# 实验报告 Hash技术字符串检索

## 主要数据结构和流程

### 版本一：数组实现array\_search

1.该结构用来存储程序执行过程的字符比较次数、内存占用大小、模式串个数、查找串的个数、查找成功个数

**struct** {

**long** cmpnum

**long** mem;

**long** pattern\_num;

**long** word\_num;

**long** compatible;

}Global\_stats;

2.保存模式串的结构

**struct** pattern{

**char** s[LEN];

};

3.字符比较函数

**int** byte\_cmp(**char** a, **char** b);

4.字符串比较函数

**int** string\_cmp(**char** \*a, **char** \*b);

5.动态申请内存函数

**void**\* bupt\_malloc(size\_t size);

6.快速排序函数

**void** QuickSort(**struct** pattern \*\* a, **long** left, **long** right);

7. 二分查找函数

**int** binary\_search(**char** \* key,**struct** pattern \*\* a,**long** n);

流程：从文本文件读取所有模式串，通过动态申请一个结构体数组来保存顺序所有模式串。然后通过快速排序，使字符串按照字典顺序排列。接着从文本文件中读取查找串，以二分查找方式对数组进行查找，并将结果写入result.txt文件中。最后释放动态申请的内存空间。

### 版本二：哈希表实现hashtable\_search

1.该结构用来存储程序执行过程的字符比较次数、内存占用大小、模式串个数、查找串的个数、查找成功个数

**struct** {

**long** cmpnum

**long** mem;

**long** pattern\_num;

**long** word\_num;

**long** compatible;

}Global\_stats;

2.保存模式串的结构

**struct** pattern{

**char** s[LEN];

};

3.字符比较函数

**int** byte\_cmp(**char** a, **char** b);

4.字符串比较函数

**int** string\_cmp(**char** \*a, **char** \*b);

5.动态申请内存函数

**void**\* bupt\_malloc(size\_t size);

6.hash函数

**unsigned** **int** PJWHash (**const** **char**\* str, **unsigned** **int** len);

7.查找散列表函数

**int** search(**char** \* s,**int** len, **struct** pattern \*\* a);

8.释放链表内存函数

**void** free\_vector(**struct** pattern \* p);

流程：从文本文件读取所有模式串，通过动态申请一个结构体的指针数组来创建散列表，并选择一个hash函数，计算每个模式串的地址，并以拉链法处理冲突。接着从文本文件中读取查找串，计算其散列地址，比较模式串是否相同，并将结果写入result.txt文件中。最后释放动态申请的内存空间。

### 版本三：Bloom Filter实现

1.该结构用来存储程序执行过程的字符比较次数、内存占用大小、模式串个数、查找串的个数、查找成功个数

**struct** {

**long** cmpnum

**long** mem;

**long** pattern\_num;

**long** word\_num;

**long** compatible;

}Global\_stats;

2.字符比较函数

**int** byte\_cmp(**char** a, **char** b);

3.字符串比较函数

**int** string\_cmp(**char** \*a, **char** \*b);

4.动态申请内存函数

**void**\* bupt\_malloc(size\_t size);

5. MurmurHash函数

uint32\_t murmur3\_32(**const** uint8\_t\* key, size\_t len, uint32\_t seed);

6.二进制向量填充函数

**void** fill\_vector(**char** \* a, size\_t len, **char** \* vector);

7.查找函数

**int** word\_filter(**char** \* a, size\_t len, **char** \* vector);

流程：取差错率为0.0001%，通过公式计算得到适合的二进制向量长度和hash函数个数。从文本文件读取所有模式串，动态创建一个合适的二进制向量，并通过多次hash，将对应的二进制位填充成1，依次操作所有模式串，则完成布隆过滤器。然后，从文本文件中读取每个查找串，对其进行多次hash，若对应的二进制位均为1，则认为该串查找成功。将所有查找结果写入result.txt文件中，并释放动态申请的空间。

参数计算松露：

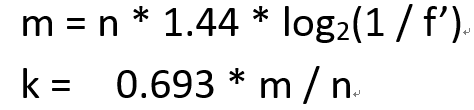
f: 期望的错误率

n: 待存储的字符串个数

m: 需开辟的存储空间位数

k: 哈希函数的个数

取f=0.0001%，n=1270000

公式：

得到 m= 43000000，k=23

## 实验过程

1. 该实验在Mac OS上进行，所以安装Xcode作为集成开发环境。
2. 根据版本1要求，设计数据结构，编写C语言代码，构建可执行程序，并运行程序生成结果。
3. 根据版本2要求，设计数据结构，编写C语言代码，构建可执行程序，并运行程序生成结果。
4. 根据版本3要求，设计数据结构，编写C语言代码，构建可执行程序，并运行程序生成结果。
5. 编写实验报告并优化代码。

## 遇到的问题

1. C语言语法知识遗忘，导致代码的错误编写。
2. Mac OS环境下，对集成开发环境Xcode不熟悉，使得在代码调试和构建过程中遇到问题。
3. 模式串文件编码为GBK，与实验操作环境不匹配，需要运用其他工具转码为UTF-8。
4. 对实验三的MurmurHash函数陌生，不理解如何使用。
5. 对于位操作的掌握不足。

## 结果指标

### 版本一：数组实现array\_search

* CPU运行时间：75s
* 申请内存：47.8MB
* 字符比较次数：24650703186
* 匹配字符串个数：516819

### 版本二：哈希表实现hashtable\_search

* CPU运行时间： <1s
* 申请内存：58.2MB
* 字符比较次数：13241852
* 匹配字符串个数：516819

### 版本三：Bloom Filter实现

* CPU运行时间： <1s
* 申请内存：5.1MB
* 字符比较次数：12857979
* 匹配字符串个数：516819

## 结论和总结

比较三个版本的程序，发现三个程序均能得到正确结果。然而，通过数组方式存储模式串，字符比较次数最大，需要运行最长的时间，并且时间花销巨大；通过hash表搜索，时间开销大大减少，但占用了最多的内存；通过Bloom Filter方式，不仅所需内存大大减少，运行时间也和hash方式相差不大，字符比较次数最少。

本次实验，体现了在数据庞大的情况下，不同数据结构的程序执行的差异。通过比较三个版本的程序，了解了Bloom Filter的准确度换空间的思想，这对于数据量巨大的大数据时代下，支撑很多网络应用的实现，具有重大的价值和学习意义。