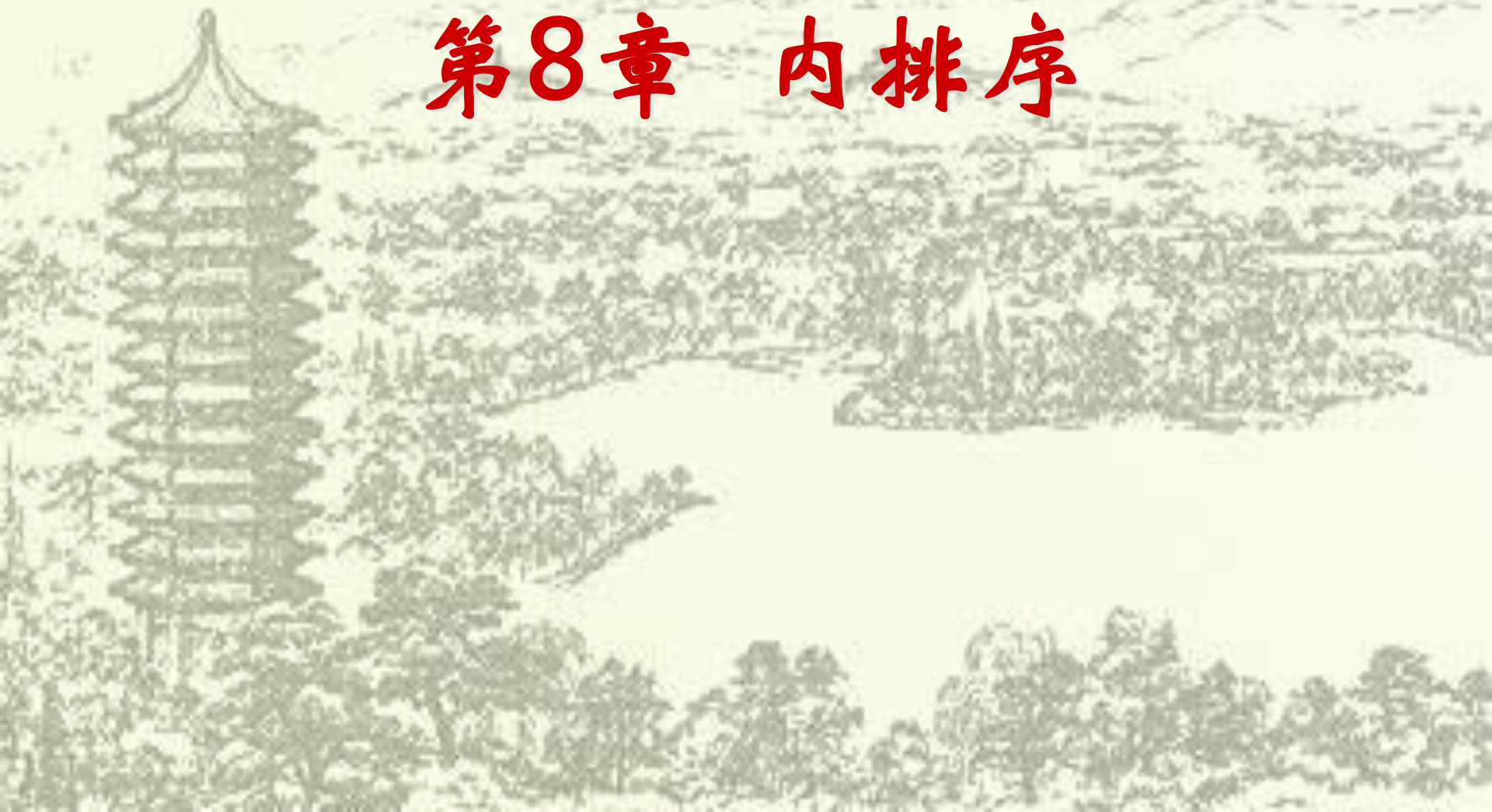


数据结构与算法

第8章 内排序



大纲



- 8.1 排序问题的基本概念
- 8.2 插入排序 (Shell排序)
- 8.3 选择排序 (堆排序)
- 8.4 交换排序
 - 8.4.1 冒泡排序
 - 8.4.2 快速排序
- 8.5 归并排序
- 8.6 分配排序和索引排序
- 8.7 排序算法的时间代价
- 8.8 内排序知识点总结

8.1 基本概念



- 记录(Record): 结点, 进行排序的基本单位
- 关键码(Key): 唯一确定记录的一个或多个域
- 排序码(Sort Key): 作为排序运算依据的一个或多个域
- 序列(Sequence): 线性表
 - 由记录组成

什么是排序？



□ 排序

- 将序列中的记录按照排序码顺序排列起来
- 排序码域的值具有不减(或不增)的顺序

□ 内排序

- 整个排序过程在内存中完成



排序问题

- 给定一个序列 $R = \{r_1, r_2, \dots, r_n\}$
 - 其排序码分别为 $k = \{k_1, k_2, \dots, k_n\}$
- 排序的目的：将记录按排序码重排
 - 形成新的有序序列 $R' = \{r'_1, r'_2, \dots, r'_n\}$
 - 相应排序码为 $k' = \{k'_1, k'_2, \dots, k'_n\}$
- 排序码的顺序
 - 其中 $k'_1 \leq k'_2 \leq \dots \leq k'_n$ ，称为不减序
 - 或 $k'_1 \geq k'_2 \geq \dots \geq k'_n$ ，称为不增序

正序 vs. 逆序



- “正序” 序列：待排序序列正好符合排序要求
- “逆序” 序列：把待排序序列逆转过来，正好符合排序要求
- 例如，要求不升序列

■ 08 12 34 96

■ 96 34 12 08

逆序！

正序！



排序的稳定性

□ 稳定性

- 存在多个具有相同排序码的记录
- 排序后这些记录的相对次序保持不变

□ 例如，

- 34 12 34' 08 96
- 08 12 34 34' 96

稳定！



排序的稳定性

□ 稳定性

- 存在多个具有相同排序码的记录
- 排序后这些记录的相对次序保持不变

□ 例如，

- 34 12 34' 08 96
- 08 12 34' 34 96



□ 稳定的

- 形式化证明

□ 不稳定，反例说明

- 34 12 34' 08 96
- 08 12 34' 34 96



排序算法的衡量标准

- 时间代价：记录的比较和移动次数
- 空间代价
- 算法本身的繁杂程度



8.2 插入排序

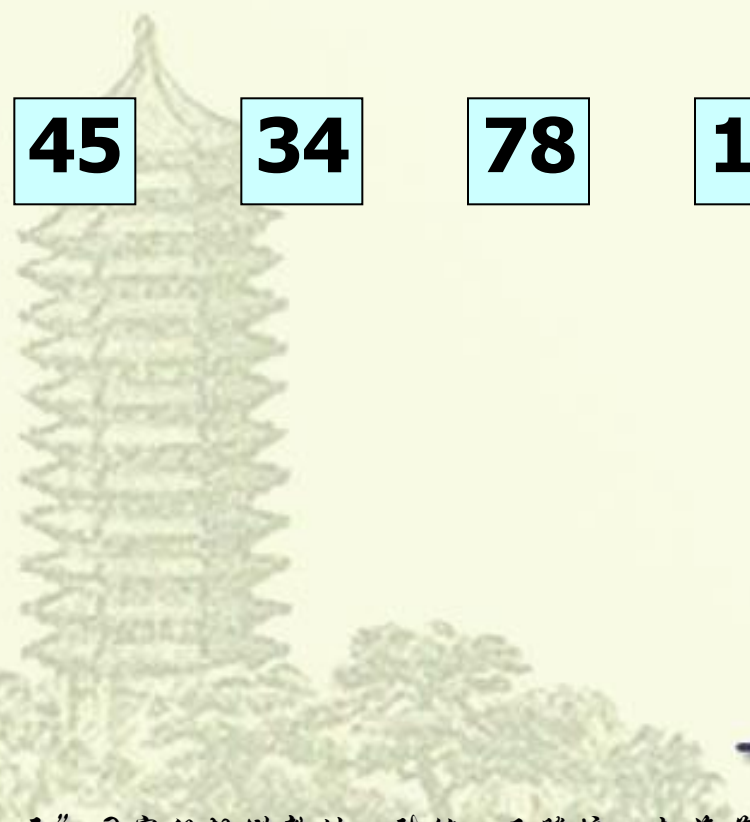
□ 8.2.1 直接插入排序

□ 8.2.2 Shell排序





插入排序动画



45 34 78 12 *34* 32 29 64

插入排序算法

```
template <class Record>
void ImprovedInsertSort (Record Array[], int n) {
    //Array[]为待排序数组, n为数组长度
    Record TempRecord;           // 临时变量
    for (int i=1; i<n; i++) {     // 依次插入第i个记录
        TempRecord=Array[i];
        //从i开始往前寻找记录i的正确位置
        int j = i-1;
        //将那些大于等于记录i的记录后移
        while ((j>=0) &&
            (Compare::lt(TempRecord, Array[j]))) {
            Array[j+1] = Array[j];    j = j - 1;
        }
        //此时j后面就是记录i的正确位置, 回填
        Array[j+1] = TempRecord;
    }
}
```

