> 2018年10月26日

本文主要做一些链表的常见题目，题目从`LeetCode`上摘取，通过练习加深对链表的掌握和理解。

定义链表的节点类：

```
    class ListNode {
        int val;
        ListNode next;

        ListNode(int x) {
            val = x;
        }
    }
```

1，反转链表

题选自`LeetCode`206题：

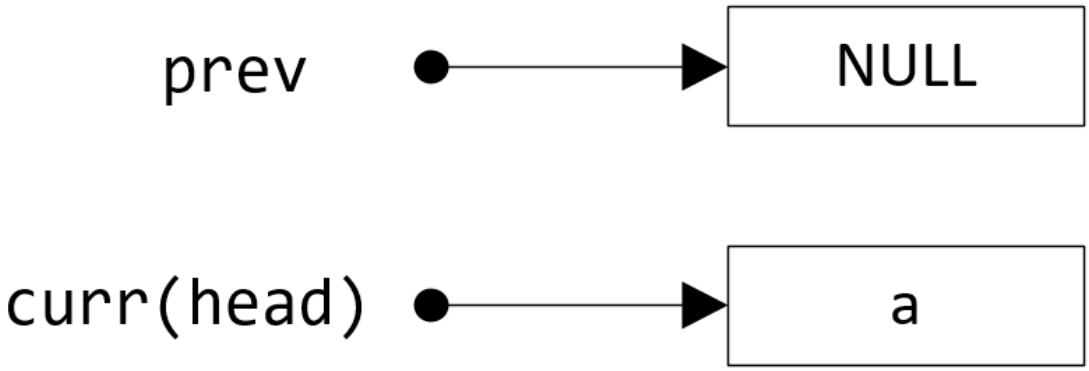
```
反转一个单链表。

