# 数据结构与算法

老师：罗召勇

B站视频地址：<https://www.bilibili.com/video/BV1Zt411o7Rn?p=1>

# 第1章 概述

## 一、数据结构概述

### 1、什么是数据结构

数据结构：

数据结构是指相互之间存在着一种或多种关系的数据元素的集合和该集合中数据元素之间的关系组成。

简而言之：数据结构就是数据之间的关系。

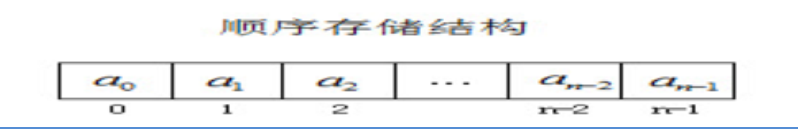
### 2、数据的存储结构

数据的存储结构：

即数据是以什么方式，存入内存中的

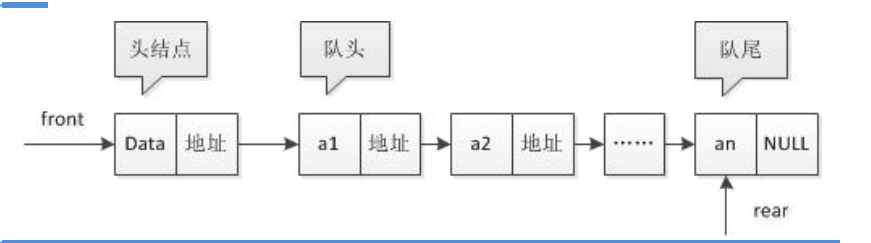
2.1顺序存储结构

即数组结构



2.2链式存储结构

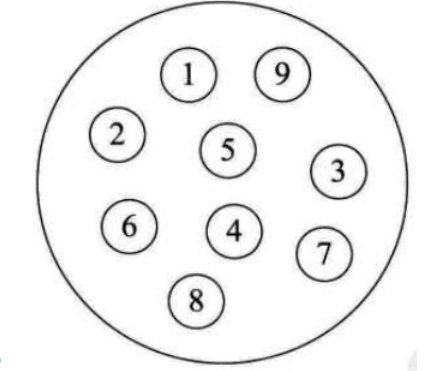
即链表结构



### 3、数据的逻辑结构

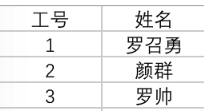
3.1集合结构

数据之间是相互独立的，没有关系



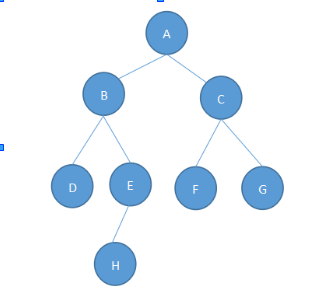
3.2线性结构

数据之间是一对一的关系



3.3树形结构

数据之间是一对多关系



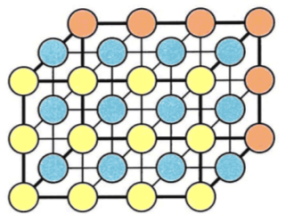
比如：

文件夹就是树形结构，一个文件夹里面可以有多个子文件夹...

Xmind思维导图也是树形结构。

3.4图形结构

数据之间关系比较复杂



## 二、算法概述

1、算法定义：

同一问题的不同解决方法，就是算法。

2、算法的特性

* 输入：算法可以有0到多个输入
* 输出：每个算法至少要有1个输出
* 有穷性：一个算法不能无限执行，必须能在有限时间内得出结果
* 确定性：1个输入，对应一个输出，结果是确定的
* 可行性：算法要能解决实际问题（废话）

3、算法的基本要求

正确性

可读性

健壮性

时间复杂度：算法执行占用的时间

空间复杂度：算法执行占用的空间

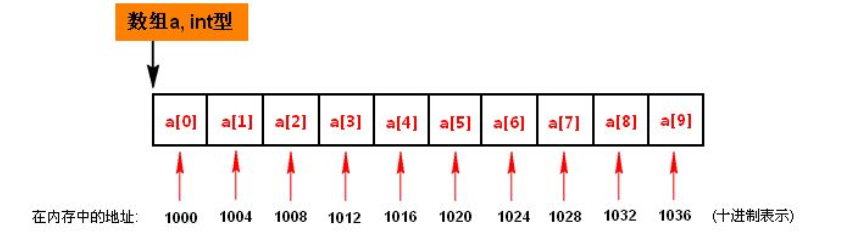
算法没有最好的，只有最适合的。

# 第2章 线性结构

## 一、数组

### 1、数组的基本使用

数组：



#### 1.1添加元素1

数组的基本用法参考以前的《笔记1》，这里就不再叙述了

老师这里介绍的是最简单的，在数组的末尾添加元素。

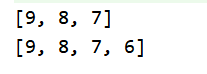
拓展：

还得学会在指定索引处添加元素。（未学）

代码：

**public static void** main(String[] args) {  
 //解决数组长度不可变的问题  
 **int**[] arr = **new int**[]{9, 8, 7};  
 //快速查看数组中给的元素  
 System.***out***.println(Arrays.*toString*(arr));//[9, 8, 7]  
 //要插入的目标元素  
 **int** dest = 6; //destination目标的缩写  
 /\*下面是算法\*/  
 //创建一个新的数组，长度是原数组的长度+1。千万注意这里[]中是数组的长度，而非索引  
 **int**[] newArr = **new int**[arr.**length** + 1];  
 //遍历原数组，把原数组中的所有元素复制到新数组中  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 newArr[i] = arr[i];  
 }  
 //把目标元素放入新数组的最后  
 newArr[arr.**length**] = dest;  
 //用新数组替换原数组（这步注意）  
 arr = newArr;  
 System.***out***.println(Arrays.*toString*(arr));//[9, 8, 7, 6]  
}

效果：



#### 1.2添加元素2

添加元素到指定索引处

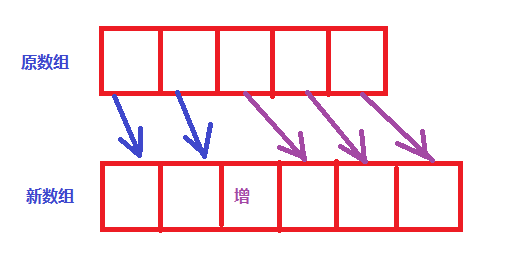
思路：

其实我觉得和后面删除元素的思想类似，都是遍历比较长的那个数组。

删除元素时，是在原数组中，以要删除的元素为分界线，原数组中左边元素索引与新数组的一致，右边元素索引比新数组多1。

添加元素时，是在新数组中，以要增加的元素为分界线，新数组中左边元素索引与原数组的一致，右边元素索引比原数组多1。

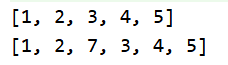
（其实不管是遍历新数组还是原数组都可以，只是我自己一开始想到的是遍历比较长的那个数组，方便以新增的元素为分界线，所以在添加元素的时候，我这里写的是遍历新数组。）



代码：

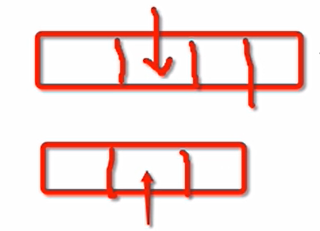
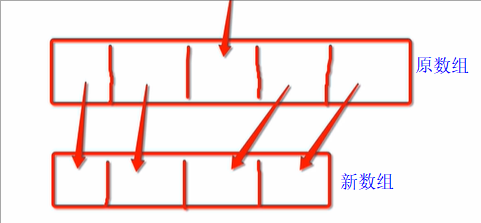
//数组\_添加元素到指定索引处  
**public static void** main(String[] args) {  
 //定义原数组  
 **int**[] arr = **new int**[]{1, 2, 3, 4, 5};  
 //查看原数组初始值  
 System.***out***.println(Arrays.*toString*(arr));//[1, 2, 3, 4, 5]  
 //要新增的元素  
 **int** element = 7;  
 //要新增的元素的索引  
 **int** elementIndex = 2;  
  
 /\*下面是算法\*/  
 //定义一个新的数组，长度是原数组长度+1  
 **int**[] newArr = **new int**[arr.**length** + 1];  
 //遍历新数组  
 **for** (**int** i = 0; i < newArr.**length**; i++) {  
 //新数组中，在新增元素的左边，索引与原数组一致，直接复制  
 **if**(i < elementIndex){  
 newArr[i] = arr[i];  
 //插入新增元素  
 }**else if**(i == elementIndex){  
 newArr[i] = element;  
 //新数组中，在新增元素的右边，索引比原数组多1,i是新数组的索引  
 }**else** {  
 newArr[i] = arr[i - 1];  
 }  
 }  
 //用新数组替换原数组  
 arr = newArr;  
 System.***out***.println(Arrays.*toString*(arr));//[1, 2, 7, 3, 4, 5]  
}

效果：



#### 1.3删除元素

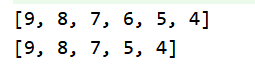
思路：



代码：

//数组\_删除1个指定索引的元素  
**public static void** main(String[] args) {  
 //定义一个数组，作为原数组  
 **int**[] arr = **new int**[]{9, 8, 7, 6, 5, 4};  
 //原数组中要删除的元素的索引  
 **int** index = 3;  
 //原数组初始数据  
 System.***out***.println(Arrays.*toString*(arr));//[9, 8, 7, 6, 5, 4]  
  
 /\*下面是算法\*/  
 //创建一个新数组，长度是原数组的长度-1  
 **int**[] newArr = **new int**[arr.**length** - 1];  
 //遍历新数组  
 **for** (**int** i = 0; i < newArr.**length**; i++) {  
 //index左边的元素，原数组中元素的索引与新数组一致，故直接复制  
 **if**(i < index){  
 newArr[i] = arr[i];  
 //index自己对应的元素跳过，原数组中index右边的元素索引比新数组大1  
 }**else** {  
 newArr[i] = arr[i + 1];//关键理解这一步  
 }  
 }  
 //用新数组替换旧数组  
 arr = newArr;  
 System.***out***.println(Arrays.*toString*(arr));//[9, 8, 7, 5, 4]  
}

效果：



### 2、面向对象的数组

其实就是描述了集合的底层是如何实现的，老师写了一个简易版的，我就不重复写了。

因为我看了下，其实用到的核心基础知识，就是上面基本使用中的2个小问题：

添加元素、删除元素。

把这两个玩熟练就可以了。

听完老师这部分的讲解，我个人感觉集合就可以看作是数组的API，提供了很多方便操作数组的方法。CRUD等。

### 3、查找算法

这里介绍的主要是针对于数组的。

#### 3.1线性查找

需求：

查找数组中指定元素的索引

代码：

//定义一个数组  
**private int**[] **arr** = **new int**[]{2, 3, 5, 6, 8, 4, 9};  
//查找数组中指定元素的索引  
@Test  
**public void** test1(){  
 //目标元素  
 **int** target = 8;  
 //目标元素的索引  
 **int** index = -1;  
 **for** (**int** i = 0; i < **arr**.**length**; i++) {  
 **if**(**arr**[i] == target){  
 index = i;  
 **break**;  
 }  
 }  
 System.***out***.println(**"index = "** + index);  
}

效果：



Tips：

上述代码中，如果有重复元素的话，查到的是第一个元素的索引

#### 3.2二分法查找

需求：

查找数组中指定元素的索引。

局限性：

这种思路不论数组长度是奇数还是偶数都可以。但是只适用于元素大小已经按序排列好了的数组，

下面的代码演示的情况只适用于元素大小已经排好序，并且排序方式为为升序，即随着索引的增加，元素越来越大。

代码：

//二分法查找元素的索引  
@Test  
**public void** test2() {  
 //目标数组  
 **int**[] arr = **new int**[]{1, 2, 3, 4, 5, 6, 7, 8, 9};  
 //目标元素  
 **int** target = 9;  
 //记录开始索引  
 **int** begin = 0;  
 //记录结束索引  
 **int** end = arr.**length** - 1;  
 //记录中间元素的索引(理解这一步)  
 /\*当数组长度为奇数时，中间元素只有一个  
 \* 当数组长度为偶数时，中间元素有两个，mid对应其中索引较小的元素\*/  
 **int** mid = (begin + end) / 2;  
 //记录目标元素的位置  
 **int** index = -1;  
 //循环查找  
 **while** (**true**){  
 //当元素不存在时,千万注意，不能加上等于=，  
 //存在begin=end的情况，而且元素也是存在的，比如target=9或target=1时  
 **if**(begin > end){  
 **break**;  
 }  
 //比中间元素小，说明目标元素在左边  
 **if**(target < arr[mid]){  
 end = mid - 1;  
 }**else if**(target == arr[mid]){  
 index = mid;  
 **break**;  
 //比中间元素大，说明秒元素在右边  
 }**else** {  
 begin = mid + 1;  
 }  
  
 //更新中间元素的索引(这一步千万别忘，否则就死循环了)  
 mid = (begin + end) / 2;  
 }  
 System.***out***.println(**"index = "** + index);  
}

效果：



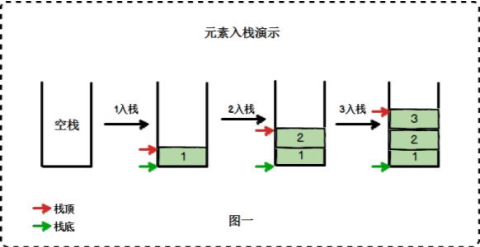
## 二、栈

栈要与队列一起对比理解：

栈：出口入口是同一个。类似于升降电梯，先进去的人，最后出来。

队列：出口入口在两端。先进先出。

图解：



栈的基本操作：

push：压入元素。其实完全等同于在原数组最后添加一个元素。

pop：推出栈顶元素。其就是将原数组的最后一个元素删除，并返回这个元素。

用到的基本知识，也就是前面数组中的基本操作，就不展示代码了。

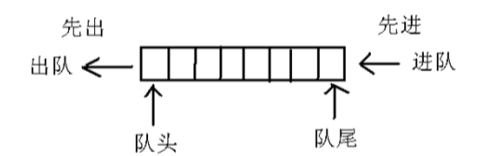
## 三、队列

### 1、概述

队列：

入口和出口在两端。先进先出。

图解：





队列的基本操作：

入队：其实和入栈操作原理完全一样。就相当于在原数组的最后添加一个元素。

出队：

这里就和出栈不一样了。出队是删除原数组中的第一个元素，即删除索引为0的元素，并将这个元素返回。

代码就不展示了，这个只需删除数组中的第一个元素，比之前数组基本操作中的删除指定索引的元素还简单，甚至直接套用删除指定元素的思想也可以。

### 2、队列类

代码：

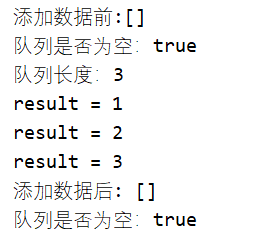
*/\*\*  
 \* 队列  
 \*/*@Data  
**public class** Myqueue {  
 **private int**[] **elements**;  
  
 //无参构造，将成员elements初始化，得到一个int类型的数组，长度为0  
 **public** Myqueue() {  
 **this**.**elements** = **new int**[0];  
 }  
  
 //入队：即在数组elements的末尾添加一个元素  
 **public void** add(**int** element){  
 //定义一个新数组，长度是原数组elements的长度+1  
 **int**[] newArr = **new int**[**elements**.**length** + 1];  
 //遍历原数组，将原数组中的全部元素存入新数组中  
 **for** (**int** i = 0; i < **elements**.**length**; i++) {  
 newArr[i] = **elements**[i];  
 }  
 //将要添加的元素element添加到新数组的最后  
 newArr[**elements**.**length**] = element;  
 //用新数组替换原数组(不要忘记这一步)  
 **elements** = newArr;  
 }  
  
 //出队:取出数组中的第一个元素，即索引为0的元素，并将原数组中的该元素删除  
 **public int** poll(){  
 //要返回的元素  
 **int** element = **elements**[0];  
 //定义一个新数组，长度为原数组elements的长度-1  
 **int**[] newArr = **new int**[**elements**.**length** - 1];  
 //遍历新数组，将原数组中索引0右边的元素全部赋值到新数组中  
 **for** (**int** i = 0; i < newArr.**length**; i++) {  
 newArr[i] = **elements**[i + 1];  
 }  
 //用新数组替换原数组  
 **elements** = newArr;  
 //这里很容易错，千万注意不要写成了elements[0]，  
 //因为原数组中已经更新了，现在的elements[0]已经不再是更新前的elements[0]了  
 //这也是为什么我们要在一开始，就将elements[0]存储到element中的原因  
 **return** element;  
 }  
  
 //判断原数组elements长度是否为空  
 **public boolean** isEmpty(){  
 **return elements**.**length** == 0;  
 }  
  
 //返回队列中元素的个数  
 **public int** length(){  
 **return this**.**elements**.**length**;  
 }  
  
 //查看队列中的元素  
 @Override  
 **public** String toString() {  
 **return** Arrays.*toString*(**elements**);  
 }  
}

### 3、测试使用

代码：

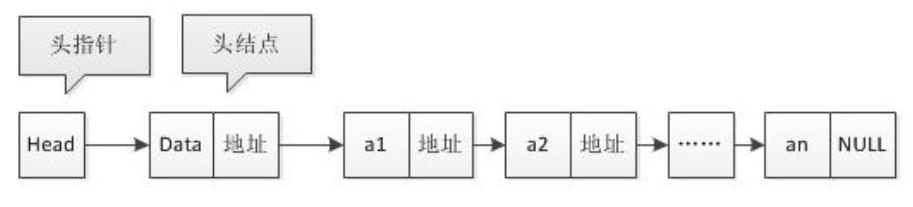
//测试队列  
@Test  
**public void** test1() {  
 Myqueue queue = **new** Myqueue();  
 System.***out***.println(**"添加数据前:"** + queue);  
 System.***out***.println(**"队列是否为空："** + queue.isEmpty());  
 queue.add(1);  
 queue.add(2);  
 queue.add(3);  
 System.***out***.println(**"队列长度："** + queue.length());  
 **while** (!queue.isEmpty()){  
 **int** result = queue.poll();  
 System.***out***.println(**"result = "** + result);  
 }  
 System.***out***.println(**"添加数据后: "** + queue);  
 System.***out***.println(**"队列是否为空："** + queue.isEmpty());  
}

效果：



## 四、单链表

图解：



### 1、节点基本用法

我们创建一个类，作为节点，取名为Node。代表的就是下图：



它至少需要包含2个东西：

① 节点内容，即数据

② 下一个节点

#### 1.1 Node类

目前该类中包括两个成员变量：

data：数据

next：下一个节点

提供的方法：

append：在后面追加节点

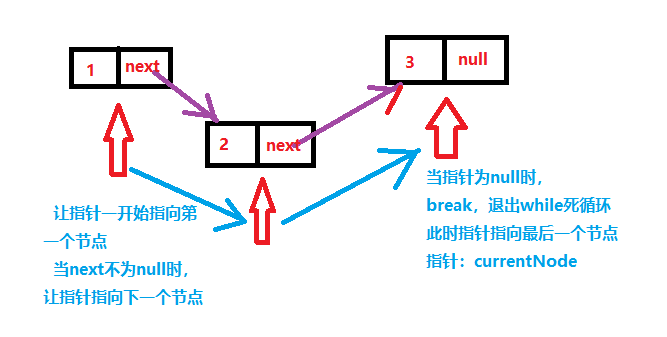
next：获取成员next，即获取下一个节点。

getData：获取成员data，即获取数据。

isLast：判断next是否为null，即判断当前节点是不是最后一个节点。

*/\*\*  
 \* 节点类  
 \*/***public class** Node {  
 //节点内容，即数据  
 **int data**;  
 //下一个节点  
 Node **next**;  
  
 //创建Node对象时，传入数据，对data进行初始化  
 **public** Node(**int** data) {  
 **this**.**data** = data;  
 }  
  
 //追加：即传入一个node，将其赋值给最后一个节点的成员next  
 **public** Node append(Node node){  
 //设置一个指针，初始指向当前节点  
 Node currentNode = **this**;  
 **while** (**true**){  
 //获取指针的下一个节点  
 Node nextNode = currentNode.**next**;  
 **if** (nextNode == **null**){  
 **break**;  
 }  
 //当指针的下一个节点不为null时，让指针指向下一个节点  
 currentNode = nextNode;  
 }  
 //这样while循环结束后，指针就已经指向最后一个节点了  
 //将node复制给指针的成员next，即将node复制给最后一个节点的成员next  
 currentNode.**next** = node;  
 //返回当前节点,即返回调用append方法的对象。  
 **return this**;  
 }  
 //获取下一个节点。即获取成员next  
 **public** Node next(){  
 **return this**.**next**;  
 }  
  
 //获取当前节点的数据  
 **public int** getData(){  
 **return this**.**data**;  
 }  
  
 //判断当前节点是不是最后一个  
 **public boolean** isLast(){  
 **return this**.**next** == **null**;  
 }  
}

append方法的思路分析：

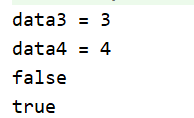


#### 1.2测试类

代码：

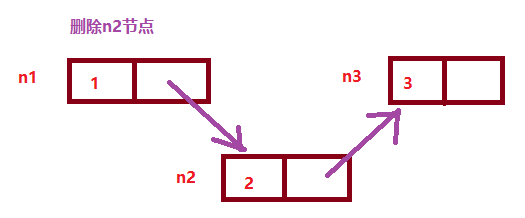
*/\*\*  
 \* node的测试类  
 \*/***public class** TestNode {  
 **public static void** main(String[] args) {  
 //创建节点  
 Node n1 = **new** Node(1);  
 Node n2 = **new** Node(2);  
 Node n3 = **new** Node(3);  
 //在n1节点后面追加节点n2、n3、以及第四个匿名节点  
 n1.append(n2).append(n3).append(**new** Node(4));  
 //获取第三个节点的数据，即获取第三个节点的成员next的data属性。  
 **int** data3 = n1.next().next().getData();  
 //获取第四个节点的数据  
 **int** data4 = n1.next().next().next().getData();  
 System.***out***.println(**"data3 = "** + data3);//data3 = 3  
 System.***out***.println(**"data4 = "** + data4);//data4 = 4  
 //判断当前是不是最后一个节点  
 System.***out***.println(n1.isLast());//false  
 System.***out***.println(n3.**next**.isLast());//true  
 }  
}

效果：



### 2、删除节点

将单链表中，当前节点的下一个节点删除



思路：

假设当前节点为n1，要删除下一个节点n2。

首先获取第三个节点n3，然后将其赋值给n1的成员next即可。

#### 2.1方法

只展示Node类中的方法

代码：

//删除当前节点的下一个节点  
**public void** removeNext(){  
 //获取下下个节点，即n3  
 Node newNext = **this**.next().next();  
 //将n3赋值给当前节点的成员next  
 **this**.**next** = newNext;  
}

#### 2.2测试

代码：

**public static void** main(String[] args) {  
 //创建节点  
 Node n1 = **new** Node(1);  
 Node n2 = **new** Node(2);  
 Node n3 = **new** Node(3);  
 //在n1节点后面追加节点n2、n3、以及第四个匿名节点  
 n1.append(n2).append(n3).append(**new** Node(4));  
  
