中国科学院大学
University of Chinese Academy of Sciences

2016-2017 春季学期 数据结构课程

实验报告

多关键字排序

1511　李　勰　2015K80099*****
1511　徐易难　2015K80099*****
1511　张远航　2015K80099*****

2016~2017 学年第二学期

2017 年 7 月 5 日

目录

人员分工

李勰：编写稀疏矩阵运算库中的十字链表和 CSR 压缩存储及相关算术操作，实现命令行界面；撰写报告；

张远航：编写稀疏矩阵运算库中的向量及三元组存储，辅助李勰实现代数余子式；用 LaTeX 完成实验报告和 PPT 的撰写；课堂汇报；

徐易难：完成多关键字排序的相关代码并实现 Python 可视化，观察关键字和记录个数增加时的现象并分析其成因；撰写报告。

实验二　多关键字排序

1　需求分析

1. 按照用户给定的排序优先顺序，对多关键字的数据进行排序，并输出排序结果。
2. 待排序的记录数和关键字数均由用户给定，各关键字的范围为 1 至 999。
3. 在对各个关键字进行排序时采用两种策略：其一是利用稳定的内部排序法，其二是利用“分配”和“收集”的方法，并综合比较这两种策略。
4. 分别按照 LSD 和 MSD 策略进行排序，并进行比较。
5. 分析、比较排序时间与各影响因素的关系。

2　概要设计

2.1　单条记录的抽象数据类型定义

```
ADT MyRecord {
    数据对象（私有）:
        std::vector<KeyType> keys 存放各关键字的内容
        int next 存放下一条记录在数据库中的位置
    基本操作:
        MyRecord()
        操作结果：构造器，构造一个空的记录，其下一条记录为宏定义EMPTY_NEXT
        MyRecord(const std::vector<KeyType> &data)
        操作结果：构造器，构造一个内容为data的记录，其下一条记录为宏定义
            EMPTY_NEXT
        MyRecord(const std::vector<KeyType> &data, int nextVal)
        操作结果：构造器，构造一个内容为data的记录，其下一条记录为nextVal
        std::vector<KeyType> getData() const
        操作结果：访问器，返回该记录中的数据
        KeyType getData(int n) const
        操作结果：访问器，返回该记录中第n个数据
        int getNext() const
        操作结果：访问器，返回该记录的下一条记录所在位置
```

MyRecord &setNext(unsigned pos)
操作结果：更改器，将该记录的下一条记录所在位置设为pos
MyRecord &setData(unsigned n, KeyType d)
操作结果：更改器，将该记录第n个数据设为d
const MyRecord &print() const;
操作结果：访问器，将该记录打印输出
MyRecord &print();
操作结果：访问器，将该记录打印输出
}ADT MyRecord

2.2　数据库的抽象数据类型定义

ADT MyDatabase{
数据对象（私有）：
std::vector<MyRecord> records 数据库中的所有记录
unsigned keynum 关键字数量
unsigned recnum 记录数
int head 第一个记录所在位置
int tail 最后一个记录所在位置
基本操作：
MyDatabase()
操作结果：构造器，构造一个空的数据库
MyDatabase(int)
操作结果：构造器，构造一个含有对应关键字数量的数据库
int getKeyNum() const
操作结果：访问器，返回数据库的关键字数量
int getRecNum() const
操作结果：访问器，返回数据库中记录的数量
std::vector<MyRecord> getData() const
操作结果：访问器，用std::vector返回数据库中所有的记录
MyRecord getData(int) const
操作结果：访问器，返回数据库中对应物理位置的记录
int getHead() const
操作结果：访问器，返回数据库第一个记录的位置
int getTail() const
操作结果：访问器，返回数据库最后一个记录的位置
int getMid() const;
操作结果：访问器，返回数据库中最中间记录的位置
int getMid(int, int) const;
操作结果：访问器，返回数据库中两条记录的中间记录的位置
bool empty() const

操作结果：访问器，如果数据库为空，则返回true，否则为false

MyDatabase &add(const std::vector<KeyType> &);

操作结果：更改器，向数据库中添加一条记录

MyDatabase &setKeyNum(int);

操作结果：更改器，将数据库的关键字数量设置对应数字

const MyDatabase &print() const;

