Chapter 4 LinkedList

```
/*
Dummy node
Dummy Node的使用多针对Single List没有前向指针的问题, 保证链表的head不会在删除操作中丢失, 基本用法如下
ListNode dummy = new ListNode(Integer.MIN_VALUE);
ListNode current = dummy;
...
while (current != null) {
    ...
    current.next = blablabla;
    ...
    current = current.next;
}
...
return dummy.next;
除此之外, 还有一种用法比较少见, 就是使用dummy node来进行head的删除操作, 比如Remove Duplicates From Sorted List II, 一般的方法current = current.next是无法删除head元素的, 所以这个时候如果有一个dummy node在head的前面.
综合以上两种情况, 我理解是dummy node使用在头结点 (head) 无法确定的时候, 所谓无法确定包括:
head可能被删除
head可能被修改
*/

Remove Duplicates from Sorted List
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.
Answer:
public class deleteDuplicatesSol {
        public static ListNode deleteDuplicates(ListNode head) {
                if (head == null) {
                        return null;
                }

                ListNode dummy = new ListNode(0);
                dummy.next = head;
                ListNode cur = dummy;
                while (cur != null) {
                        if (cur.next != null && cur.next.next != null && cur.next.val ==
cur.next.next.val) {
                                cur.next = cur.next.next;
                        } else {
                                cur = cur.next;
                        }
                }
                return dummy.next;
```

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

	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		ListNode n1 = new ListNode(1);
		ListNode n2 = new ListNode(1);
		ListNode n3 = new ListNode(2);
		n1.next = n2;
		n2.next = n3;
		n3.next = null;

		ListNode tmp = new ListNode(0);
		tmp = n1;
		while (tmp != null) {
			System.out.print(tmp.val);
			tmp = tmp.next;
		}
		System.out.print("\n");
		ListNode rst = new ListNode(0);
		rst = deleteDuplicates(n1);
		while (rst != null) {
			System.out.print(rst.val);
			rst = rst.next;
		}
	}
}
/*
 * output:
 * 112
 * 12
 */
```

Remove Duplicates from Sorted List II
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.
Answer:

```
public class deleteDuplicates2Sol {
	public static ListNode deleteDuplicates1(ListNode head) {
		if (head == null) {
			return null;
		}

		ListNode dummy = new ListNode(0);
		dummy.next = head;
		ListNode cur = dummy;
		boolean del = false;
		while (cur != null) {
			if (cur.next != null
					&& cur.next.next != null
					&& cur.next.val == cur.next.next.val) {
				cur.next = cur.next.next;
				del = true;
			} else {
				if (del) {
					cur.next = cur.next.next;
					del = false;
				} else {
					cur = cur.next;
				}
			}
		}
		return dummy.next;
	}

	public static ListNode deleteDuplicates2(ListNode head) {
		if (head == null || head.next == null) {
			return head;
		}

		ListNode dummy = new ListNode(0);
		dummy.next = head;
		ListNode cur = dummy;

		while (cur != null) {
			if (cur.next != null
					&& cur.next.next != null
					&& cur.next.val == cur.next.next.val) {
				int val = cur.next.val;
```

```
                        while (cur.next != null && cur.next.val == val) {
                            cur.next = cur.next.next;
                        }
                    } else {
                        cur = cur.next;
                    }
                }
                return dummy.next;
            }

            public static class ListNode {
                int val;
                ListNode next;
                ListNode(int x) {
                    val = x;
                }
            }

            public static void main(String[] args) {
                ListNode n1 = new ListNode(1);
                ListNode n2 = new ListNode(1);
                ListNode n3 = new ListNode(1);
                ListNode n4 = new ListNode(2);
                ListNode n5 = new ListNode(3);
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n5;
                n5.next = null;

                ListNode tmp = new ListNode(0);
                tmp = n1;
                while (tmp != null) {
                    System.out.print(tmp.val);
                    tmp = tmp.next;
                }

                System.out.print("\n");

                ListNode rst1 = new ListNode(0);
                rst1 = n1;
                rst1 = deleteDuplicates1(rst1);
                while (rst1 != null) {
                    System.out.print(rst1.val);
                    rst1 = rst1.next;
                }

                System.out.print("\n");
```

```
                ListNode rst2 = new ListNode(0);
                rst2 = n1;
                rst2 = deleteDuplicates2(rst2);
                while (rst2 != null) {
                        System.out.print(rst2.val);
                        rst2 = rst2.next;
                }
        }
}
/*
 *outputs:
 *11123
 *23
 *23
 */
```

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

```
/*
变的是结点之间的相互关系
注意"指代某结点"和"next指针指向某结点"之间的区别,
ListNode A = ListNode B, 效果是用ListNode A指代ListNode B代表的结点, 方便指针滑动,
ListNode B不动, 滑动ListNode A进行具体操作,
ListNode tmp = head.next, 用ListNode tmp指代ListNode head的next指针指向的结点,
head.next = pre, 用ListNode head的next指针指向pre代表的结点,
pre = head, 用ListNode pre指代ListNode head代表的结点,
head = tmp, 用ListNode head指代ListNode tmp代表的结点,
*/
public class reverseListSol {
        public static ListNode reverseList1(ListNode head) {
                ListNode pre = null;
                while (head != null) {
                        ListNode tmp = head.next;
                        head.next = pre;
                        pre = head;
                        head = tmp;
                }
                return pre;
        }

        public static ListNode reverseList2(ListNode head) {
                if (head == null || head.next == null) {
                        return head;
                }

