



Database Management System(DBMS)

设计报告

Meng Yit Koh | SE102 | 517030990022

目录

目录 1

一、 基本信息.....**Error! Bookmark not defined.**2

二、 提供接口..... 3

三、 模块实现..... 5

四、 功能实现..... 6

五、 特点..... 9

 1、 不定长 value..... 9

 2、 空间的重新使用 9

 3、 缓存设计 9

 4、 B+树链表范围查找 10

六、 测试及分析..... 11

 1、 正确性测试**Error! Bookmark not defined.**

 1.1 大量数据测试.....

 1.2 非寻常测试.....**Error! Bookmark not defined.**

 1.3 综合测试.....**Error! Bookmark not defined.**

 2、 性能测试**Error! Bookmark not defined.**

 2.1 B+树的时间复杂度**Error! Bookmark not defined.**

 2.2 关于存储缓存性能优化的测试**Error! Bookmark not defined.**

 2.3 关于空间的重新利用**Error! Bookmark not defined.**

七、 致谢以及参考.....**Error! Bookmark not defined.**19

一 基本信息

项目名称 Project Name: Database Management System (DBMS)

主要数据结构 Data Structure: B+ tree

开发环境 Development Platform: Windows 10 64-bit

Intel i5 4th-gen

8gb RAM

概述 Description:

结合了增删改查功能的键值数据库

<key, value> Database querying system with ability to perform insertion, deletion, modification and querying.

二 提供接口

我已经特别写了一个类来表示我的常用接口，名为 DBMS.h。里面有详细的 API 功能及介绍。如有不明白可以阅读。

我提供了<int, string>这种类型的<key,value>存储，其最终 string 的长度可以不定。

我简单地说一下我的接口（API）：

```
void openDB(int method); // do required initialization to open index and data files
```

```
void closeDB(); // do required ending procedures
```

```
bool insertDB(int key, std::string value); // insert key-value data into this database
```

```
bool changeDB(int key, std::string newValue); // change value of a value based on key  
given
```

```
bool deleteDB(int key); // delete unwanted data
```

```
std::string getDB(int key); // search for data
```

```
std::vector<int> getRangeDB(int key1, int key2); // search for range of keys
```

中文注释：

```
void openDB(int method);
```

- method 是开文件用途。由于 C++默认不同开文件读取方式，我启用了两种开文件方式。
- 0 为创建文件，1 为读写文件

```
void closeDB();
```

- 关闭 index 和 value 文件

`bool insertDB(int key, std::string value);`

- 存入 key-value。
- 如果成功就返回 `true`。反之 `false`。

`bool changeDB(int key, std::string newValue);`

- 根据 key 修改 value。
- 如果成功返回 `true`。反之 `false`。

`bool deleteDB(int key);`

- 根据 key 删除数据。
- 成功返回 `true`。反之 `false`。

`std::string getDB(int key);`

- 根据 key 得 value。
- 返回数据。

`std::vector<int> getRangeDB(int key1, int key2);`

- 根据两个 key 得到他们之间存在的 key。
- 返回各个 key 的 vector 列表。

三 模块实现

由于实现数据库的代码量很多，为了让我不要乱，我实现并使用了很多模块很多类，以让我更容易控制程序并在出问题方便找到问题所在。

以下是我的各个模块及其简单介绍：

main.cpp

- Main connection and interface for Database.

DBMS.h

- Some useful and often used API for Database.
- Used to connect difficultly understandable classes in other modules.
- Main entrance to Index and Data query.
- Implementation in DBMS.cpp

Index.h

- To store index and enable multi-level indexing based on BPNode.
- Adapt the idea of B+ tree.
- Backbone for pointer search to get value's offset.
- Implementation in Index.cpp

DataFile.h

- To store values and give position of disk's offset to Index to save location.
- Gather extra unwanted spaces for use by extra data.
- Implementation in DataFile.cpp

Data.h

- Small structure to store offset and length of values for storage later.
- Helps DataFile to gather rubbish spaces for later use.
- Implementation in Data.cpp

