# 三月份刷题测试题

## 面试题 02.02. 返回倒数第 k 个节点

采用双指针法，前指针先向前走k-1步，之后前后指针同步走，直到前指针的next为空，证明走到了表尾，这时候吧后指针指向的节点值返回即可。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {number}
 */
var kthToLast = function(head, k) {
    let hair = new ListNode(-1,head);
    while(--k){
        head = head.next;
    }
    while(head){
        head = head.next;
        hair = hair.next;
    }
    return hair.val;
};
```

## 剑指 Offer 22. 链表中倒数第k个节点

与上题相同，只不过返回值有所变化。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
```

```
 */
var getKthFromEnd = function(head, k) {
    let hair = new ListNode(-1,head);
    while(--k){
        head = head.next;
    }
    while(head){
        head = head.next;
        hair = hair.next;
    }
    return hair;
};
```

## 剑指 Offer 35. 复杂链表的复制

1. 创建两个指针，一个指向头指针
2. 遍历整个链表，复制每个节点，并将值复制成一样的，然后拼接到原节点的后面
3. 找到一个克隆节点，然后进行修正random,并将克隆节点的random指向克隆节点
4. 拆分链表，分成原节点链表和克隆节点链表
5. 返回我们的克隆节点链表

```
/**
 * // Definition for a Node.
 * function Node(val, next, random) {
 *    this.val = val;
 *    this.next = next;
 *    this.random = random;
 * };
 */

/**
 * @param {Node} head
 * @return {Node}
 */
var copyRandomList = function(head) {
    if (!head) return null;
    let p = head, q;
    while (p) {
        q = new ListNode(p.val);
        q.random = p.random;
        q.next = p.next;
        p.next = q;
        p = q.next;
    }
    p = head.next;
    while (p) {
        p.random && (p.random = p.random.next);
        (p = p.next) && (p = p.next);
    }
```

```
    p = q = head.next;
    while (q.next) {
        head.next = head.next.next;
        q.next = q.next.next;
        head = head.next;
        q = q.next;
    }
    head.next = null;
    return p;
};
```

## 面试题 02.03. 删除中间节点

单向链表是不能删除自身的，所以我们可以将待删除的节点的值改变为待删除节点的下一个节点的值，然后我们删除待删除节点的下一个节点。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} node
 * @return {void} Do not return anything, modify node in-place instead.
 */
var deleteNode = function(node) {
    node.val = node.next.val;
    node.next = node.next.next;
};
```

## 445. 两数相加 II

1. 用栈，依次压入值，再出栈进行相加操作；
2. 相加结果是从个位开始，生成的节点，依次追到到结果链表上即可

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} l1
 * @param {ListNode} l2
 * @return {ListNode}
 */
```

```
var addTwoNumbers = function(l1, l2) {
    let stack1 = [],stack2 = [];
    let hair = new ListNode(-1);
    while(l1){
        stack1.push(l1.val);
        l1 = l1.next;
    }
    while(l2){
        stack2.push(l2.val);
        l2 = l2.next;
    }
    let ten = 0;
    while(stack1.length|| stack2.length||ten){
        let num1 = stack1[stack1.length-1]===undefined?0:stack1.pop(),
        num2 = stack2[stack2.length-1]===undefined?0:stack2.pop();
        let val = num1+num2+ten;
        ten = val/10 |0;
        let temp = new ListNode(val%10,hair.next);
        hair.next = temp;
    }
    return hair.next;
};
```

## 143. 重排链表

1. 找到链表中点后分割其为 left 链表、right 链表两部分;
2. 翻转 right 链表;
3. 接着从 left 链表的左侧, 翻转后的 right 链表的左侧各取一个值进行交替拼接;

```
/**

 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {void} Do not return anything, modify head in-place instead.
 */
var reorderList = function (head) {
    let hair = new ListNode(-1, head);
    let left = hair, right = hair;
    while (right && right.next) {
        right = right.next;
        right = right.next;
        left = left.next;
    }
    right = left.next;
    left.next = null;
    left = head;
```

```
    right = reverse(right);
    while (left && right) {
        let lNext = left.next;
        let rNext = right.next;
        right.next = left.next;
        left.next = right;
        left = lNext;
        right = rNext;
    }
    return hair.next;
};
var reverse = function (head) {
    let temp = new ListNode(-1);
    while (head) {
        let next = head.next;
        head.next = temp.next;
        temp.next = head;
        head = next;
    }
    return temp.next;
}
```

