数据结构与算法
第 8 章 内排序

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

排序算法小结

- 特点?
- 复杂度?
- 核心操作?

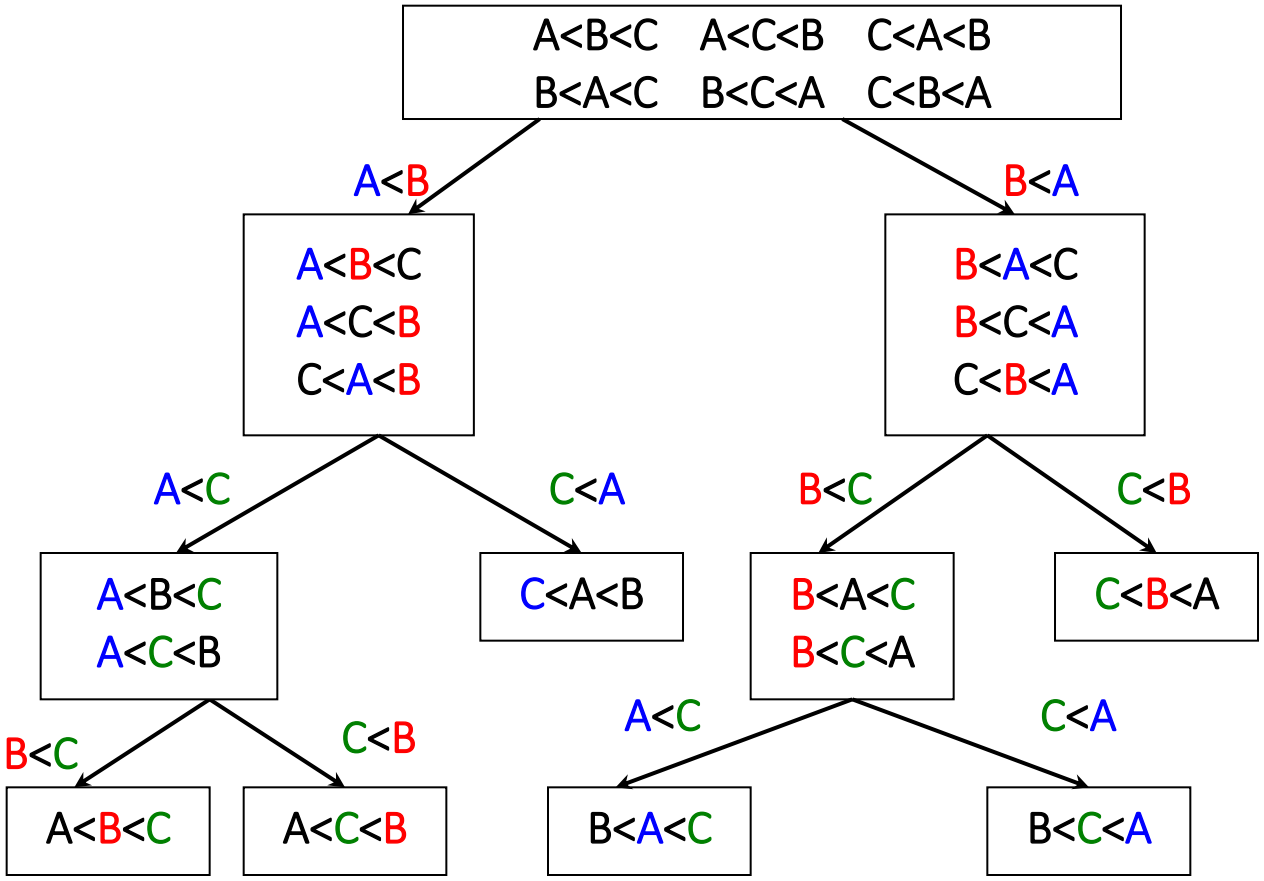
排序问题的下限

- 排序问题的时间代价在 $\Omega(n)$ （单趟扫描起码的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为 $\Omega(n \log n)$
  - 用 **判定树**（Decision Tree）阐明