                ListNode rst = reverseList2(head.next);
```

```
                head.next.next = head;
                head.next = null;
                return rst;
        }

        public static class ListNode {
                int val;
                ListNode next;
                ListNode(int x) {
                        val = x;
                }
        }

        public static void main(String[] args) {
                ListNode n1 = new ListNode(1);
                ListNode n2 = new ListNode(2);
                ListNode n3 = new ListNode(3);
                n1.next = n2;
                n2.next = n3;
                n3.next = null;

                ListNode tmp = n1;
                while (tmp != null) {
                        System.out.print(tmp.val);
                        tmp = tmp.next;
                }
                System.out.print("\n");
                ListNode rst1 = n1;
                rst1 = reverseList1(rst1);
                while (rst1 != null) {
                        System.out.print(rst1.val);
                        rst1 = rst1.next;
                }
                System.out.print("\n");
                reverseList1(n3);
                ListNode rst2 = n1;
                rst2 = reverseList2(rst2);
                while (rst2 != null) {
                        System.out.print(rst2.val);
                        rst2 = rst2.next;
                }
        }
}
/*
 *outputs:
 *123
 *321
 *321
 */
```

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.
Answer:

```
public class reverseBetweenSol {
	public static ListNode reverseBetween(ListNode head, int m, int n) {
		if (head == null || head.next == null) {
			return head;
		}
		if (m >= n) {
			return head;
		}

		ListNode dummy = new ListNode(0);
		dummy.next = head;
		ListNode pre = dummy;
		for (int i = m; i > 1; i--) {
			pre = pre.next;
		}
		ListNode reverseTail = pre.next;
		pre.next = null;
		ListNode cur = reverseTail;
		for (int j = n - m + 1; j > 0; j--) {
			if (j == 1) {
				reverseTail.next = cur.next;
			}
			ListNode tmp = cur.next;
			cur.next = pre.next;
			pre.next = cur;
			cur = tmp;
		}
		return dummy.next;
	}

	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		ListNode n1 = new ListNode(1);
```

```
                ListNode n2 = new ListNode(2);
                ListNode n3 = new ListNode(3);
                ListNode n4 = new ListNode(4);
                ListNode n5 = new ListNode(5);
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n5;
                n5.next = null;

                ListNode tmp = n1;
                while (tmp != null) {
                        System.out.print(tmp.val);
                        tmp = tmp.next;
                }
                System.out.print("\n");
                ListNode rst = reverseBetween(n1, 2, 4);
                while (rst != null) {
                        System.out.print(rst.val);
                        rst = rst.next;
                }
        }
}
/*
 * output:
 * 12345
 * 14325
 */
```

Partition Array
Given an array nums of integers and an int k, partition the array (i.e move the elements in "nums") such that:
All elements < k are moved to the left
All elements >= k are moved to the right
Return the partitioning index, i.e the first index i nums[i] >= k.
Example
If nums=[3,2,2,1] and k=2, a valid answer is 1.
Note
You should do really partition in array nums instead of just counting the numbers of integers smaller than k.
If all elements in nums are smaller than k, then return nums.length
Challenge
Can you partition the array in-place and in O(n)?
Answer:

```
public class partitionArraySol {
        public static int partitionArray1(int[] nums, int k) {
                ArrayList<Integer> l = new ArrayList<Integer>();
                ArrayList<Integer> r = new ArrayList<Integer>();
                ArrayList<Integer> rst = new ArrayList<Integer>();
```

```
		int len = nums.length;
		int ans = 0;
		for (int i = 0; i < len; i++) {
			if (nums[i] < k) {
				ans++;
				l.add(nums[i]);
			} else {
				r.add(nums[i]);
			}
		}
		len = l.size();
		for (int i = 0; i < len; i++) {
			rst.add(l.get(i));
		}
		len = r.size();
		for (int i = 0; i < len; i++) {
			rst.add(r.get(i));
		}
		return ans;
	}

	public static int partitionArray2(int[] nums, int k) {
		int i = 0;
		int j = nums.length - 1;
		while (i <= j) {
			while (i <= j && nums[i] < k) {
				i++;
			}
			while (i <= j && nums[j] >= k) {
				j--;
			}
			if (i <= j) {
				int tmp = nums[i];
				nums[i] = nums[j];
				nums[j] = tmp;
				i++;
				j--;
			}
		}
		return i;
	}

	public static void main(String[] args) {
		int[] test = {3, 2, 2, 1};
		System.out.print(partitionArray1(test, 2));
		System.out.print("\n");
		System.out.print(partitionArray2(test, 2));
	}
}
```

```
/*
 * outputs:
 * 1
 * 1
 */
```

Partition List
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.
Answer:

```
public class partitionListSol {
	public static ListNode partition(ListNode head, int x) {
		if (head == null) {
			return null;
		}

