第一章 链表

文章目录

链表

链表基础

```
public class ListNode {
    int val;       // 存储当前结点数据域
    ListNode next; // 存储下一个结点指针域
    ListNode(int x) { val = x; }
}
```

leetcode

例1a：链表逆序（206）

题目描述
反转一个单链表。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

算法思路
定义指针p指向头结点，指针q指向尾结点，从p开始逐一遍历，采用头插法插入到q之后
程序代码

```
 public ListNode reverseList(ListNode head) {
// 定义指针p指向头结点，指针q指向尾结点，从p开始逐一遍历，采用头插法插入到q之后
            ListNode p = head;  // 指针p指向头结点
            ListNode q = head;
            ListNode h = head;
            Integer length = 0;
            if(head == null) return head;
            while(q.next != null) {q = q.next;length++;}    // 指针q指向原链表尾结点

            while(length-->0) {
                    // p 从头依此遍历到尾结点，采用头插法依此插入到q之后
                    h = p.next;
                    p.next = q.next;
                    q.next = p;
                    p = h;
            }
            return q;
        }
```

例1b：链表逆序2（92）

题目描述
反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。
说明:
1 ≤ m ≤ n ≤ 链表长度。
示例:

```
输入: 1->2->3->4->5->NULL, m = 2, n = 4
输出: 1->4->3->2->5->NULL
```

算法思路
将m到n之间所有元素的指针逆转，并将m位置前一结点的指针指向n结点，将n结点的下一结点指向m结点

程序代码

```
public ListNode reverseBetween(ListNode head, int m, int n) {
                // 思路：将m到n之间所有元素的指针逆转，并将m位置前一结点的指针指向n结点，将n结点的下一结点指向m结点
            ListNode start = head;  // start指针指向位置m结点
            ListNode start_before = head;       // start_before指针指向位置m结点的前一个位置
            ListNode p = head;  // p指针为操作指针
            ListNode before = head; // before指针为索引指针，指向p的上一个指针
            ListNode after = head;  // after指针尾索引指针，指向p的下一个指针
                int idx = 1;        //idx 为指针所在位置
            if(head == null) return head;

            while(idx<m) {before = p;p = p.next;idx++;}
            // 指向起始位置m
            start_before = before;
            start = p;
            after = p.next;
            while(idx<n) {
                    // 指针由位置m遍历到n，将m到n结点的指针逆转
                    before = p;
                    p = after;
                    after = p.next;

                    p.next = before;
                    idx++;
            }
            // 指向结束位置n
            // 将m位置前一结点的指针指向n结点，将n结点的下一结点指向m结点
            start.next = after;
            // 单独处理头结点
            if(m==1)head = p;    // 若m位置为起始结点，则头结点为逆转链表的尾指针
            else start_before.next = p;// 若m位置不为起始结点，则头结点为仍为原结点

            return head;
        }
```

例2：链表求交点（160）

题目描述

编写一个程序，找到两个单链表相交的起始节点。

算法思路

遍历A链表，对A链表的每一个结点，遍历B链表，看是否相等

程序代码

```
// 160. 相交链表
        // 编写一个程序，找到两个单链表相交的起始节点。
        public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
            ListNode pa = headA;        //  pa为A链表的索引指针
            ListNode pb = headB;        //  pb为B链表的索引指针
            if(headA == null || headB == null)return null;
            // 遍历A链表，对A链表的每一个结点，遍历B链表，看是否相等
            while(pa!=null) {
                    pb = headB;
                    while(pb!=null) {
                        if(pa == pb)return pb;
                        pb = pb.next;
                    }
                    pa = pa.next;
            }
            return null;
        }
```

例3：链表求环（141）

题目描述

给定一个链表，判断链表中是否有环。

算法思路

1. 遍历链表，将链表中结点对应的指针（地址），插入set
2. 在遍历时插入节点前，需要在set中查找，第一个在set中发现的结点地址，就是链表环的起点

程序代码

```
        // 141.环形链表
        // 给定一个链表，判断链表中是否有环。
        // 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。
        // 如果 pos 是 -1，则在该链表中没有环。
        public boolean hasCycle(ListNode head) {
                // 1.遍历链表，将链表中结点对应的指针（地址），插入set
                // 2.在遍历时插入节点前，需要在set中查找，第一个在set中发现的结点地址，就是链表环的起点
                Set nodeList = new HashSet<>();
                ListNode p = head;
                if(head == null)return false;
                while(p!=null) {
                    // set求环起始节点，该节点时遍历时，第一个在set中已经出现的结点，即环的开始
                    if(nodeList.contains(p))return true;
                    nodeList.add(p);
                    p = p.next;
                }
            return false;
        }
```

