数据结构与算法

# 数据结构与算法概述

数据结构与算法描述是独立与编程语言的，它描述的是数据在硬盘中的存储方式以级解决问题的思路，所以可以在任何编程语言中使用。

本门课程则是使用Java语言来学习数据结构与算法，所以，请至少已经掌握JavaSE。

既然数据结构与算法描述的是底层的存储结构和原理以及解决问题的思路，所以这门课程注定是枯燥无味，乏味无趣的。然而，掌握了数据结构与算法以后，则可以从上层调用API的层次上升到去底层解决复杂问题的层次。记住，程序 = 数据结构 + 算法。

注意，数据结构与算法是两个概念，但是通常放到一起学习才能够发挥其巨大威力。

# 数据结构概述

## 什么是数据结构

数据结构描述的是数据与数据之间的关系，通过不同的关系解决不同的问题。通常数据之间的关系从两个方面来分析：

* 数据的存储结构
* 数据的逻辑结构

## 数据的存储结构

数据的存储结构指的是数据在计算机中的存储结构，通常来讲是指在内存中的存储结构，个别情况也指在物理硬盘中的存储结构，主要有两种存储结构

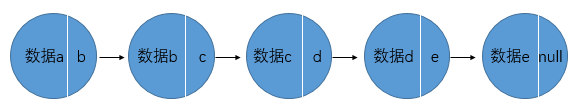
* 顺序存储结构
* 链式存储结构

### 顺序存储结构

顺序存储结构是将数据放在连续的存储单元中，其数据间的逻辑关系和物理关系是一致的。数组就是顺序存储结构的典型代表。

### 链式存储结构

链式存储结构是将数据放在一个个独立的存储单元中，各个存储单元并不相互紧邻挨着，而是任意位置。每一个存储单元都记录了本身要存储的数据以及下一个存储单元的地址，最后一个存储单元不记录下一个存储单元的地址。



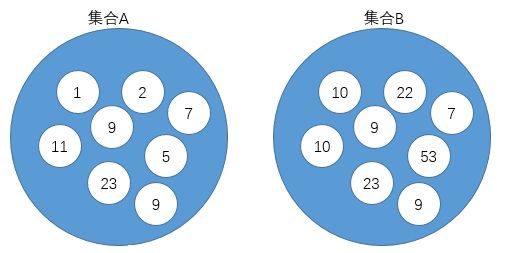
## 数据的逻辑结构

数据的逻辑结构则侧重数据与数据之间的关系，这种关系是人为加上的。主要有如下几种逻辑关系

* 集合结构
* 线性结构
* 树形结构
* 图形结构

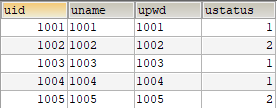
### 集合结构

集合结构中的多个数据元素同属于一组，数据元素之间是并列的关系，并且除此之外没有其它关系。



### 线性结构

线性结构中的数据与数据之间都是有关系的，它们之间是一对一的关系。即：每一个数据都前一条数据(除了第一条)和后一条数据(除了最后一条)



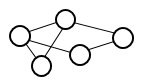
注意：每一行叫做一条记录，可以看到除了第一条记录和最后一条记录之外，每一条记录都有一个前一条记录和一个后一条记录，这就是线性结构。

### 树形结构

树形结构中的数据是一对多的关系。也就是说一个数据元素种有多个子元素。这种结构很常见，比如Windows系统中的目录结构。比如Html，xml等。

### 图形结构

图形结构则是多对多的关系，也就是说一个数据元素可以对应多个其它数据元素，这个数据元素也可以被多个其它数据元素所对应。



# 算法概述

算法其实就是解决具体问题的思路和方法，它具有如下五个明确的特性

* 输入
* 输出
* 有穷性
* 确定性
* 可行性

学习算法并不是单纯的为了学习已有的算法，更多的是为了学习培养一种解决问题的思路

例如 ： 求 1-100的和

**public** **class** Demo\_算法概述 {

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

/\*求 1 -100的和 \*/

**int** total = 0;

//第一种算法（思路）：使用普通的for循环

**for** (**int** i = 1; i <= 101; i++) {

total+=i;

}

System.***out***.println(total);

total = 0;

//第二种算法（思路）： 高斯算法

total = (100+1)\*100 / 2;

System.***out***.println(total);

}

}

## 算法的特性

### 输入

每一个算法都要求可以具有零到多个输入

### 输出

每一个算法都要求运行完毕以后，至少有一个输出结果

### 有穷性

一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。

### 确定性

确定性指示算法在相同条件下的输出结果应该是一致不变的

### 可行性

可行性指示算法是可以解决实际问题的算法，否则类似与“屠龙术”的算法是没有意义的

## 算法的基本要求

* 正确性

正确性要求算法可以正确的解决问题，不能说算法设计出来了但是运算出来的结果是错误的。

* 可读性

算法的步骤应该是明确易懂的，否则其他人无法读懂，这就是算法的可读性

* 健壮性

算法应该是强壮的，可以在处理问题中处理任何的其它异常情况。

* 时间复杂度

算法运行所需要消耗的时间就是时间复杂度

* 空间复杂度

算法运行所需要消耗的内存就是空间复杂度

时间复杂度和空间复杂度是成反向比例的，从来没有见过某个算法的运行即占用了很少的时间，又使用了很少的内存。所以算法没有最好的，只有最合适的。

## 时间复杂度和空间复杂度

### 时间复杂度

### 空间复杂度

# 线性结构

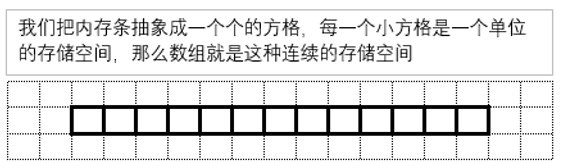
线性结构属于逻辑结构的划分。主要有以下几种：

* 数组
* 栈
* 队列
* 单链表
* 循环链表
* 双链表
* 递归
* 排序算法

## 数组结构

数组：一块连续的存储空间，从存储结构上划分属于顺序存储结构。从逻辑结构上划分属于线性结构。

数组结构示意图



数组结构具有如下特点

* 长度固定：声明数组的时候必须指定长度，且长度不可以被更改
* 数组的每个存储单元都具有角标，角标从0开始
* 数组中的元素都是通过数组的角标来访问，无论是赋值还是获取值。

思考一：如果要添加的元素数量大于数组的长度怎么办？

假设有原数组int [] srcArr = new int[]{1,3,2},现在需要再添加一个元素7到数组中。

解决办法：

1. 新建数组是原数组长度+1：int [] desArr = new int[srcArr.length+1]
2. 将srcArr中的元素依次取出并依次放入到desArr中(使用for循环)
3. 将目标元素7放入到新数组desArr的最后一个位置:desArr[srcArr.length] = 7
4. 将新建数组desArr替换为原数组：srcArr = desArr

思考二：如何删除数组中的元素

删除数组中的元素也很简单，思路也是先创建一个新的数组

假设有原数组int [] srcArr = new int[]{1,3,2,7,9,6},现在请删除数组中的第N个元素。(本题假设N=5)

解决办法：

1. 创建一个新数组，长度是原数组长度-1：int[] desArr = new int[srcArr-1];
2. 将原数组中第N个元素之前和第N个元素之后的元素依次放入到新数组中
3. 将新数组替换成原数组

**示例代码：**

**public** **class** Demo\_删除数组中的元素 {

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

**int** [] srcArr = **new** **int**[] {1,3,2,7,9,6};

//第5个元素角标是4

**int** n = 4;

**int** [] desArr = **new** **int**[srcArr.length-1];

**for** (**int** i = 0; i < desArr.length; i++) {

**if** (i<n) {

desArr[i] = srcArr[i];

}**else** {

desArr[i] = srcArr[i+1];

}

}

srcArr = desArr;

System.***out***.println(Arrays.*toString*(srcArr));

}

}

面向对象的数组-手写ArrayList集合

当需要频繁的操作数组中的元素的时候，难免会考虑将数组封装成一个Java类，提供常用的添加，删除，修改，查找等方法，这个时候这个Java类其实就是ArrayList集合。这样就将长度固定的数组变成了“长度可变”的数组。

### 手写ArrayList

**创建MyArrayList**

**public** **class** Demo03\_ArrayList<T> {

// private T [] elements = new T[DEFAULTSIZE]; 不能直接创建泛型数组，编译报错;

// private T [] elementsx = (T[]) new Object[3]; //不强转也会报错

**private** Object[] elements;

**public** Demo03\_ArrayList() {

elements = **new** Object[0];

}

// 添加元素到数组末尾

**public** **void** add(T element) {}

// 添加元素到指定位置

**public** **void** insert(**int** index, T element) {}

// 删除元素

**public** **void** delete(**int** index) {}

// 修改元素

**public** **void** set(**int** index, T t) {}

// 获取元素

**public** T get(**int** index) { }

// 获取数组长度

**public** **int** size() {}

// 重写toString，直接打印数组中的元素而不是打印数组的内存地址

**public** String toString() {

**return** Arrays.*toString*(elements);

}

}

**添加元素**

// 添加元素到数组末尾

**public** **void** add(T element) {

Object[] temp = **new** Object[elements.length + 1];

**for** (**int** i = 0; i < elements.length; i++) {

temp[i] = elements[i];

}

temp[elements.length] = element;

elements = temp;

}

// 添加元素到指定位置

**public** **void** insert(**int** index, T element) {

Object[] temp = **new** Object[elements.length + 1];

**for** (**int** i = 0; i < elements.length; i++) {

**if** (i < index) {

temp[i] = elements[i];

} **else** {

temp[i + 1] = elements[i];

}

}

temp[index] = element;

elements = temp;

}

**删除元素**

// 删除元素

**public** **void** delete(**int** index) {

Object[] temp = **new** Object[elements.length - 1];

**for** (**int** i = 0; i < temp.length; i++) {

**if** (i < index) {

temp[i] = elements[i];

} **else** {

temp[i] = elements[i + 1];

}

}

elements = temp;

}

**修改元素**

// 修改元素

**public** **void** set(**int** index, T t) {

elements[index] = t;

}

**查找元素**

// 获取元素

**public** T get(**int** index) {

**return** (T) elements[index];

}

**获取集合长度**

// 获取数组长度

**public** **int** size() {

**return** elements.length;

}

**打印集合**

// 重写toString，直接打印数组中的元素而不是打印数组的内存地址

**public** String toString() {

**return** Arrays.*toString*(elements);

}

## 查找算法

当需要查找数组是否存在目标元素的时候，无论你写了怎样的代码，都可以称之为一个“算法”，而最常用的算法则是线性查找和二分法查找

### 线性查找

线性查找，顾名思义，就是将数组中的元素依次取出来，和目标元素进行比对，仅当数组中的某一个元素和目标元素“相等”时，才认为数组中存在这个要查找的元素

示例代码：

public class Demo04\_查找数组中的元素 {

**public** **static** **int** 线性查找(**int**[] arr,**int** target) {

**int** targetElementIndex = -1;

**for** (**int** i = 0; i < arr.length; i++) {

**if** (target == arr[i]) {

targetElementIndex = i;

**break**;

}

}

**return** targetElementIndex;

}

}

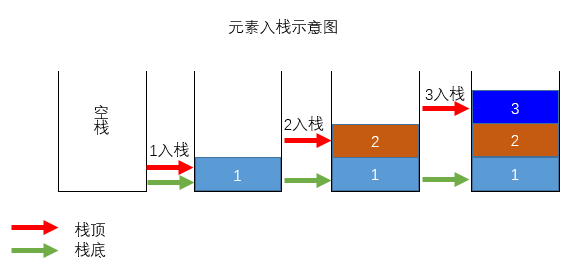
### 二分法查找

## 栈结构

栈结构是一种先进后出的结构，先放入的元素在下面，后放入的元素在上面。取元素的时候只能从上面依次取出来。最上面的元素是栈顶元素，最底部的是栈底元素。

栈结构从逻辑结构上划分属于线性结构，从存储结构上划分属于顺序存储结构

栈结构示意图



解释 ：

* 1号元素入栈以后栈顶元素和栈底元素都是1号元素
* 2号元素入栈以后栈顶元素是2号栈底元素是1号元素
* 3号元素入栈以后栈顶元素是3号栈底元素是1号元素
* 取元素的时候必须先取最后的3号，然后再是2号，最先入栈的1号只能最后取出来

这就是栈结构：先进后出，典型应用场景是手枪的弹夹(左轮手枪除外)。

### 栈结构示例代码

/\*\*

\*

\* 思路 ： 元素入栈每次都添加到数组的末尾

\* 取元素的时候每次都取数组的最后一个，即实现先进后出的栈结构

\*

\* **@author** xiaoka

\*

\*/

**public** **class** Demo05\_栈结构<T> {

**private** Object[] elements;

**public** Demo05\_栈结构() {

**super**();

elements = **new** Object[0];

}

// 压入元素进栈

**public** **void** push(T element) {}

//取出栈顶元素

**public** T pop() {}

// 查看栈顶元素，注意不是取出来栈顶元素

**public** T peek() {}

// 查看栈内是否为空

**public** **boolean** isEmpty() {

**return** elements.length == 0;

}

// 查看栈中内容

**public** **void** showStackElements() {

System.***out***.println(elements.length + " " + Arrays.*toString*(elements));

}

}

**添加元素进入栈结构**

// 压入元素进栈

**public** **void** push(T element) {

// 创建一个新数组，长度是原数组的长度+1

Object[] temp = **new** Object[elements.length + 1];

// 把原数组中的元素依次放入到新数组中

//copyArrElement(elements, temp);

**for** (**int** i = 0; i < elements.length; i++) {

temp[i] = elements[i];

}

// 把目标元素添加到新数组的末尾

temp[elements.length] = element;

// 使用新数组替换旧数组

elements = temp;

}

**从栈结构取元素**

// 取出栈顶元素

**public** T pop() {

**if** (elements.length == 0) {

**throw** **new** RuntimeException("Stack is empty");

}

// 每次都取出数组的最后一个元素

Object topElement = elements[elements.length - 1];

// 创建一个新数组，长度是原数组的长度-1

Object[] temp = **new** Object[elements.length - 1];

// 将原数组中的元素依次放入到新数组中

**for** (**int** i = 0; i < temp.length; i++) {

temp[i] = elements[i];

}

// 使用新数组替换旧数组

elements = temp;

**return** (T) topElement;

}

**查看栈顶元素**

// 查看栈顶元素，注意不是取出来栈顶元素

**public** T peek() {

**if** (elements.length == 0) {

**throw** **new** RuntimeException("Stack is empty");

}

**return** (T) elements[elements.length - 1];

}

**查看栈结构是否为空**

// 查看栈内是否为空

**public** **boolean** isEmpty() {

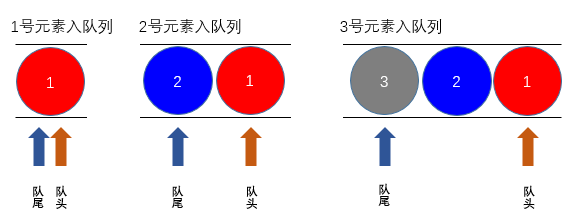
**return** elements.length == 0;

}

## 队列结构

队列结构属于先进先出的数据结构，从逻辑结构上划分属于线性结构，从存储结构上划分属于顺序存储结构。

队列结构示意图



队头先入队列，所以最先出队列

### 队列结构示例代码

/\*\*

\* 实现思路 :

\* 每次都将元素添加到数组的最后一个位置

\* 取元素每次都取数组的第一个位置

\*/

**public** **class** Demo06\_队列结构 <T>{

**private** Object[] elements;

**public** Demo06\_队列结构() {

**super**();

elements = **new** Object[0];

}

// 入队 ： 添加元素到队列中

**public** **void** add(T element) {}

// 出队 ： 每次都从数组的第0个元素取元素就能达到先进先出

**public** T poll() {}

// 判断队列是否为空

**public** **boolean** isEmpty() {

**return** elements.length == 0;

}

//遍历队列中的元素

**public** **void** showQueue() {

System.***out***.println(elements.length + " " + Arrays.*toString*(elements));

}

}

**入队**

// 入队 ： 添加元素到队列中

**public** **void** add(T element) {

//新建数组，是原数组长度 + 1

Object[] temp = **new** Object[elements.length+1];

//把原数组中的元素复制到新数组中

**for** (**int** i = 0; i < elements.length; i++) {

temp[i] = elements[i];

}

//把目标元素添加到新数组的末尾

temp[elements.length] = element;

//使用新数组替换旧数组

elements = temp;

}

**出队**

// 出队 ： 每次都从数组的第0个元素取元素就能达到先进先出

**public** T poll() {

**if**(elements.length == 0)

**throw** **new** RuntimeException("Queue is empty");

//取出来数组中的第0个元素

Object top = elements[0];

//创建一个新数组，长度是原数组的长度 - 1

Object[] temp = **new** Object[elements.length-1];

//将原数组中的元素复制到新数组中

**for** (**int** i = 0; i < temp.length; i++) {

temp[i] = elements[i+1];

}

//使用新数组替换旧数组

elements = temp;

**return** (T) top;

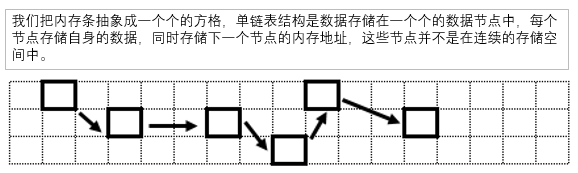
}

## 单向链表结构

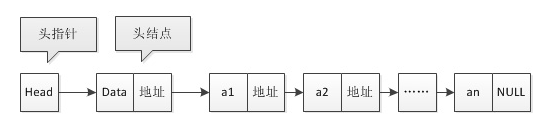
### 单向链表结构简介

单向链表结构从逻辑结构上划分属于线性结构，从存储结构上划分属于链式存储结构。

单向链表结构示意图一



单向链表结构示意图二



这种数据结构要求每一个节点存储本身要存储的数据，同时存储下一个节点的内存地址，最后一个数据节点不需要存储下一个节点内存的地址，有点儿像火车。

### 单向链表示例代码

#### 创建单向链表类

**public** **class** Node<T> {

// 存储本身需要存储的数据

**private** T data;

// 存储下一个节点的内存地址

**private** Node<? **extends** T> next;

**public** Node(T data) {

**super**();

**this**.data = data;

}

//getter方法，略

}

#### 添加节点到链表(一)

//只需要将当前节点的下一个地址指向要追加的目标节点即可

**public** **void** appendX(Demo07\_单链表<? **extends** T> endNode) {

**this**.next = endNode;

}

***一：测试代码***

**public** **class** Test {

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

Node<String> a = **new** Node<>("a");

Node<String> b = **new** Node<>("b");

Node<String> c = **new** Node<>("c");

//a追加节点b

a.appendX(b);

//b追加节点c

b.appendX(c);

//c追加节点d

c.appendX(**new** Node<String>("d"));

}

}

***二：分析***

链表结构的特点是只知道头节点，当查询数据的时候只能从头节点开始查询，然后沿着链表依次一个个查询链表中的节点元素。

appendX()方法虽然也能够实现链表，但是并不是正确的逻辑，因为它添加节点的逻辑要求知道链表中的每一个节点

a.appendX(b);

//b追加节点c，那么就必须知道b节点的位置才能够让添加c节点到链表a中

b.appendX(c);

因此appendX()方法的逻辑是错误的。

我们应该实现的逻辑是这样的

a.appendX(b);

a.appendX(c);

a.appendX(d);

也就是说始终使用a节点来添加节点元素，依然实现a→b→c→d的效果

#### 添加节点到链表(二)

/\* 添加节点到链表 \*/

**public** Node<T> append(Node<? **extends** T> endNode) {

// 获取头节点

Node currentNode = **this**;

**while** (**true**) {

// 不断获取链表中的下一个节点

Node temp = currentNode.next;

// 仅当下一个节点为null的时候，说明到达了链表的末尾节点

**if** (temp == **null**) {

**break**;

}

// 将下一个节点替换成当前节点

currentNode = temp;

}

// 将目标节点追加到末尾节点

currentNode.next = endNode;

// 返回this就可以实现链式编程，否则就只能n1.append(n2);n1.append(n3);

**return** **this**;

}

***一：测试代码***

**public** **class** Test {

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

Node<String> a = **new** Node<>("a");

Node<String> b = **new** Node<>("b");

Node<String> c = **new** Node<>("c");

a.append(b).append(c); //因为返回了this才可以实现链式编程

a.append(**new** Node<String>("d"));

}

}

#### 打印链表

/\*展示列表中所有的节点数据\*/

**public** **void** showNodeList() {

Node currentNode = **this**;

**while** (**true**) {

System.***out***.print(currentNode.getData() + " ");

currentNode = currentNode.next;

**if** (currentNode == **null**) {

**break**;

}

}

}

#### 查找节点

/\* 查找节点 \*/

**public** **boolean** findNode(Node<T> node) {

Node currentNode = **this**;

**while** (currentNode.next != **null**) {

**if** (currentNode.getData().equals(node.getData()))

**return** **true**;

currentNode = currentNode.next;

}

// 判断列表的最后一个元素

**if** (currentNode.getData().equals(node.getData()))

**return** **true**;

**return** **false**;

}

#### 删除节点(一)

n1→n2→n3→n4，这是目前的链表 假设要删除n3节点，链表就会是这样n1→n2→n4

如果直接删除n3节点，我们只能获得n3的下一个节点n4，无法获得n3的上一个节点n2

那么就无法将n2→n4

所以我们应该通过n2来删除n3节点，也是就是说用上一个节点删除下一个节点的方式来删除一个节点

/\* 删除节点\*/

**public** **void** removeNext() {

// this = n2,this.next=n3,this.next.next = n4

// n2→n4

**this**.next = next.next;

}

***一：测试代码***

//移除b节点

a.removeNext();

***二：分析***

目前这种方式有两个缺陷

1. ：只能够删除头节点以外的节点
2. ：最大的缺点，要想删除d节点，就只能用c.removeNext(),无法实现真正的链表，因为真正的链表中是只能够找到a节点的，应该用a节点移除d节点。

#### 删除节点(二)

**public** **void** removeNode(Node<T> node) {

// 使用上一个节点删除当前(下一个)节点

removNextNode(node);

}

**private** **void** removNextNode(Node<T> node) {

Node currentNode = **this**;

**while** (currentNode.next != **null**) {

// 如果当前元素的下一个节点就是要删除的目标节点

// 就直接使用当前节点指向目标节点的下一个节点

**if** (currentNode.next.getData().equals(node.getData())) {

currentNode.next = currentNode.next.next;

**break**;

}

currentNode = currentNode.next;

}

}

***一：测试代码***

a.removeNode(b);

a.removeNode(c);

a.removeNode(**new** Node<String>("d"));

***二：分析***

这种方式可以实现a删除b,a删除c等等节点，然而这种方式无法删除头节点

#### 删除节点(三)

**private** Node<T> removNextNode(Node<T> node) {

Node currentNode = **this**;

//判断头节点是否是要删除的节点

**if** (**this**.getData().equals(node.getData())) {

**return** (Node<T>) **this**.next;

}

**while** (currentNode.next != **null**) {

**if** (currentNode.next.getData().equals(node.getData())) {

currentNode.next = currentNode.next.next;

**break**;

}

currentNode = currentNode.next;

}

**return** **this**;

}

***一：测试代码***

Node<String> node = a.removeNode(a);

node = node.removeNode(d);

***二：分析***

这种方式可以删除任何节点，包括头节点，实现思路是重新生成引用，注意，并不是重新创建了链表中的节点对象。缺点是，并没有改变a链表中的关系。

#### 删除节点(四)

以上三中删除节点的思路都或多或少有点儿问题，第四种思路则是当要删除头节点的时候，则把第二个节点的数据复制给第一个节点的数据，然后删除第二个节点，以变相达到删除头节点的目的。

**private** **void** removNextNode(Node<T> node) {

Node currentNode = **this**;

//判断头节点是否是要删除的节点

**if** (**this**.getData().equals(node.getData())) {

//将第二个节点的数据复制给头节点数据

**this**.data = (T) **this**.next.getData();

//删除第二个节点即可

**this**.next = **this**.next.next;

**return** ;

}

**while** (currentNode.next != **null**) {

**if** (currentNode.next.getData().equals(node.getData())) {

currentNode.next = currentNode.next.next;

**break**;

}

currentNode = currentNode.next;

}

}

***一：测试代码***

a.removeNode(a);

a.removeNode(d);

***二：分析***

这种方式是变相的实现了删除头节点，其实就是复制第二个节点数据给头节点以后再删除第二个节点，中间并没有创建任何对象，也没有生成新的引用对象，但是它有一个专门判断头节点的过程。

通过单链表结构示例图二，我们可以创建一个这样的链表，头节点不存储数据，只作为链表的头指针而存在，所有的数据都存储在第二个节点以及以后的节点中，这样就可以在删除节点的时候不用再专门判断头节点。

**private** **void** removNextNode(Node<T> node) {

Node currentNode = **this**;

**while** (currentNode.next != **null**) {

**if** (currentNode.next.getData().equals(node.getData())) {

currentNode.next = currentNode.next.next;

**break**;

}

currentNode = currentNode.next;

}

}

***一：测试代码***

//创建头指针节点，不存数据

Node<String> head = **new** Node<>();

Node<String> a = **new** Node<>("a");

Node<String> b = **new** Node<>("b");

Node<String> c = **new** Node<>("c");

Node<String> d = **new** Node<>("d");

head.append(a).append(b).append(c);

head.append(d);

head.removeNode(a);

head.removeNode(d);

***二：分析***

这种方式的缺点是会有一个“无用”的头节点在内存中，优点是对于链表节点的操作方便了很多。

由于有了头节点，那么在查找节点的时候没有必要再专门判断头节点元素，在删除节点的时候也没有必要再专门的判断头节点元素

也由于有了头节点元素，在对链表的操作中，应该始终从第二个节点开始

/\*展示列表中所有的节点数据\*/

**public** **void** showNodeList() {

//头节点不存储数据，直接从第二个开始即可

Node currentNode = **this**.next;

. . .