 //删除n2节点的下一个节点n3  
 n2.removeNext();  
 System.***out***.println(n1.next().next().getData());//4  
}

效果：



### 3、展示节点

#### 3.1方法

注意：

这里我们实现的只是展示当前节点，以及当前节点后面的所有节点的data。

但是并没有展示当前节点前面的节点的data。

代码：

//展示当前节点，及其后面所有节点信息，即展示每个节点的成员data  
//注意，当前节点前面的节点没有展示  
**public void** show(){  
 //设置一个指针，初始指向当前节点  
 Node currentNode = **this**;  
 //注意这样写，当指针指向最后一个节点的时候，不会进入while循环  
 //因此while循环结束后，要单独再打印最后一个节点的data  
 **while** (currentNode.next() != **null**){  
 //当指针的成员next不为null时，即指针指向的不是最后一个节点时  
 System.***out***.print(currentNode.getData() + **" "**);  
 //打印完后，让指针指向下一个节点  
 currentNode = currentNode.next();  
 }  
 //展示最后一个节点的data  
 System.***out***.println(currentNode.getData());  
}

#### 3.2测试

代码：

**public static void** main(String[] args) {  
 //创建节点  
 Node n1 = **new** Node(1);  
 Node n2 = **new** Node(2);  
 Node n3 = **new** Node(3);  
 //在n1节点后面追加节点n2、n3、以及第四个匿名节点  
 n1.append(n2).append(n3).append(**new** Node(4));  
 n1.show();//1 2 3 4  
 n3.show();//3 4  
}

效果：

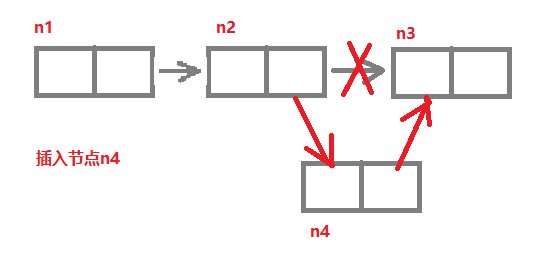


### 4、插入节点

需求：

在当前节点后面插入一个新的节点

图解：



思路：

比如以上图为例，我们把n2看作是当前节点，我们要在n2节点与n3节点之间插入一个新的节点n4。

首先找到节点n3，然后将n4节点赋值给n2节点的成员next。

再将n3赋值给n4节点的成员next。就结束了

#### 4.1方法

代码：

//插入节点：在当前节点的后面，插入一个新的节点  
**public void** after(Node n4){  
 //当前节点的下一个节点n3  
 Node n3 = **this**.next();  
 //将新节点n4赋值给当前节点n2的成员next  
 **this**.**next** = n4;  
 //将节点n3赋值给节点n4的成员next  
 n4.**next** = n3;  
}

#### 4.2测试

代码：

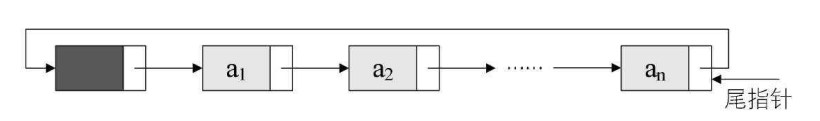
**public static void** main(String[] args) {  
 //创建节点  
 Node n1 = **new** Node(1);  
 Node n2 = **new** Node(2);  
 Node n3 = **new** Node(3);  
 Node n4 = **new** Node(4);  
 //在n1节点后面追加节点n2、n3  
 n1.append(n2).append(n3);  
 //增加节点n4之前，所有节点的信息  
 n1.show();//1 2 3  
 //在n2节点的后面增加一个新节点n4  
 n2.after(n4);  
 //增加节点n4后，所有节点的信息  
 n1.show();//1 2 4 3  
}

效果：



## 五、循环链表

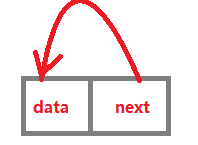
图解：



最后一个节点的成员变量next为第一个节点。

换言之，循环链表中，没有最后一个节点。

### 1、基本类



代码：

*/\*\*  
 \* 循环链表  
 \*/*@Data  
**public class** LoopNode {  
 //节点内容，即数据  
 **private int data**;  
 //下一个节点  
 **private** LoopNode **next** = **this**;  
  
 **public** LoopNode(**int** data) {  
 **this**.**data** = data;  
 }  
}

lombok坐标：

<!--lombok-->  
<**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.16.22</**version**>  
</**dependency**>

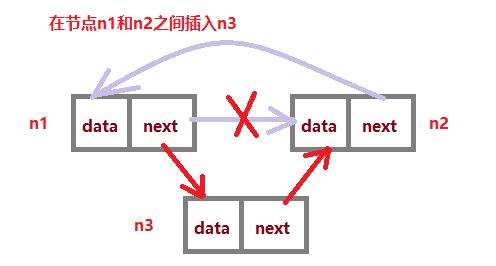
### 2、插入节点

#### 2.1概述

需求：

在当前节点后面插入一个新的节点。

图解：



分析：

其实思路和单链表的思路完全一样，甚至删除方法代码直接套用都行。

假设当前节点为n1，要在n1与n2节点之间插入一个节点n3。

将n3节点赋值给n1节点的成员next。

将n2节点赋值给n3节点的next

#### 2.2方法

代码：

//插入节点：在当前节点后面插入一个新节点  
**public void** after(LoopNode n3){  
 //获取当前节点后面的节点n2  
 LoopNode n2 = **this**.**next**;  
 //将新节点n3赋值给当前节点的成员next  
 **this**.**next** = n3;  
 //将n2赋值给新节点n3的成员next  
 n3.**next** = n2;  
}

#### 2.3测试

代码：

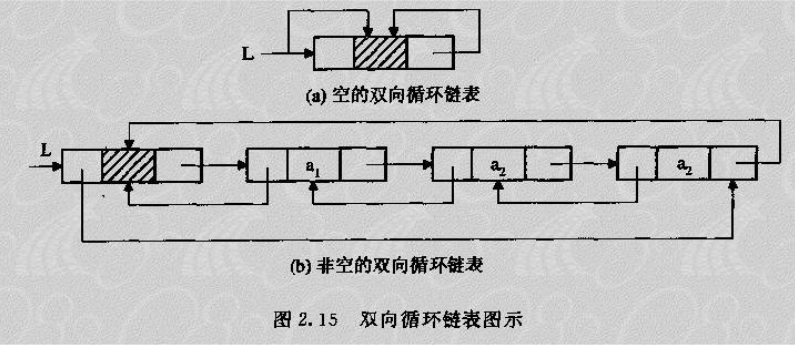
**public class** LoopNodeTest {  
 //定义3个成员，3个都是独立的循环链表，自己跟自己循环  
 **private** LoopNode **n1** = **new** LoopNode(1);  
 **private** LoopNode **n2** = **new** LoopNode(2);  
 **private** LoopNode **n3** = **new** LoopNode(3);  
 **private** LoopNode **n4** = **new** LoopNode(4);  
  
 //插入节点  
 @Test  
 **public void** test1() {  
 //增加节点  
 **n1**.after(**n2**);  
 **n2**.after(**n3**);  
 **n3**.after(**n4**);  
 System.***out***.print(**n1**.getNext().getData() + **" "**);//2  
 System.***out***.print(**n2**.getNext().getData() + **" "**);//3  
 System.***out***.print(**n3**.getNext().getData() + **" "**);//4  
 System.***out***.print(**n4**.getNext().getData() + **" "**);//1  
 }  
}

效果：



## 六、双向循环链表

图解：



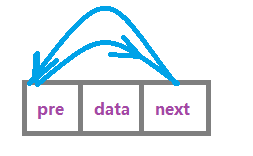
前面学的链表都是单项的，即只能找到当前节点的下一个节点。

在双向循环链表中，每个节点既能找到当前节点的下一个节点，也能找到它的上一个节点。并且由于循环的特点，因此该链表中是没有最后一个节点的。

### 1、基本类

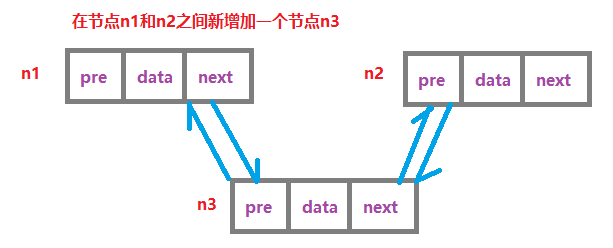
*/\*\*  
 \* 双向循环链表  
 \*/*@Data  
**public class** DoubleNode {  
 //节点的内容：即数据  
 **private int data**;  
 //上一个节点  
 **private** DoubleNode **pre** = **this**;  
 //下一个节点  
 **private** DoubleNode **next** = **this**;  
  
 **public** DoubleNode(**int** data) {  
 **this**.**data** = data;  
 }  
}

图解：



### 2、插入节点

#### 2.1思路



需求：

假设n1为当前节点，n2为n1后面的节点。

在节点n1和n2之间插入一个新的节点n3。

思路：

让n1的成员next指向n3，让n3的成员pre指向n1，即表示n1的后一个节点是n3，而n3的前一个节点是n1。

让n3的成员next指向n2，让n2的成员pre指向n3

#### 2.2方法

#### 2.3测试

##### 2.3.1插入前

首先，我们定义三个节点，刚定义的时候，他们是相互独立的

代码：

**public class** DoubleNodeTest {  
 //定义三个成员，现在是相互独立的，自己和自己循环  
 **private** DoubleNode **n1** = **new** DoubleNode(1);  
 **private** DoubleNode **n2** = **new** DoubleNode(2);  
 **private** DoubleNode **n3** = **new** DoubleNode(3);  
  
 //插入节点  
 @Test  
 **public void** test1() {  
 System.***out***.print(**n1**.getPre().getData() + **" "**);//1  
 System.***out***.print(**n1**.getNext().getData() + **" "**);//1  
 System.***out***.print(**n1**.getData() + **" "**);//1  
 }  
}

效果：



这说明节点1的上一个节点、下一个节点都是它自己。也说明现在刚定义的这3个节点都是自己和自己循环，彼此之间是相互独立的。

##### 2.3.2插入节点n2

//插入节点  
@Test  
**public void** test1() {  
 //插入节点  
 **n1**.after(**n2**);  
 System.***out***.print(**n1**.getPre().getData() + **" "**);//2  
 System.***out***.print(**n1**.getNext().getData() + **" "**);//2  
 System.***out***.print(**n1**.getData() + **" "**);//1  
}

效果：



这说明节点n1的上一个和下一个节点都是n2

##### 2.3.3插入节点n3

//插入节点  
@Test  
**public void** test1() {  
 //插入节点  
 **n1**.after(**n2**);  
 **n2**.after(**n3**);  
 System.***out***.print(**n1**.getPre().getData() + **" "**);//3  
 System.***out***.print(**n1**.getNext().getData() + **" "**);//2  
 System.***out***.print(**n1**.getData() + **" "**);//1  
}

效果：



这说明n1的上一个节点是n3，下一个节点时n2

## 七、递归

参考以前的笔记《基础300集》，里面关于递归的介绍更加详细。

这里只是简单温习一下。

递归：

就是方法自己调用自己。 recurrence。

注意事项：

必须要能停下来

构造方法，禁止递归

使用前提：

当调用方法的时候，方法的主体不变，只是每次调用方法时的参数不同。

### 1、入门案例

代码：

**public class** RecurrenceTest {  
  
 **public void** print(**int** i) {  
 //递归停止的条件  
 **if**(i > 0){  
 //每次递归，所干的事情  
 System.***out***.print(i + **" "**);  
 //调用自己，产生递归，  
 //并说明每次调用，参数如何变化  
 print(i - 1);  
 }  
 }  
  
 //测试print方法  
 @Test  
 **public void** test1() {  
 print(5);  
 }  
}

效果：



我认为，写递归方法的时候，需要明确的点：

* 递归停止的条件
* 当传入参数为n时，整个方法代表什么意义，这样就能理解传入参数为（n-1）时，整个方法代表的意义。（难点，也是关键）

只要完全清楚当传入参数为n时，整个方法代表的含义，所有问题，就都将迎刃而解。我们必须在清楚这一点的基础上再去写代码，或者读代码，就会轻松很多。

### 2、斐波那契列数

利用递归，求第n项斐波那契数。

代码：

//求第n项目斐波那契数列  
**public int** getFeibo(**int** n){  
 **if**(n == 1 || n == 2){  
 **return** 1;  
 }**else** {  
 //因为传入参数为n时，整个方法代表第n项，因此当参数  
 //为(n-1)和(n-2)时，整个方法分别代表第(n-1)项、第(n-2)项  
 **return** getFeibo(n - 1) + getFeibo(n - 2);  
 }  
}  
@Test  
**public void** test2() {  
 //斐波那契数列：1 1 2 3 5 8 13...  
 //规律：a(n) = a(n-1) + a(n-2)  
 **int** result = getFeibo(7);  
 System.***out***.println(**"result = "** + result);//result = 13  
}

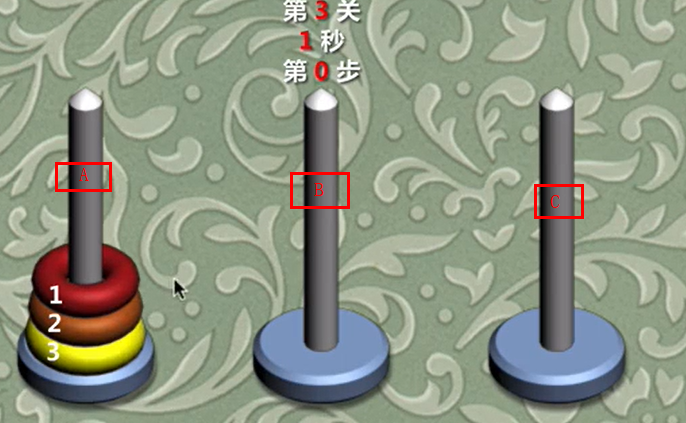
效果：



### 3、汉诺塔问题

#### 3.1概述

问题描述：



如上图：

需要把A柱上的三个环，全部移动到C柱上，就算完成。

要求是：

1、每次只能移动一个环

2、小环不能被放在大环的下面。

第一次，A柱上有1个环

第二次，A柱上有2个环

一次类推

第n次，A柱上有n个环

这就是汉诺塔问题。

分析：

假设初始A柱上有n个环。

n=2时，A柱上有2个环：1和2，我们是先将1放在B柱上，再将2放到C柱上，然后再将1放到C柱上。总之n=2的情况是可以解决的。

n=3时，A柱上有3个环：1、2、3。但我们仍然可以将其看作两个环，即将1、2整体看作一个环。我们是期望将1、2整体放在B柱上，再将3放到C柱上，然后再将1、2整体放到C柱上。这样n=3的情况是可以解决的。

这样一看，当我们将1、2看作一个整体时，进行的操作与n=2时的操作是完全一样的。

同理，n=4时，A柱上有4个环，我们还是将其看作成是2个环。即将1、2、3整体看作一个环。这样进行的操作就与n=2时的操作是完全一致的了。

小结：

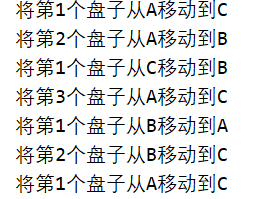
按操作的种类划分，要解决的只有2种情况，即n=1和n=2这两种情况，因为当n>=2时，都可以看作是n=2的情况来处理。

#### 3.2代码

代码：

*/\*\*  
 \* 递归\_\_汉诺塔问题  
 \*/***public class** RecurrenceTest1 {  
 **public static void** main(String[] args) {  
 *hanoi*(3, **'A'**, **'B'**, **'C'**);  
 }  
  
 */\*\*  
 \*  
 \** ***@param n*** *当一共有n个盘子时  
 \** ***@param from*** *从from柱开始移动  
 \** ***@param rest*** *除了from柱、和to柱之外的那个柱子，不一定就是中间的那个柱子（注意）  
 \** ***@param dest*** *最终移动到to柱  
 \* 当第一个参数为n时，整个方法的含义：将这n个盘子，从from柱移动到dest柱  
 \*/* **public static void** hanoi(**int** n, **char** from, **char** rest, **char** dest ){  
 **if**(n == 1){  
 System.***out***.println(**"将第1个盘子从"** + from + **"移动到"** + dest );  
 }**else** {  
 //当n >= 2时  
 //将上面的n-1个盘子，从from柱，移动到rest柱上。即从左边的柱子，移动到中间的柱子上  
 *hanoi*(n-1, from, dest, rest);  
 System.***out***.println(**"将第"** + n + **"个盘子从"** + from + **"移动到"** + dest );  
 //将上面的n-1个盘子，从rest柱，移动到dest柱上。即从中间的柱子，移动到右边的柱子上  
 *hanoi*(n-1,rest, from, dest);  
 }  
 }  
}

效果：



测试正确。

小结：

hanoi（n，参数2，参数3，参数4）

关键是要弄懂当第一个参数为n时，整个方法的含义：表示将上面的n个盘子，从“参数2”代表的柱子，移动到“参数4”代表的柱子上。

有一个误区：误认为参数3代表的就一定是中间的柱子，这是不对的，这只不过是我们在main方法中调用hanoi方法时，给参数3传入的实参是B柱，即中间那个柱子，但是“参数3”真正的含义，是代表除了“参数2”、“参数4”之外，剩下的那个柱子，可不一定就是中间那个柱子。

# 第3章 排序算法

## 一、两个复杂度

1、如何衡量一个算法的优劣：

1.1时间复杂度

算法执行所需的时间

1.2空间复杂度

算法执行所占用的内存

2、语句频度T(n)：

一个算法中的语句执行次数称为语句频度，记为T(n)。

举例：



执行for循环时，需要执行很多次，这里老师将的是要执行n+1次，也就是100次，我不理解，我认为是执行了100次。

不过这不重要，总之我们很明显能观察到，for循环要得出结果，需要执行很多次，但是下面的直接计算，只需要执行一次。

3、时间复杂度

 一般情况下，算法中的基本操作语句的重复执行次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n) / f(n) 的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作

T(n)=Ｏ( f(n) )，称Ｏ( f(n) )  为算法的渐进时间复杂度，简称时间复杂度。

T(n) 不同，但时间复杂度可能相同。 如：T(n)=n²+5n+6 与 T(n)=3n²+3n+2 它们的T(n) 不同，但时间复杂度相同，都为O(n²)。

计算时间复杂度的方法：

* 去掉常数项
* 去掉低次项
* 去掉最高次项的系数

这么做的原因是，当n很大的时候，按照上面的3个规则得到的结果，与原结果相差不大，所以推出了这三个规则。

4、常见的时间复杂度：

* 常数阶O(1)
* 对数阶O(log2n)
* 线性阶O(n)
* 线性对数阶O(nlog2n)
* 平方阶O(n2)
* 立方阶O(n3)
* k次方阶O(nk)
* 指数阶O(2n)

随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

5、平均时间复杂度和最坏时间复杂度：

5.1平均时间复杂度：

是指所有可能的输入实例均以等概率出现的情况下，该算法的运行时间。

5.2最坏时间复杂度：

即最坏情况下的时间复杂度。一般讨论的时间复杂度都指的是最坏情况下的时间复杂度。

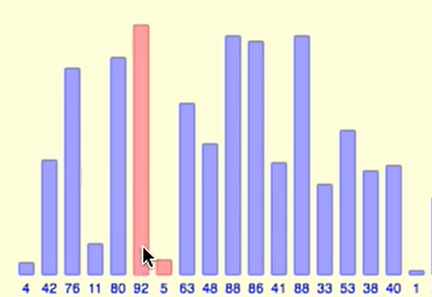
 这样做的原因是：如果最坏的情况都能容忍，那所有情况都能容忍了。

## 二、常用的排序算法

### 1、交换排序

#### 1.1冒泡排序

图解：



我的理解：

随意挑选一个元素，将它与另一个元素进行比较，筛选出二者中比较大的元素，再将结果与另一个元素比较，再次筛选出二者中的较大者，以此类推，最终会得到这堆元素中最大的元素。

然后再用这样的方法，可以找出第二大的元素，同理，以此类推，最终会将所有元素按照从小到大的顺序进行排列。

规律：

当数组长度为4时

第一轮 需比较3次 选出第一大的数

第二轮 需比较2次 选出第二大的数

第三轮 需比较1次 选出第三大的数

因此，当数组长度为n时

需比较 n-1轮

第i轮 需比较n-i次

而在for循环中

for(i = 0; i < arr.length-1; i++){}循环的次数就等于数组的长度arr.length，也就是n。

也就是说看for循环时，[0，n-1）在我的眼中就直接看作是n。

因此如果要表示n-i，就应该是[0，n-1-i)，用for循环来表示，就是

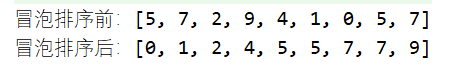
for(j = 0; arr.length-1-i; j++)。

理解了这个之后，就能更好的理解双循环了。

代码实现：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{5, 7, 2, 9, 4, 1, 0, 5, 7};  
 System.***out***.println(**"冒泡排序前："** + Arrays.*toString*(arr));  
 *bubbleSort*(arr);  
 System.***out***.println(**"冒泡排序后："** + Arrays.*toString*(arr));  
}  
  
*/\*\*  
 \* 数组长度为4时  
 \* 第1轮 比较3次  
 \* 第2轮 比较2次  
 \* 第3轮 比较1次  
 \* 故数组长度为n时，  
 \* 一共要 比较n-1轮  
 \* 第i轮 比较n-i次  
 \*/***public static void** bubbleSort(**int**[] arr){  
 //一共比较几轮，即比较数组长度n-1轮，  
 //直接把[0,n-1)看作就是数组长度n，后面-1，代表的就是n-1，这样更好理解  
 **for** (**int** i = 0; i < arr.**length** - 1; i++) {  
 //第i轮需要比较n-i次，而[0，n-1)代表的就是n，那么[0，n-1-i)代表的就是n-i  
 **for** (**int** j = 0; j < arr.**length** - 1 - i; j++) {  
 **if**(arr[j] > arr[j+1]){  
 //当左边>右边时，就将其互换，这样就能将最大的放在后面  
 //下面三行代码，不去深究，直接看作是互换  
 **int** temp = arr[j + 1];  
 arr[j + 1] = arr[j];  
 arr[j] = temp;  
 }  
 }  
 }  
}

执行结果：



#### 1.2快速排序

快速排序：

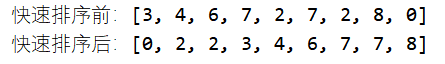


上图是我自己的理解。

代码实现：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{3, 4, 6, 7, 2, 7, 2, 8, 0};  
 System.***out***.println(**"快速排序前："** + Arrays.*toString*(arr));  
 *quickSort*(arr,0,arr.**length**-1);  
 System.***out***.println(**"快速排序后："** + Arrays.*toString*(arr));  
}  
  