判定树

- 对n个记录的序列，共有n!个排列结果
  - 结点列举当前状态下可能的记录排列集合
  - 边表示记录间的比较（判断）
  - 叶结点代表一种排序结果

用DT模拟基于比较的排序



基于比较的排序的下限

- 判定树的深度为log(n!)
  - 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

- 最差情况需要log(n!)次比较，即Ω(n·log n)，亦即，任何基于比较的排序算法都需要Ω(n·log n)次比较

思考和讨论

1. 本章讨论的排序算法都是基于数组实现的，可否应用于动态链表？性能上是否有差异?
2. 试总结并证明各种排序算法的稳定性，若算法稳定，如何修改可以使之不稳定？若算法不稳定，可否通过修改使之稳定?
3. 试调研STL中的各种排序函数是如何组合各种排序算法的

排序算法小结

- 特点：
  - 基于比较
  - 时间下限
- 有无不通过比较就可实现排序的方法？即，可否设计某些算法，使其不用对待排序元素进行一一比较?
- Such algorithms require special assumptions regarding the type and/or range of the key values to be sorted

其他排序方法

- 日常实例
  - 图书馆中书目管理
    - 作者姓名分类，首字母，从左到右
  - 邮局分发邮件
    - ZIP Code 具有同样的长度
  - 整数
    - [23, 123, 234, 567, 3] => [123, 23, 234, 3, 567]
    - [023,123, 234, 567, 003] => [003, 023,123, 234, 567]

分配排序和基数排序

- 不必记录间的两两比较
- 需事先知道记录序列的一些具体情况

- 基本思想
  - 将排序码**分解成**若干部分，通过对各个部分排序码的分别排序，最终达到对整个排序码的排序
- 主要介绍两类
  - 桶式排序
  - 基数排序

