Siyao's Blog

【九章算法基础班】课程笔记——链表

2017-12-11 | 算法， 九章算法 | 浏览 次
5,288 | 26

考点重要程度：链表 -> DFS/BFS ->DP

基础

test:

```
//print() 打印完整链表
ListNode node1 = new ListNode(1);
ListNode node2 = new ListNode(2);
ListNode node3 = new ListNode(3);

ListNode head = node1;
node1.next = node2;
node2.next = node3;

print(head);
//1->2->3
node1 = node2;
print(head);
//1->2->3

ListNode包括一个值和一个指针，head占4Byte(32bit)空间，head实际上是一个指针，通过head所指向的地址去找对应节点存储的值和下一个指针。

链表结构：
    [1,] ->  [2,] -> [3,]
      ↑        ↑       ↑
   head,n1    n2     n3
   4byte     4byte   4byte

node1和node2都是指向节点的指针，如果令node1 = node2,那么只是node1存储的地址和node2存储的地址一样了，但是链表的机构没有改变，所以输出依然是:
1->2->3

如果要改变链表的结构，需要node.next = balabala
```

相关习题

Remove Duplicates from Sorted List

题目

> Given a sorted linked list, delete all duplicates such that each element appear only *once*.
>
> For example,
>
> Given 1->1->2 , return 1->2 .
>
> Given 1->1->2->3->3 , return 1->2->3 .

分析

删掉链表中有重复的节点，保留一个

因为删除某个节点node，需要让node的前序节点.next = node.next，因此需要构造一个dummy node，让其指向前序节点，这样需要删除head的时候就可以令dummy.next = node.next。初始化时令dummy.next=head

最后返回dummy.next

代码

```
public ListNode deleteDuplicates(ListNode head) {
  ListNode prev;//用于记录重复元素第一次出现的位置
  ListNode curt = head;//用于向后遍历链表

  while(curt != null){
    //遇到重复元素
    if(curt.next != null && curt.val == curt.next.val){
      prev = curt;//记录第一个出现的元素
      int val = curt.val;//存储当前节点的值，用于后续判断是否和当前值相等
      //curt向后移动，直到和curt值不相等停止
      while (curt != null && curt.val == val){
        curt = curt.next;
      }
      //curt == null || curt.val != val;此时curt指向后面第一个和它值不相等的元素
      //将prev.next指向第一个不相等的元素
      prev.next = curt;
    }
    //如果没有遇到重复元素，curt继续后移一位
    else {
      curt = curt.next;
    }
  }
  return head;
}
```

Remove Duplicates from Sorted List II

题目

> Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only *distinct* numbers from the original list.
>
> For example,
>
> Given 1->2->3->3->4->4->5 , return 1->2->5 .
>
> Given 1->1->1->2->3 , return 2->3 .

分析

删除链表中重复出现的节点，全部删掉一个都不保留

因为删除某个节点node，需要让node的前序节点.next = node.next，删除全部重复的元素可能删掉head元素，因此需要构造一个dummy node，让其指向head的前序节点，也就是dummy.next = head。这样需要删除head的时候就可以令dummy.next = head.next。

最后反回dummy.next

代码

```
public ListNode deleteDuplicates(ListNode head) {
  ListNode dummy = new ListNode(0);//虚拟节点用于指向head
  dummy.next = head;

  ListNode prev = dummy;
  ListNode curt = head;

  while(curt != null){
    //遇到重复元素
    if(curt.next != null && curt.val == curt.next.val){
      int val = curt.val;//存储当前节点的值，用于后续判断是否和当前值相等
      //curt向后移动，直到和curt值不相等停止
      while (curt != null && curt.val == val){
        curt = curt.next;
      }
      //curt == null || curt.val != val;此时curt指向后面第一个和它值不相等的元素
      //将prev.next指向第一个不相等的元素
      prev.next = curt;
    }
    //如果没有遇到重复元素，prev和curt都后移一位
    else {
      prev = prev.next;
      curt = curt.next;
      }
    }
    return dummy.next;
  }
```

Reverse Linked List

题目

> Reverse a singly linked list.、
>
> Hint:
>
> A linked list can be reversed either iteratively or recursively. Could you implement both?

