

# 计算机应用编程实验

## Hash技术字符串检索

---

熊永平@网络技术研究院

ypxiong@bupt.edu.cn

教三楼233

2019.8

# 实验背景

- 一些实际问题
  - ▣ 如何实现在搜索引擎输入框自动提示？
  - ▣ 一个查询串的重复度越高，说明查询它的用户越多，也就越热门。大型搜索引擎每小时有几十亿个查询请求，如何统计搜索引擎最热门前100个查询？
  - ▣ 在实现一个编辑器时，如何对输入的单词进行拼写检查？
  - ▣ GFW每次google中输入\*\*词后，如何在\*\*名单中查找并reset？
  - ▣ 搜索引擎的网络爬虫，每天要爬取几十亿网页，哪些URL是爬过的？
  - ▣ 收到一封邮件后，能否快速在几亿个垃圾邮件黑名单地址里快速判断发件人是否在黑名单里面？
  - ▣ 检测引擎中包含几千万条特征字符串规则，如何在10G网络流量环境下检测网络流中的恶意软件特征？

# 海量字符串检索

- 问题
  - 在给定的海量个数的字符串中查找特定的字符串
- 挑战
  - 实际需求
    - 几亿规模
  - 数据量大
    - 200,000,000量级
  - 外存便宜
    - 存储成本低
  - 内存不够大?
    - $200,000,000 * 40\text{bytes} = 8000,000,000\text{bytes} = 8\text{G}$

# 版本一：数组实现array\_search

- 设计一个结构体数组
- 存储字符串
- 最大的空间开销
- 最多的计算开销

# 版本二：哈希表实现 hashtable\_search

- 思想

- ▣ 根据设定的哈希函数  $H(key)$  和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集（区间）上，并以关键字在地址集中的“象”作为相应记录在表中的存储位置，如此构造所得的查找表称之为“哈希表”。
  - ▣ 这一过程称为哈希造表或者散列，所得的存储位置成为哈希地址或者散列地址

- 冲突处理方法：拉链法

- ▣ 所有关键字为同义词的记录存储在同一线性链表中。
  - ▣ 凡是哈希地址为  $i$  的记录都插入到头指针为  $ChainHash[i]$  的链表中。



# 核心结构

例如：关键字集合

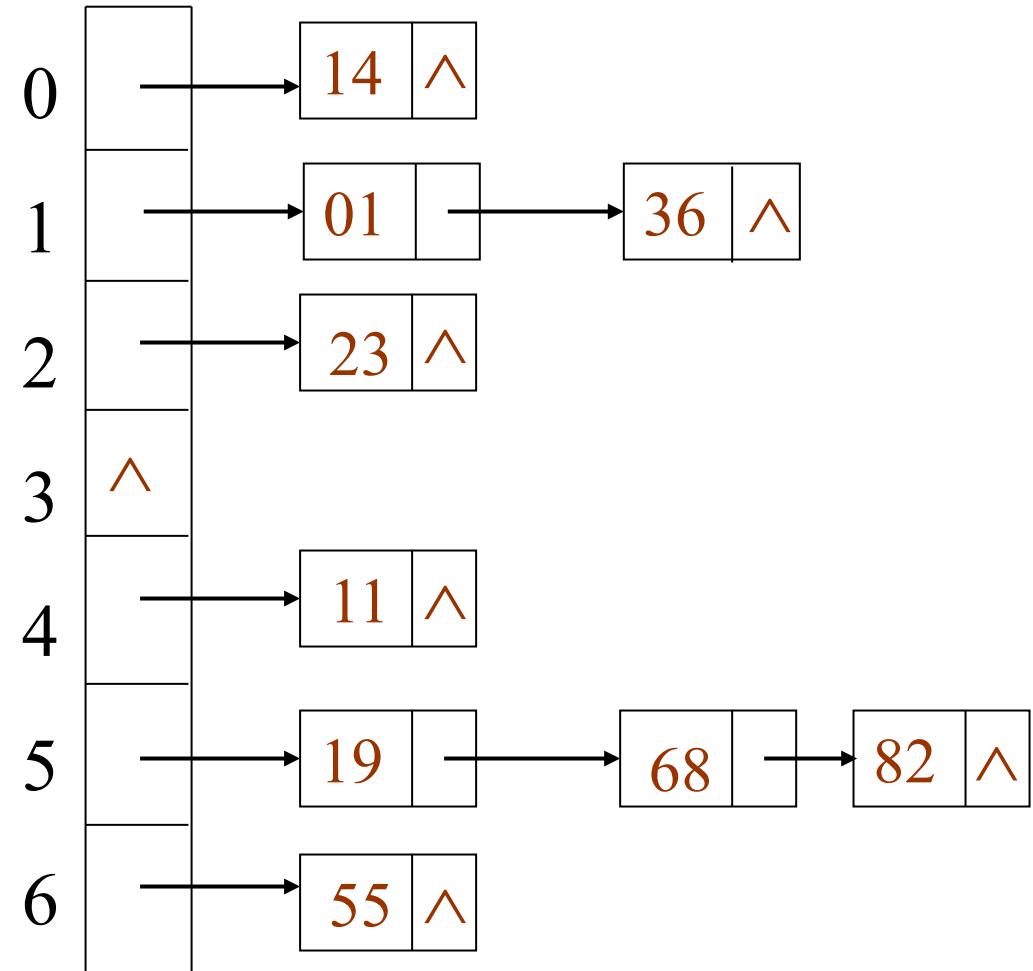
```
{ 19 01 , 23 14  
55 , 68 , 11 82 ,  
36 }
```

