**目录**

[堆排序算法的总体情况](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label3)

[时间复杂度](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label4)

[算法稳定性](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label5)

* **[完整参考代码](http://www.cnblogs.com/jingmoxukong/p/4303826.html" \l "_label6)**

[JAVA版本](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label7)

* [**参考资料**](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label8)
* [**相关阅读**](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_label9)

[回到顶部](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_labelTop)

**堆的概念**

在介绍堆排序之前，首先需要说明一下，堆是个什么玩意儿。

**堆**是一棵**顺序存储**的**完全二叉树**。

其中每个结点的关键字都**不大于**其孩子结点的关键字，这样的堆称为**小根堆**。

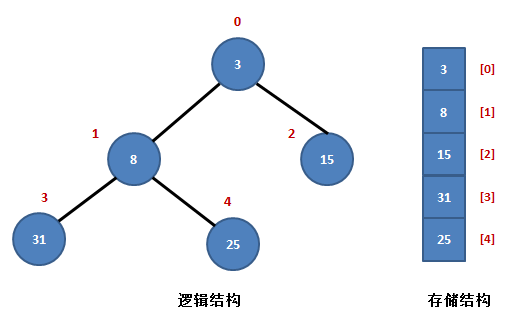
其中每个结点的关键字都**不小于**其孩子结点的关键字，这样的堆称为**大根堆**。

举例来说，对于n个元素的序列{R0, R1, ... , Rn}当且仅当满足下列关系之一时，称之为堆：

**(1) Ri <= R2i+1 且 Ri <= R2i+2 (小根堆)**

**(2) Ri >= R2i+1 且 Ri >= R2i+2 (大根堆)**

其中i=1,2,…,n/2向下取整;



如上图所示，序列R{3, 8, 15, 31, 25}是一个典型的小根堆。

堆中有两个父结点，元素3和元素8。

元素3在数组中以R[0]表示，它的左孩子结点是R[1]，右孩子结点是R[2]。

元素8在数组中以R[1]表示，它的左孩子结点是R[3]，右孩子结点是R[4]，它的父结点是R[0]。可以看出，它们**满足以下规律**：

设当前元素在数组中以**R[i]**表示，那么，

(1) 它的**左孩子结点**是：**R[2\*i+1]**;

(2) 它的**右孩子结点**是：**R[2\*i+2]**;

(3) 它的**父结点**是：**R[(i-1)/2]**;

(4) R[i] <= R[2\*i+1] 且 R[i] <= R[2i+2]。

[回到顶部](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_labelTop)

**要点**

首先，按堆的定义将数组R[0..n]调整为堆（这个过程称为创建初始堆），交换R[0]和R[n]；

然后，将R[0..n-1]调整为堆，交换R[0]和R[n-1]；

如此反复，直到交换了R[0]和R[1]为止。

以上思想可归纳为两个操作：

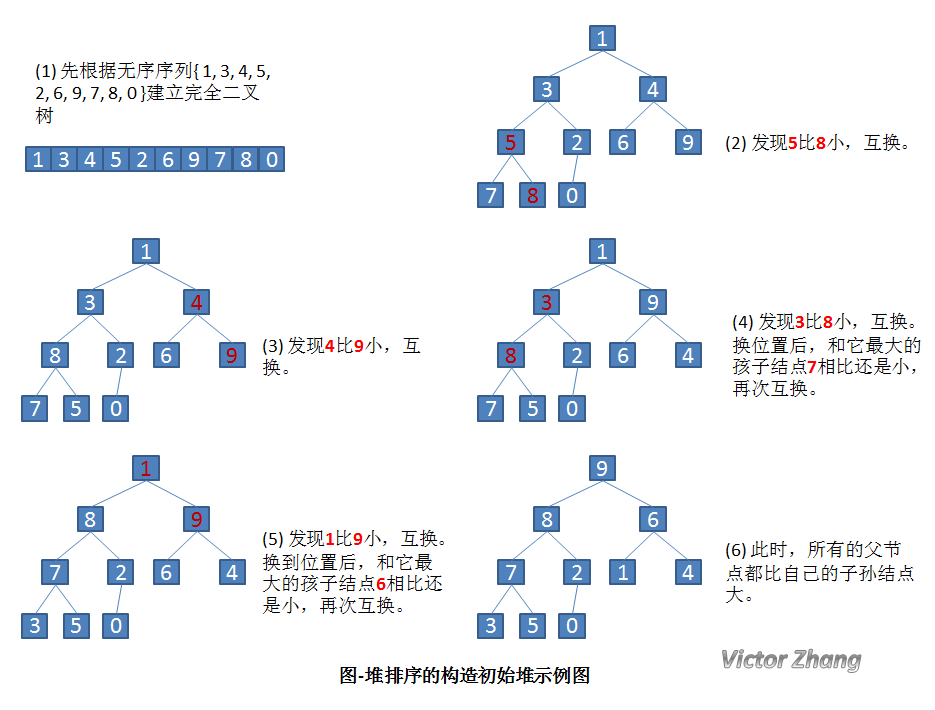
（1）根据初始数组去**构造初始堆**（构建一个完全二叉树，保证所有的父结点都比它的孩子结点数值大）。

