阅读：14,088　　作者：解学武

简单选择排序、树形选择排序和堆排序算法及其C语言实现

上一节　　下一节

本节介绍三种选择排序算法，分别为：简单选择排序、树形选择排序和堆排序。

简单选择排序

该算法的实现思想为：对于具有 n 个记录的无序表遍历 n-1 次，第 i 次从无序表中第 i 个记录开始，找出后序关键字中最小的记录，然后放置在第 i 的位置上。

例如对无序表 {56，12，80，91，20} 采用简单选择排序算法进行排序，具体过程为：

- 第一次遍历时，从下标为 1 的位置即 56 开始，找出关键字值最小的记录 12，同下标为 0 的关键字 56 交换位置：

12	56	80	91	20

- 第二次遍历时，从下标为 2 的位置即 56 开始，找出最小值 20，同下标为 2 的关键字 56 互换位置：

12	20	80	91	56

- 第三次遍历时，从下标为 3 的位置即 80 开始，找出最小值 56，同下标为 3 的关键字 80 互换位置：

12	20	56	91	80

- 第四次遍历时，从下标为 4 的位置即 91 开始，找出最小是 80，同下标为 4 的关键字 91 互换位置：

12	20	56	80	91

- 到此简单选择排序算法完成，无序表变为有序表。

简单选择排序的实现代码为：

```
01.  #include <stdio.h>
02.  #include <stdlib.h>
03.  #define MAX 9
04.  //单个记录的结构体
05.  typedef struct {
06.      int key;
07.  }SqNote;
08.  //记录表的结构体
09.  typedef struct {
10.      SqNote r[MAX];
```

```
    int length;
}SqList;
//交换两个记录的位置
void swap(SqNote *a,SqNote *b){
    int key=a->key;
    a->key=b->key;
    b->key=key;
}
//查找表中关键字的最小值
int SelectMinKey(SqList *L,int i){
    int min=i;
    //从下标为 i+1 开始，一直遍历至最后一个关键字，找到最小值所在的位置
    while (i+1<L->length) {
        if (L->r[min].key>L->r[i+1].key) {
            min=i+1;
        }
        i++;
    }
    return min;
}
//简单选择排序算法实现函数
void SelectSort(SqList * L){
    for (int i=0; i<L->length; i++) {
        //查找第 i 的位置所要放置的最小值的位置
        int j=SelectMinKey(L,i);
        //如果 j 和 i 不相等，说明最小值不在下标为 i 的位置，需要交换
        if (i!=j) {
            swap(&(L->r[i]),&(L->r[j]));
        }
    }
}

int main() {
    SqList * L=(SqList*)malloc(sizeof(SqList));
    L->length=8;
    L->r[0].key=49;
    L->r[1].key=38;
    L->r[2].key=65;
    L->r[3].key=97;
    L->r[4].key=76;
    L->r[5].key=13;
    L->r[6].key=27;
    L->r[7].key=49;
    SelectSort(L);
    for (int i=0; i<L->length; i++) {
        printf("%d ",L->r[i].key);
    }
```

```
58.      return 0;
59.  }
```

运行结果：

```
13 27 38 49 49 65 76 97
```

树形选择排序

树形选择排序（又称“锦标赛排序”），是一种按照锦标赛的思想进行选择排序的方法，即所有记录采取两两分组，筛选出较小（较大）的值；然后从筛选出的较小（较大）值中再两两分组选出更小（更大）值，依次类推，直到最后选出一个最小（最大）值。同样可以采用此方式筛选出次小（次大）值等。

整个排序的过程，可以用一棵具有 n 个叶子结点的完全二叉树表示。例如对无序表 {49，38，65，97，76，13，27，49} 采用树形选择的方式排序，过程如下：

- 首先将无序表中的记录采用两两分组，筛选出各组中的较小值（如图 1 中的（a）过程）；然后将筛选出的较小值两两分组，筛选出更小的值，以此类推（如图 1 中的（b）（c）过程），最终整棵树的根结点中的关键字即为最小关键字：

