§10. 内部排序

10.1. 概述

10.1.1. 排序的含义及定义

含义：将数据元素的任意序列 => 有序序列

定义：n个记录 $\{R_1, R_2, \ldots, R_n\}$

每个含关键字 $\{K_1, K_2, \ldots, K_n\}$

若存在 1..n 的一种排列 $P_1, P_2, \ldots, P_n$，使 $K_{p1}, K_{p2}, \ldots, K_{pn}$有序，则使 $\{R_1, R_2, \ldots, R_n\} => \{R_{p1}, R_{p2}, \ldots, R_{pn}\}$的过程称为排序

★ 若 $K_{p1} \leqslant K_{p2} \leqslant \ldots \leqslant K_{pn}$，则称为升序排序

若 $K_{p1} \geqslant K_{p2} \geqslant \ldots \geqslant K_{pn}$，则称为降序排序

★ 若K_i为主关键字(唯一)，则排序结果唯一

若K_i为次关键字(不唯一),假设$K_i=K_j$ (i<>j)，如果排序前R_i在R_j前，排序后仍维持R_i在R_j前，则称排序方法是稳定的，否则称排序方法不稳定

- 稳定/不稳定的方法都是正确的

★ 若K_i由记录的多个字段组成，则称为多关键字排序,

若K_i由记录的一个字段组成，则称为单关键字排序

- 对于多关键字，第i个关键字只有在第i-1个关键字相等的情况下才局部有序，除第1关键字外，均整体无序

§10. 内部排序

10.1. 概述

10.1.1. 排序的含义及定义

10.1.2. 排序的分类

★ 按数据位置分

外部排序：部分记录在内存中，需进行内外存交换

内部排序：所有记录均在内存中

★ 按排序方法分

基于插入的排序方法

基于交换的排序方法

基于选择的排序方法

基于归并的排序方法

基于分配的排序方法

★ 按工作量分

简单的排序方法 $O(n^2)$

先进的排序方法 $O(n*\log n)$

基数排序 $O(d*n)$

§10. 内部排序

10.1. 概述

10.1.1. 排序的含义及定义

10.1.2. 排序的分类

10.1.3. 排序数据的存储形式

顺序表：一维数组形式，逻辑相邻且物理相邻，元素可随机访问，排序时需要移动/交换元素

链表：链式结构，逻辑相邻但物理不相邻，元素不能随机访问，排序时只需要修改指针

- 不需要插入/删除，一般采用静态链表即可

线性表+索引：元素线性存储，附设辅助表，记录关键字的值及指向记录的指针，先对辅助表进行排序，结束后再调整物理表

★ 本章讨论内排序，且基于顺序表，关键字为整数若使用其它类型，需调整某些细节（赋值、比较的方法等）

§10. 内部排序

10.1. 概述

10.1.3. 排序数据的存储形式

★ 排序元素的形式定义(P.264)

```
#define  MAXSIZE  20    //按需修改
typedef  int  KeyType; //按需修改

typedef struct {
    KeyType  key;
    InfoType otherinfo;
} RecordType; //元素类型

typedef struct {
    RecordType r[MAXSIZE+1];//从[1]开始,[0]暂闲置
    int length;
} sqlist;  //顺序表
```

10.1.4. 排序算法的评价标准

★ 排序过程中元素的比较次数

★ 排序过程中元素的移动次数

★ 排序过程中所需要的辅助存储空间

§10. 内部排序

10.2. 插入排序

10.2.1. 直接插入排序

★ 基本方法

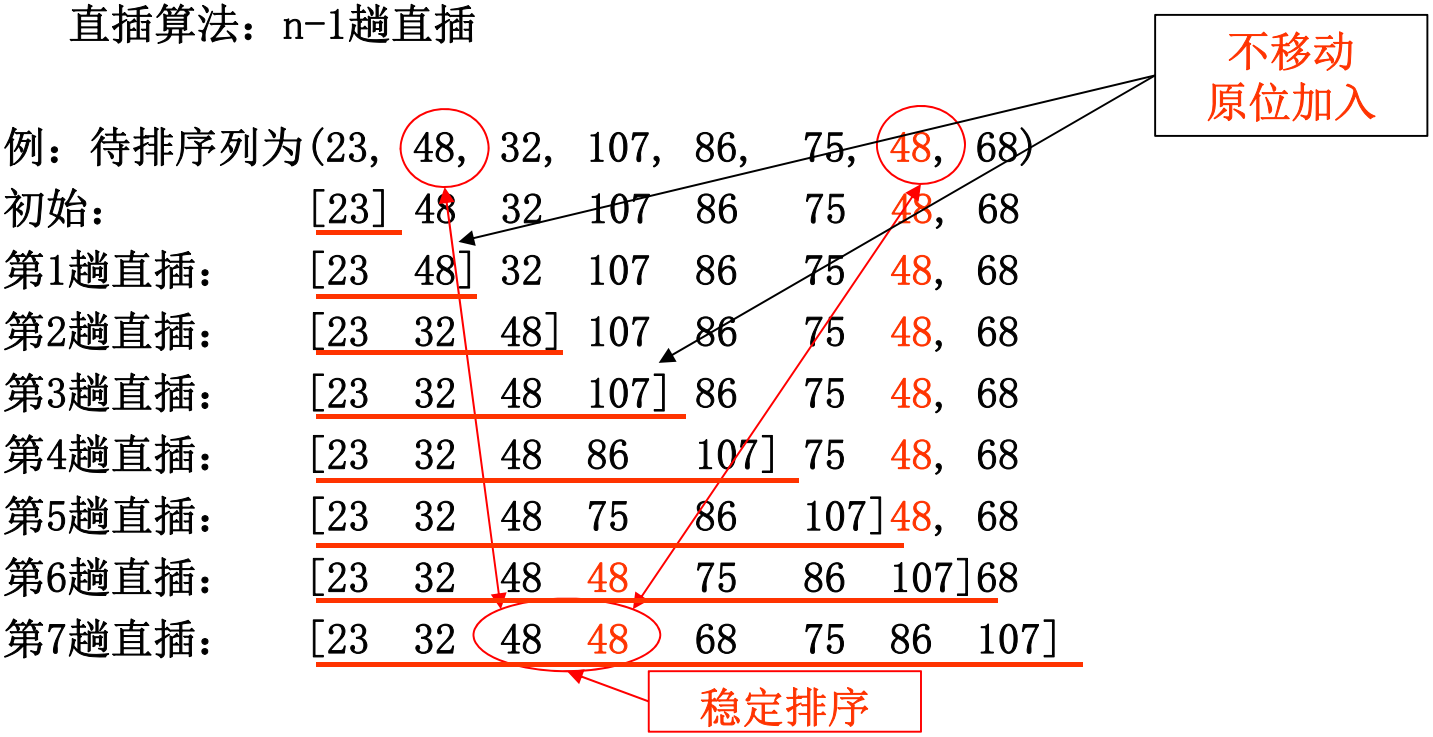
每次将一个新元素插入已经有序的表中，形成扩大后的新表并保持新表仍有序，重复到所有元素均有序

- 初始认为仅第1个元素有序(即有序表长度为1)
- 有序表表长每次增1

★ 排序过程

一趟直插：寻找插入位置，向后移动元素并插入

直插算法：n-1趟直插



★ 算法实现(P.265 算法10.1)

```
void InsertSort(SqList &L)
{
    int i, j;
    for(i=2; i<=L.length; i++) {  //循环n-1趟
        if (LT(L.r[i].key, L.r[i-1].key)) {
            L.r[0] = L.r[i];  //设监视哨，不用判下标越界
            L.r[i] = L.r[i-1];
            for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)
                L.r[j+1] = L.r[j];
            L.r[j+1] = L.r[0]; //r[i]已被覆盖，只能r[0]
            }          若if不成立，则表示r[i]原位加入，什么也不用做
        }
}
```

上面算法中红色框中的内容替换为下面的代码

```
for(j=i-1; LT(L.r[0].key, L.r[j].key); j--)
    L.r[j+1] = L.r[j];
```

问：行不行？那种更好？

§10. 内部排序

10.2. 插入排序

10.2.1. 直接插入排序

★ 效率分析　　$O(n^2)$

空间：一个辅助空间　R[0]

时间：一趟直插：最好：1次比较，0次移动

最坏：i次比较，i+1次移动(含[0])

平均：(最好+最坏)/2

n-1趟直插：最好：n-1次比较，0次移动

最坏：$\sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2}$　比较

$\sum_{i=2}^{n} (i+1) = \frac{(n+4)(n-1)}{2}$　移动

平均：(最好+最坏)/2

§10. 内部排序

10.2. 插入排序

10.2.2. 折半插入排序

★ 基本方法

在有序表中查找插入位置时不采用从后向前依次比较并同时移动的方法，而是采用先折半查找插入位置，再移动并插入的方法

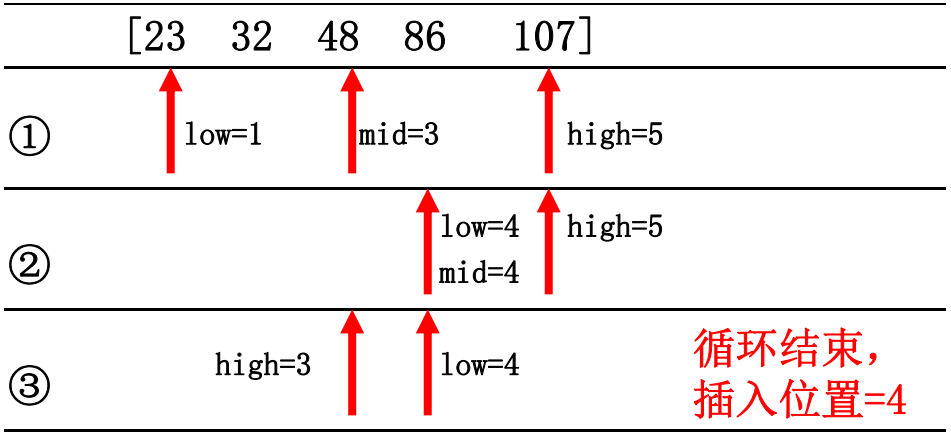
§10. 内部排序

10.2. 插入排序

10.2.2. 折半插入排序

★ 基本方法

例1：有序序列为(23, 32, 48, 86, 107)，要插入75



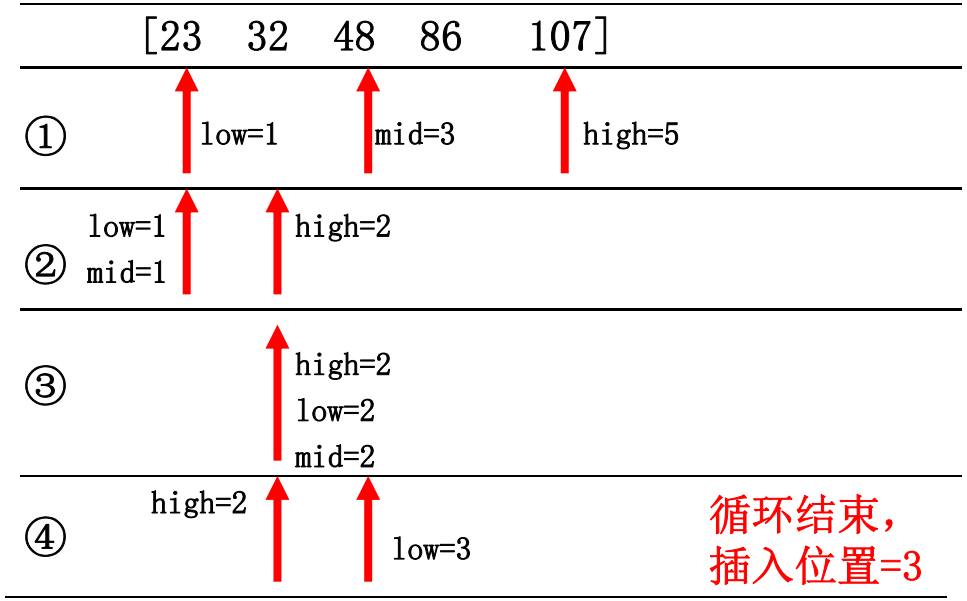
§10. 内部排序

10.2. 插入排序

10.2.2. 折半插入排序

★ 基本方法

例2：有序序列为(23, 32, 48, 86, 107)，要插入37



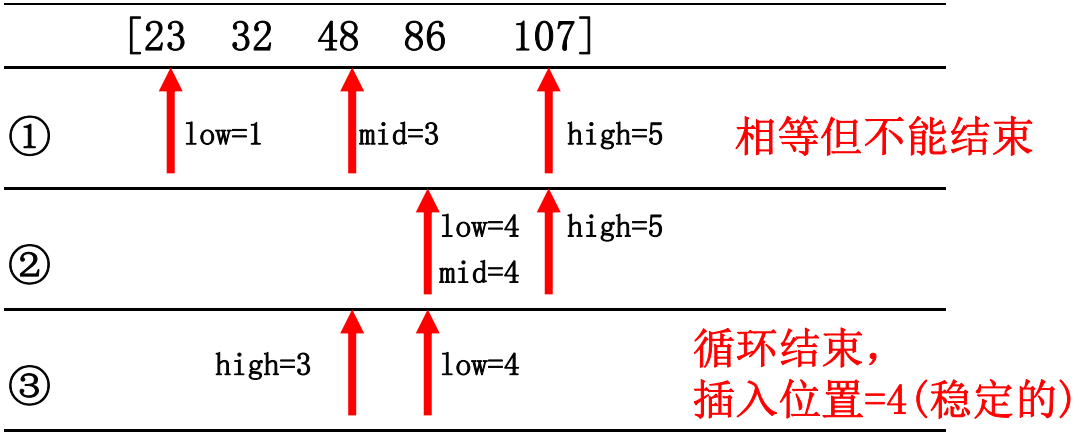
§10. 内部排序

10.2. 插入排序

10.2.2. 折半插入排序

★ 基本方法

例3：有序序列为(23, 32, 48, 86, 107)，要插入48



★ 算法实现(P.267 算法10.2)

```
void BInsertSort(SqList &L)
{
    for(i=2; i<=L.length; i++) {
        L.r[0] = L.r[i];  //仅暂存，不起监视哨的作用
        low  = 1;
        high = i-1;
        while(low<=high) {
            m = (low+high)/2; //自动向下取整
            if (LT(L.r[0].key, L.r[m].key))
                high = m-1;
            else
                low = m+1;
            }
        //执行到此，找到插入位置：high+1 或 low
        /* 元素向后移动 */
        for(j=i-1; j>=high+1; j--)
            L.r[j+1] = L.r[j];

        L.r[high+1] = L.r[0]; //r[i]已被覆盖，只能r[0]
        } // end of for
}
```