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

头插法

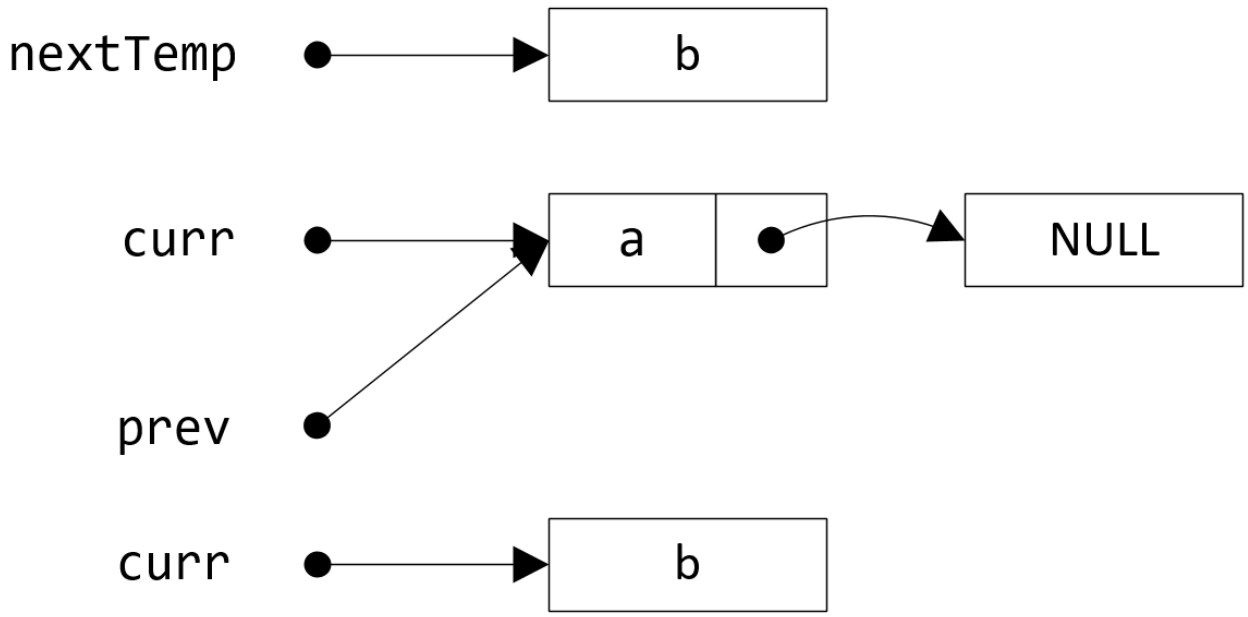
```
    public static ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }
```

假设反转如下链表：

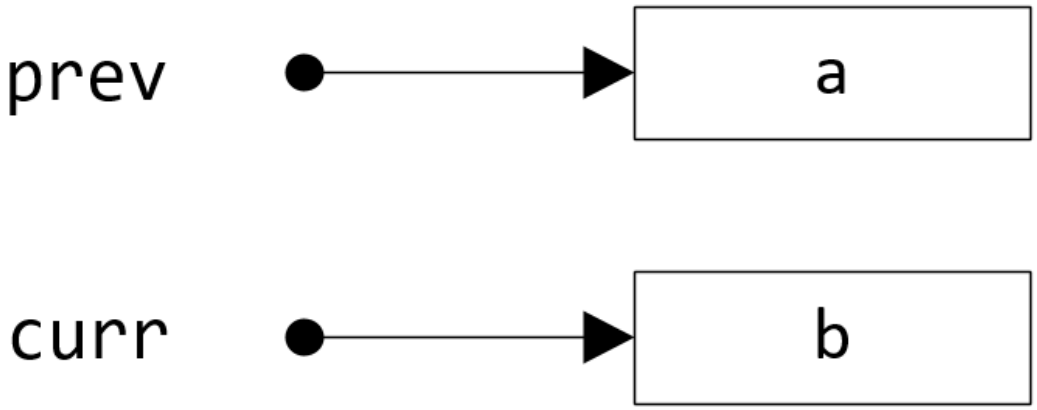
第一次循环时，`curr`与`prev`为：



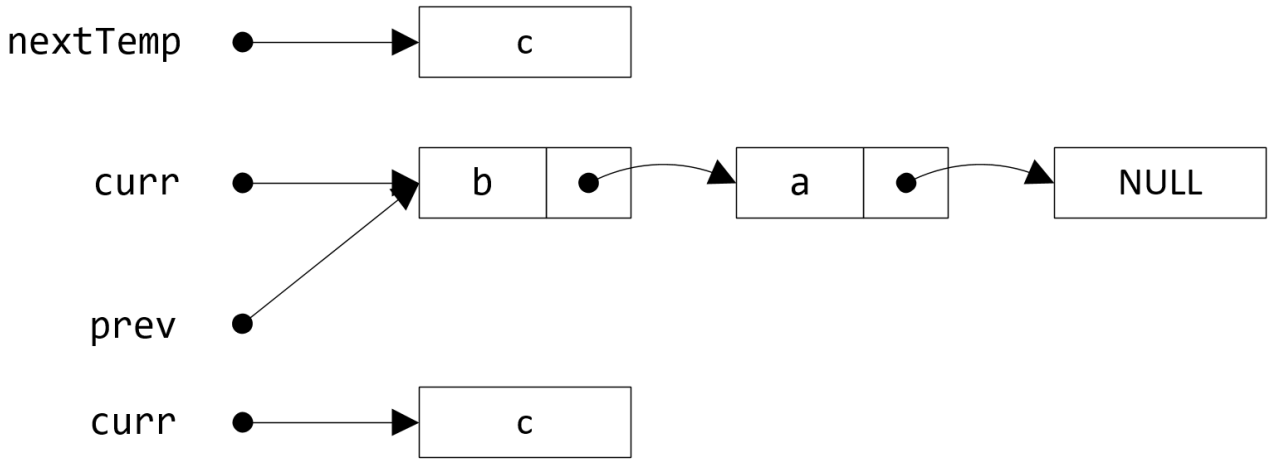
第一次循环后各个属性为：



