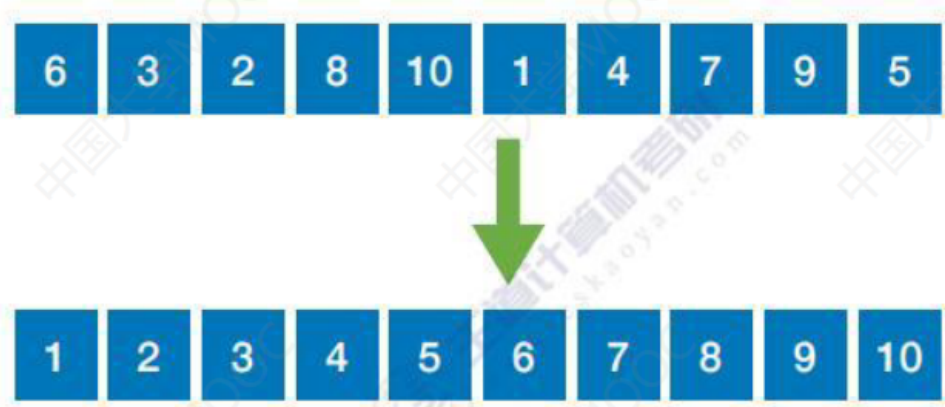
第八章 排序

8.1 排序的基本概念

8.1.1 排序的定义

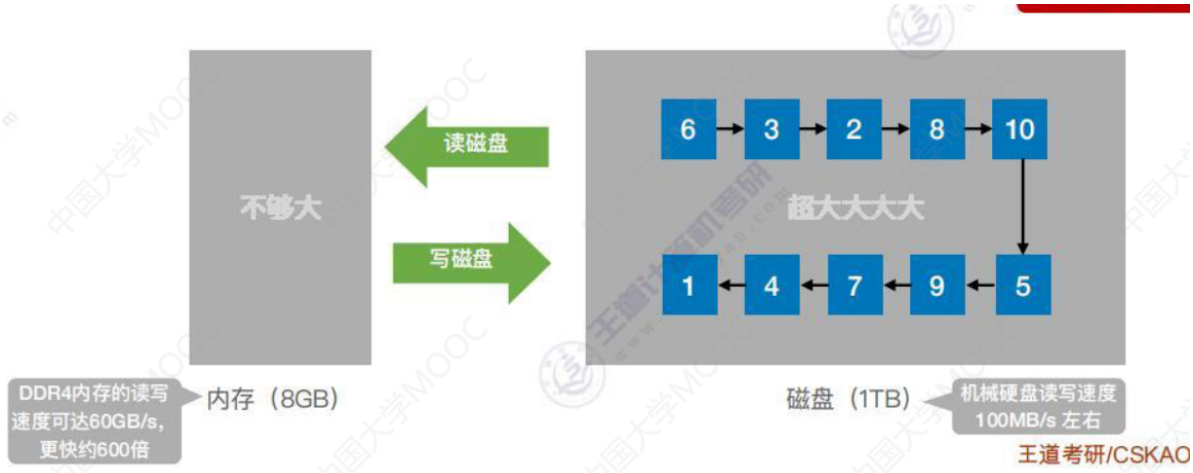
排序(sort)，就是重新排列表中的元素，使表中的元素满足按关键字有序的过程。



排序算法的评价指标、时间复杂度、空间复杂度、**算法的稳定性**

排序算法的分类：

- 内部排序：数据都在内存中，重点关注如何使算法的时间复杂度和空间复杂度更低。
- 外部排序：数据太多，无法全部放入内存中，所以还要关注如何使磁盘的读/写次数更少。



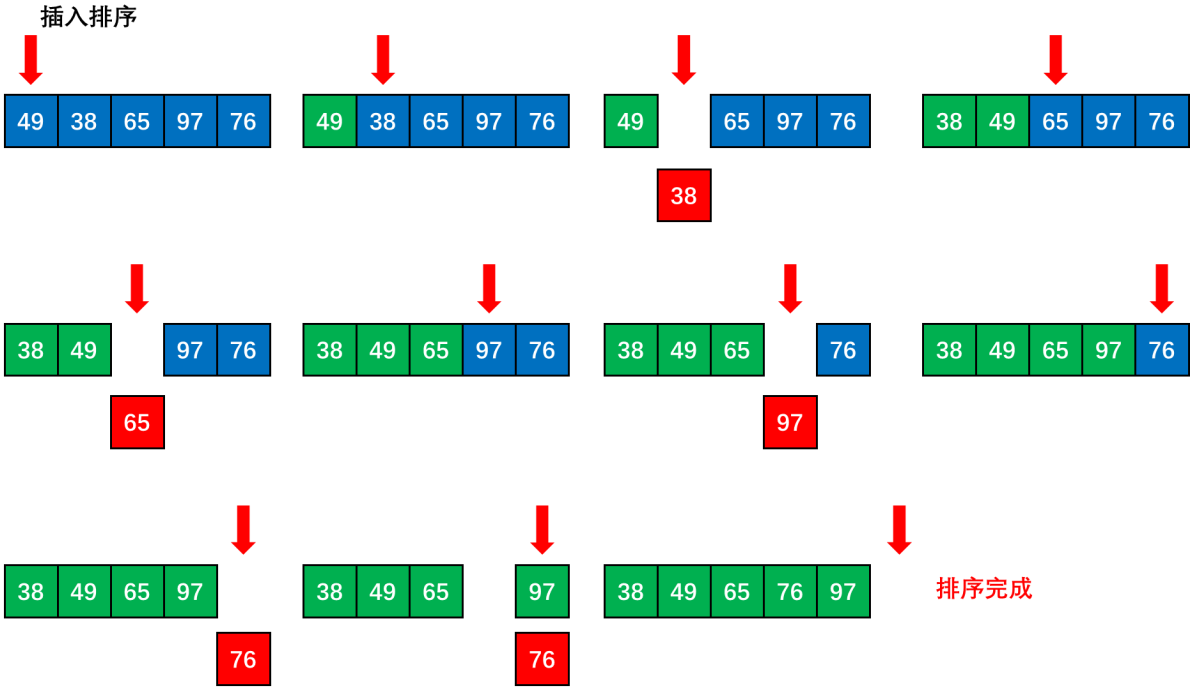
8.2 插入排序

8.2.1 直接插入排序

1、算法思想

每次将一个待排序的记录按照其关键字大小插入到前面已经排好序的子序列中，直到全部记录插入完成。

演示如下：



2、代码

不带哨兵：

```
void InsertSort(int A[], int n) {
    int i, j, temp;
    for (i = 1; i < n; i++) {        // 将各个元素插入到已排好序的序列中
        if (A[i] < A[i - 1]) {       // 若A[i]关键字小于前驱
            temp = A[i];             // 用temp暂存A[i]
            for (j = i-1; j >= 0 && A[j] > temp; --j) {   // 检查所有前面排序好的元素
                A[j + 1] = A[j];     // 所有大于temp的元素都向后移动一位
            }
            A[j + 1] = temp;         // 复制到插入的位置
        }
    }
}
```

带哨兵：（不用每轮循环都判断j>=0）

```
void InsertSortWithGuard(int A[], int n) {
    int i, j;
    for (i = 2; i <= n; i++) {
        if (A[i] < A[i - 1]) {
            A[0] = A[i];         //复制为哨兵，A[0]不放元素
            for (j = i - 1; A[0] < A[j]; --j)
                A[j + 1] = A[j];
            A[j + 1] = A[0];
        }
    }
}
```

3、算法效率分析

- 空间复杂度：$O(1)$
- 时间复杂度：主要来自对比关键字、移动元素，若有n个元素，则需要n-1趟处理
  - 最好情况：原本就有序，共$n-1$趟处理，每一趟只需要对比关键字一次，不用移动元素。时间复杂度：$O(n)$
  - 最坏情况：原本为逆序，$n-1$ 趟处理，第 i 趟要移动 $i+1$ 次元素 $(i=1,2,3,\ldots,n)$。时间复杂度：$O(n^2)$
- 稳定性：移动时不会改变值相等元素的先后顺序，故插入排序是**稳定的。**