相等也不能结束
与查找不同

无法像直接插入一样
边比较边移动，
必须先查找，再移动

§10. 内部排序

10.2. 插入排序

10.2.2. 折半插入排序

★ 效率分析

减少比较次数，移动次数不变，仍为$O(n^2)$

- 待排数据量比较大时适用
- 最好状态（正序）下效率比较直接插入低，总体效率高于直接插入

§10. 内部排序

10.2. 插入排序

10.2.3. 希尔排序

★ 基本方法

以一个逐渐缩小的增量序列(最后一个为1，且增量序列无1之外的公因子)，将元素分为若干组，对每组元素进行直接插入排序(整体表现为跳跃式)，使组中元素增多时，元素已基本有序，从而提高效率(因为直接插入在基本有序时效率高)

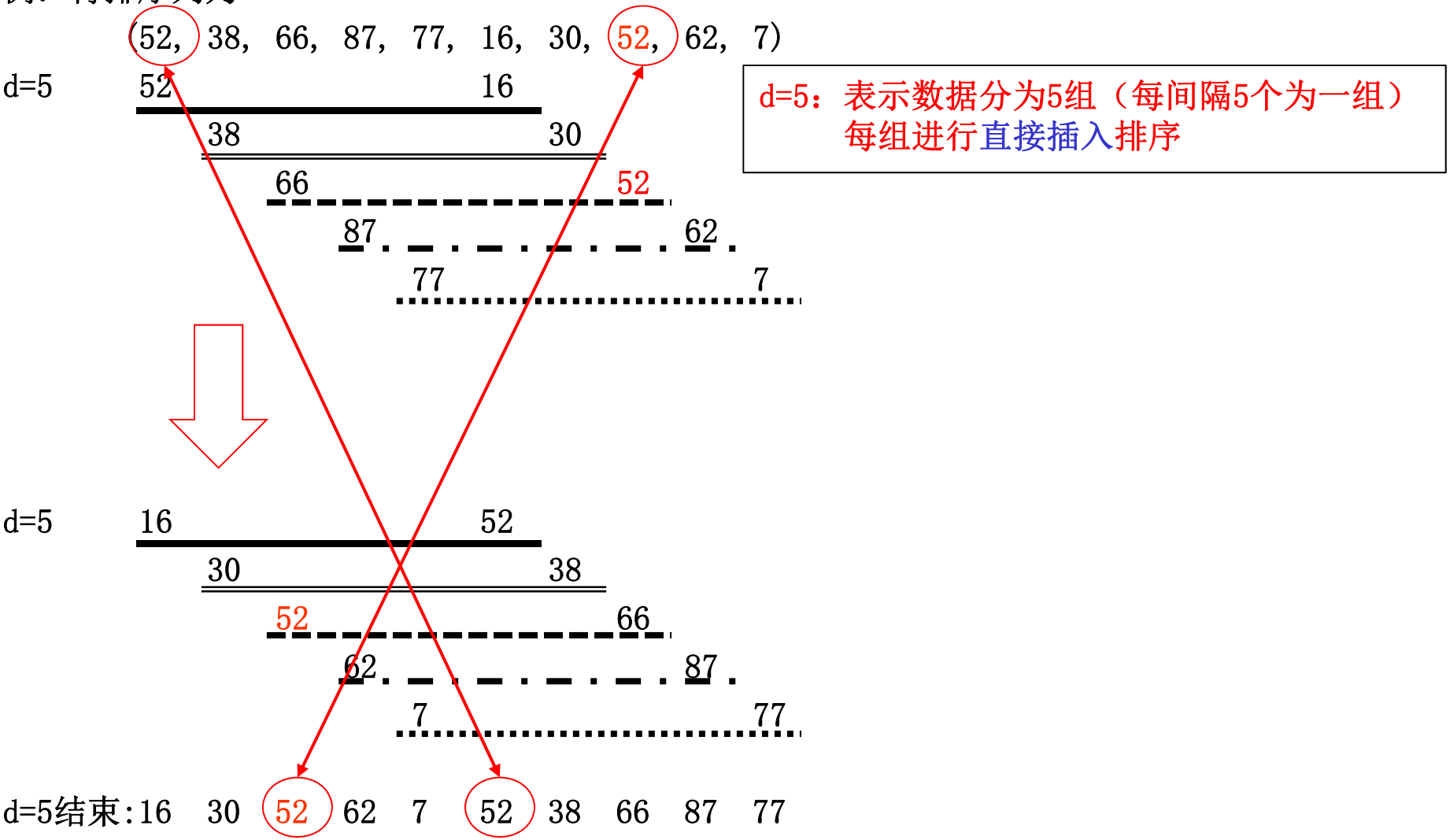
§10. 内部排序

10.2. 插入排序

10.2.3. 希尔排序

例：待排序列为

不稳定排序



d=5：表示数据分为5组（每间隔5个为一组）
每组进行直接插入排序

§10. 内部排序

10.2. 插入排序

10.2.3. 希尔排序

例：待排序列为

(52, 38, 66, 87, 77, 16, 30, 52, 62, 7)

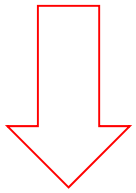
d=5结束:16 30 52 62 7 52 38 66 87 77

d=3 16 62 38 77

30 7 66

52 52 87

d=3：表示数据分为3组（每间隔3个为一组）
每组进行直接插入排序



d=3 16 38 62 77

7 30 66

52 52 87

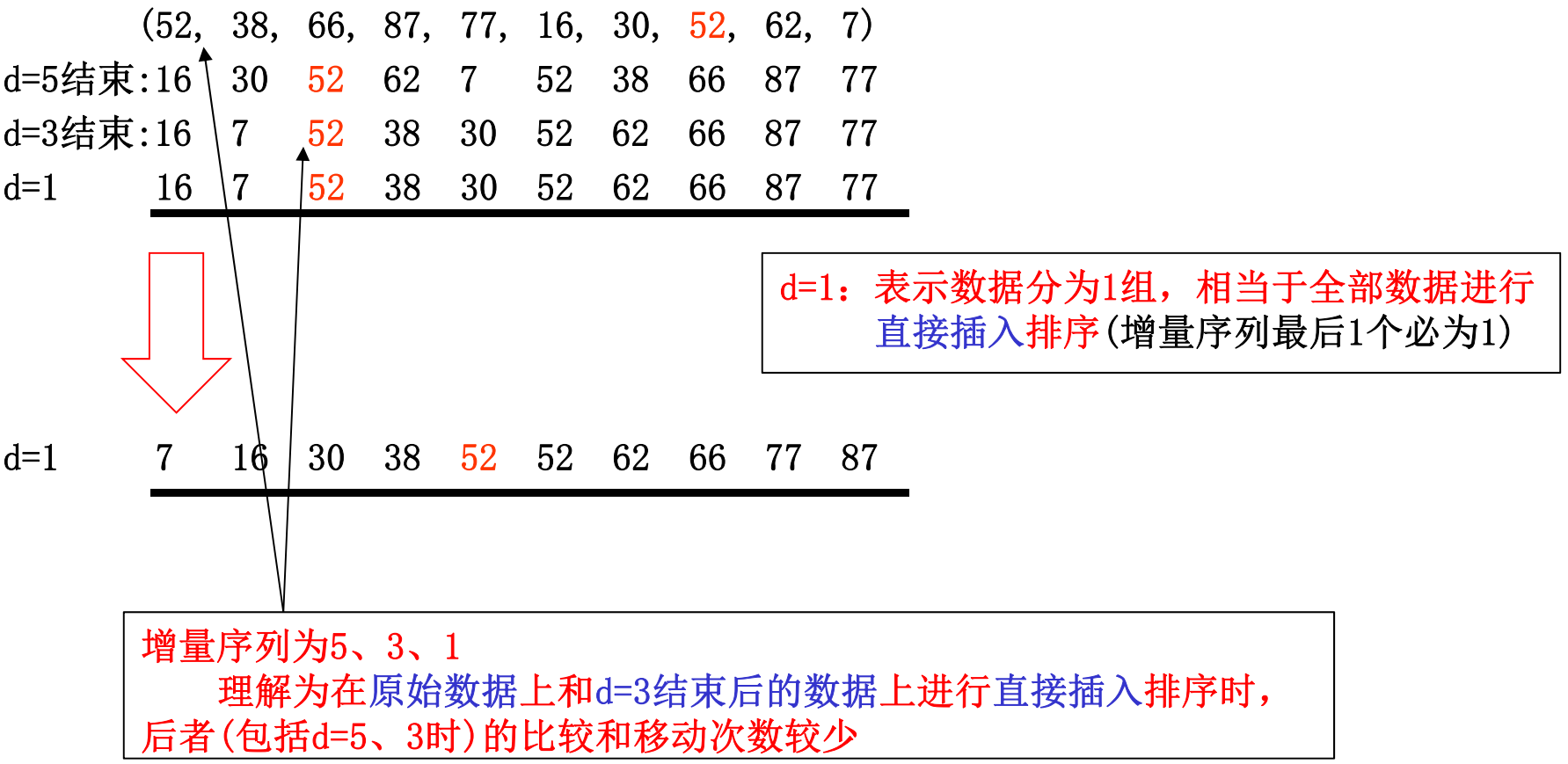
d=3结束:16 7 52 38 30 52 62 66 87 77

§10. 内部排序

10.2. 插入排序

10.2.3. 希尔排序

例：待排序列为



★ 算法实现(P.272 算法10.4)

```
void ShellInsert(SqList &L, int dk)
{
    for(i=dk+1; i<=length; i++)
        if (LT(L.r[i].key, L.r[i-dk].key)) {
            L.r[0] = L.r[i]; //暂存，非监视哨
            /* j=i-dk ：相当于每组的j=i-1
               j-=dk  ：相当于每组j--
               j>0：若j<=0则保证了插入位置已找到 */
            for(j=i-dk;j>0 && LT(L.r[0].key, L.r[j].key);j-=dk)
                L[j+dk] = L.r[j]; //记录后移
            L.r[j+dk] = L.r[0]; //插入指定位置
            }
}
```

i=dk+1：前dk个理解为共dk组，每组的第1个，则dk+1理解为从第1组的i=2开始进行直插

i++：i变为dk+2，理解为做完第1组第2个，第2组继续从i=2开始直插

例：(52, 38, 66, 87, 77, 16, 30, 52, 62, 7)

若：dk=5，则i=6， L.r[6] =16，进行(52,16)组直插
i=7， L.r[7] =30，进行(38,30)组直插
i=8， L.r[8] =52，进行(66,52)组直插
i=9， L.r[9] =62，进行(87,62)组直插
i=10，L.r[10]=77，进行(77,7) 组直插
循环结束

dk=5结束后:16 30 52 62 7 52 38 66 87 77

若：dk=3，则i=4， L.r[4] =62，(16,62,38,77)组的62
i=5， L.r[5] =7， (30,7,66)组的7
i=6， L.r[6] =52，(52,52,87)组的52
i=7， L.r[7] =38，(16,62,38,77)组的38
i=8， L.r[8] =66，(30,7,66)组的66
i=9， L.r[9] =87，(52,52,87)组的87
i=10，L.r[10]=77，(16,62,38,77)组的77
循环结束

dlta[]：放增量序列，要保证最后一个是1
t：dlta数组的大小

★ 算法实现(P.272 算法10.5)

```
void ShellSort(SqList &L, int dlta[], int t)
{
    /* 循环做t趟shell排序 */
    for(k=0; k<t; k++)
        ShellInsert(L, delta[k]);
}
```

§10. 内部排序

10.2. 插入排序

10.2.3. 希尔排序

效率分析：

★ 取决于增量序列(要求序列无除1外的公因子)，一般是$\lceil n/2 \rceil$

约 $O(n^{1.3})$ - $O(n^{1.5})$

§10. 内部排序

10.3. 交换排序(书:快速排序)