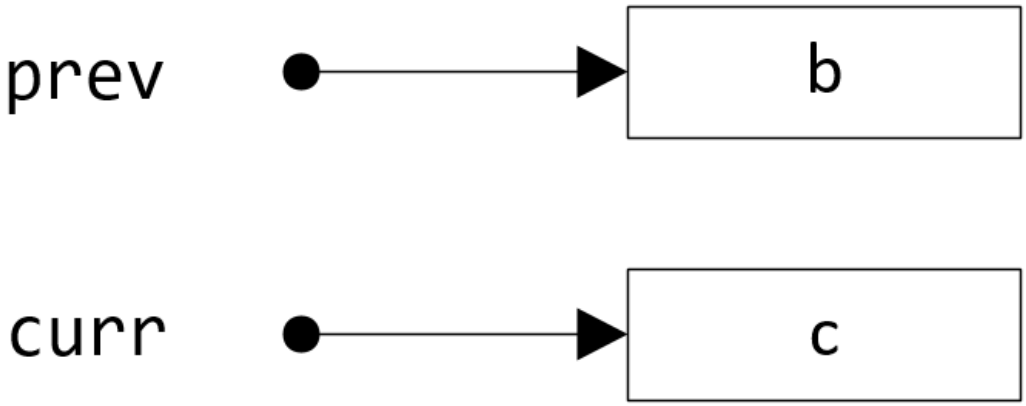
第二次循环时，`curr`与`prev`为：



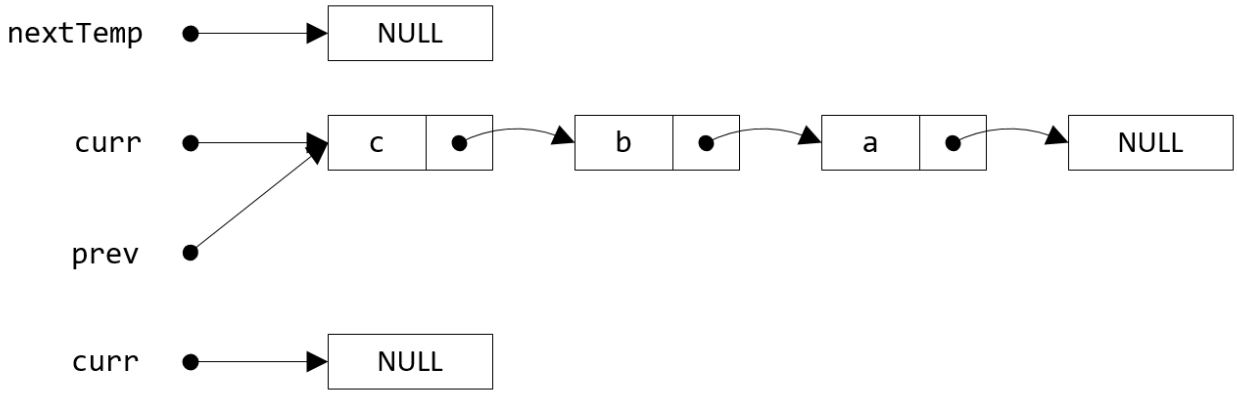
第二次循环后各个属性为：



第三次循环时，`curr`与`prev`为：



第三次循环后各个属性为：

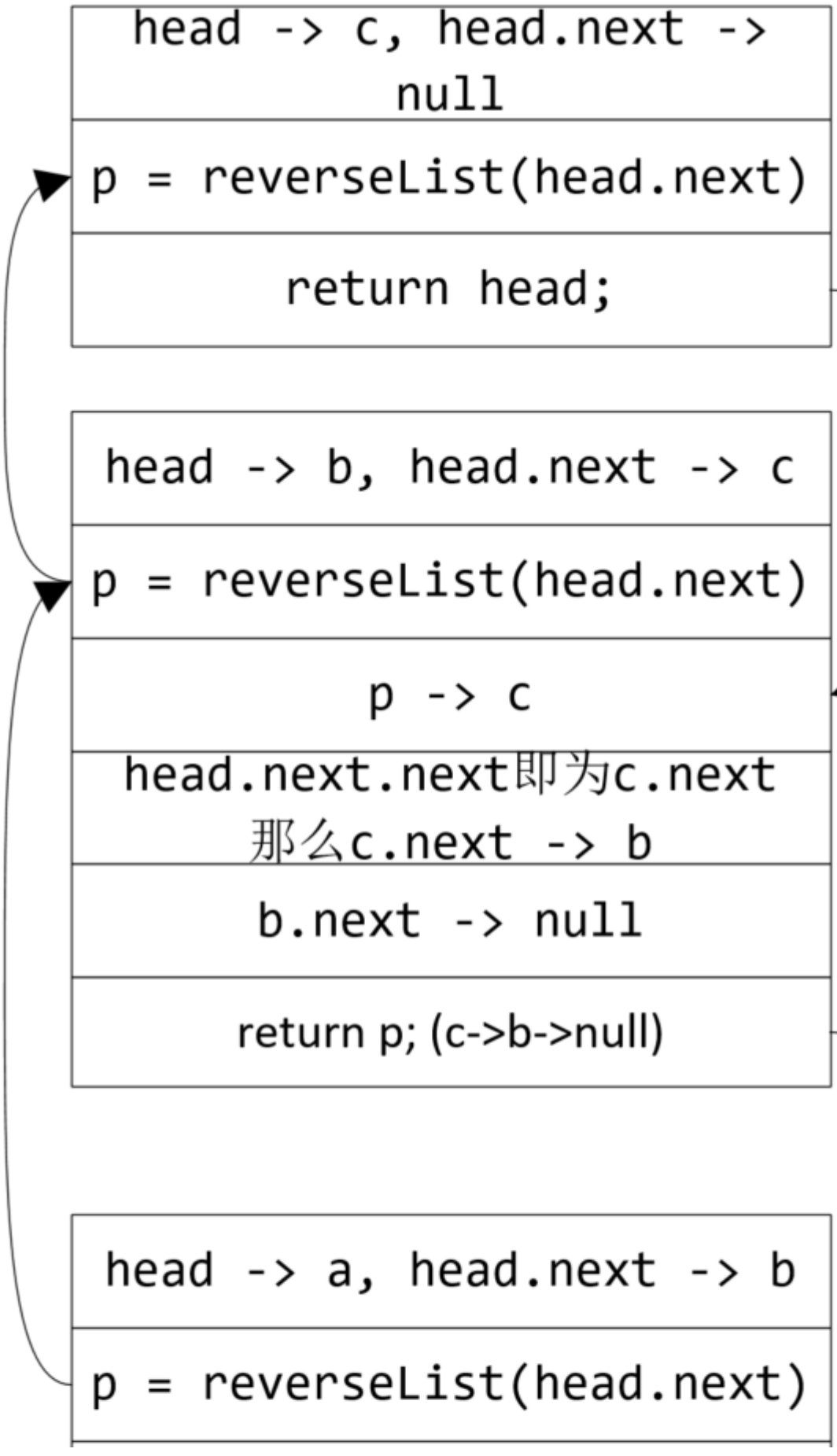


三次循环后，可以看到把链表反转了，其时间复杂度为 $O(N)$，空间复杂度为 $O(1)$。

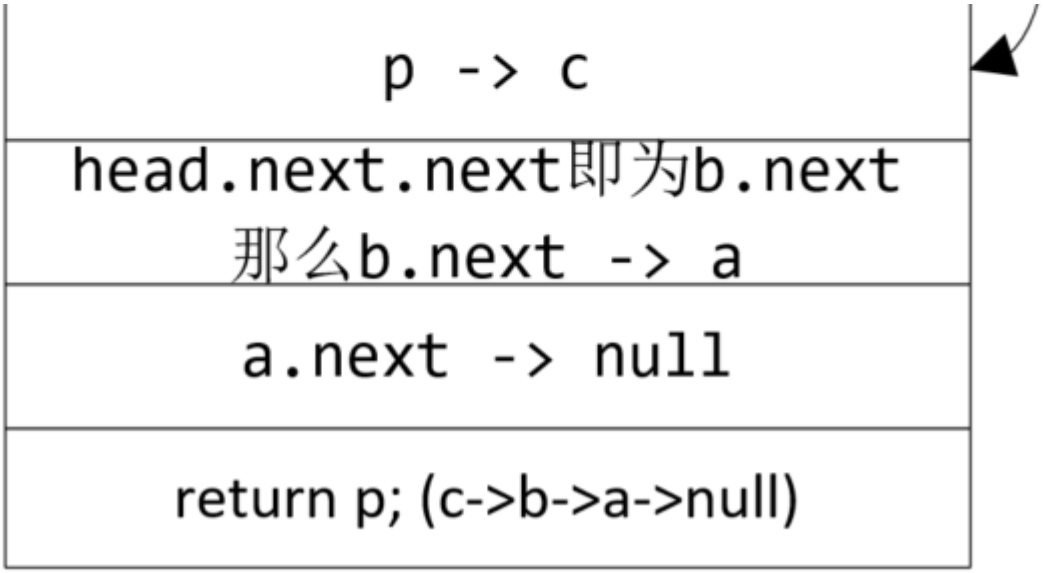
递归

