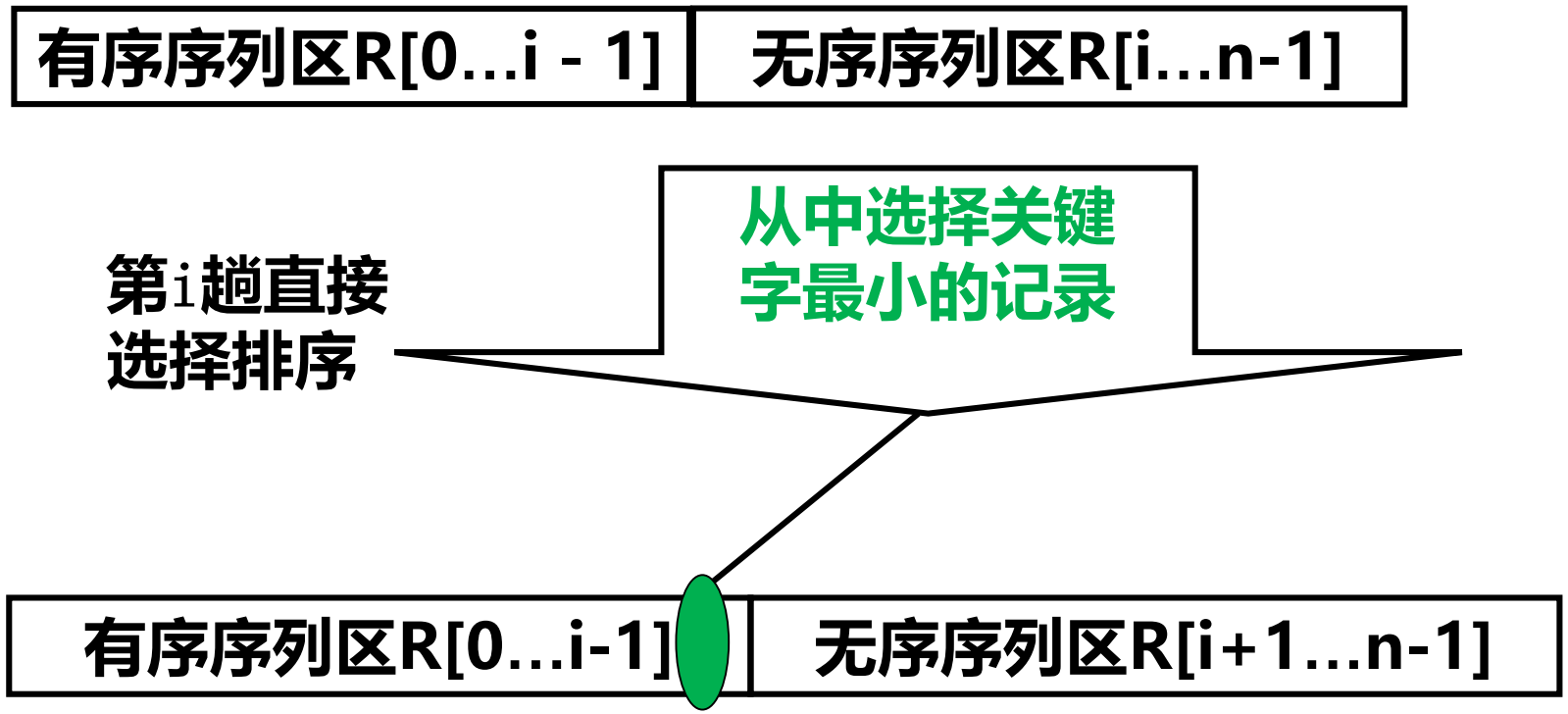
8.3.1直接选择排序

首先在所有记录中选出排序码**最小**的记录，与**第一个记录**交换，然后在**其余**的记录中再选出排序码**最小**的记录与**第二个记录**交换，以此类推，直到所有记录排好序

思考：直接选择排序的比较次数与文件初始状态有关吗?

