第7章 排序算法

#数据结构与算法

> 面对海量数据，选择合适的算法显得尤为重要。
> 在实际进行时，往往搭配**预处理**来提高效率。

本章的**三个要求**：

1. 每种算法思想都要掌握,有没有所谓的先决条件?
2. 查找时要清楚比较次数（ASL）和移动次数。（稳定性）
3. 掌握每种查找的最好情况和最坏情况，平均情况。

表 7-1 各种排序方法的比较

排序方法	时间复杂度			辅助存储	稳定性
	最好情况	平均情况	最坏情况		
直接插入排序	O(n)	O(n2)	O(n2)	O(1)	√
希尔排序	O(n1.5)			O(1)	×
冒泡排序	O(n)	O(n2)	O(n2)	O(1)	√
快速排序	O(nlog2n)	O(nlog2n)	O(n2)	O(log2n)	×
简单选择排序	O(n2)	O(n2)	O(n2)	O(1)	×
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	×
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)	×
基数排序	O(d(n+rd))	O(d(n+rd))	O(d(n+rd))	O(rd)	√

7.1 排序的基本概念

假定文件是由 n 个记录组成的序列，其相应的关键字序列为 $(K_1, K_2, \ldots, K_n)$。

排序就是将数据元素的任意序列，按关键字排成人们需要的序列（通常是升序或降序）。

排序结果的唯一性

- 若待排序序列中所有关键字均不相同，则排序结果是**唯一**的。
- 若存在关键字相同的记录，则排序结果可能**不唯一**，这取决于排序算法的稳定性。

排序算法的稳定性

若待排序序列中存在两个或两个以上关键字相等的记录 R_i 和 R_j $(i<j)$，排序后它们的相对次序保持不变，即 R_i 仍在 R_j 之前，则称该排序算法是**稳定的**；否则称为**不稳定的**。