```java
    public static ListNode reverseList1(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        //head是p的前一个节点
        ListNode p = reverseList1(head.next);
        //相当于p.next=head
        head.next.next = head;
        //使p的尾节点为null
        head.next = null;
        return p;
    }
```

其栈的递归调用过程如下：


head -> c, head.next -> null
p = reverseList(head.next)
return head;
head -> b, head.next -> c
p = reverseList(head.next)
p -> c
head.next.next即为c.next
那么c.next -> b
b.next -> null
return p; (c->b->null)
head -> a, head.next -> b
p = reverseList(head.next)

递



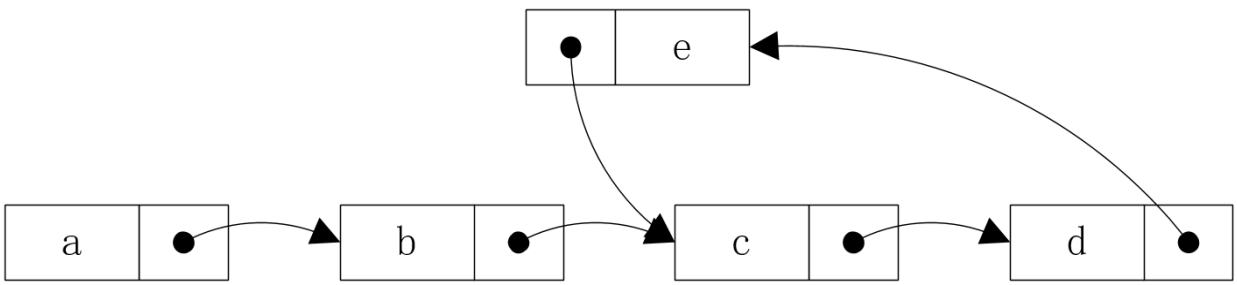
归其时间复杂度为 $O(N)$，空间复杂度为 $O(N)$。

2，检测链表中是否有环

题选自`LeetCode`141题：

