一、 实验目的

- 用 C 语言模拟实现理论课上的算法和思想
- 掌握底层实现关系选择，连接，集合的交、并、差操作的基本逻辑
- 加深对算法 I/O 复杂度和内外存环境对查询实现影响的理解
- 加深对索引减少 I/O 次数作用的理解

二、 实验环境

- 实验开发环境为 Windows 11 家庭中文版
- 所使用的 IDE 有 CLion 和 CodeBlocks，最终导出 CodeBlocks 的工程文件
- 基于 ExtMem 程序库

三、 实验内容

- **实验数据：**
  - ✧ 关系 R 具有两个属性 A 和 B，其中 A 和 B 的属性值均为 int 型（4 个字节），A 的值域为[100, 140]，B 的值域为[4000, 5000]。
  - ✧ 关系 S 具有两个属性 C 和 D，其中 C 和 D 的属性值均为 int 型（4 个字节）。C 的值域为[120, 160]，D 的值域为[4000, 6000]。
- **实验任务：**

  基于 ExtMem 程序库，用 C 语言实现以下 5 个任务（及附加题），要求使用有限内存（Buffer）实现，不可定义长度大于 10 的数组，将结果存放在磁盘上：

  (1) 基于线性搜索的关系选择算法：模拟实现

  $$\text{select S.C,S.D from S where S.C} = 130$$

  打印 I/O 读写次数。

(2) 实现两阶段多路归并排序算法（TPMMS）：利用内存缓冲区将关系 R 和 S 分别排序，打印 I/O 读写次数。

(3) 基于索引的关系选择算法：利用 (2) 中的排序结果为关系 S 建立索引文件，利用索引文件选出 S.C=130 的元组，打印 I/O 读写次数。

(4) 基于排序的连接操作算法（Sort-Merge-Join）：模拟实现

select S.C,S.D,R.A,R.B from S inner join R on S.C＝R.A

即对关系 S 和 R 计算 S.C 连接 R.A，打印连接次数。

(5) 基于排序或散列的两趟扫描算法：实现并 S∪R，交 S∩R，差 S－R。并打印并、交、差操作后的元组个数。

四、 实验过程

注：每次执行写入操作都要判断输出块是否已满，在算法结束后都要判断输出快是否不为空。在之后的伪代码中为了节约篇幅省略了以下伪代码：

```
// 当执行 write … to w 时，默认会执行下面判断
if w is full then
  writeBlockToDisk(w)
  clear w
end

// 在每个伪代码的结尾，都应判断输出块是否还有元组未写入
if w is not empty then
  writeBlockToDisk(w)
end
```

(1) 实现基于线性搜索的关系选择算法

问题分析：

线性搜索算法过程简单，只需要读入一遍该关系的所有元组即可找到满足条件的元组，注意读写细节即可。

核心代码分析：

$w \leftarrow$ getNewBlockInBuffer( )　　// w 是输出块

clear w

foreach *block in* S **do**　　// 循环关系 S 的每个磁盘块

　$r \leftarrow$ readBlockFromDisk(*block*)　// r 是读入块

　foreach *row in* r **do**　　// 遍历所有元组

　　$X \leftarrow S.C$

　　if $X = 130$ **then**

　　write *row* to w　　// 写入输出块中

　　end

　endfor

endfor

实验结果：

```
--------------------------------
基于线性搜索的选择算法 S.C = 130
--------------------------------
读入数据块17
读入数据块18
(X = 130,Y = 4208)
(X = 130,Y = 5202)
读入数据块19
读入数据块20
读入数据块21
读入数据块22
读入数据块23
读入数据块24
读入数据块25
读入数据块26
读入数据块27
(X = 130,Y = 5083)
读入数据块28
读入数据块29
读入数据块30
读入数据块31
读入数据块32
读入数据块33
读入数据块34
读入数据块35
读入数据块36
(X = 130,Y = 4983)
读入数据块37
读入数据块38
读入数据块39
(X = 130,Y = 4371)
读入数据块40
读入数据块41
读入数据块42
读入数据块43
读入数据块44
(X = 130,Y = 4180)
读入数据块45
读入数据块46
读入数据块47
读入数据块48
注：结果写入磁盘：100

满足选择条件的元组一共6个.

IO读写一共33次.
```