		ListNode dummyLeft = new ListNode(0);
		ListNode dummyRight = new ListNode(0);
		ListNode left = dummyLeft;
		ListNode right = dummyRight;
		while (head != null) {
			if (head.val < x) {
				left.next = head;
				left = head;
			} else {
				right.next = head;
				right = head;
			}
			head = head.next;
		}
		right.next = null;
		left.next = dummyRight.next;
		return dummyLeft.next;
	}

	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		ListNode n1 = new ListNode(1);
```

```
                ListNode n2 = new ListNode(4);
                ListNode n3 = new ListNode(3);
                ListNode n4 = new ListNode(2);
                ListNode n5 = new ListNode(5);
                ListNode n6 = new ListNode(2);
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n5;
                n5.next = n6;
                n6.next = null;
                ListNode rst = partition(n1, 3);
                while (rst != null) {
                        System.out.print(rst.val);
                        rst = rst.next;
                }
        }
}
```

Sort List
Sort a linked list in O(n log n) time using constant space complexity.
Answer:

```
/*
our question is to sort whole list, different with binary search, which is to find one target in O(log
n) time,
       c if n = 1,
T(n) = {
       2T(n / 2) + cn if n > 1,
T(n) = 2T(n / 2) + cn
    = 2(2T(n / 4) + cn / 2) + cn = 4(T / 4) + 2cn
    ...
    = nT(1) + logn * cn
that is, O(n log n)
为什么不用QuickSort? 因为对于链表随机访问太耗时, 而heap sort不可行
*/
/*
 0 1 2 3
 2 8 1 5
 pivot = 2
 i = 0 j = 3
 <- j--
 0 1 2 3
 1 8 2 5
 i = 0 j = 2
 -> i++
 0 1 2 3
 1 2 8 5
 i = 1 j = 2 at this time j = i + 1, move on
 {1} 2 {8 5}
```

pivot = 1
...
pivot = 8
...
{1} 2 {5 8}
快速排序是找出一个元素(理论上可以随便找一个)作为基准(pivot), 然后对数组进行分区操作, 使基准左边元素的值都不大于基准值, 基准右边的元素值都不小于基准值, 如此作为基准的元素调整到排序后的正确位置,
最优情况下, Partition每次都划分得很均匀, 如果排序n个关键字, 其递归树的深度就为logn + 1, 即仅需递归logn次, 需要时间为T(n)的话, 第一次Partiation应该是需要对整个数组扫描一遍, 做n次比较。然后, 获得的枢轴将数组一分为二, 那么各自还需要T(n / 2)的时间(注意是最好情况，所以平分两半)。于是不断地划分下去, 我们就有了下面的不等式推断,
T(1) = 0,
T(n) <= 2T(n / 2) + n
T(n) <= 2(2T(n / 4) + n / 2) + n = 4T(n / 4) + 2n
...
T(n) <= nT(1) + logn * n = O(n log n)
最坏情况是每次划分选取的基准都是当前无序区中关键字最小(或最大)的记录, 划分的结果是基准左边的子区间为空(或右边的子区间为空), 而划分所得的另一个非空的子区间中记录数目, 仅仅比划分前的无序区中记录个数减少一个。时间复杂度为O(n^2)
需要执行n - 1次递归调用, 且第i次划分需要经过n - i次关键字的比较才能找到第i个记录, 也就是枢轴的位置, 比较次数
(n - 1) + (n - 2) + ... + 1 = n * (n - 1) / 2
关键字的比较和交换是跳跃进行的, 快速排序是一种不稳定的排序方法
*/

```
public class sortListSol {
	// merge sort
	public static ListNode sortList1(ListNode head) {
		if (head == null || head.next == null) {
			return head;
		}

		ListNode mid = findMiddle(head);
		ListNode right = sortList1(mid.next);
		mid.next = null;
		ListNode left = sortList1(head);
		return merge(left, right);
	}

	private static ListNode findMiddle(ListNode head) {
		ListNode slow = head;
		ListNode fast = head.next;
		while (fast != null && fast.next != null) {
			fast = fast.next.next;
			slow = slow.next;
		}
		return slow;
	}
```

```
	private static ListNode findMiddle2(ListNode head) {
		ListNode slow = head;
		ListNode fast = head;
		while (fast != null && fast.next != null && fast.next.next != null) {
			slow = slow.next;
			fast = fast.next.next;
		}
		return slow;
	}

	private static ListNode merge(ListNode head1, ListNode head2) {
		ListNode dummy = new ListNode(0);
		ListNode tail = dummy;
		while (head1 != null && head2 != null) {
			if (head1.val < head2.val) {
				tail.next = head1;
				head1 = head1.next;
			} else {
				tail.next = head2;
				head2 = head2.next;
			}
			tail = tail.next;
		}
		if (head1 != null) {
			tail.next = head1;
		} else {
			tail.next = head2;
		}
		return dummy.next;
	}
	// quick sort
	public static ListNode sortList2(ListNode head) {
		if (head == null) {
			return null;
		}

		return quickSort(head);
	}

	public static boolean isDuplicate(ListNode head) {
		while (head != null) {
			if (head.next != null && head.next.val != head.val) {
				return false;
			}
			head = head.next;
		}
		return true;
	}
```

