计算机应用编程实验报告

计算机应用编程实验 2：Hash 技术字符串检索

2020110656 高垚 2020111562 杨恩

一 实验目标与意义

随着网络技术的普及，互联网的信息逐渐庞大，如何在海量且冗余的信息中找到用户所需的结果是互联网每天都要应对的难题。信息的基数与增长速度，用户查找内容的多样性，查询结果的实时动态变化都使精确搜索成为互联网面临的挑战。

本实验的目的是设计一套能够在庞大的词库中寻找特定目标的算法。需要通过 Hash 链表和 Bloom 过滤器两种方法在给定的具有 127 万多个词语的 dict.txt 词典中，查找有 1.7 万个词的 string.txt 文本中匹配的词。

二 系统设计与实现

根据实验目标要求，综合考虑时间复杂度和空间复杂度，在实验第一部分采用 Hash 链表作为海量搜索的基础，判断 string.txt 文本中哪一行的词条是 dict.txt 词典中拥有。实验第二部分利用 Bloom 过滤器快速检测词条是否是所搜索的目标。

2.1 Hash 链表

Hash 算法是一种利用 Hash 函数，将信息编码成一组关键字，这一组关键字将被映射到一个有限的、地址连续的地址集 (区间) 上，如图 1，在查找方面，其时间复杂度下降到 O(1)；在添加与删除节点方面的时间性能也极快，是一种利用空间成本，节约时间的一种算法。本方法利用 Hash 链表解决冲突。实验中，涉及到的技术有 Hash 链表的构建与查找，文件的读取与写入工作。

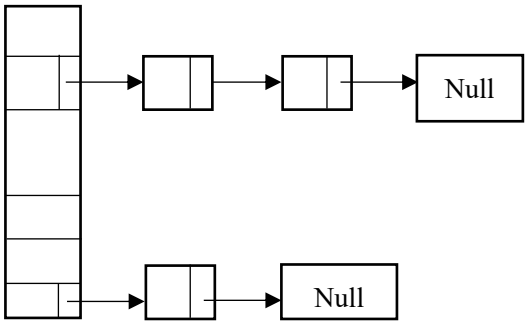


图 1 Hash 链表

2.1.1 实验思路

本节概括了实验的流程。

- 为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 打开用到的文件，包括给定的两个输入文本 string.txt 和 dict.txt，以及输出文本 result.txt。
2. 粗过滤的第一步：给 string 文本中出现的前三个字置为 1。
3. 计算 string 文本中 Hash 的值。
4. 计算 string 文本中，每一行的 Hash 值。

5. 根据 string 文本构建 Hash 表，以及将计算的 Hash 结果存入 Hash 表中。若产生冲突，则采用链式链表解决。
6. 粗过滤的第二步：判断 dict 文本中的前三个字符是否出现在 string 文本。如果出现过，则继续利用 Hash 值精确搜索，否则跳过这一行。
7. 如果成功查找到，则写入 result 文本中。
其中涉及到的函数有：

```
/*本实验中用到的函数*/
HashNode* create_new_node(char* str)//根据 string 文本创建 Hash 表，并为其赋值。
int get_hash(char* str)// 根据 string 文本中每一行的文本计算 Hash 值。
FileHandler* open_file(char* file_name, char* open_type) //创建指向文本的指针和内存缓冲区。
int read_line(FileHandler* file_handler, char** str) **//读取文本的内容，保存至内存。
int write_line(FileHandler* file_handler, char* str)//把内存中的值写入创建的文本中。
int close_file(FileHandler* file_handler)//关闭文件，释放空间。
```

2.1.2 预实验

（1）Hash 函数的选择

Hash 函数有多种，其中包括 SDBMHash，RSHash，JSHash，PJWHash，ELFHash，BKDRHash，DJBHash 和 APHash 等算法。而合适的 Hash 算法不仅能够节省时间，还可以节省空间开销。本文重点比较了其中的九个算法，从中选择一个在时间利用率和空间占用率综合性能最好的一个算法。

为了更好地比较各个算法之间的差异，我们采用了以下手段

- 只计算 Hash 算法的时间。由于 Hash 算法较快，如果包含了 printf 函数，则时间开销会不准确，计算的时间大部分都来源于 printf 函数，这对于计算 Hash 算法的误差特别大。

- 计算 Hash 算法计算 100000 次的时间。由于 Hash 算法很快，时间花费小，如果仅计算一次的时间，很难直接看出各个算法之间的误差，所以多次计算，可以明显地看出各个算法的差异。