链表反转

分析

```
 null   [1,] -> [2,] -> [3,] -> [4,] -> [,5,]
  ↑       ↑
 prev   curt

1. 用temp记录下curt.next（因为后面要修改curt.next）
 null   [1,] -> [2,] -> [3,] -> [4,] -> [,5,]
  ↑       ↑      ↑
 prev   curt    temp

2. 将curt.next指向其前序节点prev，此时原来的后续链断掉:
 null <- [1,]  [2,] -> [3,] -> [4,] -> [,5,]
  ↑       ↑      ↑
 prev   curt    temp
3. 将prev移到curt位置，curt移动到原来的curt.next,即temp:
null <- [1,]  [2,] -> [3,] -> [4,] -> [,5,]
         ↑      ↑       ↑
        prev   curt    temp

ListNode temp = curt.next;
curt.next = prev
prev = curt;
curt = temp;
```

代码

```
public ListNode reverseList(ListNode head) {
  ListNode prev = null;
  ListNode curt = head;

  while(curt != null){
    ListNode temp = curt.next;
    curt.next = prev;
    prev = curt;
    curt = temp;
  }
  return prev;
}
```

Reverse Linked List II

> Reverse a linked list from position *m* to *n*. Do it in-place and in one-pass.
>
> For example:
>
> Given `1->2->3->4->5->NULL` , *m* = 2 and *n* = 4,
>
> return `1->4->3->2->5->NULL` .
>
> **Note:**
>
> Given *m*, *n* satisfy the following condition:

> 1 ≤ *m* ≤ *n* ≤ length of list.

将链表的第m-n位置上的元素反转

分析

```
[1,]->[2,]->...->[m-1,]->[m,]->...->[n,]->[n+1,]->...
翻转m和n之间的部分，分为三个步骤：
1. 找到m-1和m的点，设为prev和curt
2. 将m~n反转
3. 把m-1.next指向n;把m.next指向n.next
```

代码

```
public ListNode reverseBetween(ListNode head, int m, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;
        ListNode curt = head;

        int i = 1;
        //寻找第m个节点
        while (i < m){
            curt = curt.next;
            prev = prev.next;
            i++;
        }//此时prev指向第m-1个节点，curt指向第m个节点

        //记录下m节点和m-1节点位置，用于反转后连接
        ListNode m_node = curt;
        ListNode m_prev = prev;

        //将m到n反转
        while (i <= n){
            ListNode temp = curt.next;
            curt.next = prev;
            prev = curt;
            curt = temp;
            i++;
        }//此时curt指向第n+1个节点，prev指向第n个节点

        //将m的前序节点的next指向第n个节点
        m_prev.next = prev;
        //将m节点的next指向第n+1个节点
        m_node.next = curt;

        return dummy.next;
    }
```

Partition List

题目

Given a linked list and a value *x*, partition it such that all nodes less than *x* come before nodes greater than or equal to *x*.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,

Given `1->4->3->2->5->2` and *x* = 3,

return `1->2->2->4->3->5` .

给定一个链表和一个数x，将链表中比x小的排在左边，大于等于x的数字排在右边，数字的相对顺序保持不变

分析

将链表排成两队，小于x的一队，大于等于x的一队，然后把两个链表连起来。

链表的结构会发生变化，所以需要两个dummy node，一个用来指向小的队dummy_low，一个用来指向大的队dummy_high。

解题步骤：

1. 遍历数组，将比x小的元素放到dummy_low队伍后面，将比x大的元素放到dummy_high队伍后面
2. 结束后将两个链表连接起来：dummy_low.next指向dummy_high.next
3. 将链表结尾置空：tail.next = null,否则会保留原始节点的next。
4. 返回dummy_low.next;