桶式排序

- 事先知道序列中的记录都处于某个小区间段[0，m)内
- 将具有相同值的记录分配到同一个桶中，然后按照编号依次从桶中取出记录，组成一个有序序列

桶式排序示例

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

待排数组: 7 3 8 9 6 1 8’ 1’ 2

	0	1	2	3	4	5	6	7	8	9
每个桶count：	0	2	1	1	0	0	1	1	2	1

＋＋＋＋

	0	1	2	3	4	5	6	7	8	9
前若干桶的累计count_->后继起始下标：	0	2	3	4	4	4	5	6	8	9

桶排序示例

待排数组：

7	3	8	9	6	1	8’	1’	2

每个桶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) {
    Record *TempArray = new Record[n];  // 临时数组
    int *count = new int[max];          // 桶容量计数器
    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]; // c [i]记录i+1的起址
    for (i = n-1; i >= 0; i--)          // 尾部开始，保证稳定性
        Array[--count[TempArray[i]]] = TempArray[i];
}
```

桶式排序分析

- 数组长度为n，所有记录区间[0,m)上
- 时间代价
  - 统计计数时间：$\Theta(n+m)$，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

基数排序

- 桶式排序只适合 *m* 很小的情况
- 基数排序
  - 当 *m* 很大时，可以将记录的排序码拆分为多个部分

基数排序

- 假设长度为 n的序列

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

  记录的排序码 k包含d个子排序码

  $k = (k_{d-1}, k_{d-2}, \ldots, k_1, k_0)$

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

  $$(k_{i,d-1}, k_{i,d-2}, \ldots, k_{i,1}, k_{i,0}) \le (k_{j,d-1}, k_{j,d-2}, \ldots, k_{j,1}, k_{j,0})$$

