly linked list, reverse the list, and return the reversed list.



Approach 1:

- First take a dummy node that will be assigned to NULL.
- Then another pointer "Next pointer" which will be initialized to **head->next**
- Update the next pointer of the head to the dummy node. (**head.next->dummy**) // *reverse operation*
- Then assign Dummy is set as head and Head is set as next (**dummy= head, head=next**)
- Then assign next as **head->next** and continue process of reversing





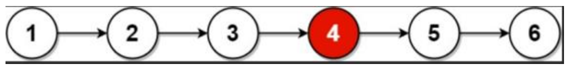
Basic Code:

```
ListNode prev=null;

while(head!=null){
      ListNode  temp=head.next;
        head.next=prev;
        prev=head;
        head=temp;
     }

     return prev;
```

Approach 2: Recursive

2. Find the middle of LinkedList (Leetcode 876)

Given the head of a singly linked list, return the middle node of the linked list.
If there are two middle nodes, return the second middle node.



Approach 1: Naïve Approach
Traverse through the Linked List while maintaining a count of nodes (say n), and then traversing for 2nd time for n/2 nodes to get to the middle of the list.

Approach 2: Tortoise-Hare-Approach

Initialize two pointers. Move slow ptr by one step and simultaneously fast ptr by two steps until fast ptr is NULL or next of fast ptr is NULL.

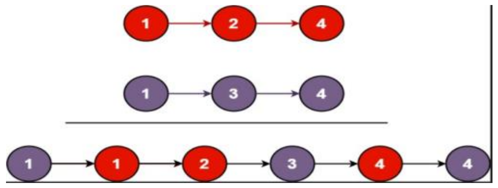
Code:

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

Time Complexity: O(N)
Space Complexity: O(1)

3. Merge two sorted Linked List (use method used in mergeSort)

You are given the heads of two sorted linked lists list1 and list2.
Merge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.



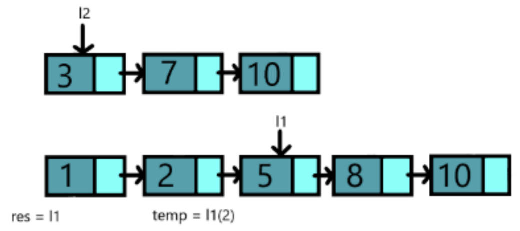
Approach 1: Using an externally linked list to store answers.

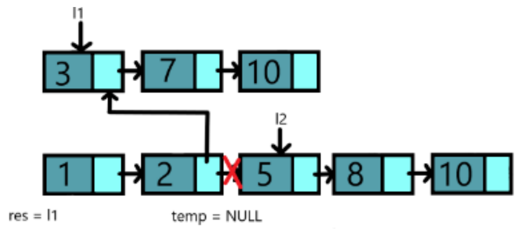
- Create a new dummy null node. This will be the head of the new list. Another pointer to keep track of traversals in the new list.
- Find the smallest among two nodes pointed by the head pointer of both input lists, and store that data in a new list created. Move the head pointer to the next node of the list whose value is stored in the new list.
- Repeat the above steps till any one of the head pointers stores NULL. Copy remaining nodes of the list whose head is not NULL in the new list.

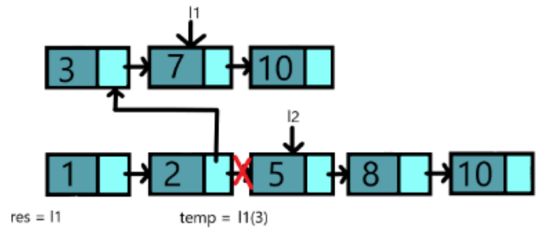
Time Complexity: O(N+M).
Space Complexity: O(N+M).

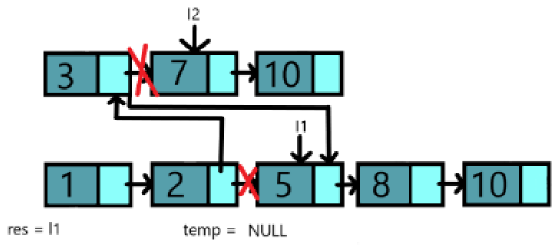
Approach 2: Inplace method without using extra space.

- Create two pointers, say l1 and l2. Compare the first node of both lists and find the small among the two. Assign pointer l1 to the smaller value node.
- Create a pointer, say res, to l1. An iteration is basically iterating through both lists till the value pointed by l1 is the highest value which is less than or equal to the value pointed by l2.
- Start iteration. Create a variable, say, temp. It will keep track of the last node sorted list in an iteration.
- Once an iteration is complete, link node pointed by temp to node pointed by l2. Swap l1 and l2.
- If any one of the pointers among l1 and l2 is NULL, then move the node pointed by temp to the next higher value node.









Code:

```
     if(list1.val> list2.val) {
     //swap
     }
     ListNode res=list1;
     ListNode tmp=null;

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

while (list1! =null && list1.val<=list2.val){
        tmp=list1;
        list1=list1.next;
       }
     tmp.next=list2;

   //swap
  }

  return res;
```

Swap fn:

```
ListNode temp=list1;
       list1=list2;
```

Time Complexity: O(N+M).
Space Complexity: O (1).