- 重复循环 50 次。由于一次测量可能会有偏差值，不具有统计学意义。因此多次计算，减少实验偶然性带来的误差。

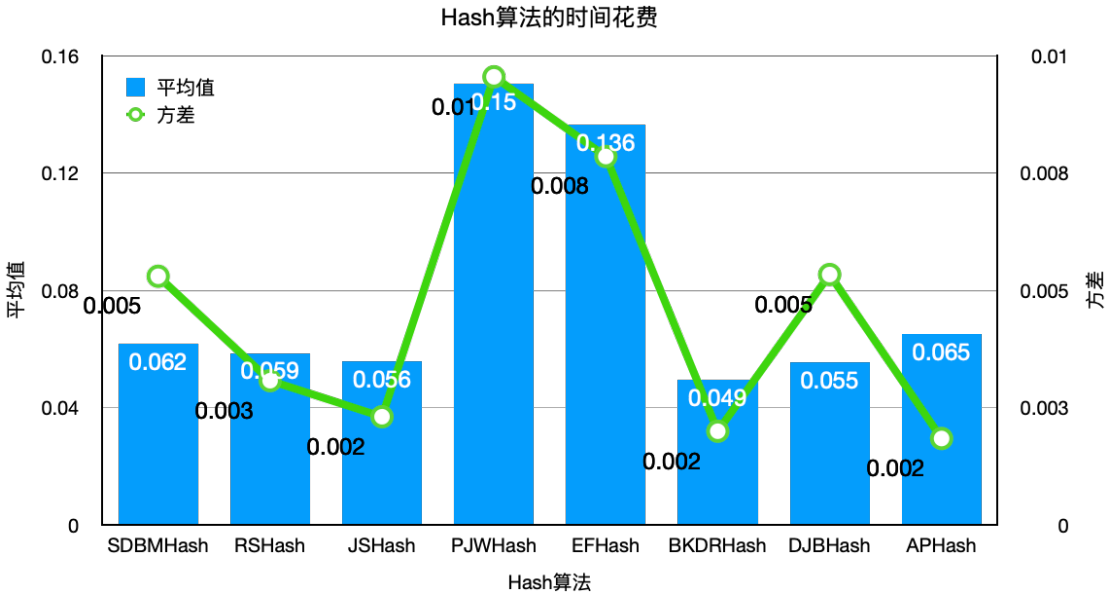


图 2 不同 Hash 算法执行的时间

得出来的结果如上图。蓝色柱体代表了各个 Hash 算法的实际计算时间的平均值，因此柱体越低代表消耗时间越少。而绿色的曲线表示 Hash 算法计算 50 次的时间方差，如果方差越低，则表示 Hash 算法消耗的时间变化不大。从结果图中，可以看出 BKDRHash 的时间利用最少，所消费的时间也最稳定。所以在时间方面，BKDRHash 的性能最优。

因此，本实验选择 BKDRHash 作为 Hash 的算法。

（2）粗过滤

当我们在给每一个 string.txt 中的字符都创建一个节点时，会浪费大量的时间和空间。因为 string.txt 文本当中只有一部分的词是我们想要得到的，如果能够首先过滤一些根本就不可能是我们想要的词组，那么搜索的次数可以大大减少。

我们创建了一个三维数组，给 string.txt 中出现的每行前三个字节构成的数组标为 1。则当通过 dict.txt 对比的时候，可以先判断 dict.txt 每行的前三个字节是否在 string.txt 中出现过。如果前三个字节在 string.txt 中出现过，则继续利用 Hash 算法精确搜索，否则放弃搜索。其数据代码如下：

```
/*粗过滤操作 */
char list[256][256][256] = { 0 }; //创建三维数组，首先初始化为 0
while (read_line(string_file, &str) == 1) {
        list[str[0] & 0xff][str[1] & 0xff][str[2] & 0xff] = (char)1;
};//把在 string_file 中出现的三个字的数组置为 1;
while (read_line(dict_file, &str)) {
        if (list[str[0] & 0xff][str[1] & 0xff][str[2] & 0xff]) {
        ...//如果前三个字在 string_file 中出现过，则继续搜索，否则跳过。
    }
};
```

下图所表示的是粗过滤前后，实验计算所花费的时间。根据下图可知，通过添加了粗过滤算法，过滤了 964953 行，实验时间从 1.14 微秒，减少到 1.03 微秒。可看到，通过粗过滤

