第二章：线性表

1. 线性表的概述

➢ **线性表的概念**

线性表属于最基本、最简单、也是最常用的一种数据结构，从逻辑上划分它属于线性结构。一个线性表是由 n 个具有相同特性的数据元素组成的有限序列，数据元素之间具有一种线性的或“一对一”的逻辑关系，如下图所示：

从严谨的角度上来讲，线性表应该满足以下三个要求：

(1) 第一个数据元素没有直接前驱，这个数据元素被称为开始节点；

(2) 最后一个数据元素没有直接后继，这个数据元素被称为终端节点；

(3) 除了第一个和最后一个数据元素外，其它数据元素有且仅有一个直接前驱和一个直接后继。

➢ **线性表的特点**

(1) 顺序性

在线性表中，相邻数据元素之间存在着序偶关系，也就是存在着先后关系。

(2) 相同数据类型

在线性表中，每一个数据元素都属于相同数据类型。相同数据类型意味着在内存中存储时，每个元素会占用相同的内存空间，便于后续的查询定位。

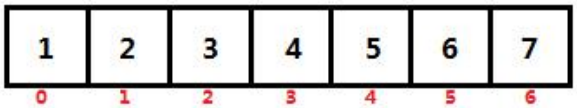
(3) 有限性

在线性表中，数据元素的个数 n 就是为线性表的长度，n 是一个有限值。当 n=0 时线性表为空表。在非空的线性表中每个数据元素在线性表中都有唯一确定的序号，例如第一个元素的序号是 0，第 i 个元素的序号为 i-1。在一个具有 n > 0 个数据元素的线性表中，数据元素序号的范围是[0, n-1]。

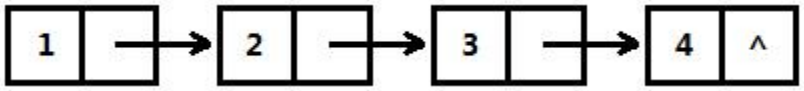
➢ **线性表的存储结构**

线性表拥有两种不同的存储结构，分别为：

(1) 顺序存储结构（顺序表）：顺序表是采用一组地址连续的存储单元来依次存放线性表中的各个元素。



(2) 链式存储结构（链表）：链表中的存储元素的地址不一定是连续的，元素节点中存放数据元素以及相邻元素的地址信息。



2. 顺序表

2.1 顺序表的特点

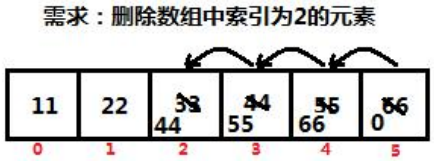
- **顺序表根据索引查询元素的特点**

因为顺序表（此处，也就是数组）的内存空间是连续的，并且存储的数据属于相同数据类型。因此，我们可以通过数组的“首地址+索引”就能快速的找到数组中对应索引的元素值，从而得出数组的优点：查找快。

索引操作数组原理：数组首地址 + 存放数据的字节数*索引。

- **顺序表删除元素的特点**

需求：删除数组{11， 22， 33， 44， 55， 66}索引为 2 的元素，删除后的数组为：{11， 22， 44， 55， 66}。



实现步骤：

(1) 把删除索引之后的元素往前挪动一位（从前往后）。

(2) 把数组的最后一个元素设置为默认值。

- **顺序表插入元素的特点**

需求：在数组{11， 22， 44， 55， 66}索引为 2 的位置插入元素 33，插入后的数组为：{11， 22， 33， 44， 55， 66}。

实现步骤：

(1) 先检查数组是否需要扩容。如果数组的空间长度和数组添加元素个数相等，则需做扩容操作。

1．定义一个空间长度更大的新数组。

2．把原数组中的元素全部拷贝到新数组中。

3．让原数组指向新数组，也就是让原数组保存新数组的地址值。

(2) 把插入索引及其之后的元素往后挪动一位（从后往前挪动）。

(3) 把插入的元素赋值到插入索引的位置中。

➢ **顺序表优劣势的总结**

优点：无须关心表中元素之间的关系，所以不用增加额外的存储空间；可以根据索引快速地操作表中任意位置的元素，并且操作任何一个元素的耗时都是一样。

缺点：插入和删除操作需要移动大量元素。使用前需事先分配好内存空间，当线性表长度变化较大时，难以确定存储空间的容量。分配空间过大会造成存储空间的巨大浪费，分配的空间过小，难以适应问题的需求。

2.2 模拟 ArrayList 实现

```
package com.bjsxt.p1.arraylist;
/**
 * 需要实现以下几个方法
 *  1)添加元素的方法  --> add(Object element)
 *  2)根据索引获取元素的方法  --> get(int index)
 *  3)根据索引删除元素的方法  --> remove(int index)
 *  4)根据索引插入元素的方法  --> add(int index, Object element)
 */
public class ArrayList {
    /**
     * 定义一个数组，用于保存集合中的数据
     */
    private Object[] elementData;
    /**
     * 定义一个变量，用于保存数组中实际存放元素的个数
     */
    private int size;

    /**
     * 获取数组中实际存放元素的个数
     * @return
     */
    public int size() {
        return this.size;
    }

    /**
     * 无参构造方法（默认设置 elementData 数组的空间长度为 10）
     */
    public ArrayList() {
        this.elementData = new Object[10];
    }
```

```
    /**
     * 有参构造方法（指定设置 elementData 数组的空间长度）
     * @param cap 需要设置 elementData 的空间长度
     */
    public ArrayList(int cap) {
        // 1.判断 cap 变量是否合法
        if(cap < 0)
            throw new RuntimeException("参数不合法，cap:" + cap);
        // 2.实例化 elementData 数组
        this.elementData = new Object[cap];
    }

    /**
     * 添加元素
     * @param element 需要添加的元素
     */
    public void add(Object element) {
        // 1.判断数组是否需要扩容
        ensureCapacityInternal();
        // 2.把 element 添加进入数组中
        elementData[size] = element;
        // 3.更新 size 的值
        size++;
    }

    /**
     * 根据索引获取元素值
     * @param index 索引值
     * @return 数组中 index 索引对应的元素值
     */
    public Object get(int index) {
        // 1.判断索引是否合法，合法的取值范围：[0, size - 1]
        rangeCheck(index);
        // 2.根据索引获取对应的元素值
        return elementData[index];
    }

    /**
     * 根据索引删除元素
     * @param index 索引值
     */
    public void remove(int index) {
        // 1.判断索引是否合法，合法取值范围：[0, size - 1]
        rangeCheck(index);
        // 2.把删除索引之后的元素往前挪动一位
        // 2.1 先获得删除索引及其之后的所有索引值
```

```
        for(int i = index; i < size - 1; i++) { // i = size - 1
            // 2.2 把后一个元素往前挪动一位
            elementData[i] = elementData[i + 1]; // i + 1 = size
        }
        // 3.把最后一个实际添加的元素设置为默认值
        elementData[size - 1] = null;
        // 4.更新 size 的值
        size--;
    }

    /**
     * 检查索引是否合法（get 和 remove）
     * @param index 需要检查的索引值
     */
    private void rangeCheck(int index) {
        // 判断索引是否合法，合法取值范围：[0, size - 1]
        if(index < 0 || index >= size) {
            throw new ArrayIndexOutOfBoundsException("索引异常，index:" + index);
        }
    }

    /**
     * 根据索引插入元素
     * @param index 插入元素的索引位置
     * @param element 需要插入的元素
     */
    public void add(int index, Object element) {
        // 1.判断索引是否合法，合法的取值范围：[0, size]
        // --> 插入的元素可以在实际添加元素的最末尾
        if(index < 0 || index > size) {
            throw new ArrayIndexOutOfBoundsException("索引异常，index:" + index);
        }
        // 2.判断数组是否需要扩容
        ensureCapacityInternal();
        // 3.插入索引及其之后的元素往后挪动一位（从后往前挪动）
        // 3.1 获得插入索引及其之后的所有索引值
        for(int i = size - 1; i >= index; i--) {
            // 3.2 把前一个元素往后挪动一位
            elementData[i + 1] = elementData[i];
        }
        // 4.在插入索引位置实现赋值操作
        elementData[index] = element;
        // 5.更新 size 的值
        size++;
    }
```

```
    /**
     * 判断数组是否需要执行扩容操作
     */
    private void ensureCapacityInternal() {
        // 1.当数组的空间长度等于数组实际存放元素的个数时，这时就需扩容操作
        if(elementData.length == size) {
            // 2.创建一个比原数组空间长度更大的新数组
            Object[] newArr = new Object[elementData.length * 2 + 1];
            // 3.把原数组中的元素拷贝进入新数组中
            for(int i = 0; i < size; i++) {
                newArr[i] = elementData[i];
            }
            // 4.让原数组保存新数组的地址值
            elementData = newArr;
        }
    }
}
```

2.3 经典面试题

2.3.1 数组的反转

- **实现方案一**

引入一个外部数组变量，用于保存反序后的数组，然后把原数组中的元素倒序保存于新创建的数组中，新建数组保存的元素就是反转之后的结果。

- **代码实现**

```
package com.bjsxt.p2.arraytest;
import java.util.Arrays;
public class Test01 {
    public static void main(String[] args) {
        int[] arr = {11, 22, 33, 44, 55, 66};
        int[] newArr = reverseOrderArray(arr);
        System.out.println(Arrays.toString(newArr));
    }
    /**
     * 实现数组的反转
     * @param arr 需要反转的数组
     * @return 反转之后的数组
     */
    public static int[] reverseOrderArray(int[] arr) {
        // 1.定义一个新数组，用于保存反转之后的结果
        int[] newArr = new int[arr.length];
        // 2.把 arr 数组中的所有元素倒序的存入 newArr 数组中
        // 2.1 通过循环获得 arr 数组中的每一个元素
        for(int i = 0; i < arr.length; i++) {
```

```
            // 2.2 把 arr 数组中的元素倒序存入 newArr 数组中
            newArr[arr.length - 1 - i] = arr[i];
        }
        // 3.把反转之后的数组返回
        return newArr;
    }
}
```

