实用算法设计——排序

主讲：娄文启

louwenqi@ustc.edu.cn

6 排序

6.1 简单排序

冒泡排序、选择排序、插入排序、希尔排序

6.2 复杂排序

快速排序、堆排序、归并排序、分配排序

6.3 案例分析：（查找、排序）

什么是排序？

- 对一组记录（或指向记录的指针）进行排序。
- 本质：基于每个记录的**KEY**对记录进行排序。
- 目的：便于后续的更快捷的检索/查找。
- 比如，客户的消费信息=（客户编号，交易日期，购买商品的列表，……）
  - 按交易日期对每个客户的消费信息进行排序；
  - 按客户编号对每个客户的消费信息进行排序；
- 再比如，电话簿、病历、档案室中的档案、图书馆和各种词典的目录表等，几乎都需要对有序数据进行操作

排序的广泛应用

- 对800万个7位正整数（每个整数取值不超过8百万）进行排序；
- 有一个1G大小的一个文件，文件中每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词；
- 从海量日志数据中提取出某日访问百度次数最多的那个IP；
- 从100万个数中找出最大的100个数 ；
- 对于上亿个数据（有重复），统计其中出现次数最多的前N个数据。

- 关键问题：
  1、存储什么？如何存储一组数据；（选择数据结构：逻辑和物理）
  2、如何排序？（选择排序策略）

排序的基本特征

- 稳定性:
  - 稳定的排序：维持要排序的记录中预先存在的顺序
  - 例：4条记录的KEY的初始状态为：序列B1 A1 B2 A2（基于其数字部分是有序的）。要求：基于其字母部分对序列进行排序。
  - 若得到的序列为A1 A2 B1 B2，则称为稳定的排序；否则为不稳定的排序。
- 对哨兵的需求：
  - 通常不考虑需要哨兵的排序：确定哨兵的KEY不方便，甚至不可能。
- 对链表进行排序的能力：
  - 通常的排序算法都设计为用于数组
  - 仅部分排序算法可以很容易的修改为基于链表的排序算法
  - 教材上只考虑两种非常适用于链表的排序方法：插入排序和快速排序
- 对待排序数据初始特征的依赖性：待排序数据可能为三类
  - **Case1**:正序；
  - **Case2**:逆序；
  - **Case3**:随机；
- 对额外存储空间的需求
  - 需额外的临时空间来存储一个或更多待排序的记录；
  - 隐式的额外空间需求：递归方式实现的排序算法对栈空间的需求

- 内部排序技术与外部排序技术：
  - 内部排序：把待排序的所有记录都加载进内存，然后对内存中的这些记录进行排序；（课程重点）
  - 外部排序：待排序的记录无法一次加载到内存中，因此每次排序时只有部分记录加载到内存中并进行内部排序，即待排序的一部分记录仍驻留在大容量的存储设备（硬盘等）
    - 比如，归并排序：将多个有序的子文件合成一个有序的大文件。

内部排序算法的分类

- 基于比较操作的排序算法：**平均时**间复杂度≥$O(N\log_2 N)$
  - 交换排序：**两两比较待排序记录的关键字，如果逆序就进行交换，直到所有记录都排好序为止。**
    - 冒泡排序
    - **快速排序（分区交换排序）**
  - 选择排序（Selection Sort）的基本思想是：不断从待排记录序列中选出关键字最小的记录放到已排序记录序列的后面，直到n个记录全部插入已排序记录序列中。
    - 简单选择排序
    - **堆排序（树型选择排序）**
  - 插入排序：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子表中的适当位置，直到全部记录插入完成为止。
    - 直接插入排序
    - 折半插入排序
    - 希尔排序
- 基于某种映射函数的排序（分配排序）：平均时间复杂度为线性级别
  - 桶排序：将待排序列的关键字k映射到第i个桶中。(T(N)=O(N))
  - 基数排序：将待排数据中的每组关键字依次进行桶分配
    - 比如，将每个数值的个位，十位，百位分成三个关键字

"分配"和"收集"

如何解决排序相关的问题？

- 直接利用现有的库函数和数据结构：
  - C中库函数:qsort()
  - C++中库函数:sort()
  - C++：STL中的SET模板（《编程珠玑》习题1.1的答案）
- 根据应用场景，选择合适的排序算法并实现：
  - 冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序、分配排序、基数排序
- 利用特定的数据结构：
  - 二叉搜索树：将待排序数据加载到二叉搜索树中，然后遍历输出结果。
  - B树：对海量数据进行排序
    - 将待排序数据的KEY加载到B树中，然后遍历输出结果。

6.1 简单排序——直接插入排序

- 算法的基本设计思想：
  - 逐个处理待排序的记录：将每个记录（目标记录）与前面已经排好序的记录序列进行比较，并将其插入到合适的位置。

表 5-3　插入排序

遍　历	一组有序的字母	一组无序的字母
0		C A B E D
1	C	A B E D
2	A C	B E D
3	A B C	E D
4	A B C E	D
5	A B C D E	