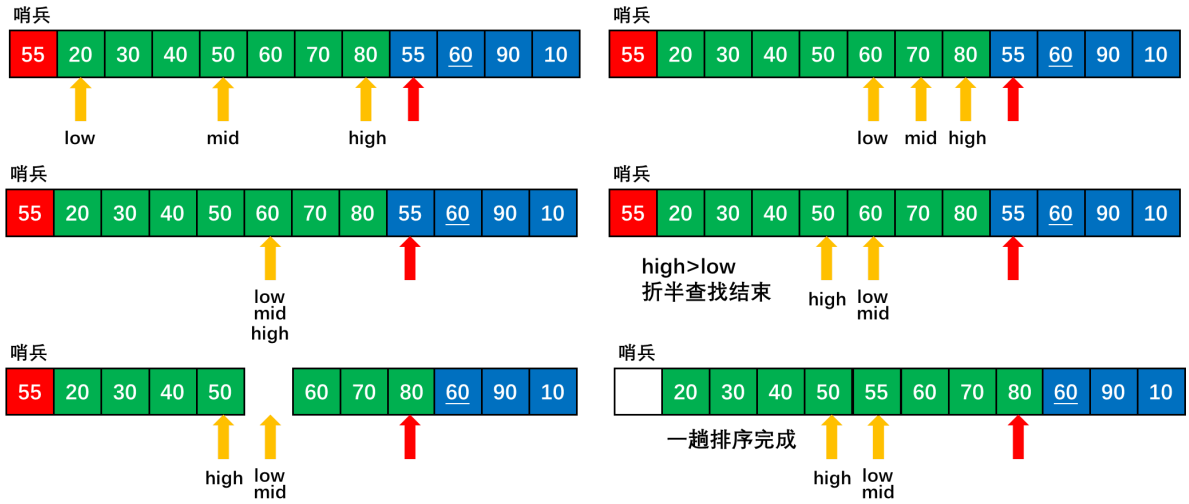
8.2.2 折半插入排序——直接插入排序的优化

1、算法思路

先用折半查找找到应该插入的位置，再移动元素。当low>high时，折半查找停止，将$[low,i-1]$内的元素全部右移，并将$A[0]$复制到low所指位置

> 当$A[mid]$==$A[0]$时，为了算法的稳定性，应该继续在mid所指位置右边寻找插入位置。

以下是一趟排序的示例：



2、代码实现

```
void BinaryInsertSort(int A[], int n) {
    int i, j, low, high, mid;
    for (i = 2; i <= n; i++) {  // 依次将A[2]~A[n]插入到前面的已排序序列
        A[0] = A[i];            // 将A[i]暂存到A[0] (哨兵)
        low = 1;                // 设置折半查找的范围（low为第一个元素）
        high = i - 1;           // 设置折半查找的范围 (high为待排序元素指针的前一位)
        while (low <= high) {    // 折半查找（默认递增有序）
            mid = (low + high) / 2;
            if (A[mid] > A[0]) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        for (j = i - 1; j >= high + 1; --j) {    // 统一右移元素，空出插入位置
            A[j + 1] = A[j];
        }
        A[high + 1] = A[0];          // 将元素插入，一趟排序完成
    }
}
```

与直接插入排序相比，比较关键字的次数减少了，但是移动元素的次数没有变。时间复杂度仍为 $O(n^2)$

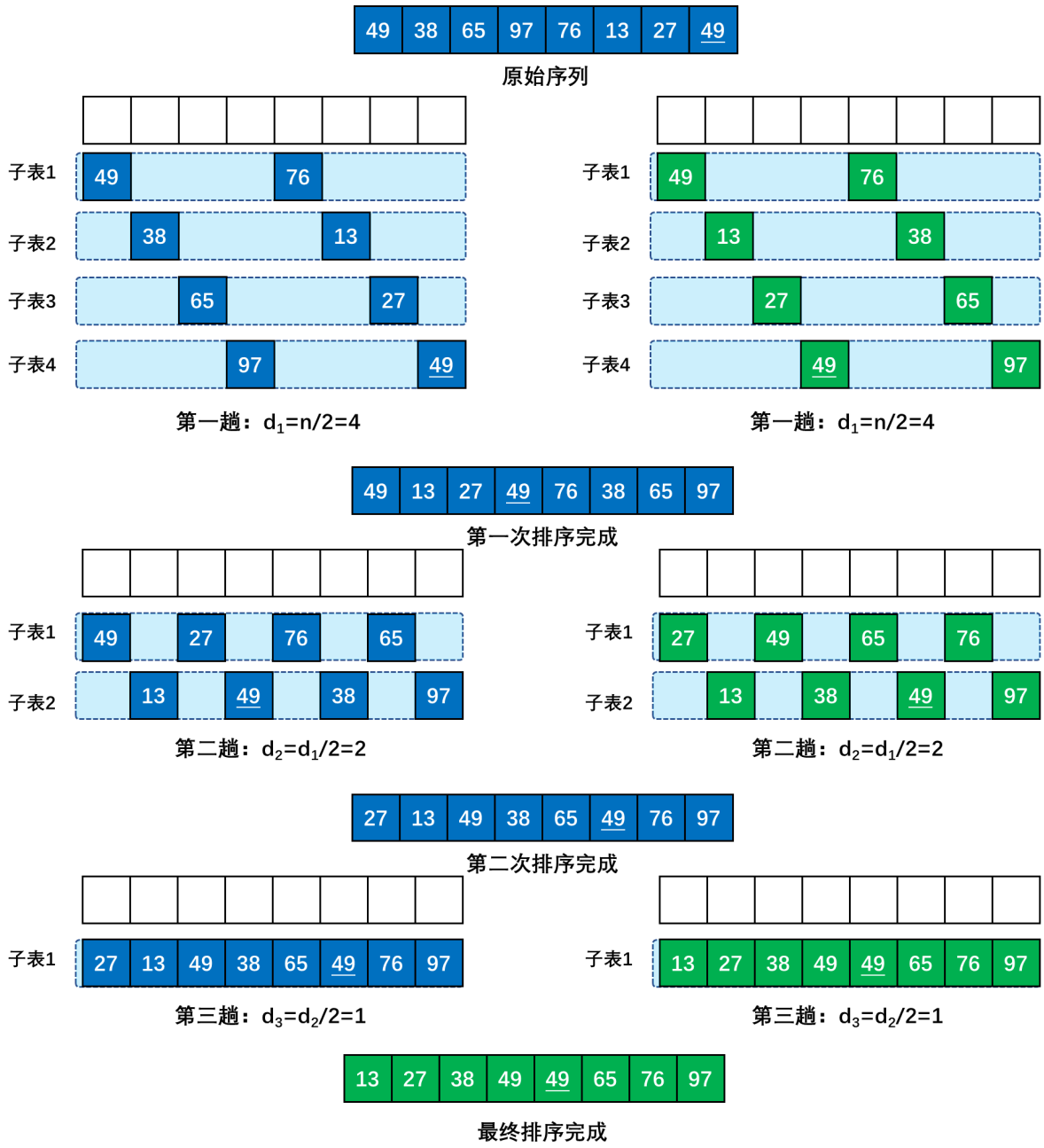
8.2.3 希尔排序

1、概念

算法步骤：现将待排序表分割成若干形如$L[i,i+d,i+2d,\ldots,i+kd]$的特殊**子表**，对各个子表分别进行直接插入排序。缩小增量 d，重复进行上述过程，直到d=1为止。

算法思想：先追求表中元素部分有序，在逐渐逼近全局有序

希尔排序的示例如下：



> 【注意】考试时出现的第一个增量**不一定**是n/2，可能会出现各种增量

2、代码实现

```
void ShellSort(int A[], int n) {
    int d, i, j;
    for (d = n / 2; d >= 1; d = d / 2) {     // 步长变化
        for (i = d + 1; i <= n; ++i) {
            if (A[i] < A[i - d]) {           // 将A[i]插入增量有序子表
                A[0] = A[i];                 // 暂存在A[0]
                for (j = i - d; j > 0 && A[0] < A[j]; j -= d) {
                    A[j + d] = A[j];         // 记录后移，查找插入的位置
                }
                A[j + d] = A[0];             // 插入
            }
        }
    }
}
```

