数据结构

第二十讲 选择排序和交换排序

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年12月21日

课程内容

- 选择排序
- 交换排序

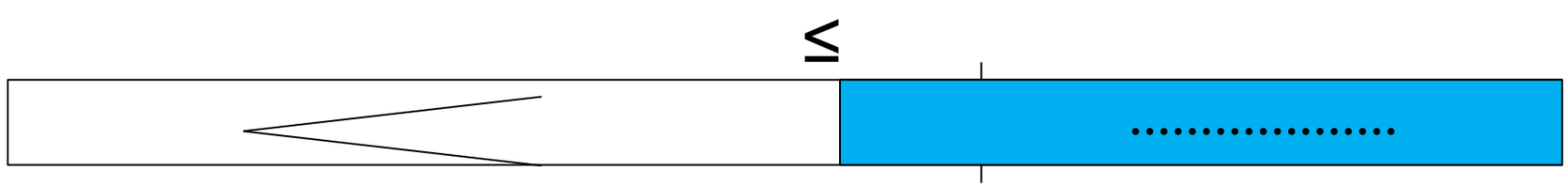
选择排序

- 基本思想：
  - 维护最小的 i 个记录的已排序序列；
  - 每次从剩余未排序的记录中选取关键码最小的记录，排在已排序序列之后，作为序列的第 $i+1$ 个记录；

- 直接选择排序
- 堆排序

直接选择排序

- 以空排序序列开始; 每次从未排序记录中选排序码最小的记录，与未排序段的第一个记录交换;直到所有记录排好序。
- 直接选择排序的比较次数与文件初始状态无关。
- 通过循环顺序比较，维护已找到的最小记录下标。循环结束时得到未排序段最小记录的下标，通过交换扩展已排序序列。
- 未排序段剩下一个元素时就不必再选择。
- 直接选择排序采用顺序存储方式。



已排序段中的最大元素小于等于所有未排序元素

直接选择排序示例

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

直接选择排序算法

```
void  selectSort(SortObject * pvector) {
    int i, j, k;
    RecordNode temp, *data = pvector->record;
    for( i = 0; i < pvector->n-1; i++ ) {                  /* 做n-1次选择排序 */
        k = i;
        for (j = i+1; j < pvector->n; j++)                 /* 在无序段找最小记录*/
            if (data[j].key < data[k].key) k = j;
        if (k != i) {                                      /* 需要时交换记录 */
            temp = data[i];
            data[i] = data[k];
            data[k] = temp;
        }
    }
}
```

算法分析

- 直接选择排序的时间复杂度：
  - 记录移动：最好时 0；最坏时 $O(n)$；
  - 比较：$n(n\text{-}1)/2$ (总是这样)；
  - 总的时间复杂度：$O(n^2)$；
- 稳定性: 不稳定
  - 其中的交换操作是导致不稳定的根源。如果改为移动前面未排序元素腾出空位后存入, 可以把算法改为稳定的。
- 直接选择排序没有适应性，对任何序列都需要 $O(n^2)$ 次比较。
- 实际试验说明其平均排序效率低于直接插入排序算法。

堆排序

- 直接选择排序效率较低:
  - 选择排序的主要操作是比较;
  - 第 i 趟排序需要作 n-i 次比较;
  - 没有利用已做的比较。
- 利用已有比较结果可以提高排序速度。
- 堆排序:[J.Williams 1964]
  - 先把待排序的记录构造成堆;
  - 然后通过从堆中不断选出最大 / 小元素。

关于堆的回顾

- **堆:** n**个元素的序列**$K=\{k_0, k_1, \ldots, k_{n-1}\}$**称为堆，当且仅当满足条件**
  - **(1)** $k_i \geq k_{2i+1}$**且**$k_i \geq k_{2i+2}$**，或者**
  - **(2)** $k_i \leq k_{2i+1}$**且**$k_i \leq k_{2i+2}$**，**$(i=0,1,\ldots,\lfloor n/2 \rfloor -1)$**。**
- **堆与完全二叉树的顺序表示: 堆序性**
  - **满足条件(1)的堆,在完全二叉树中等价于： 每个子二叉树的根均大于等于其左,右子结点;**
    - **根结点最大, 称为大根堆（本节使用）**
  - **满足条件(2)的堆,在完全二叉树中等价于： 每个子二叉树的根均小于等于其左,右子结点;**
    - **根结点最小,称为小根堆（优先队列中使用）**

