计算机应用编程 实验一

大规模字符串检索

熊永平@网络技术研究院

ypxiong@bupt.edu.cn

周一10:10-12:00@3-134

2015.9.21

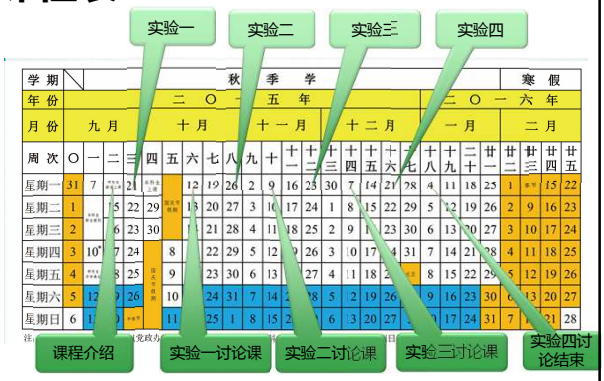
课程相关

- 课程邮箱
  - user/pwd:buptprog@163.com/bupt112358
  - 课件和实验文件打包

分组和讨论课

- 总人数113人
  - 分为10组
  - 每组11人，讨论技术方案和调研
  - 每组分为若干实验单元，每单元2-3人，评分单元
- 讨论课
  - 11组每组派人做报告
  - 由组长安排
  - 组长名单：周姿宇、郭享、李屹、周悦越、闭蓉、汤敬浩、杜炜、李放、韩松月、李俊

课程表



4

实验一：海量字符串查找

5

- 1、实验任务
- 2、核心算法之Trie树
- 3、核心算法之BloomFilter
- 4、代码提交说明

6

实验背景

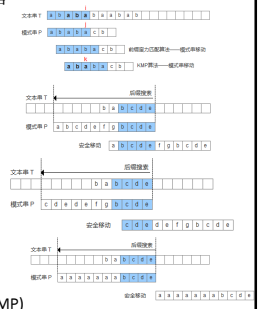
- 一些实际问题
  - 如何实现在搜索引擎输入框自动提示？
  - 一个查询串的重复度越高，说明查询它的用户越多，也就越热门。大型搜索引擎每小时有几十亿个查询请求，如何统计搜索引擎最热门前100个查询？
  - 在实现一个编辑器时，如何对输入的单词进行拼写检查？
  - GFW每次google中输入**词后，如何在**名单中查找并reset？
  - 搜索引擎的网络爬虫，每天要爬取几十亿网页，哪些URL是爬过的？
  - 收到一封邮件后，能否快速在几亿个垃圾邮件黑名单地址里快速判断发件人是否在黑名单里面？
  - 检测引擎中包含几千万条特征字符串规则，如何在10G网络流量环境下检测网络流中的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

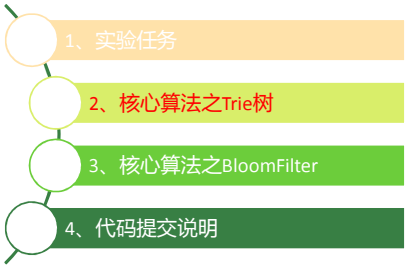
实验一：海量字符串查找

- 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*40bytes=8000,000,000bytes=8G

根据以往的学习思路

- 字符串集合转换成一个大字符串
- 字符串匹配
  - Brute Force
    - Strstr()
  - KMP算法
    - 前缀匹配算法
  - Boyer-Moore
    - 后缀匹配算法
- 问题
  - 存储空间
- 复杂度
  - best case O(n/m)
  - worst case *still* O(nm)(BM)、O(n)(KMP)





