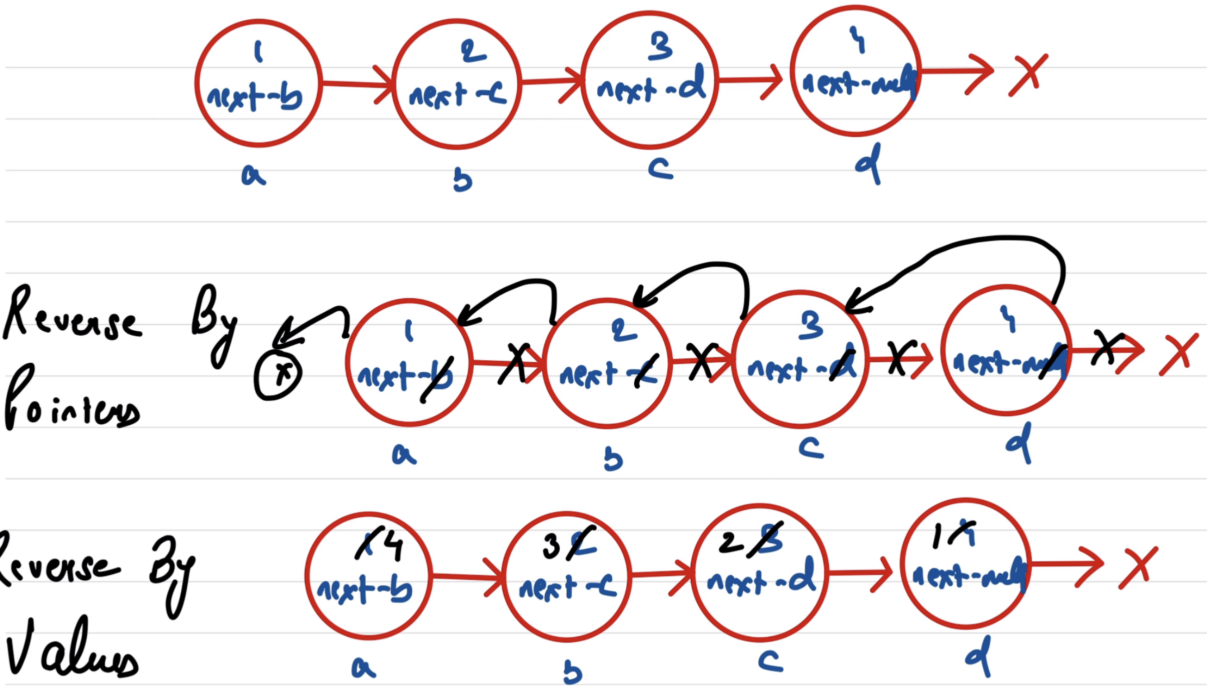
Problem Solving On LL

Created	@August 10, 2022 8:26 PM
Class	
Type	
Materials	
Reviewed	☐

Problem 1

Given a Singly Linked List, reverse the linked list. We have to devise the mechanism to reverse a LL, both by values and by pointers.



So by saying reverse by pointers, we mean that for every node we should only change the value of `next` that will help us to manipulate where the node points. Whereas, by saying reverse by value we need to change the value of the node and not hamper the next parameter.

Reverse by Pointers

For reversing a LL by pointers, we have two strategies.

- Iterative
- Recursive

Pointer Reverse Iterative

At last in the LL, we want to update every node's next pointer to the previous node. So with every node, we have to maintain it's previous node also.