没有关系

8.3.1直接选择排序


有序序列区R[0…i - 1]
无序序列区R[i…n-1]
第i趟直接
选择排序
从中选择关键
字最小的记录
有序序列区R[0…i-1]
无序序列区R[i+1…n-1]

8.3.1直接选择排序

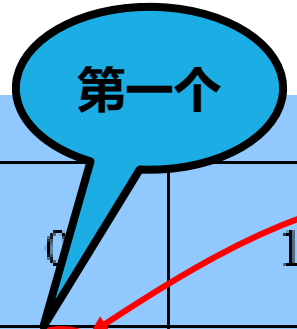


表 8-3 直接选择排序每趟结果

下标 / 趟	0	1	2	3	4	5	6	7
初始序列	15	13(1)	9	46	4	18	13(2)	7
i = 1 （4）	4	13(1)	9	46	15	18	13(2)	7
i = 2 （7）	4	7	9	46	15	18	13(2)	13(1)
i = 3 （9）	4	7	9	46	15	18	13(2)	13(1)
i = 4 (13)	4	7	9	13(2)	15	18	46	13(1)
i = 5 (13)	4	7	9	13(2)	13(1)	18	46	15
i = 6 (15)	4	7	9	13(2)	13(1)	15	46	18
i = 7 (18)	4	7	9	13(2)	13(1)	15	18	46
i = 8 (46)	4	7	9	13(2)	13(1)	15	18	46

算法8-5

```
1   void  SelectSort(SortArr *sortArr)
2   {
3           int i, j;
4           int minPos; //记录最小元素的下标
5           for( i = 0; i < sortArr->cnt-1; i++ ) // n-1趟选择排序
6           {
7                   minPos= i; //记录下最小的值所在的数组下标
8                   for (j = i+1; j < sortArr->cnt; j++) //在无序区中寻找
9                   {
10                    if(sortArr->recordArr[j].key<sortArr->recordArr[minPos].key)
11                    minPos = j;
12                  }
13                  if (minPos != i)  //说明需要交换
14                          Swap(sortArr, minPos, i); //交换记录
15          }
16  }
```

算法分析

时间复杂度：

比较：$\sum_{i=1}^{n-1}(n-i)$=n(n-1)/2
移动：最好：0　　；最坏：3（n-1）
平均时间复杂度：T(n)=O(n^2)

空间复杂度：S(n)=O(1)

稳定性：不稳定

8.3.1直接选择排序：举例

49 38 65 97 49 13 27 76

[13] 38 65 97 49 49 27 76

[13 27] 65 97 49 49 38 76

[13 27 38] 97 49 49 65 76

[13 27 38 49] 97 49 65 76

[13 27 38 49 49] 97 65 76

[13 27 38 49 49 65] 97 76

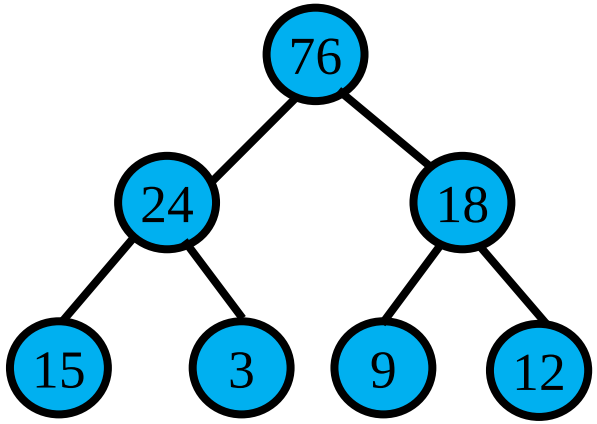
[13 27 38 49 49 65 76] 97

[13 27 38 49 49 65 76 97]