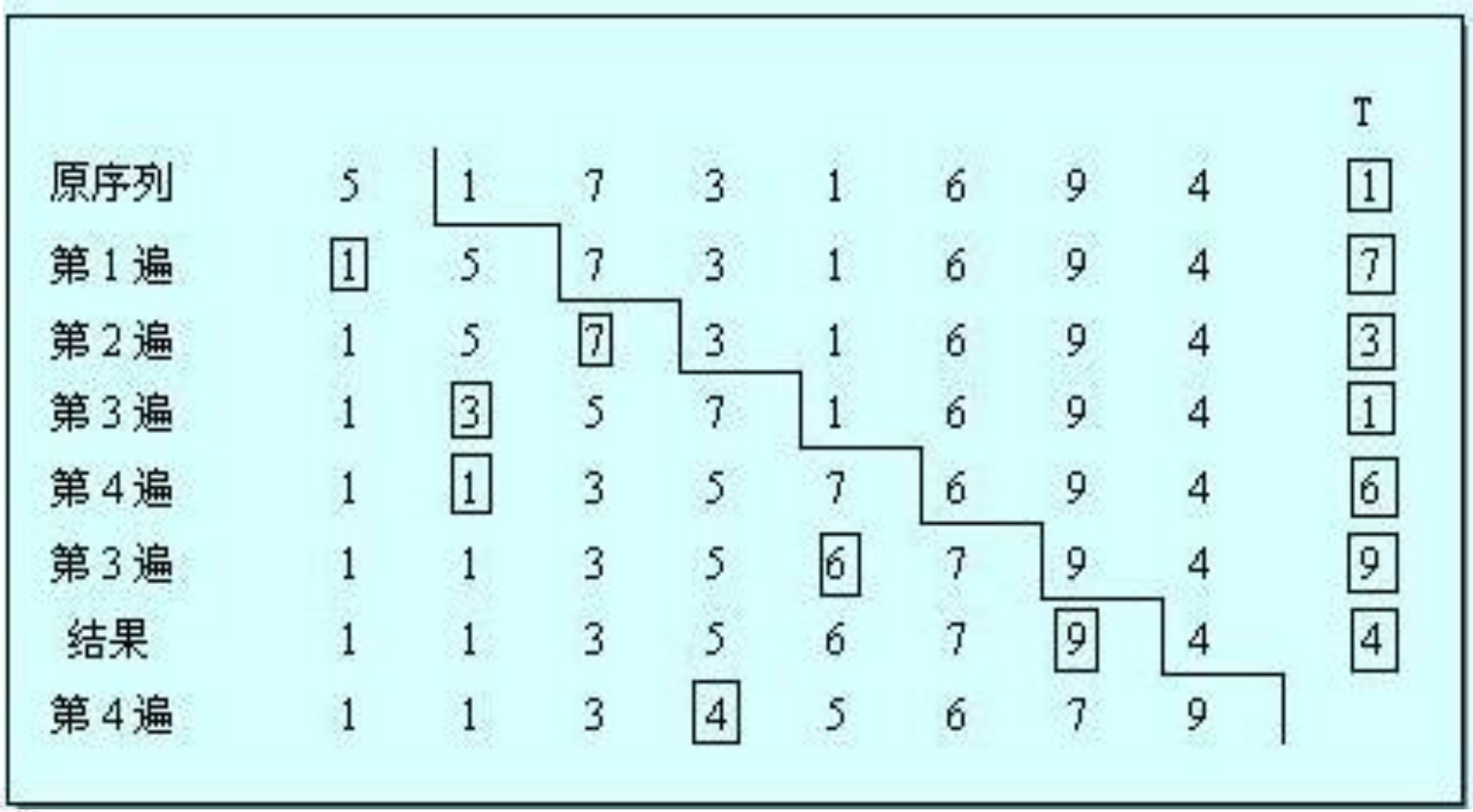
- 关键：

1）扫描已排好序的记录序列的方向：从前往后？从后往前？

2）比较，并依据比较结果进行相应处理。（假设采用从后往前的扫描）

  - 若目标记录<当前记录，则将当前记录向后移动一个；
  - 否则，则将目标记录插入到当前记录之后；

示例：（采用从后往前的扫描）



将待序列表分成左右两部分：左边为有序表（有序序列），右边为无序表（无序序列）。整个排序过程就是将右边无序表中的记录逐个插入到左边的有序表中，构成新的有序序列。

【例】假设有7个待排序的记录，它们的关键字分别为23，4，15，8，19，24，15，用直接插入法进行排序。

【解】直接插入排序过程如图1所示。方括号[ ]中为已排好序的记录的关键字，有两个记录的关键字都为15，为表示区别，将后一个15加下划线。

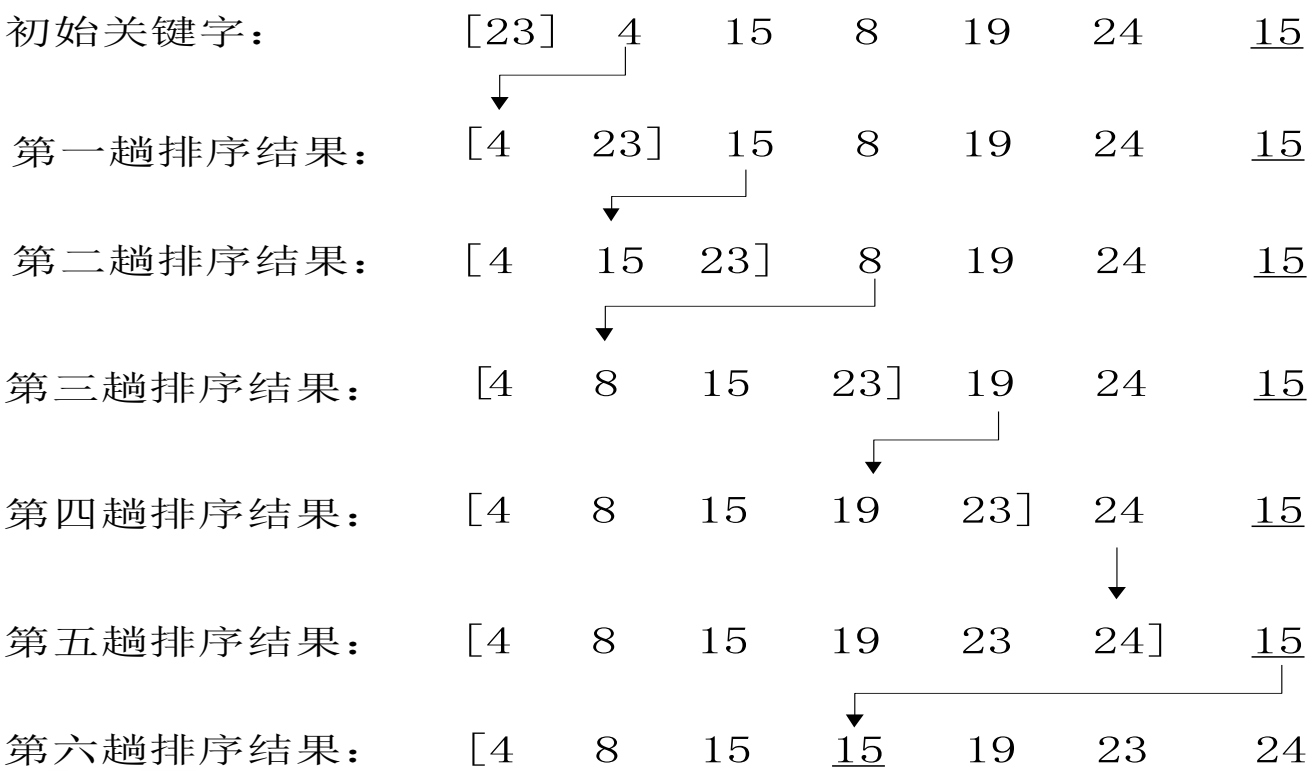


图1　直接插入排序

算法的实现1

```
void isort1()
{   int i, j;
    for (i = 1; i < n; i++)
        for (j = i; j > 0 && x[j-1] > x[j]; j--)
            swap(j-1, j);
}

void swap(int i, int j)
{   DType t = x[i];
    x[i] = x[j];
    x[j] = t;
}
```

算法的实现2

```
/* Write swap function inline */
void isort2()
{  int i, j;
   DType t;
   for (i = 1; i < n; i++)
        for (j = i; j > 0 && x[j-1] > x[j]; j--) {
              t = x[j];
              x[j] = x[j-1];
              x[j-1] = t;
        }
}
```

使用内联代码来替换函数！

算法的实现3

```
/* Move assignments to and from t out of loop */
void isort3()
{   int i, j;
    DType t;
    for (i = 1; i < n; i++) {
        t = x[i];
        for (j = i; j > 0 && x[j-1] > t; j--)
            x[j] = x[j-1];
        x[j] = t;
    }
}
```

1次移动操作算作1/3次交换.

使用“移位”操作替换“交换”操作！（“半交换”）

1）将目标记录复制到临时存储变量中；

2）在数组的有序部分中创建一个“空位”，并将这个空位向前移动，直至找到它的正确位置；

3）将目标记录插入到空位中。

3个直接插入排序程序的性能对比

- 对随机的n个整数进行排序：平均情况下 $T(n)=O(n^2)$

程序	C 代码行数	纳秒
插入排序 1（isort1）	3	$11.9\ n^2$
插入排序 2（isort2）	5	$3.8\ n^2$
插入排序 3（isort3）	5	$3.2\ n^2$

- 空间性能：该算法仅需要一个记录的辅助存储空间，空间复杂度为O(1)。

- 稳定性：由于该算法在搜索插入位置时遇到关键字值相等的记录时就停止操作，不会把关键字值相等的两个数据交换位置，所以该算法是**稳定**的。

- 时间性能：整个算法执行for循环n-1次，每次循环中的基本操作是比较和交换，其总次数取决于数据表的初始特性，可能有以下几种情况：
  - **最好情况：如果初始记录序列为“正序”序列，**则在每趟排序中关键字的比较次数为1，而交换次数为0。所以，整个排序过程中的比较次数和移动次数分别为(n-1)和0， 因而其时间复杂度为O(n).
  - **最坏情况：如果初始记录序列为“逆序”序列，则**在第i(1≤i≤n-1)趟排序时，关键字的比较次数为i，而交换次数为i/3。所以，整个排序过程中的比较次数和交换次数分别为：

$$\text{总比较次数} C_{\max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$\text{总交换次数} M_{\max} = \sum_{i=1}^{n-1} i/3 = \frac{n(n-1)}{6}$$

  - **平均情况：**可认为出现各种排列的概率相同，因此取上述两种情况的平均值，作为直接插入排序关键字的比较次数和记录交换次数，约为$n^2/4$。所以其时间复杂度为$O(n^2)$。
- 根据上述分析得知：当原始序列越接近有序时，该算法的执行效率就越高。

- 基于数组的直接插入排序算法中的基本操作：比较和“半交换”（移位）

- 优点：
  - 对几乎有序的记录排序时间开销为**O(n)**；
  - 可用于优化快速排序算法

6.1 简单排序——折半插入排序

- **是直接插入排序方法的改进版。**
- 直接插入排序的基本操作是在有序表中进行查找和插入，而在**有序表**中查找插入位置，可以通过**折半查找**的方法实现，由此进行的插入排序称之为折半插入排序。
- 折半查找只是减少了比较次数，但是元素的**移动次数**不变。

6.1 简单排序——希尔排序

- 是直接插入排序方法的改进版。
- 直接插入排序为何缓慢？
  - 每次扫描记录序列，只能定位一个目标记录的合法位置
- 算法的基本设计思想：先将整个待排记录序列分割成若干小组（子序列），分别在组内进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。**其中，每个组中的记录的位置相隔h，且h稳定递减。**
- 关键：**使用稳定递减的h序列，以保证h的最后一个取值是1。**
- 改进的出发点：直接插入排序对几乎有序的记录排序时间开销为**O(N)**
- **T(n)=O(n$^?$)**，复杂度与增量序列有关。**3/4 or 4/3**

示例：

表 5-4　希尔排序的实际应用

0：	待排序的项目	E	F	B	G	H	D	C	A
1：	将待排序的项目视作 4 对项目，每一对中包含两个项目				G				A
				B				C	
			F				D		
		E				H			
2：	对每一对项目进行排序，给出这种顺序	E	D	B	A	H	F	C	G
3：	现在把它们视作两组项目，每一组中包含 4 个项目		D		A		F		G
		E		B		H		C	
4：	对每一组进行排序，给出这种顺序	B	A	C	D	E	F	H	G
5：	最后，将 8 个项目作为一组进行排序	A	B	C	D	E	F	G	H

- 将记录A移动到正确的位置，只进行3次交换；
- h的取值依此为4,2,1；

- 算法的实现

```
void ShellSort ( Element **Array, int N, CompFunc Compare )
{
   int step, h;
   /* Find starting h */
   for ( h = 1; h <= N / 9; h = 3*h + 1 )
            ;
   /* Now loop thru successively smaller h's */
   for ( ; h > 0; h /= 3 )
   {   /* Look at hth thru Nth elements */
      for ( step = h; step < N; step++ )
      {   int i;
         Element *temp;
         /* Now, look to the left and find our spot */
         temp = Array[step];
         for ( i = step - h; i >= 0; i -= h )
         {  if ( Compare ( temp, Array[i] ) < 0 )
            {   /* Not there yet, so make room */
               Array[i + h] = Array[i];            }
            else /* Found it! */
               break;        }
         Array[i + h] = temp; /* Now insert original value from Array[step] */
      } }}
```

h=1,4,13,40,……

h=……,40,13,4, 1

算法分析

（1）算法中约定初始增量d_1为已知；

（2）算法中采用简单的取增量值的方法，从第二次起取增量值为其前次增量值的一半。在实际应用中，可能有多种取增量的方法，并且不同的取值方法对算法的时间性能有一定的影响，因而一种好的取增量的方法是改进希尔排序算法时间性能的关键。