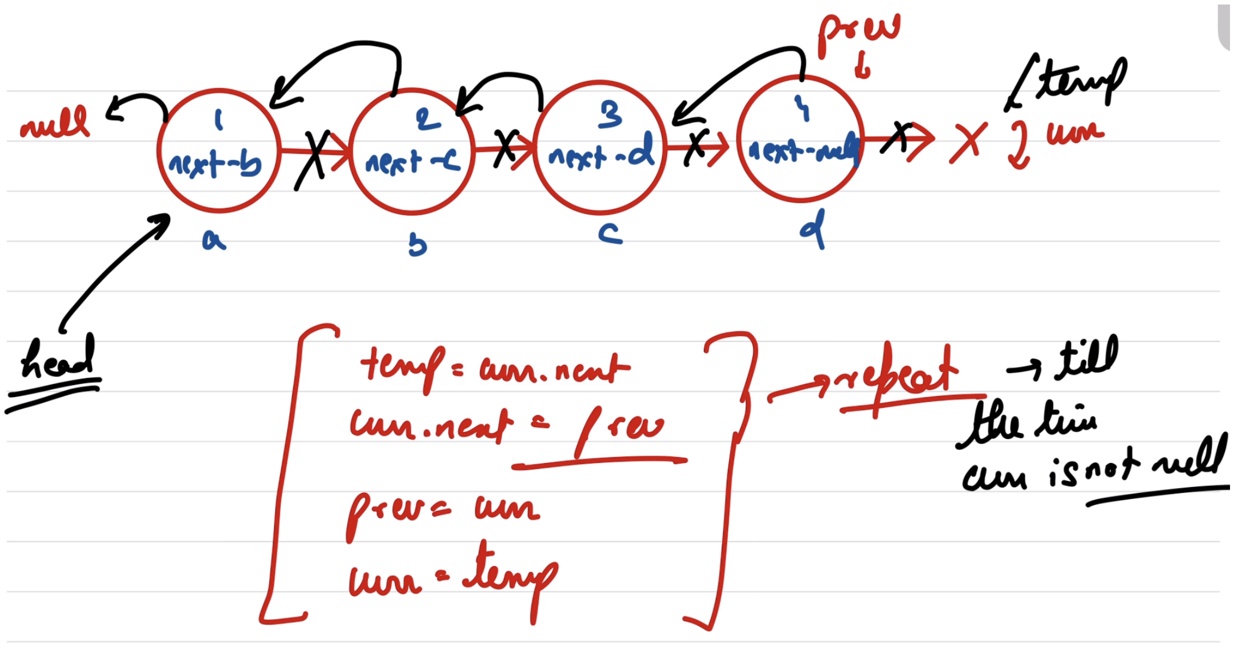
In a Singly LL, we only have access to the head, so we need to start with the head only. So if the current node is head, what should be it's previous node ? (hint: prev node will the new value of current node's next pointer).

After the reversal, head's next will point to null, so when the current node is Head then previous should be null.

Now we want to mark the next of current node to previous by doing `curr.next = null` but there is a problem. If we execute this statement, we will loose the access of the remaining list. That means, we need to maintain an access to the remaining list.

So before we execute `curr.next` `= null` we need to store the access of remaining list. So in a new temp variable we can store `curr.next` by doing `temp = curr.next`.

By doing this, we will be able to reverse the first node, and manage the access to the remaining list. So now we have to just repeat the process on the remaining list. So in the next iteration, curr will become prev, and temp will become curr. And we should repeat this process till the time curr is null. At the last of the iteration, curr will be null, temp will be null and prev will be the last node. And this last node should be the head, so we need to update the head to the prev.