10.3.1. 冒泡排序

★ 基本方法

两两比较相邻的待排序记录的关键字，对不满足顺序要求的相邻记录进行交换

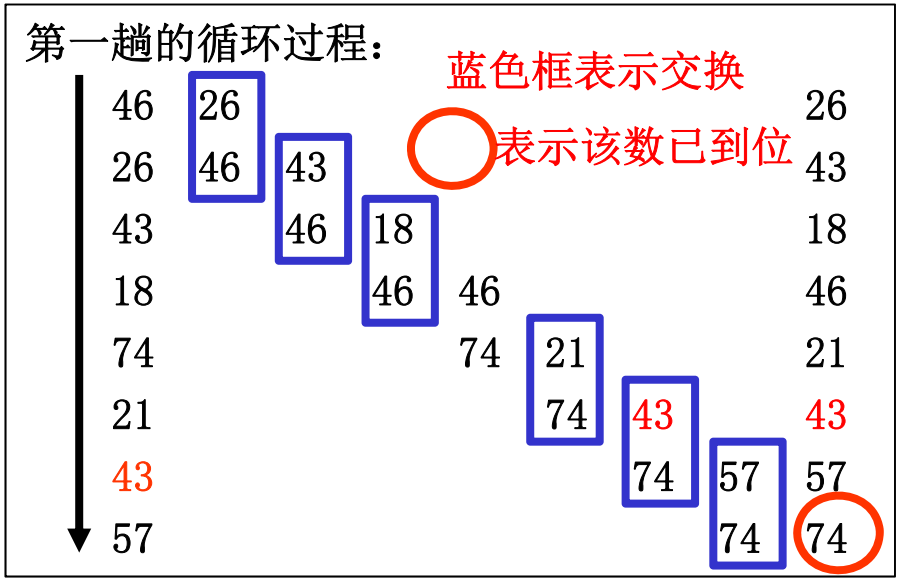
★ 排序过程

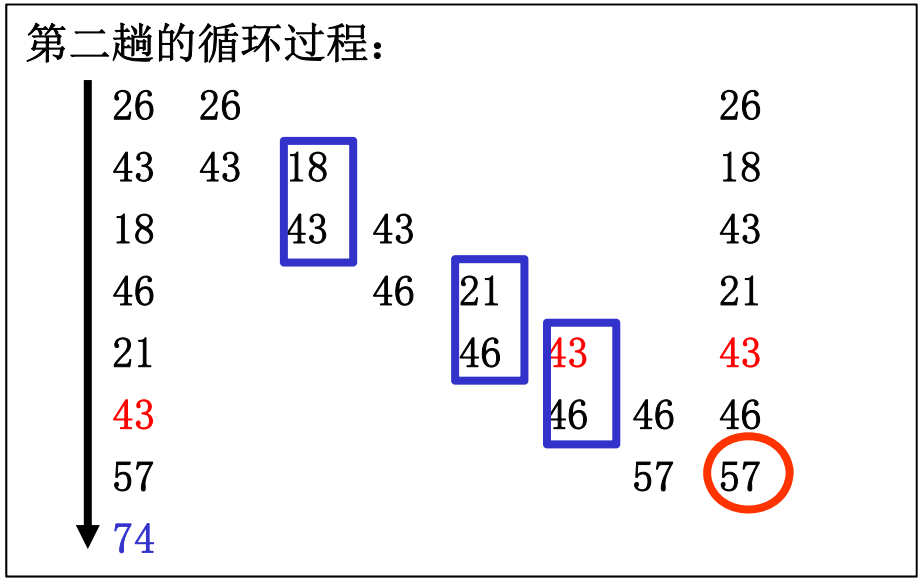
一趟冒泡：从1 ~ i-1循环，若$R_{j+1}<R_j$，则交换完成后，最大数在最后一个

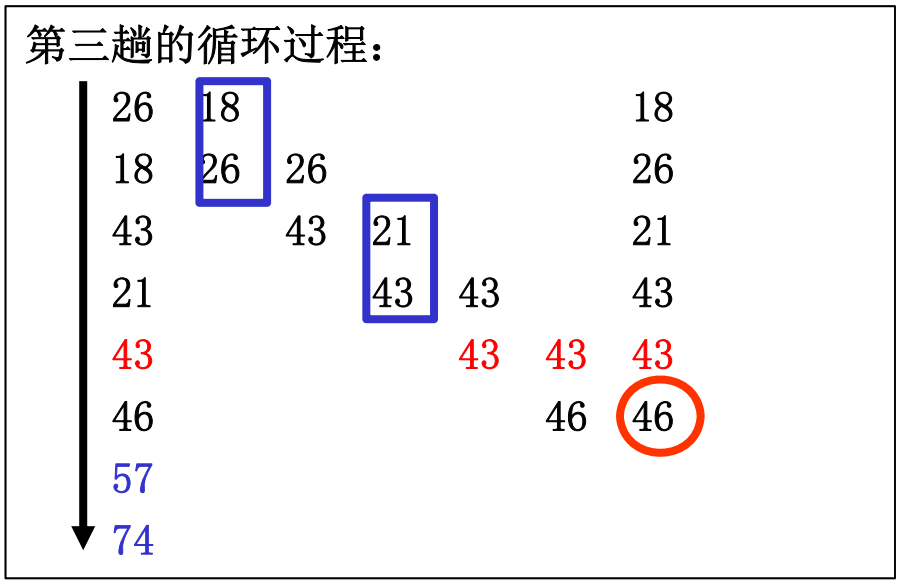
冒泡排序：n-1次一趟冒泡

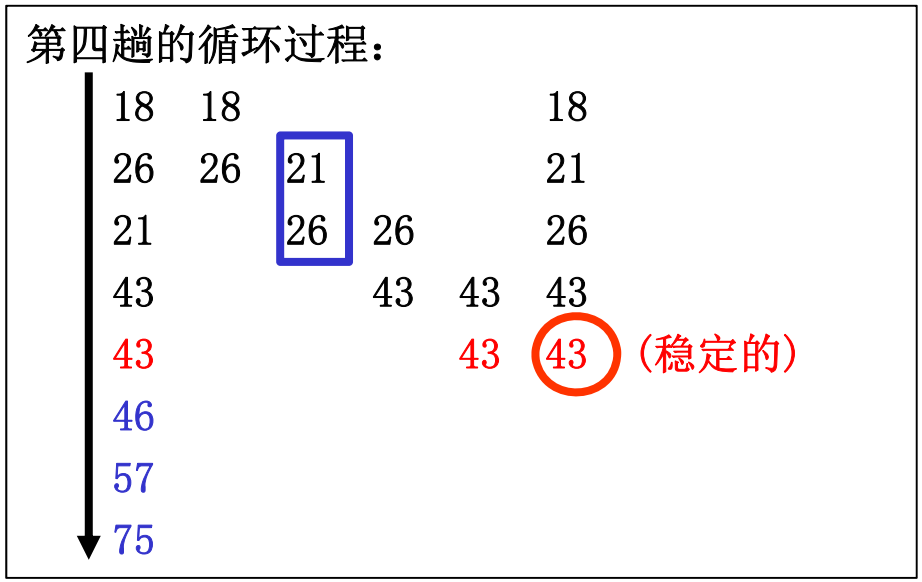
§10. 内部排序

例：待排序列为{46，26，43，18，74，21，43，57}





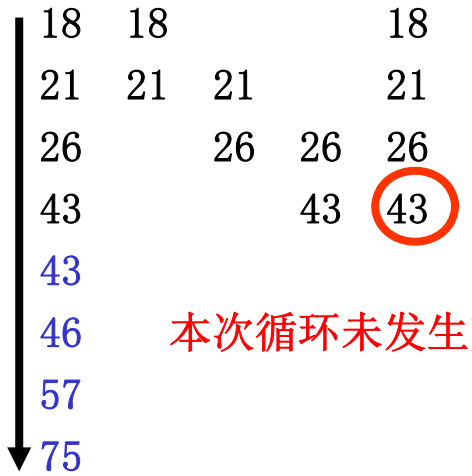




§10. 内部排序

例：待排序列为{46, 26, 43, 18, 74, 21, 43, 57}

第五趟的循环过程：



第六趟的循环过程：

第七趟的循环过程：

第五趟：

本次循环未发生交换
=> 数据已全部有序
=> 排序完成

第六、七趟没有必要运行

★ 算法实现

```
void BubbleSort(SqList &L)
{
    change_flag=1; //先置1
    for(i=L.length; change_flag && i>=2; i--) {
        change_flag = 0;    //每趟置交换标记为0
        for(j=1; j<i; j++) //无大括号
            if (LT(L.r[j+1].key, L.r[j].key) {
                temp     = L.r[j+1];
                L.r[j+1] = L.r[j];
                L.r[j]   = temp;
                change_flag=1; //置交换标记为1
                } // end of if
        } //end of for i
}
```

§10. 内部排序

10.3. 交换排序（书：快速排序）

10.3.1. 冒泡排序

$O(n^2)$

效率分析：

最好：初始有序，n-1次比较，0次交换

最坏：初始逆序，比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

交换次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

交换移动次数：$\sum_{i=2}^{n}3(n-i)=\frac{3n(n-1)}{2}$

§10. 内部排序

10.3. 交换排序(书:快速排序)

10.3.2. 快速排序

★ 排序过程(递归)

① 从待排序数据序列中任选一个元素(一般是第1个)做为基准元素(也称为枢轴、支点)，经过比较交换后放在指定位置，使基准位置左边均小于基准元素，右边均大于基准元素

② 基准元素左右两边重复步骤①，使全部元素有序

★ 基准元素的定位

① 设指针low，high，分别指向待排记录的第一个元素和最后一个元素，将基准元素(第1个)暂存入[0]中

② 先从high开始,当 L.r[high].key < L.r[0].key 时,将 L.r[high] 移入 L.r[low] 中，令low=low+1，执行步骤③，否则令high=high-1，重复步骤②

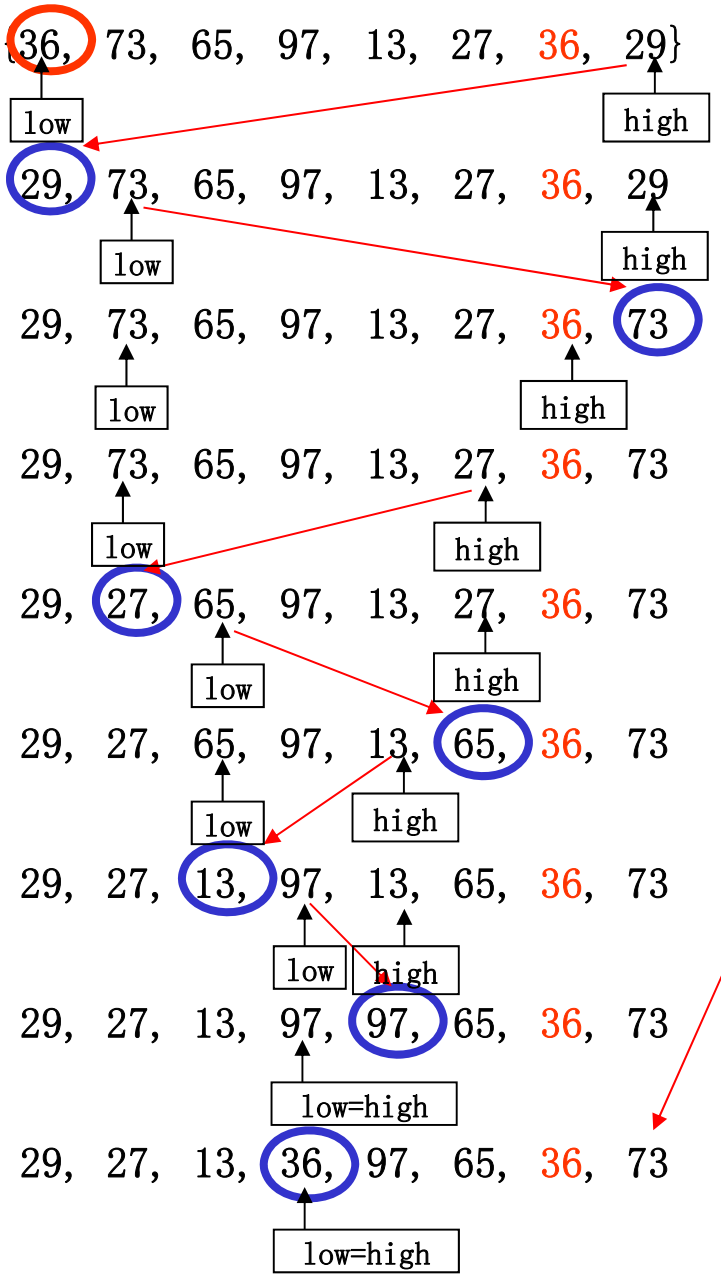
③ 再从low开始，当 L.r[low].key > L.r[0].key 时，将 L.r[low] 移入 L.r[high] 中，令high=high-1，执行步骤② ，否则令low=low+1，重复③

④ 重复②③直到low==high为止，[low]就是基准元素应在位置，将 L.r[0] 移入 L.r[low] 中，定位完成

§10. 内部排序

例：待排序列为 {36, 73, 65, 97, 13, 27, 36, 29}

初始：



L.r[0].key=36

第1趟结束后：	[29	27	13]	36	[97	65	36	73]
第2趟结束后：	[13	27]	29	36	[73	65	36]	97
第3趟结束后：	13	[27]	29	36	[36	65]	73	97
第4趟结束后：	13	27	29	36	36	[65]	73	97
最终结果：	13	27	29	36	36	65	73	97

稳定???