（3）希尔排序开始时增量较大，分组较多，每组的记录数较少，故各组内直接插入过程较快。随着每一趟中增量d_i逐渐缩小，分组数逐渐减少，虽各组的记录数目逐渐增多，但由于已经按d_{i-1}作为增量排过序，使序列表较接近有序状态，所以新的一趟排序过程也较快。因此，希尔排序在效率上较直接插入排序有较大的改进。希尔排序的时间复杂度约为$O(n^{1.25})$，它实际所需的时间取决于各次排序时增量的取值。大量研究证明，若增量序列取值较合理，希尔排序时关键字比较次数和记录移动次数约为$O(n\log_2 n)^2$。由于其时间复杂度分析较复杂，在此不做讨论。

（4）希尔排序会使关键字相同的记录交换相对位置，所以希尔排序是**不稳定**的。

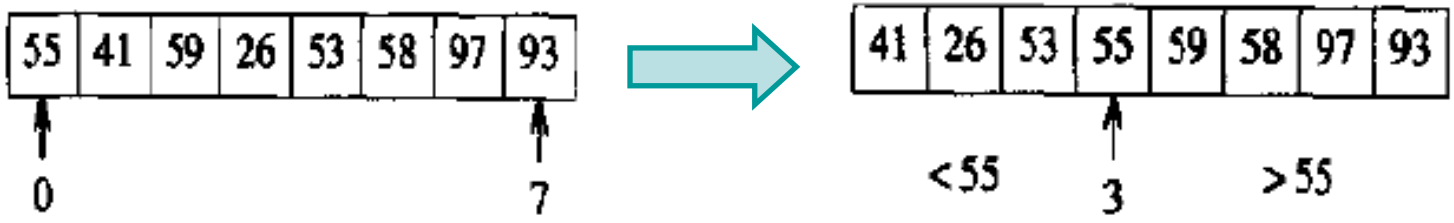
总结：简单排序算法

- 插入排序的原理：向有序序列中依次插入无序序列中待排序的记录，直到无序序列为空，对应的有序序列即为排序的结果，其主旨是“插入”。
- 交换排序的原理：先比较大小，如果逆序就进行交换，直到有序。其主旨是“若逆序就交换”。
- 选择排序的原理：先找关键字最小的记录，再放到已排好序的序列后面，依次选择，直到全部有序，其主旨是“选择”

排序方法	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
直接插入排序	O（n^2）	O（n^2）	O（n）	O(1)	稳定	简单
折半插入排序	O（n^2）	O（n^2）	O（nlogn）	O(1)	稳定	一般
希尔排序	O($n^{1.3}$)			O(1)	不稳定	较复杂
简单选择排序	O（n^2）	O（n^2）	O（n^2）	O(1)	不稳定	简单
冒泡排序	O（n^2）	O（n^2）	O（n）	O(1)	稳定	简单

6.2 复杂排序——快速排序

- 是最广泛使用的高级排序算法。
- **ANSI C库中的函数qsort()、C++中库函数sort()通常是基于快速排序。**
- 快速排序：分区交换排序
- 采用分治法：将待排序的数组分成两个小部分，然后分别对它们进行递归快速排序。



- 算法的基本设计思想：

1）若待排序的数组Array中只有一个元素，则退出；

2）否则，选择一个元素Array[i]作为基准；

3）将待排序的数组Array按照基准Array[i]划分为两个子数组A1和A2

  - A1中的元素都小于等于基准Array[i]
  - A2中的元素都大于等于基准Array[i];
  - 与基准Array[i]相等的元素既可以在A1中，也可以在A2中。
  - Array =A1∪A2 ∪{Array[i]}.

4）对子数组A1进行快速排序；

5）对子数组A2进行快速排序。

- 关键问题：
  1）如何将数组Array按照基准Array[i]划分为两个子数组A1和A2。
  2）选择哪个元素作为基准？
- 策略：
  - 方法1：单向划分 & 将数组中第一个元素作为基准。
  - 方法2：双向划分& 将数组中第一个元素作为基准。
  - 方法3：双向划分& 将随机选择的元素作为基准。
  - 方法4：利用迭代消除递归
  - 方法5：只对大的数组使用快速排序，而对小数组使用插入排序。

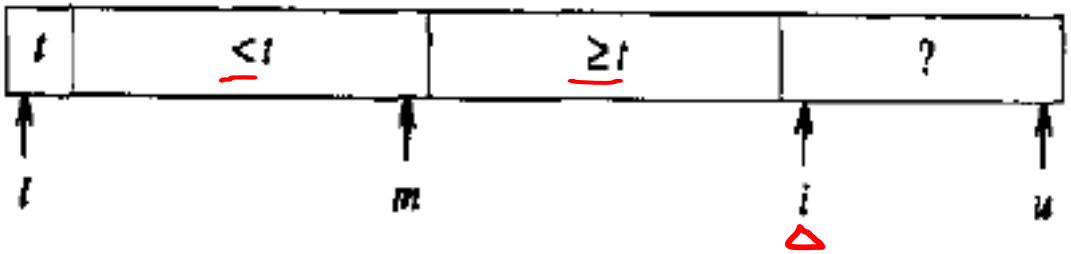
【例】假设有8个记录，关键字的初始序列为{45, 34, 67, 95, 78, 12, 26, 45}，用快速排序法进行排序.

一趟快速排序过程：

1.	45	34	67	95	78	12	26	45	(A1为空数组)
2.	45	34	67	95	78	12	26	45	(A1中有1元素)
3.	45	34	67	95	78	12	26	45	(A1中有1元素)
4.	45	34	67	95	78	12	26	45	(A1中有1元素)
5.	45	34	67	95	78	12	26	45	(A1中有1元素)
6.	45	34	12	95	78	67	26	45	(A1中有2元素)
7.	45	34	12	26	78	67	95	45	(A1中有3元素)
8.	45	34	12	26	78	67	95	45	(A1中有3元素)
9.	26	34	12	45	78	67	95	45	(将基准放中间)

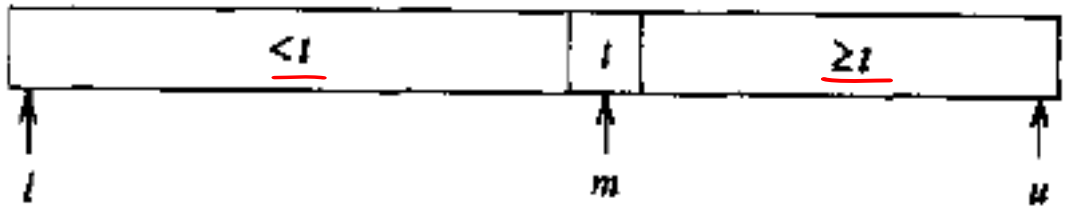
m：表示存放小于基准的元素的子数组A1的最末元素的下标。

循环过程中数组始终满足不变式：



循环终止时数组为：

子数组的划分结果为：



递归栈深度为7

【例】假设有8个记录，关键字的初始序列为{1, 2, 3, 4, 5, 6, 7, 8}，用快速排序法进行排序.

一趟快速排序过程：

1.	1	2	3	4	5	6	7	8	(A1为空数组)
2.	1	2	3	4	5	6	7	8	(A1为空数组)
3.	1	2	3	4	5	6	7	8	(A1为空数组)
4.	1	2	3	4	5	6	7	8	(A1为空数组)
5.	1	2	3	4	5	6	7	8	(A1为空数组)
6.	1	2	3	4	5	6	7	8	(A1为空数组)
7.	1	2	3	4	5	6	7	8	(A1为空数组)
8.	1	2	3	4	5	6	7	8	(A1为空数组)
9.	1	2	3	4	5	6	7	8	(将基准放中间)

m：表示存放小于基准的元素的子数组A1的最末元素的下标。

递归栈深度为7

【例】假设有8个记录，关键字的初始序列为{8, 7, 6, 5, 4, 3, 2, 1}，用快速排序法进行排序.

一趟快速排序过程：

1.	8	7	6	5	4	3	2	1	(A1为空数组)
2.	8	7	6	5	4	3	2	1	(A1中有1元素)
3.	8	7	6	5	4	3	2	1	(A1中有2元素)
4.	8	7	6	5	4	3	2	1	(A1中有3元素)
5.	8	7	6	5	4	3	2	1	(A1中有4元素)
6.	8	7	6	5	4	3	2	1	(A1中有5元素)
7.	8	7	6	5	4	3	2	1	(A1中有6元素)
8.	8	7	6	5	4	3	2	1	(A1中有7元素)
9.	1	7	6	5	4	3	2	8	(将基准放中间)

m:表示存放小于基准的元素的子数组A1的最末元素的下标。

示例

- 假设有8个记录，它们的关键字完全相同，即：关键字的初始序列为$\{8, 8^1, 8^2, 8^3, 8^4, 8^5, 8^6, 8^7\}$，用快速排序法进行排序.