3、算法效率分析

- 空间复杂度：$O(1)$
- 时间复杂度：未知，但优于直接插入排序
- 稳定性：不稳定
- 适用性：仅适用于顺序表

> 高频题型：给出增量序列，分析每一趟排序后的状态

8.3 交换排序

> 基于“交换”的排序：根据序列中两个元素关键字的比较结果来对换这两个记录在序列中的位置

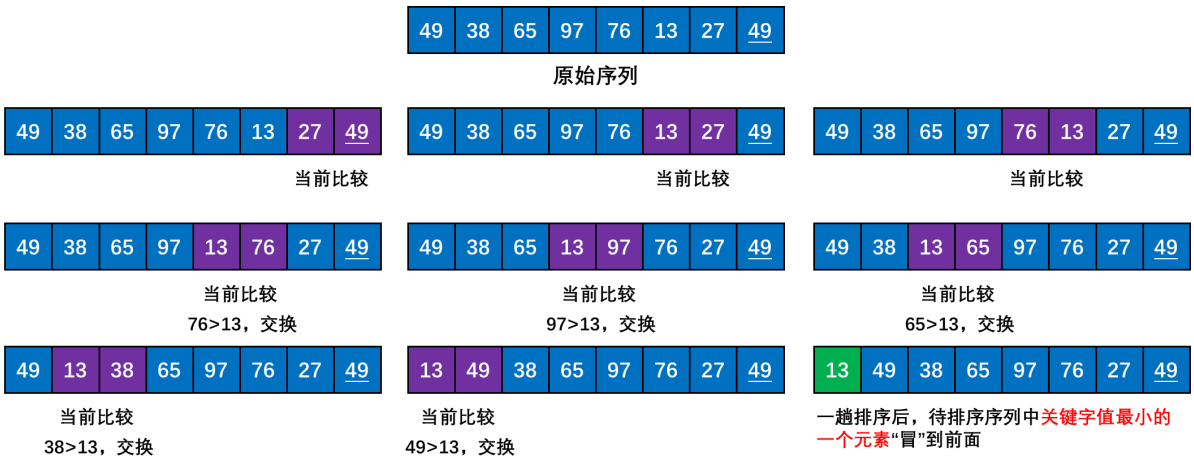
8.3.1 冒泡排序

1、算法思想

从后往前（或从前往后）两两比较相邻元素的值，若为逆序（$A[i-1] > A[i]$），则交换它们，直到序列比较完。称这样的过程为“一趟”冒泡排序。

> 若某一趟排序过程中未发生“交换”，则算法可提前结束，

以下是冒泡排序的一趟过程：



2、代码实现

```
void BubbleSort(int A[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        bool flag = false;          // 表示此趟排序是否发生交换
        for (int j = n - 1; j > i; --j) {   // 一趟冒泡
            if (A[j - 1] > A[j]) {         // 若为逆序
                swap(A[j - 1], A[j]);  // 交换元素
                flag = true;
            }
        }
        if (flag == false) {       // flag为false说明本次遍历后没有发生交换，表已经有序
            return;
        }
    }
}
```

3、算法效率分析

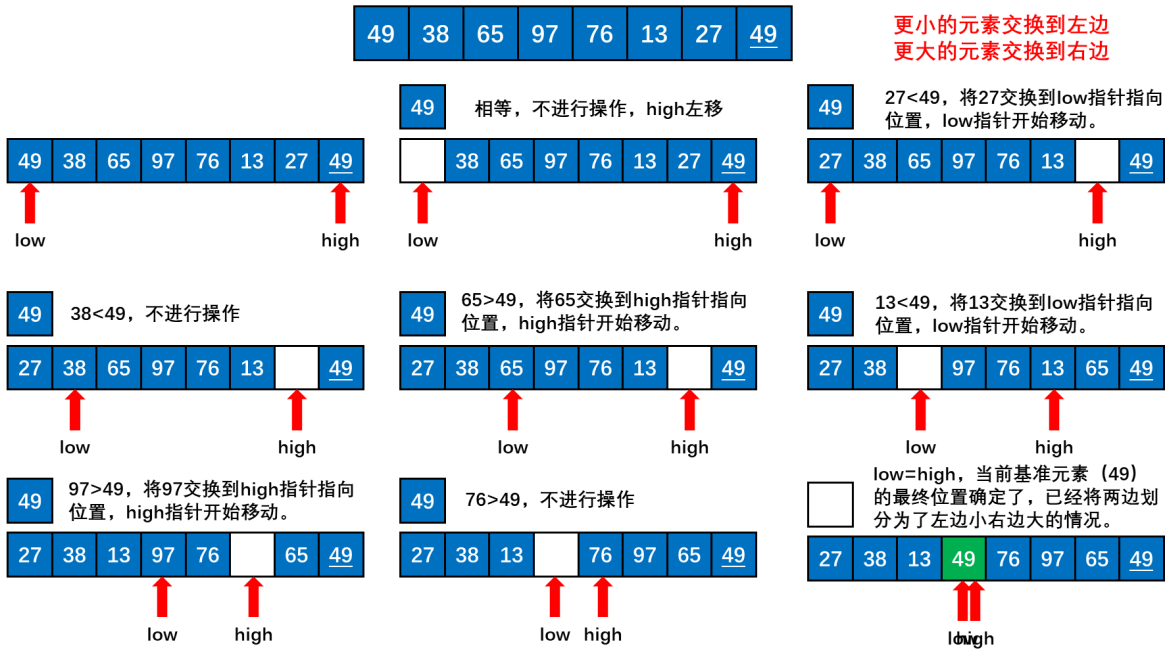
- 空间复杂度：$O(1)$
- 时间复杂度
  - 最好时间复杂度（有序）：$O(n)$
  - 最坏时间复杂度（逆序）：$O(n^2)$
  - 平均时间复杂度：$O(n^2)$
- 稳定性：只有`A[j-1]>A[j]`时才交换，因此算法是**稳定的**
- 适用性：顺序表和链表都适用

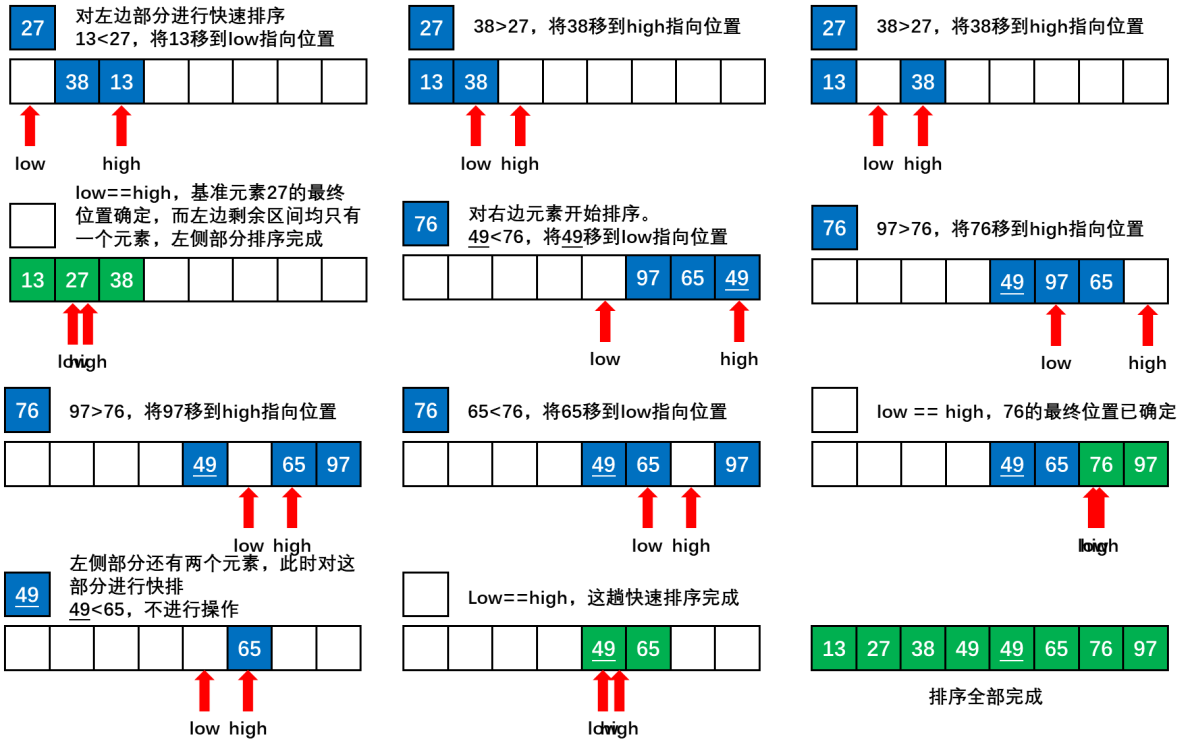
8.3.2 快速排序

1、算法思想

在待排序表 $L[1...n]$ 中任取一个元素 $pivot$ 作为枢轴（或者基准，通常去首元素），通过一趟排序将待排序表划分为独立的两部分 $L[1...k-1]$ 和 $L[K+1...n]$，使得 $L[1...k-1]$ 中的所有元素小于 $pivot$，$L[k+1...n]$中的所有元素大于等于 $pivot$，则 $pivot$放在了其最终位置 $L(k)$ 上，这个过程称为一个**划分**。然后分别递归地对两个子表重复上述过程，直到每部分内只有一个元素或空为止，即**所有元素放在了其最终位置上。**