★ 算法实现(P.274 算法10.6(b))

```
int Partition(SqList &L, int low, int high)
{   L.r[0] = L.r[low];  //暂存于[0]中
    pivotkey = L.r[low].key; //取出来，访问时快些
    while(low<high) {
        while(low<high && L.r[high].key>=pivotkey)
            high--;
        L.r[low] = L.r[high]; // while退出后执行此句
        while(low<high && L.r[low].key<=pivotkey)
            low++;
        L.r[high] = L.r[low]; // while退出后执行此句
        }
    L.r[low] = L.r[0];  //基准元素到位
    return low;  //返回基准位置的下标
}
```

★ 算法实现(P.275 算法10.7)

```
void QSort(SqList &L, int low, int high)
{
    if (low<high) {
       pivotloc = Partition(L, low, high);
       QSort(L, low, pivotloc-1);   //左半区
       QSort(L, pivotloc+1, high);  //右半区
       }
}
```

★ 算法实现(P.276 算法10.8)

```
void QuickSort(SqList &L)
{
    QSort(L, 1, L.length);
}
```

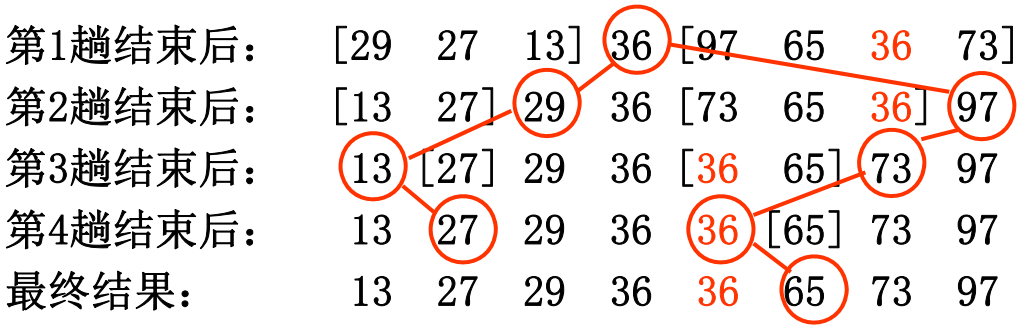
§10. 内部排序

10.3. 交换排序(书:快速排序)

10.3.2. 快速排序

★ 效率分析

每趟排序选择的基准数据形成一棵二叉查找树



最好：平衡二叉树

平均查找长度$O(\log_2 n)$，n个结点$O(n\log_2 n)$

最坏：高度为n的二叉树

平均查找长度$O(n)$，n个结点 $O(n^2)$

$O(n\log_2 n)$

§10. 内部排序

10.4. 选择排序

10.4.1. 直接选择排序

★ 基本方法

每次从待排序列中选择一个关键字最小的记录放在已排好的序列中，重复操作直到有序

★ 排序过程

第i趟直选：从R[i]-R[n]的待排序列中选择关键字值最小的记录，与R[i]进行交换

直选排序：重复n-1次

例：待排序列为 {48, 36, 65, 97, 15, 27, 48}

第1趟结束后： [15] 36 65 97 48 27 48

第2趟结束后： [15 27] 65 97 48 36 48

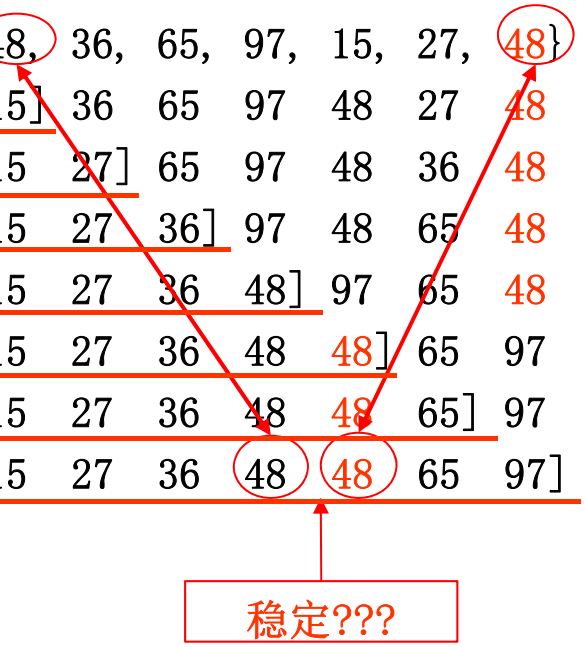
第3趟结束后： [15 27 36] 97 48 65 48

第4趟结束后： [15 27 36 48] 97 65 48

第5趟结束后： [15 27 36 48 48] 65 97

第6趟结束后： [15 27 36 48 48 65] 97

最终结果： [15 27 36 48 48 65 97]



★ 算法实现(P.277算法10.9改进,合并SelectMinKey函数)

```
void SelectSort(SqList &L)
{
    for(i=1; i<L.length; i++) { //循环n-1次
        m=i;
        for(j=i+1; i<=L.length; j++)
            if (LT(L.r[j].key, L.r[m].key))
                m=j;                          内循环查找[i..n]中最小值的位置
        if (m!=i) {
            L.r[0] = L.r[i];
            L.r[i] = L.r[m];      L.r[0]当temp，m和i互换
            L.r[m] = L.r[0];
           }
        }
}
```

§10. 内部排序

10.4. 选择排序

10.4.1. 直接选择排序

$O(n^2)$

★ 效率分析

空间：1个辅助空间(可用R[0])

时间：比较次数：$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$

交换次数：最好：0次

最坏：n-1次

交换移动次数：最好：0次

最坏：3(n-1)次

§10. 内部排序

10.4. 选择排序

10.4.2. 堆排序

★ 堆的定义

n个元素的关键字序列$(k_1, k_2, \ldots k_n)$，当且仅当对于所有的$i=1,2,\ldots,\lfloor n/2 \rfloor$，

都满足 $\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases}$ 小顶堆（小根堆） 或 $\begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases}$ 大顶堆（大根堆） 时，则称为堆

- 当作完全二叉树理解，则任意非叶结点均不大于或不小于其左右孩子结点的值
- 堆顶元素是整个序列中最小/最大的元素

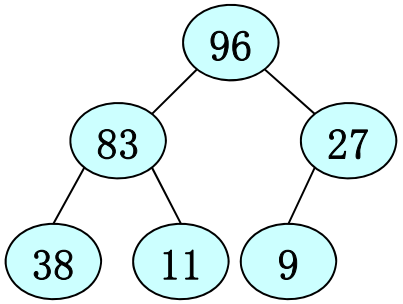
例1：96, 83, 27, 38, 11, 9

i=1：96≥83 && 96≥27

i=2：83≥38 && 83≥11

i=3：27≥9 && 2i+1不存在

大顶堆



例2：12, 36, 24, 85, 47, 30, 53, 91, 88

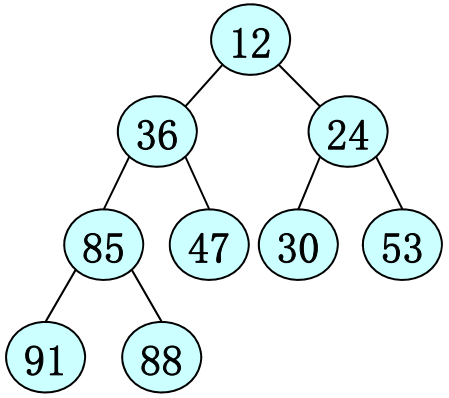
i=1：12≤36 && 12≤24

i=2：36≤85 && 36≤47

i=3：24≤30 && 24≤53

i=4：85≤91 && 85≤88

小顶堆



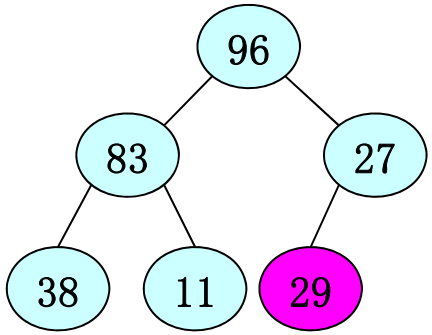
例3：96, 83, 27, 38, 11, 29

i=1：96≥83 && 96≥27

i=2：83≥38 && 83≥11

i=3：27≥29（不满足）

不是堆



§10. 内部排序

10.4. 选择排序

10.4.2. 堆排序

★ 排序过程

将待排序的n个关键字，按堆定义排成一个序列(建堆)，然后输出堆顶元素(最大值/最小值)，其余关键字再次排列成堆(调整堆)，再次输出堆顶元素，重复至所有关键字有序

★ 堆的建立(小顶堆)

① 将待排序关键字按从上到下，从左至右的方法排放成一棵完全二叉树

② 从最后一个非叶结点(⌊n/2⌋)开始至根节点，依次进行筛选运算，算法如下：

(A) 将R_i与R_{2i}/R_{2i+1}中关键字值较小的一个(假设编号为j)进行比较

(B) 若$R_i \leqslant R_j$，筛选运算结束，否则，R_i与R_j互换

(C) 检查交换后的子树是否为堆，若不为堆，重复至叶子结点为止

★ 堆的建立(大顶堆)

① 将待排序关键字按从上到下，从左至右的方法排放成一棵完全二叉树

② 从最后一个非叶结点(⌊n/2⌋)开始至根节点，依次进行筛选运算，算法如下：

(A) 将R_i与R_{2i}/R_{2i+1}中关键字值较大的一个(假设编号为j)进行比较

(B) 若$R_i \geqslant R_j$，筛选运算结束，否则，R_i与R_j互换

(C) 检查交换后的子树是否为堆，若不为堆，重复至叶子结点为止

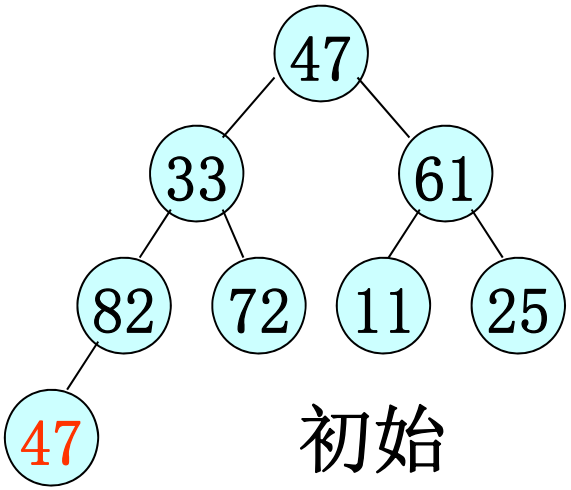
★ 堆的调整(适用于小顶堆/大顶堆)

① 输出堆顶元素

② 将序列中的最后一个元素放至堆顶，进行筛选运算至形成堆为止

③ 重复①②至所有元素输出

例：待排序列为 {47， 33， 61， 82， 72， 11， 25， 47}

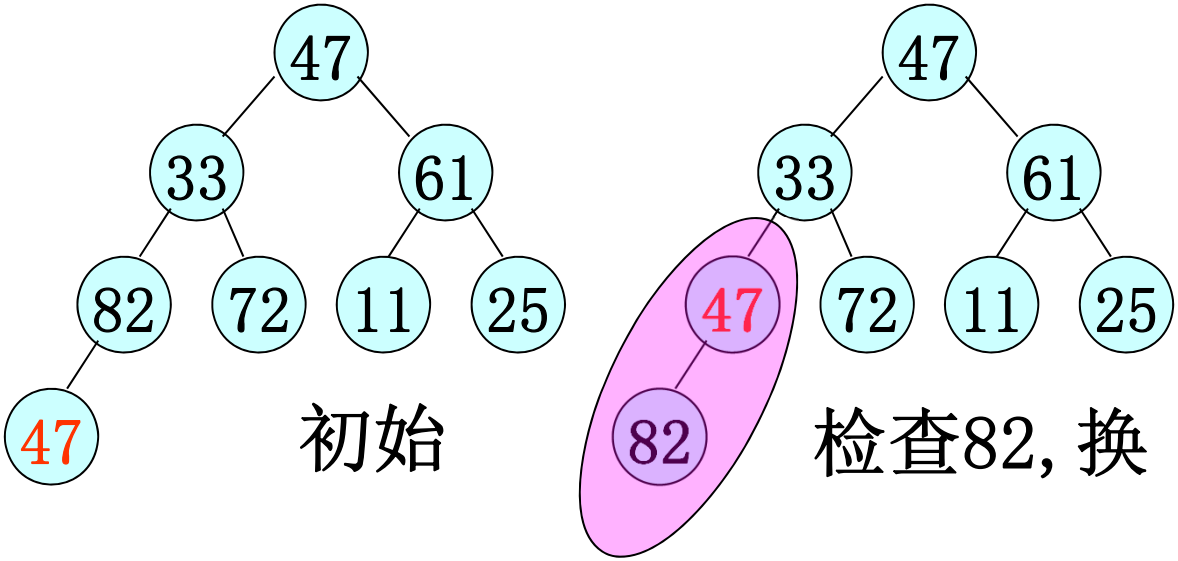


初始

数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	47	33	61	82	72	11	25	47

堆的建立过程

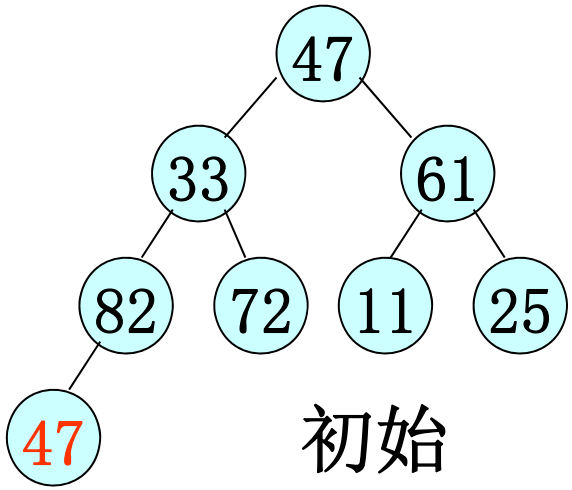
例：待排序列为 {47， 33， 61， 82， 72， 11， 25， 47}



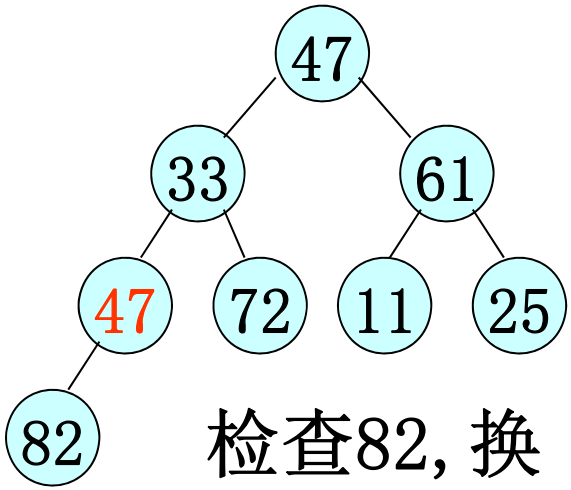
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	47	33	61	47	72	11	25	82