算法的实现1

- 方法1：单向划分 & 将数组的首元素作为基准。

```
/* Simplest version, Lomuto partitioning */
void qsort1(int l, int u)
{   int i, m;
    if (l >= u)
        return;
    m = l;//m表示存放小于基准的元素的子数组A1的最末元素的下标。
    //A1中采用尾插法来添加新元素
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    qsort1(l, m-1);
    qsort1(m+1, u);
}
```

递归调用的终止条件。

完成一趟快排

算法的实现1的优化（《编程珠玑》习题11.2）

```
/* Sedgewick's version of Lomuto, with sentinel */
void qsort2(int l, int u)
{   int i, m;
    if (l >= u)
        return;
    m = i = u+1;// m表示存放大于基准的元素的子数组A2的开始元素
                   的下标， A2中采用头插法来添加新元素
    do {
        do i--; while (x[i] < x[l]);    //将x[l]作为哨兵
        swap(--m, i);
    } while (i > l);
    qsort2(l, m-1);
    qsort2(m+1, u);}
```

完成一趟快排

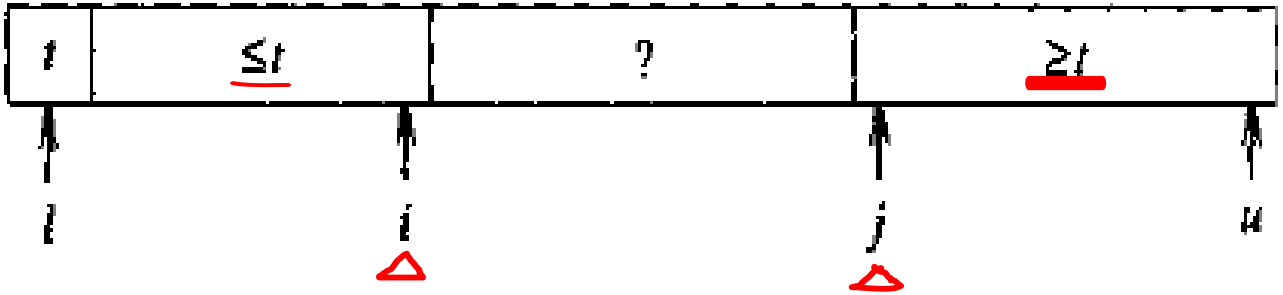
优化点：引入哨兵！

- 方法1的性能分析：
  - 对于随机顺序的待排序数组，$T(n)=O(n\log_2 n)$，栈深度为$O(\log_2 n)$
    - 简单推导：每层递归中都执行了$O(n)$次比较操作，而总计有$O(\log_2 n)$趟快排，即$O(\log_2 n)$次递归；
    - 精确的论证：$T(n)=2T(n/2)+O(n)$；
    - 大多数算法教材已证明了任何基于比较的排序至少需要$O(n\log_2 n)$次比较，因此快速排序接近最优排序；
  - 对于非随机的待排序数组，何时会出现最坏情况？
    - 比如，待排序数组由n个相同元素组成：插入排序的$T(n)=O(n)$，快速排序的$T(n)=O(n^2)$（此时快排的栈深度为n）
    - 比如，待排序数组有序（升序/降序）时，快速排序的$T(n)=O(n^2)$（此时快排的栈深度为n）

算法的实现2

- 方法2：双向划分& 将数组首元素作为基准。
- 主要设计思想：

循环过程中数组始终满足不变式：



i向右移过小元素，遇到大元素时停止；

j向左移过大元素，遇到小元素时停止；

若i,j对应的下标不交叉，则交换两个下标处对应的元素。

【例】假设有8个记录，关键字的初始序列为{8, 8^1, 8^2, 8^3, 8^4, 8^5, 8^6, 8^7}，用快速排序法进行排序.

i向右移过小元素；

j向左移过大元素；

若i, j不交叉，则交换；

直至i,j交叉。

一趟快速排序过程：

1.	8	8^1	8^2	8^3	8^4	8^5	8^6	8^7	(A1和A2为空数组)
2.	8	8^7	8^2	8^3	8^4	8^5	8^6	8^1	(A1和A2中各1元素)
3.	8	8^7	8^6	8^3	8^4	8^5	8^2	8^1	(A1和A2中各2元素)
4.	8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	(A1和A2中各3元素)
5.	8	8^7	8^6	8^5	8^4	8^3	8^2	8^1	
6.	8^5	8^7	8^6	8	8^4	8^3	8^2	8^1	(将基准放分界处)

递归栈深度为3！

i表示存放≤基准的元素的子数组A1的最末元素的下标；

j表示存放≥基准的元素的子数组A2的首元素的下标。

```
/* Two-way partitioning */
void qsort3(int l, int u)
{   int i, j;
    DType t;
    if (l >= u)
        return;
    t = x[l]; //基准
    i = l;
    j = u+1;
    for (;;) {
        do i++; while (i <= u && x[i] < t); //i向右移过小元素，遇到大元素停止；
        do j--; while (x[j] > t);  //j向左移过大元素，遇到小元素时停止；
        if (i > j)
            break;
        swap(i, j);
    }
    swap(l, j);
    qsort3(l, j-1);
    qsort3(j+1, u);
}
```

完成一趟快排

思考：为何判断条件不包括等号？

避免待排序数组中所有元素都相同时，时间复杂度降为$O(n^2)$。

- 方法2的性能分析：
  - 对于随机顺序的待排序数组，$T(n)=O(n\log_2 n)$，栈深度为$O(\log_2 n)$
    - 简单推导：每层递归中都执行了$O(n)$次操作，而总计有$O(\log_2 n)$次递归
    - 精确的论证：$T(n)=2T(n/2)+O(n)$
    - 大多数算法教材已证明了任何基于比较的排序至少需要$O(n\log_2 n)$次比较，因此快速排序接近最优排序；

有改进！对于非随机的待排序数组，

- 待排序数组由n个相同元素组成：插入排序的$T(n)=O(n)$，快速排序的$T(n)=O(n\log_2 n)$；
- 待排序数组有序（升序/降序）排列时：快速排序的$T(n)=O(n^2)$；

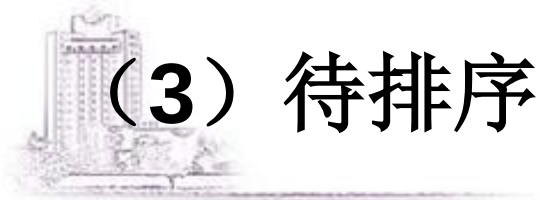
练习

- 请采用快速排序的方法2进行升序排序，并记录每次快排的比较次数和栈深度。

（1）待排序的序列为
(132,357,784,473,612,61,60,578,437,207)

（2）待排序的序列为(10,9,8,7,6,5,4,3,2,1)

（3）待排序的序列为(1,2,3,4,5,6,7,8,9,10)



