

哈尔滨工业大学(深圳)

# 《数据库》实验报告

## 实验四

### 查询处理算法的模拟实现

学 院: 计算机科学与技术

姓 名: 吕弋卿

学 号: 210110315

专 业: 计算机科学与技术

日 期: 2023-11-09

## 一、 实验目的

*阐述本次实验的目的。*

理解索引的作用，掌握关系选择、连接、集合的交、并、差等操作的实现算法，理解算法的 I/O 复杂性。

## 二、 实验环境

*阐述本次实验的环境。*

操作系统：Windows 11

集成环境：Code::Blocks 20.03

## 三、 实验内容

*阐述本次实验的具体内容。*

本次实验中，基于磁盘读写模拟程序库 ExtMem，实现了一系列数据库查询处理算法，包括以下部分：

1. 基于线性搜索的关系选择算法
2. 两阶段多路归并排序算法
3. 基于索引的关系选择算法
4. 基于排序的连接操作算法
5. 基于排序的集合交算法
6. 基于排序的集合并算法（附加题，已实现）
7. 基于排序的集合差算法（附加题，已实现）

## 四、实验过程

对实验中的5个题目分别进行分析，并对核心代码和算法流程进行讲解，用自然语言描述解决问题的方案。写明结果存储在磁盘的位置，并给出程序正确运行的结果截图。

### (0). 主程序流程

```
1  #include "defines.h"
2
3  int main(void) {
4
5      printTitle("基于线性搜索的选择算法 S.C=107");
6      linearSearch();
7
8      printTitle("两阶段多路归并排序算法");
9      mergeSort(1);
10     mergeSort(2);
11
12     printTitle("基于索引的关系选择算法");
13     constructIndex();
14     traverseIndex();
15
16     printTitle("基于排序的连接操作算法");
17     sortedInnerjoin();
18
19     printTitle("基于排序的集合交操作算法");
20     intersectSchema();
21
22     printTitle("基于排序的集合并操作算法");
23     unionSchema();
24
25     printTitle("基于排序的集合差操作算法");
26     differenceSchema();
27
28     printf("\n5个实验任务(包括附加题三种算法)已完成，输入任意字符退出：");
29     getchar();
30     return 0;
31 }
32
```

### (1) 实现基于线性搜索的关系选择算法

问题分析：

调用 extmem 中的 getBlockFromDisk() 方法,按顺序读入 17.blk 到 48.blk, 对于每个 blk 进行顺序遍历,调用 memcpy(),itoa(),atoi()等函数将其转化为由两个 int 型数据组成的二元组 Tuple,并将 Tuple.X,即 S.C 与 107 作比较,若相等,则打印到控制台,并写入内存缓冲块,内存缓冲块满之后,则写入磁盘块。遍历结束后若写缓冲块中有数据,也需写入磁盘块。最后,统计符合 S.C=107 的元组数量以及 IO 操作的次数。

```

while(TRUE) { //开始读磁盘中的Block

    bufferedReader = readBlockFromDisk(bufferedReaderAddress, &buf);
    printf("读入数据块%d\n", bufferedReaderAddress);

    for (int i=0; i<TUPLES_PER_BLOCK; i++) {

        Tuple bufferedTuple; //从读缓冲区获得元组,从char*转化为int型
        memcpy(temp, bufferedReader+i*TUPLIE_LENGTH, ELEMENT_LENGTH);
        bufferedTuple.x = atoi(temp);
        memcpy(temp, bufferedReader+i*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
        bufferedTuple.y = atoi(temp);

        if (bufferedTuple.x == 107) { //选择条件: S.C=107
            printf("(X=%d, Y=%d)\n", bufferedTuple.x, bufferedTuple.y); //打印满足选择条件的元组
            memcpy(bufferedWriter+foundCount*TUPLIE_LENGTH, bufferedReader+i*TUPLIE_LENGTH, TUPLIE_LENGTH); //将满足条件的元组添加利写缓冲区
            foundCount++;

            if (foundCount == TUPLES_PER_BLOCK) { //此块已写7个元组?
                itoa(bufferedWriterAddress+writeCount+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10); //将下个block的地址添加到写缓冲区
                if (writeBlockToDisk(bufferedWriter, bufferedWriterAddress + writeCount, &buf) != 0) { //将缓冲区内的数据写入磁盘block
                    perror("Writing Block Failed!\n");
                    return;
                }
                writeCount++;
                foundCount = 0; //重置块内元组计数器
                freeBlockInBuffer(bufferedWriter, &buf);
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
        }

        memcpy(temp, bufferedReader+TUPLES_PER_BLOCK*TUPLIE_LENGTH, ELEMENT_LENGTH); //读取block末尾的数据标识下个block地址
        bufferedReaderAddress = atoi(temp);
        if (bufferedReaderAddress >= SRC_S_INIT_ADDR+BLOCK_S_LENGTH) { //若读地址已过48则读终止
            break;
        }
        freeBlockInBuffer(bufferedReader, &buf);
    }
}

if (foundCount) { //将写缓冲区内的剩余数据写入磁盘block
    memcpy(bufferedWriter+foundCount*TUPLIE_LENGTH, emptyString, (TUPLES_PER_BLOCK-foundCount)*TUPLIE_LENGTH);
    itoa(bufferedWriterAddress+writeCount+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10); //将下个block的地址添加到写缓冲区
    if (writeBlockToDisk(bufferedWriter, bufferedWriterAddress+writeCount, &buf) != 0) { //写磁盘
        perror("Writing Block Failed!\n");
        return;
    }
    writeCount++;
}
for (int i=0; i<writeCount; i++) {
    printf("注: 结果写入磁盘:%d\n", bufferedWriterAddress + i);
}
setColor(RED);
printf("满足选择条件的元组一共:%d个\n", (writeCount-1)*TUPLES_PER_BLOCK + foundCount);
printf("IO读写一共%d次\n", buf.numIO);
setColor(WHITE);
freeBuffer(&buf);
return;

```

实验结果:

CodeBlocks 编译运行结果如下:

```

-----
基于线性搜索的选择算法 S.C=107
-----
读入数据块17
(X=107, Y=241)
(X=107, Y=209)
读入数据块18
(X=107, Y=317)
读入数据块19
读入数据块20
读入数据块21
读入数据块22
(X=107, Y=363)
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
读入数据块29
读入数据块30
读入数据块31
(X=107, Y=393)

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=107, Y=222)
读入数据块37
读入数据块38
读入数据块39
读入数据块40
(X=107, Y=356)
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
(X=107, Y=248)
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
注: 结果写入磁盘:101
满足选择条件的元组一共:8个
IO读写一共34次

```

共选择元组 8 个, 写入磁盘块 100.blk, 101.blk:

(107,241) (107,209) (107,317) (107,363) (107,393) (107,222) (107,356) (107,248)

IO 次数为 34 次，其中，读 17~48.blk 占 32 次，写 100,101.blk 占 2 次

## (2) 实现两阶段多路归并排序算法 (TPMMS)

问题分析：

由于内存缓冲区 buf 中最多存放 8 个块，其中 1 个作为读缓冲，1 个作为写缓冲，则剩余 6 个块可用，因此选择 6 路归并。开始 WHILE(TRUE) 的大循环，每次迭代选择不多于 6 各块，先对其进行块内排序，再作多路归并。下为内排序的实现：

```
while (TRUE) {
    bufferedWriter = getNewBlockInBuffer(&buf);
    remnantBlock = termReadAddress-readAddress;    //剩余未参与排序的块数
    Tuple tempTuple;
    if(++remnantBlock >= 6) {
        remnantBlock = 6;    //大于6块则选择前6块进行排序
    }
    for(int i=0; i<remnantBlock; i++) {    //读入block并进行内部排序
        if ((bufferedBlock[i] = readBlockFromDisk(readAddress+i, &buf)) == NULL) {
            perror("Reading Block Failed!\n");
            return;
        }
        extractTuple(bufferedBlock[i], bufferedTuple);
        for(int j=0; j<TUPLES_PER_BLOCK; j++) {    //内排序，采用选择排序
            temp = bufferedTuple[j].x;    //每轮选择Tuple中X值最小的元组提至最前
            order = j;
            for (int k=j; k<TUPLES_PER_BLOCK; k++) {
                if (bufferedTuple[k].x < temp) {
                    temp = bufferedTuple[k].x;
                    order = k;
                }
            }
            tempTuple = bufferedTuple[j];
            bufferedTuple[j] = bufferedTuple[order];
            bufferedTuple[order] = tempTuple;
            itoa(bufferedTuple[j].x, tempConvert, 10);    //内排序后写回block
            memcpy(bufferedBlock[i]+j*TUPLE_LENGTH, tempConvert, ELEMENT_LENGTH);
            itoa(bufferedTuple[j].y, tempConvert, 10);
            memcpy(bufferedBlock[i]+j*TUPLE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
        }
    }
}
```

对读入的 $\leq 6$  块内排序完毕后进行第一次多路归并，以 indexVector[6]的每一维来标记一个维度已归并元组的下标，以 tempTuple[6]暂存每路当前的首个元组，再在这 6 个元素中，选择 Tuple.X 最小的元组，记录其下标为 order，将其写入写缓冲，并将此路的 indexVector+1，以标识此路的此元组已归并完毕。若某路的 indexVector==TUPLES\_PER\_BLOCK，也即此路已归并的元组数等于块中可容纳的元组总数(7)，将此路的 tempTuple.X 置为一个上界数(宏定义 UPPER\_LIMIT)，以确保以后不会再选择此路元素。当写缓冲区中有 7 个元组时，写磁盘并清空写缓冲。如此进行若干次后，有若干组(每组 $\leq 6$  块)已归并完毕。