- **实现方案二**

直接对数组中的元素进行收尾交换。这样避免了新建一个数组来保存反转之后的结果，并且循环遍历的次数也降为“实现方案一”的一半，从而提高了算法的效率。

- **代码实现**

```
package com.bjsxt.p2.arraytest;
import java.util.Arrays;
public class Test02 {
    public static void main(String[] args) {
        int[] arr = {11, 22, 33, 44, 55, 66};
        reverseOrderArray(arr);
        System.out.println(Arrays.toString(arr));
    }
    /**
     * 实现数组的反转
     * @param arr 需要反转的数组
     */
    public static void reverseOrderArray(int[] arr) {
        // 1.通过循环，获得数组前半部分的元素
        for(int i = 0; i < arr.length / 2; i++) {
            // 2.把 arr[i]和 arr[arr.length - 1 - i]做交换
            int temp = arr[i];
            arr[i] = arr[arr.length - 1 - i];
            arr[arr.length - 1 - i] = temp;
        }
    }
}
```

2.3.2 找数组中重复的元素

- **题目描述**

在一个长度为 n 的数组里的所有元素都在[0, n-1]范围内。数组中某些元素是重复的，但不知道有几个元素是重复的，也不知道每个元素重复几次。请找出数组中任意一个重复的元素。例如，如果输入长度为 6 的数组{0, 2, 4, 1, 4, 3}，那么输出重复的元素 4。

注意：数据不合法和找不到重复数据的情况，则输出-1 即可。

- **思路分析**

从头到尾依次扫描数组中每一个元素。当扫描到第 i 个元素时，比较该位置数值 m 是否等于 i。若是，接着扫描下一个元素；否则，将其与第 m 个元素进行比较。若相等，则返回该重复元素；否

则，交换两个元素，继续重复前面的过程。

如果数组元素不重复，则每个元素和对应的索引值肯定相等，而我们要做的就是：把数组元素放在正确的位置上面。

- **代码实现**

```
package com.bjsxt.p2.arraytest;
public class Test03 {
    public static void main(String[] args) {
        int[] arr = {0, 3, 4, 1, 2, 8};
        int repeatNumber = getRepeatNumber(arr);
        System.out.println(repeatNumber);
    }

    /**
     * 查找数组中的重复元素
     * @param arr 需要查找的数组
     * @return 返回数组中重复的元素值。如果元素值不合法和找不到重复元素的情况，则返回-1 即可。
     */
    public static int getRepeatNumber(int[] arr) {
        // 1.判断 arr 为 null 或 arr.length 等于 0 的情况
        if(arr == null || arr.length == 0) {
            return -1;
        }
        // 2.通过循环，遍历数组中的所有元素
        for(int i = 0; i < arr.length; ) {
            // 3.判断数组元素是否合法
            if(arr[i] < 0 || arr[i] >= arr.length) {
                return -1;
            }
            // 4.处理 arr[i]等于 i 的情况
            if(arr[i] == i) {
                i++; // 继续遍历数组中的下一个元素
            }
            // 5.处理 arr[i]不等于 i 的情况
            else {
                // 5.1 如果 arr[i]和 arr[arr[i]]相等，则证明找到了重复的元素
                if(arr[i] == arr[arr[i]]) {
                    return arr[i];
                }
                // 5.2 如果 arr[i]和 arr[arr[i]]不相等，则交换 arr[i]和 arr[arr[i]]的值
                else {
                    int temp = arr[i];
                    arr[i] = arr[arr[i]];
                    arr[temp] = temp; // 切记
                }
            }
        }
        // 6.执行到此处，证明数组中就不存在重复的元素
        return -1;
    }
}
```

2.3.3 使奇数位于偶数前面

- **题目描述**

输入一个整型数组，实现一个方法来调整该数组中的元素的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

- **思路分析**

题目要求所有奇数都应该在偶数前面，所以我们应该只需要维护两个下标值，让一个下标值从前往后遍历，另外一个下标值从后往前遍历，当发现第一个下标值对应到偶数，第二个下标值对应到奇数的时候，我们就直接对调两个值。直到第一个下标到了第二个下标的后面的时候退出循环。

- **代码实现**

```
package com.bjsxt.p2.arraytest;
import java.util.Arrays;
public class Test04 {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
        replaceOrderArray(arr);
        System.out.println(Arrays.toString(arr));
    }
    /**
     * 使奇数位于偶数前面
     * @param arr 需要调整奇偶数位置的数组
     */
    public static void replaceOrderArray(int[] arr) {
        // 1.处理 arr 为 null 的情况
        if(arr == null)
            throw new NullPointerException("空指针异常，arr:" + arr);
        // 2.定义两个下标，min 的初始值为 0，max 的初始值为 arr.length - 1
        int min = 0, max = arr.length - 1;
        // 3.定义一个循环，用于调整数组中奇偶数的位置
        while(min < max) { // 如果 min 小于 max，则一直调整数组中元素的位置
            // 4.让 min 从前往后找，如果 arr[min]的值为偶数，则停止查找
            while (min < max && arr[min] % 2 != 0) {
                min++;
            }
            // 5.让 max 从后往前找，如果 arr[max]的值为奇数，则停止查找
            while(min < max && arr[max] % 2 == 0) {
                max--;
            }
            // 6.如果 min 的值不等于 max，则交换 arr[min]和 arr[max]的值
            if(min != max) {
                int temp = arr[min];
                arr[min] = arr[max];
                arr[max] = temp;
            }
        }
    }
}
```

3. 链表

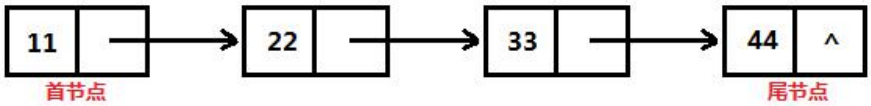
3.1 单链表

3.1.1 单链表概述

- **单链表的定义**

单链表采用的是链式存储结构，使用一组地址任意的存储单元来存放数据元素。在单链表中，存储的每一条数据都是以节点来表示的，每个节点的构成为：元素（存储数据的存储单元） + 指针（存储下一个节点的地址值），单链表的节点结构如下图所示：

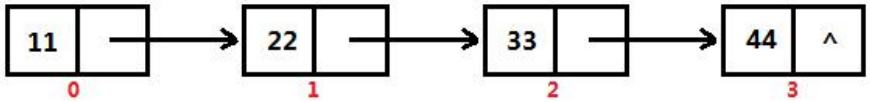
另外，单链表中的开始节点，我们又称之为首节点；单链表中的终端节点，我们又称之为尾节点。如下图所示：



- **根据序号获取节点的操作**

在线性表中，每个节点都有一个唯一的序号，该序号是从 0 开始递增的。通过序号获取单链表的节点时，我们需要从单链表的首节点开始，从前往后循环遍历，直到遇到查询序号所对应的节点时为止。

以下图为例，我们需要获得序号为 2 的节点，那么就需要依次遍历获得“节点 11”和“节点 22”，然后才能获得序号为 2 的节点，也就是“节点 33”。

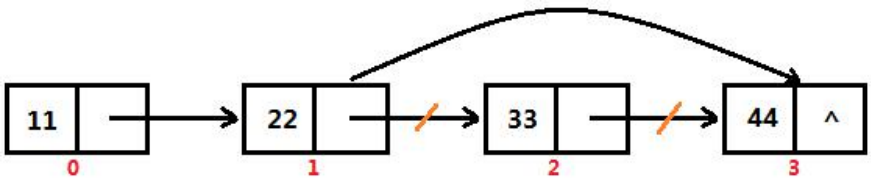


因此，在链表中通过序号获得节点的操作效率是非常低的，查询的时间复杂度为 O(n)。

- **根据序号删除节点的操作**

根据序号删除节点的操作，我们首先应该根据序号获得需要删除的节点，然后让“删除节点的前一个节点”指向“删除节点的后一个节点”，这样就实现了节点的删除操作。

以下图为例，我们需要删除序号为 2 的节点，那么就让“节点 22”指向“节点 44”即可，这样就删除了序号为 2 的节点，也就是删除了“节点 33”。

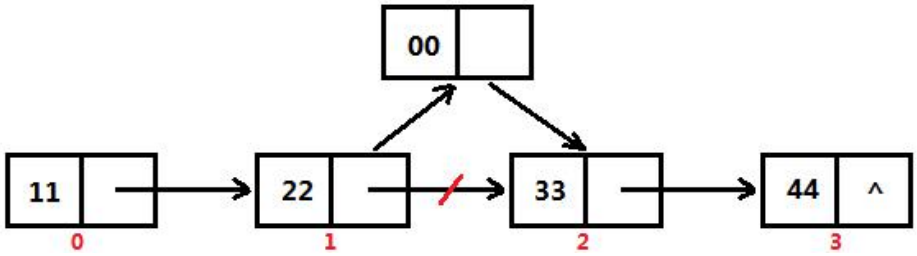


通过序号来插入节点，时间主要浪费在找正确的删除位置上，故时间复杂度为 O(n)。但是，单论删除的操作，也就是无需考虑定位到删除节点的位置，那么删除操作的时间复杂度就是 O(1)。

- **根据序号插入节点的操作**

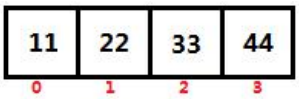
根据序号插入节点的操作，我们首先应该根据序号找到插入的节点位置，然后让“插入位置的上一个节点”指向“新插入的节点”，然后再让“新插入的节点”指向“插入位置的节点”，这样就实现了节点的插入操作。

以下图为例，我们需要在序号为 2 的位置插入元素值“00”，首先先把字符串“00”封装为一个节点对象，然后就让“节点 22”指向“新节点 00”，最后再让“节点 00”指向“节点 33”，这样就插入了一个新节点。



通过序号来插入节点，时间主要浪费在找正确的插入位置上，故时间复杂度为 O(n)。但是，单论插入的操作，也就是无需考虑定位到插入节点的位置，那么插入操作的时间复杂度就是 O(1)。

- **顺序表和单链表的比较**



顺序表

单链表

(1) 存储方式比较

顺序表采用一组地址连续的存储单元依次存放数据元素，通过元素之间的先后顺序来确定元素之间的位置，因此存储空间的利用率较高。

单链表采用一组地址任意的存储单元来存放数据元素，通过存储下一个节点的地址值来确定节点之间的位置，因此存储空间的利用率较低。

(2) 时间性能比较

顺序表查找的时间复杂度为 O(1)，插入和删除需要移动元素，因此时间复杂度为 O(n)。若是需要频繁的执行查找操作，但是很少进行插入和删除操作，那么建议使用顺序表。