8.3.2 堆排序 heapsort

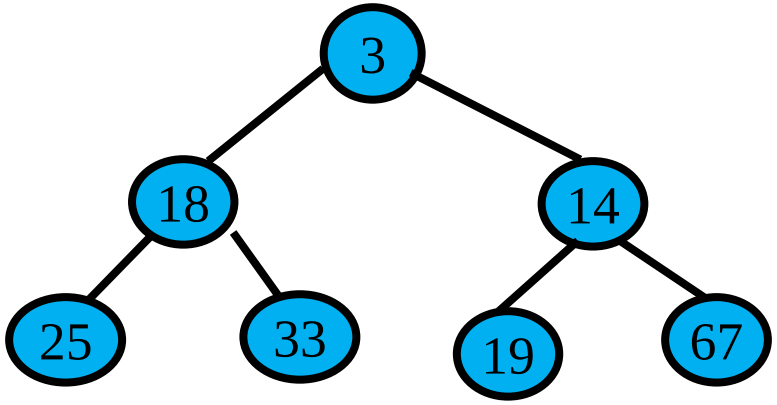
堆是一种特殊的二叉树，可以看作完全二叉树的顺序存储序列

- 所有非叶子结点关键字的值均小（大）于其左右孩子
- 二叉树中任意子树也是堆
- 根结点即是最小（大）值



$$\begin{cases} k_i \geq k_{2i+1} \\ k_i \geq k_{2i+2} \end{cases}$$

大根堆



$$\begin{cases} k_i \leq k_{2i+1} \\ k_i \leq k_{2i+2} \end{cases}$$

小根堆

堆排序基本思想

1. 将序列 $\{k_1,k_2,\ldots,k_n\}$ n个记录建成堆
（堆顶元素必为序列中n个元素的最大值（或最小值））
2. 交换第一个元素与最后一个元素
3. 剩余n-1个记录重新调整为一个堆
4. 重复2、3，直到堆中只有一个记录