  - k_{d-1} 称为**最高排序码**
  - k_0 称为**最低排序码**

基数排序示例

- 若要对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，整个序列有序

基数排序分类

- 高位优先法 MSD

  Most Significant Digit first

- 低位优先法 LSD

  Least Significant Digit first

示例

黑桃♠(S) > 红心♥(H) > 方片♦(D) > 梅花♣ (C)

♠3 ♥J ♣8 ♥9 ♠9 ♦3 ♣1 ♦7

- 高位先排(MSD)，递归分治
  - 先按花色：♣8 ♣1 ♦3 ♦7 ♥J ♥9 ♠3 ♠9
  - 再按面值：♣1 ♣8 ♦3 ♦7 ♥9 ♥J ♠3 ♠9
- 低位先排(LSD)`，要求稳定排序！
  - 先面值：♣1 [♠3 ♦3] ♦'7 ♣8 [♥9 ♠'9] ♥'J
  - 再花色：[♣1 ♣'8] [♦3 ♦'7] [♥9 ♥'J] [♠3 ♠'9]

高位优先法 MSD

- **步骤**：
  1. **先**对**高位**k_{d-1}进行桶式排序，将序列分成若干个桶；
  2. 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；
  3. **依次**重复，直到对**最低位**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}，…，k_1，k_0)；
  4. 最后将**所有的桶**依次连接在一起，成为一个有序序列

- 一个**分**、**分**、…、**分**、**收**的过程

低位优先法 LSD

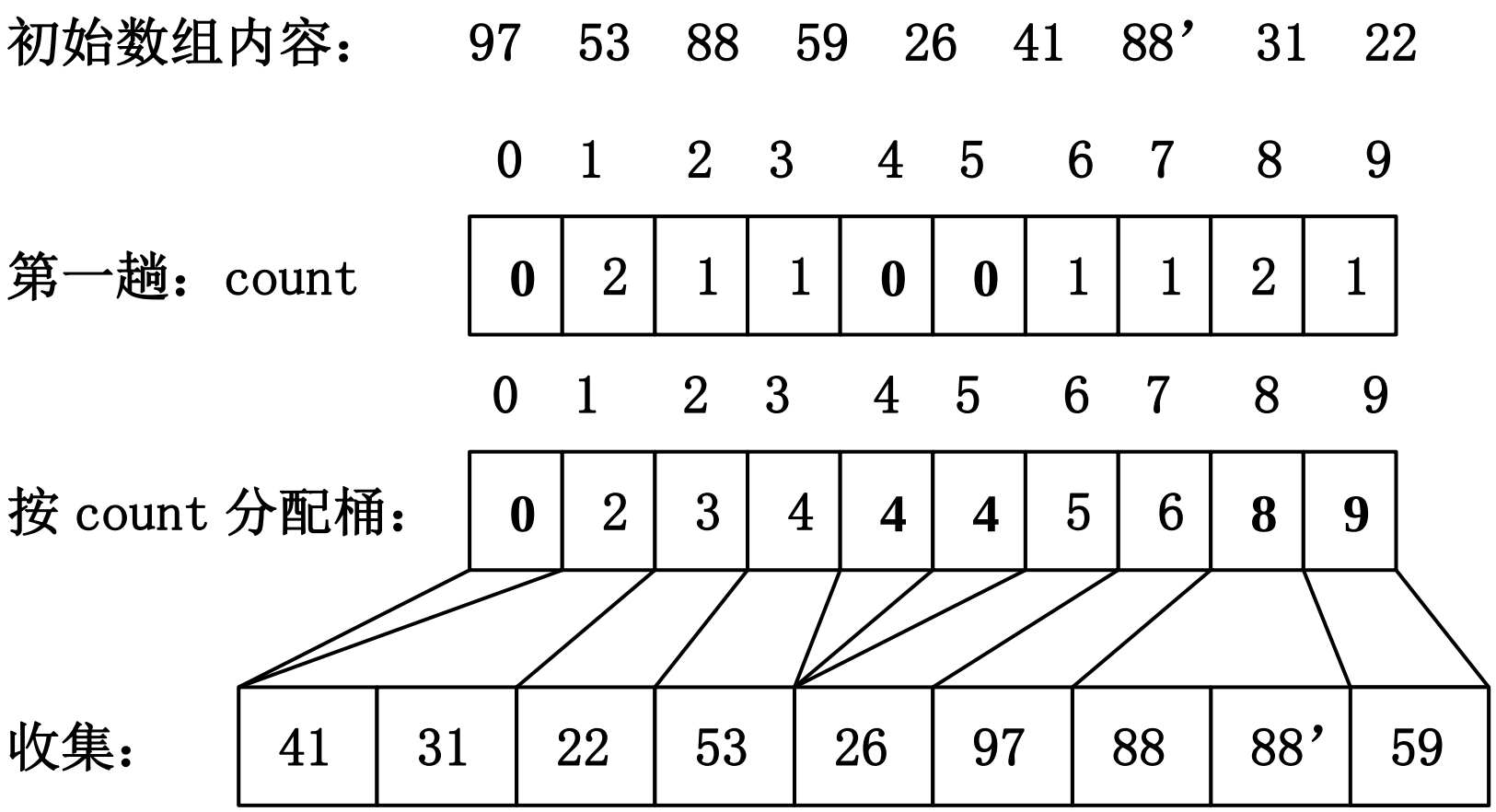
- **步骤**
  1. 从最低位k_0开始排序，分配后回收；
  2. 对于排好的序列再用次低位k_1排序，分配后回收；
  3. 依次重复，直至对最高位k_{d-1}排好序后，整个序列成为有序的

- **一个分、收；分、收；…；分、收的过程**
  - 比较简单，易于计算机处理

基数排序的实现

- 主要讨论 LSD
  - 基于顺序存储
  - 基于链式存储
- 待排序原始数组R
  - 长度为 n
  - 基数为 r
  - 排序码个数为 d

基于顺序存储的基数排序

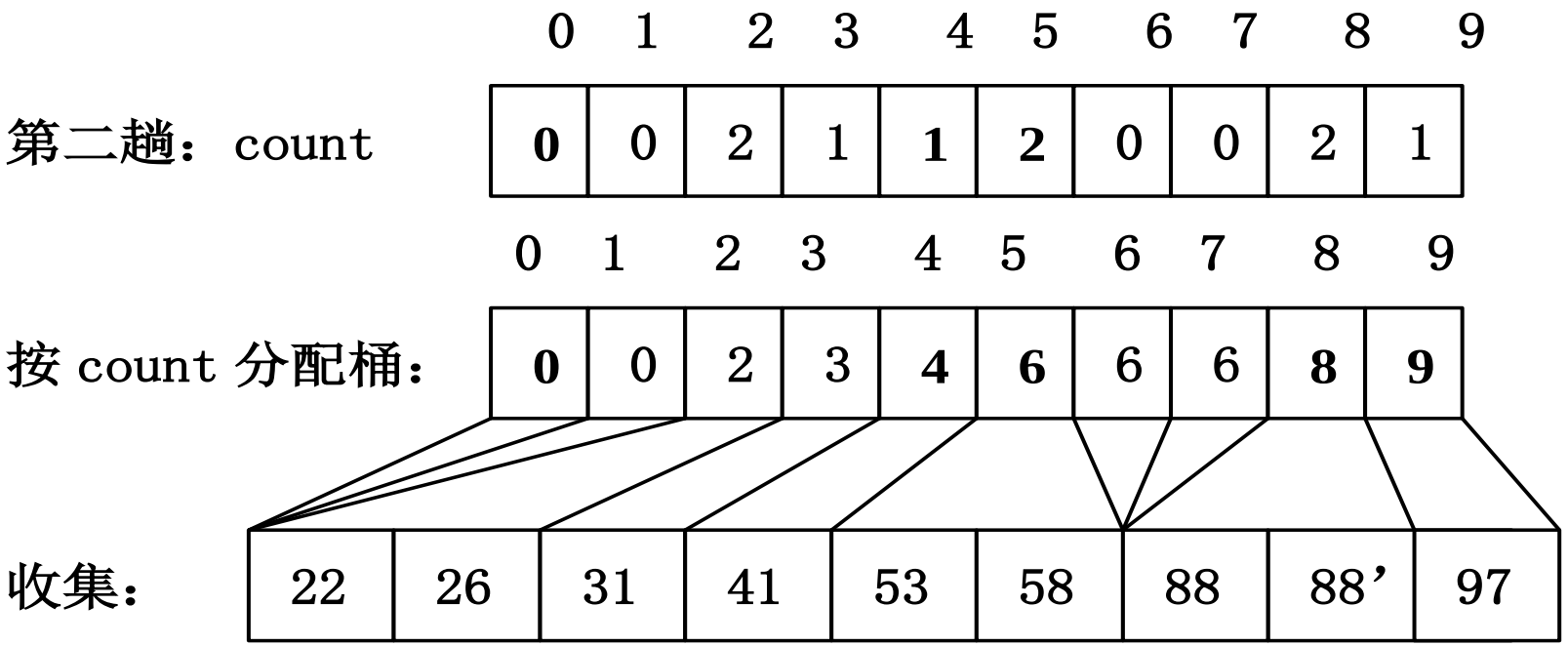


(a) 第一趟分配个位

基于顺序存储的基数排序

第一趟收集结果

41	31	22	53	26	97	88	88’	59



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) {
      Record  *TempArray = new Record[n];
      int *count = new int[r];          int i, j, k;
      int Radix = 1;                                  // 模进位，用于取Array[j]的第i位
      for (i = 1; i <= d; i++)  {                     // 对第 i 个排序码分配
            for (j = 0; j < r; j++)
                  count[j] = 0;                       // 初始计数器均为0
            for (j = 0; j < n; j++) {                 // 统计每桶记录数
                  k = (Array[j] / Radix) % r;          // 取第i位
                  count[k]++;                          // 相应计数器加1
            }
      for (j = 1; j < r; j++)                         // 给桶划分下标界
                  count[j] = count[j-1] + count[j];
            for (j = n-1; j >= 0; j--)  {             // 从数组尾部收集
                  k = (Array[j] / Radix ) % r;        // 取第 i 位
                  count[k]--;                         // 桶剩余量计数器减1
                  TempArray[count[k]] = Array[j];      // 入桶
            }
            for (j = 0; j < n; j++)                   // 内容复制回 Array 中
                  Array[j] = TempArray[j];
            Radix *= r;                               // 修改模Radix
      }
}
```