(2) 实现两阶段多路归并排序算法（TPMMS）

问题分析：

因为缓存块数有限，不能一次完成排序工作，得进行两趟排序。第一趟完成子集合划分，进行子集合内部的排序（冒泡排序）；第二趟进行子集合间的归并排序（从每个子集合中找到最小的元组，放入输出块中）。这里首先要求子集合内部的块数不能大于缓存块数（8 块），并且子集合的数量也不能大于缓存块数（8 块），且要留有一块作为输出块。因为关系 S 有 32 块（关系 R 块数少，且和关系 S 的排序无关，故不用考虑），当每个子集合有 6 块时，需要分为 6 个子集合，均满足上述要求。

核心代码分析：

```
w ← getNewBlockInBuffer( )      // w 是输出块
clear w
// 第一趟排序，假设时对关系S排序（R同理）
foreach subset in S do                    // 循环关系S的每个子集合
  foreach block in subset do              // 循环子集合的每个磁盘块，下标为i
    r_i ← readBlockFromDisk(block_i)      // r 是读入块数组
  endfor
  // 对子集合进行冒泡排序
  for cnt ← 1 to subset × size do         // 外循环次数为子集合元组数
    foreach row in subset do              // 遍历子集合所有元组
      if (X,Y)_row > (X,Y)_next_row then  // 比较前后两个元组，大者置后
        swap(row, next_row)
      end
    endfor
  endfor
  foreach row in subset do                // 遍历子集合所有元组（已排序，位于r中）
    write row to w                        // 写入输出块中
  endfor
endfor
```