操作结果：访问器，将该数据库打印输出

MyDatabase &print();

操作结果：访问器，将该数据库打印输出

MyDatabase &mergeSort_LSD();

操作结果：更改器，对数据库按照默认关键字优先级进行LSD归并排序

MyDatabase &mergeSort_MSD();

操作结果：更改器，对数据库按照默认关键字优先级进行MSD归并排序

MyDatabase &mergeSort(unsigned);

操作结果：更改器，对数据库按照对应关键字进行归并排序

MyDatabase &mergeSort_LSD(const std::vector<unsigned> &);

操作结果：更改器，对数据库按照给定的关键字优先级进行LSD归并排序

MyDatabase &mergeSort_MSD(const std::vector<unsigned> &);

操作结果：更改器，对数据库按照给定的关键字优先级进行MSD归并排序

MyDatabase &radixSort_LSD();

操作结果：更改器，对数据库按照默认关键字优先级进行LSD基数排序

MyDatabase &radixSort_MSD();

操作结果：更改器，对数据库按照默认关键字优先级进行MSD基数排序

MyDatabase &radixSort(unsigned);

操作结果：更改器，对数据库按照对应关键字进行基数排序

MyDatabase &radixSort_LSD(const std::vector<unsigned> &);

操作结果：更改器，对数据库按照给定的关键字优先级进行LSD基数排序

MyDatabase &radixSort_MSD(const std::vector<unsigned> &);

操作结果：更改器，对数据库按照给定的关键字优先级进行LSD基数排序

std::vector<int> getOrder() const;

操作结果：访问器，返回数据库中记录的次序

MyDatabase &setOrder(const std::vector<int> &);

操作结果：更改器，设置数据库中记录的次序

MyDatabase &resetOrder();

操作结果：更改器，重置数据库中记录的次序为默认次序，即按照物理地址排序

} ADT MyDataBase

3　详细设计

3.1　模块设计

本程序包含四个模块，分别为随机数生成器、单条记录的实现模块、数据库实现模块以及测试用主程序。主程序根据不同的输入建立对应数据库，并向其中添加利用随机数生成器或输入的数据产生的记录。

3.2　LSD 和 MSD 策略的实现

LSD 排序根据优先级从低到高依次调用对应关键字的排序函数，全部调用完即可。

MSD 排序优先依据高关键字进行排序，在高一级值相等时，递归在对应范围内进行低一级关键字进行排序。此处针对我们使用的数据结构进行了特别优化，每次排序时并不是在原数据库上直接进行的，而是对原数据的次序进行移动，结束后，将新得到的次序赋值给原数据库即可。

3.3　基数排序和归并排序的实现

基数排序对原题做了进一步的处理，将单个数据继续分为三位进行 LSD 策略的排序，因此基数取的是 10。通过分配、收集，每次可处理一位，三次后完成一个关键字的排序。归并排序递归进行，对于两部分排序好的子序列，遍历使得两部分从小到大排列有序即可。

3.4　演示用程序

方便起见，利用 Python 实现了简单的交互界面来与项目交互。用 Python 调用本程序，并采集输出，进一步处理得到相关数据。可以导出为 `csv` 格式（逗号分隔文件）并对他们进行比较，还可以利用 `matplotlib` 库得到排序时间与各参数的关系。

3.5　核心代码段展示

归并排序

```
MyDatabase &MyDatabase::mergeSort(unsigned key, int low, int high, std::vector<int> &order)
// merge sort in order[low], order[high]
{
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(key, low, mid, order);
        mergeSort(key, mid + 1, high, order);
        int i = low, j = mid + 1, len = high - low + 1;
        std::vector<int> sub;
        while (i <= mid || j <= high){
            if (i <= mid && j <= high) {
                if (getData(order[i], key) <= getData(order[j], key))
                    sub.push_back(order[i++]);
                else
                    sub.push_back(order[j++]);
            }
            else if (i <= mid)
```

```
18              sub.push_back(order[i++]);
19          else
20              sub.push_back(order[j++]);
21      }
22      for (i = 0; i != len; i++)
23          order[low + i] = sub[i];
24   }
25   return *this;
26 }
```