操作，时间花费少了 10%。

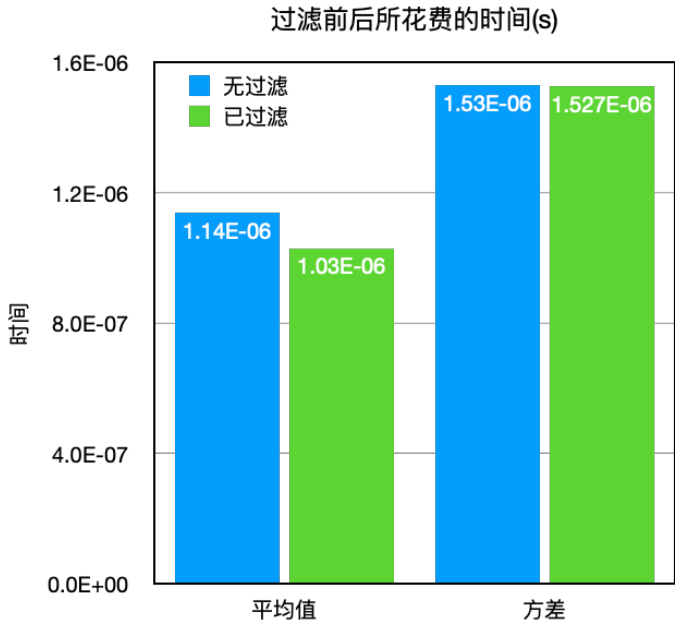


图 3 过滤前后所执行的时间

2.1.3 Hash 表的建立

利用 Hash 算法查找的核心是建立合理的 Hash 列表。 Hash 列表中的每一个节点包含了，内部的关键词和指向下一个节点的指针。在创建新的 Hash 列表时，首先要给节点设置空值。然后再根据给定的内容，计算其 Hash 值，赋值到 Hash 节点中，然后返回新的节点。其中包含了两种情况，一种是不发生冲突时，节点可以直接设置在原先处；若发生了冲突，节点需要找到这个位置的下一个节点，然后再进行赋值操作。其数据结构设计如下：

```
/* 构造 Hash 列表的结构 */
typedef struct HashNode {
    char str[100];
    struct HashNode* next;
}HashNode;
/* 构造连续的有 23767 个位置的 Hash 列表 */
HashNode hash_table[23767] = { "",NULL };   //主 Hash 表
/*创建 Hash 列表 */
HashNode* create_new_node(char* str) {
     return new_node;
};
/*根据 string.txt 文件创建 Hash 列表 */
while (read_line(string_file, &str) == 1) {
    list[str[0] & 0xff][str[1] & 0xff][str[2] & 0xff] = (char)1;   //此字符串的前三个字节出现，在三维数组中对应位置标记为 1
    int  hash_key = get_hash(str);
    if (!strlen(hash_table[hash_key].str)) {   //若 hash 表中对应位置的 node 为空，则把其文本内容设置为当前字符串
        strcpy(hash_table[hash_key].str, str);
```

```
        }
    else{   //若 hash 表中对应位置的 node 不为空，则创建新 node 放在 node 链的末尾
        HashNode* new_node = create_new_node(str);          //创建新的 node
        HashNode* node_pointer = &hash_table[hash_key];     //node 指针先指向 node 链第一个节点
        while (node_pointer->next) {    //循环遍历，直到找到最后一个节点
                node_pointer = node_pointer->next;
            }
        node_pointer->next = new_node;  //最后一个 node 的 next 指针指向新创建的 node
        }
    }
```

2.1.4 文件的操作

本实验涉及到了三个文件，其中两个为输入文件 dict.txt 和 string.txt，和一个输出文件 result.txt。dict.txt 文件是包含了 127 万个词的文件，而 string.txt 包含了 1.7 万个词。result.tx 文件中所包含的词包含了输入两个文件的交集。

本实验中，利用的文件操作涉及到了指针。其数据结构设计如下：

```
/* 文件指针的建立 */
typedef struct FileHandler {
    FILE* file;//
    char file_name[99];
    int open_status;
    int point;
    unsigned int buffer_size;
    char* buffer;
}FileHandler;
/*创建文件 */
FileHandler* open_file(char* file_name, char* open_type) {
return file_handler;
};
```

(1) 文件的读取

本实验利用缓存区读取文件。首先通过全局变量为缓存区设定大小。当缓存区内部填充满时，再进行读取。这样读取的次数就大大减少了。文件读取的数据结构如下：

```
/* 文件的读取 */
int read_line(FileHandler* file_handler, char** str) {
return 1; //读取成功
}
```