注意：排序算法的稳定性是针对所有输入实例而言的。一个算法不能因为对某个特定实例产生了稳定的结果就被认为是稳定的。稳定性由算法本身决定。

存储结构

通常的存储方式有如下三种：

1. 顺序表
2. 链表
3. 二叉树等树形结构

7.2 简单排序

> 简单排序算法时间复杂度均为$O(n^2)$，基本有序的时候“快”。

7.2.1 简单插入排序

基本思想：将待排序序列分为已排序区和未排序区。每次从未排序区取出一个元素，在已排序区中从后向前扫描，找到相应位置并插入。

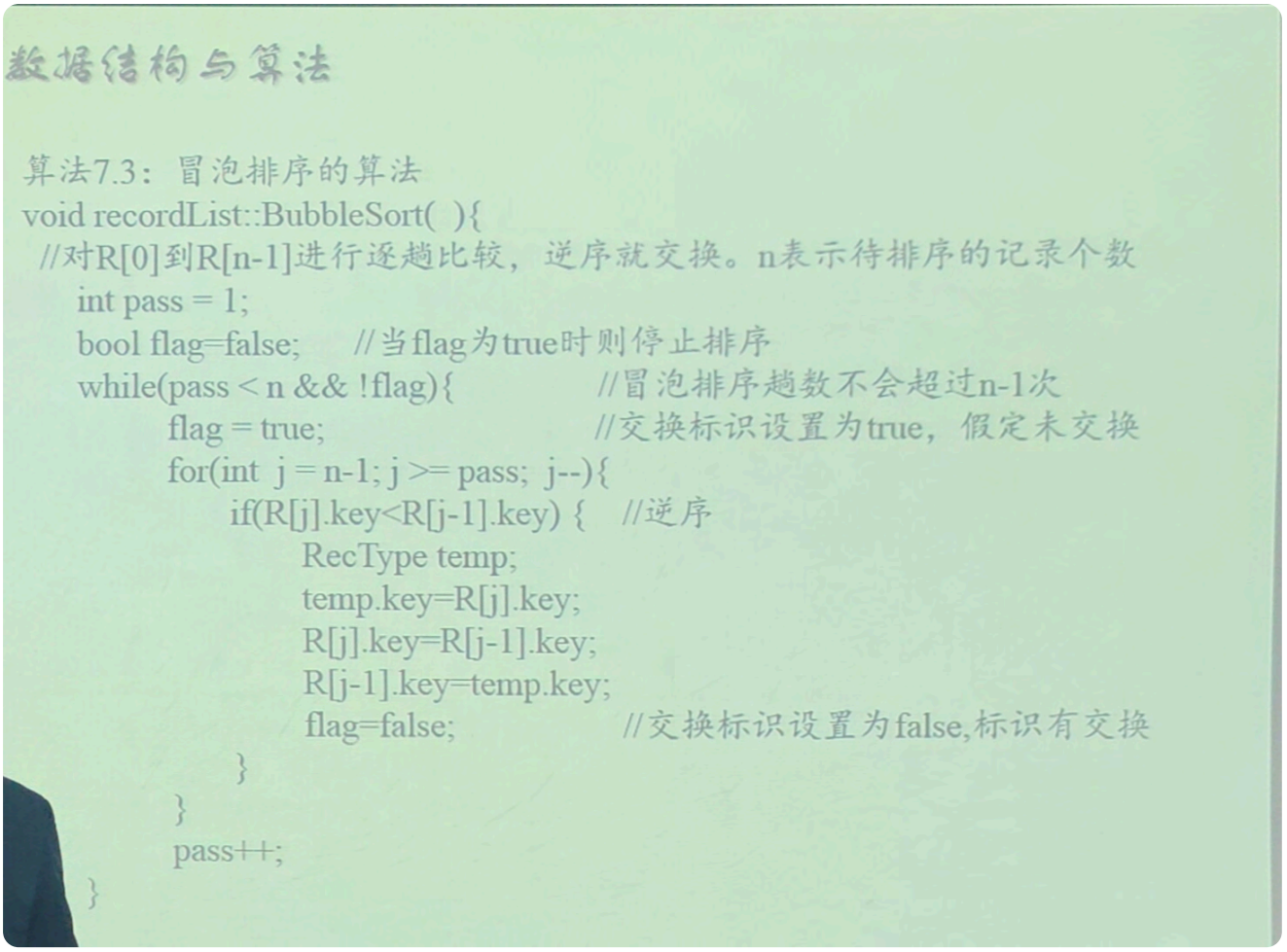
```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

7.2.2 简单冒泡排序

基本思想：重复地遍历待排序序列，一次比较两个相邻的元素，如果它们的顺序错误就把它们交换过来。

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}
```

优化后：加入flag记录某一趟是否已经交换过。



> 冒泡排序法是自底向上相邻元素两两比较，逆序则交换，使最小元素上移。
> 还有另一种方法是自顶向下相邻元素两两比较，逆序交换，最大元素下移，这种方法称为吊桶法，吊桶法与冒泡法的排序过程类似。

7.2.3 简单选择排序

基本思想：每次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。(存放下标而不是直接赋值)

```
void selectionSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      int min_idx = i;
      for (int j = i + 1; j < n; j++) {
         if (arr[j] < arr[min_idx])
            min_idx = j;
      }
      swap(arr[i], arr[min_idx]);
   }
}
```