基数排序

```
1  MyDatabase &MyDatabase::radixSort(unsigned key, int low, int high, std::vector<int> &order)
2  // radix sort in range(order[low], order[high])
3  // sort by changing vector order
4  // a helper function
5  {
6      int numLength = getLength(MAX_DATA_NUM);
7      for (int pos = 0; pos != numLength; pos++) {
8          // distribution part
9          std::vector<std::vector<int>> collect(NUM_BASE);
10         for (int i = low; i <= high; i++){
11             auto rec = order[i];
12             collect[getDigit(getData(rec, key), pos)].push_back(rec);
13         }
14         // collection part
15         unsigned i = low;
16         for (auto col: collect)
17             for (auto pos: col)
18                 order[i++] = pos;
19     }
20     return *this;
21 }
```

MSD 策略

```
1  MyDatabase &MyDatabase::sort_MSD(int type, const std::vector<unsigned> &priority)
2  // type == 0: merge sort
3  // type == 1: radix sort
4  {
5      if (keynum != 0){
6          auto order = getOrder();
7          std::stack<int> from, to;
8          from.push(0);
9          to.push(recnum - 1);
10         for (unsigned i = 0; (!from.empty()) && i != keynum; i++){
11             while (!from.empty()){
12                 if (type == 0)
13                     mergeSort(priority[i], from.top(), to.top(), order);
14                 else if (type == 1)
15                     radixSort(priority[i], from.top(), to.top(), order);
16                 from.pop();
17                 to.pop();
18             }
19             // update range of recursion
```

```
20          unsigned low = 0;
21          for (unsigned j = 0; j != recnum; j++){
22              if (j != recnum - 1){
23                  bool notEqualNow = false, notEqualBefore = false;
24                  notEqualNow = (getData(order[j], priority[i]) != getData(order[j + 1], priority[i]));
25                  if (i != 0)
26                      notEqualBefore = (getData(order[j], priority[i - 1]) != getData(order[j + 1], priority
                            [i - 1]));
27                  if (notEqualNow || notEqualBefore){
28                      if (j != low){
29                          from.push(low);
30                          to.push(j);
31                      }
32                      low = j + 1;
33                  }
34              }
35              else if (j == recnum - 1 && j != low){
36                  from.push(low);
37                  to.push(j);
38              }
39          }
40      }
41      setOrder(order);
42  }
43  return *this;
44 }
```

4　调试分析

4.1　程序特点

本项目的多关键字排序实现比较简单，思路清晰，具体实现的重点主要是对相关类的封装上。对于排序来说，相关的实现较为传统，但仍然高效，其中一个原因是对于特定数据结构的特殊优化。归并排序需要大量的移动操作，对应于所采用数据结构中的下一记录需要频繁变动，因此在实现排序时，并未对原始数据库进行操作，而是对其得到的记录次序进行操作，仅仅在完成时，对数据库进行更改，这样大量减少了访存次数。

项目的最大优点在于对类的封装，在基本的数据结构的基础上，对其进行了进一步的抽象，最大限度地保证了数据的安全，同时，也有利于数据结构的进一步优化。然而，尽管本项目的实现重点在数据库类的封装，但还未实现更多高级的功能，如查找。然而，基于所用存储结构的线性关系，查找将会是比较高效的。同时，基于数据库所提供的接口，也能够很方便地设计出更多的可用函数。

基于 Python 的可扩展性，本项目利用 `matplotlib` 库实现了对排序时间的可视化分析，可以方便地得到不同策略、排序方式、关键字数量、记录数下的排序时间，并得到曲线图，这给研究不同因素对排序时间的影响提供了极大的便利。

4.2　实验发现

基数排序中，采用 LSD 策略所需的排序时间随着关键字数量的增大而线性增加，但采用 MSD 策略时排序时间则会在达到一定峰值后，转而平稳甚至开始下降。因此，在关键字较少

的情况下，采用 LSD 策略能够有效地减少排序时间，而当关键字数量变大时，MSD 策略则更优。考虑其原因，关键字数量较少时，MSD 策略的递归消耗了大量的时间，而 LSD 则流程非常规律，因此此时 MSD 慢于 LSD；当关键字数量增加，LSD 策略由于需要对每一个关键字都进行一次排序，因此消耗时间迅速上升，MSD 策略由于递归层次较少，速度较快。