//对数组arr中索引在[start,end]范围内的元素进行排序  
//排序的内容：每次都将arr[start]作为标准数，将[start,end]范围内的元素以arr[start]标准数为界限  
//分成左右两边，左边的元素都比标准数小，右边的元素都比标准数大，中间是标准数  
**public static void** quickSort(**int**[] arr, **int** start, **int** end){  
 **if** (start <= end) {  
 //将索引为start的元素作为标准数standard  
 **int** standard = arr[start];  
 //记录low指针和high指针的初始索引位置  
 **int** low = start;  
 **int** high = end;  
 //当low指针在high指针左边时，进入循环  
 **while** (low < high){  
 **while** (low<high && arr[high] >= standard){  
 //当high指针指向的元素>=标准值时，保持元素不变，让high指针左移，即high--  
 //注意这里用的while，而不是if。  
 //如果是if的话，那么high指针只会移动一次，如果是while的话，high指针就会一直左移  
 //直到high指针指向的元素<标准值，才会退出while循环。这就是while与if的区别，一定要注意。  
 high--;  
 }  
 //退出了while循环，说明此时high指针指向的元素<标准值  
 //那么就用high指针指向的元素，将low指针指向的元素进行覆盖  
 arr[low] = arr[high];  
 //然后开始玩low指针  
 //注意：进入外面的while循环的条件是low<high<为什么我们在里面的while循环这里又写了一次low<high呢？  
 //这难道不是重复了吗？  
 //其实不重复，因为大while里面有两个小while，进入外面的大while的时候，毫无疑问，肯定是满足low<high的，  
 //但是进入之后，第一个小while里面对high进行了各种操作，这样在退出第一个小while时，我们已经无法保证low<high了  
 //所以在第二个小while这里，必须加上low<while(这个细节一定要注意，很重要)  
 **while** (low<high && arr[low] <= standard){  
 //当low指针指向的元素<=标准值时，保持元素不变，让low指针一直右移  
 low++;  
 }  
 //退出了while循环，说明此时low指针指向的元素>标准值  
 //那么就用low指针指向的元素，将high指针指向的元素进行覆盖  
 arr[high] = arr[low];  
 }  
 //退出外面的while循环时，说明此时low=high,即low指针与high指针重合，都指向同一个元素  
 //那么就用标准值，将这个元素进行覆盖  
 arr[low] = standard;  
 //到此就算完成了一次递归，得到的结果是以arr[0]为标准数，  
 //arr[0]左边的元素都比arr[0]小，右边的元素都比arr[0]大  
 //对标准数arr[0]左边的数进行递归，此时low=high，此时arr[0]的索引为low或high  
 *quickSort*(arr,start,low-1);  
 //对标准数arr[0]右边的数进行递归  
 //这里注意第二个参数low + 1，这里一定要+1，这样传入quickSort的参数才会发生变化，最后才能停止递归。  
 *quickSort*(arr,low+1, end);  
 }  
}

效果：



### 2、插入排序

#### 2.1直接插入排序

##### 2.1.1概述

直接插入排序：

比如有一个数组arr，数组长度为n。一个指针，指针从左往右移动。

第一次，指针指向左边第二个元素。

第二次，指针指向左边第三个元素，然后让前面3个元素从小到大排列。

第三次，指针指向左边第四个元素，然后让前面4个元素从小到大排列。

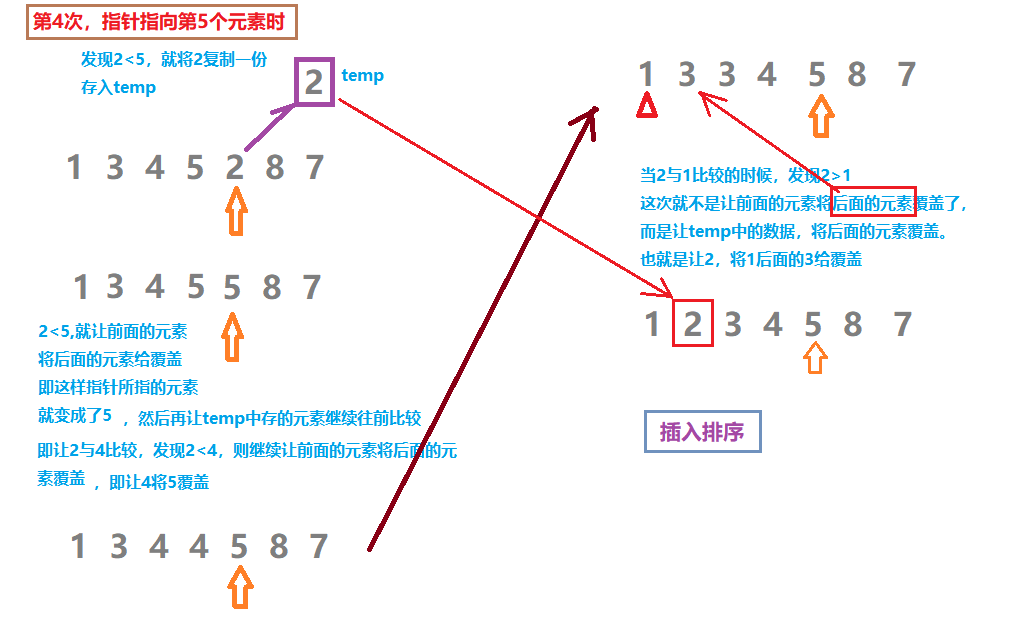
以此类推。

第n-1次，指针指向左边第n个元素，这样，前n个元素就完成了从小到达排列。

关键：

当指针指向第n个元素时，如何让前面n个元素完成排序。

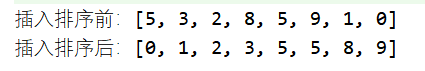
比如第4次，指针指向左边第5个元素，如何实现让前面5个元素完成排序。



##### 2.1.2代码实现

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{5, 3, 2, 8, 5, 9, 1, 0};  
 System.***out***.println(**"插入排序前："** + Arrays.*toString*(arr));  
 *insertSort*(arr);  
 System.***out***.println(**"插入排序后："** + Arrays.*toString*(arr));  
  
}  
  
/\*  
 插入排序  
 假设数组长度为n，那么  
 指针需要移动的次数：n-1  
 第i次，指针指向第i+i个元素,即arr[i]  
 \*/  
**public static void** insertSort(**int**[] arr){  
 //指针需要移动的次数n-1，指针每移动一次，前面的元素就需要排序一次  
 **for** (**int** i = 1; i <arr.**length** ; i++) {  
 //第i次，指针指向第i+1个元素，即arr[i]  
 //如果指针所指的元素，比左边的元素小，则需要排序  
 **if**(arr[i-1] > arr[i]){  
 //先将指针所指的元素存入temp  
 **int** temp = arr[i];  
 **int** j;  
 //反向遍历指针前面的元素，与temp进行比较。  
 **for**(j = i-1; j>=0 && arr[j]>temp;j--){  
 //当前面的元素>temp的时候，就将前面的元素复制给后面的元素  
 arr[j + 1] = arr[j];  
 }  
 //将temp赋值给后面的元素，就是图例中2>1的情况，或者是到了j=-1的情况。  
 arr[j+1] = temp;  
 }  
 }  
}

效果：



我个人感觉这个看图很简单，但是代码写起来不容易。

因为有个双循环：

arr[ i ]：当前指针指向的元素，将其存入temp

arr[ j ]: 反向遍历指针左边的元素时，被拿来与temp比较的元素。

双循环的时候，一定要清楚arr[ i ]和arr[ j ]各自代表的含义，否则很容易晕。

#### 2.2希尔排序

原理我懂，代码也勉强能看懂，但是我自己写的错了，我就不懂了。

直接插入排序有一个不足之处：

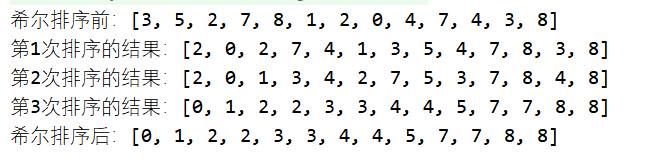
就是当后面出现一个很小的数字时，将会和前面所有比他大的数字都比较一遍，这样效率就很低了。

希尔排序的出现，就能稍微改进这个不足，因为它通过分组的机制，可以有几率将后面比较小的数字，提前排序到前面来。

代码：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{3, 5, 2, 7, 8, 1, 2, 0, 4, 7, 4, 3, 8};  
 System.***out***.println(**"希尔排序前："** + Arrays.*toString*(arr));  
 *shellSort*(arr);  
 System.***out***.println(**"希尔排序后："** + Arrays.*toString*(arr));  
}  
  
//老师写的  
**public static void** shellSort(**int**[] arr){  
 **int** k = 1;  
 //遍历所有步长  
 **for** (**int** d = arr.**length** / 2; d>0; d = d/2){  
 //遍历所有的元素  
 //因为索引在[0,d-1]的元素就是每一组的第一个元素  
 //这里i从d开始，代表遍历的元素范围是[每一组的第二个元素, 每一组的最后一个元素], 即每组的第一个元素就不遍历了  
 **for** (**int** i=d; i<arr.**length**; i++){  
 //遍历本组中所有的元素  
 **for** (**int** j = i-d; j >= 0; j = j-d){  
 //如果当前元素>加上步长后的那个元素，就将它们两个换位  
 **if**(arr[j] > arr[j+d]){  
 **int** temp = arr[j + d];  
 arr[j + d] = arr[j];  
 arr[j] = temp;  
 }  
 }  
 }  
 System.***out***.println(**"第"** + k + **"次排序的结果："** + Arrays.*toString*(arr));  
 k++;  
 }  
}

效果：



代码只是略懂。

我自己也写了个，但是有问题，不知道哪里错了

### 3、选择排序

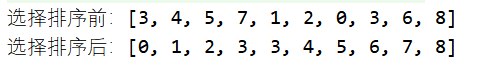
选择排序：

比如要给一堆数字进行排序。方法是，先找出这堆数字中最小的，放在第一位，然后再找出剩下的数字中最小的，放在第二位，以此类推，进行排序。这样的排序方法，叫选择排序。

代码：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{3, 4, 5, 7, 1, 2, 0, 3, 6, 8};  
 System.***out***.println(**"选择排序前："** + Arrays.*toString*(arr));  
 *selectSort*(arr);  
 System.***out***.println(**"选择排序后："** + Arrays.*toString*(arr));  
}  
//选择排序  
**public static void** selectSort(**int**[] arr){  
 //遍历所有元素  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 //定义最小元素的索引，初始值为当前遍历的元素的索引  
 **int** minIndex = i;  
 //遍历当前元素后面的所有元素，将arr[minIndex]与后面的每一个元素进行比较  
 //得出arr[minIndex]与后面所有元素之中，最小元素的索引，并将其赋值给minIndex  
 **for** (**int** j = i+1; j < arr.**length**; j++) {  
 //当后面被遍历的元素比arr[minIndex]还要小时，就将后面这个元素的索引赋值给minIndex  
 //千万注意：if条件中必须是arr[minIndex]，而不是arr[i](易错点)  
 //因为如果是arr[i]的话，比如arr[i]=3，后面的元素为1,2。那么最终minIndex会变成是元素2的索引  
 //也就是说,如果这里是arr[i]，那么得到的minIndex，将会是最后一个比arr[i]小的元素的索引，  
 //但是最后一个比arr[i]小的元素不一定就是最小的，比如2比3小，但是我们真正想要获取的是1的索引，  
 //所以这里必须是arr[minIndex]，这样就会将每次比较得到的最小值，再去与其他元素进行比较，而不是每次  
 //都只拿arr[i]去比较。  
 **if**(arr[minIndex] > arr[j]){  
 minIndex = j;  
 }  
 }  
 //后面的元素全部遍历完后，就知道了arr[i]与后面所有元素中最小元素的索引，即minIndex  
 **if** (minIndex != i){  
 //说明后面存在比arr[i]更小的元素，那么就将arr[i]与这个元素进行交换  
 **int** temp = arr[i];  
 arr[i] = arr[minIndex];  
 arr[minIndex] = temp;  
 }  
 }  
}

效果：



### 4、归并排序

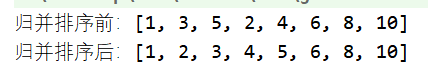
归并排序：



代码实现：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{1, 3, 5, 2, 4, 6, 8, 10};  
 System.***out***.println(**"归并排序前："** + Arrays.*toString*(arr));  
 *mergeSort*(arr,0,arr.**length**-1);  
 System.***out***.println(**"归并排序后："** + Arrays.*toString*(arr));  
}  
  
//归并排序：将数组中索引范围在[low,high]的元素进行排序  
**public static void** mergeSort(**int**[] arr, **int** low, **int** high){  
 **if** (low < high){  
 //定义中间位置mid，将arr分为左右两个数组  
 **int** mid = (low + high) / 2;  
 //将左边的数组进行排序  
 *mergeSort*(arr, low, mid);  
 //将右边的数组进行排序  
 *mergeSort*(arr, mid + 1, high);  
 //这样左右两边的数组就已经是排好序的了，这样就能使用merge方法，对该数组进行排序了  
 *merge*(arr, low, mid, high);  
 }  
}  
  
//对数组中索引范围在[low,high]内的元素进行排序  
//使用前提：左边的数组，与右边的数组必须是已经排好序的  
**public static void** merge(**int**[] arr, **int** low, **int** mid, **int** high){  
 //定义新数组，用于存储arr数组中，索引在[low,high]范围内的元素  
 **int**[] temp = **new int**[high - low + 1];  
 //定义指针j，指向左边的数组中的元素  
 **int** i = low;  
 //定义指针j，指向右边的数组中的元素  
 **int** j = mid + 1;  
 //定义指针Index，指向新数组temp中的元素  
 **int** index = 0;  
 //当左右两边的数组都没有遍历完时，即指针i、j都没有移到数组外面时  
 **while** (i <= mid && j <= high){  
 //当左边数组的元素更小时  
 **if**(arr[i] < arr[j]){  
 //将左边的元素存入新数组中  
 temp[index] = arr[i];  
 //指针i右移  
 i++;  
 }**else** {  
 //当右边数组的元素更小时，则将右边的元素存入新数组中,指针j右移  
 temp[index] = arr[j];  
 j++;  
 }  
 //新数组中每存一个元素，就将新数组的指针右移1位  
 index++;  
 }  
 //while循环结束后，说明左右两个数组中，肯定有一个数组的元素被存完了，即指针移到了数组外面。  
 //而另一个数组中仍有数组存留  
 //当右边的数组有多余元素时  
 **while** (j <= high){  
 //则将右边数组中的全部元素，直接存入新数组中  
 temp[index] = arr[j];  
 j++;  
 index++;  
 }  
 //当左边的数组有多余元素时  
 **while** (i <= mid){  
 //将左边的数组中全部元素，直接存入新数组中  
 temp[index] = arr[i];  
 i++;  
 index++;  
 }  
 //将新数组中的元素，全部存入原数组arr中  
 **for** (**int** k = 0; k < temp.**length**; k++) {  
 //这里也容易错，千万别忘了我们只是将arr中的[low,high]部分的元素存入了temp[]中  
 //因此temp[0]对应的是arr[low]  
 //因此temp[k]对应的是arr[k + low]  
 arr[k + low] = temp[k];  
 }  
}

效果：

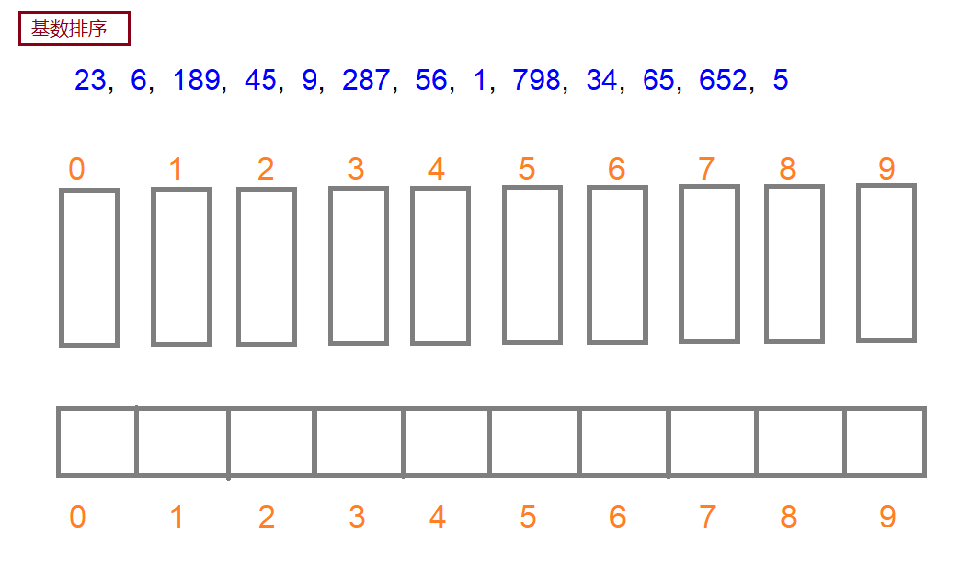


### 5、基数排序

#### 5.1基本用法

##### 5.1.1概述

基数排序：



上面的数组arr是要排序的数组，下面有10个桶，其实就是10个数组。最下面还有一个数组count，里面有10个元素，代表10个指针，每个桶都有一个指针，比如3号桶的指针就是count[3]。

指针一开始指向桶中索引0的位置，比如3号桶的指针初始值为0，即

count[3]=0。

3号桶中每存一个元素，指针就右移一位，即count[3]++。

那么每个元素应该放在几号桶呢？

第一轮：

按照个位排序，比如23的个位是3，那么23就应该放在3号桶。

以此类推，将所有元素全部存放在这10个桶中，先存进去的元素，就先取出来。

然后从0号桶~9号桶，依次将元素取出，赋值给原数组arr。

注意，每个桶中的元素取完后，记得将每个桶的指针退回到索引0的位置，这里不需要将每个桶中的所有元素重置为0，因为我们取每个桶的元素时，是根据指针遍历的，而不是根据桶中存有的元素的数量遍历的，因此遍历的范围就是[0，指针指向的索引位置)。这样即使桶中还存有上一轮的元素，我们也可以不用管它，因为这些元素只会在指针所指向的位置，或者是指针右边，我们根本不会去取。

以此类推，当0~9号桶中的元素全部赋值给原数组arr后，第一轮就算完成。

同理，第二轮，根据十位排序。

第三轮，根据百位排序。

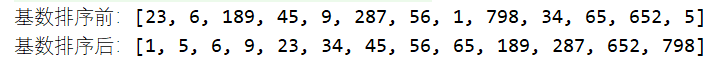
然后，排序完成！

##### 5.1.2代码实现

代码：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{23, 6, 189, 45, 9, 287, 56, 1, 798, 34, 65, 652, 5};  
 System.***out***.println(**"基数排序前："** + Arrays.*toString*(arr));  
 *radixSort*(arr);  
 System.***out***.println(**"基数排序后："** + Arrays.*toString*(arr));  
}  
  
//基数排序  
**public static void** radixSort(**int**[] arr){  
 //定义max，用于存储arr中最大的元素  
 **int** max = arr[0];  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 **if**(max <= arr[i]){  
 max = arr[i];  
 }  
 }  
 //for循环结束，就找出了arr中的最大值，并且已经赋值给了max  
 //max有几位，那么就要循环几轮  
 **int** maxLength = (max + **""**).length();  
 //定义一个临时存储的二维数组，第一个[10]代表temp中有10个数组，代表0~9号桶  
 //[arr.length]代表这10个数组中，每个数组的长度。因为最多也就将arr中所有的元素都放在一个桶里  
 **int**[][] temp = **new int**[10][arr.**length**];  
 //给每个桶都定义一个指针，这样一共需要10个指针。  
 //每个指针都用于标记放入桶中的元素的位置，也就是索引，便于存放和取出。  
 //count[0]，就代表0号桶的指针指向的位置，即索引  
 **int**[] count = **new int**[10];  
 //遍历maxLength轮  
 **for** (**int** i = 1, n = 1; i <= maxLength; i++, n = n\*10) {  
 //第i轮，遍历arr中的所有元素  
 **for** (**int** j = 0; j < arr.**length**; j++) {  
 **int** yushu = arr[j] / n % 10;  
 //余数是几，就将arr[j]放在几号桶里，那么arr[j]应该放在yushu号桶的什么位置呢？  
 //就根据yushu号桶的指针指向的位置来决定  
 //假设余数是2，一开始2号桶的指针指向索引0的位置，即count[2] = 0  
 //当2号桶存入第1个元素时，元素应该存放在索引0的位置，  
 //也就是存入前指针指向的位置，即count[2]=0的位置，然后指针右移1位，count[2]=1  
 //当2号桶存入第2个元素时，元素应该存放在索引1的位置，  
 //也就是存入前指针指向的位置，即count[2]=1的位置，然后指针右移1位，count[2]=2  
 //因此，arr[j]存入yushu号桶的索引位置，就是该桶指针指向的位置，即count[yushu]  
 temp[yushu][count[yushu]] =arr[j];  
 //yushu桶中每存入一个元素，yushu桶的指针都往右移1位  
 count[yushu]++;  
 }  
 //给arr数组定义一个指针index，指针指向的索引位置，  
 //就是将桶中的元素取出来时，要存入arr中的索引位置,指针一开始执行索引0的位置。  
 **int** index = 0;  
 //for循环结束，说明第i轮结束，已经将所有元素存放到了0~9号桶中  
 //将元素一次从0~9号桶中取出来  
 //遍历所有的桶,因为有几个桶就有几个指针，所以指针数量与桶的数量是一致的  
 **for** (**int** k = 0; k < count.**length**; k++) {  
 //遍历k号桶中的所有元素，那么k号桶里有多少个元素呢？k号桶的指针指向的位置是count[k]  
 //因为指针指向的位置是下一个元素要存入桶中的位置，也就是说指针所指的索引位置是空的，没有元素  
 //指针的左边才有元素，因此k号桶中元素的索引范围应该是[0,count[k])  
  
 **for** (**int** l = 0; l < count[k]; l++) {  
 //取出k号桶中的第l索引个元素，即temp[k][l],将其存入arr[]  
 arr[index] = temp[k][l];  
 //arr中每存一个元素，指针就右移一位  
 index++;  
 }  
  
 //k号桶的元素全部赋值到了arr中以后，必须将k号桶的指针退回到索引0的位置  
 //因为取的时候，我们是按照指针所在的位置来取的，这样即使指针所处的位置有元素，或者指针后面有元素  
 //我们都不会取，我们只会取指针左边的元素，所以只需要将指针退回到最左边索引0的位置，  
 //而不需要将k号桶的元素全部重置为0  
 count[k] = 0;  
 }  
 //for循环结束，那么第i轮中，就成功将元素从0~9号桶中取出来了  
 }  
 //这个for循环结束，说明这3轮就都结束了  
}

效果：



#### 5.2优化

基数排序之队列实现

主要就是对于桶的优化，因为这10桶中的元素都是，先存进去的，就先取出来，所以桶可以看作成是一个队列的结构。

之前想要存放数组时，是定义了一个二维数组temp，int[10][arr.length]。

每个桶都是一个一维数组，都是作为二维数组temp的元素，而数据则存放在桶中，也就是一维数组中。

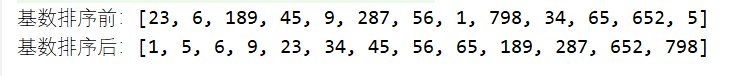
现在我们直接定义一个队列数组，来替换掉二维数组temp。每个桶就是一个队列，我们将元素存放到队列中。