算法分析



□ 稳定

□ 空间代价： $\Theta(1)$

□ 时间代价：

■ 最佳情况： $n-1$ 次比较， $2(n-1)$ 次移动， $\Theta(n)$

■ 最差情况：比较和交换次数为

$$\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$$

■ 平均情况： $\Theta(n^2)$



8.2.2 Shell排序

□ 直接插入排序的两个性质：

- 在最好情况（序列本身已是有序的）下时间代价为 $\Theta(n)$
- 对于短序列，直接插入排序比较有效

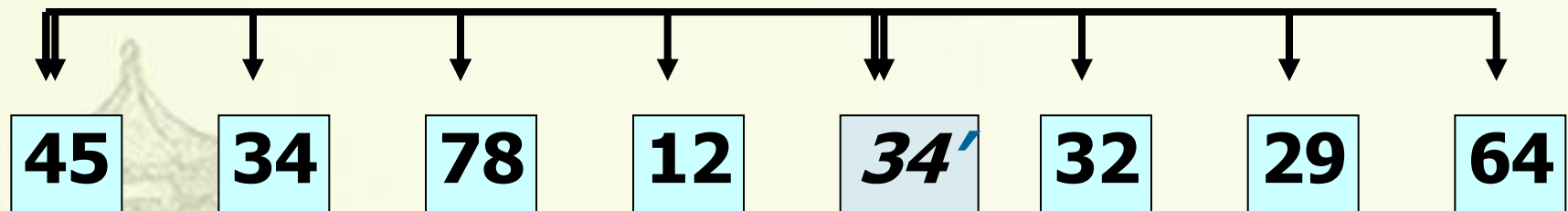
□ Shell排序有效地利用了直接插入排序的两个性质



Shell排序算法思想

- 先将序列转化为若干小序列，在这些小序列内进行插入排序
- 逐渐扩大小序列的规模，而减少小序列个数，使得待排序序列逐渐处于更有序的状态
- 最后对整个序列进行扫尾直接插入排序，从而完成排序

shell排序动画



“增量每次除以2递减”的Shell排序



```
template <class Record>
void ShellSort(Record Array[], int n)
{ // Shell排序, Array[]为待排序数组, n为数组长度
  int i, delta;
  for (delta = n/2; delta > 0; delta /= 2)
    // 增量delta每次除以2递减
    for (i = 0; i < delta; i++)
      // 分别对delta个子序列进行插入排序
      ModInsSort(&Array[i], n-i, delta); // "&"
      传 Array[i]的地址, 待处理数组长度为n-i
    // 如果增量序列不能保证最后一个delta间距为1, 可
    以安排下面这个扫尾性质的插入排序
  // ModInsSort(Array, n, 1);
}
```

针对增量而修改的插入排序算法



```
template <class Record>
void ModInsSort(Record Array[], int n, int delta)
{ // 修改的插入排序算法，参数delta表示当前的增量
  int i, j;
  for (i = delta; i < n; i += delta)
    // 对子序列中第i个记录，寻找合适的插入位置
    for (j = i; j >= delta; j -= delta) {
      // j以delta为步长向前寻找逆置对进行调整
      if (Array[j] < Array[j-delta]) // 逆置对
        swap(Array, j, j-delta);
      // 交换Array[j]和Array[j-delta]
      else break;
    }
}
```

算法分析



- 不稳定
- 空间代价： $\Theta(1)$
- 增量每次除以2递减，时间代价： $\Theta(n^2)$
- 选择适当的增量序列，可以使得时间代价接近 $\Theta(n)$



Shell排序选择增量序列

□ 增量每次除以2递减”时，效率仍然为 $\Theta(n^2)$

□ 问题：选取的增量之间并不互质

- 间距为 2^{k-1} 的子序列都是由那些间距为 2^k 的子序列组成的
- 上一轮循环中这些子序列都已经排过序了，导致处理效率不高



□ Hibbard增量序列

■ $\{2^k - 1, 2^{k-1} - 1, \dots, 7, 3, 1\}$,

□ Shell(3)和Hibbard增量序列的Shell排序的效率可以达到 $\Theta(n^{3/2})$

□ 选取其他增量序列还可以更进一步减少时间代价



8.3 选择排序

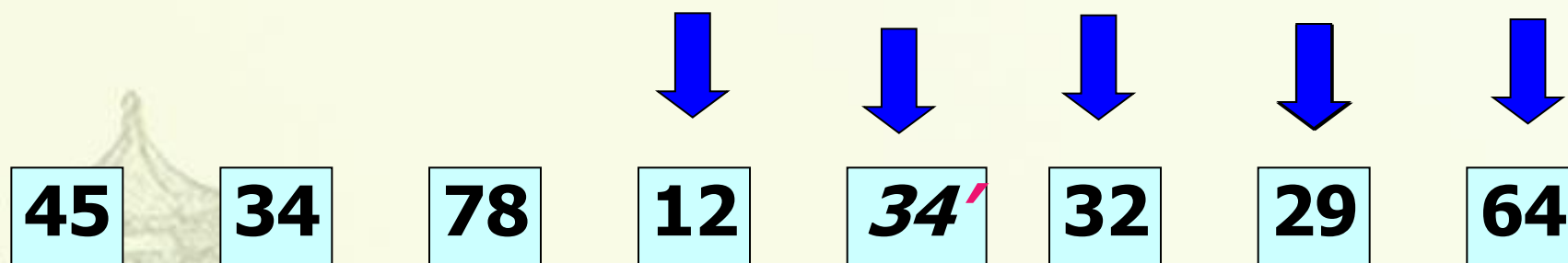
□ 8.3.1 直接选择排序

- 选出剩下的未排序记录中的最小记录，然后直接与数组中第 i 个记录交换，比冒泡排序减少了移动次数

□ 8.3.2 堆排序

- 堆排序：基于最大值堆来实现

直接选择排序动画





直接选择排序

```
template <class Record>
void SelectSort(Record Array[], int n)
{ // 依次选出第i小的记录，即剩余记录中最小的那个
  for (int i=0; i<n-1; i++) {
    // 首先假设记录i就是最小的
    int Smallest = i;
    // 开始向后扫描所有剩余记录
    for (int j=i+1; j<n; j++)
      // 如果发现更小的记录，记录它的位置
      if (Compare::lt(Array[j], Array[Smallest]))
        Smallest = j;
    // 将第i小的记录放在数组中第i个位置
    swap(Array, i, Smallest);
  }
}
```



直接选择排序性能分析

□ 不稳定

□ 空间代价： $\Theta(1)$

□ 时间代价：

- 比较次数： $\Theta(n^2)$ ，与冒泡排序一样
- 交换次数： $n-1$
- 总时间代价： $\Theta(n^2)$



8.3.2 堆排序

□ 选择类内排序

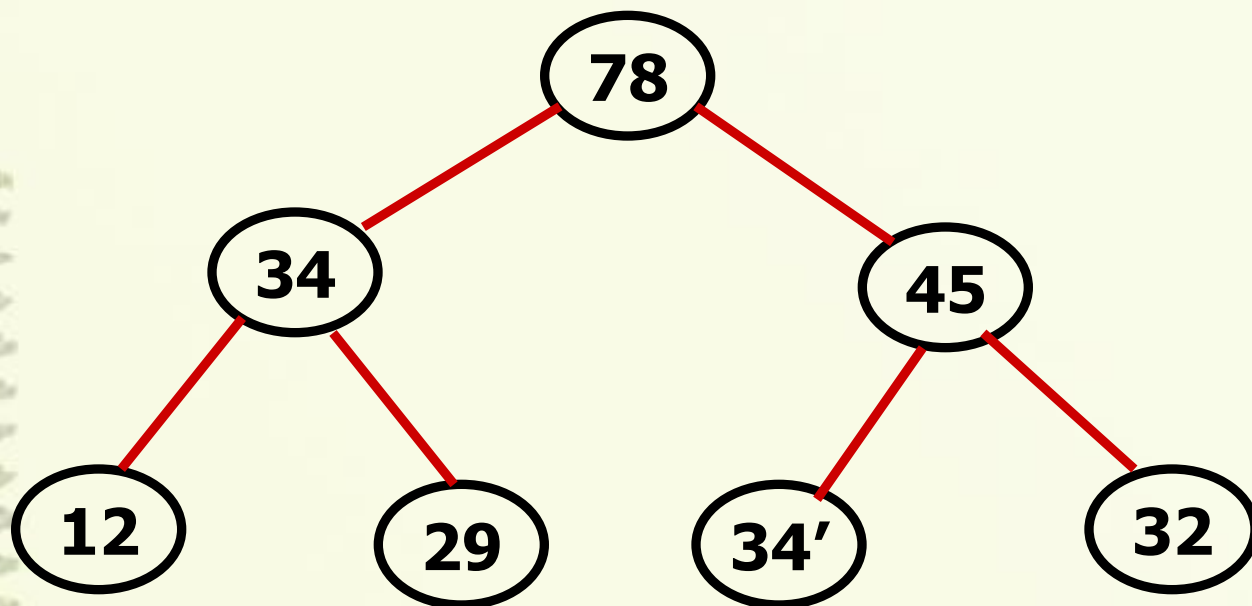
- 直接选择排序：直接从剩余记录中线性查找最大记录
- 堆排序：基于最大值堆来实现，效率更高