以下是**完整的一趟快排**流程：





2、代码实现（递归）

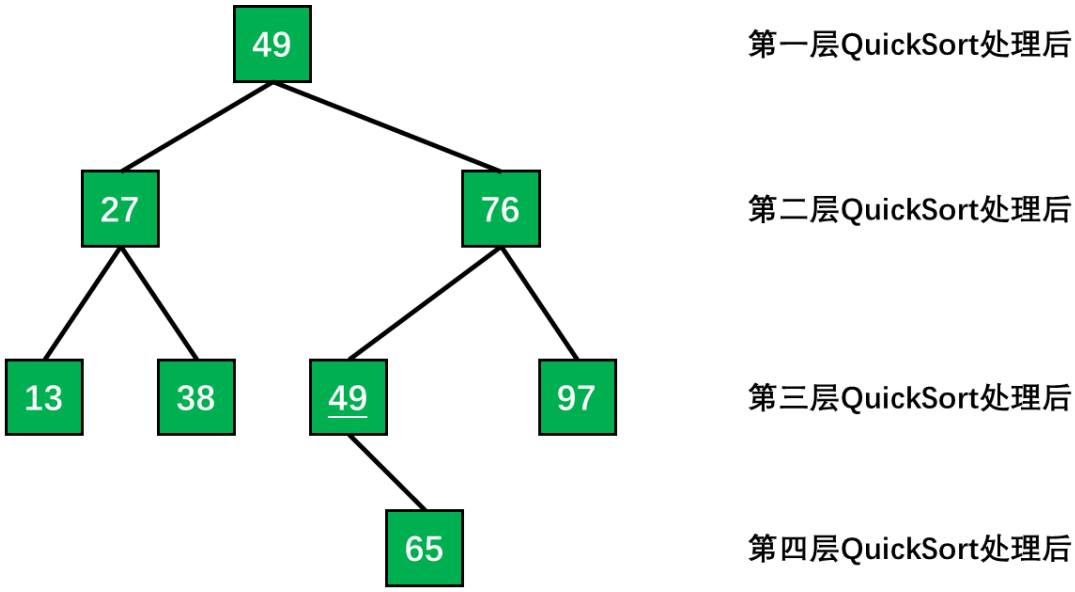
```
// 用第一个元素将待排序序列划分为左右两个部分
int Partition(int A[], int low, int high) {
    int pivot = A[low];          // 第一个元素作为枢轴
    while (low < high) {         // 用low和high搜索枢轴的最终位置
        while (low < high && A[high] >= pivot) {
            --high;
        }
        A[low] = A[high];        // 比枢轴小的元素移动到左端
        while (low < high && A[low] <= pivot) {
            ++low;
        }
        A[high] = A[low];        // 比枢轴大的元素移动到右端
    }
    A[low] = pivot;              // 枢轴元素存放的最终位置
    return low;                  // 返回枢轴元素的最终位置
}

// 快速排序
void QuickSort(int A[], int low, int high) {
    if (low < high) {                               // 递归跳出的条件
        int pivotpos = Partition(A, low, high);    // 划分
        QuickSort(A, low, pivotpos - 1);      // 划分左子表
        QuickSort(A, pivotpos + 1, high);      // 划分右子表
    }
}
```

3、算法效率分析

- 空间复杂度：$O(\text{递归层数})$
  - 最好空间复杂度：$O(nlog_2n)$
  - 最坏空间复杂度：$O(n)$
- 时间复杂度：$O(n*\text{递归层数})$

    - 最好时间复杂度：$O(nlog_2n)$
    - 最坏时间复杂度：$O(n^2)$
- 稳定性：不稳定
- 快速排序是**所有内部排序算法中，平均性能最优的排序算法。**

把n个元素组织成二叉树，二叉树的层数就是递归调用的层数。



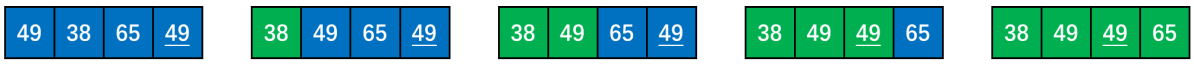
8.4 选择排序

8.4.1 简单选择排序

1、算法思想

每一趟从待排序序列中选取最大（或最小）的元素加入有序子序列。必须进行n-1趟处理。

其示例如下：



2、代码实现

```
void SelectSort(int A[], int n) {
    for (int i = 0; i < n - 1; ++i) {        // 一共进行n-1趟
        int min = i;                         // 记录最小元素位置
        for (int j = i + 1; j < n; ++j) {    // 在A[i...n-1]中选择最小的元素
            if (A[j] < A[min]) {
                min = j;                     // 更新最小元素位置
            }
        }
        if (min != i) {
            swap(A[i], A[min]);        // 移动元素位置
        }
    }
}
```

3、算法效率分析

- 空间复杂度：$O(1)$
- 时间复杂度：$O(n^2)$
- 稳定性：不稳定
- 适用性：即可用于顺序表，也可用于链表

8.4.2 堆排序

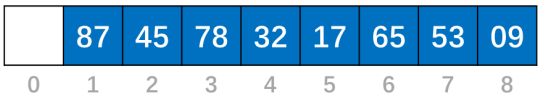
1、什么是堆?

若n个关键字序列 $L[1...n]$ 满足下面某一条性质，则称为堆：

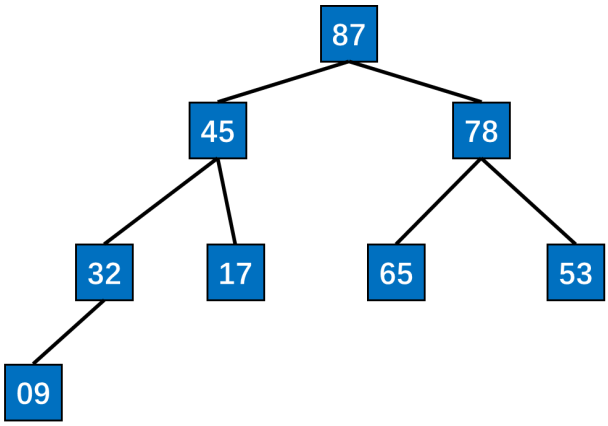
- 若满足：$L(i) \geq L(2i)$且$L(i) \geq L(2i+1), (1 \leq i \leq n/2)$，则称为大根堆（大顶堆）
- 若满足：$L(i) \leq L(2i)$且$L(i) \leq L(2i+1), (1 \leq i \leq n/2)$，则称为小根堆（小顶堆）