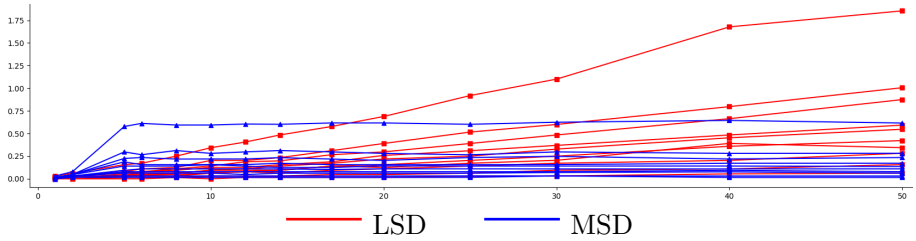


图 2.1: 基数排序中，关键字数量对消耗时间的影响

与之对比，在归并排序中，LSD 策略与 MSD 策略下的消耗时间变化情况类似，但 MSD 始终比 LSD 快，这是由归并所需要的递归所造成的，MSD 大大地减少了递归的消耗。

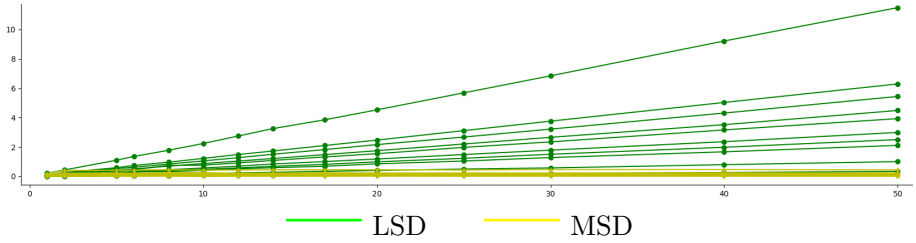


图 2.2: 归并排序中，关键字数量对消耗时间的影响

如前所述，关键字数量对消耗时间的影响在 LSD、MSD 策略下各不相同，MSD 下随着关键字数量的增长，消耗时间几乎不变，而 LSD 下则会线性增加。

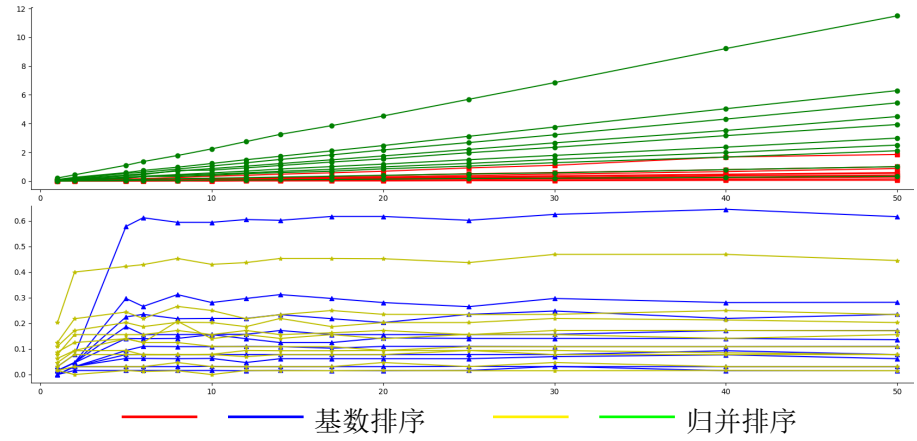


图 2.3: 关键字数量对消耗时间的影响（上图为 LSD）

记录数对消耗时间的影响较为简单，随着记录数增加，消耗时间亦线性增加。值得一提的是，当采用 MSD 策略时，在记录数相对较少的情况下，基数排序速度快于归并排序，而当记录数增加，基数排序的消耗时间会大于归并排序。这个现象的产生，需要从基数排序的具体实现上考虑，分配、收集方法需要更复杂的数据结构以及操作来来支撑其运作，而归并排序相比之下则简单不少。