□ 选择类外排序

- 置换选择排序
- 赢者树、败方树

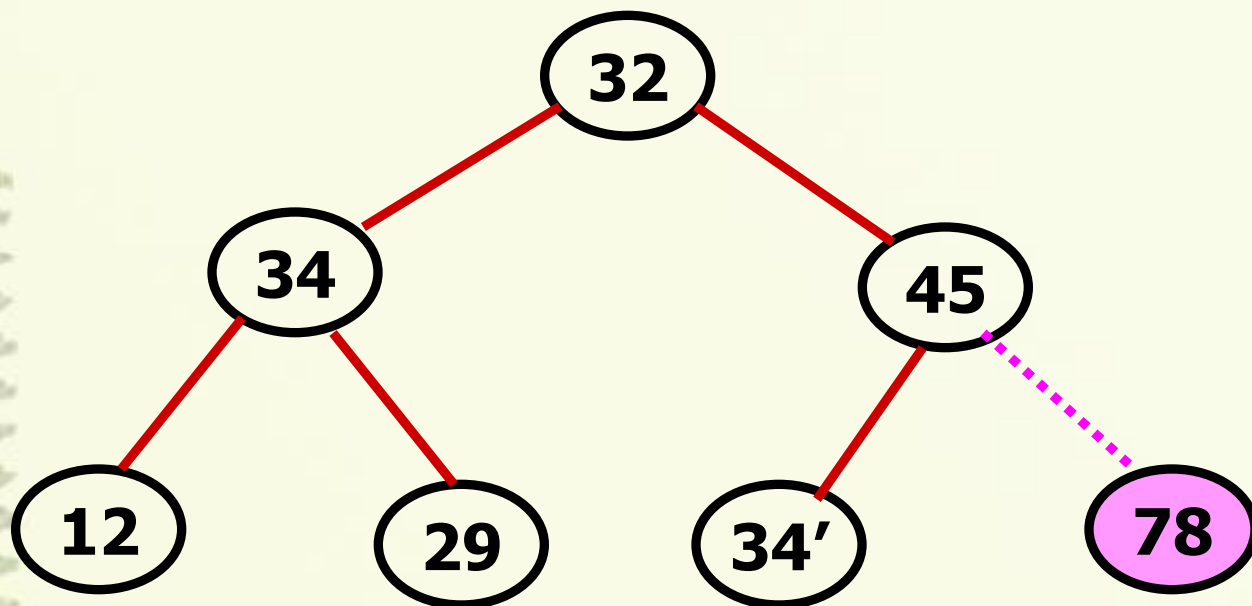


最大值堆排序过程示意图



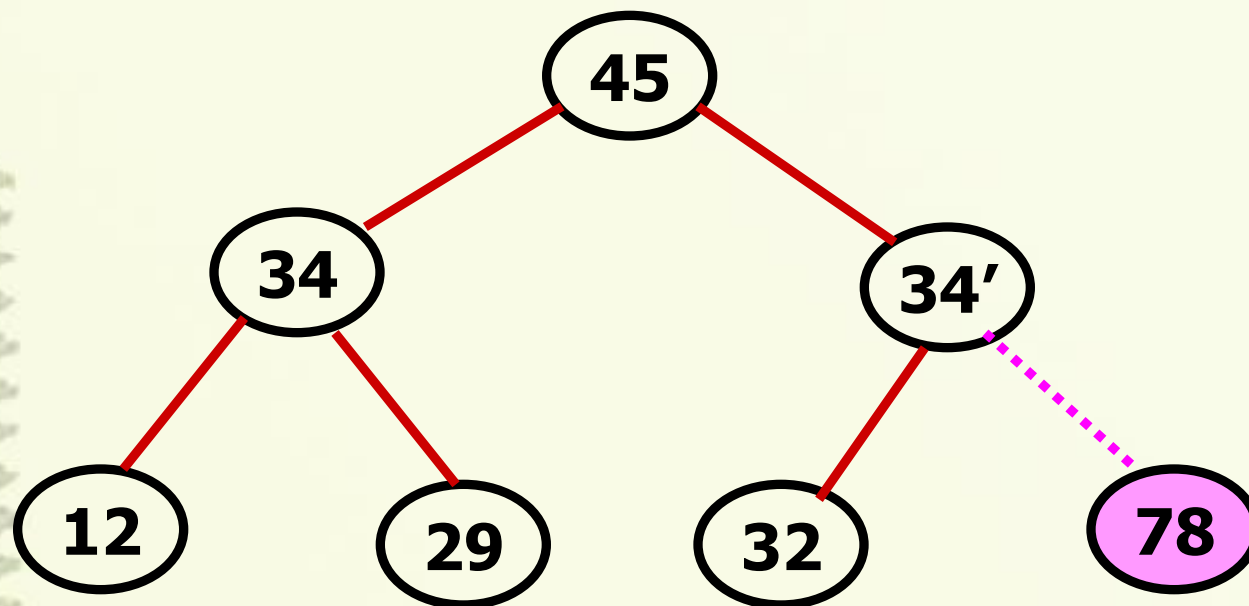


最大值堆排序过程示意图



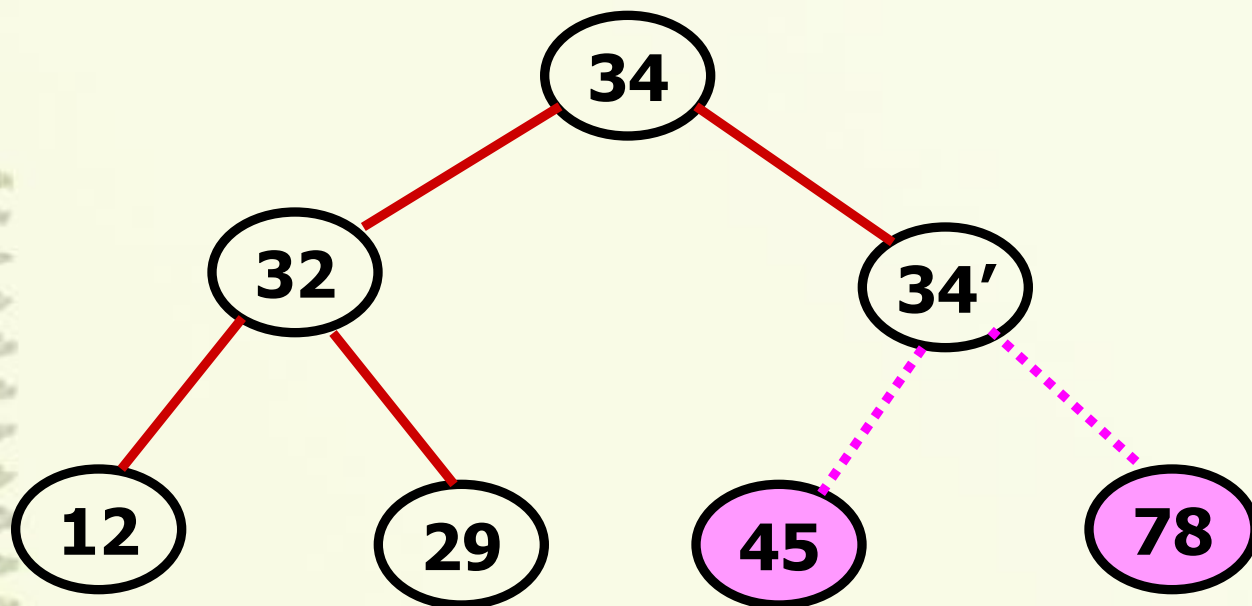


最大值堆排序过程示意图





最大值堆排序过程示意图





堆排序算法

```
template <class Record>
void sort(Record Array[], int n)
{
    int i;
    MaxHeap<Record> max_heap =
        MaxHeap<Record>(Array,n,n); // 建堆
    for (i = 0; i < n-1; i++)
        // 依次找出剩余记录中的最大记录，即堆顶
        max_heap.RemoveMax();
        // 算法操作n-1次，最小元素不需要出堆
}
```




算法分析

- 建堆: $\Theta(n)$
- 删除堆顶: $\Theta(\log i)$
- 一次建堆, n 次删除堆顶
- 总时间代价为 $\Theta(n \log n)$
- 空间代价为 $\Theta(1)$



8.4 交换排序

□ 8.4.1 冒泡排序

□ 8.4.2 快速排序





8.4.1 冒泡排序

□ 算法思想

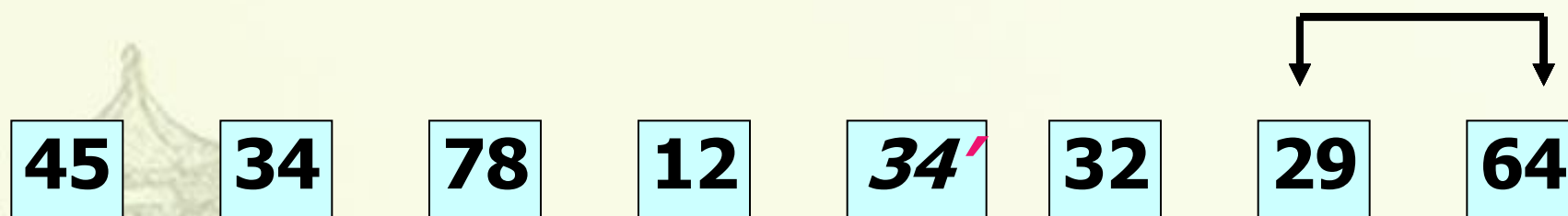
- 不停地比较相邻的记录，如果不满足排序要求，就交换相邻记录，直到所有的记录都已经排好序

- 检查每次冒泡过程中是否发生过交换，如果没有，则表明整个数组已经排好序了，排序结束

- 避免不必要的比较



冒泡排序动画





冒泡排序算法

```
template <class Record>
void BubbleSort(Record Array[], int n) {    // 优化
    的冒泡排序, Array[]为待排序数组, n为数组长度
    bool NoSwap;                          // 是否发生了交换的标志
    int i, j;
    for (i = 0; i < n-1; i++) {
        NoSwap = true;                    // 标志初始为真
        for (j = n-1; j > i; j--)
```



冒泡排序算法

```
for (j = n-1; j > i; j--)  
    if (Array[j] < Array[j-1]) { // 判断 (Array[j-1],  
        Array[j]) 是否逆置  
        swap(Array, j, j-1); // 交换逆置对 Array[j], Array[j-1]  
        NoSwap = false; // 发生了交换, 标志变为假  
    }  
    if (NoSwap) // 如果没发生过交换, 表示已排好序, 结束算法  
        return;  
}
```




算法分析

□ 稳定

□ 空间代价： $\Theta(1)$

□ 时间代价：
■ 比较次数： $\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = \Theta(n^2)$

