Leetcode→[Linked List] 1

707. Design linked list.

```
//Singly list.

class MyLinkedList {

    var list = [Int]()

    init() {

    }

    func get(_ index: Int) -> Int {
        return list.count > index ? list[index] : -1
    }

    func addAtHead(_ val: Int) {
        list.insert(val, at: 0)
    }

    func addAtTail(_ val: Int) {
        list.append(val)
    }

    func addAtIndex(_ index: Int, _ val: Int) {
        guard list.count >= index else { return }
        list.insert(val, at: index)
    }

    func deleteAtIndex(_ index: Int) {
        guard list.count > index else { return }
        list.remove(at: index)
    }
}
-------------------------------------------------------------------------------
class Node {
    var item: Int
    var next: Node?
    init(_ item: Int) {
        self.item = item
    }
}


class MyLinkedList {
    private var head: Node?
    private var count = 0
    /** Initialize your data structure here. */
    init() {}

    /** Get the value of the index-th node in the linked list. If the index is invalid, return -1. */
    func get(_ index: Int) -> Int {
        if index >= count { return -1 }
        var node = head
        for _ in 0..<index {
            node = node?.next
        }
        return node!.item
    }

    /** Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first no
de of the linked list. */
    func addAtHead(_ val: Int) {
        var node = Node(val)
        node.next = head
        head = node
        count += 1
    }

    /** Append a node of value val to the last element of the linked list. */
    func addAtTail(_ val: Int) {
        if count == 0 {
            addAtHead(val)
            return
        }
        var current = head
        while current?.next != nil {
            current = current?.next
        }
        var node = Node(val)
        current?.next = node
        count += 1
    }

    /** Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. */
    func addAtIndex(_ index: Int, _ val: Int) {
        if index > count { return }
        if index == 0 {
            addAtHead(val)
            return
        }

        var current = head
        var prev: Node?
        for _ in 0..<index {
            prev = current
            current = current?.next
        }
        var node = Node(val)
        prev?.next = node
        node.next = current
        count += 1
    }

    /** Delete the index-th node in the linked list, if the index is valid. */
    func deleteAtIndex(_ index: Int) {
        if index >= count { return }
        if index == 0 {
            head = head?.next
            count -= 1
            return
        }

        var current = head
        var prev: Node?
        for _ in 0..<index {
            prev = current
            current = current?.next
        }
        prev?.next = current?.next
        count -= 1
    }
}
```

```
class MyLinkedList {

    var intArray: [Int] = []
    init() {

    }

    func get(_ index: Int) -> Int {
        if index >= intArray.count {
            return -1
        }
        return intArray[index]
    }

    func addAtHead(_ val: Int) {
        intArray.insert(val, at: 0)
    }

    func addAtTail(_ val: Int) {
        intArray.append(val)
    }

    func addAtIndex(_ index: Int, _ val: Int) {
        if index <= intArray.count {
            intArray.insert(val, at: index)
        }
    }

    func deleteAtIndex(_ index: Int) {
        if index < intArray.count {
            intArray.remove(at: index)
        }
    }
}
```

```
/**
 * Your MyLinkedList object will be instantiated and called as such:
 * let obj = MyLinkedList()
 * let ret_1: Int = obj.get(index)
 * obj.addAtHead(val)
 * obj.addAtTail(val)
 * obj.addAtIndex(index, val)
 * obj.deleteAtIndex(index)
 */

---------------------------------------------------------------------------
//Doubly list.

public class ListNode {
    public var val: Int
    public var next: ListNode?
    public var prev: ListNode?
    public init(_ val: Int) {
        self.val = val
    }
}

class MyLinkedList {

    var head:ListNode = ListNode(-1)
    var tail:ListNode = ListNode(-1)
    var count = 0
    init() {
        head.next = tail
        tail.prev = head
    }

    /** Get the value of the index-th node in the linked list. If the index is invalid, return -1. */
    func get(_ index: Int) -> Int {
        guard count > index && index >= 0 else {
            return -1
        }
        var node:ListNode? = head.next
        var i = index
        while i > 0 {
            node = node?.next
            i -= 1
        }
        return node!.val
    }

    /** Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node
    func addAtHead(_ val: Int) {
        let nNode = ListNode(val)
        nNode.next = head.next
        nNode.prev = head
        head.next?.prev = nNode
        head.next = nNode
        count += 1
    }

    /** Append a node of value val to the last element of the linked list. */
    func addAtTail(_ val: Int) {
        let nNode = ListNode(val)
        nNode.next = tail
        nNode.prev = tail.prev
        tail.prev?.next = nNode
        tail.prev = nNode
        count += 1
    }

    /** Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node wi
    func addAtIndex(_ index: Int, _ val: Int) {
        guard index <= count && index >= 0 else {
            return
        }
        var node:ListNode? = head
        var i = index
        while i > 0 {
            node = node?.next
            i -= 1
        }
        let nNode = ListNode(val)
        nNode.next = node!.next
        nNode.prev = node
        node!.next?.prev = nNode
        node!.next = nNode
        count += 1
    }

    /** Delete the index-th node in the linked list, if the index is valid. */
    func deleteAtIndex(_ index: Int) {
        guard count > index && index >= 0 else {
            return
        }
        var node:ListNode? = head.next
        var i = index
        while i > 0 {
            node = node?.next
            i -= 1
        }
        node?.prev?.next = node?.next
        node?.next?.prev = node?.prev
        count -= 1
    }
}
```