> 简单选择排序的关键字比较次数和待排序的初始序列**无关**，简单排序算法是一种不稳定的排序算法。

7.3 高级排序

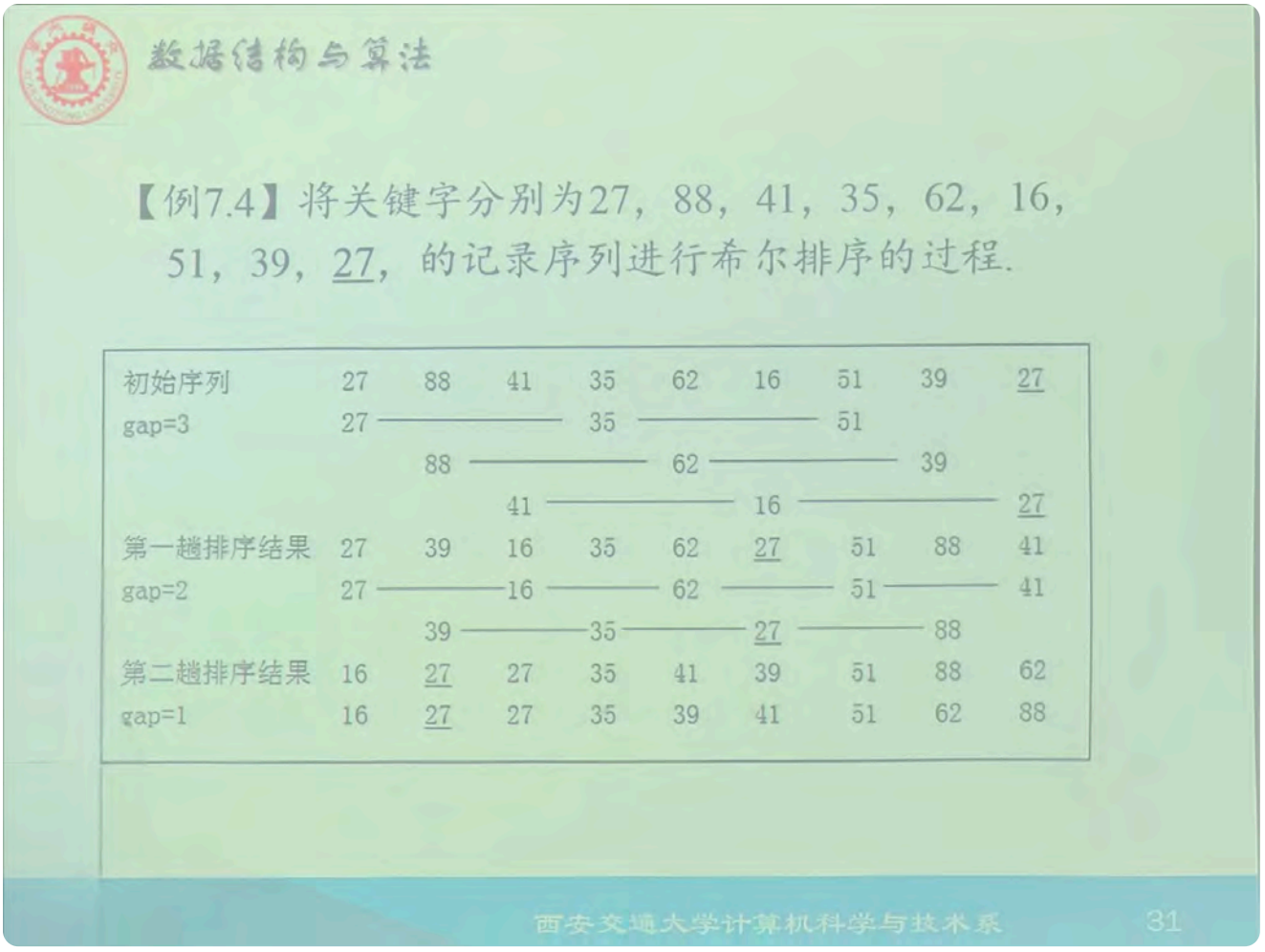
> 高级排序的时间复杂度基本都是$O(nlog(n))$

7.3.1 希尔排序（Shell）

> 希尔排序是对简单插入排序算法的一种改进，又称为**缩小增量排序**

基本思想：将数组按增量d分组，对每组进行插入排序，逐步缩小d直到1，然后对全部数进行选择排序。
稳定性：不稳定。

```
void shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int key = arr[i];
            int j = i;
            while (j >= gap && arr[j - gap] > key) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = key;
        }
    }
}
```



在最后一趟排序之前，希尔排序的序列就已经基本有序了。

7.3.2 快速排序

基本思想：选择一个基准元素，将数组分成小于和大于基准的两部分，递归排序子数组。
稳定性：不稳定。

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
```

```
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

- 最好情况：每次正好在中间分，类似于完全二叉树

7.3.3 归并排序（Merge）

需要掌握2路归并算法，明白算法流程即可。

归并排序（Merging Sort）是一种高级排序方法，它同快速排序一样，也是基于**分治法**的。
算法流程：
（1）先将n个待排序记录进行子表划分，通过不断的划分使每个待排序记录序列的长度为1；
（2）然后将相邻位置的两个待排序记录进行归并，得到[n/2]个长度为2或1的有序子序列，然后两两归并，……，这样重复进行，直到得到一个长度为n的有序序列为止。

7.3.4 树形排序：锦标赛选择排序

> 待排序关键字不能相同，否则不能判断胜者。

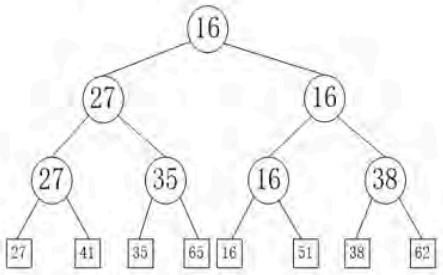
基本思想：模仿体育比赛的淘汰赛制度。首先将 n 个关键字两两比较，得到 [n/2] 个胜者，然后对这些胜者再进行两两比较，如此重复，直到选出一个总冠军（最小或最大值）。