堆的建立过程

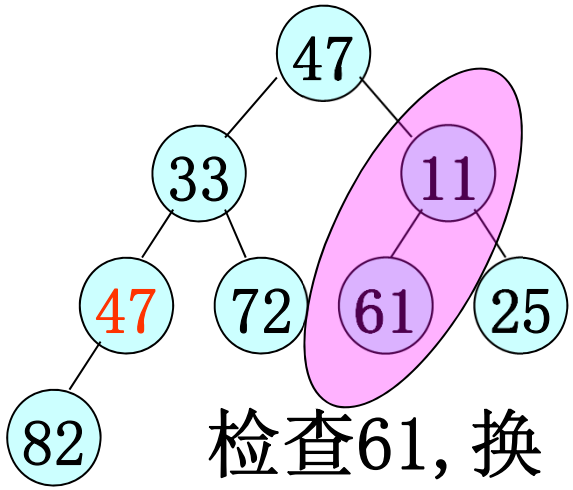
例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



初始



检查82, 换

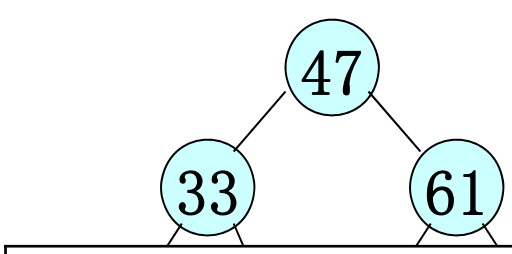


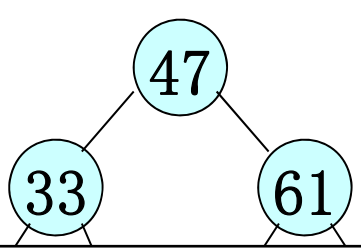
检查61, 换

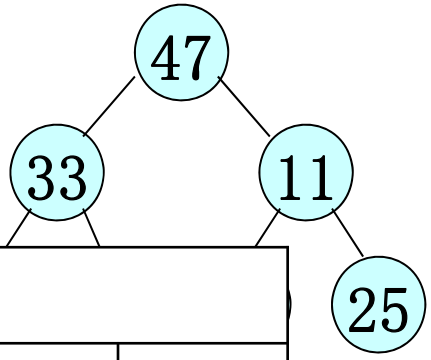
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	47	33	11	47	72	61	25	82

堆的建立过程

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



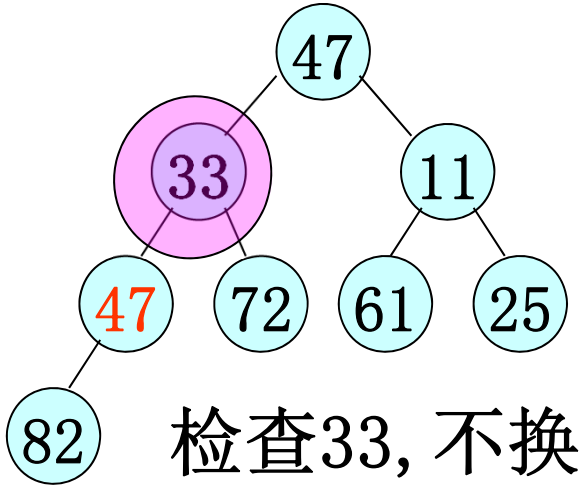




数组的实际情况：

下标	0	1	2	3	4	5	6	7	8
值	闲置	47	33	11	47	72	61	25	82

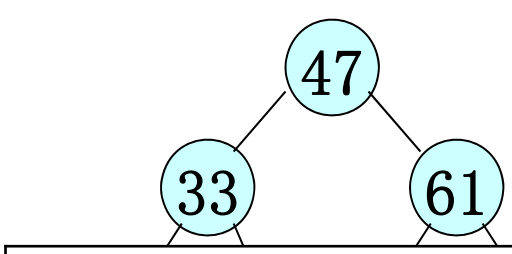
, 换

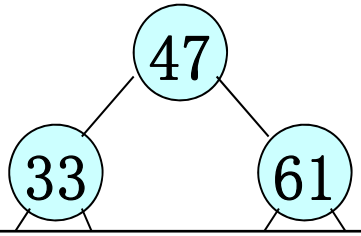


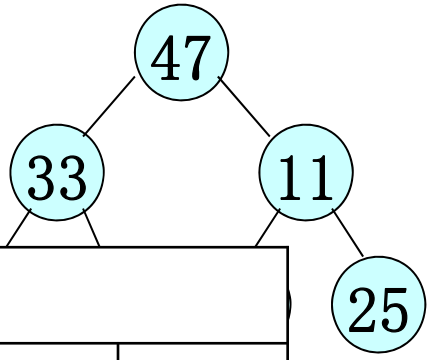
检查33, 不换

堆的建立过程

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



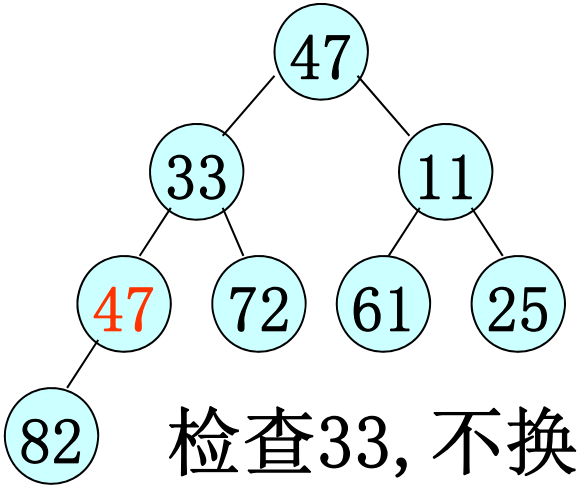




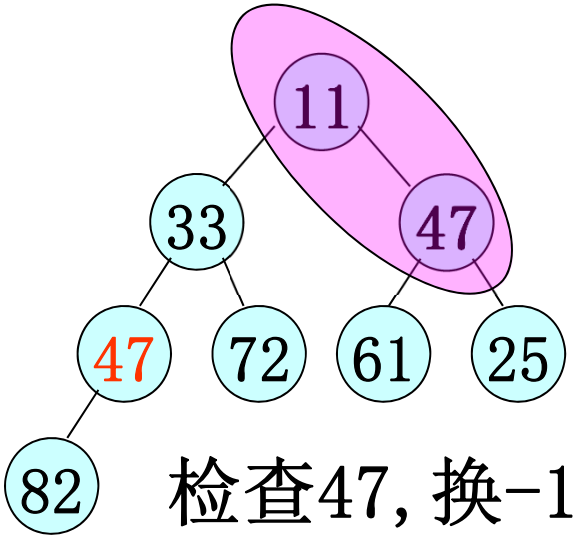
数组的实际情况：

下标	0	1	2	3	4	5	6	7	8
值	闲置	11	33	47	47	72	61	25	82

, 换



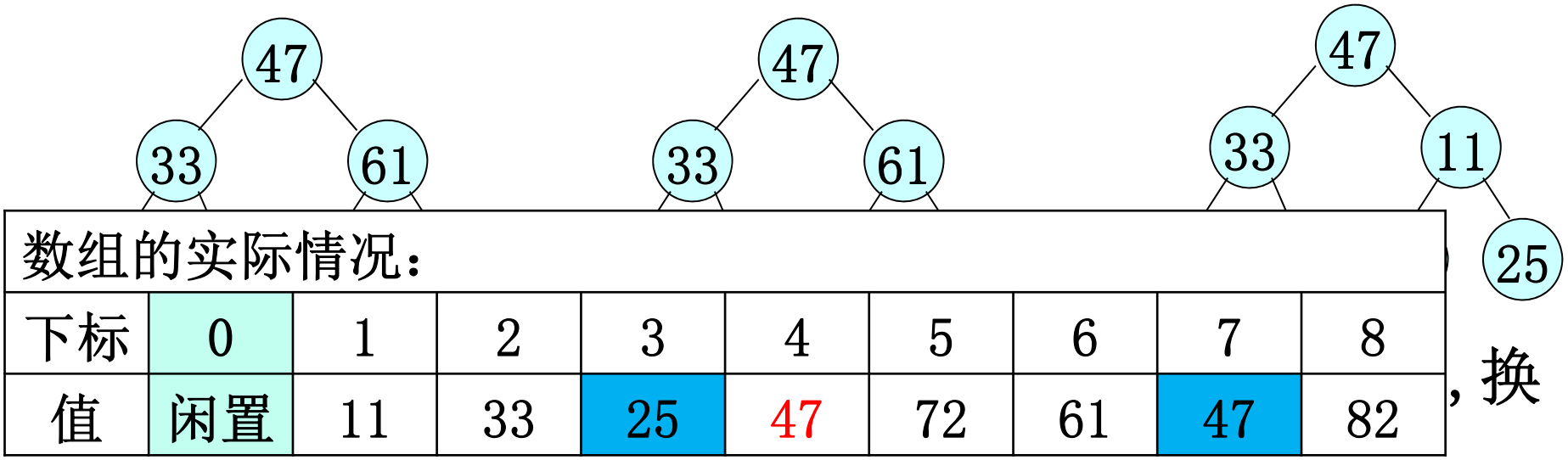
检查33, 不换



检查47, 换-1

堆的建立过程

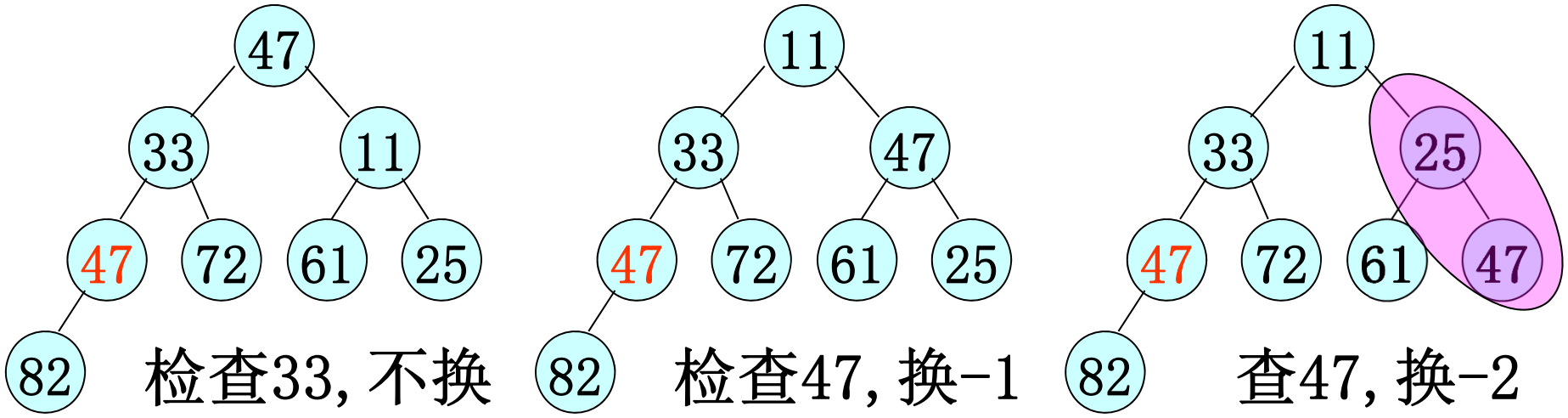
例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



数组的实际情况：

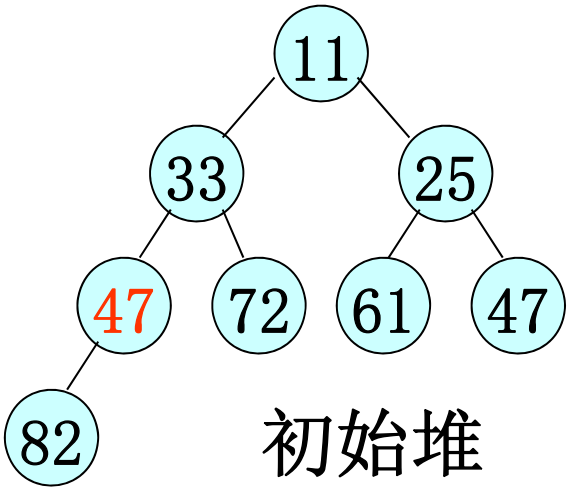
下标	0	1	2	3	4	5	6	7	8
值	闲置	11	33	25	47	72	61	47	82

, 换



堆的建立过程

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

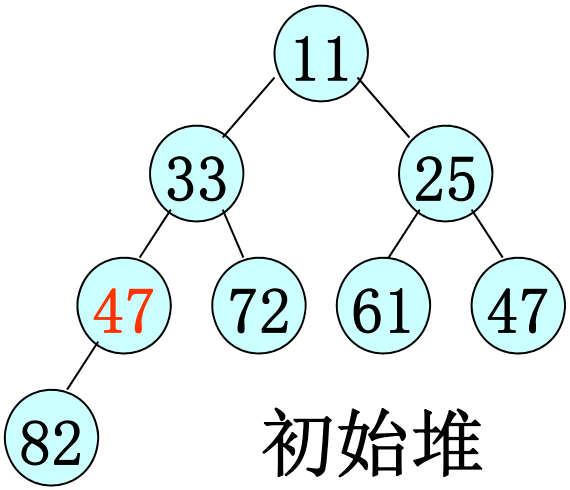


初始堆

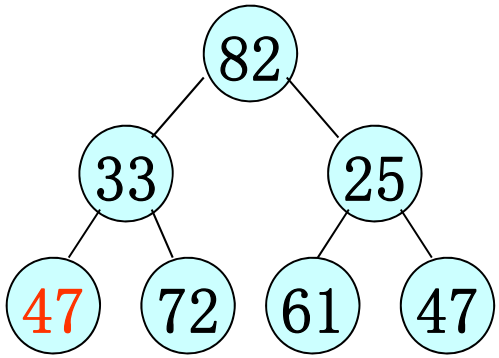
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	11	33	25	47	72	61	47	82