筛选

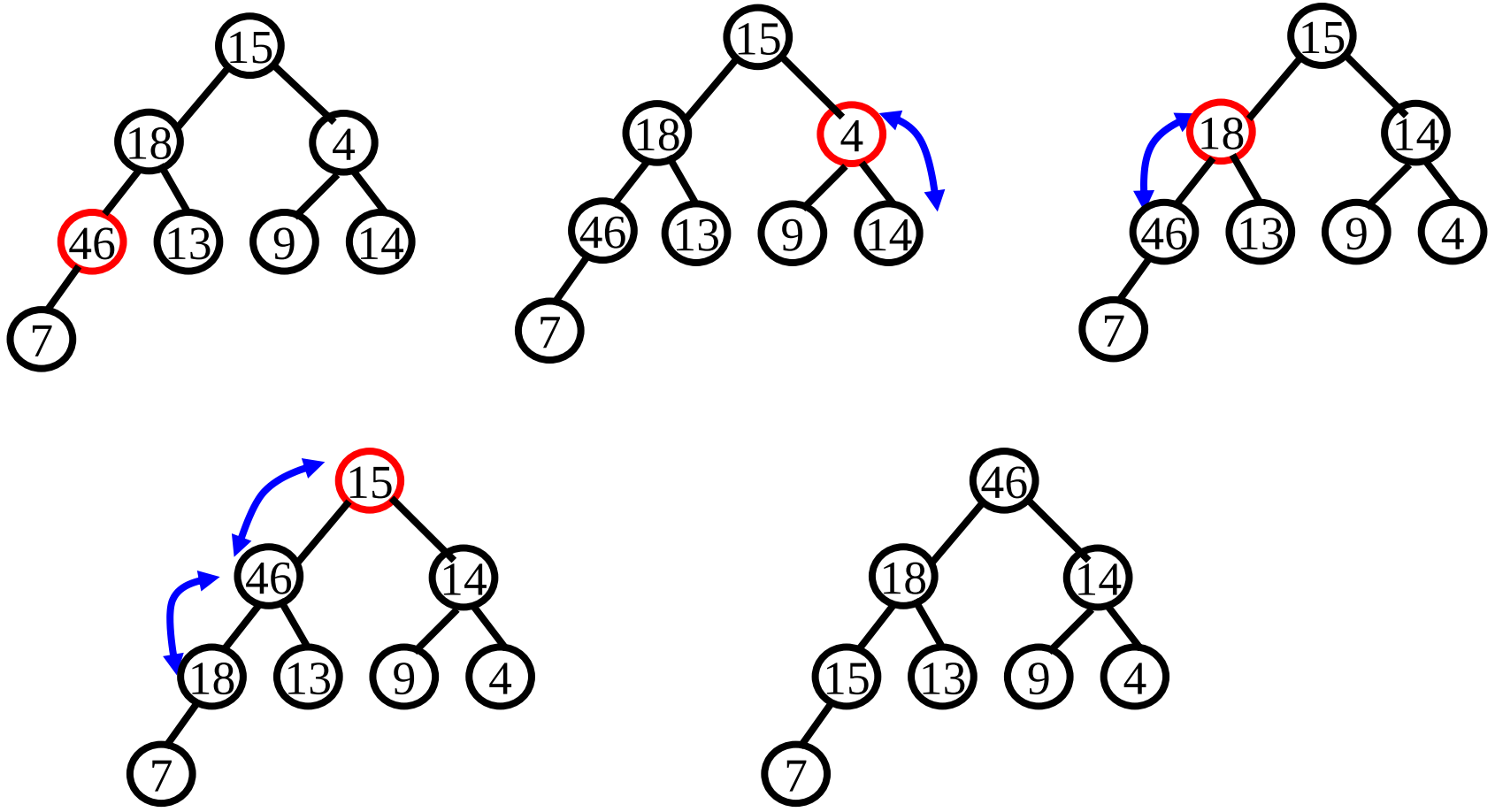
分析：要将无序序列的记录调整为一个堆，必须把这个完全二叉树中以每个结点为根的子树调整为堆；在完全二叉树中，所有序号$i>\lfloor n/2\rfloor-1$的结点都是叶子结点，只需要依次将以序号$\lfloor n/2\rfloor-1$到0的结点作为根的子树都调整为堆即可