MyQueue[ ] temp = new MyQueue[10]();

改造后的代码：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{23, 6, 189, 45, 9, 287, 56, 1, 798, 34, 65, 652, 5};  
 System.***out***.println(**"基数排序前："** + Arrays.*toString*(arr));  
 *radixSort*(arr);  
 System.***out***.println(**"基数排序后："** + Arrays.*toString*(arr));  
}  
  
//基数排序  
**public static void** radixSort(**int**[] arr){  
 //定义max，用于存储arr中最大的元素  
 **int** max = arr[0];  
 **for** (**int** i = 0; i < arr.**length**; i++) {  
 **if**(max <= arr[i]){  
 max = arr[i];  
 }  
 }  
 //for循环结束，就找出了arr中的最大值，并且已经赋值给了max  
 //max有几位，那么就要循环几轮  
 **int** maxLength = (max + **""**).length();  
 //定义一个队列数组，长度为10，表示里面有10个队列，用于存放数据。  
 //用队列的替换掉之前桶的概念，虽然干的事都是一样的，但是队列操作更简单，因为桶是数组，操作比较麻烦，存取数据都需要指针。  
 Myqueue[] temp = **new** Myqueue[10];  
 //注意，一定记得给temp中的给个元素，即给每个队列赋值，否则后面会报空指针异常。  
 //因为当数组中元素为引用类型时，初始值默认为null，即temp中一开始存了10个null。后面我们用null调用队列的各种方法时  
 //就会报空指针异常  
 //将temp中的每个队列进行初始化  
 **for** (**int** i = 0; i < temp.**length**; i++) {  
 temp[i] = **new** Myqueue();  
 }  
 //遍历maxLength轮  
 **for** (**int** i = 1, n = 1; i <= maxLength; i++, n = n\*10) {  
 //第i轮，遍历arr中的所有元素  
 **for** (**int** j = 0; j < arr.**length**; j++) {  
 **int** yushu = arr[j] / n % 10;  
 //余数是几，就将arr[j]放在几号队列里，我们不需要记录该元素存放的位置了  
 //也就是就不需要给队列定义指针了。（以前桶是数组结构的时候），我们担心如果不定义指针，那么  
 //就会导致每次存入或取出的元素都在数组的同一个索引位置。但是队列就不用担心了  
 //因为队列先进先出的特点，存入的元素，一定是存放在队列的末尾，即放在最右边  
 //而取元素的时候，一定是从队列的开头先取，即从队列最左边开始取。  
 temp[yushu].add(arr[j]);  
 }  
 //给arr数组定义一个指针index，指针指向的索引位置，  
 //就是将队列中的元素取出来时，要存入arr中的索引位置,指针一开始执行索引0的位置。  
 **int** index = 0;  
 //for循环结束，说明第i轮结束，已经将所有元素存放到了0~9号队列中  
 //将元素一次从0~9号桶中取出来  
 //遍历所有的队列，将所有队列中的数据都取出来  
 **for** (**int** k = 0; k < temp.**length**; k++) {  
 //将k号队列中的数据全部取出来，存入原数组arr中  
 **while** (!temp[k].isEmpty()){  
 arr[index] = temp[k].poll();  
 index++;  
 }  
 }  
 }  
}

效果：



显然，队列的思想更好理解，代码写起来也更简单。

但是前提必须要先有一个队列的基本类，这里我们用的是前面写的

Myqueue，代码参考：[2、队列类](#_2、队列类)

### 6、堆排序

#### 6.1概念

这部分是我学了完全二叉树的遍历之后，才开始学习的。

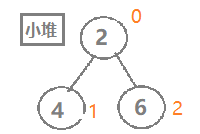
小堆排序：（这个概念是我自创的）

**小堆：**

就是针对最多只有3个节点的，呈三角形排序的二叉树。或者是2个节点，也可以是一个节点。

注意：

我定义的“小堆”数量最少一个，最最多3个，而且当数量为三个是，必须是按三角形排列的，如下图

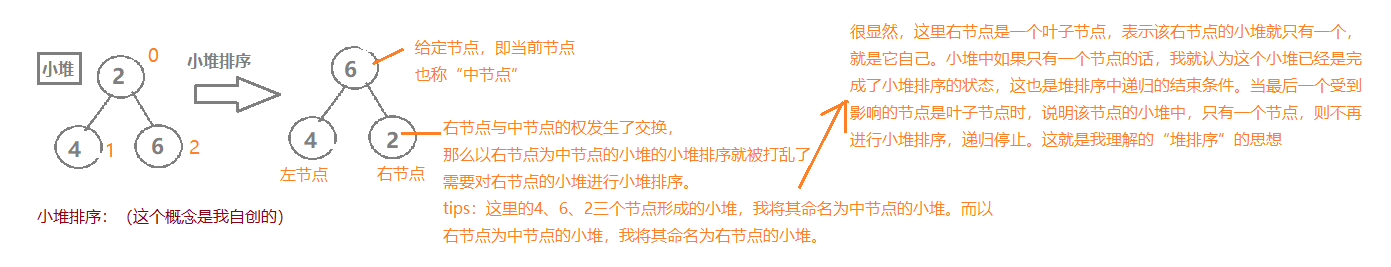


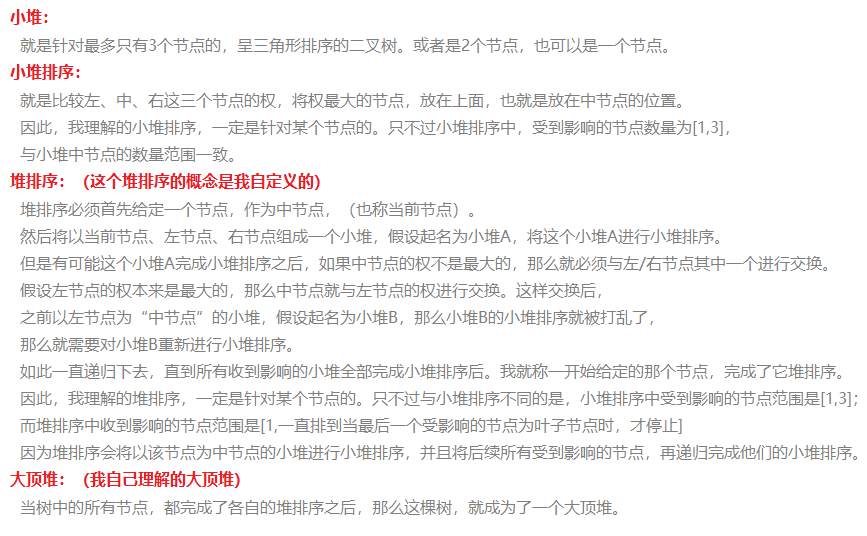
**小堆排序：**

就是比较左、中、右这三个节点的权，将权最大的节点，放在上面，也就是放在中节点的位置。

因此，我理解的小堆排序，一定是针对某个节点的。只不过小堆排序中，受到影响的节点数量为[1,3]，与小堆中节点的数量范围一致。

比如上图的小堆，完成小堆排序后的结果为：





**堆排序：（这个堆排序的概念是我自定义的）**

堆排序必须首先给定一个节点，作为中节点，（也称当前节点）。

然后将以当前节点、左节点、右节点组成一个小堆，假设起名为小堆A，将这个小堆A进行小堆排序。

但是有可能这个小堆A完成小堆排序之后，如果中节点的权不是最大的，那么就必须与左/右节点其中一个进行交换。

假设左节点的权本来是最大的，那么中节点就与左节点的权进行交换。这样交换后，

之前以左节点为“中节点”的小堆，假设起名为小堆B，那么小堆B的小堆排序就被打乱了，

那么就需要对小堆B重新进行小堆排序。

如此一直递归下去，直到所有收到影响的小堆全部完成小堆排序后。我就称一开始给定的那个节点，完成了它堆排序。

因此，我理解的堆排序，一定是针对某个节点的。只不过与小堆排序不同的是，小堆排序中受到影响的节点范围是[1,3]；

而堆排序中收到影响的节点范围是[1,一直排到当最后一个受影响的节点为叶子节点时，才停止]

因为堆排序会将以该节点为中节点的小堆进行小堆排序，并且将后续所有受到影响的节点，再递归完成他们的小堆排序。

**大顶堆：（我自己理解的大顶堆）**

当树中的所有节点，都完成了各自的堆排序之后，那么这棵树，就成为了一个大顶堆。

大顶堆完成后的效果就是：该树中的所有“中节点”的权，都比该“中节点”的左右子节点的权要大。

**大堆排序：**

第一轮：

在堆排序中，我们必须给定一个节点、数组、以及数组受到影响的元素的长度。

在大堆排序中，我们只需要给定一个数组，即可对该数组完成排序。

大堆排序的思路：

首先将树中的每一个节点都进行堆排序，让这棵树成为大顶堆。

当然了，叶子节默认是已经完成了堆排序的，就不用再进行堆排序，因此只需要从最后一个非叶子节点开始，进行堆排序。当最后一个非叶子节点完成堆排序之后，就对倒数第二个非叶子节点进行堆排序，依次类推，直到对第一个非叶子节点（也就是根节点）进行堆排序结束后。那么我就可以认为该树中所有的节点都已经完成了堆排序，我就认为这棵树已经成为了大顶堆。

此时大顶堆中，根节点的权肯定是最大的，我们就将根节点的权与最后一个节点的权进行交换，然后将最后一个节点从这棵树中“剔除”，代表我们找到了这棵树中最大的权，这样第一轮结束。

第二轮：

第一轮我们筛选出了原树中最大的权，第二轮我们需要筛选出新树中最大的权，就相当于是原树中第二大的权。

回顾我们是如何将原树变成新树的：

我们是将原树的第一个节点（根节点）与最后一个节点的权进行了交换，然后将最后一个节点从原树中“剔除”掉了，这样就得到了新树。

那么我们如何让新树成为新的大顶堆呢？

对比原树与新树，当原树成为大顶堆之后，我们只交换了第一个节点和最后一个节点，而其他节点我们是完全没去动的。

这说明，其他节点，仍然是已经完成了堆排序的状态，现在只有第一个节点（即为A节点）和最后一个节点（记为B节点）变动了。所以我们只需要再对A节点和B节点进行堆排序，就可以认为原树中所有的节点都完成了堆排序，这样就原树就又变成大顶堆了。

但是B节点是叶子节点，B节点的小堆中节点数量只有一个，就是它自己，所以我认为叶子节点就是已经出于小堆排序完成的状态，那么叶子节点肯定就也是出于堆排序完成的状态了。再加上原树还将B节点给剔除了。因此我们真正需要重新进行堆排序的节点，就只有一个，就是A节点。

也就是说，我们只需要对新树的根节点进行堆排序，那么新树就会成为一个大顶堆。

#### 6.2代码

代码：

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{9, 6, 8, 7, 0, 1, 10, 4, 2};  
 *heapSort*(arr);  
 System.***out***.println(**"排序后:"** + Arrays.*toString*(arr));  
}  
  
//排序  
**public static void** heapSort(**int**[] arr){  
 //最后一个节点的索引为arr.length-1  
 //利用公式，索引为n的节点的父节点的索引为(n-1)/2  
 //因为叶子节点我默认它就是堆排序完成的状态，所以就不用再重复进行堆排序了，  
 //所以只需要从倒数第一个非叶子节点开始进行堆排序，一直到根节点进行堆排序之后，  
 //就可以认为这棵树中的所有节点都完成了堆排序，那么这个树就是一个大顶堆  
 //定义开始索引位置:是最后一个非子叶节点，即最后一个节点的父节点  
 **int** start = (arr.**length** - 2) / 2;  
 //从倒数第一个节点开始，一直排到根节点，中间的所有节点都进行堆排序  
 //调整为大顶堆  
 **for**(**int** i = start; i >= 0; i--){  
 *maxHeap*(arr, arr.**length**, i);  
 }  
 //到此，该树中所有节点都已经出于堆排序完成的状态，因此现在这棵树就是一个大顶堆  
 //倒序遍历arr  
 **for**(**int** i = arr.**length**-1; i > 0; i--){  
 //将根节点的权，与i指向的节点的权arr[i]进行交换  
 //第一轮arr[i]是arr中的最后一个节点，即arr[arr.length-1]  
 //第二轮arr[i]是arr中的倒数第二个节点，即arr[arr.length-2],依次类推  
 **int** temp = arr[0];  
 arr[0] = arr[i];  
 arr[i] = temp;  
 //当根节点与i指向的节点的权交换后，根节点的堆排序被打乱，因此需要对根节点重新进行堆排序  
 //第二个参数为i，表示将最后一个节点“剔除”  
 *maxHeap*(arr, i, 0);  
 }  
}  
  
*/\*\*  
 \*  
 \** ***@param arr*** *进行堆排序的数组  
 \** ***@param size*** *数组中进行堆排序的长度，即数组中进行排序的元素的索引范围是[0,size)  
 \** ***@param index*** *当前节点的索引，指向进行堆排序的节点  
 \*  
 \* 方法功能描述：  
 \* 对数组arr中索引为index的节点进行堆排序。并且只对索引在[0,size)范围内的节点进行排序  
 \*/***public static void** maxHeap(**int**[] arr, **int** size, **int** index){  
 //当前节点的左子叶节点的索引  
 **int** leftNode = 2 \* index + 1;  
 //当前节点的右子叶节点的索引  
 **int** rightNode = 2 \* index + 2;  
 //定义一个指针max，指向(当前节点、左节点、右节点)中权最大的节点，同时也是该节点的索引  
 **int** max = index;  
 //对当前节点的小堆，进行小堆排序  
 //如果当前节点存在左右子节点  
 //将当前节点分别与左右子节点进行对比，找出最大节点，然后max指针就指向该节点  
 **if**(leftNode < size && arr[leftNode] > arr[max]){  
 max = leftNode;  
 }  
 **if** (rightNode < size && arr[rightNode] > arr[max]){  
 max = rightNode;  
 }  
 //现在指针max已经指向这三个中最大的节点了  
 **if**(max != index){  
 //如果指针max指向的不是当前节点，那么就让当前节点的权，与max指向的节点的权进行交换  
 **int** temp = arr[max];  
 arr[max] = arr[index];  
 arr[index] = temp;  
 //交换以后，虽然index指针和max指针没有移动，但是index指针所指节点的权已经是三个节点中最大的了  
 //这样，index指向的节点，也就是当前节点的小堆排序就完成了，但是max指向的节点的权发生了变化，  
 //说明max节点的小堆排序被打乱了，需要重新对max指向的节点进行小堆排序  
 *maxHeap*(arr, size, max);  
 }  
}

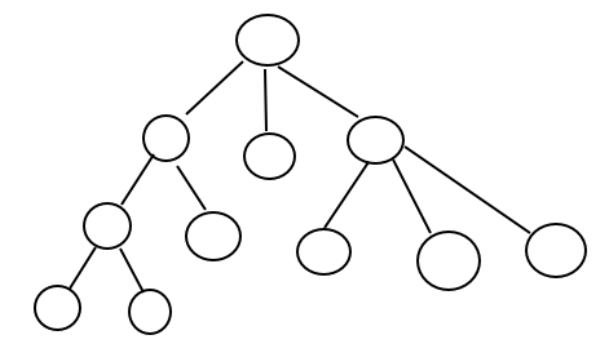
效果：



# 第4章 树结构

## 一、树结构概述

树结构图：



为什么要使用树结构：

前面的数据结构中，学了很多种，但是最底层的只有2种：数组和链表，但是我们都知道他们各有优劣。

数组：有索引，查找快，增删慢

链表：无索引，查找慢，增删快。

那么能不能引入一个新的结构，能够实现查找快、增删快呢？

树的基本概念



概念：

双亲节点、子节点、路径、节点的度、节点的权、叶子节点、子树、层、

树的高度、森林。

## 二、二叉树

### 1、什么是二叉树

#### 1.1二叉树

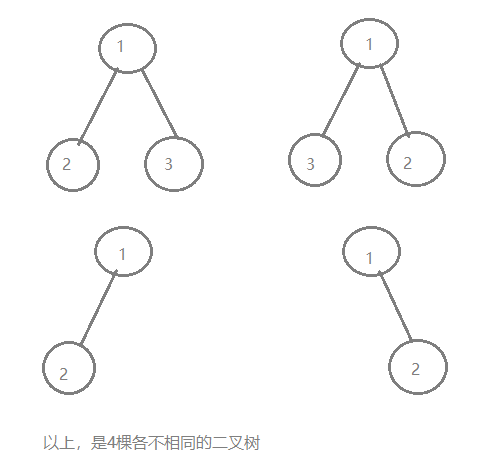
二叉树：

一棵树中，如果任何一个节点的子节点个数都不超过2，那么这棵树就叫二叉树。

二叉树的子节点分为：左节点和有节点。

理解：

下图是4个不同的二叉树



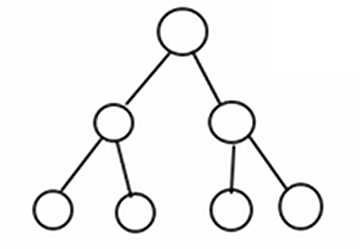
#### 1.2满二叉树

满二叉树：

所有叶子节点都在最后一层，并且节点的总数为2n-1。n是树的高度。

图解：

下面就是一棵满二叉树



#### 1.3完全二叉树

完全二叉树：

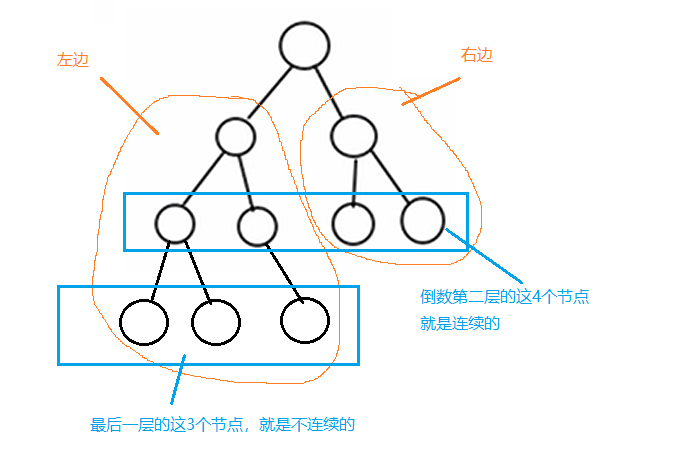
所有叶子节点都在最后一层或倒数第二层，并且最后一层的叶子节点在左边连续，倒数第二节的叶子节点在右边连续。

这里要理解几个概念：

左边和右边、连续

图解：

下图这个，就不是完全二叉树。

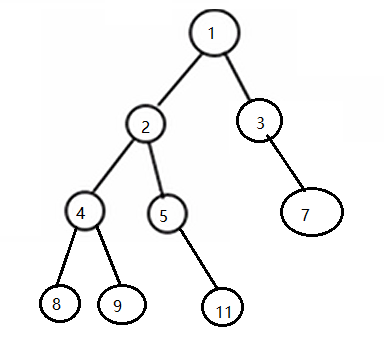


还有一个判断是否为完全二叉树的方法：

假设我们给要判断的数的每个节点赋上数字，规则是从上至下，从左至右，从1开始赋值。

当所有节点全部赋值后，如果数字是连续的，那么这棵树就是完全二叉树。

例如：



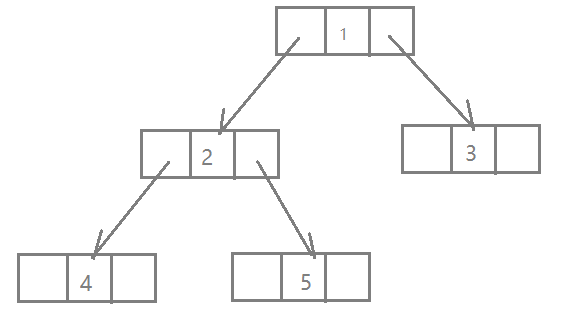
可以看到，数字不是连续的，漏了6和10，说明这棵树不是完全二叉树。

显然这个方法更容易判断是不是完全二叉树。

### 2、创建二叉树

#### 2.1概述

数的模型构建思路：



#### 2.2树

BinaryTree：

*/\*\*  
 \* 二叉树  
 \*/*@Data  
**public class** BinaryTree {  
 //根节点  
 **private** TreeNode **root**;  
}

#### 2.3节点

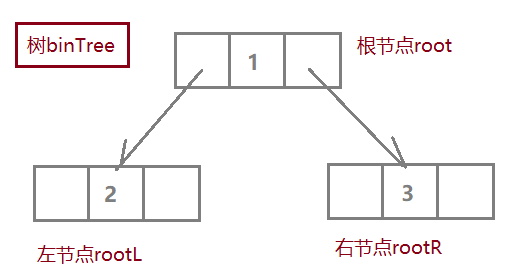
TreeNode：

*/\*\*  
 \* 节点  
 \*/*@Data  
**public class** TreeNode {  
 //节点的权  
 **private int value**;  
 //左儿子  
 **private** TreeNode **leftNode**;  
 //右儿子  
 **private** TreeNode **rightNode**;  
  
 **public** TreeNode(**int** value) {  
 **this**.**value** = value;  
 }  
}

#### 2.4测试类

*/\*\*  
 \* 二叉树测试类  
 \*/***public class** BinaryTreeTest {  
 **public static void** main(String[] args) {  
 //创建一棵树  
 BinaryTree binTree = **new** BinaryTree();  
 //创建一个节点，作为根节点  
 TreeNode root = **new** TreeNode(1);  
 //为树的成员root（根节点）设置值  
 binTree.setRoot(root);  
 //创建一个左节点  
 TreeNode rootL = **new** TreeNode(2);  
 //创建一个右节点  
 TreeNode rootR = **new** TreeNode(3);  
 //让树的根节点root指向左节点  
 root.setLeftNode(rootL);  
 //让树的根节点root指向右节点  
 root.setRightNode(rootR);  
 }  
}

到此为止，测试类中已经完成的模型如下：



### 3、遍历二叉树

#### 3.1概述

树的遍历顺序有3种：

前序遍历、中序遍历、后续遍历。

这里要用到递归的思想。

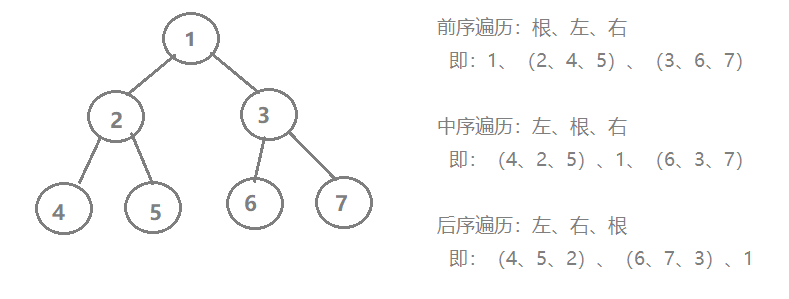
先从最简单的开始

假设树上只有3个节点：根节点、左节点，右节点。简记为（根、左、右）。



* 前序遍历：中、左、右。即1、2、3
* 中序遍历：左、中、右。即2、1、3
* 后序遍历：左、右、中。即2、3、1

稍微复杂一点的，我们给上面的数加一层：



Tips：

其实上面的笔记不太好，应该将“根”，改成“中”。这样比较恰当。

因为一个二叉树中，可以有很多个“中”，但是“根”永远只有一个，就是最顶上那个节点。

#### 3.2树

增加一个方法：frontShow（）

*/\*\*  
 \* 二叉树  
 \*/*@Data  
**public class** BinaryTree {  
 //根节点  
 **private** TreeNode **root**;  
  