大根堆：

> 逻辑视角：大根堆中，根≥左、右



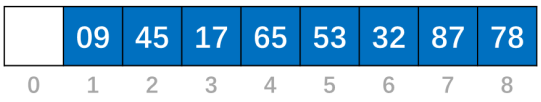
存储视角



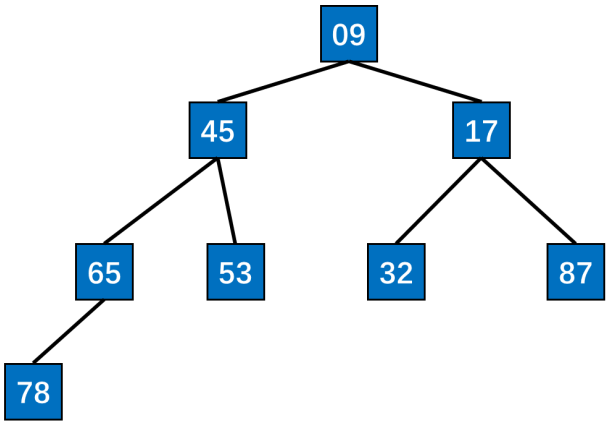
逻辑视角

小根堆：

> 逻辑视角：根≤左、右



存储视角



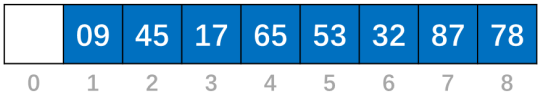
逻辑视角

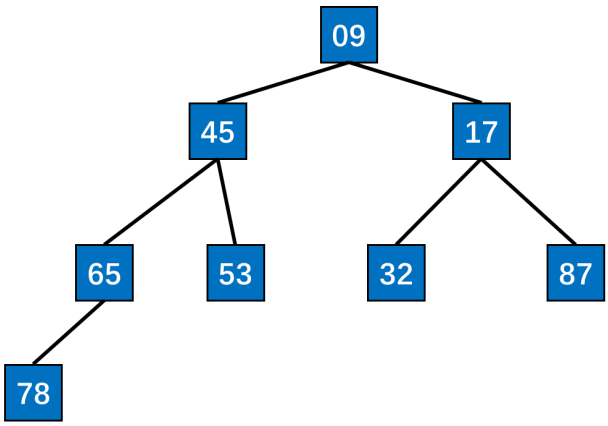
2、基于“堆”来进行排序

（1）建立大根堆

思路：把所有非终端结点都检查一遍，是否满足大根堆的要求，如果不满足，则进行调整。（不满足，将当前结点与更大的一个孩子互换）

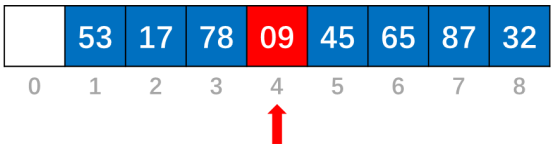
建立大根堆的一趟流程如下：

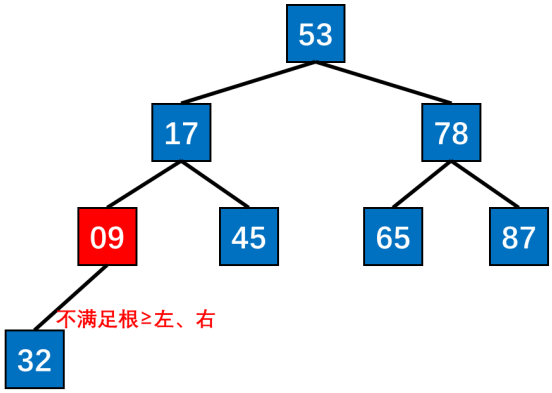

09 45 17 65 53 32 87 78
0 1 2 3 4 5 6 7 8

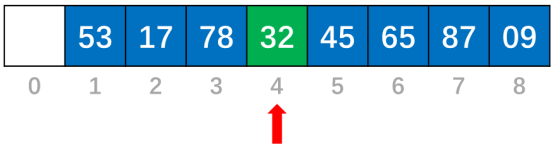

09
45
17
65
53
32
87
78

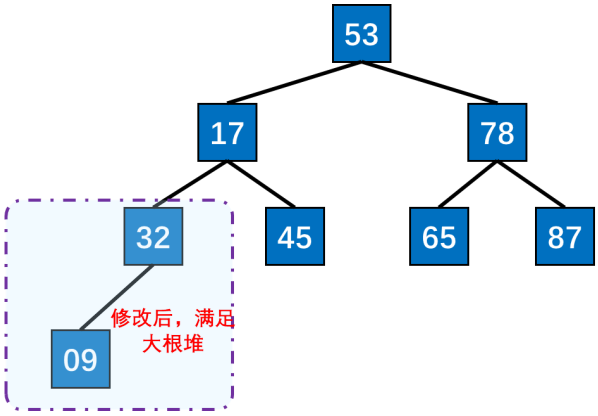
存储视角

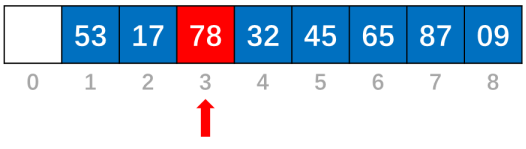
逻辑视角

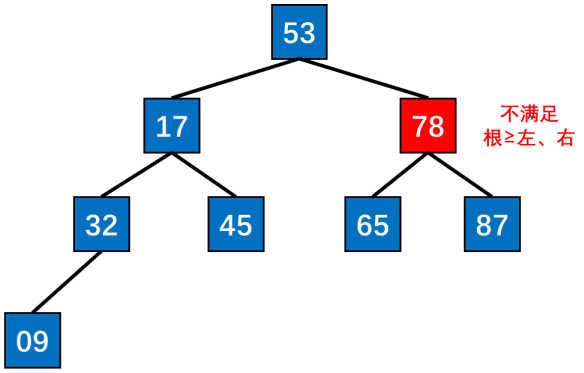

53 17 78 09 45 65 87 32
0 1 2 3 4 5 6 7 8

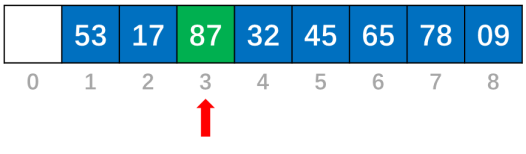

53
17
78
09
45
65
87
32
不满足根≥左、右

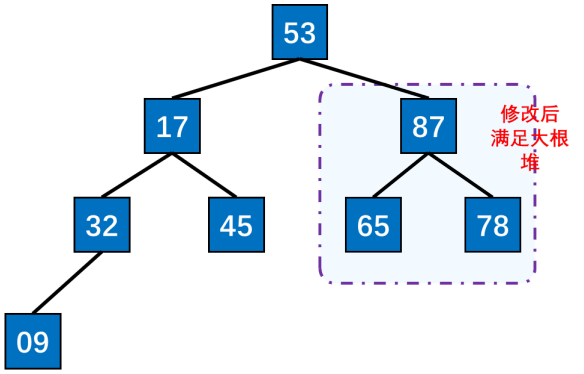

53 17 78 32 45 65 87 09
0 1 2 3 4 5 6 7 8

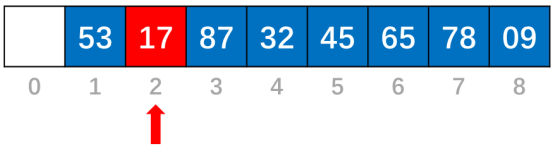

53
17
78
32
45
65
87
09
修改后，满足
大根堆

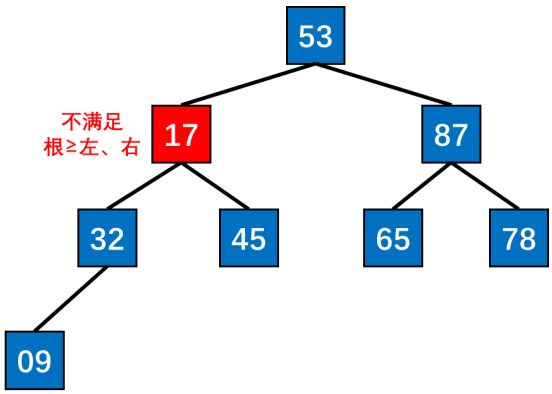

53 17 78 32 45 65 87 09
