前言
一、冒泡排序
0.冒泡排序的实现
1.第一趟排序
2.第二趟排序
3. 代码简化
4. 冒泡排序优化
二、选择排序
零. 选择排序介绍和稳定性说明
一、第一趟排序
二、第二趟排序
三、代码简化
三、插入排序
零、插入排序介绍
一、第一趟排序
二、第二趟排序
三、简化代码
四、快速排序
零、快速排序的介绍
一、第一趟快速排序
二、递归分析与代码实现
五、归并排序
零、归并排序的介绍
一、演算归并排序过程
1.1 第一步
1.2 第二步
1.3 第三步
二、归并排序前提分析(分治法)
三、归并代码实现
六、希尔排序
一、希尔排序介绍
二、希尔排序体验
三、希尔排序代码实现
七、堆排序
一、堆排序介绍
二、堆排序体验
三、堆排序代码实现
八、基数排序（桶排序）
一、基数排序(桶排序)介绍
二、基数排序体验
2.1第一趟分配与回收
2.2第二趟分配与回收
2.3第三趟分配与回收
2.4第四趟分配与回收
三、基数排序代码编写
3.1求出数组最大的值
3.2代码实现

四、桶排序（基数排序）总结
九、递归
递归介绍
一、求和
二、数组内部的最大值
三、冒泡排序递归写法
四、斐波那契数列
五、汉诺塔算法
六、总结
十、链表
一、回顾与知新
1.1回顾数组
2.2链表说明
二、Java实现链表
2.1创建链表(增加节点)
2.2遍历链表
2.3插入节点
2.4获取链表的长度
2.5删除节点
2.6 对链表进行排序
2.7找到链表中倒数第k个节点
2.8查询链表的中间节点
2.9通过递归从尾到头输出单链表
2.10反转链表
三、链表总结
十一、栈
一、数据结构【栈】就是这么简单
1.1数据结构【栈】介绍
二、数据结构【栈】 代码实现
2.1进栈
2.2遍历栈
2.3判断该栈是否为空
2.4出栈
2.5清空栈
十二、队列
一、数据结构【队列】就是这么简单
1.1数据结构【队列】 代码实现
1.2初始化队列
1.3判断队列是否满了
1.4入队
1.5遍历
1.6判断该队列是否为空
1.7出队
十三、二叉树
一、二叉树就是这么简单
1.1树的一些概念
1.2静态创建二叉树
1.3遍历二叉树
二、动态创建二叉树

2.1动态创建二叉树体验
2.2代码实现
三、查询二叉查找树相关
3.1查询树的深度
3.1查询树的最大值

前言

这个文档的内容**纯手打**，如果想要看更多的干货文章，关注我的公众号：**Java3y**。有更多的原创技术文章和干货！

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！公众号有我的**联系方式**

- 🔥**Java精美脑图**
- 🔥**Java学习路线**
- 🔥**开发常用工具**
- 🔥**更多精美电子书、文档**

在公众号下回复「**888**」即可获取！！

想要说明的是，排序算法/数据结构的代码可能不是最优解，代码的实现都是以比较容易理解的方式去写的。几乎每句代码都有对应的注释，应该是能看懂的。

这份PDF对应的**源码**：https://github.com/ZhongFuCheng3y/JavaArithmetic

- 八大排序对应的代码位置：`sort.*`
- 链表的代码位置：`datastructure.LinkedList_old.LinkListText`
- 栈的代码位置：`datastructure.stack_old.StackTest`
- 队列的代码位置：`datastructure.queue_old.QueueTest`
- 二叉树的代码位置：`datastructure.tree_old.TreeTest`

学习不能盲目，跟着我，会让你事半功倍

文档允许随意传播，但不能修改任何内容。

电子书的整理也是挺不容易，如果你觉得有帮助，想要打赏作者，那么可以通过这个收款码打赏我，**金额不重要，心意最重要**。主要是我可以通过这个打赏情况来预计大家对这本电子书的评价，嘻嘻

推荐使用微信支付
3y (**成)
微信支付
支付宝 ALIPAY
支付就用支付宝

一、冒泡排序

0.冒泡排序的实现

来源百度百科：

> 冒泡排序（Bubble Sort，台湾译为：泡沫排序或气泡排序）是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端，故名。

算法描述：

1. **`i`从0开始，`i`与`i+1`比较，如果`i>i+1`，那么就互换**
2. **`i`不断增加，直到`i<n-1`（n是数组元素的个数，`n-1`是数组已经最后一个元素），一趟下来，可以让数组元素中最大值排在数组的最后面**

从最简单开始，首先我们创建一个数组，该数组有5位数字：

```
int[] arrays = {2, 5, 1, 3, 4};
```

1.第一趟排序

下面我们根据算法的描述来进行代码验算**(第一趟排序)**：

```
//使用临时变量，让两个数互换
int temp;

//第一位和第二位比
if (arrays[0] > arrays[1]) {
  //交换
  temp = arrays[0];
  arrays[0] = arrays[1];
  arrays[1] = temp;
}
```

```
//第二位和第三位比
if (arrays[1] > arrays[2]) {
  temp = arrays[1];
  arrays[1] = arrays[2];
  arrays[2] = temp;
}

//第三位和第四位比
if (arrays[2] > arrays[3]) {
  temp = arrays[2];
  arrays[2] = arrays[3];
  arrays[3] = temp;
}

//第四位和第五位比
if (arrays[3] > arrays[4]) {
  temp = arrays[3];
  arrays[3] = arrays[4];
  arrays[4] = temp;
}
```

如果前一位的数比后一位的数要大，那么就交换，直到将数组的所有元素都比较了一遍！

经过我们第一趟比较，我们可以发现：**最大的值就在数组的末尾了！**

```
System.out.println(arrays);   arrays: {2, 1, 3, 4, 5}
```

2.第二趟排序

第二趟排序跟第一趟排序一样，也是用前一位与后一位比较，如果前一位比后一位要大，那就交换。值得注意的是：**并不需要与最后一位比较了，因为在第一趟排序完了，最后一位已经是最大的数了**。同理，**我们第二趟排序完了之后，倒数第二位也是第二大的数了。**

第二趟排序的代码如下：

```
//第一位和第二位比
if (arrays[0] > arrays[1]) {
  //交换
  temp = arrays[0];
  arrays[0] = arrays[1];
  arrays[1] = temp;
}

//第二位和第三位比
if (arrays[1] > arrays[2]) {
```

```
  temp = arrays[1];
  arrays[1] = arrays[2];
  arrays[2] = temp;
}

//第三位和第四位比
if (arrays[2] > arrays[3]) {
  temp = arrays[2];
  arrays[2] = arrays[3];
  arrays[3] = temp;
}

//第四位不需要和第五位比了，因为在第一趟排序结束后，第五位是最大的了。
```

结果：**我们的第二大数已经排在了倒数第二位了**

```
67
68        System.out.println(arrays);   arrays: {1, 2, 3, 4, 5}
69
70
```

3. 代码简化

值得说明的是：上面的结果**看起来**已经是排序好的了，其实是我在测试时数据还不足够乱，**如果数据足够乱的话，是需要4(n-1)趟排序的**！

但我们从上面的代码就可以发现：**第一趟和第二趟的比较、交换代码都是重复的，并且我们的比较都是写死的(0,1,2,3,4)，并不通用**！

我们的数组有5位数字

- 第一趟需要比较**4次**
- 第二趟需要比较**3次**

我们可以根据上面规律推断出：

- 第三趟需要比较**2次**
- 第四躺需要比较**1次**

再从上面的规律可以总结出：**5位数的数组需要4躺排序的，每躺排序之后次数减1(因为前一趟已经把前一趟数的最大值确定下来了)！**

于是我们可以**根据for循环和变量将上面的代码进行简化**：

```
int temp;

//外层循环是排序的趟数
for (int i = 0; i < arrays.length - 1 ; i++) {

  //内层循环是当前趟数需要比较的次数
  for (int j = 0; j < arrays.length - i - 1; j++) {
```

```
    //前一位与后一位与前一位比较，如果前一位比后一位要大，那么交换
    if (arrays[j] > arrays[j + 1]) {
      temp = arrays[j];
      arrays[j] = arrays[j + 1];
      arrays[j + 1] = temp;
    }
  }
}
```

4. 冒泡排序优化

从上面的例子我们可以看出来，如果数据足够乱的情况下是需要经过4躺比较才能将数组完整排好序。**但是我们在第二躺比较后就已经得到排好序的数组了。**

但是，我们的程序在第二趟排序后仍会执行第三趟、第四趟排序。这是没有必要的，因此我们可以对其进行优化一下：

- **如果在某躺排序中没有发生交换位置，那么我们可以认为该数组已经排好序了。**
  - 这也不难理解，因为我们**每趟排序的目的就是将当前趟最大的数置换到对应的位置上，没有发生置换说明就已经排好序了。**

代码如下：

```
//装载临时变量
int temp;

//记录是否发生了置换， 0 表示没有发生置换、 1 表示发生了置换
int isChange;

//外层循环是排序的趟数
for (int i = 0; i < arrays.length - 1; i++) {

  //每比较一趟就重新初始化为0
  isChange = 0;

  //内层循环是当前趟数需要比较的次数
  for (int j = 0; j < arrays.length - i - 1; j++) {

    //前一位与后一位与前一位比较，如果前一位比后一位要大，那么交换
    if (arrays[j] > arrays[j + 1]) {
      temp = arrays[j];
      arrays[j] = arrays[j + 1];
      arrays[j + 1] = temp;

      //如果进到这里面了，说明发生置换了
      isChange = 1;
```

```
    }
  }
  //如果比较完一趟没有发生置换，那么说明已经排好序了，不需要再执行下去了
  if (isChange == 0) {
    break;
  }
}
```

效果如下：

```
        //记录执行了多少趟
        int num = 0;  num: 2


        //外层循环是排序的趟数
        for (int i = 0; i < arrays.length; i++) {

            //每比较一趟就重新初始化为0
            isChange = 0;

            //内层循环是当前趟数需要比较的次数
            for (int j = 0; j < arrays.length - i - 1; j++) {

                //前一位与后一位与前一位比较，如果前一位比后一位要大，那么交换
                if (arrays[j] > arrays[j + 1]) {
                    temp = arrays[j];
                    arrays[j] = arrays[j + 1];
                    arrays[j + 1] = temp;


                    //如果进到这里面了，说明发生置换了
                    isChange = 1;

                }
            }
            //如果比较完一趟没有发生置换，那么说明已经排好序了，不需要再执行下去了
            if (isChange == 0) {
                break;
            }
            num++;

        }

        System.out.println(num);  num: 2
        System.out.println(arrays);  arrays: {1, 2, 3, 4, 5}
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

二、选择排序

零. 选择排序介绍和稳定性说明

来源百度百科：

> 选择排序(Selection sort)是一种简单直观的排序算法。它的工作原理是每一次从待排序的数据元素中选出最小(或最大)的一个元素，存放在序列的起始(末尾)位置，直到全部待排序的数据元素排完。选择排序是不稳定的排序方法（比如序列[5， 5， 3]第一次就将第一个[5]与[3]交换，导致第一个5挪动到第二个5后面）。

上面提到了选择排序是**不稳定的排序方法**，那我们的冒泡排序是不是稳定的排序方法呢？稳定的意思指的是什么呢？

判断某排序算法是否稳定，我们可以简单理解成：排序前2个相等的数其在序列的**前后位置顺序和排序后它们两个的前后位置顺序相同**

- 如果相同，则是稳定的排序方法。
- 如果不相同，则是不稳定的排序方法

如果排序前的数组是`[3,3,1]`，假定我们使用选择排序的话，那第一趟排序后结果就是`[1,3,3]`。这个数组有两个相同的值，它俩在`array[0]`和`array[1]`，结果经过排序，`array[0]`的跑到了`array[2]`上了。

那么这就导致：**2个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序不相同**，因此，我们就说它是不稳定的

再回到上面的问题，上一篇说讲的冒泡排序是稳定的，主要原因是：**俩俩比较的时候，没有对相等的数据进行交换(因为没必要)**。因此它不存在2个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序不相同。

那么稳定排序的好处是什么？

- 参考知乎回答@独行侠的回答：

> 如果我们只对一串数字排序，那么稳定与否确实不重要，因为一串数字的属性是单一的，就是数字值的大小。但是排序的元素往往不只有一个属性，例如我们对一群人按年龄排序，但是人除了年龄属性还有身高体重属性，在年龄相同时如果不想破坏原先身高体重的次序，就必须用稳定排序算法.

很清晰的指出，**只有当在“二次”排序时不想破坏原先次序，稳定性才有意义**

一、第一趟排序

它的工作原理是每一次从待排序的数据元素中选出最小(或最大)的一个元素，存放在序列的起始(末尾)位置，直到全部待排序的数据元素排完

首先，我们创建数组，找到它最大的值(这就很简单了):