}

/\* 查找节点 \*/

**public** **boolean** findNode(Node<T> node) {

Node currentNode = **this**.next;

. . .

}

#### 末尾节点判断

// 当前节点是否是最后一个节点

**public** **boolean** isLast() {

**return** **this**.next == **null**;

}

#### 插入节点(一)

//将目标节点插入到当前节点后面

**public** **void** after(Node<T> node) {

//获得当前节点的下一个节点

Node<T> temp = (Node<T>) **this**.next;

//将当前节点的下一个节点指向目标节点

**this**.next = node;

//将以前的节点连接到目标节点

node.next = temp;

}

***一：测试代码***

//插入头节点

head.after(**new** Node<String>("1"));

//插入尾节点

d.after(**new** Node<String>("2"));

//插入中间节点

c.after(**new** Node<String>("3"));

***二：分析***

使用了带头指针的单链表以后确实方便了很多，但是现在这个插入节点的逻辑不够完善，目前依然需要通过中间节点来插入节点，请实现如下逻辑

head.after(**原节点,目标节点**);

也就是说，插入节点的时候也始终使用头指针节点来插入节点，将目标节点插入到指定的原节点后面

示例 ：

执行之前的链表 ：headN→n1→n2→n3→n4

head.after(n3,n7);

执行完以后链表 ：headN→n1→n2→n3→n7→n4

#### 插入节点(二)

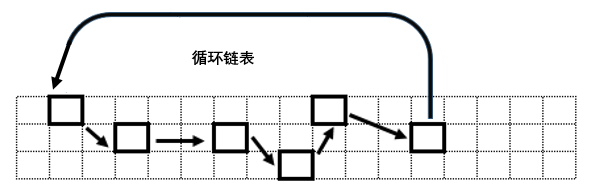
请实现插入节点(一)分析中留下的问题，补充到当前word中。

## 单向循环链表

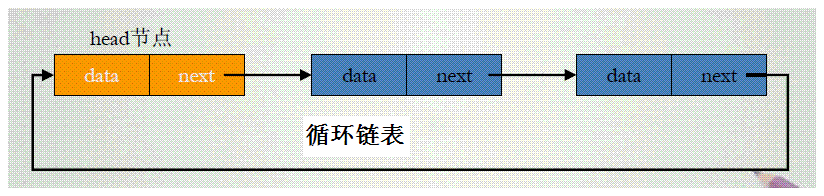
### 单向循环链表结构简介

单向循环链表结构从逻辑结构上划分属于线性结构，从存储结构上划分属于链式存储结构。它和单链表唯一的区别是循环链表的尾节点的下一个指向了头节点，换句话说它已经没有了尾节点。

单向循环链表示意图一：



单向循环链表示意图二：



### 单向循环链表示例代码

#### 创建单向循环链表类

**public** **class** LoopNode<T> {

**private** T data;

// 循环链表没有最后一个节点

// 当循环链表中仅有一个的时候，节点的下一个指向自己

**private** LoopNode<T> next = **this**;

**public** LoopNode(T data) {

**super**();

**this**.data = data;

}

**public** T getData() {

**return** data;

}

**public** LoopNode<T> getNext() {

**return** next;

}

}

***一：测试代码***

LoopNode<String> n1 = **new** LoopNode<>("1");

System.***out***.println(n1.getNext().getData());

#### 插入节点

在单链表中有末尾节点，所以可以实现追加节点的逻辑，因为追加节点是追加到单链表的末尾，而循环链表中没有末尾节点，所以不能追加节点，只能插入节点，插入节点很简单，只要随便找一个节点将目标节点追加即可，使用单链表插入节点(一)即可给循环链表插入节点。

**public** **void** after(LoopNode<T> node) {

//取出当前节点的下一个节点

LoopNode<T> temp = **this**.next;

//把新节点添加到当前节点后

**this**.next = node;

//把原来的下一个节点作为新节点的下一个节点

node.next = temp;

}

***一：测试代码***

LoopNode<String> n1 = **new** LoopNode<>("1");

LoopNode<String> n2 = **new** LoopNode<>("2");

LoopNode<String> n3 = **new** LoopNode<>("3");

n1.after(n2);

n2.after(n3);

System.***out***.println(n1.getNext().getData()); //1

System.***out***.println(n2.getNext().getData()); //2

System.***out***.println(n3.getNext().getData()); //3

***二：分析***

#### 删除节点(一)

使用单链表删除节点(一)来删除循环链表中的节点

//删除下一个节点

**public** **void** removeNext() {

**this**.next = next.next;

}

***一：测试代码***

n3.removeNext();

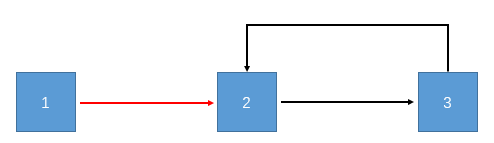
System.***out***.println(n1.getNext().getData()); //2

System.***out***.println(n2.getNext().getData()); //3

System.***out***.println(n3.getNext().getData()); //2

***二：分析***

这种删除的方法其实是”假删除”,因为在本次代码中虽然删除了n1节点，但是n1节点依然指向了n2节点，所以内存中就等于有了这两个链表



黑色的循环链表

n2→n3→n2→n3. . .

红色的单链表

n1→n2→n3

#### 删除节点(二)

//彻底删除下一个节点

**public** **void** removeNext(LoopNode<T> node) {

**this**.next = next.next;

node.next = **null**;

}

***一：测试代码***

n3.removeNext(n1);

//空指针异常，因为n1已经不再指向任何下一个节点

//System.***out***.println(n1.getNext().getData());

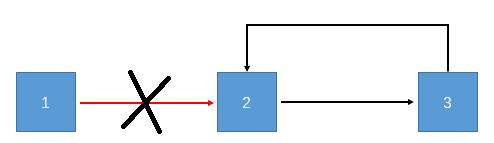
System.***out***.println(n2.getNext().getData()); //3

System.***out***.println(n3.getNext().getData()); //2

***二：分析***

因为传入了参数，则可以灵活的将n1指向n2的next属性值为null，从而将n1彻底从循环链表中删除。

注意：n1对象依然是存在的，只是n1.next不再指向了任何其它节点



## 双向链表结构

### 双向链表结构简介

双向链表结构从逻辑结构上划分属于线性结构，从存储结构上划分属于链式存储结构。

该结构的特点是每一个数据节点都分成了三部分

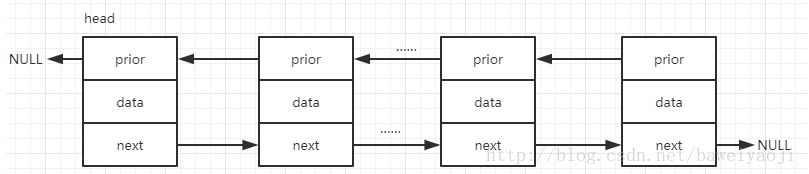
第一部分 ： 存储上一个节点的地址，用prior表示

第二部分 ： 存储本节点需要存储的数据，用data表示

第三部分 ： 存储下一个节点的地址，用next表示

由于头节点没有上一个节点，所以头节点的prior指向null,尾节点没有下一个节点，所以尾节点的next指向null。

双向链表结构示意图



## 双向循环链表结构

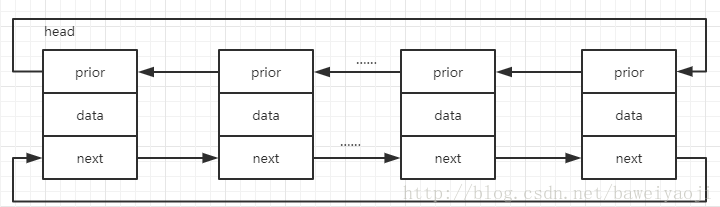
### 双向循环链表结构简介

双向循环链表结构与双向链表结构一样，每部分的功能也一致，不同点是

由于头节点没有上一个节点，所以头节点的prior指向尾节点，尾节点没有下一个节点，所以尾节点的next指向头节点，以此来构成一个循环链表。

如果双向循环链表中直有一个节点元素，那么该节点元素的prior和next均指向自己。

双向循环链表结构示意图



### 双向循环链表示例代码

#### 创建双向循环链表类

**public** **class** DoubleLoopNode<T> {

//上一个节点，链表中仅有一个节点就指向自己

**private** DoubleLoopNode<? **extends** T> prior = **this**;

//下一个节点，链表中仅有一个节点就指向自己

**private** DoubleLoopNode<? **extends** T> next = **this**;

//当前节点数据

**private** T data;

**public** DoubleLoopNode(T data) {

**super**();

**this**.data = data;

}

//获取数据

**public** T getData() {

**return** **this**.data;

}

//获取下一个节点

**public** DoubleLoopNode<T> next() {

**return** (DoubleLoopNode<T>) **this**.next;

}

//获取上一个节点

**public** DoubleLoopNode<T> pre() {

**return** (DoubleLoopNode<T>) **this**.prior;

}

}

***一：测试代码***

DoubleLoopNode<String> head = **new** DoubleLoopNode<String>("1");

System.***out***.println(head.pre().getData()); // 1

System.***out***.println(head.getData()); // 1

System.***out***.println(head.next().getData());// 1

***二：分析***

双向循环链表中仅有一个数据节点，那么该节点的前一个和后一个均指向自己，所以获得的数据都是1。

#### 添加节点

循环链表中没有头节点和尾节点，所以添加节点的时候随意拿一个节点添加其它节点即可。

//给当前节点后添加一个节点

**public** **void** after(DoubleLoopNode<T> newNextNode) {

//获取当前节点原来的下个节点

DoubleLoopNode<T> srcNextNode = (DoubleLoopNode<T>) **this**.next;

//将新节点追加到当前节点

**this**.next = newNextNode;

newNextNode.prior = **this**;

//将原来的节点追加到新节点后面

newNextNode.next = srcNextNode;

srcNextNode.prior = newNextNode;

}

***一：测试代码***

DoubleLoopNode<String> one = **new** DoubleLoopNode<String>("1");

DoubleLoopNode<String> two = **new** DoubleLoopNode<String>("2");

DoubleLoopNode<String> three = **new** DoubleLoopNode<String>("3");

/\*

\* 1 的上一个和下一个都是 2

\* 2 的上一个和下一个都是1

\* 1 → 2

\* 2 ← 1

\*/

one.after(two);

System.***out***.println(one.next().getData()); // 2

System.***out***.println(one.pre().getData()); // 2

System.***out***.println(two.next().getData()); // 1

System.***out***.println(two.pre().getData()); // 1

// 1 → 2 → 3 → 1

// 3 ← 1 ← 2 ← 3

two.after(three);

System.***out***.println(one.next().getData()); //2

System.***out***.println(two.next().getData()); //3

System.***out***.println(three.next().getData()); //1

System.***out***.println(one.pre().getData()); //3

System.***out***.println(two.pre().getData()); //1

System.***out***.println(three.pre().getData());//2

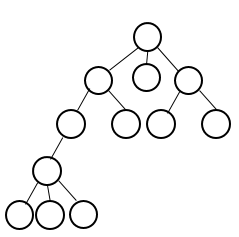
***二：分析***

通过示例代码可以发现，双向循环链表是没有头节点和尾节点的，添加的时候只要使用任意节点将目标节点添加到链表中即可。

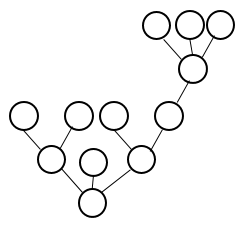
# 树形结构

## 树形结构简介

树形结构从逻辑机构上划分就是单独的树形结构，从存储结构上划分既有链式存储结构又有顺序存储存储结构的二叉树。下图就是树形结构。



如果将图片倒过来看，就更加接近一棵树的形状 ： 最下面的是根，依次是分支，分支上依然有其它子分支…，所以这种结构叫做树形结构。



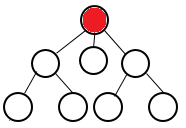
### 为什么需要树形结构

线性结构中数组结构具有查询快，增删慢的特点，而线性结构中的链表结构则是查询慢，增删快。树形结构则兼具两者的优点，查询快，增删也快，故此需要树形结构。

### 树形结构的基本概念

* 根节点

通常画树形结构都是由上往下开始，那么最上面的就是根节点，图中红色节点就是根节点

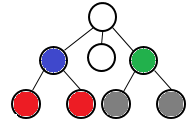


* 双亲节点

自己的上一层属于自己的双亲节点，如图

蓝色节点就是红色节点的双亲节点，绿色节点就是灰色节点的双亲节点

根节点则是蓝色，空心，绿色节点三个节点的双亲节点。



* 子节点

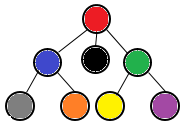
自己的下一层就属于自己的子节点

蓝，黑，绿节点是红色节点的子节点

灰，橙是蓝色节点的子节点

黑色节点没有自己的子节点

黄，紫是绿色节点的子节点



* 路径

路径分为绝对路径和相对路径

绝对路径是从根节点到目标节点所要经过的节点

相对路径是从当前节点开始，到目标节点所要经过的节点

例如现在在灰色节点，要到达紫色节点

绝对路径则是 ： 红色 – 绿色 – 紫色

相对路径则是 ： 灰色 – 蓝色 - 红色 – 绿色 – 紫色

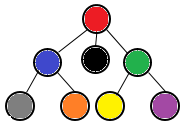
* 节点的度

一个节点有X个子节点，那么它的度就是X。如下图

红色根节点的度是3

蓝色和绿色阶段的度是2

黑色，灰色，橙色，黄色，紫色节点的度都是0.



* 节点的权

节点的权指节点中存储的数字的值。如下图

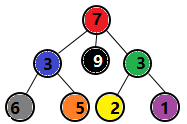
红色节点的权是7

蓝色节点的权是3

黑色节点的权是9

绿色节点的权也是3

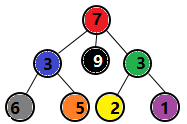
…



* 叶子节点

没有子节点的节点就是叶子节点，如下图

9，6 ，5， 2， 1这5个节点都是叶子节点



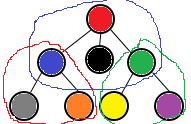
* 子树

子树指的是整个树形结构中的部分树形结构，如下图

蓝色圈中是一颗子树

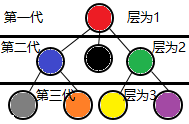
红色圈中是一颗子树

绿色圈中也是一颗子树



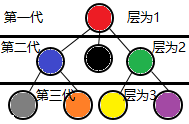
* 层

层的概念是指一代一代的节点。每一代的节点位于同一层。层值从根节点开始，每一层都加1。



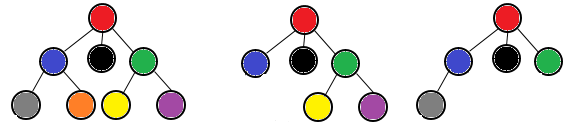
* 树的高度

树的高度就是最大层数，如图树的高度就是3



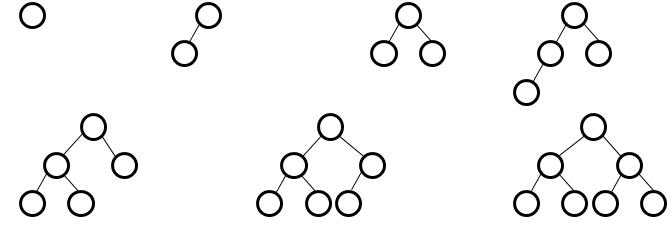
* 森林

多棵树共同组成的结构就叫做森林。如图三棵树组成一个森林



## 二叉树概述

对于一颗树而言，每一个节点的子节点数量都是<=2的，这就二叉树。如下图，都是二叉树。而上图则不是二叉树，因为红色根节点具有了三个字节点。



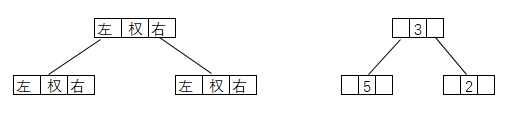
需要注意的是二叉树严格区分左节点和右节点，例如下图，虽然他们的权是一样的，但是他们是不同的二叉树，因为他们的权放在了不同的左右节点。



## 二叉树的分类

## 创建二叉树

二叉树中的每个节点应该具备三个部分 ： 权，左子节点，右子节点，如下图



**示例代码**

* 创建树的节点类

**public** **class** Demo10\_TreeNode {

// 节点的权

**private** **int** value;

// 左子节点

**private** Demo10\_TreeNode leftNode;

// 右子节点

**private** Demo10\_TreeNode rightNode;

// 构造方法

**public** Demo10\_TreeNode(**int** value) {

**super**();

**this**.value = value;

}

// ---getter/setter/toString()—略--

}

* 创建树类

**public** **class** Demo10\_BinaryTree {

//树的根节点

**private** Demo10\_TreeNode rootNode;

// ---getter/setter/toString()—略--

}

* 测试类

**public** **class** Test {

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

// 创建一颗树 : 因为目前没有任何根节点，所以它是一颗空树

Demo10\_BinaryTree tree = **new** Demo10\_BinaryTree();

// 创建根节点

Demo10\_TreeNode rootNode = **new** Demo10\_TreeNode(3);

// 创建左子节点

Demo10\_TreeNode leftNode = **new** Demo10\_TreeNode(5);

// 创建右子节点

Demo10\_TreeNode rightNode = **new** Demo10\_TreeNode(2);

// 给根节点连接左子节点

rootNode.setLeftNode(leftNode);

// 给根节点连接右子节点

rootNode.setRightNode(rightNode);

// 将根节点添加到树中

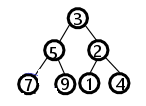
tree.setRootNode(rootNode);

}

}

## 遍历二叉树

有二叉树如下



二叉树的遍历思想有三种

* 前序遍历

前序遍历的顺序是 ： 根 – 左 – 右，对应上图的取出顺序则是 3-5-7-9-2-1-4

红色子树顺序是根左右，黄色色子树内部也是根左右，根节点的顺序而言也是根(3),左(红色)，右(黄色)

* 中序遍历

中序遍历的顺序是 ： 左 – 根 – 右，对应上图的取出顺序则是 7-5-9-3-1-2-4

从根节点3开始遍历，3的左节点是5，而5具有子节点7，所以首先遍历出7

红色子树的顺序是左根右，黄色子树内部也是左根右，根节点的顺序也是左（红色），根（3），右（黄色）

* 后序遍历

后序遍历的顺序是 ： 左 – 右 – 根，对应上图的取出顺序则是 7-9-5-1-4-2-3

红色子树的顺序是左右根，黄色子树内部也是左右根，根节点的顺序也是左（红色），右（黄色），根（3）

注意 ： 遍历是从根节点开始递归遍历，所以子树遍历完成以后才遍历父树。记住，根节点在X序就是X序遍历（X取值前中后的某一个）。

### 前序遍历示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

/\* 前序遍历 \*/

**public** **void** qianXuBianLi() {

// 自己永远是根节点，所以首先打印自己的权值

System.***out***.println(value);

// 然后遍历左子节点的值

**if** (leftNode != **null**) {

leftNode.qianXuBianLi();

}

// 然后遍历右字节点值

**if** (rightNode != **null**) {

rightNode.qianXuBianLi();

}

}

}

### 中序遍历示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

/\*中序遍历\*/

**public** **void** zhongXuBianLi() {

// 首先遍历左子节点

**if** (leftNode != **null**) {

leftNode.zhongXuBianLi();

}

// 然后遍历根节点

System.***out***.println(value);

// 最后遍历右子节点

**if** (rightNode != **null**) {

rightNode.zhongXuBianLi();

}

}

}

### 后序遍历示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

/\*后序遍历\*/

**public** **void** houXuBianLi() {

// 首先遍历左子节点

**if** (leftNode != **null**) {

leftNode.houXuBianLi();

}

// 然後遍历右子节点

**if** (rightNode != **null**) {

rightNode.houXuBianLi();

}

// 最后遍历根节点

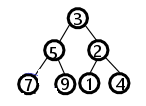
System.***out***.println(value);

}

}

### 测试代码

创建如下二叉树，使用三种顺序遍历



修改“树的代码”，应当使”树”具有遍历的能力

/\*\*

\* 树类

\*/

**public** **class** Demo10\_BinaryTree {

// 树的根节点

**private** Demo10\_TreeNode rootNode;

**public** Demo10\_TreeNode getRootNode() {

**return** rootNode;

}

**public** **void** setRootNode(Demo10\_TreeNode rootNode) {

**this**.rootNode = rootNode;

}

**public** **void** qianXuBianLi() {

rootNode.qianXuBianLi();