单链表查找的时间复杂度为 O(n)，插入和删除无需移动元素，因此时间复杂度为 O(1)。若是需要频繁的执行插入和删除操作，但是很少进行查找操作，那么建议使用链表。

补充：根据序号来插入和删除节点，需要通过序号来找到插入和删除节点的位置，那么整体的时间复杂度为 O(n)。因此，单链表适合数据量较小时的插入和删除操作，如果存储的数据量较大，那么就建议使用别的数据结构，例如使用二叉树来实现。

(3) 空间性能比较

顺序表需要预先分配一定长度的存储空间，如果事先不知道需要存储元素的个数，分配空间过大就会造成存储空间的浪费，分配空间过小则需要执行耗时的扩容操作。

单链表不需要固定长度的存储空间，可根据需求来进行临时分配，只要有内存足够就可以分配，在链表中存储元素的个数是没有限制的，无需考虑扩容操作。

3.1.2 模拟 SingleLinkedList 实现

```
package com.bjsxt.p3.singlelinkedlist;

/**
 * 模拟单链表的实现
 */
public class SingleLinkedList {
    /**
     * 用于保存单链表中的首节点
     */
    private Node headNode;
    /**
     * 用于保存单链表中的尾节点
     */
    private Node lastNode;
    /**
     * 用于保存单链表中节点的个数
     */
    private int size;

    /**
     * 添加元素
     * @param element 需要添加的数据
     */
    public void add(Object element) {
        // 1.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 2.处理单链表为空表的情况
        if(headNode == null) {
            // 2.1 把 node 节点设置为单链表的首节点
            headNode = node;
            // 2.2 把 node 节点设置为单链表的尾节点
            lastNode = node;
        }
        // 3.处理单链表不是空表的情况
        else {
            // 3.1 让 lastNode 指向 node 节点
            lastNode.next = node;
            // 3.2 更新 lastNode 的值
            lastNode = node;
        }
        // 4.更新 size 的值
        size++;
    }

    /**
     * 根据序号获取元素
     * @param index 序号
```

```
     * @return 序号所对应节点的数据值
     */
    public Object get(int index) {
        // 1.判断序号是否合法，合法取值范围：[0, size - 1]
        if(index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.根据序号获得对应的节点对象
        Node node = node(index);
        // 3.获取并返回 node 节点的数据值
        return node.data;
    }

    /**
     * 根据序号删除元素
     * @param index 序号
     */
    public void remove(int index) {
        // 1.判断序号是否合法，合法取值范围：[0, size - 1]
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.处理删除节点在开头的情况
        if (index == 0) {
            // 2.1 获得删除节点的后一个节点
            Node nextNode = headNode.next;
            // 2.2 设置 headNode 的 next 值为 null
            headNode.next = null;
            // 2.3 设置 nextNode 为单链表的首节点
            headNode = nextNode;
        }
        // 3.处理删除节点在末尾的情况
        else if (index == size - 1) {
            // 3.1 获得删除节点的前一个节点
            Node preNode = node(index - 1);
            // 3.2 设置 preNode 的 next 值为 null
            preNode.next = null;
            // 3.3 设置 preNode 为单链表的尾节点
            lastNode = preNode;
        }
        // 4.处理删除节点在中间的情况
        else {
            // 4.1 获得 index-1 所对应的节点对象
            Node preNode = node(index - 1);
            // 4.2 获得 index+1 所对应的节点对象
            Node nextNode = preNode.next.next;
            // 4.3 获得删除节点并设置 next 值为 null
            preNode.next.next = null;
```

```
            // 4.4 设置 preNode 的 next 值为 nextNode
            preNode.next = nextNode;
        }
        // 5.更新 size 的值
        size--;
    }

    /**
     * 根据序号插入元素
     * @param index 序号
     * @param element 需要插入的数据
     */
    public void add(int index, Object element) {
        // 1.判断序号是否合法，合法取值范围：[0, size]
        if(index < 0 || index > size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 3.处理插入节点在开头位置的情况
        if(index == 0) {
            // 3.1 设置 node 的 next 值为 headNode
            node.next = headNode;
            // 3.2 设置 node 节点为单链表的首节点
            headNode = node;
        }
        // 4.处理插入节点在末尾位置的情况
        else if(index == size) {
            // 4.1 设置 lastNode 的 next 值为 node
            lastNode.next = node;
            // 4.2 设置 node 节点为单链表的尾节点
            lastNode = node;
        }
        // 5.处理插入节点在中间位置的情况
        else {
            // 5.1 获得 index-1 所对应的节点对象
            Node preNode = node(index - 1);
            // 5.2 获得 index 所对应的节点对象
            Node curNode = preNode.next;
            // 5.3 设置 preNode 的 next 为 node
            preNode.next = node;
            // 5.4 设置 node 的 next 为 curNode
            node.next = curNode;
        }
        // 6.更新 size 的值
        size++;
    }
```

```
    /**
     * 根据序号获得对应的节点对象
     * @param index 序号
     * @return 序号对应的节点对象
     */
    private Node node(int index) {
        // 1.定义一个零时节点，用于辅助单链表的遍历操作
        Node tempNode = headNode;
        // 2.定义一个循环，用于获取 index 对应的节点对象
        for(int i = 0; i < index; i++) {
            // 3.更新 tempNode 的值
            tempNode = tempNode.next;
        }
        // 4.返回 index 对应的节点对象
        return tempNode;
    }

    /**
     * 获取单链表中节点的个数
     * @return
     */
    public int size() {
        return this.size;
    }
    /**
     * 节点类
     */
    private static class Node {
        /**
         * 用于保存节点中的数据
         */
        private Object data;
        /**
         * 用于保存指向下一个节点的地址值
         */
        private Node next;
        /**
         * 构造方法
         * @param data
         */
        public Node(Object data) {
            this.data = data;
        }
    }
}
```

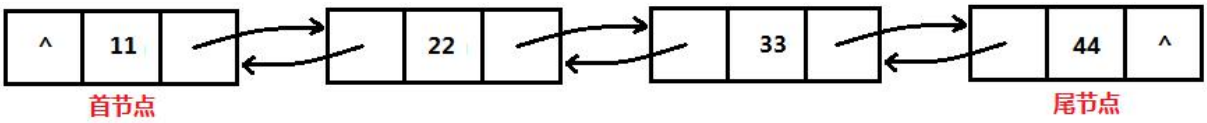
3.2 双链表

3.2.1 双链表概述

➢ **双链表的定义**

双链表也叫双向链表，它依旧采用的是链式存储结构。在双链表中，每个节点中都有两个指针，分别指向直接前驱节点（保存前一个节点的地址值）和直接后继节点（保存后一个节点的地址值），如下图所示。

所以，从双链表中的任意一个节点开始，都可以很方便地访问它的直接前驱节点和直接后继节点，如下图所示。

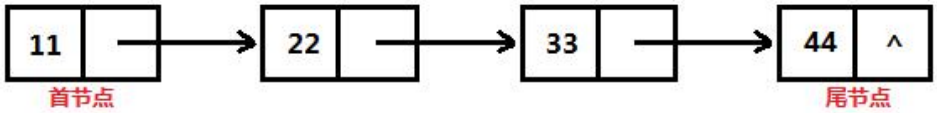


➢ **单链表和双链表的区别**

逻辑上没有区别，他们均是完成线性表的内容，主要的区别是结构上的构造有所区别。

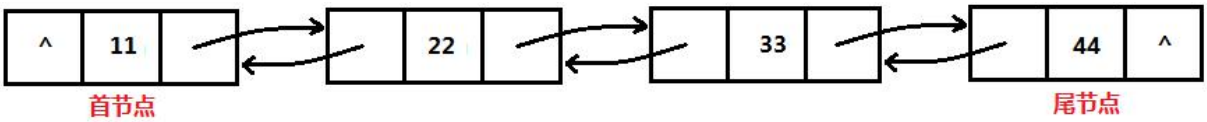
(1) 单链表

对于一个节点，有储存数据的 data 和指向下一个节点的 next。也就是说，单链表的遍历操作都得通过前节点—>后节点。



(2) 双链表

对于一个节点，有储存数据的 data 和指向下一个节点的 next，还有一个指向前一个节点的 pre。也就是说，双链表不但可以通过前节点—>后节点，还可以通过后节点—>前节点。