算法的实现3

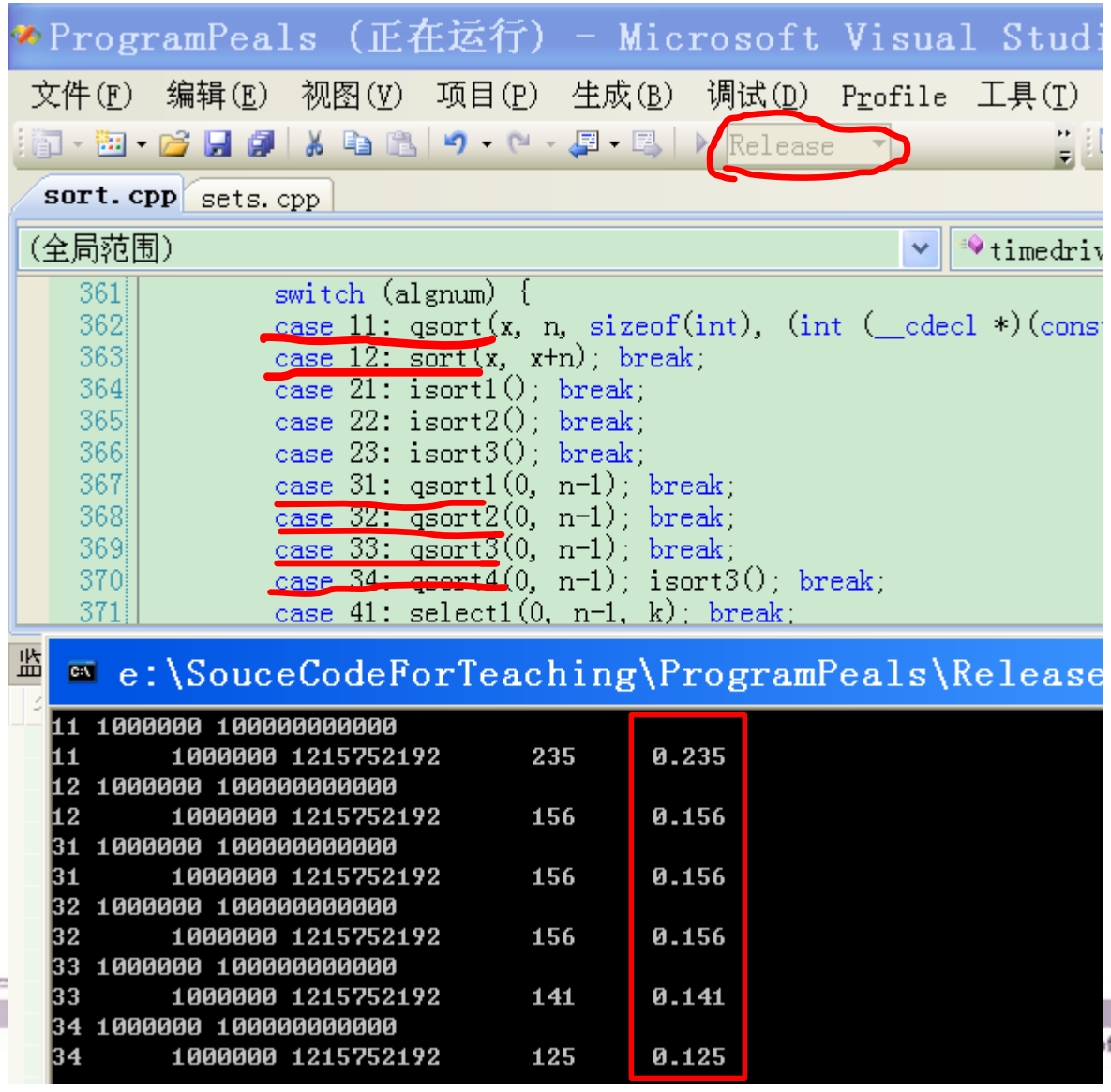
- 方法3：双向划分& 将随机选择的元素作为基准。

- 相比于方法2，改进点包括：
  1）将随机选择的元素作为基准；
  2）对于小的子数组，采用插入排序方法；
  3）用内联代码来替换函数调用：展开循环体中swap函数的代码。

```
/* qsort3 + randomization + isort small subarrays + swap inline */
int cutoff = 50;
void qsort4(int l, int u)
{   int i, j;
    DType t, temp;
    if (u - l < cutoff)
        return;
    swap(l, randint(l, u));
    t = x[l]; //基准
    i = l;
    j = u+1;
    for (;;) {
        do i++; while (i <= u && x[i] < t);
        do j--; while (x[j] > t);
        if (i > j) break;
        temp = x[i]; x[i] = x[j]; x[j] = temp;
    }
    swap(l, j);
    qsort4(l, j-1);
    qsort4(j+1, u);
}
```

完成一趟快排

性能对比：实际运行时间


ProgramPeals (正在运行) - Microsoft Visual Studi
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) Profile 工具(T)
Release
sort.cpp sets.cpp
(全局范围)
timedriv
361 switch (algnum) {
362 case 11: qsort(x, n, sizeof(int), (int (__cdecl *)(cons
363 case 12: sort(x, x+n); break;
364 case 21: isort1(); break;
365 case 22: isort2(); break;
366 case 23: isort3(); break;
367 case 31: qsort1(0, n-1); break;
368 case 32: qsort2(0, n-1); break;
369 case 33: qsort3(0, n-1); break;
370 case 34: qsort4(0, n-1); isort3(); break;
371 case 41: select1(0, n-1, k); break;
e:\SouceCodeForTeaching\ProgramPeals\Release
11 1000000 100000000000
11 1000000 1215752192 235 0.235
12 1000000 100000000000
12 1000000 1215752192 156 0.156
31 1000000 100000000000
31 1000000 1215752192 156 0.156
32 1000000 100000000000
32 1000000 1215752192 156 0.156
33 1000000 100000000000
33 1000000 1215752192 141 0.141
34 1000000 100000000000
34 1000000 1215752192 125 0.125

总结：性能对比

程序	代码行数	纳秒
C 库 qsort	3	$137n\log_2 n$
快速排序 1（qsort1）	9	$60n\log_2 n$
快速排序 2（qsort2）	9	$56n\log_2 n$
快速排序 3（qsort3）	14	$44n\log_2 n$
快速排序 4（qsort4）	15+5	$36n\log_2 n$
C++库 sort	1	$30n\log_2 n$

其他的提速方法

1）改进栈利用率：

- 对于大的子数组的排序利用迭代循环实现，以消除递归调用快排函数；
- 对于小的子数组的排序仍利用递归调用快排函数；

2）利用了static类型的变量，以减少栈空间的需求。

- 因为若没有使用static类型的变量，则递归调用时需传递这些地址，从而会减慢执行速度。
- 但是，这样处理，如果在多线程的环境中，不同的任务同时调用该函数将会产生相互干扰。

3）不使用显式的数组索引，而是直接使用等价的指针，以保证快速排序的稳定性。

总结：快速排序 vs. 插入排序

- 若系统自带的排序函数（比如**qsort(), sort()**），能满足需求，则无需自己编写代码
- 若待排序的数组所包含的元素个数**n**较小时，则可以考虑直接用插入排序；
- 若待排序的数组所包含的元素个数**n**较大时，则可以考虑使用精心编码实现的快速排序；

6.2 复杂排序——堆排序

- 堆排序：树型选择排序：
- 特点：
  - T(n)=O(nlog$_2$n)，实际运行时间比快速排序慢些；
  - 是一种良好的通用排序算法，不存在最坏情况。
    - 希尔排序也是一种良好的通用排序算法，T(n)=O(n$^{1.25}$)
- 数据结构：堆——实质上是一棵二叉树；
- 基本操作：
  - **siftup**：当在堆的尾部插入新元素后，需重新获取堆性质。
  - **siftdown**：用新元素替换堆中的旧根后，需重新获取堆性质。

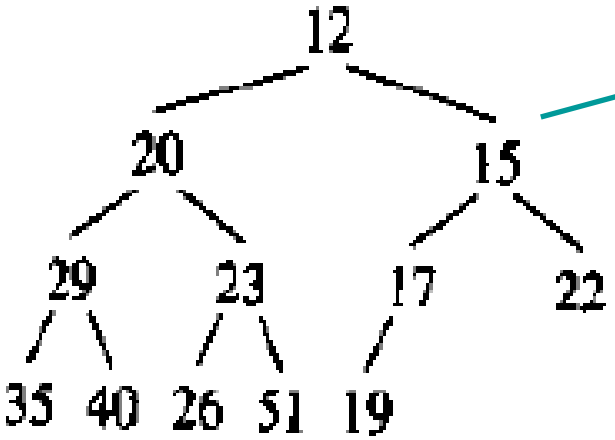
什么是堆？

- 堆是具有如下性质的二叉树：
  - 1）任何节点的值都≤（≥）其子节点的值，即堆的根节点中存放最小（大）元素；
  - 2）最多在两层上具有叶子节点，其中最底层的叶子节点尽可能的靠左分布。隐含：含有n个节点的堆，所有节点到根节点的距离都不超过$\log_2 n$。（类似 完全二叉树）

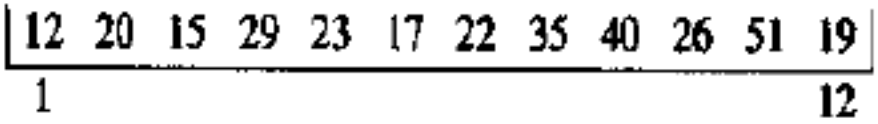
（逻辑数据结构）

堆的示例

- 由12个整数构成的（小根）堆：

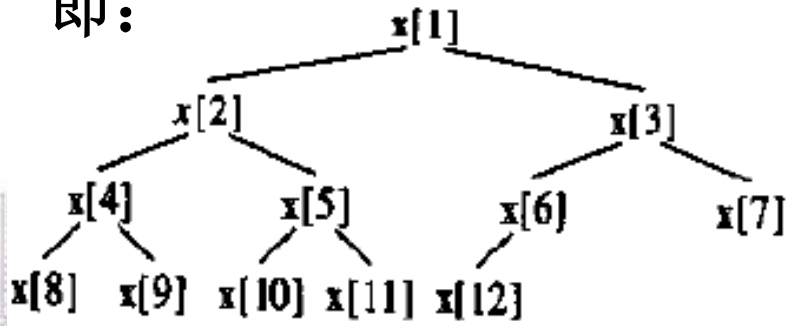


heap(1,12)为真



（采取顺序存储结构：利用数组）

即：

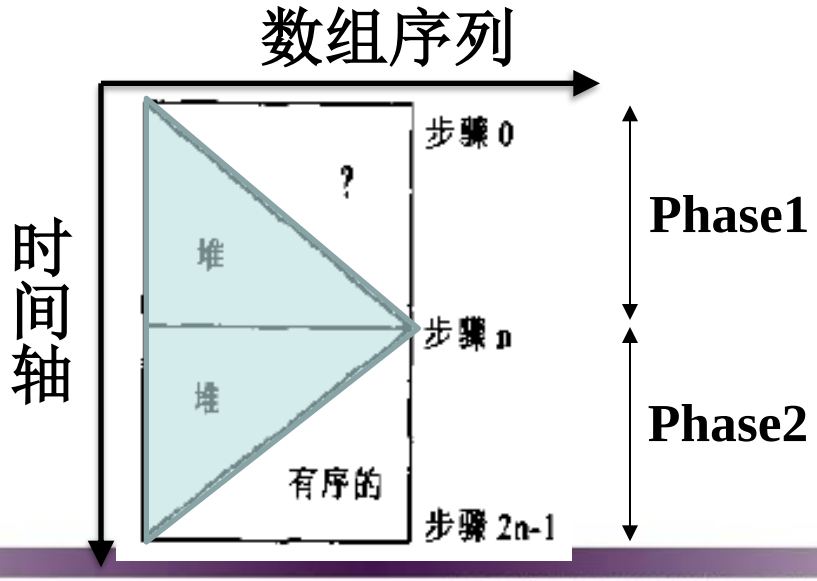


Q：下标为n/2的节点是？

A：最接近尾部的非叶子节点

如何使用堆进行排序？

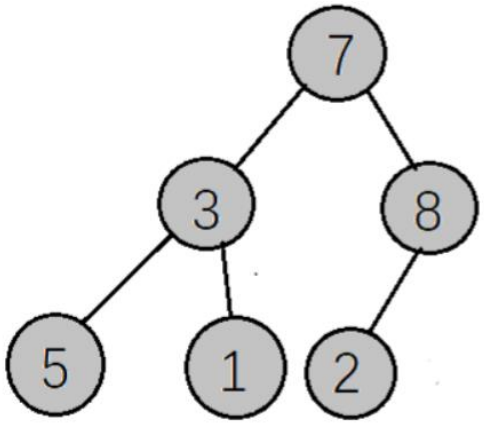
- 两阶段的过程：
  - **Phase1**：基于数组序列，建立大根堆；（堆中的根节点存放数组的最大值）
  - **Phase2**：依次提取大根堆的根节点，将其与末尾元素进行交换，将剩余**n-1**个元素重新构造成一个堆……重复此过程