}

**public** **void** zhongXuBianLi() {

rootNode.zhongXuBianLi();

}

**public** **void** houXuBianLi() {

rootNode.houXuBianLi();

}

}

**public** **class** Test {

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

//创建树

Demo10\_BinaryTree tree = **new** Demo10\_BinaryTree();

// 创建根节点

Demo10\_TreeNode threeNode = **new** Demo10\_TreeNode(3);

Demo10\_TreeNode fiveNode = **new** Demo10\_TreeNode(5);

Demo10\_TreeNode twoNode = **new** Demo10\_TreeNode(2);

Demo10\_TreeNode sevenNode = **new** Demo10\_TreeNode(7);

Demo10\_TreeNode nineNode = **new** Demo10\_TreeNode(9);

Demo10\_TreeNode oneNode = **new** Demo10\_TreeNode(1);

Demo10\_TreeNode fourNode = **new** Demo10\_TreeNode(4);

threeNode.setLeftNode(fiveNode);

threeNode.setRightNode(twoNode);

fiveNode.setLeftNode(sevenNode);

fiveNode.setRightNode(nineNode);

twoNode.setLeftNode(oneNode);

twoNode.setRightNode(fourNode);

tree.setRootNode(threeNode);

//从根节点开始遍历

tree.qianXuBianLi();

System.***out***.println("--------------");

tree.zhongXuBianLi();

System.***out***.println("--------------");

tree.houXuBianLi();

}

}

## 查找节点

从一颗树中查找某个节点是否存在，只需要遍历二叉树所有的节点即可判断出来这个节点是否存在与树中。遍历二叉树有三种顺序，那么查询某个节点也就存在了三种顺序。

### 前序查找示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

**public** Demo10\_TreeNode qianXuChaZhao(**int** value) {

Demo10\_TreeNode targetNode = **null**;

System.***out***.println(**this**.value);

// 首先判断根节点是否是要查找的节点

**if** (**this**.value == value) {

targetNode = **this**;

}

**if** (targetNode != **null**) {

// 说明根节点就是目标节点，返回即可

**return** targetNode;

}

// 其次遍历左子节点，查找左子节点是否存在要查找的节点

**if** (leftNode != **null**) {

targetNode = leftNode.qianXuChaZhao(value);

}

**if** (targetNode != **null**) {

// 说明在子节点查找到了目标节点，返回即可

**return** targetNode;

}

// 最后遍历右子节点，查询右子节点是否存在要查找的节点

**if** (rightNode != **null**) {

targetNode = rightNode.qianXuChaZhao(value);

}

**return** targetNode;

}

}

### 中序查找示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

**public** Demo10\_TreeNode zhongXuChaZhao(**int** value) {

Demo10\_TreeNode targetNode = **null**;

// 首先遍历左子节点

**if** (leftNode != **null**) {

targetNode = leftNode.zhongXuChaZhao(value);

}

**if** (targetNode != **null**) {

// 说明在左子节点查找到了目标节点

**return** targetNode;

}

System.***out***.println(**this**.value);

**if** (**this**.value == value) {

targetNode = **this**;

}

**if** (targetNode != **null**) {

// 说明在当前根节点查找到了目标节点

**return** targetNode;

}

// 最后遍历右子节点

**if** (rightNode != **null**) {

targetNode = rightNode.zhongXuChaZhao(value);

}

**return** targetNode;

}

}

### 后序查找示例代码

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

**public** Demo10\_TreeNode houXuChaZhao(**int** value) {

Demo10\_TreeNode targetNode = **null**;

// 首先遍历左子节点

**if** (leftNode != **null**) {

targetNode = leftNode.houXuChaZhao(value);

}

**if** (targetNode != **null**) {

// 说明在左子节点查找到了目标节点

**return** targetNode;

}

//其次遍历右子节点

**if** (rightNode != **null**) {

targetNode = rightNode.houXuChaZhao(value);

}

**if** (targetNode != **null**) {

// 说明在右子节点查找到了目标节点

**return** targetNode;

}

System.***out***.println(**this**.value);

**if** (**this**.value == value) {

targetNode = **this**;

}

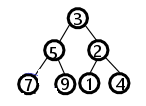
**return** targetNode;

}

}

### 测试代码

创建如下树，去树中寻找指定节点



修改“树的代码”，应当使”树”具有查找的能力

**public** **class** Demo10\_BinaryTree {

// ---其它代码---略---

**public** Demo10\_TreeNode qianXuChaZhao(**int** value) {

//从根节点开始查找

**return** rootNode.qianXuChaZhao(value);

}

**public** Demo10\_TreeNode zhongXuChaZhao(**int** value) {

**return** rootNode.zhongXuChaZhao(value);

}

**public** Demo10\_TreeNode houXuChaZhao(**int** value) {

**return** rootNode.houXuChaZhao(value);

}

}

**public** **class** Test {

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

//创建树过程---略---

Demo10\_TreeNode qianXuChaZhao = tree.qianXuChaZhao(3);

System.***out***.println("---------------------");

Demo10\_TreeNode zhongXuChaZhao = tree.zhongXuChaZhao(3);

System.***out***.println("---------------------");

Demo10\_TreeNode houXuChaZhao = tree.houXuChaZhao(3);

System.***out***.println((qianXuChaZhao == zhongXuChaZhao) == (houXuChaZhao == threeNode));

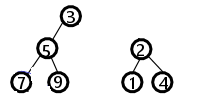
}

}

## 删除子节点

首先你得明白，删除一个子节点，其实是删除一颗子树。即，随着某个节点的删除，该节点的子节点，子节点的子节点…往下整颗子树都会被删除。

如图所示，要删除子节点2，则2整颗子树都会被删除。所以如果要删除的节点是根节点，则整颗树被删除，只有删除的节点是叶子节点才不会删除一颗子树。



执行删除也很简单，只需要使用三种遍历顺序中的任何一种顺序来遍历树，当某个节点与要删除的节点值一致，将这个节点置为null即可。当然，具体执行的时候依然要用父节点来删除子节点，至于原因，后面有思考总结。

**示例代码**

修改“树的代码”，应当使”树”具有删除的能力

**public** **class** Demo10\_BinaryTree {

// ---其它代码---略---

**public** **void** deleteNode(**int** value) {

//判断根节点是否是要被删除的节点

**if** (rootNode.getValue() == value) {

//删除根节点

rootNode = **null**;

} **else** {

rootNode.deleteNode(value);

}

}

}

**删除节点**

**public** **class** Demo10\_TreeNode {

// -- 其它代码 – 略

**public** **void** deleteNode(**int** value) {

// 自己永远是父节点

Demo10\_TreeNode parentNode = **this**;

System.***out***.println(**this**.value);

**if** (parentNode.leftNode != **null**) {

// 左子节点就是要被删除的节点

**if** (parentNode.leftNode.value == value) {

parentNode.leftNode = **null**;

**return**;

} **else** {

leftNode.deleteNode(value);

}

}

**if** (parentNode.rightNode != **null**) {

// 右子节点就是要被删除的节点

**if** (parentNode.rightNode.value == value) {

parentNode.rightNode = **null**;

**return**;

} **else** {

rightNode.deleteNode(value);

}

}

}

}

**测试代码**

**public** **class** Test {

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

//创建树过程---略---

tree.deleteNode(2);

tree.qianXuBianLi();

}

}

关于删除子节点的一点儿思考

* 一个节点无法自己删除自己，所以必须要用父节点来删除子节点。现在假设我要让一个节点自己删除自己，那么代码应该是这样的

**public** **void** deleteNode(**int** value) {

//如果当前节点就是要被删除的节点

**if** (**this**.value == value) {

//this = null; 无法将this置为null,也就无法自己删除自己

**return**;

}

}

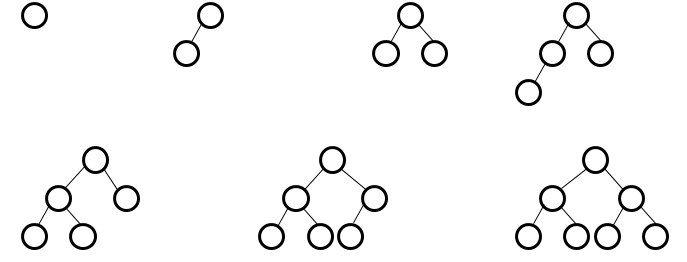
* 无论使用前序，中序，后序哪种方式遍历树，都可以把遍历的路径看成一个单向链表，在单向链表章节我们已经分析过了，要删除一个节点则必须使用父节点来删除子节点。

## 顺序存储的二叉树

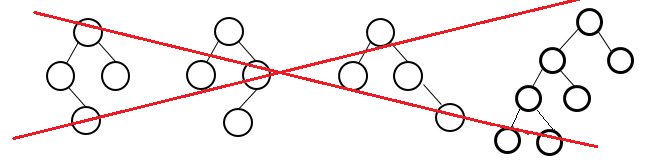
顺序存储结构是将数据放在连续的存储单元中，这里突出的重点是“连续”（注意并不是排序）。完全二叉树的特点就是从上到下，从左往右，节点之间是连续的。所以通常来讲顺序存储的二叉树就是指完全二叉树，当然，满二叉树也属于完全二叉树。

### 完全二叉树

完全二叉树生成节点顺序是从上往下，从左往右，中间不间断。并且，所有的子节点都在最后一层或者倒数第二层。图中有七颗树，这七颗树都属于完全二叉树



下面的图例则都不是完全二叉树



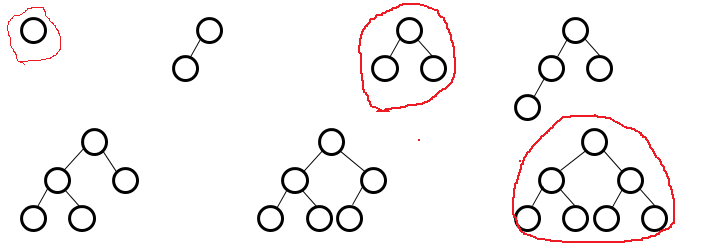
其实判断是否是完全二叉树特别简单，只需要按照从上往下，从左往右的顺序走，如果中间没有间断的节点就能一直走到最后一个叶子节点，则是完全二叉树。

### 满二叉树

满二叉树需要具备如下要求

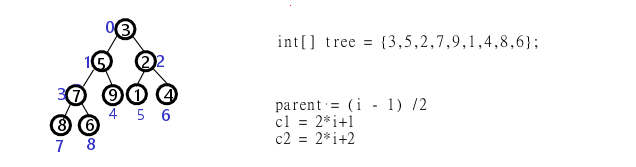
1. 所有的叶子节点都在最后一层
2. 节点的数量等于2^n – 1,其中n的值等于树的高度。

如图所示二叉树，只有红色圈中的三颗树属于满二叉树。



### 完全二叉树的遍历

对于完全二叉树的遍历我们当然可以用之前的前序/中序/后序三种顺序来遍历，然而完全二叉树比普通的树多了“连续”的特点。由于中间不会有断开的情况，基于这种特点，我们可以将完全二叉树看成一个数组，只需要按照完全二叉树的特点依次从上往下，从左往右，即可确定完全二叉树中的每个节点在数组中的角标。对于每个节点的父节点，左子节点，右子节点图中也给出了计算公式。



示例代码

完全二叉树要求从上往下，从左往右是连续的，其实这就是前序遍历的顺序，基于这样的原因，遍历二叉树应该使用前序顺序来遍历。

**public** **class** Demo11\_CompleteTree {

//注意：我们已经将完全二叉树看成了一个数组，所以树的节点都是数组中的元素

**private** **int** data[];

**public** Demo11\_CompleteTree(**int**[] data) {

**super**();

**this**.data = data;

}

**public** **void** qianXuBianLi(**int** index) {

System.***out***.println(data[index]);

**if** (2 \* index + 1 < data.length) {

// 说明具有左子节点,继续遍历左子节点

qianXuBianLi(2 \* index + 1);

}

**if** (2 \* index + 2 < data.length) {

// 说明具有左子节点,继续遍历左子节点

qianXuBianLi(2 \* index + 2);

}

}

}

测试代码

**public** **class** Test {

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

**int**[] arr = {3,5,2,7,9,1,4,8,6};

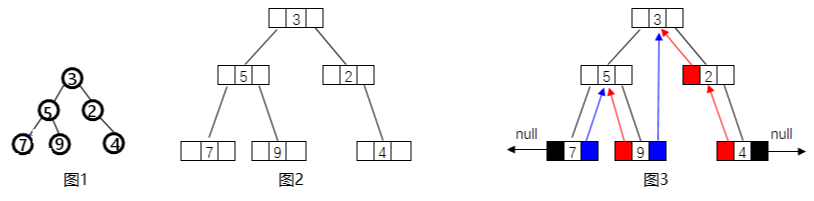
Demo11\_CompleteTree completeTree = **new** Demo11\_CompleteTree(arr);

completeTree.qianXuBianLi(1);

}

}

## 线索二叉树



### 为什么要研究线索二叉树

图1 ： 一个普通的二叉树

图2 ： 展示图1二叉树在内存中的表现形式。

当用二叉树链表作为二叉树的存储结构时，可以很方便地找到某个节点的左右子节点，但一般情况下，无法直接找到该节点在某种遍历序列中的前驱节点(前一个节点)和后继节点(后继节点)。那么如何寻找特定遍历序列中二叉树的前驱和后继节点呢？

解决的办法有三种 ：

1. 再次遍历寻找 ： 浪费时间
2. 再增设前驱节点，后继节点的变量，用于存储前驱和后继节点 ： 增加存储负担
3. 利用二叉链表中空指针域

如果某个节点的左字节点为空，则将空的左子节点指向其前驱节点；

如果某个节点的右子节点为空，则将空的右子节点指向其后继节点

这种改变指向的指针称为“线索”

加上了线索的二叉树称为线索二叉树（Threaded Binary Tree），对二叉树按某种遍历次序使其变为线索二叉树的过程叫做线索化。

图2中，二叉树中没有子节点的节点例如7，9，4等，它们的左子节点，右子节点都指向了null，造成了一定的空间浪费。我们可以让没有字节点的节点的左子节点指向它的前一个节点，右子节点指向它的后一个节点，让这个“单项链表”变成“双向链表”，例如节点9。这个过程就是线索化。

图3 ：中序线索化二叉树示例图，我们可以看到节点9的前一个节点是5，5就是9的前驱节点，9的后一个节点是3，3就是9的后继节点。感兴趣的同学可以将前序线索化二叉树，后序线索化二叉树分别给画出来。

### 线索化二叉树示例代码

为什么第一个节点的左子节点和最后一个子节点的右子节点依然指向了null ？Debug即可明了。

/\*\*

\* 线索二叉树节点

\*/

**public** **class** Demo13\_ThreadedBinaryNode {

**private** **int** data;

// 左子节点

**private** Demo13\_ThreadedBinaryNode leftNode;

// 右子节点

**private** Demo13\_ThreadedBinaryNode rightNode;

// 0：标志左子节点是指向本来的左子节点， 1 ：标志左子节点是指向前驱节点

**private** **int** leftFlag;

// 0：标志右子节点是指向本来的右子节点， 1 ：标志右子节点是指向后继节点

**private** **int** rightFlag;

}

/\*\*

\* 线索二叉树

\*/

**public** **class** Demo12\_ThreadedBinaryTree {

// 树的根节点

**private** Demo13\_ThreadedBinaryNode rootNode;

//临时存储前驱节点

**private** Demo13\_ThreadedBinaryNode preNode;

//中序线索化二叉树

**public** **void** inThreadOrder() {

inThreadOrder(rootNode);

}

/\*\*

\* 中序线索化二叉树

\* **@param** node ： 当前节点

\* debug运行下列树，代码会一目了然

\* 5

\* 7 9

\*/

**private** **void** inThreadOrder(Demo13\_ThreadedBinaryNode node) {

**if** (node == **null**) {

**return**;

}

//处理左子节点

inThreadOrder(node.getLeftNode());

// 将左子节点指向前驱节点

**if** (node.getLeftNode() == **null**) {

node.setLeftNode(preNode);

node.setLeftFlag(1);

}

// 前一个节点的后继节点指向当前节点

**if** (preNode != **null** && preNode.getRightNode() == **null**) {

preNode.setRightNode(node);

preNode.setRightFlag(1);

}

//将当前节点临时存储起来，它就是前驱节点

preNode = node;

//处理右子节点

inThreadOrder(node.getRightNode());

}

}

运行测试代码

**public** **class** Test {

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

// 创建树 ： 略

// 创建根节点 ： 略

tree.inThreadOrder();

}

}

### 线索二叉树的遍历

/\*\*

\* 线索二叉树

\*/

**public** **class** Demo12\_ThreadedBinaryTree {

// --------其它代码-----略----

//遍历线索二叉树

**public** **void** threadedIterator() {

//当前遍历的节点

Demo13\_ThreadedBinaryNode currentNode = rootNode;

**while**(currentNode != **null**) {

//找到第一个节点

**while**(currentNode.getLeftFlag() == 0) {

currentNode = currentNode.getLeftNode();

}

//打印当前节点

System.***out***.println(currentNode.getData());

//找到最后一个后继节点

**while**(currentNode.getRightFlag() == 1) {

currentNode = currentNode.getRightNode();

System.***out***.println(currentNode);

}

//更换当前节点

currentNode = currentNode.getRightNode();

}

}

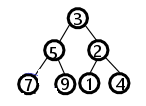
}

## 赫夫曼树

### 什么是赫夫曼树？

第一个概念 ：叶节点的带权路径

从根节点开始，到达目标叶节点所经过的节点数量乘以叶节点的权值就是叶节点的带权路径。



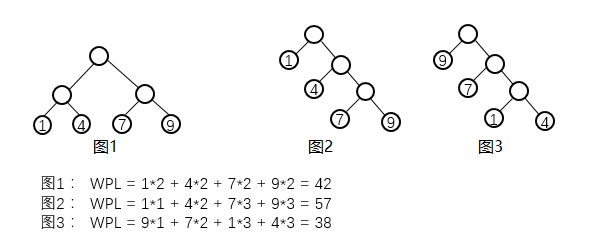
叶节点7的带权路径值为 ： 7\*2=14

叶节点9的带权路径值为 ： 9\*2=18

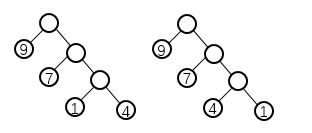
依次类推计算，即可求出每个叶节点的带权路径值。

第二个概念 ：树的带权路径长度

所有叶节点的带权路径之和就是树的带权路径长度，简称WPL。如下三张图，叶节点相同，但是WPL值却不一样。



树的带权路径长度值最小的树就是赫夫曼树，也叫最优二叉树。通过三幅图可以得出结论：赫夫曼树是叶节点权值越大距离根节点越近。注意，赫夫曼树并不是唯一的，例如下面两幅图。



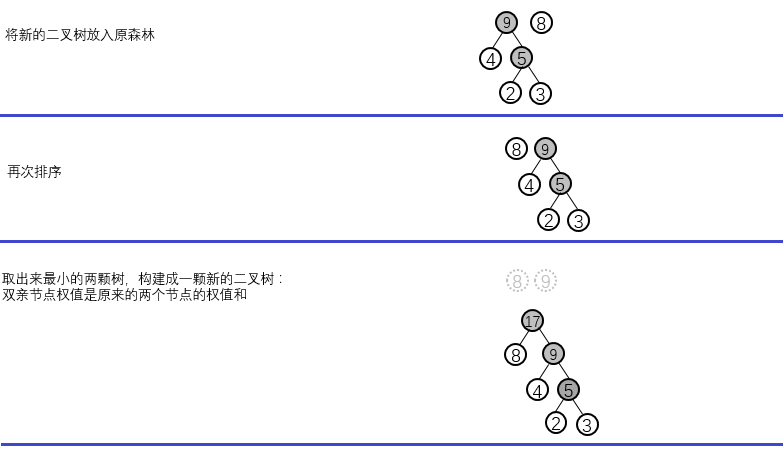
两棵树都是赫夫曼树，但是却不是一样的树，所以赫夫曼树并不是唯一的

### 如何构建赫夫曼树？

我们已经知道赫夫曼树是叶节点权值越大，距离根节点越近，通过这个特点，我们只需要根据叶节点的权值大小按顺序来构建二叉树即可构建出赫夫曼树。

构建赫夫曼树







**示例代码**

/\*\*

\* 实现Comparable接口方便排序

\* 当然也可以不实现改接口，手动排序

\* **@author** xiaoka

\*

\*/

**public** **class** Demo14\_HuffmanNode **implements** Comparable<Demo14\_HuffmanNode> {

**private** **int** value;

**public** Demo14\_HuffmanNode left;

**public** Demo14\_HuffmanNode right;

**public** Demo14\_HuffmanNode(**int** value) {

**super**();

**this**.value = value;

}

@Override

**public** **int** compareTo(Demo14\_HuffmanNode node) {

**return** **this**.value - node.value;

}

}

/\*\*

\* 赫夫曼树

\*

\* **@author** xiaoka

\*

\*/

**public** **class** Demo15\_HuffmanTree {

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

**int**[] arr = { 3, 7, 8, 29, 5, 11, 23, 14 };

Demo14\_HuffmanNode huffmanTree = *createHuffmanTree*(arr);

System.***out***.println(huffmanTree);

}

**public** **static** Demo14\_HuffmanNode createHuffmanTree(**int**[] arr) {

// 首先将数组中的每个元素都创建成一个二叉树（每颗树只有一个节点）

List<Demo14\_HuffmanNode> nodeList = **new** ArrayList<Demo14\_HuffmanNode>();

**for** (**int** value : arr) {

nodeList.add(**new** Demo14\_HuffmanNode(value));

}

// 仅当只有一颗树的时候，说明哈夫曼树创建完成

**while** (nodeList.size() > 1) {

// 排序，取出最小的两棵树，严格来讲应该自己实现排序逻辑，偷个懒…

Collections.*sort*(nodeList);

// 取出来两个最小的二叉树

Demo14\_HuffmanNode left = nodeList.remove(0);

Demo14\_HuffmanNode right = nodeList.remove(0);

// 创建一颗新的二叉树

**int** oneValue = left.getValue();

**int** twoValue = right.getValue();

Demo14\_HuffmanNode parentNode = **new** Demo14\_HuffmanNode(oneValue + twoValue);

// 给父节点添加左右子节点

parentNode.left = left;

parentNode.right = right;

// 将新的二叉树放入到集合中

nodeList.add(parentNode);

}

**return** nodeList.get(0);

}

}

### 赫夫曼树的应用

#### 赫夫曼编码

##### 定长编码

因为计算机使用二进制，所以在远程通信中，需要将传输的电文（字符串）转换成二进制进行网络传输。

假设要传输的字符串为：ABACCDA

假设编码为如下编码（注意：每个字符固定使用两个长度来表示） ：

A——00

B——01

C——10

D——11

那么字符串ABACCDA在网络中就要编码成00010010101100进行传输。这种方式就叫做定长编码方式。解码的时候也很简单，只需要每次取出来两位即可解码成原来的字符。

这种定长的编码方式固然可以解决问题，但是有一个很大的缺点 ： 浪费空间。在ASCII码表中，每一个字符都使用一个字节(8位)来表示，浪费空间更甚。基于这样的原因，我们希望可以将数据进行压缩以后再存储，再传输等等。