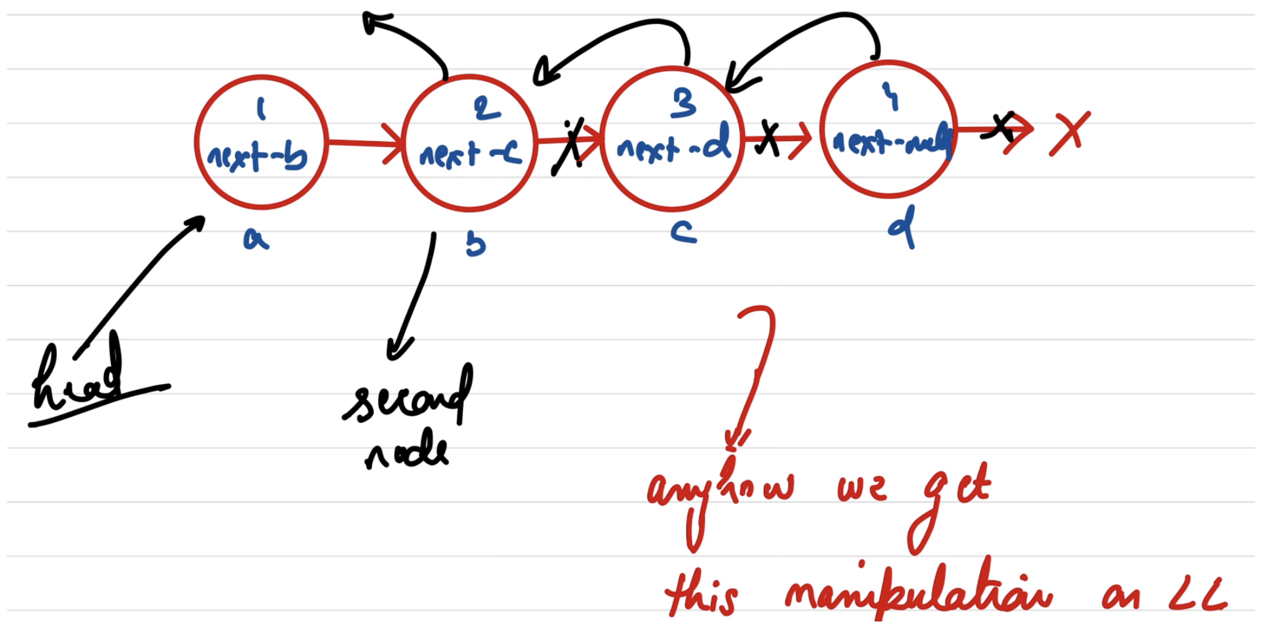


```
reverseLLPointerIterative() {
  /**
     * Time: O(n)
     * Space: O(1)
     */
  let prev = null;
  let curr = this.head;
  while(curr != null) {
    let temp = curr.next;
    curr.next = prev;
    prev = curr;
    curr = temp;
  }