四 功能实现

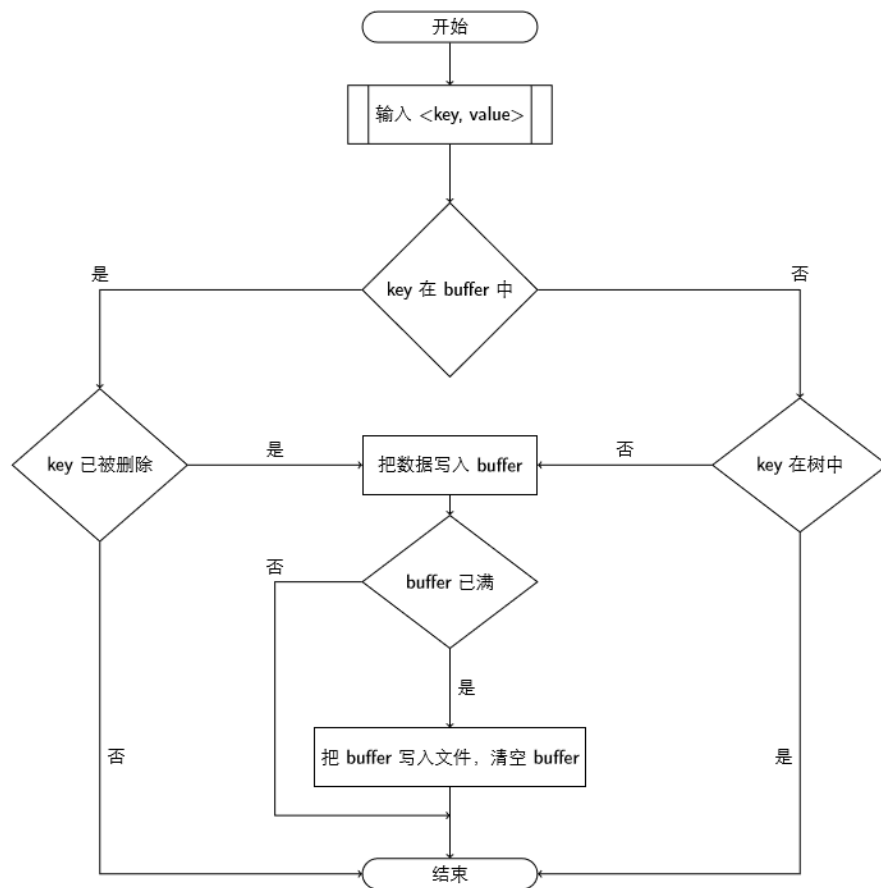
1. 打开（open）

- 根据我做的界面，用户能很清楚地知道要如何开启数据库。我提供两种方案：
一，重新制造数据文件。
二，使用原有的数据文件。
- 初始化数据读写和需要的缓存。

2. 关闭（close）

- 强制进行一致性刷新，关闭索引以及数据文件。

3. 插入（store）



4. 删除 (delete)

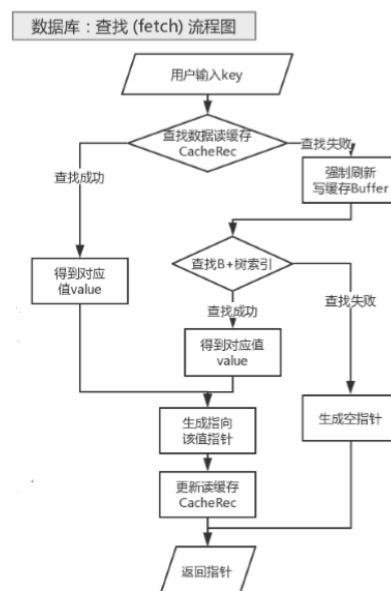
- 程序会对文件进行树的从根起的解析。为了保证性能，只有被碰到的树节点才会被解析成树节点。
- 找到叶节点及对应的 key 后就解析起信息，得到 value 的所在位置。
- 将 key-value 两两一起删除。文件的空位将会被记录以让别的数据使用。

5. 修改 (modify)