基于数组的基数排序算法分析

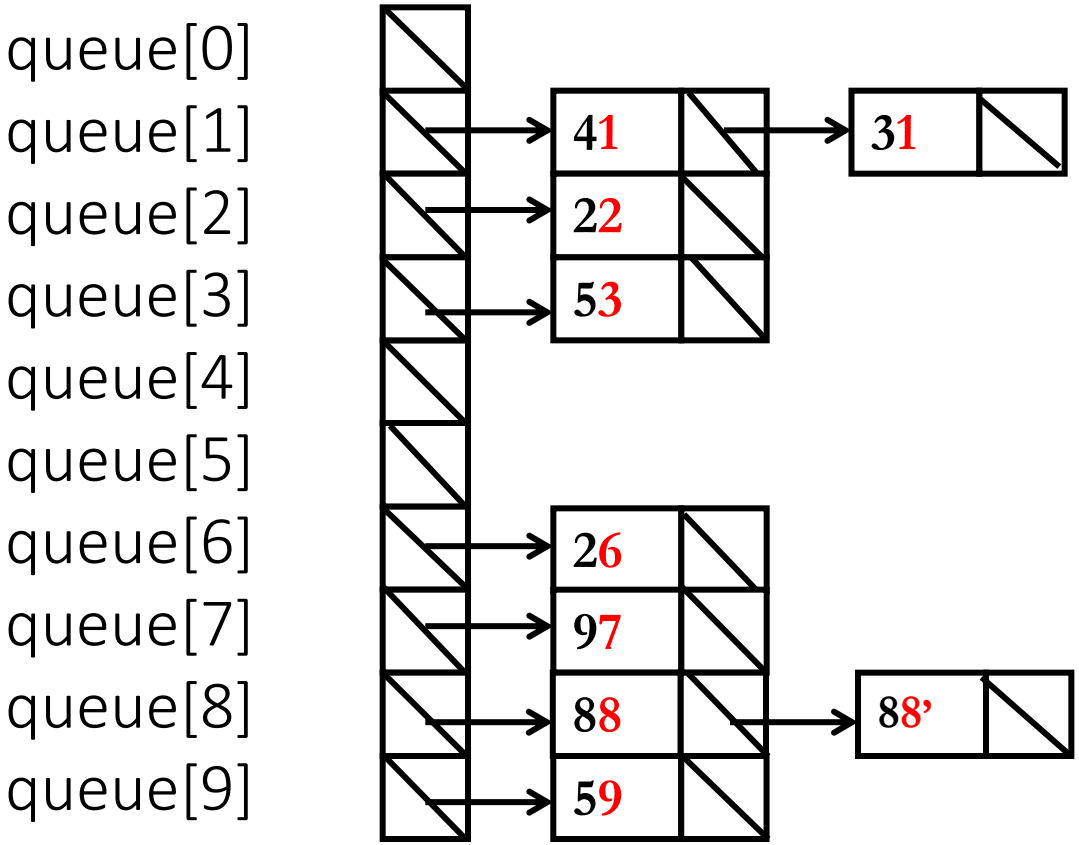
- 空间代价　Θ(n+r)
  - 临时数组，n
  - r 个计数器
- 时间代价　Θ(d ·(n+r))
  - 桶式排序：　Θ(n+r)
  - d 次桶式排序