哈希函数为

$$H(key) = key \bmod 7$$

采用链地址法来构造哈希表。

$$ASL = (6 \times 1 + 2 \times 2 + 3 \times 1) / 9 = 13 / 9$$



# 哈希表分析

- **查找过程**
  - 利用线性表存储集合元素
  - 利用Hash函数计算元素对应的地址entry，并在对应的区域存储该元素
  - 实际查找通过先hash计算entry，再遍历链表匹配元素是否相同（字符串匹配，必须逐个字符比较）
- **问题**
  - 解决冲突：选择合适的hash函数
  - 合适的哈希表大小

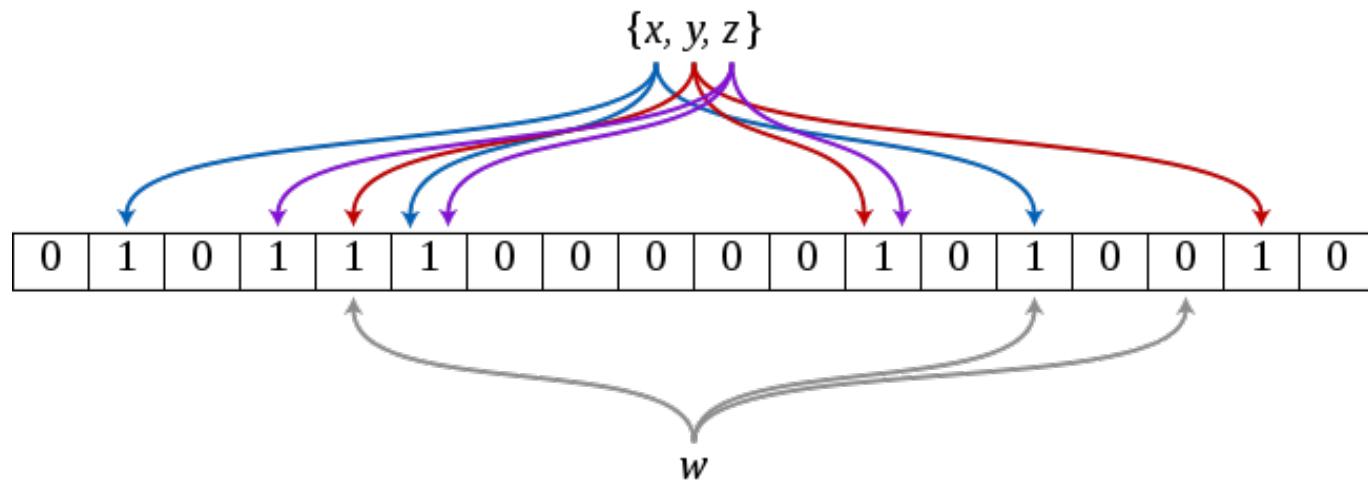
# 版本三： Bloom Filter实现

- 背景
  - ▣ 1970年Burton Bloom论文《Space/time trade-offs in hash coding with errors》中提出
  - ▣ 吴军《数学之美》中称之为布隆过滤器
- 概念
  - ▣ **一个很长的二进制向量和一系列随机hash函数**
  - ▣ 用于检索一个元素是否在一个集合中
  - ▣ 准确率换空间思想延伸
- 优点
  - ▣ 空间效率和查询时间都远超过一般的算法
- 缺点
  - ▣ 有一定的误识别率和删除困难