 //前序遍历  
 **public void** frontShow(){  
 **root**.frontShow();  
 }  
}

#### 3.3节点

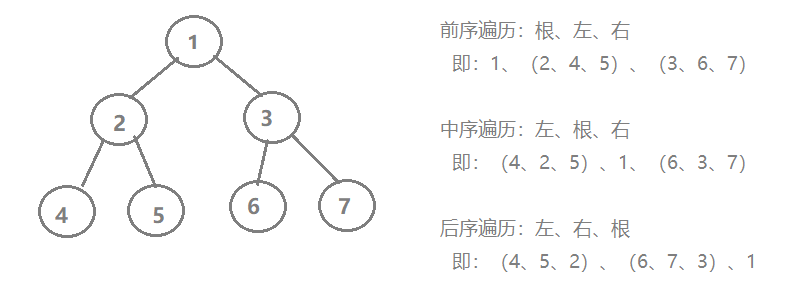
前序遍历：

//前序遍历：根、左、右  
**public void** frontShow() {  
 //打印根节点的权  
 System.***out***.print(**value** + **" "**);  
 //左节点存在时，左节点也调用前序遍历  
 **if**(**leftNode** != **null**){  
 **leftNode**.frontShow();  
 }  
 //右节点存在时，右节点也调用前序遍历  
 **if**(**rightNode** != **null**){  
 **rightNode**.frontShow();  
 }  
 //当左右节点为null时，就不再调用前序遍历了，递归停止。  
}

#### 3.4测试类

**public static void** main(String[] args) {  
 //创建一棵树  
 BinaryTree binTree = **new** BinaryTree();  
 //创建一个节点，作为根节点  
 TreeNode n1 = **new** TreeNode(1);  
 //为树的成员root（根节点）设置值  
 binTree.setRoot(n1);  
 //创建一个左节点  
 TreeNode n2 = **new** TreeNode(2);  
 //创建一个右节点  
 TreeNode n3 = **new** TreeNode(3);  
 //让n1的左儿子指向n2  
 n1.setLeftNode(n2);  
 //让n1的右儿子指向n3  
 n1.setRightNode(n3);  
 //设置第三层的4、5、6、7节点  
 TreeNode n4 = **new** TreeNode(4);  
 TreeNode n5 = **new** TreeNode(5);  
 TreeNode n6 = **new** TreeNode(6);  
 TreeNode n7 = **new** TreeNode(7);  
 //让n2的左儿子指向n4  
 n2.setLeftNode(n4);  
 //让n2右儿子指向n4  
 n2.setRightNode(n5);  
 //让n3的左儿子指向n6  
 n3.setLeftNode(n6);  
 //让n3的右儿子指向n7  
 n3.setRightNode(n7);  
 //前序遍历  
 binTree.frontShow();  
}

模型：



执行结果：



中序遍历和后序遍历的原理，与前序遍历类似，只是修改一下顺序而已，就不再叙述了。

### 4、二叉树中节点的查找

#### 4.1概述

二叉树的遍历有三种顺序。

二叉树中节点的查找也有三种顺序。

这里就只介绍其中一种：前序查找。

思路：

将二叉树进行遍历，获取每个节点的权，然后将传入的参数与节点的权进行比较，当二者相等时，就返回该节点。

即：给我一个权，就能获取该权对应的节点的对象。

#### 4.2树

增加一个方法：前序查找

@Data  
**public class** BinaryTree {  
 //根节点  
 **private** TreeNode **root**;  
  
 //前序遍历  
 **public void** frontShow(){  
 **root**.frontShow();  
 }  
 //前序查找  
 **public** TreeNode frontSearch(**int** i){  
 **return root**.frontSearch(i);  
 }  
}

#### 4.3节点

这里仅展示前序查找方法、以及重写后的toString方法：

TreeNode：

//前序查找  
**public** TreeNode frontSearch(**int** i){  
 TreeNode target = **null**;  
 //先比较根节点的权  
 **if**(**this**.**value** == i){  
 //当根节点查到了时，直接返回  
 **return this**;  
 }**else** {  
 //再比较左节点  
 **if**(**leftNode** != **null**){  
 target = **leftNode**.frontSearch(i);  
 }  
 **if** (target != **null**){  
 //当左边查到了时，直接返回  
 **return** target;  
 }  
 //最后比较右节点  
 **if**(**rightNode** != **null**){  
 target = **rightNode**.frontSearch(i);  
 }  
 }  
 //返回右节点的查找结果，如果每查到就返回null  
 **return** target;  
}  
  
@Override  
**public** String toString() {  
 **return this**.**value** + **""**;  
}

#### 4.4测试

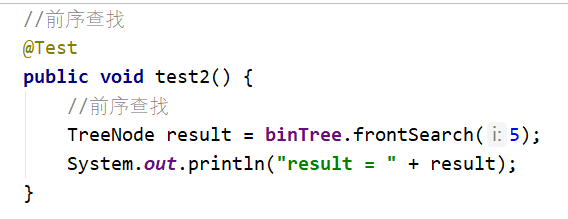
关键看test2方法的结果就行了：

**public class** BinaryTreeTest {  
 //定义成员binTree，作为树  
 **private** BinaryTree **binTree**;  
  
 //前置方法，构建树的模型  
 @Before  
 **public void** setUp() **throws** Exception {  
 //创建一棵树  
 **binTree** = **new** BinaryTree();  
 //创建一个节点，作为根节点  
 TreeNode n1 = **new** TreeNode(1);  
 //为树的成员root（根节点）设置值  
 **binTree**.setRoot(n1);  
 //创建一个左节点  
 TreeNode n2 = **new** TreeNode(2);  
 //创建一个右节点  
 TreeNode n3 = **new** TreeNode(3);  
 //让n1的左儿子指向n2  
 n1.setLeftNode(n2);  
 //让n1的右儿子指向n3  
 n1.setRightNode(n3);  
 //设置第三层的4、5、6、7节点  
 TreeNode n4 = **new** TreeNode(4);  
 TreeNode n5 = **new** TreeNode(5);  
 TreeNode n6 = **new** TreeNode(6);  
 TreeNode n7 = **new** TreeNode(7);  
 //让n2的左儿子指向n4  
 n2.setLeftNode(n4);  
 //让n2右儿子指向n4  
 n2.setRightNode(n5);  
 //让n3的左儿子指向n6  
 n3.setLeftNode(n6);  
 //让n3的右儿子指向n7  
 n3.setRightNode(n7);  
 }  
  
 //前序遍历  
 @Test  
 **public void** test1() {  
 //前序遍历  
 **binTree**.frontShow();  
 }  
  
 //前序查找  
 @Test  
 **public void** test2() {  
 //前序查找  
 TreeNode result = **binTree**.frontSearch(10);  
 System.***out***.println(**"result = "** + result);  
 }  
}

结果：



当查找的是5时：

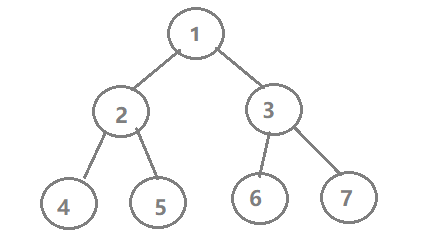




注意一个细节：

为什么每次我们调用方法时，不论是遍历还是查找，都是用树的对象来调用呢？这根直接用节点对象调用有什么区别吗？

有区别，很关键。看下图：



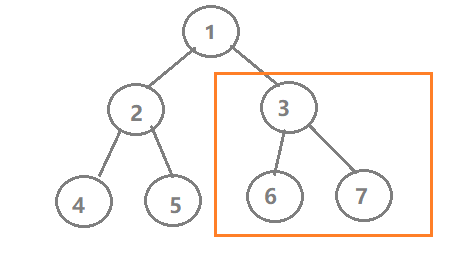
比如说，如果我们用节点3来调用遍历或者是查找的方法，那么我们遍历或者查找的范围就只是：3、6、7而已。

但是我们用树对象来调用就不一样了，因为树对象中所有方法，都是树对象的成员变量root来调用的，因此，我们用树对象来调用时，就相当于在用根节点1来调用方法。那么我们遍历或者查找的范围就是整棵树了。

这就是区别。

### 5、删除子树

#### 5.1概述



比如我们要删除上图中的3、6、7。

我们只需要知道节点3，就能将其整个删除。

这回树的代码就比较重要了

#### 5.2树

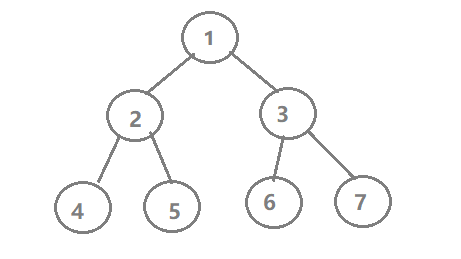
//删除子树:即删除当前节点i，以及所有子孙后代节点  
**public void** delete(**int** i){  
 **if**(**root**.getValue()== i){  
 //当给的节点就是根节点时，直接让根节点为null  
 //就代表将根节点自己及其所有子孙后代都给删除了  
 **root** = **null**;  
 }**else** {  
 //当要删除的不是根节点时，调用TreeNode的delete方法，删除这个子树  
 //注意可不是BinaTree的delete方法哦，  
 //即这里树的delete方法中是没有递归的，真正的递归要在节点的delete方法中  
 **root**.delete(i);  
 }  
}

#### 5.3节点

//在当前节点的子孙后代中查找，删除权为i的节点，及其子孙节点  
//注意，他不会判断当前节点的权是不是i  
**public void** delete(**int** i){  
 TreeNode parent = **this**;  
 //判断左儿子  
 **if**(parent.**leftNode** != **null** && parent.**leftNode**.**value** == i){  
 parent.**leftNode** = **null**;  
 }  
 //判断右儿子  
 **if** (parent.**rightNode** != **null** && parent.**rightNode**.**value** == i){  
 parent.**rightNode** = **null**;  
 }  
 //递归检查并删除左儿子  
 parent = **leftNode**;  
 **if** (parent != **null**){  
 parent.delete(i);  
 }  
  
 //递归检查并删除右儿子  
 parent = **rightNode**;  
 **if**(**rightNode** != **null**){  
 parent.delete(i);  
 }  
  
}

#### 5.4测试

树模型：



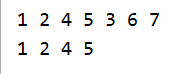
##### 5.4.1测试1

当树调用树的delete方法删除时，如果传入3，那么会将3、6、7节点全部删除

代码：

//删除节点  
@Test  
**public void** test3() {  
 **binTree**.frontShow();  
 **binTree**.delete(3);  
 System.***out***.println();  
 **binTree**.frontShow();  
}

效果：



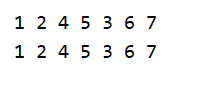
##### 5.4.2测试2

如果是节点3调用节点的delete方法，传入3时，不会删除，因为节点的delete方法不会判断自己的权，只会去判断其子孙后代的权

代码：

//删除节点  
@Test  
**public void** test3() {  
 **binTree**.frontShow();  
 **n3**.delete(3);  
 System.***out***.println();  
 **binTree**.frontShow();  
}

效果：



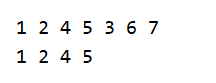
##### 5.4.3测试3

当节点1调用节点的delete方法，传入3时，会将3、6、7删除

代码：

//删除节点  
@Test  
**public void** test3() {  
 **binTree**.frontShow();  
 **n1**.delete(3);  
 System.***out***.println();  
 **binTree**.frontShow();  
}

效果：



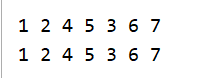
##### 5.4.4测试4

当节点1调用节点的delete方法，传入1时，不会将树删除

代码：

//删除节点  
@Test  
**public void** test3() {  
 **binTree**.frontShow();  
 **n1**.delete(1);  
 System.***out***.println();  
 **binTree**.frontShow();  
}

效果：



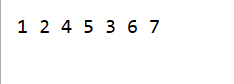
##### 5.4.5测试5

当树调用树的delete方法，传入1时，会将整个树删除

代码：

//删除节点  
@Test  
**public void** test3() {  
 **binTree**.frontShow();  
 **binTree**.delete(1);  
 System.***out***.println();  
 **binTree**.frontShow();  
}

效果：



小结：

搞了这么多测试，就只为了说明一个问题：

就是节点的delete方法和树的delete方法是不一样的，节点的delete方法，是不会判断节点自己的权的，只会判断当前节点的所有子孙后代的权。

但是树的delete方法，就会进行判断。这样才能更好的理解递归。

这么一看，我反而觉得老师写的方法不好，我们应该将全部的判断逻辑都写在节点方法当中，也就是将对当前节点自己的权的判断逻辑，也写到节点的delete方法中去，让树的方法中，只负责调用节点的方法就行了。这样处理显然更好，而且也会更容易理解。

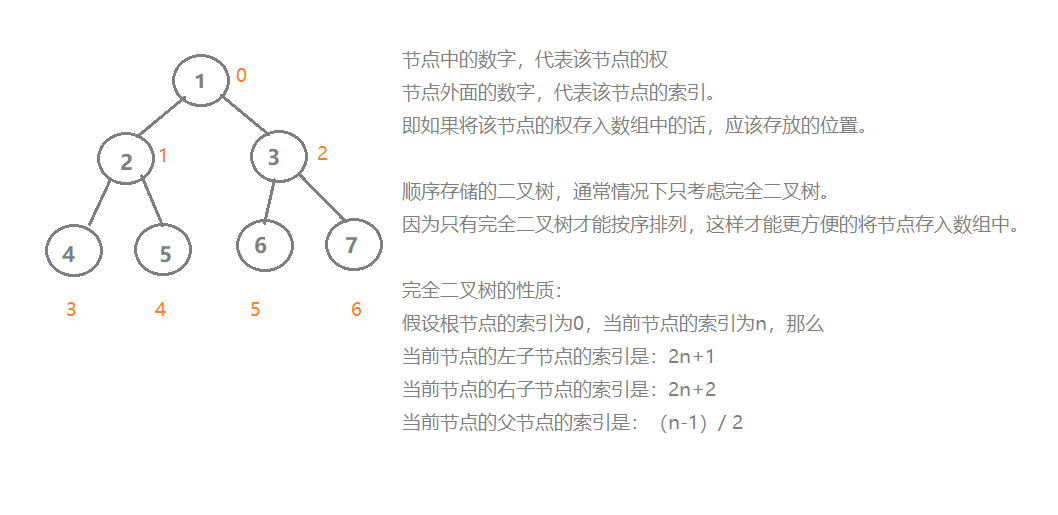
### 6、顺序存储的二叉树

#### 6.1简介

前面所学的是链式存储的二叉树，与链表有相似之处。

现在我们来学习顺序存储的二叉树，与数组进行结合。

图解：



节点中的数字，代表该节点的权

节点外面的数字，代表该节点的索引。

即如果将该节点的权存入数组中的话，应该存放的位置。

顺序存储的二叉树，通常情况下只考虑完全二叉树。

因为只有完全二叉树才能按序排列，这样才能更方便的将节点存入数组中。

完全二叉树的性质：

假设根节点的索引为0，当前节点的索引为n，那么

当前节点的左子节点的索引是：2n+1

当前节点的右子节点的索引是：2n+2

当前节点的父节点的索引是：（n-1）/ 2

#### 6.2基本类

因为完全二叉树的节点是可以按序排列的，因此，我们可以将数组，看做成是一个完全二叉树。

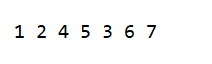
代码：

@Data  
**public class** ArrayBinaryTree {  
 //定义成员data，可以将data看作成是一棵完全二叉树  
 **private int**[] **data**;  
  
 **public** ArrayBinaryTree(**int**[] data) {  
 **this**.**data** = data;  
 }  
  
 //遍历索引为0的节点（根节点），及其子树  
 //也就是遍历整个树，即遍历整个data数组。  
 **public void** frontShow(){  
 frontShow(0);  
 }  
  
 //遍历索引为index的节点，及其子树  
 //直接将index索引对应的节点，看作是当前节点  
 **public void** frontShow(**int** index){  
 **if**(**data** == **null** || **data**.**length** == 0){  
 **return**;  
 }  
 //打印当前节点，即索引为index的节点  
 System.***out***.print(**data**[index] + **" "**);  
 //当前节点的左子节点的索引为2\*index+1  
 **if** (2\*index + 1 < **data**.**length**){  
 //当左子节点存在时，递归打印  
 frontShow(2\*index + 1);  
 }  
 //当前节点的右子节点的索引为2\*index + 2  
 **if** (2\*index + 2 < **data**.**length**){  
 //当右子节点存在时，递归打印  
 frontShow(2\*index + 2);  
 }  
 }  
}

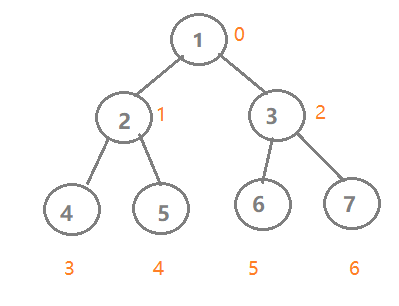
#### 6.3测试类

//测试前序遍历  
@Test  
**public void** test1() {  
 **int**[] arr = **new int**[]{1, 2, 3, 4, 5, 6, 7};  
 ArrayBinaryTree tree = **new** ArrayBinaryTree(arr);  
 tree.frontShow();  
}

效果：



结合模型图：



外面的是索引，从0~6。里面的权就是数组中的元素。

使用的是二叉树的前序遍历：中、左、右

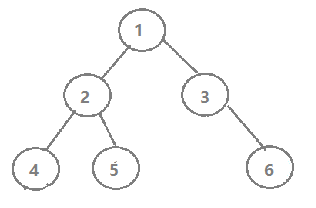
测试是没问题的。

## 三、线索二叉树

这块只听了个大概，感觉没有完全掌握，不过暂时先这样吧。

### 1、概述

问题引入：



中序遍历：左、中、右

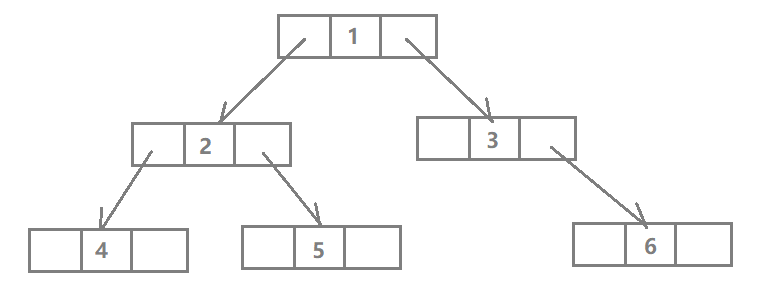
结果：（4、2、5）、1、（3、6）

问题：

在中序遍历的时候，如果我们希望知道节点2的前一个被遍历的节点是什么，我们无法得知。

因为每次遍历完一个节点之后，信息是没有被保留的，直接就去遍历下一个节点了。因此我们无法得知当前被遍历的节点的前一个节点是什么。

此外，我们也无法得知下一个将要被遍历的节点是什么。（这点我不是很懂），我自己理解的就是，我们只能看到当前被遍历的节点信息，其他的都看不到。

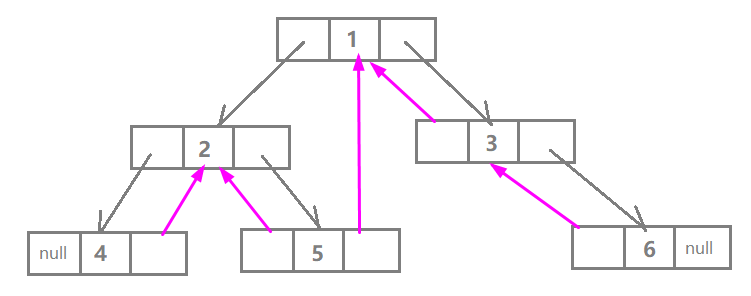


可以发现，所有叶子节点（4、5、6）的左右子节点都是null

而且节点3的左子节点也是null

与其像这样浪费格子，对于节点中，如果有左右子节点为null的话，我们就让该节点的左子节点指向当前节点的前一个节点，让该节点的右子节点指向当前节点的后一个节点：

中序线索化模型图：



线索化二叉树时：

* 当前节点的前一个节点：叫当前节点的前驱节点
* 当前节点的后一个节点：叫当前节点的后继节点

但是这样又会存在一个问题：

就是我们怎么才能知道，当前节点左边的格子，代表的到底是当前节点的左子节点，还是前驱节点呢？同理，右边也是一样，无法判断是右子节点还是后继节点。

因此肯定需要设置一个标记，因为不论是左边的格子，还是右边的各自，同时肯定只能代表一个含义。比如左边的格子，代表的含义要么是左子节点，要么就是上一个节点，当然了，要么也可能是null。总之只能是其中一个，不可能同时代表多个含义的。

### 2、实现

#### 2.1树

ThreadedBinaryTree：

*/\*\*  
 \* 二叉树\_\_线索化  
 \*/*@Data  
**public class** ThreadedBinaryTree {  
 //根节点  
 **private** ThreadedNode **root**;  
 //临时存储前驱节点  
 **private** ThreadedNode **pre**;  
  
 //中序线索化二叉树，真正使用的是这个方法  
 **public void** threadNodes(){  
 threadNodes(**root**);  
 }  
  
 //中序线索化二叉树(仅内部使用)  
 **private void** threadNodes(ThreadedNode node) {  
 //理解下面叙述的“当前节点”  
 //每次调用该方法时，所传入的参数，  
 //就认为是这一次调用该方法的当前节点  
  
 //如果当前节点为null，直接返回，递归结束  
 **if**(node == **null**){  
 **return**;  
 }  
 //处理当前节点的左子树  
 threadNodes(node.getLeftNode());  
  
 //如果当前节点的左边为null  
 **if**(node.getLeftNode() == **null**){  
 //那么就让当前节点的左边指向当前节点的前驱节点pre  
 node.setLeftNode(**pre**);  
 //然后将当前节点的左边标记改为1，  
 //表示当前节点的左边指向的的是“前驱节点”，而不是“左子节点”  
 node.setLeftType(1);  
 }  
 //如果前驱节点不为null，并且前驱节点的右边为null,  
 **if** (**pre** != **null** && **pre**.getRightNode() == **null**){  
 //那么就让前驱节点的右边指向前驱节点的后继节点，即指向当前节点  
 **pre**.setRightNode(node);  
 //然后将前驱节点的右边标记改为1  
 //表示前驱节点的右边指向的是“后继节点”，而不是前驱节点的“右子节点”  
 **pre**.setRightType(1);  
 }  
  
 //指针pre指向当前节点，那么对于这句代码上面的所有代码来说  
 //pre就完全可以理解为当前节点的前驱节点（这一步是我认为最难理解的）  
 **pre** = node;  
  