代码

```java
import java.util.List;

public class PartitionList {
    class ListNode {
        int val;
        ListNode next;

        ListNode(int val) {
            this.val = val;
        }
    }
    public ListNode establish(int[] array){
        ListNode dummy = new ListNode(0);
        ListNode curt = dummy;
        for(int item : array){
            ListNode node = new ListNode(item);
            curt.next = node;
            curt = curt.next;
        }
        curt.next = null;
        return dummy.next;
    }

    public ListNode partition(ListNode head, int x) {
        ListNode dummy_low = new ListNode(0);
        ListNode dummy_high = new ListNode(0);

        ListNode prev_low = dummy_low;//用于向小链表插入
        ListNode prev_high = dummy_high;//用于向大链表插入

        //分别放到两个队伍里
        ListNode curt = head;
        while(curt != null){
            if(curt.val < x){
                prev_low.next = curt;
                prev_low = prev_low.next;
            }
            else{
                prev_high.next = curt;
                prev_high = prev_high.next;
            }
            curt = curt.next;
        }
        //将两链表连接
        prev_low.next = dummy_high.next;
        prev_high.next = null;

        return dummy_low.next;
    }
    public void printList(ListNode head){
        while(head != null){
            System.out.println(head.val);
            System.out.println(" -> ");
            head = head.next;
        }
        System.out.println("null");
    }
    public static void main(String[] args){
        PartitionList test = new PartitionList();
        int[] array = {1,4,3,2,5,2};
        int x = 3;
        ListNode head = test.establish(array);
        head = test.partition(head,x);
        test.printList(head);
    };
}
```

Merge Two Sorted Lists

题目

> Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.
>
> **Example:**
>
> ```
> > Input: 1->2->4, 1->3->4
> > Output: 1->1->2->3->4->4
> >
> ```

分析

链表结构可能改变， 所以需要dummy node

需要一个prev指针记录当前节点，初始化指向dummy_node两个curt指针分别指向两个链表当前节点，用于比较，将比较小的接在prev后面，知道两个curt中有一个为空，将另一个链表的后面直接接到prev后面。

最后返回dummy.next

代码

```java
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode prev = dummy;
        ListNode curt1 = l1;
        ListNode curt2 = l2;
        while (curt1 != null && curt2 != null){
            if(curt1.val < curt2.val){
                prev.next = curt1;
                curt1 = curt1.next;
            }
            else {
                prev.next = curt2;
                curt2 = curt2.next;
            }
            prev = prev.next;
        }
        if(curt1 == null){
            prev.next = curt2;
        }
        else {
            prev.next = curt1;
        }
        return dummy.next;
    }
}
```

Sort List

题目

Sort a linked list in $O(n \log n)$ time using constant space complexity.

将链表排序，时间复杂度为 $O(n \log n)$

分析

时间复杂度为nlogn的排序：quick sort\merge sort\heap sort

quick sort和merge sort的区别：

1. 算法流程

   quik sort：整体有序 -> 局部有序、不稳定排序

  - 整体有序：选定一个元素，比它小的都在它左边，比它大的都在它右边
  - 局部有序：然后再对左段和右段分别做快排

merge sort：局部有序 -> 整体有序、稳定排序

  - 局部有序：选取中点将序列分成左右两段，对左右两边分别排序
  - 整体有序：将左右两边sort list 进行merge操作使得整个list有序

1. 排序的稳定性

```
[2 , 1' , 1'', 1''' , 4 , 3' , 3'']
merge sort的merge操作不会改变元素的相对顺序，所以是稳定排序
quick sort会改变元素的相对位置，所以不是稳定排序
```

2. 时间复杂度

   quick sort平均时间复杂度$O(nlogn)$，最坏时间复杂度$O(n^2)$

   merge sort时间复杂度：$O(nlogn)$

3. 空间复杂度

   quick sort：$O(1)$

   merge sort：$O(n)$，但是在链表中不需要开辟额外的空间

解题步骤：

merge sort

具体步骤：

1. merge sort在链表中找中点，有两种方法：
   1. 遍历一遍，得到链表的长度n，则中间位置是n/2，再从头遍历一遍，到n/2的位置停止，找到中点
   2. 设置两个错位指针，一个slow一个fast，初始化都指向head，slow每次向右移动一位，fast每次向右移动两位，fast移动到末尾的时候，head指向中间，取到链表中点
2. 对左右两段递归进行排序
3. merge两段有序链表