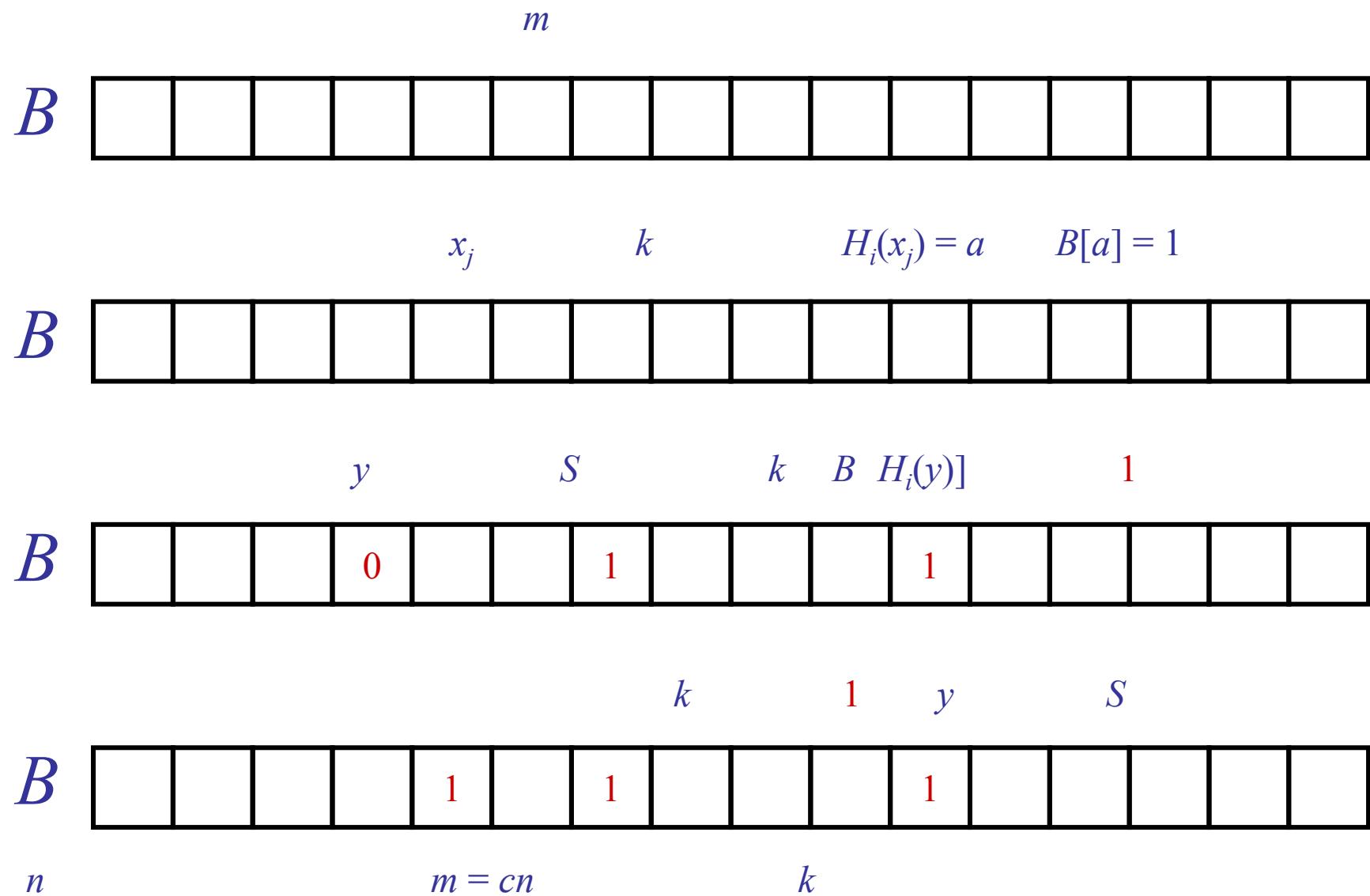
# Bloom Filter构建

- 定义

- 将n个元素集合 $S=\{x_1, x_2, \dots, x_n\}$
- 一个包含m位的二进制位数组存储
- K个相互独立的哈希函数映射到 $\{1, \dots, m\}$ 的范围
- S集合中的每个元素用k个hash函数映射到， $\{1, \dots, m\}$ 范围内，将相应的位置为1



# Bloom Filter原理



# 错误率估计

- 当集合 $S=\{x_1, x_2, \dots, x_n\}$ 的所有元素都被 $k$ 个哈希函数映射到 $m$ 位的位数组中时，位数组中某一位是0的概率是：

$$q = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

- 构建BF后某一位为1的概率就是 $1-q$
