数据库开发说明和测试报告

殷凡

515030910040

2017.7.28

1. 开发说明

1.1总体说明

本次项目以c++语言编写简易数据库，数据库为<key:value>的简单形式，在本项目中，限定key为整数且不考虑溢出问题，value为字符串类型，不可为空，长度最长为19（其中第20位为\0字符）。主体程序面向用户提供四种主要操作，分别为查找、添加、删除和修改。文件中数据结构主要采用B+树，实现了对删除的结点的空间回收。数据库cache模拟系统中的cache以利用文件读取的局部性来增加读写速度。文件以二进制形式打开以便于管理。

1.2文件设计说明

1.2.1索引文件设计说明

索引文件前4个字节为根节点所在地址，若为0则树为空，初始时。接着8个字节为第一个空白位置，初始时为8，即文件尾。然后依次是每个节点。每个节点分为三个部分，第一部分为12个字节，四个整数，分别是父节点地址、父节点在节点中的位置（从1开始）和当前节点关键码的个数，根节点父节点地址为0。第二部分为当前节点的关键码和其孩子的地址，若节点为叶节点，则为当前节点的关键码和关键码对应的值在数据文件中的地址的负数（因此可以根据孩子地址的正负来直接区别内部节点和叶子结点）。第三部分为下一个叶子节点的地址，若节点为内部节点，则该部分无意义。空白位置组成单项链表，最后一项始终为文件末尾。删除节点后将地址链接到链表头部。

1.2.2数据文件设计说明

数据文件前四个字节为第一个空白位置，初始时为4。之后为数据，每条数据占用20个字节。空白位置组成单项链表，最后一项始终为文件末尾。删除节点后将地址链接到链表头部。

1.2.3日志文件设计说明

用户版和正确性测试版中每次操作会将相关信息写入日志文件，主要用于程序调试。

1.2.5正确性测试文件及性能测试文件设计说明

文件第一个数字为需要进行的操作，数字1、2、3的含义与用户版对应，其后是需要操作的key，根据操作种类确定后面是否有value。正确性测试文件中4的含义为测试所有key。性能测试文件名最后数字问测试循环单位，而非测试数据量。

1.3类设计说明

1.3.1Cache类（Cache.h）

1.3.1.1总体说明

cache类模拟系统中的cache，将最近使用过的数据放入其中，利用局部性加快数据的读写速度。cache大小固定，为16384，即2^14。cache中每个数据有四个值：int key、string value、bool dirty、bool valid。其中valid表示这个数据是否有效，dirty表示这个key对应的值是否被修改过。

1.3.1.2主要函数

1. Cache()

构造函数，确定cache大小，将所有有效位置为0。

1. bool select(const int key, string &value)

从cache中取值，将取回的值放在value中，返回值表示是否取值成功。

1. int insert(const int key, const string &value,int \*oldKey,string \*oldValue)

向cache中插入值，若cache中原来有值且被修改过，则将原来的key和value存入\*oldKey和\*oldValue。返回值含义为0表示插入失败，1表示原来没有值或值没有被修改，2表示原来有值且被修改过。

1. void remove(const int key)

从cache中删除，若cache中存有key且有效位为true，则将有效位置为false。

1. bool update(const int key, string &value)

更新cache中的值，若cache中没有key，返回false，否则更新值且将dirty置为true。

1. void save(vector<int> \*key, vector<string> \*value)

将cache中的内容写会，所有没有保存修改的key和value存入\*key和\*value，交由索引文件和数据文件保存。

1.3.2Database类

1.3.2.1总体说明

Database类为数据库的主体实现。持有变量scale为B+树的阶数。三个文件流分别对应索引文件、数据文件和日志文件，常量ZERO用于写入空指针。

1.3.2.2主要流程（对应用户版）

1.3.2.2.1查找

（1）用户输入关键码

（2）查找cache，若找到，返回对应的值

（3）否则查找索引文件，若没有找到，返回false

（4）否则根据地址查找数据文件，取出相应的值

（5）将关键码和值插入cache中

（6）若有必要，更新cache中原来的关键码和值

（7）根据返回值输出结果

1.3.2.2.2添加

1. 用户输入关键码和值
2. 若cache中存在关键码，返回false
3. 否则查找索引文件，若存在关键码，返回false
4. 否则在数据文件中插入值并获得位置
5. 根据查找结果在索引文件中加入关键码和数据地址
6. 将关键码和值插入cache中
7. 若有必要，更新cache中原来的关键码和值
8. 根据返回值输出结果