代码

```
merge sort:
//寻找链表中点,中间偏前
public ListNode findMid(ListNode head){
  if(head == null){
    return head;
  }
  ListNode slow = head;
  ListNode fast = head;
  while(fast.next != null && fast.next.next != null) {
    slow = slow.next;
    fast = fast.next.next;
  }
  return slow;
}

//merge两段有序链表
public ListNode merge(ListNode left_h,ListNode right_h){
  ListNode dummy = new ListNode(0);
  ListNode prev = dummy;
  ListNode curt_l = left_h;
  ListNode curt_r = right_h;
  while (curt_l != null && curt_r != null){
    if(curt_l.val < curt_r.val){
      prev.next = curt_l;
```

```
      prev = prev.next;
      curt_l = curt_l.next;
    }
    else {
      prev.next = curt_r;
      prev = prev.next;
      curt_r = curt_r.next;
    }
  }
  if(curt_l == null){
    prev.next = curt_r;
  }
  if(curt_r == null){
    prev.next = curt_l;
  }
  return dummy.next;
}

//递归调用merge sort
public ListNode sortList(ListNode head) {
  if(head == null){
    return null;
  }
  if(head.next == null){
    return head;
  }
  //寻找链表中点mid
  ListNode mid = findMid(head);

  //链表中点.next之后的链表排序
  ListNode right = sortList(mid.next);

  //链表中点之前包括中点的链表排序
  mid.next = null;
  ListNode left = sortList(head);

  //merge两段有序链表
  ListNode res = merge(left,right);
  return res;
}
```

Reorder List

题目

> Given a singly linked list *L*: *L*0→*L*1→…→*L***n*-1→*L*n,
>
> reorder it to: *L*0→*L***n*→*L*1→*L***n*-1→*L*2→*L***n*-2→…
>
> You must do this in-place without altering the nodes' values.
>
> For example,
>
> Given `{1,2,3,4}` , reorder it to `{1,4,2,3}` .

分析

题目需要从后向前访问链表，但是链表是单向的，所以需要reverse反转操作，再和原链表merge。

具体步骤：

1. 找到链表中点，mid
2. 将mid之后的链表反转
3. 将mid之前的链表和mid之后反转的链表做merge操作

代码

```
//找到中间位置的元素
public ListNode findMid(ListNode head){
    if(head == null){
        return head;
        }
        ListNode slow = head;
        ListNode fast = head;
        while(fast.next != null && fast.next.next != null){
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    //链表反转
    public ListNode reverse(ListNode head){
        ListNode prev = null;
        ListNode curt = head;
        while(curt != null){
            ListNode temp = curt.next;
            curt.next = prev;
            prev = curt;
            curt = temp;
        }
        return prev;
    }
    //交替merge
    public void merge(ListNode left,ListNode right){
        ListNode prev = left;
        ListNode curt1 = left.next;
        ListNode curt2 = right;
        int i = 0;
        while(curt1 != null && curt2 != null){
            if(i % 2 == 0){
                prev.next = curt2;
                curt2 = curt2.next;
            }
            else {
                prev.next = curt1;
                curt1 = curt1.next;
            }
            prev = prev.next;
            i++;
        }
        if(curt1 == null){
            prev.next = curt2;
        }
        else {
            prev.next = curt1;
        }
    }
    //
    public void reorderList(ListNode head) {
        //边界条件
        if(head == null || head.next == null){
            return;
        }
        //找到mid
        ListNode mid = findMid(head);

        //将mid之后的部分反转
        ListNode right = reverse(mid.next);
        mid.next = null;

        //前半部分和反转后的后半部分merge
        merge(head,right);
    }
```

Fast-slow pointer

1. Middle of Linked List

寻找指针链表中点，快慢指针，快指针每次走两步，慢指针每次走一步，快指针走到链表结尾时，慢指针在中间

2 .Remove Nth Node From End of List

题目

> Given a linked list, remove the nth node from the end of list and return its head.

> For example,
>
> ```
> >    Given linked list: 1->2->3->4->5, and n = 2.
> >
> >    After removing the second node from the end, the linked list becomes 1->2->3->5.
> >
> >
> ```

>

> **Note:**
>
> Given *n* will always be valid.
>
> Try to do this in one pass.

删除掉从末尾开始第n个节点

分析

两个指针，fast先走n+！步slow再出发，当fast==null时，slow指向倒数第n+1个节点，删掉slow后面的节点：slow.next = slow.next.next

代码

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode slow = dummy;
    ListNode fast = dummy;
    //fast先走n+1步
    int i = 0;
    while(i <= n){
        fast = fast.next;
        i++;
    }
    //fast、slow同时向后遍历
    while(fast != null){
        fast =fast.next;
        slow = slow.next;
    }
    //删除节点
    slow.next = slow.next.next;

    return dummy.next;
}
```

3. Linked List Cycle

题目

> Given a linked list, determine if it has a cycle in it.
>
> Follow up:
>
> Can you solve it without using extra space?

给定一个链表，判断是否有圈

分析

方法一：

判断node是否有被重复访问

从head出发，把所有访问过的点放到一个hash表里，空间复杂度O(n)

方法二：

一个快指针一个慢指针，如果路径上有环，快慢指针一定会相遇

初始化：slow = head;fast = head.next

代码