```
int[] arrays = {2, 3, 1, 4, 3, 5, 1, 6, 1, 2, 3, 7, 2, 3};

//假定max是最大的
int max = 0;


for (int i = 0; i < arrays.length; i++) {
  if (arrays[i] > max) {
    max = arrays[i];
  }
}
```

效果如下:

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

    int[] arrays = {2, 3, 1, 4, 3, 5, 1, 6, 1, 2, 3, 7, 2, 3};  arrays: {2, 3, 1, 4, 3,

    //假定max是最大的
    int max = 0;  max: 7

    for (int array : arrays) {  arrays: {2, 3, 1, 4, 3, 5, 1, 6, 1, 2, + 4 more}
        if (array > max) {
            max = array;
        }
    }
    System.out.println(max);  max: 7
```

随后这个最大的数和数组末尾的数进行交换:

```
//使用临时变量，让两个数互换
int temp;
temp = arrays[11];
arrays[11] = arrays[13];
arrays[13] = temp;
```

那么经过第一趟排序，我们的最大值已经到了数组的末尾了。

二、第二趟排序

再次从数组获取最大的数(除了已经排好的那个）：

```
int max2 = 0;
for (int i = 0; i < (arrays.length - 1); i++) {
  if (arrays[i] > max2) {
    max2 = arrays[i];
  }
}

System.out.println(max2);
```

效果如下：

再将获取到的最大值与数组倒数第二位交换：

```
temp = arrays[7];
arrays[7] = arrays[12];
arrays[12] = temp;
```

经过第二次排序，已经能够将数组最大两个数进行排序了

三、代码简化

从前两趟排序其实我们就可以摸出规律了：

- 一个数组是需要`n-1`趟排序的(因为直到剩下一个元素时，才不需要找最大值)
- **每交换1次，再次找最大值时就将范围缩小1**
- 查询当前趟数最大值实际上不用知道最大值是多少(上面我查出最大值，还要我手动数它的角标)，**知道它的数组角标即可**，交换也是根据角标来进行交换

第一趟：遍历数组14个数，获取最大值，将最大值放到数组的末尾`[13]`
第二趟：遍历数组13个数，获取最大值，将最大值放到数组倒数第二位`[12]`

....

数组有14个数，需要13趟排序。

```
//记录当前趟数的最大值的角标
int pos ;

//交换的变量
int temp;

//外层循环控制需要排序的趟数
for (int i = 0; i < arrays.length - 1; i++) {
  //新的趟数、将角标重新赋值为0
  pos = 0;

  //内层循环控制遍历数组的个数并得到最大数的角标
  for (int j = 0; j < arrays.length - i; j++) {
    if (arrays[j] > arrays[pos]) {
      pos = j;
    }
  }
```

```
    //交换
    temp = arrays[pos];
    arrays[pos] = arrays[arrays.length - 1 - i];
    arrays[arrays.length - 1 - i] = temp;
}

System.out.println("公众号Java3y" + arrays);
```

效果如下：

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

三、插入排序

零、插入排序介绍

来源百度百科：

> 插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为O(n^2)。是稳定的排序方法。

将一个数据插入到**已经排好序的有序数据**中

- 将要排序的是一个乱的数组`int[] arrays = {3, 2, 1, 3, 3};`
- 在未知道数组元素的情况下，我们**只能把数组的第一个元素作为已经排好序的有序数据**，也就是说，把`{3}`看成是已经排好序的有序数据

一、第一趟排序

用数组的第二个数与第一个数(**看成是已有序的数据**)比较

- 如果比第一个数大，那就不管他
- 如果比第一个数小，将第一个数往后退一步，将第二个数插入第一个数去

```
int temp;
if (arrays[1] > arrays[0]) {
  //如果第二个数比第一个数大，直接跟上

} else {
  //如果第二个数比第一个数小，将第一个数后退一个位置(将第二个数插进去)
  temp = arrays[1];
  arrays[1] = arrays[0];
  arrays[0] = temp;

}

System.out.println("公众号Java3y" + arrays);
```

效果如下：

```
int[] arrays = {3, 2, 1, 3, 3};  arrays: {2, 3, 1, 3, 3}



//第一趟排序--------------------
int temp;
if (arrays[1] > arrays[0]) {
    //如果第二个数比第一个数大，直接跟上

} else {
    //如果第二个数比第一个数小，将第一个数后退一个位置(将第二个数插进去)
    temp = arrays[1];
    arrays[1] = arrays[0];
    arrays[0] = temp;

}

System.out.println("公众号Java3y" + arrays);  arrays: {2, 3, 1, 3, 3}
```

二、第二趟排序

用数组的第三个数与已是有序的数据`{2,3}`(刚才在第一趟排的)比较

- 如果比第二位的数大，那就不管它
- 如果比第二位的数小，那就将第二的位置退一个位置，让第三个数和第一位比较
  - 如果第三个数比第一位大，那么将第三个数插入到第二的位置上
  - 如果第三个数比第一位小，那么将第一位后退一步，将第三个数插入到第一的位置上

```
//第二趟排序--------------------

if (arrays[2] > arrays[1]) {
  //如果第三个数比第二个数大，直接跟上

} else {
  //如果第三个数比第二个数小，将第二个数往后退一个位置，让第三个数跟第一个数比
  temp = arrays[2];
```

```
    arrays[2] = arrays[1];

    //如果第三个数比第一个大，那就插入到第二个数中
    if (temp > arrays[0]) {
      arrays[1] = temp;
    } else {

      //如果第三个数比第一个小，将第三个数插入到第一个数前面
      int swapTemp = arrays[0];
      arrays[0] = temp;
      arrays[1] = swapTemp;

    }

  }
  System.out.println("公众号Java3y" + arrays);
```

效果如下：

```
if (arrays[2] > arrays[1]) {
    //如果第三个数比第二个数大，直接跟上

} else {
    //如果第三个数比第二个数小，将第二个数往后退一个位置，让第三个数跟第一个数比
    temp = arrays[2];
    arrays[2] = arrays[1];

    //如果第三个数比第一个大，那就插入到第二个数中
    if (temp > arrays[0]) {
        arrays[1] = temp;
    } else {

        //如果第三个数比第一个小，将第三个数插入到第一个数前面
        int swapTemp = arrays[0];
        arrays[0] = temp;
        arrays[1] = swapTemp;

    }

}
System.out.println("公众号Java3y" + arrays);   arrays: {1, 2, 3, 3, 3}
```

三、简化代码

从前两趟排序我们可以摸出的规律：

- 首先将已排序的数据看成一个整体
- 一个数组是需要`n-1`趟排序的，总是用后一位跟`已排序的数据`比较(第一趟：第二位跟`已排序的数据`比，第二趟：第三位跟`已排序的数据`比)
- 用第三位和`已排序的数据`比，实际上就是让第三位数跟两个数比较，只不过这两个数是已经**排好序**的而已。而正是因为它排好序的，我们可以**使用一个循环就可以将我们比较的数据插入进去**

```java
for (int i = 1; i < arrays.length; i++) {

  temp = arrays[i];

  //如果前一位(已排序的数据)比当前数据要大，那么就进入循环比较[参考第二趟排序]
  int j = i - 1;

  while (j >= 0 && arrays[j] > temp) {

    //往后退一个位置，让当前数据与之前前位进行比较
    arrays[j + 1] = arrays[j];

    //不断往前，直到退出循环
    j--;

  }
  //退出了循环说明找到了合适的位置了，将当前数据插入合适的位置中
  arrays[j + 1] = temp;

}
System.out.println("公众号Java3y" + arrays);
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

四、快速排序

零、快速排序的介绍

来源百度百科：

> 快速排序由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以**递归**进行，以此达到整个数据变成有序序列。

快速排序是面试出现的可能性比较高的，也是经常会用到的一种排序，应该重点掌握。

前面一个章节已经讲了递归了，那么现在来看快速排序就非常简单了。（递归可以关注我的公众号（微信搜**Java3y**）阅读。

一、第一趟快速排序

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小

百度百科的话并没有说到重点，更简单的理解是这样的：**在数组中找一个支点(任意),经过一趟排序后，支点左边的数都要比支点小，支点右边的数都要比支点大！**

现在我们有一个数组：`int arr[]={1,4,5,67,2,7,8,6,9,44};`

经过一趟排序之后，如果我选择数组中间的数作为支点：7(任意的)，那么第一趟排序后的结果是这样的：`{1,4,5,6,2,7,8,67,9,44}`

那么就实现了**支点左边的数比支点小，支点右边的数比支点大**

二、递归分析与代码实现

现在我们的数组是这样的：`{1,4,5,6,2,7,8,67,9,44}`，既然我们**比7小的在左边，比7大的在右边**，那么我们只要将”左边“的排好顺序，又将”右边“的排好序，那整个数组是不是就有序了？想一想，是不是？

又回顾一下递归：”左边“的排好顺序，”右边“的排好序，跟我们**第一趟排序的做法**是不是一致的？

只不过是参数不一样：第一趟排序是任选了一个支点，比支点小的在左边，比支点大的在右边。那么，我们想要”左边“的排好顺序，只要**在”左边“部分找一个支点**，比支点小的在左边，比支点大的在右边。

.....

在数组中使用递归依照我的惯性，往往定义两个变量：`L`和`R`，`L`指向第一个数组元素，`R`指向在最后一个数组元素

递归出口也很容易找到：如果**数组只有一个元素时**，那么就不用排序了

所以，我们可以写出这样的代码：

```
    public static void main(String[] args) {
        int[] arr = {1, 4, 5, 67, 2, 7, 8, 6, 9, 44};

        quickSort(arr, 0, 9);

        System.out.println("Java3y   " + arr);


    }

    /**
     * 快速排序
     *
     * @param arr
     * @param L   指向数组第一个元素
     * @param R   指向数组最后一个元素
     */
    public static void quickSort(int[] arr, int L, int R) {
        int i = L;
        int j = R;

        //支点
        int pivot = arr[(L + R) / 2];

        //左右两端进行扫描，只要两端还没有交替，就一直扫描
        while (i <= j) {

            //寻找直到比支点大的数
            while (pivot > arr[i])
                i++;

            //寻找直到比支点小的数
            while (pivot < arr[j])
                j--;

            //此时已经分别找到了比支点小的数(右边)、比支点大的数(左边)，它们进行交换
            if (i <= j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
                i++;
                j--;
            }
        }
        //上面一个while保证了第一趟排序支点的左边比支点小，支点的右边比支点大了。
```

```java
        //“左边”再做排序，直到左边剩下一个数(递归出口)
        if (L < j)
            quickSort(arr, L, j);

        //“右边”再做排序，直到右边剩下一个数(递归出口)
        if (i < R)
            quickSort(arr, i, R);
    }
```

效果如下：

```java
    int[] arr = {1, 4, 5, 67, 2, 7, 8, 6, 9, 44};  arr: {1, 2, 4, 5, 6, 7, 8, 9, 44, 67}

    quickSort(arr, L: 0, R: 9);

    System.out.println("Java3y   " + arr);  arr: {1, 2, 4, 5, 6, 7, 8, 9, 44, 67}


}

/**
 * 快速排序
 *
 * @param arr
 * @param L    指向数组第一个元素
 * @param R    指向数组最后一个元素
 */
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

五、归并排序

零、归并排序的介绍

来源百度百科：

> 归并排序（MERGE-SORT）是建立在归并操作上的一种有效的排序算法,该算法是采用**分治法**（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。**若将两个有序表合并成一个有序表，称为二路归并。**