```
public static ListNode quickSort(ListNode head) {
        if (head == null) {
                return null;
        }

        if (isDuplicate(head)) {
                return head;
        }

        ListNode headNew = partition(head, head.val);
        ListNode cur = headNew;
        ListNode dummy = new ListNode(0);
        dummy.next = headNew;
        ListNode pre = dummy;
        while (cur != null) {
                if (cur.val == head.val) {
                        break;
                }
                cur = cur.next;
                pre = pre.next;
        }
        pre.next = null;
        ListNode left = dummy.next;
        ListNode right = cur.next;
        cur.next = null;
        left = quickSort(left);
        right = quickSort(right);
        if (left != null) {
                dummy.next = left;
                while (left.next != null) {
                        left = left.next;
                }
                left.next = cur;
        } else {
                dummy.next = cur;
        }
        cur.next = right;
        return dummy.next;
}

public static ListNode partition(ListNode head, int x) {
        if (head == null) {
                return null;
        }
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        ListNode cur = head;
        ListNode bigDummy = new ListNode(0);
```

```
                ListNode bigTail = bigDummy;
                while (cur != null) {
                        if (cur.val >= x) {
                                pre.next = cur.next;
                                bigTail.next = cur;
                                cur.next = null;
                                bigTail = cur;
                        } else {
                                pre = pre.next;
                        }
                        cur = pre.next;
                }
                pre.next = bigDummy.next;
                return dummy.next;
        }

        public static class ListNode {
                int val;
                ListNode next;
                ListNode(int x) {
                        val = x;
                }
        }

        public static void main(String[] args) {
                ListNode n1 = new ListNode(2);
                ListNode n2 = new ListNode(8);
                ListNode n3 = new ListNode(1);
                ListNode n4 = new ListNode(5);
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = null;
                ListNode rst = sortList1(n1);
                while (rst != null) {
                        System.out.print(rst.val);
                        rst = rst.next;
                }
        }
}
```

Reorder List
Given a singly linked list L: L0→L1→…→Ln-1→Ln,
reorder it to: L0→Ln→L1→Ln-1→L2→Ln-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}.
Answer:

```
/*
4 STEP:
1. find the mid.
2. cut the list to two list.
3. REVERSE the right side.
4. MERGE the two list.
*/
public class reorderListSol {
	public static void reorderList(ListNode head) {
		if (head == null) {
			return;
		} else if (head.next == null) {
			return;
		}

		ListNode pre = findMidPre(head);
		ListNode right = pre.next;
		pre.next = null;

		right = reverse(right);
		merge(head, right);
	}

	public static ListNode findMidPre(ListNode head) {
		if (head == null) {
			return null;
		}
		ListNode slow = head;
		ListNode fast = head.next;
		while (fast != null && fast.next != null) {
			slow = slow.next;
			fast = fast.next.next;
		}
		return slow;
	}

	public static ListNode reverse(ListNode head) {
		if (head == null) {
			return null;
		}
		ListNode dummy = new ListNode(0);
		while (head != null) {
			ListNode tmp = head.next;
			head.next = dummy.next;
			dummy.next = head;
			head = tmp;
		}
		return dummy.next;
	}
```

```
	public static void merge(ListNode head1, ListNode head2) {
		ListNode dummy = new ListNode(0);
		ListNode cur = dummy;
		while (head1 != null && head2 != null) {
			cur.next = head1;
			cur = cur.next;
			head1 = head1.next;
			cur.next = head2;
			cur = cur.next;
			head2 = head2.next;
		}
		if (head1 != null) {
			cur.next = head1;
		} else {
			cur.next = head2;
		}
	}

	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
		}
	}

	public static void main(String[] args) {
		ListNode n1 = new ListNode(1);
		ListNode n2 = new ListNode(2);
		ListNode n3 = new ListNode(3);
		ListNode n4 = new ListNode(4);
		n1.next = n2;
		n2.next = n3;
		n3.next = n4;
		n4.next = null;
		reorderList(n1);
		while (n1 != null) {
			System.out.print(n1.val);
			n1 = n1.next;
		}
	}
}
```

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

```
public class hasCycleSol {
	public static boolean hasCycle1(ListNode head) {
		if (head == null) {
			return false;
		}

		ListNode slow = head;
		ListNode fast = head;
		while (fast != null && fast.next != null) {
			slow = slow.next;
			fast = fast.next.next;
			if (slow == fast) {
				return true;
			}
		}
		return false;
	}

	public static boolean hasCycle2(ListNode head) {
		if (head == null || head.next == null) {
			return false;
		}

		ListNode slow = head;
		ListNode fast = head.next;
		while (fast != null && fast.next != null) {
			slow = slow.next;
			fast = fast.next.next;
			if (slow == fast) {
				return true;
			}
		}
		return false;
	}

	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
			next = null;
		}
	}

	public static void main(String[] args) {
		ListNode n1 = new ListNode(1);
		ListNode n2 = new ListNode(2);
		ListNode n3 = new ListNode(3);
		ListNode n4 = new ListNode(4);
```