下图为第一次归并的算法实现：

```

int indexVector[6] = {0}; //第1次多路归并, indexVector的每个维标识一个block中归并进度的下标
while (!isMerged(indexVector, remnantBlock)) {
    Tuple element[6]; //暂存每路中的极小值
    for (int i = 0; i < remnantBlock; i++) {
        if (indexVector[i] < TUPLES_PER_BLOCK) {
            memcpy(tempConvert, bufferedBlock[i]+indexVector[i]*TUPLIE_LENGTH, ELEMENT_LENGTH);
            element[i].x = atoi(tempConvert);
            memcpy(tempConvert, bufferedBlock[i]+indexVector[i]*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
            element[i].y = atoi(tempConvert);
        } else {
            element[i].x = UPPER_LIMIT; //此路已归并完毕, 将此路的S.C设为上界, 确保以后不会再选择此路
        }
    }
    temp = UPPER_LIMIT;
    for (int i = 0; i < remnantBlock; i++) {
        if (element[i].x < temp) {
            temp = element[i].x;
            order = i; //选择6路中极小值的下标
        }
    }
    itoa(element[order].x, tempConvert, 10); //将极小值并入总集
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH, tempConvert, ELEMENT_LENGTH);
    itoa(element[order].y, tempConvert, 10);
    memcpy(bufferedWriter+(writeOrder++)*TUPLIE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
    if (writeOrder % TUPLES_PER_BLOCK == 0) {
        itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLIE_LENGTH, 10);
        if (writeBlockToDisk(bufferedWriter, writeAddress, &buf) != 0) {
            perror("Writing Block Failed!\n");
            return;
        }
        writeOrder = 0;
        writeAddress++;
        bufferedWriter = getNewBlockInBuffer(&buf);
    }
    if (indexVector[order] < TUPLES_PER_BLOCK) { //若此块还未归并完毕
        indexVector[order]++; //此块的归并进度下标自增
    }
    readAddress += remnantBlock;
    freeBlockInBuffer(bufferedWriter, &buf);
    for (int i = 0; i < remnantBlock; i++) {
        freeBlockInBuffer(bufferedBlock[i], &buf);
    }
}

```

接下来进行第二次多路归并, 由于 R 有 16 个块, S 有 32 个块,  $\log_6 16$  和  $\log_6 32$  均小于 2, 因此两次 6 路归并即可完成所有的排序。第二次归并与第一次思路相似, 但有一些值得注意的点: 先计算路数以及最后一路含有的块数 (每个块都是满的, 因此不用考虑残块), 在代码中用 segment 和 fragment 作标识, 每路归并进度由 fragmentIndexVector[6] 中的一个维度标识。与第一次归并不同的是, 每路有多个块, 当一个块归并完毕后需要读入此路的下一个块继续参与归并。

下图为第二次归并算法的实现：

```

while (TRUE) {
    for (int i = 0; i < segmentCount; i++) { //复制每个segment中剩余block的第一个元组
        if (fragmentIndexVector[i] < TUPLES_PER_BLOCK) {
            memcpy(tempConvert, bufferedBlock[i]+fragmentIndexVector[i]*TUPLIE_LENGTH, ELEMENT_LENGTH);
            bufferedTuple[i].x = atoi(tempConvert);
            memcpy(tempConvert, bufferedBlock[i]+fragmentIndexVector[i]*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
            bufferedTuple[i].y = atoi(tempConvert);
        } else {
            bufferedTuple[i].x = UPPER_LIMIT;
        }
    }
    temp = UPPER_LIMIT;
    for (int i = 0; i < segmentCount; i++) {
        if (bufferedTuple[i].x < temp) {
            temp = bufferedTuple[i].x;
            order = i;
        }
    }
    itoa(bufferedTuple[order].x, tempConvert, 10); //选择最小元组写入写缓冲区
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH, tempConvert, ELEMENT_LENGTH);
    itoa(bufferedTuple[order].y, tempConvert, 10);
    memcpy(bufferedWriter+writeOrder*TUPLIE_LENGTH+ELEMENT_LENGTH, tempConvert, ELEMENT_LENGTH);
}

```

```

if ((++writeOrder) == TUPLES_PER_BLOCK) {
    itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
    if (writeBlockToDisk(bufferedWriter, (writeAddress++), &buf) != 0) {
        perror("Writing Block Failed!\n");
        return;
    }
    writeOrder = 0;
    bufferedWriter = getNewBlockInBuffer(&buf);
}
if (fragmentIndexVector[order] == 6) {
    if (order == segmentCount-1) {
        if (segmentIndexVector[order] == fragmentCount-1) {
            fragmentIndexVector[order] = 7;
        } else {
            segmentIndexVector[order]++;
            fragmentIndexVector[order] = 0;
            freeBlockInBuffer(bufferedBlock[order], &buf);
            bufferedBlock[order] = readBlockFromDisk(initWriteAddress+order*6+segmentIndexVector[order], &buf);
        }
    } else {
        if (segmentIndexVector[order] == 5) {
            fragmentIndexVector[order] = 7;
        } else {
            segmentIndexVector[order]++;
            fragmentIndexVector[order] = 0;
            freeBlockInBuffer(bufferedBlock[order], &buf);
            bufferedBlock[order] = readBlockFromDisk(initWriteAddress+order*6+segmentIndexVector[order], &buf);
        }
    }
} else {
    fragmentIndexVector[order]++;
}
if (isMerged(fragmentIndexVector, segmentCount)) {
    break;
}
}
}

```

在两次归并中，使用了一些磁盘 block 作为中间结果的暂存块，在归并完成后，将这些暂存块清除。最后，在控制台打印 IO 次数。

实验结果：

R 的排序结果存储在 301~316.blk, S 的排序结果存储在 317~348.blk

控制台输出 IO 次数：

-----  
两阶段多路归并排序算法  
-----

对R的多路归并排序完成，IO读写96次  
对S的多路归并排序完成，IO读写192次

打开 301,302,303.blk 进行检验，发现确实已按照 R.A 进行升序排序，并在块尾标识了下个块的地址，证明了此算法的正确性。

|         |     |    |  |
|---------|-----|----|--|
| 301.blk | 文件  | 编辑 | 查看   |
| 81      | 247 | 81 | 215 81 259 82 273 83 269 83 289 84 208 302 |
| 302.blk | 文件  | 编辑 | 查看   |
| 85      | 264 | 87 | 214 87 260 88 255 90 259 92 258 95 207 303 |
| 303.blk | 文件  | 编辑 | 查看   |
| 95      | 257 | 95 | 247 95 265 95 236 96 272 96 203 97 229 304 |

### (3) 实现基于索引的关系选择算法

问题分析：

首先要建立索引，基于任务 2 的排序结果，索引建立流思路如下：顺序读取排序后的块，获取 Tuple.X，用两个 int 型数据来存储索引键，currentKey 和 nextKey，每次读 Tuple 都更新 nextKey=Tuple.X，并与 currentKey 比较，相等则继续读取下一个 Tuple，不等则建立索引，并更新 currentKey=nextKey。从性能和存储角度考虑，本次实验选择建立稀疏索引，即对于每个 block，只建立一个索引，键为其中第一次 currentKey≠nextKey 时的 currentKey 值。将 317~348.blk 中的 Tuple 看成一个序列，索引键对应的索引值为这个键第一次出现时的序列号。例如在这个序列中，(x,y)是第 24 个元组，则索引键值为：(x,24)。建立索引的算法实现如下图：

```
for(int readAddress=SORTED_S_INIT_ADDR; readAddress<SORTED_S_INIT_ADDR+BLOCK_S_LENGTH; readAddress++) {
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    Tuple bufferedTuple[TUPLES_PER_BLOCK];
    extractTuple(bufferedReader, bufferedTuple);
    int currentKey = bufferedTuple[0].x;
    int nextKey = currentKey;
    for(int i=1; i<TUPLES_PER_BLOCK; i++) {
        if((nextKey=bufferedTuple[i].x) != currentKey) { //找到相异元素则建立索引
            int indexValue = (readAddress-SORTED_S_INIT_ADDR)*TUPLES_PER_BLOCK+i; //索引值
            char temp[ELEMENT_LENGTH+1];
            itoa(bufferedTuple[i].x, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLE_LENGTH, temp, ELEMENT_LENGTH);
            sprintf(temp, "%-4d", indexValue);
            memcpy(bufferedWriter+writeCount*TUPLE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
            if (++writeCount == 7) {
                itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
                printf("索引写入磁盘: %d\n", writeAddress);
                if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return;
                }
                indexLength++;
                writeCount = 0;
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
            freeBlockInBuffer(bufferedReader, &buf);
            break;
        }
        currentKey = nextKey;
    }
    if (i == TUPLES_PER_BLOCK-1) {
        freeBlockInBuffer(bufferedReader, &buf);
    }
}
}
```