141.Linked list cycle.

```
Floyd's Cycle Finding Algorithm
//I got it.

func hasCycle(_ head: ListNode?) -> Bool {


     guard let head = head, let next = head.next else {
            return false
        }


        var slow: ListNode? = head
        var fast: ListNode? = next

        while slow !== fast {
            if slow == nil || fast == nil {
                return false
            }
            slow = slow?.next
            fast = fast?.next?.next
        }
        return true

    }
```

```
extension ListNode : Equatable{
    public static func ==(lhs : ListNode, rhs : ListNode)->Bool{
        return ObjectIdentifier(lhs) == ObjectIdentifier(rhs)
    }
}

class Solution {
    func hasCycle(_ head: ListNode?) -> Bool {

        guard let head = head, let _ = head.next else { return false }

        var s : ListNode? = head
        var f : ListNode? = head

        while s != nil && f != nil{
            f = f?.next?.next
            s = s?.next

            if s == f{
                return true
            }
        }

        return false
    }
}
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     public var val: Int
 *     public var next: ListNode?
 *     public init(_ val: Int) {
```

```
 *         self.val = val
 *         self.next = nil
 *     }
 * }
 */

class Solution {
    func hasCycle(_ head: ListNode?) -> Bool {

    var slow = head
    var fast = head?.next


        while fast != nil {

            slow = slow?.next
            fast = fast?.next?.next

             if slow === fast {
                return true
            }

        }
        return false


    }
}

//operator function '==' requires that 'ListNode' conform to 'Equatable' in solution.swift
```

Time complex: O(n)

Space complex: O(1)

142. Linked List Cycle 2

Floyd's Tortoise and Hare

Step 1: determine if there is a cycle.

Step 2: find the entrance point.

Floyd's algorithm is separated into two distinct *phases*. In the first phase, it determines whether a cycle is present in the list. If no cycle is present, it returns `null` immediately, as it is impossible to find the entrance to a nonexistant cycle. Otherwise, it uses the located "intersection node" to find the entrance to the cycle.