■ 交换次数最多为 $\Theta(n^2)$ ，最少为0，平均为 $\Theta(n^2)$ 。

■ 最大，平均时间代价均为 $\Theta(n^2)$ 。

■ 最小时间代价为 $\Theta(n)$ ：最佳情况下只运行第一轮循环



8.4.2 快速排序

□ 算法思想：

- 选择轴值（pivot）
- 将序列划分为两个子序列L和R，使得L中所有记录都小于或等于轴值，R中记录都大于轴值
- 对子序列L和R递归进行快速排序

□ 20世纪十大算法

- 1962 London的Elliot Brothers Ltd的Tony Hoare提出的快速排序

□ 基于分治法的排序：快速、归并



分治策略的基本思想

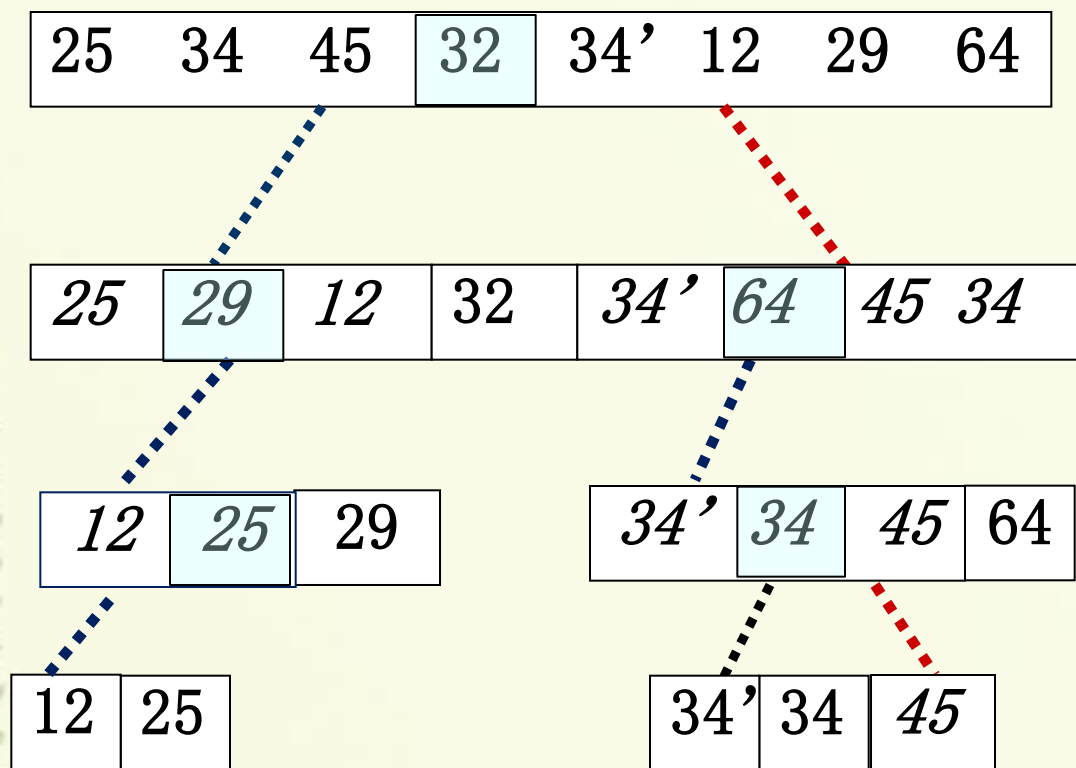
□ 分治策略的实例

- BST查找、插入、删除算法
- 快速排序、归并排序
- 二分检索

□ 主要思想

- 划分
- 求解子问题(子问题不重叠)
- 综合解

快速排序分治思想



最终排序结果: **12 25 29 32 34' 34 45 64**



轴值选择

□ 尽可能使L, R长度相等

□ 选择策略:

- 选择最左边记录
- 随机选择
- 选择平均值



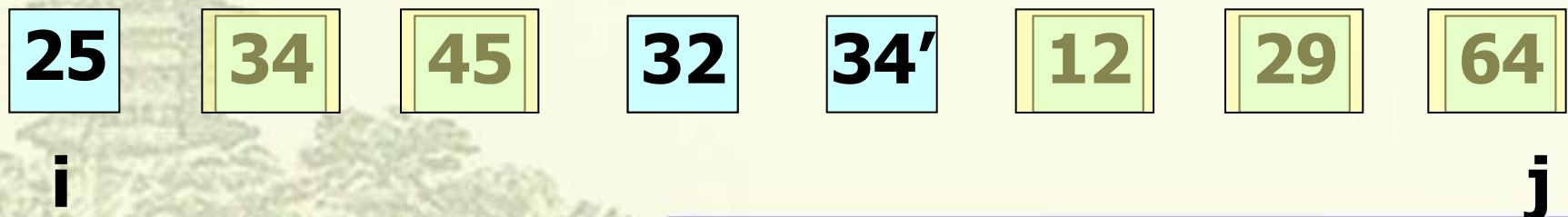
分割过程 (Partition)

□ 整个快速排序的关键，轴值位于正确位置，分割后使得

- L中所有记录位于轴值左边
- R中记录位于轴值右边

一次分割过程

- 选择轴值并存储轴值
- 最后一个元素放到轴值位置
- 初始化下标*i*, *j*, 分别指向头尾
- *i*递增直到遇到比轴值大的元素, 将此元素覆盖到*j*的位置;
*j*递减直到遇到比轴值小的元素, 将此元素覆盖到*i*的位置
- 重复上一步直到*i*==*j*, 将轴值放到*i*的位置, 完毕





快速排序算法

```
template <class Record>
void QuickSort(Record Array[], int left, int right)
{ // Array[]为待排序数组, left,right分别为数组两端
    if (right <= left)
        return; // 如果子序列中只有0或1个记录, 就不需排序
    int pivot = SelectPivot(left, right); // 选择轴值
    swap(Array, pivot, right); // 分割前先将轴值放到数组末端
    pivot = Partition(Array, left, right); // 分割后轴值已到达正确位置
    // 对左子序列进行递归快速排序
    QuickSort(Array, left, pivot-1);
    // 对右子序列进行递归快速排序
    QuickSort(Array, pivot +1, right);
}
```



轴值选择函数

```
int SelectPivot(int left, int right)
{ // 选择轴值, 参数left,right分别表示序列的左右端下标
  return (left+right)/2; // 选择中间记录作为轴值
}
```



分割函数

```
template <class Record>
int Partition(Record Array[], int left, int right)
{ // 分割函数，分割后轴值已到达正确位置
    int l = left; // l为左指针，r为右指针
    int r = right;
    Record TempRecord = Array[r];
    // 将轴值存放在临时变量中
```



```
while (l != r) { // 开始分割, l, r 不断向中间移动, 直到相遇
    // l 指针向右移动, 越过那些小于等于轴值的记录
    // 直到找到一个大于轴值的记录
    while (Array[l] <= TempRecord && r > l)
        // "<=" 也可以改写为 "<", 但增加记录移动
        l++;
    if (l < r) { // 若 l, r 尚未相遇, 将逆置元素换到右边空位
        Array[r] = Array[l];
        r--;    // r 指针向左移动一步
    }
}
```



```
// r指针向左移动, 越过那些大于等于轴值的记录
```

```
// 直到找到一个小于轴值的记录
```

```
while (Array[r] >= TempRecord && r > l)
```

```
// ">="也可以改写为 ">", 但增加记录移动
```

```
    r--;
```

```
    if (l < r) { // 若l,r尚未相遇, 将逆置元素换到左边空位
```

```
        Array[l] = Array[r];
```

```
        l++; // l指针向右移动一步
```

```
    }
```

```
} //end while
```

```
Array[l] = TempRecord;
```

```
// 把轴值回填到分界位置l上
```

```
return l;
```

```
// 返回分界位置l
```

```
}
```




时间代价

□ 长度为 n 的序列，时间为 $T(n)$

■ $T(0) = T(1) = 1$

□ 选择轴值时间为常数

□ 分割时间为 cn

■ 分割后长度分别为 i 和 $n-i-1$

■ 对子序列进行快速排序所需时间分别为 $T(i)$ 和 $T(n-1-i)$

□ 求解递推方程

$$T(n) = T(i) + T(n-1-i) + cn$$



最差情况

$$T(n) = T(n-1) + cn$$

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

M

$$T(2) = T(1) + c(2)$$

□ 总的时间代价为：

$$T(n) = T(1) + c \sum_{i=2}^n i = \Theta(n^2)$$



最佳情况

$$T(n) = 2T(n/2) + cn$$

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c$$

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c$$

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c$$

M

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$



□ $\log n$ 次分割

$$\frac{T(n)}{n} = \frac{T(1)}{1} + c \log n$$

$$T(n) = cn \log n + n = \Theta(n \log n)$$



□ 假设在每次分割时，轴值处于最终排序好的数组中位置的概率是一样的

■ 也就是说，轴值将数组分成长度为0和 $n-1$ ，1和 $n-2$ ，...的子序列的概率是相等的，都为 $1/n$



□ $T(i)$ 和 $T(n-1-i)$ 的平均值均为

$$T(i) = T(n-1-i) = \frac{1}{n} \sum_{k=0}^{n-1} T(k)$$

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k)) = cn + \frac{2}{n} \sum_{k=0}^{n-1} T(k)$$



$$nT(n) = (n+1)T(n-1) + 2cn - c$$

$$T(n) = \Theta(n \log n)$$

算法分析



□ 最差情况：

- 时间代价： $\Theta(n^2)$
- 空间代价： $\Theta(n)$

□ 最佳情况：

- 时间代价： $\Theta(n \log n)$
- 空间代价： $\Theta(\log n)$

□ 平均情况：

- 时间代价： $\Theta(n \log n)$
- 空间代价： $\Theta(\log n)$

算法分析（续）



□ 不稳定

□ 可能优化：

- 轴值选择
 - RQS
- 小子串不递归
- 消除递归



教材上的优化快速排序

□ 子数组小于某个长度（阈值 $n=28$ ）时，
不递归

■ 块与块之间有序

□ 最后对整个数组进行一次插入排序



8.5 归并排序

□ 算法思想

- 简单地将原始序列划分为两个子序列
- 分别对每个子序列递归排序
- 最后将排好序的子序列合并为一个有序序列，即归并过程

归并思想



先
划
分

再
归
并

(25 34 45 32 78 12 34' 64)

(25 34 45 32)(78 12 34' 64)

(25 34)(45 32)(78 12)(34' 64)

(25)(34)(45)(32)(78)(12)(34')(64)

(25 34)(32 45)(12 78)(34' 64)

(25 32 34 45)(12 34' 64 78)

(12 25 32 34 34' 45 64 78)



两路归并排序

```
template <class Record>
void MergeSort(Record Array[], Record TempArray[], int
    left, int right)
{ // Array为待排序数组, left, right两端
    int middle;
    if (left < right) { // 序列中只有0或1个记录, 不用排序
        middle = (left + right) / 2; // 从中间划为两个子序列
        // 对左边一半进行递归
        MergeSort(Array, TempArray, left, middle);
        // 对右边一半进行递归
        MergeSort(Array, TempArray, middle+1, right);
        Merge(Array, TempArray, left, right, middle); // 归并
    }
}
```



归并函数

// 两个有序子序列都从左向右扫描，归并到新数组

```
template <class Record>
```

```
void Merge(Record Array[], Record TempArray[], int  
    left, int right, int middle) {
```

```
    int i, j, index1, index2;
```

```
    for (j = left; j <= right; j++)
```

```
        // 将数组暂存入临时数组
```

```
        TempArray[j] = Array[j];
```

```
    index1 = left;                // 左边子序列的起始位置
```

```
    index2 = middle+1;           // 右边子序列的起始位置
```

```
    i = left;                    // 从左开始归并
```



```
while (index1 <= middle && index2 <= right) {  
    // 取较小者插入合并数组中  
    if (TempArray[index1] <= TempArray[index2])// 相等时左  
        边优先  
        Array[i++] = TempArray[index1++];  
    else Array[i++] = TempArray[index2++];  
}  
while (index1 <= middle)  
    // 只剩左序列，可以直接复制  
    Array[i++] = TempArray[index1++];  
while (index2 <= right)  
    // 与上个循环互斥，直接复制剩余的右序列  
    Array[i++] = TempArray[index2++];  
}
```

R.Sedgewick 优化归并思想



先
划
分

再
归
并

(25 34 45 32 78 12 34' 64)

(25 34 45 32)(78 12 34' 64)

(25 34)(45 32)(78 12)(34' 64)

(25)(34)(45)(32)(78)(12)(34')(64)