建立索引后，要根据索引进行搜索，因为建立的是稀疏索引，因此每次需要读入相邻的两个索引  $initIndex$ ,  $termIndex$ ，并判断要搜索的键是否在  $(initIndex, termIndex]$  区间内，若在，则从  $initIndex$  对应的索引值出开始查询，若  $R.S$  与搜索值相等则写入缓存，缓存满则将结果写入磁盘，在遇到第一个相异值时结束。此外，由于区间采用半开半闭，若  $initIndex \geq$  搜索值，则还需回溯至上个索引块进行查询。下为查询的第一阶段，查找索引键值的算法实现：

```
for(int readAddress = INDEX_INIT_ADDR; readAddress < INDEX_INIT_ADDR+indexLength; readAddress++) { //遍历索引块
    printf("读入索引块%d\n", readAddress);
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    Tuple bufferedTuple[TUPLES_PER_BLOCK];
    extractTuple(bufferedReader, bufferedTuple);
    for(int i=0; i<TUPLES_PER_BLOCK-1; i++) {
        if (i == 0 && bufferedTuple[i].x > 107) { //每个索引块的第二个索引键与上个块的最后一个索引键对比
            if (readAddress == INDEX_INIT_ADDR) { //第一个索引块无需和前块对比
                initIndex = dataAddrOffset;
                termIndex = bufferedTuple[i].y+dataAddrOffset;
                break;
            } else { //回溯上个块
                char* tempReader = readBlockFromDisk(readAddress-1, &buf);
                Tuple tempTuple[TUPLES_PER_BLOCK];
                extractTuple(tempReader, tempTuple);
                if (tempTuple[TUPLES_PER_BLOCK-1].x <= 107) {
                    initIndex = tempTuple[TUPLES_PER_BLOCK-1].y+dataAddrOffset;
                    termIndex = bufferedTuple[i].y+dataAddrOffset;
                    freeBlockInBuffer(tempReader, &buf);
                    break;
                }
            }
        }
        if (bufferedTuple[i].x <= 107 && bufferedTuple[i+1].x > 107) { //如果S.C=107在索引区间内
            initIndex = bufferedTuple[i].y + dataAddrOffset;
            termIndex = bufferedTuple[i+1].y + dataAddrOffset;
            printf("已找到索引\n");
            isIndexFound = true;
            break;
        }
    }
    freeBlockInBuffer(bufferedReader, &buf);
    if(isIndexFound){
        break;
    }
}
if (initIndex == termIndex) {
    printf("Can not find by index!\n");
    return;
}
```

下为查询的第二阶段，根据索引值和偏移量查找数据块的算法实现：

```
for (int readAddress = initReadAddress; readAddress <= termReadAddress; readAddress++) {
    Tuple bufferedTuple[7]; //根据索引值读取磁盘数据块
    printf("读入数据块%d\n", readAddress);
    bufferedReader = readBlockFromDisk(readAddress, &buf);
    extractTuple(bufferedReader, bufferedTuple);
    int currentIndex = 0, termIndex = 7;
    if(readAddress == initReadAddress){
        currentIndex = initReadIndex;
    }
    if(readAddress == termReadAddress) {
        termIndex = termReadIndex;
    }
    while(currentIndex < termIndex) {
        if (bufferedTuple[currentIndex].x == 107) {
            validTuples++;
            printf("(X=%d, Y=%d)\n", bufferedTuple[currentIndex].x, bufferedTuple[currentIndex].y);
            char temp[ELEMENT_LENGTH+1];
            itoa(bufferedTuple[currentIndex].x, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLE_LENGTH, temp, ELEMENT_LENGTH);
            itoa(bufferedTuple[currentIndex].y, temp, 10);
            memcpy(bufferedWriter+writeCount*TUPLE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
            if (++writeCount == TUPLES_PER_BLOCK) {
                itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
                printf("注：结果写入磁盘:%d\n", writeAddress);
                if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
                    perror("Writing Block Failed!\n");
                    return;
                }
                writeCount = 0;
                bufferedWriter = getNewBlockInBuffer(&buf);
            }
        }
        currentIndex++;
    }
    freeBlockInBuffer(bufferedReader, &buf);
}
```

实验结果：

索引块写在 350~354.blk 中，基于索引的搜索结果写在 120,121.blk 中。

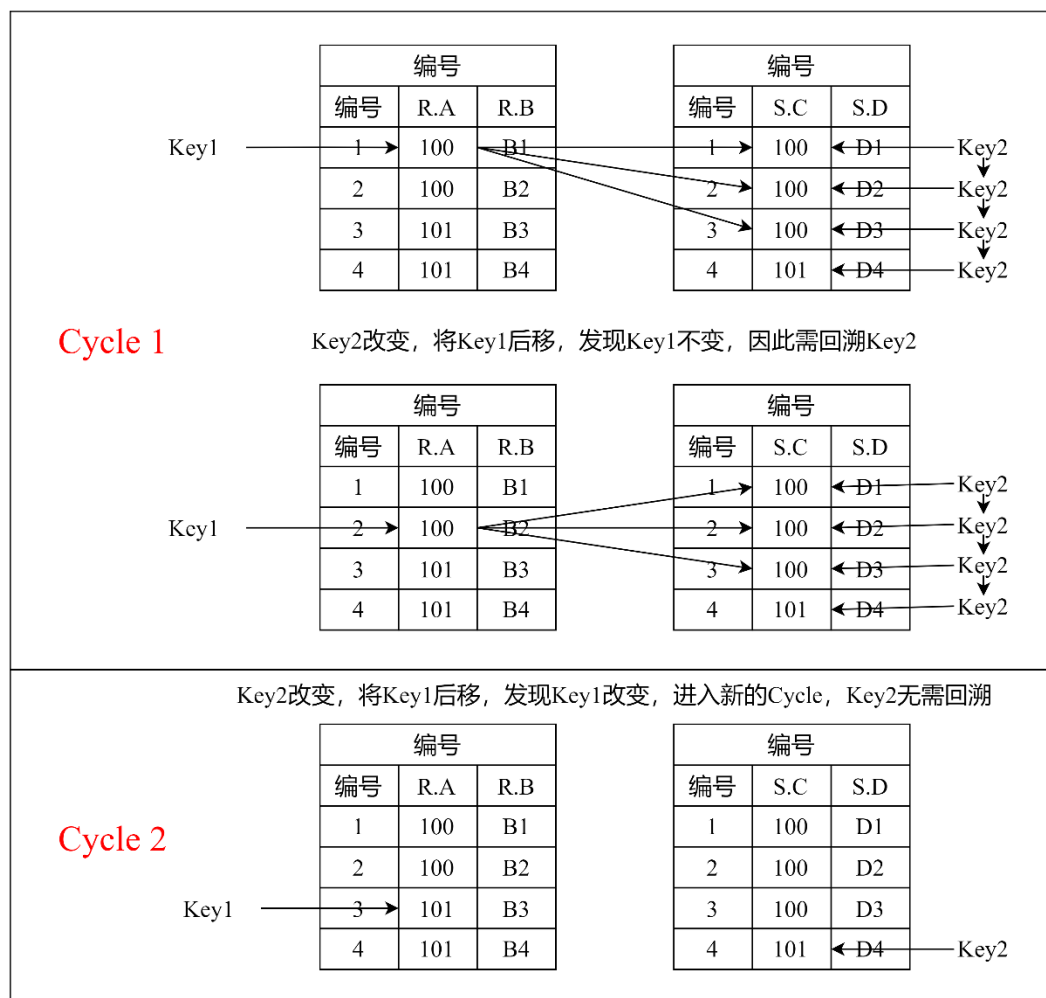
控制台输出：可见，基于索引查找只用了 6 次 IO，相比线性查找 34 次 IO 有极大优化。

```
-----  
基于索引的关系选择算法  
-----  
开始建立索引：  
索引写入磁盘：350  
索引写入磁盘：351  
索引写入磁盘：352  
索引写入磁盘：353  
索引写入磁盘：354  
建立索引完成，索引长度为5个Block，IO次数为：37  
根据索引进行查找：  
读入索引块350  
已找到索引  
读入数据块319  
(X=107, Y=241)  
读入数据块320  
(X=107, Y=209)  
(X=107, Y=317)  
(X=107, Y=363)  
(X=107, Y=393)  
(X=107, Y=222)  
(X=107, Y=356)  
注：结果写入磁盘:120  
(X=107, Y=248)  
读入数据块321  
注：结果写入磁盘:121  
满足选择条件的元组一共:8个  
IO读写一共6次
```

**(4) 实现基于排序的连接操作算法 (Sort-Merge-Join)**

问题分析：

这个算法基于任务 2 的已排序的数据集（即为第一趟），基本思想为：对关系模式 R,S 分别进行迭代，以 R.A 为 Key1，将 Key1 值改变设为一个 Cycle，S.C 为 Key2，以 Key1 为基准将 Key2 向后迭代，当第一次 Key1=Key2 时，对 S 和 R 进行 Join 操作，再将 Key2 后移并 Join 直至 Key2 值改变为止。但考虑到可能有连续的几个 R 中的元组，其 R.A 都等于 Key1，因此，需要将 Key1 后移并进行检查，若 Key1 改变，则此次 Cycle 已完成，进入下一个 Cycle 即可；若 Key1 未变，则说明还有 R.A 相同的元组需要和 S 中的元组连接，此时，就需要将 Key2 回溯至此 Cycle 的初始位置，再对当前的 Key1 标识的元组进行 Join。下为示意图：