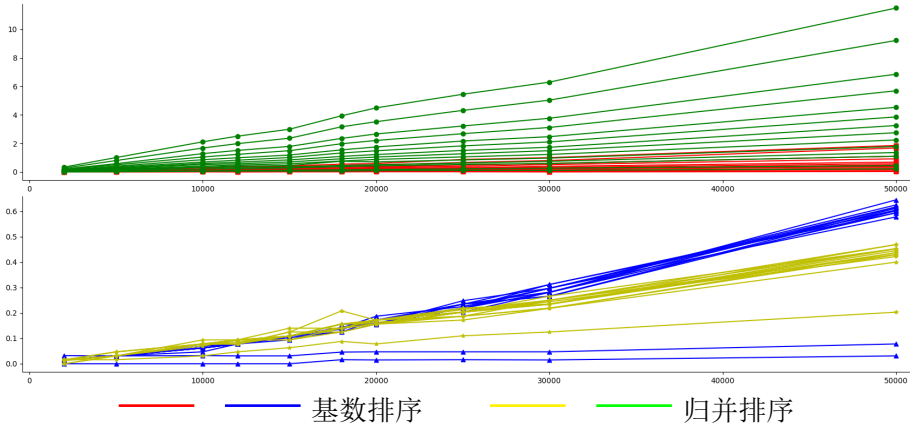


图 2.4: 记录数对消耗时间的影响（上图为 LSD）

5　用户手册

1. 本程序运行环境为 Windows 10，可执行文件为 `MultiKeySort.exe` 。C++ 代码采用 C++11 标准，开发所用 Python 版本为 Python 3.6.0。

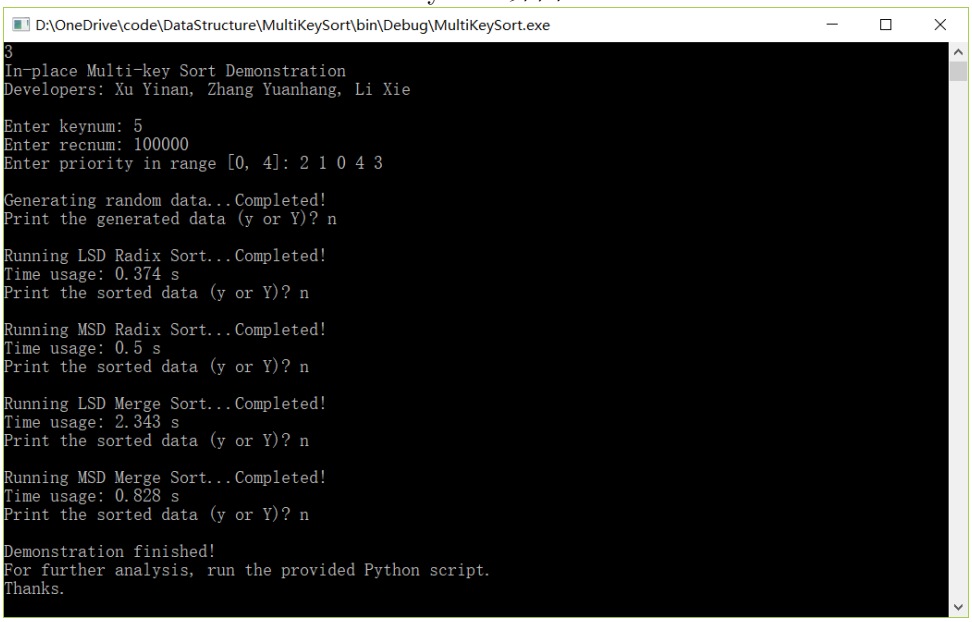
2. 如需用 `MultiKeySort.exe` 进行演示，打开后输入 3 并回车，之后依次输入关键字数量、记录数及排序优先级，程序将会提示是否打印各次排序结果，输入 y 或者 Y 打印。该程序打开后输入 $0, 1, 2$ 将会进入数据分析的模式，将不会打印除有效信息外的任何数据，仅供 Python 脚本调用时使用。

3. 如用 Python 进行演示，双击 `run.bat` 将自动运行脚本，开启后点击各按键即可实现相应功能。

6　测试结果

实验结果分析见 4.2 节。

Python 界面



命令行版本