```
1 给定一个链表，判断链表中是否有环。
```

如下链表就是有环：



使用HashSet

```java
public boolean hasCycle(ListNode head) {
    Set<ListNode> nodesSeen = new HashSet<>();
    while (head != null) {
        if (nodesSeen.contains(head)) {
            return true;
        } else {
            nodesSeen.add(head);
        }
        head = head.next;
    }
    return false;
}
```

利用`Set`中不能有相同元素这一特性，在往`nodesSeen`集合中添加元素时，一旦有相同元素就返回`true`，表示有环，若没有环，那么遇到`null`节点时，会结束循环，并返回`false`，表示没有环。

使用快慢指针

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

试想这么一个场景，甲乙两人绕着标准操场（400m类似椭圆形）跑步，甲的速度比乙快，因为操场时有环的，那么在某一时刻，甲肯定会追上乙，与乙相遇；其中`slow`表示慢的指针，每次只走一步，而`fast`表示快的指针，每次走两步，一旦`slow`与`fast`相等，即它们都指向同一个元素，终止循环，并返回`true`，表示有环；若`fast`(偶数情况)或`fast.next`(奇数情况)指向`null`，表示这个链表没有环，返回`false`。

3，合并两个有序链表

题选自`LeetCode`21题：

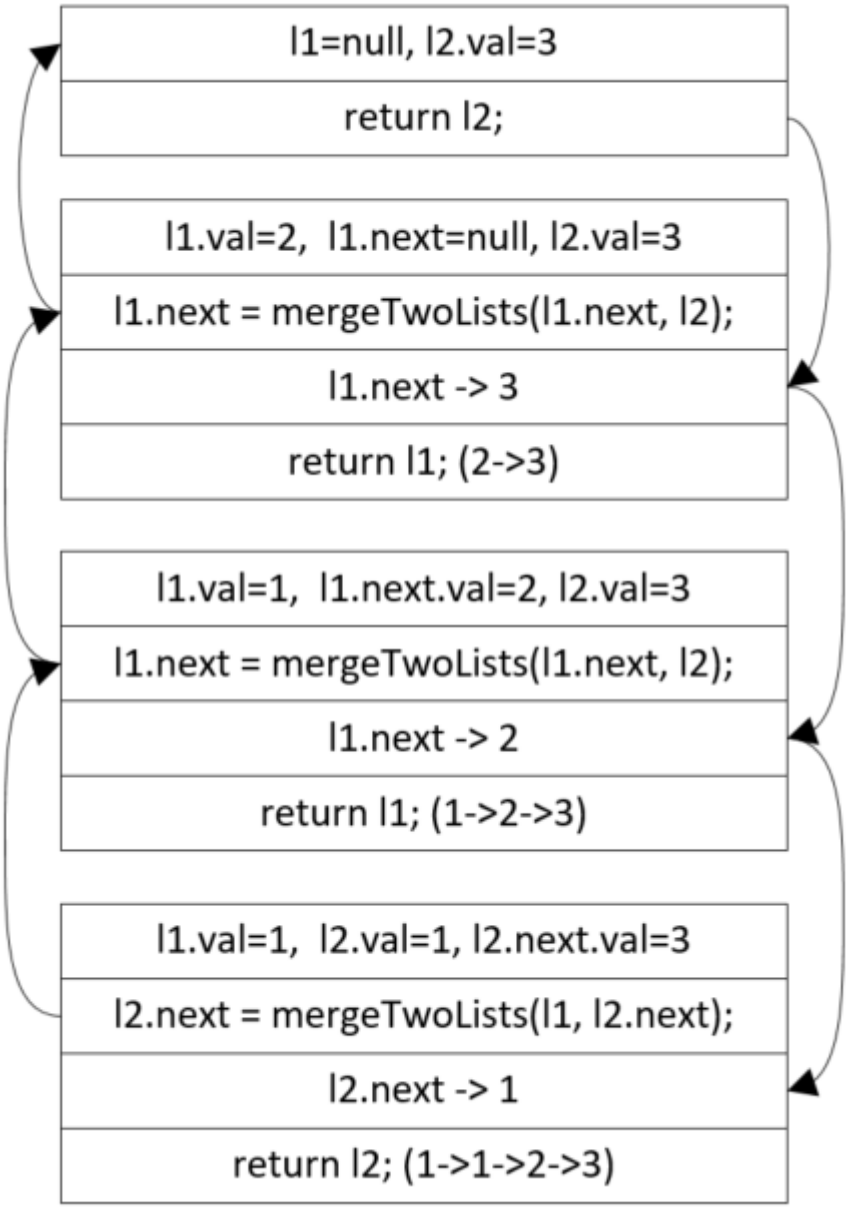
```
将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

```java
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        ListNode listNode = new ListNode(0);
        ListNode curr = listNode;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                curr.next = l1;
                l1 = l1.next;
            } else {
                curr.next = l2;
                l2 = l2.next;
            }
            curr = curr.next;
        }
        if (l1 != null) {
            curr.next = l1;
        }
        if (l2 != null) {
            curr.next = l2;
        }
        return listNode.next;
    }
```