关于堆的回顾

- 初始建堆:
  - 方法一： 通过把待排序的文件中的记录逐个插入到空堆中。
    - 需要另外开辟与原来文件同样大小的空间。
  - 方法二：利用原始文件的空间，调用一系列sift操作实现。
- 存储结构:
  - 堆排序采用顺序存储方式。

堆排序的基本过程：初始建堆

(1) 把顺序表中元素看作一棵完全二叉树的结点；

(2) 表的后一半元素是叶，每个元素(自然)是一个堆（一个元素的完全二叉树都是堆)；

(3) 对前一半的元素，从后向前逐个考虑和处理:

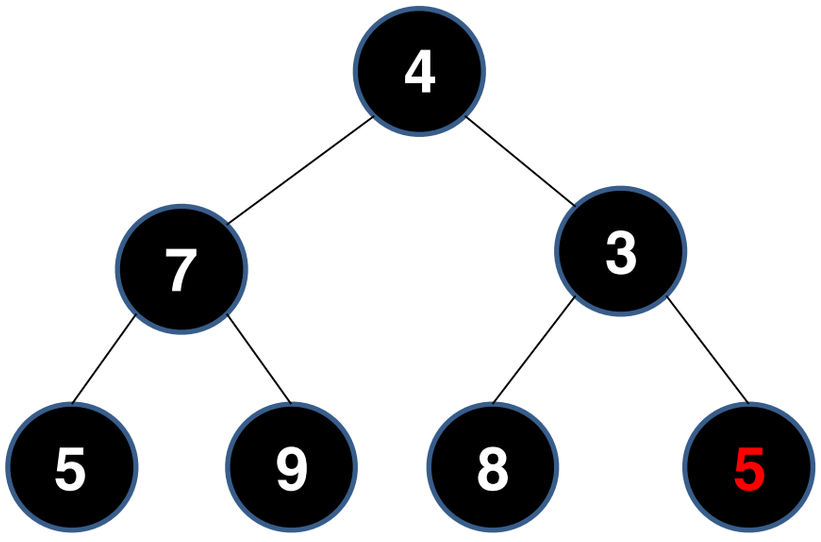
(a) 遇到的每个元素e都是一棵子树的根，其左右子树已经是堆，通过一次筛选，可以把以e为根的子树调整为堆；