Phase 2

Given that phase 1 finds an intersection, phase 2 proceeds to find the node that is the entrance to the cycle. To do so, we initialize two more pointers: `ptr1`, which points to the head of the list, and `ptr2`, which points to the intersection. Then, we advance each of them by 1 until they meet; the node where they meet is the entrance to the cycle, so we return it.

```
func detectCycle(_ head: ListNode?) -> ListNode? {

//Phase 2

        guard let head = head else { return nil }
        var ptr1 = intersection(head)

        if ptr1 == nil { return nil }

        var ptr2: ListNode? = head

        while ptr1 !== ptr2 {
            ptr1 = ptr1?.next
            ptr2 = ptr2?.next
        }

        return ptr1
    }
```

```
//Phase 1
    func intersection(_ head: ListNode?) -> ListNode? {
        var fast = head
        var slow = head


        while (fast != nil && fast?.next != nil) {
            slow = slow?.next
            fast = fast?.next?.next

            if fast === slow {
                return fast
            }
        }

        return nil
    }
```

```
//无论时间复杂度还是空间复杂度都是最优解。
 * Definition for singly-linked list.
 * public class ListNode {
 *     public var val: Int
 *     public var next: ListNode?
 *     public init(_ val: Int) {
 *         self.val = val
 *         self.next = nil
 *     }
 * }
 */

class Solution {
    func detectCycle(_ head: ListNode?) -> ListNode? {
        var walker = head
        var runner = head

        while walker != nil {
            walker = walker?.next
            runner = runner?.next?.next

            if runner != nil && runner === walker {
                var ptr1 = head
                var ptr2 = walker

                while ptr1 !== ptr2 {

                    ptr1 = ptr1?.next
                    ptr2 = ptr2?.next
                }

                return ptr1
            }
        }

        return nil
    }
}
```

```
func detectCycle(_ head: ListNode?) -> ListNode? {

//此处，必须同时指向head，不然后面会无限循环。
        var slow = head
        var fast = head

        while fast != nil{
            slow = slow?.next
            fast = fast!.next?.next

            if slow === fast { break }
        }

        if fast == nil {
            return nil
        }

        slow = head
        while slow !== fast {
            slow = slow?.next
            fast = fast?.next
        }
        return slow
    }
```

Time complex: O(n)

Space complex: O(1).

160.intersaction of two linked list.

```
//Method 1. Hash Set.

//A=[1,2,3,4,5,6], length:N
//B=[7,8,9,5,6]  , length:M
// var hashset { 遍历B,加到这里}
//遍历A,遇到和hash set里一样的第一个数字，即result。


//设一个空hashset。
var hashset = headA:ListNode

//遍历B，并且都加到hashset里。
while headB != nil{
 for i in headB{
headB.insert(i)
headB = headB.next
}

//遍历A。
while headA != nil{
 for i in headA{
   if hashSet.contains(of: i){
return i
}
}
return nil
}


//Time complex: O(N+M)
//Space complex: O(M)


func getIntersectionNode(_ headA: ListNode?, _ headB: ListNode?) -> ListNode? {
        if headA == nil || headB == nil { return nil }

        var a = headA
        var b = headB

        var inA: Set<ListNode> = []

        while a != nil {
            inA.insert(a!)
            a = a?.next
        }

        while b != nil {
            if inA.contains(b!) {
                return b
            }

            b = b?.next
        }

        return nil
    }
}

extension ListNode: Equatable, Hashable {
    public static func == (lhs: ListNode, rhs: ListNode) -> Bool {
        return lhs === rhs
    }

    public func hash(into hasher: inout Hasher) {
        hasher.combine(val + (next?.val ?? 0))
    }

-------------------------------------------------------------------------------
```

```
//Method 2. 双指针.
//P1 P2一起走
//P1遍历A (length N), 到头后跳到B的head，继续往前遍历
//P2遍历B (length M), 到头后跳到A的head，继续往前遍历
//这样p1 p2跳到head时处在相同距离
//where P1 == P2 is the intersection point.
//Time complex: O(N+M)
//Space complex: O(1)


//1.
func getIntersectionNode(_ headA: ListNode?, _ headB: ListNode?) -> ListNode? {

     var tempA = headA
     var tempB = headB

        while tempA !== tempB {
            if tempA == nil {
                tempA = headB
            } else {
                tempA = tempA?.next
            }

            if tempB == nil {
                tempB = headA
            } else {
                tempB = tempB?.next
            }
        }

        return tempA //OR tempB
    }

//2. space complex is better.

func getIntersectionNode(_ headA: ListNode?, _ headB: ListNode?) -> ListNode? {

        if headA == nil || headB == nil {
            return nil
        }
        var p1 = headA, p2 = headB
        while p1 !== p2 {
            p1 = p1 == nil ? headB : p1?.next
            p2 = p2 == nil ? headA : p2?.next
        }
        return p1
    }






-------------------------------------------------------------------------------
//没看！
//Method 3. Traversing backwards both lists.

func getIntersectionNode(_ headA: ListNode?, _ headB: ListNode?) -> ListNode? {
   if headA == nil && headB == nil {return nil}
  if headA === headB {
    return headA
  }

  var arrayA = [headA]
  var arrayB = [headB]


  var nA = headA
  var nB = headB

  while let next = nA?.next {
    arrayA.append(next)
    nA = next

  }

  while let next = nB?.next {
    arrayB.append(next)
    nB = next
  }

  var minArray = arrayA.count > arrayB.count ? arrayB : arrayA
  var maxArray = arrayA.count > arrayB.count ? arrayA : arrayB
  minArray = minArray.reversed()
  maxArray = maxArray.reversed()

  guard minArray.first! === maxArray.first! else {
    return nil
  }

  for i in 1..<minArray.count {
    if minArray[i] !== maxArray[i]{
      return minArray[i - 1]
    }
  }

  return minArray.last!

    }
```