0 1 2 3 4 5 6 7 8

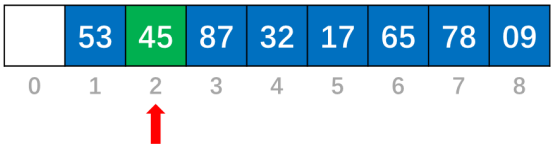

53
17
78
32
45
65
87
09
不满足
根≥左、右

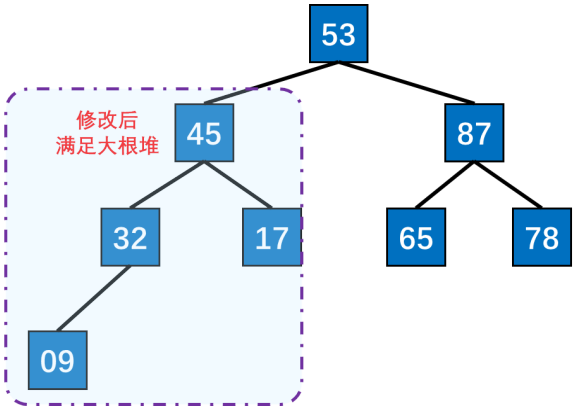

53 17 87 32 45 65 78 09
0 1 2 3 4 5 6 7 8

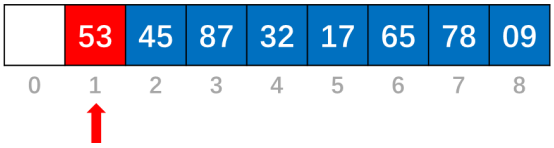

53
17
87
32
45
65
78
09
修改后
满足大根
堆

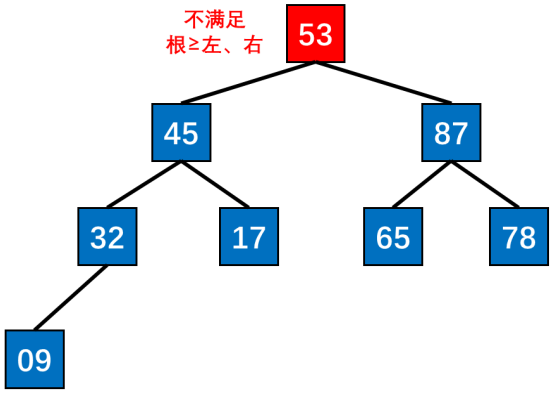

53 17 87 32 45 65 78 09
0 1 2 3 4 5 6 7 8

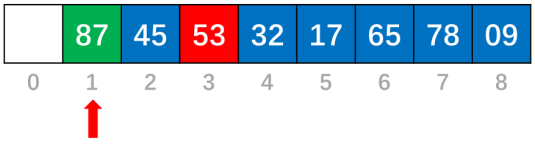

不满足
根≥左、右
53
17
87
32
45
65
78
09

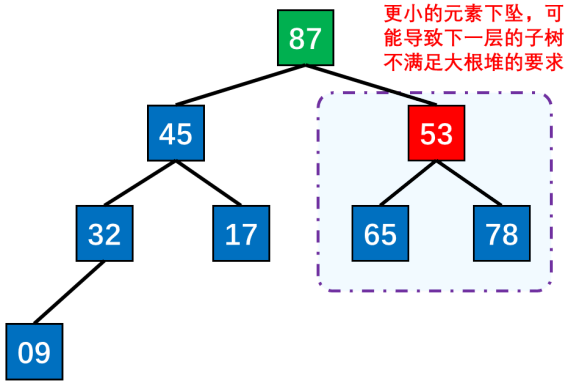

53 45 87 32 17 65 78 09
0 1 2 3 4 5 6 7 8

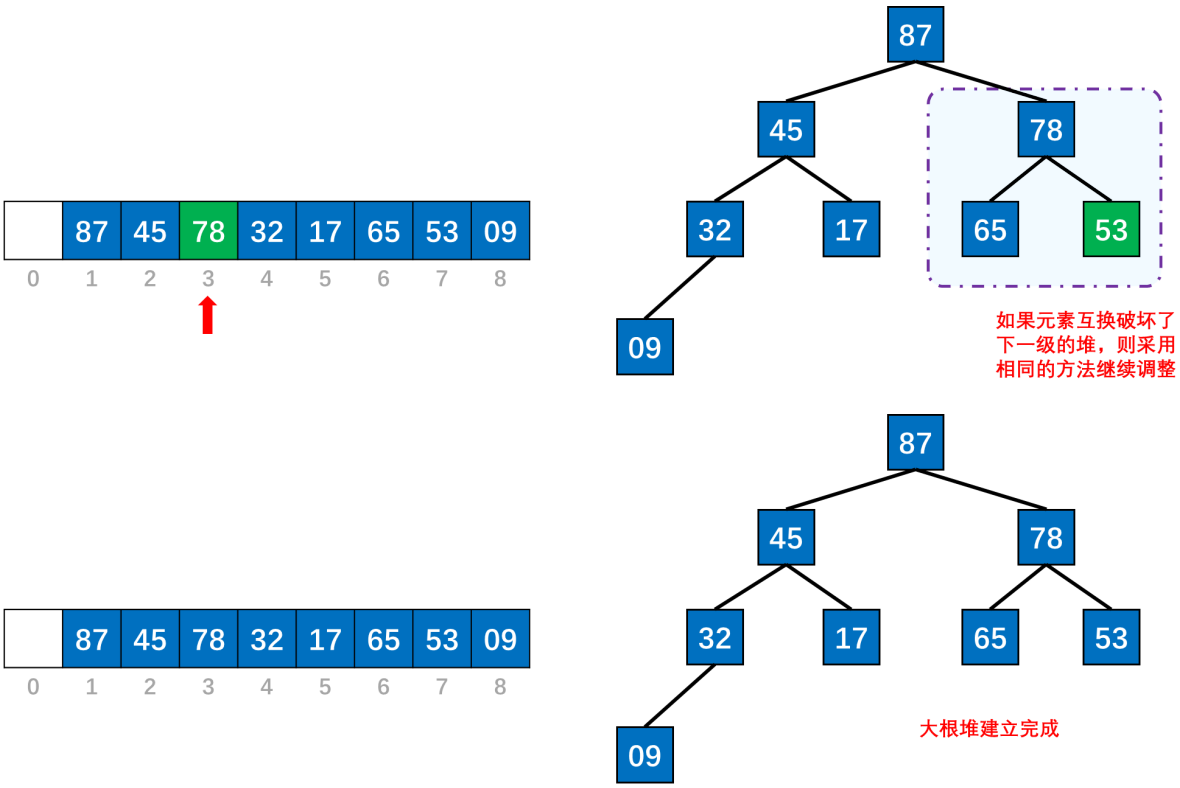

修改后
满足大根堆
53
45
87
32
17
65
78
09


53 45 87 32 17 65 78 09
0 1 2 3 4 5 6 7 8


不满足
根≥左、右
53
45
87
32
17
65
78
09


87 45 53 32 17 65 78 09
0 1 2 3 4 5 6 7 8


更小的元素下坠，可
能导致下一层的子树
不满足大根堆的要求
87
45
53
32
17
65
78
09



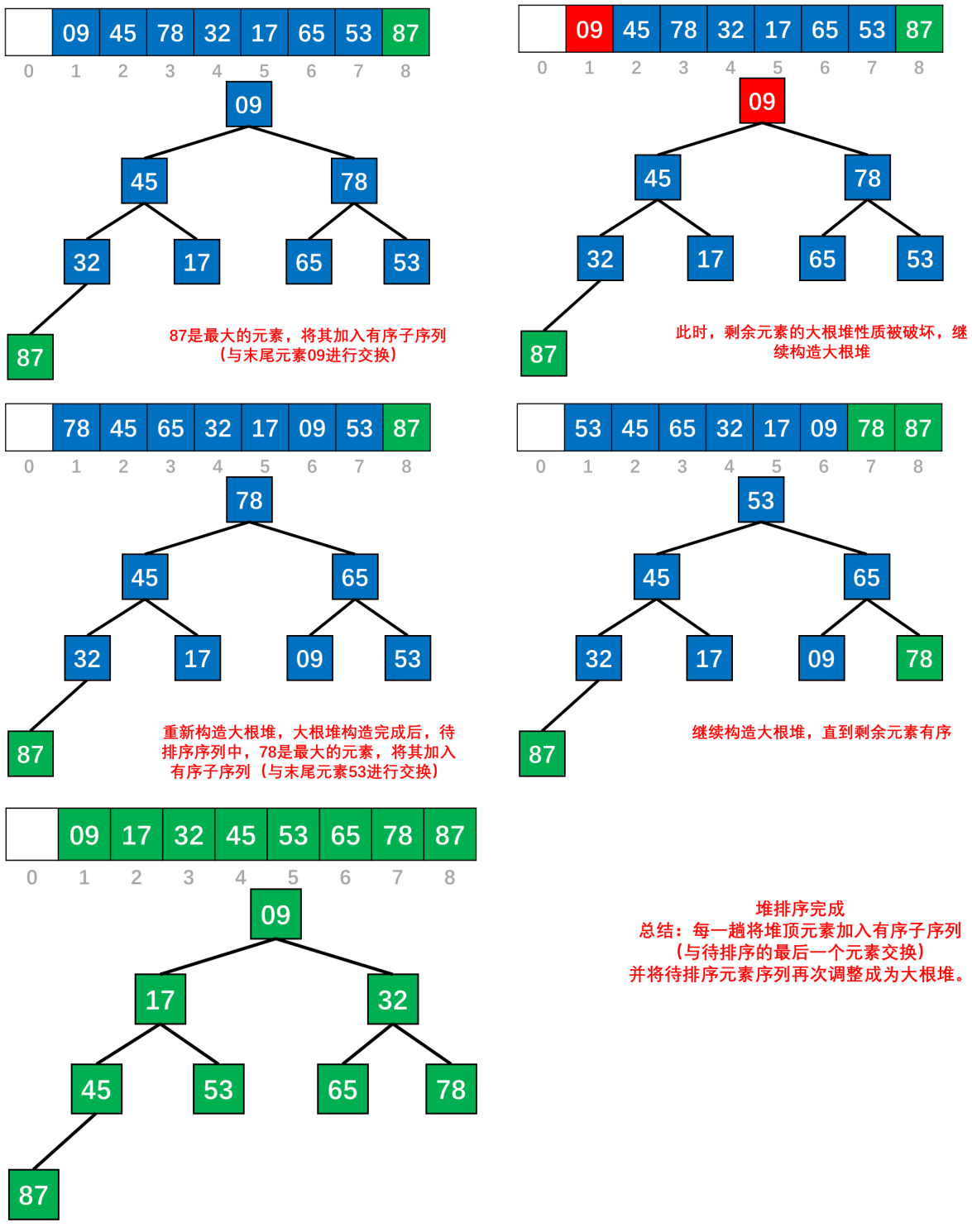
代码实现：

```
// 将以k为根的子树调整为大根堆
void HeadAdjust(int A[], int k, int len) {
    A[0] = A[k];         // A[0]暂存子树的根节点
    for (int i = 2 * k; i <= len; i *= 2) { // 沿着key较大的子节点向下筛选
        if (i < len && A[i] < A[i + 1]) {