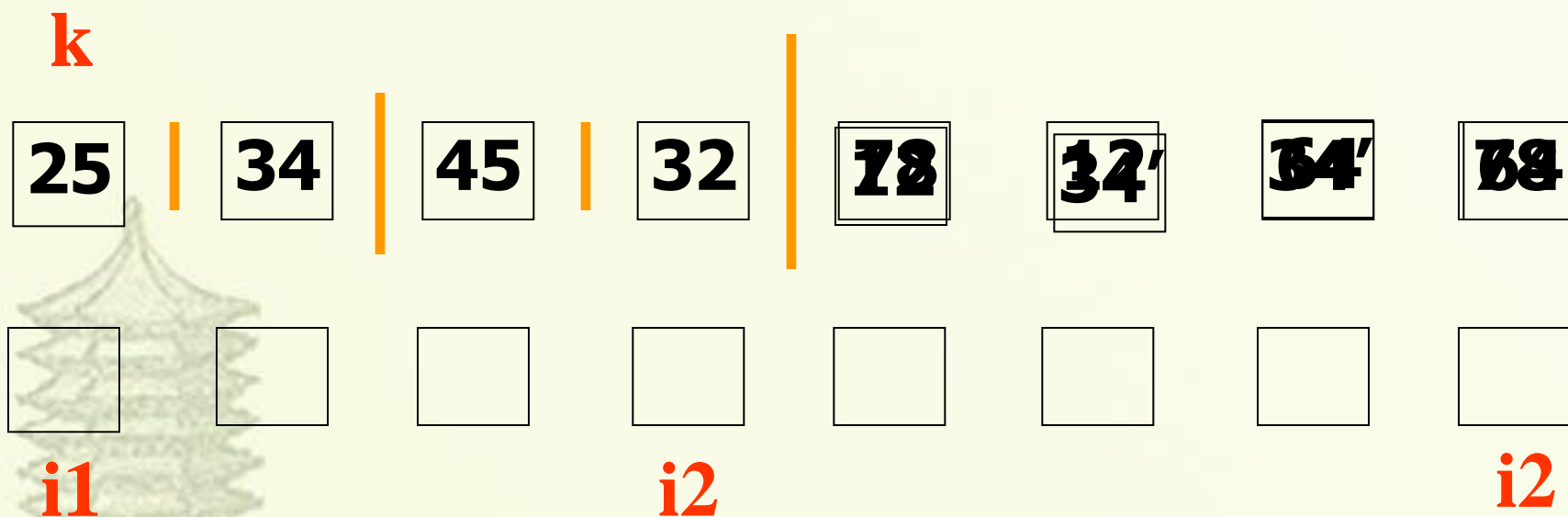
(25 34)(45 32)(12 78)(64 34')

(25 32 34 45)(78 64 34' 12)

(12 25 32 34 34' 45 64 78)



R. Sedgewick 优化归并





优化的归并排序

```
#define THRESHOLD 28
template <class Record>
void ModMergeSort(Record Array[], Record TempArray[],
    int left, int right)
{ // Array为待排序数组, left, right两端
    int middle;
    // 如果序列长度大于阈值(28最佳), 递归进行归并
    if (right-left+1 > THRESHOLD) {
        middle = (left + right) / 2; // 从中间划为两个子序列
        // 对左边一半进行递归
        ModMergeSort(Array, TempArray, left, middle);
```




优化的归并排序 (cont.)

```
// 对右边一半进行递归
ModMergeSort(Array, TempArray ,middle+1,right);
// 对相邻的有序序列进行归并
ModMerge(Array, TempArray ,left,right,middle);
}
else InsertSort(&Array[left],right-left+1);
// 小长度子序列进行插入排序, "&"传 Array[left]的地址
}
```



优化的归并函数

// 优化的Sedgwick两个有序子序列归并，右子序列逆置了，都从两端向中间扫描，归并到新数组

```
template <class Record>
```

```
void ModMerge(Record Array[],Record  
TempArray[],int left,int right,int middle) {
```

```
    //归并过程
```

```
    int index1,index2; // 两个子序列的起始位置
```

```
    int i,j,k;
```

```
    for (i = left; i <= middle; i++)
```

```
        // 复制左边的子序列
```

```
        TempArray[i] = Array[i];
```



```
// 复制右边的子序列，但顺序颠倒过来
for (j = 1; j <= right-middle; j++)
    TempArray[right-j+1] = Array[j+middle];
// 开始归并，取较小者插入合并数组中
for (index1 = left, index2 = right, k = left; k <=
right; k++)
    // 为保证稳定性，相等时左边优先
    if (TempArray[index1] <= TempArray[index2])
        Array[k] = TempArray[index1++];
    else Array[k] = TempArray[index2--];
}
```



时间代价

- 划分时间
- 两个子序列的排序时间
- 归并时间, n

算法分析



- 空间代价： $\Theta(n)$
- 总时间代价： $\Theta(n \log n)$
- 不依赖于原始数组的输入情况，最大、最小以及平均时间代价均为 $\Theta(n \log n)$ 。

算法分析（续）



□ 稳定

□ 优化：

- 同优化的快速排序一样，对基本已排序序列直接插入排序
- R.Sedgewick优化：归并时从两端开始处理，向中间推进，简化了边界判断



8.6 分配排序和基数排序

- 不需要进行纪录之间两两比较
- 需要事先知道记录序列的一些具体情况





8.6.1 桶式排序

- 事先知道序列中的记录都位于某个小区间段 $[0, m)$ 内
- 将具有相同值的记录都分配到同一个桶中，然后依次按照编号从桶中取出记录，组成一个有序序列



$$\text{count}[i] = \text{count}[i-1] + \text{count}[i];$$

待排数组: 7 3 8 9 6 1 8' 1' 2

第一趟count:

0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

+ + + +

后继起始下标:

0	2	3	4	4	4	5	6	8	9
---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9



桶排序示意

待排数组: 7 3 8 9 6 1 8' 1' 2

第一趟count

0	1	2	3	4	5	6	7	8	9
0	2	1	1	0	0	1	1	2	1

后继起始下标:

0	0	2	4	4	4	5	6	7	9
---	---	---	---	---	---	---	---	---	---

收集:

0	1	2	3	4	5	6	7	8
1	1'	2	3	6	7	8	8'	9



桶式排序算法

```
template <class Record>
void BucketSort(Record Array[], int n, int max)
{ // Array数组长度为n, 所有记录都位于区间[0,max)上
  Record *TempArray = new Record[n];
  // 临时数组
  int *count = new int[max];
  // 桶容量计数器, count[i] 存储了第i个桶中的记录数
  int i;
  for (i = 0; i < n; i++)    // 把序列复制到临时数组
    TempArray[i] = Array[i];
```



```
for (i = 0; i < max; i++)    // 所有计数器初始都为0
    count[i] = 0;
for (i = 0; i < n; i++)    // 统计每个取值出现的次数
    count[Array[i]]++;
for (i = 1; i < max; i++)    // 统计小于等于i的元素个数
    count[i] = count[i-1] + count[i];
    // count[i]记录了i+1的开始位置
// 从尾部开始按顺序输出，保证排序的稳定性
for (i = n-1; i >= 0; i--)
    Array[--count[TempArray[i]]] = TempArray[i];
    // TempArray[i]放到Array[count[TempArray[i]-1]中
}
```


算法分析



□ 数组长度为 n , 所有记录区间 $[0, m)$ 上

□ 时间代价:

- 统计计数时: $\Theta(n+m)$
- 输出有序序列时循环 n 次
- 总的时间代价为 $\Theta(m+n)$
- 适用于 m 相对于 n 很小的情况

□ 空间代价:

- m 个计数器, 长度为 n 的临时数组, $\Theta(m+n)$

□ 稳定



8.6.2 基数排序

- 桶式排序只适合 m 很小的情况
- 当 m 很大时，可以将一个记录的值即排序码拆分为多个部分来进行比较——基数排序

基数排序



□ 假设长度为 n 的序列

$$R = \{ r_0, r_1, \dots, r_{n-1} \}$$

记录的排序码 K 包含 d 个子排序码

$$K = (k_{d-1}, k_{d-2}, \dots, k_1, k_0)$$



- R对排序码($k_{d-1}, k_{d-2}, \dots, k_1, k_0$)有序
- 对于任意两个记录 R_i, R_j ($0 \leq i < j \leq n-1$), 都满足

$$(k_{i,d-1}, k_{i,d-2}, \dots, k_{i,1}, k_{i,0}) \leq (k_{j,d-1}, k_{j,d-2}, \dots, k_{j,1}, k_{j,0})$$

- 其中 k_{d-1} 称为最高排序码
- k_0 称为最低排序码

例子



例如：如果我们要对0到9999之间的整数进行排序

- 将四位数看作是由四个排序码决定，即千、百、十、个位，其中千位为最高排序码，个位为最低排序码。基数 $r=10$ 。
- 可以按千、百、十、个位数字依次进行4次桶式排序
- 4趟分配排序后，整个序列就排好序了



基数排序分为两类

- 高位优先法 (MSD, Most Significant Digit first)
- 低位优先法 (LSD, Least Significant Digit first)



□ 黑桃(S) > 红心(H) >
方片(D) > 梅花(C)

□ S3 HJ C8 H9 S9 D3 C1 D7



S3 HJ C8 H9 S9 D3 C1 D7

□ MSD方法（递归分治）

- 先按花色：C8 C1 D3 D7 HJ H9 S3 S9
- 再按面值：C1 C8 D3 D7 H9 HJ S3 S9

□ LSD方法

- 先按面值：C1 S3 D3 D'7 C'8 H9 S'9 H'J
- 再按花色：C1 C'8 D3 D'7 H9 H'J S3 S'9

• 要求稳定排序！

高位优先法 (*MSD, Most Significant Digit first*)



- 先对高位 k_{d-1} 进行桶式排序，将序列分成若干个桶中
- 然后对每个桶再按次高位 k_{d-2} 进行桶式排序，分成更小的桶
- 依次重复，直到对 k_0 排序后，分成最小的桶，每个桶内含有相同的排序码($k_{d-1}, k_{d-2}, \dots, k_1, k_0$)
- 最后将所有的桶依次连接在一起，成为一个有序序列
- 这是一个分、分、...、分、收的过程

低位优先法 (*LSD, Least Significant Digit first*)