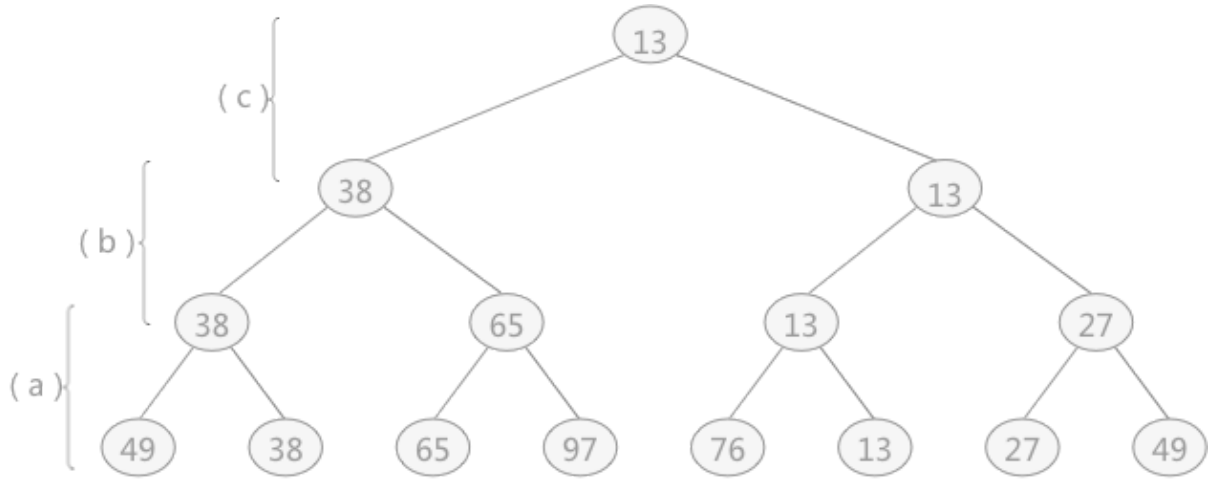


图 1 树形选择排序（一）

- 筛选出关键字 13 之后，继续重复此方式找到剩余记录中的最小值，此时由于关键字 13 已经筛选完成，需要将关键字 13 改为“最大值”，继续重复此过程，如图 2 所示：

