**北 京 邮 电 大 学**

**计算机应用编程实践**

**实验报告**



|  |  |
| --- | --- |
| 题目： | 海量字符串检索 |
| 姓名： | 田昌昊 |
| 学号： | 2019110680 |
| 班级： | 2019111307 |

计算机学院

2019年9月

目录

[一、实验要求与实验内容 3](#_Toc20334046)

[1. 实验要求： 3](#_Toc20334047)

[2. 挑战： 3](#_Toc20334048)

[3. 程序要求 3](#_Toc20334049)

[二、实验过程 3](#_Toc20334050)

[1. 主要的数据结构和流程 3](#_Toc20334051)

[1.1 版本一array\_search 3](#_Toc20334052)

[1.2 版本二hashtable\_search 4](#_Toc20334053)

[1.3 版本三bf\_search 4](#_Toc20334054)

[2. 实验过程 4](#_Toc20334055)

[3. 优化过程 5](#_Toc20334056)

[三、遇到的问题 5](#_Toc20334057)

[1. 文件编码不一致 5](#_Toc20334058)

[2. 读取words出错 5](#_Toc20334059)

[3. 对hash值计算出错 5](#_Toc20334060)

[四、结果指标 6](#_Toc20334061)

[1. 优化前结果 6](#_Toc20334062)

[2. 优化后结果 7](#_Toc20334063)

[五、结论和总结 8](#_Toc20334064)

# 一、实验要求与实验内容

## 1. 实验要求：

在给定的海量个数的字符串中查找特定的字符串。

## 2. 挑战：

1. 实际需求
   1. 几亿规模
2. 数据量大
   1. 200,000,000量级
3. 外存便宜
   1. 存储成本低
4. 内存不够大
   1. 200,000,000\*40bytes=8000,000,000bytes=8G

## 3. 程序要求

1. 分别实现三个版本，程序分别命名为：array\_search、hashtable\_search和bf\_search
2. 输入数据
   1. 字典串pattern.txt(127w个)
   2. 待匹配的98w字符串：words.txt
3. 实验结果result.txt
   1. 在模式串中的输出yes，不在就输出no
      1. Keyword1 yes
      2. Keyword2 no
      3. 最后一行输出四个数字，用空格分割：
         * 检索结构占用内存量（kb单位）
         * 字符比较次数
         * words总个数
         * 成功检索到的word总个数

# 二、实验过程

## 1. 主要的数据结构和流程

### 1.1 版本一array\_search

主要数据结构如下：

typedef struct Array {

char\* str;

}\*ArrayPtr;

本程序先读取words文件和patterns文件，获取数据，存于Array结构数组中，其中对内存的使用动态分配并记录。然后进行检索工作：遍历words数组，并从patterns数组中遍历查询是否存在与之相等的字符串，如果检索到相同的字符串，则返回并将检索成功结果写入文件；如果检索结束仍然未找到相同字符串，则返回并将检索失败结果写入文件。最后将统计信息写入文件最后一行。

### 1.2 版本二hashtable\_search

主要数据结构如下：

typedef struct Array {

char\* str;

}\*ArrayPtr;

typedef struct List {

char\* str;

struct List\* next;

}\*ListPtr;

typedef struct Dict {

int num;

ListPtr next;

}\*DictPtr;

本程序先读取words文件和patterns文件，获取数据，其中words存于Array结构数组中，patterns存于Dict结构数组中，Dict结构为字典链表结构，并对内存的使用动态分配并记录。然后进行检索工作：遍历words数组，并获取word对应的hash值，从patterns结构中查询，并遍历节点的链表，查询是否存在与之相等的字符串，如果检索到相同的字符串，则返回并将检索成功结果写入文件；如果检索结束仍然未找到相同字符串，则返回并将检索失败结果写入文件。最后将统计信息写入文件最后一行。

### 1.3 版本三bf\_search

主要数据结构如下：

typedef struct Array {

char\* str;

}\*ArrayPtr;

typedef struct BloomFilter {

char\* bits;

}\*BFPtr;