（2）每次**交换第一个和最后一个元素，输出最后一个元素**（最大值），然后把剩下元素**重新调整**为大根堆。

当输出完最后一个元素后，这个数组已经是按照从小到大的顺序排列了。

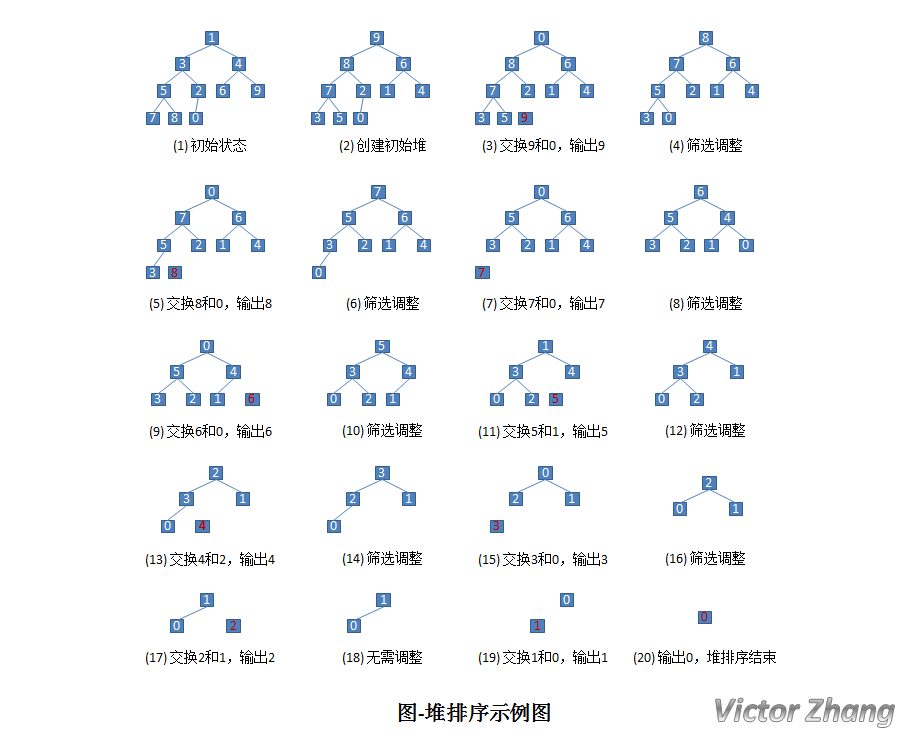
先通过详细的实例图来看一下，如何构建初始堆。

设有一个无序序列 { 1, 3, 4, 5, 2, 6, 9, 7, 8, 0 }。

[](http://images2015.cnblogs.com/blog/318837/201604/318837-20160422104522991-406805984.png)

构造了初始堆后，我们来看一下完整的堆排序处理：

还是针对前面提到的无序序列 { 1, 3, 4, 5, 2, 6, 9, 7, 8, 0 } 来加以说明。

[](http://images2015.cnblogs.com/blog/318837/201604/318837-20160422104524038-1723180638.png)

