



Leetcode → [LinkedList] 2

328. Odd Even linked list

```
//奇数分成一个list
//偶数分成一个list
//奇数.next连偶数head

func oddEvenList(_ head: ListNode?) -> ListNode? {

    var odd = head
    var even = head?.next
    let evenHead = even

    while even != nil && even?.next != nil {
        odd?.next = odd?.next?.next
        even?.next = even?.next?.next

        odd = odd?.next
        even = even?.next
    }

    odd?.next = evenHead

    return head
}
```

Time complex: $O(n)$

Space complex: $O(1)$

234. Palindrome linked list

```
//Method 1.
//step 1. iterate the original list.
//step 2. store them to an array.(array 1)
//step 3. reverse the array.(array 2)
//step 4. compare 2 arrays, if same, return true, else return false.
//Space:  $O(2n)$ , Time:  $O(3n)$ 

//Method 2.
//1. find 2 halves. ( 2 runner technique: ending node of the 1st half will be the starting node of the 2nd half.)
//slow指针走一步, fast指针走两步, 当fast.next = nil时, slow指针刚好在一半。
//2. reverse the 2nd half. (参考206题:how to reverse a linked list) .
//3. compare 2 halves. (p1放在head1, p2放在head2, 往后遍历, 当p2 = nil时, break)
//4. reverse the 2nd half back。
//5. return res.
//Time  $O(4n)$ , Space  $O(1)$ 

//pseudocode

//find the end of first half.
ListNode fast = head
ListNode slow = head

while fast.next != nil && fast.next.next != nil {
    fast = fast.next.next
    slow = slow.next
}
```

```

    }

    return slow (at the middle)

//reverse the 2nd half.
ListNode prev = nil
ListNode curr = head

while curr != nil {
    ListNode nextTemp = curr.next
    curr.next = prev
    prev = curr
    curr = nextTemp
}
return prev

//compare
firstHalfEnd = endOfFirstHalf(head)
secondHalfStart = reverseList(firstHalfEnd.next)

p1 = head
p2 = secondHalfStart
boolean res = true

while p2 != nil && res = true {
    if p1.val != p2.val{
        return false
    }else{
        p1 = p1.next
        p2 = p2.next
    }
}

firstHalfEnd.next = reverse(secondHalfStart)
return res
}

```

//Method 1.

```

func isPalindrome(_ head: ListNode?) -> Bool {
    var currentNode = head
    var array: [Int] = []
    while currentNode != nil {
        array.append(currentNode?.val ?? -1)
        currentNode = currentNode?.next
    }
    return array == array.reversed() ? true : false
}

```

//Method 2.

```

func isPalindrome(_ head: ListNode?) -> Bool {
    var slow = head
    var fast = head

    //find middle (slow)
    while fast != nil && fast?.next != nil {
        fast = fast?.next?.next
        slow = slow?.next
    }

    //reverse second half

    var prev: ListNode? = nil
    while slow != nil {
        var tmp = slow?.next
        slow?.next = prev
        prev = slow
        slow = tmp
    }

    //check palindrome

```

```

var left = head
var right = prev
while right != nil {
    if left?.val != right?.val {
        return false
    }
    left = left?.next
    right = right?.next
}
return true
}

-----

func isPalindrome(_ head: ListNode?) -> Bool {
    var slow = head
    var fast = head
    while fast?.next != nil {
        fast = fast?.next?.next
        slow = slow?.next
    }

    var secondHalf = slow?.reverse()
    var firstHalf = head

    while secondHalf != nil {
        if firstHalf?.val != secondHalf?.val {
            return false
        }
        firstHalf = firstHalf?.next
        secondHalf = secondHalf?.next
    }
    return true
}

}

extension ListNode {
    func reverse() -> ListNode? {
        if self == nil {
            return nil
        } else {
            var pre : ListNode? = nil
            var current: ListNode? = self
            while current != nil {
                let next = current?.next
                current?.next = pre
                pre = current
                current = next
            }
            return pre
        }
    }
    return nil
}

}

-----

func isPalindrome(_ head: ListNode?) -> Bool {
    var fastNode = head
    var slowNode = head

    while fastNode != nil {
        fastNode = fastNode?.next?.next
        slowNode = slowNode?.next
    }

    fastNode = head
    slowNode = reverseLinkedList(slowNode)

    while fastNode != nil, slowNode != nil {
        guard fastNode?.val == slowNode?.val else { return false }
        fastNode = fastNode?.next
        slowNode = slowNode?.next
    }
}