方法：从无序序列的第$\lfloor n/2\rfloor-1$个元素起，至第一个元素止，进行反复筛选

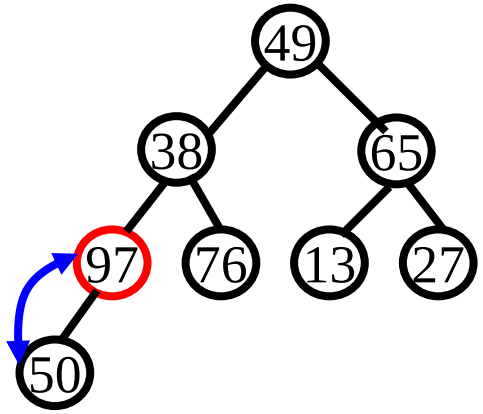
8.3.2 堆排序

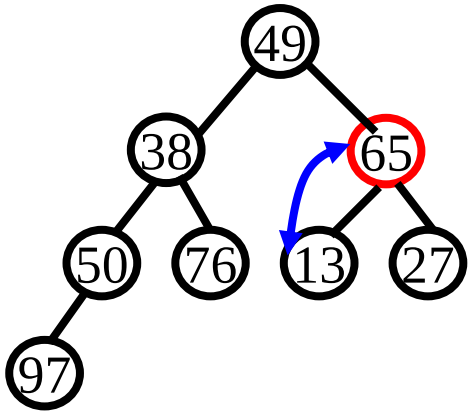
堆排序演示

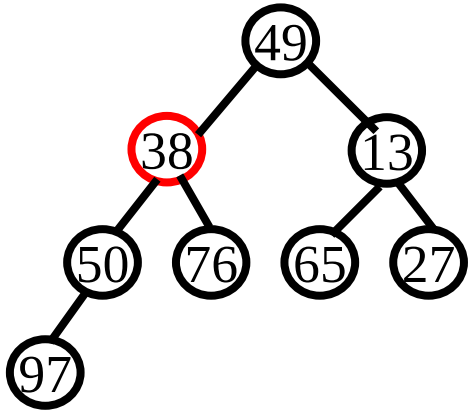
举例： 含8个元素的无序序列（15， 18， 4， 46， 13， 9， 14， 7）
建立大根堆

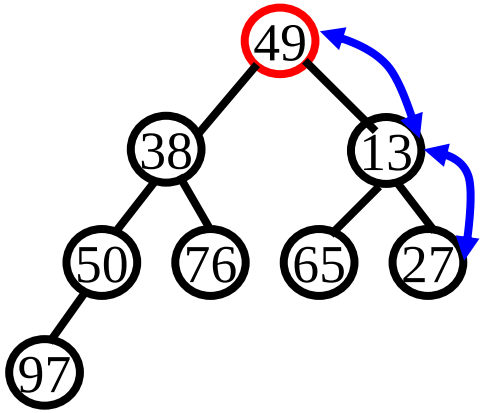


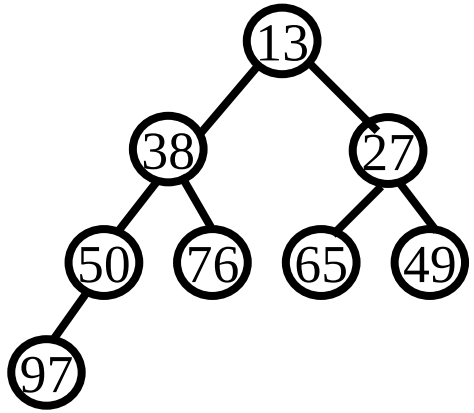
课堂练习： 含8个元素的无序序列（49，38，65，97，76，13，27，50），建立小根堆











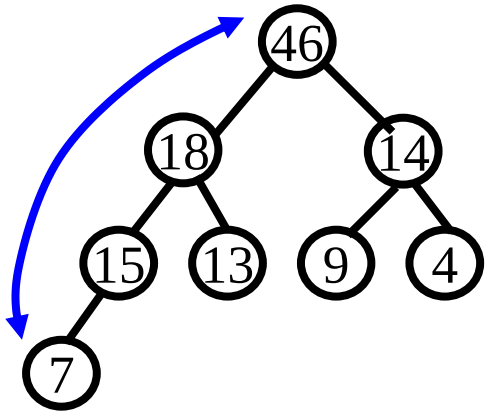
堆排序基本思想

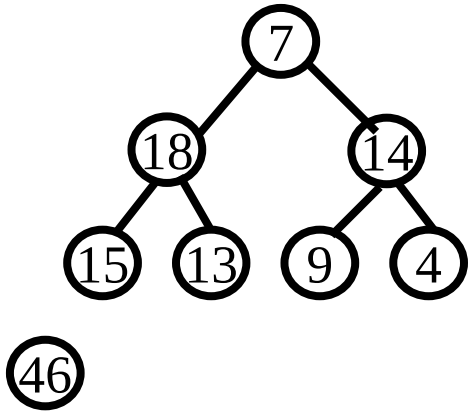
1. 将序列 $\{k_1,k_2,\ldots,k_n\}$ n个记录建成堆
（堆顶元素必为序列中n个元素的最大值（或最小值））
2. 交换第一个元素与最后一个元素
3. 剩余n-1个记录重新调整为一个堆
4. 重复2、3，直到堆中只有一个记录