```
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n2;
                System.out.print(hasCycle1(n1));
                System.out.print(hasCycle2(n1));
        }
}
```

Linked List Cycle II
Given a linked list, return the node where the cycle begins. If there is no cycle, return null.
Follow up:
Can you solve it without using extra space?
Answer:

```
/*
现在有两个指针, 第一个指针, 每走一次走一步, 第二个指针每走一次走两步, 如果他们走了t次之后相遇在K点
指针一 走的路是 t = X + nY + K
指针二 走的路是 2t = X + mY + K
把等式一代入到等式二中, 有
X + K = (m - 2n)Y
X和K的关系是基于Y互补的。等于说, 两个指针相遇以后, 再往下走X步就回到Cycle的起点了。这就可以有O(n)的实现了。
*/
public class detectCycleSol {
        public static ListNode detectCycle(ListNode head) {
                if (head == null) {
                        return null;
                }

                ListNode s = head;
                ListNode f = head;
                ListNode cross = null;
                while (f != null && f.next != null) {
                        s = s.next;
                        f = f.next.next;
                        if (s == f) {
                                cross = s;
                                break;
                        }
                }

                if (cross == null) {
                        return null;
                }

                s = head;
                while (true) {
                        if (s == f) {
```

```
                        return s;
                    }
                    s = s.next;
                    f = f.next;
                }
        }

        public static class ListNode {
                int val;
                ListNode next;
                ListNode(int x) {
                        val = x;
                        next = null;
                }
        }

        public static void main(String[] args) {
                ListNode n1 = new ListNode(1);
                ListNode n2 = new ListNode(2);
                ListNode n3 = new ListNode(3);
                ListNode n4 = new ListNode(4);
                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n2;
                System.out.print(detectCycle(n1).val);
        }
}
```

Merge k Sorted Lists
Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.
Answer:
/*
这道题目在分布式系统中非常常见, 来自不同client的sorted list要在central server上面merge起来
SOL 1 Divide Conquer
先把k个list分成两半, 然后继续划分, 直到剩下两个list就合并起来, 合并时会用到Merge Two Sorted Lists这道题,
假设总共有k个list, 每个list的最大长度是n,
T(k) = 2T(k / 2) + O(nk)
= 2(2T(k / 2^2) + O(nk / 2)) + O(nk) = 4T(k / 4) + 2O(nk)
= 4(2T(k / 2^3) + O(nk / 2^2)) + 2O(nk) = 8T(k / 2^3) + 3O(nk)
...
= O(nklogk)
空间复杂度的话是递归栈的大小O(logk).
SOL 2, SOL 3,
维护一个大小为k的堆, 每次取堆顶的最小元素放到结果中, 然后读取该元素的下一个元素放入堆中, 重新维护好。因为每个链表是有序的, 每次又是去当前k个元素中最小的, 所以当所有链表都读完时结束, 这个时候所有元素按从小到大放在结果链表中。

这个算法每个元素要读取一次, 即是k*n次, 然后每次读取元素要把新元素插入堆中要logk的复杂度, 所以总时间复杂度是O(nklogk)。
空间复杂度是堆的大小, 即为O(k).
note: “空间复杂度”指占内存大小, “堆排序”每次只对一个元素操作, 是就地排序, 所用辅助空间O(1), 注意和本题区别
*/

```
public class mergeKListsSol {
	// SOL 1
	public static ListNode mergeKLists1(List<ListNode> lists) {
		if (lists == null || lists.size() == 0) {
			return null;
		}
		return helper1(lists, 0, lists.size() - 1);
	}

	public static ListNode helper1(List<ListNode> lists, int l, int r) {
		if (l < r) {
			int mid = l + (r - 1) / 2;
			return merge(helper1(lists, l, mid), helper1(lists, mid + 1, r));
		}
		return lists.get(l);
	}

	public static ListNode merge(ListNode n1, ListNode n2) {
		ListNode dummy = new ListNode(0);
		ListNode cur = dummy;
		while (n1 != null && n2 != null) {
			if (n1.val < n2.val) {
				cur.next = n1;
				n1 = n1.next;
			} else {
				cur.next = n2;
				n2 = n2.next;
			}
			cur = cur.next;
		}
		if (n1 != null) {
			cur.next = n1;
		} else {
			cur.next = n2;
		}
		return dummy.next;
	}
	// SOL 2 min heap
	public static ListNode mergeKLists2(List<ListNode> lists) {
		if (lists == null || lists.size() == 0) {
			return null;
		}
```

```
		Queue<ListNode> heap2 = new PriorityQueue<ListNode>(lists.size(),
Comparator2);
		for (int i = 0; i < lists.size(); i++) {
			if (lists.get(i) != null) {
				heap2.add(lists.get(i));
			}
		}
		ListNode dummy = new ListNode(0);
		ListNode tail = dummy;
		while (!heap2.isEmpty()) {
			ListNode head = heap2.poll();
			tail.next = head;
			tail = head;
			if (head.next != null) {
				heap2.add(head.next);
			}
		}
		return dummy.next;
	}

	private static Comparator<ListNode> Comparator2 = new Comparator<ListNode>() {
		public int compare(ListNode left, ListNode right) {
			if (left == null) {
				return 1;
			} else if (right == null) {
				return -1;
			}
			return left.val - right.val;
		}
	};
	// SOL 3 min heap
	public static ListNode mergeKLists3(List<ListNode> lists) {
		if (lists == null || lists.size() == 0) {
			return null;
		}
		int size = lists.size();
		PriorityQueue<ListNode> heap3 = new PriorityQueue<ListNode>(size,
				new Comparator<ListNode>() {
			public int compare(ListNode o1, ListNode o2) {
				return o1.val - o2.val;
			}
		});

		for (ListNode node : lists) {
			if (node != null) {
				heap3.offer(node);
			}
		}
		ListNode dummy = new ListNode(0);
```