- False positive rate就是一个不在集合中的字符串经过K次Hash后对应的位都为1的概率：

$$f = (1 - e^{-kn/m})^k = (1 - q)^k$$

# 哈希函数个数k

- Hash函数的个数k不是越大越好， k如何取， 才能使得f最小

$$f = (1 - q)^k = (1 - e^{-kn/m})^k$$

$$\Rightarrow f = \exp(k \ln(1 - e^{-kn/m}))$$

令  $g = k \ln(1 - e^{-kn/m})$  #### 所以当g最小时， f最小

$$\Rightarrow g = -\frac{m}{n} \ln(q) \ln(1 - q)$$

- 所以，  $q=1/2$  时错误率最小， 也就是让一半的位空着

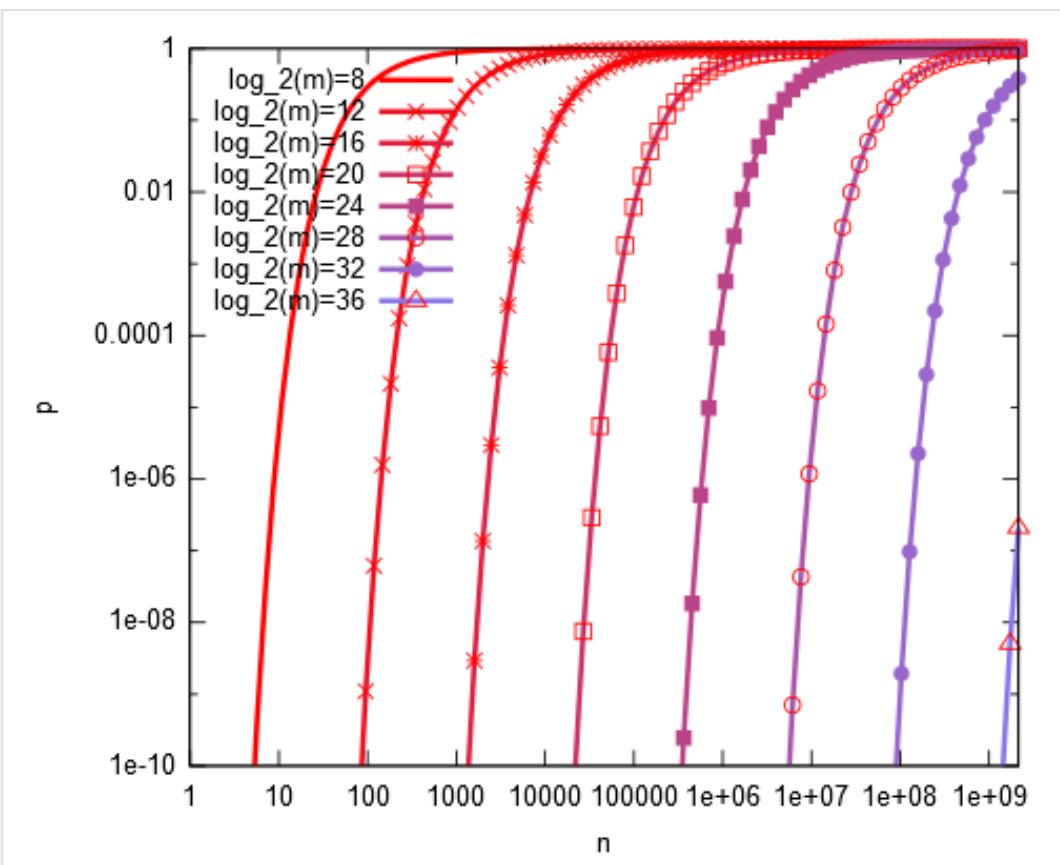
$$k = \ln 2 * \frac{m}{n} \approx 0.693 * \frac{m}{n}$$