在上述的 Key2 回溯过程的具体实现中，有时会涉及到 Block 的回溯，因此实现的

代码较长，如下图：

```

463 d sortedInnerJoin() {
464     Buffer buf;
465     if (!initBuffer(520, 64, &buf)) {
466         perror("Buffer Initialization Failed!\n");
467         return;
468     }
469     int innerJoinKey = 0, innerJoinOperationTicker = 0;
470     char* bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR, &buf);
471     char* bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR, &buf);
472     char* bufferedWriter = getNewBlockInBuffer(&buf);
473     int writeAddress = SORTED_INNERJOIN_INIT_ADDR;
474     int writeCount = 0, writeIndexR = 0, writeIndexS = 0, currentIndexR = 0, currentIndexS = 0;
475     Tuple bufferedTupleR[TUPLES_PER_BLOCK], bufferedTupleS[TUPLES_PER_BLOCK];
476     extractTuple(bufferedReaderR, bufferedTupleR);
477     extractTuple(bufferedReaderS, bufferedTupleS);
478     bool isJoinedR = false, isJoinedS = false; // 记录前一个元素和当前元素是否发生了连接
479     while(TRUE) { // 开始遍历两个关系模式
480         if (bufferedTupleR[currentIndexR].x == bufferedTupleS[currentIndexS].x) { // R.A=S.C, 可以进
481             innerJoinOperationTicker++;
482             isJoinedS = true; // 将连接结果写入block缓冲
483             char temp[ELEMENT_LENGTH+1];
484             itoa(bufferedTupleS[currentIndexS].x, temp, 10);
485             memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
486             itoa(bufferedTupleS[currentIndexS].y, temp, 10);
487             memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
488             itoa(bufferedTupleR[currentIndexR].x, temp, 10);
489             memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+2*ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
490             itoa(bufferedTupleR[currentIndexR].y, temp, 10);
491             memcpy(bufferedWriter+writeCount*4*ELEMENT_LENGTH+3*ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
492             writeCount++;
493             if (writeCount == 3) { // 每个块可容纳3个4元组
494                 memset(bufferedWriter+48, 0, 4*ELEMENT_LENGTH+ELEMENT_LENGTH); // 写入磁盘
495                 itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
496                 printf("注：结果写入磁盘:%d\n", writeAddress);
497                 if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
498                     perror("Writing Block Failed!\n");
499                     return;
500                 }
501                 writeCount = 0;
502                 freeBlockInBuffer(bufferedWriter, &buf);
503                 bufferedWriter = getNewBlockInBuffer(&buf);
504             }
505             if ((++currentIndexS) == TUPLES_PER_BLOCK) {
506                 if ((++writeIndexS) == BLOCK_S_LENGTH) {
507                     break;
508                 }
509                 currentIndexS = 0;
510                 freeBlockInBuffer(bufferedReaderS, &buf);
511                 bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+writeIndexS, &buf);
512                 extractTuple(bufferedReaderS, bufferedTupleS);
513             }
514             } else if (bufferedTupleR[currentIndexR].x > bufferedTupleS[currentIndexS].x) { // R.A>S.C
515                 currentIndexS++;
516                 if (currentIndexS == TUPLES_PER_BLOCK) {
517                     writeIndexS++;
518                 }
519             }
520             if (writeIndexS == BLOCK_S_LENGTH) {
521                 break;
522             }
523             currentIndexS = 0;
524             freeBlockInBuffer(bufferedReaderS, &buf);
525             bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
526             extractTuple(bufferedReaderS, bufferedTupleS);
527         } else {
528             if (bufferedTupleR[currentIndexR].x == innerJoinKey && isJoinedR && !isJoinedS) { // R
529                 if (currentIndexS == 0) {
530                     if (writeIndexS == 0) {
531                         break;
532                     }
533                     } else {
534                         currentIndexS = 6;
535                         freeBlockInBuffer(bufferedReaderS, &buf);
536                         bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+(-writeIndexS), &buf);
537                         extractTuple(bufferedReaderS, bufferedTupleS);
538                     }
539                     } else {
540                         currentIndexS--;
541                     }
542                     while(bufferedTupleS[currentIndexS].x == bufferedTupleR[currentIndexR].x) {
543                         if (currentIndexS == 0) {
544                             if (writeIndexS == 0) {
545                                 break;
546                             }
547                             currentIndexS = 6;
548                             freeBlockInBuffer(bufferedReaderS, &buf);
549                             bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR+(-writeIndexS), &buf);
550                             extractTuple(bufferedReaderS, bufferedTupleS);
551                         }
552                         } else {
553                             currentIndexS--;
554                         }
555                     } else {
556                         isJoinedR = isJoinedS;
557                         isJoinedS = false;
558                         innerJoinKey = bufferedTupleR[currentIndexR++].x;
559                         if (currentIndexR == TUPLES_PER_BLOCK) {
560                             if ((++writeIndexR) == BLOCK_R_LENGTH) {
561                                 break;
562                             }
563                             } else {
564                                 currentIndexR = 0;
565                                 freeBlockInBuffer(bufferedReaderR, &buf);
566                                 bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + writeIndexR, &buf);
567                                 extractTuple(bufferedReaderR, bufferedTupleR);
568                             }
569                             }
570                             }
571                             if (writeCount) {
572                                 memset(bufferedWriter+writeCount*4*ELEMENT_LENGTH, 0, 64-writeCount*4*ELEMENT_LENGTH);
573                                 itoa(writeAddress+1, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
574                                 printf("注：结果写入磁盘:%d\n", writeAddress);
575                                 if (writeBlockToDisk(bufferedWriter, writeAddress++, &buf) != 0) {
576                                     perror("Writing Block Failed!\n");
577                                 }
578                             }

```

实验结果：

控制台输出如下：

连接结果写在磁盘块 400~478.blk 中

```

-----
基于排序的连接操作算法
-----
注：结果写入磁盘:400
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
连接完成，共执行连接操作235次
IO操作次数为：160

```

(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:

这里分析基于散列的两趟扫描集合交算法, (并和差思路类似, 写在附加题中):

这个算法与集合连接(Join)算法的思路相似, 第一趟排序已完成, 在第二趟查找迭代中, Join 中做的是, 如果  $R.A=S.C$ , 则  $\text{Tuple}(R) \times \text{Tuple}(S)$ ; 而对于集合交(Intersection), 则需要  $R.A=S.C$  且  $R.B=S.D$  时, 保存 Tuple。虽然操作不同, 但回溯的思想与 Join 类似。