过程描述：

> 归并过程为：比较a[i]和b[j]的大小，若a[i]≤b[j]，则将第一个有序表中的元素a[i]复制到r[k]中，并令i和k分别加上1；否则将第二个有序表中的元素b[j]复制到r[k]中，并令j和k分别加上1，如此循环下去，直到其中一个有序表取完，然后再将另一个有序表中剩余的元素复制到r中从下标k到下标t的单元。归并排序的算法我们通常用递归实现，先把待排序区间[s,t]以中点二分，接着把左边子区间排序，再把右边子区间排序，最后把左区间和右区间用一次归并操作合并成有序的区间[s,t]。

原理：

> 归并操作的工作原理如下：

> 第一步：申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

> 第二步：设定两个指针，最初位置分别为两个已经排序序列的起始位置

> 第三步：比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

> 重复步骤3直到某一指针超出序列尾

> 将另一序列剩下的所有元素直接复制到合并序列尾

下面我就来做个小小的总结：

- 将**两个**已**排好序**的数组**合并**成**一个有序**的数组,称之为归并排序
- 步骤：遍历两个数组，比较它们的值。谁比较小，谁先放入大数组中，直到数组遍历完成

一、演算归并排序过程

现在我有两个已经排好顺序的数组：`int[] arr1 = {2, 7, 8}`和`int[] arr2 = {1, 4, 9}`，我还有一个大数组来装载它们`int[] arr = new int[6];`

1.1 第一步

那么，我将两个数组的值进行比较，**谁的值比较小，谁就放入大数组中**！

首先，拿出`arr1[0]`和`arr2[0]`进行比较，显然是`arr2[0]`比较小，因此将`arr2[0]`放入大数组中，**同时`arr2`指针往后一格**

所以，现在目前为止`arr = {1}`

1.2 第二步

随后，拿`arr1[0]`和`arr2[1]`进行比较，显然是`arr1[0]`比较小，将`arr1[0]`放入大数组中，**同时`arr1`指针往后一格**

所以，现在目前为止`arr = {1,2}`

1.3 第三步

随后，拿`arr1[1]`和`arr2[1]`进行比较，显然是`arr2[1]`比较小，将`arr2[1]`放入大数组中，**同时`arr2`指针往后一格**

所以，现在目前为止`arr = {1,2,4}`

.....

遍历到最后，我们会将**两个**已排好序的数组**变成一个**已排好序的数组`arr = {1,2,4,7,8,9}`

二、归并排序前提分析(分治法)

从上面的演算我们就直到，归并排序的**前提是需要两个已经排好顺序的数组**，那**往往不会有**两个已经排好顺序的数组给我们的呀**(一般是杂乱无章的一个数组)**，那这个算法是不是很鸡肋的呢？？

其实并不是的，首先**假设**题目给出的数组是这样子的：`int[] arr = {2, 7, 8, 1, 4, 9};`

当我们要做归并的时候就以`arr[3]`也就元素为1的那个地方分开。是然后用一个`指针L`指向`arr[0]`，一个`指针M`指向`arr[3]`，用一个`指针R`指向`arr[5]`(数组最后一位)。有了指针的帮助，我们就**可以将这个数组切割成是两个有序的数组了**（操作的方式就可以和上面一样了）

可是上面说了，一般给出的是**杂乱无章**的一个数组，现在还是达不到要求。比如给出的是这样一个数组：`int[] arrays = {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, 8};`

此时，我们就得用到**分治**的思想了：

- 那么我们也可以这样想将`int[] arr = {2, 7, 8, 1, 4, 9};`数组分隔成一份一份的，`arr[0]`它是一个有序的"数组",`arr[1]`它也是一个有序的"数组",利用指针(L,M,R)又可以**像**操作两个数组一样进行排序。最终合成`{2,7}`.....再**不断拆分合并**，最后又回到了我们的`arr = {1,2,4,7,8,9}`，因此**归并排序是可以排序杂乱无章的数组的**

这就是我们的分治法--->**将一个大问题分成很多个小问题进行解决，最后重新组合起来**

三、归并代码实现

实现步骤：

1. 拆分
2. 合并

.....

```
    public static void main(String[] args) {
        int[] arrays = {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, 8};
        mergeSort(arrays, 0, arrays.length - 1);

        System.out.println("公众号：Java3y" + arrays);
    }

    /**
     * 归并排序
     *
     * @param arrays
     * @param L      指向数组第一个元素
     * @param R      指向数组最后一个元素
     */
    public static void mergeSort(int[] arrays, int L, int R) {

        //如果只有一个元素，那就不用排序了
        if (L == R) {
            return;
        } else {

            //取中间的数，进行拆分
            int M = (L + R) / 2;

            //左边的数不断进行拆分
            mergeSort(arrays, L, M);

            //右边的数不断进行拆分
            mergeSort(arrays, M + 1, R);

            //合并
            merge(arrays, L, M + 1, R);
```

```
    }
}


/**
 * 合并数组
 *
 * @param arrays
 * @param L      指向数组第一个元素
 * @param M      指向数组分隔的元素
 * @param R      指向数组最后的元素
 */
public static void merge(int[] arrays, int L, int M, int R) {

    //左边的数组的大小
    int[] leftArray = new int[M - L];

    //右边的数组大小
    int[] rightArray = new int[R - M + 1];

    //往这两个数组填充数据
    for (int i = L; i < M; i++) {
        leftArray[i - L] = arrays[i];
    }
    for (int i = M; i <= R; i++) {
        rightArray[i - M] = arrays[i];
    }

    int i = 0, j = 0;
    // arrays数组的第一个元素
    int  k = L;


    //比较这两个数组的值，哪个小，就往数组上放
    while (i < leftArray.length && j < rightArray.length) {

        //谁比较小，谁将元素放入大数组中,移动指针，继续比较下一个
        if (leftArray[i] < rightArray[j]) {
            arrays[k] = leftArray[i];

            i++;
            k++;
        } else {
            arrays[k] = rightArray[j];
            j++;
            k++;
        }
    }
```

```
        //如果左边的数组还没比较完，右边的数都已经完了，那么将左边的数抄到大数组中(剩下的都是大数字)
        while (i < leftArray.length) {
            arrays[k] = leftArray[i];
            i++;
            k++;
        }
        //如果右边的数组还没比较完，左边的数都已经完了，那么将右边的数抄到大数组中(剩下的都是大数字)
        while (j < rightArray.length) {
            arrays[k] = rightArray[j];
            k++;
            j++;
        }
    }
```

我debug了一下第一次的时候，就可以更容易理解了：

将大数组的前两个进行拆分，然后用数组装载起来

```
    /**
     * 合并数组
     *
     * @param arrays
     * @param L       指向数组第一个元素
     * @param M       指向数组分隔的元素
     * @param R       指向数组最后的元素
     */
    public static void merge(int[] arrays, int L, int M, int R) {  arrays: {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, +

        //左边的数组的大小
        int[] leftArray = new int[M - L];  leftArray: {9}

        //右边的数组大小
        int[] rightArray = new int[R - M + 1];  rightArray: {2}

        //往这两个数组填充数据
        for (int i = L; i < M; i++) {
            leftArray[i - L] = arrays[i];  L: 0
        }
        for (int i = M; i <= R; i++) {  R: 1
            rightArray[i - M] = arrays[i];  M: 1  arrays: {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, + 1 more}
        }

        System.out.println("公众号: Java3y" + leftArray + rightArray);  leftArray: {9}  rightArray: {2}
```

比较小数组的元素哪个小，哪个小就先放入大数组中

```
    //如果左边的数组还没比较完，右边的数都已经完了，那么将左边的数抄到大数组中(剩下的都是大数字)
    while (i < leftArray.length) {
        arrays[k] = leftArray[i];  leftArray: {9}

        i++;  i: 1
        k++;
    }
    //如果右边的数组还没比较完，左边的数都已经完了，那么将右边的数抄到大数组中(剩下的都是大数字)
    while (j < rightArray.length) {
        arrays[k] = rightArray[j];  rightArray: {2}

        k++;  k: 2
        j++;  j: 1
    }
    System.out.println("公众号: Java3y" + arrays);  arrays: {2, 9, 5, 1, 3, 2, 9, 5, 2, 1, + 1 more}
```

上面的两个步骤不断循环，最后得出有序的数组：

```
    int[] arrays = {9, 2, 5, 1, 3, 2, 9, 5, 2, 1, 8};  arrays: {1, 1, 2, 2, 2, 3, 5, 5, 8, 9, + 1 more}
    mergeSort(arrays, L:0, R:arrays.length - 1);

    System.out.println("公众号：Java3y" + arrays);  arrays: {1, 1, 2, 2, 2, 3, 5, 5, 8, 9, + 1 more}
}
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

六、希尔排序

一、希尔排序介绍

来源百度百科：

> 希尔排序(Shell's Sort)是插入排序的一种又称“缩小增量排序”（Diminishing Increment Sort），是直接插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因D.L.Shell于1959年提出而得名。

从上面我们很容易看出来，它是**插入排序的高级版**

回顾一下插入排序：

- 将数据**插入到已有序**的数列中
  - 排序前：将**每个**元素看成有序的数列
  - 第一趟排序后：得到一个有序数列，其大小为2
  - 第二趟排序后：得到一个有序数列，其大小为3
  - 第三趟排序后：得到一个有序数列，其大小为4
  - .....每一趟插入排序，都可以将一个无序值插入一个有序数列，**直至全部值有序**
- 如果给出的数组**足够乱**的话，那么插入排序所耗费的时间是O(n^2)

既然希尔排序是插入排序的高级版，那它做了哪些优化呢？？让我们来看看：

- 希尔排序在排序前：将一个序列**分成了好几个序列**
- 在第一趟排序时：**将这几个序列做插入排序**。排序后，**部分**较大的数字往后靠，**部分**较小的数字往前靠
- 在第二趟排序时：将这个序列又分了好几个序列做插入排序(但比第一次分的数要少)。注意:如果第一次分5个，第二次可能就2个了)。排序后，**部分**较大的数字往后靠，**部分**较小的数字往前靠
- .....
- 在第n趟排序时：将这个序列又分了好几个序列(直到剩下一个序列)，从宏观上看，此序列就基本是有序的了。这时就用简单插入排序将数列**直至已序**

从直观上看希尔排序：

- **就是把数列进行分组(不停使用插入排序)，直至从宏观上看起来有序，最后插入排序起来就容易了(无须多次移位或交换)。**

那么，上面那里说了将一个序列分成好几个序列，那么到底怎么分呢？比如有10个元素的序列，分成几个才合适？每次缩减又是多少呢？

从专业的角度上讲，将一个序列分成好几个序列，用一个数来表示：那个数称为**增量**。显然的是，增量是不断递减的(直到增量为1)

往往的：如果一个数列有10个元素，我们第一趟的增量是5，第二趟的增量是2，第三趟的增量是1。如果一个数列有18个元素，我们第一趟的增量是9，第二趟的增量是4，第三趟的增量是2，第四趟的增量是1

很明显我们可以用一个序列来表示增量：`{n/2,(n/2)/2...1}`，**每次增量都**`/2`

二、希尔排序体验

现在我们有一个数组，该数组有6个元素

```
int[] arrays = {2, 5, 1, 3, 4, 6};
```

排序前：

- 将该数组看成三个（arrays.length/2)数组，分别是:`{2,3}`,`{5,4}`,`{1,6}`

第一趟排序：

- 对三个数组分别进行插入排序，因此我们三个数组得到的结果为`{2,3}`,`{4,5}`,`{1,6}`
    - 此时数组是这样子的：`{2, 4, 1, 3, 5, 6}`

第二趟排序：

- 增量减少了，上面增量是3，此时增量应该为1了，因此把`{2, 4, 1, 3, 5, 6}`看成一个数组(**从宏观上是有序的了**)，对其进行插入排序，**直至有序**

可能我举的例子不够好(没看到很好的效果)，我们来看看网上的图片，加深一下希尔排序的过程：

原始数组 以下数据元素颜色相同为一组

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]

对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，
然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]

https://www.cnblogs.com/chengxiao/

对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。
再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下

经过上面的“宏观调控”，整个数组的有序化程度成果喜人。
此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

三、希尔排序代码实现