- 程序会对文件进行树的从根起的解析。为了保证性能，只有被碰到的树节点才会被解析成树节点。
- 找到叶节点及对应的 key 后就解析起信息，得到 value 的所在位置。
- 将 value 删除，另置放空间给新数据，让 key 指向新空间。
- 将树节点重新写入 Index。

6. 查找 (search)

- 若 key 不存在于树中，返回空，终止操作。
- 若 key 存在于树中，获取对应子节点的索引信息，进而在数据文件中读取对应 value，返回 value。



7. 范围查找（range search）

- 得到了两个 key 之间的所有 key 再进行查找（search）。

五 特点

1. 不定长 value

- 在该数据设计的两种 key-value 类型中，value 类型可以不定长长度。
- 实现方法：在索引文文件中保存 value 的位置和长度。

2. 空间的重新利用

```
class DataDel
{
private:
    std::vector<Data> spaces; // stores a vector of extra spaces

public:
    DataDel() {}

    void insertSpace(int offset, int length); // give space to DataDel
    void sortDataDel();
    int getSpace(int dataLength);
};
```

- 利用这些接口，可以实现废弃空间重用。
- 考虑到现今社会数据只会增长不会减少。我就没实现 Index 的垃圾收集。我就只将 Index 的位置记下，以后有新数据时可以重新占用该位置。
- 这样可以保证效率也可以减少资源浪费，毕竟数据只增不减。

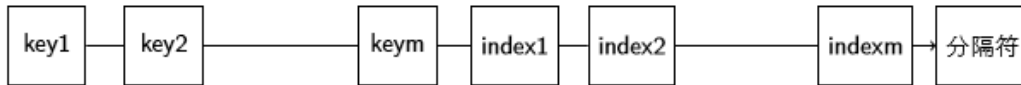
3. 缓存设计

- 我考虑到一般用户的习惯我，刚刚写入的数据往往会再一次查阅，又因为要一致性刷新的需要，我们需要记录一定数量的近期储存的<key,value>到缓存中方便下一次的读取。
- 在拿到 key-value 后，我虽然会立即将树写入文件，可是不会将树擦掉。这样当用户想重新查找时，可以更高效地查找因为树还在程序（内存）里。

4. B+树链表范围查找

- 在写入 B+树的链表后，我可以在不完全读取索引文件的情况下，对数据所在的树结构进行操作。

叶子节点



其中，一条 index 信息即是一对 $\langle \text{offset}, \text{len} \rangle$ 的组合数据；

内部结点



- 另一点需要强调的是 B+树的操作中会涉及大量的节点上溢和下溢的操作，这往往需要递归地操作大量的节点信息，碍于模式 B 的设计限制，必然会导致大量的文件读写操作，反而会导致数据量较大时，难以忍受的时间延迟，无法满足日常需求。也考虑到现今数据的爆发性增长，数据只增不减。所以我只处理上溢问题。如果节点 key 数量太少也不成问题，因为迟早会有数据塞满那个位置。

六 测试分析

1.正确性测试

大量数据测试

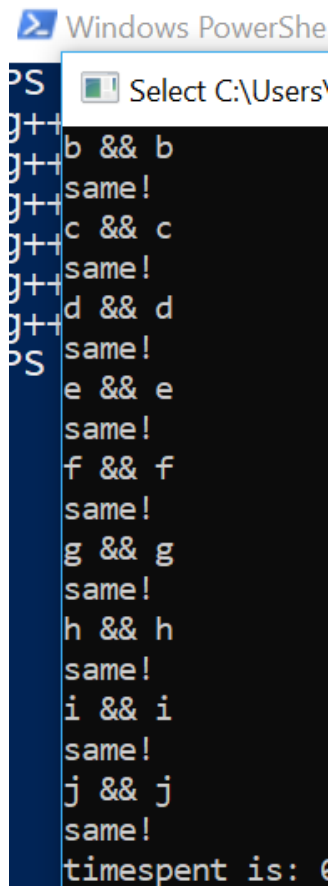
- 建立数据库，分别执行下列操作做
- 所及储存 TIMES 条数据
- 随机删除部分数据
- 读取全部数据
- 随机修改数据
- 读取全部数据，进行对比
- 随机储存 TIMES 条数据
- 读取全部数据，进行对比
- 该测试主要是为了验证大量数据下的最正确性
- TIMES 分别取成 1000,10000,100000,1000000