输出：

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



初始堆



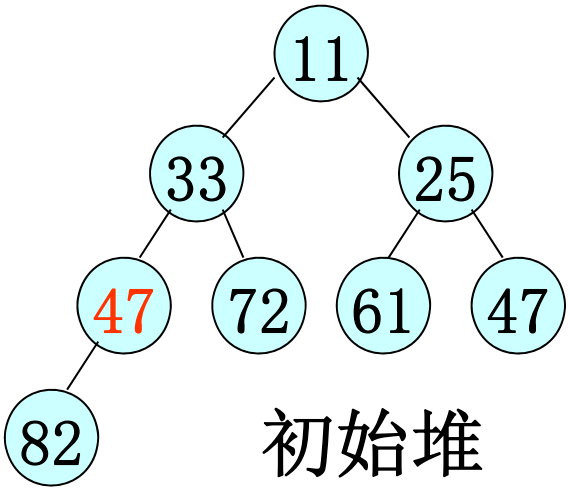
出11，82至顶

数组的实际情况：（将11和82互换，11不再参与排序）

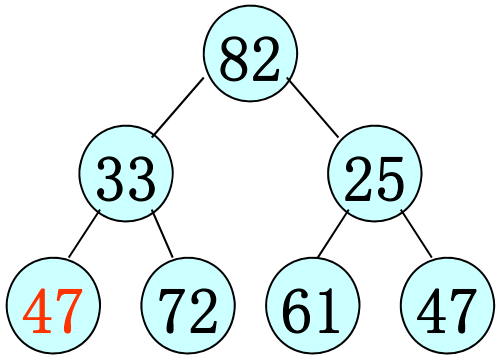
下标	0	1	2	3	4	5	6	7	8
值	闲置	82	33	25	47	72	61	47	11

输出：11

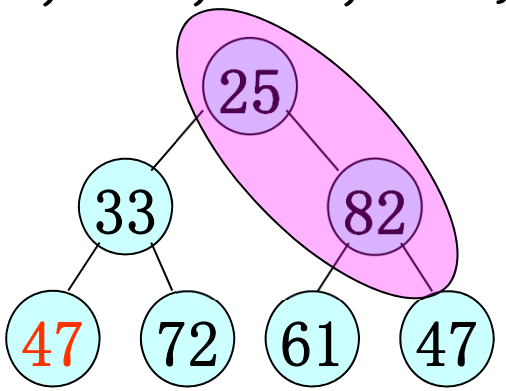
例：待排序列为 {47，33，61，82，72，11，25，47}



初始堆



出11，82至顶

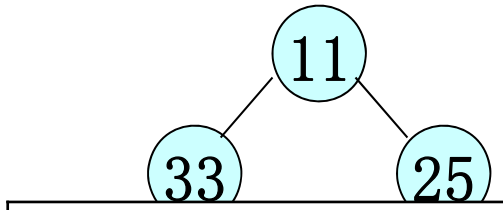


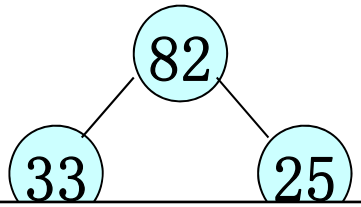
调整82-1

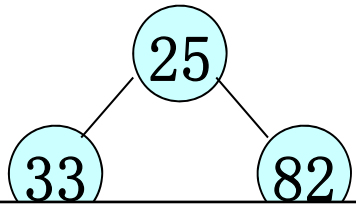
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	25	33	82	47	72	61	47	11

输出：11

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

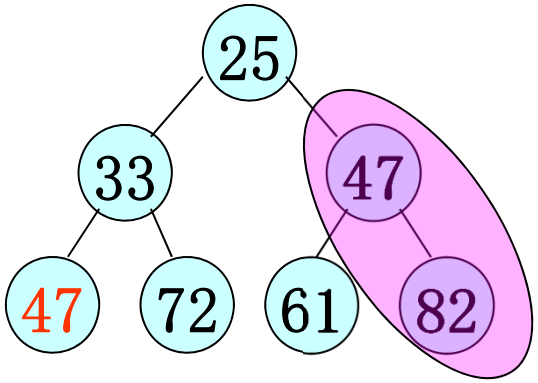






数组的实际情况：

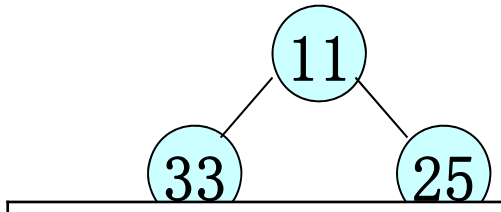
下标	0	1	2	3	4	5	6	7	8
值	闲置	25	33	47	47	72	61	82	11

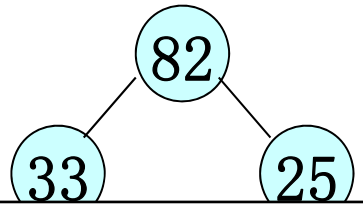


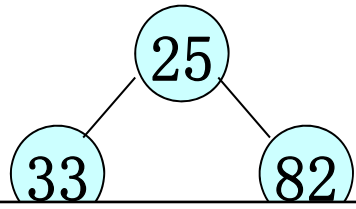
调整82-2

输出：11

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

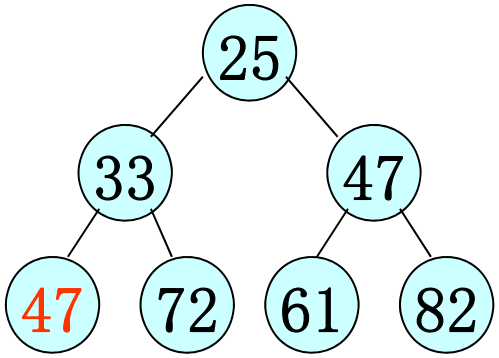




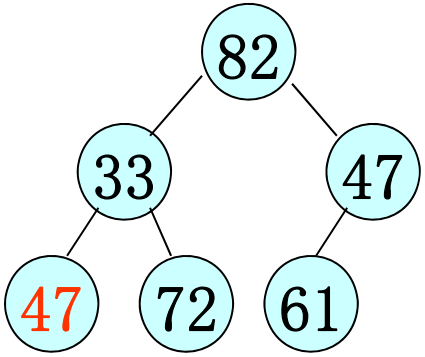


数组的实际情况：（将25和82互换，25不再参与排序）

下标	0	1	2	3	4	5	6	7	8
值	闲置	82	33	47	47	72	61	25	11



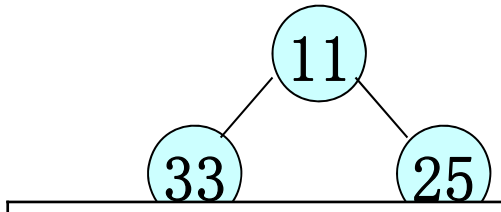
调整82-2

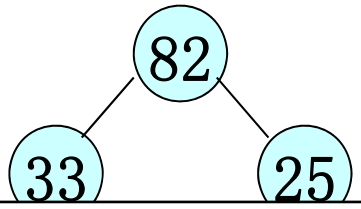


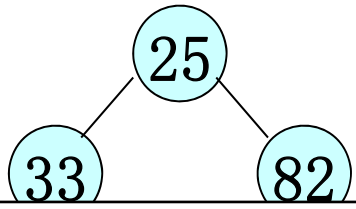
出25，82至顶

输出：11 25

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

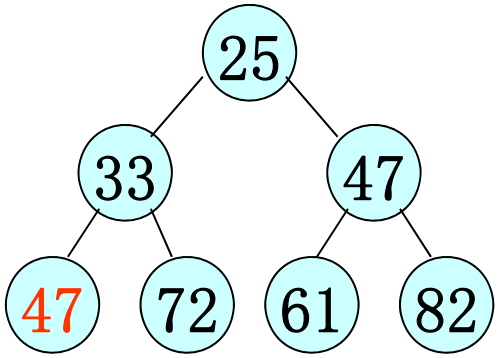




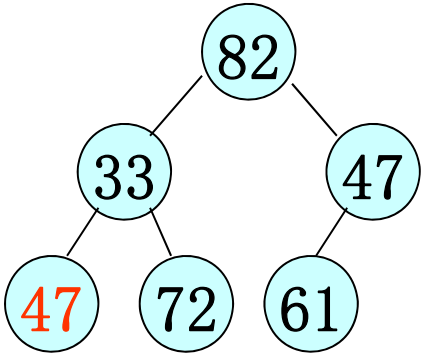


数组的实际情况：

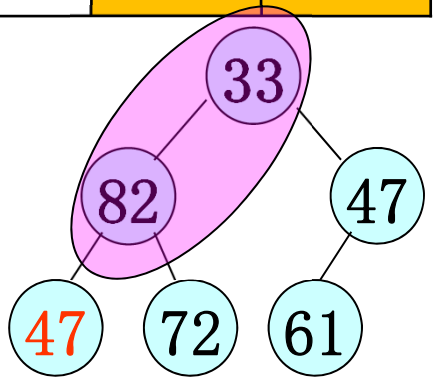
下标	0	1	2	3	4	5	6	7	8
值	闲置	33	82	47	47	72	61	25	11



调整82-2



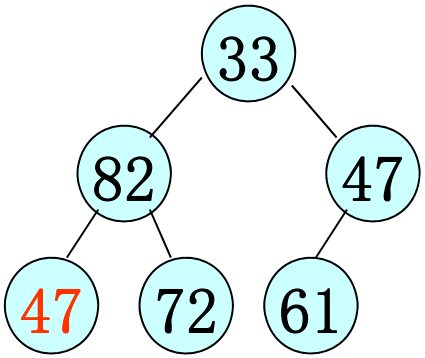
出25，82至顶



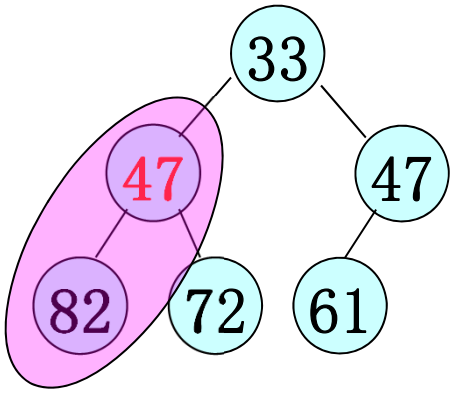
调整82-1

输出：11 25

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



调整82-1

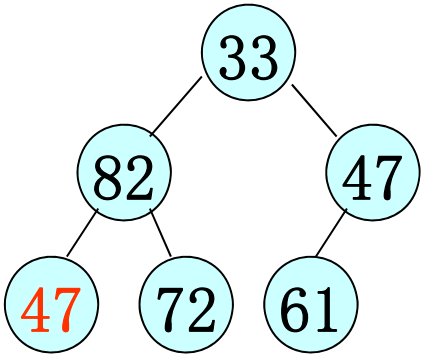


调整82-2

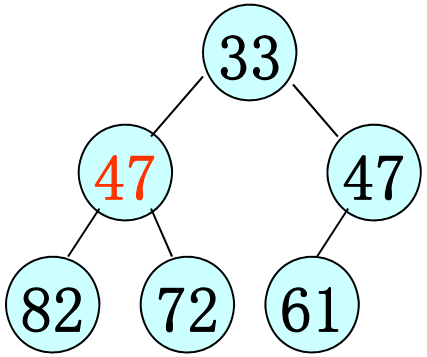
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	33	47	47	82	72	61	25	11

输出：11 25

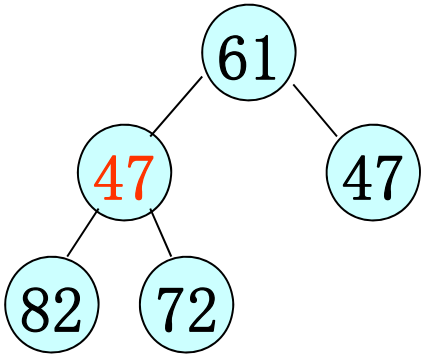
例：待排序列为 {47，33，61，82，72，11，25，47}



调整82-1



调整82-2

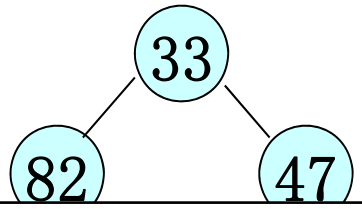


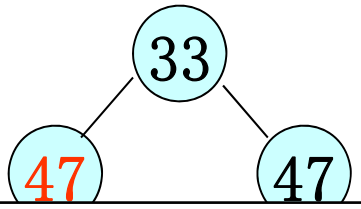
出33，61至顶

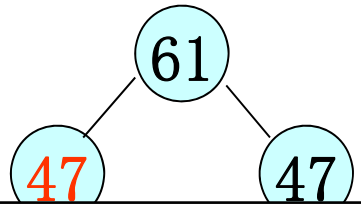
数组的实际情况：（将33和61互换，33不再参与排序）									
下标	0	1	2	3	4	5	6	7	8
值	闲置	61	47	47	82	72	33	25	11

输出：11 25 33

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

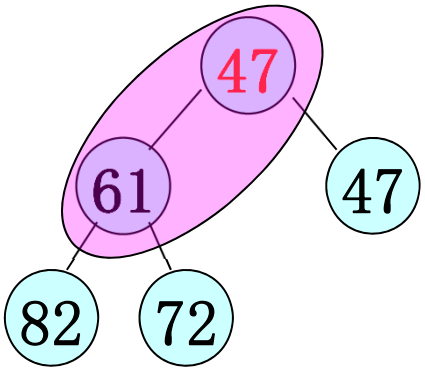






数组的实际情况：

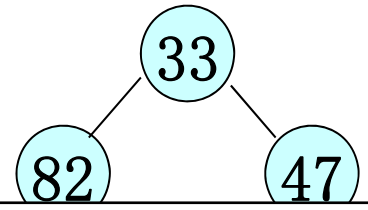
下标	0	1	2	3	4	5	6	7	8
值	闲置	47	61	47	82	72	33	25	11