# 需要的存储空间m

- 推导结果公式: (假设, 允许的false rate为 $p$ )

$$m = n * \log_2 e * \log_2(1/p)$$

$$\approx n * 1.44 * \log_2(1/p)$$



# 参数取值举例

目标：降低错误率

通过设计布隆过滤器的4个参数来实现

f: 期望的错误率	---- 0.001
n: 待存储的字符串个数	---- 1500w
m: 需开辟的存储空间位数	
k: 哈希函数的个数	

$$m = n * 1.44 * \log_2(1 / f')$$

$$k = 0.693 * m / n$$

$$m = 215260940 \text{ bit} = 25M$$

$$K = 10$$

# 测试与参数选择

- 进行3组实验，每组取5个N
  - 取 $FP1 = 0.01\%$ ,  $N=[50W, 100W, 300W, 500W, 1000W]$
  - 取 $FP2 = 0.001\%$ ,  $N=[50W, 100W, 300W, 500W, 1000W]$
  - 取 $FP3 = 0.00001\%$ ,  $N=[50W, 100W, 300W, 500W, 1000W]$

N	(Vector size)m			内存			(Hash num) k			X		
	FP1	FP2	FP3	FP1	FP2	FP3	FP1	FP2	FP3	FP1	FP2	FP3
50W	958W	1198 W	1677W	1M	1M	1M	13	17	23	37091	3691	38
100W	1917 W	2396 W	3355W	2M	2M	3M	13	17	23	36958	3770	47
300W	5751 W	7188 W	10064 W	6M	8M	11M	13	17	23	36585	3689	38
500W	9585 W	11981 W	16773 W	11 M	14M	19M	13	17	23	36569	3701	45
1000W	19170 W	23962W	33547 W	22 M	28M	39M	13	17	23	36533	3552	41

