经典排序算法分析和代码-上篇

前言:这一篇文章中我们将讨论数组排序的问题，对于数据量比较大的，不能在内存中完成排序的，

必须在磁盘上完成排序类型叫作外部排序，本篇将不讨论。

对于内部排序的一些相关知识:

存在几种容易的算法以$O(N^2)$排序，如插入排序。

有一种算法叫做谢尔排序(ShellSort)，它编程非常简单，以$O(N^2)$运行，并在实践中很有效。

还有一些稍微复杂的$O(N\log N)$的排序算法。

任何通用的排序算法均需要$\Omega(N\log N)$次比较。

1.插入排序

最简单的排序算法之一是插入排序(insertion sort)。这是一个对少量元素进行排序的有效算法，

插入排序基于这样一种假设：位置0到位置 p-1上的元素已经是经过排序的，下图显示了一个简单的数组在每一趟排序后的情况。

初始状态	34	8	64	51	32	21	移到的位置
After $p=1$	8	34	64	51	32	21	1
After $p=2$	8	34	64	51	32	21	0
After $p=3$	8	34	51	64	32	21	1
After $p=4$	8	32	34	51	64	21	3
After $p=5$	8	21	32	34	51	64	4

或者可以这样理解，插入排序的机理与打扑克牌时，整理手中牌时的做法一样，在开始摸牌时手中牌是空的，

牌面朝下放在桌子上。接着依次从桌上取一张牌插入到左手牌中正确的位置上，当牌摸完时，左手中的牌就成了有序的状态。

下面给出算法具体代码:


```cpp
//插入排序
void InsertSort(int Nums[], int Length)
{
    for (int i = 0; i < Length; i++)
    {
        int j, temp = Nums[i];//将第 i 个数保存起来
        for (j = i - 1; Nums[j] > temp && j >= 0; j--)//依次将第 i 个数与前面的数比较
        {
            Nums[j + 1] = Nums[j];//如果前面的书比第 i 个数大的，则向后移动
        }
        Nums[j + 1] = temp;  //最后将空出的位置装入上面保存的第 i 个数
    }
}
```

2.谢尔排序

谢尔排序(Shellsort)的名称源于它的发明者 Donald Shell，他通过比较相距一定间隔的元素来工作，

各趟比较所用的距离随着算法的进行二减小，知道只比较相邻元素的最后一趟排序位置。由于这个原因，

谢尔排序有时也叫作缩减增量排序。

初始状态	81	94	11	96	12	35	17	95	28	58	41	75	15
5 排序之后	35	17	11	28	12	41	75	15	96	58	81	94	95
3 排序之后	28	12	11	35	15	41	58	17	94	75	81	96	95
1 排序之后	11	12	15	17	28	35	41	58	75	81	94	95	96

上图可以看到增量分别为5，3，1的排列状态，5排序之后，从第一个元素开始相隔为5的元素变为有序，

同理3排序和1排序之后元素全部为有序了。

下面给出算法的具体代码

[cpp] view plaincopy

```cpp
//谢尔排序
void ShellSort(int Nums[], int Length)
{
    for (int gap = Length / 2; gap > 0; gap /= 2)
    {
        //根据增量 gap 进行插入排序
        for (int i = gap; i < Length; i += gap)
        {
            int j, temp = Nums[i];
            for (j = i; j - gap >= 0; j -= gap)
            {
                if (temp < Nums[j - gap])
```

```
                    Nums[j] = Nums[j - gap];
                else
                    break;
            }
            Nums[j] = temp;
        }
    }
}
```