(b) 所有元素都处理完时(处理完数组的首元素时)，整个表里的全部元素就形成了一个大根堆。

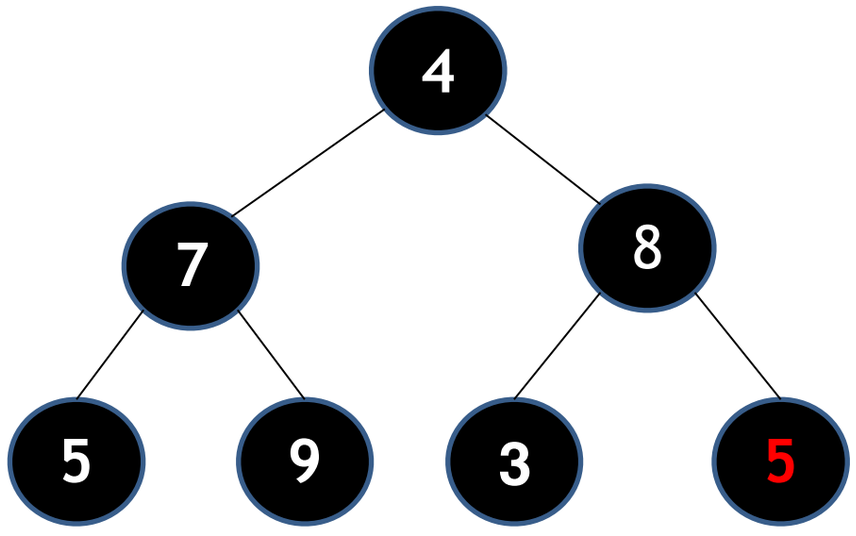
大根堆初始建堆示例

初始序列为: 4, 7, 3, 5, 9, 8, 5,

n=7。

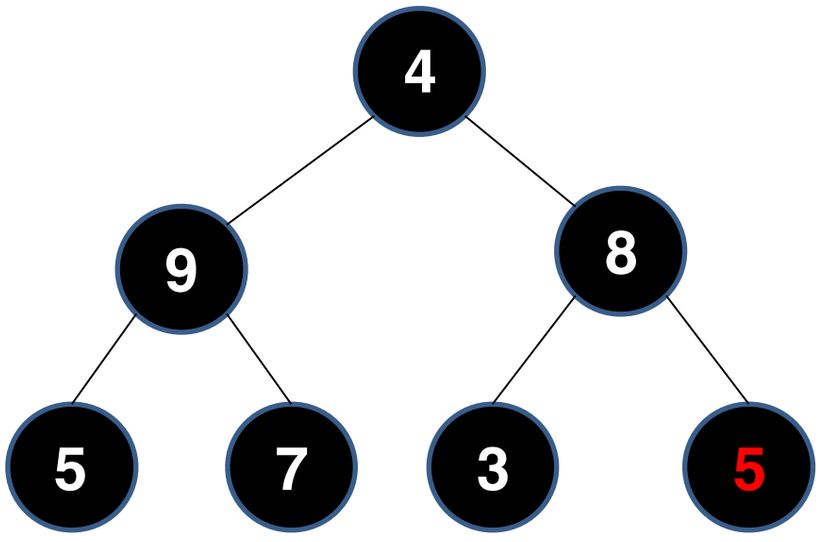


(1)初始完全二叉树，从3开始建堆。

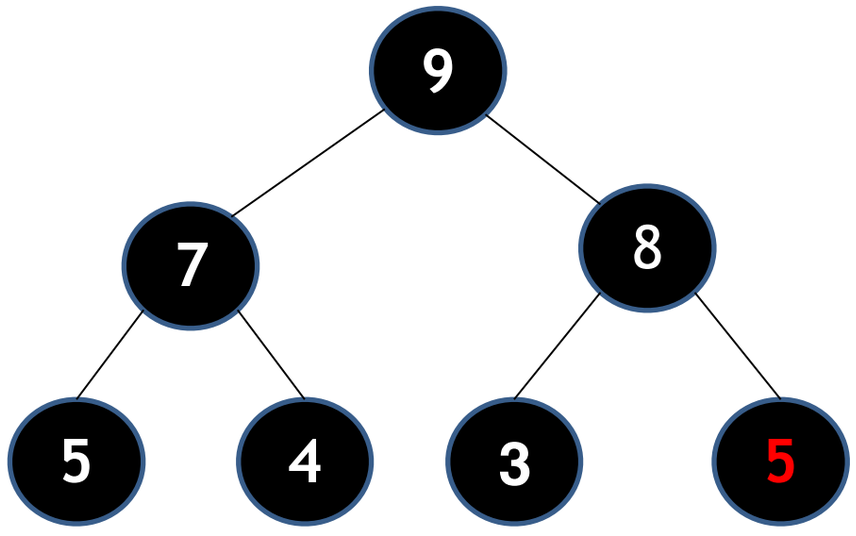


(2)初始完全二叉树，从7开始建堆。

大根堆初始建堆示例



(3)初始完全二叉树，从4开始建堆。



(4)得到初始堆。