非寻常测试:

- 建立数据库，分别执行以下操作
- 随机储存 10000 条数据
- 循环以下步骤多次
- 储存一个特定关键字以及值
- 删除该关键字，读取该关键字
- 删除该关键字，读取该关键字
- 修改该关键字，读取该关键字
- 存储该关键字以及一个不同的值
- 该测试主要是为了测试设计删除，存储，修改的异常操作
- 每次循环都返回 value 预想的结果。

综合测试

- 建立一个数据库。
- 随机插入数据。
- 关闭程序，重新开启程序读取数据文件。
- 随机读取一条数据
- 每循环 37 次，随机删除一条记录
- 每循环 11 次，随机添加一条记录并且读取该条记录
- 每循环 17 次，随机替换一条记录为新记录。
- 该测试主要是设计综合存储，删除，修改，读取，操作
- 在每次循环中，都得到与插入是相同的值，以下为测试截图：



```
Windows PowerShell
PS Select C:\Users\
g++ b && b
g++ same!
g++ c && c
g++ same!
g++ d && d
g++ same!
PS e && e
same!
f && f
same!
g && g
same!
h && h
same!
i && i
same!
j && j
same!
timespent is: 6
```

2. 性能测试

B+树的时间复杂度

Insertion:

The image displays two screenshots of a Visual Studio Code editor window, showing C++ code for testing the performance of B+ tree insertion. The code is organized into two versions of a test function, each with its own output window.

Top Screenshot: The code defines a vector of strings, inserts 100,000 samples, and measures the time taken for insertion. The output window shows the result: `timespent is: 455.93`.

```
vector<string> samples;
samples.push_back("ioveu");
samples.push_back("fu");
samples.push_back("hi there!@#$%^&*()_+");
samples.push_back("haha");
samples.push_back("Ratatouille");
samples.push_back("This is a testing sample!!");

int counter = 100000;
while (counter != 0) {
    test.insertDB(counter, samples[rand()%(samples.size())]);
    counter--;
}

test.closeDB();

end = clock();
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

cout << "timespent is: " << time_spent << endl;

system("pause");
return 0;
```

Bottom Screenshot: The code defines a vector of strings, inserts 10,000 samples, and measures the time taken for insertion. The output window shows the result: `timespent is: 3.955`.

```
DBMS test("");
test.openDB(0);
test.closeDB();
test.openDB(1);

vector<string> samples;
samples.push_back("ioveu");
samples.push_back("fu");
samples.push_back("hi there!@#$%^&*()_+");
samples.push_back("haha");
samples.push_back("Ratatouille");
samples.push_back("This is a testing sample!!");

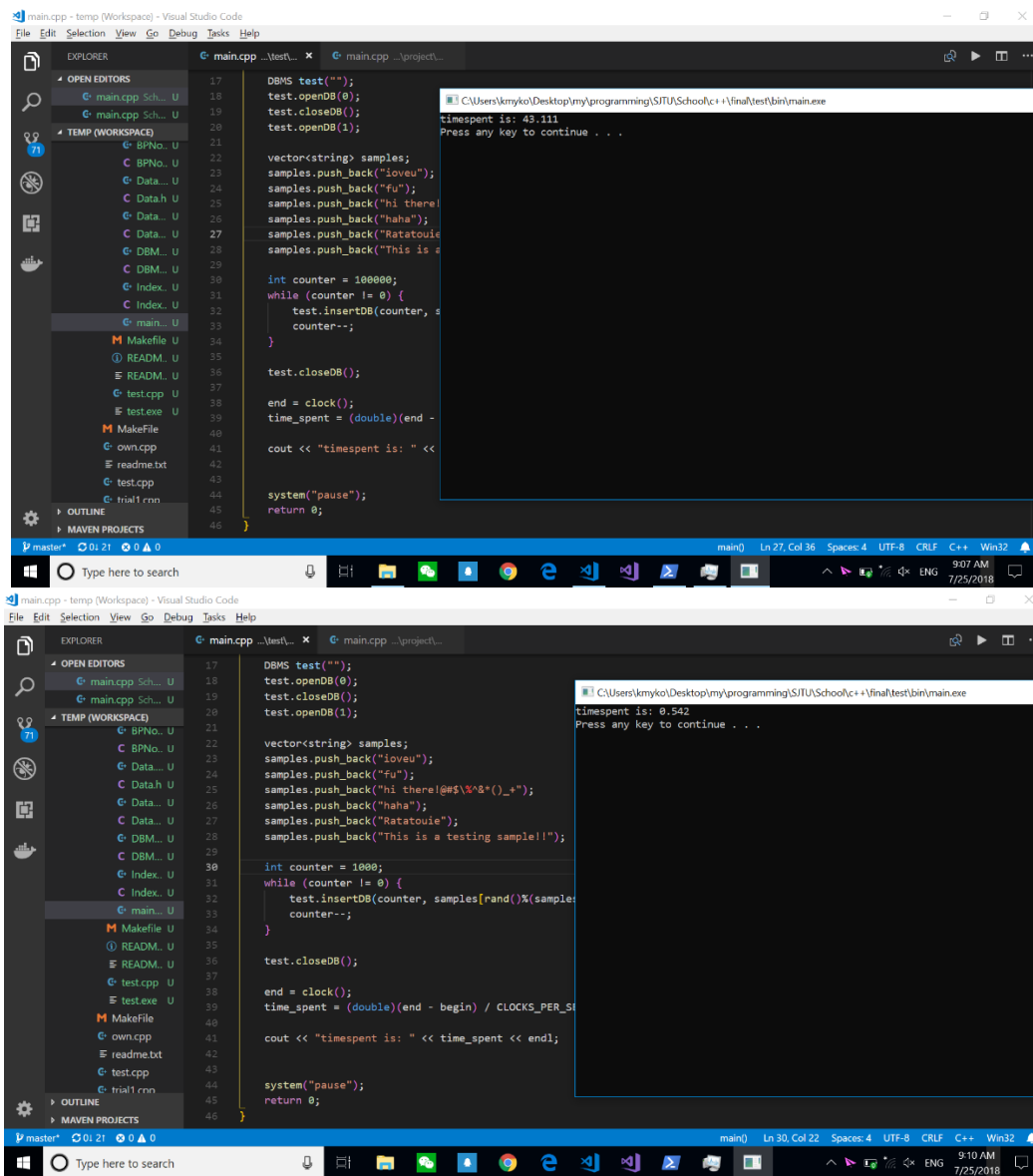
int counter = 10000;
while (counter != 0) {
    test.insertDB(counter, samples[rand()%(samples.size()-1)]);
    counter--;
}

test.closeDB();

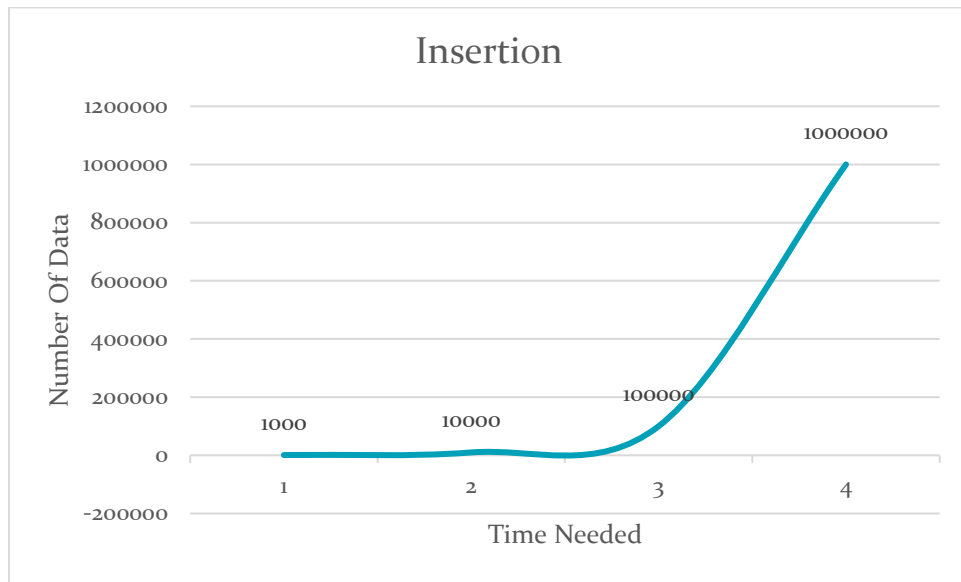
end = clock();
time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

cout << "timespent is: " << time_spent << endl;

system("pause");
return 0;
```



| Number of data | Time needed |
|----------------|-------------|
| 1000 | 0.542S |
| 10000 | 3.955S |
| 100000 | 43.111S |
| 1000000 | 455.93S |



可见，得到的数据作图可以肯定 $O(\log(n))$ 复杂度。

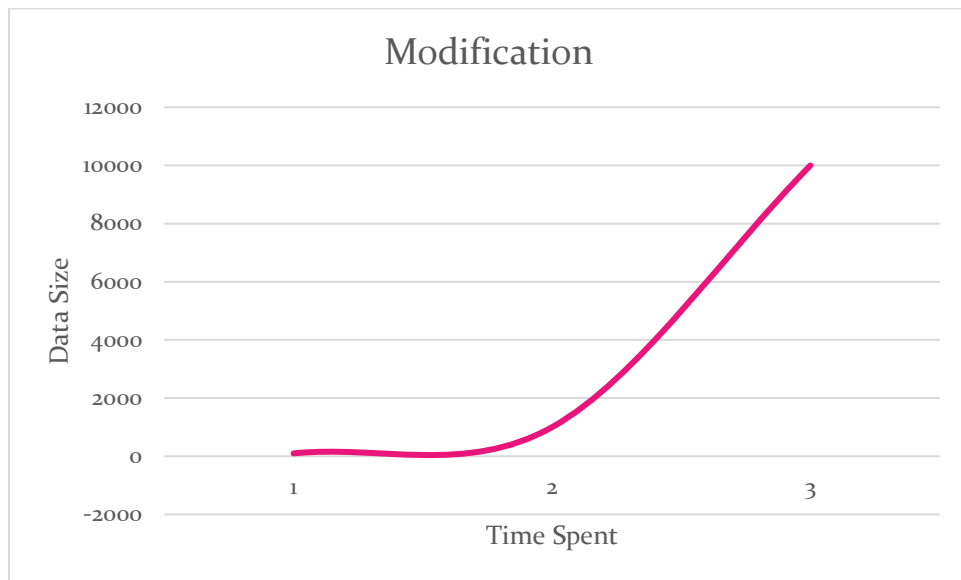
Deletion:

我使用测试系统进行测试得到的结果惊人。1000000 的数据量进行删除只需 2 秒。因此我觉得没必要研究删除的时间复杂度了。肯定是在 $O(1)$ 或 $O(\log(n))$ 之间。否则无法那么快。如果想要测试删除更多数据，Insertion 需要的时间相较也多太多了。因此我可以下的结论是 B+树的删除很有效率。可能需要更多数据才能看到不同。

Modification:

由于我的 API 设计 modification 和 insertion 基本一致。所以可以说他们两的时间复杂度一致。都属于 $O(\log(n))$ 。

| Data Sizes | Time Taken To Complete (s) |
|------------|----------------------------|
| 100 | 0.061 |
| 1000 | 0.054 |
| 10000 | 52.425 |



很明显，改值也是 $O(\log(n))$

Search:

由于使用的数据结构是 B+ 树。无论单值还是范围查找肯定都非常有效率。

以下便是我进行的一些测试截图。各种测试程序可以在 test 文件夹找到。

The image contains two screenshots of the Visual Studio Code editor. The top screenshot shows the 'main.cpp' file with a loop that inserts 1000 random samples into a database. The terminal output shows 'timespent is: 0.524' and 'Press any key to continue...'. The bottom screenshot shows the same code with a loop that performs 1000 range searches. The terminal output shows 'timespent is: 52.425' and 'Press any key to continue...'. Both screenshots show the Explorer, Search, and Run and Debug panels.

```
180 samples.push_back("loveu");
181 samples.push_back("fu");
182 samples.push_back("hi there!@#$%^&*()_~");
183 samples.push_back("haha");
184 samples.push_back("Rabbit");
185 samples.push_back("The");
186
187 int counter = 1000;
188 while (counter != 0) {
189     test.insertDB(counter);
190     counter--;
191 }
192
193 clock_t begin, end;
194 double time_spent;
195 begin = clock();
196
197 int counter2 = 1000;
198 while (counter2 != 0) {
199     test.changeDB(counter2);
200     counter2--;
201 }
202
203 test.closeDB();
204
205 end = clock();
206 time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
207
208 cout << "timespent is: " << time_spent << endl;
209
210 system("pause");
211 return 0;
```

```
Index key;
DataFile value;
std::string keyPath; // stores the file path to index file
std::string valuePath; // stores the file path to data file
```

```
timespent is: 0.524
Press any key to continue . . .
```

```
timespent is: 52.425
Press any key to continue . . .
```

查找基本也是 $O(\log(n))$

关于存储缓存性能优化的测试

将全部资料都存到缓存然后等内存不够时在一次过存入硬盘虽然性能很好，可是对于我觉得用户数据才是最重要的东西。性能当然要好不然没人会为了一个数据等几个小时但是如果数据不安全再好性能的数据库都没用。数据的安全性应该放在第一的考量范围。

如果将数据先存在内存里，等时间适合再存进硬盘中不能保证数据安全。因为如果数据还没存进硬盘中突然停电或者程序突然崩溃，数据就没了。

因此，我对缓存储存不是很看重。用户储存数据后，我第一件事就是建树，插值然后直接存树索引及存值入硬盘。

或许你会觉得这样太浪费效率。为了补全，我设计了读取缓存。当我们插值后树在内存里建好后我不会将它们删除。这样就能保证下次读数据能从内存里读取。而不是从硬盘重新读取。只有程序退出或内存不够我才会从内存回收树。

这样的设计不仅能解决最重要的安全问题，同时也保证最常用情况下性能的保证。

关于空间的重新利用

具体可以参考 Data.h

我设计了可以记录删除数据后剩下的位子位置（offset）及长度（length）。下次用户再次插值程序将会先从记录里寻找位置。如果没有合适的位置程序才会占用文件的新位置。

七 致谢以及参考

这次大作业虽然让我获益良多可是代码量也实在太多了。可能是我还不是很厉害，这个数据库从构思到实现用了我超过两个星期。因为隔着的时间也较久，之前写过的代码功能过后又忘了导致实现时有些困难。整个数据库的实现难点我觉得是 B+树的实现。毕竟网上的代码又复杂又不齐全。我是参考 Java 和 python 的 B+树实现然后将他们转换成 C++的。当然由于 Classes API 很多，当程序报错误时我只能慢慢用 debugger 或 std::cout 慢慢深入地查下去，非常费时。

虽然很辛苦但是很高兴老师能给我机会让我实现这次的数据库。它不仅让我更了解数据库的操作也让我对 B+树数据结构有更深入的了解。也难怪 linux 线程也在用 B+树做索引(indexing)。感谢同学的帮忙也感谢助教们在群里的谈话让我有所启发。感谢戚老师，任老师，李老师的教导。感谢他们让我认识数据结构。

由于我的能力有限，设计时写了许多不必要的函数。虽然我最后也已经尽量清理了可是难免还会漏掉一些。B+树的实现我也大多参考网上的内容写出的。也由于我的中文能力有限。所以这报告一些比较简单的部分我用了一些英文来讲解。希望老师助教们不会介意。内容中如果有误导的地方也请见谅因为我的报告有 20%的内容利用了 Google 翻译。

谢谢

References

<https://www.sqlite.org/c3ref/intro.html>

<https://use-the-index-luke.com/sql/anatomy/the-tree>

<https://www.geeksforgeeks.org/database-file-indexing-b-tree-introduction/>

https://cstack.github.io/db_tutorial/parts/part7.html

<https://www.youtube.com/watch?v=aZjYr87r1b8>

...还有很多的 Google!