上面代码中的 gap 增量叫做谢尔增量，增量的使用对于算法性能有影响，使用谢尔增量时，

谢尔排序的最坏情形运行时间为$O(N^2)$，Hibbard 提出一个稍微不同的增量序列

1,3,7....$2^k - 1$,

使用 Hibbard 增量的谢尔排序的最坏情形运行时间为$O(N^{3/2})$。

3.堆排序

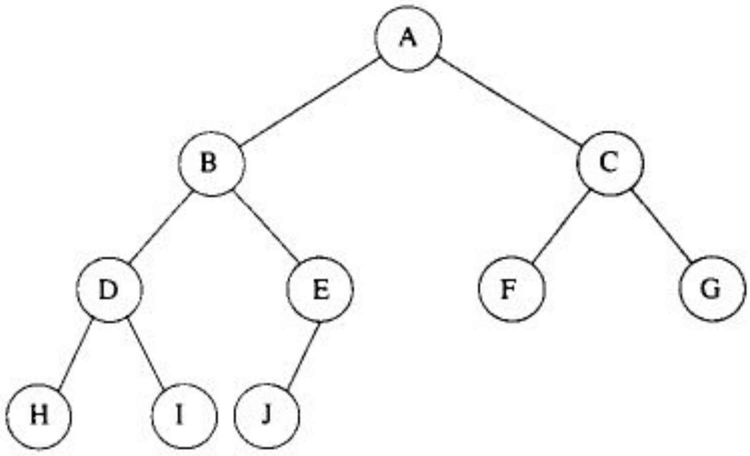
对排序的运行时间为$O(N \log N)$,也是一种原地排序算法(任何时候数组中只有常数个元素存储在输入数组以外),

堆的数据结构可以构成一个有效的优先队列,

它的应用可以参考我以前的文章 http://blog.csdn.net/itcastcpp/article/details/12999595 。

堆是一棵被填满的二叉树，但底部可以例外，底部的节点从左到右的填充，这样的树被称为完全二叉树(complete binary tree)。

如下图:



一棵完全二叉树

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F	G	H	I	J			

完全二叉树有一个很有规律，可以用一个数组表示，而不需要链。对于数组任一位置 i 上的元素，其左儿子节点在2i 上，

右儿子节点在2i+1上，其父节点在2/i 上。因此这里不仅不需要链，遍历该树所需要的操作也及简单，

在大部分计算机上运行得非常快，这种实现的唯一问题在于，最大堆的大小需要事先估计，

但一般情况下这不成问题(而且如果需要我们可以重新调整)。

那么对于堆排序我们可以结合上图的树来理解就非常容易了，最后一个元素位置为 p，

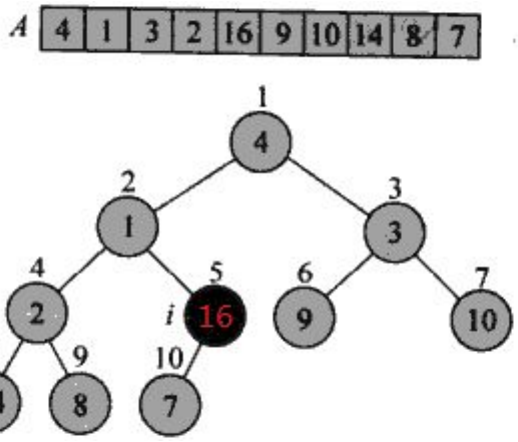
那么我们将 p/2节点和之前分别进行调整，使得1 到 p/2的节点都比其儿子节点大，从 p/2开始调整，

递减到第一个根节点，即可以得到根节点最大，再将根节点和最后节点 p 交换，然后将节点 p 排除在树之外，

从(p-1)/2的节点开始调整，把最大的根节点跟第 p-1个节点交换，那么重复上面过程，直到只剩下根节点，

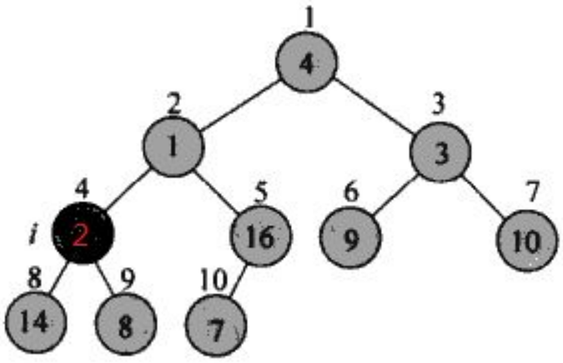
即完成了一次堆排序，排序结果位从小到大排列。

我们现在对下面实例进行分析:



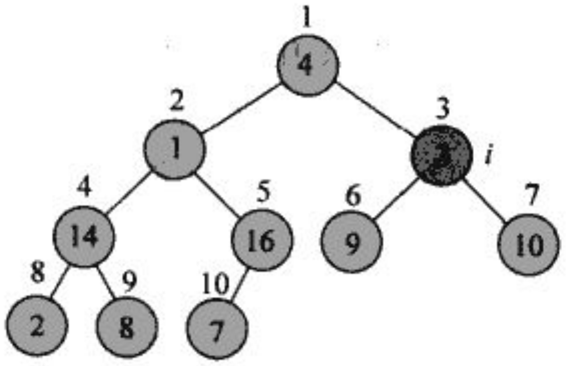
可以看到上面有10个元素，那么先对第5个元素，发现节点5大于10，不用交换，

进入下一步，第4个节点和他的第8、9两个儿子节点比较：

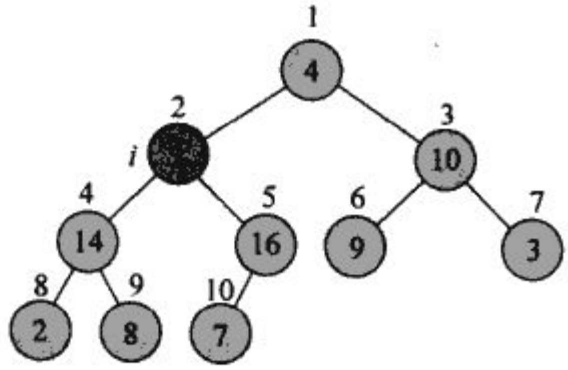


b)

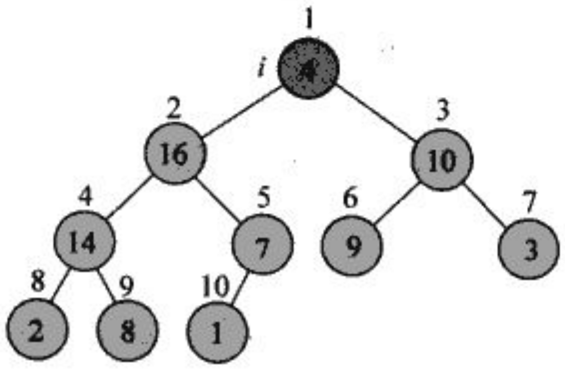
发现第四个节点比第八个节点小，此时交换节点，然后处理继续处理第3个节点，依次类推，直到第一个节点(根节点)如下图：



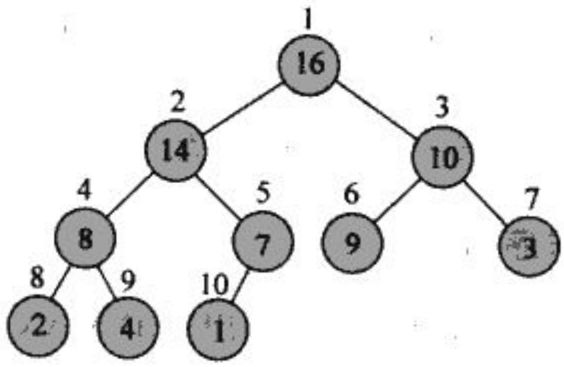
c)



d)



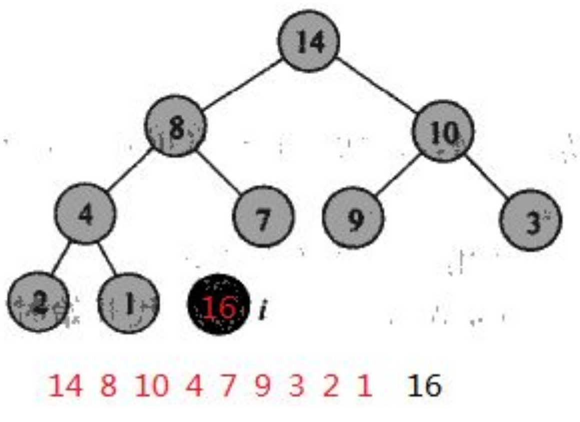
e)



f)