筛选

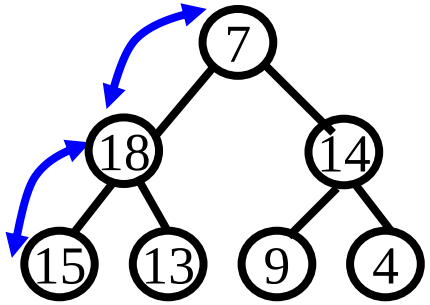
方法：调整根节点，将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆

举例： 含8个元素的无序序列（15， 18， 4， 46， 13， 9， 14， 7）堆排序

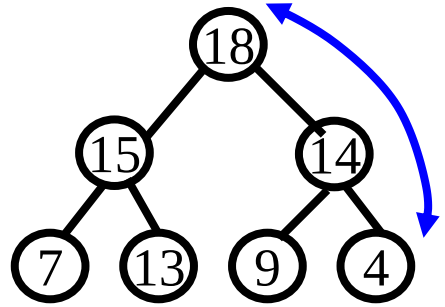


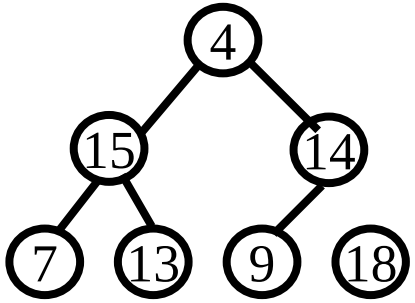


排序好的元素：46

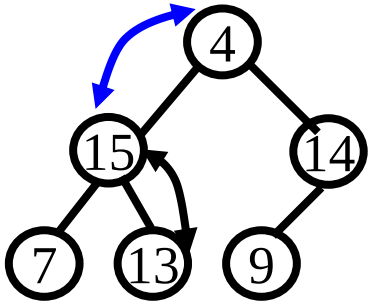


调整为堆

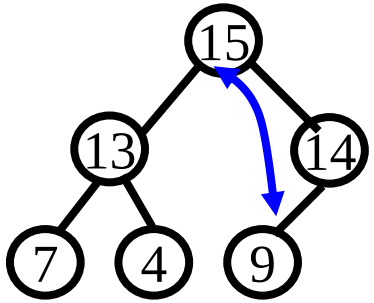


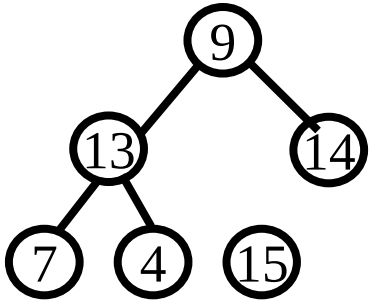


排序好的元素：18 46



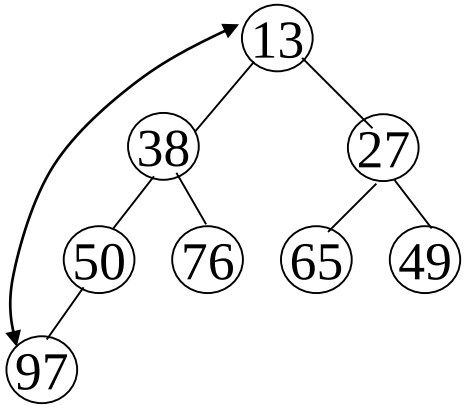
调整为堆

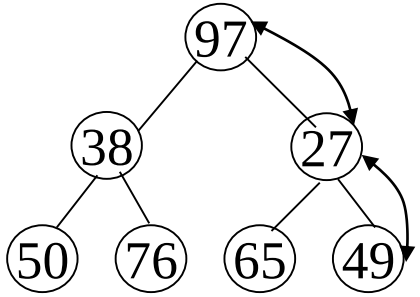




排序好的元素： 15 18 46

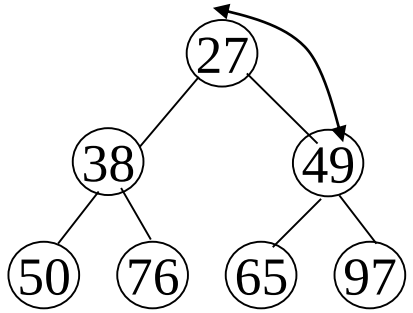
课堂练习： 含8个元素的无序序列（49，38，65，97，76，13，27，50），建立小根堆





