门徒计划——链表

1.链表的基础知识

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

抽象概念：链表代表了一种唯一指向思想

链表适合用于存储一些经常增加、删除的数据

2.链表的访问

环形链表

思路：快慢指针

1.首先，快指针每次向前移动两步，慢指针每次向前移动一步，进行遍历整个列表。

2.接着，当快指针的next节点为null，或者快指针本身节点为null时，说明该链表没有环，遍历结束。

3.思考：我们再来看一下有环的情况是怎样的?

4.我们定义两个指针，一个慢指针（用红色标记），一个快指针（用黄色标记），并且，一开始，慢指针和快指针同时指向head节点。然后，快指针每次向前移动两步，慢指针每次向前移动一步，开始遍历链表。如果链表有环，那么快慢指针一定会相遇，指向同一个节点，当指向同一个节点时，遍历结束。

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

```
 *     this.next = null;
 * }
 */

/**
 * @param {ListNode} head
 * @return {boolean}
 */
var hasCycle = function (head) {
    if (!head) return false;
    let pre = head, cur = head;
    while (cur && cur.next) {
        pre = pre.next;
        cur = cur.next.next;
        if (pre === cur) {
            return true;
        }
    }
    return false;
};
```

环形链表II

思路总结：使用快慢指针，

1.我们假设链表头到入环口的距离为a，从入环口到相遇点的距离为b，从相遇点到入环口的距离为c。

2.慢指针走过a+b的距离，快指针走过a+n(b+c)+b的距离。由于快指针是慢指针的二倍，所以:2(a+b)= a+n(b+c)+b,而我们实际上并不用关心n是多少，有可能是10，也有可能是1,因此上述公式可以简化为：a=c;

3.因此当快慢指针相遇后，重新定义一个新指针从a的起始位置向后移动，慢指针继续向后移动。

4.新指针从a的起始位置向后移动，慢指针继续向后移动。

5.当新指针和慢指针相遇时，就是入环点

```
/**
 * 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) return null;
    let pre = head, cur = head;
    while (cur && cur.next) {
        pre = pre.next;
        cur = cur.next.next;
        if (pre === cur) {
            let temp = head;
            while (pre !== temp) {
                pre = pre.next;
                temp = temp.next;
            }
            return pre;
        }
    }
    return null;
};
```

快乐数

思路：

1.我们举个例子：当输入值为19时，平方和就转换成了：19——>82——>68 ——>100——>1;

2.题目就可以转化为，判断一个链表是否有环。如果遍历某个节点为1，说明没环，就是快乐数。如果遍历到重复的节点值，说明有环，就不是快乐数。

3.再举个例子，当输入值为116时，平方和构成的链表就是：
116——>38——>73——>58——>89——>145——>42——>20——>4——>16——>37

```
/**
 * @param {number} n
 * @return {boolean}
 */
var isHappy = function (n) {
    let pre = n, cur = getNext(n);
    while (cur !== pre && cur !== 1) {
        pre = getNext(pre);
        cur = getNext(getNext(cur));
    }
    return cur === 1;
};
var getNext = function (n) {
    let t = 0;
    while (n) {
        t += (n % 10) * (n % 10);
        n = Math.floor(n / 10);
    }
    return t;
}
```

3.链表的反转

反转链表

思路：

1.定义指针——pre，pre指向空，定义指针——cur,cur指向我们的头节点。
定义指针——next，next指向cur所指向节点的下一个节点。这样我们的指针就初始化完毕了。

2.首先，我们将cur指针所指向的节点指向pre指针所指向的节点。然后移动指针pre到指针cur所在的位置，移动cur到next所在的位置

3.然后移动指针pre到指针cur所在的位置，移动cur到next所在的位置。
此时，我们已经反转了第一个节点。

4.将我们的next指针指向cur指针所指向节点的下一个节点。然后重复上述操作

5.当cur指针指向null的时候，我们就完成了整个链表的反转

```
/**
 * 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 {ListNode}
 */
var reverseList = function (head) {
    if (!head) return null;
    let pre = null, cur = head;
    while (cur) {
        [cur.next, pre, cur] = [pre, cur, cur.next];
        // let next = cur.next;
        // cur.next = pre;
        // pre = cur;
        // cur = next;
    }
    return pre;
};
```