Phase1——堆构建

- 方法一
  - 首先将数组从上至下按顺序排列，转换成二叉树：一个无序堆。
  - 从第一个非叶子节点开始，向上维护堆属性。

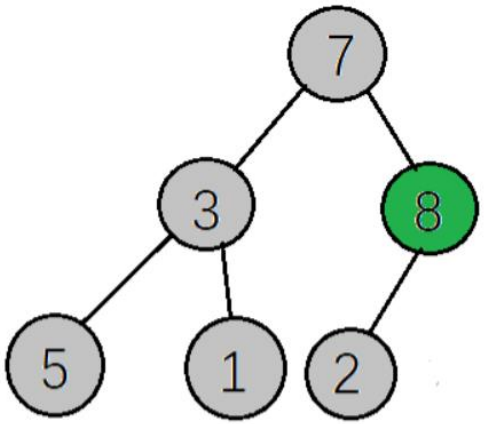
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase1——堆构建

- 方法一
  - 首先将数组从上至下按顺序排列，转换成二叉树：一个无序堆。
  - 从第一个非叶子节点开始，向上维护堆属性。

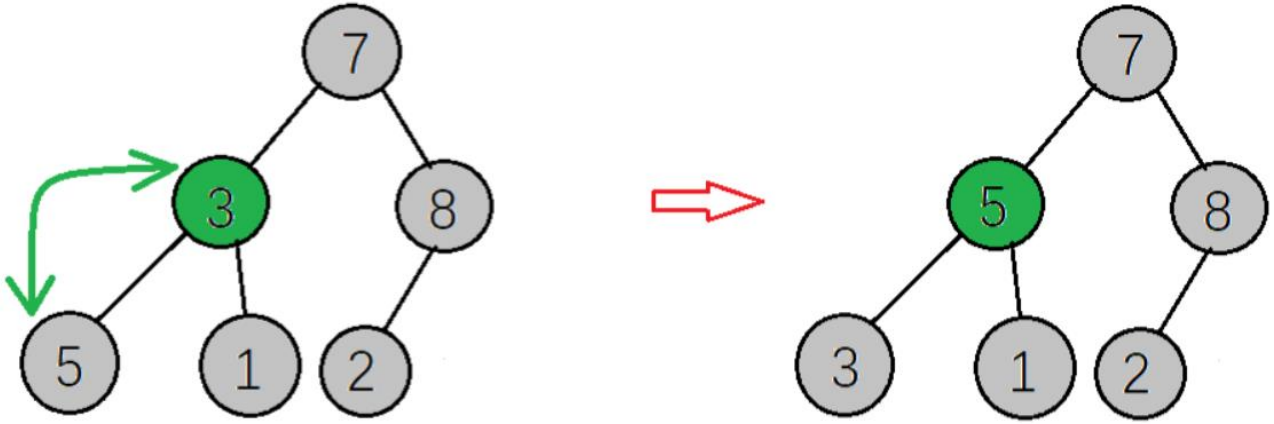
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase1——堆构建

- 方法一
  - 首先将数组从上至下按顺序排列，转换成二叉树：一个无序堆。
  - 从第一个非叶子节点开始，向上维护堆属性。

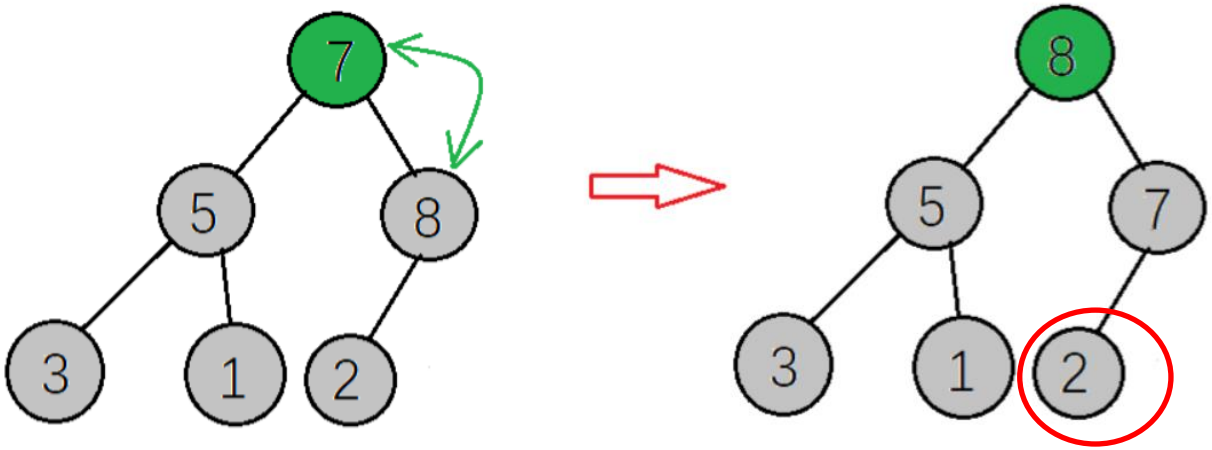
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase1——堆构建

- 方法一
  - 首先将数组从上至下按顺序排列，转换成二叉树：一个无序堆。
  - 从第一个非叶子节点开始，向上维护堆属性。

```
int a[6] = {7, 3, 8, 5, 1, 2};
```



如果不是2，是7.5呢？

Phase1——堆构建

- 方法一

```
void heap_init(int arr[], int len) {
    int i;
    //初始化，i从最后一个父节点开始调整，i从0开始
    for (i = len / 2 - 1; i >= 0; i--)
        max_heapify(arr, i, len - 1);
}
```

```
void max_heapify(int arr[], int start, int end) {
    //建立父节点指标和子节点指标
    int dad = start;
    int son = dad * 2 + 1;
    while (son <= end) { //若子节点指标在范围内才做比较
        if (son + 1 <= end && arr[son] < arr[son + 1]) //选择大子节点
            son++;
        if (arr[dad] > arr[son]) //如果父节点大于子节点代表调整完毕，直接跳出函数
            return;
        else { //否则交换父子内容再继续子节点和孙节点比较
            swap(&arr[dad], &arr[son]);
            dad = son;
            son = dad * 2 + 1;
        }}}
```