19. remove Nth node from end of list.

```
// Method 1. Two Loops. -》小小福, 以下为phthon pseudocode。
//1. 计算linked list的长度
//2. 根据l和n, 计算出即将要删除的node的位置, (即删除的node之前的node, i.e, prev。
//3. 进行删除操作。
//一般如果要删除node, 要在linked list最前面加一个dummy node。
//所以最后返回dummynode.next,而不是head。


// 举例： 删除倒数第二个: n=2->即：要删除的node是3.
// D->1->2->3->4->NULL
// D->1->2->4->Null

//identify the length
length = 0
node = head //此时的head是1.还没create dummy head。
while node{
    node = node.next
    length += 1
}

//因为要涉及到删除的操作了, 所以create dummy head
dummy = ListNode(0)
dummy.next = head //把dummy head接到1上

//identify the location to be removed.
node = dummy
for i in range(length-n){
    node= node.next
}

//perform removal.
prev = node
succ = node.next.next
prev.next = succ

//return result
return dummy.next //( 避免1也被删除掉了)


//Time complex: O(2n) = O(n) 第一遍计算linked list的长度, 第二遍去找要删除的位置。
//Space complex: O(1)
------------------------------------------------------------------------------
//Method 2. Single Loops. (better solution).

//1.创建2个指针, 举例为题目问的n
//2. 同时同方向向前遍历2个指针, 知道右边指针运动到末尾
//3. 要删除的就是左边指针的那个node。
// 注意：对于singlely linked list, 为了删掉一个node, 需要知道这个node之前node (i.e: prev) 的信息,
//所以我们让left pointer指向prev的位置 (即：要删除的node的parent) , 那么当right pointer.next == none
//的时候, left pointer就到了prev的位置了。
//Time complex: O(n)
//Space complex: O(1)
```

```
//create a dymmy head.
dymmy = ListNode(0)
dummy.next = head

//create 2 pointers
left = dummy
right = dummy
for i in range(n){
right = right.next
} //这样right ponter此时就指向了2.

//shift left pointer and right pointer.
while right.next != nil{
   left = left.next
   right = right.next
}

//delete prev.next
prev = next
succ = prev.next.next
prev.next = succ

//return result
return dummy.next


-------------------------------------------------------------------------
//Siwft Answer.

func removeNthFromEnd(_ head: ListNode?, _ n: Int) -> ListNode? {

    if head == nil { return nil }

    var validHead = head

    var fastPointer: ListNode? = head
    var slowPointer: ListNode? = head

    for i in 1...n {
      if fastPointer?.next == nil && i != n {
        return nil
      }
      fastPointer = fastPointer?.next
    }

    if fastPointer == nil {
      validHead = validHead?.next
    } else {
      fastPointer = fastPointer?.next
      while fastPointer != nil {
        fastPointer = fastPointer?.next
        slowPointer = slowPointer?.next
      }

      var temp = slowPointer?.next
      slowPointer?.next = temp?.next

      temp = nil
    }

    return validHead
    }
----------------------------------------------------------------------------
func removeNthFromEnd(_ head: ListNode?, _ n: Int) -> ListNode? {
        var count = 1
        var node = head
        while node?.next != nil {
            count+=1
            node = node?.next
        }
        let targetIndex = count-n-1
        if targetIndex < 0 {
            return head?.next
        }
        print(targetIndex)
        var leftNode: ListNode? = head
        for i in 0..<targetIndex {
            leftNode = leftNode?.next
        }
        print(leftNode?.val)
        leftNode?.next = leftNode?.next?.next
        return head
    }
----------------------------------------------------------------------------
//Good and Simple !

func removeNthFromEnd(_ head: ListNode?, _ n: Int) -> ListNode? {
        var dummyNode = ListNode(0, head)
        var rearNode: ListNode? = dummyNode
        var frontNode: ListNode? = dummyNode

        var count = 0

        while count < n {
            frontNode = frontNode?.next
            count += 1
        }

        while frontNode?.next != nil {
            rearNode = rearNode?.next
            frontNode = frontNode?.next
        }

        rearNode?.next = rearNode?.next?.next

        return dummyNode.next
    }
--------------------------------------------------------------------------------
//空间复杂度最小！

func removeNthFromEnd(_ head: ListNode?, _ n: Int) -> ListNode? {

        if head == nil {
            return nil
        }

        var p : ListNode? = nil
        var l = head
        var r = head

        for i in 1..<n {
            r = r?.next
        }

        while r?.next != nil {
            r = r!.next
            p = l
            l = l!.next
        }

        if l!.val == r!.val {
            r = nil
        }
        if p == nil {
            return l!.next
        }

        p?.next = l!.next
        l?.next = nil

        return head
    }
```