##### 变长编码

如果将编码设计成长度不等的二进制编码，即让要传输的字符串中出现**次数较多**的字符串采用**尽可能短**的编码，则转换的二进制字符串就能减少。例如

假设要传输的字符串为：ABACCDA

假设编码为如下编码（注意：出现次数越多，编码越短） ：

A——0

B——00

C——1

D——01

那么字符串ABACCDA在网络中就要编码成000011010进行传输。这样由原来的定长14位就压缩成了变长的9位，数据已经压缩了30%-40%。基于这样的原因，我们肯定要使用变长编码来压缩数据进行传输，存储等等。

需要指出的是利用赫夫曼树实现的赫夫曼编码则通常可以将数据压缩将近一半。实际上某些情况下，甚至可以将数据压缩到只剩下30%。

现在来对变长编码进行解码 ：

在解码过程中，可以发现，由于变长编码中的长度不确定性，导致了重码现象。比如对000011010进行解码，可以发现前4个0至少可以解码成如下三种情况。

00 00

0 0 0 0

0 00 0

所以在设计变长编码的时候一定不能出现重码现象，否则数据则无法解码还原。解决这个问题的关键点就是 ：

在设计出来的变长编码中，每个字符的编码都不能是其它任何字符的编码的前缀。即 : 每一个字符的编码都是唯一的，没有歧义。

这样就能够根据编码将数据还原。而这种编码就叫做前缀编码。

##### 哈夫曼编码/前缀编码

哈夫曼编码就是一种前缀编码，而且可以使得编码的总长最短。所以哈夫曼编码就是最优前缀码。

**实现思路**

**编码**

1. 统计字符集中每个字符在电文中出现的次数，要求次数越多，编码越短。
2. 利用哈夫曼树的特点 ：权越大的叶子离根越近。将每个字符的**次数值**作为权值，构造哈夫曼树。这样**次数值**越大的节点，路径越短。
3. 给哈夫曼树的每个分支上标记0或者1（左0右1）
   1. 所有节点的左分支标记0，
   2. 右分支标记1
4. 把从根节点到每个叶子节点的路径上的标号连接起来，作为该叶子节点所代表的字符的编码

**解码**

构造哈夫曼树

依次读入二进制码

读入0，则走向左孩子；读入1，则走向右孩子

一旦到达某叶子节点，即可译出字符

再次从根节点出发，如此往复，直到译出所有字符

**哈夫曼编码代码实现**

对字符串“abcABC”进行哈夫曼编码和解码

**public** **class** Demo17\_HuffmanCode {

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

|  |  |  |
| --- | --- | --- |
| 代码 | | 运行效果 |
| String msgString = "abcABC"; | |  |
| /\*  \* 由于计算机底层本质使用ASCII码，所以统计每个字符出现的次数本质是统计每个字节出现的次数  \* 所以要首先获取字符串的原始字节 :1 2 3 4 5 6 ... 97 98 99 ... 127  \*  \*/  **byte**[] bytes = msgString.getBytes();  System.***out***.println("原始字节 ：" + Arrays.*toString*(bytes));  System.***out***.println("原始字节长度 : " + bytes.length + "字节");  System.***out***.println("原始二进制长度 : " + bytes.length \* 8 + "比特位"); | | 原始字节 ：  [97, 98, 99, 65, 66, 67]  原始字节长度 : 6字节  原始二进制长度 : 48比特位 |
| //统计每个字节出现的次数  Map<Byte, Integer> countsMap = *countBytes*(bytes);  System.***out***.println("每个字节出现的次数 ：");  System.***out***.println(countsMap); | | 每个字节出现的次数 ：  {97=1, 65=1, 98=1, 66=1, 99=1, 67=1} |
| // 接下来应根据每个字节的出现次数来创建哈夫曼树 出现次数越多，离根节点越近 | |  |
| // 创建森林  ArrayList<Demo16\_HuffmanNode> list = *createForest*(countsMap);  System.***out***.println("创建的森林 ： ");  System.***out***.println(list); | | 创建的森林 ：  [data=97, weight=1, data=65, weight=1, data=98, weight=1, data=66, weight=1, data=99, weight=1, data=67, weight=1]  遍 |
| // 创建赫夫曼树  Demo16\_HuffmanNode tree = *createHuffmanTree*(list);  System.***out***.println("遍历哈夫曼树 ： ");  // 通过遍历来检查生成的哈夫曼树是否正确  tree.*qianXuBianLi*(tree); | | 遍历哈夫曼树 ：  data=99, weight=1  data=67, weight=1  data=97, weight=1  data=65, weight=1  data=98, weight=1  data=66, weight=1 |
| // 遍历哈夫曼树所有的叶节点，生成哈夫曼编码表  Map<Byte, String> manCode = *getHuffManCode*(tree);  System.***out***.println("哈夫曼编码表 ： ");  System.***out***.println(manCode); | | 哈夫曼编码表 ：  {97=100, 65=101, 98=110, 66=111, 99=00, 67=01} |
| // 使用哈夫曼码表来对原始数据(字节)进行重新编码 该编码就是前奏编码，是哈夫曼编码  String huffCode = *huffCode*(bytes, manCode);  System.***out***.println("字符串哈夫曼编码 : ");  System.***out***.println(huffCode);  System.***out***.println("哈夫曼编码长度 : " + huffCode.length() + " 压缩率 ：" + huffCode.length() / (bytes.length \* 8f)); | | 开始生成哈夫曼编码 :  第1次, 使用字节97生成编码 ：100  第2次, 使用字节98生成编码 ：110  第3次, 使用字节99生成编码 ：00  第4次, 使用字节65生成编码 ：101  第5次, 使用字节66生成编码 ：111  第6次, 使用字节67生成编码 ：01  字符串哈夫曼编码 :  1001100010111101  哈夫曼编码长度 : 16 压缩率 ：0.33333334 |
| /\*  \* 验证哈夫曼编码是否正确  \* 每个字符出现的次数 ✖ 每个字符的编码长度 ，  \* 累加的和就是哈夫曼编码的长度  \*  \*/  **int** sum = 0;  **for** (Map.Entry<Byte, String> entry : manCode.entrySet()) {  String code = entry.getValue();  Integer cout = countsMap.get(entry.getKey());  sum = sum + Integer.*valueOf*(code.length()) \* cout;  }  System.***out***.println("验证哈夫曼编码 ： ");  System.***out***.println("计算值 ： " + sum);  System.***out***.println("生成的值 ：" + huffCode.length()); | | 验证哈夫曼编码 ：  计算值 ： 16  生成的值 ：16 |
| //到这里应该解码了，为什么不是进行解码而是往下进行呢？  //假设现有字符串"小钻风"，使用UTF-8编码，应该是3byte \* 3 = 9字节 \* \* 8=72比特位  //假设"小钻风" 的哈夫曼编码是 "1111000011110001"，  //如果直接存储字符串类型的哈夫曼编码，那么每一个字符"1"和每一个字符"0"都需要一个字节，  //这样算下来，字符串"1111 0000 1111 0001"就总共需要 16字节 \* 8 = 128个比特位  //所以，我们应该将字符串"1111 0000 1111 0001"转换成字节类型，  //这样就只需要 两个字节 \*8 = 16比特位即可。  //所以，我们的做法是将每八位截取出来，转换成一个字节，这才是真正需要的编码数据  **byte**[] realCode = *createRealCode*(huffCode);  System.***out***.println("=========编码完成============="); | | 开始生成哈夫曼字节 ：  第1次截取 ：字符串编码 ： 10011000 哈夫曼字节 : -104  第2次截取 ：字符串编码 ： 10111101 哈夫曼字节 : -67  生成的哈夫曼字节长度 : 2  =========编码完成============= |
| System.***out***.println("=========解码阶段============="); | | |
| //编码阶段是将二进制字符串变成十进制数字  //1111 0000 → -16(十进制数字)  //1111 0001 → -15(十进制数字)  //所以解码阶段就是将十进制数字变成二进制字符串  //-16(十进制数字) → 1111 0000  //-15(十进制数字) → 1111 0001  String decode = *decode*(realCode); | | 第1次生成 ：哈夫曼字节 :-104 生成的值为 ： 10011000  第2次生成 ：哈夫曼字节 :-67 生成的值为 ： 10111101  解码成的字符串哈夫曼编码 :  1001100010111101  验证解码的哈夫曼编码 : 16 |
| // 根据字符串类型的哈夫曼编码和编码表解码出来原始字节  **byte**[] byteString = *decode*(decode, manCode); | 开始解码工作 :  1  10  100  第1次解码 ： 100 对应 ：97 解码出来字符 : a  1  11  110  第2次解码 ： 110 对应 ：98 解码出来字符 : b  0  00  第3次解码 ： 00 对应 ：99 解码出来字符 : c | 1  10  101  第4次解码 ： 101 对应 ：65 解码出来字符 : A  1  11  111  第5次解码 ： 111 对应 ：66 解码出来字符 : B  0  01  第6次解码 ： 01 对应 ：67 解码出来字符 : C  编码总长 (缺少末尾字节)：16 已使用长度 :16 |
| //根据原始字节还原数据  System.***out***.println(**new** String(byteString)); | | abcABC |

}

}

1. 创建节点类

**public** **class** Demo16\_HuffmanNode **implements** Comparable<Demo16\_HuffmanNode> {

// 节点权值

Byte data;

// 节点的路径（当前权值的出现次数）

**int** weight;

Demo16\_HuffmanNode left;

Demo16\_HuffmanNode right;

@Override

**public** **int** compareTo(Demo16\_HuffmanNode node) {

// 不是按照权值排序，而是按照出现的次数排序（路径）

**return** **this**.weight - node.weight;

}

//其它代码 略

}

1. 统计每个字符出现的次数 ：

**private** **static** Map<Byte, Integer> countBytes(**byte**[] bytes) {

Map<Byte, Integer> countsMap = **new** HashMap<Byte, Integer>();

**for** (**byte** b : bytes) {

Integer tempCount = countsMap.get(b);

**if** (tempCount == **null**) {

countsMap.put(b, 1);

} **else** {

countsMap.put(b, tempCount + 1);

}

}

**return** countsMap;

}

1. 创建森林，为创建赫夫曼树做准备

**private** **static** ArrayList<Demo16\_HuffmanNode> createForest(Map<Byte, Integer> countsMap) {

ArrayList<Demo16\_HuffmanNode> list = **new** ArrayList<>();

Set<Entry<Byte, Integer>> entrySet = countsMap.entrySet();

**for** (Entry<Byte, Integer> entry : entrySet) {

Byte data = entry.getKey();

Integer count = entry.getValue();

Demo16\_HuffmanNode node = **new** Demo16\_HuffmanNode(data, count);

list.add(node);

}

**return** list;

}

1. 创建赫夫曼树，仅当森林中只有一颗树说明赫夫曼树创建完成

**private** **static** Demo16\_HuffmanNode createHuffmanTree(ArrayList<Demo16\_HuffmanNode> list) {

**while** (list.size() > 1) {

// 排序森林

Collections.*sort*(list);

// 获取出现次数最少的两个节点

Demo16\_HuffmanNode left = list.remove(0);

Demo16\_HuffmanNode right = list.remove(0);

// 创建新的二叉树

Demo16\_HuffmanNode parentNode = **new** Demo16\_HuffmanNode(left.weight + right.weight);

parentNode.left = left;

parentNode.right = right;

// 放入森林

list.add(parentNode);

}

**return** list.remove(0);

}

1. 前序遍历

**public** **static** **void** qianXuBianLi(Demo16\_HuffmanNode node) {

**if** (node == **null**) {

**return**;

}

**if** (node.data != **null**) {

System.***out***.println(node);

}

*qianXuBianLi*(node.left);

*qianXuBianLi*(node.right);

}

1. 获取哈夫曼编码表

**private** **static** Map<Byte, String> getHuffManCode(Demo16\_HuffmanNode tree) {

**if** (tree == **null**) {

**return** **null**;

}

// 记录已经经过的路径

StringBuilder pathBuilder = **new** StringBuilder();

//存储哈夫曼码表

Map<Byte, String> huffCodeMap = **new** HashMap<Byte, String>();

*getPath*(tree.left, "0", pathBuilder, huffCodeMap);

*getPath*(tree.right, "1", pathBuilder, huffCodeMap);

**return** huffCodeMap;

}

**private** **static** **void** getPath(Demo16\_HuffmanNode node, String code, StringBuilder pathBuilder,

Map<Byte, String> huffCodeMap) {

// 拼接之前已经经过的路径

StringBuilder path = **new** StringBuilder(pathBuilder);

path.append(code);

**if** (node.data == **null**) {

// 说明不是叶子节点，继续递归

*getPath*(node.left, "0", path, huffCodeMap);

*getPath*(node.right, "1", path, huffCodeMap);

} **else** {

// 这是一个叶子节点

huffCodeMap.put(node.data, path.toString());

}

}

1. 根据哈夫曼码表编译出来哈夫曼编码

**private** **static** String huffCode(**byte**[] bytes, Map<Byte, String> manCode) {

// 拼接原始数据编码以后的编码

StringBuilder tempCharsetSb = **new** StringBuilder();

ArrayList<String> list = **new** ArrayList<String>();

/\*\*

\* java中对于字符串的长度有限制吗? 是否可以考虑将每8个字节放入到一个集合中?

\*/

System.***out***.println("开始生成哈夫曼编码 : ");

**for** (**int** i = 0; i < bytes.length; i++) {

**byte** temp = bytes[i];

// 获取每个字节对应的哈夫曼编码

String tempCharset = manCode.get(temp);

System.***out***.println("第" + (i + 1) + "次, 使用字节" + temp + "生成编码 ：" + tempCharset);

tempCharsetSb.append(tempCharset);

}

**return** tempCharsetSb.toString();

}

1. 将字符串类型的哈夫曼编码转换成字节类型（二进制转换成十进制）

**private** **static** **byte**[] createRealCode(String huffCode) {

System.***out***.println("开始生成哈夫曼字节 ： ");

**int** length = huffCode.length();

**if** (length % 8 == 0) {

length = length / 8;

} **else** {

length = length / 8 + 1;

}

**byte**[] realHuffBytes = **new** **byte**[length];

**int** index = 0;

**int** codelength = huffCode.length();

**for** (**int** i = 0; i < codelength; i += 8) {

String tempByte;

**int** end = i + 8;

**if** (end > codelength) {

tempByte = huffCode.substring(i);

} **else** {

tempByte = huffCode.substring(i, end);

}

// 将tempByte作为2进制，转换成十进制

**byte** realByte = (**byte**) Integer.*parseInt*(tempByte, 2);

realHuffBytes[index] = realByte;

index++;

System.***out***.println("第" + index + "次截取 ：" + "字符串编码 ： " + tempByte + " 哈夫曼字节 : " + realByte);

}

System.***out***.println("生成的哈夫曼字节长度 : " + realHuffBytes.length);

**return** realHuffBytes;

}

1. 将字节哈夫曼编码还原成字符类型的编码（十进制转换成二进制）

**private** **static** String decode(**byte**[] encode) {

StringBuilder stringBuilder = **new** StringBuilder();

String byteString;

**for** (**int** i = 0; i < encode.length; i++) {

**int** temp = encode[i];

// 让所有数字都变成8位的长度，否则转换成二进制的正数不够8位

temp |= 256;

// 从十进制数字变成二进制字符串

byteString = Integer.*toBinaryString*(temp);

// 由于使用int类型的十进制转换成的二进制，所以转换成的字符串是32位的，

// 只需要八位，所以截取后八位即可

byteString = byteString.substring(byteString.length() - 8);

// 拼接出来字符串类型的哈夫曼编码

stringBuilder.append(byteString);

System.***out***.println("第" + (i + 1) + "次生成 ：" + "哈夫曼字节 :" + temp + " 生成的值为 ： " + byteString);

}

String string = stringBuilder.toString();

System.***out***.println("解码成的字符串哈夫曼编码 : ");

System.***out***.println(string);

System.***out***.println("验证解码的哈夫曼编码 : ");

System.***out***.println(string.length());

**return** string;

}

1. 将字符串类型的哈夫曼编码还原成原始字节

@SuppressWarnings("unlikely-arg-type")

**private** **static** **byte**[] decode(String huffCode, Map<Byte, String> huffCodeMap) {

System.***out***.println("开始解码工作 : ");

HashMap<String, Byte> map = **new** HashMap<String, Byte>();

**for** (Map.Entry<Byte, String> entry : huffCodeMap.entrySet()) {

map.put(entry.getValue(), entry.getKey());

}

ArrayList<Byte> byteList = **new** ArrayList<Byte>();

**int** end = 0, start = 0;

**int** count = 1;

**int** length = huffCode.length();

**int** realLength;

String baWei = huffCode.substring(length - 8);

**byte** flag = (**byte**) Integer.*parseInt*(baWei, 2);

**if** (flag < 0) {

realLength = length;

} **else** {

realLength = length - 8;

}

**while** (end <= realLength) {

String code = huffCode.substring(start, end);

Byte byte1 = map.get(code);

**if** (byte1 != **null**) {

System.***out***.println(code);

byteList.add(byte1);

start = end;

**byte** temp = byte1;

System.***out***.println("第" + (count) + "次解码 ： " + code + " 对应 ：" + byte1 + " 解码出来字符 : " + ((**char**) temp));

count++;

} **else** {

System.***out***.println(code);

}

end++;

}

System.***out***.println("编码总长 (缺少末尾字节)：" + huffCode.length() + " 已使用长度 :" + start);

// 处理末尾编码

**if** (start < realLength) {

System.***out***.println("进入IF");

String code = huffCode.substring(start, realLength);

**for** (**int** i = 1; i < 8; i++) {

StringBuilder zeroCode = **new** StringBuilder(code);

String substring = huffCode.substring(length - i);

zeroCode.append(substring);

Byte byteCode = map.get(zeroCode.toString());

**if** (byteCode != **null**) {

byteList.add(byteCode);

**break**;

}

}

}

**byte**[] bytes = **new** **byte**[byteList.size()];

**for** (**int** i = 0; i < bytes.length; i++) {

bytes[i] = byteList.get(i);

}

**return** bytes;

}

问题 :

* 为什么哈夫曼编码能够保证是前缀编码？

因为没有哪个叶节点是其它某个叶节点的双亲节点，所以每个叶节点的编码就不可能是其它叶节点编码的前缀

* 为什么哈夫曼编码能够保证字符编码的总长最短？

因为哈夫曼树的带权路径长度最短，故字符编码的总长度最短。

##### 文件压缩和解压缩

//压缩文件 ： 将原始字节编码成哈夫曼字节，同时将哈夫曼字节和哈夫曼编码写入到压缩文件中

**private** **static** **void** zipFile(String srcfile, String desFile) **throws** Exception {

FileInputStream inputStream = **new** FileInputStream(**new** File(srcfile));

**byte**[] orignalBytes = **new** **byte**[inputStream.available()];

inputStream.read(orignalBytes);

Map<Byte, Integer> countsMap = *countBytes*(orignalBytes);

ArrayList<Demo16\_HuffmanNode> list = *createForest*(countsMap);

Demo16\_HuffmanNode tree = *createHuffmanTree*(list);

Map<Byte, String> manCode = *getHuffManCode*(tree);

String huffCode = *huffCode*(orignalBytes, manCode);

**byte**[] realCode = *createRealCode*(huffCode);

FileOutputStream outputStream = **new** FileOutputStream(**new** File(desFile));

ObjectOutputStream oos = **new** ObjectOutputStream(outputStream);

oos.writeObject(realCode);

//将码表也写入压缩文件，以备解压使用

oos.writeObject(manCode);

oos.close();

outputStream.close();

inputStream.close();

}

//解压缩文件 ： 读取压缩文件中的哈夫曼码表和哈夫曼字节，进行解码还原

**private** **static** **void** unZip(String src, String des) **throws** Exception {

FileInputStream inputStream = **new** FileInputStream(**new** File(src));

ObjectInputStream ois = **new** ObjectInputStream(inputStream);

**byte**[] datas = (**byte**[]) ois.readObject();

Map<Byte, String> charSet = (Map<Byte, String>) ois.readObject();

String huffCode = *decode*(datas);

datas = *decode*(huffCode, charSet);

FileOutputStream os = **new** FileOutputStream(**new** File(des));

os.write(datas);

os.close();

ois.close();

inputStream.close();

}

## 二叉排序树

我们知道，数组结构查找元素快，增删元素慢，而链表结构则是增删元素快，查找元素慢。那么有没有一种结构就是增删元素效率不是那么的低，查找元素效率也不是那么的低呢？答案是肯定的。

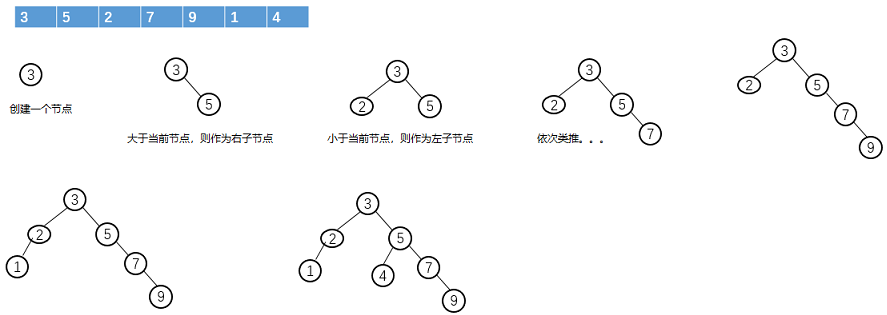
### 二叉排序树简介

二叉排序树，又叫二叉查找树，二叉搜索树，是一种兼具了数组和链表两种结构的优点的一种树形结构。二叉排序树无论在增删还是查询元素时，效率都是比较高的。

二叉排序树的特点 ：

* 当前节点的权值大于左子节点的权值
* 当前节点的权值小于右子节点的权值
* 左右子树也分别为二叉排序树

如图所示 ： 使用数组中的元素来创建一个二叉排序树。



### 创建二叉排序树

二叉排序树节点类

**public** **class** Demo18\_BinarySortNode {

Demo18\_BinarySortNode left;

Demo18\_BinarySortNode right;

**int** value;

}

二叉排序树类

**public** **class** Demo19\_BinarySortTree {

Demo18\_BinarySortNode root;

}

测试类

**public** **class** Test {

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

**int**[] arr = { 3, 5, 2, 7, 9, 1, 4 };

Demo19\_BinarySortTree tree = **new** Demo19\_BinarySortTree();

**for** (**int** i : arr) {

//添加节点

tree.addNode(**new** Demo18\_BinarySortNode(i));

}

//中序遍历，检验创建的结果

tree.midShow();

}

}

**public** **class** Demo19\_BinarySortTree {

//添加节点

**public** **void** addNode(Demo18\_BinarySortNode node) {

**if** (root == **null**) {

root = node;

**return**;

}

root.addNode(node);

}

//中序遍历排序二叉树

**public** **void** midShow() {

**if** (root == **null**) {

System.***out***.println(root);

**return**;

}

root.midShow();

}

}

**public** **class** Demo18\_BinarySortNode {

// 添加节点

**public** **void** addNode(Demo18\_BinarySortNode node) {

**if** (node == **null**) {

**return**;

}

**int** temp = node.value;

**if** (temp < value) {

**if** (left == **null**) {

left = node;