```
public static void shellSort(int[] arrays) {


  //增量每次都/2
  for (int step = arrays.length / 2; step > 0; step /= 2) {

    //从增量那组开始进行插入排序，直至完毕
    for (int i = step; i < arrays.length; i++) {

      int j = i;
```

```
        int temp = arrays[j];

        // j - step 就是代表与它同组隔壁的元素
        while (j - step >= 0 && arrays[j - step] > temp) {
          arrays[j] = arrays[j - step];
          j = j - step;
        }
        arrays[j] = temp;
      }
    }


}
```

我们发现希尔排序代码其实非常简单(相比对堆排序)，理解起来也不难，**就用增量来将数组进行分隔，直到增量为1。底层干的还是插入排序干的活～**

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

七、堆排序

一、堆排序介绍

来源百度百科：

> 堆排序(Heapsort)是指利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种。可以利用数组的特点快速定位指定索引的元素。堆分为大根堆和小根堆，是**完全二叉树**。

前面我已经有二叉树入门的文章了，当时讲解的是二叉查找树，那上面所说的完全二叉树是怎么样的一种二叉树呢？？还有满二叉树又是怎么的一种二叉树呢？？甚至还有完满二叉树？？

- 完全二叉树： 除了最后一层之外的其他每一层都被完全填充，并且所有结点都保持向**左对齐**。
- 满二叉树：除了叶子结点之外的每一个结点都有两个孩子，**每一层(当然包含最后一层)都被完全填充**。

- 完满二叉树：**除了叶子结点之外的每一个结点都有两个孩子结点。**

下面用图来说话：

- 完全二叉树(Complete Binary Tree)：

- 满二叉树(Full Binary Tree)：

- 完满二叉树(Perfect Binary Tree)：

参考资料：

- http://www.cnblogs.com/idorax/p/6441043.html

简单来说：**堆排序是将数据看成是完全二叉树、根据完全二叉树的特性来进行排序的一种算法**

- **最大堆要求节点的元素都要不小于其孩子，最小堆要求节点元素都不大于其左右孩子**
- **那么处于最大堆的根节点的元素一定是这个堆中的最大值**

这里我们讨论**最大堆：当前每个父节点都大于子节点**

完全二叉树有个特性：`左边子节点位置 = 当前父节点的两倍 + 1`，`右边子节点位置 = 当前父节点的两倍 + 2`

二、堆排序体验

现在我们有一个完全二叉树：左子树和右子树都符合最大堆-->`父>子`

但是我们会发现：根元素所在的数并不符合，明显的是：**1是小于7的**

我们就对其进行交换，交换完之后我们会发现：**右子树又不符合了～**

因为，右子树变成了这样：

最后，我们**将右子数的最大值也交换到右子树的根元素上**

于是我们第一次的建堆操作就完成了！

可以发现的是：**一次堆建立完之后，我们的最大值就在了堆的根节点上**

随后将堆顶最大值和数组最后的元素进行替换，我们就完成了一趟排序了。

接下来，剩下的数不断进行建堆，交换就可以完成我们的堆排序了

.....建堆，交换....建堆，交换...建堆，交换...建堆，交换..

三、堆排序代码实现

比较当前父节点是否大于子节点，如果大于就交换，直到一趟建堆完成～

```java
    /**
     * 建堆
     *
     * @param arrays           看作是完全二叉树
     * @param currentRootNode 当前父节点位置
     * @param size             节点总数
     */
    public static void heapify(int[] arrays, int currentRootNode, int size) {

        if (currentRootNode < size) {
            //左子树和右字数的位置
            int left = 2 * currentRootNode + 1;
            int right = 2 * currentRootNode + 2;

            //把当前父节点位置看成是最大的
            int max = currentRootNode;

            if (left < size) {
                //如果比当前根元素要大，记录它的位置
                if (arrays[max] < arrays[left]) {
                    max = left;
                }
            }
            if (right < size) {
                //如果比当前根元素要大，记录它的位置
                if (arrays[max] < arrays[right]) {
                    max = right;
                }
            }
            //如果最大的不是根元素位置，那么就交换
            if (max != currentRootNode) {
                int temp = arrays[max];
                arrays[max] = arrays[currentRootNode];
                arrays[currentRootNode] = temp;

                //继续比较，直到完成一次建堆
                heapify(arrays, max, size);
            }
        }
    }
```

值得注意的是：**在上面体验堆排序时，我们是左子树和右子数都是已经有`父>子`这么一个条件的了。**

- 显然，一个普通的数组并不能有这种条件(父>子)，因此，我们**往往是从数组最后一个元素来进行建堆**

```java
    /**
     * 完成一次建堆，最大值在堆的顶部(根节点)
     */
    public static void maxHeapify(int[] arrays, int size) {

    // 从数组的尾部开始，直到第一个元素(角标为0)
        for (int i = size - 1; i >= 0; i--) {
            heapify(arrays, i, size);
        }

    }
```

完成第一次建堆之后，其它下边的分叶子节点都在自己该在的位置了，我们会发现**最大值会在数组的首位**：

```
maxHeapify(arrays, arrays.length);
System.out.println("公众号: Java3y" + arrays);   arrays: {4321, 432, 234, 23, 6, 8, 4, 5, 7, 5, + 9 more}
```

然后让数组最后一位与当前堆顶(数组第一位)进行**交换**即可排序，不断调整位置即可，全部的代码如下：

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

        int[] arrays = {6, 3, 8, 5,2,-1,-5,-2,-6,345,7, 5, 1, 2, 23, 4321,
432, 3, 2, 34234, 2134, 1234, 5, 132423, 234, 4, 2, 4, 1, 5, 2, 5};

        // 完成一次建堆..
        maxHeapify(arrays, arrays.length - 1);
        int size = arrays.length - 1;

        for (int i = 0; i < arrays.length; i++) {
            //交换
            int temp = arrays[0];
            arrays[0] = arrays[(arrays.length - 1) - i];
            arrays[(arrays.length - 1) - i] = temp;

            // 调整位置
            heapify(arrays, 0, size);
            size--;
        }
        System.out.println("公众号: Java3y" + arrays);
    }

    /**
     * 完成一次建堆，最大值在堆的顶部(根节点)
```

```java
     */
    public static void maxHeapify(int[] arrays, int size) {
        for (int i = size - 1; i >= 0; i--) {
            heapify(arrays, i, size);
        }
    }

    /**
     * 建堆
     *
     * @param arrays           看作是完全二叉树
     * @param currentRootNode 当前父节点位置
     * @param size             节点总数
     */
    public static void heapify(int[] arrays, int currentRootNode, int size) {

        if (currentRootNode < size) {
            //左子树和右字数的位置
            int left = 2 * currentRootNode + 1;
            int right = 2 * currentRootNode + 2;

            //把当前父节点位置看成是最大的
            int max = currentRootNode;

            if (left < size) {
                //如果比当前根元素要大，记录它的位置
                if (arrays[max] < arrays[left]) {
                    max = left;
                }
            }
            if (right < size) {
                //如果比当前根元素要大，记录它的位置
                if (arrays[max] < arrays[right]) {
                    max = right;
                }
            }
            //如果最大的不是根元素位置，那么就交换
            if (max != currentRootNode) {
                int temp = arrays[max];
                arrays[max] = arrays[currentRootNode];
                arrays[currentRootNode] = temp;

                //继续比较，直到完成一次建堆
                heapify(arrays, max, size);
            }
        }
    }
```

效果如下：

```
            int temp = arrays[0];
            arrays[0] = arrays[(arrays.length -
            arrays[(arrays.length - 1) - i] = te

        }

        System.out.println("公众号: Java3y" + arr


    }

    /**
     * 完成一次建堆，最大值在堆的顶部(根节点)
     */
    public static void maxHeapify(int[] arrays,

        for (int i = size - 1; i >= 0; i--) {
            heapify(arrays, i, size);
        }
```

Main › main()

```
arrays = {int[19]@468}
    0 = 1
    1 = 1
    2 = 2
    3 = 2
    4 = 2
    5 = 3
    6 = 3
    7 = 4
    8 = 4
    9 = 5
    10 = 5
    11 = 5
    12 = 6
    13 = 7
    14 = 8
    15 = 23
    16 = 234
    17 = 432
    18 = 4321
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

八、基数排序（桶排序）

一、基数排序(桶排序)介绍

来源360百科：

> 基数排序(radix sort)属于"分配式排序"(distribution sort)，又称"桶子法"(bucket sort)或bin sort，顾名思义，它是透过键值的部份资讯，将要排序的元素分配至某些"桶"中，藉以达到排序的作用，基数排序法是属于稳定性的排序，其时间复杂度为O (nlog(r)m)，其中r为所采取的基数，而m为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。

从上面的简单介绍，是并不了解基数排序是怎么弄的～基数排序不同与其他的7种排序，其他7种排序本质上都是按照交换或者比较来进行排序，但是基数排序并不是，它是按照**分配，回收(分配到不同的位置上，然后回收)..不断分配..回收**来进行排序，直到有序..

听上去好像很高大上，很难的样子，其实不然。基数排序挺简单的，下面我就来看一下基数排序的流程....

我们有9个桶，将数组的数字按照数值**分配**桶中：

上面我们发现：如果**将桶按顺序**进行**回收**，那么我们的排序就完成了～

可是，一般我们的数组元素都**不仅仅是个位数的数字**的呀，那么高位数的数字又怎么弄呢？？比如：23,44,511,6234这些高位数..

其实也是一样的：

- **第一趟桶排序**将数字的个位数**分配**到桶子里面去，然后**回收**起来，此时数组元素的所有**个位数**都已经排好顺序了
- 第二趟桶排序将数字的十位数分别**分配**到桶子里面去，然后**回收**起来，此时数组元素的所有**个位数和十位数**都已经排好顺序了**(如果没有十位数、则补0)**
- 第三趟桶排序将数字的百位数分别**分配**到桶子里面去，然后**回收**起来，此时数组元素的所有**个位数和十位数和百位数**都已经排好顺序了**(如果没有百位数、则补0)**
- .....
- 直至全部数(个、十、百、千位...)排好顺序，那么这个数组就是有序的了。

机智的同学可能就会发现了，关于这个桶我们可以用**二维数组**来进行存放。

10个桶子就是10列，如果分配时有的数字相同的话，那么就弄成多行~

二、基数排序体验

首先我们有以下这个数组：

```
int[] arrays = {6,  4322, 432, 344, 55 };
```

现在我们有10个桶子，每个桶子下能装载`arrays.length`个数字..

```
int[][] buckets = new int[arrays.length][10];
```

效果如下：

0	1	2	3	4	5	6	7	8	9

2.1第一趟分配与回收

将数组的每个个位数进行分配到不同的桶子上：

0	1	2	3	4	5	6	7	8	9
						6			
		4322							
		432							
				344					
					55				

分配完之后，我们按照顺序来进行回收：得到的结果应该是这样子的：`{4322,432,344,55,6}`

2.2第二趟分配与回收

将数组的每个十位数进行分配到不同的桶子上(像6这样的数，往前边补0)：

于是我们可以得到这样的排序：

0	1	2	3	4	5	6	7	8	9
6									
		4322							
			432						
				344					
					55				

分配完之后，我们按照顺序来进行回收：得到的结果应该是这样子的：`{6,4322,432,344,55}`

2.3第三趟分配与回收

将数组的每个百位数进行分配到不同的桶子上(像6、55这样的数，往前边补0)：

于是我们可以得到这样的排序：

0	1	2	3	4	5	6	7	8	9
6									
55			4322						
				432					
			344						

分配完之后，我们按照顺序来进行回收：得到的结果应该是这样子的：`{6,55,4322,344,432}`

2.4第四趟分配与回收

将数组的每个百位数进行分配到不同的桶子上(像6、55，344，432这样的数，往前边补0)：

于是我们可以得到这样的排序：

0	1	2	3	4	5	6	7	8	9
6									
55									
344									
432									
				4322					

分配完之后，我们按照顺序来进行回收：得到的结果应该是这样子的：`{6,55,344,432,4322}`

此时我们的数组就已经可以排好序了~~~过程就是这样子，其实不难就只有两个步骤：

- **将数组的每一位放进桶子里**
- **回收**
- 循环.....

三、基数排序代码编写

我们的基数排序是按照个、十、百、千位...来进行存放的。前面的演示是已经知道数组元素的数据的情况下来进行存放，但是一般我们是**不去理会数组内元素的值的**。那如果位数很多(万位)或者都是个位数，这个条件我们怎么去处理呢?

我们可以这样做：**先求出数组最大的值，然后不断/10，只要它能大于0，那么它的位数还有~**：

3.1求出数组最大的值

这个我在前面写递归的时候就有这个代码了，我就直接搬去递归的代码过来了，顺便**复习一哈**吧：

- 当然了，**更好的是直接用for循环来找出来就行了(易读性好一些)**