  this.head = prev;
}
```

Pointer Reverse Recursive

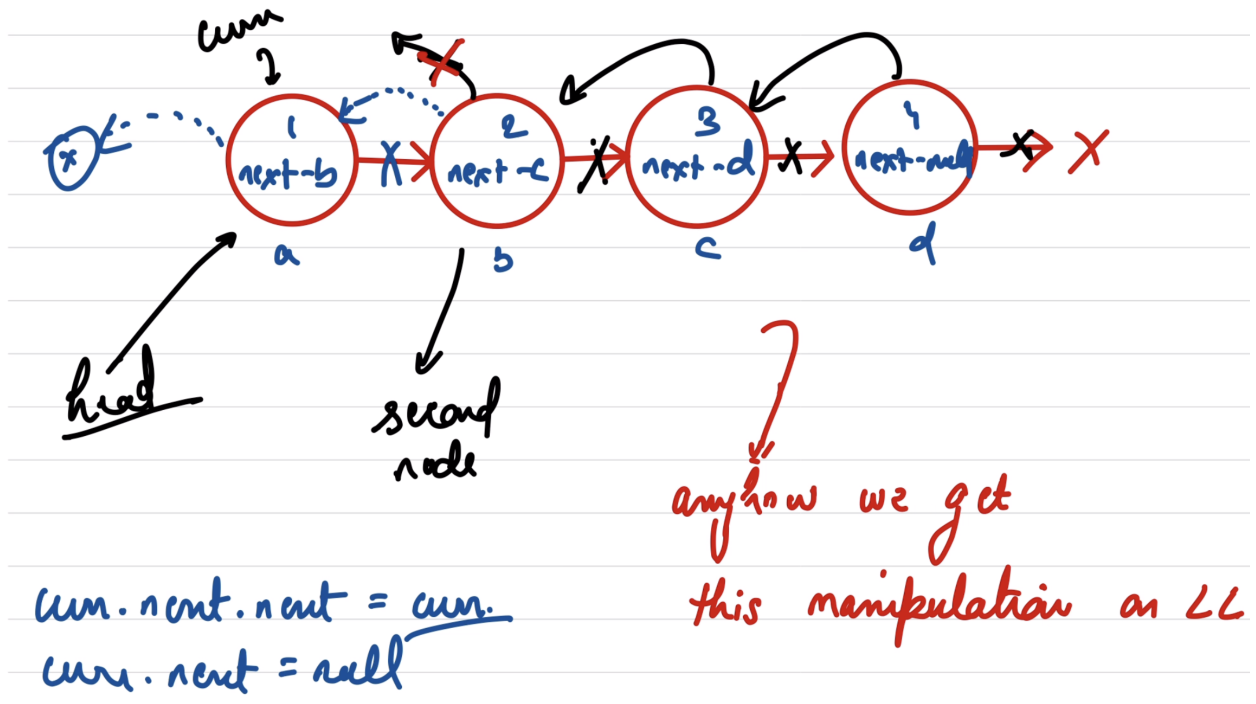
If we have to reverse the LL recursively, then we need to make some assumption. As in recursion, we just care about self work and leave the remaining task for recursion let's see what can be done here.