// 第二趟排序

foreach *subset in S* **do** // 循环关系S的每个子集合，下标为i

// *disk_index* 记录当前每个缓存块对应的磁盘块号

$disk\_index_i \leftarrow the\ first\ disk\ number\ of\ subset_i$

$r_i \leftarrow \text{readBlockFromDisk}(disk\_index_i)$ // r 是读入块数组

endfor

$index[\ ] \leftarrow 0$ // 每个缓存块的比较位置，初始为0

while *exist* r_i *is not empty* **do**

// 找到每个缓存块中当前比较位置的最小的元组

$min\_row \leftarrow \min((X,Y)_i)$

$min\_pos \leftarrow \text{argmin}((X,Y)_i)$

write *min_row* to w // 写入输出块中

$index[min\_pos] \leftarrow index[min\_pos] + 8$

// 如果当前缓存块全部比较完毕，读入该子集合的下一个磁盘块

if $index[min\_pos]$ *is max* **then**

$index[min\_pos] \leftarrow 0$

$\text{freeBlockInBuffer}(r_i)$

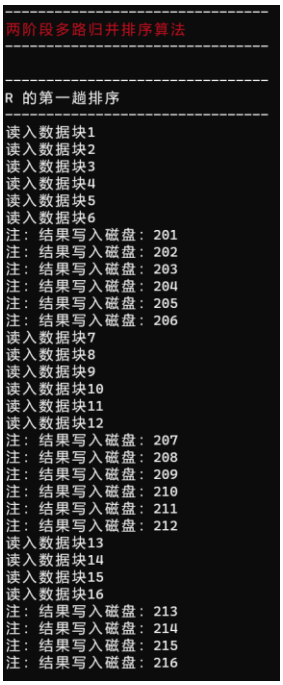
$disk\_index_i++$

$r_i \leftarrow \text{readBlockFromDisk}(disk\_index_i)$

end

endwhile

实验结果：



```
--------------------------------
S 的第一趟排序
--------------------------------
读入数据块17
读入数据块18
读入数据块19
读入数据块20
读入数据块21
读入数据块22
注：结果写入磁盘：217
注：结果写入磁盘：218
注：结果写入磁盘：219
注：结果写入磁盘：220
注：结果写入磁盘：221
注：结果写入磁盘：222
读入数据块23
读入数据块24
读入数据块25
读入数据块26
读入数据块27
读入数据块28
注：结果写入磁盘：223
注：结果写入磁盘：224
注：结果写入磁盘：225
注：结果写入磁盘：226
注：结果写入磁盘：227
注：结果写入磁盘：228
读入数据块29
读入数据块30
读入数据块31
读入数据块32
读入数据块33
读入数据块34
注：结果写入磁盘：229
注：结果写入磁盘：230
注：结果写入磁盘：231
注：结果写入磁盘：232
注：结果写入磁盘：233
注：结果写入磁盘：234
读入数据块35
读入数据块36
读入数据块37
读入数据块38
读入数据块39
读入数据块40
注：结果写入磁盘：235
注：结果写入磁盘：236
注：结果写入磁盘：237
注：结果写入磁盘：238
注：结果写入磁盘：239
注：结果写入磁盘：240
读入数据块41
读入数据块42
读入数据块43
读入数据块44
读入数据块45
读入数据块46
注：结果写入磁盘：241
注：结果写入磁盘：242
注：结果写入磁盘：243
注：结果写入磁盘：244
注：结果写入磁盘：245
注：结果写入磁盘：246
读入数据块47
读入数据块48
注：结果写入磁盘：247
注：结果写入磁盘：248
```

```
--------------------------------
S 的第二趟排序
--------------------------------
读入数据块217
读入数据块223
读入数据块229
读入数据块235
读入数据块241
读入数据块247
注：结果写入磁盘：317
注：结果写入磁盘：318
注：结果写入磁盘：319
读入数据块242
读入数据块218
读入数据块224
注：结果写入磁盘：320
注：结果写入磁盘：321
读入数据块230
注：结果写入磁盘：322
注：结果写入磁盘：323
读入数据块243
注：结果写入磁盘：324
读入数据块225
注：结果写入磁盘：325
注：结果写入磁盘：326
读入数据块236
读入数据块231
注：结果写入磁盘：327
注：结果写入磁盘：328
读入数据块219
读入数据块244
注：结果写入磁盘：329
读入数据块248
读入数据块226
读入数据块232
注：结果写入磁盘：330
注：结果写入磁盘：331
读入数据块237
注：结果写入磁盘：332
读入数据块233
注：结果写入磁盘：333
注：结果写入磁盘：334
读入数据块238
注：结果写入磁盘：335
读入数据块220
读入数据块245
注：结果写入磁盘：336
注：结果写入磁盘：337
注：结果写入磁盘：338
读入数据块221
注：结果写入磁盘：339
读入数据块239
读入数据块246
注：结果写入磁盘：340
读入数据块227
读入数据块234
注：结果写入磁盘：341
注：结果写入磁盘：342
读入数据块222
注：结果写入磁盘：343
读入数据块228
注：结果写入磁盘：344
读入数据块240
注：结果写入磁盘：345
注：结果写入磁盘：346
注：结果写入磁盘：347
注：结果写入磁盘：348

IO读写一共192次.
```

(3) 实现基于索引的关系选择算法

问题分析：

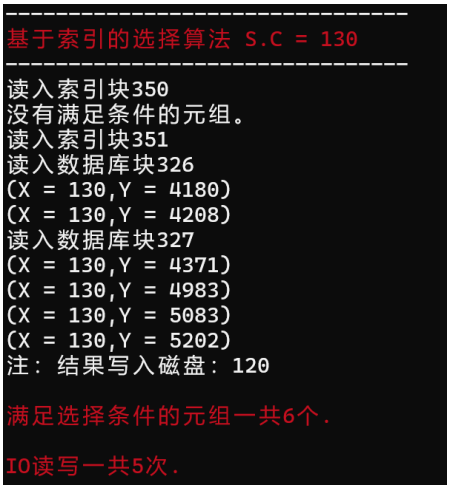
利用(2)的排序结果建立稀疏索引：索引字段是 S.C 的值，指向这个值第一次出现在有序磁盘块（(2)排序结果的磁盘块）的块号。因为索引的值对应元组在磁盘中是连续分布的，只需要遍历索引块，找到目标值对应的磁盘块号，根据这个块号在磁盘块中查找这个值存在的元组即可，要注意满足条件的连续元组可能会跨块。