例4：链表划分（86）

题目描述

给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

算法思路

1. 定义两个链表，分别存储小于x的节点与大于等于x的结点。
2. 将这两个链表合并 。

程序代码

```
        // 86.分隔链表
        // 给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。
        // 你应当保留两个分区中每个节点的初始相对位置。
        public ListNode partition(ListNode head, int x) {
                if(head == null)return null;
                if(head.next == null)return head;
                ListNode smallerHead = null; // 较小链表的链表头
                ListNode smallerP = null; //    较小链表的索引指针
                ListNode biggerHead = null; //  较大链表的链表头
                ListNode biggerP = null; // 较大链表的索引指针

                ListNode p = head;
                while(p != null) {
                    if(p.val < x) { // 若该结点的值<x，则通过尾插法插入较小链表
                        if(smallerHead == null) {
                            smallerHead = p;
                            smallerP = p;
                        }else {
                            smallerP.next = p;
                            smallerP = p;
                        }
                    }else {         // 若该结点的值>x，则通过尾插法插入较大链表
                        if(biggerHead == null) {
                            biggerHead = p;
                            biggerP = p;
                        }else {
                            biggerP.next = p;
                            biggerP = p;
                        }
                    }
                    p = p.next;
                }
                // 将较小结点链表与较大结点链表连接
                if(smallerHead == null)return biggerHead;   // 只有较大链表
                else if(biggerHead == null) return smallerHead; //  只有较小链表
                smallerP.next = biggerHead;
                biggerP.next = null;
                return smallerHead;
            }
```

例5：复杂链表的复制（138）

题目描述

给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。

要求返回这个链表的深拷贝。

算法思路

1. 从 head 节点开始遍历链表。下图中，我们首先创造新的 head 拷贝节点。并将新建结点加入字典中。
2. 如果当前节点 i 的 random 指针指向一个节点 j 且节点 j 已经被拷贝过，我们将直接使用已访问字典中该节点的引用而不会新建节点。
   如果当前节点 i 的 random 指针指向的节点 j 还没有被拷贝过，我们就对 j 节点创建对应的新节点，并把它放入已访问节点字典中。
3. 如果当前节点 i 的 next 指针指向一个节点 j 且节点 j 已经被拷贝过，我们将直接使用已访问字典中该节点的引用而不会新建节点。
   如果当前节点 i 的 next 指针指向的节点 j 还没有被拷贝过，我们就对 j 节点创建对应的新节点，并把它放入已访问节点字典中。

程序代码

```
        // 138. 复制带随机指针的链表
        // 给定一个链表，每个节点包含一个额外增加的随机指针，该指针可以指向链表中的任何节点或空节点。
        // 要求返回这个链表的深拷贝。
        public Node copyRandomList(Node head) {
            // 返回深度拷贝后的链表
                // 深度拷贝：构造生成一个完全新的链表，即使将原链表毁坏，新链表可独立使用
                // 算法步骤：
                // 1. 从 head 节点开始遍历链表。下图中，我们首先创造新的 head 拷贝节点。并将新建结点加入字典中。
                // 2. 如果当前节点 i 的 random 指针指向一个节点 j 且节点 j 已经被拷贝过，我们将直接使用已访问字典中该节点的引用而不会新建节点。
                // 如果当前节点 i 的 random 指针指向的节点 j 还没有被拷贝过，我们就对 j 节点创建对应的新节点，并把它放入已访问节点字典中。
                // 3. 如果当前节点 i 的 next 指针指向一个节点 j 且节点 j 已经被拷贝过，我们将直接使用已访问字典中该节点的引用而不会新建节点。
                // 如果当前节点 i 的 next 指针指向的节点 j 还没有被拷贝过，我们就对 j 节点创建对应的新节点，并把它放入已访问节点字典中。
                if(head == null)return head;

                Node copy_head = new Node(head.val,null,null);  // 拷贝结点的头结点
                Map<Node,Node> node_map = new HashMap<Node,Node>(); // 结点字典，存储已拷贝<原链表结点,拷贝链表结点>键值对
                node_map.put(head, copy_head);  //将头结点插入字典

                Node old_p = head;      // p为原链表索引结点
                Node copy_p = copy_head;        // copy_p为拷贝链表的索引指针
                while(old_p != null) {
                    copy_p.random = copyCloneNode(old_p.random,node_map);       // 拷贝结点的random引用
                    copy_p.next = copyCloneNode(old_p.next,node_map);       // 拷贝结点的next引用

                    old_p = old_p.next;
                    copy_p = copy_p.next;
                }

                return copy_head;   // 拷贝链表头结点
            }

            public Node copyCloneNode(Node oldNode,Map<Node,Node> nodeMap) {
                // 对原结点进行深度拷贝
                if(oldNode == null )return null;
```

```
            else if(nodeMap.containsKey(oldNode)) {
                // 该结点已存在字典中
                return nodeMap.get(oldNode);
            }else {
                // 该结点不存在，构造新结点并加入字典中
                Node copyNode = new Node(oldNode.val,null,null);
                nodeMap.put(oldNode, copyNode);
                return copyNode;
            }
        }
```