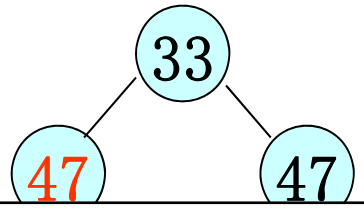


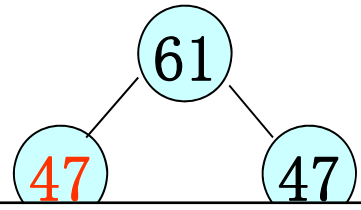
调整61

输出：11 25 33

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

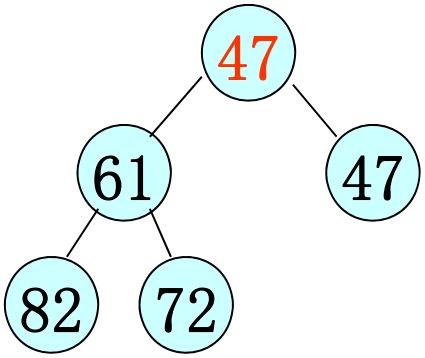




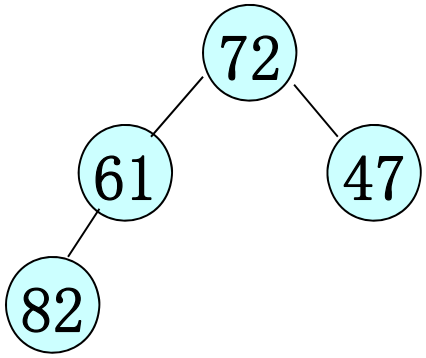


数组的实际情况：（将47和72互换，47不再参与排序）

下标	0	1	2	3	4	5	6	7	8
值	闲置	72	61	47	82	47	33	25	11



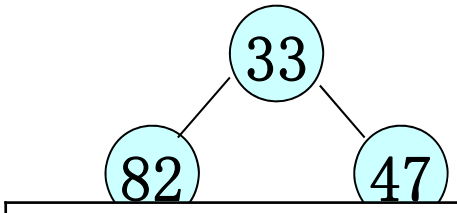
调整61

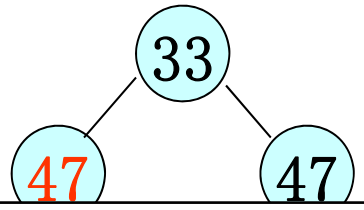


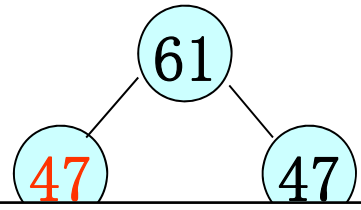
出47，72至顶

输出：11 25 33 47

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

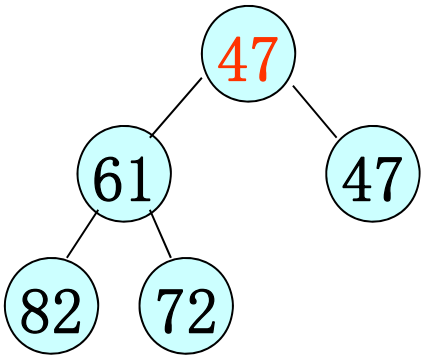




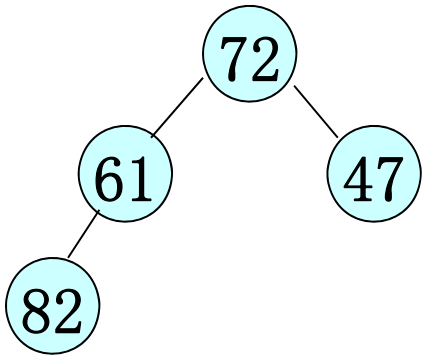


数组的实际情况：

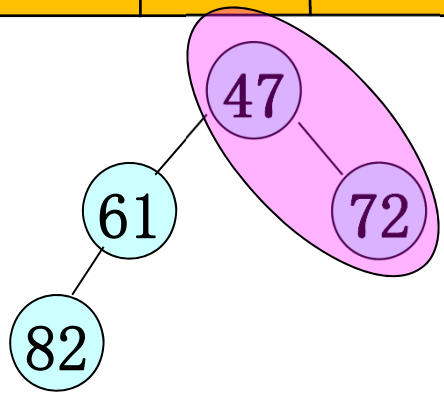
下标	0	1	2	3	4	5	6	7	8
值	闲置	47	61	72	82	47	33	25	11



调整61



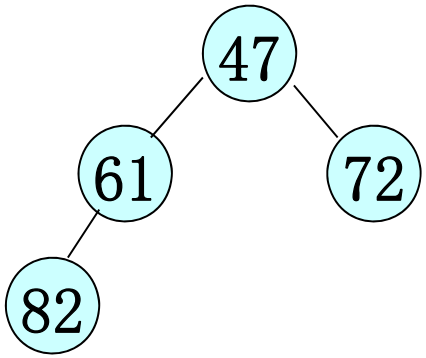
出47，72至顶



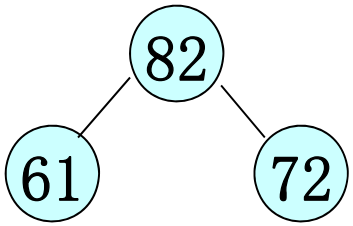
调整72

输出：11 25 33 47

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}



调整72



出47，82至顶

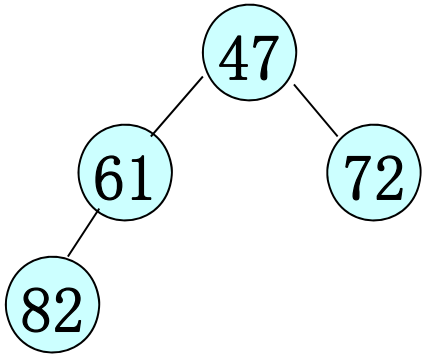
数组的实际情况：（将47和82互换，47不再参与排序）

下标	0	1	2	3	4	5	6	7	8
值	闲置	82	61	72	47	47	33	25	11

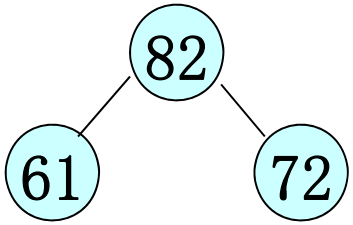
不稳定

输出：11 25 33 47 47

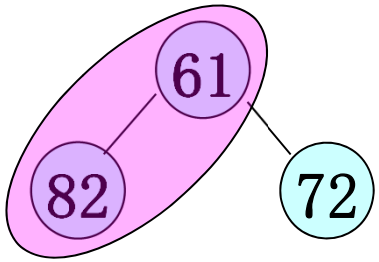
例：待排序列为 {47， 33， 61， 82， 72， 11， 25， 47}



调整72



出47，82至顶

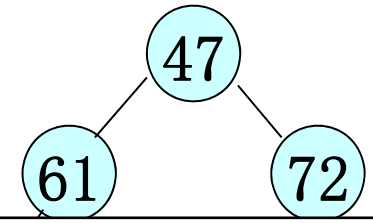


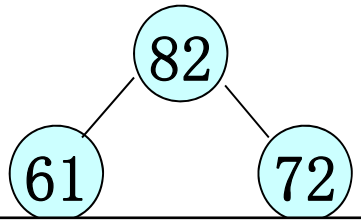
调整82

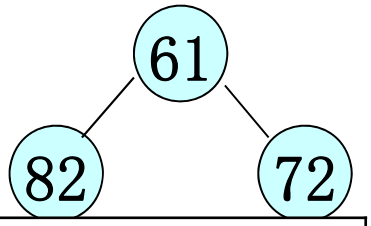
数组的实际情况：									
下标	0	1	2	3	4	5	6	7	8
值	闲置	61	82	72	47	47	33	25	11

输出：11 25 33 47 47

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

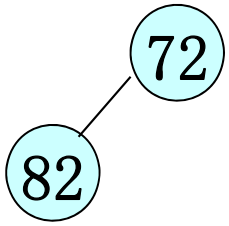






数组的实际情况：（将61和72互换，61不再参与排序）

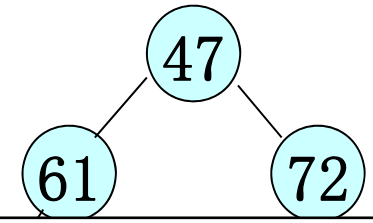
下标	0	1	2	3	4	5	6	7	8
值	闲置	72	82	61	47	47	33	25	11

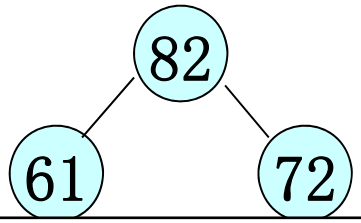


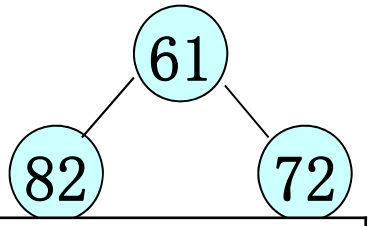
出61，72至顶

输出：11 25 33 47 47 61

例：待排序列为 {47， 33， 61， 82， 72， 11， 25， 47}

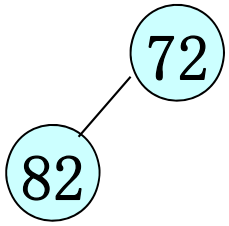






数组的实际情况：（将72和82互换，72不再参与排序）

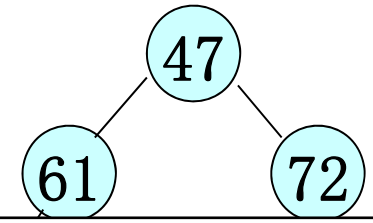
下标	0	1	2	3	4	5	6	7	8
值	闲置	82	72	61	47	47	33	25	11

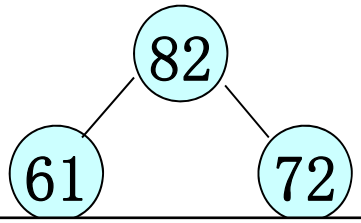


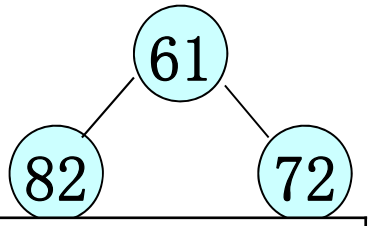
出61，72至顶　　出72，82至顶

输出：11 25 33 47 47 61 72

例：待排序列为 {47, 33, 61, 82, 72, 11, 25, 47}

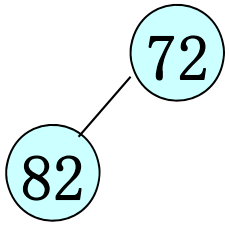






数组的实际情况：（仅剩一个82，不再参与排序）

下标	0	1	2	3	4	5	6	7	8
值	闲置	82	72	61	47	47	33	25	11



小顶堆筛选算法：
★ 如果边排序边输出，则输出为递增序列
★ 如果希望全部排序完成再输出整个数组，则输出为递减序列

出61，72至顶　　出72，82至顶　　出82，堆空

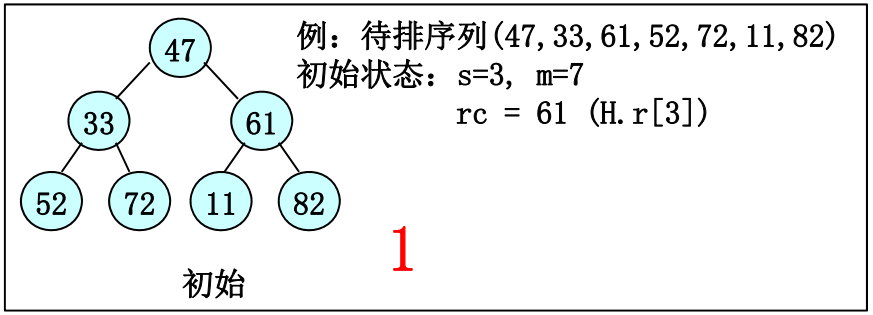
输出：11 25 33 47 47 61 72 82

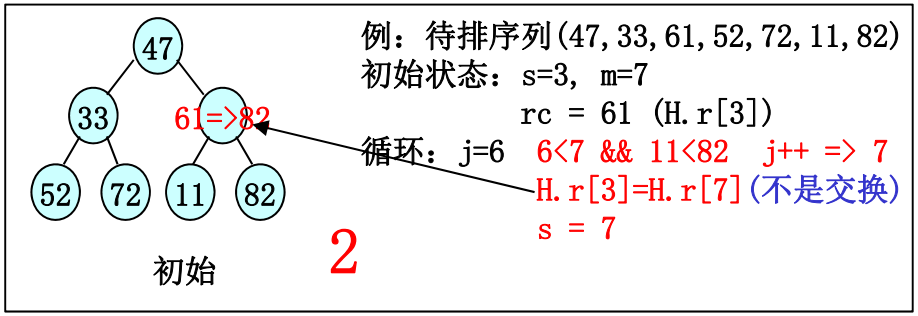
★ 算法实现(筛选算法 P.282 算法10.10)