反转链表II

思路：

1.首先我们定义一个虚拟头节点，起名叫做hair，将它指向我们的真实头节点。

2.定义一个指针pre指向虚拟头节点。

3.定义一个指针cur指向pre指针所指向节点的下一个节点。

4.让我们的pre指针和cur指针同时向后移动，直到我们找到了第m个节点

5.定义指针con和tail，con指向pre所指向的节点，tail指向cur指针所指向的节点。

6.con所指向的节点，将是我们将部分链表反转后，部分链表头节点的前驱节点。tail则是部分链表反转后的尾节点。

7.开始我们的链表反转，首先定义一个指针third指向cur所指向的节点的下一个节点，然后，将cur所指向的节点指向pre所指向的节点，将pre指针移动到cur指针所在的位置。将cur指针移动到third指针所在的位置，直到我们的pre指针指向第n个节点

8.重复上述步骤

9.此时pre指针指向了第m个节点并且将第m到第n个节点之间反转完毕。

10.我们将con指针所指向的节点指向pre指针所指向的节点。

11.将tail指针所指的节点指向cur指针所指的节点，整理一下，显示出最终的链表。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} left
 * @param {number} right
 * @return {ListNode}
 */
var reverseBetween = function (head, left, right) {
    if (!head) return null;
    let ret = new ListNode(-1, head), pre = ret, cnt = right - left
+ 1;
    while (--left) {
        pre = pre.next;
    }
    pre.next = reverse(pre.next,cnt);
    return ret.next;
};
var reverse = function (head, n) {
    let pre = null, cur = head;
    while (n--) {
        [cur.next, pre, cur] = [pre, cur, cur.next];
    }
    head.next = cur;
    return pre;
}
```

K个一组翻转链表

思路：

1.首先我们创建一个虚拟头节点hair，并将虚拟头节点指向链表的头节点。

2.定义指针pre指向虚拟头节点,定义指针tail指向pre所指的节点。

3.移动tail指针，找到第K个节点

4.反转从head节点到tail节点之间的链表，我们可以参照前面的反转链表方法，将反转链表操作抽取出来成为一个方法命名为reverse

5.我们向reverse方法中传入head节点以及tail指针所指向的节点
6.定义一个指针prev指向tail指针所指节点的下一个节点，定义指针P指向head节点，定义指针next指向P指针所指向节点的下一个节点

7.将指针P所指的节点指向指针prev所指的节点。

8.将pre指针挪动到P指针所指针的节点上。

9.将P指针挪动到next指针所指的节点上。

10.next指针则继续指向P指针所指节点的下一个节点

11.重复上述步骤

12.当指针prev与指针tail指向同一节点的时候，我们的K个一组的链表反转完成了，然后将这部分链表返回

13.让pre指针所指的节点指向tail指针所指的节点

14.pre指针移动到head指针所在的位置，head指针移动到pre指针所指节点的下一个节点

15.tail指针再次指向pre指针所指的节点

16.然后tail节点再移动K步，如果tail节点为空，证明后面的节点不足K个，返回链表。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
```

```
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var reverseKGroup = function (head, k) {
    if (!head) return null;
    let ret = new ListNode(-1, head), pre = ret;
    do {
        pre.next = reverse(pre.next, k);
        for (let i = 0; i < k && pre; i++) {
            pre = pre.next;
        }
        if (!pre) break;
    } while (1);
    return ret.next;
};

var reverse = function (head, n) {
    let pre = head, cur = head, con = n;
    while (--n&&pre) {
        pre = pre.next;
    }
    if (!pre) return head;
    pre = null;
    while (con--) {
        [cur.next, pre, cur] = [pre, cur, cur.next];
    }
    head.next = cur;
    return pre;
}
```