- 从最低位 k_0 开始排序
- 对于排好的序列再用次低位 k_1 排序；
- 依次重复，直至对最高位 k_{d-1} 排好序后，整个序列成为有序的
- 这是一个分、收；分、收；...；分、收的过程
 - 比较简单，计算机常用



基数排序的实现

- 基于顺序存储
- 基于链式存储





基于顺序存储的基数排序

只讨论LSD

- 原始输入数组 R 的长度为 n
- 基数为 r
- 排序码个数为 d

初始数组内容: 97 53 88 59 26 41 88' 31 22

0 1 2 3 4 5 6 7 8 9

第一趟: count

0	2	1	1	0	0	1	1	2	1
---	---	---	---	---	---	---	---	---	---

0 1 2 3 4 5 6 7 8 9

按 count 分配桶:

0	2	3	4	4	4	5	6	8	9
---	---	---	---	---	---	---	---	---	---

收集:

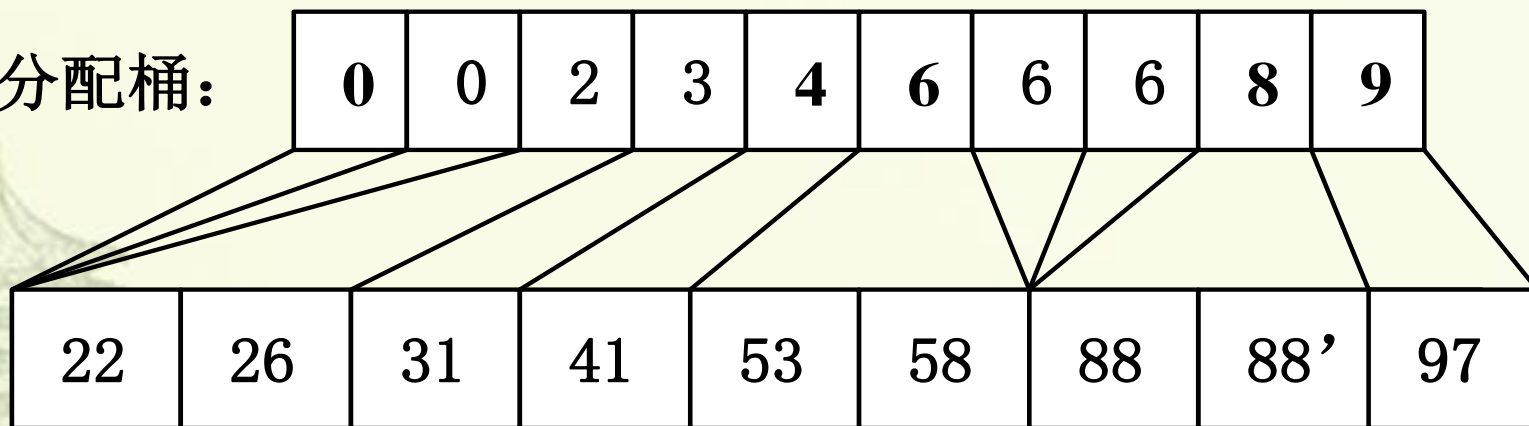
41	31	22	53	26	97	88	88'	59
----	----	----	----	----	----	----	-----	----

(a) 第一趟分配个位

第二趟: count

0	1	2	3	4	5	6	7	8	9
0	0	2	1	1	2	0	0	2	1

按 count 分配桶:



收集:

最终排序结果: 22 26 31 41 53 59 88 88' 97

(b) 第二趟分配十位



基于数组的基数排序

```
template <class Record>
void RadixSort(Record Array[], int n, int d, int r)
{ // n为数组长度, d为排序码个数, r为基
    // 辅助排序的临时数组
    Record *TempArray = new Record[n];
    int *count = new int[r];
        // 计数器, count[i] 存储了第i个桶中的记录数
    int i, j, k;
    int Radix = 1; // 模进位, 用于取Array[j]的第i位
```



// 分别对第*i*个排序码进行分配

for (i = 1; i <= d; i++) {

for (j = 0; j < r; j++)

count[j] = 0;

// 初始计数器均为0

for (j = 0; j < n; j++) {

// 统计每个桶中的记录数

k = (Array[j] / Radix) % r; // 取第*i*位排序码

count[k]++;

// 相应计数器加1

}

for (j = 1; j < r; j++)

// 将TempArray中的位置依次分配给*r*个桶

count[j] = count[j-1] + count[j];

// count[i]记录了*i*+1的开始位置

// 元素*i*应该从Array[count[i]-1]往前追溯

// 存放count[i]-count[i-1]个



```
// 从数组尾部，把记录收集到相应桶
for (j = n-1; j >= 0; j--) {
    // 取第j位排序码，Array[j]应该放到count[k]-1处
    k = (Array[j] / Radix) % r;
    count[k]--;    // 桶剩余量的计数器减1
    // Array[j]放到TempArray中，下标位置在上面已经被减1
    TempArray[count[k]] = Array[j];
}
for (j = 0; j < n; j++)
    // 将临时数组中的内容复制到Array中
    Array[j] = TempArray[j];
Radix *= r;    // 往左进一位，修改模Radix
}
```

算法分析



□ 空间代价：

■ 临时数组, n

■ r 个计数器

□ 总空间代价 $\Theta(n+r)$

算法分析（续）



□ 时间代价

- 桶式排序： $\Theta(n+r)$
- d次桶式排序
- $\Theta(d \cdot (n+r))$

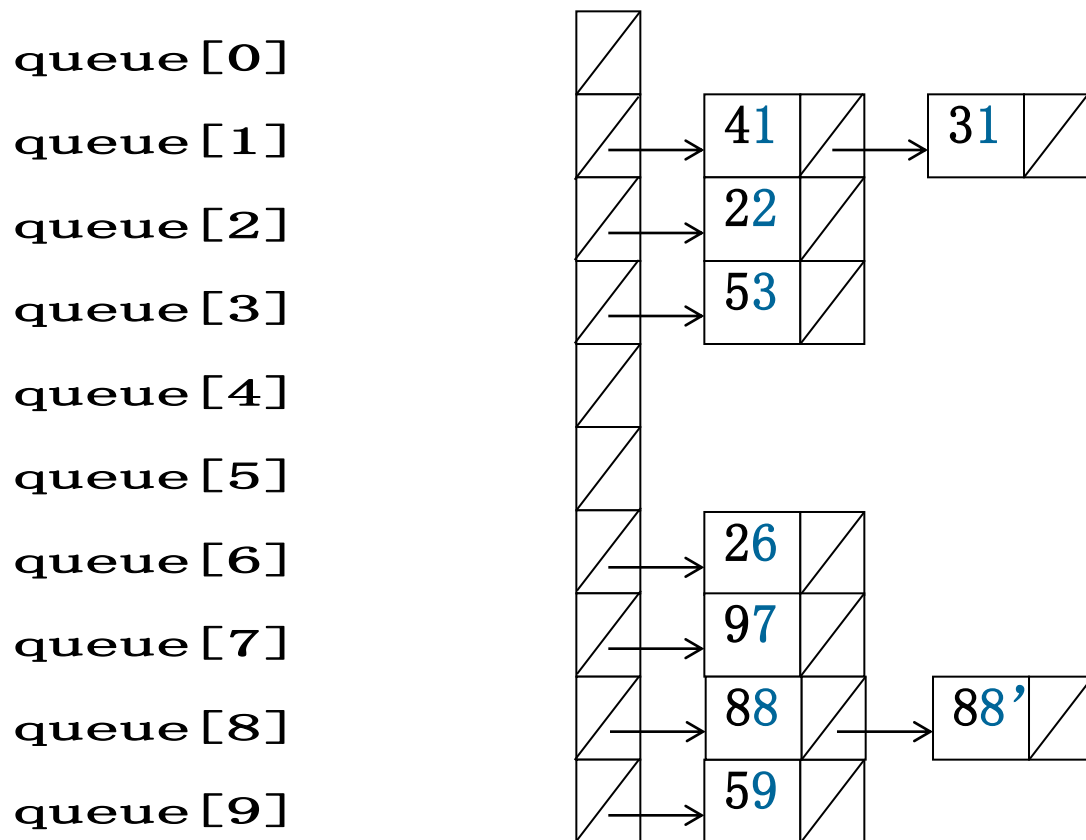


基于静态链的基数排序

- 将分配出来的子序列存放在 r 个(静态链组织的)队列中
- 链式存储避免了空间浪费情况

97	53	88	59	26	41	88'	31	22
----	----	----	----	----	----	-----	----	----

(a) 初始链表内容

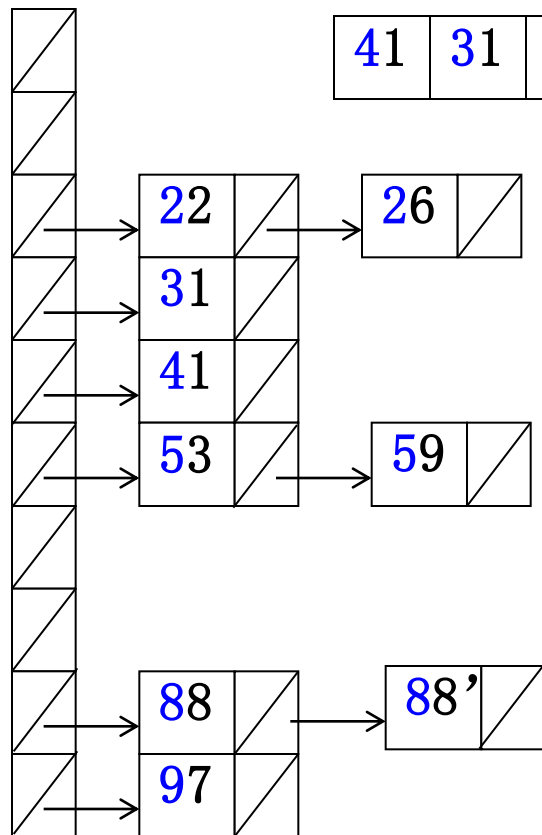


(b) 第一趟分配

(c) 第一趟收集

41	31	22	53	26	97	88	88'	59
----	----	----	----	----	----	----	-----	----

queue[0]
queue[1]
queue[2]
queue[3]
queue[4]
queue[5]
queue[6]
queue[7]
queue[8]
queue[9]



(d) 第二趟分配

(d) 第二趟分配

(e) 第二趟收集结果
(最终结果)

