Linked List 2

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

Here we are only gonna discuss the approach, but you will have to implement it yourselves and in case you are stuck then refer to the solution section of the corresponding problem.

Midpoint of LL

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. Fast pointer will move ahead two pointers at a time, while the slower 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.

They will move like these:

slow = slow -> next;

fast = fast -> next -> next;

Also, be careful with the even length scenario of the linked lists. For odd length there will be only one middle element, but for the even length there will be two middle elements. The above approach will return the first middle element and the other one(in case of even length list) 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. Now try to code it...

Mergesort over 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 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 and merging part of the divided lists can also be done using the merge sorted linked lists code as discussed above. Basically the functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.

Reverse the linked list

Recursive approach:

Basically, 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 code below, for better understanding...

```
Node* reverseLL(Node *head) {
        if(head == NULL || head -> next == NULL) {            //Base case
                return head;
        }

        Node *smallAns = reverseLL(head -> next);             // Recursive call

        Node *temp = smallAns;                                // small answer that
        while(temp -> next != NULL) {                         // stores the reversed
                temp = temp -> next;                          // list by traversing the
        }                                                     // reversed list and
                                                              // then appending the
        temp -> next = head;                                  // next element to it.
        head -> next = NULL;
        return smallAns;
}
```

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

There is another recursive approach in which we can simply use the O(n) approach. What we will be doing is creating a pair class that will be storing the reference of not only the 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. Checkout the code for your reference...

```
class Pair {
        public :                            //Pair class about which we were talking above
                Node *head;
                Node *tail;
};

Pair reverseLL_2(Node *head) {
        if(head == NULL || head -> next == NULL) {          //Base case
                Pair ans;
                ans.head = head;
                ans.tail = head;
                return ans;
        }

        Pair smallAns = reverseLL_2(head -> next);          //Recursive call

        smallAns.tail -> next = head;                       // you can see that the time
        head -> next = NULL;                                // is reduced as we do
        Pair ans;                                           //not need to find the tail
        ans.head = smallAns.head;                           // pointer each time
        ans.tail = head;
        return ans;
}
```

Now improving this code further...

A simple observation is that the tail is always the 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...

```
Node* reverseLL_3(Node *head) {
        if(head == NULL || head -> next == NULL) {                  //Base case
                return head;
        }
        Node *smallAns = reverseLL_3(head -> next);                 //Recursive call

        Node *tail = head -> next;                                  //Small calculation
        tail -> next = head;                                        //discussed above
        head -> next = NULL;
        return smallAns;
}
```

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. Current pointer will be 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.

You will be solving this problem yourself now...

Variations of the linked list

In the lecture notes of Linked list - 1, we have already seen the three different types of linked list and discussed them diagrammatically also. Prefer to that section for the reference...

Practice problems

Try over the following link to practice some good questions related to linked lists:

https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=linked-lists