代码如下:

```
mem.c x test.c x AuxiliaryFunctions.c x defines.h x *CoreFunction.c x extmem.h x
605 while(TRUE) {
606     if (bufferedTupleR[currentIndexR].x == bufferedTuples[currentIndexS].x) { //如果R.A=S.C, 则比较
607         if (!currentIndexS) { //回溯S, 从首次交点开始
608             if (writeIndexS) {
609                 currentIndexS = 0;
610                 freeBlockInBuffer(bufferedReaderS, &buf);
611                 bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + (--writeIndexS), &buf);
612                 extractTuple(bufferedReaderS, bufferedTuples);
613             }
614         }
615         else {
616             currentIndexS--;
617         }
618         if (writeIndexS != 0 || currentIndexS != 0) {
619             while(bufferedTuples[currentIndexS].x == bufferedTupleR[currentIndexR].x) { //R.A=S.C, 且S
620                 if (!currentIndexS) {
621                     if (!writeIndexS) {
622                         break;
623                     }
624                     else {
625                         currentIndexS = TUPLES_PER_BLOCK;
626                         freeBlockInBuffer(bufferedReaderS, &buf);
627                         bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + (--writeIndexS), &buf);
628                         extractTuple(bufferedReaderS, bufferedTuples);
629                     }
630                 }
631                 currentIndexS--;
632             }
633             if ((++currentIndexS) == TUPLES_PER_BLOCK) {
634                 if ((++writeIndexS) == BLOCK_S_LENGTH) {
635                     break;
636                 }
637                 currentIndexS = 0;
638                 freeBlockInBuffer(bufferedReaderS, &buf);
639                 bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
640                 extractTuple(bufferedReaderS, bufferedTuples);
641             }
642         }
643     }
644     while(bufferedTuples[currentIndexS].x == bufferedTupleR[currentIndexR].x) { //R.A=S.C
645         if (bufferedTuples[currentIndexS].y == bufferedTupleR[currentIndexR].y) { //R.B=S.D
646             intersectionCount++;
647             char temp[ELEMENT_LENGTH*1];
648             itoa(bufferedTuples[currentIndexS].x, temp, 10); //找到交元素, 写入block
649             memcpy(bufferedWriter+writeIndexS*TUPLE_LENGTH, temp, ELEMENT_LENGTH);
650             itoa(bufferedTuples[currentIndexS].y, temp, 10);
651             memcpy(bufferedWriter+writeIndexS*TUPLE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
652             if ((++writeIndexS) == TUPLES_PER_BLOCK) {
653                 memset(bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 0, TUPLE_LENGTH);
654                 itoa(writeIndexS, bufferedWriter+TUPLES_PER_BLOCK*TUPLE_LENGTH, 10);
655                 printf("注: 结果写入磁盘: %d\n", writeIndexS);
656                 if (writeBlockToDisk(bufferedWriter, (writeIndexS++), &buf) != 0) {
657                     perror("Writing Block Failed!\n");
658                     return;
659                 }
660                 writeIndexS = 0;
661                 memcpy(bufferedWriter, emptyString, 64);
662                 bufferedWriter = getNewBlockInBuffer(&buf);
663             }
664         }
665         if ((++currentIndexS) == TUPLES_PER_BLOCK) {
666             if ((++writeIndexS) == BLOCK_S_LENGTH) {
667                 break;
668             }
669             if ((++writeIndexS) == BLOCK_S_LENGTH) {
670                 break;
671             }
672             currentIndexS = 0;
673             freeBlockInBuffer(bufferedReaderS, &buf);
674             bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
675             extractTuple(bufferedReaderS, bufferedTuples);
676         }
677         intersectKey = bufferedTupleR[currentIndexR].x;
678         if ((++currentIndexR) == TUPLES_PER_BLOCK) {
679             readIndexR++;
680             if (readIndexR == BLOCK_R_LENGTH) {
681                 break;
682             }
683             currentIndexR = 0;
684             freeBlockInBuffer(bufferedReaderR, &buf);
685             bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + readIndexR, &buf);
686             extractTuple(bufferedReaderR, bufferedTupleR);
687         }
688         if (currentIndexS == TUPLES_PER_BLOCK) {
689             if ((++writeIndexS) == BLOCK_S_LENGTH) {
690                 break;
691             }
692             currentIndexS = 0;
693             freeBlockInBuffer(bufferedReaderS, &buf);
694             bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
695             extractTuple(bufferedReaderS, bufferedTuples);
696         }
697         else if (bufferedTupleR[currentIndexR].x > bufferedTuples[currentIndexS].x) {
698             intersectKey = bufferedTuples[currentIndexS].x;
699             if ((++currentIndexS) == TUPLES_PER_BLOCK) {
700                 if ((++writeIndexS) == BLOCK_S_LENGTH) {
701                     break;
702                 }
703                 currentIndexS = 0;
704                 freeBlockInBuffer(bufferedReaderS, &buf);
705                 bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + writeIndexS, &buf);
706                 extractTuple(bufferedReaderS, bufferedTuples);
707             }
708         }
709         else {
710             if (bufferedTupleR[currentIndexR].x == intersectKey) {
711                 if (currentIndexS == 0) {
712                     if (writeIndexS) {
713                         currentIndexS = 0;
714                         freeBlockInBuffer(bufferedReaderS, &buf);
715                         bufferedReaderS = readBlockFromDisk(SORTED_S_INIT_ADDR + (--writeIndexS), &buf);
716                         extractTuple(bufferedReaderS, bufferedTuples);
717                     }
718                 }
719                 else {
720                     currentIndexS--;
721                 }
722             }
723             if ((++currentIndexR) == TUPLES_PER_BLOCK) {
724                 if ((++readIndexR) == BLOCK_R_LENGTH) {
725                     break;
726                 }
727                 currentIndexR = 0;
728                 freeBlockInBuffer(bufferedReaderR, &buf);
729                 bufferedReaderR = readBlockFromDisk(SORTED_R_INIT_ADDR + readIndexR, &buf);
730                 extractTuple(bufferedReaderR, bufferedTupleR);
731             }
732         }
733     }
734 }
```

## 实验结果:

控制台输出如下:

集合交结果写在磁盘块 500,501.blk 中

```
基于排序的集合交操作算法
(X=101, Y=252)
(X=102, Y=221)
(X=104, Y=260)
(X=105, Y=286)
(X=105, Y=241)
(X=108, Y=291)
(X=116, Y=275)
注: 结果写入磁盘: 500
(X=119, Y=259)
(X=121, Y=265)
(X=130, Y=232)
(X=144, Y=290)
注: 结果写入磁盘: 501
R和S的交集中共有11个元组
IO操作共99次
```

## 五、 附加题

对剩余的两种集合操作进行问题分析，并给出程序正确运行的结果截图。

集合交(Intersection)在第<四>部分已分析过，这里分析并和差。

### 1. 并

这个算法也是基于排序结果的，对 R、S 分别建立迭代指针，因 R.A 的值域下界小于 S.C，因此从 R.A 开始迭代，将遇到的元组拷贝至写缓存中(满了就写磁盘，与前面相同)，直到第一次遇到 R.A=S.C 为止。此时依然是使用回溯操作，对于 Key1，设置一个 Flag 来表示是否遇到相同的元组，每次 Key1 后移时置为 false，每轮中，Key1 不变，Key2 后移，将 Key2 标识的元组拷贝至写缓存，并检查是否有 S.C=R.A，没有则直接将 Key1 标识的元组写入写缓存，继续检查是否满足 S.D=R.B，若有则将 Flag 置为 true，每轮结束后若 Flag 为 false(即 S 中无与此 R 相同的元组)，则将 Key1 标识的元组写入写缓存。算法的核心思想即如果有相同元组则只写一次。实现如下：

```
extmem.c test.c AuxiliaryFunctions.c defines.h *CoreFunction.c extmem.h
773 while(!completeFlag) {
774     if(bufferedReaderR > bufferedDataS) {
775         if(!isUnite) {
776             itoa(bufferedReaderS, temp, 10);
777             memcpy(bufferedReaderS+readIndex*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
778             memcpy(temp, bufferedReaderS+writeIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
779             bufferedTuples.y = atoi(temp);
780             memcpy(bufferedReaderR+readIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
781             unionTucker++;
782             if(++readIndex == TUPLES_PER_BLOCK) {
783                 itoa(writeAddress+writeCount+1, bufferedWriter+readIndex*TUPLIE_LENGTH, 10);
784                 if (writeBlockToDisk(bufferedReaderR, writeAddress+writeCount, &buf) != 0) {
785                     perror("Writing Block Failed!\n");
786                     return;
787                 }
788                 readIndex = 0;
789                 printf("注:结果写入磁盘:%d\n", writeAddress+(writeCount++));
790                 bufferedWriter = getNewBlockInBuffer(&buf);
791             }
792             isUnite = 0;
793             if(++writeIndexS == TUPLES_PER_BLOCK) {
794                 if(++readAddressS <= SORTED_S_INIT_ADDR+BLOCK_S_LENGTH-1) {
795                     freeBlockInBuffer(bufferedReaderS, &buf);
796                     bufferedReaderS = readBlockFromDisk(readAddressS, &buf);
797                     writeIndexS = 0;
798                 } else {
799                     completeFlag = true;
800                 }
801             }
802             if(!completeFlag) {
803                 memcpy(temp, bufferedReaderS+writeIndex*TUPLIE_LENGTH, ELEMENT_LENGTH);
804                 currentDataS = atoi(temp);
805                 previousDataS = bufferedDataS;
806             }
807             bool retraceFlag = !completeFlag && unionFlag && currentDataS == bufferedDataS;
808             if(retraceFlag) {
809                 if(readAddressR != currentAddressKey) {
810                     freeBlockInBuffer(bufferedReaderR, &buf);
811                     bufferedReaderR = readBlockFromDisk(currentAddressKey, &buf);
812                 }
813                 writeIndexR = currentTupleKey;
814                 readAddressR = currentAddressKey;
815                 memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
816                 bufferedDataR = atoi(temp);
817             } else {
818                 unionFlag = false;
819                 if(currentDataS != bufferedDataS) {
820                     bufferedDataS = currentDataS;
821                     isShifted = true;
822                 }
823             }
824             } else if(bufferedReaderR < bufferedDataS) {
825                 itoa(bufferedReaderR, temp, 10);
826                 memcpy(bufferedReaderR+readIndex*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
827                 memcpy(temp, bufferedReaderR+writeIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
828                 bufferedTuples.y = atoi(temp);
829                 memcpy(bufferedReaderR+readIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
830                 unionTucker++;
831                 if(++readIndex == TUPLES_PER_BLOCK) {
832                     itoa(writeAddress+writeCount+1, bufferedWriter+readIndex*TUPLIE_LENGTH, 10);
833                     if (writeBlockToDisk(bufferedReaderR, writeAddress+writeCount, &buf) != 0) {
834                         perror("Writing Block Failed!\n");
835                         return;
836                     }
837                     readIndex = 0;
838                     printf("注:结果写入磁盘:%d\n", writeAddress+(writeCount++));
839                     bufferedWriter = getNewBlockInBuffer(&buf);
840                 }
841                 if(previousDataR != bufferedDataR && isShifted) {
842                     isShifted = 0;
843                     currentTupleKey = writeIndexR;
844                     currentAddressKey = readAddressR;
845                 }
846                 if(++writeIndexR == TUPLES_PER_BLOCK) {
847                     if(++readAddressR <= SORTED_R_INIT_ADDR+BLOCK_R_LENGTH-1) {
848                         writeIndexR = 0;
849                         freeBlockInBuffer(bufferedReaderR, &buf);
850                         bufferedReaderR = readBlockFromDisk(readAddressR, &buf);
851                     } else {
852                         completeKeyFlag = true;
853                         bufferedDataR = UPPER_LIMIT;
854                     }
855                 }
856                 if(!completeKeyFlag) {
857                     memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
858                     currentDataR = atoi(temp);
859                     previousDataR = bufferedDataR;
860                     bufferedDataR = currentDataR;
861                 }
862             }
863             if(!completeFlag) {
864                 previousDataR = bufferedDataR;
865                 memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
866                 currentDataR = atoi(temp);
867                 if(currentDataR != bufferedDataR) {
868                     bufferedDataR = currentDataR;
869                     currentTupleKey = writeIndexR;
870                     currentAddressKey = readAddressR;
871                 }
872                 else if(bufferedReaderS == bufferedDataR) {
873                     memcpy(temp, bufferedReaderS+writeIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
874                     bufferedTuples.y = atoi(temp);
875                     memcpy(temp, bufferedReaderR+writeIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
876                     bufferedTuples.y = atoi(temp);
877                     unionFlag = true;
878                     if(bufferedReaderS.y == bufferedTuples.y && previousDataS == bufferedDataS) {
879                         isUnite = true;
880                     } else if(bufferedReaderS.y != bufferedTuples.y && previousDataS != bufferedDataS) {
881                         itoa(bufferedReaderR, temp, 10);
882                         memcpy(bufferedReaderR+readIndex*TUPLIE_LENGTH, temp, ELEMENT_LENGTH);
883                         itoa(bufferedReaderR, temp, 10);
884                         memcpy(bufferedReaderR+readIndex*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
885                         unionTucker++;
886                         if(++readIndex == TUPLES_PER_BLOCK) {
887                             itoa(writeAddress+writeCount+1, bufferedWriter+readIndex*TUPLIE_LENGTH, 10);
888                             if (writeBlockToDisk(bufferedReaderR, writeAddress+writeCount, &buf) != 0) {
889                                 perror("Writing Block Failed!\n");
890                                 return;
891                             }
892                             readIndex = 0;
893                             printf("注:结果写入磁盘:%d\n", writeAddress+(writeCount++));
894                             bufferedWriter = getNewBlockInBuffer(&buf);
895                         }
896                     }
897                     if(previousDataR != bufferedDataR && isShifted) {
898                         isShifted = 0;
899                         currentTupleKey = writeIndexR;
900                         currentAddressKey = readAddressR;
901                     }
902                     if(++writeIndexR == TUPLES_PER_BLOCK) {
903                         if(++readAddressR <= SORTED_R_INIT_ADDR+BLOCK_R_LENGTH-1) {
904                             writeIndexR = 0;
905                             freeBlockInBuffer(bufferedReaderR, &buf);
906                             bufferedReaderR = readBlockFromDisk(readAddressR, &buf);
907                         } else {
908                             completeKeyFlag = true;
909                             bufferedDataR = UPPER_LIMIT;
910                         }
911                     }
912                     if(!completeKeyFlag) {
913                         memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
914                         currentDataR = atoi(temp);
915                         previousDataR = bufferedDataR;
916                         bufferedDataR = currentDataR;
917                     }
918                 }
919             }
920         }
921     }
922 }
```