总结： Here we provide a template for you to solve the two-pointer problem in the linked list.

```
// Initialize slow & fast pointers
ListNode slow = head;
ListNode fast = head;
/**
 * Change this condition to fit specific problem.
 * Attention: remember to avoid null-pointer error
 **/
while (slow != null && fast != null && fast.next != null) {
    slow = slow.next;           // move slow pointer one step each time
    fast = fast.next.next;      // move fast pointer two steps each time
    if (slow == fast) {         // change this condition to fit specific problem
        return true;
    }
}
return false;   // change return value to fit specific problem
```

1. Always examine if the node is null before you call the next field.

Getting the next node of a null node will cause the null-pointer error. For example, before we run `fast = fast.next.next`, we need to examine both `fast` and `fast.next` is not null.

2. Carefully define the end conditions of your loop.

206. Reserve linked list. Recursive方法没看！

```
//Method 1. Iterative

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

        var current = head
        var reversedLinkedList : ListNode?
        var next : ListNode?

        while current != nil
        {
            next = current?.next
            current?.next = reversedLinkedList
            reversedLinkedList = current
            current = next
        }
        return reversedLinkedList

    }

//Time complex: O(n)
//Space complex: O(1)




//Space complex is the best.
func reverseList(_ head: ListNode?) -> ListNode? {
        var currentNode: ListNode?
        var node = head
        while let nextNode = node {
            let next = nextNode.next

            nextNode.next = currentNode
            currentNode = nextNode

            node = next
        }

        return currentNode
    }

//篮子王解法。明白了的！3 pointers!-> prev, head1, next.
func reverseList(_ head: ListNode?) -> ListNode? {
        var head1 = head
        var prev: ListNode? //(point to dummy head)
        while(head1 != nil) {
            let next = head1!.next
            head1!.next = prev
            prev = head1
            head1 = next
        }
        return prev
```

203. Remove linked list elements

```
func removeElements(_ head: ListNode?, _ val: Int) -> ListNode? {

var dummy = ListNode(0)
dummy.next = head

        var prev = dummy
        var cur = head

        while cur != nil {
            if cur?.val == val {
                prev.next = cur?.next
            } else {
                prev = cur!
            }

            cur = cur?.next
        }

        return dummy.next
}
```