3.2.2 模拟 DoubleLinkedList 实现

```
package com.bjsxt.p4.doublelinkedlist;

/**
 * 模拟双链表的实现
 */
public class DoubleLinkedList {
    /**
     * 用于保存双链表中的首节点
     */
    private Node headNode;
    /**
     * 用于保存双链表中的尾节点
```

```
     */
    private Node lastNode;
    /**
     * 用于保存双链表中节点的个数
     */
    private int size;

    /**
     * 获取双链表中节点的个数
     * @return
     */
    public int size() {
        return this.size;
    }

    /**
     * 添加元素
     * @param element 需要添加的数据
     */
    public void add(Object element) {
        // 1.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 2.处理双链表为空表的情况
        if(headNode == null) {
            // 2.1 把 node 节点设置为双链表的首节点
            headNode = node;
            // 2.2 把 node 节点设置为双链表的尾节点
            lastNode = node;
        }
        // 3.处理双链表不是空表的情况
        else {
            // 3.1 设置 lastNode 的 next 值为 node
            lastNode.next = node;
            // 3.2 设置 node 的 pre 值为 lastNode
            node.pre = lastNode;
            // 3.3 设置 node 节点为双链表的尾节点
            lastNode = node;
        }
        // 4.更新 size 的值
        size++;
    }

    /**
     * 根据序号获得元素
     * @param index 序号
     * @return 序号所对应的节点对象
     */
    public Object get(int index) {
        // 1.判断序号是否合法，合法取值范围：[0, size - 1]
        if(index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException("序号不合法，index：" + index);
        }
        // 2.根据序号获得对应的节点对象
        Node node = node(index);
        // 3.获得并返回 node 节点数据值
        return node.data;
    }

    /**
     * 根据序号删除元素
     * @param index 序号
     */
    public void remove(int index) {
        // 1.判断序号是否合法，合法取值范围：[0, size - 1]
        if(index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.处理删除节点为首节点的情况
        if(index == 0) {
            // 2.1 获得删除节点的下一个节点
            Node nextNode = headNode.next;
            // 2.2 设置 headNode 的 next 值为 null
            headNode.next = null;
            // 2.3 设置 nextNode 的 pre 值为 null
            if(nextNode != null) {
                nextNode.pre = null;
            }
            // 2.4 设置 nextNode 为双链表的首节点
            headNode = nextNode;
        }
        // 3.处理删除节点为尾节点的情况
        else if(index == size - 1) {
            // 3.1 获得删除节点的前一个节点
            Node preNode = lastNode.pre;
            // 3.2 设置 preNode 的 next 值为 null
            preNode.next = null;
            // 3.3 设置 lastNode 的 pre 值为 null
            lastNode.pre = null;
            // 3.4 设置 preNode 为双链表的尾节点
            lastNode = preNode;
        }
        // 4.处理删除节点为中间节点的情况
        else {
            // 4.1 获得删除节点对象
            Node delNode = node(index);
            // 4.2 获得删除节点的前一个节点
            Node preNode = delNode.pre;
            // 4.3 获得删除节点的后一个节点
```

```
            Node nextNode = delNode.next;
            // 4.4 设置 preNode 的 next 值为 nextNode
            preNode.next = nextNode;
            // 4.5 设置 nextNode 的 pre 值为 preNode
            nextNode.pre = preNode;
            // 4.6 设置 delNode 的 next 值和 pre 值为 null
            delNode.next = null;
            delNode.pre = null;
        }
        // 5.更新 size 的值
        size--;
    }

    /**
     * 根据序号插入元素
     * @param index 序号
     * @param element 需要插入的数据
     */
    public void add(int index, Object element) {
        // 1.判断序号是否合法，合法取值范围：[0, size]
        if(index < 0 || index > size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 3.处理插入节点在开头的情况
        if(index == 0) {
            // 3.1 设置 node 的 next 值为 headNode
            node.next = headNode;
            // 3.2 设置 headNode 的 pre 值为 node
            headNode.pre = node;
            // 3.3 设置 node 节点为双链表的首节点
            headNode = node;
        }
        // 4.处理插入节点在末尾的情况
        else if(index == size) {
            add(element);
            return;
        }
        // 5.处理插入节点在中间的情况
        else {
            // 5.1 获得插入位置所对应的节点对象
            Node curNode = node(index);
            // 5.2 获得插入位置前面的哪一个节点对象
            Node preNode = curNode.pre;
            // 5.3 设置 node 节点和 preNode 节点之间的连线
            preNode.next = node;
            node.pre = preNode;
```

```
            // 5.4 设置 node 节点和 curNode 节点之间的连线
            node.next = curNode;
            curNode.pre = node;
        }
        // 6.更新 size 的值
        size++;
    }

    /**
     * 根据序号获得对应的节点对象
     * @param index 序号
     * @return 序号所对应的节点对象
     */
    private Node node(int index) {
        // 1.如果查找的节点在双链表的前半区，则从前往后查找
        if(index < size / 2) {
            // 1.1 定义一个零时节点，用于辅助双链表的遍历
            Node nextNode = headNode;
            // 1.2 定义一个循环，从前往后查找 index 所对应的节点对象
            for(int i = 0; i < index; i++) {
                // 1.3 更新 nextNode 的值
                nextNode = nextNode.next;
            }
            // 1.4 返回 index 所对应的节点对象
            return nextNode;
        }
        // 2.如果查找的节点在双链表的后半区，则从后往前查找
        else {
            // 2.1 定义一个零时节点，用于辅助双链表的遍历
            Node preNode = lastNode;
            // 2.2 定义一个循环，从后往前查找 index 所对应的节点对象
            for(int i = size - 1; i > index; i--) {
                // 2.3 更新 preNode 的值
                preNode = preNode.pre;
            }
            // 2.4 返回 index 所对应的节点对象
            return preNode;
        }
    }

    /**
     * 节点类
     */
    private static class Node {
        /**
         * 用于保存节点中的数据
         */
        private Object data;
        /**
```

```
        * 用于保存上一个节点的地址值
        */
        private Node pre;
        /**
        * 用于保存下一个节点的地址值
        */
        private Node next;
        /**
        * 构造方法
        * @param data
        */
        public Node(Object data) {
            this.data = data;
        }
    }
}
```

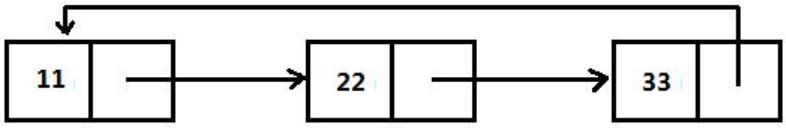
3.3 环形链表

3.3.1 环形链表概述

环形链表依旧采用的是链式存储结构，它的特点就是设置首节点和尾节点相互指向，从而实现让整个链表形成一个环。在我们实际开发中，常见的环形链表有：

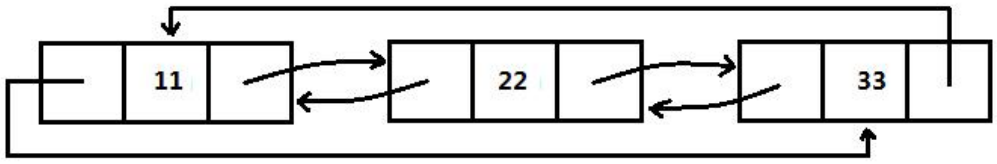
(1) 环形单链表

在单链表中，尾节点的指针指向了首节点，从而整个链表形成一个环，如下图所示：



(2) 环形双链表

在双链表中，尾节点的指针指向了首节点，首节点的指针指向了尾节点，从而整个链表形成一个环，如下图所示：



3.3.2 模拟 CycleSingleLinkedList 实现

```
package com.bjsxt.p5.cyclesinglelinkedlist;

/**
 * 模拟环形单链表的实现
```

```
 */
public class CycleSingleLinkedList {
    /**
     * 用于保存单链表中的首节点
     */
    private Node headNode;
    /**
     * 用于保存单链表中的尾节点
     */
    private Node lastNode;
    /**
     * 用于保存单链表中节点的个数
     */
    private int size;

    /**
     * 获取单链表中节点的个数
     * @return
     */
    public int size() {
        return this.size;
    }

    /**
     * 添加元素
     * @param element 需要添加的数据
     */
    public void add(Object element) {
        // 1.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 2.处理单链表为空表的情况
        if(headNode == null) {
            // 2.1 把 node 节点设置为单链表的首节点
            headNode = node;
            // 2.2 把 node 节点设置为单链表的尾节点
            lastNode = node;
        }
        // 3.处理单链表不是空表的情况
        else {
            // 3.1 让 lastNode 指向 node 节点
            lastNode.next = node;
            // 3.2 更新 lastNode 的值
            lastNode = node;
        }
        // 4.设置 lastNode 的 next 值为 headNode
        lastNode.next = headNode;
        // 5.更新 size 的值
        size++;
```

```
    }

    /**
     * 根据序号获取元素
     * @param index 序号
     * @return 序号所对应节点的数据值
     */
    public Object get(int index) {
        // 1.如果序号的取值小于 0，则证明是不合法的情况
        if(index < 0) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.根据序号获得对应的节点对象
        Node node = node(index);
        // 3.获取并返回 node 节点的数据值
        return node.data;
    }

    /**
     * 根据序号删除元素
     * @param index 序号
     */
    public void remove(int index) {
        // 1.判断序号是否合法，合法取值范围：[0, size - 1]
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.处理删除节点在开头的情况
        if (index == 0) {
            // 2.1 获得删除节点的后一个节点
            Node nextNode = headNode.next;
            // 2.2 设置 headNode 的 next 值为 null
            headNode.next = null;
            // 2.3 设置 nextNode 为单链表的首节点
            headNode = nextNode;
            // 2.4 设置 lastNode 的 next 值为 headNode
            lastNode.next = headNode;
        }
        // 3.处理删除节点在末尾的情况
        else if (index == size - 1) {
            // 3.1 获得删除节点的前一个节点
            Node preNode = node(index - 1);
            // 3.2 设置 preNode 的 next 值为 null
            preNode.next = null;
            // 3.3 设置 preNode 为单链表的尾节点
            lastNode = preNode;
```

```
            // 3.4 设置 lastNode 的 next 值为 headNode
            lastNode.next = headNode;
        }
        // 4.处理删除节点在中间的情况
        else {
            // 4.1 获得 index-1 所对应的节点对象
            Node preNode = node(index - 1);
            // 4.2 获得 index+1 所对应的节点对象
            Node nextNode = preNode.next.next;
            // 4.3 获得删除节点并设置 next 值为 null
            preNode.next.next = null;
            // 4.4 设置 preNode 的 next 值为 nextNode
            preNode.next = nextNode;
        }
        // 5.更新 size 的值
        size--;
        // 6.判断 size 的值是否为 0，如果 size 的值为 0，则设置 headNode 和 lastNode 为
null
        if(size == 0) {
            headNode = null;
            lastNode = null;
        }
    }

    /**
     * 根据序号插入元素
     * @param index 序号
     * @param element 需要插入的数据
     */
    public void add(int index, Object element) {
        // 1.判断序号是否合法，合法取值范围：[0, size]
        if(index < 0 || index > size) {
            throw new IndexOutOfBoundsException("序号不合法，index:" + index);
        }
        // 2.把需要添加的数据封装成节点对象
        Node node = new Node(element);
        // 3.处理插入节点在开头位置的情况
        if(index == 0) {
            // 3.1 设置 node 的 next 值为 headNode
            node.next = headNode;
            // 3.2 设置 node 节点为单链表的首节点
            headNode = node;
            // 3.3 设置 lastNode 的 next 值为 headNode
            lastNode.next = headNode;
        }
        // 4.处理插入节点在末尾位置的情况
```

```
        else if(index == size) {
            // 4.1 设置 lastNode 的 next 值为 node
            lastNode.next = node;
            // 4.2 设置 node 节点为单链表的尾节点
            lastNode = node;
            // 4.3 设置 lastNode 的 next 值为 headNode
            lastNode.next = headNode;
        }
        // 5.处理插入节点在中间位置的情况
        else {
            // 5.1 获得 index-1 所对应的节点对象
            Node preNode = node(index - 1);
            // 5.2 获得 index 所对应的节点对象
            Node curNode = preNode.next;
            // 5.3 设置 preNode 的 next 为 node
            preNode.next = node;
            // 5.4 设置 node 的 next 为 curNode
            node.next = curNode;
        }
        // 6.更新 size 的值
        size++;
    }

    /**
     * 根据序号获得对应的节点对象
     * @param index 序号
     * @return 序号对应的节点对象
     */
    private Node node(int index) {
        // 0.判断环形单链表是否为空表
        if(headNode == null) {
            throw new NullPointerException("环形单链表为空表");
        }
        // 1.定义一个零时节点，用于辅助单链表的遍历操作
        Node tempNode = headNode;
        // 2.定义一个循环，用于获取 index 对应的节点对象
        for(int i = 0; i < index % size; i++) {
            // 3.更新 tempNode 的值
            tempNode = tempNode.next;
        }
        // 4.返回 index 对应的节点对象
        return tempNode;
    }

    /**
     * 节点类
```

```
     */
    private static class Node {
        /**
         * 用于保存节点中的数据
         */
        private Object data;
        /**
         * 用于保存指向下一个节点的地址值
         */
        private Node next;
        /**
         * 构造方法
         * @param data
         */
        public Node(Object data) {
            this.data = data;
        }
    }
}
```