} **else** {

left.addNode(node);

}

} **else** {

**if** (right == **null**) {

right = node;

} **else** {

right.addNode(node);

}

}

}

//中序遍历二叉树

**public** **void** midShow() {

**if** (left != **null**) {

left.midShow();

}

System.***out***.println(value);

**if** (right != **null**) {

right.midShow();

}

}

}

### 查找节点

**public** **class** Test {

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

// ----其它代码 ---略---

Demo18\_BinarySortNode node = tree.searchValue(5);

System.***out***.println(node);

}

}

**public** **class** Demo19\_BinarySortTree {

// 查找节点值

**public** Demo18\_BinarySortNode searchValue(**int** value) {

**if** (root == **null**) {

**return** **null**;

}

**return** root.searchValue(value);

}

}

**public** **class** Demo18\_BinarySortNode {

//查找节点

**public** Demo18\_BinarySortNode searchValue(**int** value) {

**if** (**this**.value == value) {

**return** **this**;

}

**if** (**this**.value > value && left != **null**) {

**return** left.searchValue(value);

}

**if** (right != **null** && **this**.value < value) {

**return** right.searchValue(value);

}

**return** **null**;

}

}

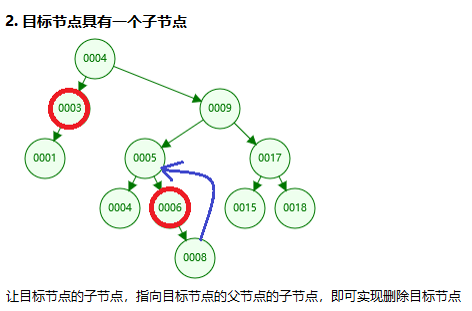
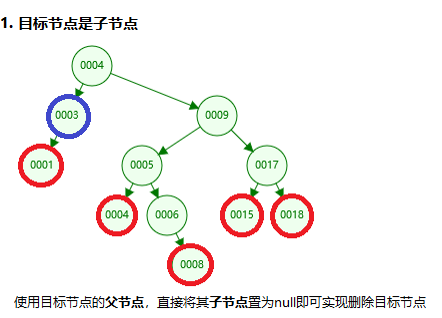
### 删除节点

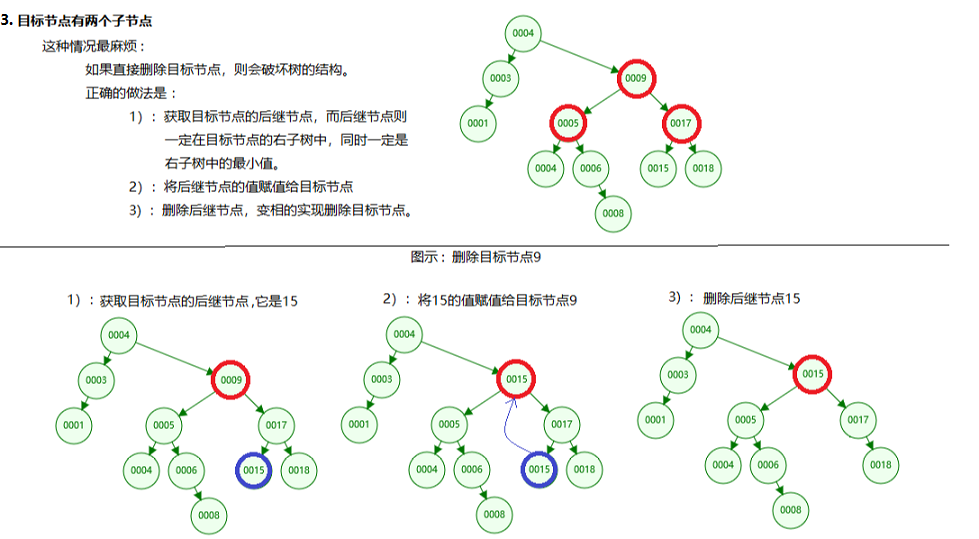
二叉排序树的删除相对来讲比较麻烦，因为它涉及到的情况比较多。总得来说，分为如下三种情况。

目标节点 ： 将要被删除的节点。

1. 目标节点是子节点
2. 目标节点具有一个子节点
3. 目标节点具有两个子节点

这三种情况的实现思路分别是：





删除节点示例代码

**public** **class** Demo18\_BinarySortNode {

//查找节点

**public** Demo18\_BinarySortNode searchValue(**int** value) {}

//查找父节点

**public** Demo18\_BinarySortNode searchParentNode(**int** value) {

**if** ((left != **null** && left.value == value) || (right != **null** && right.value == value)) {

**return** **this**;

}

**if** (**this**.value > value) {

**if** (left != **null**) {

**return** left.searchParentNode(value);

}

}

**if** (**this**.value < value) {

**if** (right != **null**) {

**return** right.searchParentNode(value);

}

}

**return** **null**;

}

}

**public** **class** Demo19\_BinarySortTree {

// 删除节点

**public** **void** deleteNode(**int** value) {

**if** (root == **null**) {

**return**;

}

// 找到需要删除的目标节点

Demo18\_BinarySortNode targetNode = searchValue(value);

// 目标节点不存在

**if** (targetNode == **null**) {

**return**;

}

// 找到目标节点的父节点

Demo18\_BinarySortNode parentNode = searchParentNode(value);

// 1.要删除的目标节点是子节点

**if** (targetNode.left == **null** && targetNode.right == **null**) {

//3. 目标节点具有两个子节点

} **else** **if** (targetNode.left != **null** && targetNode.right != **null**) {

} **else** {

//2. 目标节点有一个子节点

// 目标节点只有左子节点

**if** (targetNode.left != **null**) {

**return**;

}

// 目标节点只有右子节点

**if** (targetNode.right != **null**) {

}

}

}

}

1. **目标节点是子节点**

|  |  |
| --- | --- |
| **if** (targetNode.left == **null** && targetNode.right == **null**) {  **if** (parentNode != **null**) {  **if** (parentNode.left == **null**) { //目标节点8，父节点6  // 目标节点是父节点的右子节点，让父节点的右子节点置为null就是删除目标节点  parentNode.right = **null**;  } **else** **if** (parentNode.right == **null**) { //目标节点1，父节点3  // 目标节点是父节点的左子节点  parentNode.left = **null**;  } **else** {  **if** (parentNode.left.value == value) { //目标节点15或者18，父节点17  // 目标节点是父节点的左子节点  parentNode.left = **null**;  } **else** {  parentNode.right = **null**;  }  }  }  } | C:\Users\xiaoka\Desktop\图片1.png |

1. **目标节点具有一个子节点**

**2.1处理右子节点**

|  |  |  |
| --- | --- | --- |
| **else** {  //2. 目标节点有一个子节点  // 目标节点只有左子节点  **if** (targetNode.left != **null**) {  **return**;  }  // 目标节点只有右子节点  **if** (targetNode.right != **null**) { //目标节点6，父节点是5  // 目标节点是父节点的左子节点  **if** (parentNode.left != **null** && parentNode.left.value == value) {  parentNode.left = targetNode.right;  } **else** {  parentNode.right = targetNode.right;  }  }  } | | C:\Users\xiaoka\Desktop\图片2.png |
| 如果树的结构是这样子的，而且要删除的还是根节点，那么以前的方式则行不通，因为根节点没有父节点。所以要换一种实现的思路。  C:\Users\xiaoka\Desktop\a.png | | |
| // 目标节点只有右子节点  **if** (targetNode.right != **null**) {  // 要删除的节点是根节点  **if** (parentNode == **null**) { // 目标节点4，父节点为null  、 //删除右子树中的最小节点，返回获取被删除节点的值  **int** minNode = deleteMinNode(targetNode.right);  targetNode.value = minNode;  } **else** {  // 目标节点是父节点的左子节点 //目标节点6，父节点是5  **if** (parentNode.left != **null** && parentNode.left.value == value) {  parentNode.left = targetNode.right;  } **else** {  parentNode.right = targetNode.right;  }  }  } |  | |
| **2.2处理左子节点** | | |
| **else** {  //2. 目标节点有左子节点  // 目标节点只有左子节点  **if** (targetNode.left != **null**) { //目标节点3，父节点为4  // 目标节点是父节点的左子节点  **if** (parentNode.left != **null** && parentNode.left.value == value) {  parentNode.left = targetNode.left;  } **else** {  // 目标节点是父节点的右子节点  parentNode.right = targetNode.left;  }  **return**;  }  // 目标节点只有右子节点  **if** (targetNode.right != **null**) {  }  } | C:\Users\xiaoka\Desktop\图片2.png | |
| 与删除右子节点的情况相仿，如果树的结构只有左子树，同时要删除的节点是根节点，同样要换一种思路来实现。 | | |
| // 目标节点有一个子节点  // 目标节点只有左子节点  **if** (targetNode.left != **null**) {  **if** (parentNode == **null**) {  // 左子树应该删除最大的节点  **int** minNode = deleteMaxNode(targetNode.left);  targetNode.value = minNode;  } **else** {  // 目标节点是父节点的左子节点  **if** (parentNode.left != **null** && parentNode.left.value == value) {  parentNode.left = targetNode.left;  } **else** {  // 目标节点是父节点的右子节点  parentNode.right = targetNode.left;  }  }  **return**;  }  // 目标节点只有右子节点  **if** (targetNode.right != **null**) {  }  } |  | |

1. **目标节点具有两个子节点**

**3.1目标节点不是根节点**

|  |
| --- |
|  |
| **else** **if** (targetNode.left != **null** && targetNode.right != **null**) {  /\* 当目标节点具有两个子节点的时候，  \* 删除目标节点的后继节点，这个后继节点一定在目标节点的右子树中，并且是右子树中的最左边，一定是最小值节点  \* 将后继节点的值赋值给目标节点，即可变相实现删除目标节点  \*/  **int** minValue = deleteMinNode(targetNode.right);  // 替换值即可  targetNode.value = minValue;  } **else** { |
| **3.2目标节点是根节点** |
| /\*  \* 当目标节点是5，第一次找到的是根节点5，而它有左右子节点，这样就会进入无限递归  \* 解决思路 ： 临时将根节点5的值随意更改，然后就会删除掉子节点5，最后再将根节点的值恢复  \* 5  \* 3 6  \* 5 7  \*  \*/  // 要删除的节点就是根节点，同时要删除的节点值与根节点值一样  **if** (parentNode == **null** && targetNode.value == root.value) {  // 随意更改一下根节点的值  root.value = value - 1;  // 删除另外一个和根节点值一样的节点  // 恢复根节点的值  **int** minValue = deleteMinNode(root.right);  root.value = minValue;  } **else** {  **int** minValue = deleteMinNode(targetNode.right);  // 替换值即可  targetNode.value = minValue;  } |

**测试代码**

**public** **class** Test {

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

**int**[] arr = { 4, 3, 9, 5, 6, 1, 8, 17, 18, 15, 4, 15 };

Demo19\_BinarySortTree tree = **new** Demo19\_BinarySortTree();

**for** (**int** i : arr) {

tree.addNode(**new** Demo18\_BinarySortNode(i));

}

**for** (**int** i = 20; i > 0; i--) {

tree.deleteNode(i);

tree.midShow();

System.***out***.println("---" + i);

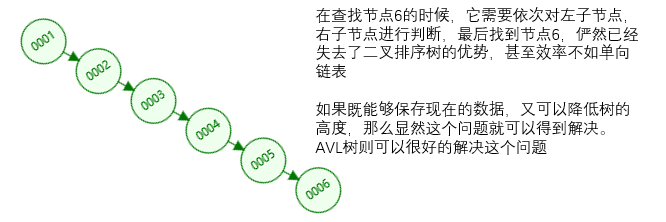
}

}

}

## AVL树

### 二叉排序树的问题

当二叉排序树是这样的结构的时候，我们发现，二叉排序树已经失去了它的查找性能。而使用平衡二叉树则可以保证效率问题。  


### AVL树简介

AVL是俄国科学家Adelson-Velskii and Lanids的首字母组合。在[计算机科学](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6/9132)中，**AVL树**是最先发明的自平衡二叉查找树。在AVL树中任何节点的两个子树的高度最大差别为1，所以它也被称为**高度平衡树**。增加和删除可能需要通过一次或多次[树旋转](https://baike.baidu.com/item/%E6%A0%91%E6%97%8B%E8%BD%AC)来重新平衡这个树。

**AVL树的特点**

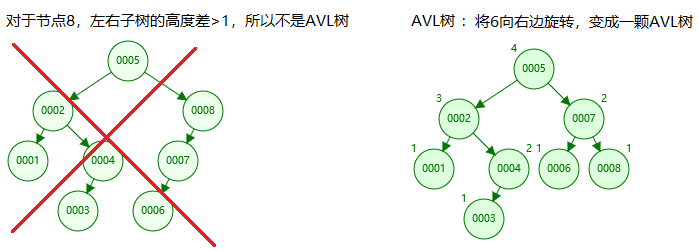
AVL树本质上还是一棵二叉搜索树，它的特点是：

1.本身首先是一棵二叉搜索树。

2.带有平衡条件：每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1。

也就是说，AVL树，本质上是带了平衡功能的二叉排序树。

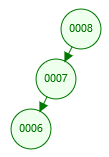
图示AVL树



### AVL树的实现

#### 右旋转

假设现有一个二叉排序树如此，现在要通过旋转将它变成一颗平衡二叉树。



**思路和示例代码**

|  |  |
| --- | --- |
| /\*  \* 平衡二叉树节点  \*/  **public** **class** Demo20\_AVLNode {  Demo20\_AVLNode left;  Demo20\_AVLNode right;  **int** value;  } | /\* 平衡二叉树 \*/  **public** **class** Demo20\_AVLTree {  Demo20\_AVLNode root;  **public** **void** addNode(Demo20\_AVLNode node) {  **if** (root == **null**) {  root = node;  **return**;  }  root.addNode(node);  }  } |

每次添加完一个节点以后都要通过左右子树的高度来检查，是否为平衡二叉树

|  |  |
| --- | --- |
| // 添加节点  **public** **void** addNode(Demo20\_AVLNode node) {  // AVL树依然是二叉排序树  **int** tempValue = node.value;  **if** (tempValue < **this**.value) {  **if** (left == **null**) {  left = node;  } **else** {  left.addNode(node);  }  } **else** {  **if** (right == **null**) {  right = node;  } **else** {  right.addNode(node);  }  }  // 节点添加完成以后，需要检查是否是一颗AVL树，如果不是，则需要进行节点的旋转  **int** calcLeftHeight = calcLeftHeight();  **int** calcRightHeight = calcRightHeight();  //左子树高度大于右子树高度，那么就应该右旋转  **if** (calcLeftHeight - calcRightHeight > 1) {  rightRotate();  }  } | // 计算当前节点的高度值  **public** **int** caclHeight() {  **int** leftHeight, rightHeight;  **if** (left == **null**) {  // 注意 ： 这并不是求子节点的高度，而是当前节点的高度  // 子树为null,那么当前节点高度已经是1  leftHeight = 1;  } **else** {  leftHeight = left.caclHeight() + 1;  }  **if** (right == **null**) {  rightHeight = 1;  } **else** {  rightHeight = right.caclHeight() + 1;  }  **return** (leftHeight > rightHeight ? leftHeight : rightHeight);  //return Math.max(left == null ? 0 : left.caclHeight(),  // right == null ? 0 : right.caclHeight()) + 1;  } |
| //左子树高度大于右子树高度，那么就应该右旋转  **if** (calcLeftHeight - calcRightHeight > 1) {  rightRotate();  } |  |
| **private** **void** rightRotate() {  //1. 使用当前节点的值创建一个新节点  Demo20\_AVLNode newNode = **new** Demo20\_AVLNode(value);  } |  |
| **private** **void** rightRotate() {  //2. 将当前节点的右子树设置给新节点的右子树  newNode.right = right;  } |  |
| **private** **void** rightRotate() {  //3. 将当前节点左子树的右子树设置给新节点的左子树  newNode.left = left.right;  } |  |
| **private** **void** rightRotate() {  //4. 将当前节点左子节点的值赋值给当前节点  **this**.value = left.value;  } |  |
| **private** **void** rightRotate() {  //5. 删除当前节点的左子节点  //（将当前节点左子节点的左子节点设置给当前节点的左子节点）  **this**.left = left.left;  } |  |
| **private** **void** rightRotate() {  //6. 把新节点设置为当前节点的右子节点  **this**.right = newNode;  } |  |
| **private** **void** rightRotate() {  Demo20\_AVLNode newNode = **new** Demo20\_AVLNode(value);  //2. 将当前节点的右子树设置给新节点的右子树  newNode.right = right;  //3. 将当前节点左子树的右子树设置给新节点的左子树  newNode.left = left.right;  **this**.value = left.value;  **this**.left = left.left;  **this**.right = newNode;  } |  |

**思路/代码分析（一） ： 为什么需要第2，3步？**

可以看到，我们的第2，3步骤是给新节点分别设置左，右子节点，但是上次却是设置了两个空节点，那么2，3步骤到底是否有必要进行设置呢？

答案是肯定的。

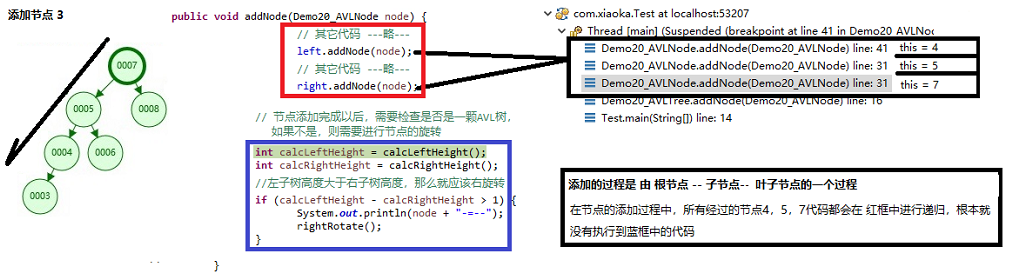
当AVL树是下图所示的时候，就可以发现，2，3步骤其实是将之前的某些节点“挪动”到新节点下，不再是挪动“空节点了”，最终以达到平衡整颗二叉树的目的。

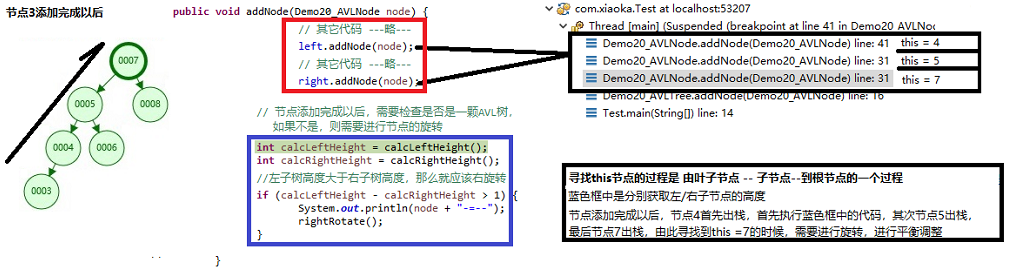
|  |
| --- |
|  |
|  |

**思路/代码分析（二） ： 什么时候进行了每颗子树的高度检查—this节点的寻找**

要想保证是一颗平衡二叉树，则必须在每个节点添加完成后，再对二叉树中的每个节点进行左右子树的高度检查。左右子树高度差（平衡因子）大于1的节点就是this节点。

这是一次充分理解递归的过程。





**总结**

旋转的过程就是操作新节点的过程

1. 使用this节点创建新节点
2. 给新节点的右子树赋值 ： 将this的右子树赋值给新节点右子树，用于保存this的右子树数据
3. 给新节点的左子树赋值 ： 将this的左子节点的右子树赋值给新节点的左子树，用于保存另外的数据，之所以这样是因为this的左子节点将要被删除
4. 删除this节点的左子节点 ： 刚刚已经将this节点左子节点的右子树给保存到了新节点中了，所以删除左子节点以后，左子节点的右子树数据依然存在
5. 使用新节点替代this的右子节点，整个过程完成

#### 左旋转

使用总结的思路对下列二叉排序树进行左旋转，使之高度平衡。

|  |  |  |
| --- | --- | --- |
|  | **public** **void** addNode(Demo20\_AVLNode node) {  // 其它代码---略---  // 如果右子树高度大于左子树高度，那么就应该左旋转  **if** (calcRightHeight - calcLeftHeight > 1) {  leftRotate();  }  } | // 左旋转  **private** **void** leftRotate() {  Demo20\_AVLNode newNode = **new** Demo20\_AVLNode(value);  newNode.left = **this**.left;  newNode.right = **this**.right.left;  **this**.value = right.value;  **this**.right = right.right;  **this**.left = newNode;  } |

#### 双旋转

|  |  |
| --- | --- |
| **public** **void** addNode(Demo20\_AVLNode node) {  // 其它代码 -- 略 --  // 左子树高度大于右子树高度，那么就应该右旋转  **if** (calcLeftHeight - calcRightHeight > 1) {  //双旋转  **if** (left != **null** && left.calcRightHeight() > left.calcLeftHeight()) {  // 首先使用left作为根节点，进行左旋转  left.leftRotate();  // 让后再让整个this进行右旋转  rightRotate();  } **else** {  //单旋转  rightRotate();  }  }  // 如果右子树高度大于左子树高度，那么就应该左旋转  **if** (calcRightHeight - calcLeftHeight > 1) {  //双旋转  **if** (right != **null** && right.calcLeftHeight() > right.calcRightHeight()) {  // 首先使用right作为根节点，进行右旋转  right.rightRotate();  // 然后再让整个this进行左旋转  leftRotate();  } **else** {  //单旋转  leftRotate();  }  }  } |  |

#### 总结

在平衡二叉树的旋转中，总共分为如下四种情况

|  |  |  |  |
| --- | --- | --- | --- |
| **单旋转** | | **双旋转** | |
| LL型 | RR型 | LR型 | RL型 |
| 左子树高 ：  向右顺时针旋转调整平衡 | 右子树高 ：  向左逆时针旋转调整平衡 | 左子树的右子树高 ：  先让左子树的右子树向左旋转，整体变成LL形  再整体向右顺时针旋转调整平衡 | 右子树的左子树高 ：  先让右子树的左子树向右旋转，整体变成RR型  再整体向左逆时针旋转调整平衡 |

## B/B+树

### 2-3树/2-3-4树

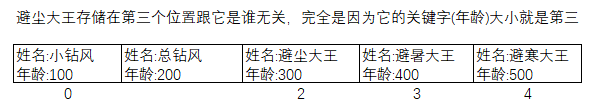
### B/B+树

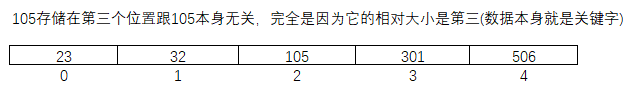
# 散列表

## 散列查找法的思想

对于一组数据而言，无论使用线性查找，二分法查找还是二叉排序树来查找，在查找过程中都只考虑数据元素中各个关键字之间的大小关系。数据在存储结构中的位置和数据本身并没有直接的关系。

例如 ：





上诉这种通过比较数据元素关键字大小来查找的办法，其查找时间与数据元素的个数有直接的关系。当数据元素很多的时候，查找的时候需要进行大量的无效的比较，致使查找速度很慢。即使使用效率相对较高的二分法查找，也需要进行大量无效的比较，造成一定的时间浪费。