堆排序的基本过程：选择和堆重建

(1) 交换堆顶元素与堆中的最后一个元素并重建堆：

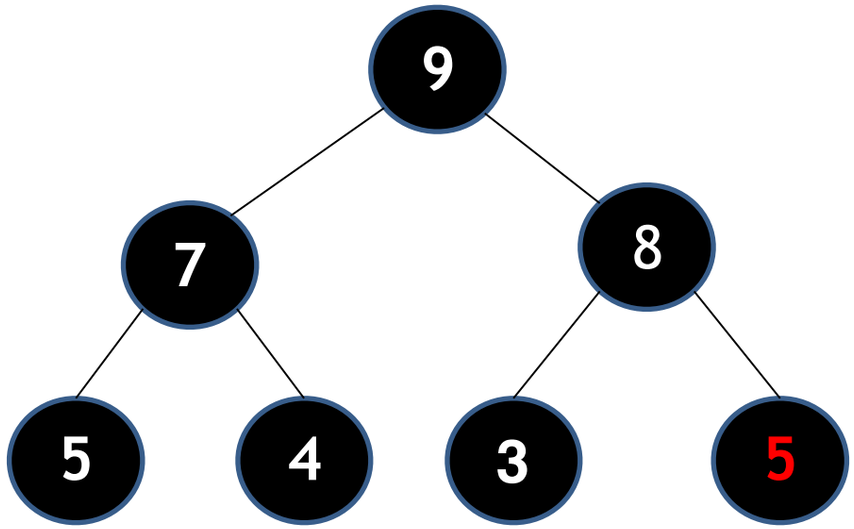
(a) 已排序的序列增加一个元素；

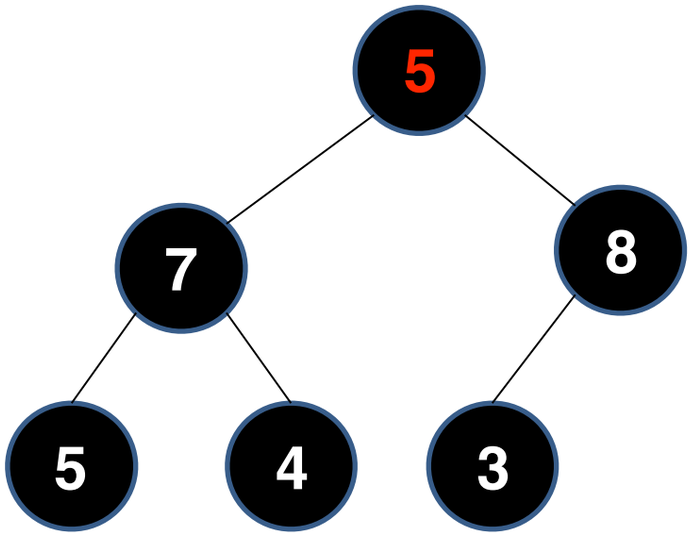
(b) 而且把堆的最后元素放在两个堆上；

(c) 重建堆：通过一次筛选，使数组前面一段重新成为一个新堆（堆里的元素减少了一个）。

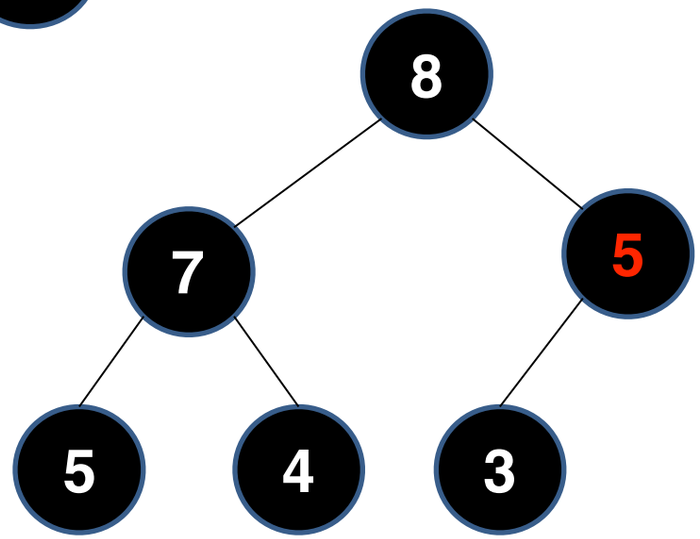
(2) 反复做上一步，直至整个堆里只剩下一个元素时（它必然是所有元素里最小的）排序完成。