1.3.2.2.3删除

1. 用户输入关键码
2. 在cache中删除此关键码
3. 查找索引文件，若关键码不存在，返回false
4. 否则根据查找结果在数据文件中删除值
5. 根据查找结果在索引文件中删除关键码
6. 根据返回值输出结果

1.3.2.2.4修改

1. 用户输入关键码和值
2. 在cache中更新关键码和值，若成功，返回true
3. 否则在索引文件中查找关键码，若没找到，返回false
4. 否则根据查找结果在数据文件中更新值
5. 将关键码和值插入cache中
6. 若有必要，更新cache中原来的关键码和值
7. 根据返回值输出结果

1.3.2.3主要函数

1. Database();构造函数，文件不存在初始化文件
2. void dataFile\_find(const int dataAddress, string &value);数据文件查找
3. int dataFile\_add(const string &value);数据文件添加，返回数据地址
4. void dataFile\_delete(const int dataAddress);数据文件删除
5. void dataFile\_replace(const int dataAddress, string &value);数据文件替换
6. int indexFile\_find(int key, int \*indexAddress, int \*pos, int \*size, int \*dataAddress);

索引文件查找，\*size为最后一个节点关键码个数。返回值含义为0表示树为空或key小于最小值，1表示命中，2表示在两节点之间，4表示大于最后一个节点最大值。若命中，\*indexAddress存放命中叶节点地址，\*pos为节点位置，\*dataAddress为值在数据文件中的地址。若不命中，\*indexAddress存放应该插入叶节点地址，\*pos为插入位置（1表示在第一个关键码之后），\*dataAddress无意义。对每个节点采用二分查找法。

1. void indexFile\_add(const int key, const int dataAddress, const int indexAddress, int pos, int size);索引文件添加
2. void indexFile\_addAndOverflow(const int key, const int dataAddress, const int indexAddress, const int pos, const int size);索引文件添加并处理上溢
3. bool indexFile\_delete(const int indexAddress, const int pos, int size);索引文件删除
4. void indexFile\_borrowLeft(const int indexAddress, int size, const int left, int leftSize, int parent, int parentPosition);索引文件当前节点向左兄弟借一个关键码
5. void indexFile\_borrowRight(const int indexAddress, int size, const int right, int rightSize, int parent, int parentPosition);索引文件当前节点向右兄弟借一个关键码
6. void indexFile\_mergeLeft(const int indexAddress, const int size, const int left, int leftSize);索引文件当前节点和左兄弟合并
7. void indexFile\_mergeRight(const int indexAddress, int size, const int right, const int rightSize);索引文件当前节点和右兄弟合并
8. void indexFile\_deleteAndUnderflow(const int indexAddress, const int pos, int size);索引文件删除并处理下溢
9. bool file\_update(const int key, string &value);更新到文件
10. bool select(const int key, string &value);对外接口查找
11. bool insert(const int key, const string &value);对外接口插入
12. bool remove(const int key);对外接口删除
13. bool update(const int key, string &value);对外接口更新
14. ~Database();析构函数，保存cache，关闭文件

1.4版本设计说明

1.4.1用户版

用户版无法测试，故没有数据结构维护正确答案，每次打开数据库若索引文件或数据文件不存在则新建文件并初始化，用户操作的相关信息会写入日志文件。

1.4.2正确性测试版

默认情况下每次清空索引文件和数据文件，从测试文件中读取测试数据进行测试。若遇到错误则停止，否则输出通过并进入手动测试界面。可以通过更改文件打开方式和使用注释中的代码进行数据库关闭后再打开的测试（主要用于测试cache）。在简易测试模式下需手动注释cache的引用进行测试。（否则数据均在cache中，难以找出错误）简易测试为针对B+树规模为5时构造层数为3的B+树，之后人为计算各种增删改查的特殊情况进行测试。随机测试模式随机生成测试数据和增删改查操作进行测试。