Trie树

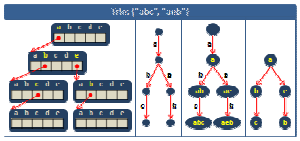
- 基于关键码分解的数据结构，叫作Trie结构（Trie树）
- 基于两个原则
  - 有一个固定的关键码集合
  - 对于结点的分层标记
- Trie树
  - 又称单词查找树、字典树，是一种树形结构，是一种用于快速检索的多叉树结构
- 典型应用
  - 统计和排序大量的字符串
  - 文本词频统计和文本检索
  - 优点：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

介绍

- 不依赖于关键码的插入顺序
  - 树的深度受到关键码精度的影响
- 最坏的情况下，深度等于存储关键码所需要的位数
  - 例如，如果关键码是0到255之间的整数，关键码的精度就是8个二进制位。
  - 如果有两个关键码：10000010和10000011，它们的前面7位都是相同的
  - 所以直到第8次划分才能将这两个关键码分开
  - 这样的搜索树深度也为8，但这是最坏的情况
  - 与B+树一样，基于关键码空间分解的树结构，其内部结点仅作为占位符引导检索过程，数据纪录只存储在叶结点中

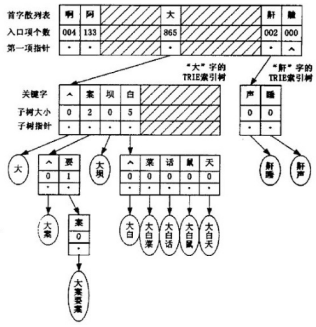
Trie应用1—英文词典

- 存储字典里面的单词
  - 英文的单词是有26个字母组成的（简单起见，忽略大小写），
  - 英文字符树每一个内部结点都有26个子结点
  - 树的高度为最长字符串长度
- 应用
  - 把要查找的关键词看作一个字符序列，并根据构成关键词字符的先后顺序构造用于检索的树结构：



Trie树应用2--中文词典

- 存储格式
  - 首字散列表
  - TRIE索引树结点包含
    - 按关键字排序的数组
    - 关键字(2字节)：单一汉字
    - 子树大小(2字节)：以从根结点到当前单元的关键字组成的子串为前缀的词的个数。
    - 子树指针(4字节)：子树大小非0时，指针指向子树，否则指向叶子。



Trie数据结构定义

ASCII 一共有 128 个不同的字符，所以一个节点需要一个 128 个元素的数组

```
1. struct TrieNode
2. {
3.     TrieNode* l[128];
4.     int n;
5.
6.     TrieNode()
7.     {
8.         memset(l, 0, sizeof(TrieNode*) * 128);
9.         n = 0;
10.    }
11. } *root = new TrieNode();
```

Trie查找

- 查找
  - 在Trie树上进行检索总是始于根结点。
  - 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索。
  - 在某个结点处相应的子树上，取得要查找关键词的第二个字母，并进一步选择对应的子树进行检索。
  - 键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找

```
1. bool lookup(char* s)
2. {
3.     TrieNode* p = root;
4.     for (; *s; ++s)
5.     {
6.         if (!p->l[*s]) return false;
7.         p = p->l[*s];
8.     }
9.     return p->n > 0;
10. }
```

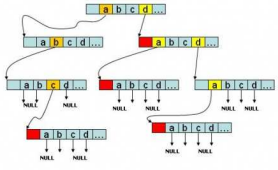
Trie树的插入

- 插入
  - 首先根据插入纪录的关键码找到需要插入的结点位置
  - 如果该结点是叶结点，那么就将为其分裂出两个子结点，分别存储这个纪录和以前的那个纪录
  - 如果是内部结点，则在那个分支上应该是空的，所以直接为该分支建立一个新的叶结点即可

```
1. void add(char* s)
2. {
3.     TrieNode* p = root;
4.     for (; *s; ++s)
5.     {
6.         if (!p->l[*s]) p->l[*s] = new TrieNode();
7.         p = p->l[*s];
8.     }
9.     p->n++;
10. }
```