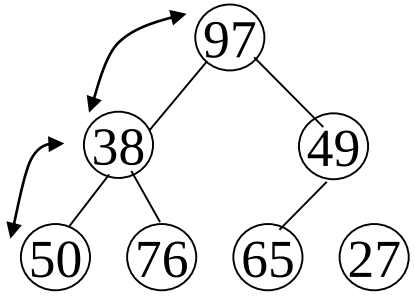
13

排序好的元素：13



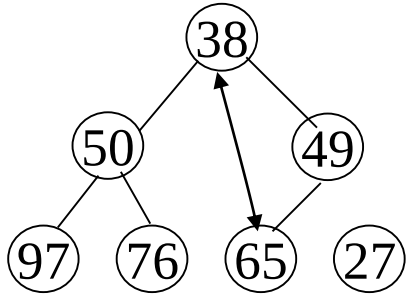
13

调整



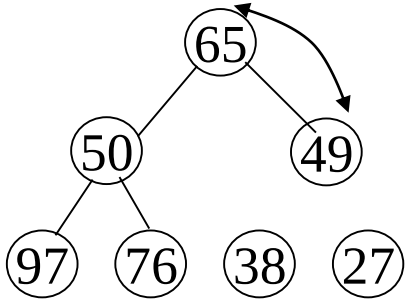
13

排序好的元素：13 27



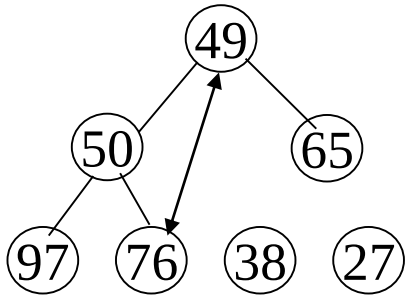
13

调整



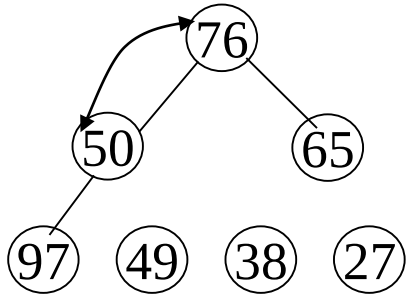
13

排好序的元素：13 27 38



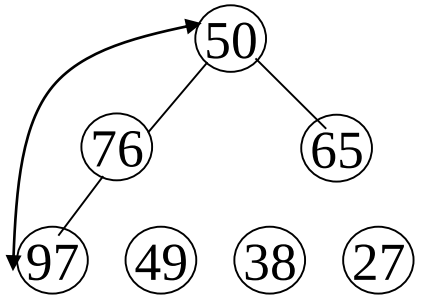
13

输出：13 27 38



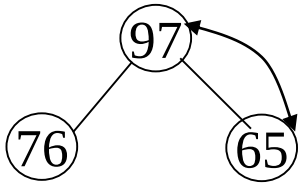
13

输出：13 27 38 49



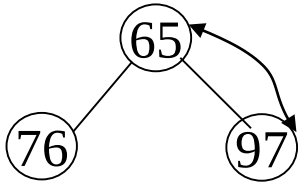
13

输出：13 27 38 49



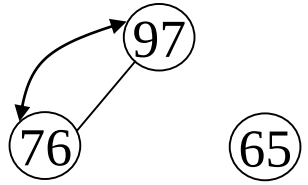
13

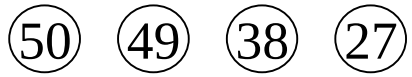
输出：13 27 38 49 50



13

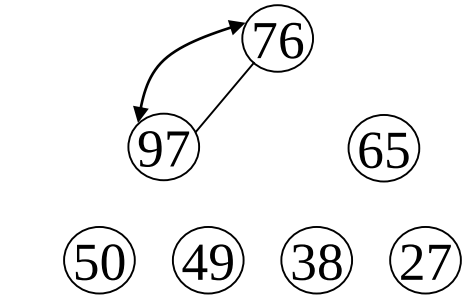
输出：13 27 38 49 50



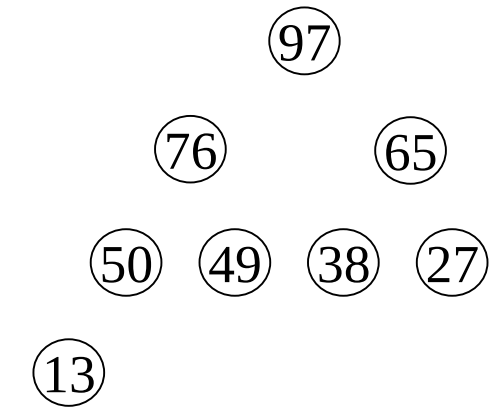


13

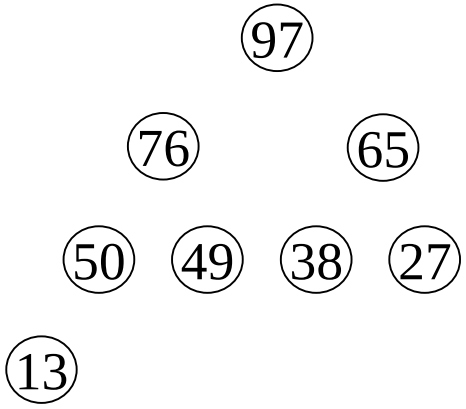
输出：13 27 38 49 50 65



输出：13 27 38 49 50 65



输出：13 27 38 49 50 65 76



输出：13 27 38 49 50 65 76 97

堆排序算法　算法8-7

```
1   void HeapSort(SortArr *sortArr,int size) //堆排序
2   {
3           int i;
4           for(i = size/2 - 1; i >= 0; i--)
5         //从倒数第一个非叶子结点开始调整,一致调整到根结点，形成堆
6                   HeapAdjust(sortArr,i,size);
7           //每次取树根元素跟未排序尾部交换，之后，再重新调整堆