大根堆排序示例



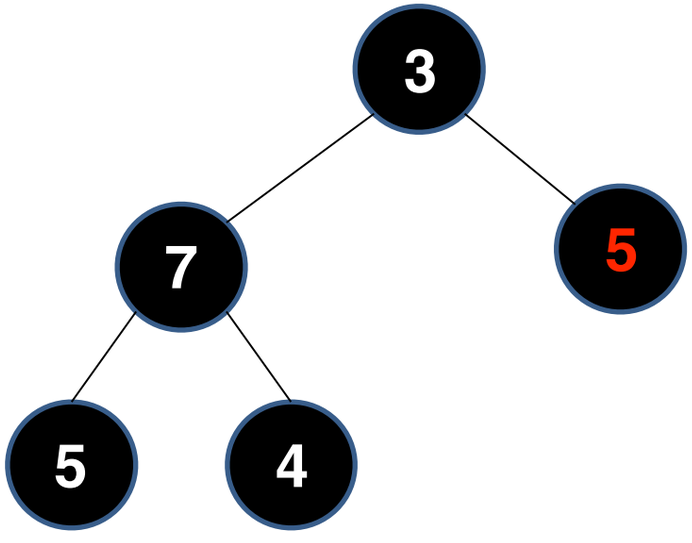


(4) 得到初始堆。

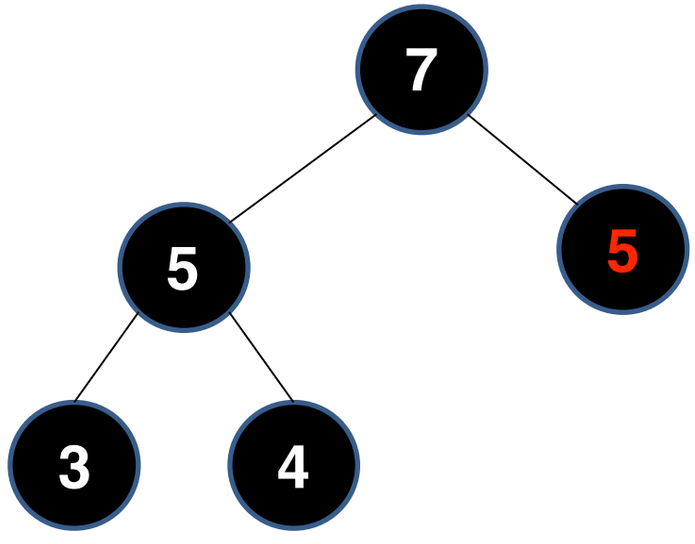


(5) 9与5互换。

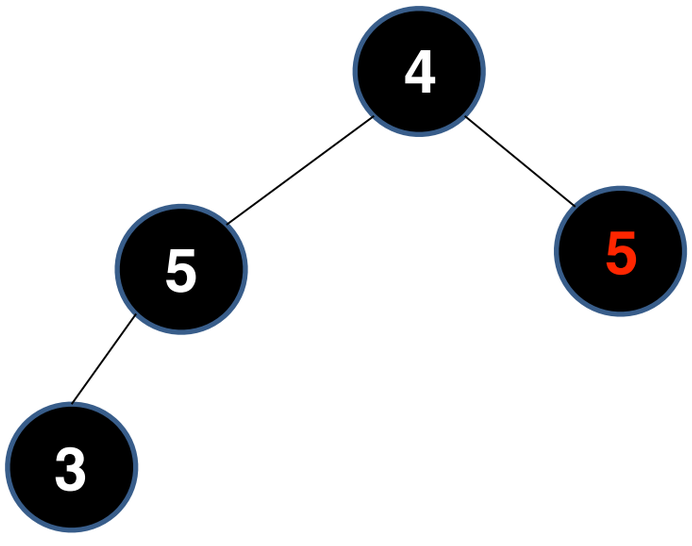
(6) 重新建堆。



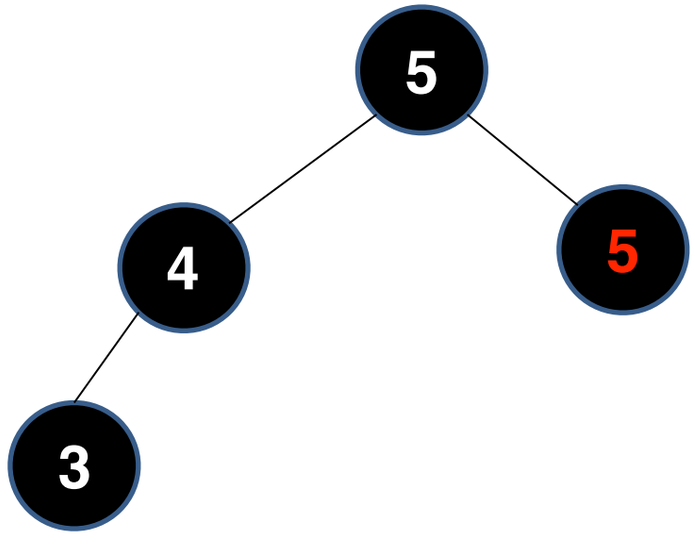
(7) 8与3互换。



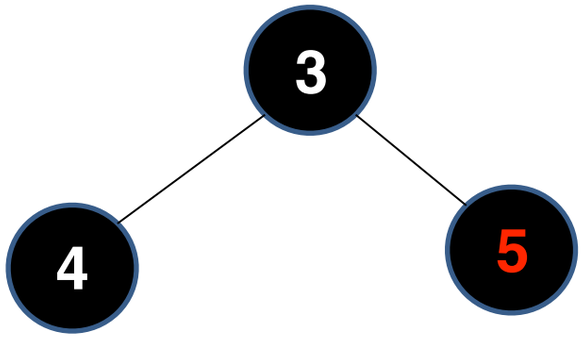
(8) 重新建堆。



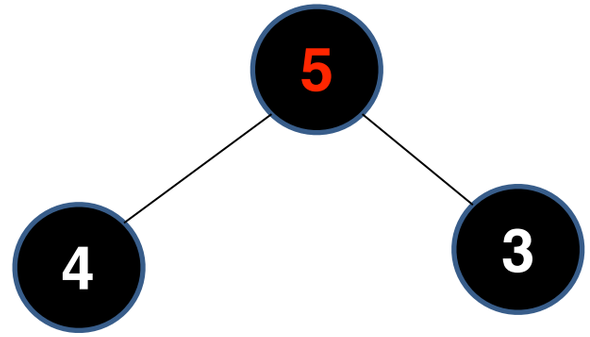
(9) 7与4互换。



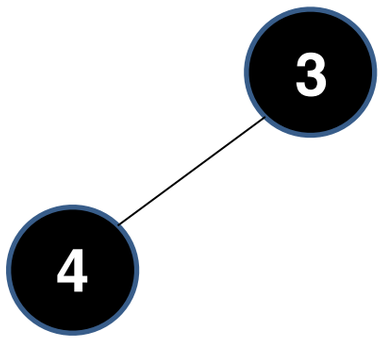
(10) 重新建堆。



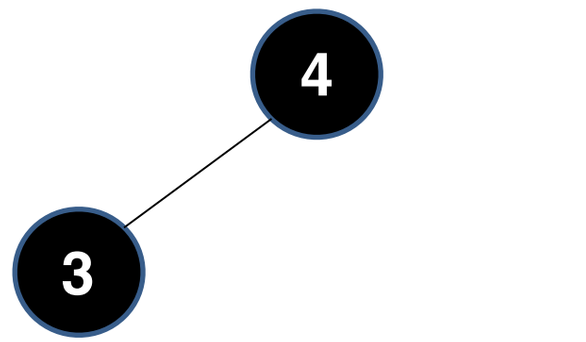
(11) 5与3互换。



(12) 重新建堆。



(13) 5与3互换。



(14) 重新建堆。

(15) 4和3互换，排序完成。

堆排序算法

```
void  heapSort(SortObject * pvector) {
  int i, n;
  RecordNode temp;
  n=pvector->n;
  for(i=n/2-1; i>=0; i--) sift(pvector,n,i);      /* 建立初始堆 */
  for(i=n-1; i>0; i--) {                          /* 进行n-1趟堆排序 */
   temp=pvector->record[0];       /* 当前堆顶和堆中最后记录互换 */
   pvector->record[0]=pvector->record[i];
   pvector->record[i]=temp;
   sift(pvector,i,0);  /*重新调整建堆，注意i在控制调整范围中的作用*/
   }
}
```

堆排序算法

```
void  sift(SortObject * pvector, int size , int p)  {
 RecordNode temp=pvector->record[p];
 int child= 2* p +1;