22	26	31	41	53	59	88	88'	97
----	----	----	----	----	----	----	-----	----



静态队列定义

```
class Node{ //结点类
public:
    int key;    //结点的关键词值
    int next;   //下一个结点在数组中的下标
};
```

```
class StaticQueue{ //静态队列类
public:
    int head;
    int tail;
};
```



基于静态链的基数排序

```
//静态链实现的基数排序，n为数组长度，d为排序码个数，r为基数
template <class Record>
void RadixSort(Record *Array, int n, int d, int r) {
    int i, first = 0;      // first指向静态链中第一个记录
    StaticQueue *queue = new StaticQueue[r]; // r个桶的静态队列
    // 初始化建链，相邻记录的静态指针链接为单链表
    for (i = 0; i < n-1; i++)
        Array[i].next = i + 1; // 静态指针域设为下一个元素的下标
    Array[n-1].next = -1; // 链尾next为空
```




```
// 对第i个排序码进行分配和收集，一共d趟  
for (i = 0; i < d; i++) {
```

```
    Distribute(Array, first, i, r, queue);
```

```
    Collect(Array, first, r, queue);
```

```
}
```

```
delete[] queue;
```

```
AddrSort(Array, n, first);
```

```
// 线性时间整理静态链表，使得数组按下标有序
```

```
}
```

// 分配过程, A中存放待排序记录, first为静态链中的第一个记录

// i为第i个排序码, r为基数

template <class Record>

void Distribute(Record *Array, int first, int i, int r, StaticQueue
*queue)

```
{
    int j, k, a, curr = first;
    for (j = 0; j < r; j++)           // 初始化r个队列
        queue[j].head = -1;
    while (curr != -1) {               // 对整个静态链进行分配
        k = Array[curr].key;           // 取第i位排序码数字k
        for (a = 0; a < i; a++)
            k = k / r;
        k = k % r;                     // 把Array[curr]分配到第k个桶中
        if (queue[k].head == -1) // 若桶空, Array[curr]就是第一个记录
            queue[k].head = curr;
        else Array[queue[k].tail].next = curr; // 否则加到桶的尾部
        // 当前记录的下标curr被标记为该桶的尾部
        queue[k].tail = curr;
        // 静态指针curr移动一位, 继续分配下一个记录
        curr = Array[curr].next;
    }
}
```



```
// 收集过程，Array中存放待排序记录，first为静态链中的第一个记录
// r为基数
template <class Record>
void Collect(Record *Array, int& first, int r, StaticQueue *queue) {
    int last, k=0; // 已收集到的最后一个记录
    while (queue[k].head == -1) // 找到第一个非空队列
        k++;
    first = queue[k].head;
    last = queue[k].tail;
    while (k < r-1) { // 继续收集下一个非空队列，若k==r-1则已是最后
        k++; // 找下一个非空队列
        // 当前队列k为空，而且k还不是最后的队列r-1
        while (k < r-1 && queue[k].head == -1)
            k++; // 试探下一个队列
        // 将这个非空序列与上一个非空序列连接起来
        if (queue[k].head != -1) {
            Array[last].next = queue[k].head;
            last = queue[k].tail; // 最后一个记录为序列的尾部记录
        }
    }
    Array[last].next = -1; // 收集完毕
}
```



线性时间整理静态链表

```
template <class Record>
void AddrSort(Record *Array, int n, int first) {int i, j;
    j = first;                // j待处理数据下标, 第一次为first
    Record TempRec;
    for (i = 0; i < n-1; i++) { // 循环n-1次, 每次处理第i大记录
        TempRec = Array[j];    // 暂存第i大的记录Array[j]
        Array[j] = Array[i];   // 当前下标i的数据存放到j位置
        Array[i] = TempRec;    // 第i大记录入位
        Array[i].next = j;     // 第i大的记录的next链要保留调换轨迹j
        j = TempRec.next;     // j移动到下一位
        while (j <= i)        // j比i小, 则是轨迹, 顺链找到数据
            j = Array[j].next;
    }
}
```

算法分析



□ 空间代价

- n 个记录空间
- r 个子序列的头尾指针
- $O(n + r)$

□ 时间代价

- 不需要移动记录本身，只需要修改记录的next指针
- $O(d \cdot (n+r))$



空间 vs 时间

□ 速度比较快的排序算法 $O(n \log n)$

- 归并、分配、快速
 - 空间消耗都比较大
- 堆
 - 空间消耗比较小

□ 对于大记录，采用地址排序



数组 vs 链表

□ 一般用数组

□ 静态链表





8.6.3 索引地址排序

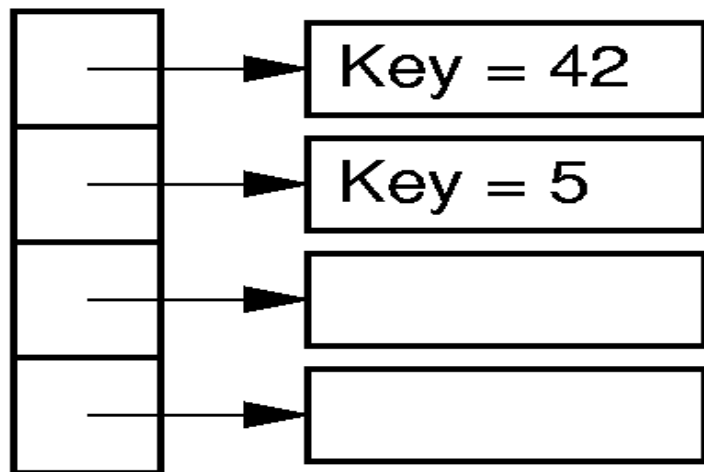
□ 动态链(new出来的指针)

□ 数组

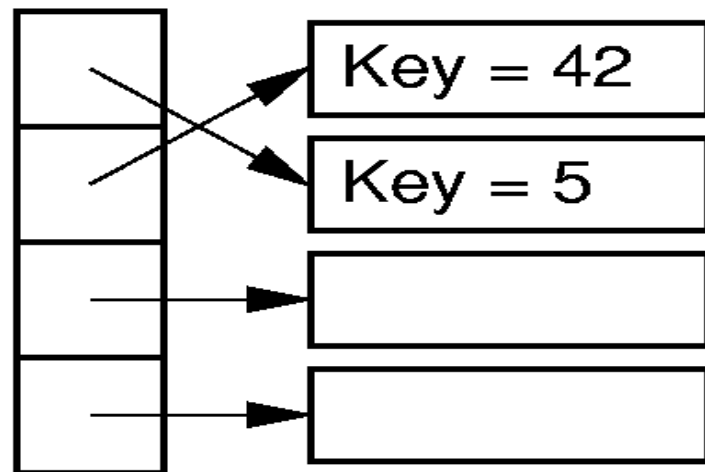
■ 建立索引数组 (通用)

■ 建立静态链表

索引数组



(a)



(b)

交换指针，减少交换记录的次数



索引方法1:

□ 结果下标IndexArray[i]存放的是Array[i]中数据应该待的位置。

□ 也就是说 $\text{Array}[\text{IndexArray}[i]] = \text{Array}[i]$
用另一个数组整理

□ 下标 0 1 2 3 4 5 6 7

□ 排序码 29 25 34 64 34' 12 32 45

□ 结果 2 1 4 7 5 0 3 6

0 1 2 3 4 5 6 7
12 25 29 32 34 34' 45 64



索引方法2:

- 结果下标IndexArray[i]存放的是Array[i]中应该摆放的数据位置。
- 也就是说 $\text{Array}[i] = \text{Array}[\text{IndexArray}[i]]$

用另一个数组整理

- | | | | | | | | | |
|-------|----|----|----|----|-----|----|----|----|
| □ 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| □ 排序码 | 29 | 25 | 34 | 64 | 34' | 12 | 32 | 45 |
| □ 结果 | 5 | 1 | 0 | 6 | 2 | 4 | 7 | 3 |

开新数组 12 25 29 32 34 34' 45 64



不开新数组

□ 只利用原Array和IndexArray

- 可以用一个临时空间
- Record TempRec;

□ 根据IndexArray内容，整理好原Array，使得Array数组按下标有序

- 在 $O(n)$ 的时间内完成

顺链整理



下标	0	1	2	3	4	5	6	7
排序码	29	25	34	64	34'	12	32	45
结果	5	1	0	6	2	4	7	3
整理	2	1	4	7	5	0	3	6

0	1	2	3	4	5	6	7
12	25	29	32	34	34'	45	64



得到索引2的方法？

□ 一般的排序方法都可以

■ 那些赋值（或交换）都换成对index数组的赋值（或交换）

□ 举例：插入排序



```
template<class Record, class int_intCompare>
void AddrSort(Record Array[], int n)
{ //Array[]为待排序数组， n为数组长度
  int *IndexArray = new int[n], TempIndex;
  int i,j,k;
  Record TempRec; //只需一个临时空间
  for (i=0; i<n; i++) IndexArray[i] = i;
  for (i=1; i<n; i++) // 依次插入第i个记录
    for (j=i; j>0; j--) //依次比较，发现逆置就交换
      if ( Compare::lt(Array[IndexArray[j]],
        Array[IndexArray[j-1]]))
        swap(IndexArray, j, j-1);
    else break; //此时i前面记录已排序
```



//根据 *IndexArray* 整理 *Array*

```
for(i=0; i<n; i++) { //调整的代价为O(n)
    j=i;
    TempRec=Array[i];
    while (IndexArray[j]!=i) {
        k=IndexArray[j];
        Array[j]=Array[k];
        IndexArray[j]=j;
        j=k;
    }
    Array[j]=TempRec;
    IndexArray[j]=j;
}
```



得到索引2的方法？

- Rank排序
- 静态链表的基数排序



Rank排序



```
template<class Record, class int_intCompare>
void Rank(Record* array, int listsize, Record *rank)
{
    for(int i=0;i<listsize;i++)
        rank[i] = 0;
    for(i=0;i<listsize;i++)
        for(int j=i+1;j<listsize;j++) {
            if (Compare::gt(Array[i], array[j])
                rank[i]++;
            else
                rank[j]++;
        }
}
```




8.7 排序算法的时间代价

- 8.7.1 简单排序算法的时间代价
- 8.7.2 排序算法的理论和实验时间
- 8.7.3 排序问题的下限



8.7.1 简单排序算法的时间代价

比较次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$



移动次数	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	0	$\Theta(n)$	$\Theta(n)$	0	0	$\Theta(n)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$



总代价	直接插入排序	改进的插入排序	二分法插入排序	冒泡排序	改进的冒泡排序	选择排序
最佳情况	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
平均情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$