Trie查找实例

- trie树中保存了abc、d、da、dda四个单词



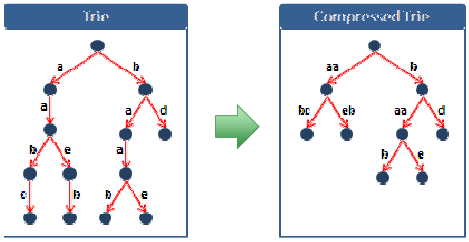
Trie查找效率分析

- 在trie树中查找一个关键字的时间和树中包含的结点数无关，而取决于组成关键字的字符数。（对比：二叉查找树的查找时间和树中的结点数有关$O(\log_2^n)$。）
- 如果要查找的关键字可以分解成字符序列且不是很长，利用trie树查找速度优于二叉查找树。
- 若关键字长度最大是5，则利用trie树，利用5次比较可以从$26^5=11881376$个可能的关键字中检索出指定的关键字。而利用二叉查找树至少要进行$\log_2 26^5=23.5$次比较。

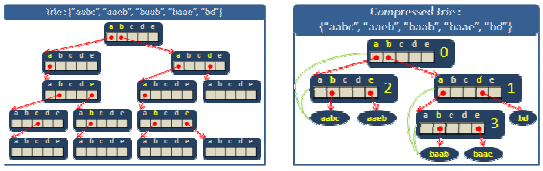
Trie树特性

- 核心思想
  - 空间换时间
  - 利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的
- 优点
  - 查找效率高，与词表长度无关
    - Trie树的查找效率只与关键词长度有关
  - 索引的插入，合并速度快
    - 注意，直接遍历Trie树需要搜索大量的无效节点
    - 可以把数据存在一个数组中，Trie只保存指针
    - 这样合并时，只需要对数组进行遍历即可
- 缺点
  - 内存空间消耗大
    - 如果是完全m叉树，节点数指数级增长
    - 不可达上限：词数×字符序列长度×字符集大小×指针长度
    - 例如：20000×6×256×4＝120M
  - 实现较复杂

Trie优化（1）--Compressed Trie



Compressed Trie实例



- 去掉没有分岔、连成直线的节点。
- 每个节点增加一个数字，纪录是第几个字符开始分岔。
- 去掉节点之后，字符串信息不完整，在树叶里储存完整字符串。
- 每个节点增加一个指针，纪录要参考哪一个叶子的字符串开头。

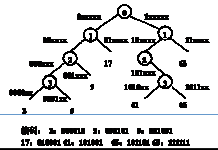
Trie优化（2）– PAT Trie

- Trie结构缺点
  - Trie结构显然也不是平衡的
  - 存取英文单词时，显然t子树下的分支比z子树下的分支多很多
  - 26个分支因子使得树的结构过于庞大，检索不便
- PATRICIA Trie（"Practical Algorithm To Retrieve Information Coded In Alphanumeric"）
  - 关键码二进制形式存储
  - 根据关键码每个二进制位的编码来划分
  - 是对整个关键码大小范围的划分

每个内部结点都代表一个位的比较，必然产生两个子结点，所以它是一个满二叉树，进行一次检索，最多只需要关键码位数次的比较即可。

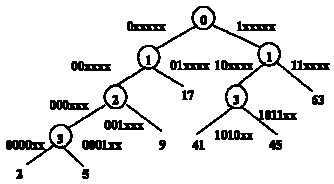
PATRICIA应用举例

- 举例（2、5、9、17、41、45、63）
  - 因为最大的数是63，用6位二进制表示即可
  - 每个结点都有一个标号，表示它是比较第几位，然后根据那一位是0还是1来划分左右两个子树
  - 标号为2的结点的右子树一定是编码形式为xx1xxx，（x表示该位或0或1，标号为2说明比较第2位）
  - 在图中检索5的话，5的编码为000101
  - 首先我们比较第0位，从而进入左子树，然后在第1位仍然是0，还是进入左子树，在第2位还是0，仍进入左子树，第3位变成了1，从而进入右子树，就找到了位于叶结点的数字5