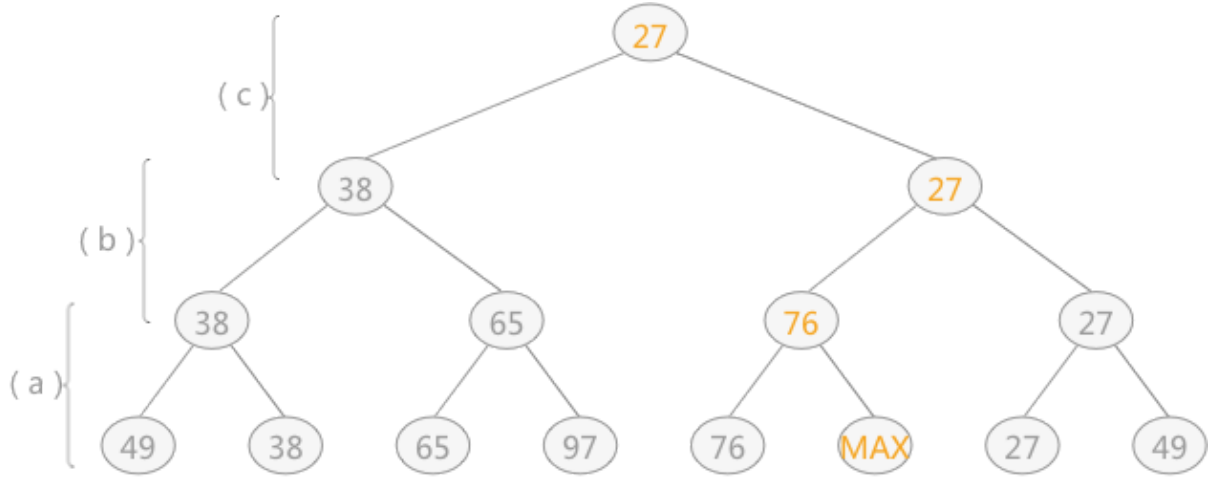


图 2 树形选择排序（二）

通过不断地重复此过程，可依次筛选出从小到大的所有关键字。该算法的时间复杂度为 O(nlogn) ，同简单选择排序相比，该算法减少了不同记录之间的比较次数，但是程序运行所需要的空间较多。

堆排序

在学习堆排序之前，首先需要了解堆的含义：在含有 n 个元素的序列中，如果序列中的元素满足下面其中一种关系时，此序列可以称之为堆。

- $k_i \le k_{2i}$ 且 $k_i \le k_{2i+1}$（在 n 个记录的范围内，第 i 个关键字的值小于第 2*i 个关键字，同时也小于第 2*i+1 个关键字）
- $k_i \ge k_{2i}$ 且 $k_i \ge k_{2i+1}$（在 n 个记录的范围内，第 i 个关键字的值大于第 2*i 个关键字，同时也大于第 2*i+1 个关键字）

对于堆的定义也可以使用完全二叉树来解释，因为在完全二叉树中第 i 个结点的左孩子恰好是第 2i 个结点，右孩子恰好是 2i+1 个结点。如果该序列可以被称为堆，则使用该序列构建的完全二叉树中，每个根结点的值都必须不小于（或者不大于）左右孩子结点的值。

以无序表 {49，38，65，97，76，13，27，49} 来讲，其对应的堆用完全二叉树来表示为：

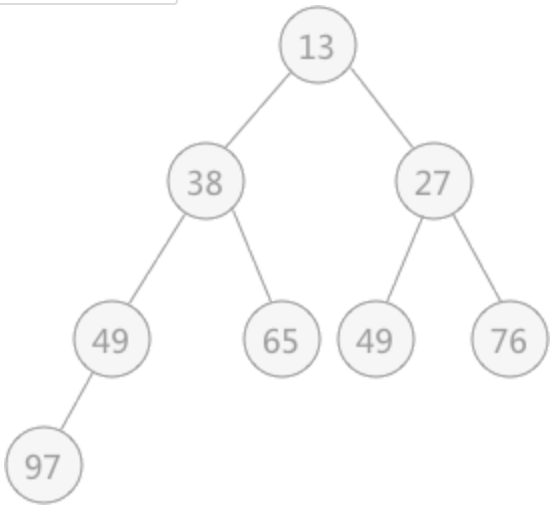


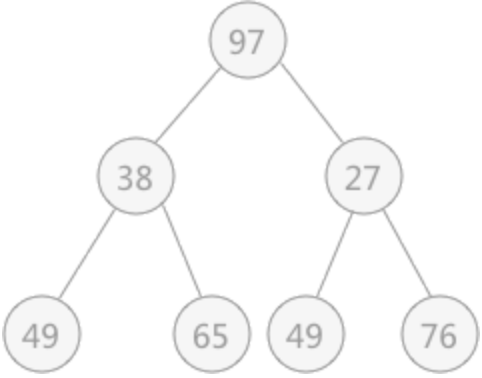
图 3 无序表对应的堆

> 提示：堆用完全二叉树表示时，其表示方法不唯一，但是可以确定的是树的根结点要么是无序表中的最小值，要么是最大值。

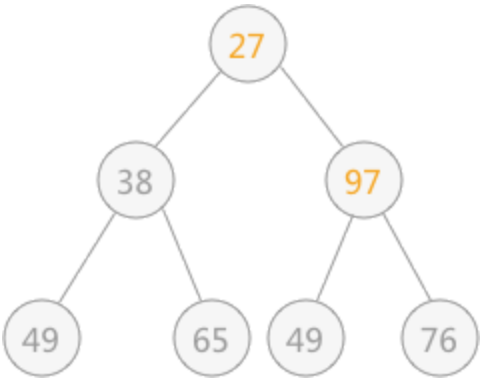
通过将无序表转化为堆，可以直接找到表中最大值或者最小值，然后将其提取出来，令剩余的记录再重建一个堆，取出次大值或者次小值，如此反复执行就可以得到一个有序序列，此过程为堆排序。