```
public boolean hasCycle(ListNode head) {
    if (head == null){
        return false;
    }
    ListNode fast = head.next;
    ListNode slow = head;
    while(fast != slow){
        //fast走到null
        if(fast == null || fast.next == null){
            return false;
        }
        fast = fast.next.next;
        slow =slow.next;
    }
    //fast和slow相遇了
    return true;
}
```

4. Linked List Cycle II

题目

> Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.
>
> **Note:** Do not modify the linked list.
>
> **Follow up**:
>
> Can you solve it without using extra space?

是否有环，如果有，找到环的入口

分析

slow从快慢指针相遇的地方出发，fast指针从初始地方出发，两个指针每次走一步，直到相遇，就是环的入口

代码

```
public ListNode detectCycle(ListNode head) {
  if (head == null){
    return null;
  }
  ListNode fast = head.next;
  ListNode slow = head;
  while(fast != slow){
    //fast走到null
    if(fast == null || fast.next == null){
      return null;
    }
    fast = fast.next.next;
    slow = slow.next;
  }

  //fast和slow相遇了
  slow = head;
  fast = fast.next;
  while(slow != fast){
    slow = slow.next;
    fast = fast.next;
  }
  return slow;
}
```

5. Rotate List

题目

> Given a list, rotate the list to the right by *k* places, where *k* is non-negative.
>
> **Example:**
>
> ```
> > Given 1->2->3->4->5->NULL and k = 2,
> > return 4->5->1->2->3->NULL.
> >
> ```

分析

将链表向后移k次

```
Given 1->2->3->4->5->NULL and k = 2,
            ↑  ↑  ↑
node.next=head ↑  ↑
      dummy.next tail.next=head
```

求解步骤：

1. 求链表长度len，如果k>len,k = k%len;
2. 找到从后往前数第k个元素，也就是从前往后数第len-k个元素node，和末尾元素tail
3. tail.next = dummy.next;dummy.next = node.next;node.next = null

代码

```
public ListNode rotateRight(ListNode head, int k) {
        if(head == null){
            return head;
        }
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode tail = dummy;
        int len = 0;
        while(tail.next != null){
            tail = tail.next;
            len++;
        }
        k = k % len;
        ListNode node = dummy;
        int i = 1;
        while(i <= len - k){
            node = node.next;
            i++;
        }
        tail.next = dummy.next;
        dummy.next = node.next;
        node.next = null;

        return dummy.next;
    }
```

Merge k Sorted Lists

题目

> Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

merge k 个有序链表

分析

一共三种方法，都需要掌握：

方法一：heap

用PriorityQueue实现

第一个参数 - 第二个参数：升序，最小堆

第二个参数 - 第一个参数：降序，最大堆

初始化：将链表头放进去

每次弹出最小的元素，放到结果链表后面，然后将其next入堆，重复上述

N：所有数的个数

K：链表个数

时间复杂度：$O(NlogK)$，heap中最多有k各元素，插入操作时间复杂度是$O(logk)$

空间复杂度：$O(K)$

```
//heap
public ListNode mergeKLists(ListNode[] lists) {
        int k = lists.length;
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>(new Comparator<ListNode>() {
           // @Override
            public int compare(ListNode o1, ListNode o2) {
                return o1.val - o2.val;
            }
        });
        ListNode dummy = new ListNode(0);
        ListNode prev = dummy;
        for(int i = 0; i < k;i++){
            ListNode head = lists[i];
            if(head != null){
                minHeap.add(head);
            }
        }
        while(!minHeap.isEmpty()){
            ListNode curt = minHeap.poll();
            prev.next = curt;
            if(curt.next != null){
                minHeap.add(curt.next);
            }
            prev = prev.next;
        }
        prev.next = null;
        return dummy.next;
    }
```

方法二：分治法

merge k 个链表

- 拆分成merge前k/2个链表得到list1和merge后k/2个链表得到list2
- 合并list1和 list2，得到结果

递归调用求解上述子问题

时间复杂度：$O(NlogK)$