3.3.3 环形单链表的约瑟夫问题

- **题目描述**

设置编号为 1，2，3，...，n 的 n 个小孩围坐一圈，约定编号为 k（1<=k<=n）的小孩从 1 开始报数，当数到 m 时对应的那个小孩出列。然后它的下一位又从 1 开始报数，当数到 m 时对应的小孩又出列，以次类推，直到所有小孩出列为止，由此产生一个出队编号的序列。

例如，上图中从编号为 2 的小孩从 1 开始报数，每次数到 3 时对应的那个小孩就出列，最后得到的出队编号就是：4、7、10、3、8、2、9、6、1、5。

- **代码实现**

```
package com.bjsxt.p5.cyclesinglelinkedlist;
/**
 * 环形单链表的约瑟夫问题
 */
public class Test02 {
    public static void main(String[] args) {
        // 创建一个环形单链表
```

```
        Node lastNode = new Node(10);
        Node node9 = new Node(9, lastNode);
        Node node8 = new Node(8, node9);
        Node node7 = new Node(7, node8);
        Node node6 = new Node(6, node7);
        Node node5 = new Node(5, node6);
        Node node4 = new Node(4, node5);
        Node node3 = new Node(3, node4);
        Node node2 = new Node(2, node3);
        Node headNode = new Node(1, node2);
        lastNode.next = headNode;
        // 执行约瑟夫方法
        josephus(headNode, lastNode, 10, 2, 3);
    }

    /**
     * 约瑟夫问题
     * @param headNode 环形单链表的首节点
     * @param lastNode 环形单链表的尾节点
     * @param size 环形单链表中节点的个数
     * @param start 从编号为 start 的小孩开始报数
     * @param count 每次数几下
     */
    public static void josephus(Node headNode, Node lastNode, int size, int start,
int count) {
        // 1.处理不合法的情况
        // 1.1 处理 headNode 为 null 的情况
        if(headNode == null) {
            throw new NullPointerException("headNode 为 null");
        }
        // 1.2 处理 start 和 count 不合法的情况
        if(start < 1 || start > size || count < 1) {
            throw new IllegalArgumentException("参数不合法");
        }
        // 2.设置编号为 start 的小孩开始报数，并且使用 headNode 指向该节点
        // 把 headNode 和 lastNode 往后移动 start-1 次
        for (int i = 0; i < start - 1; i++) {
            headNode = headNode.next;
            lastNode = lastNode.next;
        }
        // 3.定义一个循环，用于循环的执行报数操作
        while (size != 0) { // 如果 size 等于 0，则停止报数
            // 4.执行报数操作，也就是找到需要出圈的小孩，我们使用 headNode 指向需要出圈
的节点
            // 把 headNode 和 lastNode 往后移动 count-1 次
            for (int i = 0; i < count - 1; i++) {
```

```
                headNode = headNode.next;
                lastNode = lastNode.next;
            }
            // 5.输出需要出圈小孩的编号，也就是输出 headNode 保存的数据值
            System.out.println(headNode.data);
            // 6.实现小孩的出圈操作，也就是把 headNode 从环形单链表中删除
            // 6.1 获得删除节点的后一个节点
            Node nextNode = headNode.next;
            // 6.2 设置 lastNode 的 next 值为 nextNode
            lastNode.next = nextNode;
            // 6.3 设置 headNode 的 next 值为 null
            headNode.next = null;
            // 7.更新 size 的值
            size--;
            // 8.设置 headNode 指向 nextNode
            headNode = nextNode;
        }
    }

    /**
     * 节点类
     */
    private static class Node {
        /**
         * 用于保存节点中的数据值
         */
        private Object data;
        /**
         * 用于保存下一个节点的地址值
         */
        private Node next;

        /**
         * 专门为 data 做初始化的工作
         * @param data
         */
        public Node(Object data) {
            this.data = data;
        }

        /**
         * 专门为 data 和 next 做初始化的工作
         * @param data
         * @param next
         */
        public Node(Object data, Node next) {
            this.data = data;
            this.next = next;
        }
```

```
    }
}
```

3.4 经典面试题

说明：以下面试题中，所用的节点类声明如下：

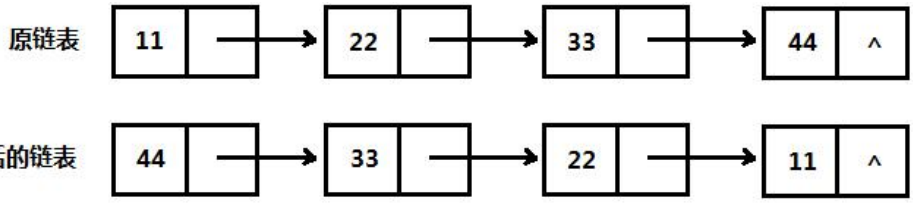
```
package com.bjsxt.p6.linkedlisttest;

/**
 * 单链表节点类
 */
public class Node {
    /**
     * 用于保存节点中的数据
     */
    private Object data;
    /**
     * 用于保存下一个节点的地址值
     */
    private Node next;

    /**
     * 专门为 data 做初始化的工作
     * @param data
     */
    public Node(Object data) {
        this.data = data;
    }

    /**
     * 专门为 data 和 next 做初始化的工作
     * @param data
     * @param next
     */
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public void setData(Object data) {
        this.data = data;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public Object getData() {
        return data;
    }
```

```
    public Node getNext() {
        return next;
    }
}
```

3.4.1 单链表的反转

- **题目描述**

  输入一个链表，反转链表后，输出新链表的表头，反转后的结果如下图所示。



- **思路分析**

  从前往后遍历原链表，把遍历出来的节点设置为反转后链表的首节点，当原链表遍历完毕，那么就等到了反转后的链表。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;
/**
 * 单链表的反转
 */
public class Test01 {
    public static void main(String[] args) {
        // 创建一个单链表
        Node lastNode = new Node(44);
        Node node3 = new Node(33, lastNode);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        System.out.println("反转之前的链表：");
        print(headNode);
        System.out.println();
        // 执行反转操作
        Node reverse = reverseLinkedList(headNode);
        System.out.println("反转之后的链表：");
        print(reverse);
    }

    /**
     * 实现单链表的反转
     * @param headNode 需要反转单链表的首节点
     * @return 反转之后链表的首节点
     */
```

```
    public static Node reverseLinkedList(Node headNode) {
        // 1.判断 headNode 为 null 和单链表只有一个节点的情况
        if(headNode == null || headNode.getNext() == null) {
            return headNode;
        }
        // 2.定义一个节点，用于保存反转后链表的首节点
        Node reverse = null;
        // 3.定义一个循环，用于实现单链表的反转操作
        while (headNode != null) {
            // 4.获得 headNode 的下一个节点
            Node nextNode = headNode.getNext();
            // 5.把 headNode 插入到反转后链表的最前面
            headNode.setNext(reverse);
            // 6.更新反转后链表的首节点
            reverse = headNode;
            // 7.更新需要反转链表的首节点
            headNode = nextNode;
        }
        // 8.返回反转后链表的首节点
        return reverse;
    }

    /**
     * 实现单链表的遍历操作
     * @param headNode 单链表中的首节点
     */
    public static void print(Node headNode) {
        // 1.定义一个零时节点，用于辅助单链表的遍历操作
        Node tempNode = headNode;
        // 2.定义一个循环，用于实现单链表的遍历操作
        while (tempNode != null) {
            // 3.输出 tempNode 中保存的数据值
            System.out.println(tempNode.getData());
            // 4.让 tempNode 指向它的下一个节点
            tempNode = tempNode.getNext();
        }
    }
}
```

3.4.2 查找单链表的中间节点

- **题目描述**

  要求不允许获得链表的长度，从而获得单链表的中间节点。如果链表的长度为偶数，返回中间

两个节点的任意一个；若为奇数，则返回中间节点。

- **思路分析**

设置两个指针 fast 和 slow 并设置默认值为首节点，然后两个指针同时往后走，fast 指针每次走两步，slow 指针每次走一步，直到 fast 指针走到最后一个节点时，此时 slow 指针所指的节点就是中间节点。注意链表为空和链表节点个数为 1 的情况。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;
/**
 * 查找单链表的中间节点
 */
public class Test02 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(55);
        Node node4 = new Node(44, lastNode);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        // 2.获得单链表的中间节点
        Node midNode = findMidNode(headNode);
        System.out.println(midNode.getData());
    }

    /**
     * 查找单链表的中间节点
     * @param headNode 单链表的首节点
     * @return 单链表的中间节点
     */
    public static Node findMidNode(Node headNode) {
        // 1.处理 headNode 为 null 或单链表只有一个节点的情况
        if (headNode == null || headNode.getNext() == null) {
            return headNode;
        }
        // 2.定义一个快指针，让它每次往后走两步
        Node fast = headNode;
        // 3.定义一个慢指针，让它每次往后走一步
        Node slow = headNode;
        // 4.定义一个循环，用于循环的执行快慢指针往后移动
        while (fast != null && fast.getNext() != null) {
            // 5.设置快慢指针往后移动
            fast = fast.getNext().getNext();
            slow = slow.getNext();
        }
        // 6.返回单链表的中间节点
```

```
        return slow;
    }
}
```