本程序先读取words文件和patterns文件，获取数据，其中words存于Array结构数组中，patterns存于BloomFilter结构中，BloomFilter结构包含一个超长的bit段，用于存储pattern的特征，并对内存的使用动态分配并记录。然后进行检索工作：遍历words数组，并获取word对应的Hash\_Num（暂定8个hash函数以及8MB的bits长度）个hash值，从BloomFilter结构中查询其bit位是否均为1，从而查询是否存在与之相等的字符串，如果检索到0位，则直接返回检索失败，将结果写入文件；如果检索全为1，则返回检索成功结果并写入文件。最后将统计信息写入文件最后一行。

## 2. 实验过程

在实验时，首先编写了文件的读取操作，并查看是否成功读取数据，后来发现两个文件的编码不一致，导致部分乱码，所以最后将两个文件均调整为utf-8无签名的编码格式进行操作。

之后首先编写了array\_search.c进行运行测试，后来发现运行时间过长，导致结果短时间不易查看。所以着手编写hashtable\_search。编写过程中从网上找到了常用的哈希函数并进行代码封装，供程序使用。完成后得到匹配正确个数为516816个words。

根据hashtable\_search得到的结果和array\_search得到的结果进行比对，发现前者的结果也正确。

版本三的过程比较顺利，直接就得到了想要的结果，后续只是对各种参数情况进行测试，并选取较优结果。

## 3. 优化过程

在完成第一版本的实验后，发现words无需存于内存中，而在匹配的时候实时从文件读取并进行匹配即可，故将三个实验的words结构均省去，对内存的使用量节省了不少。

# 三、遇到的问题

## 1. 文件编码不一致

在读取文件的过程中，发现两个文件的编码方式不一致。后续将两个文件调整为utf-8无签名编码格式进行操作。

## 2. 读取words出错

在读取文件的过程中，发现words文件一共986004行，但是却读取了986007个words。

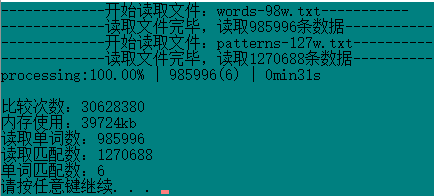
后续发现是每行读取的字符size过小，导致其中三行的单个字符串被分割为两个字符串，后续加大了每行的缓存大小size=100，得到986004个words

之后又发现文档中存在字符串为空的8行。后续筛除这些空行，得到985996个words。

ps：有同学读取words比我多了3个，后续查询发现有三个单词中间存在空格，需要注意。这三个单词分别为：脱 水、伊朗 自由词、炼 油厂。

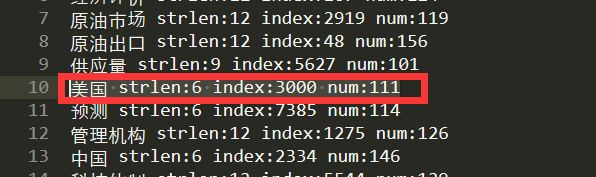
## 3. 对hash值计算出错

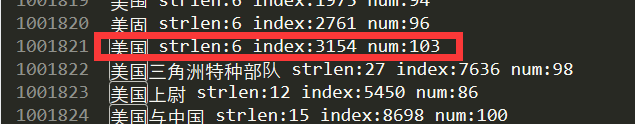
当完成hashtable\_search时，发现准确度很低，整个文件只有6个匹配成功的。查看匹配成功的字符串，均为“濮城”，可能是hash值前后不对应，而这个单词刚好偶然前后对应了。



后续DEBUG，将所有的words和patterns得到的hash值进行比对，发现在读取patterns文件并整理为Dict结构的过程中，在取hash值时传入了读取的raw值，由于该读取的数据包含最后的’\n’，导致计算的hash和去掉’\n’计算的hash值不同，从而无法匹配。

示例如下（以“美国”举例），strlen为words和patterns中的字符串长度（不是raw值，而是去除末尾’ \n’后的字符串，index为hash值，num为当前节点的链表长度）：





# 四、结果指标

## 1. 优化前结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **比较次数** | **内存使用（kb）** | **单词数/匹配数** | **单词**  **匹配数** | **运行时间** |
| **array\_search** | 219081105981 | 34561 | 985996/1270688 | 516816 | 931min2s |
| **hashtable\_search** | 26988453 | 39724 | 985996/1270688 | 516816 | 24s |
| **bf\_search** | 4754225 | 20313 | 985996/1270688 | 516816 | 3s |