(2) 文件的写入

本实验中利用的写入操作有两个特点：第一点是直接通过对缓存区进行写入，当缓存区内部填充满时，再进行写入，直到完全把结果输出到文本中；第二点是没有将查找到的结果直接进行输出，而是加入了“第 n 行：”的字符串。文件写入的数据结构如下：

```
/* 文件的写入 */
int write_line(FileHandler* file_handler, char* str) {
    char target[20];//we have 6306 numbers
    static int num = 1;//numbers will change
    char* t = target;
    sprintf(target,"%d",num);
return 0; //写入成功
};
```

2.1.5 实验分析

本实验能够在极短的时间内，获得完整且精确的结果，有以下三个原因。

（1） 直接对内存的读取与写入。

文件的读取与写入是本实验的重点。输入的数据以及输出的数据均是庞大的。

在输入方面，如果每次读取一行数据，则经过初过滤后，仍然需要读取 dict.txt 文件中的 305643 次。所以本实验采取，为内存设定一定空间的大小（本实验利用全局变量 FILE_BUFFER_SIZE 表达内存空间的大小）每次从里面读取内存大小的字符串，那么只需要读取（dict.txt 文件大小/FILE_BUFFER_SIZE +1）次，这样可节省时间。

在输出方面，如果每次输出一行数据，那么需要执行写入操作六千多次。如果把需要写入的数据全部缓存在内存中，通过对内存数据的修改，对文本执行写入操作，在时间消耗方面也会大大地减少。

（2） 粗过滤

在预实验部分中，我们对实验进行了粗过滤操作。有 305629 行在粗过滤之后留下，经过粗过滤后，时间花费率减少了 10%。

（3） 合适的 Hash 算法

在预实验部分中，我们在多种常见的 Hash 算法中，通过计算寻找最适合本实验的 Hash 算法。

2.2 Bloom 过滤

Bloom 过滤器是一串很长的二进制向量和一系列随机 Hash 函数。每个字符串通过 k 个 Hash 算法，获得 k 个 Hash 值。在一串很长的很长的二进制向量中的第 k 个位置置为 1，其余位置置为 0；因此，在查找的时候，通过计算该字符串的 k 个 Hash 值，找到它存储的位置，并判断这些存储位置的内容是 0，还是 1；如果这 k 个位置的内容都为 1，则该字符串是我们需要的目标，并进行输出。

2.2.1 参数的计算

Bloom 过滤器对于海量数据的查找，依赖于合适参数的选取。在进行实验前，需要设定好目前的容错率 f 以及待存储的字符串个数 n，以此来计算 Hash 函数的个数 k，以及需要开辟的存储空间的位数 m。所以首先设定变量，容错率 f 和待存储的字符串个数 n；再利用函数计算出需要的存储空间的位数 m 和 Hash 函数的个数 k。其数据结构如下.

```
 /* 设定变量*/
```

```
float fp=0.0000001;//期望的错误率
int n=1270574;//待存储的字符串个数
 /* 计算所需要的参数*/
 int num_m(float f, int n){//需要开辟的存储空间的位数
 };
 int num_k(int m, int n){//Hash 函数的个数 K
 };
```

2.2.2 创建连续的存储空间

根据上一步得出的空间的大小，利用 malloc 函数创建一个连续存储的空间。而 malloc 函数的参数是字节，因此我们要把它换算成位字符，并把其中包含的数据都置为 0。实际数据结构如下所示：

```
 /* 创建空间 */
char* create_m(int m){
    m=m/8+1;
    char* new_m = (char*)malloc(m);
    for(int i=0;i<m;i++)
    {
        *(new_m+i)=0;
    }
    return new_m;
}
```

2.2.3 Bloom 过滤器的构造

首先构建一个 Bloom 过滤器的操作函数。输入参数有，存储空间的首位置*m，每个关键词的 Hash 值，整数型字符 typ。给入整数型字符 typ，若 typ 为 1，那么进行写入操作。否则，进行读取操作。输出为整数型字符 1 或者 0；1 表示查找成功；否则查找失败。