```

```

    }

    return true
}

private fun reverseLinkedList(_ node: ListNode?) -> ListNode? {
    var currentNode = node
    var previousNode: ListNode?

    while currentNode != nil {
        let nextNode = currentNode?.next
        currentNode?.next = previousNode
        previousNode = currentNode
        currentNode = nextNode
    }

    return previousNode
}

```

21. Merge two sorted lists.

```

//there are 2 methods for this question. 1, iterative; 2, recursive.

//1. Iterative: list 1设置head 1, list 2设置head 2->比较两个head的大小, 小的那个存储到res里
//当其中一个list走到null时, return两一个list当前的head。
// 需要4个指针: tail/res和head共同指向空node (存储), l1, l2。
//tail往后走: tail=tail.next
//最后return head.next
// Time complex: O(m+n)
// Space complex:O(1)
//iterative比recursive 的space complex 小。

func mergeTwoLists(_ list1: ListNode?, _ list2: ListNode?) -> ListNode? {

    var res:ListNode? = ListNode(0)
    let head = res

    var l1 = list1
    var l2 = list2

    while l1 != nil && l2 != nil{
        if l1!.val <= l2!.val {
            res?.next = l1
            l1 = l1?.next
        }else{
            res?.next = l2
            l2=l2?.next
        }
        res = res?.next
    }

    // 当一个list走完, 接剩下的nodes.
    if l1 != nil{
        res?.next = l1
    }else{
        res?.next = l2
    }

    return head?.next
}

```

```

}

-----

//2. recursive: 先用上述方法找到最小值, 设置成res/head (存储), 然后用recursive的方法 (还是两两一组,
//比较大小, 小值存储到res)
//consider BASE CASE!!!-> 当一个list走到nil, res.next连另一个list的当前head。
//if l1==nil, return l2; else if l2 == nil, return l1.
//返回res/head。
//Time complex:O(m+n)
//Space complex: O(m+n) ->因为占用了多余的stacks。

func mergeTwoLists(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    //Base case.
    if(l1 == nil) {return l2}
    if(l2 == nil) {return l1}

    var result:ListNode? = nil

    if(l1!.val <= l2!.val) {
        result = l1
        result!.next = mergeTwoLists(l1!.next, l2)
    }
    else {
        result = l2
        result!.next = mergeTwoLists(l1, l2!.next)
    }
    return result
}

```

2. Add Two Numbers

```

//花花酱 (python)
//dummy = tail - ListNode(0)
//while l1 or l2 or carry:
//sum = l1.val + l2.val + carry
//tail.next = ListNode(sum%10)(remian的那个位数)
//tail = tail.next
//carry = sum/= 10
//l1 = l1.next ? l1.val:0
//l2 = l2.next ? l2.val:0
//return dummy.next

//Time complex: O(max(n,m))
//Space complex: O(max(n,m))

func addTwoNumbers(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    var l1: ListNode? = l1
    var l2: ListNode? = l2

    var result: ListNode? = ListNode(0)
    var head = result

    var carry = 0

    while l1 != nil || l2 != nil || carry > 0 {

        let firstValue = l1?.val ?? 0
        let secondValue = l2?.val ?? 0

        let sum = firstValue + secondValue + carry

        let value = sum % 10
        carry = sum / 10
    }
}

```

```

        result?.next = ListNode(value)
        result = result?.next
        l1 = l1?.next
        l2 = l2?.next
    }
    return head?.next
}

```

430.Flatten a Multilevel doubly linked list

```

//Method:Preorder DFS by recursive.
//Time complex: O(n)
//Space complex:O(n).

func flatten(_ head: Node?) -> Node? {
    if head == nil { return nil}

    var curr = head

    while curr != nil {

        if curr?.child != nil {
            var next = curr?.next
            curr?.next = curr?.child
            curr?.next?.prev = curr
            curr?.child = nil

            var pointer = curr?.next
            while pointer?.next != nil {
                pointer = pointer?.next }

            pointer?.next = next
            next?.prev = pointer
        }

        curr = curr?.next
    }

    return head
}

```

708.Insert into a singly circular linked list.

```

//Method:two-pointers: prev , curr

//Termination Condition-> prev==head

// there are 3 scenarios we need to analyse:

// Scenario 1. min.val <InsertVal < max.val -> then insert val at prev.val <= insertVal <= curr.val

// Scenario 2. InsertVal < min.val || InsertVal > max.val -> locate the tail node(where prev.val > curr.val)
// then insert insertVal between head and tail nodes, which is exactly Scenario 1.

//Scenario 3. all nodes are uniform values-> iterate whole list first, to determine if