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

(a)初始链表内容

97	53	88	59	26	41	88’	31	22

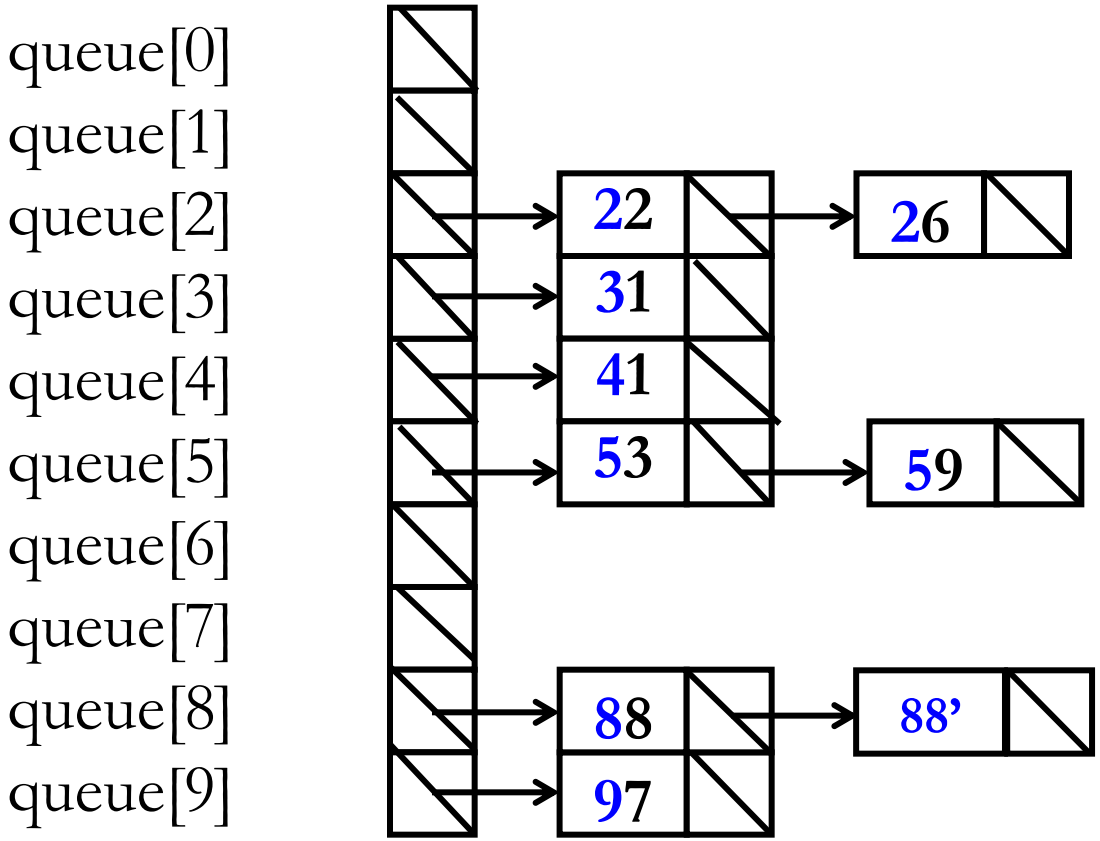


(b)第一趟分配

(c) 第一趟收集结果

41	31	22	53	26	97	88	88’	59

41	31	22	53	26	97	88	88’	59



(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;                    // 结点的关键码值
    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

```
template <class Record>
void RadixSort(Record *Array, int n, int d, int r) {
    int i, first = 0;                          // first指向第一个记录
    StaticQueue *queue = new StaticQueue[r];
    for (i = 0; i < n-1; i++)
        Array[i].next = i + 1;                 // 初始化静态指针域
    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);                 // 整理后，按下标有序
}
```

基于静态链的基数排序

```
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++)    queue[j].head = -1;
    while (curr != -1)  {                    // 对整个静态链进行分配
        k = Array[curr].key;
        for (a = 0; a < i; a++)              // 取第i位排序码数字k
            k = k / r;
        k = k % r;
        if (queue[k].head == -1)             // 把数据分配到第k个桶中
            queue[k].head = curr;
        else  Array[queue[k].tail].next = curr;
        queue[k].tail = curr;
        curr = Array[curr].next;             // curr移动，继续分配
    }
}
```

基于静态链的基数排序

```
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
        k++;
        while (k < r-1 && queue[k].head == -1)
            k++;
        if (queue[k].head != -1)  {           // 试探下一个队列
            Array[last].next = queue[k].head;
            last = queue[k].tail;             // 最后一个为序列的尾部
        }
    }
    Array[last].next = -1;                    // 收集完毕
}
```

链式基数排序算法代价分析

- 空间代价　O(n+r)
  - n个记录空间
  - r个子序列的头尾指针
- 时间代价　O(d·(n+r))
  - 不需要移动记录本身，只需修改记录的next指针

基数排序效率

- 时间代价为 $\Theta(d \cdot n)$， 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**
  - 即，$d >= \log_r n$，即在 $\Omega(\log n)$ 中

思考

- 桶排事先知道序列中的记录都位于某个小区间段 [0..m) 内。m 多大合适？超过这个范围怎么办?
- 桶排中，count 数组的作用是什么？为什么桶排要从后往前收集?

- 顺序和链式基数排序的优劣?
- 链式基数排序的结果整理?