PATRICIA压缩优化

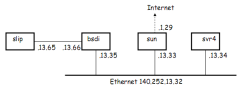
- 优化
  - 在区分2和5、41和45时，第3个二进制位的比较不能区别它们，可以将它省略，得到一棵更为简洁的树 。



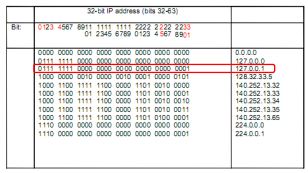
编码：2：000010　5：000101　9：001001
17：010001　41：101001　45：101101 63：111111

PATRICIA 应用：路由表查找

- Example Net
- Routing Table

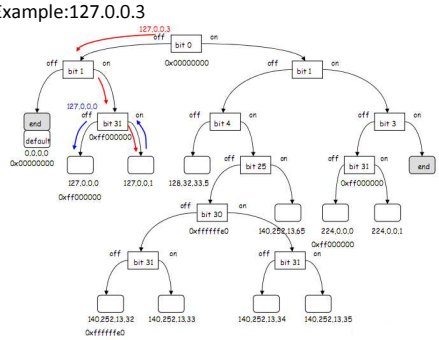


Destination	Gateway	Flags	Ref	Use	Interface
default	140.252.13.33	UGS	0	3	le0
127	127.0.0.1	UGSR	0	2	le0
127.0.0.1	127.0.0.1	UH	1	55	le0
128.32.33.5	140.252.13.33	UGHR	2	16	le0
140.252.13.32	link#1	UC	0	0	le0
140.252.13.33	8:0:20:3:f6:42	UHL	11	55146	le0
140.252.13.34	0:0:c0:c2:9b:26	UHL	0	3	le0
140.252.13.35	0:0:c0:6f:2d:40	UHL	1	12	le0
140.252.13.65	140.252.13.66	UH	0	41	sl0
224	link#1	UC	0	0	le0
224.0.0.1	link#1	UHL	0	5	le0



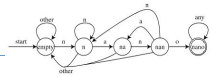
PATRICIA应用：路由表查找

- Example:127.0.0.3



Trie优化（3）--Trie 图

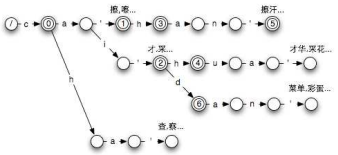
- 背景
  - 反病毒引擎一般都用字符串作为恶意软件特征，在大规模网络流量中如何判断是否出现恶意软件？
- 问题
  - 给一个很长很长的母串长度为n，然后给m个小模式串。
  - 求这m个模式串里边有多少个是母串的字串。
- Trie图
  - 一个确定性有限自动机DFA，由Trie树为基础构造
  - 每点都是一个状态，状态之间的转换用有向边来表示
  - 用母串作为DFA的输入，在DFA 上行走，走到终止节点，就意味着匹配了相应的模式串



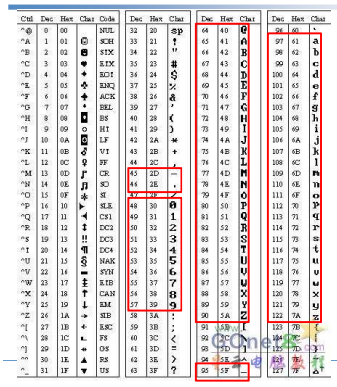
Trie图实例—输入法

- 举例：
  - 音节表

string[] _spellMusicCode = new string[]{ “a”,“ai”, “an”, “ang”, “ao”, “ba”, “bai”, “ban”, “bang”, “bao”, “bei”, “ben”, “beng”, “bi”, “bian”, “biao”, “bie”, “bin”, “bing”, “bo”,“bu”,“ca”, “cai”, “can”, “cang”, “cao”, “ce”, “ceng”, “cha”, “chai”, “chan”, “chang”, “chao”, “che”, “chen”, “cheng”, “chi”, “chong”, “chou”, “chu”,“chuai”, “chuan”, “chuang”, “chui”, “chun”, “chuo”, “ci”, “cong”, “cou”, “cu”,……