堆排序过程的代码实现需要解决两个问题：
1. 如何将得到的无序序列转化为一个堆？
2. 在输出堆顶元素之后（完全二叉树的树根结点），如何调整剩余元素构建一个新的堆？

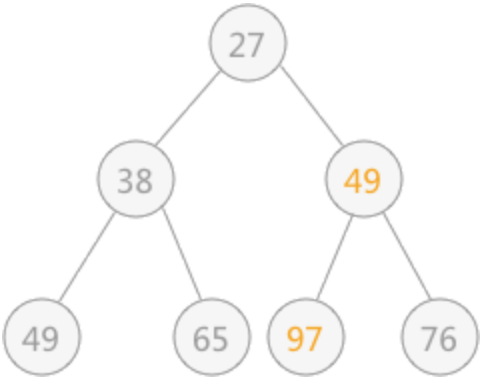
首先先解决第 2 个问题。图 3 所示为一个完全二叉树，若去除堆顶元素，即删除二叉树的树根结点，此时用二叉树中最后一个结点 97 代替，如下图所示：



此时由于结点 97 比左右孩子结点的值都大，破坏了堆的结构，所以需要进行调整：首先以 堆顶元素 97 同左右子树比较，同值最小的结点交换位置，即 27 和 97 交换位置：

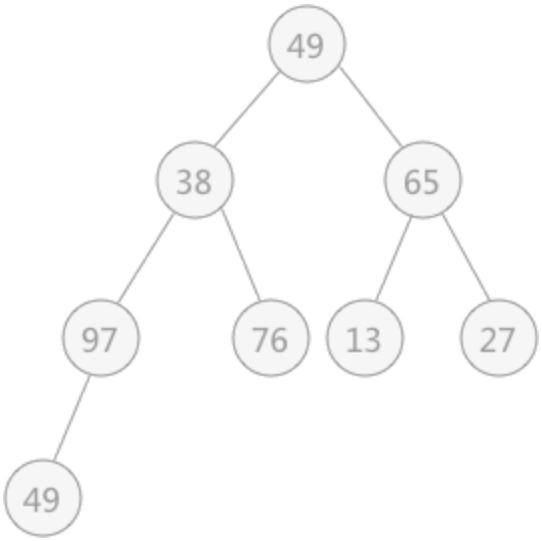


由于替代之后破坏了根结点右子树的堆结构，所以需要进行和上述一样的调整，即令 97 同 49 进行交换位置：



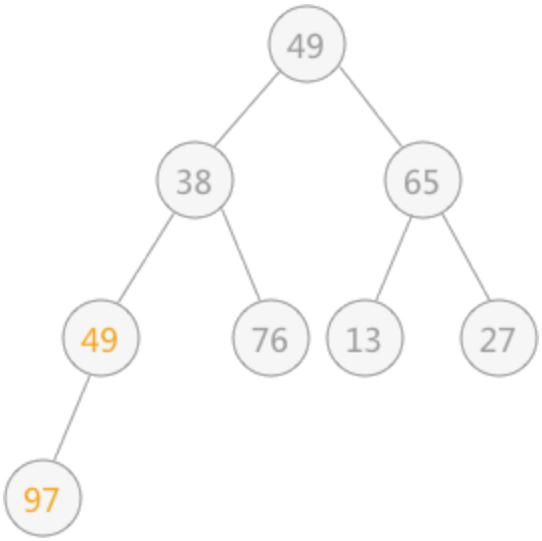
通过上述的调整，之前被破坏的堆结构又重新建立。从根结点到叶子结点的整个调整的过程，被称为“筛选”。