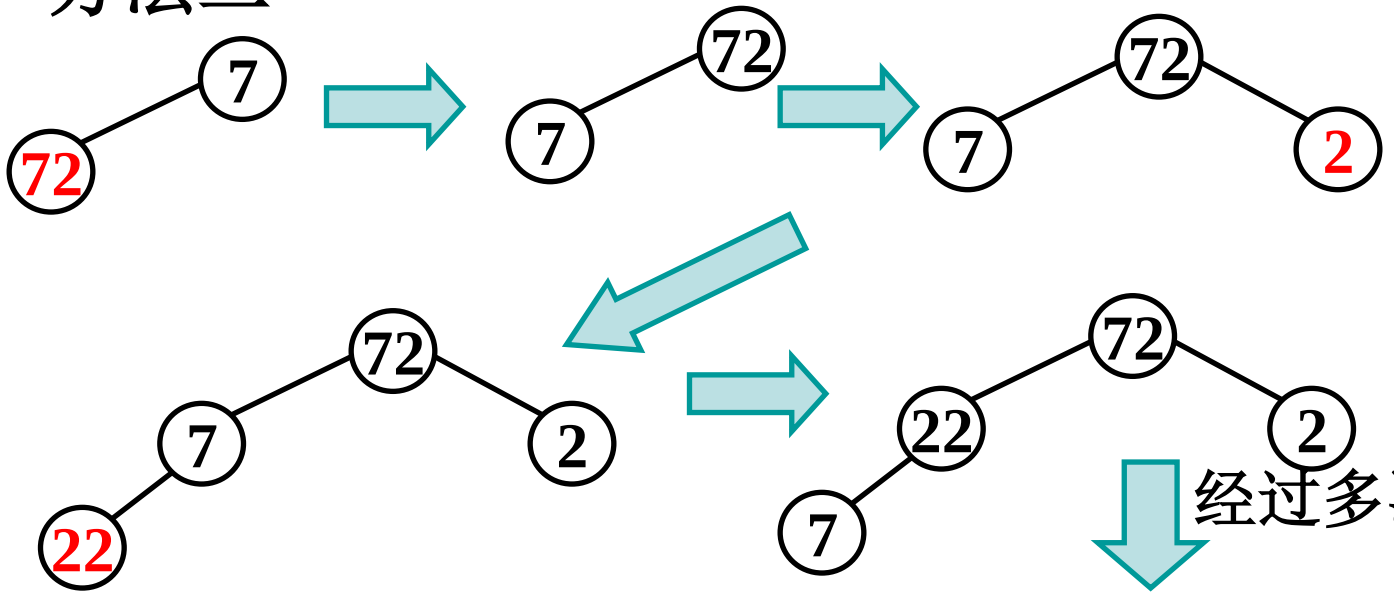
Phase1——堆构建

- 方法二
  - 初始化一个两结点堆。
  - 将剩余元素逐个插如：先插入到叶结点，再根据堆的性质上移。

待排序的序列为{7,72,2,22,36,87,59,40,3,79}，采用siftup方法构造初始堆。

Phase1——堆构建

- 方法二



待排序的序列为{7,72,2,22,36,87,59,40,3,79}，采用siftup方法构造初始堆。

Phase1——堆构建

- 方法二

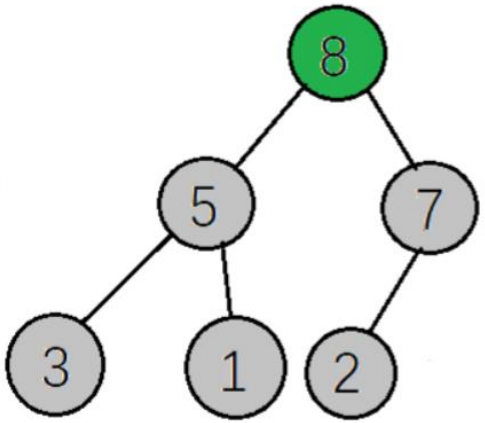
```
void heap_init()
{
    int i;
    for (i = 2; i <= n; i++)
        siftup(i);
}
```

```
void siftup(int u)
{
    int i, p;
    i = u;
    for (;;) {
        if (i == 1) break;
        p = i / 2;      // p为当前结点的父节点
        if (x[p] >= x[i]) break;
        swap(p, i);
        i = p;
    }
}
```

Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新：

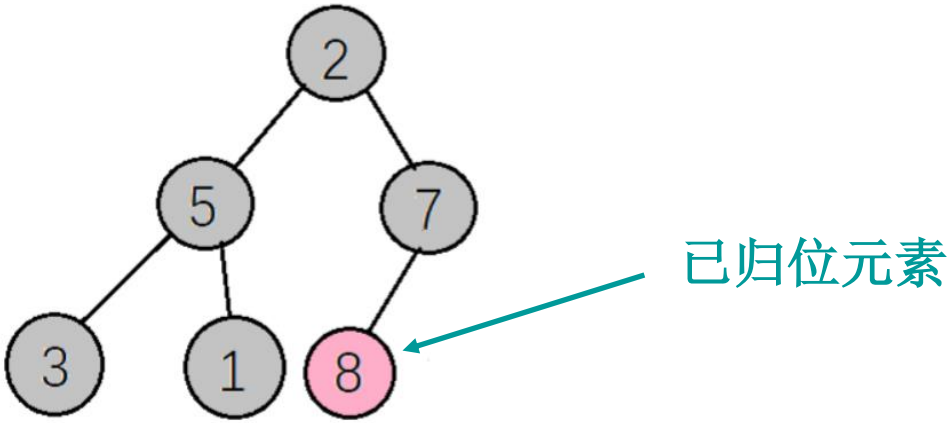
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新

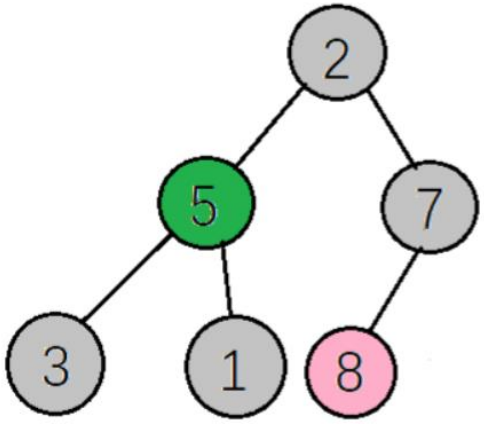
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新

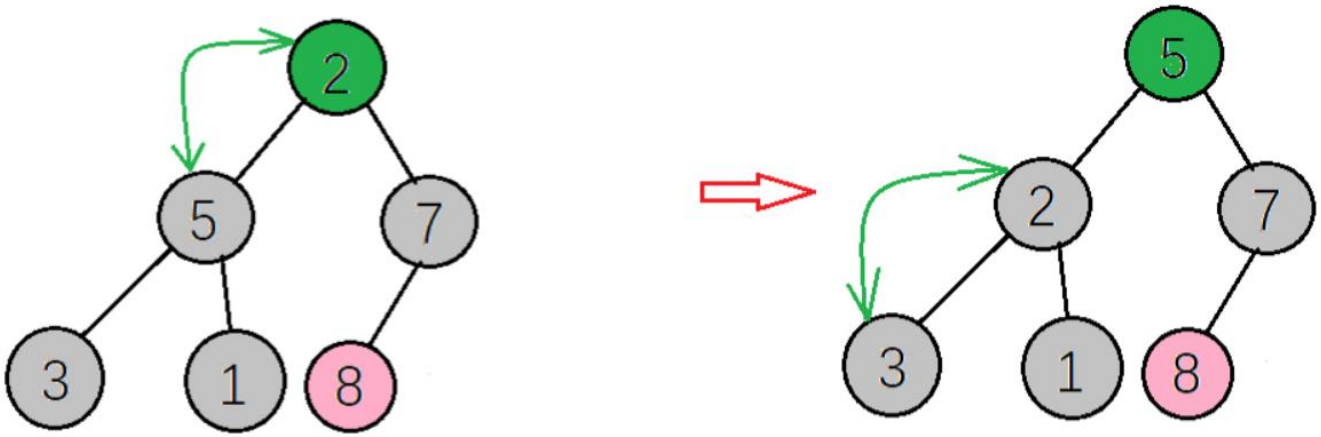
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新

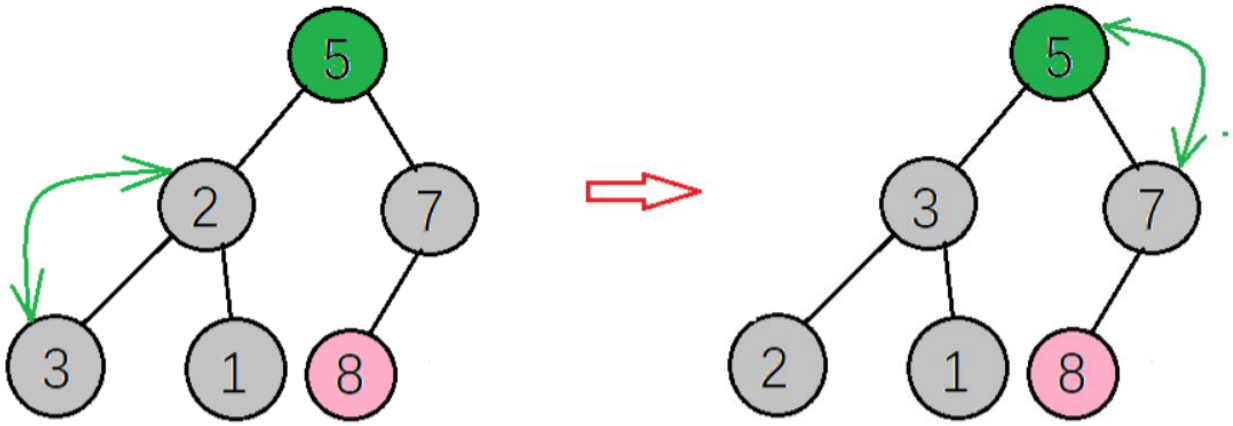
```
int a[6] = {7, 3, 8, 5, 1,
```



Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新

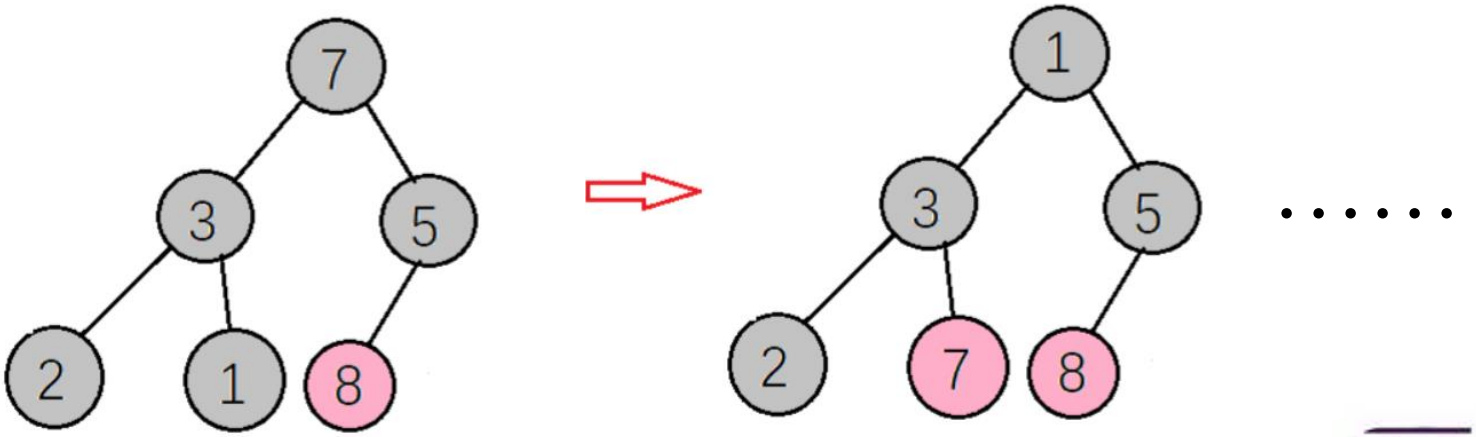
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