本实验的Trie树应用



Email格式：
- 开头为数字或字母
- 中间可含“_”、“-”、“.”3类字符
- **不区分大小写**
- local part 为 “@”前面的部分，最多64个字符
- domain part 为 “@”后面的部分，最多255个字符
- 总长度不超过255+64+“@”=320字符

1、实验任务

2、核心算法之Trie树

3、核心算法之BloomFilter

4、代码提交说明

31

Hash思想回顾

- 思想
  - 把任意长度的输入通过Hash算法，变换成固定长度的输出
    - 把一些不同长度的信息转化成杂乱的128/256位的编码
    - 代表算法：MD5, SHA
  - 特点
    - 一般是压缩映射，不可逆映射
    - 把一个大范围映射到一个小范围
- 使用领域
  - 信息安全基础设施实现加密、消息摘要等
  - 数字指纹：文档去重、内容检索
  - 内容安全：数据防泄漏

数据结构之Hash Table

- 思想
  - 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
  - 实际查找通过先计算，再匹配元素是否相同（字符串匹配）
- 问题
  - 解决冲突
  - 加大表空间，
  - 查找复杂（
  - 准确率100%

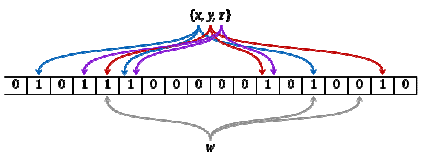
如何改进？

Bloom Filter概念

- 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映射函数**
  - 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={$x_1, x_2,...,x_n$}
  - 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,...,m}的范围
  - S集合中的每个元素用k个hash函数映射到，{1,...,m}范围内，将相应的位置为1



Bloom Filter原理

初始化时m 位数组置零

B | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

对集合中的每个元素x_j 分别进行k 次hash，If $H_i(x_j) = a$, set $B[a] = 1$.

B | 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 0

要检测y是否在集合S，测试所有k 个B $[H_i(y)]$是否都为1.

B | 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 0

可能出现false positive：即所有k个值都是1，但y不在集合S中

B | 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 0

n items　　$m = cn$ bits　　k hash functions

36

错误率估计

- 当集合S={$x_1, x_2, ..., x_n$}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：

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

- 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

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

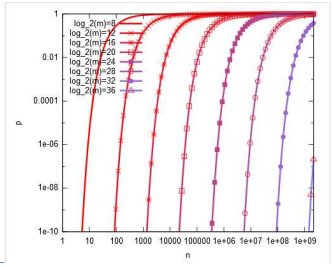
需要的存储空间m

- 推导结果公式：(假设，允许的false rate为p)

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

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

False positive rate随Hash函数个数和存储空间增加成指数级降低



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FP 2=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	19170W	23962W	33547 W	22 M	28M	39M	13	17	23	36533	3552	41

Hash算法选择

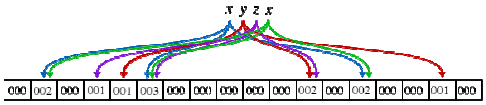
- K个hash越相互独立效果越好
- 常用的HASH算法
  - http://www.partow.net/programming/hashfunctions/
  - unsigned int RSHash (char* str, unsigned int len);
  - unsigned int JSHash (char* str, unsigned int len);
  - unsigned int PJWHash (char* str, unsigned int len);
  - unsigned int ELFHash (char* str, unsigned int len);
  - 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/