核心代码分析：

```
w ← getNewBlockInBuffer( )          // w 是输出块
clear w
// 建立索引
foreach block in S do                // 循环已排序关系S的每个块
  r ← readBlockFromDisk(block)
  foreach row in r do
    X ← S.C
    write (X, block number) to w     // 将（索引字段，对应块号）写入输出块
  endfor
endfor
// 利用索引查找
foreach block in Index do            // 循环索引的每个块
  r ← readBlockFromDisk(block)
  foreach row in r do
    X ← S.C
    if X = 130 then
      disk_num ← Y                   // Y中记录的就是对应的磁盘块号
      break                          // 结束两重循环
    end
  endfor
endfor
// first_meet标记是否还没遇到S.C = 130的元组
// not_meet标记是否还有S.C = 130的元组（退出循环标志）
first_meet ← 0, not_end ← 1
```

```
while not_end = 1 do
  r ← readBlockFromDisk(disk_num)
  foreach row in r do
    X ← S.C
    if X = 130 then
      first_meet ← 1
      write row to w                    // 写入输出块
    else if first_meet = 1 then         // 当X不等于130时，后面的X必大于130
      not_end ← 0
      break
    end
  endfor
  disk_num ++
  freeBlockInBuffer(r)
endwhile
```

实验结果：



(4) 实现基于排序的连接操作算法（Sort-Merge-Join）

问题分析：

利用(2)的排序结果进行类似归并排序的连接操作。为两个关系都分配一个缓存块。在连接时固定 R 的位置不动，往下遍历关系 S 的元组（可能跨块导致 S 的缓存块改变）。当连接结束时，还原 S 的位置，将 S 的缓存块也同时还原（可以再用另一个缓存块来缓存，这样就不用再次从磁盘读取了）。然后根据当前元组的值，判断 R 的元组后移还是 S 后移（谁小谁后移，相等情况 R 后移，因为 R 的这个元组已经与所有 S 等值的元组连接完毕）。

核心代码分析：

$w \leftarrow$ getNewBlockInBuffer( )　　// w 是输出块
$r \leftarrow$ readBlockFromDisk(R)　　// r 读入关系R的第一个磁盘块
$s \leftarrow$ readBlockFromDisk(S)　　// s 读入关系S的第一个磁盘块
$start \leftarrow s$　　// 缓存块s的备份

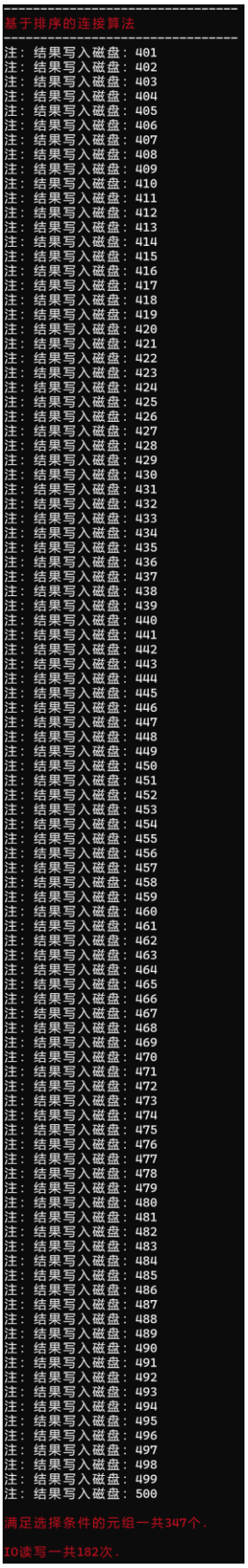
while *both r and s are not the end* **do**　　// 还未遍历完

  while $X_{row_r} = X_{row_s}$**do**
    write row_r *and* row_s to w　　// 连接
    if row_s *is the last row of* s **then**　　// r不动，s后移
      $s \leftarrow$ readBlockFromDisk(S)
    end
    $row_s \leftarrow$ *next row in* s
  endwhile