 while(child<size)  {
    if((child<size-1)&&(pvector->record[child].key < pvector->record[child+1].key))
      child++;                                          /*选择比较大的子结点 */
    if(temp.key < pvector->record[child].key)  {
      pvector->record[p]=pvector->record[child];        /*将大的子结点上移 */
      p=child;  child= 2*p +1;
      }
    else  break;                                        /* 调整结束 */
    }
  pvector->record[p]=temp;                              /* 将temp放入正确位置 */
}
```

算法分析

- 堆排序总的时间复杂度$T(n) = O(n*log_2n)$：
  - 初始建堆比较次数为：$O(n)$；
  - 排序中比较次数：$O(n*log_2n)$ (完全树高度为$O(log_2n)$)；
  - 移动次数小于比较次数；
  - 最坏情况下的时间复杂度：$O(n*log_2n)$；
- 仅需一个记录大小的辅助空间，开销为$O(1)$。
- 是一种高效排序算法，适用于n值较大的情况。
- 堆排序不稳定：
  - 如不同分支中有排序码相同的结点，算法不能保证它们在最终排序序列里的位置关系。

交换排序

- 基本思想:
  - 每发现某两项次序颠倒（逆序），则交换它们;
  - 重复这一过程，直到不需要交换为止。

- 不同确定逆序的方式和交换方式:
  - 起泡排序
  - 快速排序

Sir C.A.R. Hoare

(1934-)

起泡排序

- 基本思想：
  - 顺序比较相邻记录，发现逆序就进行交换；
  - 通过不断比较和交换，最终得到一个排序序列。
- 容易证明：
  - 只要每对相邻记录的顺序正确（前一记录不大于后一记录，假定要求按上升序排序），则整个序列为一个排序序列。
  - 这也是通过局部性质得到全局性质的一个实例。