相信，通过以上两幅图，应该能很直观的演示堆排序的操作处理。

**核心代码**

[复制代码](javascript:void(0);)

public void HeapAdjust(int[] array, int parent, int length) {  
    int temp = array[parent]; // temp保存当前父节点  
    int child = 2 \* parent + 1; // 先获得左孩子  
   
    while (child < length) {  
        // 如果有右孩子结点，并且右孩子结点的值大于左孩子结点，则选取右孩子结点  
        if (child + 1 < length && array[child] < array[child + 1]) {  
            child++;  
        }  
   
        // 如果父结点的值已经大于孩子结点的值，则直接结束  
        if (temp >= array[child])  
            break;  
   
        // 把孩子结点的值赋给父结点  
        array[parent] = array[child];  
   
        // 选取孩子结点的左孩子结点,继续向下筛选  
        parent = child;  
        child = 2 \* child + 1;  
    }  
   
    array[parent] = temp;  
}  
   
public void heapSort(int[] list) {  
    // 循环建立初始堆  
    for (int i = list.length / 2; i >= 0; i--) {  
        HeapAdjust(list, i, list.length - 1);  
    }  
   
    // 进行n-1次循环，完成排序  
    for (int i = list.length - 1; i > 0; i--) {  
        // 最后一个元素和第一元素进行交换  
        int temp = list[i];  
        list[i] = list[0];  
        list[0] = temp;  
   
        // 筛选 R[0] 结点，得到i-1个结点的堆  
        HeapAdjust(list, 0, i);  
        System.out.format("第 %d 趟: \t", list.length - i);  
        printPart(list, 0, list.length - 1);  
    }  
}

[复制代码](javascript:void(0);)

[回到顶部](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_labelTop)

**算法分析**

**堆排序算法的总体情况**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **排序类别** | **排序方法** | **时间复杂度** | | | **空间复杂度** | **稳定性** | **复杂性** |
| **平均情况** | **最坏情况** | **最好情况** |
| 选择排序 | 堆排序 | O(nlog2n) | O(nlog2n) | O(nlog2n) | O(1) | 不稳定 | 较复杂 |

**时间复杂度**

堆的存储表示是**顺序的**。因为堆所对应的二叉树为完全二叉树，而完全二叉树通常采用顺序存储方式。

当想得到一个序列中第**k**个最小的元素之前的部分排序序列，最好采用堆排序。

因为堆排序的时间复杂度是**O(n+klog2n)**，若k≤n/log2n，则可得到的时间复杂度为**O(n)**。

**算法稳定性**

堆排序是一种**不稳定**的排序方法。

因为在堆的调整过程中，关键字进行比较和交换所走的是该结点到叶子结点的一条路径，

因此对于相同的关键字就可能出现排在后面的关键字被交换到前面来的情况。

[回到顶部](http://www.cnblogs.com/jingmoxukong/p/4303826.html#_labelTop)

**完整参考代码**

**JAVA版本**

**代码实现**

以下范例是对上文提到的无序序列 { 1, 3, 4, 5, 2, 6, 9, 7, 8, 0 } 进行排序。

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

**运行结果**

[复制代码](javascript:void(0);)

排序前:    1   3   4   5   2   6   9   7   8   0    
第 1 趟:   8   7   6   5   2   1   4   3   0   9    
第 2 趟:   7   5   6   3   2   1   4   0   8   9    
第 3 趟:   6   5   4   3   2   1   0   7   8   9    
第 4 趟:   5   3   4   0   2   1   6   7   8   9    
第 5 趟:   4   3   1   0   2   5   6   7   8   9    
第 6 趟:   3   2   1   0   4   5   6   7   8   9    
第 7 趟:   2   0   1   3   4   5   6   7   8   9    
第 8 趟:   1   0   2   3   4   5   6   7   8   9    
第 9 趟:   0   1   2   3   4   5   6   7   8   9    
排序后:    0   1   2   3   4   5   6   7   8   9