1.4.3性能测试版

默认情况下每次清空索引文件和数据文件，从测试文件中读取数据测试数据进行测试。测试完成后显示测试所用时间（单位为毫秒），用户输入0退出。

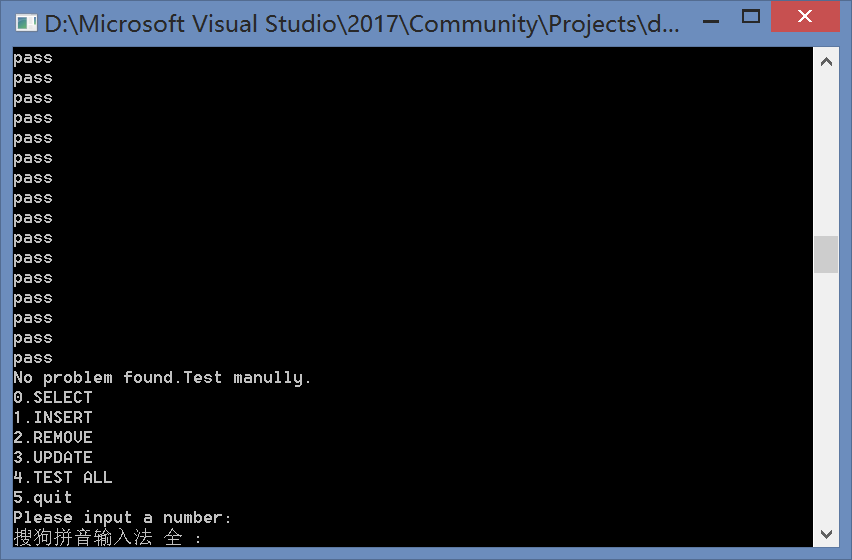
1.5其他说明

辅助项目auxiliary目录下CreateTestFile项目用于生成正确性测试和性能测试所需要的文件。

1. 测试报告

2.1正确性测试

分别运行简易测试和随机测试文件，测试结果均没有异常。



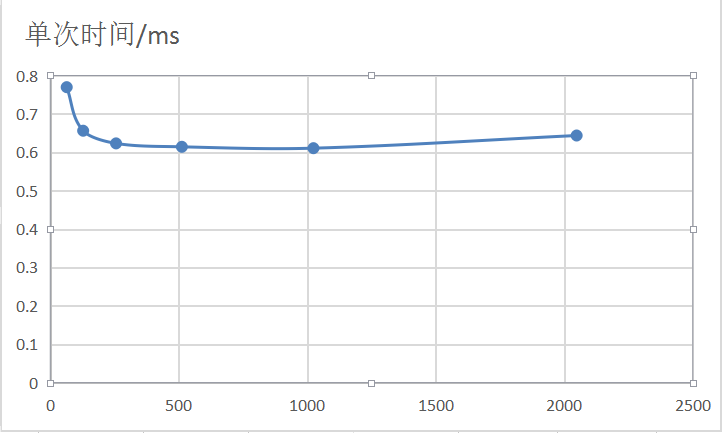
2.2性能测试

2.2.1测试结果

当B+树的规模不同，数据量为1000000，插入次数为10000、循环测试次数为10000、删除次数为1000（删除后会随机读取10条记录）时，平均每次操作所用时长如下（总操作次数1033088）

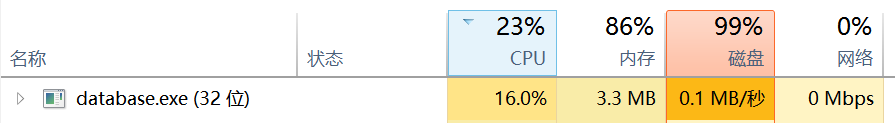
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 规模 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 操作总时间/ms | 795088 | 677865 | 643788 | 634600 | 630898 | 664984 |
| 单次操作时间/ms | 0.76962 | 0.65615 | 0.62317 | 0.61427 | 0.61069 | 0.64369 |