3.4.3 在 O(1)时间删除单链表节点

- **题目描述**

  提供单链表的首节点和删除节点，定义一个方法在 O(1)时间删除该节点。

- **思路分析**

  常规的做法是从链表的首节点开始遍历，找到需要删除的节点的前驱节点，把它的 next 指向要删除节点的下一个节点，平均时间复杂度为 O(n)，不满足题目要求。

  那是不是一定要得到被删除的节点的前一个节点呢？其实不用的。我们可以很方面地得到要删除节点的下一个节点，如果我们把下一个节点的内容复制到要删除的节点上覆盖原有的内容，再把下一个节点删除，那就相当于把当前要删除的节点删除了。举个栗子，我们要删除的节点 i，先把 i 的下一个节点 j 的内容复制到 i，然后把 i 的指针指向节点 j 的下一个节点。此时再删除节点 j，其效果刚好是把节点 i 给删除了。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;

/**
 * 在 O(1)时间删除单链表节点
 */
public class Test03 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(55);
        Node node4 = new Node(44, lastNode);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        // 2.测试删除节点的情况
        Node head = deleteNode(headNode, node3);
        print(head);
    }

    /**
     * 实现单链表的遍历操作
     * @param headNode 单链表中的首节点
     */
    public static void print(Node headNode) {
        // 1.定义一个零时节点，用于辅助单链表的遍历操作
        Node tempNode = headNode;
        // 2.定义一个循环，用于实现单链表的遍历操作
        while (tempNode != null) {
```

```
            // 3.输出 tempNode 中保存的数据值
            System.out.println(tempNode.getData());
            // 4.让 tempNode 指向它的下一个节点
            tempNode = tempNode.getNext();
        }
    }

    /**
     * 在 O(1)时间删除单链表节点
     * @param headNode 单链表中的首节点
     * @param delNode 需要删除的节点
     * @return 删除节点后单链表的首节点
     */
    public static Node deleteNode(Node headNode, Node delNode) {
        // 1.处理 headNode 为 null 和 delNode 为 null 的情况
        if (headNode == null || delNode == null) {
            throw new NullPointerException("headNode 或 delNode 为 null");
        }
        // 2.处理删除节点在开头的情况
        if (headNode == delNode) {
            // 2.1 获得删除节点的后一个节点
            Node nextNode = headNode.getNext();
            // 2.2 设置 headNode 的 next 值为 null
            headNode.setNext(null);
            // 2.3 返回删除节点后单链表的首节点
            return nextNode;
        }
        // 3.处理删除节点在末尾的情况
        else if(delNode.getNext() == null) {
            // 3.1 获得删除节点的前一个节点
            // a ) 定义一个零时节点，用于辅助单链表的遍历
            Node tempNode = headNode;
            // b)定义一个循环，用于找到尾节点的前一个节点
            while (tempNode.getNext() != delNode) {
                // c ) 让 tempNode 指向它的下一个节点
                tempNode = tempNode.getNext();
            }
            // 3.2 设置 tempNode 的 next 值为 null
            tempNode.setNext(null);
            // 3.3 返回删除节点后链表的首节点
            return headNode;
        }
        // 4.处理删除节点在中间的情况
        else {
            // 4.1 获得删除节点的后一个节点
```

```
            Node nextNode = delNode.getNext();
            // 4.2 把 nextNode 中保存的数据值赋值给删除节点
            delNode.setData(nextNode.getData());
            // 4.3 从单链表中删除 nextNode 节点
            // a ) 设置 delNode 的 next 值为 nextNode 的下一个节点
            delNode.setNext(nextNode.getNext());
            // b)设置 nextNode 的 next 值为 null
            nextNode.setNext(null);
            // 4.4 返回删除节点后单链表的首节点
            return headNode;
        }
    }
}
```

3.4.4 查找单链表倒数第 k 个节点

- **题目描述**

  查找单链表倒数第 k 个节点，要求：不允许遍历获得单链表的长度。

- **思路分析**

  这里需要声明两个指针，也就是 first 指针和 second 指针，并且设置他们的初始值都是单链表的首节点，然后让 first 指针往后移动 k-1 个位置，此时 first 和 second 就间隔了 k-1 个位置，然后整体向后移动这两个指针，直到 first 指针走到单链表的尾节点时，此时 second 指针所指向的位置就是倒数第 k 个节点的位置。

- **代码实现**

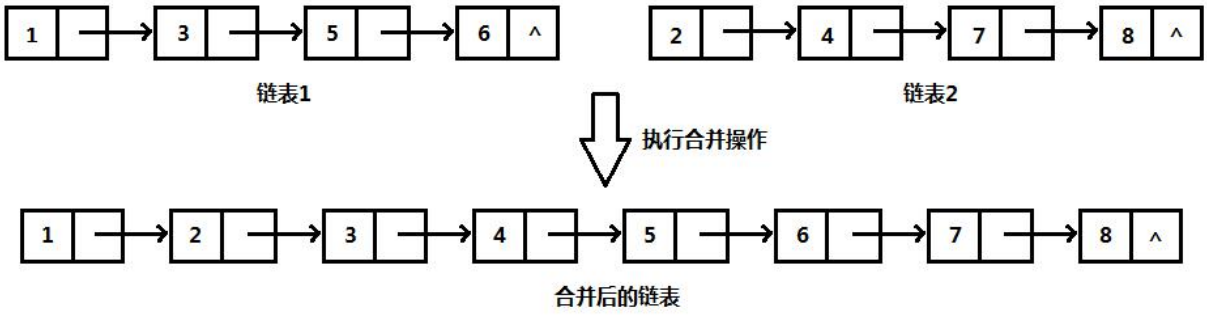
```
package com.bjsxt.p6.linkedlisttest;

/**
 * 查找单链表倒数第 k 个节点
 */
public class Test04 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(55);
        Node node4 = new Node(44, lastNode);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        // 2.查找单链表倒数第 k 个节点
        Node node = getLastIndexOfNode(headNode, 5);
        System.out.println(node.getData());
    }

    /**
     * 查找单链表倒数第 k 个节点
     * @param headNode 单链表的首节点
```

```
     * @param k 表示单链表倒数第 k 个节点
     * @return 返回单链表中倒数第 k 个节点
     */
    public static Node getLastIndexOfNode(Node headNode, int k) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            throw new NullPointerException("headNode 为 null");
        }
        // 此处 k 的合法取值范围：[1, 单链表长度]
        // 2.定义 first 和 second 指针，并且设置它们的初始值为单链表的首节点
        Node first = headNode, second = headNode;
        // 3.通过循环，让 first 指针往后移动 k-1 次
        for (int i = 0; i < k - 1; i++) {
            first = first.getNext();
        }
        // 4.通过循环，每次让 first 和 second 往后移动一次。
        // 当 first 指向的节点为单链表的尾节点时，则停止移动
        while (first.getNext() != null) {
            // 5.设置 first 和 second 每次往后移动一次
            first = first.getNext();
            second = second.getNext();
        }
        // 6.返回单链表中倒数第 k 个节点
        return second;
    }
}
```

3.4.5 合并两个有序的单链表

- **题意分析**

要求：合并两个有序的单链表，合并之后的单链表依然有序。



- **思路分析**

挨着比较链表 1 和链表 2。注意：两个链表都为空和其中一个为空的情况。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;
```

```java
/**
 * 合并两个有序的单链表
 * 链表 1：1 --> 3 --> 5 --> 6
 * 链表 2: 2 --> 4 --> 7 --> 8
 * 合并之后的链表：1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8
 */
public class Test05 {
    public static void main(String[] args) {
        // 1.创建两个单链表
        // 链表 1
        Node node1_4 = new Node(6);
        Node node1_3 = new Node(5, node1_4);
        Node node1_2 = new Node(3, node1_3);
        Node node1_1 = new Node(1, node1_2);

        // 链表 2
        Node node2_4 = new Node(8);
        Node node2_3 = new Node(7, node2_4);
        Node node2_2 = new Node(4, node2_3);
        Node node2_1 = new Node(2, node2_2);

        // 2.合并两个有序的单链表
        Node headNode = mergeList(node1_1, node2_1);
        print(headNode);
    }

    /**
     * 实现单链表的遍历操作
     * @param headNode 单链表中的首节点
     */
    public static void print(Node headNode) {
        // 1.定义一个零时节点，用于辅助单链表的遍历操作
        Node tempNode = headNode;
        // 2.定义一个循环，用于实现单链表的遍历操作
        while (tempNode != null) {
            // 3.输出 tempNode 中保存的数据值
            System.out.println(tempNode.data);
            // 4.让 tempNode 指向它的下一个节点
            tempNode = tempNode.next;
        }
    }

    /**
     * 合并两个有序的单链表
     * @param head1 链表 1 的首节点
     * @param head2 链表 2 的首节点
     * @return 返回合并后链表的首节点
```

```
     */
    public static Node mergeList(Node head1, Node head2) {
        // 1.处理 head1 和 head2 都为 null 的情况
        if (head1 == null && head2 == null) {
            return null;
        }
        // 2.处理 head1 和 head2 其中一个为 null 的情况
        if (head1 == null) {
            return head2;
        }
        if (head2 == null) {
            return head1;
        }
        // 3.定义 headNode 和 lastNode，分别作为合并后链表的首节点和尾节点
        Node headNode = null, lastNode = null;
        // 4.获取 head1 和 head2 中数据值较小的节点，并设置为合并后链表的首节点和尾节点
        if (head1.data > head2.data) {
            headNode = head2;
            lastNode = head2;
            // 更新 head2 的值，让 head2 指向它的下一个节点
            head2 = head2.next;
        }
        else {
            headNode = head1;
            lastNode = head1;
            // 更新 head1 的值，让 head1 指向它的下一个节点
            head1 = head1.next;
        }
        // 5.定义一个循环，用于依次的获取链表 1 和链表 2 中数据值较小的节点，并把该节点
添加到合并后链表的末尾
        while (head1 != null && head2 != null) {
            // 处理 head1 的数据值大于 head2 的情况
            if (head1.data > head2.data) {
                lastNode.next = head2;
                lastNode = head2;
                // 更新 head2 的值，让 head2 指向它的下一个节点
                head2 = head2.next;
            }
            // 处理 head2 的数据值大于 head1 的情况
            else {
                lastNode.next = head1;
                lastNode = head1;
                // 更新 head1 的值，让 head1 指向它的下一个节点
                head1 = head1.next;
            }
        }
        // 6.循环执行完毕，如果某个链表的首节点不为 null，那么我们就将这个链表的首节点
及其之后的节点添加到合并后链表的末尾
```

```
        if (head1 == null) { // 意味着 head2 可能不为 null
            lastNode.next = head2;
        }
        else { // 意味着 head1 可能不为 null
            lastNode.next = head1;
        }
        // 7.返回合并后链表的首节点
        return headNode;
    }

    /**
     * 节点类
     */
    private static class Node {

        /**
         * 用于保存节点中的数据
         */
        private int data;
        /**
         * 用于保存下一个节点的地址值
         */
        private Node next;

        /**
         * 专门为 data 做初始化的工作
         * @param data
         */
        public Node(int data) {
            this.data = data;
        }

        /**
         * 专门为 data 和 next 做初始化的工作
         * @param data
         * @param next
         */
        public Node(int data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
}
```