Phase2——堆排序

- 排序的过程就是提取根节点后不断维护最大堆的过程
- 假设现在已经构建了最大堆，若从非叶结点开始更新

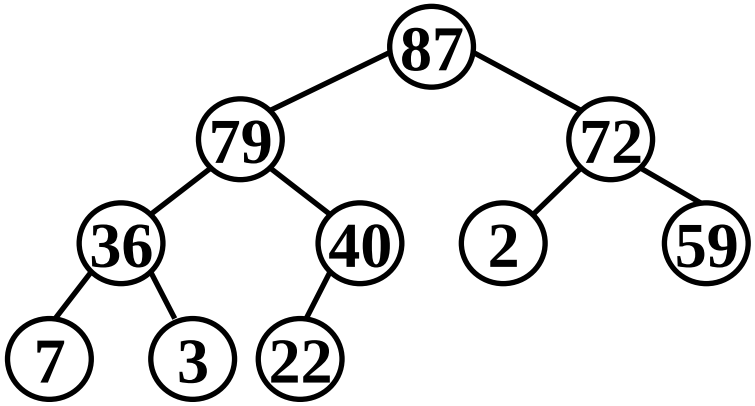
```
int a[6] = {7, 3, 8, 5, 1, 2};
```



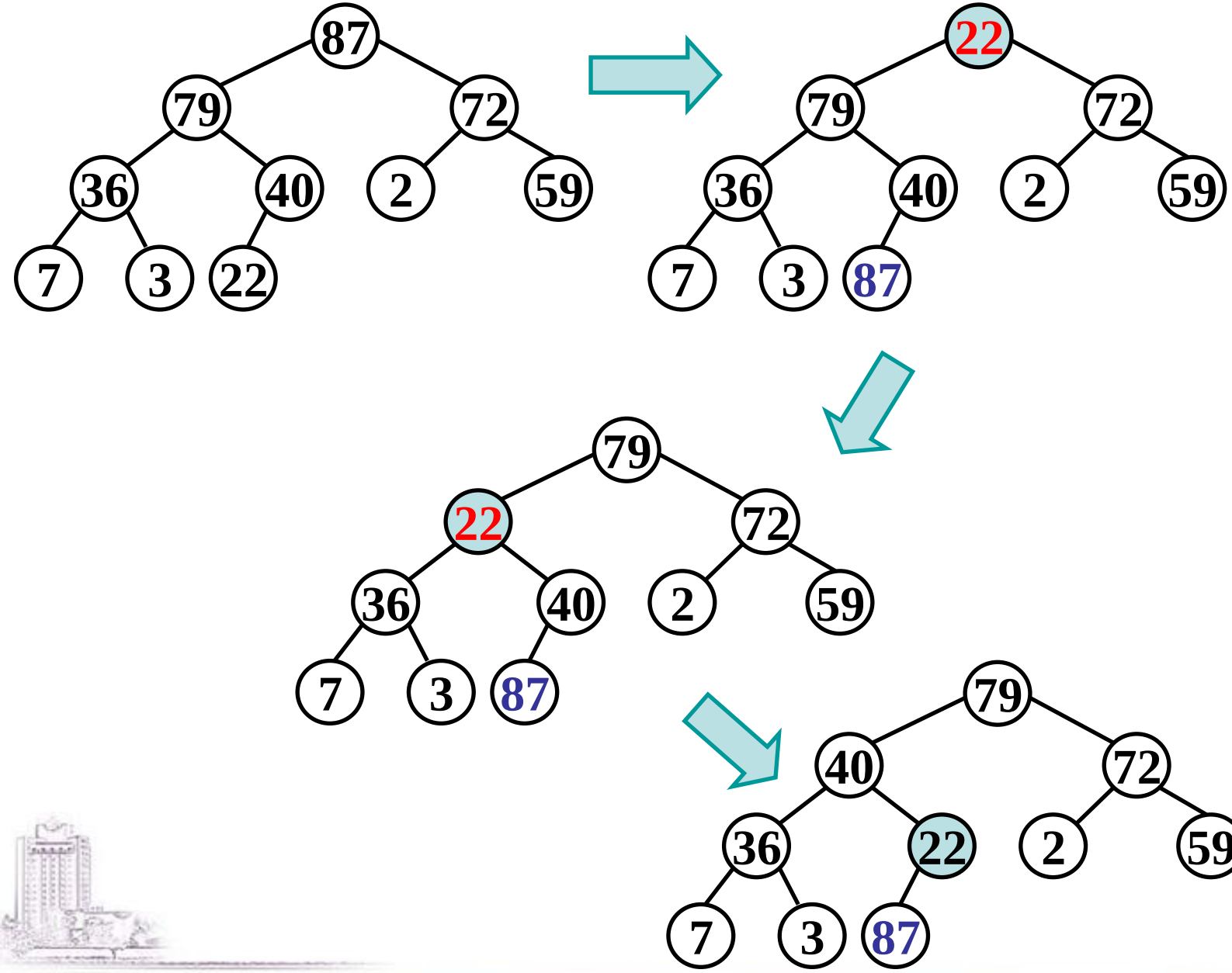
Phase2——堆排序

- 若从根节点向下更新：
- 前提是除根节点以外，其余结点保持最大（小）堆性质

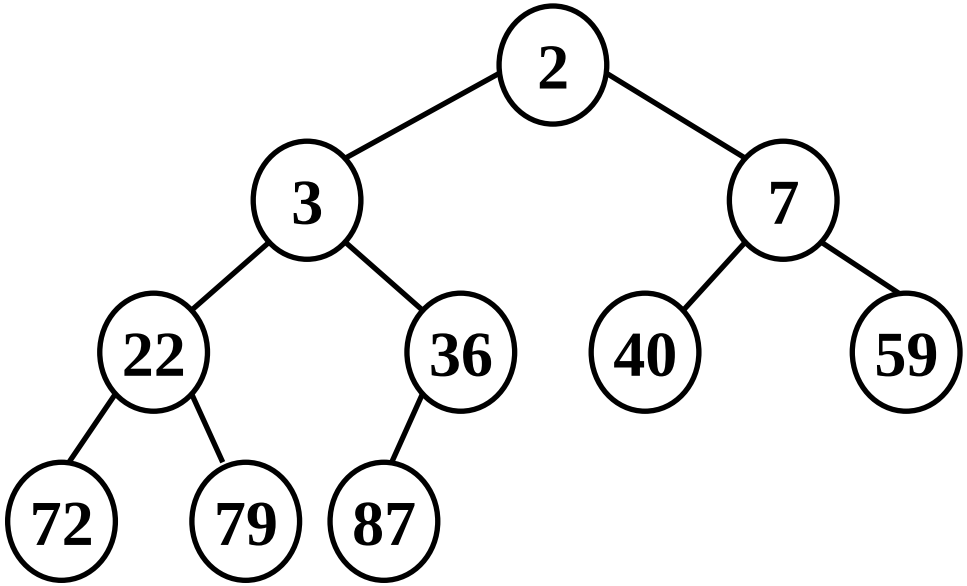
{2,3,7,22,36,40,59,72,79,87}



{2,3,7,22,36,40,59,72,79,87}



- 经过9次提取根节点，并向下筛选重新调整新根，可得到最终的有序序列为：
  {2,3,7,22,36,40,59,72,79,87}



堆排序的实现

```
void hsort()
{
    heap_init()
    for (i = n; i >= 2; i--) {
        swap(1, i);
        siftdown1(1, i-1);
    }
}
```

```
void siftdown(int l, int u)
{    int i, c;
     i = l;
     for (;;) {
         c = 2*i;    // c是左孩子结点
         if (c > u)  break;   //已是叶节点
         if (c+1 <= u && x[c+1] > x[c])
                 c++;
         // c是最大的孩子结点
         if (x[i] > x[c])        break;
         swap(i, c);
         i = c;
      }
}
```

边栏：STL中的priqueue类

- Insert操作和extraction操作在最坏情况下的时间开销时$O(\log_2 n)$；
- 需要额外存储(n+1)个元素

```
template<class T>
void pqsort(T v[], int n)
{   priqueue<T> pq(n);
    int i;
    for (i = 0; i < n; i++)
        pq.insert(v[i]);
    for (i = 0; i < n; i++)
        v[i] = pq.extractmin();
}
```