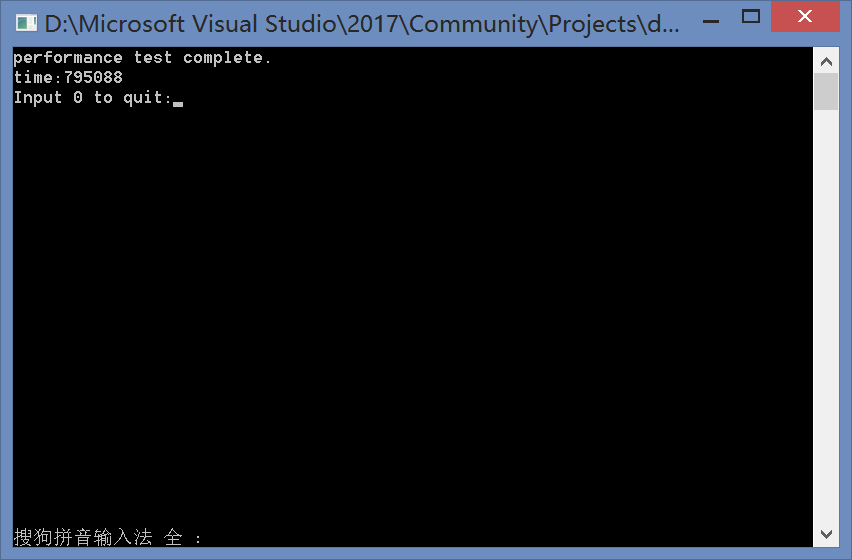
数据折线图如下

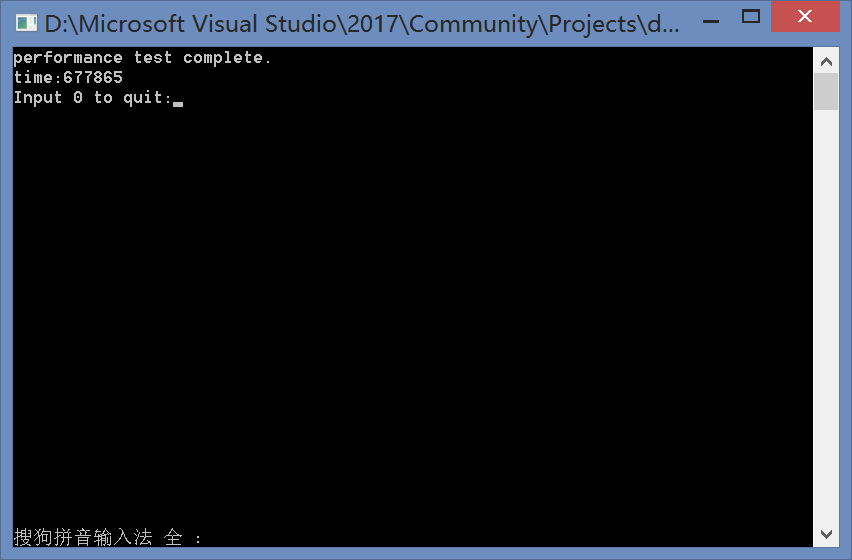


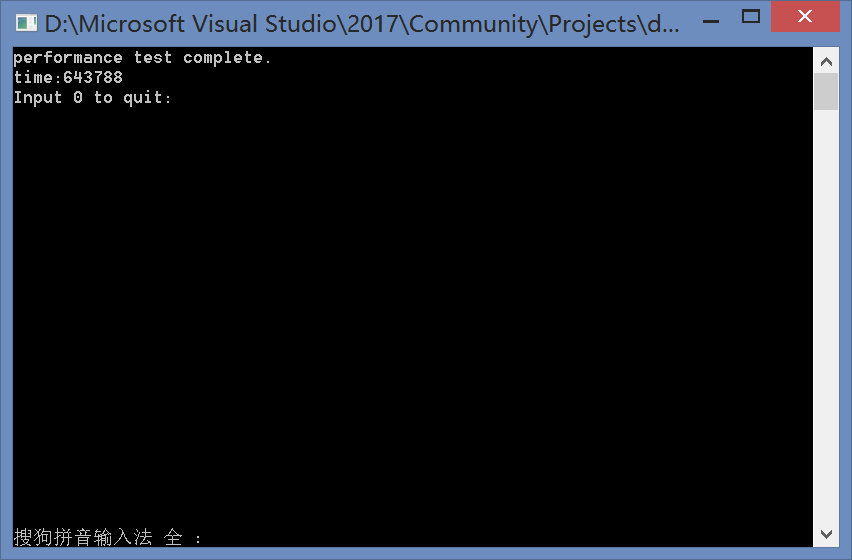
在测试中，程序所占CPU为15%左右，所占内存为2-3M。由此推测程序没有进入死循环且没有内存泄漏。