解决第一个问题使用的就是不断筛选的过程，如下图所示，无序表 {49，38，65，97，76，13，27，49} 初步建立的完全二叉树，如下图所示：

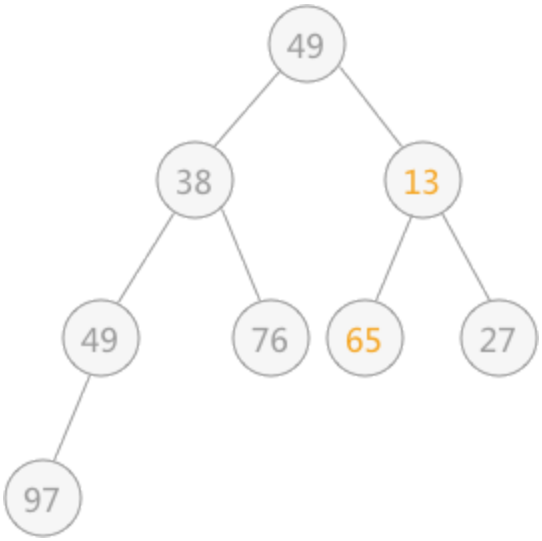


在对上图做筛选工作时，规律是从底层结点开始，一直筛选到根结点。对于具有 n 个结点的完全二叉树，筛选工作开始的结点为第 ⌊n/2⌋个结点（此结点后序都是叶子结点，无需筛选）。

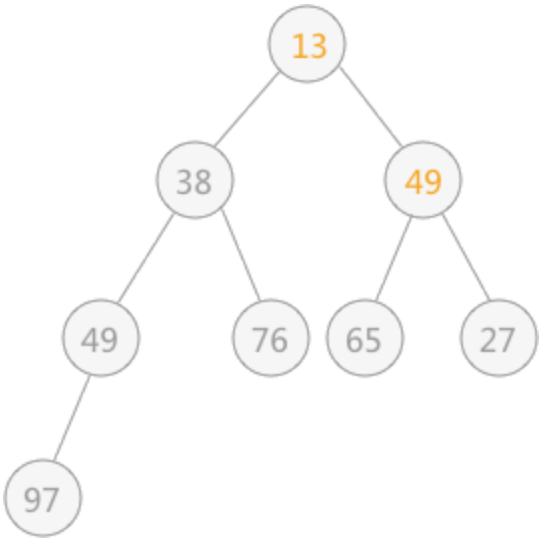
所以，对于有 9 个结点的完全二叉树，筛选工作从第 4 个结点 97 开始，由于 97 > 49 ,所以需要相互交换，交换后如下图所示：



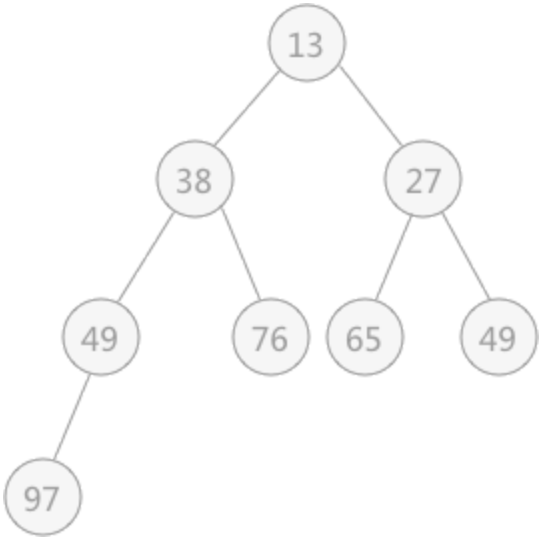
然后再筛选第 3 个结点 65 ， 由于 65 比左右孩子结点都大，则选择一个最小的同 65 进行交换，交换后的结果为：