```
    /**
     * 递归，找出数组最大的值
     * @param arrays 数组
     * @param L      左边界，第一个数
     * @param R      右边界，数组的长度
     * @return
     */

    public static int findMax(int[] arrays, int L, int R) {
```

```
        //如果该数组只有一个数，那么最大的就是该数组第一个值了
        if (L == R) {
            return arrays[L];
        } else {

            int a = arrays[L];
            int b = findMax(arrays, L + 1, R);//找出整体的最大值

            if (a > b) {
                return a;
            } else {
                return b;
            }
        }
```

3.2代码实现

```
public static void radixSort(int[] arrays) {

  int max = findMax(arrays, 0, arrays.length - 1);

  //需要遍历的次数由数组最大值的位数来决定
  for (int i = 1; max / i > 0; i = i * 10) {

    int[][] buckets = new int[arrays.length][10];

    //获取每一位数字(个、十、百、千位...分配到桶子里)
    for (int j = 0; j < arrays.length; j++) {

      int num = (arrays[j] / i) % 10;

      //将其放入桶子里
      buckets[j][num] = arrays[j];
    }

    //回收桶子里的元素
    int k = 0;

    //有10个桶子
    for (int j = 0; j < 10; j++) {
      //对每个桶子里的元素进行回收
      for (int l = 0; l < arrays.length ; l++) {

        //如果桶子里面有元素就回收(数据初始化会为0)
        if (buckets[l][j] != 0) {
          arrays[k++] = buckets[l][j];

        }
```

```
        }

    }

  }
}
```

搞了一堆数测试了一哈：

四、桶排序（基数排序）总结

基数排序(桶排序)要理解起来并不困难，不过值得注意的是：**基数排序对有负数和0的数列难以进行排序**

- 因此，往往有0和负数的数组一般我们都不用基数来进行排序

基数排序的要点就两个：

- 分配：按照元素的大小来放入不同的桶子里
- 回收：将桶子里的元素**按桶子顺序**重新放到数组中
- 重复.....两个步骤

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

九、递归

递归介绍

递归在程序语言中简单的理解是：**方法自己调用自己**

递归其实和循环是非常像的，循环**都**可以改写成递归，递归**未必**能改写成循环，这是一个充分不必要的条件。

- 那么，有了循环，为什么还要用递归呢？？**在某些情况下(费波纳切数列，汉诺塔)，使用递归会比循环简单很多很多**
- 话说多了也无益，让我们来感受一下递归吧。

我们初学编程的时候肯定会做过类似的练习：

- `1+2+3+4+....+100(n)`求和
- 给出一个数组，求该数组内部的最大值

我们要记住的是，想要用递归必须知道两个条件：

- **递归出口(终止递归的条件)**
- **递归表达式(规律)**

技巧：在递归中常常是将问题**切割成两个部分(1和整体的思想)**，这能够让我们快速找到递归表达式(规律)

一、求和

如果我们使用`for`循环来进行求和`1+2+3+4+....+100`，那是很简单的：

```
int sum = 0;
for (int i = 1; i <= 100; i++) {

  sum = sum + i;

}
System.out.println("公众号：Java3y：" + sum);
```

前面我说了，for循环都可以使用递归来进行改写，而使用递归必须要知道两个条件：1、递归出口，2、递归表达式(规律)

首先，我们来找出它的规律：`1+2+3+...+n`，这是一个求和的运算，那么我们可以假设`X=1+2+3+...+n`，可以将`1+2+3+...+(n-1)`看成是一个**整体**。而这个整体做的事又和我们的**初始目的(求和)**相同。以我们的高中数学知识我们又可以将上面的式子看成`X=sum(n-1)+n`

好了，我们找到我们的递归表达式(规律)，它就是`sum(n-1)+n`,那递归出口呢，这个题目的递归出口就有很多了，我列举一下：

- 如果`n=1`时，那么就返回`1`
- 如果`n=2`时，那么就返回`3`(1+2)
- 如果`n=3`时，那么就返回`6`(1+2+3)

当然了，我肯定是使用一个最简单的递归出口了：`if(n=1) return 1`

递归表达式和递归出口我们都找到了，下面就代码演示：

递归出口为1：

```
    public static void main(String[] args) {
        System.out.println("公众号：Java3y：" + sum(100));
    }

    /**
     *
     * @param n 要加到的数字，比如题目的100
     * @return
     */
    public static int sum(int n) {

        if (n == 1) {
            return 1;
        } else {
            return sum(n - 1) + n;
        }
    }
```

递归出口为4：

```java
    public static void main(String[] args) {
        System.out.println("公众号：Java3y：" + sum(100));
    }

    /**
     *
     * @param n 要加到的数字，比如题目的100
     * @return
     */
    public static int sum(int n) {

        //如果递归出口为4，(1+2+3+4)
        if (n == 4) {
            return 10;
        } else {
            return sum(n - 1) + n;
        }
    }
```

结果都是一样的。

二、数组内部的最大值

如果使用的是循环，那么我们通常这样实现：

```java
int[] arrays = {2, 3, 4, 5, 1, 5, 2, 9, 5, 6, 8, 3, 2};

//将数组的第一个假设是最大值
int max = arrays[0];

for (int i = 1; i < arrays.length; i++) {

  if (arrays[i] > max) {
    max = arrays[i];
  }
}

System.out.println("公众号：Java3y：" + max);
```

那如果我们用递归的话，那怎么用弄呢？首先还是先要找到**递归表达式(规律)和递归出口**

- 我们又可以运用1和整体的思想来找到规律
  - 将数组第一个数->`2`与数组后面的数->`{3, 4, 5, 1, 5, 2, 9, 5, 6, 8, 3, 2}`进行**切割**，将数组后面的数看成是一个**整体**`X={3, 4, 5, 1, 5, 2, 9, 5, 6, 8, 3, 2}`，那么我们就可以看成是**第一个数和一个整体进行比较**`if(2>X) return 2  else(2<X) return X`

    - 而我们要做的就是找出这个**整体**的最大值与`2`进行比较。找出**整体**的最大值又是和我们的**初始目的(找出最大值)**是一样的
    - 也就可以写成`if( 2>findMax() )return 2 else return findMax()`
- 递归出口，如果数组只有1个元素时，那么这个数组最大值就是它了。

使用到数组的时候，我们通常为数组设定左边界和右边界，这样比较好地进行切割

- L表示左边界，往往表示的是数组第一个元素，也就会赋值为0(角标为0是数组的第一个元素)
- R表示右边界，往往表示的是数组的长度，也就会赋值为`arrays.length-1`（长度-1在角标中才是代表最后一个元素)

那么可以看看我们递归的写法了：

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

        int[] arrays = {2, 3, 4, 5, 1, 5, 2, 9, 5, 6, 8, 3, 1};

        System.out.println("公众号：Java3y：" + findMax(arrays, 0, arrays.length
- 1));


    }


    /**
     * 递归，找出数组最大的值
     * @param arrays 数组
     * @param L      左边界，第一个数
     * @param R      右边界，数组的长度
     * @return
     */

    public static int findMax(int[] arrays, int L, int R) {

        //如果该数组只有一个数，那么最大的就是该数组第一个值了
        if (L == R) {
            return arrays[L];
        } else {

            int a = arrays[L];
            int b = findMax(arrays, L + 1, R);//找出整体的最大值

            if (a > b) {
                return a;
            } else {
                return b;
            }
        }
```

```
    }
```

三、冒泡排序递归写法

在冒泡排序章节中给出了C语言的递归实现冒泡排序，那么现在我们已经使用递归的基本思路了，我们使用Java来重写一下看看：

冒泡排序：俩俩交换，在第一趟排序中能够将最大值排到最后面，外层循环控制排序趟数，内层循环控制比较次数

以递归的思想来进行改造：

- 当第一趟排序后，我们可以将数组最后一位(R)和数组前面的数(L,R-1)进行**切割**，数组前面的数(L,R-1)看成是一个**整体**，这个整体又是和我们的**初始目的(找出最大值，与当前趟数的末尾处交换)**是一样的
- 递归出口：当只有一个元素时，即不用比较了：`L==R`

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

        int[] arrays = {2, 3, 4, 5, 1, 5, 2, 9, 5, 6, 8, 3, 1};
        bubbleSort(arrays, 0, arrays.length - 1);
        System.out.println("公众号：Java3y：" + arrays);


    }

    public static void bubbleSort(int[] arrays, int L, int R) {

        int temp;

        //如果只有一个元素了，那什么都不用干
        if (L == R) ;

        else {
            for (int i = L; i < R; i++) {
                if (arrays[i] > arrays[i + 1]) {
                    temp = arrays[i];
                    arrays[i] = arrays[i + 1];
                    arrays[i + 1] = temp;
                }
            }

            //第一趟排序后已经将最大值放到数组最后面了

            //接下来是排序"整体"的数据了
            bubbleSort(arrays, L, R - 1);

        }
    }
```

四、斐波那契数列

接触过C语言的同学很可能就知道什么是费波纳切数列了，因为往往做练习题的时候它就会出现，它也是递归的典型应用。

菲波那切数列长这个样子：`{1 1 2 3 5 8 13 21 34 55..... n }`

数学好的同学可能很容易就找到规律了：**前两项之和等于第三项**

例如：

```
1 + 1 = 2
2 + 3 = 5
13 + 21 = 34
```

如果让我们求出第n项是多少，那么我们就可以很简单写出对应的递归表达式了：`Z = (n-2) + (n-1)`

递归出口在本题目是需要有两个的，因为**它是前两项加起来才得出第三项的值**

同样地，那么我们的递归出口可以写成这样：`if(n==1) retrun 1 if(n==2) return 2`

下面就来看一下完整的代码吧：

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

        int[] arrays = {1, 1, 2, 3, 5, 8, 13, 21};
        //bubbleSort(arrays, 0, arrays.length - 1);

        int fibonacci = fibonacci(10);
```

```
        System.out.println("公众号：Java3y：" + fibonacci);


    }

    public static int fibonacci(int n) {
        if (n == 1) {
            return 1;
        } else if (n == 2) {
            return 1;
        } else {
            return (fibonacci(n - 1) + fibonacci(n - 2));
        }

    }
```

五、汉诺塔算法

图片来源百度百科：

玩汉诺塔的规则很简单：

- 有三根柱子，原始装满大小不一的盘子的柱子我们称为A，还有两根空的柱子，我们分别称为B和C(任选)
- 最终的目的就是**将A柱子的盘子全部移到C柱子中**
    - 移动的时候有个规则：**一次只能移动一个盘子，小的盘子不能在大的盘子上面(反过来：大的盘子不能在小的盘子上面)**

我们下面就来玩一下：

- 只有一个盘子：
    - 将**A柱子的盘子直接移动到C柱子中**
    - 完成游戏
- 只有两个盘子：
    - 将A柱子上的**小**盘子移动到B柱子中
    - 将A柱子上的**大**盘子移动到C柱子中
    - 最后将在B柱子的**小**盘子移动到C柱子**大**盘子中

    - 完成游戏
- 只有三个盘子：
    - 将A柱子**小**的盘子移动到C柱子中
    - 将A柱子上的**中**盘子移动到B柱子中
    - 将C柱子**小**盘子移动到B柱子**中**盘子中
    - 将A柱子的**大**盘子移动到C柱子中
    - 将B柱子的**小**盘子移动到A柱子中
    - 将B柱子的**中**盘子移动到C柱子中
    - 最后将A柱子的**小**盘子移动到C柱子中
    - 完成游戏

.....

