**Tries树**

基本思想（以字母树为例）：  
1、插入过程  
 对于一个单词，从根开始，沿着单词的各个字母所对应的树中的节点分支0向下走，直到单词遍历完，将最后的节点标记为红色，表示该单词已插入Trie树。  
2、查询过程  
 同样的，从根开始按照单词的字母顺序向下遍历trie树，一旦发现某个节点标记不存在或者单词遍历完成而最后的节点未标记为红色，则表示该单词不存在，若最后的节点标记为红色，表示该单词存在。

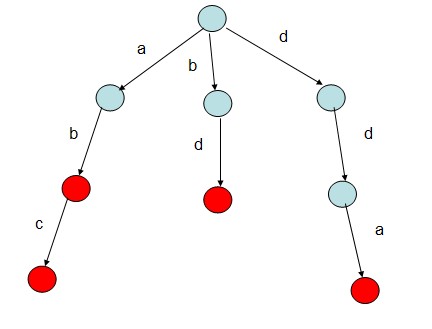
复杂度：

（1） 插入、查找的时间复杂度均为O(N)，其中N为字符串长度。

（2） 空间复杂度是26^n级别的，非常庞大（可采用双数组实现改善）。

字典树的数据结构：  
    利用串构建一个字典树，这个字典树保存了串的公共前缀信息，因此可以降低查询操作的复杂度。  
    下面以英文单词构建的字典树为例，这棵Trie树中每个结点包括26个孩子结点，因为总共有26个英文字母(假设单词都是小写字母组成)。  
    则可声明包含Trie树的结点信息的结构体:

1. **typedef** **struct** Trie\_node
2. {
3. **int** count;                    // 统计单词前缀出现的次数
4. **struct** Trie\_node\* next[26];   // 指向各个子树的指针
5. **bool** exist;                   // 标记该结点处是否构成单词
6. }TrieNode , \*Trie;

 其中next是一个指针数组，存放着指向各个孩子结点的指针。  
      如给出字符串"abc","ab","bd","dda"，根据该字符串序列构建一棵Trie树。则构建的树如下:  
  
 Trie树的根结点不包含任何信息，第一个字符串为"abc"，第一个字母为'a'，因此根结点中数组next下标为'a'-97的值不为NULL，其他同理，构建的Trie树如图所示，红色结点表示在该处可以构成一个单词。很显然，如果要查找单词"abc"是否存在，查找长度则为O(len)，len为要查找的字符串的长度。而若采用一般的逐个匹配查找，则查找长度为O(len\*n)，n为字符串的个数。显然基于Trie树的查找效率要高很多。  
如上图中：Trie树中存在的就是abc、ab、bd、dda四个单词。在实际的问题中可以将标记颜色的标志位改为数量count等其他符合题目要求的变量。  
已知n个由小写字母构成的平均长度为10的单词，判断其中是否存在某个串为另一个串的前缀子串。下面对比3种方法：  
  
1、 最容易想到的：即从字符串集中从头往后搜，看每个字符串是否为字符串集中某个字符串的前缀，复杂度为O(n^2)。  
  
2、 使用hash：我们用hash存下所有字符串的所有的前缀子串。建立存有子串hash的复杂度为O(n\*len)。查询的复杂度为O(n)\* O(1)= O(n)。  
  
3、 使用Trie：因为当查询如字符串abc是否为某个字符串的前缀时，显然以b、c、d....等不是以a开头的字符串就不用查找了，这样迅速缩小查找的范围和提高查找的针对性。所以建立Trie的复杂度为O(n\*len)，而建立+查询在trie中是可以同时执行的，建立的过程也就可以成为查询的过程，hash就不能实现这个功能。所以总的复杂度为O(n\*len)，实际查询的复杂度只是O(len)。  
**三、Trie树的操作**  
    在Trie树中主要有3个操作，插入、查找和删除。一般情况下Trie树中很少存在删除单独某个结点的情况，因此只考虑删除整棵树。  
1、插入  
  假设存在字符串str，Trie树的根结点为root。i=0，p=root。  
  1)取str[i]，判断p->next[str[i]-97]是否为空，若为空，则建立结点temp，并将p->next[str[i]-97]指向temp，然后p指向temp；  
   若不为空，则p=p->next[str[i]-97]；  
  2)i++，继续取str[i]，循环1)中的操作，直到遇到结束符'\0'，此时将当前结点p中的 exist置为true。  
2、查找  
  假设要查找的字符串为str，Trie树的根结点为root，i=0，p=root   
  1)取str[i]，判断判断p->next[str[i]-97]是否为空，若为空，则返回false；若不为空，则p=p->next[str[i]-97]，继续取字符。  
  2)重复1)中的操作直到遇到结束符'\0'，若当前结点p不为空并且 exist 为true，则返回true，否则返回false。

3. trie树的应用：

### 1. 字符串检索，词频统计，搜索引擎的热门查询

        事先将已知的一些字符串（字典）的有关信息保存到trie树里，查找另外一些未知字符串是否出现过或者出现频率。

        举例：

       1） 。

       2）给出N 个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。

       3）给出一个词典，其中的单词为不良单词。单词均为小写字母。再给出一段文本，文本的每一行也由小写字母构成。判断文本中是否含有任何不良单词。例如，若rob是不良单词，那么文本problem含有不良单词。

       4）1000万字符串，其中有些是重复的，需要把重复的全部去掉，保留没有重复的字符串

**5）寻找热门查询**：搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，这些查询串的重复读比较高，虽然总数是1千万，但是如果去除重复和，不超过3百万个。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。请你统计最热门的10个查询串，要求使用的内存不能超过1G。

### 2. 字符串最长公共前缀

       Trie树利用多个字符串的公共前缀来节省存储空间，反之，当我们把大量字符串存储到一棵trie树上时，我们可以快速得到某些字符串的公共前缀。举例：

      1) 给出N 个小写英文字母串，以及Q 个询问，即询问某两个串的最长公共前缀的长度是多少.  解决方案：

        首先对所有的串建立其对应的字母树。此时发现，对于两个串的最长公共前缀的长度即它们所在结点的公共祖先个数，于是，问题就转化为了离线  （Offline）的最近公共祖先（Least Common Ancestor，简称LCA）问题。

       而最近公共祖先问题同样是一个经典问题，可以用下面几种方法：

        1. 利用并查集（Disjoint Set），可以采用采用经典的Tarjan 算法；

        2. 求出字母树的欧拉序列（Euler Sequence ）后，就可以转为经典的最小值查询（Range Minimum Query，简称RMQ）问题了；

### 3.  排序

       Trie树是一棵多叉树，只要先序遍历整棵树，输出相应的字符串便是按字典序排序的结果。

        举例： 给你N 个互不相同的仅由一个单词构成的英文名，让你将它们按字典序从小到大排序输出。