为了解决这个问题，我们可以在数据元素的存储位置和数据元素关键字之间使用一个函数来建立某种对应关系，当需要查找某个数据元素的时候，无需进行大量无效的比较，直接通过这种对应关系就可以获得该数据元素的存储位置。**这就是散列查找法的思想。**

对数据元素关键字进行某种运算，获得数据元素的存储地址的函数就叫做**散列函数。**

使用散列函数对数据元素关键字运算出来的存储地址就叫做**散列地址**

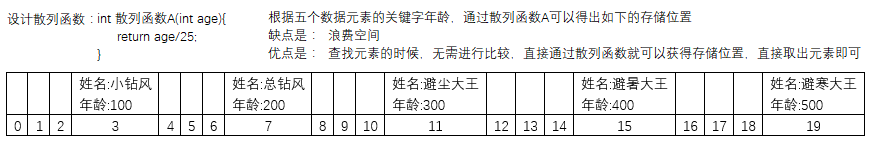
**即 ：**

数据的存储位置 = **散列函数**(关键字){

//自己设计函数的逻辑

}

例如 ：



## 散列表的概念

在一个有限的连续的存储空间中，数据元素的存储位置与关键字之间存在对应关系，这种结构就是散列表。

通常散列表的存储空间是一个一维数组，散列地址就是数组的下标。

散列表也叫哈希表，散列函数也叫做哈希函数。

* **散列函数的构造方法**

散列函数的作用是根据数据元素的关键字通过某种逻辑运算，获得该数据元素的存储地址。使用这样的原则可以构造出来多种多样的散列函数。

通常来讲，构造散列函数的方法有如下几种：

1. 数字分析法

如果数据元素的关键字的位数比散列表的地址码位数多，则可以从关键字中提取数字分布比较均匀的若干位作为散列地址。

例如 ： 数据元素 {姓名 ： 小钻风，学号 ：10086}，可以提取86作为散列地址。

1. 平方取中法

通常在选定散列函数时不一定能够知道关键字的全部情况，取其中哪几位也不一定合适，而一个数平方后的中间几位数和数字的每一位都相关，如果取关键字平方后的中间几位或者其组合作为散列地址，则使随机分布的关键字得到的散列地址也是随机的，具体所取的位数由表长决定。

平方取中法是一种较为常用的构造散列函数的方法。

1. 折叠法

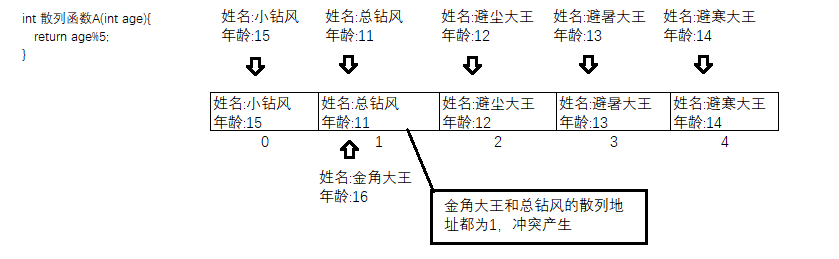
将关键字分隔成位数相同的几部分，然后取这几部分的叠加和作为散列地址，这种方法称为折叠法。

1. 保留余数法

这是最常用的构造散列函数的方法。假设散列表的长度位len,那么无论关键字进行何种运算，最后只要取余len,其结果一定是0到len范围之内，这样就能够保证最后的散列地址一定在散列表的地址空间内。

* **地址冲突的问题**

现有散列表长度为5，散列函数使用保留余数法，将下列数据元素放入散列表中。



选择一个“好”的散列函数可以在一定程度上减少冲突，但却无法绝对的避免冲突问题的产生。基于这样的原因，就有了解决地址冲突的办法。

* **处理冲突的办法**

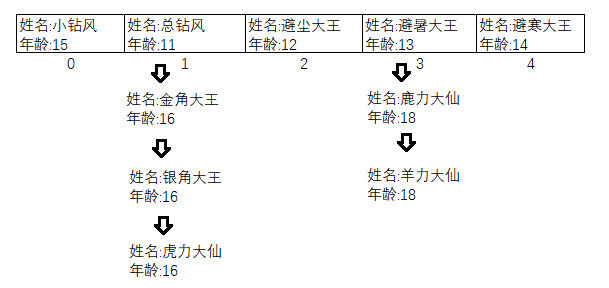
解决冲突的办法，大致来说有两种

1. 开放地址法

将冲突的数据元素，再次进行其它运算，用以重新获得一个散列地址，例如进行二次哈希运算，或者进行线性探测，随机探测等。

1. 链地址法

将产生冲突的数据元素变成一个单链表，这样就能够保存所有的数据元素，从而解决地址冲突的问题。



## 散列表实现

众所周知，jdk中的HashMap就是一个散列表的典型实例。这次就带大家手写一个HashMap，去体会散列表的高效。

**创建HashMap类**

**public** **class** Demo22\_HashMap<K, V> {

// 定义散列表的一维数组

Node<K, V>[] nodeArray = **null**;

// 散列表的长度(数据元素的个数)

**private** **int** size;

// 众高手验证，默认长度16最好使

**private** **static** **final** **int** ***DEFAULT\_LENGTH*** = 16;

**public** Demo22\_HashMap() {

// 初始化散列表的一维数组

nodeArray = **new** Node[***DEFAULT\_LENGTH***];

}

// 定义散列函数,获得数据元素的散列地址

**public** **int** myHashCode(K k) {}

// 获取散列表数据元素的个数

**public** **int** size() {

**return** size;

}

// 添加数据元素到散列表

**public** **void** put(K k, V v) {}

// 根据关键字获取数据元素

**public** V get(K k) {}

// 根据关键字刪除数据元素

**public** **void** remove(K k) { }

// 模拟jdk中Map的高内聚

**class** Node<K, V> {

K key;

V value;

// 单链表

Node<K, V> next;

**public** Node(K key, V value) {

**this**.key = key;

**this**.value = value;

}

**public** K getKey() {

**return** **this**.key;

}

**public** V getValue() {

**return** **this**.value;

}

}

}

**设计散列函数**

// 定义散列函数,获得数据元素的散列地址，里边的逻辑可以随意的设计，但是一般使用数字分析法、平方取中法、折叠法、保留余数法中的一种

**public** **int** myHashCode(K k) {

// 获取key的内存地址

**int** code = k.hashCode();

// 返回散列地址（保留余数法）

**return** code % ***DEFAULT\_LENGTH***;

}

**根据关键字获取数据元素**

// 根据关键字获取数据元素

**public** V get(K k) {

**if** (k == **null**) {

**return** **null**;

}

// 获取散列地址

**int** index = myHashCode(k);

Node<K, V> node = nodeArray[index];

// 循环单链表

**while** (node != **null**) {

K key = node.getKey();

**if** (k.equals(key)) {

**return** node.getValue();

}

node = node.next;

}

**return** **null**;

}

**添加数据到散列表**

// 添加数据元素到散列表

**public** **void** put(K k, V v) {

// 直接根据关键字获取数据元素

V tempValue = get(k);

// 获取散列地址

**int** index = myHashCode(k);

// 获取旧链表

Node oldChain = nodeArray[index];

// 散列表中没有当前数据元素

**if** (tempValue == **null**) {

// 创建一个新节点

Node newNode = **new** Node(k, v);

// 将新节点添加到旧链表的头部

newNode.next = oldChain;

// 将新节点添加到散列表

nodeArray[index] = newNode;

size++;

} **else** {

// 散列表中已经有当前关键字，只能是覆盖操作

**while** (oldChain != **null**) {

Object key = oldChain.getKey();

**if** (k.equals(key)) {

oldChain.setValue(v);

**break**;

}

oldChain = oldChain.next;

}

}

}

**根据关键字删除数据元素**

// 根据关键字刪除数据元素

**public** **void** remove(K k) {

**int** index = myHashCode(k);

Node<K, V> node = nodeArray[index];

**if** (node != **null**) {

// 头节点就是要删除的数据元素

**if** (node.getKey().equals(k)) {

size--;

nodeArray[index] = node.next;

**return**;

}

}

// n1→n2→n3→n4 删除单向链表中的某个节点

**while** (node != **null**) {

Node<K, V> next = node.next;

**if** (next != **null**) {

**if** (k.equals(next.getKey())) {

// 要使用n1来删除n2 node是n1,next是n2 next.next是n3

node.next = next.next;

size--;

**break**;

}

}

node = next;

}

}

**遍历散列表**

@Override

**public** String toString() {

**for** (Node node : nodeArray) {

**while** (node != **null**) {

System.***out***.print("key :"+ node.getKey() + " value :" + node.getValue() + " → ");

node = node.next;

}

System.***out***.println();

}

**return** "";

}

**测试代码**

**public** **class** Test {

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

Demo22\_HashMap<Integer,Integer> hashMap = **new** Demo22\_HashMap<Integer,Integer>();

**for** (**int** i = 1; i <= 20; i++) {

hashMap.put(i, i);

}

//移除操作

hashMap.remove(1);

//覆盖操作

hashMap.put(15, 999);

//遍历操作

System.***out***.println(hashMap);

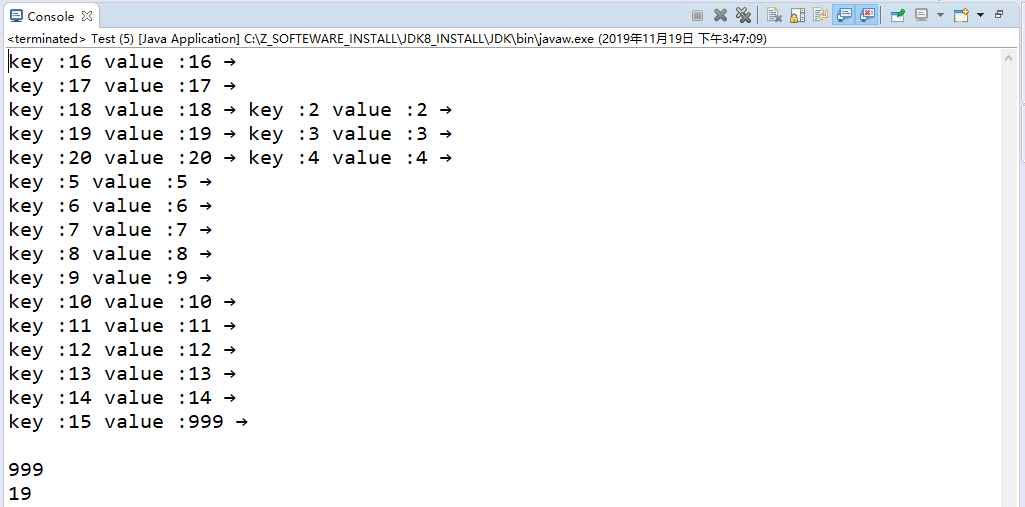
//获取操作

System.***out***.println(hashMap.get(15));

System.***out***.println(hashMap.size());

}

}



注意 ： 我们这里并不是要讨论JDK中的HashMap，只是学习一下散列表。对于HashMap中的扩容，调整平衡等优化，我们会在专门的Map专题中介绍。

# 图形结构

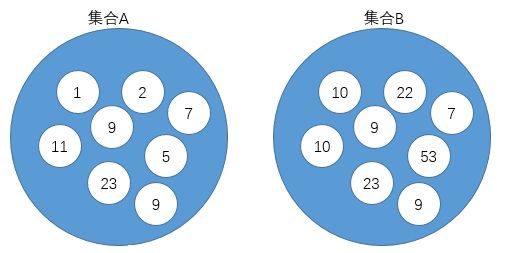
图形结构中相关的内容还是比较多的，我们这次并不会全面的介绍图形结构，只是让大家对图形结构有一个初步的入门即可。

回顾：

数据结构按逻辑结构划分可以有如下四种结构

* 集合结构

集合结构中的数据元素，除了“同属一个集合外”，再无其它任何关系。



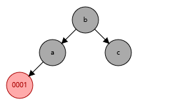
* 线性结构

在数组，线性表，栈，队列等等线性结构中，每一个数据元素都有一个前驱，一个后继数据，属于一对一的关系。



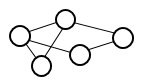
* 树形结构

树形结构则属于一对多的关系，即没一个节点具有一个双亲节点，但是却有多个子节点。



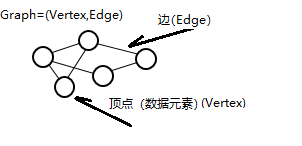
* 图形结构

图形结构属于复杂的数据结构，它可以用来表示人物的社会关系等应用场景。属于多对多的关系。即一个数据元素有多个前驱和多个后继数据元素。



## 图的基本概念和术语

图形结构使用顶点和边来描述。每一个数据元素都叫做一个顶点，而连接顶点与顶点之间的逻辑连线则叫做边。在图形结构中，顶点是有穷的非空集合，边则是有穷集合。



* 有向图

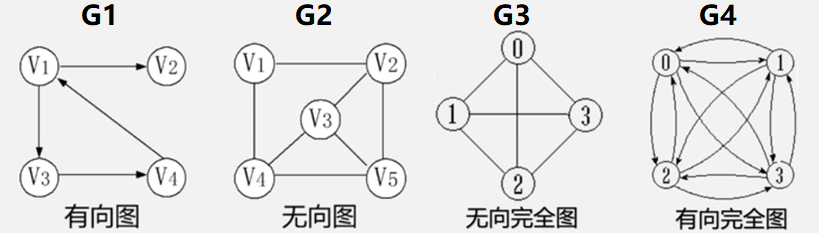
G1图中有四个顶点，四条边，而且四条边都有方向,这就叫做有向图。

* 无向图

G2图中有五个顶点，七条边，而且七条边都没有方向，这就叫做无向图。

* 完全图

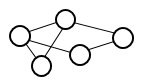
G3,G4两图中，任意两个点都有一条边相连，这就是完全图



邻接：邻接表示的是有边的两个顶点之间的关系。例如在G2图中存在G（V3,V5），则称V3和V5互为邻接点。

## 图的存储结构

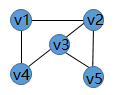
* 邻接矩阵

图形结构其实是数据元素之间的多对多的关系，由于这种结构比较复杂，直接使用图形的结构来进行物理存储则是很困难的。

而且图没有顺序存储结构。虽然是这样，但我们可以借助二维数组来表示元素之间的关系，这就是数组表示法，也叫做邻接矩阵。

除了使用邻接矩阵来存储图形结构，当然还有其它的方式，我们这里不再介绍。

* 邻接矩阵存储原理



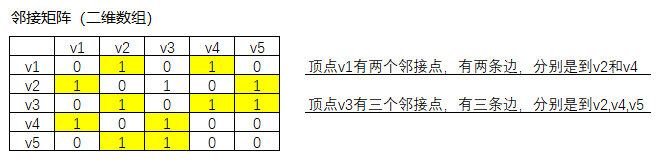
1. 建立顶点表

使用一个一维数组来记录各个顶点的信息，假设图中有五个顶点，那么顶点表的长度就是5。

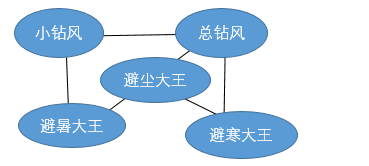


1. 建立邻接矩阵

使用一个二维数组，来表示各个顶点之间的关系，两个顶点之间有边，则存储1，没有边，则存储0。二维数组的长度就是顶点的数量。



## 图形结构示例代码



|  |  |
| --- | --- |
| **创建数据顶点类**  **public** **class** Demo21\_Student\_Vertex {  **private** String name;  **private** **int** age;  } | **创建图形类**  **public** **class** Demo21\_Student\_Graph {  // 顶点表  **private** Demo21\_Student\_Vertex svArr[];  **private** **int** index;  // 邻接矩阵  **private** **int**[][] adjMat;  **public** Demo21\_Student\_Graph(**int** size) {  svArr = **new** Demo21\_Student\_Vertex[size];  // 邻接矩阵的大小就是顶点表的长度  adjMat = **new** **int**[size][size];  }  } |
| **图形类**  // 添加元素到顶点表  **public** **void** addElement(Demo21\_Student\_Vertex sv) {  svArr[index] = sv;  index++;  } | **图形类**  // 添加边  **public** **void** addEdge(Demo21\_Student\_Vertex sv1, Demo21\_Student\_Vertex sv2) {  **int** index1 = -1, index2 = -1;  // 寻找顶点元素下标  **for** (**int** i = 0; i < svArr.length; i++) {  Demo21\_Student\_Vertex tempSt = svArr[i];  **if** (sv1.equals(tempSt)) {  index1 = i;  }  **if** (sv2.equals(tempSt)) {  index2 = i;  }  //两个顶点的下标都已经获得  **if** (index1 > -1 && index2 > -1) {  **break**;  }  }  //添加边  adjMat[index1][index2] = 1; //v1和v2有边  adjMat[index2][index1] = 1;//那么v2到v1也肯定有边  } |

**测试代码**

**public** **class** Test {

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

Demo21\_Student\_Vertex v1 = **new** Demo21\_Student\_Vertex("小钻风", 300);

Demo21\_Student\_Vertex v2 = **new** Demo21\_Student\_Vertex("总钻风", 500);

Demo21\_Student\_Vertex v3 = **new** Demo21\_Student\_Vertex("避尘大王", 700);

Demo21\_Student\_Vertex v4 = **new** Demo21\_Student\_Vertex("避暑大王", 800);

Demo21\_Student\_Vertex v5 = **new** Demo21\_Student\_Vertex("避寒大王", 900);

Demo21\_Student\_Graph graph = **new** Demo21\_Student\_Graph(5);

graph.addElement(v1);

graph.addElement(v2);

graph.addElement(v3);

graph.addElement(v4);

graph.addElement(v5);

//添加边

graph.addEdge(v1, v2);//小钻风-总钻风

graph.addEdge(v1, v4);//小钻风-避暑大王

graph.addEdge(v2, v3);//总钻风 - 避尘大王

graph.addEdge(v2, v5);//总钻风-避寒大王

graph.addEdge(v3, v5);//避尘大王 - 避寒大王

graph.addEdge(v3, v4);//避尘大王 - 避暑大王

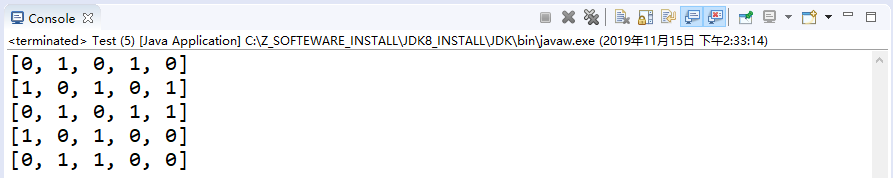
**for** (**int**[] arr : graph.adjMat) {

System.***out***.println(Arrays.*toString*(arr));

}

}

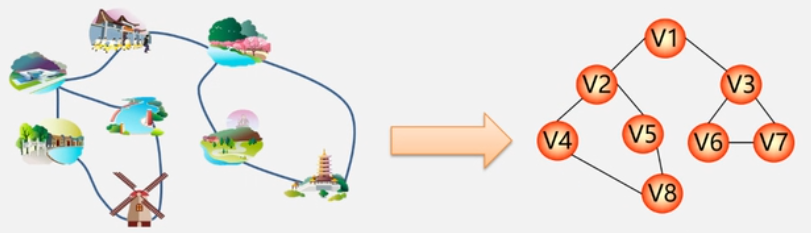
}



## 图的遍历

从已给的图中的某一个顶点出发，沿着**一些边访遍图中的所有顶点，且使每个顶点仅被访问一次**，就叫做图的遍历，它是图的基本运算。

而遍历的本质则是寻找每个顶点的邻接点的过程。



* 图的特点

图中可能存在回路，且图的任一顶点都可能与其它顶点相通。在访问完成某个顶点以后可能会沿着某些边又回到了曾经访问过的顶点。

* 怎样避免重复访问？

解决思路 ： 设置一个辅助数组visited[n]，用来标记每个被访问过的顶点。

初始状态visited[n]中所有的值为0。

顶点i被访问，修改visted[i] = 1,防止多次访问。

* 常用的遍历思想

深度优先搜索（Depth\_First Search --- DFS）

广度优先搜索 (Breadth\_First Search ---BFS)

**深度优先遍历**

深度优先遍历的思想是“一条道走到黑”，走不动了再后退，后退到第一个还有邻接点的点，再次前进，如此完成图的遍历

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**无向图的深度优先遍历示例代码**

**public** **class** Demo21\_Student\_Graph {

// 深度优先遍历的辅助数组---其它代码---略---

**private** **int** dfs[];

**public** Demo21\_Student\_Graph(**int** size) {

// 初始化遍历的辅助数组 ---其它代码---略---

dfs = **new** **int**[size];

}

}

// 深度优先遍历无向图

**public** **void** dfsSearch(Demo21\_Student\_Vertex sv) {

**if** (sv == **null**) {

**return**;

}

// 遍历当前节点

System.***out***.println(sv);

**int** currentIndex = -1;

// 获得当前顶点的下标

**for** (**int** i = 0; i < svArr.length; i++) {

Demo21\_Student\_Vertex currentvertex = svArr[i];

**if** (sv.equals(currentvertex)) {

currentIndex = i;

**break**;

}

}

// 当前顶点已经被访问过

dfs[currentIndex] = 1;

// 选择当前顶点的邻接点们

**int**[] nearVer = adjMat[currentIndex];

//当递归出栈的时候就会在for循环中“后退”，找到没有遍历的邻接点继续递归

**for** (**int** i = 0; i < nearVer.length; i++) {

**int** nearValue = nearVer[i];

//它是当前节点的邻接点

**if** (nearValue == 1) {

//这个邻接点还没有被访问过

**if** (dfs[i] == 0) {

//遍历这个邻接点

dfsSearch(svArr[i]);

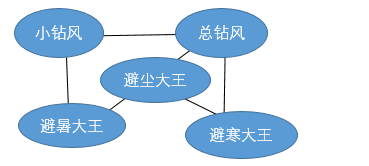
}

}

}

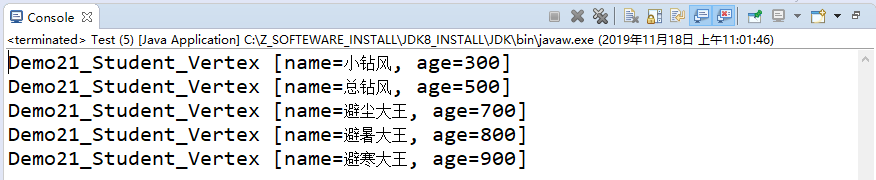
}

**测试代码**



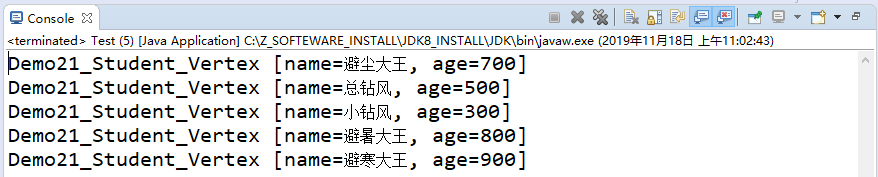
//从小钻风开始遍历

graph.dfsSearch(v1);



//从避尘大王开始遍历

graph.dfsSearch(v3);



# 算法

## 递归算法

### 斐波那契数列

**public** **class** Demo01\_斐波那契数列 {

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

// 1 1 2 3 5 8 13 21 34 ...