So we don't know how to recursively reverse the whole list but if someone can give a reversed list from the second node, then can we do something ?

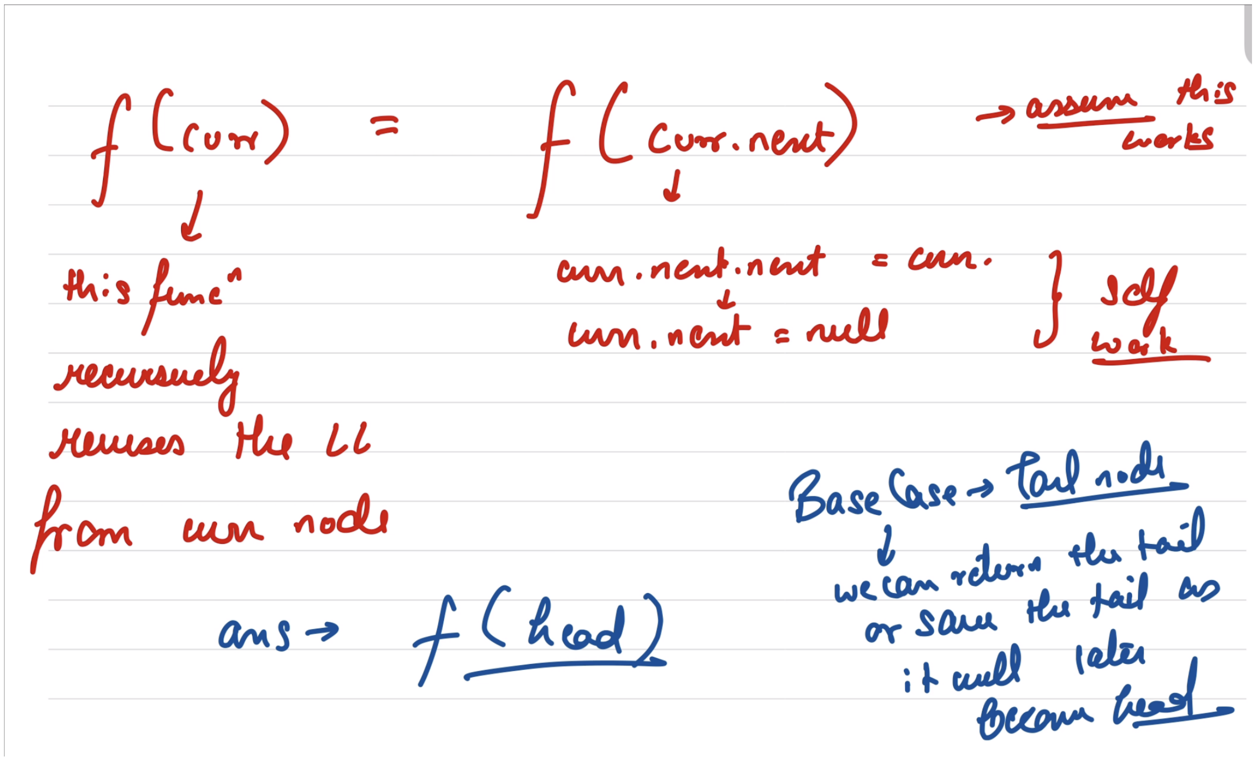


So if we have the LL from the second node already reversed, then we want our second node's next to point to first node i.e. if the first node is `curr` then `curr.next.next = curr`

`curr.next` is the second node, and `curr.next.next` is the next pointer of the second node, and we are making it point to the curr node.



For a base case, we can say that the operation `curr.next.next = curr` cannot be done on tail node so we need to stop recursion there. And this will be the base case and we should return this tail node also.


f(curr) = f(curr.next)
→ assume this works
this func^n recursively reverses the LL from curr node
curr.next.next = curr.
curr.next = null
self work
Base Case → Tail node
we can return the tail or save the tail as it will later become head
ans → f(head)

```
f(curr) {
  /**
     * Time: O(n)
     * Space: O(n) // due to call stack space, call stack will have n entries when we reac
h tail
     */
  if(curr.next == null) {
    // if node's next is null it is the tail node
    return curr;
  }
  let tail = this.f(curr.next); // recursively assume it reverses
  curr.next.next = curr; // self work
  curr.next = null; // self work
  return tail;
}
reverseLLPointersRecursive() {
  let tail = this.f(this.head);
  this.head = tail;
}
```

Value reverse iteratively

So in value reverse, we will not manipulate the pointers, instead we will try to just change the data values of a node.

So just think how we can reverse an array ? In an array for every index i, we swap it's value with `n-i-1` index.

So we will try to do the same for LL, for every node at index i, we will try to swap it with node at index `n-i-1`. But we can't do direct access of a node in a LL, so we need to put a loop and then reach there.

For this approach we need to first get the length as well.

```
reverseLLValueIterative() {
  /**
     * Time: O(n^2)
     * Space: O(1)
     */
  let len = 0;
    let temp = this.head;
    while(temp != null) {
      temp = temp.next;
      len++;
    }
    let curr = this.head;
    for(let i = 0; i < len/2; i++) {
      let temp = this.head;
      // console.log(curr.data);
      for(let j = 0; j < len - i - 1; j++) {
        temp = temp.next;
      }
      let v = curr.data;
      curr.data = temp.data;
      temp.data = v;
      curr = curr.next;
    }
}
```

In iterative version we cant optimise further but, we can optimise in recursive version.

Value reverse Recursive

So to swap values recursively in a LL, for every node we need to access it's opposite counter part. Whenever in LL we want to access an opposite counterpart like for index `i` we want `n-i-1` in recursion, we can maintain a global kind of variable to act as `start` and we can have a variable in recursive function that can act as `end`

