# 实验报告

## 数据特征

分别基于hdu.txt和poj.txt各构造了3份规模为10000、100000、500000的数据，命名为hdu1.txt、hdu2.txt、hdu3.txt（poj同上）。数据的构造方法是：对于文本中的每一个用户名，每次生成一个随机数，如果为1则在查询操作时不改变该用户名（即最后能成功查询）；否则会在查询操作时随机地将该用户名丢弃末尾若干字符（即最后不能成功查询）。

## Hash实现思路

说明：
1.ASCII的单位元的值是它本身，UTF8的单位元的值是连续的1-3个字符组合成的十六进制数，例如0x72,0xc2a1,0xe384b5等。
2.对于每个单位(ascii或utf8)，记录它的秩为k，值为a[k]，各单位元计算后累计和为ans。

方法：
**1.独立链法**： ans+=a[k]*(32<<k);最后ans%=30000000
冲突处理：

```
if (!op) //插入
{
    if (!(~linkpoints[ans].value))//该处未冲突，则直接插入
    {
        linkpoints[ans].key = temp;
        linkpoints[ans].value = val;
    }
    else//出现冲突
    {
        if (linkpoints[ans].next == NULL)
        {
            linkpoints[ans].next = new linkpoint;
            linkpoints[ans].next->key = temp;
            linkpoints[ans].next->value = val;
            linkpoints[ans].next->next = NULL;
        }
        else
        {
            linkpoint* p = linkpoints[ans].next;
            while (p->next != NULL)
                p = p->next;
            p->next = new linkpoint;
            p->next->key = temp;
            p->next->value = val;
            p->next->next = NULL;
        }
    }
}
else //查找
{
    int flag = 0;
    for (linkpoint* p = &linkpoints[ans]; p != NULL; p = p->next)//遍历该处链表
    {
        if (!temp.compare(p->key))
        {
            flag = 1;
            cout << p->value << endl;
            break;
        }
    }
    if (!flag)
        cout << "-1" << endl;
}
```

在插入时如果对应位置为空，就直接插入，否则沿着该处拓展的列表找到末尾并伸展出这个新的位置，查找时则直接遍历链表即可。

**2.双向平方探测法**：ans+=llabs[temp[k]*(1<<k)]; ans%=prime prime=4n+3，具体数值随数据量的变化而变化，在代码中给了一些具体的值。
冲突处理：

```
if (!op) //插入
{
    int limit = (prime - 1) >> 1;
    for (int k = 0; k <= limit; k++)
    {
        long long bias = (long long)k * k;
        long long t1 = (ans + bias) % prime;
        long long t2 = (((ans - bias) % prime) + prime) % prime;
        if (!(~points[t1].value))//该处为空
        {
            points[t1].key = temp;
            points[t1].value = val;
            break;
        }
        if (!(~points[t2].value))//该处为空
        {
            points[t2].key = temp;
            points[t2].value = val;
            break;
        }
    }
}
else //查找
{
    int flag = 0;
    int limit = (prime - 1) >> 1;
    if (!points[ans].key.empty())
    {
        for (int k = 0; k <= limit; k++)
        {
            long long bias = (long long)k * k;
            long long t1 = (ans + bias) % prime;
            long long t2 = (((ans - bias) % prime) + prime) % prime;
            if (points[t1].key.empty() && points[t2].key.empty())
                break;
            if (temp == points[t1].key)//查询到了
            {
                flag = 1;
                cout << points[t1].value << endl;
                break;
            }
```

```
            if (temp == points[t2].key)//查询到了
            {
                flag = 1;
                cout << points[t2].value << endl;
                break;
            }
        }
    }
    if (!flag)
        cout << "-1" << endl;
}
```

如上图所示，增量分别为0,1,-1,4,-4,9,-9.....双向地试探直到找到空位置插入为止，查找时方法相同。

**3.再散列法**：ans计算同独立链，额外定义一个值为hash2，hash2+=llabs((long long)temp[k] * ((k << 1) + k + 1));
冲突处理：

```
if (!op) //插入
{
    for (int k = 0;; k++)
    {
        int bias = (ans + hash2 * k) % mod3;
        if (!(~points[bias].value))//该处为空
        {
            points[bias].key = temp;
            points[bias].value = val;
            break;
        }
    }
}
else //查找
{
    int flag = 0;
    if (!points[ans].key.empty())
    {
        for (int k = 0;; k++)
        {
            int bias = (ans + hash2 * k) % mod3;
            if (points[bias].key.empty())
                break;
            if (temp == points[bias].key)//查询到了
            {
                flag = 1;
                cout << points[bias].value << endl;
                break;
            }
        }
    }
    if (!flag)
        cout << "-1" << endl;
}
```

如上图所示，增量分别为0,hash2,2*hash2*,3hash2.....直到找到空位置插入为止，查找时方法相同。

## 测试结果

见附件"时间数据.xlsx"
说明：UTF8和ASCII分别代表针对两种编码设计的函数，1、2、3分别代表独立链法、双平方试探法和再散列法，例如UTF8-1代表针对UTF8编码的独立链法函数。

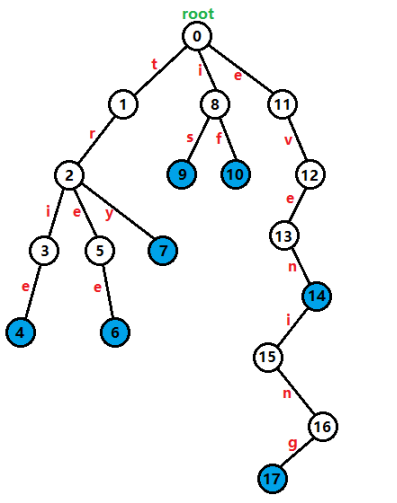
## 回答问题

1.实际效果上当成ASCII处理耗费的时间较少（双平方试探法反常，但可能是hash函数的选择问题），但两者相差时间不多，即使在500000数据量下也不到1s，可能的原因是UTF8的字符处理花费了额外的时间。

2.性能：再散列>独立链>双平方试探。可能的原因：再散列法的增量数据较大，具有更好的随机性，更能避免冲突；独立链法在每次插入和查找花费的时间小于双平方试探，因为独立链每次最多查找n次（n为之前重复个数），而双平方试探大于n次，实际效果上，当数据越大时，两者的时间差越明显，500000数据量时已经相差>5s。

3.可能使hash的结果分布比预期更不均匀，出现更多的冲突，降低运行时间。

4.输入数据：ASCII字符串  数据结构：字典树
例子：如下图所示，为字符串trie、tree、try、is、if、evening建树，对于每个字符串，只在叶子节点处储存它对应的数字即可。




设一共有n个字符串，最长字符串的长度为k
建树时间：O(n*k)

单次查找/修改时间：O(k）
由于字典树保证不会发生冲突，且不用对字符串进行hash处理，故对于大规模字符串的存储与查找其实际效果往往比hash表更好。

## 参考资料

https://blog.csdn.net/WhereIsHeroFrom/article/details/112271312
https://baike.baidu.com/item/UTF-8/481798