## 主要数据结构和流程

总体数组结构是单链表+指针数组完成。

创建了五个类，分别是readFile、linkList、Hash、Query，功能分别是完成文件读取，链表创建添加、创建hash表、对指定内容在hash表中进行查询并返回统计结果、定义了十一个hash函数。

readFile类中主要有两个函数，分别是readFile()读物文件和getNextLine()获得下一行的字符串。

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  
};

linkList类中主要是addLink()函数将需要添加的指针添加到该链表头部，实现头部插入，减少插入时间。

class linkList{//链表类  
private:  
 linkList \*next; //指向下一个链表的指针  
 string value; //该节点的值  
public:  
 linkList() {  
 next = nullptr;  
 }  
 void addLink(linkList \*nextLink){ //将nextLink节点添加到this节点之后  
 linkList \*temp = next;  
 next = nextLink;  
 nextLink->setNext(temp);  
 }  
 void setNext(linkList \* temp){ //设置该节点next指针  
 next = temp;  
 }  
 linkList \* getNext(){ //返回该节点next指针  
 return next;  
 }  
 void setValue(string str) {   
 value = std::move(str);  
 }  
 string getValue(){  
 return value;  
 }  
};

Hash类中主要函数为getHashKey()返回字符串的hash值、hashDict()创建hash表。

class Hash{  
private:  
 readFile \*file; //读取文件的指针  
 linkList \*hashList; //hash表的头节点  
 string dictPath; //字典路径  
 int selectHashFunc;//选择hash的标志  
 string selectHashName;//选择的hash的名字  
public:  
 Hash(const string& dictPath,int selectHashFunc=0){  
 this->dictPath = dictPath;  
 this->selectHashFunc = selectHashFunc;  
 file = new readFile(dictPath);  
 selectHashNameFunc();  
 N = file->getLines();  
 M = N / a;  
  
 hashList = hashDict();  
  
 }  
 linkList \*getHashList() const;//返回headList指针  
 void selectHashNameFunc();//确定选择的hash的名字  
 unsigned int getHashKey(char \*str); //返回字符串的hash值  
 void printResult(); //输出结果  
 linkList\* hashDict(); // 对字典建立hash表  
  
};

Query类中主要函数为query()查询一个文件中所有字符出现情况的统计、queryHash()查询一个字符串是否在hash表中。

class Query{  
private:  
 readFile \*file; //读取文件的指针  
 Hash \*hash; //hash表的指针  
 string wordsPath; //文件的地址  
 int needQueryNum;//需要查询的单词数  
 int alreadyFindNum;//查询到的次数  
 int sumLength;//查询的总长度  
 int alreadySumLength; //查询到的计算的总长度  
public:  
 Query(Hash \*hash,string wordsPath=""){  
 this->wordsPath = std::move(wordsPath);  
 this->hash = hash;  
 needQueryNum = 0;  
 alreadyFindNum = 0;  
 sumLength = 0;  
 alreadySumLength = 0;  
 }  
 pair<bool,int> queryHash(const string& str); //<是否查到，查询次数> 查询str是否在hash表中  
 void query(); //查询wordsPath文件在hash表中出现的结果  
 void printResult();//输出结果  
};

HashFunc类主要定义实现了是一个hash函数。

class HashFunc{  
public:  
 static unsigned int RSHash(char \*str);  
 static unsigned int JSHash(char \*str);  
 static unsigned int PJWHash(char \*str);  
 static unsigned int ELFHash(char \*str); //对str求解hash  
 static unsigned int BKDRHash(char \*str);  
 static unsigned int SDBMHash(char \*str);  
 static unsigned int DJBHash(char \*str);  
 static unsigned int DEKHash(char \*str);  
 static unsigned int BPHash(char \*str);  
 static unsigned int FNVHash(char \*str);  
 static unsigned int APHash(char \*str);  
};

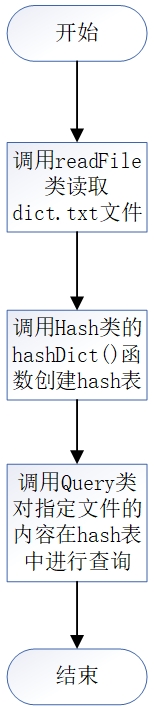


图1 总体流程

总体流程如图1所示，首先是读取文件内容，然后调用Hash类的hashDict()创建hash表，最后使用Query类对word.txt中的内容进行查询，并将统计结果进行返回。

## 实验过程

将dict.txt和words.txt放到lab1\_1目录下。

运行主程序，会提示输入装填因子，输入装填因子之后，程序就会自主运行，然后分别使用十一个hash函数创建hash表，然后进行查询操作，最后输出十一个hash表的查询统计结果。