  if $start \neq s$ **then**　　// 还原缓存块s
    $s \leftarrow start$
  end

  if $X_{row_r} < X_{row_s}$ **then**　　// 类似归并排序，谁小谁后移
    if row_r *is the last row of* r **then**
      $r \leftarrow$ readBlockFromDisk(R)
    end
    $row_r \leftarrow$ *next row in* r
  else
    if row_s *is the last row of* s **then**
      $s \leftarrow$ readBlockFromDisk(S)
      $start \leftarrow s$　　// 备份s
    end
    $row_s \leftarrow$ *next row in* s
  end
endwhile

实验结果：


基于排序的连接算法
注：结果写入磁盘：401
注：结果写入磁盘：402
注：结果写入磁盘：403
注：结果写入磁盘：404
注：结果写入磁盘：405
注：结果写入磁盘：406
注：结果写入磁盘：407
注：结果写入磁盘：408
注：结果写入磁盘：409
注：结果写入磁盘：410
注：结果写入磁盘：411
注：结果写入磁盘：412
注：结果写入磁盘：413
注：结果写入磁盘：414
注：结果写入磁盘：415
注：结果写入磁盘：416
注：结果写入磁盘：417
注：结果写入磁盘：418
注：结果写入磁盘：419
注：结果写入磁盘：420
注：结果写入磁盘：421
注：结果写入磁盘：422
注：结果写入磁盘：423
注：结果写入磁盘：424
注：结果写入磁盘：425
注：结果写入磁盘：426
注：结果写入磁盘：427
注：结果写入磁盘：428
注：结果写入磁盘：429
注：结果写入磁盘：430
注：结果写入磁盘：431
注：结果写入磁盘：432
注：结果写入磁盘：433
注：结果写入磁盘：434
注：结果写入磁盘：435
注：结果写入磁盘：436
注：结果写入磁盘：437
注：结果写入磁盘：438
注：结果写入磁盘：439
注：结果写入磁盘：440
注：结果写入磁盘：441
注：结果写入磁盘：442
注：结果写入磁盘：443
注：结果写入磁盘：444
注：结果写入磁盘：445
注：结果写入磁盘：446
注：结果写入磁盘：447
注：结果写入磁盘：448
注：结果写入磁盘：449
注：结果写入磁盘：450
注：结果写入磁盘：451
注：结果写入磁盘：452
注：结果写入磁盘：453
注：结果写入磁盘：454
注：结果写入磁盘：455
注：结果写入磁盘：456
注：结果写入磁盘：457
注：结果写入磁盘：458
注：结果写入磁盘：459
注：结果写入磁盘：460
注：结果写入磁盘：461
注：结果写入磁盘：462
注：结果写入磁盘：463
注：结果写入磁盘：464
注：结果写入磁盘：465
注：结果写入磁盘：466
注：结果写入磁盘：467
注：结果写入磁盘：468
注：结果写入磁盘：469
注：结果写入磁盘：470
注：结果写入磁盘：471
注：结果写入磁盘：472
注：结果写入磁盘：473
注：结果写入磁盘：474
注：结果写入磁盘：475
注：结果写入磁盘：476
注：结果写入磁盘：477
注：结果写入磁盘：478
注：结果写入磁盘：479
注：结果写入磁盘：480
注：结果写入磁盘：481
注：结果写入磁盘：482
注：结果写入磁盘：483
注：结果写入磁盘：484
注：结果写入磁盘：485
注：结果写入磁盘：486
注：结果写入磁盘：487
注：结果写入磁盘：488
注：结果写入磁盘：489
注：结果写入磁盘：490
注：结果写入磁盘：491
注：结果写入磁盘：492
注：结果写入磁盘：493
注：结果写入磁盘：494
注：结果写入磁盘：495
注：结果写入磁盘：496
注：结果写入磁盘：497
注：结果写入磁盘：498
注：结果写入磁盘：499
注：结果写入磁盘：500
满足选择条件的元组一共347个.
IO读写一共182次.