起泡排序算法

- 设待排序记录顺序存在$R_0, R_1, R_2, \ldots, R_{n-1}$中；
  - 顺序比较$(R_0,R_1), (R_1,R_2), \ldots, (R_{n-2},R_{n-1})$，遇到相邻记录的顺序颠倒则交换它们。一遍比较和交换的结果将保证最大记录移到R_{n-1}，称为一次起泡。
  - 再对存放在$R_0, R_1, R_2, \ldots, R_{n-2}$中$n-1$个记录作同样处理，结果将保证次大记录移到 R_{n-2}。
  - ……
  - $n-1$次起泡一定能完成排序。
  - 可以增加一个标志noswap，用于记录本次起泡是否进行了交换，若无交换则表示排序已经完。

起泡排序示例

初始序列为　49,38,65,97,76,13,27,49'

i=1:　38　49　65　76　13　27　49'　[97]

i=2:　38　49　65　13　27　49'　[76　97]

i=3:　38　49　13　27　49'　[65　76　97]

i=4:　38　13　27　49　[49'　65　76　97]

i=5:　13　27　38　[49　49'　65　76　97]

i=6:　13　27　[38　49　49'　65　76　97]

i=7:　13　[27　38　49　49'　65　76　97]

起泡排序算法

```
void  bubbleSort(SortObject * pvector) {
    int i, j, noswap;
    RecordNode temp, *data = pvector->record;
    for(i = 0; i < pvector->n-1; i++) {                    /* 做n-1次起泡 */
        noswap = TRUE;                                     /* 置交换标志 */
        for (j = 0; j < pvector->n-i-1; j++)               /* 从前向后扫描 */
            if (data[j+1].key < data[j].key) {             /* 交换记录 */
                temp = data[j];
                data[j] = data[j+1];
                data[j+1] = temp;
                noswap = FALSE;
            }
        if ( noswap ) break;                               /* 一遍起泡未发生交换，算法结束 */
    }
}
```

起泡排序算法分析

- 时间复杂度为$T(n) = O(n^2)$：
  - 最坏时间复杂度$O(n^2)$；
  - 平均时间复杂度$O(n^2)$；
  - 最好情况时间复杂度$O(n)$；
    - 起泡排序算法具有适应性。
- 起泡排序算法辅助空间为$S(n) = O(1)$。
- 起泡排序算法是稳定的。
- 同直接插入排序算法实验相比，起泡排序需要相对复杂的程序，而且花费的时间更高。

快速排序

- 起泡排序中总的比较和移动次数较多：
  - 相邻两个记录比较和交换，每次交换只能上移或下移一个位置。
- 快速排序也称为分区交换排序，其基本思想为：
  - 设法把待排序序列按某种标准分为大小两组；
    - 通常选择第一个记录R_0为区分标准，把R_0移动到中间，所有大于该排序码的记录移到R_0的右边，所有小于该排序码的记录移到R_0的左边。
  - 而后可以递归地分别对两组记录采用同样方式排序；
  - 划分到每个子部分只包含一个记录，整个序列的排序完成。

快速排序算法一趟中各次交换示例

初始: [49 38 65 97 76 13 27 49]

i↑ j↑

j向左扫描: [□ 38 65 97 76 13 27 49]

i↑ j↑

第一次交换后: [27 38 65 97 76 13 □ 49]

i↑ j↑

i向右扫描: [27 38 65 97 76 13 □ 49]

i↑ j↑

第二次交换后: [27 38 □ 97 76 13 65 49]

i↑ j↑

快速排序各趟结果示例

初始: [49 38 65 97 76 13 27 49]

[27 38 13] 49 [76 97 65 49]

[13] 27 [38] 49 [76 97 65 49]

13 27 [38] 49 [76 97 65 49]

13 27 38 49 [76 97 65 49]

13 27 38 49 [49 65] 76 [97]

13 27 38 49 49 [65] 76 [97]

13 27 38 49 49 65 76 [97]

13 27 38 49 49 65 76 97

快速排序算法实现

- 需要在顺序表内部完成排序，使用尽可能少的辅助空间:

  (1) 最简单的划分方式是取序列中第一个记录，以它的排序码为标准，把排序码小的记录移到表的一边，排序码大的记录移到另一边；

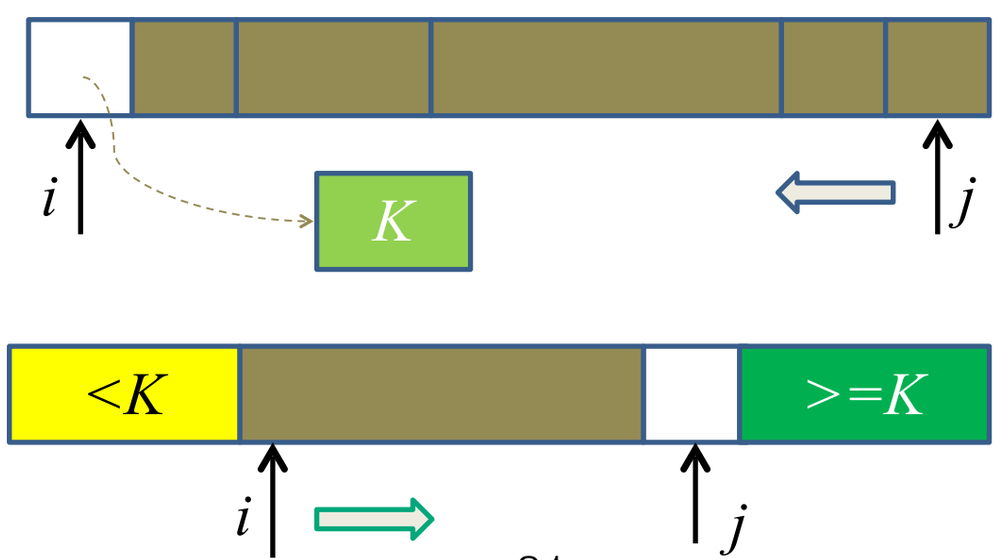
  - 显然这个标准被移动到了它在最终递增序列的位置；
  - 存在不同的选择标准和移动记录的方式，形成了快速排序的不同实现。

  (2) 一次划分完毕后，中间空位就是作为标准的记录的位置；

  (3) 而后对两边的记录序列用同样方式分别处理（递归）。

一遍快速排序的一种做法

- 设 i 和 j 初值分别是序列第一个和最后记录的位置；
- 取出第一个记录，设其排序码为 K(划分标准)；
- 从 j 所指位置起向前搜索，找到第一个排序码小于 K 的记录并将其存入前面空位；从 i 所指位置起向后搜索，找到第一个排序码大于 K 的记录并将其存入上一步留下的空位；
- 重复地交替进行上述两个动作直到 i 不小于 j 为止。



快速排序算法

```
void quickSort(SortObject * pvector, int l, int r) {
    int i, j;
    RecordNode temp, *data = pvector->record;
    if (l >= r) return;                                  /* 只有一个记录或无记录，则无须排序 */
    i = l ;  j = r ; temp = data[i];
    while(i != j) {                                      /* 找R_l的最终位置 */
       while( i < j && data[j].key >= temp.key)
          j--;                                           /* 向左扫描找排序码小于temp.key的记录 */
       if (i < j) data[i++] = data[j];
       while( i < j && data[i].key <= temp.key)
          i++;                                           /* 向右扫描找排序码大于temp.key的记录 */
       if (i < j) data[j--] = data[i];
    }
    data[i] = temp;                                      /* 将 R_l 存入其最终位置 */
    quickSort(pvector, l, i-1);                          /* 递归处理左区间 */
    quickSort(pvector, i+1, r);                          /* 递归处理右区间 */
}
```

快速排序算法分析

- 快速排序的记录移动次数不大于比较次数，所以其最坏时间复杂度应为$O(n^2)$，最坏情况出现在待排序序列为有序时.
- 最好时间复杂度为$O(n*log_2n)$，如果每次划分能把序列分为长度差不多的两段，就可以得到 $O(n*log_2n)$。
- 为减少最坏情况的出现，可采用“三者取中”规则，每趟划分前，比较lst[l].key、lst[r].key和lst[(l+r)/2].key的大小，取中间的记录与lst[l]交换，用它的排序码划分。
- 快速排序的平均时间复杂度是$O(n*log_2n)$。
- 算法需要栈空间实现递归。栈大小取决于递归深度，最多不超过n。若每次先处理短的一半，递归深度将不超过log_2n，所以快速排序的辅助空间为$O(log_2n)$。
- 常见快速排序算法是不稳定的。

本讲重点

- 选择排序：直接选择排序、堆排序
- 交换排序：起泡排序、快速排序