控制台输出如下：

集合并结果写在磁盘块 600~646.blk 中

```

-----
基于排序的集合并操作算法
-----
注:结果写入磁盘:600
注:结果写入磁盘:601
注:结果写入磁盘:602
注:结果写入磁盘:603
注:结果写入磁盘:604
注:结果写入磁盘:605
注:结果写入磁盘:606
注:结果写入磁盘:607
注:结果写入磁盘:608
注:结果写入磁盘:609
注:结果写入磁盘:637
注:结果写入磁盘:638
注:结果写入磁盘:639
注:结果写入磁盘:640
注:结果写入磁盘:641
注:结果写入磁盘:642
注:结果写入磁盘:643
注:结果写入磁盘:644
注:结果写入磁盘:645
注:结果写入磁盘:646
R和S的并集中共有325个元组
IO操作共131次

```

## 2. 差

与上述回溯思想类似，只是以 Key2 为主迭代键，设置 Flag，若有与其相同的 Key1 指向的元组，则不写入写缓存，若没有，则将 Key2 指向的元组写入写缓存，并将 Key2 后移，并判断 Key1 是否需要回溯，如此循环直到遍历完 S 集即可。代码如下：

```

ktmem.c  test.c  AuxiliaryFunctions.c  defines.h  CoreFunction.c  extmem.h
947  while(!completeFlag) {
948      if(bufferedReaderR > bufferedDataS) {
949          if(!isDifferentiated) {
950              differenceTicker++;
951              itoa(bufferedReaderR, temp, 10);
952              memcpy(temp, bufferedReaderS+writeIndexS*TUPLIE_LENGTH, ELEMENT_LENGTH);
953              memcpy(temp, bufferedReaderS+writeIndexS*TUPLIE_LENGTH+ELEMENT_LENGTH, ELEMENT_LENGTH);
954              bufferedTupleS.y = atoi(temp);
955              memcpy(bufferedReaderR+readIndexR*TUPLIE_LENGTH+ELEMENT_LENGTH, temp, ELEMENT_LENGTH);
956              if(++readIndexR == TUPLES_PER_BLOCK) {
957                  itoa(writeAddress+writeCount+1, bufferedWriter+readIndexR*TUPLIE_LENGTH, 10);
958                  if (writeBlockToDisk(bufferedReaderR, writeAddress+writeCount, &buf) != 0) {
959                      perror("Writing Block Failed!\n");
960                      return;
961                  }
962                  printf("注:结果写入磁盘:%d\n", writeAddress+(writeCount++));
963                  readIndexR = 0;
964                  bufferedWriter = getNewBlockInBuffer(&buf);
965              }
966          }
967          isDifferentiated = false;
968          if(++writeIndexS == TUPLES_PER_BLOCK) {
969              if(++readAddressS <= SORTED_S_INIT_ADDR+BLOCK_S_LENGTH-1) {
970                  freeBlockInBuffer(bufferedReaderS, &buf);
971                  writeIndexS = 0;
972                  bufferedReaderS = readBlockFromDisk(readAddressS, &buf);
973              } else {
974                  completeFlag = true;
975              }
976          }
977          if(!completeFlag) {
978              memcpy(temp, bufferedReaderS+writeIndexS*TUPLIE_LENGTH, ELEMENT_LENGTH);
979              currentDataS = atoi(temp);
980              previousDataS = bufferedDataS;
981          }
982          bool retraceFlag = !completeFlag && differenceFlag && currentDataS == bufferedDataS;
983          if(retraceFlag) {
984              if(readAddressR != currentAddressKey) {
985                  freeBlockInBuffer(bufferedReaderR, &buf);
986                  bufferedReaderR = readBlockFromDisk(currentAddressKey, &buf);
987              }
988              writeIndexR = currentTupleKey;
989              readAddressR = currentAddressKey;
990              memcpy(temp, bufferedReaderR+writeIndexR*TUPLIE_LENGTH, ELEMENT_LENGTH);
991              bufferedDataR = atoi(temp);
992          } else {
993              if(currentDataS != bufferedDataS) {
994                  isShifted = false;
995                  bufferedDataS = currentDataS;
996              }
997              differenceFlag = false;
998          }
999      } else if(bufferedReaderR < bufferedDataS) {
1000          if(++writeIndexR == TUPLES_PER_BLOCK) {
1001              if(++readAddressR <= SORTED_R_INIT_ADDR+BLOCK_R_LENGTH-1) {
1002                  writeIndexR = 0;
1003                  freeBlockInBuffer(bufferedReaderR, &buf);
1004                  bufferedReaderR = readBlockFromDisk(readAddressR, &buf);
1005              } else {
1006                  completeFlag = true;
1007              }
1008          }
1009      }
1010  }
1011  }
1012  }
1013  }
1014  }
1015  }
1016  }
1017  }
1018  }
1019  }
1020  }
1021  }
1022  }
1023  }
1024  }
1025  }
1026  }
1027  }
1028  }
1029  }
1030  }
1031  }
1032  }
1033  }
1034  }
1035  }
1036  }
1037  }
1038  }
1039  }
1040  }
1041  }
1042  }
1043  }
1044  }
1045  }
1046  }
1047  }
1048  }
1049  }
1050  }
1051  }
1052  }
1053  }
1054  }
1055  }
1056  }
1057  }
1058  }
1059  }
1060  }
1061  }
1062  }
1063  }
1064  }
1065  }
1066  }
1067  }
1068  }
1069  }
1070  }
1071  }
1072  }
1073  }
1074  }
1075  }
1076  }
1077  }
1078  }
1079  }
1080  }
1081  }
1082  }
1083  }
1084  }
1085  }
1086  }
1087  }
1088  }
1089  }
1090  }
1091  }
1092  }
1093  }
1094  }
1095  }
1096  }
1097  }
1098  }
1099  }
1100  }
1101  }
1102  }
1103  }
1104  }
1105  }
1106  }
1107  }
1108  }
1109  }
1110  }
1111  }
1112  }
1113  }
1114  }
1115  }
1116  }
1117  }
1118  }
1119  }
1120  }
1121  }
1122  }
1123  }
1124  }
1125  }
1126  }
1127  }
1128  }
1129  }
1130  }
1131  }
1132  }
1133  }
1134  }
1135  }
1136  }
1137  }
1138  }
1139  }
1140  }
1141  }
1142  }
1143  }
1144  }
1145  }
1146  }
1147  }
1148  }
1149  }
1150  }
1151  }
1152  }
1153  }
1154  }
1155  }
1156  }
1157  }
1158  }
1159  }
1160  }
1161  }
1162  }
1163  }
1164  }
1165  }
1166  }
1167  }
1168  }
1169  }
1170  }
1171  }
1172  }
1173  }
1174  }
1175  }
1176  }
1177  }
1178  }
1179  }
1180  }
1181  }
1182  }
1183  }
1184  }
1185  }
1186  }
1187  }
1188  }
1189  }
1190  }
1191  }
1192  }
1193  }
1194  }
1195  }
1196  }
1197  }
1198  }
1199  }
1200  }
1201  }
1202  }
1203  }
1204  }
1205  }
1206  }
1207  }
1208  }
1209  }
1210  }
1211  }
1212  }
1213  }
1214  }
1215  }
1216  }
1217  }
1218  }
1219  }
1220  }
1221  }
1222  }
1223  }
1224  }
1225  }
1226  }
1227  }
1228  }
1229  }
1230  }
1231  }
1232  }
1233  }
1234  }
1235  }
1236  }
1237  }
1238  }
1239  }
1240  }
1241  }
1242  }
1243  }
1244  }
1245  }
1246  }
1247  }
1248  }
1249  }
1250  }
1251  }
1252  }
1253  }
1254  }
1255  }
1256  }
1257  }
1258  }
1259  }
1260  }
1261  }
1262  }
1263  }
1264  }
1265  }
1266  }
1267  }
1268  }
1269  }
1270  }
1271  }
1272  }
1273  }
1274  }
1275  }
1276  }
1277  }
1278  }
1279  }
1280  }
1281  }
1282  }
1283  }
1284  }
1285  }
1286  }
1287  }
1288  }
1289  }
1290  }
1291  }
1292  }
1293  }
1294  }
1295  }
1296  }
1297  }
1298  }
1299  }
1300  }
1301  }
1302  }
1303  }
1304  }
1305  }
1306  }
1307  }
1308  }
1309  }
1310  }
1311  }
1312  }
1313  }
1314  }
1315  }
1316  }
1317  }
1318  }
1319  }
1320  }
1321  }
1322  }
1323  }
1324  }
1325  }
1326  }
1327  }
1328  }
1329  }
1330  }
1331  }
1332  }
1333  }
1334  }
1335  }
1336  }
1337  }
1338  }
1339  }
1340  }
1341  }
1342  }
1343  }
1344  }
1345  }
1346  }
1347  }
1348  }
1349  }
1350  }
1351  }
1352  }
1353  }
1354  }
1355  }
1356  }
1357  }
1358  }
1359  }
1360  }
1361  }
1362  }
1363  }
1364  }
1365  }
1366  }
1367  }
1368  }
1369  }
1370  }
1371  }
1372  }
1373  }
1374  }
1375  }
1376  }
1377  }
1378  }
1379  }
1380  }
1381  }
1382  }
1383  }
1384  }
1385  }
1386  }
1387  }
1388  }
1389  }
1390  }
1391  }
1392  }
1393  }
1394  }
1395  }
1396  }
1397  }
1398  }
1399  }
1400  }
1401  }
1402  }
1403  }
1404  }
1405  }
1406  }
1407  }
1408  }
1409  }
1410  }
1411  }
1412  }
1413  }
1414  }
1415  }
1416  }
1417  }
1418  }
1419  }
1420  }
1421  }
1422  }
1423  }
1424  }
1425  }
1426  }
1427  }
1428  }
1429  }
1430  }
1431  }
1432  }
1433  }
1434  }
1435  }
1436  }
1437  }
1438  }
1439  }
1440  }
1441  }
1442  }
1443  }
1444  }
1445  }
1446  }
1447  }
1448  }
1449  }
1450  }
1451  }
1452  }
1453  }
1454  }
1455  }
1456  }
1457  }
1458  }
1459  }
1460  }
1461  }
1462  }
1463  }
1464  }
1465  }
1466  }
1467  }
1468  }
1469  }
1470  }
1471  }
1472  }
1473  }
1474  }
1475  }
1476  }
1477  }
1478  }
1479  }
1480  }
1481  }
1482  }
1483  }
1484  }
1485  }
1486  }
1487  }
1488  }
1489  }
1490  }
1491  }
1492  }
1493  }
1494  }
1495  }
1496  }
1497  }
1498  }
1499  }
1500  }
1501  }
1502  }
1503  }
1504  }
1505  }
1506  }
1507  }
1508  }
1509  }
1510  }
1511  }
1512  }
1513  }
1514  }
1515  }
1516  }
1517  }
1518  }
1519  }
1520  }
1521  }
1522  }
1523  }
1524  }
1525  }
1526  }
1527  }
1528  }
1529  }
1530  }
1531  }
1532  }
1533  }
1534  }
1535  }
1536  }
1537  }
1538  }
1539  }
1540  }
1541  }
1542  }
1543  }
1544  }
1545  }
1546  }
1547  }
1548  }
1549  }
1550  }
1551  }
1552  }
1553  }