8           for (i = size - 1; i >= 1; i--)
9           {
10                  Swap(sortArr, 0, i); //交换
11                  //重新调整一个元素的位置就可以了（刚调整到树根位置的那个值）
12                  HeapAdjust(sortArr, 0, i);
13          }
14  }
```

算法8-6

```
void HeapAdjust(SortArr *sortArr, int father, int size) //调整过程
{
    int lchild; int rchild; int max;
        //将father中的值放到堆中正确的位置上
        while (father < size){
                lchild = father * 2 + 1;  rchild = lchild + 1;    //左孩子,右孩子
                if( lchild >= size)   break;
                //寻找father,lchild,rchild中最大的，将最大值与father值做交换
                max = lchild;
                //右孩子的下标不要越界了
                if(rchild < size && sortArr->recordArr[rchild].key >sortArr->recordArr[lchild].key)
                        max = rchild;
                 if(sortArr->recordArr[father].key < sortArr->recordArr[max].key)
                        Swap(sortArr, father, max); father = max;
                else  break;
        }
}
```

堆排序算法

时间复杂度：

初始建堆比较次数为：O(n)；

调整重建堆中比较次数为：　<$O(n*\log_2 n)$；

移动次数小于比较次数；

在最坏的情况下，时间复杂度也是$O(n\log_2 n)$；

空间效率：仅需一个记录大小的辅助空间temp；

稳定性：堆排序是不稳定的；

堆排序适用于n值较大的情况；