 //处理当前节点的右子树  
 threadNodes(node.getRightNode());  
 }  
  
 //中序遍历  
 **public void** midShow(){  
 **root**.midShow();  
 }

}

#### 2.2节点

ThreadedNode

*/\*\*  
 \* 线索节点  
 \*/*@Data  
**public class** ThreadedNode {  
 //节点的权  
 **private int value**;  
 //左儿子  
 **private** ThreadedNode **leftNode**;  
 //右儿子  
 **private** ThreadedNode **rightNode**;  
 //标识指针的类型，int类型的成员变量，初始值默认为0  
 **private int leftType**;  
 **private int rightType**;  
  
 **public** ThreadedNode(**int** value) {  
 **this**.**value** = value;  
 }

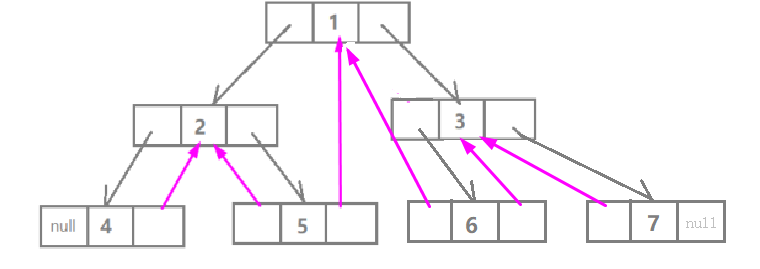
//中序遍历：左、中、右  
 **public void** midShow(){  
 //左节点存在时，左节点也调用前序遍历  
 **if**(**leftNode** != **null**){  
 **leftNode**.midShow();  
 }  
 //打印中节点的权  
 System.***out***.print(**value** + **" "**);  
  
 //右节点存在时，右节点也调用前序遍历  
 **if**(**rightNode** != **null**){  
 **rightNode**.midShow();  
 }  
 //当左右节点为null时，就不再调用前序遍历了，递归停止。  
 }

}

#### 2.3测试类

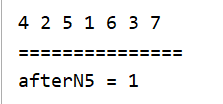
**public class** ThreadedBinaryTreeTest {  
 //定义成员binTree，作为树  
 **private** ThreadedBinaryTree **binTree**;  
 **private** ThreadedNode **n5**;  
  
  
 //前置方法，构建树的模型  
 @Before  
 **public void** setUp() **throws** Exception {  
 //创建一棵树  
 **binTree** = **new** ThreadedBinaryTree();  
 //创建一个节点，作为根节点  
 ThreadedNode n1 = **new** ThreadedNode(1);  
 //为树的成员root（根节点）设置值  
 **binTree**.setRoot(n1);  
 //创建一个左节点  
 ThreadedNode n2 = **new** ThreadedNode(2);  
 //创建一个右节点  
 ThreadedNode n3 = **new** ThreadedNode(3);  
 //让n1的左儿子指向n2  
 n1.setLeftNode(n2);  
 //让n1的右儿子指向n3  
 n1.setRightNode(n3);  
 //设置第三层的4、5、6、7节点  
 ThreadedNode n4 = **new** ThreadedNode(4);  
 **n5** = **new** ThreadedNode(5);  
 ThreadedNode n6 = **new** ThreadedNode(6);  
 ThreadedNode n7 = **new** ThreadedNode(7);  
 //让n2的左儿子指向n4  
 n2.setLeftNode(n4);  
 //让n2右儿子指向n5  
 n2.setRightNode(**n5**);  
 //让n3的左儿子指向n6  
 n3.setLeftNode(n6);  
 //让n3的右儿子指向n7  
 n3.setRightNode(n7);  
 }  
  
 //中序线索化二叉树  
 @Test  
 **public void** test2() {  
 //中序遍历二叉树  
 **binTree**.midShow();  
 System.***out***.println();  
 System.***out***.println(**"==============="**);  
 //中序线索化二叉树  
 **binTree**.threadNodes();  
 //获取n5节点的后继节点  
 ThreadedNode afterN5 = **n5**.getRightNode();  
 System.***out***.println(**"afterN5 = "** + afterN5.getValue());  
 }  
  
}

模型结构：



中序遍历：（4、2、5）、1、（6、3、7）

测试结果：



这里线索化之后，就不能使用直接调用以前的方法进行查找了

### 3、遍历

暂时了解即可吧，虽然代码每一步干了什么事都看得懂，但是无法理解为什么这样竟然可以实现遍历，真是无法理解。

#### 3.1方法

方法写在树中，而不是节点中

代码：

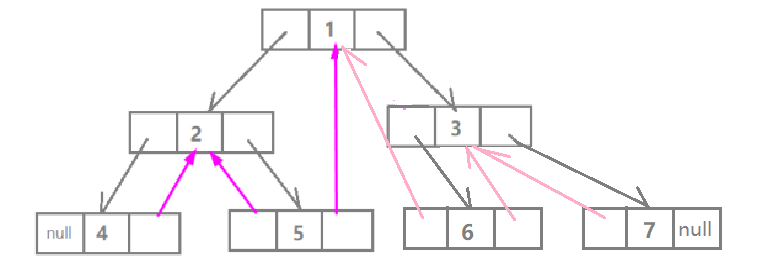
//中序遍历线索二叉树  
**public void** threadIterate(){  
 //定义一个指针node，一开始指向根节点  
 ThreadedNode node = **root**;  
 //只要指针所指向的节点不为null，那么就进入while循环  
 **while** (node != **null**){  
 //指针node所指向的节点，为描述方便，我在后面就直接简记为指针node了  
 //当指针node的左边类型为0时，代表指针的左边是左子节点，而不是前驱节点  
 **while** (node.getLeftType() == 0){  
 //那么就让指针node移向这个左子节点  
 node = node.getLeftNode();  
 }  
 //while结束后，此时指针node已经移动到了树的“最左下角”  
 //我们打印这个最左下角节点的权  
 System.***out***.print(node.getValue() + **" "**);  
 //当指针node的右边的类型是1时，  
 //代表指针的右边是后继节点，而不是右子节点，那么就进入while循环  
 **while** (node.getRightType() == 1){  
 //让指针移向右边的后继节点  
 node = node.getRightNode();  
 //打印指针node的权，也就是移动前，指针node的后继节点的权  
 System.***out***.print(node.getValue() + **" "**);  
 }  
 //while循环结束，那么此时指针node所指向的节点的右边肯定不是后继节点  
 //要么是null，要么就是右子节点  
 //让指针移向右边的节点  
 node = node.getRightNode();  
 }  
}

#### 3.2测试

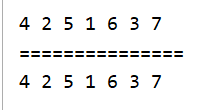
代码：

//遍历线索二叉树  
@Test  
**public void** test3() {  
 //中序遍历二叉树  
 **binTree**.midShow();  
 System.***out***.println(**"\r\n"** + **"==============="**);  
 //中序线索化二叉树  
 **binTree**.threadNodes();  
 //遍历线索二叉树  
 **binTree**.threadIterate();  
}

模型图：



效果：



很奇怪，边看代码，边看模型走了一遍，发现确实能够遍历，但是就是不知道怎么想到这个遍历思路的。

暂时先不仔细深究了。

## 四、赫夫曼树

### 1、赫夫曼树概述

1.1最优二叉树：

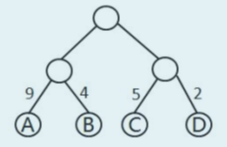
它是n个带权叶子结点构成的所有二叉树中，带权路径长度最小的二叉树。最优二叉树，我们也称之为赫夫曼树。

1.2叶(子)节点的带权路径：

带权路径 = 路径长度 × 叶节点的权

路径长度：从根节点一直到叶节点，中间经过了几段，那么该叶子节点到根节点的路径长度就是几。

如图：



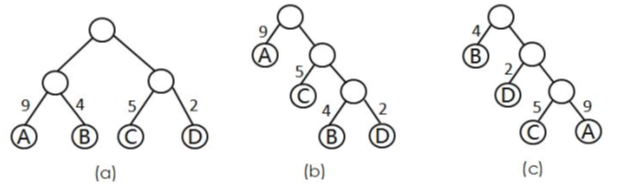
上面的二叉树中，有A、B、C、D这4个叶节点，他们的权分别是9、4、5、2

其中根节点到A节点的路径长度是2，则

A节点的带权路径为：2 × 9 = 18

1.3树的带权路径长度：

WPL(weighted path length)：树中所有叶子结点的带权路径长度之和。



（a）WPL=9x2+4x2+5x2+2x2=18+8+10+4=40

（b）WPL=9x1+5x2+4x3+2x3=9+10+12+6=37

（c）WPL=4x1+2x2+5x3+9x3=4+4+15+27=50

上面三棵树中的最优二叉树是(b)这棵树。

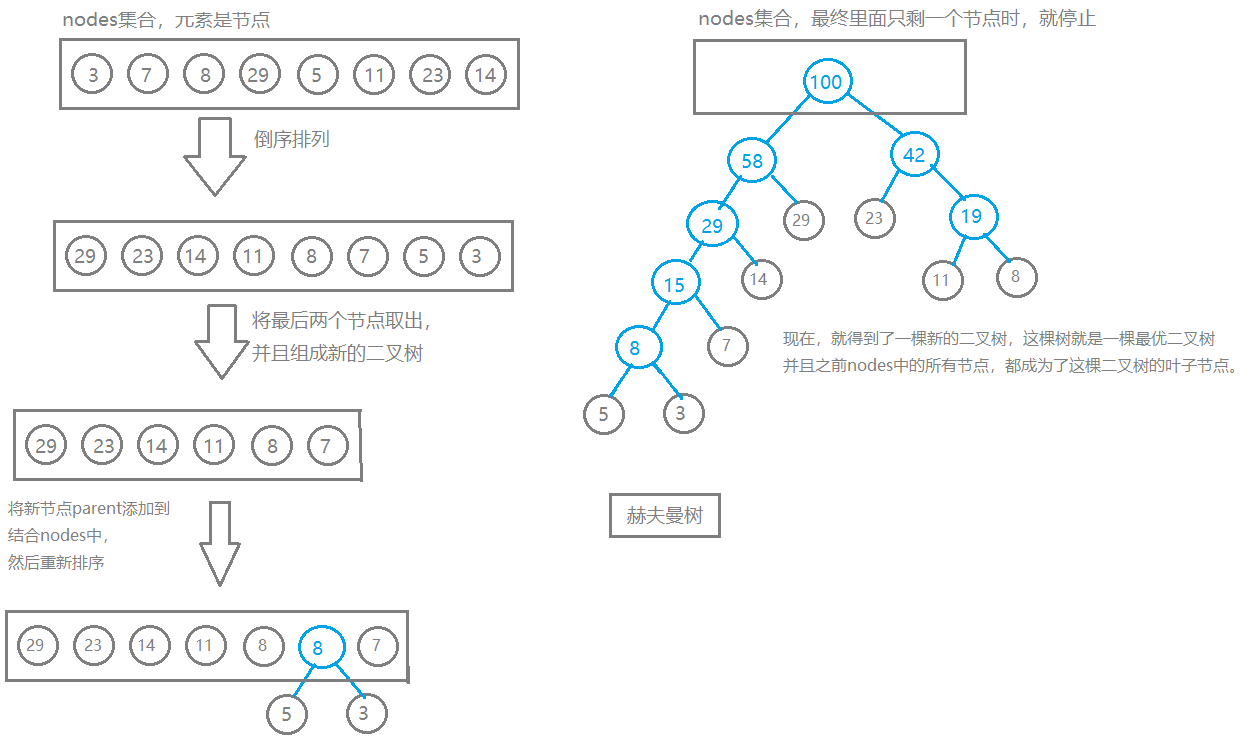
可以看出，如果希望树的带权路径小一些的话，那我们应该让权越大的叶子节点，离根节点越近。即当叶子节点的权没法改变时，我们可以改变路径长度。

### 2、代码实现

代码实现赫夫曼树

#### 2.1思路

图解：



#### 2.2节点

Node：

*/\*\*  
 \* 节点  
 \*/*@Data  
**public class** Node **implements** Comparable{  
 **private int value**;  
 **private** Node **left**;  
 **private** Node **right**;  
  
 **public** Node(**int** value) {  
 **this**.**value** = value;  
 }  
  
  
 @Override  
 **public int** compareTo(Object o) {  
 Node node = (Node) o;  
 //将集合中的Node元素，按照value的大小，倒序排列，即从大到小排列  
 **return** node.getValue() - **this**.getValue();  
 }  
 @Override  
 **public** String toString() {  
 **return this**.getValue() + **" "**;  
 }  
}

#### 2.3测试类

代码：

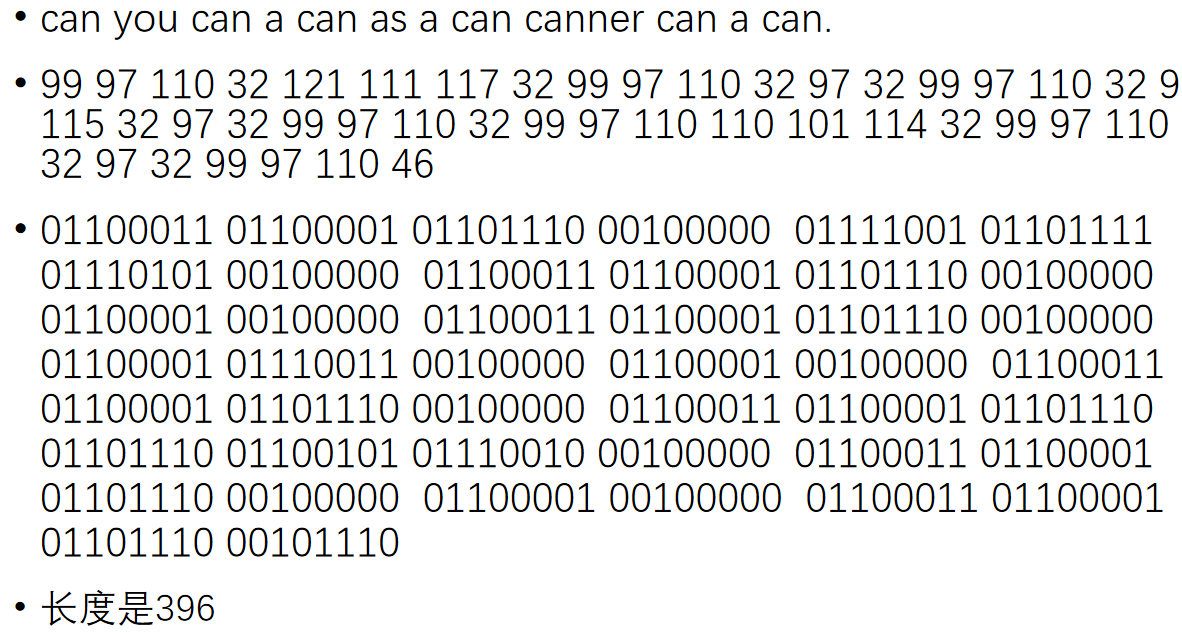
**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{3, 7, 8, 29, 5, 11, 23, 14};  
 *createHuffmanTree*(arr);  
}  
  
**public static** Node createHuffmanTree(**int**[] arr){  
 //创建一个集合，元素是一个个独立的节点。  
 //每个独立的节点，都可以视为一棵独立的二叉树，只不过是二叉树里面只有一个节点而已  
 List<Node> nodes = **new** ArrayList<>();  
 //将数组arr中的元素，赋值给集合中的所有节点元素，作为他们各自的权  
 **for** (**int** value : arr) {  
 nodes.add(**new** Node(value));  
 }  
 //将集合nodes中的所有节点（二叉树），按照根节点的权，从大到小排列  
 //这样根节点权最大的二叉树在集合的最左边，权最小的在最右边  
 Collections.*sort*(nodes);  
 //nodes中装的是每个二叉树的根节点  
 //只要集合nodes中的节点（二叉树）的数量>1，就进入while循环  
 **while** (nodes.size()>1){  
 //取出nodes中根节点的权最小的两个节点，并且将这两个节点从nodes中移除  
 Node left = nodes.remove(nodes.size() - 1);  
 Node right = nodes.remove(nodes.size() - 1);  
 //创建一个新的节点，作为left和right的父节点，父节点的权，正好是他们两个的和  
 Node parent = **new** Node(left.getValue() + right.getValue());  
 //将left和right绑定在parent上  
 parent.setLeft(left);  
 parent.setRight(right);  
 //然后将parent存入集合nodes中去  
 nodes.add(parent);  
 }  
 System.***out***.println(nodes);  
 **return null**;  
}

效果：



### 3、赫夫曼编码

#### 3.1定长编码



编码时：

每一个字符，都转为其ASCII码。再将每一个数，转为一个8位二进制数，比如99用八位二进制数表示就是0110 0011

解码时：

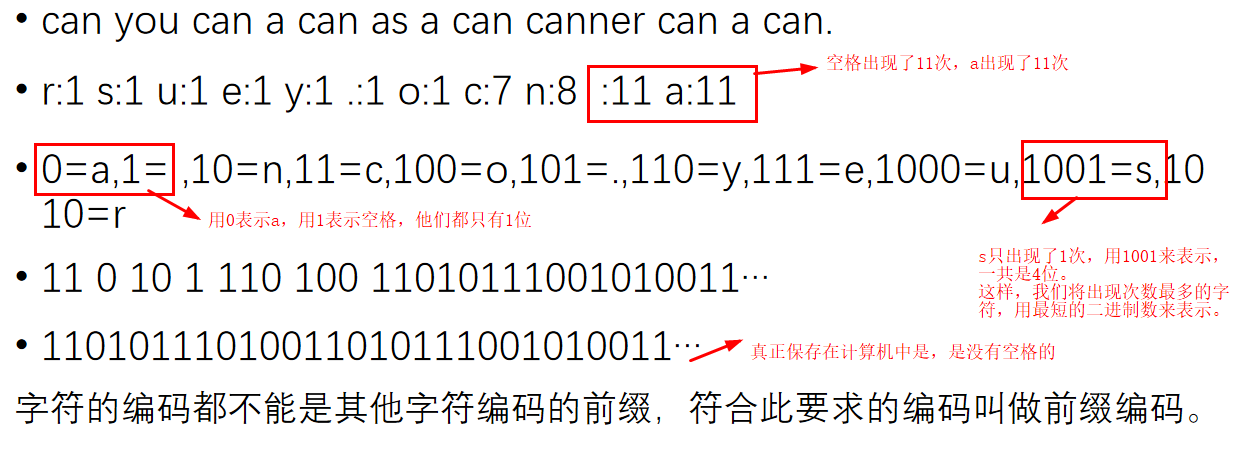
也是每8位二进制为一组，转换成对应的10进制数，再通过ASCII码，将其转为对应的字符

缺点：

很明显，这是最简单的编码方式，但是也是最占用资源的方式，就上面那一句话“can you can a can as a can canner can a can.”。用二进制数来表达时，就需要396位。

#### 3.2非定长编码

图解：



编码时：

首先统计每个字符出现的次数，然后将出现次数最多的字符，用最小的二进制数来表示。比如a出现了11次，次数最多，就用0来表示，这样就只占用了1位。就算把所有的1都编码完，也会需要占用11位，这样就大大的节省了空间。

缺点：

在解码的时候，存在很大的问题。因为最终存储在计算机中时，时没有空格的，所以解码就会出现一个问题。

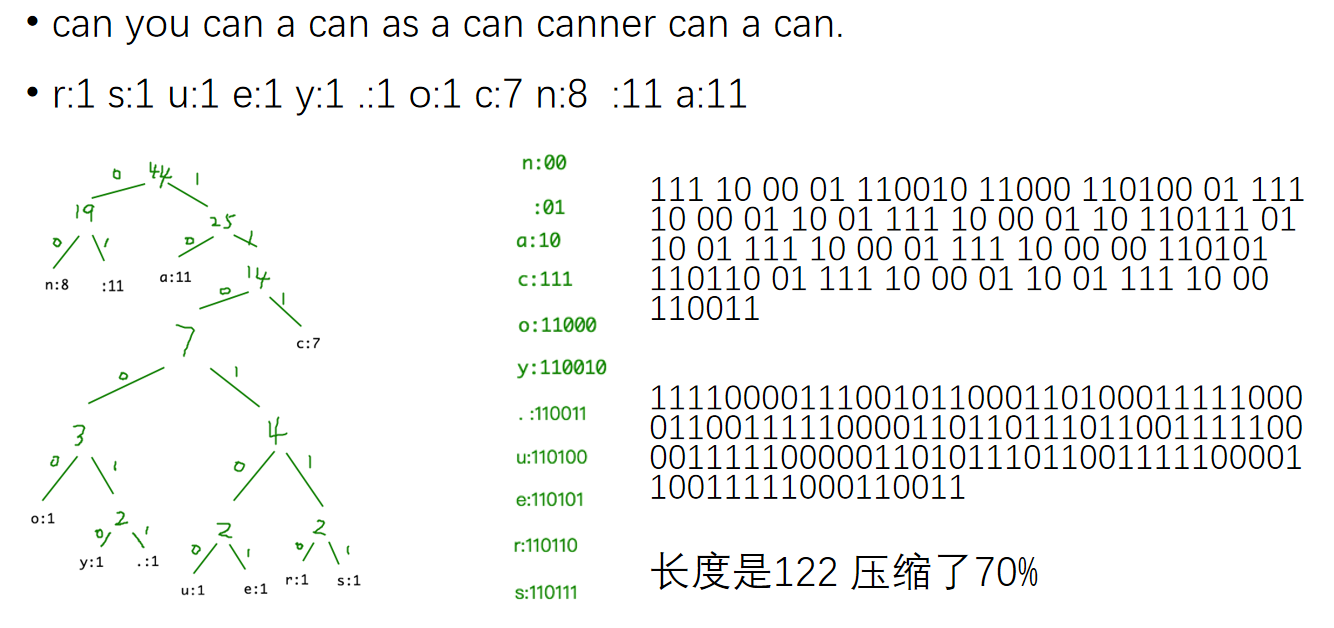
比如说对同一个11，解码的时候，就有多种解码方式，比如可以看作是两个1，即两个空格，也可以看作是一个11，即c。

因此可以得出一个结论：  
 为了解码的时候，对于同一个二进制数，只能有唯一的一个解码方式。必须满足任意一个字符的编码，都不能是其他字符编码的前缀。

符合此要求的编码叫做前缀编码。

#### 3.3赫夫曼编码

图解：



思路：

第一步，同样是统计每个字符出现的次数。然后我们开始思考如何编码。

我们知道，总长度=每个字符各自的长度×该字符出现的次数。

要想让总长度最短，我们应该让出现次数最多的字符的长度最短，让出现次数第二多的字符长度第二短，依次类推。

这不就是赫夫曼树的思想么？

如果我们用节点来表示各个不同的字符，用节点的权来表示字符出现的次数，用路径来表示二进制数与字符的对应规则，不就可以了吗？

对于一个节点，如果要找到它的左子节点，我们就让路径用0表示；如果要找到右子节点，我们就让路径用1表示。

按照这个规则设计一个赫夫曼树，这样不仅最终总长度是最短的，而且找到每一个叶子节点的路径也是唯一的，即满足前缀编码的要求。

这就是赫夫曼编码的思想。

### 4、代码实现

代码实现赫夫曼编码

内容比较多，该类的名称为：HuffmanCodeTest

也可以用everything直接搜类名，在IDEA里面看比较方便。

代码：

//main方法  
**public static void** main(String[] args) {  
 String msg = **"can you can a can as a can canner can a can."**;  
 //获取msg的初始字节数组，每一个字节元素，就代表一个字符  
 **byte**[] bytes = msg.getBytes();  
 System.***out***.println(**"压缩前的数组："** + Arrays.*toString*(bytes));  
 //进行赫夫曼编码  
 **byte**[] b = *huffmanZip*(bytes);  
 System.***out***.println(**"压缩后的数组："** + Arrays.*toString*(b));  
 System.***out***.println(**"压缩前的数组长度："** + bytes.**length**);  
 System.***out***.println(**"压缩后的数组长度："** + b.**length**);  
}  
  