然后筛选第 2 个结点，由于其符合要求，所以不用筛选；最后筛选根结点 49 ，同 13 进行交换，交换后的结果为：



交换后，发现破坏了其右子树堆的结构，所以还需要调整，最终调整后的结果为：



所以实现堆排序的完整代码为：

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 9
//单个记录的结构体
typedef struct {
    int key;
}SqNote;
//记录表的结构体
typedef struct {
    SqNote r[MAX];
    int length;
}SqList;
//将以 r[s]为根结点的子树构成堆，堆中每个根结点的值都比其孩子结点的值大
void HeapAdjust(SqList * H,int s,int m){
    SqNote rc=H->r[s];//先对操作位置上的结点数据进行保存，放置后序移动元素丢失。
    //对于第 s 个结点，筛选一直到叶子结点结束
    for (int j=2*s; j<=m; j*=2) {
        //找到值最大的孩子结点
        if (j+1<m && (H->r[j].key<H->r[j+1].key)) {
            j++;
        }
        //如果当前结点比最大的孩子结点的值还大，则不需要对此结点进行筛选，直接略过
        if (!(rc.key<H->r[j].key)) {
            break;
        }
        //如果当前结点的值比孩子结点中最大的值小，则将最大的值移至该结点，由于 rc 记录着该结点的值，所以
        H->r[s]=H->r[j];
        s=j;//s相当于指针的作用，指向其孩子结点，继续进行筛选
    }
    H->r[s]=rc;//最终需将rc的值添加到正确的位置
}
//交换两个记录的位置
void swap(SqNote *a,SqNote *b){
    int key=a->key;
    a->key=b->key;
    b->key=key;
}
void HeapSort(SqList *H){
    //构建堆的过程
    for (int i=H->length/2; i>0; i--) {
        //对于有孩子结点的根结点进行筛选
        HeapAdjust(H, i, H->length);
    }
    //通过不断地筛选出最大值，同时不断地进行筛选剩余元素
    for (int i=H->length; i>1; i--) {
        //交换过程，即为将选出的最大值进行保存大表的最后，同时用最后位置上的元素进行替换，为下一次筛选做
```

```
47.          swap(&(H->r[1]), &(H->r[i]));
48.          //进行筛选次最大值的工作
49.          HeapAdjust(H, 1, i-1);
50.      }
51.  }
52.
53.  int main() {
54.      SqList * L=(SqList*)malloc(sizeof(SqList));
55.      L->length=8;
56.      L->r[1].key=49;
57.      L->r[2].key=38;
58.      L->r[3].key=65;
59.      L->r[4].key=97;
60.      L->r[5].key=76;
61.      L->r[6].key=13;
62.      L->r[7].key=27;
63.      L->r[8].key=49;
64.      HeapSort(L);
65.      for (int i=1; i<=L->length; i++) {
66.          printf("%d ",L->r[i].key);
67.      }
68.      return 0;
69.  }
```

运行结果为：

```
13 27 38 49 49 65 76 97
```

> 提示:代码中为了体现构建堆和输出堆顶元素后重建堆的过程，堆在构建过程中，采用的是堆的第二种关系，即父亲结点的值比孩子结点的值大；重建堆的过程也是如此。

堆排序在最坏的情况下，其时间复杂度仍为 `O(nlogn)` 。这是相对于快速排序的优点所在。同时堆排序相对于树形选择排序，其只需要一个用于记录交换（rc）的辅助存储空间，比树形选择排序的运行空间更小。

总结

本节介绍了三种选择排序：简单选择排序、树形选择排序和堆排序。树形选择排序的产生是考虑到为了减少简单选择排序过程中做比较操作的次数；堆排序的产生是考虑到树形选择排序在运行时申请的辅助存储空间多。要根据特定地情况，选择合适的选择排序算法。

‹ 上一节　　下一节 ›