算法过程：

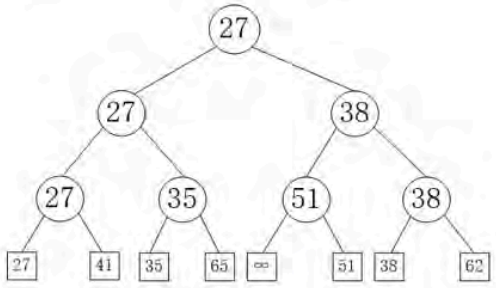
1. **构建选择树**：将 n 个待排序记录作为叶子节点，构建一棵完全二叉树（即选择树）。树中每个非叶子节点的值都是其左右孩子中关键字较小（或较大）者。
2. **输出并调整**：输出根节点的记录。然后将该记录对应的叶子节点的值设为一个极大（或极小）的哨兵值，表示它已出局。
3. **重建**：从被修改的叶子节点开始，沿着其父节点路径一路向上，重新比较并调整节点值，直到根节点。新的根节点就是次小（或次大）的记录。
4. 重复步骤 2 和 3，直到所有记录都被输出。

★注：使用锦标赛选择排序的时候，待排序记录的关键字不能相同，如果相同的话，将无法判断谁是优胜者。

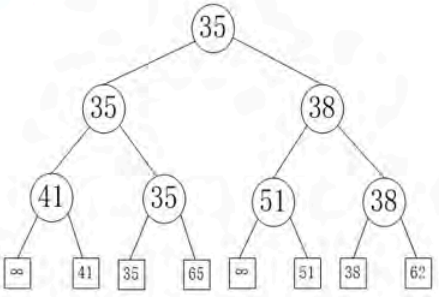
【例7.7】对关键字序列{27，41，35，65，16，51，38，62}进行锦标赛选择排序，各趟排序过程如下图所示。