递归

```java
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null) {
            return l2;
        }
        if (l2 == null) {
            return l1;
        }
        if (l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
```

以`l1 = 1 -> 2`，`l2 = 1 -> 3`为例，其栈递归调用图如下：



4，删除链表的倒数第 n 个节点

题选自`LeetCode`19题：

```
给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：
给定一个链表: 1->2->3->4->5, 和 n = 2.
当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

解法1

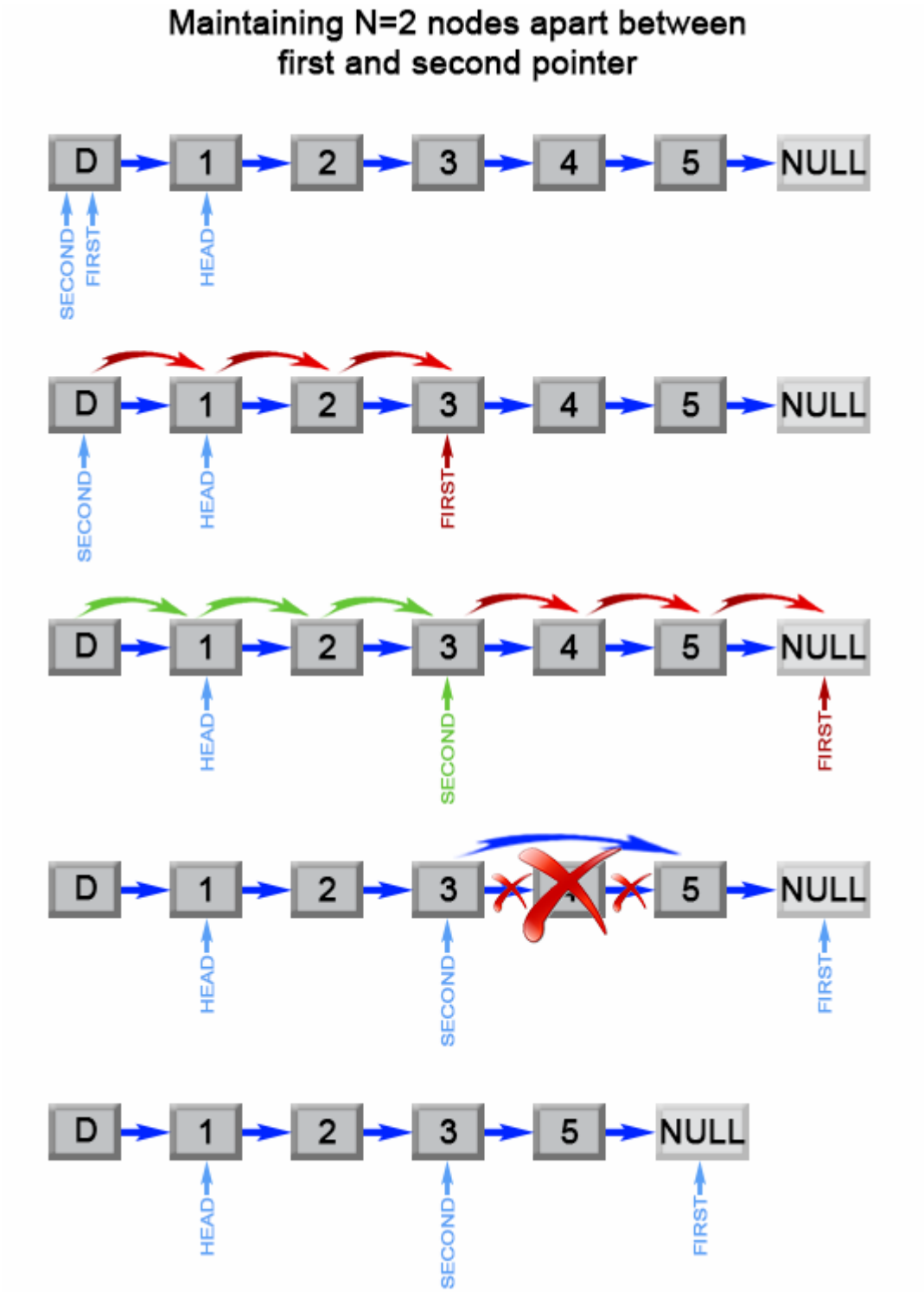
```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    int length  = 0;
    ListNode first = head;
    while (first != null) {
        length++;
        first = first.next;
    }
    length -= n;
    first = dummy;
    while (length > 0) {
        length--;
        first = first.next;
    }
    first.next = first.next.next;
    return dummy.next;
}
```

假设链表的长度为L，那么删除倒数第n个节点，即删除整数第`L-n+1`个节点，那么就需要获得其前一个节点，即第`L-n`个节点

解法2

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode first = dummy;
    ListNode second = dummy;

    for (int i = 1; i <= n + 1; i++) {
        first = first.next;
    }

    while (first != null) {
        first = first.next;
        second = second.next;
    }
    second.next = second.next.next;
    return dummy.next;
}
```