原因

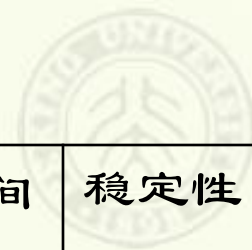


- 一个长度为 n 序列平均有 $n(n-1)/4$ 对逆置
- 任何一种只对相邻记录进行比较的排序算法的平均时间代价都是 $\Theta(n^2)$



8.7.2 排序算法的理论和实验时间

算法	最大时间	平均时间	最小时间	辅助空间 代价	稳定性
直接插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
二分法插入排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(1)$	稳定
冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	稳定
改进的冒泡排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$	稳定
选择排序	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	不稳定



算法	最大时间	平均时间	最小时间	辅助空间 代价	稳定性
Shell排序(3)	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(n^{3/2})$	$\Theta(1)$	不稳定
快速排序	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(\log n)$	不稳定
归并排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	稳定
堆排序	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(1)$	不稳定
桶式排序	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(n+m)$	稳定
基数排序	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(d \cdot (n+r))$	$\Theta(n+r)$	稳定

小结



- n 很小或基本有序时插入排序比较有效
- Shell排序选择增量以3的倍数递减
 - 需要保证最后一趟增量为1
- 综合性能快速排序最佳



测试环境

□ 硬件环境：

- CPU: Intel P4 3G
- 内存: 1G
- 软件环境:
- windows xp
- Visual C++ 6.0



随机生成待排序数组

```
//设置随机种子
inline void Randomize() { srand(1); }
// 返回一个0到n之间的随机整数值
inline int Random(int n)
{ return rand() % (n); }
//产生随机数组
ELEM *sortarray = new ELEM[1000000];
for(int i=0;i<1000000;i++)
sortarray[i]=Random(32003);
```



时间测试

```
#include <time.h>
/* Clock ticks macro - ANSI version */
#define CLOCKS_PER_SEC 1000

clock_t tstart = 0; // Time at beginning
// Initialize the program timer
void Settime() { tstart = clock(); }
// The elapsed time since last Settime()
double Gettime()
{ return (double)((double)clock() - (double)tstart)/
  (double)CLOCKS_PER_SEC; }
```



排序的时间测试

```
Settime();
for (i=0; i<ARRAYSIZE; i+=listsize) {
    sort<int,intintCompare>(&array[i], listsize);
    print(&array[i], listsize);
}
cout << "Sort with list size " << listsize << ",
array size " << ARRAYSIZE << ", and threshold " <<
THRESHOLD << ": " << Gettime() << " seconds\n";
```


数组规模	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
直接插入排序	0.00000047	0.000020	0.001782	0.1752	17.917	——	0.00011	0.35094
选择排序	0.00000110	0.000041	0.002922	0.2778	36.500	——	0.27781	0.29109
冒泡排序	0.00000160	0.000156	0.015620	1.5617	207.69	——	0.00006	2.44840
Shell排序(2)	0.00000156	0.000036	0.000640	0.0109	0.1907	3.0579	0.00156	0.00312
Shell排序(3)	0.00000078	0.000016	0.000281	0.0038	0.0579	0.8204	0.00125	0.00687
堆排序	0.00000204	0.000027	0.000344	0.0042	0.0532	0.6891	0.00406	0.00375
快速排序	0.00000169	0.000021	0.000266	0.0030	0.0375	0.4782	0.00190	0.00199
优化快排/16	0.00000172	0.000020	0.000265	0.0020	0.0235	0.3610	0.00082	0.00088
优化快排/28	0.00000062	0.000011	0.000141	0.0018	0.0235	0.2594	0.00063	0.00063
归并排序	0.00000219	0.000028	0.000375	0.0045	0.0532	0.5969	0.00364	0.00360



数组规模	10	100	1K	10K	100K	1M	10K 正序	10K 逆序
优化归并/16	0.00000063	0.000014	0.000188	0.0030	0.0375	0.4157	0.00203	0.00265
优化归并/28	0.00000062	0.000013	0.000204	0.0027	0.0360	0.4156	0.00172	0.00265
顺序基数/8	0.00000610	0.000049	0.000469	0.0048	0.0481	0.4813	0.00484	0.00469
顺序基数/16	0.00000485	0.000034	0.000329	0.0032	0.0324	0.3266	0.00328	0.00313
链式基数/2	0.00002578	0.000233	0.002297	0.0234	0.2409	3.4844	0.02246	0.02281
链式基数/4	0.00000922	0.000075	0.000719	0.0075	0.0773	1.3750	0.00719	0.00719
链式基数/8	0.00000704	0.000048	0.000466	0.0049	0.0502	0.9953	0.00469	0.00469
链式基数/16	0.00000516	0.000030	0.000266	0.0028	0.0295	0.6570	0.00281	0.00281
链式基数/32	0.00000500	0.000027	0.000235	0.0028	0.0297	0.5406	0.00263	0.00266





8.7.3 排序问题的下限

- 排序问题的时间代价在 $\Omega(n)$ (起码I/O 时间) 到 $O(n \log n)$ (平均, 最差情况)之间
- 基于比较的排序算法的下限也为 $\Theta(n \cdot \log n)$

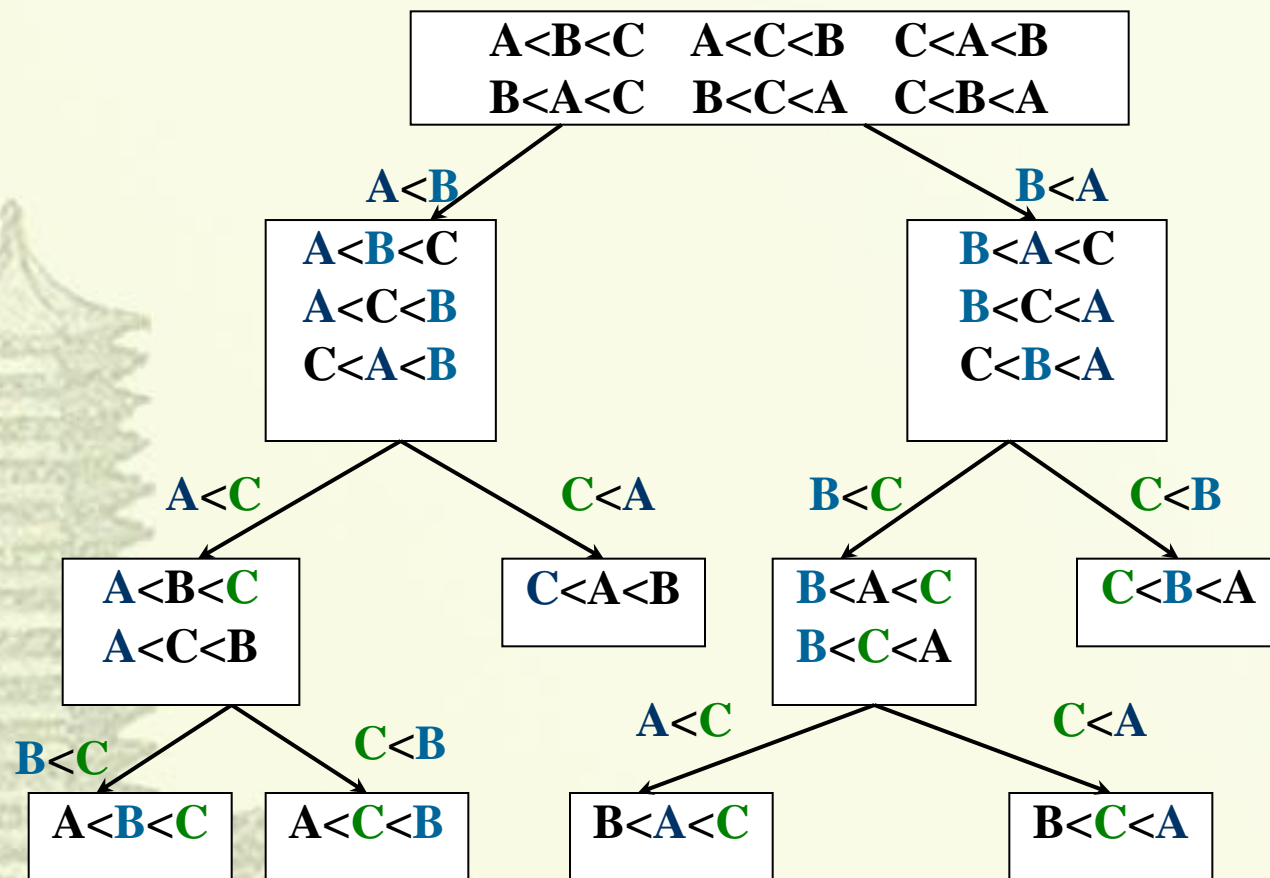


判定树(Decision Tree)

- 判定树中叶结点的最大深度就是排序算法在最差情况下需要的最大比较次数
- 叶结点的最小深度就是最佳情况下的最小比较次数



用判定树模拟基于比较的排序





基于比较的排序的下限

- 对 n 个记录，共有 $n!$ 个叶结点
- 判定树的深度为 $\log(n!)$
- 在最差情况下需要 $\log(n!)$ 次比较，即 $\Omega(n \cdot \log n)$
- 在最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较



排序问题的时间代价

- 在最差情况下任何基于比较的排序算法都需要 $\Omega(n \log n)$ 次比较，因此最差情况时间代价就是 $\Omega(n \cdot \log n)$
- 那么排序问题本身需要的运行时间也就是 $\Omega(n \cdot \log n)$
- 所有排序算法都需要 $\Theta(n \cdot \log n)$ 的运行时间，因此可以推导出排序问题的时间代价为 $\Theta(n \cdot \log n)$

基数排序效率



□ 时间代价为 $\Theta(d n)$ ，实际上还是 $\Theta(n \log n)$

■ 没有重复关键码的情况，需要 n 个不同的编码来表示它们

• 也就是说， $d \geq \log_r n$ ，即在 $\Omega(\log n)$ 中



小结

□ 基本概念

- 排序码、正逆序、稳定性

□ 三种 $O(n^2)$ 的简单排序

- 插入、选择、冒泡
- 比较次数、移动次数（交换vs移动）

□ Shell排序

- 基于插入排序
- 建立分区



□ 基于分治法的排序

- 快速排序、归并排序

□ 堆排序

- 选择排序

□ 分配排序和基数排序

- 桶排序、静态链表、地址排序

□ 排序算法的理论和实验时间代价

- 判定树



The End