```

```

//it contains a single unique value. Then insert InsertVal right after the head node. (entrance point).

//Time complex:O(n)
//Space complex: O(1)
-----
func insert(_ head: Node?, _ insertVal: Int) -> Node? {

    //consider the edge condition.
    guard head != nil else {
        let newNode = Node(insertVal)
        newNode.next = newNode
        return newNode
    }

    var curr: Node? = head?.next
    var prev = head
    let newNode = Node(insertVal)

    var inserted = false

    while(prev!.next != head) {
        if
            // insert when:
            // 1. prev <= insertVal <= next
            // 2. insertVal > max (insert at the tail)
            // 3. insertVal < min (insert at the tail)

            (prev!.val <= insertVal && curr!.val >= insertVal) ||
            (insertVal > prev!.val && prev!.val > curr!.val) ||
            (insertVal < curr!.val && prev!.val > curr!.val)
        {
            prev!.next = newNode
            newNode.next = curr
            inserted = true
            break
        }

        curr = curr!.next
        prev = prev!.next
    }

    // All values were equal
    if(!inserted) {
        prev!.next = newNode
        newNode.next = curr
    }

    return head
}

```

138.copy list with random pointer.

方法1: hashmap。

Time complex: O(n)

Space complex: O(n)

```

func copyRandomList(_ head: Node?) -> Node? {
    //设置一个dummy head.
    var clonedHead = Node(0)
    //设置一个空map。
    var map = [Node? : Node?]() // [OldNode : NewNode]

    // (每一个数clone,连next pointer) -> 循环
    var current: Node? = head
    var clonedCurrent: Node? = clonedHead
    while current != nil {
        let clone = Node(current!.val)
        map[current] = clone
        current = current?.next
        clonedCurrent?.next = clone
        clonedCurrent = clonedCurrent?.next
    }

    //连random pointer.
    current = head
    clonedCurrent = clonedHead.next
    while current != nil {
        let random = map[current?.random, default: nil]
        clonedCurrent?.random = random
        current = current?.next
        clonedCurrent = clonedCurrent?.next
    }
    //返回dummy head.next
    return clonedHead.next
}

```

方法2: 3 pointers . Iterative

1, Traverse the original list and clone the nodes as you go and place the cloned copy next to its original node. 遍历第一次，复制原nodes，新node放在原node后面, 连上

```

cloned_node.next = original_node.next
original_node.next = cloned_node

```

2, 遍历第二次，连random pointer

3, 遍历第三次，重新调整next pointer, 让整体的list变回两个list：（原始list 和 复制list）

A.next = A'.next

A'.next = B.next

B.next = B'.next

B'.next = C.next

C.next = C'.next = nil

4. Time complex: O(n)

Space complex: O(1)

```

func copyRandomList(_ head: Node?) -> Node? {

    let dummy = Node(Int.min)
    var temp = dummy
    var old = head

    // Pass One: Create New Nodes
    while old != nil {
        let new = Node(old!.val)

        new.next = old?.next
        old?.next = new
        old = new.next
    }

    // Pass Two: Add Random Pointers
    old = head
    while old != nil {
        let new = old?.next
        new?.random = old?.random?.next
        old = new?.next
    }

    // Pass Three: Remove Old Nodes & Repair List
    old = head
    while old != nil {
        let new = old?.next
        temp.next = new
        temp = temp.next!
        old?.next = new?.next
        old = old?.next
    }

    return dummy.next
}

```

61. rotate list.

```

// step 1. 把这个linked list连成闭环。
//Find the old tail and connect it with the head old_tail.next = head to close the ring.
//Compute the length of the list n at the same time.
//step 2. 找到new head: n - k % n, 和new tail: n - k % n - 1
//step 3. 打破闭环: new_tail.next = nil, return new_head.
//Time complex: O(n)
//Space complex: O(1)

func rotateRight(_ head: ListNode?, _ k: Int) -> ListNode? {

    if head == nil || k == 0 {
        return head
    }

    var head = head
    var last = head

    var length = 1

    while last?.next != nil {
        last = last?.next
        length+=1
    }

    last?.next = head

```

```
var rotations = k%length
var skip = length-rotations
var newLast = head
var i = 0
while i<skip-1 {
    newLast = newLast?.next
    i+=1
}

head = newLast?.next
newLast?.next = nil
return head
}
```