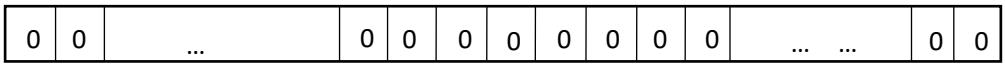
首先，因为 char 所包含的字符是一个字节，包含了八位，而我们在读取和写入操作时，是以位为单位操作的。在进行写入操作时，我们设置了一个八位的字符型变量 b，用二进制表示为 10,000,000，然后对他进行位操作，即根据计算的 Hash 值 h，把 1 向右移动 h 位。然后把内存中的那一位置 1。在进行读取操作时，同样设置一个八位的字符型变量 b，并根据 Hash 值 h 对 b 进行位操作。然后内存的部分与 b 进行与操作。如果为 0，那么内存中的那个部分为 0，则直接返回 0，该字符不符合我们的预期。实际数据结构如下所示：

```
 /* 构造 Bloom 过滤器 */
int manipulate_m(char*m,unsigned int h,int typ)
{
    if(typ==1)//写
    {
    unsigned char b=(char)128;//1000,0000 永远只有 1 位是 1
        b=b>>(h%8);
        *(m+h/8)=*(m+h/8)|b;
    }
    if(typ==0)//读，判断是 0 是 1;
```

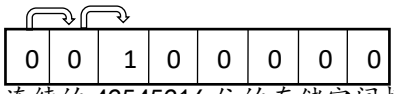
```
    {
        unsigned char b=(char)128;
        b=b>>(h%8);//0001,0000 假如 m+k/8 为 a,若 a 为 0101，0000。
b=0001,0000;0001,0000
        if(!(*(m+h/8)&b))
            return 0;//0 则跳过这一行
    }
    return 1;
}
```

其具体操作如图所示：

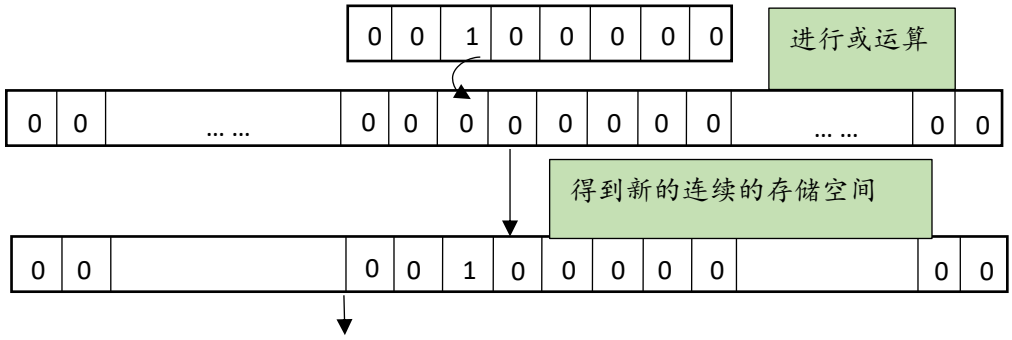
1、把连续的 42545216 位的存储空间全部置 0



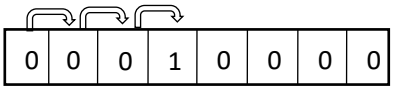
2、根据 dict.txt 文本中每行字符的 Hash 值 h，使 b 位左移 h%8（本图假设 h%8=2）



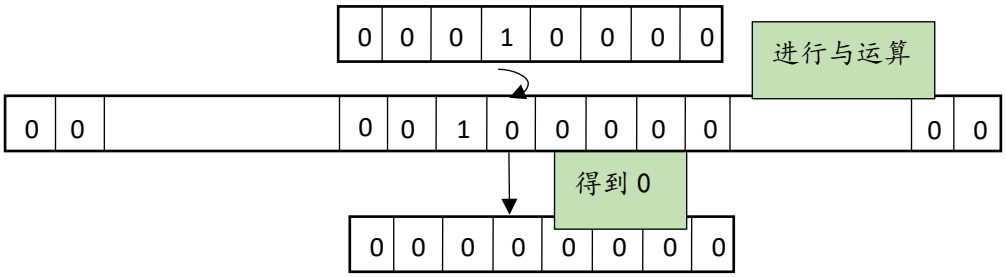
3、把连续的 42545216 位的存储空间根据 dict.txt 文本中每行字符的 Hash 值 h 置 1。