            i++;             // 取key较大的子节点的下标
        }
        if (A[0] >= A[i]) { // 筛选结束
            break;
        } else {
            A[k] = A[i];     // 将A[i]调整到双亲结点上
            k = i;           // 修改k值，以便继续向下筛选
        }
    }
    A[k] = A[0];             // 将被筛选结点的值放入最终位置
}

// 建立大根堆
void BuildMaxHeap(int A[], int len) {
    for (int i = len / 2; i > 0; --i) { // 从后往前调整所有非终端节点
        HeadAdjust(A,i,len);
    }
}
```

（2）堆排序

每一趟将堆顶元素加入有序子序列（与**待排序序列的**最后一个元素互换），并将待排序元素序列再次调整成为**大根堆。**

其示例如下：



> 【注意】基于大根堆的堆排序得到的是**递增序列**

```
void HeapSort(int A[], int len) {
    BuildMaxHeap(A, len);
    for (int i = len; i > 1; --i) {
        swap(A[i], A[1]);
        HeadAdjust(A, 1, i - 1);
    }
}
```

（3）算法效率分析

- 时间复杂度：$O(n)+O(nlog_2n)=O(nlog_2n)$
- 空间复杂度：$O(1)$
- 稳定性：不稳定
- 基于大根堆的堆排序得到递增序列，基于小根堆的堆排序得到递减序列。

（4）堆的插入删除

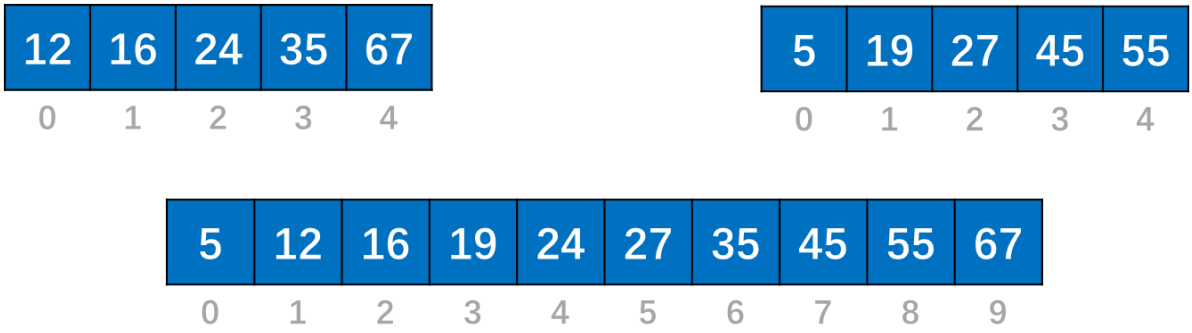
- 堆的插入：对于大（或小）根堆，要插入的元素放到表尾，然后与父节点对比，若新元素比父节点更大（或小），则将二者互换。新元素就这样一路“上升”，直到无法继续上升为止。
- 堆的删除：被删除的元素用堆底元素替换，然后让该元素不断“下坠”，直到无法下坠为止。

8.5 归并排序和基数排序

8.5.1 归并排序

1、什么是归并?