1554  }
1555  }
1556  }
1557  }
1558  }
1559  }
1560  }
1561  }
1562  }
1563  }
1564  }
1565  }
1566  }
1567  }
1568  }
1569  }
1570  }
1571  }
1572  }
1573  }
1574  }
1575  }
1576  }
1577  }
1578  }
1579  }
1580  }
1581  }
1582  }
1583  }
1584  }
1585  }
1586  }
1587  }
1588  }
1589  }
1590  }
1591  }
1592  }
1593  }
1594  }
1595  }
1596  }
1597  }
1598  }
1599  }
1600  }
1601  }
1602  }
1603  }
1604  }
1605  }
1606  }
1607  }
1608  }
1609  }
1610  }
1611  }
1612  }
1613  }
1614  }
1615  }
1616  }
1617  }
1618  }
1619  }
1620  }
1621  }
1622  }
1623  }
1624  }
1625  }
1626  }
1627  }
1628  }
1629  }
1630  }
1631  }
1632  }
1633  }
1634  }
1635  }
1636  }
1637  }
1638  }
1639  }
1640  }
1641  }
1642  }
1643  }
1644  }
1645  }
1646  }
1647  }
1648  }
1649  }
1650  }
1651  }
1652  }
1653  }
1654  }
1655  }
1656  }
1657  }
1658  }
1659  }
1660  }
1661  }
1662  }
1663  }
1664  }
1665  }
1666  }
1667  }
1668  }
1669  }
1670  }
1671  }
1672  }
1673  }
1674  }
1675  }
1676  }
1677  }
1678  }
1679  }
1680  }
1681  }
1682  }
1683  }
1684  }
1685  }
1686  }
1687  }
1688  }
1689  }
1690  }
1691  }
1692  }
1693  }
1694  }
1695  }
1696  }
1697  }
1698  }
1699  }
1700  }
1701  }
1702  }
1703  }
1704  }
1705  }
1706  }
1707  }
1708  }
1709  }
1710  }
1711  }
1712  }
1713  }
1714  }
1715  }
1716  }
1717  }
1718  }
1719  }
1720  }
1721  }
1722  }
1723  }
1724  }
1725  }
1726  }
1727  }
1728  }
1729  }
1730  }
1731  }
1732  }
1733  }
1734  }
1735  }
1736  }
1737  }
1738  }
1739  }
1740  }
1741  }
1742  }
1743  }
1744  }
1745  }
1746  }
1747  }
1748  }
1749  }
1750  }
1751  }
1752  }
1753  }
1754  }
1755  }
1756  }
1757  }
1758  }
1759  }
1760  }
1761  }
1762  }
1763  }
1764  }
1765  }
1766  }
1767  }
1768  }
1769  }
1770  }
1771  }
1772  }
1773  }
1774  }
1775  }
1776  }
1777  }
1778  }
1779  }
1780  }
1781  }
1782  }
1783  }
1784  }
1785  }
1786  }
1787  }
1788  }
1789  }
1790  }
1791  }
1792  }
1793  }
1794  }
1795  }
1796  }
1797  }
1798  }
1799  }
1800  }
1801  }
1802  }
1803  }
1804  }
1805  }
1806  }
1807  }
1808  }
1809  }
1810  }
1811  }
1812  }
1813  }
1814  }
1815  }
1816  }
1817  }
1818  }
1819  }
1820  }
1821  }
1822  }
1823  }
1824  }
1825  }
1826  }
1827  }
1828  }
1829  }
1830  }
1831  }
1832  }
1833  }
1834  }
1835  }
1836  }
1837  }
1838  }
1839  }
1840  }
1841  }
1842  }
1843  }
1844  }
1845  }
1846  }
1847  }
1848  }
1849  }
1850  }
1851  }
1852  }
1853  }
1854  }
1855  }
1856  }
1857  }
1858  }
1859  }
1860  }
1861  }
1862  }
1863  }
1864  }
1865  }
1866  }
1867  }
1868  }
1869  }
1870  }
1871  }
1872  }
1873  }
1874  }
1875  }
1876  }
1877  }
1878  }
1879  }
1880  }
1881  }
1882  }
1883  }
1884  }
1885  }
1886  }
1887  }
1888  }
1889  }
1890  }
1891  }
1892  }
1893  }
1894  }
1895  }
1896  }
1897  }
1898  }
1899  }
1900  }
1901  }
1902  }
1903  }
1904  }
1905  }
1906  }
1907  }
1908  }
1909  }
1910  }
1911  }
1912  }
1913  }
1914  }
1915  }
1916  }
1917  }
1918  }
1919  }
1920  }
1921  }
1922  }
1923  }
1924  }
1925  }
1926  }
1927  }
1928  }
1929  }
1930  }
1931  }
1932  }
1933  }
1934  }
1935  }
1936  }
1937  }
1938  }
1939  }
1940  }
1941  }
1942  }
1943  }
1944  }
1945  }
1946  }
1947  }
1948  }
1949  }
1950  }
1951  }
1952  }
1953  }
1954  }
1955  }
1956  }
1957  }
1958  }
1959  }
1960  }
1961  }
1962  }
1963  }
1964  }
1965  }
1966  }
1967  }
1968  }
1969  }
1970  }
1971  }
1972  }
1973  }
1974  }
1975  }
1976  }
1977  }
1978  }
1979  }
1980  }
1981  }
1982  }
1983  }
1984  }
1985  }
1986  }
1987  }
1988  }
1989  }
1990  }
1991  }
1992  }
1993  }
1994  }
1995  }
1996  }
1997  }
1998  }
1999  }
2000  }
2001  }
2002  }
2003  }
2004  }
2005  }
2006  }
2007  }
2008  }
2009  }
2010  }
2011  }
2012  }
2013  }
2014  }
2015  }
2016  }
2017  }
2018  }
2019  }
2020  }
2021  }
2022  }
2023  }
2024  }
2025  }
2026  }
2027  }
2028  }
2029  }
2030  }
2031  }
2032  }
2033  }
2034  }
2035  }
2036  }
2037  }
2038  }
2039  }
2040  }
2041  }
2042  }
2043  }
2044  }
2045  }
2046  }
2047  }
2048  }
2049  }
2050  }
2051  }
2052  }
2053  }
2054  }
2055  }
2056  }
2057  }
2058  }
2059  }
2060  }
2061  }
2062  }
2063  }
2064  }
2065  }
2066  }
2067  }
2068  }
2069  }
2070  }
2071  }
2072  }
2073  }
2074  }
2075  }
2076  }
2077  }
2078  }
2079  }
2080  }
2081  }
2082  }
2083  }
2084  }
2085  }
2086  }
2087  }
2088  }
2089  }
2090  }
2091  }
2092  }
2093  }
2094  }
2095  }
2096  }
2097  }
2098  }
2099  }
2100  }
2101  }
2102  }
2103  }
2104  }
2105  }
2106  }
2107  }
2108  }
2109  }
2110  }
2111  }
2112  }
2113  }
2114  }
2115  }
2116  }
2117  }
2118  }
2119  }
2120  }
2121  }
2122  }
2123  }
2124  }
2125  }
2126  }
2127  }
2128  }
2129  }
2130  }
2131  }
2132  }
2133  }
2134  }
2135  }
2136  }
2137  }
2138  }
2139  }
2140  }
2141  }
2142  }
2143  }
2144  }
2145  }
2146  }
2147  }
2148  }
2149  }
2150  }
2151  }
2152  }
2153  }
2154  }
2155  }
2156  }
2157  }
2158  }
2159  }
2160  }
2161  }
2162  }
2163  }
2164  }
2165  }
2166  }
2167  }
2168  }
2169  }
2170  }
2171  }
2172  }
2173  }
2174  }
2175  }
2176  }
2177  }
2178  }
2179  }
2180  }
2181  }
2182  }
2183  }
2184  }
2185  }
2186  }
2187  }
2188  }
2189  }
2190  }
2191  }
2192  }
2193  }
2194  }
2195  }
2196  }
2197  }
2198  }
2199  }
2200  }
2201  }
2202  }
2203  }
2204  }
2205  }
2206  }
2207  }
2208  }
2209  }
2210  }
2211  }
2212  }
2213  }
2214  }
2215  }
2216  }
2217  }
2218  }
2219  }
2220  }
2221  }
2222  }
2223  }
2224  }
2225  }
2226  }
2227  }
2228  }
2229  }
2230  }
2231  }
2232  }
2233  }
2234  }
2235  }
2236  }
2237  }
2238  }
2239  }
2240  }
2241  }
2242  }
2243  }
2244  }
2245  }
2246  }
2247  }
2248  }
2249  }
2250  }
2251  }
2252  }
2253  }
2254  }
2255  }
2256  }
2257  }
2258  }
2259  }
2260  }
2261  }
2262  }
2263  }
2264  }
2265  }
2266  }
2267  }
2268  }
2269  }
2270  }
2271  }
2272  }
2273  }
2274  }
2275  }
2276  }
2277  }
2278  }
2279  }
2280  }
2281  }
2282  }
2283  }
2284  }
2285  }
2286  }
2287  }
2288  }
2289  }
2290  }
2291  }
2292  }
2293  }
2294  }
2295  }
2296  }
2297  }
2298  }
2299  }
2300  }
2301  }
2302  }
2303  }
2304  }
2305  }
2306  }
2307  }
2308  }
2309  }
2310  }
2311  }
2312  }
2313  }
2314  }
2315  }
2316  }
2317  }
2318  }
2319  }
2320  }
2321  }
2322  }
2323  }
2324  }
2325  }
2326  }
2327  }
2328  }
2329  }
2330  }
2331  }
2332  }
2333  }
2334  }
2335  }
2336  }
2337  }
2338  }
2339  }
2340  }
2341  }
2342  }
2343  }
2344  }
2345  }
2346  }
2347  }
2348  }
2349  }
2350  }
2351  }
2352  }
2353  }
2354  }
2355  }
2356  }
2357  }
2358  }
2359  }
2360  }
2361  }
2362  }
2363  }
2364  }
2365  }
2366  }
2367  }
2368  }
2369  }
2370  }
2371  }
2372  }
2373  }
2374  }
2375  }
2376  }
2377  }
2378  }
2379  }
2380  }
2381  }
2382  }
2383  }
2384  }
2385  }
2386  }
2387  }
2388  }
2389  }
2390  }
2391  }
2392  }
2393  }
2394  }
2395  }
2396  }
2397  }
2398  }
2399  }
2400  }
2401  }
2402  }
2403  }
2404  }
2405  }
2406  }
2407  }
2408  }
2409  }
2410  }
2411  }
2412  }
2413  }
2414  }
2415  }
2416  }
2417  }
2418  }
2419  }
2420  }
2421  }
2422  }
2423  }
2424  }
2425  }
2426  }
2427  }
2428  }
2429  }
2430  }
2431  }
2432  }
2433  }
2434  }
2435  }
2436  }
2437  }
2438  }
2439  }
2440  }
2441  }
2442  }
2443  }
2444  }
2445  }
2446  }
2447  }
2448  }
2449  }
2450  }
2451  }
2452  }
2453  }
2454  }
2455  }
2456  }
2457  }
2458  }
2459  }
2460  }
2461  }
2462  }
2463  }
2464  }
2465  }
2466  }
2467  }
2468  }
2469  }
2470  }
2471  }
2472  }
2473  }
2474  }
2475  }
2476  }
2477  }
2478  }
2479  }
2480  }
2481  }
2482  }
2483  }
2484  }
2485  }
2486  }
2487  }
2488  }
2489  }
2490  }
2491  }
2492  }
2493  }
2494  }
2495  }
2496  }
2497  }
2498  }
2499  }
2500  }
2501  }
2502  }
2503  }
2504  }
2505  }
2506  }
2507  }
2508  }
2509  }
2510  }
2511  }
2512  }
2513  }
2514  }
2515  }
2516  }
2517  }
2518  }
2519  }
2520  }
2521  }
2522  }
2523  }
2524  }
2525  }
2526  }
2527  }
2528  }
2529  }
2530  }
2531  }
2532  }
2533  }
2534  }
2535  }
2536  }
2537  }
2538  }
2539  }
2540  }
2541  }
2542  }
2543  }
2544  }
2545  }
2546  }
2547  }
2548  }
2549  }
2550  }
2551  }
2552  }
2553  }
2554  }
2555  }
2556  }
2557  }
2558  }
2
```