```
     result
     ↗    ↖
   ↗      ↗ ↖
↗  ↖    ↗   ↗  ↖
1       2 |     3 |     4       5
```

```
    //分治
    //merge两个List
    public ListNode MergeTwoList(ListNode left,ListNode right){
        ListNode dummy = new ListNode(0);
        ListNode prev = dummy;
        ListNode prev1 = left;
        ListNode prev2 = right;
        while(prev1 != null && prev2 != null){
            if(prev1.val < prev2.val){
```

```
                prev.next = prev1;
                prev1 = prev1.next;
                prev = prev.next;
            }
            else{
                prev.next = prev2;
                prev2 = prev2.next;
                prev = prev.next;
            }
        }
        if(prev1 == null){
            prev.next = prev2;
        }
        if(prev2 == null){
            prev.next = prev1;
        }
        return dummy.next;
    }

    //分治法mergek个数组
    public ListNode divideMergeKList(ListNode[] lists,int start,int end) {
        if(start == end){
            return lists[start];
        }

        //拆分成两部分
        ListNode left = divideMergeKList(lists,start,start+(end-start)/2);
        ListNode right = divideMergeKList(lists,start+(end-start)/2+1,end);
        //合并两部分结果返回
        return MergeTwoList(left,right);
    }
    //调用分治法
    public ListNode DividemergeKLists(ListNode[] lists){
        int len = lists.length;
        if(len ==0){
            return null;
        }
        return divideMergeKList(lists,0,len-1);
    }
```

方法三：两两合并

1、2合并，3、4合并，....n

向上递归合并

时间复杂度：$O(NlogK)$

如果是1、2合并，然后忽然3合并，...n

时间复杂度O(NK)

```
 //两两合并
public ListNode mergeKListsOneByOne(ListNode[] lists){
  if(lists.length == 0){
    return null;
  }
  List<ListNode> newlists = new ArrayList<>();
  for(int i = 0; i < lists.length;i++) {
    newlists.add(lists[i]);
  }
  while (newlists.size() > 1){
    List<ListNode> listTemp = new ArrayList<>();
    for(int i = 0;i+1 < newlists.size();i+=2){
      listTemp.add(MergeTwoList(newlists.get(i),newlists.get(i+1)));
    }
    if(newlists.size() % 2 == 1){
      listTemp.add(newlists.get(newlists.size()-1));
    }
    newlists = listTemp;
  }
  return newlists.get(0);
}
```

Copy List with Random Pointer

题目

> A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.
>
> Return a deep copy of the list.

给定一个链表，每个节点除包含一个next指针以外，还有一个指向任意节点的random pointer，clone链表

分析

方法一：

hash_map

先按next指针复制链表，把原链表老节点和新链表新节点的映射关系存入hash_map，再遍历一遍原链表，按照hash_map中的对应关系，把random pointer在对应的新节点中标出。

空间复杂度$O(n)$

```
public RandomListNode copyRandomList(RandomListNode head) {
       RandomListNode dummy = new RandomListNode(0);
       RandomListNode prevNew = dummy;
       RandomListNode prev = head;
       HashMap<RandomListNode,RandomListNode> map = new HashMap<>();
       //将链表和next指针复制，对应点存入hashmap
       while(prev != null){
           RandomListNode temp = new RandomListNode(prev.label);
           prevNew.next = temp;
           map.put(prev,prevNew.next);
           prev = prev.next;
           prevNew = prevNew.next;
       }
       //复制random poiner
       prev = head;
       prevNew = dummy.next;
       while(prev != null){
           prevNew.random = map.get(prev.random);
           prev = prev.next;
           prevNew = prevNew.next;
       }
       return dummy.next;
   }
```

方法二：

代码

```
public RandomListNode copyRandomList2(RandomListNode head) {
  RandomListNode prev = head;
  //将每个元素都复制一份，插在原来元素的后面一位上
  while(prev != null){
    RandomListNode node = new RandomListNode(prev.label);
    node.next = prev.next;
    prev.next = node;
    prev = prev.next.next;
  }

  //加入random pointer
  prev = head;
  while(prev != null){
    if(prev.random != null){
      prev.next.random = prev.random.next;
    }
    prev = prev.next.next;
  }
  //删掉原来元素;
  RandomListNode dummy = new RandomListNode(0);
  dummy.next = head;
  prev = dummy;
  RandomListNode curt = head;
  while(curt != null){
```