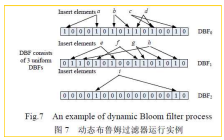
扩展：计数型Bloom Filter

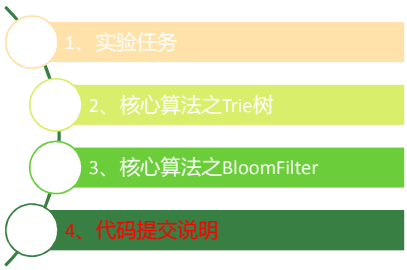
- 标准BF
  - 没法删除元素
  - m长数组，每unit用1位表示，只能表示[0, 1]
- 如果每个unit用多位表示，如3位，则可以表示[0,1,...,7]



扩展：压缩Bloom Filter

- 由于标准Bloom Filter在f最小时，任意位为0的概率为1/2，所以，为了更好的在网络上传输Bloom Filter，可以对Bloom Filter进行压缩，能得到约69.3%





45

编程实验环境

- 代码和数据服务器
  - 见邮件
  - Ssh访问
- 操作系统
  - 推荐Ubuntu Linux操作系统（虚拟机或cygwin）
- 编辑器
  - Source Insight或Dev C++
- 编译和调试
  - GCC/G++
  - gdb
  - Makefile
- 使用的库
  - STL

46

实验输入和输出

- 输入数据文件
  - 总共1500万个email，每个email地址一行

```
timothy15447jv4028@yahoo.com
timothy15447jv58@hotpop.com
timothy15447jv5@yahoo.com
timothy15447jv741@yahoo.com
timothy15447fde@imailbox.com
timothy15447fde@flashmail.com
timothy15447edf4794@yahoo.com
timothy15447edf@imailbox.com
timothy15447edfe@imailbox.com
timothy15447edfqq5017@yahoo.com
timothy15447mq3413@yahoo.com
timothy15447vj365v@hotpop.com
```

  - 待检测数据1000个email，每个email地址一行
- 输出要求
  - 每行输出一个1000行结果的文件，每行一个yes/no，表示1000个email是否在1500万email中

实验说明

- 代码文件
  - 代码文件
    - main.c bloom.c bloom.h trie.c trie.h hash.c hash.h
    - Makefile编译规则文件
    - Readme说明程序的编译和使用方法
  - 命令行格式
    - ./strsearch emaillist.txt checklist.dat checkedresult.dat
  - 输入文件
    - emaillist.dat，1500万个email
    - checklist.dat，1000个待检测的email
  - 输出文件（要求程序生成）
    - checkresult.dat
- 文档
  - 实验XXXXX设计文档.doc

实验代码框架之main()

```
int main(int argc, char *argv[])
{
        fp_strpool=fopen(argv[1], "r");
        fp_checkedstr=fopen(argv[2],"r");
        fp_result=fopen(argv[3],"w");

        trie_check(fp_strpool,fp_checkedstr,fp_result);

        fputs(fp_result,"----------trie end----------");

        bf_check(fp_strpool,fp_checkedstr,fp_result);

        fclose(fp_strpool);
        fclose(fp_checkedstr);
        fclose(fp_result);
}
```

实验代码框架之trie_check()

```
int trie_check(fp_strpool,fp_checkedstr,fp_result)
{
        int pos = -1;
        Trie trie = trie_create();
        while(fgets(line, 1024, fp_strpool)){
                trie_insert(trie, line);
        }
        while(fgets(line, 1024, fp_checkedstr)){
                pos++;
                if(!trie_search(trie, line)){
                        fprintf("%d\n", pos);
                }
        }
}
```

实验代码框架之bf_check()

```
int bf_check(fp_strpool,fp_checkedstr,fp_result)
{
        int pos = -1;
        void (*hashfamily[17])(const char *) = {hash1,hash2,hash3....};
        BF bf= bf_create(239620000,hashfamily); //大约29M

        while(fgets(line, 1024, fp_strpool)){
                bf_add(bf, line);
        }
        while(fgets(line, 1024, fp_checkedstr)){
                pos++;
                if(!bf_search(trie, line)){
                        fprintf("%d\n", pos);
                }
        }
}
```