使用双指针，`first`在前面跑，因为要铲除倒数第n个节点，那么就要获取到倒数第`n+1`个节点，所以使`first`与`second`的距离保持为`n+1`：



5，链表的中间结点

题选自`LeetCode`876题：

```
1 给定一个带有头结点 head 的非空单链表，返回链表的中间结点。
```

```
如果有两个中间结点，则返回第二个中间结点。

示例 1:
输入：[1,2,3,4,5]
输出：此列表中的结点 3 (序列化形式：[3,4,5])
返回的结点值为 3 。 (测评系统对该结点序列化表述是 [3,4,5])。
注意，我们返回了一个 ListNode 类型的对象 ans，这样:
ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及
ans.next.next.next = NULL.

示例 2:
输入：[1,2,3,4,5,6]
输出：此列表中的结点 4 (序列化形式：[4,5,6])
由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。
```

使用数组

```java
    public ListNode middleNode(ListNode head) {
        ListNode[] A = new ListNode[100];
        int t = 0;
        while (head.next != null) {
            A[t++] = head;
            head = head.next;
        }
        return A[t / 2];
    }
```

使用快慢指针

```java
    public ListNode middleNode(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }
```

6，LRU缓存机制

题选自`LeetCode`146题

```
运用你所掌握的数据结构，设计和实现一个  LRU (最近最少使用) 缓存机制。它应该支持以下操作:
获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值 (总是正数)，否则返回 -1。
写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:
你是否可以在 O(1) 时间复杂度内完成这两种操作?

示例:

LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

$O(N)$的解法

```java
public class LRUCache {
    private int capacity;
    private HashMap<Integer, Integer> cacheData;
    private ArrayDeque<Integer> deque;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        cacheData = new HashMap<Integer, Integer>();
        deque = new ArrayDeque<>();
    }

    public int get(int key) {
        if (cacheData.containsKey(key)) {
```

```
            deque.remove(key);
            deque.add(key);
            return cacheData.get(key);
        }
        return -1;
    }

    public void put(int key, int value) {
        if (cacheData.containsKey(key)) {
            deque.remove(key);
        }
        if (deque.size() == capacity) {
            cacheData.remove(deque.pollFirst());
        }
        cacheData.put(key, value);
        deque.add(key);
    }
}

```

因为`ArrayDeque`中的`remove`的时间复杂度为$O(N)$，因此总的时间复杂度为$O(N)$。

$O(1)$解法

```
public class LRUCacheByList {
    private int size;
    private int capacity;
    private HashMap<Integer, Node> cacheData;
    private Node head;
    private Node tail;

    public LRUCacheByList(int capacity) {
        this.capacity = capacity;
        cacheData = new HashMap<>();
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if (cacheData.containsKey(key)) {
```

```
            Node node = cacheData.get(key);
            remove(node);
            addLast(node);
            return node.val;
        }
        return -1;
    }

    public void put(int key, int value) {
        if (cacheData.containsKey(key)) {
            Node node = cacheData.get(key);
            node.val = value;
            remove(node);
            addLast(node);
            return;
        }

        Node node = new Node(key, value);
        addLast(node);
        cacheData.put(key, node);
        size++;

        if (size > capacity) {
            cacheData.remove(removeFirst());
            size--;
        }
    }

    private void addLast(Node node) {
        node.prev = tail.prev;
        node.next = tail;

        tail.prev.next = node;
        tail.prev = node;
    }

    private int removeFirst() {
        Node next = head.next;
        Node nextNext = next.next;

