Linked-List 2

Now moving further with the topic, let's try to solve some problems now...

The Midpoint of A Linked List

The Trivial Approach- Two Passes

- This approach requires us to traverse through the linked list twice i.e. 2 passes.
- In the first pass, we will calculate the **length** of the linked list. After every iteration, update the **length** variable.
- In the second pass, we will find the element of the linked list at the **(length-1)/2th** position. This element shall be the middle element in the linked list.
- However, we wish to traverse the linked list only once, therefore let us see another approach.

The Optimal Approach- One Pass

- The midpoint of a linked list can be found out very easily by taking two pointers, one named **slow** and the other named **fast**.
- As their names suggest, they will move in the same way respectively.
- The **fast** pointer will move ahead **two pointers at a time**, while the **slow** pointer one will move at a speed of **a pointer at a time**.
- In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.
- These pointers will be updated like this:
    - `slow = slow.next`
    - `fast = fast.next.next`

Java Code

```java
public static Node returnMiddle(Node headNode){
    if (headNode == null || headNode.next == null)
        return head;
    Node slow = headNode; //Slow pointer
    Node fast = headNode.next; //Fast Pointer
    while (fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow; // Slow pointer shall point to our middle element
}
```

Note:

- For odd length there will be only one middle element, but for the even length there will be two middle elements.
- In case of an even length LL, both these approaches will return the first middle element and the other one will be the direct **next** of the first middle element.

Merge Two sorted linked lists

- We will be merging the linked list, similar to the way we performed merge over two sorted arrays.
- We will be using the two **head** pointers, compare their data and the one found smaller will be directed to the new linked list, and increase the **head** pointer of the corresponding linked list.
- Just remember to maintain the **head** pointer separately for the new sorted list.
- And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list.
- Try to implement this approach on your own.

Mergesort over a linked list

- Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list.
- Just the difference is that in the case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach.
- The merging part of the divided lists can also be done using the **merge sorted linked lists code** as discussed above.
- The functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.
- Try to implement this approach on your own.

Reverse the linked list

Recursive approach:

- In this approach, we will store the last element of the list in the small answer, and then update that by adding the next last node and so on.
- Finally, when we will be reaching the first element, we will assign the **next** to **null**.
- Follow the Java code below, for better understanding.

```
public static Node reverseLinkedList(Node head){
    if (headNode == null || headNode.next == null)
        return head;

    Node smallHead = reverseLinkedList(head.next);
    Node tail = smallHead;
    while(tail.next != null){
        tail = tail.next;
    }
    tail.next = head;
    head.next = null;
    return smallHead;
}
```

After calculation, you can see that this code has a time complexity of **O(n^2)**. Now let's think about how to improve it.

Recursive approach (Optimal):

- There is another recursive approach to the order of **O(n)**.
- What we will be doing is that head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing.
- Check out the given code for your reference:

```
class Pair{
    Node head;
    Node tail;
    public Pair(Node head, Node tail){
        this.head = head;
        this.tail = tail;
    }
}

class main{
    private static Pair reverse2Helper(Node head){
         if (head == null || head.next == null)
              return new Pair(head, head);

         Pair p = reverse2Helper(head.next);
         p.tail.next= head;
         head.next = null;
         return new Pair(p.head,head);
    }

    private static Node reverse2(Node head){
         return reverse2Helper(head).head;
    }
    // Main driver function can be written by yourself
}
```

Now let us try to improve this code further.

A simple observation is that the **tail** is always **head.next**. By making the recursive call we can directly use this as our **tail** pointer and reverse the linked list by **tail.next = head**. Refer to the code below, for better understanding.

```
public static Node reverse3(Node head){
      if (head == null || head.next == null)
           return head;

      smallHead = reverse3(head.next);
      tail = head.next;
      tail.next = head;
      head.next = null;
      return smallHead;
}
```

Iterative approach:

- We will be using three-pointers in this approach: **previous, current,** and **next.**
- Initially, the **previous** pointer would be **null** as in the reversed linked list, we want the original head to be the last element pointing to **null** .
- The **current** pointer will point to the current node whose **next** will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element.
- Similarly, iteratively, we will keep updating the pointers as **current** to the **next**, **previous** to the **current,** and **next** to **current**'s **next**.

Refer to the given Java code for better understanding:

```
public static Node reverse(Node head){
      if (head == null || head.next == null)
           return head;

    Node prev = null;
    Node curr = head.next;
    Node next = curr.next;

    while (next != null){
        curr.next = prev;
        prev = curr;
        curr = next;
        next = next.next;
    }
    curr.next = prev;
    return curr;
}
```