从前三次玩法中我们就可以发现的规律：

- 想要将最大的盘子移动到C柱子，就必须将**其余的盘子移到B柱子处(借助B柱子将最大盘子移动到C柱子中[除了最大盘子，将所有盘子移动到B柱子中])**[递归表达式]
- 当C柱子有了最大盘子时，所有的盘子在B柱子。现在的目的就是**借助A柱子将B柱子的盘子都放到C柱子中(和上面是一样的，已经发生递归了)**
- **当只有一个盘子时，就可以直接移动到C柱子了(递归出口)**
    - **A柱子称之为起始柱子，B柱子称之为中转柱子，C柱子称之为目标柱子**
    - 从上面的描述我们可以发现，起始柱子、中转柱子它们的角色是会变的（A柱子开始是起始柱子，第二轮后就变成了中转柱子了。B柱子开始是目标柱子，第二轮后就变成了起始柱子。总之，**起始柱子、中转柱子的角色是不停切换的**)

简单来说就分成三步：

1. 把 n-1 号盘子移动到中转柱子
2. 把最大盘子从起点移到目标柱子
3. 把中转柱子的n-1号盘子也移到目标柱子

那么就可以写代码测试一下了(回看上面玩的过程)：

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

```java
        int[] arrays = {1, 1, 2, 3, 5, 8, 13, 21};
        hanoi(3, 'A', 'B', 'C');
        System.out.println("公众号：Java3y" );
    }

    /**
     * 汉诺塔
     * @param n n个盘子
     * @param start 起始柱子
     * @param transfer 中转柱子
     * @param target 目标柱子
     */
    public static void hanoi(int n, char start, char transfer, char target) {


        //只有一个盘子，直接搬到目标柱子
        if (n == 1) {
            System.out.println(start + "---->" + target);
        } else {

            //起始柱子借助目标柱子将盘子都移动到中转柱子中(除了最大的盘子)
            hanoi(n - 1, start, target, transfer);
            System.out.println(start + "---->" + target);

            //中转柱子借助起始柱子将盘子都移动到目标柱子中
            hanoi(n - 1, transfer, start, target);

        }
    }
```

我们来测试一下看写得对不对：

```
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
公众号：Java3y----------------------------------------------------
A---->C
A---->B
C---->B
A---->C
B---->A
B---->C
A---->C
公众号：Java3y----------------------------------------------------
```

六、总结

递归的确是一个比较难理解的东西，好几次都把我绕进去了....

要使用递归首先要知道两件事：

- 递归出口(终止递归的条件)
- 递归表达式(规律)

在递归中常常用”整体“的思想，在汉诺塔例子中也不例外：**将最大盘的盘子看成1，上面的盘子看成一个整体**。那么我们在演算的时候就很清晰了：**将”整体“搬到B柱子，将最大的盘子搬到C柱子，最后将B柱子的盘子搬到C柱子中**

因为我们人脑无法演算那么多的步骤，递归是用计算机来干的，只要我们找到了递归表达式和递归出口就要**相信**计算机能帮我们搞掂。

在编程语言中，递归的本质是方法自己调用自己，只是参数不一样罢了。

最后，我们来看一下如果是5个盘子，要运多少次才能运完：

```
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
公众号：Java3y--------------------------------------------------
A---->C
A---->B
C---->B
A---->C
B---->A
B---->C
A---->C
A---->B
C---->B
C---->A
B---->A
C---->B
A---->C
A---->B
C---->B
A---->C
B---->A
B---->C
A---->C
B---->A
C---->B
C---->A
B---->A
B---->C
A---->C
A---->B
C---->B
A---->C
B---->A
B---->C
A---->C
公众号：Java3y--------------------------------------------------
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

十、链表

一、回顾与知新

说起链表，我们先提一下数组吧，跟数组比较一下就很理解链表这种存储结构了。

1.1回顾数组

数组我们无论是C、Java都会学过：

- 数组是一种**连续存储**线性结构，元素类型相同，大小相等

数组的优点：

- **存取速度快**

数组的缺点：

- 事先必须知道数组的长度
- 插入删除元素很慢
- 空间通常是有限制的
- **需要大块连续的内存块**
- 插入删除元素的效率很低

2.2链表说明

看完了数组，**回到我们的链表：**链表是**离散存储**线性结构

> n个节点离散分配，彼此通过指针相连，每个节点只有一个前驱节点，每个节点只有一个后续节点，首节点没有前驱节点，尾节点没有后续节点。

链表优点：

- 空间没有限制
- 插入删除元素很快

链表缺点：

- 存取速度很慢

链表相关术语介绍，我还是通过上面那个图来说明吧：

确定一个链表我们只需要头指针，通过头指针就可以把整个链表都能推导出来了～

链表又分了好几类：

- 单向链表
  - 一个节点指向下一个节点
- 双向链表
  - 一个节点有两个指针域
- 循环链表
  - 能通过任何一个节点找到其他所有的节点，将两种(双向/单向)链表的最后一个结点指向第一个结点从而实现循环

操作链表要时刻记住的是：

- **节点中指针域指向的就是一个节点了！**

二、Java实现链表

算法：

- 遍历
- 查找
- 清空
- 销毁
- 求长度
- 排序
- 删除节点
- 插入节点

我将head节点定义在成员变量上：

```
private static Node head = new Node();
```

首先，我们定义一个类作为节点

- 数据域
- 指针域

为了操作方便我就直接定义成public了。

```
public class Node {

    //数据域
    public Integer data;

    //指针域，指向下一个节点
    public Node next;

    public Node() {
    }

    public Node(int data) {
        this.data = data;
    }

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```

2.1创建链表(增加节点)

向链表中插入数据：

- 找到尾节点进行插入
- 即使头节点.next为null，不走while循环，也是将头节点与新节点连接的(**我已经将head节点初始**

化过了，因此没必要判断头节点是否为null)～

```
/**
 * 向链表添加数据
 *
 * @param value 要添加的数据
 */
public static void addData(int value) {

    //初始化要加入的节点
    Node newNode = new Node(value);

    //临时节点
    Node temp = head;

    // 找到尾节点
    while (temp.next != null) {
        temp = temp.next;
    }

    // 已经包括了头节点.next为null的情况了～
    temp.next = newNode;

}
```

2.2遍历链表

上面我们已经编写了增加方法，现在遍历一下看一下是否正确～～～

从首节点开始，不断往后面找，直到后面的节点没有数据：

```
/**
 * 遍历链表
 *
 * @param head 头节点
 */
public static void traverse(Node head) {


    //临时节点，从首节点开始
    Node temp = head.next;

    while (temp != null) {

        if (temp.data != null) {
            System.out.println("关注公众号Java3y：" + temp.data);
        }
```

```
            //继续下一个
            temp = temp.next;
        }
    }
```

结果：

```
    public static void main(String[] args) {
        insertData( value: 3);
        insertData( value: 4);
        insertData( value: 5);
        insertData( value: 6);


        traverse(head);


    }

    /**
     * 遍历链表
     *
     * @param head 头节点
     */
    public static void traverse(Node head) {
```

```
LinkListText
LinkListText
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
关注公众号Java3y: 3
关注公众号Java3y: 4
关注公众号Java3y: 5
关注公众号Java3y: 6
```

2.3插入节点

1. 插入一个节点到链表中，首先得判断这个位置是否是合法的，才能进行插入～
2. **找到想要插入的位置的上一个节点就可以了～**

```
    /**
     * 插入节点
     *
     * @param head  头指针
     * @param index 要插入的位置
     * @param value 要插入的值
     */
```

```java
    public static void insertNode(Node head, int index, int value) {


        //首先需要判断指定位置是否合法，
        if (index < 1 || index > linkListLength(head) + 1) {
            System.out.println("插入位置不合法。");
            return;
        }

        //临时节点，从头节点开始
        Node temp = head;

        //记录遍历的当前位置
        int currentPos = 0;

        //初始化要插入的节点
        Node insertNode = new Node(value);

        while (temp.next != null) {

            //找到上一个节点的位置了
            if ((index - 1) == currentPos) {

                //temp表示的是上一个节点

                //将原本由上一个节点的指向交由插入的节点来指向
                insertNode.next = temp.next;

                //将上一个节点的指针域指向要插入的节点
                temp.next = insertNode;

                return;

            }

            currentPos++;
            temp = temp.next;
        }

    }
```

2.4获取链表的长度

获取链表的长度就很简单了，遍历一下，每得到一个节点+1即可～

```java
    /**
     * 获取链表的长度
```

```
     * @param head 头指针
     */
    public static int  linkListLength(Node head) {

        int length = 0;

        //临时节点，从首节点开始
        Node temp = head.next;

        // 找到尾节点
        while (temp != null) {
            length++;
            temp = temp.next;
        }

        return length;
    }
```

2.5删除节点

删除某个位置上的节点其实是和插入节点很像的， 同样都要找到上一个节点。**将上一个节点的指针域改变一下，就可以删除了～**

```
    /**
     * 根据位置删除节点
     *
     * @param head  头指针
     * @param index 要删除的位置
     */
    public static void deleteNode(Node head, int index) {


        //首先需要判断指定位置是否合法，
        if (index < 1 || index > linkListLength(head) + 1) {
            System.out.println("删除位置不合法。");
            return;
        }

        //临时节点，从头节点开始
        Node temp = head;

        //记录遍历的当前位置
        int currentPos = 0;


        while (temp.next != null) {

            //找到上一个节点的位置了
            if ((index - 1) == currentPos) {
```

```
                //temp表示的是上一个节点

                //temp.next表示的是想要删除的节点

                //将想要删除的节点存储一下
                Node deleteNode = temp.next;

                //想要删除节点的下一个节点交由上一个节点来控制
                temp.next = deleteNode.next;


                //Java会回收它，设置不设置为null应该没多大意义了(个人觉得,如果不对请指出哦～)
                deleteNode = null;

                return;

            }
            currentPos++;
            temp = temp.next;
        }
    }
```

2.6 对链表进行排序

前面已经讲过了8种的排序算法了【八种排序算法总结】，这次挑简单的冒泡排序吧(其实我是想写快速排序的，尝试了一下感觉有点难.....)

```
  /**
     * 对链表进行排序
     *
     * @param head
     *
     */
    public static void sortLinkList(Node head) {


        Node currentNode;

        Node nextNode;

        for (currentNode = head.next; currentNode.next != null; currentNode = currentNode.next) {

            for (nextNode = head.next; nextNode.next != null; nextNode = nextNode.next) {
```

```
                if (nextNode.data > nextNode.next.data) {

                    int temp = nextNode.data;
                    nextNode.data = nextNode.next.data;

                    nextNode.next.data = temp;

                }
            }


        }
    }
```

2.7找到链表中倒数第k个节点

这个算法挺有趣的：设置两个指针p1、p2，**让p2比p1快k个节点，同时向后遍历，当p2为空，则p1为倒数第k个节点**

```
    /**
     * 找到链表中倒数第k个节点(设置两个指针p1、p2，让p2比p1快k个节点，同时向后遍历，当p2为空，则p1为倒数第k个节点
     *
     * @param head
     * @param k     倒数第k个节点
     */
    public static Node findKNode(Node head, int k) {

        if (k < 1 || k > linkListLength(head))
            return null;
        Node p1 = head;
        Node p2 = head;

        // p2比p1快k个节点
        for (int i = 0; i < k - 1; i++)
            p2 = p2.next;


        // 只要p2为null，那么p1就是倒数第k个节点了
        while (p2.next != null) {

            p2 = p2.next;
            p1 = p1.next;
        }
        return p1;


    }
```

2.8查询链表的中间节点

这个算法也挺有趣的：**一个每次走1步，一个每次走两步，走两步的遍历完，然后走一步的指针，那就是中间节点**

```
/**
 * 查询单链表的中间节点
 */

public static Node searchMid(Node head) {

    Node p1 = head;
    Node p2 = head;


    // 一个走一步，一个走两步，直到为null，走一步的到达的就是中间节点
    while (p2 != null && p2.next != null && p2.next.next != null) {

        p1 = p1.next;
        p2 = p2.next.next;

    }

    return p1;


}
```

2.9通过递归从尾到头输出单链表

```
/**
 * 通过递归从尾到头输出单链表
 *
 * @param head 头节点
 */
public  static  void printListReversely(Node head) {
    if (head != null) {
        printListReversely(head.next);
        if (head.data != null) {
            System.out.println(head.data);
        }
    }
}
```