其中hashtable\_search字典大小为10000，采用的是BDKRHash函数，因为该函数的效果最优，能够均匀分配各字符串，每个节点得到的链表长度均在110-130之间。而bf\_search采用的bits为3MB长度，hash函数为8个。hash函数如下，顺序排列，前8个即为所采用的：

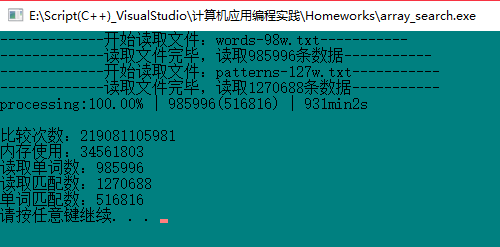
unsigned int(\*Hash\_List[])(char\*, unsigned int) = {

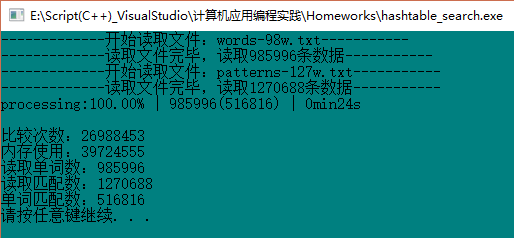
BKDRHash, RSHash, JSHash, PJWHash, ELFHash,

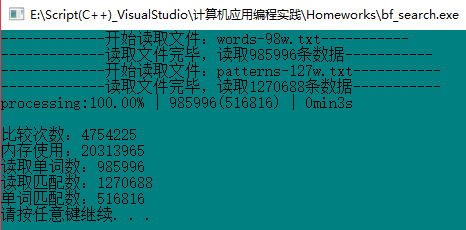
SDBMHash, DJBHash, DEKHash, BPHash, FNVHash,

APHash, MyHash};

上述结果运行输出如下：







## 2. 优化后结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **比较次数** | **内存使用（kb）** | **单词数/匹配数** | **单词**  **匹配数** | **运行时间** |
| **array\_search** | 219081105981 | 18010 | 985996/1270688 | 516816 | 890min21s |
| **hashtable\_search** | 26988453 | 23051 | 985996/1270688 | 516816 | 18s |
| **bf\_search** | 5874234 | 3072 | 985996/1270688 | 516816 | 4s |

其中hashtable\_search字典大小为10000，采用的是BDKRHash函数，因为该函数的效果最优，能够均匀分配各字符串，每个节点得到的链表长度均在110-130之间。而bf\_search采用的bits为3MB长度，hash函数为10个。hash函数如下，顺序排列，前10个即为所采用的：

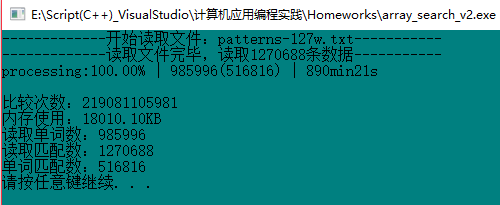
unsigned int(\*Hash\_List[])(char\*, unsigned int) = {

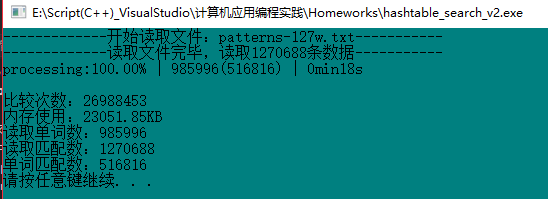
BKDRHash, RSHash, JSHash, PJWHash, ELFHash,

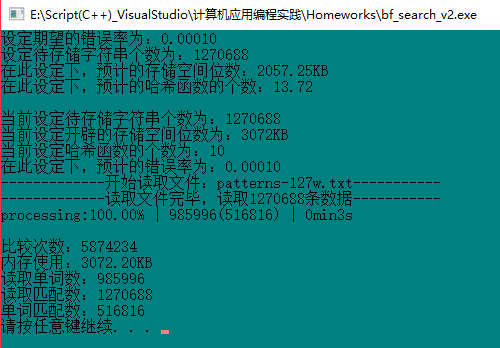
SDBMHash, DJBHash, DEKHash, BPHash, FNVHash,

APHash, MyHash};

上述结果运行输出如下：







# 五、结论和总结

通过本次实验，进行了大数据的字符串匹配，体会到了暴力算法的耗时，同时也体会到了数据结构的重要性，以及对内存使用的注意事项。

完成此次任务的成就感还是满强的。