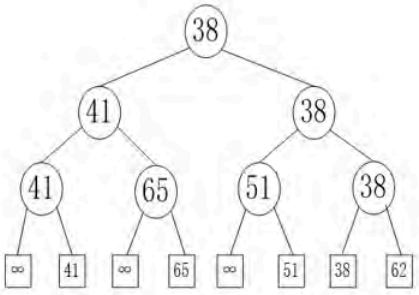
（a)第一趟排序，输出 16



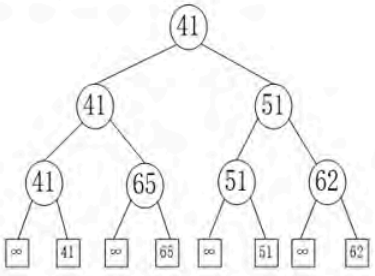
（b）把 16 置为∞，进行第二趟排序，输出 27



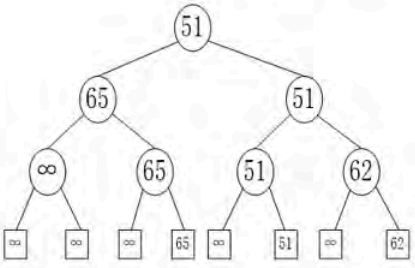
（c）把 27 置为∞,排序输出 35



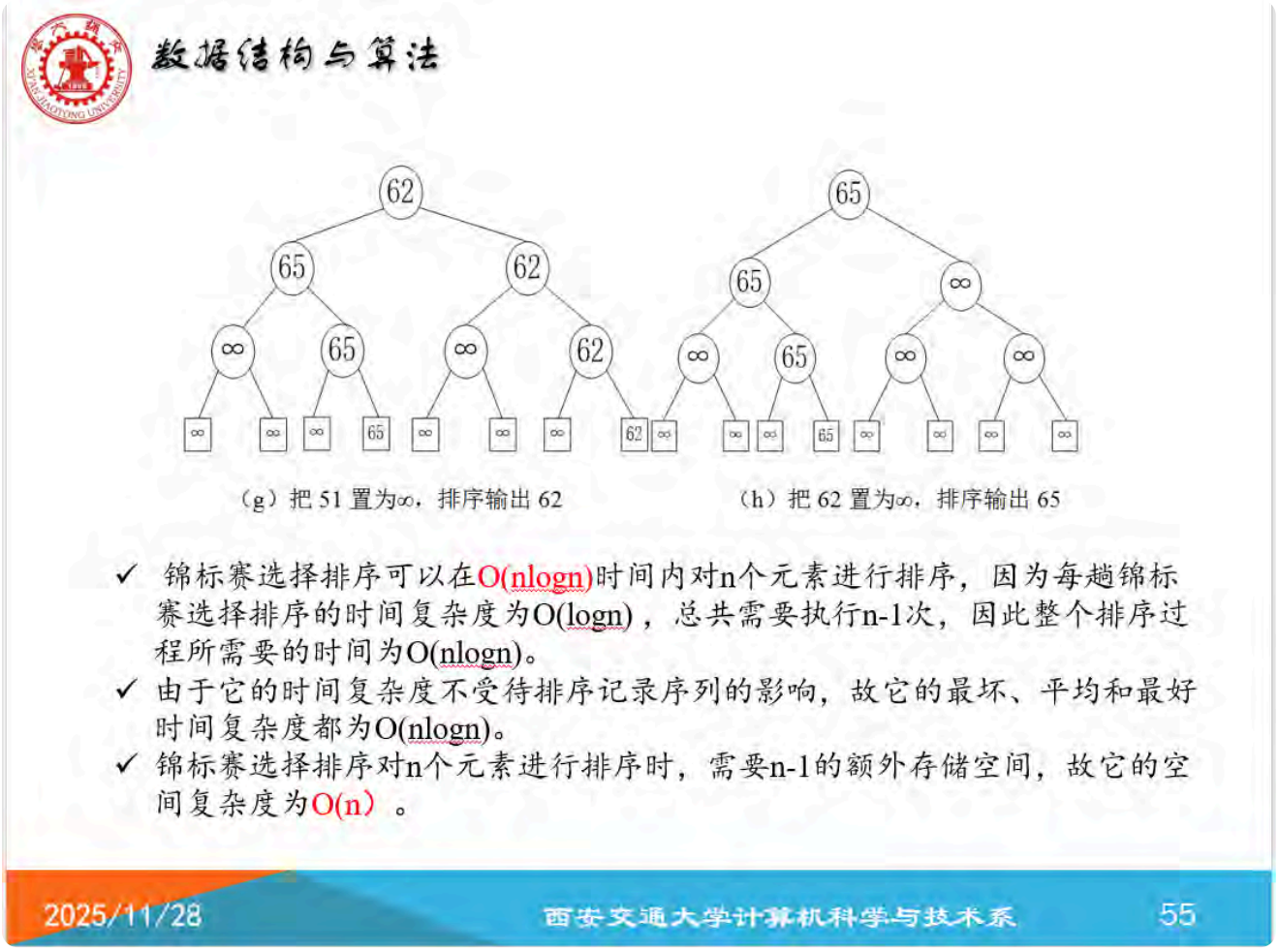
（d）把 35 置为∞，排序输出 38



(e)把 38 置为∞，排序输出 41



（f）把 41 置为∞，排序输出 51



稳定性：不稳定。

缺点：需要大量的空间；没有利用先前的结果。

7.3.5 树形排序：堆排序（HeapSort）

堆排序解决了上一个排序算法的的两个缺点。

算法具体过程：
S1：建立初始堆；
S2：（循环做，直到n=1）
{ 交换R[1]和R[n];
n--;
做筛选;
}

- 堆排序的时间复杂度为O(nlog2n)。
- 堆排序算法过程中，用到了一个临时记录的变量，故其空间复杂度为O(1)。
- 堆排序是一种不稳定的排序算法。

> 堆排序算法在初始序列基本有序的时候的时候，处理的效率较慢。

7.4 关键字比较排序下界问题

注：7.4和7.5上课一带而过了,重点还是之前的算法

7.5 非关键字比较的排序

7.5.1 基数排序

需要理解每一趟发生的变化，这种排序现在不太常用，可以用于桥牌的大小比较。

7.5.2 多关键字排序

（省略）

7.6 综合评价

（1）从平均时间性能而言，一般认为快速排序最佳，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。对于n值很大而关键字位数较小的序列，基数排序的速度最快。
（2）从稳定性而言，一般简单排序算法是稳定的，高级排序算法不稳定，但归并排序是稳定的。

上一章 第6章 图
下一章 第8章 查找算法