控制台输出如下：

连接结果写在磁盘块 700~731.blk 中

```
-----  
基于排序的集合差操作算法  
-----
```

```
注:结果写入磁盘:700  
注:结果写入磁盘:701  
注:结果写入磁盘:702  
注:结果写入磁盘:703  
注:结果写入磁盘:704  
注:结果写入磁盘:705  
注:结果写入磁盘:706  
注:结果写入磁盘:707  
注:结果写入磁盘:708  
注:结果写入磁盘:709  
注:结果写入磁盘:710  
注:结果写入磁盘:711  
注:结果写入磁盘:712  
注:结果写入磁盘:713  
注:结果写入磁盘:714  
注:结果写入磁盘:715  
注:结果写入磁盘:716  
注:结果写入磁盘:717  
注:结果写入磁盘:718  
注:结果写入磁盘:719  
注:结果写入磁盘:720  
注:结果写入磁盘:721  
注:结果写入磁盘:722  
注:结果写入磁盘:723  
注:结果写入磁盘:724  
注:结果写入磁盘:725  
注:结果写入磁盘:726  
注:结果写入磁盘:727  
注:结果写入磁盘:728  
注:结果写入磁盘:729  
注:结果写入磁盘:731  
R和S的差集中共有213个元组  
IO操作共135次
```