//赫夫曼编码——总方法  
**private static byte**[] huffmanZip(**byte**[] bytes) {  
 //先统计每一个byte出现的次数，并存入一个集合中  
 List<Node> nodes = *getNodes*(bytes);  
 //创建一个赫夫曼树  
 Node tree = *createHuffmanTree*(nodes);  
 //根据赫夫曼树的根节点，获取该树的赫夫曼编码  
 //编码是一个map，key代表字符，value代表字符的路径  
 Map<Byte, String> huffCodes = *getCodes*(tree);  
 //编码  
 **byte**[] b = *zip*(bytes, huffCodes);  
 **return** b;  
}  
  
*/\*\*  
 \* 进行赫夫曼编码  
 \** ***@param bytes*** *需要进行编码的字符数组  
 \** ***@param huffCodes*** *赫夫曼编码表，是一个map  
 \** ***@return*** *\*/***private static byte**[] zip(**byte**[] bytes, Map<Byte, String> huffCodes) {  
 StringBuilder sb = **new** StringBuilder();  
 //bytes中的每一个字节元素，就代表一个字符，我们遍历这个bytes数组  
 //根据字符，在赫夫曼表中其对应的路径，将其挨个添加到字符串sb后面  
 **for** (**byte** b : bytes) {  
 String newByte = huffCodes.get(b);  
 sb.append(newByte);  
 }  
 //因为现在sb里面全是0、1，我们每8位，就存一个字节，那么能存多少个字节呢  
 //定义长度，也就是能存多少个字节  
 **int** len;  
 **if**(sb.length() % 8 == 0){  
 //如果刚好能除尽，那么商是多少，就说明能长度是几  
 len = sb.length() / 8;  
 }**else** {  
 //如果除不尽，那么余数也将其看作一个字节  
 len = sb.length() / 8 + 1;  
 }  
 //定义by，用于存储压缩后的byte  
 **byte**[] by = **new byte**[len];  
 //定义指针，指向by中元素应该存储的索引位置  
 **int** index = 0;  
 **for** (**int** i = 0; i < sb.length(); i = i + 8 ) {  
 //定义字符串strByte，用于存8位二进制数  
 String strByte;  
 **if**(i + 8 > sb.length()){  
 //最后余下的几位  
 strByte = sb.substring(i);  
 }**else** {  
 //subString截取的索引范围是[i,i+8)  
 strByte = sb.substring(i, i + 8);  
 }  
 //将每一个8位二进制数strByte转为10进制数  
 **byte** byt = (**byte**) Integer.*parseInt*(strByte, 2);  
 by[index] = byt;  
 index++;  
 }  
 **return** by;  
}  
  
//定义一个成员变量sb,用于临时存储路径  
**private static** StringBuilder *sb* = **new** StringBuilder();  
//定义一个成员变量huffCodes，用于临时存储赫夫曼编码。  
//即huffCodes这个map中的每一个元素存的是：叶节点的字符和路径  
**private static** Map<Byte, String> *huffCodes* = **new** HashMap<>();  
  
*/\*\*  
 \* 根据赫夫曼树的根节点，获取该树的赫夫曼编码  
 \** ***@param tree*** *\** ***@return*** *\*/***private static** Map<Byte, String> getCodes(Node tree) {  
 //当根节点为null时，直接返回  
 **if**(tree == **null**){  
 **return null**;  
 }  
 *getCodes*(tree.getLeft(), **"0"**, *sb*);  
 *getCodes*(tree.getRight(),**"1"**, *sb*);  
 **return** *huffCodes*;  
}  
  
*/\*\*  
 \* 获取以当前节点为根节点的赫夫曼树的编码表  
 \** ***@param node*** *当前节点  
 \** ***@param code*** *从上一个节点移动到当前节点的方向，“0”代表左边，“1”代表右边  
 \** ***@param sb*** *移动之前存储的路径  
 \* 这个方法是最难理解的，因为有递归，而且用到了2个临时成员变量。  
 \*/***private static void** getCodes(Node node, String code, StringBuilder sb) {  
 StringBuilder sb2 = **new** StringBuilder(sb);  
 sb2.append(code);  
 **if**(node.getData() == **null**){  
 *getCodes*(node.getLeft(), **"0"**, sb2);  
 *getCodes*(node.getRight(), **"1"**, sb2);  
 }**else** {  
 //当node的data不为null时，说明该node是一个叶节点  
 *huffCodes*.put(node.getData(), sb2.toString());  
 }  
}  
  
  
*/\*\*  
 \* 创建赫夫曼树  
 \** ***@param nodes*** *\** ***@return*** *\*/***private static** Node createHuffmanTree(List<Node> nodes) {  
 **while** (nodes.size() > 1){  
 //排序：将nodes中的元素，按照成员weight的大小，倒序排列  
 Collections.*sort*(nodes);  
 //取出两个权最低的节点,并将这两个节点从集合nodes中删除  
 Node left = nodes.remove(nodes.size() - 1);  
 Node right = nodes.remove(nodes.size() - 1);  
 //创建新的节点parent  
 Node parent = **new** Node(**null**, left.getWeight() + right.getWeight());  
 //将刚才取出的left和right绑定为parent的左右子节点  
 parent.setLeft(left);  
 parent.setRight(right);  
 //将节点parent添加到nodes中  
 nodes.add(parent);  
 }  
 **return** nodes.get(0);  
}  
  
  
  
*/\*\*  
 \* 传入一个字节数组bytes，返回一个节点集合  
 \* 集合中每一个节点元素都保存了两个信息:data、weight  
 \* data代表数组bytes中的字节，weight代表数组bytes中该字节出现的次数  
 \** ***@param bytes*** *\** ***@return*** *\*/***private static** List<Node> getNodes(**byte**[] bytes) {  
 List<Node> nodes = **new** ArrayList<>();  
 //传入一个数组，统计该数组中有哪些元素，以及每个元素出现的次数  
 Map<Byte, Integer> counts = **new** HashMap<>();  
 **for** (**byte** b : bytes) {  
 //获取字符b出现的次数count  
 Integer count = counts.get(b);  
 //千万注意：这里不要写成了count==0，否则会报空指针异常  
 //因为count是Integer类型，是引用类型，而不是int类型。  
 //因此当b不存在的时候，count为null，而不是为0，这一点千万要注意。  
 **if**(count == **null**){  
 //count为0，说明该字符不存在，那么就存入counts这个map中,并将count设为1  
 counts.put(b, 1);  
 }**else** {  
 //当字符存在时，就再存一次，只不过让count+1  
 counts.put(b, count + 1);  
 }  
 }  
 //把每一个键值对，转为一个node对象存入集合nodes中  
 //entry方式遍历counts这个Map集合  
 Node node = **null**;  
 Set<Map.Entry<Byte, Integer>> entries = counts.entrySet();  
 **for** (Map.Entry<Byte, Integer> entry : entries) {  
 Byte data = entry.getKey();  
 Integer weight = entry.getValue();  
 node = **new** Node(data, weight);  
 nodes.add(node);  
 }  
 **return** nodes;  
}

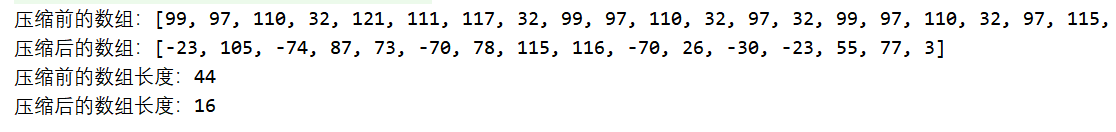
效果：

压缩前的数组：[99, 97, 110, 32, 121, 111, 117, 32, 99, 97, 110, 32, 97, 32, 99, 97, 110, 32, 97, 115, 32, 97, 32, 99, 97, 110, 32, 99, 97, 110, 110, 101, 114, 32, 99, 97, 110, 32, 97, 32, 99, 97, 110, 46]

压缩后的数组：[-23, 105, -74, 87, 73, -70, 78, 115, 116, -70, 26, -30, -23, 55, 77, 3]

压缩前的数组长度：44

压缩后的数组长度：16



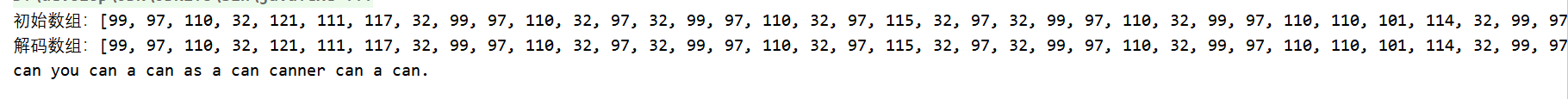
### 5、解码

我认为，目前阶段，了解即可。

代码：

//main方法  
**public static void** main(String[] args) {  
 String msg = **"can you can a can as a can canner can a can."**;  
 //获取msg的初始字节数组，每一个字节元素，就代表一个字符  
 **byte**[] bytes = msg.getBytes();  
 //进行赫夫曼编码压缩  
 **byte**[] b = *huffmanZip*(bytes);  
 //使用赫夫曼编码表解码  
 **byte**[] newBytes = *decode*(*huffCodes*, b);  
 System.***out***.println(**"初始数组："** + Arrays.*toString*(bytes));  
 System.***out***.println(**"解码数组："** + Arrays.*toString*(newBytes));  
 System.***out***.println(**new** String(newBytes));  
  
}  
  
*/\*\*  
 \* 使用指定的赫夫曼编码表进行解码  
 \** ***@param huffCodes*** *编码、解码所共同使用的赫夫曼编码表  
 \** ***@param bytes*** *需要被解码的数组  
 \** ***@return*** *\*/***private static byte**[] decode(Map<Byte, String> huffCodes, **byte**[] bytes) {  
 StringBuilder sb = **new** StringBuilder();  
 //把bytes数组中的所有元素，由10进制数转为8位二进制数，然后变为字符串追加到sb后面  
 **for** (**int** i = 0; i < bytes.**length**; i++) {  
 **byte** b = bytes[i];  
 //判断b是不是数组bytes中的最后一个元素  
 **boolean** flag = (i == bytes.**length** - 1);  
 //将每一个10进制的b，都转为一个8位二进制数，然后将结果追加到sb后面  
 sb.append(*byteToBitString*(!flag, b));  
 }  
 //把赫夫曼编码表的键值对调换位置，让key变为路径，而value变为字符  
 Map<String, Byte> map = **new** HashMap<>();  
 **for** (Map.Entry<Byte, String> entry : huffCodes.entrySet()) {  
 map.put(entry.getValue(), entry.getKey());  
 }  
 //创建一个集合，用于存byte  
 List<Byte> list = **new** ArrayList<>();  
 //处理字符串  
 **for** (**int** i = 0; i < sb.length();) {  
 **int** count = 1;  
 **boolean** flag = **true**;  
 Byte b = **null**;  
 **while** (flag){  
 String key = sb.substring(i, i + count);  
 b = map.get(key);  
 **if** (b == **null**){  
 //如果根据[i,count)截取出来的路径没有找到对应的字符  
 //则让指针count右移  
 count++;  
 }**else** {  
 //如果找到了,就退出while循环。  
 flag = **false**;  
 }  
 }  
 //每进一次while循环，就会找出一个字符,将其添加到list  
 list.add(b);  
 i = i + count;  
 }  
 //把集合list转为数组  
 **byte**[] b = **new byte**[list.size()];  
 **for** (**int** i = 0; i < b.**length**; i++) {  
 b[i] = list.get(i);  
 }  
 **return** b;  
}  
  
*/\*\*  
 \* 将10进制数b，转成8位二进制的数，并以字符串形式返回  
 \** ***@param flag*** *标记，判断是否需要在左边补0  
 \** ***@param b*** *\** ***@return*** *\*/***private static** String byteToBitString(**boolean** flag, **byte** b){  
 **int** temp = b;  
 //进行“或”操作，256的二进制数为：1 0000 0000  
 //其实就是将temp的二进制数补全成8位，即在左边补0  
 **if**(flag){  
 temp = temp | 256;  
 }  
 String str = Integer.*toBinaryString*(temp);  
 **if**(flag){  
 //如果之前补过0，就截取str的最后8位数字返回  
 **return** str.substring(str.length() - 8);  
 }**else** {  
 //如果没有补0的话，就直接将str返回  
 **return** str;  
 }  
}

效果：



### 6、文件压缩与解压

暂时跳过。

## 五、二叉排序树

### 1、概述

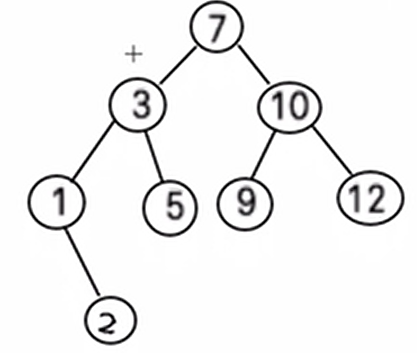
二叉排序树：

也叫二叉查找树、二叉搜索树：BST。

对于二叉树中的任何一个非叶子节点，它与它的左右子节点之间权的大小排序为：

左子节点 < 当前节点 < 右子节点

图解：



当我们要插入一个2时，只需要在节点1的右子节点处添加，这样添节点的效率就很高。

当我们要查找时，比如查找12，首先在根节点处，12>7，那么就往根节点右边找，发现12>10，就往10的右子节点找，这样就找到12了，这样查找的效率也很高。

### 2、基本用法

包含了添加节点、中序遍历排序二叉树

Tips：

中序遍历排序二叉树时，遍历的结果正好是节点的权值从小到达排列

#### 2.1树

*/\*\*  
 \* 二叉排序树  
 \*/***public class** BinarySortTree {  
 **private** Node **root**;  
  
 //向二叉排序树中添加节点  
 **public void** add(Node node){  
 **if**(**root** == **null**){  
 **root** = node;  
 }**else** {  
 **root**.add(node);  
 }  
 }  
  
 //中序遍历  
 **public void** midShow(){  
 **if** (**root** != **null**){  
 **root**.midShow(**root**);  
 //遍历结束后换行  
 System.***out***.println();  
 }  
 }  
}

#### 2.2节点

*/\*\*  
 \* 二叉排序树的节点  
 \*/*@Data  
**public class** Node {  
 **private int value**;  
 **private** Node **left**;  
 **private** Node **right**;  
  
 **public** Node(**int** value) {  
 **this**.**value** = value;  
 }  
  
 */\*\*  
 \* 中序遍历  
 \** ***@param node*** *当前节点  
 \* 中序遍历二叉排序树的结果：权值正好从小到大  
 \*/* **public void** midShow(Node node) {  
 **if** (node == **null**){  
 **return**;  
 }  
 //遍历左边  
 midShow(node.getLeft());  
 //打印当前节点  
 System.***out***.print(node.getValue() + **" "**);  
 //遍历右边  
 midShow(node.getRight());  
 }  
  
 */\*\*  
 \* 向子树中添加节点  
 \** ***@param node*** *\*/* **public void** add(Node node) {  
 **if**(node == **null**){  
 **return**;  
 }  
 //哪个节点调用当前这个add方法，哪个节点就是当前节点  
 //判断传入的节点node的权，与当前节点的权，哪个更大  
 **if**(node.getValue() < **this**.getValue()){  
 //node比当前节点小时，应该将node放到当前节点的左边  
 **if**(**this**.getLeft() == **null**){  
 //如果当前节点的左边为空，就直接放到左边  
 **this**.setLeft(node);  
 }**else** {  
 //如果当前节点的左边不为空，就递归下去  
 **this**.getLeft().add(node);  
 }  
 }**else** {  
 //node比当前节点大时，应该将node放到当前节点的右边  
 **if**(**this**.getRight() == **null**){  
 //如果当前节点的右边为空，就直接放到右边  
 **this**.setRight(node);  
 }**else** {  
 //如果当前节点的右边不为空，就递归下去  
 **this**.getRight().add(node);  
 }  
 }  
 }  
   
}

#### 2.3测试类

**public static void** main(String[] args) {  
 **int**[] arr = **new int**[]{7, 3, 10, 12, 5, 1, 9};  
 //创建一棵二叉排序树  
 BinarySortTree bst = **new** BinarySortTree();  
 //往树中添加节点  
 **for** (**int** i : arr) {  
 bst.add(**new** Node(i));  
 }  
 //中序遍历二叉树  
 bst.midShow();  
}

结果：



### 3、查找节点

#### 3.1树

//查找节点  
**public** Node search(**int** value){  
 **if** (**root** == **null**){  
 **return null**;  
 }**else** {  
 **return root**.search(value);  
 }  
}

#### 3.2节点

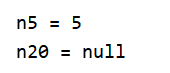
*/\*\*  
 \* 查找节点  
 \** ***@param value*** *要查找的权  
 \** ***@return*** *\*/***public** Node search(**int** value) {  
 **if**(value == **this**.getValue()){  
 **return this**;  
 }**else if**(value < **this**.getValue()){  
 //value比当前节点的权小时，就继续往当前节点的左边查找  
 **if** (**this**.getLeft() == **null**){  
 **return null**;  
 }  
 //左子节点存在时  
 **return this**.getLeft().search(value);  
 }**else** {  
 //value比当前节点的权大时，就继续往右边查找  
 **if**(**this**.getRight() == **null**){  
 **return null**;  
 }  
 //右子节点存在时  
 //这一步放在else里面，与不放的效果是完全一样的，因为当右边为空时，就直接就return了(很重要，一定要理解)  
 //那么这边这个就不会再执行了。也就是说，当value偏大时，要么return null，  
 //要么继续往右边查找，这两种情况，只可能是二选一。  
 **return this**.getRight().search(value);  
 }  
}

@Override  
**public** String toString() {  
 **return this**.getValue() + **" "**;  
}

#### 3.3测试

//查找节点  
@Test  
**public void** test2() {  
 Node n5 = **bst**.search(5);  
 Node n20 = **bst**.search(20);  
 System.***out***.println(**"n5 = "** + n5);  
 System.***out***.println(**"n20 = "** + n20);  
}

效果：



我已经重写了Node的toString方法了，让它直接返回Node的权。

### 4、删除节点

#### 4.1概述

首先注意一点：

二叉排序树里面的删除节点，和我们以前学的其他二叉树的删除节点可不一样。

我们之前学其他二叉树中，删除节点都指的是将以该节点为根节点的子树，整个都给删除了。

而二叉排序树里面，删除节点，真就只是删除一个节点而已，如果这个节点左右还有子树、或者子节点的话，他们会重新排列。

所以二叉排序树里面的删除节点，又分为多种情况。

① 要删除的节点是叶子节点时，这种情况最简单，因为删除后不需要从新排列

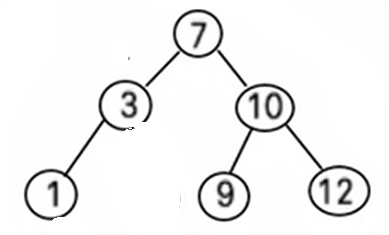
② 要删除的节点的左右子节点有且仅有一个时。（注意哦，这里说的是子节点，可不是子树哦，因为我们只需要管要删除的节点的左右子节点就行了，并不需要进行递归删除）

③ 要删除的节点的左右子节点都存在时。

简单说一下思路：

第①个就不说了，直接删就行。

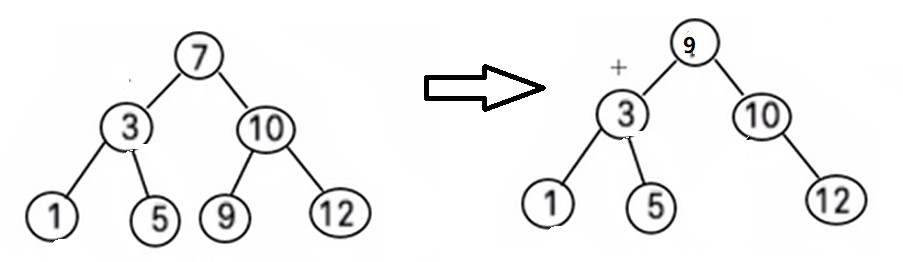
第②种情况：



比如我们要删除节点3，此时3是有一个子节点的，比如上图就是左子节点1，

那么我们只需要让节点7的左子节点指向节点1，这样节点3就被删除了。

第③种情况：



比如我们要删除节点7。

我们操作的时候，是先找到7的右子树中最小的节点，比如上图中就是9，我们将节点9删除，并将9的值赋值给7，这样就相当于将7给删除了。

当然，9可能也有子节点，但是只能是9的右子节点，否则9就不是7的右子树中最小的了。我们直接调用整个方法，将9删除即可。

#### 4.2树

*/\*\*  
 \* 二叉排序树  
 \*/***public class** BinarySortTree {  
 **private** Node **root**;  
  
 */\*\*  
 \* 从以root节点为根节点的树中，删除权为value的节点  
 \*/* **public void** delete(**int** value){  
 **if** (**root** == **null**){  
 **return**;  
 }  
 //先找到权为value的节点，也就是要删除的节点  
 Node target = search(value);  
 **if**(target == **null**){  
 //如果该节点不存在  
 **return**;  
 }  
 //如果该节点存在，就先找到它的父节点  
 Node parent = searchParent(value);  
 //如果要删除的节点，即target是叶子节点时  
 **if**(target.getLeft() == **null** && target.getRight() == **null**){  
 //如果value节点是parent的左子节点  
 **if**(parent.getLeft().getValue() == value){  
 //那么就让parent的左边指向null，就代表将value节点从parent上删除了  
 parent.setLeft(**null**);  
 }**else** {  
 //当value是parent的右子节点时，就让parent的右边指向null  
 parent.setRight(**null**);  
 }  
  
 //当要删除的节点target的左右子节点都存在时  
 }**else if** (target.getLeft() != **null** && target.getRight() != **null**){  
 //删除target的右子树中值最小的节点，并获取这个最小节点的权  
 **int** min = deleteMin(target.getRight());  
 //将这个最小值min，赋值给target  
 target.setValue(min);  
  
 //当要删除的节点tatget的左右子节点中，有且仅有一个存在时  
 }**else** {  
 //当仅有的节点是target的左子节点时  
 **if** (target.getLeft() != **null**){  
 //再判断target是parent的左子节点还是右子节点，并将target从parent上删除  
 //如果target是parent的左子节点  
 **if**(parent.getLeft().getValue() == value){  
 //让parent的左子节点指向target的左子节点  
 parent.setLeft(target.getLeft());  
 //如果target是parent的右子节点  
 }**else** {  
 //让parent的右子节点指向target的左子节点  
 parent.setRight(target.getLeft());  
 }  
  