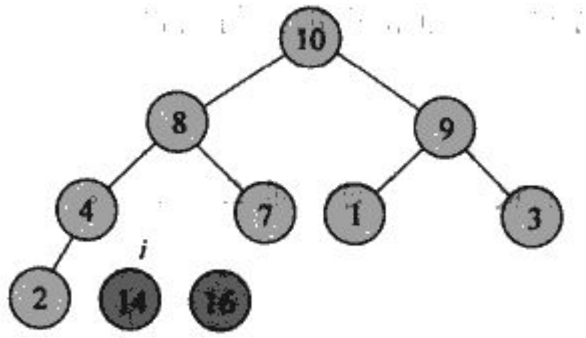
最后得到图上图 F，可以看到根节点16是最大的。

这时我们将根节点和最后一个节点互换，并且从数种去掉，那么只剩下9个节点了。

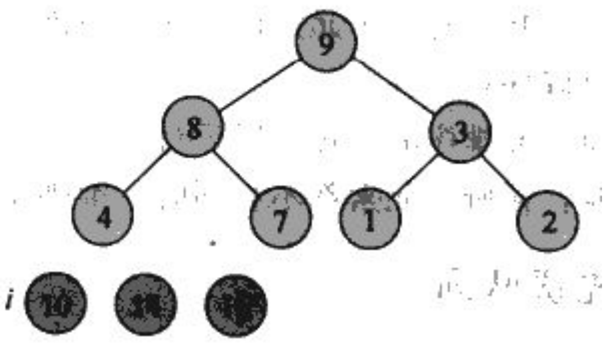


我们接着调整根节点在整棵数种的位置，发现根节点比左右子节点都大，不用调整，

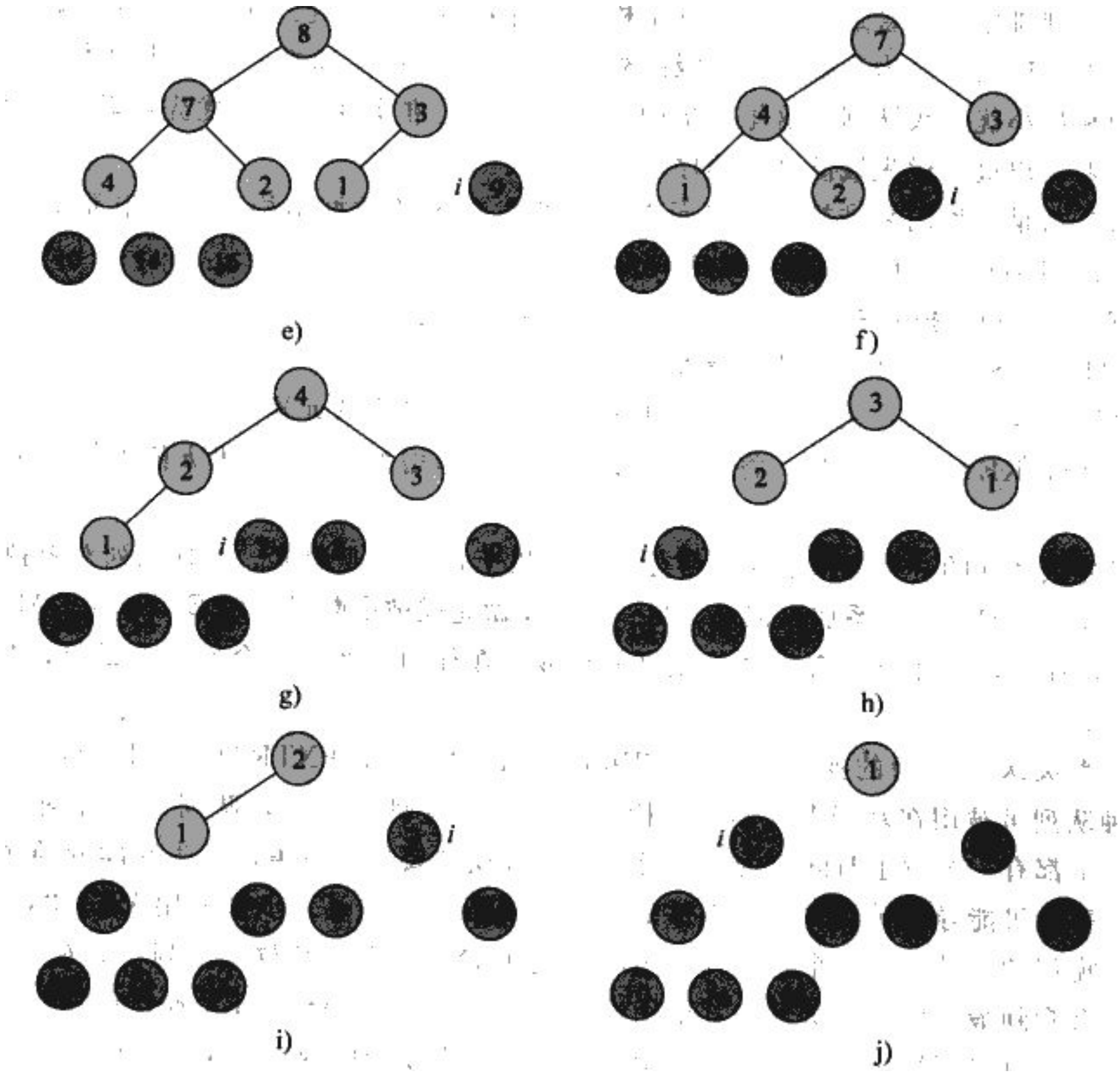
这时我们继续将根节点14和最后一个节点1互换，那么1就变成了根节点，再调整1在根节点中的位置：