实验结果图如图1所示。

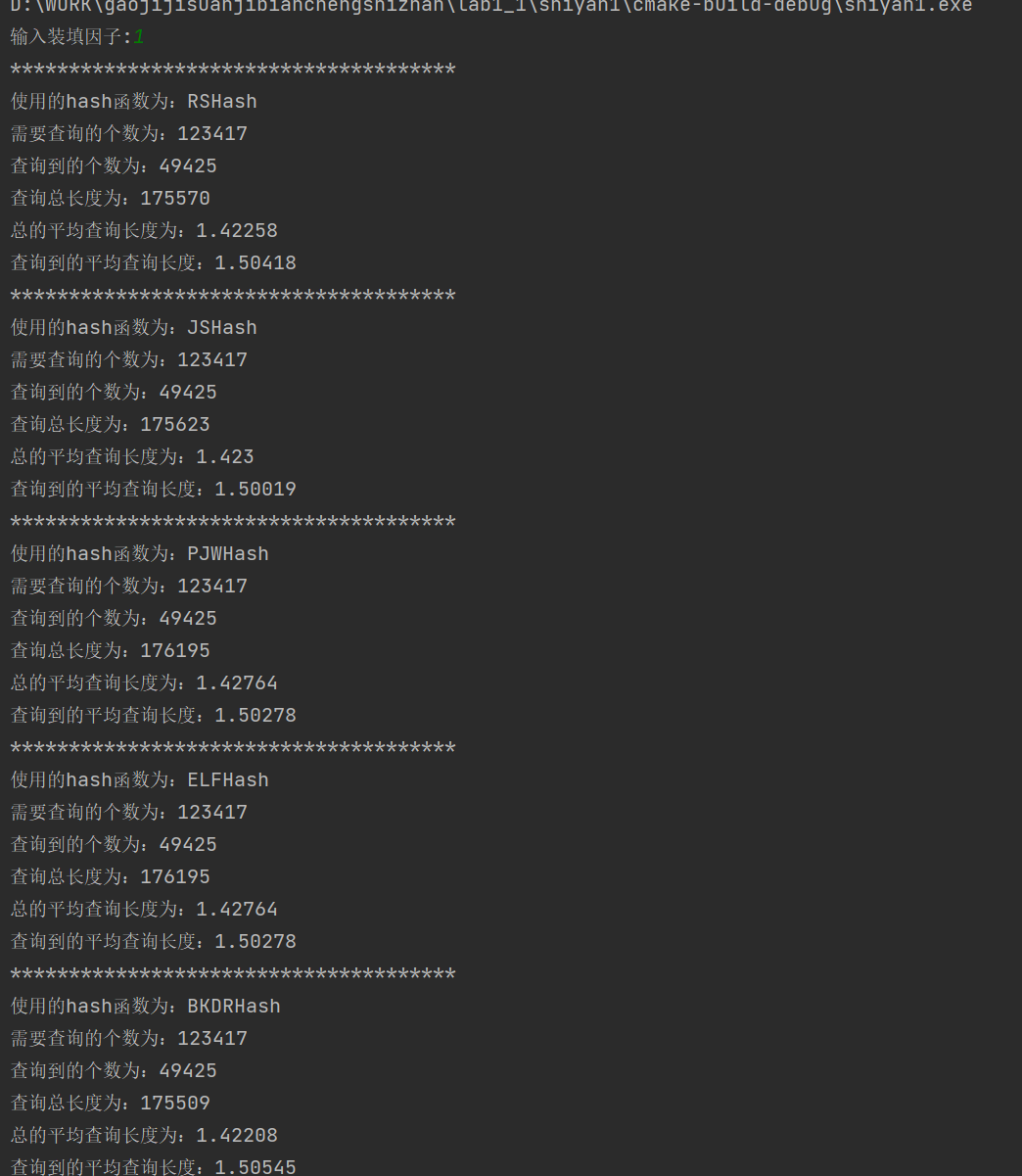


图1

## 遇到的问题

最开始运行程序的时候，发现不同的装填因子会产生不同的查询到的单词个数，这明显是错误的，因为查询到的个数应当是确定且唯一的，最后检查程序中间结果，发现是在进行查询的时候，链表的最后一个节点总是被忽略了，这样就导致了不同的装填因子产生了不同的错误结果。

## 结果指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指标\装填因子 | 1 | 2 | 10 | 50 |
| cpu内存（MB） | 61.2 | 51 | 50.9 | 51.6 |
| 平均比较次数 | 1.50418 | 1.99694 | 5.99033 | 25.9662 |

## 结论和总结

当装填因子比较大的时候，例如10 50，此时的平均比较此时大约是装填因子的一半左右，说明hash函数的分布是非常均匀的，我之前试过单纯的使用一个简单的求余的hash函数的平均比较次数相比较上课给出的hash函数会大一个数量级，这充分表明了一个好的hash函数对于查找效率是有非常大的提升的。

对课上给的十一个hash函数都进行了实现，并观察最后的结果，发现除了BPhash函数之外，其余十个hash函数的平均比较次数和上述结果指标的平均次数差不多，但是BPhash的平均比较次数远大于这十个hash函数的比较次数，怀疑可能是BPhash的均匀分布并不是很好。