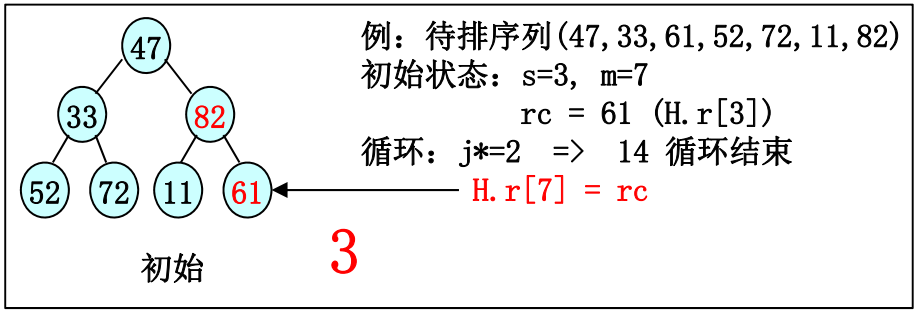
算法是大顶堆，
刚才图示是小顶堆

```
void HeapAdjust(HeapType &H, int s, int m)
{   rc = H.r[s];  //保存r[s]
    /* 在以s为根的子树中从根到叶子循环
       j=2*s：j为s的左子树  j*=2：j指向j的左子树 */
    for(j=2*s; j<=m; j*=2) {
        if (j<m && LT(H.r[j].key, H.r[j+1].key))
            j++;  //如果右子树大，则选右子树(大顶堆)
        if (!LT(rc.key, H.r[j].key))   rc.key>=H.r[j].key
            break;                     则结束，s为插入位置
        H.r[s] = H.r[j]; //根=左右子树中大的(不是交换)
        s = j; //s成为子树的根
        }
    H.r[s] = rc; //将保存的根赋值给s位置
}
```

1



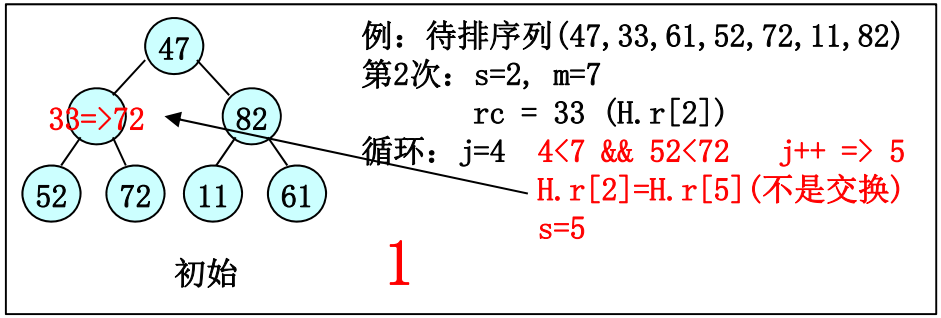


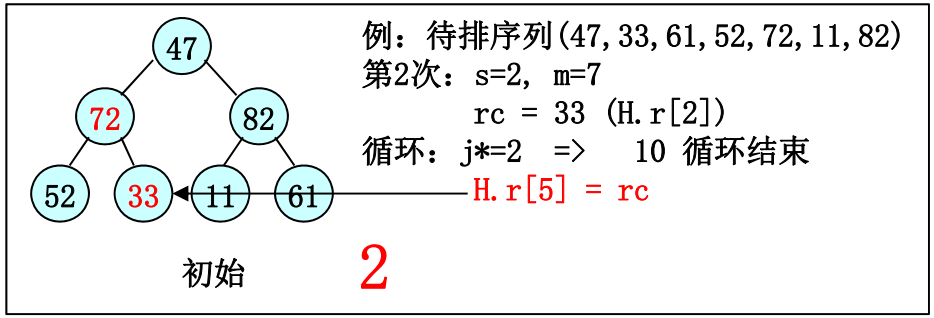


★ 算法实现(筛选算法 P.282 算法10.10)

算法是大顶堆，
刚才图示是小顶堆

```
void HeapAdjust(HeapType &H, int s, int m)
{   rc = H.r[s];  //保存r[s]
    /* 在以s为根的子树中从根到叶子循环
       j=2*s：j为s的左子树  j*=2：j指向j的左子树 */
    for(j=2*s; j<=m; j*=2) {
        if (j<m && LT(H.r[j].key, H.r[j+1].key))
            j++;  //如果右子树大，则选右子树(大顶堆)
        if (!LT(rc.key, H.r[j].key))   rc.key>=H.r[j].key
            break;                     则结束，s为插入位置
        H.r[s] = H.r[j]; //根=左右子树中大的(不是交换)
        s = j; //s成为子树的根
        }
    H.r[s] = rc; //将保存的根赋值给s位置
}
```

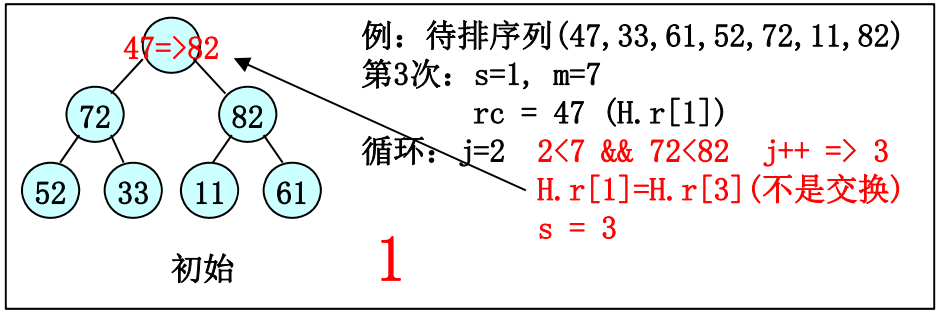


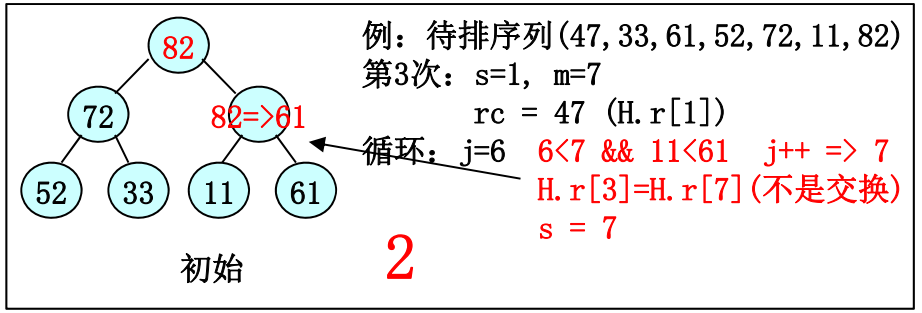


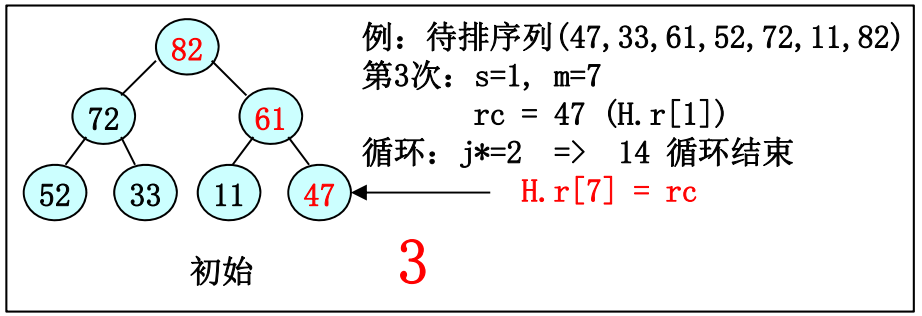
★ 算法实现(筛选算法 P.282 算法10.10)

算法是大顶堆，
刚才图示是小顶堆

```
void HeapAdjust(HeapType &H, int s, int m)
{   rc = H.r[s];  //保存r[s]
    /* 在以s为根的子树中从根到叶子循环
       j=2*s：j为s的左子树  j*=2：j指向j的左子树 */
    for(j=2*s; j<=m; j*=2) {
        if (j<m && LT(H.r[j].key, H.r[j+1].key))
            j++;  //如果右子树大，则选右子树(大顶堆)
        if (!LT(rc.key, H.r[j].key))   rc.key>=H.r[j].key
            break;                     则结束，s为插入位置
        H.r[s] = H.r[j]; //根=左右子树中大的(不是交换)
        s = j; //s成为子树的根
        }
    H.r[s] = rc; //将保存的根赋值给s位置
}
```







★ 算法实现(堆排序算法 P.282 算法10.11)

```
void HeapSort(HeapType &H)
{
    //从最后一个非叶结点到根循环，完成后为初始堆
    for (i=H.length/2; i>0; i--)
        HeapAdjust(H, i, H.length);

    for (i=H.length; i>1; i--) { //i=1是根不循环
      temp   = H.r[1];
      H.r[1] = H.r[i];
      H.r[i] = temp;
      HeapAdjust(H, 1, i-1); //前i-1个调整为堆
      }
}
```

R[1] ⇔ R[i]
i个元素的堆，堆顶交换到输到R[i]

若HeapAdjust为大顶堆，则排序完成后，H中递增排列

§10. 内部排序

10.4. 选择排序

10.4.2. 堆排序 $O(n\log_2 n)$

★ 效率分析

堆所对应的完全二叉树高度 $h=\lfloor \log_2 n \rfloor +1$

建堆过程：要比较的结点高度为1~h-1

假设都是h-1层，则 $O(\log_2 n)$

共n/2个结点 $O(n\log_2 n)$

堆排序过程：第k层(k≤h，假设都在h层)的结点

调整至顶 $O(\log_2 n)$

n-1个结点 $O(n\log_2 n)$

§10. 内部排序

10.5. 归并排序

10.5.1. 二路归并排序

★ 排序方法

通过已排序的子序列合并得到一个完整的有序序列

● 初始认为各元素单独成为子序列

★ 排序过程

第i趟归并：每个子序列长度为2^{i-1}，相邻的两个子序列合并为长度2^i的序列

归并排序：共$\lceil \log_2 n \rceil$趟

子序列归并方法：两个子序列，各比较第1个记录,关键字小的就是归并后的第1个记录，
输出后继续比较子序列，依次类推至归并完成

§10. 内部排序

10.5. 归并排序

10.5.1. 二路归并排序

例：待排序列为

{22 36 06 78 25 43 73 18 39 62 27 66 13}

1趟归并：22 36 06 78 25 43 18 73 39 62 27 66 13

2趟归并：06 22 36 78 18 25 43 73 27 39 62 66 13

3趟归并：06 18 22 25 36 43 73 78 13 27 39 62 66

4趟归并：06 13 18 22 25 27 36 39 43 62 66 73 78

子序列的归并方法：
(第2趟=>第3趟)
06 22 36 78
18 25 43 73

第1次：06与18比较，得06
第2次：22与18比较，得18
第3次：22与25比较，得22
第4次：36与25比较，得25
第5次：36与43比较，得36
第6次：78与43比较，得43
第7次：78与73比较，得73
第8次：　　得78

★ 算法实现(将L~M, M+1~H两个有序序列归并为一个)

```
void Merge(RType SR[], RType &TR[], int L, int M, int H)
{
    int i=L, j=M+1, k=L;

    while( i<=M && j<=H)    //当两个序列都未完时进行循环
        if (LT(SR[i].key,SR[j].key))
            TR[k++] = SR[i++]; //小的放入TR，指向下一个
        else
            TR[k++] = SR[j++]; //小的放入TR，指向下一个

    while(i<=M) //将序列中剩余部分放到归并结果中
        TR[k++] = SR[i++];

    while(j<=H) //将序列中剩余部分放到归并结果中
        TR[k++] = SR[j++];
}
```

不可能同时被执行到

将存储于SR中的两部分有序归并后放入TR中(大小与SR同)
类似于P.26 算法2.7 顺序表的合并

★ 算法实现(一趟归并排序，归并长度len,有n个元素)

```
void MergePass(RType SR[], RType &TR[], int len, int n)
{   /* 当两个子序列长度都为n时循环 */
    for (i=1; i+2*len-1<=n; i+=2*len)
        Merge(SR, TR, i, i+len-1, i+2*len-1);

    if (i+len-1 < n)  //两个子序列，后一个长度不到len
        Merge(SR, TR, i, i+len-1, n);
    else {            //只有一个子序列
        for(; i<=n; i++)
            TR[i] = SR[i];
        }
}
```

len=1 n=13 i=1 Merge(1, 1, 2) i=3 3 3 4 i=5 5 5 6 i=7 7 7 8 i=9 9 9 10 i=11 11 11 12 i=13 for结束 执行else	len=2 n=13 i=1 Merge(1, 2, 4) i=5 5 6 8 i=9 9 10 12 i=13 for结束 执行else	len=4 n=13 i=1 Merge(1, 4, 8) i=9 for结束 执行if Merge(9,12,13)	len=8 n=13 i=1 for循环结束 执行if Merge(1, 8, 13)	len=16 n=13 i=1 for循环结束 执行else

★ 算法实现(归并排序)

```
void MergeSort(RType R[], int n)
{   int len=1;
    RType R1[]; //申请一个R1数组,大小同R,[0]不用
    while(len<n) {
       MergePass(R, R1, len, n);
       len=len*2;
       Mergepass(R1, R, len, n);
       len=len*2;
       }
}
```

每个循环R->R1/R1->R，
保证结果在R中
可能会出现一次重复的归并
目的仅为了从R1->R

```
          {22 36 06 78 25 43 73 18 39 62 27 66 13}    R
1趟归并：  22 36 06 78 25 43 18 73 39 62 27 66 13      R1
2趟归并：  06 22 36 78 18 25 43 73 27 39 62 66 13      R
3趟归并：  06 18 22 25 36 43 73 78 13 27 39 62 66      R1
4趟归并：  06 13 18 22 25 27 36 39 43 62 66 73 78      R
```

无重复的归并

```
          {22 36 06 78 25 43 73 18}    R
1趟归并：  22 36 06 78 25 43 18 73      R1
2趟归并：  06 22 36 78 18 25 43 73      R
3趟归并：  06 18 22 25 36 43 73 78      R1   (至此已可结束)
4趟归并：  06 18 22 25 36 43 73 78      R    (重复的归并)
```

有重复的归并

§10. 内部排序

10.5. 归并排序

10.5.1. 二路归并排序

★ 效率分析

空间：大小为n的辅助数组

时间：每趟最多比较n次，移动n次　　$O(n\log_2 n)$

共 $\lceil\log_2 n\rceil$ 趟