System.***out***.println(*count*(9));

}

**public** **static** **int** count(**int** n) {

**if** (n < 3) {

**return** 1;

} **else** {

**return** *count*(n - 1) + *count*(n - 2);

}

}

}

### 汉诺塔问题

**public** **class** Demo02\_汉诺塔问题 {

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

// A B C

*transfer*(7, "A", "B", "C");

}

//无论有多数个圆盘，都只认为有两个圆盘 ： 最下面的一个，上面所有的算一个

**public** **static** **void** transfer(**int** n, String from, String middle, String to) {

**if** (n == 1) {

System.***out***.println("第1个圆盘从" + from + "移动到" + to);

} **else** {

*transfer*(n - 1, from, to, middle);

System.***out***.println("第" + n + "个圆盘从" + from + "移动到" + to);

*transfer*(n - 1, middle, from, to);

}

}

}

## 排序算法

排序算法，顾名思义，是对一些数据进行正序或者倒序的排序。众多常用的排序算法中有八种常用的排序算法，它们可以分为以下五类：

* 交换排序
* 插入排序
* 选择排序
* 归并排序
* 基数排序

### 交换排序

交换排序是在比较数据大小的时候，将数据进行交换，以达到最终的排序目标。有冒泡和快速这两种。

#### 冒泡排序

**int**[] arr = { 1, 2, 3, 45, 67, 8, 13, 2, 4, 65 };

// 控制比较多少轮

// 第一个不用和自己比较，或者说最后一个没有下一个，不用比较最后一个

// 所以比较的轮数 = arr.length-1

**for** (**int** i = 0; i < arr.length - 1; i++) {

// 每轮比较多少次

// 比较过的不用再比较，所以每轮比较的次数要-i

**for** (**int** j = 0; j < arr.length - 1 - i; j++) {

**if** (arr[j] < arr[j + 1]) {//>号则是升序

**int** temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

System.***out***.println(Arrays.*toString*(arr));

#### 快速排序

快速排序是从冒泡排序演变而来的，但是比冒泡排序效率要高很多。所以叫快速排序。

快速排序之所以快速，是因为它使用了分治法。

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](https://baike.baidu.com/item/%E9%80%92%E5%BD%92/1740695)进行，以此达到整个数据变成有序[序列](https://baike.baidu.com/item/%E5%BA%8F%E5%88%97/1302588)。

##### 实现思路

原始数组

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 9 | 7 | 6 | 4 | 3 | 2 | 1 | 8 |

随便找一个数字作为基准数字，接下来将数组中所有大于基准数字的元素放在数组的右边，比基准数字小的放在基准数字的左边为了方便起见，使用数组的第一个元素作为基准数字，在初始状态下，基准数字在数组的第一个位，我们的目标是将基准数字5挪动数组中间的某个位置，假设这个位置是K，现在就需要寻找到K,并且以K为分界点，左边的都小于基准数字5，右边的都大于基准数字5这样将数组分成了大小两个部分，这就是分治的思想。

继续使用这样的分治思想，使用相同的方法处理大和小的部分，直到完成整个数组的排序。

##### 过程图解

红色箭头从右往左开始搜索，每次都找到一个比基准数字小的数字，然后停下来等待

黑色箭头从左往右开始搜索，每次都找到一个比基准数字大的数字，然后停下来等待

交换被找到的两个数字 ：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 9 | 7 | 6 | 4 | 3 | 2 | 1 | 8 |
| 5 | 1 | 7 | 6 | 4 | 3 | 2 | 9 | 8 |

交换完成以后，红色箭头继续向左搜索，找到第一个比基准数字小的数字，黑色箭头继续向右边搜索，找到第一个比基准数字大的数字，进行交换

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 7 | 6 | 4 | 3 | 2 | 9 | 8 |
| 5 | 1 | 2 | 6 | 4 | 3 | 7 | 9 | 8 |
|  |  |  |  |  |  |  |  |  |
| 5 | 1 | 2 | 6 | 4 | 3 | 7 | 9 | 8 |
| 5 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | 8 |

当红色箭头与黑色箭头重逢，意味着一遍搜索完成，这个时候只需要将黑色箭头与基准数字进行交换以后，就可以得出最基本的排序：基准数字左边都是小于基准数字的数字，基准数字右边都是大于基准数字的数字

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 2 | 3 | 4 | 6 | 7 | 9 | 8 |
| 4 | 1 | 2 | 3 | 5 | 6 | 7 | 9 | 8 |

继续将5右边大的部分分成两部分，将5左边小的部分分成两部分，直到完成整个数组的排序。

##### 示例代码

/\*\*

\*

\* **@param** arr : 要排序的数组

\* **@param** start : 开始的位置

\* **@param** end : 结束的位置

\*/

**public** **static** **void** quickSort(**int**[] arr, **int** start, **int** end) {

**if** (start > end) {

**return**;

}

**int** standrand = arr[start];

**int** tempStart = start;

**int** tempEnd = end;

**while** (tempStart < tempEnd) {

// 数组右边：寻找小于标准数字的元素

**while** (arr[tempEnd] >= standrand && tempEnd > tempStart) {

tempEnd--;

}

// 数组左边：寻找大于标准数字的元素

**while** (arr[tempStart] <= standrand && tempStart < tempEnd) {

tempStart++;

}

//将被找到的"大数字"和被找到的"小数字"进行位置交换

**if** (tempStart < tempEnd) {

**int** temp = arr[tempEnd];

arr[tempEnd] = arr[tempStart];

arr[tempStart] = temp;

}

}

//交换基准数字

arr[start] = arr[tempStart];

arr[tempStart] = standrand;

//将基准数字右边的继续拆分成两个部分

*quickSort*(arr, tempEnd + 1, end);

//将基准数字左边边的继续拆分成两个部分

*quickSort*(arr, start, tempStart - 1);

}

##### 总结

快速排序的每一轮处理其实就是将这一轮的基准数归位，直到所有的数都归位为止，排序就结束了。快速排序之所比较快，因为相比冒泡排序，每次交换是跳跃式的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是O(N2)，它的平均时间复杂度为O(NlogN)。

### 插入排序

#### 直接插入排序

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。

##### 实现思路

假设数组的前N个元素已经是有序的，那么只需要将N+1以后的元素依次插入到前N个元素中的某个位置即可完成整个数组的排序。

而无论任何数组，只要它只有且只有一个元素，那么无论该元素值是多少，那么都是排序好的。所以我们可以假设所有的的数组的第一个元素都是排序完成的，需要做的是将第一个以后的元素依次往前排列即可。

原始数组，排序完成的只有第一个元素，此时N=1。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 1 | 7 | 5 | 4 | 1 |

接下来应该将N+1插入到有序数组N中的某个位置，即可完成N+1号元素的排序。而这个位置只需要依次和前N个元素依次比较大小即可找到。由于前N个元素已经是排序完成的，所以不用担心新插入的元素会打乱原来的N个有序元素的顺序。就好像扑克牌一样，在排列好的牌中添加一张牌，只是让排好的数量加1而已，并不会打乱原来的扑克牌顺序。这就是插入排序的原理，每次插入一个元素都会得到一个个数加1的有序数组。

首先N=1,这是初始状态的有序个数，它是角标为0的3。而N+1号就是第二个元素，它是2。将N+1号元素依次和它前面的元素进行大小比较即可找到2的位置。最终2<3,所以交换位置。本来只有一个排列好的3，插入一个2以后，排列好的元素就变成了两个 2和3。刚好符合加1的特征，最后N=2。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 7 | 5 | 4 | 1 |

N=2以后，N+1号就是第三个元素1，它依次和前边的N个有序的元素进行大小比较，寻找自己的位置。

第一次它碰到了3，1<3，交换位置，

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | 3 | 7 | 5 | 4 | 1 |

第二次它碰到了2，1<2，交换位置

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 7 | 5 | 4 | 1 |

这样1也加入到了N的序列中，最终N=3。

N=3以后，N+1号就是第4个元素7，它依次和前边的N个有序的元素进行大小比较，寻找自己的位置。

第一次它碰到了3，7>3，不交换位置，由于前边都是排序好的，所以7不用再往前进行比较大小了。它的位置已经固定

所以插入排序有一个特点，只要N+1号元素没有发生位置交换，则已经找到了自己的位置，不用再往下寻找。

如此下去，直到N=6,此时的排序应该是

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 1 |

N=3以后，N+1号就是第7个元素1，它依次和前边的N个有序的元素进行大小比较，寻找自己的位置。

第一次它碰到了7，1<7,交换位置

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 1 | 7 |

第二次它碰到了5，1<5,交换位置

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 1 | 5 | 7 |

如此下去，直到最后的1也找到自己的位置，整个数组完成排序。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 3 | 4 | 5 | 7 |

总结：

一：仔细观察整个插入排序可以发现，它其实就是逆向的冒泡排序（进行N-1遍循环，两两比较，交换位置）

二：最后一个元素是1，也就是说突然间来了一个特别小的数字的时候，整个排序效率特然变低，解决这个问题就可以用希尔排序。

##### 示例代码

**public** **static** **void** insertSort(**int**[] arr) {

//初始状态的数组只有第一个是排序好的，即N=1,而那个元素的角标是0

**for** (**int** n = 0; n < arr.length-1; n++) {

/\*

\* 角标j就是N+1号元素

\* N+1号元素需要和前边的N个元素中的每一个都进行大小比较

\* 那么就是和N,N-1,N-2,N-3...号元素都进行比较

\* 体现在代码中就是j-1(N),j-2(N-1)...

\*

\*/

**for** (**int** j = n + 1; j > 0; j--) {

//只要N+1号元素小于有序N元素中的数字，则发生位置交换，然后继续和前一个进行大小比较，直到找到N+1号元素的位置

**if** (arr[j] < arr[j - 1]) {

**int** temp = arr[j-1];

arr[j-1] = arr[j];

arr[j] = temp;

} **else** {

/\*一旦没有发生位置交换，说明N+1号元素插入到了应有的位置，不用再继续进行它的大小比较

\* 应该让下一个元素进行排序

\*/

**break**;

}

}

}

}

#### 希尔排序

希尔排序是希尔（Donald Shell）提出的一种排序方法，也属于插入排序，但是简单插入排序的高效版本，也称为缩小增量排序。基本思想是将待排序元素进行增量分组，然后在分组组内进行插入排序，随着增量的减少，每个分组组内的元素越来越多，直至增量减至1时，所有元素都分到同一个组内，执行插入排序后完成整个排序操作。

##### 基本思想

希尔排序是把数组元素按下表的一定增量（步长）分组，然后对每组使用直接插入排序算法排序。随着增量逐渐减少，每组包含的元素越来越多，当增量减至1时，整个数组恰好被分成一组，完成最后的排序，算法终止。

直接插入排序循规蹈矩，无论数组中的元素如何分布，总是一步一步对元素逐个比较，移动，插入，完成排序。例如数组[5,4,3,2,1,0],显然数组末端的元素0要回倒首位非常耗时，比较和移动的次数均是n-1次。而希尔排序在数组中采用跳跃式分组的策略，通过某个增量将数组分为若干组，然后分组进行插入排序。随着逐步缩小增量，继续按组进行插入排序操作。直至增量为1。希尔排序通过这种策略使得整个数组在初始阶段达到从宏观上基本有序：小的基本在前，大的基本在后。然后缩小增量，倒增量为1时，多数情况下只需要微调即可，不会涉及过多的数据移动。

##### 过程图解



##### 示例代码

**public** **static** **void** shellSort(**int**[] arr) {

**int** buChang = arr.length / 2;

**while** (buChang > 0) {

**for** (**int** j = 0; j < buChang; j++) {

//筛选出来每一组的数字

**for** (**int** i = j+buChang; i<arr.length; i+=buChang) {

//将每一组的数字都进行插入排序，这里的范围不再是简单的>0，而是同一组的数字进行比较，所以是大于j

**for** (**int** k = i; k>j; k-=buChang) {

**if** (arr[k] < arr[k-buChang] ) {

**int** temp = arr[k];

arr[k] = arr[k-buChang];

arr[k-buChang] = temp;

}**else** {

**break**;

}

}

}

}

buChang /= 2;

}

}

### 选择排序

#### 简单选择排序

选择排序是每次都选择出来一个最大/最小的元素，依次与数组的末尾/开始元素进行位置交换，从而完成整个数组的排序。

##### 基本思想

每次都从数组的第一个位置开始查找，使用一个变量记录本轮循环查找出来的最大元素的角标

假设数组长度为N，

第一次遍历n-1个数，找到最大的数值与最后一个元素交换(N-1)；  
第二次遍历n-2个数，找到本次最大的数值与倒数第二个元素交换（N-2）；

##### 过程图解

原始数组如下

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 2 | 7 | 9 | 3 | 5 | 4 | 6 | 8 |

第一次遍历，设置变量maxIndex = 0,

Arr[maxIndex] = 1 > 0 ,不更改maxIndex的值， maxIndex依然等于0。

Arr[maxIndex] = 1 < 2,设置maxIndex = 2

继续遍历Arr[maxIndex] =2 < 7,设置maxIndex = 3

Arr[maxIndex] =7 < 9，设置maxIndex = 4

继续遍历到数组末尾，最后maxIndex = 4。

交换arr[maxIndex] = 9和arr[末尾] = 8的值

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 2 | 7 | 8 | 3 | 5 | 4 | 6 | 9 |

第二次遍历，设置变量maxIndex = 0,使用相同的方式找到本次循环的最大数字8，和数组的倒数第二个数字6交换

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 2 | 7 | 6 | 3 | 5 | 4 | 8 | 9 |

如此往复，直到完成整个数组的排序

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 2 | 4 | 6 | 3 | 5 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 2 | 4 | 5 | 3 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 2 | 4 | 3 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  |  |  |  |  |  |  |  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

##### 示例代码

**public** **static** **void** selectSort(**int**[] arr) {

**for** (**int** i = 0; i < arr.length - 1; i++) {

**int** maxIndex = 0;

**for** (**int** j = 1; j < arr.length - i; j++) {

**if** (arr[maxIndex] < arr[j]) {

maxIndex = j;

}

}

//如果最大数字不是数组末尾的那个数字，再交换位置

**if** (maxIndex != arr.length-1 - i) {

**int** temp = arr[maxIndex];

arr[maxIndex] = arr[ arr.length-1 - i];

arr[ arr.length-1 - i] = temp;

}

}

}

#### 堆排序

##### 堆结构简介

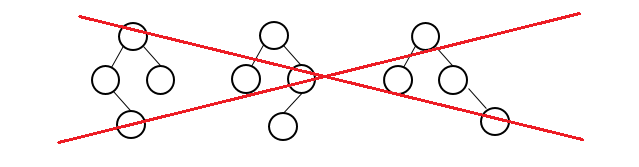
堆是一种数据结构，这种数据结构满足如下特点

* 它是一个完全二叉树
* 所有的父节点的值都大于子节点的值

###### 完全二叉树

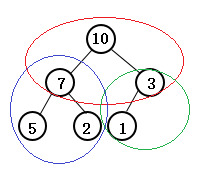
完全二叉树生成节点顺序是从上往下，从左往右，不间断。图中有七颗树，这七颗树都属于完全二叉树

下面示例几个错误的完全二叉树。



###### Parent > child

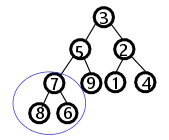
堆结构不仅仅要满足是一颗完全二叉树，同时要保证每个父节点都大于子节点的两个值，两个子节点之间则无所谓大小和左右。例如图中，10这个父节点大于两个子节点7和3就可以了，至于7和3谁是左边谁是右边则无所谓。同样，7作为父节点必须大于它的两个子节点，3作为父节点也必须大于它的一个子节点。



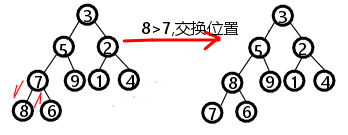
##### Hepify操作

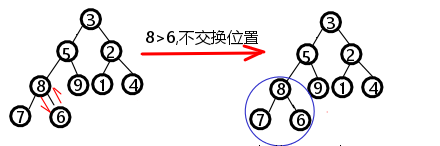
对于任意一个无序的树，都可以通过heapify操作，将其构建成一个完全二叉树。Heapify操作的具体过程如下

现有无序树如下



Hepify操作：将蓝色圆圈中的三个元素依次进行大小比较，最后将最大的元素放在parent节点，即可得到一个局部的完全二叉树，这就是一次hepify操作。



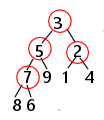


总结：hepify操作其实就是将原来较小的父节点与原来较大的子节点进行位置交换。交换完成以后，原父节点变成子节点，最大的子节点变成了父节点。

所以，只要从最后一个节点的父节点开始，依次对交换完成以后的子节点（原父节点）继续“向下”进行递归的hepifiy操作，即可构建一个完整的完全二叉树，此时，堆构建完成。

##### 构建堆结构

刚刚提到，对于无序的原始树，只要从最后一个节点的父节点开始，依次对交换完成以后的子节点（原父节点）继续“向下”进行递归的hepifiy操作，即可构建一个完整的完全二叉树。所以对于图中的原始树，只要依次对 7 2 5 3 这几个父节点中发生交换的子节点（原父节点）进行“向下”递归的hepify操作即可完成整个堆的构建。

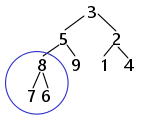
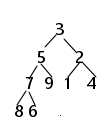


###### 构建堆过程图解

对倒数第一个父节点进行hepify操作

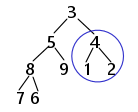
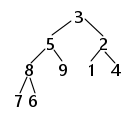
末尾节点是6，倒数第一个父节点是7，hepify操作完成以后原父节点7变成子节点，最大子节点8变成父节点。

继续向下递归对交换完成以后的子节点7（原父节点）进行hepify操作，而7没有子节点，本轮递归结束，所以递归的出口就是对进行hepify操作的节点进行是否具有子节点的判断。

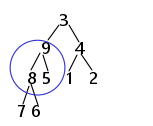
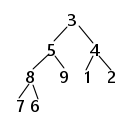


对倒数第二个父节点进行hepify操作

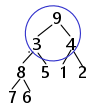
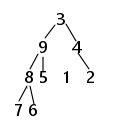
对于父节点2进行hepify操作完成以后，交换完成以后的子节点是2（原父节点），但是节点2没有子节点，本轮递归结束，对于节点2的整个hepify操作结束。



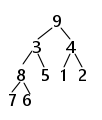
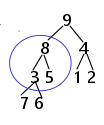
对倒数第三个父节点进行hepify操作，交换完成以后的子节点是5（原父节点），对子节点5继续进行递归的hepify操作，5没有子节点，递归结束。



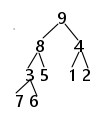
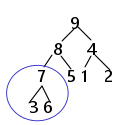
对倒数第4个父节点进行hepify操作，交换完成以后的子节点是3（原父节点）



继续对子节点3继续进行hepify操作，交换完成以后的子节点依然是3（原父节点）

对子节点3继续进行hepify操作，交换完成以后子节点一直都是3（原父节点）

对子节点3继续进行hepify操作，此时子节点3已经是最后的子节点，递归结束。

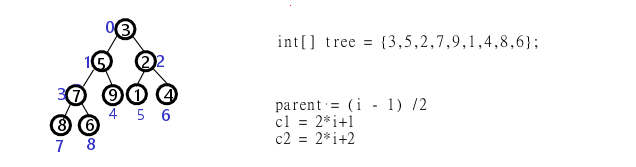
到这里堆结构构建完成，我们可以发现，由于堆结构的特点就是父节点大于子节点，所以对于某一个父节点而言，只要发生了交换，那么必然就是将原来的父节点给替换到了某个子节点的位置，所以每次递归其实是将原父节点不断的向子节点递归移动的过程。如果没发生交换，那么就说明父节点直接就是最大的节点，其实没必要在递归下去。

那么总共需要对哪些父节点进行移动呢？

通过示例图可以发现，从最后一个子节点的父节点开始，一直倒退到堆的最顶端，这中间所有的父节点都需要被递归移动。

最后的问题，如何找到最后一个子节点？如何找到某个节点的父节点以及它的两个子节点？

堆结构保证了树是连续的，中间不会有断开的情况，基于这种特点，我们可以将堆结构看成一个数组，只需要按照堆的特点依次从上往下，从左往右，即可确定堆结构中的每个节点在数组中的角标。



题外话，透过表象看本质，不得不说，数组真的好强大！！！

###### 构建堆过程示例代码

/\* 交换数组的两个元素 \*/

**public** **static** **void** swap(**int**[] arr, **int** i, **int** k) {

**int** temp = arr[i];

arr[i] = arr[k];

arr[k] = temp;

}

/\*\*

\*

\* **@param** tree 要排序的数组

\* **@param** len 要排序的元素个数，即数组的长度

\* **@param** index 要排序元素的角标

\*/

**public** **static** **void** hepify(**int** tree[], **int** len, **int** index) {

**if** (index > len) {

**return**;

}

// 获得两个子节点

**int** c1 = 2 \* index + 1;

**int** c2 = 2 \* index + 2;

// 假设父节点index就是最大值,两两比较，找到最大值的角标,注意数组角标别越界

**int** max = index;

**if** (c1 < len && tree[c1] > tree[max]) {

max = c1;

}

**if** (c2 < len && tree[c2] > tree[max]) {

max = c2;

}

// 如果最大值不是父节点，则进行交换

**if** (max != index) {

*swap*(tree, max, index);

System.***out***.println(Arrays.*toString*(tree));

//继续对交换完成后的子节点(原父节点)进行hepify操作

*hepify*(tree, len, max);

}

}  
 /\*构建堆\*/

**public** **static** **void** buildHeap(**int**[] tree, **int** len) {

//最后一个子节点：数组的末尾元素

**int** lastNode = len - 1;

//倒数第一个父节点

**int** parent = (lastNode - 1) / 2;

//从倒数第一个父节点，依次对每个父节点进行hepify操作

**for** (**int** i = parent; i >= 0; i--) {

*hepify*(tree, len, i);

}

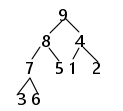
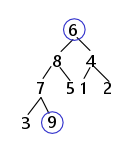
}

##### 排序

###### 排序图解

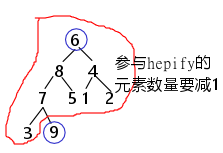
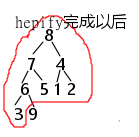
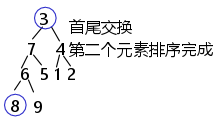
堆结构保证了连续和父节点大于子节点，这样就可以保证第一个节点的值是整个堆中的最大值。而堆排序整是利用了这样的特点进行排序：

将第一个节点与最后一个节点进行位置交换，交换完成以后，最大值在堆的末尾节点

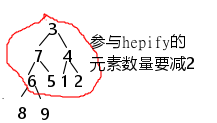
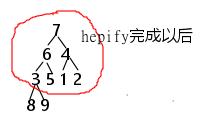
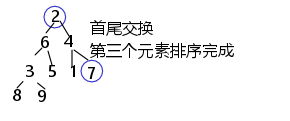
再次对第一个节点进行hepify操作，再次构建成一个稳定的堆结构，但是这次hepify操作需要“砍掉”最后一个子节点，因为它已经是被排序的节点，不能动。