## 面试题 02.08. 环路检测

我们使用两个指针，fast 与 slow。它们起始都位于链表的头部。随后，slow 指针每次向后移动一个位置，而 fast 指针向后移动两个位置。如果链表中存在环，则 fast 指针最终将再次与 slow 指针在环中相遇。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */

/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var detectCycle = function(head) {
    if (head === null) {
        return null;
    }
    let slow = head, fast = head;
    while (fast !== null) {
        slow = slow.next;
        if (fast.next !== null) {
            fast = fast.next.next;
        } else {
            return null;
        }
        if (fast === slow) {
```

```
            let ptr = head;
            while (ptr !== slow) {
                ptr = ptr.next;
                slow = slow.next;
            }
            return ptr;
        }
    }
    return null;
};
```

## 707. 设计链表

这道题目一共为5个接口，已经覆盖了链表的常见操作；

### 单向链表的实现：

```
/**
 * Initialize your data structure here.
 */
var MyLinkedList = function() {
  this.head=null
  this.rear=null
  this.len=0
};
function ListNode(val) {
  this.val = val;
  this.next = null;
}
/**
 * Get the value of the index-th node in the linked list. If the index is invalid, ret
urn -1.
 * @param {number} index
 * @return {number}
 */
MyLinkedList.prototype.get = function(index) {
  if(index<0||index>this.len-1)
    return -1
  var node=this.head
  while(index-->0){
    if(node.next==null)
      return -1
    node=node.next
  }
  return node.val
};

/**
 * Add a node of value val before the first element of the linked list. After the inse
rtion, the new node will be the first node of the linked list.
 * @param {number} val
 * @return {void}
 */
```

```
MyLinkedList.prototype.addAtHead = function(val) {
  var node=new ListNode(val)
  if(this.head==null)
    this.rear=node
  else
    node.next=this.head
  this.head=node
  this.len++
};

/**
 * Append a node of value val to the last element of the linked list.
 * @param {number} val
 * @return {void}
 */
MyLinkedList.prototype.addAtTail = function(val) {
  var node=new ListNode(val)
  if(this.head==null)
    this.head=node
  else
    this.rear.next=node
  this.rear=node
  this.len++
};

/**
 * Add a node of value val before the index-th node in the linked list. If index equal
s to the length of linked list, the node will be appended to the end of linked list. I
f index is greater than the length, the node will not be inserted.
 * @param {number} index
 * @param {number} val
 * @return {void}
 */
MyLinkedList.prototype.addAtIndex = function(index, val) {
  if(index<=0)
    return this.addAtHead(val)
  if(this.len<index)
    return
  if(index==this.len)
    return this.addAtTail(val)
  var node=this.head
  while(index-->1){
    node=node.next
  }

  var newnode=new ListNode(val)
  newnode.next=node.next
  node.next=newnode
  this.len++
};

/**
 * Delete the index-th node in the linked list, if the index is valid.
 * @param {number} index
 * @return {void}
 */
MyLinkedList.prototype.deleteAtIndex = function(index) {
  if(index<0||index>this.len-1||this.len==0)
```

```
    return
  if(index==0){
    this.head=this.head.next
    this.len--
    return
  }

  var node=this.head
  var myindex=index
  while(index-->1){
    node=node.next
  }
  if(myindex==(this.len-1)){
    this.rear=node
  }
  node.next=node.next.next
  this.len--
};

/**
 * Your MyLinkedList object will be instantiated and called as such:
 * var obj = new MyLinkedList()
 * var param_1 = obj.get(index)
 * obj.addAtHead(val)
 * obj.addAtTail(val)
 * obj.addAtIndex(index,val)
 * obj.deleteAtIndex(index)
 */
```

## 双向链表的实现：

```
var ListNode = function(val,pre,next){
    this.val = (val === undefined ? 0 : val)
    this.pre = (pre === undefined ? null : pre)
    this.next = (next === undefined ? null : next)
}
/**
 * Initialize your data structure here.
 */
var MyLinkedList = function() {
    this.head = new ListNode(-1);
    this.tail = new ListNode(-1);
    this.head.next = this.tail;
    this.tail.pre = this.head;
    this.length = 0;
};