4、根据 string.txt 文本中每行字符的 Hash 值 h，使 b 位左移 h%8（本图假设 h%8=3）



5、把 string.txt 文本中每行字符的 Hash 值 h 与新的连续的存储空间比较



6、当得出的结果为 1 时，可以判断，该字符不是我们想要的值。

2.2.4 Bloom 过滤

在利用 Bloom 过滤器算法时，首先读取 dict.txt 文件中每行的字符，然后计算该字符的

Hash 值，每个字符计算 24 次的 Hash 值，然后利用上节的函数把 Hash 值在的位置置入 1。其次读取 string.txt 文件中每行的字符，计算该字符的 Hash 值 24 次，每次都判断 Hash 值的位置是否为 0，若为 0，则跳过，进行 string.txt 文件中下一行的字符。

```
 /* 对存储空间进行 Bloom 过滤 */
while (read_line(dict_file, &str) == 1) {
        for (int seed = 0; seed < 24; seed++) {//一行计算 24 次 Hash 值
            MurmurHash3_x64_128(str, strlen(str), seed, out);//计算哈希
值
            manipulate_m(new_m, (unsigned int)out[1] % m, 1);//置 0 置 1
        }
    }
    /* 根据 dict.txt 建立内存结束 */
    /* 打开 string.txt，根据 string.txt 查询 */
while (read_line(string_file, &str) == 1){
        int t=0;
        for(int seed=0;seed<24;seed++){
            if(t==0){
            MurmurHash3_x64_128(str, strlen(str),seed,out);
            if(!(manipulate_m(new_m,(unsigned int)out[1]%m,0)))
                t=1;
            }
        }
        if(t==0)
            write_line(result_file, str);
    }
```

三 实验分工

按照上述的系统设计，实验分工如下：

高垚实现了第一部分的粗过滤，Hash 表的创建与赋值，构造文件结构与文件的读取，以及文件的关闭。第二部分的对连续存储空间的写入与读取。

杨恩实现了第一部分的计算 Hash 值与寻找合适的 Hash 函数，以及文件的写入。第二部分的计算参数（包括所需 Hash 函数的个数 k，所需存储空间的位数 m），创建存储空间。

四 实验结果与分析

4. 1 Hash 链表

经过实验，程序共爬取了 1270574 行字符，初步过滤后，剩余 305629 行，发生冲突 1447 次。运行时间为 0.131834，内存峰值为 5.9M。最终得到 3606 行字符，程序运行结果如下图 4。

```
runtime:0.131834      sring_march:6306
sh: pause: command not found
Program ended with exit code: 0
```

图 4　1.1 实验的输出

```
Error rate:0.000100 string match:6315
Error rate:0.000010 string match:6308
Error rate:0.000001 string match:6308
runtime: 11.582795 s
sh: pause: command not found
Program ended with exit code: 0
```

图 5　1.2 实验的输出

4.2 Bloom 过滤器

经过实验，程序共爬取了 1270574 行字符，三次运行时间分别为 3.984542s,3.696368s,3.777416s，总时间为 11.582795s。内存峰值为 17.2M。三次不同的容错率中，容错率为 0.0001 时，得到 6315 行字符；容错率为 0.00001 时，得到 6308 行字符；容错率为 0.000001 时，得到 6308 行字符，程序运行结果如图 5。

五 实验问题

5.1 Hash 链表

（1）报错提醒 Undefined symbol: BKDRHash(char*）

解决方法：头文件（.h）中不要引用任何.c 程序中的函数。

（2）报错提醒 conflicting types for “”

解决方法：头文件（.h）中与.c 程序中的函数的声明不一致，需要改用成一致的声明。

5.2 Bloom 过滤器

（1）如何通过字节实现对位的操作

解决方法：本实验通过位操作改变一个字节中 1 的位置，并与连续的储存空间进行或运算或者与运算，实现对位的改变，来达到 Bloom 过滤的目的。

（2）MurmurHash 的输出

解决方法：MurmurHash 的输出是由两个 16 位的数字组成的 32 位的字，然而我们只需要其中的 16 个数字，因此我们在每次选取时，只利用了前 16 位作为我们对 Hash 值的被余数，这样开辟出的存储空间也不会过大。

六 实验结论

综上所述，实验程序能够对海量数据进行筛选和搜索，并能在较快的时间内完成计算。

实验 1.1 中，可以通过全局变量，设置缓存空间的大小，以此来平衡时间和空间的利用率。所采用的算法能够建立合理大小的 Hash 链表，并且能够有效地解决冲突，快速地搜索结果。其计算结果符合实验要求。

实验 1.2 中，通过对容错率的调整，循环三次实验，以此得到三种不同的输出结果。本实验采取 MurmurHash 函数，把种子 seed 设置为 1~k 个不同的值，就得到不同的 Hash 值。本部分涉及到的参数较多，而参数与参数之间的联系又紧密，因此本实验的困难在于，短时间内计算并处理大量数据。而本实验的执行时间在十秒，能够顺利完成实验目标，符合实验预期结果。