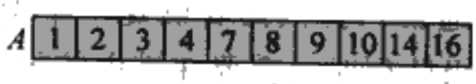


调整后如上图，根节点为10了，再与最后一个节点互换，然后调整根节点位置：



依此类推，知道所有节点从树中分离:





k)

全部分离后，排列的内容为 ，可以看到内容变为有序了。

下面给出算法代码:

[cpp] view plaincopy

```cpp
#include <stdio.h>

//这个函数调整对数组中第 n 个元素的位置
void HeapAdjust(int array[], int n, int length)
{
    int Child;
    for (int i = n; i * 2 <= length; i = Child)
    {
        Child = i * 2;
        if (Child + 1 <= length && array[Child] < array[Child + 1])
            Child++;
        //如果较大的子节点大于父节点则交换位置
        if (array[i] < array[Child])
        {
            int Temp = array[i];
            array[i] = array[Child];
            array[Child] = Temp;
        }
        else
        {
            break;
        }
    }
}

void HeapSort(int array[], int length)
{
    //调整前半部分，保证了最大的值都在前半部分
    for (int i = length / 2; i > 0; i--)
    {
        HeapAdjust(array, i, length);
    }
    for (int i = length-1; i > 0; i--)
    {
        //将最大的数移动到尾部
        int Temp = array[1];
        array[1] = array[i+1];
        array[i+1] = Temp;
        //除去尾部后，调整第一个元素位置
        HeapAdjust(array, 1, i);
    }
}
```

```c

void HeapAdjustLittle(int array[], int num, int length)
{
    //如果输入的数小于这些数，直接返回
    if (num < array[1])
    {
        return;
    }

    //如果输入的数大于数组中最小的数，则赋值，然后调整堆数组
    array[1] = num;
    int Child;
    for (int i = 1; i * 2 <= length; i = Child)
    {
        Child = i * 2;
        if (Child + 1 <= length && array[Child] > array[Child + 1])
            Child++;
        //如果较小的子节点大于父节点则交换位置
        if (array[i] > array[Child])
        {
            int Temp = array[i];
            array[i] = array[Child];
            array[Child] = Temp;
        }
        else
        {
            break;
        }
    }
}

//打印出数组内容
void PrintArray(int array[], int size)
{
    printf("最大的前%d 个数:\n", size);
    for (int i = 0; i < size; i++)
    {
        printf("%3d", array[i]);
    }
    printf("\n");
}

int myarray[] = { 0, 1, 9, 2, 8, 3, 7, 4, 6, 5 , 10};

```

```c
int main()
{
    //将前十个数进行一次堆排序，并输出结果
    HeapSort(myarray, sizeof(myarray) / 4 - 1);
    PrintArray(myarray + 1, sizeof(myarray) / 4 - 1);

    //输入数字，打印出前十个最大的数
    while (1)
    {
        int num = 0;
        scanf("%d", &num);
        HeapAdjustLittle(myarray, num, sizeof(myarray) / 4 - 1);
        PrintArray(myarray + 1, sizeof(myarray) / 4 - 1);
    }

    return 0;
}
```