# Hash算法选择

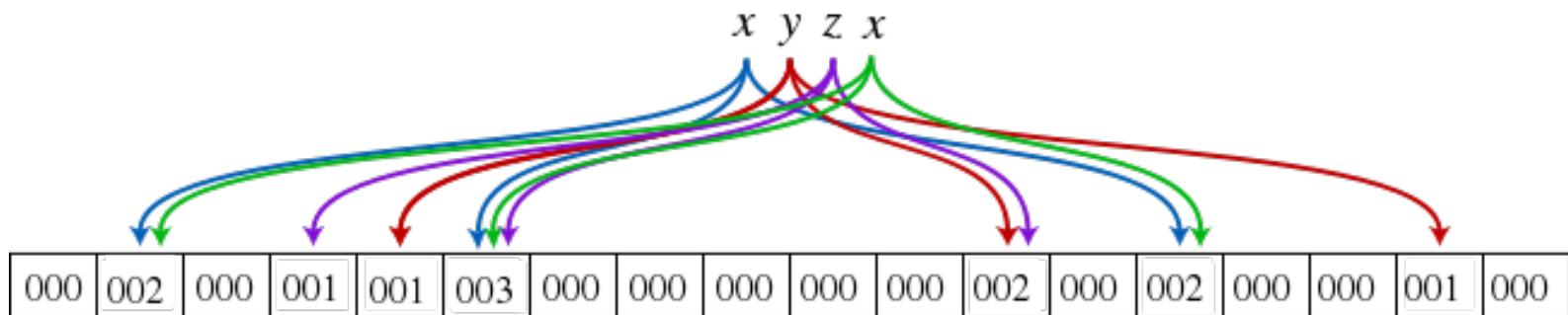
- K个hash越相互独立效果越好
- 常用的HASH算法
  - <http://www.partow.net/programming/hashfunctions/>
  - unsigned int RSHash (char\* str, unsigned int len);
  - unsigned int JSHash (char\* str, unsigned int len);
  - unsigned int PJWHash (char\* str, unsigned int len);
  - unsigned int ELFHash (char\* str, unsigned int len);
  - unsigned int BKDRHash(char\* str, unsigned int len);
  - unsigned int SDBMHash(char\* str, unsigned int len);
  - unsigned int DJBHash (char\* str, unsigned int len);
  - unsigned int DEKHash (char\* str, unsigned int len);
  - unsigned int BPHash (char\* str, unsigned int len);
  - unsigned int FNVHash (char\* str, unsigned int len);
  - unsigned int APHash (char\* str, unsigned int len);

# Hash算法不够

- Hash算法特性
  - 抗冲突性(collision-resistant), 即在统计上无法产生2个散列值相同的预映射。
  - 映射分布均匀性和差分分布均匀性
    - 散列结果中, 为0的bit和为1的bit, 其总数应该大致相等;
- 超级HASH算法
  - The requirement of designing  $k$  different independent hash functions can be prohibitive for large  $k$ . For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields.
  - MurmurHash
  - 官网: <https://sites.google.com/site/murmurhash/>

# 扩展：计数型Bloom Filter

- 标准BF
  - 无法删除元素
  - $m$  长数组，每 unit 用 1 位表示，只能表示  $[0, 1]$
- 如果每个 unit 用多位表示，如 3 位，则可以表示  $[0, 1, \dots, 7]$



# 程序要求

- 分别实现三个版本，程序分别命名为
  - array\_search、 hashtable\_search和bf\_search
- 输入数据
  - 词典串pattern.txt: 127万个
  - 待匹配的98万个字符串: words.txt
- 实验结果**result.txt**
  - 在模式串中的输出yes，不在就输出no
    - Keyword1 yes
    - Keyword2 no
    - 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

# 内存统计方法

- 数组或树都采用动态内存分配，不要用栈内存分配
- 用如下代码分配内存

```
void* bupt_malloc(size_t size){  
    if (size <= 0) {  
        return NULL;  
    }  
  
    global_stats.mem += size;  
    return malloc(size);  
}
```

# 字符比较统计方法

- 字符比较统计
- 设计一个字符比较函数

```
int byte_cmp(char a, char b)
```

```
{
```

```
    Global_stats.cmpnum++;
```

```
    Return a-b;
```

```
}
```

# 报告要求

- 实验报告
  - 主要数据结构和流程
  - 实验过程
  - 遇到的问题
  - 结果指标: cpu 内存 准确率等
  - 结论和总结

# 打分标准

- 每个版本单独打分
  - 每个实现版本的打分依据
  - 程序代码：权重80%
    - 实现逻辑能正常运行： 50
    - 检索命中串个数正确：  $x/total*20$
    - 内存开销  $(1- (x-min)/(max-min))*15$
    - 字符/字节比较次数  $(1- (x-min)/(max-min))*15$
  - 实验报告：权重20%
    - 逻辑清晰、写作规范： 40
    - 图表丰富： 20
    - 内容完整： 40
  - 演讲加分： 20分
    - 课堂讲解自己的实现

# 进度要求

- 实验进度
  - W1: 实现数组和哈希表检索
  - W2: 实现bloomfilter检索
  - W3: 编写实验报告并进行优化

# THE END