```
        ListNode tail = dummy;
        while (!heap3.isEmpty()) {
            ListNode cur = heap3.poll();
            tail.next = cur;
            tail = tail.next;
            if (cur.next != null) {
                heap3.offer(cur.next);
            }
        }
        return dummy.next;
    }

    public static class ListNode {
        int val;
        ListNode next;
        ListNode(int x) {
            val = x;
        }
    }

    public static void main(String[] args) {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(2);
        ListNode n5 = new ListNode(4);
        ListNode n6 = new ListNode(6);
        ListNode n7 = new ListNode(3);
        ListNode n8 = new ListNode(5);
        ListNode n9 = new ListNode(7);
        n1.next = n2;
        n2.next = n3;
        n3.next = null;
        n4.next = n5;
        n5.next = n6;
        n6.next = null;
        n7.next = n8;
        n8.next = n9;
        n9.next = null;
        List<ListNode> lists = new ArrayList<ListNode>();
        lists.add(n1);
        lists.add(n4);
        lists.add(n7);
        for (int i = 0; i < lists.size(); i++) {
            ListNode tmp = lists.get(i);
            while (tmp != null) {
                System.out.print(tmp.val);
                tmp = tmp.next;
            }
```

```
                System.out.print("\n");
            }
            ListNode rst1 = mergeKLists3(lists);
            while (rst1 != null) {
                System.out.print(rst1.val);
                rst1 = rst1.next;
            }
        }
}
```

Merge Two Sorted Lists
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.
Answer:

```
public class mergeTwoListsSol {
    public static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                cur.next = l1;
                l1 = l1.next;
            } else {
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        if (l1 != null) {
            cur.next = l1;
        } else {
            cur.next = l2;
        }
        return dummy.next;
    }

    public static class ListNode {
        int val;
        ListNode next;
        ListNode(int x) {
            val = x;
        }
    }

    public static void main(String[] args) {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(2);
```

```
                ListNode n5 = new ListNode(4);
                ListNode n6 = new ListNode(6);
                n1.next = n2;
                n2.next = n3;
                n3.next = null;
                n4.next = n5;
                n5.next = n6;
                n6.next = null;
                ListNode rst = mergeTwoLists(n1, n4);
                while (rst != null) {
                        System.out.print(rst.val);
                        rst = rst.next;
                }
        }
}
```

Copy List with Random Pointer
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.
Answer:

```
/*
SOL 2
head1 -> newNode1 -> head2 -> newNode2 -> head3 -> newNode3 -> ... -> Random1 -> Random1Copy -> ...
before copyNext, due to "newNode.random = head.random", so head1.random points to Random1 and newNode1.random points to Random1,
after copyNext, we need head1.random points to Random1, newNode1 points to Random1Copy, notice that Random1.next = Random1Copy, so "head.next.random = head.random.next".
*/
public class copyRandomListSol {
        // SOL 1
        public static RandomListNode copyRandomList1(RandomListNode head) {
                if (head == null) {
                        return null;
                }
                HashMap<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();
                RandomListNode dummy = new RandomListNode(0);
                RandomListNode pre = dummy;
                RandomListNode newNode;
                while (head != null) {
                        if (map.containsKey(head)) {
                                newNode = map.get(head);
                        } else {
                                newNode = new RandomListNode(head.label);
                                map.put(head, newNode);
                        }
```

```
                        pre.next = newNode;

                        if (head.random != null) {
                                if (map.containsKey(head.random)) {
                                        newNode.random = map.get(head.random);
                                } else {
                                        newNode.random = new
RandomListNode(head.random.label);
                                        map.put(head.random, newNode.random);
                                }
                        }

                        pre = newNode;
                        head = head.next;
                }
                return dummy.next;
        }
        // SOL 2
        public static RandomListNode copyRandomList2(RandomListNode head) {
                if (head == null) {
                        return null;
                }
                copyNext(head);
                copyRandom(head);
                return splitList(head);
        }

        private static void copyNext(RandomListNode head) {
                while (head != null) {
                        RandomListNode newNode = new RandomListNode(head.label);
                        newNode.random = head.random;
                        newNode.next = head.next;
                        head.next = newNode;
                        head = head.next.next;
                }
        }

        private static void copyRandom(RandomListNode head) {
                while (head != null) {
                        if (head.next.random != null) {
                                head.next.random = head.random.next;
                        }
                        head = head.next.next;
                }
        }

        private static RandomListNode splitList(RandomListNode head) {
                RandomListNode newHead = head.next;
```

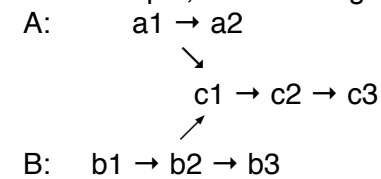
```
                while (head != null) {
                        RandomListNode tmp = head.next;
                        head.next = tmp.next;
                    head = head.next;
                    if (tmp.next != null) {
                        tmp.next = tmp.next.next;
                    }
                }
                return newHead;
        }

        public static class RandomListNode {
                int label;
                RandomListNode next, random;
                RandomListNode(int x) {
                        this.label = x;
                }
        }

        public static void main(String[] args) {
                RandomListNode node = new RandomListNode(-1);
                RandomListNode copy = copyRandomList1(node);
        }
}