例6：2个排序链表归并（21）

题目描述

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

算法思路

比较l1和l2指向结点，将较小的结点插入p指针后

程序代码

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        // 比较l1和l2指向结点，将较小的结点插入p指针后
            ListNode p1 = l1;   // 链表1的索引指针
            ListNode p2 = l2;   // 链表2的索引指针
            ListNode head = new ListNode(0);    // 新链表头指针（头结点）,真正结点从head->next开始
            ListNode p = head;  // 新链表的索引指针
            while(p1 != null && p2 != null) {
                // 比较p1与p2的大小，将较小的结点通过尾插法插入合并链表
                if(p1.val<p2.val) {
                    p.next = p1;
                    p1 = p1.next;
                }else {
                    p.next = p2;
                    p2 = p2.next;
                }
                p = p.next;
            }
            // 若有剩余的结点，依此插入合并链表
            if(p1!=null)p.next = p1;
            if(p2!=null)p.next = p2;

            return head.next;
    }
```

例7：K个排序链表归并（23）

题目描述

合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

算法思路

对k个链表进行分制，两两进行合并。

设有k个链表，平均每个链表有n个节点，时间复杂度：

第1轮：进行k/2次，每次处理2n个数字；第2轮：进行k/4次，每次处理4n个数字；...；

最后一次，进行k/(2^{logk})次，每次处理2^{logk}logk*N个值。*

*因此时间复杂度为2N*k/2 + 4N*k/4 + 8N*k/8 + ... + N*k/(2^logk)*

=NK + NK + ... + NK = O(kNlogk)

程序代码

```
    // 23.合并K个排序链表
    // 合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。
    public ListNode mergeKLists(ListNode[] lists) {
        // 对k个链表进行分制，两两进行合并。
            // 设有k个链表，平均每个链表有n个节点，时间复杂度：
            // 第1轮：进行k/2次，每次处理2n个数字；第2轮：进行k/4次，每次处理4n个数字；...；
            // 最后一次，进行k/(2^logk)次，每次处理2^logk*N个值。
            // 因此时间复杂度为2N*k/2 + 4N*k/4 + 8N*k/8 + ... + N*k/(2^logk)
            // =NK + NK + ... + NK = O(kNlogk)
            if(lists.length == 0) return null;  // 若lists为空，返回null
            if(lists.length == 1) return lists[0];  // 若只有一个链表，则直接返回该链表
            if(lists.length == 2)
                return mergeTwoLists(lists[0],lists[1]);    //若只有两个list，则直接调用mergeTwoLists

            // 拆分lists为两个子lists
            int mid = lists.length/2 + 1;
            int i = 0;
            ListNode[] sub_lists1 = new ListNode[mid];
            ListNode[] sub_lists2 = new ListNode[mid];

            for(i=0;i<mid;i++)
                sub_lists1[i] = lists[i];
            for(i=mid;i<lists.length;i++)
                sub_lists2[i-mid] = lists[i];

            ListNode l1 = mergeKLists(sub_lists1);
            ListNode l2 = mergeKLists(sub_lists2);
            // 分治处理
            return mergeTwoLists(l1,l2);
        }
```

剑指offer

例1：从尾到头打印链表（3）

题目描述

输入一个链表，按链表从尾到头的顺序返回一个ArrayList。

算法思路

1. 将链表内容逐一加入栈中

2. 栈中元素一一弹栈，加入数组中，实现逆序打印

程序代码