(5) 实现基于散列的两趟扫描算法，实现交、并、差其中一种集合操作算法

问题分析：

利用(2)的排序结果进行类似归并排序的交操作。为两个关系都分配一个缓存块。在发现元组相等时，写入输出块，然后关系 S 和 R 都后移一个元组。如果不相等，判断 R 的元组后移还是 S 后移（谁小谁后移，因为大者剩下的元组都比小者当前元组大，不会出现相等的情况）。

核心代码分析：

```
w ← getNewBlockInBuffer( )        // w 是输出块
r ← readBlockFromDisk(R)          // r 读入关系R的第一个磁盘块
s ← readBlockFromDisk(S)          // s 读入关系S的第一个磁盘块
while both r and s are not the end do      // 还未遍历完
  if (X,Y)_row_r = (X,Y)_row_s do
    write row_r to w              // 交集写入输出块
    row_s ← next row in s         // 这里忽略了跨块，操作同上一题
    row_r ← next row in r         // 这里忽略了跨块，操作同上一题
  else if (X,Y)_row_r < (X,Y)_row_s then
    row_r ← next row in r
  else
    row_s ← next row in s
  end
endwhile
```

实验结果：

```
--------------------------------
基于排序的集合的交算法
--------------------------------
(X = 122,Y = 4749)
(X = 123,Y = 4362)
(X = 125,Y = 4741)
(X = 126,Y = 4031)
(X = 128,Y = 4249)
(X = 132,Y = 4759)
(X = 134,Y = 4864)
注: 结果写入磁盘: 140
(X = 137,Y = 4020)
(X = 137,Y = 4986)
注: 结果写入磁盘: 141

S和R的交集有9个元组。
```

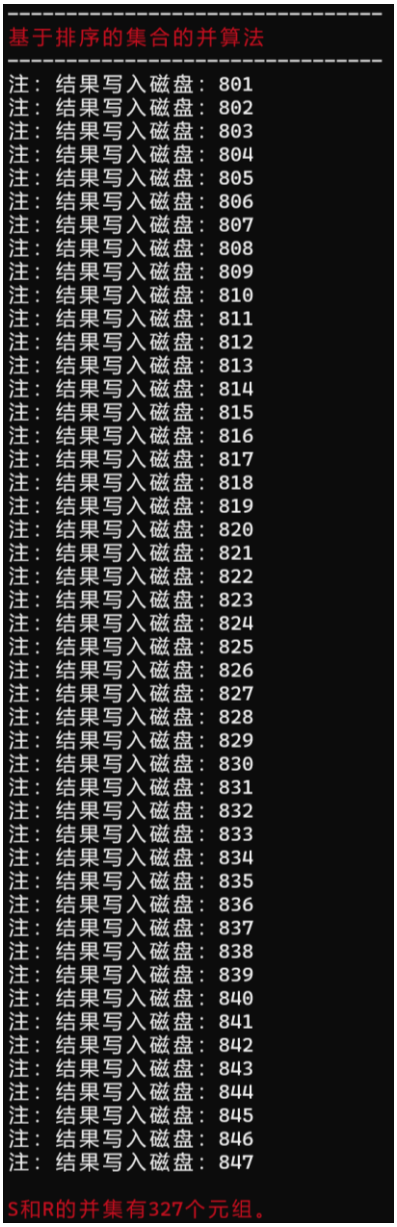
五、 附加题

- 并运算：

问题分析：

利用(2)的排序结果进行类似归并排序的并操作。为两个关系都分配一个缓存块。在发现元组相等时，只写入一次输出块，然后关系 S 和 R 都后移一个元组。如果不相等，判断 R 的元组后移还是 S 后移（谁小谁后移，因为大者剩下的元组都比小者当前元组大，不会出现相等的情况），同时写入输出块中（可以保证不会在另一个关系中重复）。当一个关系为空时，将另一个关系所有元组都写入输出块。