 //当仅有的节点是target的右子节点时  
 }**else** {  
 //再判断target是parent的左子节点还是右子节点，并将target从parent上删除  
 //如果target是parent的左子节点  
 **if**(parent.getLeft().getValue() == value){  
 //让parent的左子节点指向target的右子节点  
 parent.setLeft(target.getRight());  
 //如果target是parent的右子节点  
 }**else** {  
 //让parent的右子节点指向target的左子节点  
 parent.setRight(target.getRight());  
 }  
  
 }  
 }  
  
  
 }  
  
 */\*\*  
 \* 对与以node节点为根节点的子树，删除该子树中权最小的节点  
 \** ***@param node*** *\** ***@return*** *\*/* **private int** deleteMin(Node node) {  
 //定义一个指针，一开始指向节点node  
 Node target = node;  
 //只要指针target的左边不为null  
 **while** (target.getLeft() != **null**){  
 //就让指针一直左移  
 target = target.getLeft();  
 }  
 //退出while循环时，target就已经指向了node子树中的最小的节点  
 //从node子树中删除该节点，直接调用当前类中的delete方法删除  
 delete(target.getValue());  
 //返回最小节点的权  
 **return** target.getValue();  
 }  
  
 */\*\*  
 \* 根据value，查找树中权为value的节点的父节点  
 \** ***@param value*** *\** ***@return*** *\*/* **public** Node searchParent(**int** value){  
 **if**(**root** == **null**){  
 //根节点为空时，说明树不存在，直接返回。  
 **return null**;  
 }**else** {  
 **return root**.searchParent(value);  
 }  
 }  
  
 //查找节点  
 **public** Node search(**int** value){  
 **if** (**root** == **null**){  
 **return null**;  
 }**else** {  
 **return root**.search(value);  
 }  
 }  
  
 //向二叉排序树中添加节点  
 **public void** add(Node node){  
 **if**(**root** == **null**){  
 **root** = node;  
 }**else** {  
 **root**.add(node);  
 }  
 }  
  
 //中序遍历  
 **public void** midShow(){  
 **if** (**root** != **null**){  
 **root**.midShow(**root**);  
 //遍历结束后换行  
 System.***out***.println();  
 }  
 }  
}

#### 4.3节点

*/\*\*  
 \* 二叉排序树的节点  
 \*/*@Data  
**public class** Node {  
 **private int value**;  
 **private** Node **left**;  
 **private** Node **right**;  
  
 **public** Node(**int** value) {  
 **this**.**value** = value;  
 }  
  
 */\*\*  
 \* 查找节点  
 \** ***@param value*** *要查找的权  
 \** ***@return*** *\*/* **public** Node search(**int** value) {  
 **if**(value == **this**.getValue()){  
 **return this**;  
 }**else if**(value < **this**.getValue()){  
 //value比当前节点的权小时，就继续往当前节点的左边查找  
 **if** (**this**.getLeft() == **null**){  
 **return null**;  
 }  
 //左子节点存在时  
 **return this**.getLeft().search(value);  
 }**else** {  
 //value比当前节点的权大时，就继续往右边查找  
 **if**(**this**.getRight() == **null**){  
 **return null**;  
 }  
 //右子节点存在时  
 //这一步放在else里面，与不放的效果是完全一样的，因为当右边为空时，就直接就return了(很重要，一定要理解)  
 //那么这边这个就不会再执行了。也就是说，当value偏大时，要么return null，  
 //要么继续往右边查找，这两种情况，只可能是二选一。  
 **return this**.getRight().search(value);  
 }  
 }  
  
 */\*\*  
 \* 中序遍历  
 \** ***@param node*** *当前节点  
 \* 中序遍历二叉排序树的结果：权值正好从小到大  
 \*/* **public void** midShow(Node node) {  
 **if** (node == **null**){  
 **return**;  
 }  
 //遍历左边  
 midShow(node.getLeft());  
 //打印当前节点  
 System.***out***.print(node.getValue() + **" "**);  
 //遍历右边  
 midShow(node.getRight());  
 }  
  
 */\*\*  
 \* 向子树中添加节点  
 \** ***@param node*** *\*/* **public void** add(Node node) {  
 **if**(node == **null**){  
 **return**;  
 }  
 //哪个节点调用当前这个add方法，哪个节点就是当前节点  
 //判断传入的节点node的权，与当前节点的权，哪个更大  
 **if**(node.getValue() < **this**.getValue()){  
 //node比当前节点小时，应该将node放到当前节点的左边  
 **if**(**this**.getLeft() == **null**){  
 //如果当前节点的左边为空，就直接放到左边  
 **this**.setLeft(node);  
 }**else** {  
 //如果当前节点的左边不为空，就递归下去  
 **this**.getLeft().add(node);  
 }  
 }**else** {  
 //node比当前节点大时，应该将node放到当前节点的右边  
 **if**(**this**.getRight() == **null**){  
 //如果当前节点的右边为空，就直接放到右边  
 **this**.setRight(node);  
 }**else** {  
 //如果当前节点的右边不为空，就递归下去  
 **this**.getRight().add(node);  
 }  
 }  
 }  
  
 @Override  
 **public** String toString() {  
 **return this**.getValue() + **" "**;  
 }  
  
 */\*\*  
 \* 在以当前节点为根节点的数中，搜索value节点的父节点  
 \** ***@param value*** *\** ***@return*** *\*/* **public** Node searchParent(**int** value) {  
 //如果当前节点的左右子节点中存在一个为value，说明当前节点就是value的父节点，直接返回当前节点  
 **if**((**this**.getLeft() != **null**&& **this**.getLeft().getValue() == value) ||  
 (**this**.getRight() != **null** && **this**.getRight().getValue() == value) ){  
 **return this**;  
 //如果当前节点的左右子节点都不是value,则  
 }**else** {  
 //如果value<当前节点的权，则继续往当前节点的左边递归查找  
 **if**(**this**.getLeft() != **null** && value < **this**.getValue()){  
 **return this**.getLeft().searchParent(value);  
 //如果value>当前节点的权。则继续往当前节点的右边递归查找  
 }**else if**(**this**.getRight() != **null** && value > **this**.getValue()){  
 **return this**.getRight().searchParent(value);  
 }  
 }  
 **return null**;  
 }  
}

#### 4.4测试

//定义成员，排序二叉树  
**private** BinarySortTree **bst** = **new** BinarySortTree();  
  
//构建排序二叉树模型  
@Before  
**public void** setUp() **throws** Exception {  
 **int**[] arr = **new int**[]{7, 3, 10, 5, 12, 1, 9};  
 //往树中添加节点  
 **for** (**int** i : arr) {  
 **bst**.add(**new** Node(i));  
 }  
}  
  
//当要删除的节点的左右节点都存在时  
@Test  
**public void** test6() {  
 //比如我们要删除权为7的节点  
 **bst**.delete(7);  
 Node n9 = **bst**.search(9);  
 System.***out***.println(**"9的左节点："** + n9.getLeft());//9的左节点：3  
 System.***out***.println(**"9的右节点："** + n9.getRight());//9的右节点：10  
 **bst**.midShow();//1 3 5 9 10 12  
}  
  
//当删除的节点仅有一个节点时  
@Test  
**public void** test5() {  
 //我们先去掉树中的5，然后删除3，看能不能删掉，并且删掉后，是不是1变成了7的左子节点  
 **bst**.delete(3);  
 //获取7的左子节点的value  
 **int** value = **bst**.search(7).getLeft().getValue();  
 System.***out***.println(**"value = "** + value);//value = 1  
 **bst**.midShow();//1 7 9 10 12  
}  
  
//删除的节点是叶子节点时  
@Test  
**public void** test4() {  
 //删除叶子节点  
 **bst**.delete(9);  
 //删除后遍历这棵排序二叉树  
 **bst**.midShow();  
}  
  
//查找value的父节点  
@Test  
**public void** test3() {  
 Node node = **bst**.searchParent(12);  
 System.***out***.println(**"node = "** + node);  
}  
  
//查找节点  
@Test  
**public void** test2() {  
 Node n5 = **bst**.search(5);  
 Node n20 = **bst**.search(20);  
 System.***out***.println(**"n5 = "** + n5);  
 System.***out***.println(**"n20 = "** + n20);  
}

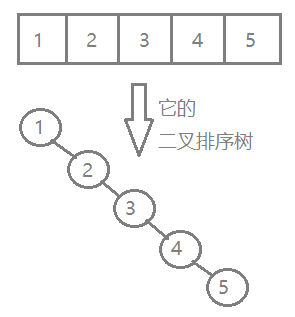
测试截图就不一一列举了，经测试是没问题的。

## 六、AVL树

### 1、概述

我们使用二叉排序树的初衷，是因为它增删的速度、查找的速度都还挺快。但是它仍然是有缺陷的。

比如有一个数组arr = ｛1、2、3、4、5｝。那么该数组的二叉排序树设计如下：



如果我们要查找5，就要一个一个查找过来，并且，每当查找到一个节点的时候，不是仅仅比较是否等于该节点就完事了，还需要判断比该节点大，还是小，这样才能判断往左边找，还是右边找。这样一来，这棵二叉排序树的查找效率甚至不如单链表。

因此我们引入了另一个概念：AVL，平衡二叉树。

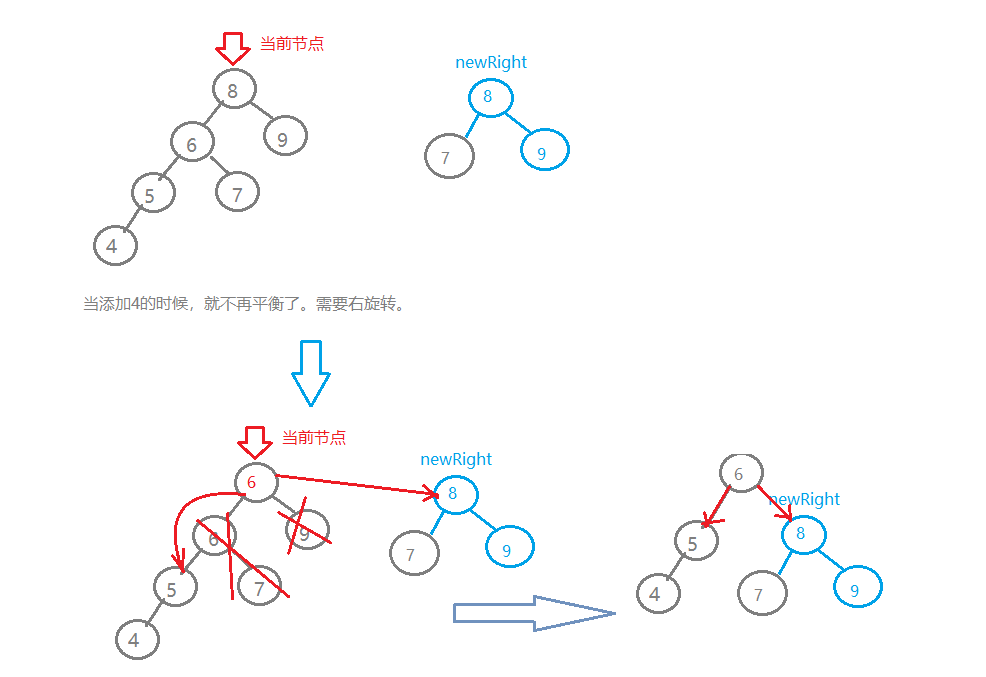
AVL：

即平衡二叉树。平衡二叉树也是一棵二叉排序树，并且对于任何一个节点而言，它的左子树与右子树之间高度差的绝对值都不超过1。

### 2、单旋转

这里我们只介绍右旋转，左旋转就不介绍了，因为思路都是一样的。

右旋转思路：



#### 2.1树

*/\*\*  
 \* 平衡排序树\_\_\_AVL  
 \*/*@Data  
**public class** AVLTree {  
 **private** Node **root**;  
  
 //向二叉排序树中添加节点  
 **public void** add(Node node){  
 **if**(**root** == **null**){  
 **root** = node;  
 }**else** {  
 **root**.add(node);  
 }  
 }  
}

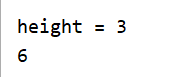
#### 2.2节点

*/\*\*  
 \* 平衡二叉树的节点\_\_\_AVL  
 \*/*@Data  
**public class** Node {  
 **private int value**;  
 **private** Node **left**;  
 **private** Node **right**;  
  
 **public** Node(**int** value) {  
 **this**.**value** = value;  
 }  
  
 */\*\*  
 \* 返回以当前节点为根节点的子树高度  
 \** ***@return*** *\*/* **public int** height(){  
 //左子树高度，当左子树为null时，返回0  
 **int** leftH = (**this**.getLeft() == **null** ? 0 : **this**.getLeft().height());  
 //右子树高度  
 **int** rightH = (**this**.getRight() == **null** ? 0 : **this**.getRight().height());  
 **return** Math.*max*(leftH, rightH) + 1;  
 }  
  
 */\*\*  
 \* 获取当前节点的左子树高度  
 \** ***@return*** *\*/* **public int** leftHeight(){  
 **if**(**this**.getLeft() == **null**){  
 **return** 0;  
 }  
 **return this**.getLeft().height();  
 }  
  
 */\*\*  
 \* 获取当前节点的右子树高度  
 \** ***@return*** *\*/* **public int** rightHeight(){  
 **if**(**this**.getRight() == **null**){  
 **return** 0;  
 }  
 **return this**.getRight().height();  
 }  
  
 */\*\*  
 \* 向子树中添加节点  
 \** ***@param node*** *\*/* **public void** add(Node node) {  
 **if**(node == **null**){  
 **return**;  
 }  
 //哪个节点调用当前这个add方法，哪个节点就是当前节点  
 //判断传入的节点node的权，与当前节点的权，哪个更大  
 **if**(node.getValue() < **this**.getValue()){  
 //node比当前节点小时，应该将node放到当前节点的左边  
 **if**(**this**.getLeft() == **null**){  
 //如果当前节点的左边为空，就直接放到左边  
 **this**.setLeft(node);  
 }**else** {  
 //如果当前节点的左边不为空，就递归下去  
 **this**.getLeft().add(node);  
 }  
 }**else** {  
 //node比当前节点大时，应该将node放到当前节点的右边  
 **if**(**this**.getRight() == **null**){  
 //如果当前节点的右边为空，就直接放到右边  
 **this**.setRight(node);  
 }**else** {  
 //如果当前节点的右边不为空，就递归下去  
 **this**.getRight().add(node);  
 }  
 }  
 //查询是否平衡  
 //当左子树高度 - 右子树高度 >= 2时 右旋转  
 **if** (**this**.leftHeight() - **this**.rightHeight() >= 2){  
 rightRotate();  
 }  
 }  
  
 //右旋转  
 **private void** rightRotate() {  
 //创建一个新节点，值就等于当前节点的值，也就是新8  
 Node newRight = **new** Node(**this**.getValue());  
 //让新节点的右子树指向当前节点的右子树：让新8指向新9  
 newRight.setRight(**this**.getRight());  
 //让新8的左子树指向7  
 newRight.setLeft(**this**.getLeft().getRight());  
 //把当前节点的左子节点的权 赋值给 当前节点。这样当前节点的权就变成了6  
 **this**.setValue(**this**.getLeft().getValue());  
 //让当前节点的左子树指向5。（相当于将6,和6的右子树都给删除了）  
 **this**.setLeft(**this**.getLeft().getLeft());  
 //让当前节点的右子节点指向新8。这样相当于同时也将9给删除了  
 **this**.setRight(newRight);  
 }  
   
}

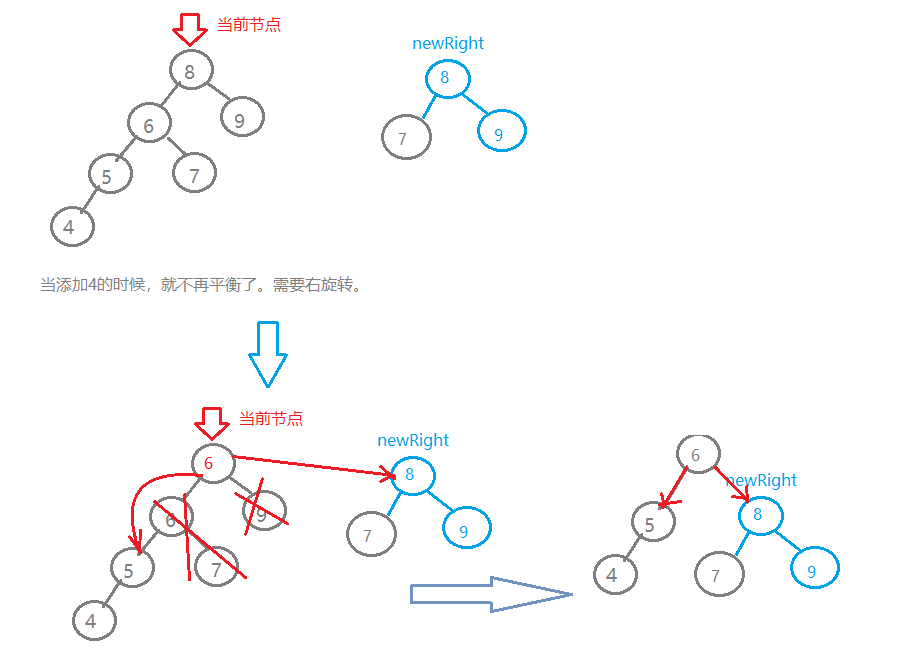
#### 2.3测试

**public class** AVLTreeTest {  
 //定义成员，平衡二叉树  
 **private** AVLTree **avl** = **new** AVLTree();  
  
 //构建平衡二叉树模型  
 @Before  
 **public void** setUp() **throws** Exception {  
 **int**[] arr = **new int**[]{8, 6, 9, 5, 7, 4};  
 //添加节点  
 **for** (**int** i : arr) {  
 **avl**.add(**new** Node(i));  
 }  
 }  
  
 //查看平衡二叉树高度  
 @Test  
 **public void** test1() {  
 **int** height = **avl**.getRoot().height();  
 System.***out***.println(**"height = "** + height);  
 System.***out***.println(**avl**.getRoot());  
 }  
}

效果：

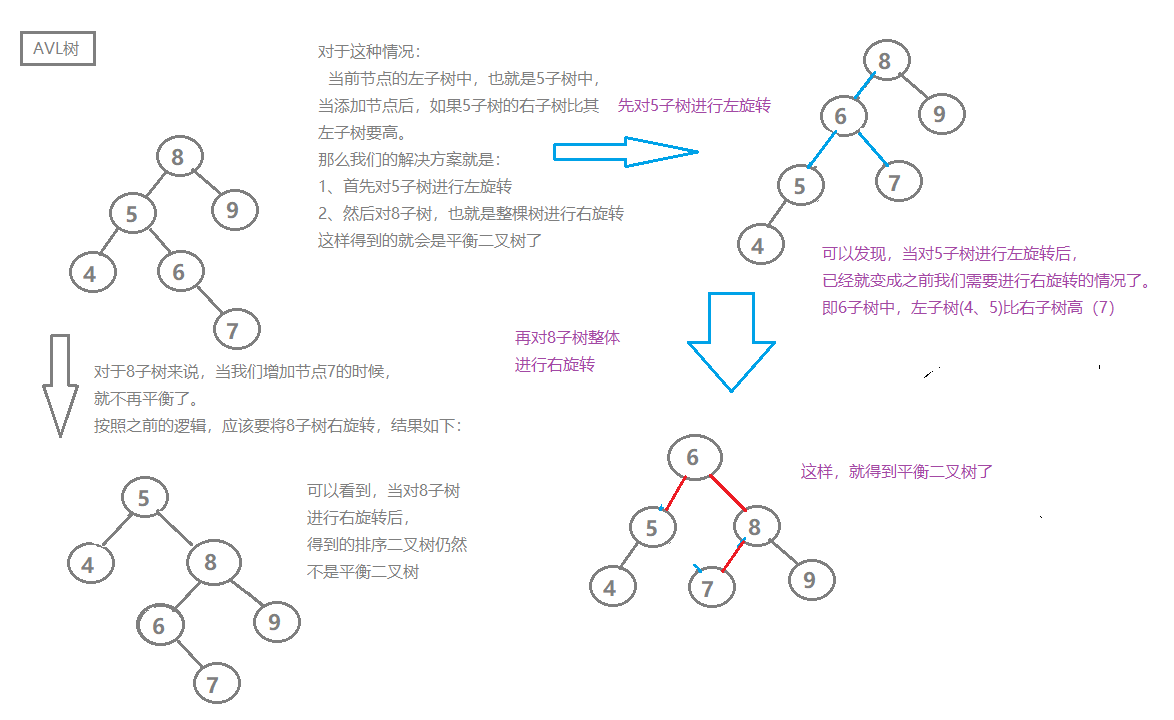


结合原理图：



### 3、双旋转

仅靠左旋转和右旋转，其实不能解决所有的情况，如下图：



这里演示的只是双旋转中的一种情况：左右

即先将根节点的左子树进行坐旋转，然后对根字树整体进行右旋转。

进行这种操作的条件：

首先因为根子树整体是要右旋转的，所以应该满足：

根节点的左子树高度 - 根节点的右子树高度 >= 2

但是发现根节点的左节点，即5，发现5的左子树高度，比5的右子树高度低，那么首先将5子树进行坐旋转。

之后在将8子树整体右旋转即可。

整体右旋转部分的代码改动为：

//查询是否平衡  
//当左子树高度 - 右子树高度 >= 2时 右旋转  
//右旋转  
**if** (**this**.leftHeight() - **this**.rightHeight() >= 2){  
 **if**(**this**.getLeft()!=**null** && **this**.getLeft().leftHeight() < **this**.getLeft().rightHeight()){  
 //将根节点的左子树进行左旋转  
 **this**.getLeft().leftRotate();  
 }  
 //根子树右旋转  
 rightRotate();  
  
}  
  
//左旋转  
**if** (**this**.leftHeight() - **this**.rightHeight() <= -2){  
 **if** (**this**.getRight()!=**null** && **this**.getRight().leftHeight() > **this**.getRight().rightHeight()){  
 //将根节点的右子树进行右旋转  
 **this**.getRight().rightRotate();  
 }  
 //根子树左旋转  
 leftRotate();  
}

左旋转：

//左旋转  
**public void** leftRotate() {  
 Node newLeft = **new** Node(**this**.getValue());  
 newLeft.setLeft(**this**.getLeft());  
 newLeft.setRight(**this**.getRight().getLeft());  
 **this**.setValue(**this**.getRight().getValue());  
 **this**.setRight(**this**.getRight().getRight());  
 **this**.setLeft(newLeft);  
}

至于另外一种双旋转的情况：右左。

道理类似，就不再演示了，代码都已经写在上面了。

## 七、多路查找树

B树中所有的叶子节点都必须在同一层

二节点：

有2个子节点的节点，就叫二节点。

B树中，二节点要么有2个子节点，要么2个子节点都为空。

前面所学的二叉树中的节点，都是二节点。每个二节点可以有2个子节点，只能存一个权value。

三节点：

有3个子节点的节点，就叫三节点。

B树中，三节点要么有3个子节点，要么3个子节点都为空。

每个三节点可以有3个子节点，最多存2个权value。

B树学的挺蒙蔽，暂时跳过。

# 第5章 哈希表