2.10反转链表

```java
/**
    * 实现链表的反转
    *
    * @param head 链表的头节点
    */
   public static Node reverseList(Node head) {

       Node pre = null;
       Node cur = head;
       while (cur != null) {
           Node next = cur.next;
           cur.next = pre;
           pre = cur;
           cur = next;
       }

       return pre;
   }
// 翻转完，使用下面的代码进行遍历吧：
public static void traverse4Reverse(Node head) {

       //临时节点，从首节点开始
       Node temp = head;

       while (temp != null) {

           if (temp.data != null) {
               System.out.println("关注公众号Java3y：" + temp.data);
           }

           //继续下一个
           temp = temp.next;
       }
   }
```

三、链表总结

操作一个链表只需要**知道它的头指针就可以做任何操作了**

- 添加数据到链表中
  - 遍历找到尾节点，插入即可(只要`while(temp.next != null)`，退出循环就会找到尾节点)
- 遍历链表
  - 从首节点(有效节点)开始，只要不为null，就输出
- 给定位置插入节点到链表中
  - 首先判断该位置是否有效(在链表长度的范围内)
  - **找到想要插入位置的上一个节点**
    - 将原本由上一个节点的指向交由插入的节点来指向
    - 上一个节点指针域指向想要插入的节点
  -

- 获取链表的长度
  - 每访问一次节点，变量++操作即可
- 删除给定位置的节点
  - 首先判断该位置是否有效(在链表长度的范围内)
  - **找到想要插入位置的上一个节点**

    - **将原本由上一个节点的指向后面一个节点**

- 对链表进行排序
    - 使用冒泡算法对其进行排序
- 找到链表中倒数第k个节点
    - 设置两个指针p1、p2，让p2比p1**快k个**节点，同时向后遍历，**当p2为空，则p1为倒数第k个节点**
- 查询链表的中间节点
    - 这个算法也挺有趣的：一个每次走1步，一个每次走两步，**走两步的遍历完，然后走一步的指针，那就是中间节点**
- 递归从尾到头输出单链表
    - 只要下面还有数据，那就往下找，**递归是从最后往前翻**。
- 反转链表
    - 解法1：循环的方式；使用一个pre节点，找到next节点。`cur.next= pre; pre = cur; cur = next`
      解法2：递归的方式；`ListNode rhead = reverseList2(head.next); head.next.next = head; head.next = null; return rhead;`

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

十一、栈

一、数据结构【栈】就是这么简单

1.1数据结构【栈】介绍

数据结构的栈长的是这个样子：

其实非常好理解，我们将栈可以**看成一个箱子**

- 往箱子里面放东西叫做入栈
- 往箱子里面取东西叫做出栈
- 箱子的底部叫做栈底
- 箱子的顶部叫做栈顶

说到栈的特性，肯定会有一句经典的言语来概括：**先进后出(LIFO, Last In First Out)**

- 往箱子里边放苹果，箱子底部的苹果想要拿出来，得先把箱子顶部的苹果取走才行

二、数据结构【栈】 代码实现

栈的分类有两种：

- 静态栈(数组实现)
- **动态栈(链表实现)**

从上一篇写链表我就认知到我的算法是有多渣了，普通的单链表操作也能把我绕得晕晕的。

由于我的链表还不是很熟，栈又不是很难，那么我就用链表来创建动态栈了！

既然是用链表，我们还是把上一篇节点的代码拿过来吧：

```
public class Node {

    //数据域
    public int data;

    //指针域，指向下一个节点
    public Node next;

    public Node() {
    }

    public Node(int data) {
        this.data = data;
    }

    public Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }

}
```

要链表用来表示栈，这次会有两个指针：

- 栈顶
- 栈底

```
public class Stack {

    public Node stackTop;
    public Node stackBottom;

    public Stack(Node stackTop, Node stackBottom) {
        this.stackTop = stackTop;
        this.stackBottom = stackBottom;
    }
```

```
    public Stack() {
    }


}
```

2.1进栈

将原本栈顶指向的节点交由新节点来指向，栈顶指向新加入的节点

```
    /**
     * 进栈
     *
     * @param stack 栈
     * @param value 要进栈的元素
     */
    public static void pushStack(Stack stack, int value) {

        // 封装数据成节点
        Node newNode = new Node(value);


        // 栈顶本来指向的节点交由新节点来指向
        newNode.next = stack.stackTop;

        // 栈顶指针指向新节点
        stack.stackTop = newNode;

    }
```

2.2遍历栈

只要栈顶元素的指针不指向栈底，那么就一直输出遍历结果：

```
    /**
     * 遍历栈(只要栈顶指针不指向栈底指针，就一直输出)
     *
     * @param stack
     */
    public static void traverse(Stack stack) {
        Node stackTop = stack.stackTop;

        while (stackTop != stack.stackBottom) {

            System.out.println("关注公众号：Java3y：" + stackTop.data);
```

```
            stackTop = stackTop.next;
        }


    }
```

测试：

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

        //初始化栈(无元素)
        Stack stack = new Stack(new Node(), new Node());

        //栈顶和栈尾是同一指向
        stack.stackBottom = stack.stackTop;

        //指向null
        stack.stackTop.next = null;


        //进栈
        pushStack(stack, 3);
        pushStack(stack, 4);
        pushStack(stack, 5);

        traverse(stack);

    }
```

结果：

这就符合了先进后出的特性了～

2.3判断该栈是否为空

很简单，只要栈顶和栈底是同一指向，那么该栈就为空

```
    /**
     * 判断该栈是否为空
     *
     * @param stack
     */
    public static void isEmpty(Stack stack) {
        if (stack.stackTop == stack.stackBottom) {

            System.out.println("关注公众号：Java3y---->该栈为空");
        } else {

            System.out.println("关注公众号：Java3y---->该栈不为空");

        }

    }
```

2.4出栈

1. 在出栈之前看看该栈是否为空，不为空才出栈...
2. 将栈顶的元素的指针(指向下一个节点)赋值给栈顶指针(完成出栈)

```java
    /**
     * 出栈(将栈顶的指针指向下一个节点)
     * @param stack
     */
    public static void popStack(Stack stack) {

        // 栈不为空才能出栈
        if (!isEmpty(stack)) {

            //栈顶元素
            Node top = stack.stackTop;

            // 栈顶指针指向下一个节点
            stack.stackTop = top.next;

            System.out.println("关注公众号：Java3y---->出栈的元素是： " + top.data);

        }
    }
```

测试出栈：

```java
            isEmpty(stack);

            // 出栈
            popStack(stack);


        }
```

StackTest › main()

Run StackTest

```
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
关注公众号：Java3y： 5
关注公众号：Java3y： 4
关注公众号：Java3y： 3
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->出栈的元素是： 5
```

多次出栈：

```
        // 出栈
        popStack(stack);
        popStack(stack);
        popStack(stack);
        popStack(stack);
        popStack(stack);


    }
```

StackTest › main()

StackTest

```
"X:\Program Files\Java\jdk1.8.0_91\bin\ja
关注公众号：Java3y：5
关注公众号：Java3y：4
关注公众号：Java3y：3
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->出栈的元素是：5
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->出栈的元素是：4
关注公众号：Java3y---->该栈不为空
关注公众号：Java3y---->出栈的元素是：3
关注公众号：Java3y---->该栈为空
关注公众号：Java3y---->该栈为空
```

2.5清空栈

当时学C的时候需要释放内存资源，可是Java不用呀，所以栈顶指向栈底，就清空栈了

```
/**
 * 清空栈
 * @param stack
 */
public static void clearStack(Stack stack) {

    stack.stackTop = null;
    stack.stackBottom = stack.stackTop;
}
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

十二、队列

一、数据结构【队列】就是这么简单

数据结构的队列长的是这个样子：

其实队列非常好理解，我们将队列可以**看成小朋友排队**

- 队尾的小朋友到指定的地点了-->出队
- 有新的小朋友加入了-->入队

相对于栈而言，队列的特性是：先进先出

- **先排队的小朋友肯定能先打到饭！**

队列也分成两种：

- 静态队列(数组实现)
- 动态队列(链表实现)

这次我就使用数组来实现静态队列了。值得注意的是：**往往实现静态队列，我们都是做成循环队列**

做成循环队列的**好处是不浪费内存资源**！

1.1数据结构【队列】 代码实现

这次我们使用的是数组来实现静态队列，因此我们可以这样设计：

```
public class Queue {


    //数组
    public int [] arrays;

    //指向第一个有效的元素
    public int front;

    //指向有效数据的下一个元素(即指向无效的数据)
    public int rear;

}
```

从上面的设计我们可以发现：**rear并不指向最后一个有效的元素，在循环队列中这样设计是非常方便的**！因为这样设计可以**让我们分得清队头和队尾**(不然循环队列不断入队或出队，位置是变化很快的)

由于我们是循环队列，所以`front`和`rear`值会经常变动，我们得把`front`和`rear`的值限定在一个范围内，不然会超出队列的长度的。

有这么一个算法：`rear= (rear+1)%数组长度`

- 比如rear的下标是2，数组的长度是6，往后面移一位是3，那么`rear = (rear+1) % 6`，结果还是3

1.2初始化队列

。此时队列为空，分配了6个长度给数组(只能装5个实际的数字，rear指向的是无效的位置的)

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

    //初始化队列
    Queue queue = new Queue();

    queue.front = 0;
    queue.rear = 0;
    queue.arrays = new int[6];

}
```

1.3判断队列是否满了

如果rear指针和front指针紧挨着，那么说明队列就满了

```
/**
 * 判断队列是否满了，front和rear指针紧挨着，就是满了
 * @param queue
 * @return
 */
public static boolean isFull(Queue queue) {
    if ((queue.rear + 1) % queue.arrays.length == queue.front) {

        System.out.println("关注公众号：Java3y--->此时队列满了！");
        return true;
    } else {
        System.out.println("关注公众号：Java3y--->此时队列没满了！");
        return false;
    }
}
```

1.4入队

1. 判断该队列是否满了

2. 入队的值插入到队尾中(具体的位置就是rear指针的位置【再次声明：rear指向的是无效元素的位置】
3. rear指针移动(再次指向无效的元素位置)

```
/**
 * 入队
 *
 * @param queue
 */
public static void enQueue(Queue queue,int value) {

    // 不是满的队列才能入队
    if (!isFull(queue)) {

        // 将新的元素插入到队尾中
        queue.arrays[queue.rear] = value;

        // rear节点移动到新的无效元素位置上
        queue.rear = (queue.rear + 1) % queue.arrays.length;
    }
}
```

1.5遍历

只要front节点不指向rear节点，那么就可以一直输出

```
/**
 * 遍历队列
 * @param queue
 *
 */
public static void traverseQueue(Queue queue) {

    // front的位置
    int i = queue.front;

    while (i != queue.rear) {

        System.out.println("关注公众号：Java3y--->" + queue.arrays[i]);

        //移动front
        i = (i + 1) % queue.arrays.length;
    }

}
```

队列没满时：

队列已满了就插入不了了(验证上面的方法是否正确)：

1.6判断该队列是否为空

只要`rear`和`front`指针指向同一个位置，那该队列就是空的了

```
    /**
     * 判断队列是否空，front和rear指针相等，就是空了
     * @param queue
     * @return
     */
    public static boolean isEmpty(Queue queue) {
        if (queue.rear  == queue.front) {
```

```
            System.out.println("关注公众号：Java3y--->此时队列空的！");
            return true;
        } else {
            System.out.println("关注公众号：Java3y--->此时队列非空！");
            return false;
        }
    }
```

1.7出队

出队的逻辑也非常简单：

1. 判断该队列是否为null
2. 如果不为null，则出队，**只要front指针往后面移就是出队了**！

```
    /**
     * 出队
     *
     * @param queue
     */
    public static void outQueue(Queue queue) {

        //判断该队列是否为null
        if (!isEmpty(queue)) {


            //不为空才出队
            int value = queue.arrays[queue.front];
            System.out.println("关注公众号：Java3y--->出队的元素是：" + value);

            // front指针往后面移
            queue.front = (queue.front + 1) % queue.arrays.length;

        }


    }
```

结果：

```java
        //出队
        outQueue(queue);
        outQueue(queue);
        outQueue(queue);
        outQueue(queue);
        outQueue(queue);
        outQueue(queue);
```

QueueTest › main()

QueueTest

```
关注公众号：Java3y--->此时队列没满了！
关注公众号：Java3y--->此时队列没满了！
关注公众号：Java3y--->此时队列没满了！
关注公众号：Java3y--->此时队列没满了！
关注公众号：Java3y--->3
关注公众号：Java3y--->1
关注公众号：Java3y--->2
关注公众号：Java3y--->4
关注公众号：Java3y--->6
关注公众号：Java3y--->此时队列非空！
关注公众号：Java3y--->出队的元素是：3
关注公众号：Java3y--->此时队列非空！
关注公众号：Java3y--->出队的元素是：1
关注公众号：Java3y--->此时队列非空！
关注公众号：Java3y--->出队的元素是：2
关注公众号：Java3y--->此时队列非空！
关注公众号：Java3y--->出队的元素是：4
关注公众号：Java3y--->此时队列非空！
关注公众号：Java3y--->出队的元素是：6
关注公众号：Java3y--->此时队列空的！

Process finished with exit code 0
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y

十三、二叉树

一、二叉树就是这么简单

本文撇开一些非常苦涩、难以理解的概念来讲讲二叉树，仅入门观看(或复习)....