        next.prev = null;
        next.next = null;
```

```
61
62          nextNext.prev = head;
63          head.next = nextNext;
64
65          return next.key;
66      }
67
68      private void remove(Node node) {
69          Node prev = node.prev;
70          Node next = node.next;
71
72          node.prev = null;
73          node.next = null;
74
75          prev.next = next;
76          next.prev = prev;
77      }
78
79      private class Node {
80          int key;
81          int val;
82          Node next;
83          Node prev;
84
85          Node(int key, int val) {
86              this.key = key;
87              this.val = val;
88          }
89      }
90  }
```

上面两种解法思路都是将数据存在`HashMap`中，使用一个双链表表示数据的“冷热”程度，最新添加的数据从链表尾部插入，最近访问的数据线将其从链表中删除，在将其从链表尾部插入；当空间满了，就删除链表头部的数据；越靠近链表表头，数据越“冷”，越靠近链表尾部，数据越“热”。

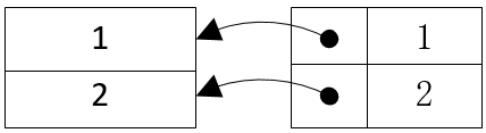
上面两种解法都是使用了`HashMap`与双链表来实现，唯一的区别就是$O(N)$的解法使用JAVA库中的`ArrayDeque`来实现双链表；而$O(1)$解法中自己实现双链表，将节点作为`HashMap`中的值，当要删除链表中某个节点，通过`HashMap`取出这个节点，直接更改`prev`与`next`即可删除，所以其时间复杂度为$O(1)$。使用库中的数据结构时，无论是`ArrayDeque`还是

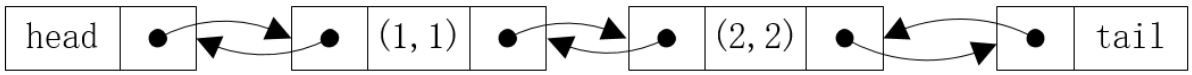
`LinkedList`，其节点信息都封装在其类里面，无法获取，因此要删除某个节点，只能从头开始遍历，找出与要删除的节点值相同的节点，然后在将其删除，因此其时间复杂度为$O(N)$。

下面将以图解形式分析`LRU缓存机制`，当执行完以下代码时：

```
LRUCache cache = new LRUCache( 2 );
cache.put(1, 1);
cache.put(2, 2);
```

`HashMap`与链表中的结构如下：

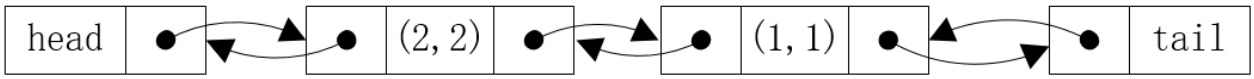




执行：

```
cache.get(1);       // 返回  1
```

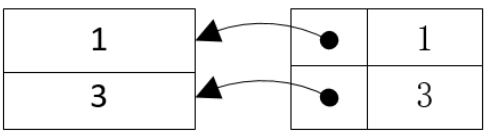
此时`HashMap`中没有改动，链表改动如下：

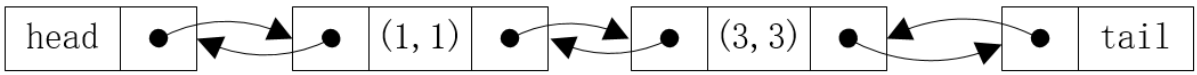


执行：

```
cache.put(3, 3);    // 该操作会使得密钥 2 作废
```

此时`HashMap`与链表改动如下：





执行：

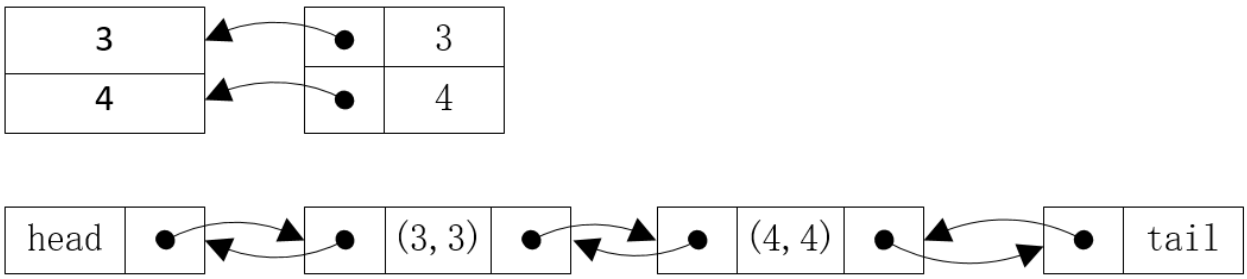
```
cache.get(2);       // 返回 -1 (未找到)
```

此时`HashMap`与链表均没有改动。

执行：

```
cache.put(4, 4);    // 该操作会使得密钥 1 作废
```

此时`HashMap`与链表改动如下：



执行：

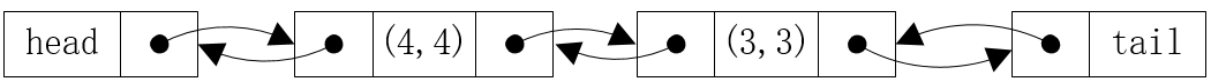
```
cache.get(1);       // 返回 -1 (未找到)
```

此时`HashMap`与链表均没有改动。

执行：

```
cache.get(3);       // 返回  3
```

此时`HashMap`中没有改动，链表改动如下：



执行：

```
cache.get(4);       // 返回  4
```

此时`HashMap`中没有改动，链表改动如下：