Convert Sorted List to Binary Search Tree
Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.
*/
/*
SOL 1
这个方法比较暴力, 每次遍历当前list, 找到中间的节点, 建立root, 分别使用递归建立左树以及右树, 并将左右树挂在root之下.
但这个算法会复杂度很高. 建立root次数为N, 每次遍历最多N次, 最坏为N平方(实际不会这么多)
SOL 2
类似SOL 3, Java不能使用指针, 所以我们自建一个自定义的类, 里面只有一个ListNode, 这样我们就能方便地修改入参了.
SOL 3
这个解法使用一个instance variable来记录当前正在操作的List Node. DFS本身的效果是, 从head直到尾部建树, 并且将currNode移动到size+1处. 这样可以在1次iterator我们的List后直接建立树.
这是一种Bottom-up的建树方法. 如果我们使用C++, 则可以将List Node直接做为入参来改变之而不需要使用实例变量.
问题是: 我们如果可以的话, 尽量不要使用实例变量, 因为它是各个Method共享的, 所以这个方法存在风险, 因为变量有可能被别的方法修改.
SOL2, SOL3 time both are O(nlogn)
*/
public class sortedListToBSTSol {
        // SOL 1
        public static TreeNode sortedListToBST1(ListNode head) {
```

```
        if (head == null) {
            return null;
        }

        ListNode fast = head;
        ListNode slow = head;
        ListNode pre = head;

        TreeNode root = null;
        if (head.next == null) {
            root = new TreeNode(head.val);
            root.left = null;
            root.right = null;
            return root;
        }

        while (fast != null && fast.next != null) {
            pre = slow;
            fast = fast.next.next;
            slow = slow.next;
        }

        pre.next = null;
        TreeNode left = sortedListToBST1(head);
        TreeNode right = sortedListToBST1(slow.next);
        root = new TreeNode(slow.val);
        root.left = left;
        root.right = right;
        return root;
    }
    // SOL 2
    public static TreeNode sortedListToBST2(ListNode head) {
        if (head == null) {
            return null;
        }

        int size = 0;
        ListNode cur = head;
        while (cur != null) {
            size++;
            cur = cur.next;
        }

        CurNode curNode = new CurNode(head);
        return helper2(curNode, size);
    }

    public static class CurNode {
        ListNode node;
```

```
        CurNode(ListNode node) {
            this.node = node;
        }
    }

    public static TreeNode helper2(CurNode cur, int size) {
        if (size <= 0) {
            return null;
        }
        TreeNode left = helper2(cur, size / 2);
        TreeNode root = new TreeNode(cur.node.val);
        cur.node = cur.node.next;
        TreeNode right = helper2(cur, size - 1 - size / 2);
        root.left = left;
        root.right = right;
        return root;
    }
    // SOL 3
    public static TreeNode sortedListToBST3(ListNode head) {
        int size;
        cur = head;
        size = getListLength(head);
        return helper3(size);
    }

    private static ListNode cur;

    private static int getListLength(ListNode head) {
        int size = 0;
        while (head != null) {
            size++;
            head = head.next;
        }
        return size;
    }

    public static TreeNode helper3(int size) {
        if (size <= 0) {
            return null;
        }

        TreeNode left = helper3(size / 2);
        TreeNode root = new TreeNode(cur.val);
        cur = cur.next;
        TreeNode right = helper3(size - 1 - size / 2);
        root.left = left;
        root.right = right;
        return root;
    }
```

```
	public static class ListNode {
		int val;
		ListNode next;
		ListNode(int x) {
			val = x;
		}
	}

	public static class TreeNode {
		int val;
		TreeNode left;
		TreeNode right;
		TreeNode(int x) {
			val = x;
		}
	}
}
```

Intersection of Two Linked Lists
Write a program to find the node at which the intersection of two singly linked lists begins.
For example, the following two linked lists:



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.
Answer:

```
/*
SOL 1
1. 得到2个链条的长度.
2. 将长的链条向前移动差值(len1 - len2)
3. 两个指针一起前进, 遇到相同的即是交点, 如果没找到, 返回null.
相当直观的解法. 空间复杂度O(1), 时间复杂度O(m+n)
*/
public class getIntersectionNodeSol {
	// SOL 1
	public static ListNode getIntersectionNode1(ListNode headA, ListNode headB) {
		if (headA == null || headB == null) {
			return null;
		}

		int len1 = getLen(headA);
```

```
        int len2 = getLen(headB);
        int cnt = Math.abs(len1 - len2);

        if (len1 > len2) {
            while (cnt > 0) {
                headA = headA.next;
                cnt--;
            }
        } else {
            while (cnt > 0) {
                headB = headB.next;
                cnt--;
            }
        }

        while (headA != null) {
            if (headA == headB) {
                return headA;
            }
            headA = headA.next;
            headB = headB.next;
        }
        return null;
    }

    public static int getLen(ListNode head) {
        int cnt = 0;
        while (head != null) {
            cnt++;
            head = head.next;
        }
        return cnt;
    }
    // SOL 2
    public static ListNode getIntersectionNode2(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode pA = headA;
        ListNode pB = headB;
        ListNode tailA = null;
        ListNode tailB = null;
        while (true) {
            if (pA == null) {
                pA = headB;
            }
            if (pB == null) {
                pB = headA;
            }
```

```
                        if (pA.next == null) {
                                tailA = pA;
                        }
                        if (pB.next == null) {
                                tailB = pB;
                        }
                        if (tailA != null && tailB != null && tailA != tailB) {
                                return null;
                        }
                        if (pA == pB) {
                                return pA;
                        }
                        pA = pA.next;
                        pB = pB.next;
                }
        }

        public static class ListNode {
                int val;
                ListNode next;
                ListNode(int x) {
                        val = x;
                }
        }

        public static void main(String[] args) {
                ListNode n1 = new ListNode(1);
                ListNode n2 = new ListNode(2);
                ListNode n3 = new ListNode(3);
                ListNode n4 = new ListNode(4);
                ListNode n5 = new ListNode(5);
                n1.next = n2;
                n2.next = n4;
                n4.next = n5;
                n5.next = null;
                n3.next = n4;
                System.out.print(getIntersectionNode2(n1, n3).val);
        }
}
```

Remove Linked List Elements
Remove all elements from a linked list of integers that have value val.
Example
Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6
Return: 1 --> 2 --> 3 --> 4 --> 5
Answer:

```
/**
 * Definition for singly-linked list.
```

```
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode cur = dummy;
        while (head != null) {
            if (head.val == val) {
                cur.next = head.next;
                head = head.next;
            } else {
                cur = head;
                head = head.next;
            }
        }
        return dummy.next;
    }
}
```

Insertion Sort List
Sort a linked list using insertion sort.
Answer:

```
/*
1. 从第一个元素开始, 该元素可以认为已经被排序
2. 取出下一个元素, 在已经排序的元素序列中从后向前扫描
3. 如果该元素(已排序)大于新元素, 将该元素移到下一位置
4. 重复步骤3, 直到找到已排序的元素小于或者等于新元素的位置
5. 将新元素插入到该位置后
6. 重复步骤2~5
time O(N^2), space O(1)
*/
/*
dummy -> null, 1 -> 2 -> 6 -> 5
pre           head
                tmp
       pre -> 1 -> null _____________
                head |          |
                ...  v          |
       pre -> 1 -> 2 -> 6 -> null, 5 -> null
                                head tmp
                pre              ^
                |__________________|    head
*/
```

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(0);
        while (head != null) {
            ListNode pre = dummy;
            while (pre.next != null && pre.next.val <= head.val) {
                pre = pre.next;
            }
            ListNode tmp = head.next;
            head.next = pre.next;
            pre.next = head;
            head = tmp;
        }
        return dummy.next;
    }
}
```

Rotate List
Given a list, rotate the list to the right by k places, where k is non-negative.
For example:
Given 1->2->3->4->5->NULL and k = 2,
return 4->5->1->2->3->NULL.
Answer:

```
/*
重点就是, 要先变成循环链表, end.next = head, 再执行ListNode headNew = pre.next, 否则, 当k = 0的时候, 会返回一个null指针, 因为pre是在最后的,
Rotate的精髓是旋转, 也就是说当n=0的时候, 应该什么也不做, 那么pre的下一个应该是头节点. 所以我们应该把end.next = head,
另外的做法, 就是把n = 0单独拿出来, 当n = 0直接returen head. 这样子就不用考虑这种特殊情况了. pre.next就一定不会是null.
*/
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
```

```
public class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null) {
            return head;
        }
        int len = getLen(head);
        k = k % len;

        ListNode end = head;
        while (k > 0) {
            end = end.next;
            k--;
        }
        ListNode pre = head;
        while (end.next != null) {
            pre = pre.next;
            end = end.next;
        }
        end.next = head;
        ListNode headNew = pre.next;
        pre.next = null;
        return headNew;
    }

    public int getLen(ListNode head) {
        int cnt = 0;
        while (head != null) {
            head = head.next;
            cnt++;
        }
        return cnt;
    }
}
```

Remove Nth Node From End of List
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.
Answer:

```
/*
使用快慢指针, 快指针先行移动n步. 用慢指针指向要移除的Node的前一个Node.
*/
/**
 * Definition for singly-linked list.
```

by Eamon 05/23/2015

```
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode slow = dummy;
        ListNode fast = dummy;
        while (n > 0) {
            fast = fast.next;
            n--;
        }
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return dummy.next;
    }
}
```

Figures: