## 主要的数据结构

创建了多个类。

Bit 位数组类，因为bloom只需要一个bit就可以进行表示，但是在c++中的基础类型最小的也是一个字节，8个bit，因此处于节省空间的角度，使用char数组模拟位数组，也就是将一个char看作是可以存储8个bit的数据，这样相比于直接使用bool型空间缩小了8倍。

class Bit {  
private:  
 long long int length;//表示这个二进制数组的长度  
 char \*head;//使用char数组模拟位数组  
 long long int charNum;//创建的char数组的长度  
public:  
 Bit(long long int length);  
 void set(long long int index, bool value);//true表示设置为1 false表示设置为0  
 bool get(long long int index); //true表示为1 false表示为0  
};

readFile类，读取文件类，这个类是复用的上个hash作业的readFile类。

class readFile{ //读取文件  
private:  
 ifstream infile; //文件变量  
 int lines;//获取的文件行数  
 void Lines(); //获取文件行数的函数  
public:  
 readFile();  
 readFile(std::string filePath);  
 int getLines(); //返货lines的值  
 string getNextLine(); //返回下个字符串，返回空表示已经读完  
};

bloom类，这个类是程序的主要操作类，里面有MurmurHash2()函数，可以根据不同的seed来返回不同的hash值；createDict()函数，创建字典；queryString()函数，查询一个字符串是否在创建的hash字典中；queryFile()函数，查询一个文件的字符串在创建的hash字典中的查询情况。

class bloom{  
private:  
 long long int M; //计算出来的数组长度  
 long long int N; //给定的字符串的个数  
 float fp;//允许错误率  
 int K; //hash函数个数  
 Bit \*bit; //位数组  
 string dictPath; //字典文件路径  
 readFile \* file; //读取dict.txt文件变量  
 int \*seed; //创建不同的hash函数时使用的随机种子  
public:  
 bloom(float fp, string dictPath, int \*seed=nullptr);  
 long long int \* getHashKey(const string& str, int num); //字符串 hash个数，返回该字符串的num个hash值  
 uint32\_t MurmurHash2(const void \* key, int len, uint32\_t seed);//hash的母函数  
 void createDict(); //构建hash字典  
 bool queryString(string str);//查询一个字符串是否存在 存在返回true 不存在返回false  
 void queryFile(string filePath,string resultFile);//对一个文件的字符串进行查询 将查询到的内容放到该文件中  
};

## 实验过程

将dict.txt和words.txt放到lab1\_2目录下。

首先创建一个bloom类，然后在里面的构造函数会根据输入的期望错误率和字典大小N计算出M位数组大小和K需要的hash函数个数，然后根据M调用Bit类创建相应大小的位数组，再调用创建字典函数对所有字符串进行hash表示并将相应的位向量置为1。最后调用queryFile()函数对整个文件的所有字符串进行查询操作，其中对于每个字符串会调用queryString()函数查询该字符串是否存在。

实验结果如图1所示。

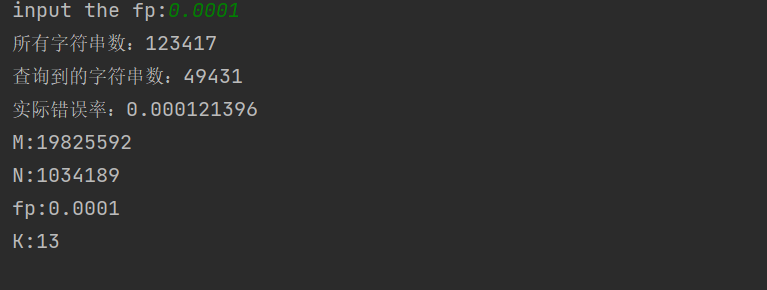


图1

## 遇到的问题

在fp=0.0001的时候，查询任务管理器该进程的内存，发现内存高达16.2MB，这和自己计算出来的以及老师上课讲的内存大小相差了一个数量级。然后我采用中间进行输入来暂停程序运行的方法来观看当时的程序内存大小，发现在进行查询(queryFile())前和查询后内存变化比较大，最后检查代码，发现是在queryString()函数中创建的long long int\*key指针，每次new，但是没有将其free掉，所以导致程序内存比较高。当将这个地方进行修改之后，程序运行内存变为了2.9M,这和上课老师讲的内存大小基本一致。

## 结果指标

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 预设错误率 | M | K | 实际错误率 | 内存 |
| 1034189 | 0.001 | 14869194 | 9 | 0.00165908 | 2.3MB |
| 1034189 | 0.0001 | 19825592 | 13 | 0.000121396 | 2.9MB |
| 1034189 | 0.00001 | 24781989 | 16 | 0.0000202327 | 3.5MB |
| 1034189 | 0.000001 | 29738388 | 19 | 0.0000202327 | 4.1MB |

## 结论和总结

在这次实验中通过与上次hash实验进行对比，发现bloom的优势非常明显，其空间复杂度直接下降了一个数量级，虽然准确率没有达到百分百，但是在预设错误率为0.00001的时候，其只是相比于正确结果多查询了一个字符串，但是其CPU内存却仅仅需要3.5MB。这种情况对于更大的字符串数据量会有比较明显的空间优化，而代价不过是有少数字符串查询不正确。

尤其是当字符串的平均长度比较长的时候，bloom的优势会更加明显，因为常规的hash存储的是每个字符串，而bloom存储的只是该位置是否为1或者0，因此当字符串长度增加的时候，bloom的使用内存不会发生变化，但是普通的hash操作内存则和字符串的长度成线性关系。

在很多时候可能并不需要百分百的争取率，这个时候就可以使用bloom来优化控件内存。