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
   Record TempRec;
   for (i = 0; i < n-1; i++) {         // 循环，每次处理第 i 个记录
      TempRec = Array[j];              // 暂存第 i 个的记录 Array[j]
      swap(Array[i], Array[j]);
      Array[i].next = j;               //  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
         j = Array[j].next;
   }
}
```

数组 vs 链表

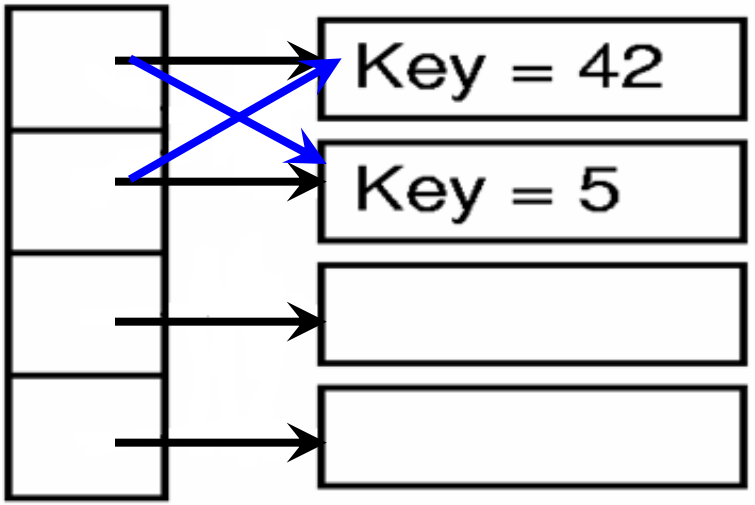
- 数组
- 静态链表

空间 vs 时间

- 速度比较快的排序算法 O(nlogn)
  - 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

索引地址排序

- 大记录序列的排序
  - 建立索引数组（通用）
  - 建立静态链表



交换指针，**减少交换记录的次数**

索引方法 #11

- 结果下标数组IndexArray[i]存放的是Array[i]中数据应该待的位置，也即，最终位置

  Array[IndexArray[i]] = 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
新数组	12	25	29	32	34	34’	45	64

索引方法 #I2

- 结果下标数组 IndexArray[i]存放的是Array[i]中应该摆放的数据的所在位置，也即，

  Array[i] = Array[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

	0	1	2	3	4	5	6	7
新数组	12	25	29	32	34	34’	45	64

不开新数组

- 只利用原Array和IndexArray
  - 可用一个临时空间： 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
I2结果	5	1	0	6	2	4	7	3
I1结果	2	1	4	7	5	0	3	6

按I2链	0	1	2	3	4	5	6	7
整理后数组	12	25	29	32	34	34’	45	64

得到索引 I2 的方法?

- 一般的排序方法都可以
  - 将排序中的那些赋值（或交换）换成对index数组的赋值（或交换）
- e.g.,：插入排序

插入排序产生索引2

```
template<class Record>
void AddrSort(Record Array[], int n) {
    // 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 ( Array[IndexArray[j]] <Array[IndexArray[j-1]]))
                swap(IndexArray, j, j-1);
            else break;                      // 此时i前面记录已排序
```

插入排序产生索引2

```
        // 根据IndexArray整理Array
        for (i=0; i<n; i++) {          // 调整为按下标有序
                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 I1 的方法?

- Rank排序
- 静态链表的基数排序

思考

- 证明地址排序整理方案的时间代价为$\theta(n)$

- 修改快速排序，得到第一种索引结果
- 采用Rank排序得到第二种索引的方法
- 对静态链的基数排序结果进行简单变换得到第二种索引的方法

Rank排序

```
template <class Record>
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 (array[i] > array[j])
                 rank[i]++;
           else
             rank[j]++;
    }
}
```

排序算法的时间代价

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

排序代价的根源

- 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$\Theta(n^2)$

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**
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

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	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,n-1]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>
#define CLOCKS_PER_SEC  1000

clock_t tstart = 0;      // 开始的时间

// 初始化计时器
void Settime()  {
    tstart = clock();
}

// 上次 Settime() 之后经过的时间
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>(&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

基数排序效率再回顾

- 时间代价为Θ(d·n)
  - 本质上 Θ(nlog 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
    - 即，d>=$\log_r n$，即在Ω(logn)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

小结

- 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

思考和讨论

1. 本章讨论的排序算法都是基于数组实现的，可否应用于动态链表？性能上是否有差异?
2. 试总结并证明各种排序算法的稳定性，若算法稳定，如何修改可以使之不稳定？若算法不稳定，可否通过修改使之稳定?
3. 试调研STL中的各种排序函数是如何组合各种排序算法的

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

小结

- 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