```
    prev.next = curt.next;
    curt.next = curt.next.next;
    prev = prev.next;
    curt = curt.next;
  }
  return dummy.next;
}
```

总结

leetcode 相关习题

Palindrome Linked List

题目

> Given a singly linked list, determine if it is a palindrome.
>
> **Follow up:**
>
> Could you do it in O(n) time and O(1) space?

给定链表，判读是否是回文串，要求时间复杂度$O(n)$，空间复杂度$O(1)$

分析

方法一：

利用stack，先进后出的性质：

1. 找到终点，过程中将前半部分链表入栈
2. 继续向后遍历，出栈，对比元素是否一致

```
public boolean isPalindromeStack(ListNode head) {
        //空链表和只有一个元素
        if(head == null || head.next == null){
            return true;
        }

        //快慢指针寻找中点,前半部分元素入栈
        Stack<Integer> stack = new Stack<>();
        ListNode mid = head;
        ListNode tail = head;
        stack.push(head.val);
        while(tail.next != null && tail.next.next != null){
            mid = mid.next;
            stack.push(mid.val);
            tail = tail.next.next;
        }
        //如果是奇数个元素，将mid弹出，无须比较
        if(tail.next == null){
            stack.pop();
        }
        mid = mid.next;
        while(mid != null){
            int temp = stack.peek();
            if(temp != mid.val){
                return  false;
            }
            stack.pop();
            mid = mid.next;
        }
        return true;
    }
```

方法二：

1. 先找到中点

2. 将后半部分的链表反转
3. 对比前后两部分是否一致

```
 public boolean isPalindrome(ListNode head) {
        if(head == null){
            return true;
        }
        //快慢指针寻找中点
        ListNode mid = head;
        ListNode tail = head;
        while(tail.next != null && tail.next.next != null){
            mid = mid.next;
            tail = tail.next.next;
        }

        //反转后半个链表
        ListNode prev = null;
        ListNode curt = mid.next;
        while(curt !=  null){
            ListNode temp = curt.next;
            curt.next = prev;
            prev = curt;
            curt = temp;
        }
        mid.next = prev;
        //对比两段元素是否一致
        ListNode first = head;
        ListNode second = mid.next;
        while(second != null){
            if(first.val != second.val){
                return false;
            }
            first = first.next;
            second = second.next;
        }
        return true;
    }
```

160.Intersection of Two Linked Lists

题目

> Write a program to find the node at which the intersection of two singly linked lists begins.
>
> For example, the following two linked lists:
>
> ```
> > A:          a1 → a2
> >                     ↘
> >                      c1 → c2 → c3
> >                     ↗
> > B:     b1 → b2 → b3
> >
> >
> ```

>

> begin to intersect at node c1.
>
> **Notes:**
>
> - If the two linked lists have no intersection at all, return `null`.
> - The linked lists must retain their original structure after the function returns.
> - You may assume there are no cycles anywhere in the entire linked structure.
> - Your code should preferably run in O(n) time and use only O(1) memory.

找到两个链表相交的地方

分析

是个技巧题，想到了就能做出来，想不到就做不出来

方法：

两个指针分别遍历，一个先A后B，一个先B后A，两指针指向节点相等即为相交处

代码

```
 public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if(headA == null || headB == null){
            return null;
        }
        ListNode n1 = headA;
        ListNode n2 = headB;
        //记录是否遍历第二个链表
        boolean flag1 = false;
        boolean flag2 = false;
        while(true){
            //两个指针都遍历结束了，没有相交节点
            if(n1 == null && flag1){
                return null;
            }
            //n1遍历完一个数组
            if(n1 == null && !flag1){
                 n1 = headB;
                 flag1 = true;
            }
            //n2遍历完一个数组
            if(n2 == null && !flag2){
                n2 = headA;
                flag2 = true;
            }
            if (n1 == n2){
                return n1;
            }
            n1 = n1.next;
            n2 = n2.next;
        }
    }

    public int findDuplicate(int[] nums) {
        for(int i = 0 ; i < nums.length;i++){
            int idx = Math.abs(nums[i]);
            if(nums[idx] < 0){
                return idx;
            }
            nums[idx] = -nums[idx];
        }
        return -1;
    }
```

‹ DeepFM论文笔记 【九章算法基础班】数与数组 ›