首先，我们来讲讲什么是树：

- 树是一种**非线性**的数据结构，相对于线性的数据结构(链表、数组)而言，**树的平均运行时间更短(往往与树相关的排序时间复杂度都不会高)**

在现实生活中，我们一般的树长这个样子的：

但是在编程的世界中，我们一般把树**“倒”**过来看，这样容易我们分析：

一般的树是有很多很多个分支的，分支下又有很多很多个分支，如果在程序中研究这个会非常**麻烦**。因为本来**树就是非线性**的，而我们计算机的**内存是线性存储**的，太过复杂的话我们无法设计出来的。

因此，我们先来研究**简单又经常用**的---> **二叉树**

1.1树的一些概念

我就拿上面的图来进行画来讲解了：

二叉树的意思就是说：**每个节点不能多于有两个儿子**，上面的图就是一颗二叉树。

- **一棵树至少会有一个节点(根节点)**
- **树由节点组成**，每个节点的数据结构是这样的：

- 因此，我们定义树的时候往往是**->定义节点->节点连接起来就成了树**，而节点的定义就是：**一个数据、两个指针(如果有节点就指向节点、没有节点就指向null)**

1.2静态创建二叉树

上面说了，**树是由若干个节点组成，节点连接起来就成了树**，而**节点由一个数据、两个指针组成**

- 因此，**创建树实际上就是创建节点，然后连接节点**

首先，使用Java类定义节点：

```
public class TreeNode {

    // 左节点(儿子)
    private TreeNode lefTreeNode;

    // 右节点(儿子)
    private TreeNode rightNode;

    // 数据
    private int value;


}
```

下面我们就拿这个二叉树为例来构建吧：

为了方便构建，我就给了它一个带参数的构造方法和set、get方法了：

```
    public TreeNode(int value) {

        this.value = value;
    }
```

那么我们现在就创建了5个节点：

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

        //根节点-->10
        TreeNode treeNode1 = new TreeNode(10);

        //左孩子-->9
        TreeNode treeNode2 = new TreeNode(9);

        //右孩子-->20
        TreeNode treeNode3 = new TreeNode(20);

        //20的左孩子-->15
        TreeNode treeNode4 = new TreeNode(15);

        //20的右孩子-->35
        TreeNode treeNode5 = new TreeNode(35)

    }
```

它们目前的状态是这样子的：

于是下面我们去把它连起来：

```
//根节点的左右孩子
treeNode1.setLefTreeNode(treeNode2);
treeNode1.setRightNode(treeNode3);

//20节点的左右孩子
treeNode3.setLefTreeNode(treeNode4);
treeNode3.setRightNode(treeNode5);
```

连接完之后，那么我们的树就创建完成了。

1.3遍历二叉树

上面说我们的树创建完成了，那怎么证明呢？？我们**如果可以像数组一样遍历它(看它的数据)，那就说明它创建完成了**～

值得说明的是：**二叉树遍历有三种方式**

- 先序遍历
  - 先访问根节点，然后访问左节点，最后访问右节点(根->左->右)
- 中序遍历
  - 先访问左节点，然后访问根节点，最后访问右节点(左->根->右)
- 后序遍历
  - 先访问左节点，然后访问右节点，最后访问根节点(左->右->根)

以上面的二叉树为例：

- 如果是**先序遍历**：`10->9->20->15->35`
- 如果是**中序遍历**：`9->10->15->20->35`
  - 可能需要解释地方：访问完10节点过后，去找的是20节点，**但20下还有子节点**，因此**先**访问的是20的左儿子15节点。由于15节点没有儿子了。所以就返回20节点，访问20节点。最后访问35节点
- 如果是**后序遍历**：`9->15->35->20->10`

- 可能需要解释地方：先访问9节点，随后应该访问的是20节点，**但20下还有子节点**，因此**先**访问的是20的左儿子15节点。由于15节点没有儿子了。所以就去访问35节点，由于35节点也没有儿子了，所以返回20节点，最终返回10节点

一句话总结：**先序(根->左->右)，中序(左->根->右)，后序(左->右->根)。如果访问有孩子的节点，先处理孩子的，随后返回**

无论先中后遍历，每个节点的遍历如果访问有孩子的节点，先处理孩子的(逻辑是一样的)

- 因此我们很容易想到**递归**
- 递归的出口就是：**当没有子节点了，就返回**

因此，我们可以写出这样的**先序遍历代码**：

```
/**
 * 先序遍历
 * @param rootTreeNode  根节点
 */
public static void preTraverseBTree(TreeNode rootTreeNode) {

    if (rootTreeNode != null) {

        //访问根节点
        System.out.println(rootTreeNode.getValue());

        //访问左节点
        preTraverseBTree(rootTreeNode.getLefTreeNode());

        //访问右节点
        preTraverseBTree(rootTreeNode.getRightNode());
    }
}
```

结果跟我们刚才说的是一样的：

我们再用**中序遍历**调用一遍吧：

```
    /**
     * 中序遍历
     * @param rootTreeNode  根节点
     */
    public static void inTraverseBTree(TreeNode rootTreeNode) {

        if (rootTreeNode != null) {

            //访问左节点
            inTraverseBTree(rootTreeNode.getLefTreeNode());

            //访问根节点
            System.out.println(rootTreeNode.getValue());

            //访问右节点
            inTraverseBTree(rootTreeNode.getRightNode());
        }
    }
```

结果跟我们刚才说的是一样的：

```
        preTraverseBTree(treeNode1);
        System.out.println("---------------公众号：Java3y");

    }


    /**
Main
Main
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
9
10
15
20
35
---------------公众号：Java3y

Process finished with exit code 0
```

有意思的是：**通过先序和中序或者中序和后序我们可以还原出原始的二叉树**，但是通过先序和后序是无法还原出原始的二叉树的

- 也就是说：**通过中序和先序或者中序和后序我们就可以确定一颗二叉树了**！

二、动态创建二叉树

上面我们是手动创建二叉树的，一般地：都是给出一个数组给你，让你将数组变成一个二叉树，此时就需要我们动态创建二叉树了。

二叉树中还有一种特殊的二叉树：**二叉查找树(binary search tree)**

- 定义：**当前根节点的左边全部比根节点小，当前根节点的右边全部比根节点大。**
    - 明眼人可以看出，这对我们来**找一个数是非常方便快捷的**

往往我们动态创建二叉树都是创建**二叉查找树**

2.1动态创建二叉树体验

假设我们有一个数组：`int[] arrays = {3, 2, 1, 4, 5};`

那么创建二叉树的步骤是这样的：

- 首先将3作为根节点

- 随后2进来了，我们跟3做比较，比3小，那么放在3的左边

- 随后1进来了，我们跟3做比较，比3小，那么放在3的左边，此时3的左边有2了，因此跟2比，比2小，放在2的左边

- 随后4进来了，我们跟3做比较，比3大，那么放在3的右边

- 随后5进来了，我们跟3做比较，比3大，那么放在3的右边，此时3的右边有4了，因此跟4比，比4大，放在4的右边

那么我们的二叉查找树就建立成功了，**无论任何一颗子树，左边都比根要小，右边比根要大**

2.2代码实现

我们的代码实现也很简单，**如果比当前根节点要小，那么放到当前根节点左边，如果比当前根节点要大，那么放到当前根节点右边。**

因为是动态创建的，因此我们得用一个类来表示根节点

```java
public class TreeRoot {

    private TreeNode treeRoot;

    public TreeNode getTreeRoot() {
        return treeRoot;
    }

    public void setTreeRoot(TreeNode treeRoot) {
        this.treeRoot = treeRoot;
    }
}
```

比较与根谁大，大的往右边，小的往左边：

```java
  /**
     * 动态创建二叉查找树
     *
     * @param treeRoot 根节点
     * @param value    节点的值
     */
    public static void createTree(TreeRoot treeRoot, int value) {


        //如果树根为空(第一次访问)，将第一个值作为根节点
        if (treeRoot.getTreeRoot() == null) {
            TreeNode treeNode = new TreeNode(value);
            treeRoot.setTreeRoot(treeNode);

        } else  {

            //当前树根
            TreeNode tempRoot = treeRoot.getTreeRoot();

            while (tempRoot != null) {
                //当前值大于根值，往右边走
                if (value > tempRoot.getValue()) {

                    //右边没有树根，那就直接插入
                    if (tempRoot.getRightNode() == null) {
                        tempRoot.setRightNode(new TreeNode(value));
                        return ;
                    } else {
                        //如果右边有树根，到右边的树根去
                        tempRoot = tempRoot.getRightNode();
                    }
```

```
                } else {
                    //左没有树根，那就直接插入
                    if (tempRoot.getLefTreeNode() == null) {
                        tempRoot.setLefTreeNode(new TreeNode(value));

                        return;
                    } else {
                        //如果左有树根，到左边的树根去
                        tempRoot = tempRoot.getLefTreeNode();
                    }
                }
            }
        }
    }
```

测试代码：

```
    int[] arrays = {2, 3, 1, 4, 5};

    //动态创建树

    TreeRoot root = new TreeRoot();
    for (int value : arrays) {
        createTree(root, value);
    }


    //中序遍历树
    inTraverseBTree(root.getTreeRoot());
    System.out.println("---------------公众号：Java3y");

    //先序遍历树
    preTraverseBTree(root.getTreeRoot());
    System.out.println("---------------公众号：Java3y");
```

三、查询二叉查找树相关

3.1查询树的深度

查询树的深度我们可以这样想：**左边的子树和右边的字数比，谁大就返回谁，那么再接上根节点+1就可以了**

```java
public static int getHeight(TreeNode treeNode) {

    if (treeNode == null) {
        return 0;
    } else {

        //左边的子树深度
        int left = getHeight(treeNode.getLefTreeNode());

        //右边的子树深度
        int right = getHeight(treeNode.getRightNode());


        int max = left;

        if (right > max) {
            max = right;
        }
        return max + 1;
    }
}
```

3.1查询树的最大值

从上面先序遍历二叉查找树的时候，细心的同学可能会发现：**中序遍历二叉查找树得到的结果是排好顺序的～**

那么，如果我们的二叉树不是二叉查找树，我们要**怎么查询他的最大值呢**?

可以这样：

- **左边找最大值->递归**
- **右边找最大值->递归**

```
/**
 * 找出树的最大值
 *
 * @param rootTreeNode
 */
public static int  getMax(TreeNode rootTreeNode) {

    if (rootTreeNode == null) {
        return -1;
    } else {
        //找出左边的最大值
```

```
            int left = getMax(rootTreeNode.getLefTreeNode());

            //找出右边的最大值
            int right = getMax(rootTreeNode.getRightNode());

            //与当前根节点比较
            int currentRootValue = rootTreeNode.getValue();

            //假设左边的最大
            int max = left;


            if (right > max) {
                max = right;
            }
            if (currentRootValue > max) {
                max = currentRootValue;
            }

            return max ;


        }
    }
```

如果文档中有任何的不懂的问题，都可以直接来找我询问，我乐意帮助你们！微信搜**Java3y**公众号有我的联系方式。更多**原创**技术文章可关注我的GitHub：https://github.com/ZhongFuCheng3y/3y