3.4.6 从尾到头打印单链表

- **方案一：使用栈来实现**

```
package com.bjsxt.p6.linkedlisttest;
```

```
import java.util.Stack;

/**
 * 从尾到头打印单链表（使用栈来实现）
 */
public class Test06 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(44);
        Node node3 = new Node(33, lastNode);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        // 2.从尾到头打印单链表
        reversePrint(headNode);
    }

    /**
     * 从尾到头打印单链表
     * @param headNode 单链表的首节点
     */
    public static void reversePrint(Node headNode) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            throw new NullPointerException("headNode 为 null");
        }
        // 2.创建一个栈
        Stack<Node> stack = new Stack<>();
        // 3.遍历单链表中的所有节点
        Node tempNode = headNode;
        while (tempNode != null) {
            // 4.把遍历出来的节点添加进入栈中
            stack.push(tempNode);
            tempNode = tempNode.getNext();
        }
        // 5.遍历栈中的所有节点
        while (!stack.isEmpty()) {
            Node node = stack.pop();
            System.out.println(node.getData());
        }
    }
}
```

- **方案二：使用递归来实现**

```
package com.bjsxt.p6.linkedlisttest;
/**
 * 从尾到头打印单链表（使用递归来实现）
 */
public class Test07 {
```

```
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(44);
        Node node3 = new Node(33, lastNode);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        // 2.从尾到头打印单链表（使用递归来实现）
        reversePrint(headNode);
    }

    /**
     * 从尾到头打印单链表（使用递归来实现）
     * @param headNode 单链表的首节点
     */
    public static void reversePrint(Node headNode) {
        // 1.判断 headNode 为 null 的情况
        if (headNode == null) {
            return;
        }
        // 2.从尾到头打印以 headNode 下一个节点为首节点的链表
        reversePrint(headNode.getNext());
        // 3.打印输出 headNode 中的数据值
        System.out.println(headNode.getData());
    }
}
```

3.4.7 判断单链表是否有环

- **思路分析**

这里也是用到两个指针，如果一个链表有环，那么用一个指针去遍历，是永远走不到头的。因此，我们用两个指针去遍历：slow 指针每次走一步，fast 指针每次走两步，如果 slow 指针和 fast 指针相遇，则证明单链表中有环。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;

/**
 * 判断单链表是否有环
 */
public class Test08 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(55);
        Node node4 = new Node(44, lastNode);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        lastNode.setNext(node2);
```

```
        // 2.判断单链表是否有环
        boolean flag = hasCycle(headNode);
        System.out.println(flag);
    }

    /**
     * 判断单链表是否有环
     * @param headNode 单链表的首节点
     * @return 如果单链表有环，则返回 true，否则返回 fasle
     */
    public static boolean hasCycle(Node headNode) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            return false;
        }
        // 2.定义一个快指针，每次往后走两步
        Node fast = headNode;
        // 3.定义一个慢指针，每次往后走一步
        Node slow = headNode;
        // 4.定义一个循环，用于判断单链表是否有环
        while (fast != null && fast.getNext() != null) {
            // 5.设置快慢指针每次往后移动
            fast = fast.getNext().getNext();
            slow = slow.getNext();
            // 6.如果 fast 和 slow 指向的是同一个节点，则证明单链表有环
            if (fast == slow) {
                return true;
            }
        }
        // 7.执行到此处，证明单链表是无环单链表
        return false;
    }
}
```

3.4.8 从有环链表中，获得环的长度

- **思路分析**

通过第五题，获得相遇的那个节点，我们拿到这个相遇的节点就好办了，这个节点肯定是在环里嘛，我们可以让这个节点对应的指针一直往下走，直到它回到原点，就可以算出环的长度了。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;

/**
 * 从有环链表中，获得环的长度
 */
public class Test09 {
```

```
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(66);
        Node node5 = new Node(55, lastNode);
        Node node4 = new Node(44, node5);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        lastNode.setNext(node2);
        // 2.从有环链表中，获得环的长度
        int size = getCycleLength(headNode);
        System.out.println(size);
    }

    /**
     * 从有环链表中，获得环的长度
     * @param headNode 单链表的首节点
     * @return 如果单链表有环，则返回环中节点的个数，否则返回 0
     */
    public static int getCycleLength(Node headNode) {
        // 1.获得快慢指针相交的节点
        Node meetNode = meetNode(headNode);
        // 2.处理 meetNode 为 null 的情况
        if (meetNode == null) {
            return 0;
        }
        // 3.定义一个变量，用于保存环中节点的个数
        int size = 0;
        // 4.从 meetNode 节点开始，遍历环中的节点
        // 4.1 定义一个零时节点，用于辅助单链表的遍历
        Node tempNode = meetNode;
        // 4.2 定义一个死循环，用于遍历环中的所有节点
        while (true) {
            // 4.3 让 tempNode 指向它的下一个节点
            tempNode = tempNode.getNext();
            // 4.4 更新 size 的值
            size++;
            // 5.如果 tempNode 和 meetNode 指向的是同一个节点，那么就需要停止遍历操作
            if (tempNode == meetNode) {
                break;
            }
        }
        // 6.返回环中节点的个数
        return size;
    }

    /**
```

```
     * 获得快慢指针相交的节点
     * @param headNode 单链表的首节点
     * @return 如果单链表有环，则返回快慢指针相交的节点，否则返回 null。
     */
    public static Node meetNode(Node headNode) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            return null;
        }
        // 2.定义一个快指针，每次往后走两步
        Node fast = headNode;
        // 3.定义一个慢指针，每次往后走一步
        Node slow = headNode;
        // 4.定义一个循环，用于判断单链表是否有环
        while (fast != null && fast.getNext() != null) {
            // 5.设置快慢指针每次往后移动
            fast = fast.getNext().getNext();
            slow = slow.getNext();
            // 6.如果 fast 和 slow 指向的是同一个节点，则证明单链表有环
            if (fast == slow) {
                return fast;
            }
        }
        // 7.执行到此处，证明单链表是无环单链表
        return null;
    }
}
```

3.4.9 单链表中，取出环的起始点

- **思路分析**

先获得单链表中环的长度（length），定义两个指针 first 和 second，初始值都设置为单链表的首节点。先让 first 指针走 length 步，然后再让 first 指针和 second 指针每次各往后走一步，当两个指针相遇时，相遇时的节点就是环的起始点。

- **代码实现**

```
package com.bjsxt.p6.linkedlisttest;

/**
 * 单链表中，取出环的起始点
 */
public class Test10 {
    public static void main(String[] args) {
        // 1.创建一个单链表
        Node lastNode = new Node(66);
        Node node5 = new Node(55, lastNode);
```

```
        Node node4 = new Node(44, node5);
        Node node3 = new Node(33, node4);
        Node node2 = new Node(22, node3);
        Node headNode = new Node(11, node2);
        lastNode.setNext(node2);
        // 2.单链表中，取出环的起始点
        Node startNode = getStartNode(headNode);
        System.out.println(startNode.getData());
    }

    /**
     * 单链表中，取出环的起始点
     * @param headNode 单链表中的首节点
     * @return 返回带环单链表中环的起始点
     */
    public static Node getStartNode(Node headNode) {
        // 1.获得带环单链表中环的长度
        int length = getCycleLength(headNode);
        // 2.处理 length 为 0 的情况，也就是处理单链表没有环的情况
        if (length == 0) {
            return null;
        }
        // 3.定义 first 和 second 指针，并且设置初始值为单链表的首节点
        Node first = headNode, second = headNode;
        // 4.让 first 指针往后移动 length 次
        for (int i = 0; i < length; i++) {
            first = first.getNext();
        }
        // 5.定义一个循环，用于获得带环单链表中环的起始点
        while (first != second) {
            // 6.设置 first 和 second 每次往后移动一步
            first = first.getNext();
            second = second.getNext();
        }
        // 6.返回带环单链表中环的起始点
        return first;
    }

    /**
     * 从有环链表中，获得环的长度
     * @param headNode 单链表的首节点
     * @return 如果单链表有环，则返回环中节点的个数，否则返回 0
     */
    public static int getCycleLength(Node headNode) {
        // 1.获得快慢指针相交的节点
        Node meetNode = meetNode(headNode);
        // 2.处理 meetNode 为 null 的情况
```

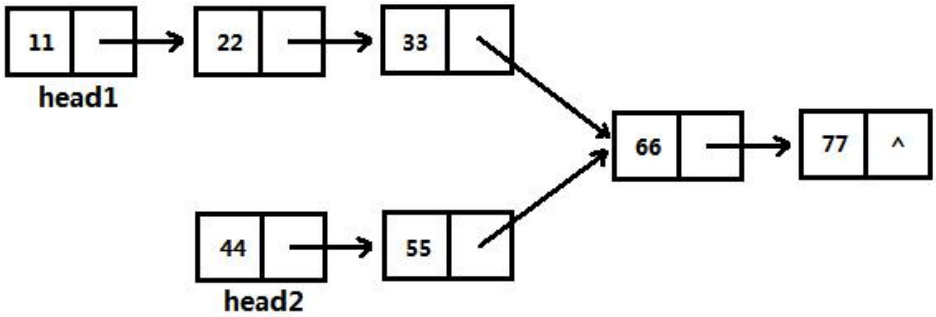
```
        if (meetNode == null) {
            return 0;
        }
        // 3.定义一个变量，用于保存环中节点的个数
        int size = 0;
        // 4.从 meetNode 节点开始，遍历环中的节点
        // 4.1 定义一个零时节点，用于辅助单链表的遍历
        Node tempNode = meetNode;
        // 4.2 定义一个死循环，用于遍历环中的所有节点
        while (true) {
            // 4.3 让 tempNode 指向它的下一个节点
            tempNode = tempNode.getNext();
            // 4.4 更新 size 的值
            size++;
            // 5.如果 tempNode 和 meetNode 指向的是同一个节点，那么就需要停止遍历操作
            if (tempNode == meetNode) {
                break;
            }
        }
        // 6.返回环中节点的个数
        return size;
    }

    /**
     * 获得快慢指针相交的节点
     * @param headNode 单链表的首节点
     * @return 如果单链表有环，则返回快慢指针相交的节点，否则返回 null。
     */
    public static Node meetNode(Node headNode) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            return null;
        }
        // 2.定义一个快指针，每次往后走两步
        Node fast = headNode;
        // 3.定义一个慢指针，每次往后走一步
        Node slow = headNode;
        // 4.定义一个循环，用于判断单链表是否有环
        while (fast != null && fast.getNext() != null) {
            // 5.设置快慢指针每次往后移动
            fast = fast.getNext().getNext();
            slow = slow.getNext();
            // 6.如果 fast 和 slow 指向的是同一个节点，则证明单链表有环
            if (fast == slow) {
                return fast;
            }
        }
```

```
        // 7.执行到此处，证明单链表是无环单链表
        return null;
    }
}
```