```
    // 3. 从头到尾打印链表
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        // 将链表内容逐一加入栈中
        // 栈中元素一一弹栈，加入数组中，实现逆序打印
        ArrayList<Integer> result = new ArrayList<Integer>();
        Stack<Integer> stack = new Stack<Integer>();
        if(listNode == null) return result;
        while(listNode != null) {
                stack.push(listNode.val);
                listNode = listNode.next;
        }
        while(!stack.isEmpty()) result.add(stack.pop());
        return result;
    }
```

例2：链表中倒数第k个节点（14）

题目描述

输入一个链表，输出该链表中倒数第k个结点。

程序代码

```
    // 14. 链表中倒数第k个节点
    // 输入一个链表，输出该链表中倒数第k个结点。
    public ListNode FindKthToTail(ListNode head,int k) {
            // 第一次遍历获得链表中节点数目
            // 第二次遍历获取链表中倒数k个节点
            if(head == null)return null;

            ListNode p = head;  // 链表指针
            int num = 0;            // 链表节点数
            while(p!=null) {
                num++;
                p = p.next;
            }
            int idx = num - k;
            if(idx <0)return null;  // 若k大于链表长度
            p = head;
            while(idx-- > 0)p = p.next;

            return p;
    }
```

例3：反转链表（15）

题目描述

输入一个链表，反转链表后，输出新链表的表头。

程序代码

```
// 15.反转链表
  // 输入一个链表，反转链表后，输出新链表的表头。
  public ListNode ReverseList(ListNode head) {
          // 1. 新建一个新链表
          // 2. 按照头插法将原链表节点逐一插入新链表
          ListNode new_head = new ListNode(0);    // 建立一个空节点做头结点（避免单独处理头结点）
          if(head == null) return null;
          ListNode p = head;  // 操作指针
          ListNode p_next = head; // 前置指针
          while(p!=null) {
              p_next = p.next;
              p.next = new_head.next;
              new_head.next = p;
              p = p_next;
          }
          return new_head.next;   // 新建一个空节点做头结点，新链表是从第二个节点开始
  }
```

例4： 合并两个排序链表（16）

题目描述

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

程序代码

```
// 16. 合并两个排序链表
 // 输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。
 public ListNode Merge(ListNode list1,ListNode list2) {
     // 1. 指针p1指向链表list1，指针p2指向链表list2
         // 2. 比较p1,p2指针指向数大小，较小则插入新合并后的链表
         // 3. 若p,q指向节点仍有剩余，则采用尾插法插入新合并后的链表

         ListNode merge_list = new ListNode(0);  // 合并后的链表
         ListNode merge_p = merge_list;          // 合并链表的操作指针
         ListNode p1 = list1;                    // list1 的操作指针
         ListNode p1_next = list1;               // list1 的索引指针
         ListNode p2 = list2;                        // list2 的操作指针
         ListNode p2_next = list2;               // list2的索引指针
         while(p1 != null && p2 != null) {
             if(p1.val <= p2.val) {              // 若p1较小，将p1插入合并链表链尾
                 p1_next = p1.next;
                 merge_p.next = p1;
                 merge_p = merge_p.next;
                 p1 = p1_next;
             }else {                             // p2 较小，将p2插入合并链表链尾
                 p2_next = p2.next;
                 merge_p.next = p2;
```

```
                merge_p = merge_p.next;
                p2 = p2_next;
            }
        }
        if(p1 != null)merge_p.next = p1;
        if(p2 != null)merge_p.next = p2;

        return merge_list.next;
    }
```

例5： 复杂链表的复制（25）

题目描述

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

程序代码

```
    // 25. 复杂链表的复制
    // 输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），
    // 返回结果为复制后复杂链表的head。
    //（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）
    Map<RandomListNode,RandomListNode> nodeDic = new HashMap<RandomListNode,RandomListNode>();  // 节点引用字典，存储键值对<原链表引用，新链表引用>
    public RandomListNode Clone(RandomListNode pHead)
    {
        // 定义一个HashMap存储原节点与复制节点的引用。解决链表有重复值时可能找不到锁引用节点。
        // 遍历原链表
        // 若节点在字典中存在，则直接引用
        // 若节点在字典中不存在，则加入复制链表，并存入地址字典
        if(pHead == null)return null;
        RandomListNode cHead = new RandomListNode(1);   // 头结点空节点（不用单独处理头结点）
        RandomListNode ptr = pHead;
        RandomListNode ctr = cHead;
        while(ptr!=null) {
            ctr.next = cloneNode(ptr.next);
            ctr.random = cloneNode(ptr.random);