MyLinkedList.prototype.getNodeOfIndex = function(index){
    let cur = this.head.next;
    while(index--){
        cur = cur.next;
    }
    return cur;
}
```

```
/**
 * Get the value of the index-th node in the linked list. If the index is invalid, ret
urn -1.
 * @param {number} index
 * @return {number}
 */
MyLinkedList.prototype.get = function(index) {

    if(index > this.length) return -1;
    return this.getNodeOfIndex(index).val;
};

/**
 * Add a node of value val before the first element of the linked list. After the inse
rtion, the new node will be the first node of the linked list.
 * @param {number} val
 * @return {void}
 */
MyLinkedList.prototype.addAtHead = function(val) {
   let temp = new ListNode(val,this.head,this.head.next);
   this.head.next.pre = temp;
   this.head.next = temp;
   this.length++;
};

/**
 * Append a node of value val to the last element of the linked list.
 * @param {number} val
 * @return {void}
 */
MyLinkedList.prototype.addAtTail = function(val) {
   let temp = new ListNode(val,this.tail.pre,this.tail);
   this.tail.pre.next = temp;
   this.tail.pre = temp;
   this.length++;
};

/**
 * Add a node of value val before the index-th node in the linked list. If index equal
s to the length of linked list, the node will be appended to the end of linked list. I
f index is greater than the length, the node will not be inserted.
 * @param {number} index
 * @param {number} val
 * @return {void}
 */
MyLinkedList.prototype.addAtIndex = function(index, val) {
    if(index>this.length) return;
    if(index<0){
        this.addAtHead(val);
    }else if(index==this.length){
        this.addAtTail(val);
    }else{
       let cur = this.getNodeOfIndex(index);
       let temp = new ListNode(val,cur.pre,cur);
       cur.pre.next = temp;
       cur.pre = temp;
    this.length++;
    }
```

```
};

/**
 * Delete the index-th node in the linked list, if the index is valid.
 * @param {number} index
 * @return {void}
 */
MyLinkedList.prototype.deleteAtIndex = function(index) {
    if(index >= 0 && index < this.length){
       this.length--;
       let cur = this.getNodeOfIndex(index);
       cur.pre.next = cur.next;
       cur.next.pre = cur.pre;
    }
};
/**
 * Your MyLinkedList object will be instantiated and called as such:
 * var obj = new MyLinkedList()
 * var param_1 = obj.get(index)
 * obj.addAtHead(val)
 * obj.addAtTail(val)
 * obj.addAtIndex(index,val)
 * obj.deleteAtIndex(index)
 */
```

## 剑指 Offer 18. 删除链表的节点

1.首先遍历整个链表，然后当head.val == val

2.修改next指针指向 让 head.val 指向 val.next

3.最后，考虑 特殊情况 当是删除链表头时

```
 /**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} val
 * @return {ListNode}
 */
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
```

```
 * @param {number} val
 * @return {ListNode}
 */
var deleteNode = function(head, val) {
    // 特殊情况 当是删除链表头时
    if (head.val === val) return head.next

    let prev = head, node = prev.next

    while (node) {
        if (node.val === val) {
            prev.next = node.next
        }
        // 遍历 不断迭代 prev 与 node
        prev = node
        node = node.next
    }

    return head
};
```

## 725. 分隔链表

1.遍历链表，将链表的总长度len求出来

2.求出每一项的长度，Math.floor,L % k = 商,商就是总共分几段

3.余数给每段循环+1个

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} root
 * @param {number} k
 * @return {ListNode[]}
 */
var splitListToParts = function (root, k) {
  let len = 0;
  let node = root;
  while (node) {
    len++;
    node = node.next;
  }
//  每一项的长度，取商
  let itemLen = Math.floor(len / k);
  let curry = len % k; // 余数，即前 curry 项多一个元素
  let m = 0; // curry 计数
  let result = [];
  let dummyHead = new ListNode(0);
```

```
  dummyHead.next = root;
  for (let i = 0; i < k; i++) {
    node = dummyHead;
    let j = 0;
    while (j < itemLen) {
      node = node ? node.next : null;
      j++;
    }
    if (m < curry) {
      node = node ? node.next : null;
      m++;
    }

    result.push(dummyHead.next || null);
    let next = node.next || null;
    if (node) node.next = null;
    dummyHead.next = next;
  }
  return result;
};
```

## 面试题 02.04. 分割链表

1.首先，声明两个链表：small（存放所有小于 x 的节点）和 large（存放大于或等于 x 的节点）

2.接着，遍历完原链表，比较每一个节点和X 的值，分别存放到对应的small链表和large链表中

3.最后 small 链表尾节点指向 large 链表的头节点便是完成对链表的分割。

```
/**
 * De\finition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} x
 * @return {ListNode}
 */
var partition = function(head, x) {
    let small = new ListNode(0);
    const smallHead = small;
    let large = new ListNode(0);
    const largeHead = large;
    while(head !== null){
        if(head.val < x){
            small.next = head;
            small = small.next;
        }else{
```

```
            large.next = head;
            large = large.next;
        }
        head = head.next;
    }
    large.next = null;
    small.next = largeHead.next;
    return smallHead.next;

};
```

## 779. 第K个语法符号

1.按着给出的例子，自己走一遍，便能看出规律：

2.如果k是个偶数，那么就是上一行的k/2个；

3.如果k是个奇数，那么就是上一行的k+1/2

```
/**
 * @param {number} N
 * @param {number} K
 * @return {number}
 */
var kthGrammar = function(N, K) {
if (N === 1) return 0;
  if (K % 2 === 0) {
    return kthGrammar(N - 1, K / 2) === 0 ? 1 : 0;
  } else {
    return kthGrammar(N - 1, Math.floor(K / 2) + 1);
  }
};
```

## 剑指 Offer 10- I. 斐波那契数列

时间复杂度: 直接求解 O(n) / 使用快速幂求解 O(lg n)
空间复杂度: O(1)
根据线性代数的知识我们知道线性方程组具有以下两种完全相同的表示形式.

第一种, 使用方程组表示:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = y_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = y_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = y_3 \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = y_n \end{cases}$$

第二种, 使用矩阵乘法进行表示:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{vmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_n \end{vmatrix} = \begin{vmatrix} y_1 \\ y_2 \\ y_3 \\ \cdots \\ y_n \end{vmatrix}$$

对于本题我们有$F_{n+1} = F_n + F_{n-1}$

写成矩阵，我们有：

$$\begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} \cdot \begin{vmatrix} F_n \\ F_{n-1} \end{vmatrix} = \begin{vmatrix} 1 \cdot F_n + 1 \cdot F_{n-1} \\ 1 \cdot F_n + 0 \cdot F_{n-1} \end{vmatrix} = \begin{vmatrix} F_{n+1} \\ F_n \end{vmatrix}$$

此式子递归展开：

$$\begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-1} \cdot \begin{vmatrix} F_2 \\ F_1 \end{vmatrix} = \begin{vmatrix} F_{n+1} \\ F_n \end{vmatrix}$$

为了保证精度我们使用了BigInt

```
/**
 * @param {number} n
 * @return {number}
```

```
 */
var fib = function (n) {
   let initialState = [1n,0n,0n,0n];
    let transformation = [1n,1n,1n,0n];
    let buffer =[0n,0n,0n,0n];
    matrixPower(transformation,n,buffer);
    let finalState = [0n,0n,0n,0n];
 matrixMultiply(buffer, initialState, finalState);

     return finalState[2];
};

var matrixPower = function(base,exponent,result){
      let temp = [1n,0n,0n,1n];
       matrixCopy(result,temp);
       let currentBase = [0n,0n,0n,0n];
       matrixCopy(currentBase,base);
      let buffer = [0n,0n,0n,0n];
    while(exponent){
        if(exponent%2){
            matrixMultiply(currentBase,result,buffer);
            matrixCopy(result,buffer);
        }
        matrixMultiply(base,base,currentBase);
        matrixCopy(base,currentBase);
        exponent = exponent / 2 | 0;
    }
}

var matrixCopy = function(destination,source){
    for(let i= 0;i<4;i++){
        destination[i] = source[i];
    }
}
var matrixMultiply = function (a, b, c) {
    let mod = 1000000007n;
    c[0] = (a[0] * b[0] + a[1] * b[2]) % mod;
    c[1] = (a[0] * b[1] + a[1] * b[3]) % mod;
    c[2] = (a[2] * b[0] + a[3] * b[2]) % mod;
    c[3] = (a[2] * b[1] + a[3] * b[3]) % mod;
}
```