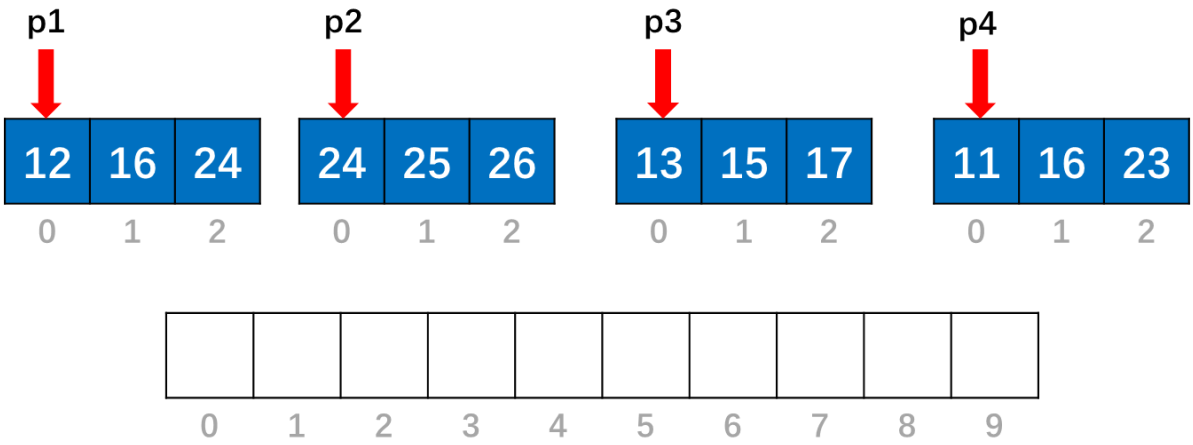
归并：将两个或多个**已经有序**的序列合并成一个。



二路归并：把**两个**已经有序的序列合并成一个。

多路归并：把**多个**已经有序的序列合并成一个。

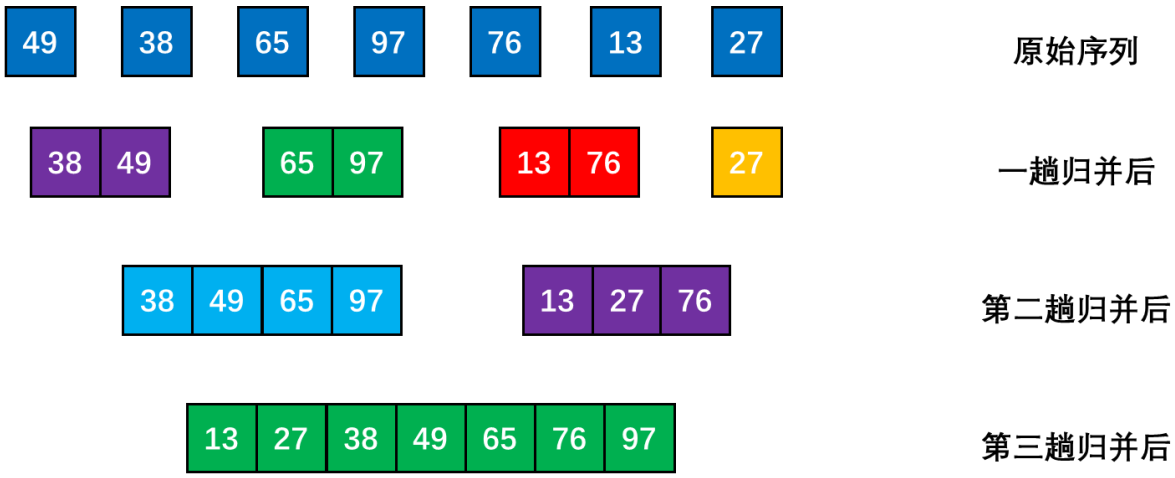
四路归并：对比p1、p2、p3、p4所指元素，选择更小的一个放入k所指位置。



> 【结论】m路归并，每选出一个元素需要对比关键字$m-1$次

2、归并排序

核心操作：将数组内的两个有序序列归并成一个。



> 归并排序一般会考察某一趟排序后得到的序列，基本上不可能考察代码。

3、代码实现

王道书中代码如下：

```
// 辅助数组B
int *B=(int *)malloc(n*sizeof(int));

// A[low,...,mid]，A[mid+1,...,high]各自有序，将这两个部分归并
void Merge(int A[], int low, int mid, int high){
    int i,j,k;
    for(k=low; k<=high; k++)
        B[k]=A[k];
    for(i=low, j=mid+1, k=i; i<=mid && j<= high; k++){
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    while(i<=mid)
        A[k++]=B[i++];
    while(j<=high)
        A[k++]=B[j++];
}

// 递归操作
void MergeSort(int A[], int low, int high){
    if(low<high){
        int mid = (low+high)/2;
        MergeSort(A, low, mid);
        MergeSort(A, mid+1, high);
        Merge(A,low,mid,high);      //归并
    }
}
```

4、算法效率分析

- 空间复杂度：$O(n)$
- 时间复杂度：$O(nlogn)$
- 稳定性：稳定

8.5.2 基数排序

1、概念

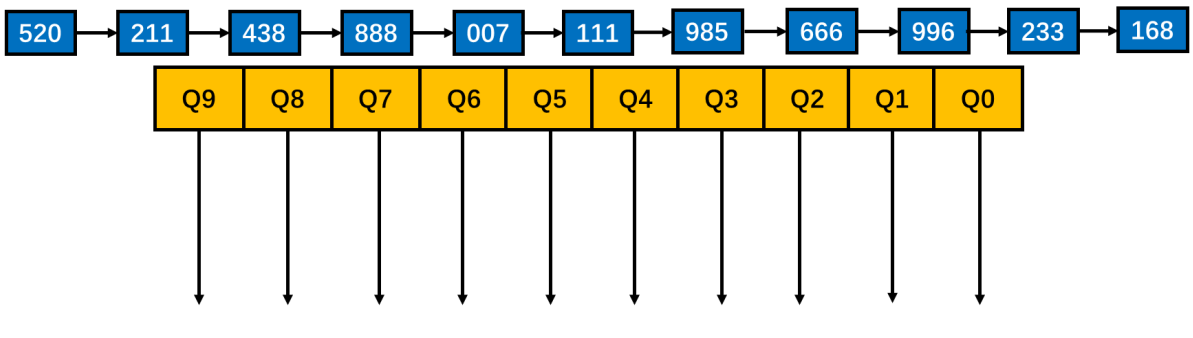
基数排序是一种很特别的排序算法，它**不急于比较和移动进行排序**，而是基于关键字各位的大小进行排序。

> 基数排序是一种借助多关键字排序的思想对单逻辑关键字进行排序的算法。

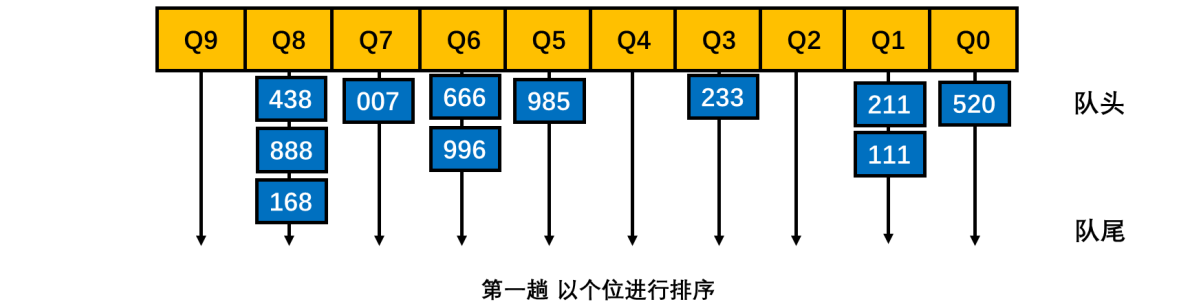
为实现多关键字排序，通常有两种方法：

1. **最高位优先**：按关键字位权重递减依次逐层划分成若干更小的子序列，最后将所有子序列依次连接成一个有序蓄力
2. **最低位优先**：按关键字权重递增依次进行排序，最终形成一个有序序列。

以下是堆排序的一个示例：

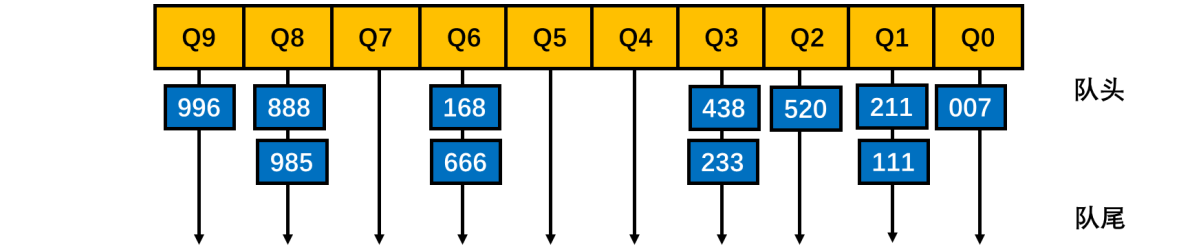


初始状态



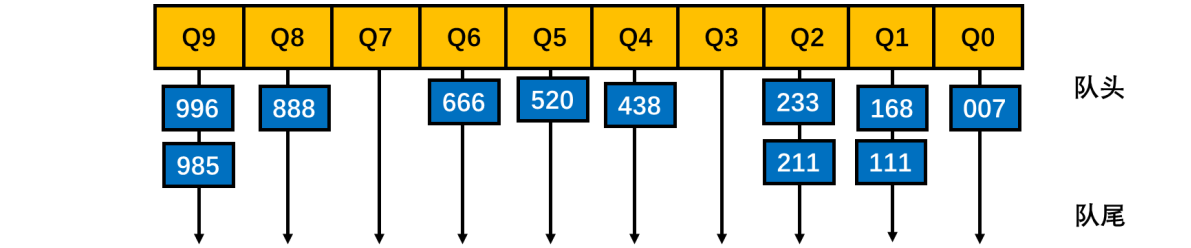
第一趟 以个位进行排序

第一趟收集结果



第二趟 以十位进行排序

第二趟收集结果



第三趟 以百位进行排序

第三趟收集结果

最终得到了一个递减序列。

2、算法过程

基数排序得到递减序列的过程如下：

初始化：设置r个空队列，$Q_{r-1},Q_{r-2},\dots,Q_0$

按照各个关键字位**权重递增**的次序（个、十、百、...）的次序，对d个关键字位分别做“分配”和“收集”

> 分配：顺序扫描各个元素，根据当前处理的关键字位，将元素插入相应的队列。
>
> 收集：把各个队列中的结点依次出队并链接。

> 基数排序不考察代码。

3、算法效率分析

整个关键字拆分为d位（或“组”），按照各个关键字位权重递增的次序，需要做d趟分配和收集。

当前处理的关键字位可能取到r个值，则需要建立r个队列。

> 如上述例子：$d=3$，$r=10$

- 空间复杂度：$O(r)$
- 时间复杂度：$O(d(n+r))$
- 稳定性：稳定

4、算法应用

基数排序适合处理：

- 数据元素的关键字可以方便地拆分成d组，且 d **较小**
- 每组关键字的取值范围不大，即 r **较小**
- 数据元素个数 n **较大**

8.6 内部排序算法的比较

算法类型	最好时间复杂度	平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
希尔排序	未知	未知，但优于直接插入排序	未知	$O(1)$	不稳定
快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(log_2n)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
2路归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定