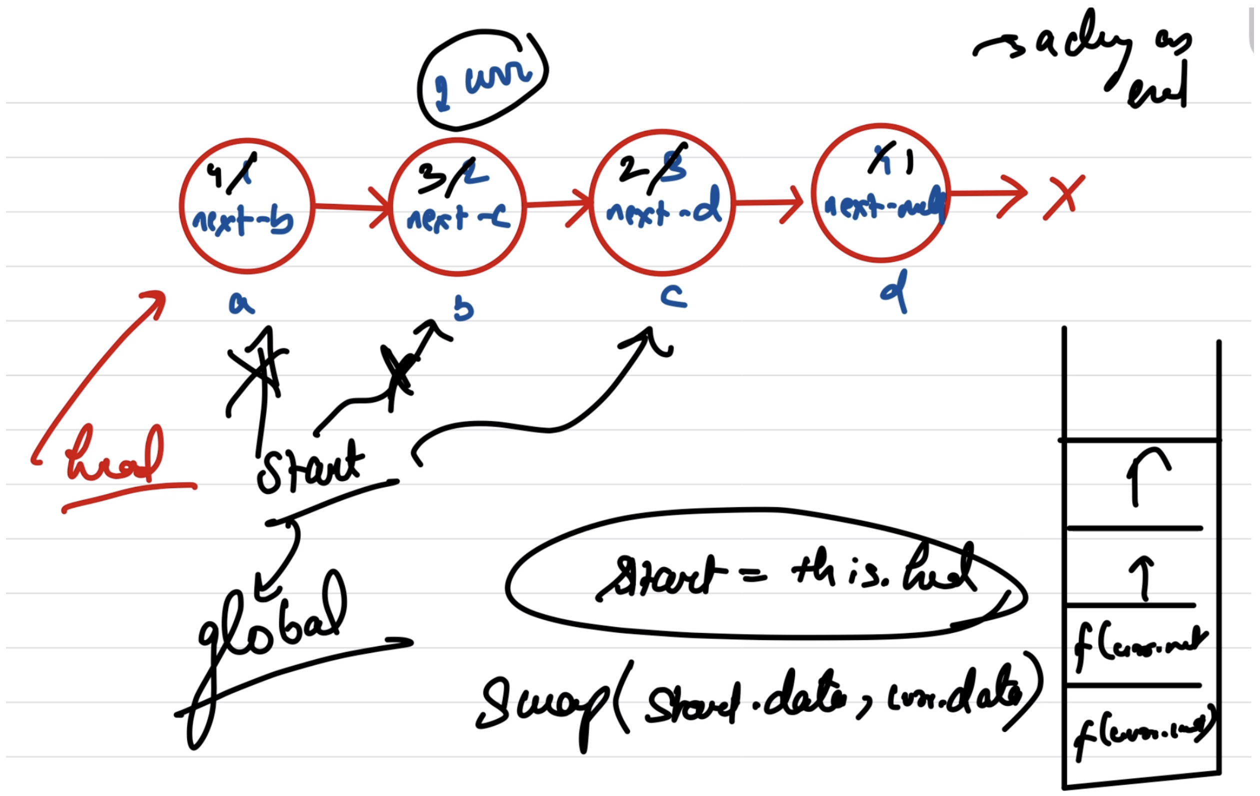
Then every time we can swap the values and before we return from recursion we move start to next node. Because returning from recursion will bring back the `end` variable to previous node as it moved forward recursively so when recursion comes back it also comes back.

```
constructor() { // changes constructor also for LL class
    // when we initialise a new linked list head will be empty
    this.head = null;
    this.start = null; // temporary variable for problem solving
    this.flag = null; // temporary variable for problem solving
}

g(curr) {
    /**
     * Time: O(n)
     * Space: O(n) // due to recursion
     */
    if(curr == null) {
      return;
    }
    this.g(curr.next);
    if(this.start == curr || curr.next == this.start) {
      this.flag = false;
    }
    if(this.flag == true) {
      let temp = this.start.data;
      this.start.data = curr.data;
      curr.data = temp;
      this.start = this.start.next;
    }
  }

  reverseLLValueRecursive() {
    this.start = this.head;
    this.flag = true;
    this.g(this.head);
  }
```

So because we are touching recursively every node once while going deep into recursion and then just come back and no for loops are there for function calls, the time will be O(n)



→ ready as end
curr
4/1
next-b
3/2
next-c
2/3
next-d
4/1
next-null
a
b
c
d
head
start
global
start = this.head
swap(start.data, curr.data)
f(curr.next)
f(curr.next)