CHAPTER 1
1.1
Answer:

```
/*
1. unicode or ascii?
*/
/*
ASCII码的取值范围是0~127, 可以用7个bit表示, char型变量的大小规定为一字节, 如果存放ASCII码则只用到低7位, 高位为0,
绝大多数计算机的一个字节是8位, 取值范围是0~255, 而ASCII码并没有规定编号为128~255的字符,
为了能表示更多字符, 各厂商制定了很多种ASCII码的扩展规范. 注意, 虽然通常把这些规范称为扩展ASCII码(Extended ASCII), 但其实它们并不属于ASCII码标准.
例如扩展ASCII码由IBM制定, 在字符终端下被广泛采用, 其中包含了很多表格边线字符用来画界面
Thus, there are in fact 256 possible unique characters for one char.
带符号的char类型取值范围是-128~127(01111111)
计算机里面所有数都是用补码表示, 负数补码是原码的反码加1
-127 (11111111) -> 10000000 -> 10000001
任何一一个原码都不可能在转成补码时变成10000000, 人为规定-128
Unicode defines (less than) 221 characters, which, similarly, map to numbers 0–221 (though not all numbers are currently assigned, and some are reserved).
Unicode is a superset of ASCII. Because Unicode characters don't generally fit into one 8-bit byte, there are numerous ways of storing Unicode characters in byte sequences, such as UTF-32 and UTF-8.
*/
public class UniCharSol {
	// time O(n) space O(1)
	public static boolean isUniqueChars1(String str) {
		if (str.length() > 256) {
			return false;
		}

		boolean[] char_set = new boolean[256];
		for (int i = 0; i < str.length(); i++) {
			int val = str.charAt(i);
			if (char_set[val]) {
				return false;
			}
			char_set[val] = true;
		}
		return true;
	}
	// time O(n) space more reduced
	public static boolean isUniqueChars2(String str) {
		int checker = 0;
		for (int i = 0; i < str.length(); i++) {
			int val = str.charAt(i) - 'a';
			if ((checker & (1 << val)) > 0) {
				return false;
```

```
                        }
                        checker |= (1 << val);
                }
                return true;
    }

        public static void main(String[] args) {
                String test = "I love Wuhan City 我爱武汉";
                System.out.print(isUniqueChars2(test));
        }
}
```

1.2
Answer:

```
/*
Java strings are not terminated with a null characters as in C or C++. Although java strings uses
internally the char array but there is no terminating null in that. String class provides a method
called length to know the number of characters in the string.
*/
public class ReverseStrSol {
        public static String reverse1(String str) {
                int len = str.length();
                char[] list = new char[len];
                int start = 0;
                int end = len - 1;

                for (int i = 0; i < len; i++) {
                        list[i] = str.charAt(i);
                }

                while (start < end) {
                        char tmp = list[start];
                        list[start] = list[end];
                        list[end] = tmp;
                        start++;
                        end--;
                }

                return Arrays.toString(list);
        }

        public static String reverse2(String str) {
                int len = str.length();
                ArrayList<Character> list = new ArrayList<Character>();

                for (int i = 0; i < len; i++) {
                        list.add(str.charAt(i));
                }
```

```
                Collections.reverse(list);
                return list.toString();
        }

        public static void main(String[] args) {
                String test = "Jun";
                System.out.println(reverse2(test));
        }
}
/*
 * output:
 * [n, u, J]
 */
```

1.3
Answer:

```
/*
1. case sensitive or not? God vs dog?
2. whitespace is significant or not?
assume yes
SOL 2
assume the character set was ascii
*/
public class PermutationJudgeSol {
        public static boolean permutation1(String s, String t) {
                if (s.length() != t.length()) {
                        return false;
                }

                return sort(s).equals(sort(t));
        }

        private static String sort(String s) {
                char[] content = s.toCharArray();
                Arrays.sort(content);
                return new String(content);
        }

        public static boolean permutation2(String s, String t) {
                if (s.length() != t.length()) {
                        return false;
                }

                int[] letters = new int[256];
                char[] s_array = s.toCharArray();
                for (char c : s_array) {
                        letters[c]++;
                }
```

```
		for (int i = 0; i < t.length(); i++) {
			int c = (int) t.charAt(i);
			if (--letters[c] < 0) {
				return false;
			}
		}
		return true;
	}

	public static void main(String[] args) {
		System.out.print(permutation2("God ", "dog"));
	}
}
```

1.4
Answer:

```
public class ReplaceSpaceSol {
	public static String replacesSpaces(char[] str, int length) {
		int spaceCount = 0;
		int newLength, i;

		for (i = 0; i < length; i++) {
			if (str[i] == ' ') {
				spaceCount++;
			}
		}

		newLength = length + spaceCount * 2;
		str[newLength] = '\0';
		for (i = length - 1; i >= 0; i--) {
			if (str[i] == ' ') {
				str[newLength - 1] = '0';
				str[newLength - 2] = '2';
				str[newLength - 3] = '%';
				newLength = newLength - 3;
			} else {
				str[newLength - 1] = str[i];
				newLength--;
			}
		}
		return Arrays.toString(str);
	}

	public static void main(String[] args) {
		String str = "um a s s";
		char[] arr = new char[str.length() + 3 * 2 + 1];
		for (int i = 0; i < str.length(); i++) {
			arr[i] = str.charAt(i);
		}
```

```
                System.out.print(replacesSpaces(arr, str.length()));
        }
}
/*
 * outputs:
 * [u, m, %, 2, 0, a, %, 2, 0, s, %, 2, 0, s, ]
 */
```

1.5
Answer:

```
public class StrCompressSol {
        // O(p + k^2) time
        // p is the size of the original string
        // k is the number of character sequences
        public static String compressBad(String str) {
                String mystr = "";
                char last = str.charAt(0);
                int count = 1;
                for (int i = 1; i < str.length(); i++) {
                        if (str.charAt(i) == last) {
                                count++;
                        } else {
                                mystr += last + "" + count;
                                last = str.charAt(i);
                                count = 1;
                        }
                }
                return mystr + last + count;
        }
        // O(N) time, O(N) space
        public static String compressBetter(String str) {
                int size = countCompression(str);
                if (size >= str.length()) {
                        return str;
                }

                StringBuffer mystr = new StringBuffer();
                char last = str.charAt(0);
                int count = 1;
                for (int i = 1; i < str.length(); i++) {
                        if (str.charAt(i) == last) {
                                count++;
                        } else {
                                mystr.append(last);
                                mystr.append(count);
                                last = str.charAt(i);
                                count = 1;
                        }
                }
```

```
            mystr.append(last);
            mystr.append(count);
            return mystr.toString();
    }

    private static int countCompression(String str) {
            if (str == null || str.isEmpty()) return 0;
            char last = str.charAt(0);
            int size = 0;
            int count = 1;
            for (int i = 1; i < str.length(); i++) {
                    if (str.charAt(i) == last) {
                            count++;
                    } else {
                            last = str.charAt(i);
                            size += 1 + String.valueOf(count).length();
                            count = 1;
                    }
            }
            size += 1 + String.valueOf(count).length();
            return size;
    }
    // O(N) time, O(N) space
    public static String compressAlternate(String str) {
            int size = countCompression(str);
            if (size >= str.length()) {
                    return str;
            }

            char[] array = new char[size];
            int index = 0;
            char last = str.charAt(0);
            int count = 1;
            for (int i = 1; i < str.length(); i++) {
                    if (str.charAt(i) == last) {
                            count++;
                    } else {
                            index = setChar(array, last, index, count);
                            last = str.charAt(i);
                            count = 1;
                    }
            }
            index = setChar(array, last, index, count);
            return String.valueOf(array);
    }

    private static int setChar(char[] array, char c, int index, int count) {
            array[index] = c;
            index++;
```

```
                char[] cnt = String.valueOf(count).toCharArray();
                for (char x : cnt) {
                        array[index] = x;
                        index++;
                }
                return index;
        }

        public static void main(String[] args) {
                String str = "aabca";
                System.out.print(compressAlternate(str));
        }
}
```

1.6
Answer:

```
/*
In other contexts there is a difference between ++i and i++, but not for loops.
*/
public class RotateSol {
        // O(N^2) time
        // matrix[i][j] i: rows, j: columns
        public static void rotate(int[][] matrix, int n) {
                for (int layer = 0; layer < n / 2; layer++) {
                        int first = layer;
                        int last = n - 1 - layer;

                        for (int i = first; i < last; i++) {
                                int offset = i - first;

                                int top = matrix[first][i];

                                matrix[first][i] = matrix[last - offset][first];

                                matrix[last - offset][first] = matrix[last][last - offset];

                                matrix[last][last - offset] = matrix[i][last];

                                matrix[i][last] = top;
                        }
                }
        }

        public static void main(String[] args) {

                int[][] matrix = AssortedMethods.randomMatrix(4, 4, 0, 3);
                AssortedMethods.printMatrix(matrix);
                rotate(matrix, 4);
                System.out.println();
```

```
                AssortedMethods.printMatrix(matrix);

                System.out.println();
                int[][] test = {{4, 2, 1, 3}, {1, 3, 2, 4}, {3, 1, 2, 4}, {2, 4, 1, 3}};
                for (int i = 0; i < test.length; i++) {
                        for (int j = 0; j < test[i].length; j++) {
                                if (test[i][j] < 10 && test[i][j] > -10) {
                                        System.out.print(" ");
                                }
                                if (test[i][j] < 100 && test[i][j] > -100) {
                                        System.out.print(" ");
                                }
                                if (test[i][j] >= 0) {
                                        System.out.print(" ");
                                }
                                System.out.print(" " + test[i][j]);
                        }
                   System.out.println();
                }
                rotate(test, 4);
                System.out.println();
                for (int i = 0; i < test.length; i++) {
                        for (int j = 0; j < test[i].length; j++) {
                                if (test[i][j] < 10 && test[i][j] > -10) {
                                        System.out.print(" ");
                                }
                                if (test[i][j] < 100 && test[i][j] > -100) {
                                        System.out.print(" ");
                                }
                                if (test[i][j] >= 0) {
                                        System.out.print(" ");
                                }
                                System.out.print(" " + test[i][j]);
                        }
                        System.out.println();
                }
        }
}
```

1.7
Answer:

```
public class SetZeroSol {
        // O(N) space
        public static void setZeros1(int[][] matrix) {
                boolean[] row = new boolean[matrix.length];
                boolean[] column = new boolean[matrix[0].length];

                for (int i = 0; i < matrix.length; i++) {
                        for (int j = 0; j < matrix[0].length; j++) {
```

```
                        if (matrix[i][j] == 0) {
                                row[i] = true;
                                column[j] = true;
                        }
                }
        }

        for (int i = 0; i < row.length; i++) {
                if (row[i]) nullifyRow(matrix, i);
        }

        for (int j = 0; j < column.length; j++) {
                if (column[j]) nullifyColumn(matrix, j);
        }
}

public static void nullifyRow(int[][] matrix, int row) {
        for (int j = 0; j < matrix[0].length; j++) {
                matrix[row][j] = 0;
        }
}

public static void nullifyColumn(int[][] matrix, int col) {
        for (int i = 0; i < matrix.length; i++) {
                matrix[i][col] = 0;
        }
}
// O(1) space
public static void setZeros2(int[][] matrix) {
        boolean rowHasZero = false;
        boolean colHasZero = false;
        // Check if first row has a zero
        for (int j = 0; j < matrix[0].length; j++) {
                if (matrix[0][j] == 0) {
                        rowHasZero = true;
                        break;
                }
        }
        // Check if first column has a zero
        for (int i = 0; i < matrix.length; i++) {
                if (matrix[i][0] == 0) {
                        colHasZero = true;
                        break;
                }
        }
        // Check for zeros in the rest of the array
        for (int i = 1; i < matrix.length; i++) {
                for (int j = 1; j < matrix[0].length; j++) {
                        if (matrix[i][j] == 0) {
```

```
                                matrix[i][0] = 0;
                                matrix[0][j] = 0;
                        }
                }
        }

        for (int i = 1; i < matrix.length; i++) {
                if (matrix[i][0] == 0) {
                        nullifyRow(matrix, i);
                }
        }

        for (int j = 1; j < matrix[0].length; j++) {
                if (matrix[0][j] == 0) {
                        nullifyColumn(matrix, j);
                }
        }

        if (rowHasZero) {
                nullifyRow(matrix, 0);
        }

        if (colHasZero) {
                nullifyColumn(matrix, 0);
        }
}

public static void main(String[] args) {
        int M = 5;
        int N = 7;
        int[][] test = new int[M][N];
        for (int i = 0; i < M; i++) {
                for (int j = 0; j < N; j++) {
                        test[i][j] = (int) (Math.random() * (6 + 1 - 0)) + 0;
                }
        }
        for (int i = 0; i < test.length; i++) {
                for (int j = 0; j < test[i].length; j++) {
                        System.out.print(" " + test[i][j]);
                }
                System.out.println();
        }
        System.out.println();
        setZeros1(test);
        for (int i = 0; i < test.length; i++) {
                for (int j = 0; j < test[i].length; j++) {
                        System.out.print(" " + test[i][j]);
                }
                System.out.println();
```

```
                }
        }
}
```

1.8
Answer:

```
/*
int indexOf(int ch): Returns the index within this string of the first occurrence of the specified character or -1 if the character does not occur.
int indexOf(int ch, int fromIndex): Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index or -1 if the character does not occur.
int indexOf(String str): Returns the index within this string of the first occurrence of the specified substring. If it does not occur as a substring, -1 is returned.
int indexOf(String str, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. If it does not occur, -1 is returned.
*/
public class IsRotationSol {
        public static boolean isRotation(String s1, String s2) {
                int len = s1.length();

                if (len == s2.length() && len > 0) {
                        String s1s1 = s1 + s1;
                        return isSubstring(s1s1, s2);
                }
                return false;
        }

        private static boolean isSubstring(String big, String small) {
                if (big.indexOf(small) >= 0) {
                        return true;
                } else {
                        return false;
                }
        }

        public static void main(String[] args) {
                System.out.print(isRotation("xy", "yx"));
        }
}
```

1.4 6th
Answer:

```
// time O(N)
public class Question1_4_6th {
```

```
	public static boolean isPermutationOfPalindrome(String phrase) {
		int bitVector = createBitVector(phrase);
		return bitVector == 0 || checkExactlyOneBitSet(bitVector);
	}

	public static int createBitVector(String phrase) {
		int bitVector = 0;
		for (char c : phrase.toCharArray()) {
			int x = getCharNumber(c);
			bitVector = toggle(bitVector, x);
		}
		return bitVector;
	}

	public static int toggle(int bitVector, int index) {
		if (index < 0) return bitVector;

		int mask = 1 << index;
		if ((bitVector & mask) == 0) {
			bitVector |= mask;
		} else {
			bitVector &= ~mask;
		}
		return bitVector;
	}

	public static boolean checkExactlyOneBitSet(int bitVector) {
		return (bitVector & (bitVector - 1)) == 0;
	}

	public static int getCharNumber(Character c) {
		int a = Character.getNumericValue('a');
		int z = Character.getNumericValue('z');
		int A = Character.getNumericValue('A');
		int Z = Character.getNumericValue('Z');

		int val = Character.getNumericValue(c);
		if (a <= val && val <= z) {
			return val - a;
		} else if (A <= val && val <= Z) {
			return val - A;
		}
		return -1;
	}

	public static void main(String[] args) {
		System.out.print(isPermutationOfPalindrome("bb"));
	}
}
```

CHAPTER 2
preface code
Answer:

```
public class PrefaceCode {
	// Creating a Linked List
	public static class Node {
		Node next = null;
		int data;

		public Node(int d) {
			data = d;
		}

		public void appendToTail(int d) {
			Node end = new Node(d);
			Node n = this;
			while (n.next != null) {
				n = n.next;
			}
			n.next = end;
		}
	}
	// Deleting a Node from a Singly Linked List
	public static Node deleteNode1(Node head, int d) {
		Node n = head;

		if (n.data == d) {
			return head.next;
		}

		while (n.next != null) {
			if (n.next.data == d) {
				n.next = n.next.next;
				return head;
			}
			n = n.next;
		}
		return head;
	}

	public static Node deleteNode2(Node head, int d) {
		Node dummy = new Node(0);
		dummy.next = head;
		Node pre = dummy;

		while (pre.next != null) {
			if (pre.next.data == d) {
				pre.next = pre.next.next;
				return dummy.next;
```

```
                }
                pre = pre.next;
            }
            return dummy.next;
        }

        public static Node reorderList1(Node head) {
            if (head == null) {
                return head;
            } else if (head.next == null) {
                return head;
            }

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

            merge(head, right);

            return head;
        }

        private static Node findMidPre(Node head) {
            if (head == null) {
                return null;
            }

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

            return slow;
        }

        private static void merge(Node head1, Node head2) {
            Node dummy = new Node(0);
            Node 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 Node reorderList2(Node head) {
            if (head == null) {
                return null;
            }

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

            Node slowNode = slow.next;
            slow.next = null;
            Node fastNode = head;

            Node newNode = new Node(0);
            Node tmp = newNode;

            while (fastNode != null && slowNode != null) {
                tmp.next = fastNode;
                tmp = tmp.next;
                fastNode = fastNode.next;
                tmp.next = slowNode;
                tmp = tmp.next;
                slowNode = slowNode.next;
            }

            if (fastNode != null) {
                tmp.next = fastNode;
            } else {
                tmp.next = slowNode;
            }

            return newNode.next;
        }

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

```
		n2.next = n3;
		n3.next = null;
		Node rst1 = deleteNode2(n1, 2);
		while (rst1 != null) {
			System.out.print(rst1.data + " ");
			rst1 = rst1.next;
		}
		System.out.println();

		Node a1 = new Node(1);
		Node a2 = new Node(2);
		Node a3 = new Node(3);
		Node b1 = new Node(1);
		Node b2 = new Node(2);
		Node b3 = new Node(3);
		a1.next = a2;
		a2.next = a3;
		a3.next = b1;
		b1.next = b2;
		b2.next = b3;
		b3.next = null;
		Node rst2 = reorderList2(a1);
		while (rst2 != null) {
			System.out.print(rst2.data + " ");
			rst2 = rst2.next;
		}
	}
}
```

2.1
Answer:

```
public class RemoveDupSol {
	// time O(N) space O(N)
	public static ListNode removeDup1(ListNode head) {
		Hashtable<Integer, Boolean> table = new Hashtable<Integer, Boolean>();

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

		while (head != null) {
			if (table.containsKey(head.val)) {
				pre.next = head.next;
			} else {
				table.put(head.val, true);
				pre = head;
			}
			head = head.next;
		}
```

```
        return dummy.next;
    }
    // time O(N^2) space O(1)
    public static ListNode removeDup2(ListNode head) {
        if (head == null) {
            return head;
        }

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

        while (cur.next != null) {
            ListNode runner = cur.next;
            while (runner.next != null) {
                if (runner.next.val == cur.next.val) {
                    runner.next = runner.next.next;
                } else {
                    runner = runner.next;
                }
            }
            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(6);
        ListNode n2 = new ListNode(1);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(6);
        ListNode n5 = new ListNode(7);
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;
        n4.next = n5;
        n5.next = null;
```

```
		ListNode rst = removeDup2(n1);

		while (rst != null) {
			System.out.print(rst.val);
			rst = rst.next;
		}
	}
}
```

2.2
Answer:

```
/*
notes about the space complex for recursive
是程序运行所以需要的额外消耗存储空间, 一般的递归算法就要有o(n)的空间复杂度了, 简单说就是递归集算时通常是反复调用同一个方法, 因为每次递归都要存储返回信息, 递归n次, 就需要n个空间
*/
public class Question2_2 {
	// Recursive space O(n)
	public static int nthToLast1(ListNode head, int k) {
		if (head == null) {
			return 0;
		}

		int i = nthToLast1(head.next, k) + 1;
		if (i == k) {
			System.out.println(head.val);
		}
		return i;
	}
	// Recursive space O(n)
	public static ListNode nthToLast2(ListNode head, int k, IntWrapper i) {
		if (head == null) {
			return null;
		}

		ListNode node = nthToLast2(head.next, k, i);
		i.value = i.value + 1;
		if (i.value == k) {
			return head;
		}
		return node;
	}

	public static class IntWrapper {
		public int value = 0;
	}
	// Iterative time O(n) space O(1)
	public static ListNode nthToLast3(ListNode head, int k) {
```

```
        if (k <= 0) return null;

        ListNode p1 = head;
        ListNode p2 = head;

        for (int i = 0; i < k - 1; i++) {
            if (p2 == null) return null;
            p2 = p2.next;
        }

        if (p2 == null) return null;

        while (p2.next != null) {
            p1 = p1.next;
            p2 = p2.next;
        }

        return p1;
    }

    public static class ListNode {
        public int val;
        public ListNode next;
        public ListNode(int x) {
            val = x;
        }
        public String printForward() {
            if (next != null) {
                return val + "->" + next.printForward();
            } else {
                return ((Integer) val).toString();
            }
        }
    }

    public static ListNode randomLinkedList(int N, int min, int max) {
        ListNode root = new ListNode(randomIntInRange(min, max));
        ListNode pre = root;
        for (int i = 1; i < N; i++) {
            int val = randomIntInRange(min, max);
            ListNode next = new ListNode(val);
            pre.next = next;
            pre = next;
        }
        return root;
    }

    public static int randomInt(int n) {
        return (int) (Math.random() * n);
```

```
	}

	public static int randomIntInRange(int min, int max) {
		return randomInt(max + 1 - min) + min;
	}

	public static void main(String[] args) {

		ListNode test = randomLinkedList(9, 0, 10);
		/*
		while (test != null) {
			System.out.print(test.val);
			test = test.next;
			if (test != null) {
				System.out.print("->");
			}
		}
		*/
		System.out.print(test.printForward());
		System.out.println();
		int nth = 3;
		System.out.println();
		nthToLast1(test, nth);
		IntWrapper i = new IntWrapper();
	System.out.print(nthToLast2(test, nth, i).val);
	System.out.println();
		System.out.print(nthToLast3(test, nth).val);
	}
}
```

2.3
Answer:

```
/*
常见错误:
"if lastNode.next == null; lastNode = null;" 注意lastNode = null代表的是用lastNode指代null, 注意
指代和指向不同
n1.next是指向一个内存, 如果n2 = null没有改变内存只是重赋值
e.g.
*/
/*
public static void main(String[] args) {
		ListNode n1 = new ListNode(0);
		ListNode n2 = new ListNode(1);
		n1.next = n2;
		n2.next = null;
		n2 = null;
		// n1.next = n2;
		if (n2 == null) {
			System.out.print("n2 is null" + "\n");
```

```
                }
                while (n1 != null) {
                        System.out.print(n1.val);
                        n1 = n1.next;
                }
        }
*/
/*
output:
n2 is null
01
*/
/*
1. In Doubly linked list no need of copying b/c it allows Bidirectinal iteration
2. If u need to delete the last node in the list, say the list is 1->2->3 and u had pointer to 3 alone, just create a new node with some dummy value. say the value is zero. now attach it to the list. it becomes 1->2->3->0 now swap the value in dummy node to the node to be deleted (here '3') and now free the memory of the dummy node. hence list becomes 1->2->0.
*/
public class Question2_3 {
        public static boolean deleteNode1(ListNode n) {
                if (n == null || n.next == null) {
                        return false;
                }

                ListNode next = n.next;
                n.val = next.val;
                n.next = next.next;
                return true;
        }

        public static class ListNode {
                public int val;
                public ListNode next;
                public ListNode(int x) {
                        val = x;
                }
                public String printForward() {
                        if (next != null) {
                                return val + "->" + next.printForward();
                        } else {
                                return ((Integer) val).toString();
                        }
                }
        }

        public static ListNode randomLinkedList(int N, int min, int max) {
                ListNode root = new ListNode(randomIntInRange(min, max));
                ListNode pre = root;
```

```
                for (int i = 1; i < N; i++) {
                        int val = randomIntInRange(min, max);
                        ListNode next = new ListNode(val);
                        pre.next = next;
                        pre = next;
                }
                return root;
        }

        public static int randomInt(int n) {
                return (int) (Math.random() * n);
        }

        public static int randomIntInRange(int min, int max) {
                return randomInt(max + 1 - min) + min;
        }

        public static void main(String[] args) {
                ListNode test = randomLinkedList(9, 0, 10);
                System.out.print(test.printForward() + "\n");
                deleteNode1(test.next.next.next.next);
                System.out.println(test.printForward());
        }
}
```

2.4
Answer:

```
public class Question2_4 {
        public static ListNode partition1(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 ListNode partition2(ListNode head, int x) {
        ListNode beforeStart = null;
        ListNode beforeEnd = null;
        ListNode afterStart = null;
        ListNode afterEnd = null;

        while (head != null) {
            ListNode next = head.next;
            head.next = null;
            if (head.val < x) {
                if (beforeStart == null) {
                    beforeStart = head;
                    beforeEnd = beforeStart;
                } else {
                    beforeEnd.next = head;
                    beforeEnd = head;
                }
            } else {
                if (afterStart == null) {
                    afterStart = head;
                    afterEnd = afterStart;
                } else {
                    afterEnd.next = head;
                    afterEnd = head;
                }
            }
            head = next;
        }

        if (beforeStart == null) {
            return afterStart;
        }

        beforeEnd.next = afterStart;
        return beforeStart;
    }

    public static ListNode partition3(ListNode head, int x) {
        ListNode beforeStart = null;
        ListNode afterStart = null;

        while (head != null) {
            ListNode next = head.next;
            if (head.val < x) {
                head.next = beforeStart;
                beforeStart = head;
```

```
                } else {
                        head.next = afterStart;
                        afterStart = head;
                }
                head = next;
        }

        if (beforeStart == null) {
                return afterStart;
        }

        ListNode head1 = beforeStart;
        while (beforeStart.next != null) {
                beforeStart = beforeStart.next;
    }
        beforeStart.next = afterStart;

        return head1;
}

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

                public String printForward() {
                        if (next != null) {
                                return val + "->" + next.printForward();
                        } else {
                                return ((Integer) val).toString();
                        }
                }
        }

        public static void main(String[] args) {

                int[] vals = {1, 3, 7, 5, 2, 9, 4};
                ListNode head = new ListNode(vals[0]);
                ListNode current = head;

                for (int i = 1; i < vals.length; i++) {
                        ListNode tmp = new ListNode(vals[i]);
                        current.next = tmp;
                        current = current.next;
                }

                System.out.println(head.printForward());
```

```
                ListNode rst = partition3(head, 5);
                System.out.println(rst.printForward());

                int len = 20;
                ListNode[] nodes = new ListNode[len];
                for (int i = 0; i < len; i++) {
                        if (i >= len / 2) {
                                nodes[i] = new ListNode(len - i - 1);
                        } else {
                                nodes[i] = new ListNode(i);
                        }
                }

                ListNode head2 = nodes[0];
                ListNode cur = head2;
                for (int i = 1; i < len; i++) {
                        ListNode tmp2 = nodes[i];
                        cur.next = tmp2;
                        cur = cur.next;
                }

                System.out.println(head2.printForward());

                ListNode rst2 = partition3(head2, 8);
                System.out.println(rst2.printForward());

        }
}
```

2.5
Answer:

```
public class Question2_5 {

        public static ListNode addList0(ListNode l1, ListNode l2) {
                if (l1 == null || l2 == null) {
                        return null;
                }

                ListNode dummy = new ListNode(0);
                ListNode tail = dummy;
                int carry = 0;
                while (l1 != null || l2 != null || carry == 1) {
                        int sum = carry;
                        if (l1 != null) {
                                sum += l1.val;
                                l1 = l1.next;
                        }
```

```
                        if (l2 != null) {
                                sum += l2.val;
                                l2 = l2.next;
                        }

                        carry = sum / 10;
                        ListNode cur = new ListNode(sum % 10);
                        tail.next = cur;
                        tail = tail.next;
                }
                return dummy.next;
        }

        public static ListNode addList1(ListNode l1, ListNode l2, int carry) {
                if (l1 == null && l2 == null && carry == 0) {
                        return null;
                }

                // ListNode rst = new ListNode();

                int value = carry;
                if (l1 != null) {
                        value += l1.val;
                }
                if (l2 != null) {
                        value += l2.val;
                }

                // rst.val = value % 10;
                ListNode rst = new ListNode(value % 10);
                ListNode cur = rst;

                if (l1 != null || l2 != null) {
                        ListNode more = addList1(l1 == null ? null : l1.next, l2 == null ? null :
l2.next, value >= 10 ? 1 : 0);
                        // rst.setNext(more);
                        cur.next = more;
                        cur = cur.next;
                }
                return rst;
        }

        /*
        public static class ListNode {
                public int val;
                public ListNode next;
                public ListNode prev;
                public ListNode last;
                public ListNode(int x, ListNode n, ListNode p) {
```

```
                val = x;
                setNext(n);
                setPrevious(p);
        }

        public ListNode() { }

        public void setNext(ListNode n) {
                next = n;

                if (this == last) {
                        last = n;
                }
                if (n != null && n.prev != this) {
                        n.setPrevious(this);
                }
        }

        public void setPrevious(ListNode p) {
                prev = p;

                if (p != null && p.next != this) {
                        p.setNext(this);
                }
        }

        public String printForward() {
                if (next != null) {
                        return val + "->" + next.printForward();
                } else {
                        return ((Integer) val).toString();
                }
        }
}
*/
public static class ListNode {
        public int val;
        public ListNode next;
        public ListNode(int x) {
                val = x;
        }
        public String printForward() {
                if (next != null) {
                        return val + "->" + next.printForward();
                } else {
                        return ((Integer) val).toString();
                }
        }
}
```

```
	public static void main(String[] args) {

		ListNode lA1 = new ListNode(9);
		ListNode lA2 = new ListNode(9);
		ListNode lA3 = new ListNode(9);
		lA1.next = lA2;
		lA2.next = lA3;
		lA3.next = null;
		ListNode lB1 = new ListNode(1);
		ListNode lB2 = new ListNode(0);
		ListNode lB3 = new ListNode(0);
		lB1.next = lB2;
		lB2.next = lB3;
		lB3.next = null;

		/*
		ListNode lA1 = new ListNode(9, null, null);
		ListNode lA2 = new ListNode(9, null, lA1);
		ListNode lA3 = new ListNode(9, null, lA2);

		ListNode lB1 = new ListNode(1, null, null);
		ListNode lB2 = new ListNode(0, null, lB1);
		ListNode lB3 = new ListNode(0, null, lB2);

		ListNode list3 = addList1(lA1, lB1, 0);
		*/
		ListNode list = addList0(lA1, lB1);
		ListNode list3 = addList1(lA1, lB1, 0);

		System.out.println("  " + lA1.printForward());
		System.out.println("+ " + lB1.printForward());
		System.out.println("= " + list3.printForward());
	}
}
/*
outputs:
  9->9->9
+ 1->0->0
= 0->0->0->1
*/
```

2.5 follow up
Answer:

```
public class Question2_5_B {

	public static class PartialSum {
		public ListNode sum = null;
		public int carry = 0;
```

```
}

public static ListNode addLists(ListNode l1, ListNode l2) {
	int len1 = length(l1);
	int len2 = length(l2);

	if (len1 < len2) {
		l1 = padList(l1, len2 - len1);
	} else {
		l2 = padList(l2, len1 - len2);
	}

	PartialSum sum = addListsHelper(l1, l2);

	if (sum.carry == 0) {
		return sum.sum;
	} else {
		ListNode rst = insertBefore(sum.sum, sum.carry);
		return rst;
	}
}

public static PartialSum addListsHelper(ListNode l1, ListNode l2) {
	if (l1 == null && l2 == null) {
		PartialSum sum = new PartialSum();
		return sum;
	}

	PartialSum sum = addListsHelper(l1.next, l2.next);

	int val = sum.carry + l1.val + l2.val;

	ListNode full_result = insertBefore(sum.sum, val % 10);

	sum.sum = full_result;
	sum.carry = val / 10;
	return sum;
}

public static int length(ListNode l) {
	if (l == null) {
		return 0;
	} else {
		return 1 + length(l.next);
	}
}

public static ListNode padList(ListNode l, int padding) {
	ListNode head = l;
```

```
        ListNode pos = head;
        for (int i = 0; i < padding; i++) {
                ListNode n = new ListNode(0);
                n.next = pos;
                pos = n;
        }
        return head;
}

public static ListNode insertBefore(ListNode list, int data) {
        ListNode node = new ListNode(data);
        ListNode cur = node;
        if (list != null) {
                cur.next = list;
        }
        return node;
}

public static class ListNode {
        public int val;
        public ListNode next;
        public ListNode(int x) {
                val = x;
        }
        public String printForward() {
                if (next != null) {
                        return val + "->" + next.printForward();
                } else {
                        return ((Integer) val).toString();
                }
        }
}

public static void main(String[] args) {
        ListNode lA1 = new ListNode(3);
        ListNode lA2 = new ListNode(1);
        ListNode lA3 = new ListNode(5);
        lA1.next = lA2;
        lA2.next = lA3;
        lA3.next = null;
        ListNode lB1 = new ListNode(5);
        ListNode lB2 = new ListNode(9);
        ListNode lB3 = new ListNode(1);
        lB1.next = lB2;
        lB2.next = lB3;
        lB3.next = null;
        ListNode list3 = addLists(lA1, lB1);

        System.out.println("  " + lA1.printForward());
```

```
                System.out.println("+ " + lB1.printForward());
                System.out.println("= " + list3.printForward());
        }
}
/*
outputs:
  3->1->5
+ 5->9->1
= 9->0->6
*/
```

2.6 LeetCode Linked List Cycle II

Answer:

```
public class Question2_6 {
        public static ListNode FindBeginning(ListNode head) {
                ListNode slow = head;
                ListNode fast = head;

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

                        if (slow == fast) {
                                break;
                        }
                }

                if (fast == null ll fast.next == null) {
                        return null;
                }

                slow = head;
                while (slow != fast) {
                        slow = slow.next;
                        fast = fast.next;
                }

                return fast;
        }

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

                public String printForward() {
                        if (next != null) {
```

```
                        return val + "->" + next.printForward();
                    } else {
                        return ((Integer) val).toString();
                    }
                }
        }

        public static void main(String[] args) {
                int list_len = 100;
                int k = 10;

                ListNode[] nodes = new ListNode[list_len];
                for (int i = 0; i < list_len; i++) {
                        nodes[i] = new ListNode(i);
                }

                ListNode head = nodes[0];
                ListNode cur = head;
                for (int i = 1; i < list_len; i++) {
                        ListNode tmp = nodes[i];
                        cur.next = tmp;
                        cur = cur.next;
                }

                // System.out.println(nodes[0].printForward());

                nodes[list_len - 1].next = nodes[k];
                ListNode loop = FindBeginning(nodes[0]);
                if (loop == null) {
                        System.out.println("No Cycle.");
                } else {
                        System.out.println(loop.val);
                }
        }
}
```

2.7
Answer:

```
public class Question2_7 {
        // Iteration
        public static boolean isPalindrome1(ListNode head) {
                ListNode fast = head;
                ListNode slow = head;

                Stack<Integer> stack = new Stack<Integer>();

                while (fast != null && fast.next != null) {
                        stack.push(slow.val);
                        slow = slow.next;
```

```
                fast = fast.next.next;
            }
            /* Has odd number of elements, so skip the middle element */
            if (fast != null) {
                slow = slow.next;
            }

            while (slow != null) {
                int top = stack.pop().intValue();
                if (top != slow.val) {
                    return false;
                }
                slow = slow.next;
            }
            return true;
        }
        // Recursive
        public static boolean isPalindrome2(ListNode head) {
            int size = 0;
            ListNode n = head;
            while (n != null) {
                size++;
                n = n.next;
            }
            Result p = isPalindromeRecurse(head, size);
            return p.result;
        }

        public static Result isPalindromeRecurse(ListNode head, int length) {
            if (head == null || length == 0) {
                return new Result(null, true);
            } else if (length == 1) {
                return new Result(head.next, true);
            } else if (length == 2) {
                return new Result(head.next.next, head.val == head.next.val);
            }

            Result res = isPalindromeRecurse(head.next, length - 2);

            if (!res.result || res.node == null) {
                return res; // Only "result" member is actually used in the call stack.
            } else {
                res.result = head.val == res.node.val;
                res.node = res.node.next;
                return res;
            }
        }

        public static class Result {
```

```
		public ListNode node;
		public boolean result;
		public Result(ListNode n, boolean res) {
			node = n;
			result = res;
		}
	}

	public static class ListNode {
		public int val;
		public ListNode next;
		public ListNode(int x) {
			val = x;
		}
		public String printForward() {
			if (next != null) {
				return val + "->" + next.printForward();
			} else {
				return ((Integer) val).toString();
			}
		}
	}

	public static void main(String[] args) {

		int len = 10;
		ListNode[] nodes = new ListNode[len];
		for (int i = 0; i < len; i++) {
			if (i >= len / 2) {
				nodes[i] = new ListNode(len - i - 1);
			} else {
				nodes[i] = new ListNode(i);
			}
		}

		ListNode head = nodes[0];
		ListNode cur = head;
		for (int i = 1; i < len; i++) {
			ListNode tmp = nodes[i];
			cur.next = tmp;
			cur = cur.next;
		}

		System.out.println(head.printForward());

		Question2_7 q = new Question2_7();
		System.out.println(q.isPalindrome1(head));
	}
}
```

2.7 (6th edition)
Answer:

```
// time O(A + B) space O(1)
public class Question2_7_6th {
	public static ListNode findIntersection(ListNode list1, ListNode list2) {
		if (list1 == null || list2 == null) {
			return null;
		}

		Result result1 = getTailAndSize(list1);
		Result result2 = getTailAndSize(list2);

		if (result1.tail != result2.tail) {
			return null;
		}

		ListNode shorter = result1.size < result2.size ? list1 : list2;
		ListNode longer = result1.size > result2.size ? list1 : list2;

		longer = getKthNode(longer, Math.abs(result1.size - result2.size));

		while (shorter != longer) {
			shorter = shorter.next;
			longer = longer.next;
		}

		return longer;
	}

	public static class Result {
		public int size;
		public ListNode tail;
		public Result(ListNode tail, int size) {
			this.tail = tail;
			this.size = size;
		}
	}

	public static Result getTailAndSize(ListNode list) {
		if (list == null) {
			return null;
		}

		int size = 1;
		ListNode cur = list;
		while (cur.next != null) {
			size++;
			cur = cur.next;
```

```
		}

		return new Result(cur, size);
	}

	public static ListNode getKthNode(ListNode head, int k) {
		ListNode cur = head;
		while (k > 0 && cur != null) {
			cur = cur.next;
			k--;
		}
		return cur;
	}

	public static ListNode findIntersection2(ListNode list1, ListNode  list2) {
		if (list1 == null || list2 == null) {
			return null;
		}

		int len1 = getLen(list1);
		int len2 = getLen(list2);
		int cnt = Math.abs(len1 - len2);

		if (len1 > len2) {
			while (cnt > 0) {
				list1 = list1.next;
				cnt--;
			}
		} else {
			while (cnt > 0) {
				list2 = list2.next;
				cnt--;
			}
		}

		while (list1 != null) {
			if (list1 == list2) {
				return list1;
			}
			list1 = list1.next;
			list2 = list2.next;
		}
		return null;
	}

	public static int getLen(ListNode list) {
		int cnt = 0;
		while (list != null) {
			cnt++;
```

```
                list = list.next;
        }
        return cnt;
}

public static class ListNode {
        public int val;
        public ListNode next;
        public ListNode(int x) {
                val = x;
        }
        public String printForward() {
                if (next != null) {
                        return val + "->" + next.printForward();
                } else {
                        return ((Integer) val).toString();
                }
        }
}

public static void main(String[] args) {
        int[] set1 = {3, 1, 5, 9};
        int[] set2 = {4, 6};
        int[] set3 = {7, 2, 1};
        ListNode[] nodes1 = new ListNode[set1.length];
        ListNode[] nodes2 = new ListNode[set2.length];
        ListNode[] nodes3 = new ListNode[set3.length];

        for (int i = 0; i < set1.length; i++) {
                nodes1[i] = new ListNode(set1[i]);
        }

        for (int i = 0; i < set2.length; i++) {
                nodes2[i] = new ListNode(set2[i]);
        }

        for (int i = 0; i < set3.length; i++) {
                nodes3[i] = new ListNode(set3[i]);
        }

        ListNode list1 = nodes1[0];
        ListNode cur1 = list1;
        for (int i = 1; i < set1.length; i++) {
                ListNode tmp = nodes1[i];
                cur1.next = tmp;
                cur1 = cur1.next;
        }
        for (int i = 0; i < set3.length; i++) {
                ListNode tmp = nodes3[i];
```

```
                        cur1.next = tmp;
                        cur1 = cur1.next;
                }

                ListNode list2 = nodes2[0];
                ListNode cur2 = list2;
                for (int i = 1; i < set2.length; i++) {
                        ListNode tmp = nodes2[i];
                        cur2.next = tmp;
                        cur2 = cur2.next;
                }
                for (int i = 0; i < set3.length; i++) {
                        ListNode tmp = nodes3[i];
                        cur2.next = tmp;
                        cur2 = cur2.next;
                }
                System.out.println(list1.printForward());
                System.out.println(list2.printForward());

                System.out.print(findIntersection2(list1, list2).val);
        }
}
```

CHAPTER 3

```
public class PrefaceCode {
	/* Implement Stack use Linked List */
	public static class MyStack<T> {
		private class StackNode<T> {
			private T data;
			private StackNode<T> next;

			public StackNode(T data) {
				this.data = data;
			}
		}

		private StackNode<T> top;

		public T pop() {
			if (top == null) throw new EmptyStackException();
			T item = top.data;
			top = top.next;
			return item;
		}

		public void push(T item) {
			StackNode<T> t = new StackNode<T>(item);
			t.next = top;
			top = t;
		}

		public T peek() {
			if (top == null) throw new EmptyStackException();
			return top.data;
		}

		public boolean isEmpty() {
			return top == null;
		}
	}

	/* Implement Queue use Linked List */
	public static class MyQueue<T> {
		private class QueueNode<T> {
			private T data;
			private QueueNode<T> next;

			public QueueNode(T data) {
				this.data = data;
			}
		}
```

```
		private QueueNode<T> first;
		private QueueNode<T> last;

		public void add(T item) {
			QueueNode<T> t = new QueueNode<T>(item);
			if (last != null) {
				last.next = t;
			}
			last = t;
			if (first == null) {
				first = last;
			}
		}

		public T remove() {
			if (first == null) throw new NoSuchElementException();
			T data = first.data;
			first = first.next;
			if (first == null) {
				last = null;
			}
			return data;
		}

		public T peek() {
			if (first == null) throw new EmptyStackException();
			return first.data;
		}

		public boolean isEmpty() {
			return first == null;
		}
	}
}
```

3.1 A
Answer:
/*
throw和throws的区别:
1. throw代表动作, 表示抛出一个异常的动作; throws代表一种状态, 代表方法可能有异常抛出
2. throw用在方法实现中, 而throws用在方法声明中
3. throw只能用于抛出一种异常, 而throws可以抛出多个异常
throws是用来声明一个方法可能抛出的所有异常信息
throw则是指抛出的一个具体的异常类型
通常在一个方法(类)的声明处通过throws声明方法(类)可能抛出的异常信息, 而在方法(类)内部通过throw声明一个具体的异常信息,
throws通常不用显示的捕获异常, 可由系统自动将所有捕获的异常信息抛给上级方法,
throw则需要用户自己捕获相关的异常, 而后在对其进行相关包装, 最后在将包装后的异常信息抛.
*/

```
public class Question3_1_A {
	public static class FixedMultiStack1 {
		private int numberOfStacks = 3;
		private int stackCapacity;
		private int[] values;
		private int[] sizes;

		public FixedMultiStack1(int stackSize) {
			stackCapacity = stackSize;
			values = new int[stackSize * numberOfStacks];
			sizes = new int[numberOfStacks];
		}

		public void push(int stackNum, int value) throws FullStackException {
			if (isFull(stackNum)) {
				throw new FullStackException();
			}

			sizes[stackNum]++;
			values[indexOfTop(stackNum)] = value;
		}

		public int pop(int stackNum) throws Exception {
			if (isEmpty(stackNum)) {
				throw new EmptyStackException();
			}

			int topIndex = indexOfTop(stackNum);
			int value = values[topIndex];
			values[topIndex] = 0;
			sizes[stackNum]--;
			return value;
		}

		public int peek(int stackNum) {
			if (isEmpty(stackNum)) {
				throw new EmptyStackException();
			}
			return values[indexOfTop(stackNum)];
		}

		public boolean isFull(int stackNum) {
			return sizes[stackNum] == stackCapacity;
		}

		public boolean isEmpty(int stackNum) {
			return sizes[stackNum] == 0;
		}
```

```
		private int indexOfTop(int stackNum) {
			int offset = stackNum * stackCapacity;
			int size = sizes[stackNum];
			return offset + size - 1;
		}
	}

	public static class FullStackException extends Exception {

		private static final long serialVersionUID = 1L;

		public FullStackException() {
			super();
		}

		public FullStackException(String message) {
			super(message);
		}
	}

	static int stackSize = 10;
	// static int[] buffer = new int[stackSize * 3];
	// static int[] stackPointer = {-1, -1, -1};

	public static void main(String[] args) throws Exception {
		FixedMultiStack1 test = new FixedMultiStack1(stackSize);
		test.push(2, 4);
		System.out.println("Peek 2: " + test.peek(2));
		test.push(0, 3);
		test.push(0, 7);
		test.push(0, 5);
		System.out.println("Peek 0: " + test.peek(0));
		test.pop(0);
		System.out.println("Peek 0: " + test.peek(0));
		test.pop(0);
		System.out.println("Peek 0: " + test.peek(0));
	}
}
```

3.1 B
Answer:

```
public class Question3_1_B {

	public static class StackData {
		public int start;
		public int pointer;
		public int size;
		public int capacity;
		public StackData(int _start, int _capacity) {
```

```
                        start = _start;
                        pointer = _start - 1;
                        capacity = _capacity;
                }

                public boolean isWithinStackCapacity(int index, int total_size) {
                        if (start <= index && index < start + capacity) {
                                return true;
                        } else  if (start + capacity > total_size && index < (start + capacity) %
total_size) {
                                return true;
                        }
                        return false;
                }
        }

        static int number_of_stacks = 3;
        static int default_size = 4;
        static int total_size = default_size * number_of_stacks;
   static StackData[] stacks = {new StackData(0, default_size),
        new StackData(default_size, default_size),
        new StackData(default_size * 2, default_size)};
        static int[] buffer = new int[total_size];

        public static int numberOfElements() {
                return stacks[0].size + stacks[1].size + stacks[2].size;
        }

        public static int nextElement(int index) {
                if (index + 1 == total_size) {
                        return 0;
                } else {
                        return index + 1;
                }
        }

        public static int previousElement(int index) {
                if (index == 0) {
                        return total_size - 1;
                } else {
                        return index - 1;
                }
        }

        public static void shift(int stackNum) {
                System.out.println("/// Shifting " + stackNum);
                StackData stack = stacks[stackNum];

                if (stack.size >= stack.capacity) {
```

```
                int nextStack = (stackNum + 1) % number_of_stacks;
                shift(nextStack);
                stack.capacity++;
        }

        for (int i = (stack.start + stack.capacity - 1) % total_size;
                        stack.isWithinStackCapacity(i, total_size);
                        i = previousElement(i)) {
                buffer[i] = buffer[previousElement(i)];
        }

        buffer[stack.start] = 0;
        stack.start = nextElement(stack.start);
        stack.pointer = nextElement(stack.pointer);
        stack.capacity--;
}

public static void expand(int stackNum) {
        shift((stackNum + 1) % number_of_stacks);
        stacks[stackNum].capacity++;
}

public static void push(int stackNum, int value) throws Exception {

        StackData stack = stacks[stackNum];

        if (stack.size >= stack.capacity) {
                if (numberOfElements() >= total_size) {
                        throw new Exception("Out of space.");
                } else {
                        expand(stackNum);
                }
        }

        stack.size++;
        stack.pointer = nextElement(stack.pointer);
        buffer[stack.pointer] = value;
}

public static int pop(int stackNum) throws Exception {
        StackData stack = stacks[stackNum];
        if (stack.size == 0) {
                throw new Exception("Trying to pop an empty stack.");
        }

        int value = buffer[stack.pointer];
        buffer[stack.pointer] = 0;
        stack.pointer = previousElement(stack.pointer);
        stack.size--;
```

```
                return value;
        }

        public static int peek(int stackNum) {
                StackData stack = stacks[stackNum];
                return buffer[stack.pointer];
        }

        public static boolean isEmpty(int stackNum) {
                StackData stack = stacks[stackNum];
                return stack.size == 0;
        }

        public static String arrayToString(int[] array) {
                StringBuilder sb = new StringBuilder();
                for (int v : array) {
                        sb.append(v + ",");
                }
                return sb.toString();
        }

        public static void main(String[] args) throws Exception {

                push(0, 10);
                push(1, 20);
                push(2, 30);
                push(1, 21);
                push(0, 11);
                push(0, 12);
                pop(0);
                push(2, 31);
                push(0, 13);
                push(1, 22);
                push(2, 31);
                push(2, 32);
                push(2, 33);
                push(2, 34);
                System.out.println("Final Stack: " + arrayToString(buffer));
                pop(1);
                push(2, 35);
                System.out.println("Final Stack: " + arrayToString(buffer));
        }
}
/*
output:
/// Shifting 0
/// Shifting 0
/// Shifting 1
Final Stack: 33, 34, 10, 11, 13, 20, 21, 22, 30, 31, 31, 32,
```

```
/// Shifting 0
/// Shifting 1
Final Stack: 33, 34, 35, 10, 11, 13, 20, 21, 30, 31, 31, 32,
*/

3.2
Answer:
public class Question3_2 {
	public static class StackWithMin1 extends Stack<NodeWithMin> {
		public void push(int value) {
			int newMin = Math.min(value, min());
			super.push(new NodeWithMin(value, newMin));
		}

		public int min() {
			if (this.isEmpty()) {
				return Integer.MAX_VALUE;
			} else {
				return peek().min;
			}
		}
	}

	public static class NodeWithMin {
		public int value;
		public int min;
		public NodeWithMin(int v, int min) {
			value = v;
			this.min = min;
		}
	}

	public static class StackWithMin2 extends Stack<Integer> {
		Stack<Integer> s2;
		public StackWithMin2() {
			s2 = new Stack<Integer>();
		}

		public void push(int value) {
			if (value <= min()) {
				s2.push(value);
			}
			super.push(value);
		}

		public Integer pop() {
			int value = super.pop();
			if (value == min()) {
				s2.pop();
```

```
                        }
                        return value;
                }

                public int min() {
                        if (s2.isEmpty()) {
                                return Integer.MAX_VALUE;
                        } else {
                                return s2.peek();
                        }
                }
        }

        public static int randomInt(int n) {
                return (int) (Math.random() * n);
        }

        public static int randomIntInRange(int min, int max) {
                return randomInt(max + 1 - min) + min;
        }

        public static void main(String[] args) {
                StackWithMin1 test1 = new StackWithMin1();
                StackWithMin2 test2 = new StackWithMin2();
                for (int i = 0; i < 15; i++) {
                        int value = randomIntInRange(0, 100);
                        test1.push(value);
                        test2.push(value);
                        System.out.println(value + ", ");
                }
                System.out.println('\n');
                for (int i = 0; i < 15; i ++) {
                        System.out.println("Popped " + test1.pop().value + ", " + test2.pop());
                        System.out.println("New min is " + test1.min() + ", " + test2.min());
                }
        }
}
/*
outputs:
40,
52,
76,
15,
10,
25,
27,
54,
42,
54,
```

```
37,
16,
47,
56,
87,
Popped 87, 87
New min is 10, 10
Popped 56, 56
New min is 10, 10
Popped 47, 47
New min is 10, 10
Popped 16, 16
New min is 10, 10
Popped 37, 37
New min is 10, 10
Popped 54, 54
New min is 10, 10
Popped 42, 42
New min is 10, 10
Popped 54, 54
New min is 10, 10
Popped 27, 27
New min is 10, 10
Popped 25, 25
New min is 10, 10
Popped 10, 10
New min is 15, 15
Popped 15, 15
New min is 40, 40
Popped 76, 76
New min is 40, 40
Popped 52, 52
New min is 40, 40
Popped 40, 40
New min is 2147483647, 2147483647
*/
```

3.3
Answer:

```
public class Question3_3 {
	public static class SetOfStacks {
		ArrayList<Stack> stacks = new ArrayList<Stack>();
		public int capacity;
		public SetOfStacks(int capacity) {
			this.capacity = capacity;
		}

		public Stack getLastStack() {
			if (stacks.size() == 0) return null;
```

```
                return stacks.get(stacks.size() - 1);
        }

        public void push(int v) {
                Stack last = getLastStack();
                if (last != null && !last.isFull()) {
                        last.push(v);
                } else {
                        Stack stack = new Stack(capacity);
                        stack.push(v);
                        stacks.add(stack);
                }
        }

        public int pop() {
                Stack last = getLastStack();
                int v = last.pop();
                if (last.size == 0) {
                        stacks.remove(stacks.size() - 1);
                }
                return v;
        }

        public int popAt(int index) {
                return leftShift(index, true);
        }

        public int leftShift(int index, boolean removeTop) {
                Stack stack = stacks.get(index);
                int removed_item;
                if (removeTop) {
                        removed_item = stack.pop();
                } else {
                        removed_item = stack.removeBottom();
                }

                if (stack.isEmpty()) {
                        stacks.remove(index);
                } else if (stacks.size() > index + 1) {
                        int v = leftShift(index + 1, false);
                        stack.push(v);
                }
                return removed_item;

        }
}

public static class Stack {
        private int capacity;
```

```
public Node top, bottom;
public int size = 0;

public Stack(int capacity) {
        this.capacity = capacity;
}

public boolean isFull() {
        return capacity == size;
}

public void join(Node above, Node below) {
        if (below != null) {
                below.above = above;
        }

        if (above != null) {
                above.below = below;
        }
}

public boolean push(int v) {
        if (size >= capacity) return false;
        size++;
        Node n = new Node(v);
        if (size == 1) {
                bottom = n;
        }
        join(n, top);
        top = n;
        return true;
}

public int pop() {
        Node t = top;
        top = top.below;
        size--;
        return t.value;
}

public boolean isEmpty() {
        return size == 0;
}

public int removeBottom() {
        Node b = bottom;
        bottom = bottom.above;
        if (bottom != null) {
                bottom.below = null;
```

```
                        }
                        size--;
                        return b.value;
                }
        }

        public static class Node {
                public Node above;
                public Node below;
                public int value;
                public Node(int value) {
                        this.value = value;
                }
        }

        public static void main(String[] args) {
                int capacity_per_substack = 5;
                SetOfStacks set = new SetOfStacks(capacity_per_substack);
                for (int i = 0; i < 34; i++) {
                        set.push(i);
                }
                for (int i = 0; i < 34; i++) {
                        System.out.println("Poped " + set.pop());
                }
        }
}
```

3.4 LeetCode: Implement Queue by Two Stacks
Answer:

```
public class Question3_4 {
        public static class MyQueue<T> {
                Stack<T> stackNewest, stackOldest;

                public MyQueue() {
                        stackNewest = new Stack<T>();
                        stackOldest = new Stack<T>();
                }

                public int size() {
                        return stackNewest.size() + stackOldest.size();
                }

                public void add(T value) {
                        stackNewest.push(value);
                }

                private void shiftStacks() {
                        if (stackOldest.isEmpty()) {
                                while (!stackNewest.isEmpty()) {
```

```
                                        stackOldest.push(stackNewest.pop());
                                }
                        }
                }

                public T peek() {
                        shiftStacks();
                        return stackOldest.peek();
                }

                public T remove() {
                        shiftStacks();
                        return stackOldest.pop();
                }
        }

        public static int randomInt(int n) {
                return (int) (Math.random() * n);
        }

        public static int randomIntInRange(int min, int max) {
                return randomInt(max + 1 - min) + min;
        }

        public static void main(String[] args) {
                MyQueue<Integer> my_queue = new MyQueue<Integer>();
                Queue<Integer> test_queue = new LinkedList<Integer>();

                for (int i = 0; i < 100; i++) {
                        int choice = randomIntInRange(0, 10);
                        if (choice <= 5) {
                                int element = randomIntInRange(1, 10);
                                test_queue.add(element);
                                my_queue.add(element);
                                System.out.println("Enqueued " + element);
                        } else if (test_queue.size() > 0) {
                                int top1 = test_queue.remove();
                                int top2 = my_queue.remove();
                                if (top1 != top2) {
                                        System.out.println("FAILURE - DIFFERENT TOPS: " +
top1 + ", " + top2);
                                }
                                System.out.println("Dequeued " + top1);
                        }

                        if (test_queue.size() == my_queue.size()) {
                                if (test_queue.size() > 0 && test_queue.peek() !=
my_queue.peek()) {
```

```
                                        System.out.println("FAILURE - DIFFERENT TOPS: " +
test_queue.peek() + ", " + my_queue.peek());
                                }
                        } else {
                                System.out.println("FAILURE - DIFFERENT SIZES");
                        }
                }
        }
}
```

3.5
Answer:

```
public class Question3_5 {
        // time O(N^2) space O(N)
        public static Stack<Integer> sort(Stack<Integer> s) {
                Stack<Integer> r = new Stack<Integer>();
                while (!s.isEmpty()) {
                        int tmp = s.pop();
                        while (!r.isEmpty() && r.peek() > tmp) {
                                s.push(r.pop());
                        }
                        r.push(tmp);
                }
                return r;
        }
        // Merge Sort
        static int c = 0;
        public static Stack<Integer> mergeSort(Stack<Integer> s2) {
                if (s2.size() <= 1) {
                        return s2;
                }

                Stack<Integer> left = new Stack<Integer>();
                Stack<Integer> right = new Stack<Integer>();
                int count = 0;
                while (s2.size() != 0) {
                        count++;
                        c++;
                        if (count % 2 == 0) {
                                left.push(s2.pop());
                        } else {
                                right.push(s2.pop());
                        }
                }

                left = mergeSort(left);
                right = mergeSort(right);

                while (left.size() > 0 || right.size() > 0) {
```

```
                if (left.size() == 0) {
                        s2.push(right.pop());
                } else if (right.size() == 0) {
                        s2.push(left.pop());
                } else if (right.peek().compareTo(left.peek()) <= 0) {
                        s2.push(left.pop());
                } else {
                        s2.push(right.pop());
                }
        }

        Stack<Integer> reverseStack = new Stack<Integer>();
        while (s2.size() > 0) {
                c++;
                reverseStack.push(s2.pop());
        }

        return reverseStack;
}
// Quick Sort
// time O(nlogn) to sort n items, time O(n^2) in worst case
public static Stack<Integer> quickSort(Stack<Integer> s3) {
        if (s3.isEmpty()) {
                return s3;
        }

        int pivot = s3.pop();

        Stack<Integer> left = new Stack<Integer>();
        Stack<Integer> right = new Stack<Integer>();
        while (!s3.isEmpty()) {
                int y = s3.pop();
                if (y < pivot) {
                        left.push(y);
                } else {
                        right.push(y);
                }
        }
        quickSort(left);
        quickSort(right);

        Stack<Integer> tmp = new Stack<Integer>();
        while (!right.isEmpty()) {
                tmp.push(right.pop());
        }
        tmp.push(pivot);
        while (!left.isEmpty()) {
                tmp.push(left.pop());
        }
```

```
                while (!tmp.isEmpty()) {
                        s3.push(tmp.pop());
                }
                return s3;
        }


        public static int randomInt(int n) {
                return (int) (Math.random() * n);
        }

        public static int randomIntInRange(int min, int max) {
                return randomInt(max + 1 - min) + min;
        }

        public static void main(String[] args) {
                Stack<Integer> test = new Stack<Integer>();
                for (int i = 0; i < 10; i++) {
                        int r = randomIntInRange(0, 1000);
                        test.push(r);
                }
                test = quickSort(test);
                while(!test.isEmpty()) {
                        System.out.println(test.pop());
                }
        }
}
```

3.6
Answer:

```
/*
note:
abstract, static, final
private, public, protected
*/
public class Question3_6 {
        public static abstract class Animal {
                private int order;
                protected String name;
                public Animal(String n) {
                        name = n;
                }
                public void setOrder(int ord) {
                        order = ord;
                }
                public int getOrder() {
                        return order;
                }
                public boolean isOlderThan(Animal a) {
```

```
                return this.order < a.getOrder();
        }
}

public static class AnimalQueue {
        LinkedList<Dog> dogs = new LinkedList<Dog>();
        LinkedList<Cat> cats = new LinkedList<Cat>();
        private int order = 0;

        public void enqueue(Animal a) {
                a.setOrder(order);
                order++;

                if (a instanceof Dog) {
                        dogs.addLast((Dog) a);
                } else if (a instanceof Cat) {
                        cats.addLast((Cat) a);
                }
        }

        public Animal dequeueAny() {
                if (dogs.size() == 0) {
                        return dequeueCats();
                } else if (cats.size() == 0) {
                        return dequeueDogs();
                }

                Dog dog = dogs.peek();
                Cat cat = cats.peek();
                if (dog.isOlderThan(cat)) {
                        return dequeueDogs();
                } else {
                        return dequeueCats();
                }
        }

        public Dog dequeueDogs() {
                return dogs.poll();
        }

        public Cat dequeueCats() {
                return cats.poll();
        }

        public int size() {
                return dogs.size() + cats.size();
        }
}
```

```
	public static class Dog extends Animal {
		public Dog(String n) {
			super(n);
		}
	}

	public static class Cat extends Animal {
		public Cat(String n) {
			super(n);
		}
	}

	public static void main(String[] args) {
		AnimalQueue animals = new AnimalQueue();
		animals.enqueue(new Cat("Callie"));
		animals.enqueue(new Cat("Kiki"));
		animals.enqueue(new Dog("Fido"));
		animals.enqueue(new Dog("Dora"));
		animals.enqueue(new Cat("Kari"));
		animals.enqueue(new Dog("Dexter"));
		animals.enqueue(new Dog("Dobo"));
		animals.enqueue(new Cat("Copa"));

		System.out.println(animals.dequeueAny().name);
		System.out.println(animals.dequeueAny().name);
		System.out.println(animals.dequeueAny().name);

		animals.enqueue(new Dog("Dapa"));
		animals.enqueue(new Cat("Kilo"));

		while (animals.size() != 0) {
			System.out.println(animals.dequeueAny().name);
		}
	}
}
```

CHAPTER 4
4.1
Answer:

```
public class Question4_1 {

	public static enum State {
		Unvisited, Visited, Visiting;
	}

	public static class Node {
		private Node adjacent[];
		public int adjacentCount;
		private String vertex;
		public State state;
		public Node(String vertex, int adjacentLength) {
			this.vertex = vertex;
			adjacentCount = 0;
			adjacent = new Node[adjacentLength];
		}

		public void addAdjacent(Node x) {
			if (adjacentCount < 30) {
				this.adjacent[adjacentCount] = x;
				adjacentCount++;
			} else {
				System.out.print("No more adjacent can be added");
			}
		}

		public Node[] getAdjacent() {
			return adjacent;
		}

		public String getVertex() {
			return vertex;
		}
	}

	public static class Graph {
		private Node vertices[];
		public int count;
		public Graph() {
			vertices = new Node[6];
			count = 0;
		}

		public void addNode(Node x) {
			if (count < 30) {
				vertices[count] = x;
```

```
                        count++;
                } else {
                        System.out.print("Graph full");
                }
        }

        public Node[] getNodes() {
                return vertices;
        }
}

public static Graph createNewGraph() {
        Graph g = new Graph();
        Node[] tmp = new Node[6];
        tmp[0] = new Node("a", 3);
        tmp[1] = new Node("b", 0);
        tmp[2] = new Node("c", 0);
        tmp[3] = new Node("d", 1);
        tmp[4] = new Node("e", 1);
        tmp[5] = new Node("f", 0);
        tmp[0].addAdjacent(tmp[1]);
        tmp[0].addAdjacent(tmp[2]);
        tmp[0].addAdjacent(tmp[3]);
        tmp[3].addAdjacent(tmp[4]);
        tmp[4].addAdjacent(tmp[5]);
        for (int i = 0; i < 6; i++) {
                g.addNode(tmp[i]);
        }
        return g;
}

public static boolean search(Graph g, Node start, Node end) {
        LinkedList<Node> q = new LinkedList<Node>();
        for (Node u : g.getNodes()) {
                u.state = State.Unvisited;
        }
        start.state = State.Visiting;
        q.add(start);
        Node u;
        while (!q.isEmpty()) {
                u = q.removeFirst();
                if (u != null) {
                        for (Node v : u.getAdjacent()) {
                                if (v.state == State.Unvisited) {
                                        if (v == end) {
                                                return true;
                                        } else {
                                                v.state = State.Visiting;
                                                q.add(v);
```

```
                                        }
                                    }
                                }
                                u.state = State.Visited;
                            }
                        }
                        return false;
                }

                public static void main(String[] args) {
                        Graph g = createNewGraph();
                        Node[] n = g.getNodes();
                        Node start = n[3];
                        Node end = n[5];
                        System.out.println(search(g, start, end));
                }
}
```

4.2
Answer:

```
public class Question4_2 {

        public static TreeNode createMinimalBST(int array[]) {
                return createMinimalBST(array, 0, array.length - 1);
        }

        public static TreeNode createMinimalBST(int arr[], int start, int end) {
                if (end < start) {
                        return null;
                }

                int mid = (start + end) / 2;
                TreeNode n = new TreeNode(arr[mid]);
                n.left = createMinimalBST(arr, start, mid - 1);
                n.right = createMinimalBST(arr, mid + 1, end);
                return n;
        }

        public static class TreeNode {
                int val;
                TreeNode left;
                TreeNode right;
                public TreeNode(int val) {
                        this.val = val;
                        left = null;
                        right = null;
                }

                public boolean isBST() {
```

```
                        if (left != null) {
                                if (val < left.val || !left.isBST()) {
                                        return false;
                                }
                        }

                        if (right != null) {
                                if (val >= right.val || !right.isBST()) {
                                        return false;
                                }
                        }

                        return true;
                }

                public int height() {
                        int leftHeight = left != null ? left.height() : 0;
                        int rightHeight = right != null ? right.height() : 0;
                        return 1 + Math.max(leftHeight, rightHeight);
                }
        }

        public static void main(String[] args) {
                int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

                TreeNode root = createMinimalBST(array);
                System.out.println("Root? " + root.val);
                System.out.println("Created BST? " + root.isBST());
                System.out.println("Height: " + root.height());
        }
}
```

4.3
Answer:

```
public class Question4_3 {
        // DFS time O(N)
        public static void createLevelLinkedList1(TreeNode root,
ArrayList<LinkedList<TreeNode>> lists, int level) {
                if (root == null) {
                        return;
                }

                LinkedList<TreeNode> list = null;
                if (lists.size() == level) {
                        list = new LinkedList<TreeNode>();
                        lists.add(list);
                } else {
                        list = lists.get(level);
                }
```

```
                list.add(root);
                createLevelLinkedList1(root.left, lists, level + 1);
                createLevelLinkedList1(root.right, lists, level + 1);
        }

        public static ArrayList<LinkedList<TreeNode>> createLevelLinkedList1(TreeNode root) {
                ArrayList<LinkedList<TreeNode>> lists = new
ArrayList<LinkedList<TreeNode>>();
                createLevelLinkedList1(root, lists, 0);
                return lists;
        }
        // BFS time O(N)
        public static ArrayList<LinkedList<TreeNode>> createLevelLinkedList2(TreeNode root) {
                ArrayList<LinkedList<TreeNode>> rst = new ArrayList<LinkedList<TreeNode>>();
                LinkedList<TreeNode> current = new LinkedList<TreeNode>();
                if (root != null) {
                        current.add(root);
                }

                while (current.size() > 0) {
                        rst.add(current);
                        LinkedList<TreeNode> parents = current;
                        current = new LinkedList<TreeNode>();
                        for (TreeNode parent : parents) {
                                if (parent.left != null) {
                                        current.add(parent.left);
                                }
                                if (parent.right != null) {
                                        current.add(parent.right);
                                }
                        }
                }
                return rst;
        }

        public static TreeNode createTreeFromArray(int[] array) {
                if (array.length > 0) {
                        TreeNode root = new TreeNode(array[0]);
                        Queue<TreeNode> queue = new LinkedList<TreeNode>();
                        queue.add(root);
                        boolean done = false;
                        int i = 1;
                        while (!done) {
                                TreeNode r = (TreeNode) queue.element();
                                if (r.left == null) {
                                        r.left = new TreeNode(array[i]);
                                        i++;
                                        queue.add(r.left);
                                } else if (r.right == null) {
```

```
                                        r.right = new TreeNode(array[i]);
                                        i++;
                                        queue.add(r.right);
                                } else {
                                        queue.remove();
                                }
                                if (i == array.length) {
                                        done = true;
                                }
                        }
                        return root;
                } else {
                        return null;
                }
        }

        public static void printResult(ArrayList<LinkedList<TreeNode>> rst) {
                int depth = 0;
                for (LinkedList<TreeNode> entry : rst) {
                        Iterator<TreeNode> i = entry.listIterator();
                        System.out.print("Linked List at depth " + depth + ":");
                        while(i.hasNext()) {
                                System.out.print(" " + ((TreeNode)i.next()).val);
                        }
                        System.out.println();
                        depth++;
                }
        }

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

        public static void main(String[] args) {
                int[] nodes_flattened = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
                TreeNode root = createTreeFromArray(nodes_flattened);
                ArrayList<LinkedList<TreeNode>> list = createLevelLinkedList2(root);
                printResult(list);
        }
}
/*
 * outputs:
 * Linked List at depth 0: 1
 * Linked List at depth 1: 2 3
```

```
 * Linked List at depth 2: 4 5 6 7
 * Linked List at depth 3: 8 9 10
 *
 */
```

4.4

```
/*
In the average case, for a balanced binary tree
T(n) = 2T(n/2) + Θ(1);
Every recursive call gives you two problems of half the size. By master theorem, this would evaluate to T(n) = Θ(n)
In the worst case, where each node has only one child.
T(n) = T(n-1) + Θ(1)
Which evaluates to T(n) = Θ(n)
*/
/*
T(n) = 2T(n / 2) + O(1)
    = 2(2T(n / 4) + O(1)) + O(1) = 4T(n / 4) + 2O(1)
    …
    = nT(1) + nO(1) = O(n)
*/
/*
首先如果一个root的树有n个点，求height是O(n) ->每个点走一遍
然后看第7行主程序，设复杂度的是f(n)，那么有:
f(n) = 2 * n/2 + 2*f(n/2)  --> 第一个2 * n/2是第10行求两个子树的高度, f(n/2) 是14行的recursion
     = n + 2 * (2 * n / 4 + 2 * f(n/4))
     = n + n + 4 * f(n/4) = ... = O(nlogn)
*/
Answer:
public class Question4_4 {
        // SOL 1 time O(N log N) ? O(N^2)
        public static int getHeight(TreeNode root) {
                if (root == null) return 0;
                return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
        }

        public static boolean isBalanced1(TreeNode root) {
                if (root == null) return true;

                int heightDiff = getHeight(root.left) - getHeight(root.right);
                if (Math.abs(heightDiff) > 1) {
                        return false;
                } else {
                        return isBalanced1(root.left) && isBalanced1(root.right);
                }
        }
        // SOL 2 time O(N) time O(H)
        public static int checkHeight(TreeNode root) {
                if (root == null) {
```

```
                return 0;
        }

        int leftHeight = checkHeight(root.left);
        if (leftHeight == - 1) {
                return -1;
        }
        int rightHeight = checkHeight(root.right);
        if (rightHeight == - 1) {
                return -1;
        }

        int heightDiff = leftHeight - rightHeight;
        if (Math.abs(heightDiff) > 1) {
                return -1;
        } else {
                return Math.max(leftHeight, rightHeight) + 1;
        }
}

public static boolean isBalanced2(TreeNode root) {
        if (checkHeight(root) == - 1) {
                return false;
        } else {
                return true;
        }
}

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

        public void insertInOrder(int d) {
                if (d <= val) {
                        if (left == null) {
                                TreeNode tmp = new TreeNode(d);
                                this.left = tmp;
                        } else {
                                left.insertInOrder(d);
                        }
                } else {
                        if (right == null) {
                                TreeNode tmp = new TreeNode(d);
                                this.right = tmp;
                        } else {
```

```
                                        right.insertInOrder(d);
                                    }
                                }
                            }
                        }

                        public static int randomInt(int n) {
                            return (int) (Math.random() * n);
                        }

                        public static int randomIntInRange(int min, int max) {
                            return randomInt(max + 1 - min) + min;
                        }

                        public static void main(String[] args) {
                            TreeNode unbalanced = new TreeNode(10);
                            for (int i = 0; i < 10; i++) {
                                unbalanced.insertInOrder(randomIntInRange(0, 100));
                            }
                            System.out.println("Root? " + unbalanced.val);
                            System.out.println("Is balanced? " + isBalanced1(unbalanced));
                        }
}
```

4.5
Answer:

```
public class Question4_5 {
        // SOL 1
        public static Integer last_printed = null;

        public static boolean checkBST1(TreeNode n) {
            if (n == null) {
                return true;
            }

            if (!checkBST1(n.left)) {
                return false;
            }

            if (last_printed != null && n.val <= last_printed) {
                return false;
            }
            last_printed = n.val;

            if (!checkBST1(n.right)) {
                return false;
            }
            return true;
        }
```

```
	// SOL 2 time O(N) space O(log N)
	public static boolean checkBST2(TreeNode n) {
		return checkBST2(n, null, null);
	}

	public static boolean checkBST2(TreeNode n, Integer min, Integer max) {
		if (n == null) {
			return true;
		}
		if ((min != null && n.val <= min) || (max != null && n.val > max)) {
			return false;
		}
		if ((!checkBST2(n.left, min, n.val)) || (!checkBST2(n.right, n.val, max))) {
			return false;
		}
		return true;
	}

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

	public static TreeNode createMinimalBST(int array[]) {
		return createMinimalBST(array, 0, array.length - 1);
	}

	public static TreeNode createMinimalBST(int arr[], int start, int end) {
		if (end < start) {
			return null;
		}

		int mid = (start + end) / 2;
		TreeNode n = new TreeNode(arr[mid]);
		n.left = createMinimalBST(arr, start, mid - 1);
		n.right = createMinimalBST(arr, mid + 1, end);
		return n;
	}

	public static void main(String[] args) {
		int[] array = {Integer.MIN_VALUE, Integer.MAX_VALUE - 2, Integer.MAX_VALUE
- 1, Integer.MAX_VALUE};
		TreeNode node = createMinimalBST(array);
		System.out.println(checkBST2(node));
	}
```

```
}
```

4.6
Answer:

```
public class Question4_6 {

	public static TreeNode inorderSucc(TreeNode n) {
		if (n == null) return null;

		if (n.parent == null || n.right != null) {
			return leftMostChild(n.right);
		} else {
			TreeNode q = n;
			TreeNode x = q.parent;
			while (x != null && x.left != q) {
				q = x;
				x = x.parent;
			}
			return x;
		}
	}

	public static TreeNode leftMostChild(TreeNode n) {
		if (n == null) {
			return null;
		}

		while (n.left != null) {
			n = n.left;
		}
		return n;
	}

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

	public static TreeNode createMinimalBST(int array[]) {
		return createMinimalBST(array, 0, array.length - 1);
	}

	public static TreeNode createMinimalBST(int arr[], int start, int end) {
		if (end < start) {
```

```
                return null;
            }

            int mid = (start + end) / 2;
            TreeNode n = new TreeNode(arr[mid]);
            n.left = createMinimalBST(arr, start, mid - 1);
            n.right = createMinimalBST(arr, mid + 1, end);
            return n;
        }

        public static TreeNode find(int d, TreeNode root) {
            if (d == root.val) {
                return root;
            } else if (d <= root.val) {
                return root.left != null ? find(d, root.left) : null;
            } else if (d > root.val) {
                return root.right != null ? find(d, root.right) : null;
            }
            return null;
        }

        public static void main(String[] args) {
            int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

            TreeNode root = createMinimalBST(array);

            for (int i = 0; i < array.length; i++) {
                TreeNode node = find(array[i], root);
                TreeNode next = inorderSucc(node);
                if (next != null) {
                    System.out.println(node.val + "->" + next.val);
                } else {
                    System.out.println(node.val + "->" + null);
                }
            }
        }
}
```

4.7
Answer:

```
// SOL 1 time O(P + D)
public class Question4_7_A {
        public static Project[] findBuildOrder(String[] projects, String[][] dependencies) {
            Graph graph = buildGraph(projects, dependencies);
            return orderProjects(graph.getNodes());
        }

        public static Graph buildGraph(String[] projects, String[][] dependencies) {
            Graph graph = new Graph();
```

```
		for (String project : projects) {
			graph.getOrCreateNode(project);
		}

		for (String[] dependency : dependencies) {
			String first = dependency[0];
			String second = dependency[1];
			graph.addEdge(first, second);
		}

		return graph;
	}

	public static Project[] orderProjects(ArrayList<Project> projects) {
		Project[] order = new Project[projects.size()];
		int endOfList = addNonDependent(order, projects, 0);
		int toBeProcessed = 0;
		while (toBeProcessed < order.length) {
			Project current = order[toBeProcessed];

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

			ArrayList<Project> children = current.getChildren();
			for (Project child : children) {
				child.decrementDependencies();
			}

			endOfList = addNonDependent(order, children, endOfList);
			toBeProcessed++;
		}

		return order;
	}

	public static int addNonDependent(Project[] order, ArrayList<Project> projects, int offset)
{
		for (Project project : projects) {
			if (project.getNumberDependencies() == 0) {
				order[offset] = project;
				offset++;
			}
		}
		return offset;
	}

	public static class Graph {
		private ArrayList<Project> nodes = new ArrayList<Project>();
```

```
        private HashMap<String, Project> map = new HashMap<String, Project>();

        public Project getOrCreateNode(String name) {
            if (!map.containsKey(name)) {
                Project node = new Project(name);
                nodes.add(node);
                map.put(name, node);
            }

            return map.get(name);
        }

        public void addEdge(String startName, String endName) {
            Project start = getOrCreateNode(startName);
            Project end = getOrCreateNode(endName);
            start.addNeighbor(end);
        }

        public ArrayList<Project> getNodes() { return nodes; }
    }

    public static class Project {
        private ArrayList<Project> children = new ArrayList<Project>();
        private HashMap<String, Project> map = new HashMap<String, Project>();
        private String name;
        private int dependencies = 0;

        public Project(String n) {
            name = n;
        }

        public void addNeighbor(Project node) {
            if (!map.containsKey(node.getName())) {
                children.add(node);
                node.incrementDependencies();
            }
        }

        public void incrementDependencies() {
            dependencies++;
        }
        public void decrementDependencies() {
            dependencies--;
        }

        public String getName() {
            return name;
        }
        public ArrayList<Project> getChildren() {
```

```
                return children;
            }
            public int getNumberDependencies() {
                return dependencies;
            }
        }
}
// SOL 2 DFS time O(P + D)
public class Question4_7_B {

    public static Stack<Project> findBuildOrder(String[] projects, String[][] dependencies) {
        Graph graph = buildGraph(projects, dependencies);
        return orderProjects(graph.getNodes());
    }

    public static Stack<Project> orderProjects(ArrayList<Project> projects) {
        Stack<Project> stack = new Stack<Project>();
        for (Project project : projects) {
            if (project.getState() == Project.State.BLANK) {
                if (!doDFS(project, stack)) {
                    return null;
                }
            }
        }
        return stack;
    }

    public static boolean doDFS(Project project, Stack<Project> stack) {
        if (project.getState() == Project.State.PARTIAL) {
            return false;
        }

        if (project.getState() == Project.State.BLANK) {
            project.setState(Project.State.PARTIAL);
            ArrayList<Project> children = project.getChildren();
            for (Project child : children) {
                if (!doDFS(child, stack)) {
                    return false;
                }
            }
            project.setState(Project.State.COMPLETE);
            stack.push(project);
        }
        return true;
    }

    public static Graph buildGraph(String[] projects, String[][] dependencies) {
        Graph graph = new Graph();
        for (String project : projects) {
```

```
                graph.getOrCreateNode(project);
        }

        for (String[] dependency : dependencies) {
                String first = dependency[0];
                String second = dependency[1];
                graph.addEdge(first, second);
        }

        return graph;
}

public static class Graph {
        private ArrayList<Project> nodes = new ArrayList<Project>();
        private HashMap<String, Project> map = new HashMap<String, Project>();

        public Project getOrCreateNode(String name) {
                if (!map.containsKey(name)) {
                        Project node = new Project(name);
                        nodes.add(node);
                        map.put(name, node);
                }

                return map.get(name);
        }

        public void addEdge(String startName, String endName) {
                Project start = getOrCreateNode(startName);
                Project end = getOrCreateNode(endName);
                start.addNeighbor(end);
        }

        public ArrayList<Project> getNodes() { return nodes; }
}

public static class Project {
        public enum State {COMPLETE, PARTIAL, BLANK};
        private State state = State.BLANK;
        public State getState() {
                return state;
        }
        public void setState(State st) {
                state = st;
        }

        private ArrayList<Project> children = new ArrayList<Project>();
        private HashMap<String, Project> map = new HashMap<String, Project>();
        private String name;
        private int dependencies = 0;
```

```
		public Project(String n) {
			name = n;
		}

		public void addNeighbor(Project node) {
			if (!map.containsKey(node.getName())) {
				children.add(node);
				node.incrementDependencies();
			}
		}

		public void incrementDependencies() {
			dependencies++;
		}
		public void decrementDependencies() {
			dependencies--;
		}

		public String getName() {
			return name;
		}
		public ArrayList<Project> getChildren() {
			return children;
		}
		public int getNumberDependencies() {
			return dependencies;
		}
	}
}
```

4.8 Leetcode: lowest common ancestor

Answer:

```
public class Question4_8 {
	// SOL 1 balanced tree time O((log N)^2), unbalanced tree time O(N^2)
	public static TreeNode commonAncestor1(TreeNode p, TreeNode q) {
		if (p == q) {
			return null;
		}

		TreeNode ancestor = p;
		while (ancestor != null) {
			if (isOnPath(ancestor, q)) {
				return ancestor;
			}
			ancestor = ancestor.parent;
		}
		return null;
	}
```

```
public static boolean isOnPath(TreeNode ancestor, TreeNode q) {
	while (q != ancestor && q != null) {
		q = q.parent;
	}
	return q == ancestor;
}
// SOL 1 B
public static TreeNode commonAncestor1B(TreeNode root, TreeNode p, TreeNode q) {
	ArrayList<TreeNode> list1 = getPath2Root(p);
	ArrayList<TreeNode> list2 = getPath2Root(q);
	int i, j;
	for (i = list1.size() - 1, j = list2.size() - 1; i >= 0 && j >= 0; i--, j--) {
		if (list1.get(i) != list2.get(j)) {
			return list1.get(i).parent;
		}
	}
	return list1.get(i + 1);
}

public static ArrayList<TreeNode> getPath2Root(TreeNode node) {
	ArrayList<TreeNode> list = new ArrayList<TreeNode>();
	while (node != null) {
		list.add(node);
		node = node.parent;
	}
	return list;
}
// SOL 2 time O(t), t is the size of the subtree for the first common ancestor
public static TreeNode commonAncestor2(TreeNode root, TreeNode p, TreeNode q) {
	if (!covers(root, p) || !covers(root, q)) {
		return null;
	} else if (covers(p, q)) {
		return p;
	} else if (covers(q, p)) {
		return q;
	}

	TreeNode sibling = getSibling(p);
	TreeNode parent = p.parent;
	while (!covers(sibling, q)) {
		sibling = getSibling(parent);
		parent = parent.parent;
	}
	return parent;
}

public static boolean covers(TreeNode root, TreeNode node) {
	if (root == null) return false;
```

```
                if (root == node) return true;
                return covers(root.left, node) || covers(root.right, node);
        }

        public static TreeNode getSibling(TreeNode node) {
                if (node == null || node.parent == null) {
                        return null;
                }
                TreeNode parent = node.parent;
                return parent.left == node ? parent.right : parent.left;
        }
   // SOL 3 time O(n) 2n / 2 + 2n / 4 + ... -> n
        public static TreeNode commonAncestorHelper(TreeNode root, TreeNode p, TreeNode
q) {
                if (root == null) {
                        return null;
                }
                boolean is_p_on_left = covers(root.left, p);
                boolean is_q_on_left = covers(root.left, q);
                if (is_p_on_left != is_q_on_left) { // Nodes are on different side
                        return root;
                }
                TreeNode child_side = is_p_on_left ? root.left : root.right;
                return commonAncestorHelper(child_side, p, q);
        }

        public static TreeNode commonAncestor3(TreeNode root, TreeNode p, TreeNode q) {
                if (!covers(root, p) || !covers(root, q)) {
                        return null;
                }
                return commonAncestorHelper(root, p, q);
        }
        // SOL 4
        public static TreeNode commonAncestor4(TreeNode root, TreeNode p, TreeNode q) {
                Result r = commonAncHelper(root, p, q);
                if (r.isAncestor) {
                        return r.node;
                }
                return null;
        }

        public static Result commonAncHelper(TreeNode root, TreeNode p, TreeNode q) {
                if (root == null) {
                        return new Result(null, false);
                }
                if (root == p && root == q) {
                        return new Result(root, true);
                }
```

```
		Result rx = commonAncHelper(root.left, p, q);
		if (rx.isAncestor) {
			return rx;
		}

		Result ry = commonAncHelper(root.right, p, q);
		if (ry.isAncestor) {
			return ry;
		}

		if (rx.node != null && ry.node != null) {
			return new Result(root, true);
		} else if (root == p || root == q) {
			boolean isAncestor = rx.node != null || ry.node != null;
			return new Result(root, isAncestor);
		} else {
			return new Result(rx.node != null ? rx.node : ry.node, false);
		}
	}

	public static class Result {
		public TreeNode node;
		public boolean isAncestor;
		public Result(TreeNode n, boolean isAnc) {
			node = n;
			isAncestor = isAnc;
		}
	}
	// SOL 5
	public static TreeNode commonAncestor5(TreeNode root, TreeNode p, TreeNode q) {
		if (root == null || root == p || root == q) {
			return root;
		}

		TreeNode left = commonAncestor5(root.left, p, q);
		TreeNode right = commonAncestor5(root.right, p, q);
		if (left != null && right != null) {
			return root;
		}
		if (left != null) {
			return left;
		}
		if (right != null) {
			return right;
		}
		return null;
	}

	public static class TreeNode {
```

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

public static TreeNode createMinimalBST(int array[]) {
        return createMinimalBST(array, 0, array.length - 1);
}

public static TreeNode createMinimalBST(int arr[], int start, int end) {
        if (end < start) {
                return null;
        }

        int mid = (start + end) / 2;
        TreeNode n = new TreeNode(arr[mid]);
        TreeNode tmp1 = createMinimalBST(arr, start, mid - 1);
        n.left = tmp1;
        if (tmp1 != null) {
                tmp1.parent = n;
        }
        TreeNode tmp2 = createMinimalBST(arr, mid + 1, end);
        n.right = tmp2;
        if (tmp2 != null) {
                tmp2.parent = n;
        }
        return n;
}

public static TreeNode find(int d, TreeNode root) {
        if (d == root.val) {
                return root;
        } else if (d <= root.val) {
                return root.left != null ? find(d, root.left) : null;
        } else if (d > root.val) {
                return root.right != null ? find(d, root.right) : null;
        }
        return null;
}

public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        TreeNode root = createMinimalBST(array);
        TreeNode n3 = find(10, root);
        TreeNode n7 = find(6, root);
```

```
                TreeNode n0 = new TreeNode(100);
                TreeNode ancestor = commonAncestor4(root, n3, n0);
                if (ancestor != null) {
                        System.out.println(ancestor.val);
                } else {
                        System.out.println("null");
                }
        }
}
```

4.9
Answer:

```
public class Question4_9 {
        public static ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
                ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();

                if (node == null) {
                        result.add(new LinkedList<Integer>());
                        return result;
                }

                LinkedList<Integer> prefix = new LinkedList<Integer>();
                prefix.add(node.val);

                ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
                ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

                for (LinkedList<Integer> left : leftSeq) {
                        for (LinkedList<Integer> right : rightSeq) {
                                ArrayList<LinkedList<Integer>> weaved = new
ArrayList<LinkedList<Integer>>();
                                weaveLists(left, right, weaved, prefix);
                                result.addAll(weaved);
                        }
                }
                return result;
        }

        public static void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
                        ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
                if (first.size() == 0 || second.size() == 0) {
                        LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
                        result.addAll(first);
                        result.addAll(second);
                        results.add(result);
                        return;
                }

                int headFirst = first.removeFirst();
```

```
                prefix.addLast(headFirst);
                weaveLists(first, second, results, prefix);
                prefix.removeLast();
                first.addFirst(headFirst);

                int headSecond = second.removeFirst();
                prefix.addLast(headSecond);
                weaveLists(first, second, results, prefix);
                prefix.removeLast();
                second.addFirst(headSecond);
        }

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

        public static void main(String[] args) {
                TreeNode n1 = new TreeNode(1);
                TreeNode n2 = new TreeNode(2);
                TreeNode n3 = new TreeNode(3);
                n2.left = n1;
                n2.right = n3;
                System.out.print(allSequences(n2).toString());
        }
}
```

4.10
Answer:

```
// time O(nm) space O(log(n) + log(m))
public class Question4_10 {
        public static boolean containsTree(TreeNode t1, TreeNode t2) {
                if (t2 == null) {
                        return true;
                }
                return subTree(t1, t2);
        }

        public static boolean subTree(TreeNode r1, TreeNode r2) {
                if (r1 == null) {
                        return false;
                } else if (r1.val == r2.val && matchTree(r1, r2)) {
                        return true;
                }
                return (subTree(r1.left, r2) || subTree(r1.right, r2));
```

```
}

public static boolean matchTree(TreeNode r1, TreeNode r2) {
        if (r2 == null && r1 == null) {
                return true;
        } else if (r1 == null || r2 == null) {
                return false;
        } else if (r1.val != r2.val) {
                return false;
        } else {
                return (matchTree(r1.left, r2.left) && matchTree(r1.right, r2.right));
        }
}

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

public static TreeNode createTreeFromArray(int[] array) {
        if (array.length > 0) {
                TreeNode root = new TreeNode(array[0]);
                Queue<TreeNode> queue = new LinkedList<TreeNode>();
                queue.add(root);
                boolean done = false;
                int i = 1;
                while (!done) {
                        TreeNode r = (TreeNode) queue.element();
                        if (r.left == null) {
                                r.left = new TreeNode(array[i]);
                                i++;
                                queue.add(r.left);
                        } else if (r.right == null) {
                                r.right = new TreeNode(array[i]);
                                i++;
                                queue.add(r.right);
                        } else {
                                queue.remove();
                        }
                        if (i == array.length) {
                                done = true;
                        }
                }
                return root;
        } else {
```

```
                return null;
            }
        }

        public static void main(String[] args) {
            int[] array1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
            int[] array2 = {2, 4, 5, 8, 9, 10, 11};
            TreeNode t1 = createTreeFromArray(array1);
            TreeNode t2 = createTreeFromArray(array2);

            if (containsTree(t1, t2)) {
                System.out.println("t2 is a subtree of t1");
            } else {
                System.out.println("t2 is not a subtree of t1");
            }

            int[] array3 = {1, 2, 3};
            TreeNode t3 = createTreeFromArray(array1);
            TreeNode t4 = createTreeFromArray(array3);

            if (containsTree(t3, t4)) {
                System.out.println("t4 is a subtree of t3");
            } else {
                System.out.println("t4 is not a subtree of t3");
            }
        }
}
```

4.11
Answer:

```
public class Question4_11 {
        // SOL 1 time O(logN)
        public static class TreeNode1 {
            private int data;
            public TreeNode1 left;
            public TreeNode1 right;
            private int size = 0;

            public TreeNode1(int d) {
                data = d;
                size = 1;
            }

            public TreeNode1 getRandomNode() {
                int leftSize = left == null ? 0 : left.size();
                Random random = new Random();
                int index = random.nextInt(size);
                if (index < leftSize) {
                    return left.getRandomNode();
```

```
                } else if (index == leftSize) {
                        return this;
                } else {
                        return right.getRandomNode();
                }
        }

        public void insertInOrder(int d) {
                if (d <= data) {
                        if (left == null) {
                                left = new TreeNode1(d);
                        } else {
                                left.insertInOrder(d);
                        }
                } else {
                        if (right == null) {
                                right = new TreeNode1(d);
                        } else {
                                right.insertInOrder(d);
                        }
                }
                size++;
        }

        public int size() {
                return size;
        }

        public int data() {
                return data;
        }

        public TreeNode1 find(int d) {
                if (d == data) {
                        return this;
                } else if (d <= data) {
                        return left != null ? left.find(d) : null;
                } else if (d > data) {
                        return right != null ? right.find(d) : null;
                }
                return null;
        }
}
// SOL 2 time O(D)
public static class Tree2 {
        TreeNode2 root = null;

        public int size() {
                return root == null ? 0 : root.size();
```

```
        }

        public TreeNode2 getRandomNode() {
            if (root == null) return null;

            Random random = new Random();
            int i = random.nextInt(size());
            return root.getIthNode(i);
        }

        public void insertInOrder(int value) {
            if (root == null) {
                root = new TreeNode2(value);
            } else {
                root.insertInOrder(value);
            }
        }
    }

    public static class TreeNode2 {
        private int data;
        public TreeNode2 left;
        public TreeNode2 right;
        private int size = 0;

        public TreeNode2(int d) {
            data = d;
            size = 1;
        }

        public TreeNode2 getIthNode(int i) {
            int leftSize = left == null ? 0 : left.size();
            if (i < leftSize) {
                return left.getIthNode(i);
            } else if (i == leftSize) {
                return this;
            } else {
                return right.getIthNode(i - (leftSize + 1));
            }
        }

        public void insertInOrder(int d) {
            if (d <= data) {
                if (left == null) {
                    left = new TreeNode2(d);
                } else {
                    left.insertInOrder(d);
                }
            } else {
```

```
                        if (right == null) {
                                right = new TreeNode2(d);
                        } else {
                                right.insertInOrder(d);
                        }
                }
                size++;
        }

        public int size() {
                return size;
        }

        public TreeNode2 find(int d) {
                if (d == data) {
                        return this;
                } else if (d <= data) {
                        return left != null ? left.find(d) : null;
                } else if (d > data) {
                        return right != null ? right.find(d) : null;
                }
                return null;
        }
    }
}
```

4.12
Answer:

```
public class Question4_12 {
    // SOL 1 Brute Force balanced time O(N log N), unbalanced time O(N^2)
    public static int countPathsWithSum1(TreeNode root, int targetSum) {
        if (root == null) return 0;

        int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);

        int pathsOnLeft = countPathsWithSum1(root.left, targetSum);
        int pathsOnRight = countPathsWithSum1(root.right, targetSum);

        return pathsFromRoot + pathsOnLeft + pathsOnRight;
    }

    public static int countPathsWithSumFromNode(TreeNode node, int targetSum, int
currentSum) {
        if (node == null) return 0;

        currentSum += node.val;

        int totalPaths = 0;
        if (currentSum == targetSum) {
```

```
                totalPaths++;
            }

            totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
            totalPaths += countPathsWithSumFromNode(node.right, targetSum,
currentSum);
            return totalPaths;
        }
        // SOL 2 DFS + HashMap time O(N)
        public static int countPathsWithSum2(TreeNode root, int targetSum) {
            if (root == null) return 0;
            HashMap<Integer, Integer> pathCount = new HashMap<Integer, Integer>();
            incrementHashTable(pathCount, 0, 1);
            return countPathsWithSum2(root, targetSum, 0, pathCount);
        }

        public static int countPathsWithSum2(TreeNode node, int targetSum, int runningSum,
HashMap<Integer, Integer> pathCount) {
            if (node == null) return 0;

            runningSum += node.val;
            incrementHashTable(pathCount, runningSum, 1);

            int sum = runningSum - targetSum;
            int totalPaths = pathCount.containsKey(sum) ? pathCount.get(sum) : 0;

            totalPaths += countPathsWithSum2(node.left, targetSum, runningSum,
pathCount);
            totalPaths += countPathsWithSum2(node.right, targetSum, runningSum,
pathCount);

            incrementHashTable(pathCount, runningSum, -1);
            return totalPaths;
        }

        public static void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int
delta) {
            if (!hashTable.containsKey(key)) {
                hashTable.put(key, 0);
            }
            hashTable.put(key, hashTable.get(key) + delta);
        }

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

```
            }
        }

        public static void main(String[] args) {
            TreeNode n1 = new TreeNode(5);
            TreeNode n2 = new TreeNode(3);
            TreeNode n3 = new TreeNode(1);
            TreeNode n4 = new TreeNode(4);
            TreeNode n5 = new TreeNode(8);
            TreeNode n6 = new TreeNode(2);
            TreeNode n7 = new TreeNode(6);
            n1.left = n2;
            n1.right = n3;
            n2.left = n4;
            n2.right = n5;
            n3.left = n6;
            n3.right = n7;
            System.out.print(countPathsWithSum2(n1, 8));
        }
}
```