3.4.10 判断两个单链表相交的第一个交点

- **思路分析**

首先遍历获得两个单链表得到它们的长度（长度分别记为 len1 和 len2），定义 longNode 指针指向长度较长的单链表的首节点，定义 shortNode 指针指向长度较短的单链表的首节点，然后让 longNode 指针往后走 Math.abs(len1-len2)步。接着，定义一个循环，每次让 longNode 指针和 shortNode 指针往后移动一步，当 longNode 指针和 shortNode 指针指向同一个节点时，则停止循环。此时，longNode 和 shortNode 指向的节点，就是两个单链表相交的第一个交点。



- **代码实现**

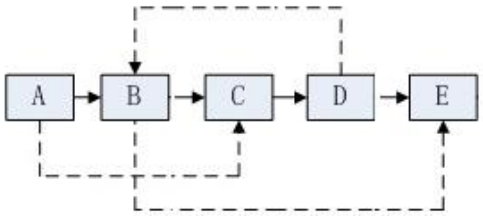
```
package com.bjsxt.p6.linkedlisttest;

/**
 * 获得两个单链表相交的第一个交点
 */
public class Test11 {
    public static void main(String[] args) {
        // 1.创建两个单链表
        Node lastNode = new Node(77);
        Node node6 = new Node(66, lastNode);
        Node node3 = new Node(33, node6);
        Node node2 = new Node(22, node3);
        Node head1 = new Node(11, node2);
        Node node5 = new Node(55, node6);
        Node head2 = new Node(44, node5);
        // 2.获得两个单链表相交的第一个交点
        Node commonNode = getFirstCommonNode(head1, head2);
        System.out.println(commonNode.getData());
    }

    /**
     * 获得两个单链表相交的第一个交点
```

```
     * @param head1 单链表 1 的首节点
     * @param head2 单链表 2 的首节点
     * @return 返回两个单链表相交的第一个交点
     */
    public static Node getFirstCommonNode(Node head1, Node head2) {
        // 1.处理 head1 或 head2 为 null 的情况
        if (head1 == null || head2 == null) {
            return null;
        }
        // 2.获得以 head1 为首节点的单链表长度
        int length1 = getLength(head1);
        // 3.获得以 head2 为首节点的单链表长度
        int length2 = getLength(head2);
        // 4.定义 longNode 指针，用于指向长度较长单链表的首节点
        Node longNode = length1 > length2 ? head1 : head2;
        // 5.定义 shortNode 指针，用于指向长度较短单链表的首节点
        Node shortNode = length1 > length2 ? head2 : head1;
        // 6.让 longNode 指针往后移动 Math.abs(length1-length2)次
        for (int i = 0; i < Math.abs(length1 - length2); i++) {
            longNode = longNode.getNext();
        }
        // 7.定义一个循环，每次让 longNode 和 shortNode 往后移动一次
        while (longNode != shortNode) {
            longNode = longNode.getNext();
            shortNode = shortNode.getNext();
        }
        // 8.返回两个单链表相交的第一个交点
        return longNode;
    }

    /**
     * 获得单链表的长度
     * @param headNode 单链表的首节点
     * @return 返回单链表的长度
     */
    public static int getLength(Node headNode) {
        // 1.定义一个变量，用于保存单链表的长度
        int length = 0;
        // 2.定义一个循环，用于实现单链表的遍历操作
        Node tempNode = headNode;
        while (tempNode != null) {
            tempNode = tempNode.getNext();
            // 3.更新 length 的值
            length++;
        }
        // 4.返回单链表的长度
        return length;
```

```
    }
}
```

3.4.11 复杂链表的复制

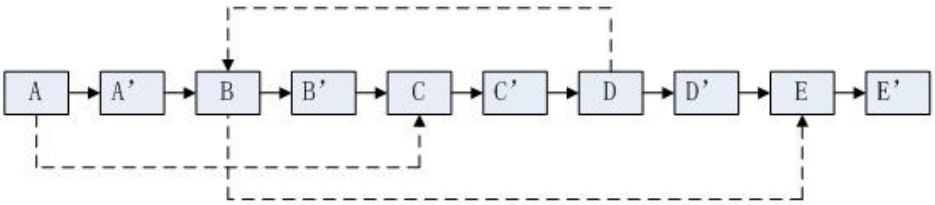
- **题目描述**



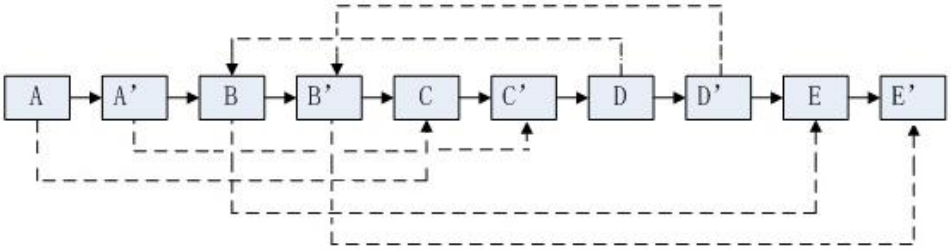
输入一个复杂链表（每个节点由三部分组成：data 用于保存节点的数据值，next 指针用于指向下一个节点，random 指针用于指向任意一个节点），返回结果为复制后复杂链表的首节点。

- **思路分析**

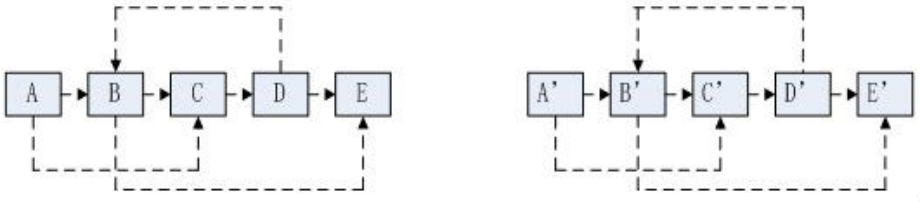
第一步：复制原始链表上的每一个节点 N，复制的结果就是对应的节点 N′ ，然后把 N′ 放在 N 之后，如下图所示：



第二步：设置每个 N'节点的 random 指针。如果原始链表上的节点 N 的 random 指向 S，则对应的复制节点 N′ 的 random 指向 S′ ，如下图所示：



第三步：把长链表拆分为两个链表。把奇数位置的节点用 next 连接起来就是原始链表，把偶数位置的节点用 next 连接起来就是复制出来的链表，如下图所示：



- **代码实现**

```java
package com.bjsxt.p6.linkedlisttest;

/**
 * 复杂链表的复制
 */
public class Test12 {
    public static void main(String[] args) {
        // 1.创建一个复杂链表
        Node lastNode = new Node("E");
        Node node4 = new Node("D", lastNode);
        Node node3 = new Node("C", node4);
        Node node2 = new Node("B", node3);
        Node headNode = new Node("A", node2);
        headNode.random = node3;
        node2.random = lastNode;
        node4.random = node2;
        // 2.复杂链表的复制
        Node cloneNode = cloneNode(headNode);
        System.out.println();
    }

    /**
     * 复杂链表的复制
     * @param headNode 复杂链表的首节点
     * @return 返回复制后复杂链表的首节点
     */
    public static Node cloneNode(Node headNode) {
        // 1.处理 headNode 为 null 的情况
        if (headNode == null) {
            return null;
        }
        // 2.复制复杂链表中的每一个节点，并且把复制的节点添加到被复制节点的后面
        Node curNode = headNode;
        while (curNode != null) {
            // 创建一个复制节点（N'）
            Node node = new Node(curNode.data);
            // 设置 node 节点的 next 值为 curNode 的后一个节点
            node.next = curNode.next;
            // 设置 curNode 的 next 值为 node
            curNode.next = node;
            // 更新 curNode 的值
            curNode = node.next;
        }
        // 3.设置每一个复制节点的 random 指针
        curNode = headNode;
        while (curNode != null) {
            // 获得复制节点（N'）
            Node node = curNode.next;
```

```
            // 处理 curNode 的 random 指针不为 null 的情况
            if (curNode.random != null) {
                // 设置 node 的 random 值为 curNode.random 的下一个节点
                node.random = curNode.random.next;
            }
            // 更新 curNode 的值
            curNode = node.next;
        }
        // 4.把长链表拆分为两个链表
        // 定义一个节点，用于保存拷贝后复杂链表的首节点
        Node head = headNode.next;
        curNode = headNode;
        while (curNode != null) {
            // 获得复制节点（N'）
            Node node = curNode.next;
            // 设置 curNode 的 next 值为 node.next
            curNode.next = node.next;
            // 设置 node 的 next 值为 node.next.next
            node.next = node.next == null ? null : node.next.next;
            // 更新 curNode 的值
            curNode = curNode.next;
        }
        // 5.返回拷贝后复杂链表的首节点
        return head;
    }

    /**
     * 节点类
     */
    private static class Node {
        /**
         * 用于保存节点中的数据
         */
        private Object data;
        /**
         * 用于保存指向下一个节点的地址值
         */
        private Node next;
        /**
         * 用于保存指向任意节点的地址值
         */
        private Node random;

        /**
         * 专门为 data 做初始化工作
         * @param data
         */
        public Node(Object data) {
```

```
            this.data = data;
        }

        /**
         * 专门为 data 和 next 做初始化工作
         * @param data
         * @param next
         */
        public Node(Object data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
}
```