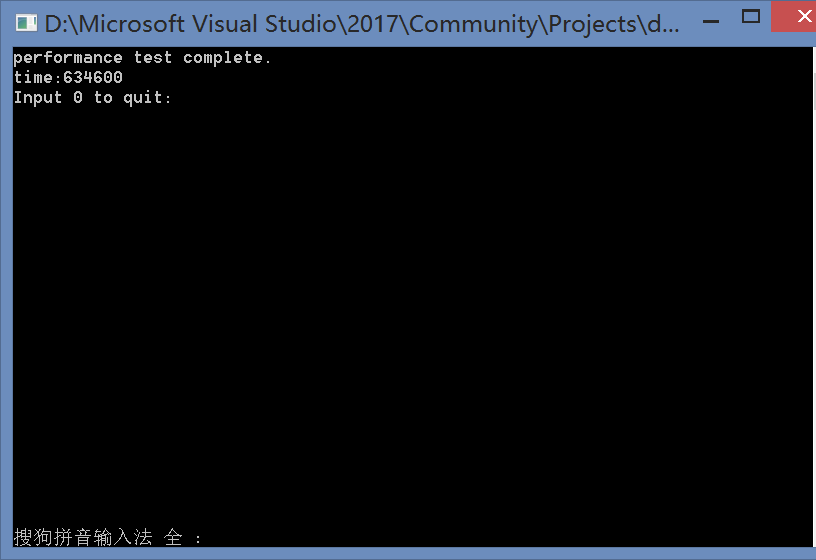
测试过程截图如下

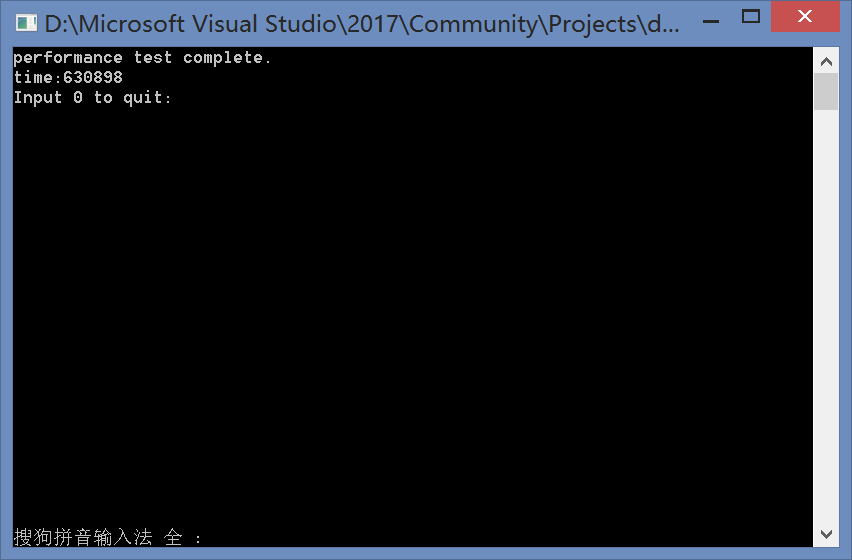


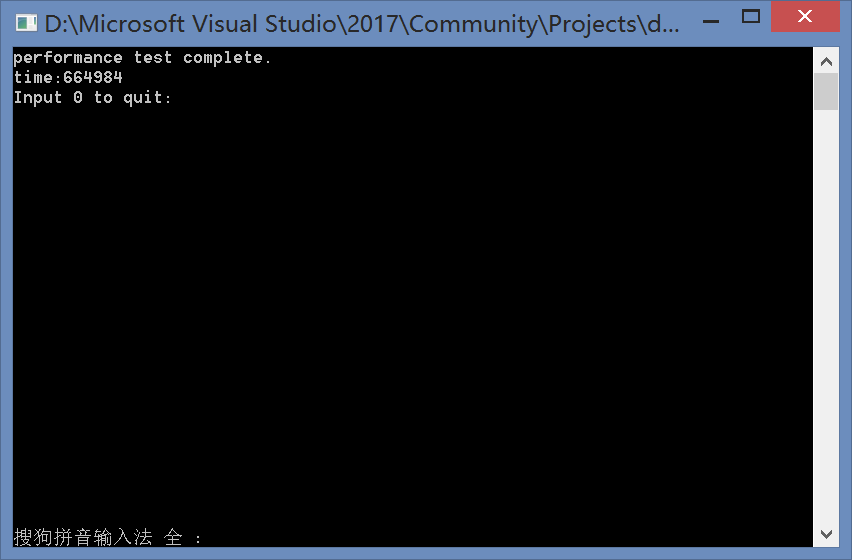












2.2.2测试分析和猜想

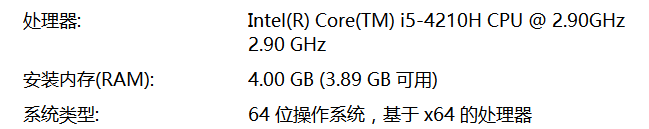
根据B+树性质，对于阶数m数据量为N的单次操作时间复杂度为logmN。在本次测试中N几乎不变，故随着m增大，单次操作所需时间减少，这在m不很大的情况下有所体现。说明B+树对查找效率有更大的帮助，当m过大时，单次查找所需时间增加，猜想这是因为结点大小超过内存访问时单个物理页的面积，从而破坏整体读写，使B+树的优点无法充分体现。这说明B+树设计的合理性，也体现了本次设计达到了预期要求。

1. 其他说明

3.1开发环境

代码编写环境：Microsoft Visual Studio 2017 community

测试环境：windows 8.1



3.2其他

本次模拟数据库的设计加强了我对数据结构的理解和应用，但由于知识有限，难免有许多设计不合理的地方，望各位老师见谅并不吝赐教，我也将进一步加强相关知识的学习。