            ptr = ptr.next;
            ctr = ctr.next;
        }
        return cHead;
    }

    public RandomListNode cloneNode(RandomListNode originNode) {
        // 复制节点
        if(originNode == null) return null;
        if(!nodeDic.containsKey(originNode)) {
            RandomListNode cloneNode = new RandomListNode(originNode.label);
        nodeDic.put(originNode, cloneNode);
        return cloneNode;
        }else {
        return nodeDic.get(originNode);
        }
    }
```

例6： 两个链表的第一个公共节点（35）

题目描述

输入两个链表，找出它们的第一个公共结点。

程序代码

```
    // 35.两个链表的第一个公共节点
    // 输入两个链表，找出它们的第一个公共结点。
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        // 定义一个HashSet，存储第一个链表pHead1中各个节点的引用
        // 再遍历pHead2，若pHead2指向节点在HashSet中存在则直接返回
        // 否则不存在，返回null
        ListNode p1 = pHead1;
        ListNode p2 = pHead2;
        HashSet<ListNode> nodeSet = new HashSet<ListNode>();
        if(pHead1 != null || pHead2 != null) {
            while(p1 != null) {
                nodeSet.add(p1);
                p1 = p1.next;
            }
            while(p2 != null) {
                if(nodeSet.contains(p2))
                    return p2;
                p2 = p2.next;
            }
        }
        return null;
    }
```

例7： 孩子们的游戏（圆圈中最后剩下的数）（45）

题目描述

每年六一儿童节,牛客都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF作为牛客的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的:首先,让小朋友们围成一个大圈。然后,他随机指定一个数m,让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,从他的下一个小朋友开始,继续0...m-1报数...这样下去...直到剩下最后一个小朋友,可以不用表演,并且拿到牛客名贵的“名侦探柯南”典藏版(名额有限哦!!-)。请你试着想下,哪个小朋友会得到这份礼品呢？(注：小朋友的编号是从0到n-1)

如果没有小朋友，请返回-1

程序代码

```
    public int LastRemaining_Solution(int n, int m) {
        // 1. 用首尾连接的循环链表连接小朋友，遍历链表，并计数
            // 2. 计数到m-1时重置计数器，并移除该节点
            // 3. 如果没有小朋友，返回-1
            if(n<1 || m<1)return -1;
            // 构造循环链表
```

```
            ListNode head = new ListNode(0);    // 头结点
            ListNode ptr = head;        // 指针节点
            for(int i=1;i<n;i++) {
                ListNode node = new ListNode(i);
                ptr.next = node;
                ptr = node;
            }
            ptr.next = head;
            // 遍历循环链表
            ptr = head;
            ListNode pre = head;    // 记录当前节点的上一个节点，用于删除链表
            while(ptr.next!=ptr) {
                for(int i=0;i<m-1;i++) {
                    pre = ptr;
                    ptr = ptr.next;
                }
                pre.next = ptr.next;
                ptr = pre.next;
            }
            return ptr.val;
    }
```

例8：链表中环入口节点（54）

题目描述

给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。

程序代码

```
    // 54.链表中环的入口节点
    //给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。
    public ListNode EntryNodeOfLoop(ListNode pHead)
    {
        HashSet<ListNode> visited = new HashSet<ListNode>();
        ListNode p = pHead; // 遍历指针
        while(p != null) {
                if(visited.contains(p))return p;
                visited.add(p);
                p = p.next;
        }
        return null;
    }
```

例9：删除链表中重复节点（55）

题目描述

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

程序代码

```
    // 55.删除链表中重复的节点
    // 在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。
    // 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5
    public ListNode deleteDuplication(ListNode pHead)
    {
            if(pHead == null) return null;
            ListNode p = pHead;     // 操作指针
            ListNode pre = pHead;   // 前置指针
            while(p!=null && p.next!=null) {
                if(p.val == p.next.val) {
                    if(p == pHead) {    // 重复头结点单独处理
                        while(p.next!=null && p.val == p.next.val)  p = p.next;// p最终指向重复节点的最后节点
                        pHead = p.next;
                        p = pHead;
                    }else {
                        while(p.next!=null && p.val == p.next.val)  p = p.next;
                        pre.next = p.next;
                        p = pre.next;
                    }
                }else {
                    pre = p;
                    p = p.next;
                }
            }

            return pHead;
    }
```