旋转链表

思路：

1.
举例，有个链表是 6——>2——>7——>4——>9,我们命名一个指针F，它指向的是链表的Head,K=2是让 F 指针往右移动两位，指向 7 这个节点

2.接下来，进行这道题的第一步，我们通过遍历，得到链表的长度length和链表的尾节点，然后命名E指针，指向尾节点

3.然后让尾结点指向链表的Head，这样就成了环

4.然后我们要计算新链表Head的前一位，通过链表length – K-1 ，在这里面就是5-2-1=2；

5.我们先命名T指针指向新链表Head的前一位，然后去找对应的节点；首先 T指针走了0步

6.然后T指针走了1步

7.接着T指针走了两步，到了新链表Head 的前一位

8.接着，通过刚得到的新链表Head的前一位 得到它的Head，命名H指针指向它

9.最后，我们断开T到H的指向，就会得到最终效果;

10.接下来思考：当K = 11时，旋转后的效果会是怎样的呢?

11.当K = 1 旋转后的效果,9——>6——>2——>7——>4;当K = 2 旋转后的效果,4——>9——>6——>2——>7;当K = 3 旋转后的效果,7——>4——>9——>6——>2;当K = 4 旋转后的效果,2——>7——>4——>9——>6;当K = 5 旋转后的效果,6——>2——>7——>4——>9;当K = 6 旋转后的效果,9——>6——>2——>7——>4;当K = 7 旋转后的效果,4——>9——>6——>2——>7;当K = 8 旋转后的效果,7——>4——>9——>6——>2;当K = 9 旋转后的效果,2——>7——>4——>9——>6;当K = 10 旋转后的效果,6——>2——>7——>4——>9;当K = 11 旋转后的效果,9——>6——>2——>7——>4;

12.这里，我们来思考一下，从旋转前到旋转后，如果K是11，我们真的要一步一步的旋转吗？这里面是否有规律可循呢?

13.观察，K = 5 和 K = 10的效果是一样的;都是6——>2——>7——>4——>9;

14.接着观察，K = 11 和 K = 6 还有 K= 1 时的效果都是一样的；都是9——>6——>2——>7——>4;

15.总结：当 K 比链表长度数值要大时，K 要对链表长度取余。因为当 K 是旋转链表长度的整数倍时，它和未旋转是一样的。

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
```

```
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} k
 * @return {ListNode}
 */
var rotateRight = function (head, k) {
    if (!head) return null;
    let pre = head, size = 1;
    while (pre.next) pre = pre.next, size++;
    pre.next = head;
    for (let i = 0; i < size - k % size - 1; i++) {
        head = head.next;
    }
    pre = head.next;
    head.next = null;
    return pre;

};
```

两两交换链表的节点

思路：

1.创建一个虚拟空节点T，指向链表里的head;

2.再分别命名三个指针，指向原链表的head，head.next 和head.next.next;

3.接下来进行两两交换；

4.首先 P指向N,接着 C指向P，然后 T指向C; 这样第一步的交换就完成了；

5.接着 T 移到到 P，然后 P 移动到 N，然后 C 移动到 P 后一位，然后 N 要放在 C 的下一个，移动完毕；

6.按着上种移动指针的顺序无限循环；

7.接着第二轮，交换 P指针和C指针的值，然后， T指针 到 P的位置；

8.接着 P指针 到 N的位置，然后，N往后移一位；

9.这里思考一下，如果，没有 9 这个节点呢？我们需要注意什么呢？

10.此时要判断 p == null || p.next==null；

```
/**
 * 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 {ListNode}
 */
var swapPairs = function (head) {
    let ret = new ListNode(-1, head), temp = ret;
    while (temp.next && temp.next.next) {
        let pre = temp.next, cur = temp.next.next;
        pre.next = cur.next;
        cur.next = pre;
        temp.next = cur;
        temp = pre;
    }
    return ret.next;
};
```

4.链表的删除

删除链表的倒数第N个结点

思路：

1.先假设我的N 是2 ，那么我们就是删除倒数第2个元素，但是链表是没有索引的，就不能按着数组的方法删除；

2.想要删除倒数第N个元素，就必须找到当前待删除的前一个元素；

3.那么如何删除倒数第2个元素呢？

4.首先，先在当前待删除元素的前一个元素命名为T指针，然后呢，在当前链表的尾节点他只指向的是NUll空指针，我们命名为这个空指针是P

5.当我们P指针指向的是NUll的时候，我们的T指针必然指向的是当前待删除的前一个元素；

6.所以T指针和P指针两者的关系和距离显而易见；T和P两者差着题中给定的N个节点，差着N+1个步；

7.接着我们将链表的头部插入一个虚拟头节点；然后将虚拟头节点指向我们链表的头节点；

8.然后让T指针在虚拟头节点的位置上，根据第6点得到T和P之间的距离，把P 放到正确位置

9.当P指向NULL时候，当前待删除节点的前一个节点就找到了

10.最后，让当前待删除的前一个节点 指向 待删除的下一个节点；也就是断开了当前待删除的节点的前一位到当前待删除的节点的指向；

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} n
 * @return {ListNode}
 */
var removeNthFromEnd = function (head, n) {
    if (!head) return null;
    let ret = new ListNode(-1, head), pre = ret, cur = head;
    while (n--) cur = cur.next;
    if (!cur) return head.next;
    while (cur) pre = pre.next, cur = cur.next;
    pre.next = pre.next.next;
    return ret.next;
};
```

删除排序链表中重复的元素

思路：

1.首先声明一个虚拟头节点，然后让这个虚拟头节点指向链表的head；

2.接着使用双指针，慢指针是T,快指针是P,P比T多走一位；

3.然后设置一个while循环，如果T.val = P.next.val，那么就是出现了重复的元素，就需要去重；

4.删除的方法就是让 T 的下一个指针指向下一个的下一个；

5.如果不相等则 T 移动到下一个位置继续循环；

6.当 T 和 P.next 的存在为循环结束条件，当二者有一个不存在时说明链表没有去重复的必要了

```
/**
 * 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 {ListNode}
 */
var deleteDuplicates = function(head) {
    if (!head) return null;
    let cur = head;
    while (cur && cur.next) {
        if (cur.val == cur.next.val) {
            cur.next = cur.next.next;
        }else {
            cur = cur.next;
        }
    }
    return head;
};
```

删除排序链表中重复的元素II

思路：

1.首先我们要新建一个节点指向head，为的是创建一条头指针为空的链；

2.因为删除重复元素时有可能需要删除第一个；

3.接着我们遍历链表来查找重复的数；

4.命名prev为前一个指针，cur为当前指针；

5.如果cur 和 cur.next的val相同，那么cur继续向前移，prev不动；

6.等到不相同了，再将prev.next指向cur.next，这样就一次性跳过重复的数；

7.当然有时候还没遇到相同数，而我们又需要更新prev；

8.所以多加一个判断条件，来判断是要更新prev还是删除相同数；

```
/**
 * 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 {ListNode}
 */
var deleteDuplicates = function (head) {
    if (!head) return null;
    let ret = new ListNode(-1, head), pre = ret, cur = head;
    while (cur&&cur.next) {
        if (cur.next.val != pre.next.val) {
            cur = cur.next;
            pre = pre.next
        } else {
            while (cur && cur.next && pre.next.val == cur.next.val)
{
                cur = cur.next;
            }
```

```
            pre.next = cur.next;
            cur = cur.next;
        }
    }
    return ret.next;
};
```