实验结果：

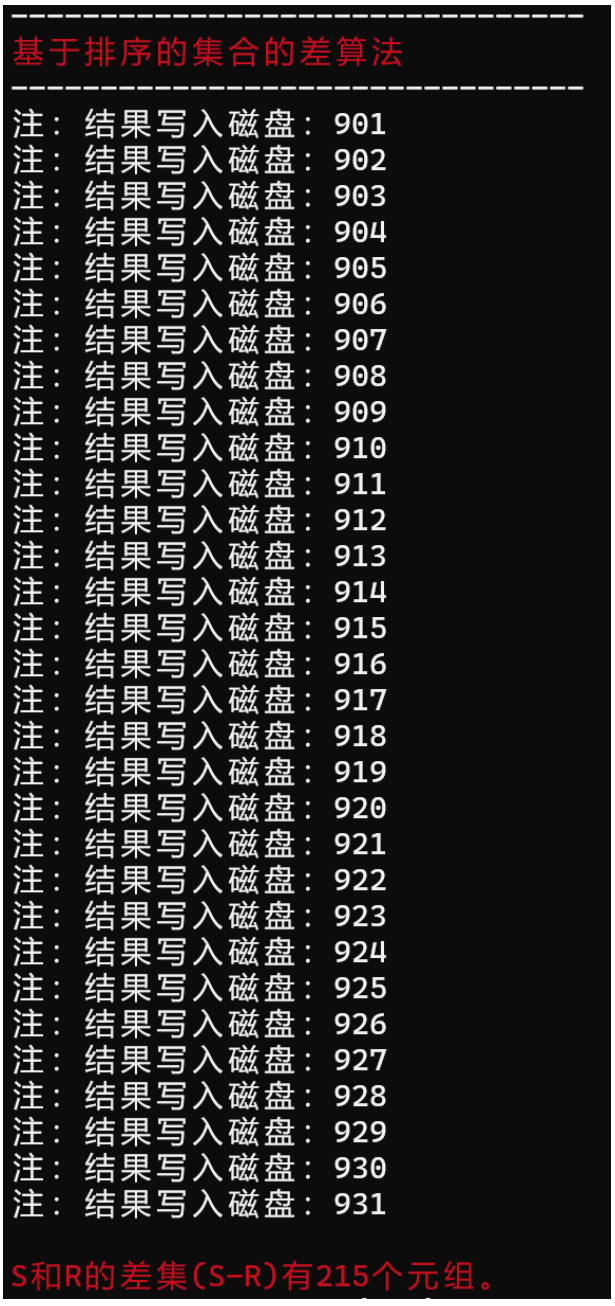


➢ 差运算：

问题分析：

利用(2)的排序结果进行类似归并排序的差操作。为两个关系都分配一个缓存块。在发现元组相等时，不写入一次输出块，然后关系 S 和 R 都后移一个元组。如果不相等，判断 R 的元组后移还是 S 后移（谁小谁后移，因为大者剩下的元组都比小者当前元组大，不会出现相等的情况），如果是 S 后移则写入输出块中（可以保证不会在 R 关系中出现）。如果 R 先空，则可以将 S 的剩下元组都写入输出块中，如果 S 空则结束算法。

实验结果：



六、 总结

- 问题：
  - 不熟悉打印语句改变颜色操作
  - 在使用 writeBlockToDisk 时没有注意到会将缓存块标记为可用，导致打印结果异常
  - 在各个实验中存在大量可以复用的代码块
  - 指针和内存释放使用不当
  - 模拟实现代码量大，不容易 Debug
- 上述问题都可以通过仔细研究 PPT 和指导书，以及 C 语言编程规范得到解决。但是解决之前还是付出比较长时间的代价。所以编程前应该想好具体的实现流程，并且对提供的代码有深刻的理解，清楚知道他们会产生怎样的影响，而不是理所当然地直接编写代码。特别是要利用好工具类，方便调试和调用函数，并且减少了代码层面的编写复杂性和冗余。
- 本次实验收获很大，对 C 语言和数据库底层实现的逻辑有了更清晰的认识，对理论课的知识有了更深刻的理解。尽管并没有实际接触最底层的硬件操作，但是在这个过程中也能充分体会各种算法对 I/O 结果的影响。