Hepify操作完成后，再次进行首尾交换

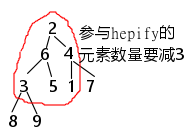
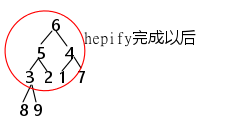
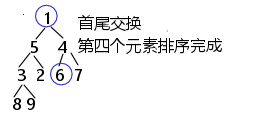
再次对第一个节点进行hepify操作，再次构建成一个稳定的堆结构，但是这次hepify操作需要“砍掉”最后两个个子节点，因为它们已经是被排序的节点，不能动。

Hepify操作完成后，再次进行首尾交换

再次对第一个节点进行hepify操作，再次构建成一个稳定的堆结构，但是这次hepify操作需要“砍掉”最后三个个子节点，因为它们已经是被排序的节点，不能动。

Hepify操作完成后，再次进行首尾交换

通过这种不断hepify操作，交换首尾的方式，即可完成整个树的排序，此时数组排序也已经完成。因为，树就是数组。

###### 示例代码

**public** **static** **void** heapSort(**int**[] tree) {

//构建堆结构

*buildHeap*(tree, tree.length);

**int** len = tree.length;

//通过for循环，不断减少hepify的元素数量

**for** (**int** i = len - 1; i >= 0; i--) {

// 首先进行首尾交换

*swap*(tree, i, 0);

// 然后对第一个节点进行hepify操作

*hepify*(tree, i, 0);

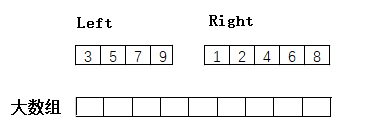
}

}

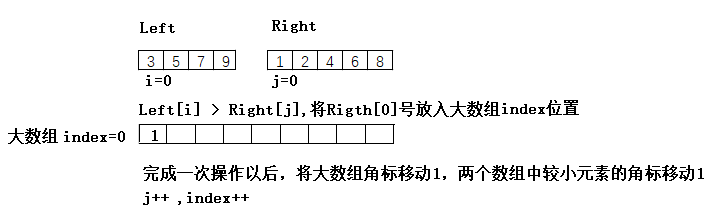
### 归并排序

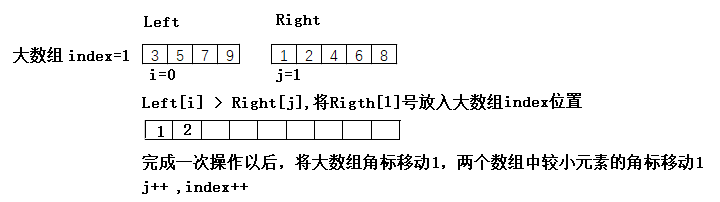
#### 合并的思想

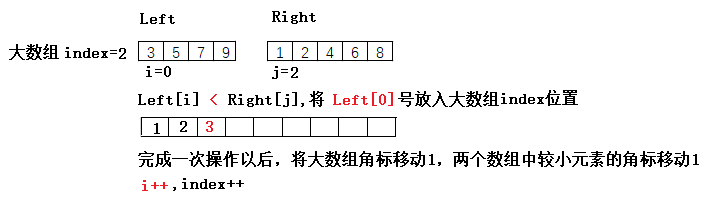
现有两个数组Left和Right。它们在其内部都已经是有序的。只需要来一个大的数组，而大数组长度是两个数组元素之和，那么只需要依次比较两个数组中的元素，将较小/大的元素放入到大数组中就可以将这两个数组合并到一起完成整体的排序。

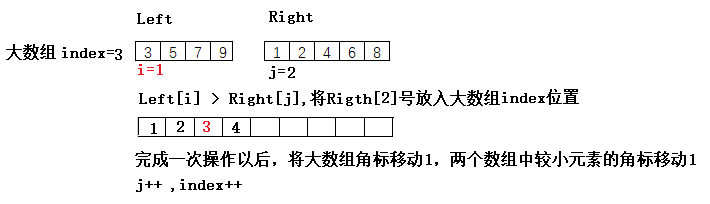


#### 合并图解

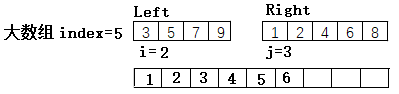
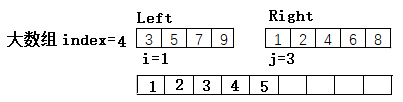


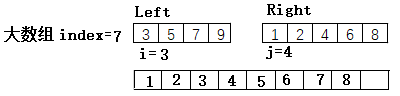
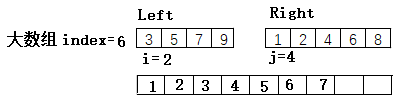






依此类推下去





当index = 8的时候，其实已经有一个数组已经把它所有的元素放入大数组中（本例是Right数组），那么接下来，只需要将另外一个还剩下若干元素的数组（本例是Left数组）中剩余的元素原封不动（因为它本来就是有序的，本例还剩下一个元素9，假设它还剩下有3个元素，那么这三个元素也必然是大于等于9且有序）的放入大数组中即可完成合并的操作，而且合并以后的大数组中的元素是排序完成的。

#### 合并示例代码

/\*\*

\*

\* **@param** leftArray 内部有序的数组

\* **@param** rigthArray 内部有序的数组

\* **@return** 排序完成的大数组

\*/

**public** **static** **int**[] merge(**int**[] leftArray, **int**[] rigthArray) {

//创建一个大数组，长度是两个数组的长度和

**int**[] bigArray = **new** **int**[leftArray.length + rigthArray.length];

//定义左，右，大三个数组的角标变量

**int** leftIndex = 0, rightIndex = 0, bigIndex = 0;

//合并两个数组到大数组中 ： 只要有一个数组已经合并完成，另外一个剩下的部分也不必在进行比较

**while** (leftIndex < leftArray.length && rightIndex < rigthArray.length) {

//左右两个数组进行元素比较

**if** (leftArray[leftIndex] < rigthArray[rightIndex]) {

//左边数组元素值较小，放入大数组中

bigArray[bigIndex] = leftArray[leftIndex];

//移动左边数组的角标变量

leftIndex++;

} **else** {

bigArray[bigIndex] = rigthArray[rightIndex];

rightIndex++;

}

//每次放入一个元素到大数组，都要移动大数组的角标变量

bigIndex++;

}

//将剩余的元素原封不动的添加到大数组中

**if** (leftIndex == leftArray.length) {

*addElement*(rigthArray, rightIndex, bigArray, bigIndex);

} **else** {

*addElement*(leftArray, leftIndex, bigArray, bigIndex);

}

**return** bigArray;

}

/\*\*

\* 将源数组的开始位置(包含)以后所有的元素添加到目标数组中，从目标数组的desIndex位置开始(包含)

\*

\* **@param** src 源数组

\* **@param** srcIndex 源数组开始的位置

\* **@param** des 目标数组

\* **@param** desIndex 目标数组开始的位置

\*/

**public** **static** **void** addElement(**int**[] src, **int** srcIndex, **int**[] des, **int** desIndex) {

**for** (**int** i = srcIndex; i < src.length; i++) {

des[desIndex] = src[i];

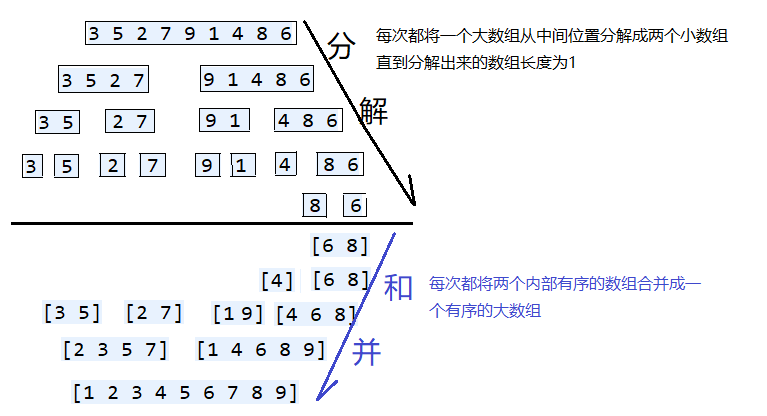
desIndex++;

}

}

#### 分治的思想

我们已经可以将两个有序数组合并成一个大数组，同样，可以使用分治的思想将一个无序的大数组分成若干小数组。当小数组的长度为1的时候，它在其内部一定是有序的，这个时候就可以将两个小数组进行合并，通过这种不断递归拆分，不断递归合并的思想即可完成整个数组的排序。



#### 分治-归并示例代码一

假设要排序的数组刚刚好左半边有序，右半边也有序，那么只需要将数组从中间拆分一次，进行合并操作即可完成整个数组的排序。

例如有数组如下[1,3,5,7,2,4,6,8]

注意 ：这段代码只是一个过渡过程。

**public** **static** **int**[] merageSort(**int**[] array) {

// 将原无序数组分成两份

**int** middle = array.length / 2;

// 左半边数组,数组内都是默认值0

**int**[] leftArray = **new** **int**[middle];

// 右半边数组

**int**[] rightArray = **new** **int**[array.length - middle];

// 给左半边数组赋值，将原来数组左半部分元素赋值给左数组[1,3,5,7]

**for** (**int** i = 0; i < leftArray.length; i++) {

leftArray[i] = array[i];

}

// 给右半边数组赋值[2,4,6,8]

**for** (**int** i = 0; i < rightArray.length; i++) {

rightArray[i] = array[middle + i];

}

// 合并成一个有序的大数组[1,2,3,4,5,6,7,8]

**int**[] merge = *merge*(leftArray, rightArray);

**return** merge;

}

#### 分治-归并示例代码二

那么对于一个无序的数组，则需要递归拆分，合并。下列代码使用[3,1,4,2]则相对容易理解。

这才是归并排序的完整代码。

**public** **static** **int**[] merageSort(**int**[] array) {//[3,1,4,2]

// 将原无序数组分成两份

**int** middle = array.length / 2;

// 左半边数组,数组内都是默认值0

**int**[] leftArray = **new** **int**[middle];

// 右半边数组

**int**[] rightArray = **new** **int**[array.length - middle];

// 给左半边数组赋值，将原来数组左半部分元素赋值给左数组[0,0]→[3,1]

**for** (**int** i = 0; i < leftArray.length; i++) {

leftArray[i] = array[i];

}

// 给右半边数组赋值[0,0]→[4,2]

**for** (**int** i = 0; i < rightArray.length; i++) {

rightArray[i] = array[middle + i];

}

// 分治的思想 ： 只要数组的元素不是1个，那么继续拆分，排序

**if** (leftArray.length > 1) {

//递归拆分-排序,本例leftArray是[3,1],它会被拆分成[3],[1]两个有序数组,合并以后就是有序的[1,3]

//具体思路可以参考示例代码一

**int**[] merageSort = *merageSort*(leftArray);

// 把排序完成的[1,3]赋值给原来无序的[3,1]

leftArray = merageSort;

}

**if** (rightArray.length > 1) {

//[2,4]

**int**[] merageSort = *merageSort*(rightArray);

//[4,2] = [2,4] 有序替换无序

rightArray = merageSort;

}

/\*

\* 在if中

\* leftArray由无序的[3,1]状态已经被调整成了有序的[1,3]状态

\* rightArray由无序的[4,2]状态已经被调整成了有序的[2,4]状态

\* 内部有序以后，直接合并即可。

\*/

**int**[] merge = *merge*(leftArray, rightArray);

**return** merge;

}

#### 分治-归并示例代码三

网上的思路 ： 将一个要排序的数组按照角标来划分左右即可，没必要真正创建左边和右边的数组，只需要创建一个临时的大数组，用来存储归并以后的元素即可。这个思路能较少的创建临时数组。相对来讲节省空间。

/\*\*

\* 合并数组

\* **@param** arr ： 要合并的数组

\* **@param** left ：要合并数组的左边起始点

\* **@param** middle ：要合并数组的中间位置

\* **@param** right ：要合并数组的右边终点

\*/

**public** **static** **void** merge(**int**[] arr, **int** left, **int** middle, **int** right) {

// 存储归并后的临时数据

**int**[] temp = **new** **int**[right - left + 1];

// 声明记录数组元素的临时角标

**int** leftIndex = left; // 数组的左半部分起始位置

**int** middleIndex = middle + 1; // 数组的右半部分起始位置

**int** index = 0; // 存储归并元素数组的角标

**while** (leftIndex <= middle && middleIndex <= right) {

// 比较数组的左右两部分

**if** (arr[leftIndex] < arr[middleIndex]) {

temp[index] = arr[leftIndex];

leftIndex++;

} **else** {

temp[index] = arr[middleIndex];

middleIndex++;

}

index++;

}

// 处理剩余的若干元素

**while** (leftIndex <= middle) {

temp[index] = arr[leftIndex];

leftIndex++;

index++;

}

**while** (middleIndex <= right) {

temp[index] = arr[middleIndex];

index++;

middleIndex++;

}

System.***out***.println(Arrays.*toString*(temp));

// 把临时数组中的元素（已经有序）还给原始数组

**for** (**int** i = 0; i < temp.length; i++) {

arr[left + i] = temp[i];

}

}

/\*\*

\*

\* **@param** arr 要排序的数组

\* **@param** left 要排序数组的左边起点

\* **@param** right 要排序数组的右边终点

\*/

**public** **static** **void** mergeSort(**int**[] arr, **int** left, **int** right) {

// 递归出口

**if** (left == right) {

**return**;

}

// 要排序数组的元素的中间位置

**int** middle = (right + left) / 2;

// 分治思想 ：处理要排序数组的左边

*mergeSort*(arr, left, middle);

// 处理要排序数组的右边

*mergeSort*(arr, middle + 1, right);

// 合并数组的左右两边

*merge*(arr, left, middle, right);

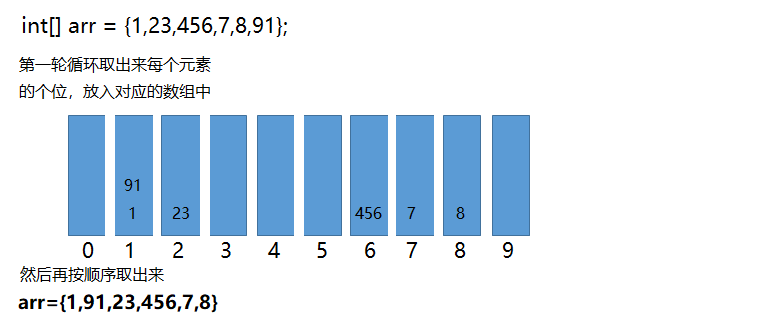
}

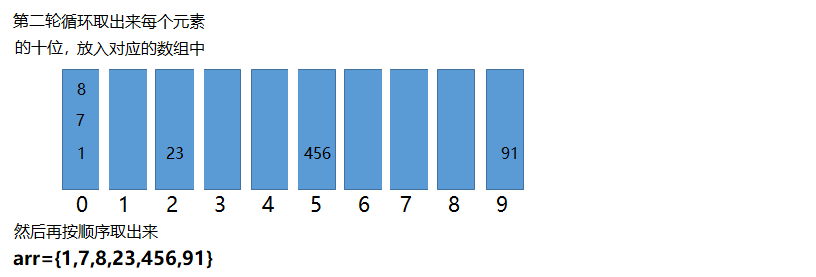
对于merge函数而言，middle参数完全可以省略，在函数内部计算求得。自己想想，很简单的哟。

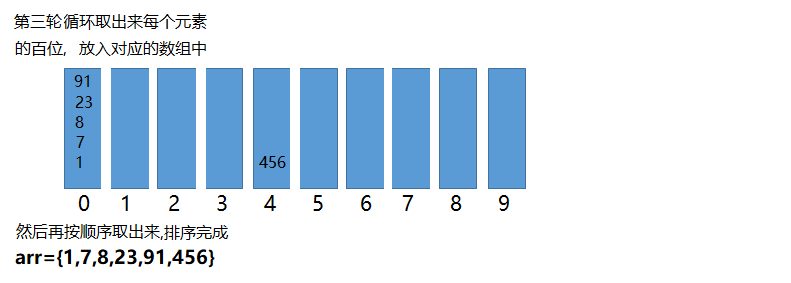
### 基数排序

基数排序是逐个比较数组元素中的每个位数上的大小的一种排序方式。

#### 基数排序图解







循环N轮即可完成排序，N的值等于数组中最大值的位数。

#### 基数排序示例代码一

**public** **static** **void** radixSort(**int**[] arr) {

//存放数组中元素的十个数组

**int**[] zero = **new** **int**[arr.length];

**int**[] one = **new** **int**[arr.length];

**int**[] two = **new** **int**[arr.length];

**int**[] three = **new** **int**[arr.length];

**int**[] four = **new** **int**[arr.length];

**int**[] five = **new** **int**[arr.length];

**int**[] six = **new** **int**[arr.length];

**int**[] seven = **new** **int**[arr.length];

**int**[] eight = **new** **int**[arr.length];

**int**[] nine = **new** **int**[arr.length];

// 获得数组中的最大数字

**int** temMaxValue = arr[0];

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

**if** (arr[i] > temMaxValue) {

temMaxValue = arr[i];

}

}

// 计算本次排序总共需要n轮

**int** n = String.*valueOf*(temMaxValue).length();

//取出"位数"的变量

**int** tempN = 1;

**for** (**int** i = 1; i <= n; i++, tempN \*= 10) { //tempN每次\*=10就可以取出来个位,十位,百位...

//十个数组，十个变量，用于记录每个数组中存储数字的个数，而且每轮循环以后要清零

**int** zIndex = 0, aIndex = 0, bIndex = 0, cIndex = 0, dIndex = 0, eIndex = 0, fIndex = 0, gIndex = 0,

hIndex = 0, iIndex = 0, totalIndex = 0;

**for** (**int** j = 0; j < arr.length; j++) {

// 第一轮取个位，第二轮取出十位，第三轮取出百位...全靠temN

**int** tempMod = arr[j] / tempN % 10;

**switch** (tempMod) {

**case** 0:

zero[zIndex] = arr[j];

zIndex++;

**break**;

**case** 1:

one[aIndex] = arr[j];

aIndex++;

**break**;

**case** 2:

two[bIndex] = arr[j];

bIndex++;

**break**;

**case** 3:

three[cIndex] = arr[j];

cIndex++;

**break**;

**case** 4:

four[dIndex] = arr[j];

dIndex++;

**break**;

**case** 5:

five[eIndex] = arr[j];

eIndex++;

**break**;

**case** 6:

six[fIndex] = arr[j];

fIndex++;

**break**;

**case** 7:

seven[gIndex] = arr[j];

gIndex++;

**break**;

**case** 8:

eight[hIndex] = arr[j];

hIndex++;

**break**;

**case** 9:

nine[iIndex] = arr[j];

iIndex++;

**break**;

}

}

// 再按顺序取出来

**for** (**int** j = 0; j < zIndex; j++) {

arr[totalIndex] = zero[j];

totalIndex++;

}

**for** (**int** j = 0; j < aIndex; j++) {

arr[totalIndex] = one[j];

totalIndex++;

}

**for** (**int** j = 0; j < bIndex; j++) {

arr[totalIndex] = two[j];

totalIndex++;

}

**for** (**int** j = 0; j < cIndex; j++) {

arr[totalIndex] = three[j];

totalIndex++;

}

**for** (**int** j = 0; j < dIndex; j++) {

arr[totalIndex] = four[j];

totalIndex++;

}

**for** (**int** j = 0; j < eIndex; j++) {

arr[totalIndex] = five[j];

totalIndex++;

}

**for** (**int** j = 0; j < fIndex; j++) {

arr[totalIndex] = six[j];

totalIndex++;

}

**for** (**int** j = 0; j < gIndex; j++) {

arr[totalIndex] = seven[j];

totalIndex++;

}

**for** (**int** j = 0; j < hIndex; j++) {

arr[totalIndex] = eight[j];

totalIndex++;

}

**for** (**int** j = 0; j < iIndex; j++) {

arr[totalIndex] = nine[j];

totalIndex++;

}

}

}

#### 基数排序示例代码二

**public** **static** **void** radixSort(**int**[] arr) {

// 用一个二位数组存储排序中临时需要的十个数组

**int**[][] tempArr = **new** **int**[10][arr.length];

//用一个一维数组存储那十个数组中每个数组每轮存储了几个数字

**int**[] indexs = **new** **int**[10];

// 获得数组中的最大数字

**int** temMaxValue = arr[0];

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

**if** (arr[i] > temMaxValue) {

temMaxValue = arr[i];

}

}

// 计算本次排序总共需要n轮

**int** n = String.*valueOf*(temMaxValue).length();

// 取出"位数"的变量

**int** tempN = 1;

**for** (**int** i = 1; i <= n; i++, tempN \*= 10) { // tempN每次\*=10就可以取出来个位,十位,百位...

**for** (**int** j = 0; j < arr.length; j++) {

// 第一轮取个位，第二轮取出十位，第三轮取出百位...全靠temN

**int** tempMod = arr[j] / tempN % 10;

//放入到指定的二维数组中

tempArr[tempMod][indexs[tempMod]] = arr[j];

//该数组记录了十个数组中每个数组放入了多数个元素

indexs[tempMod]++;

}

**int** totalIndex = 0;

// 再按顺序取出来

**for** (**int** j = 0; j < tempArr.length; j++) {

**int**[] tempOneArr = tempArr[j];

**int** tempLength = indexs[j];

**for** (**int** k = 0; k < tempLength; k++) {

arr[totalIndex] = tempOneArr[k];

totalIndex++;

}

//每轮大循环结束，记录存放了多数个元素的数字要清零，为下一轮循环准备重新计数

indexs[j] = 0;

}

}

}

#### 基数排序示例代码三

基数排序要求有十个临时数组用于临时排序，而且存入的顺序与取出的顺序一致，那么这就是一个典型的队列结构。接下来可以把那十个数组使用十个队列来替代。

**public** **static** **void** radixSort(**int**[] arr) {

// 创建十个队列

队列结构<Integer>[] myQueue = **new** 队列结构[10];

**for** (**int** i = 0; i < myQueue.length; i++) {

myQueue[i] = **new** 队列结构<Integer>();

}

// 获得数组中的最大数字

**int** temMaxValue = arr[0];

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

**if** (arr[i] > temMaxValue) {

temMaxValue = arr[i];

}

}

// 计算本次排序总共需要n轮

**int** n = String.*valueOf*(temMaxValue).length();

// 取出"位数"的变量

**int** tempN = 1;

**for** (**int** i = 1; i <= n; i++, tempN \*= 10) { // tempN每次\*=10就可以取出来个位,十位,百位...

**for** (**int** j = 0; j < arr.length; j++) {

// 第一轮取个位，第二轮取出十位，第三轮取出百位...全靠temN

**int** tempMod = arr[j] / tempN % 10;

// 放入到指定的队列中

myQueue[tempMod].add(arr[j]);

}

**int** totalIndex = 0;

// 再按顺序取出来

**for** (**int** j = 0; j < myQueue.length; j++) {

队列结构<Integer> tempQueue = myQueue[j];

**while** (!tempQueue.isEmpty()) {

arr[totalIndex] = tempQueue.poll();

totalIndex++;

}

}

}

}