数据结构与算法（9）：Trie树

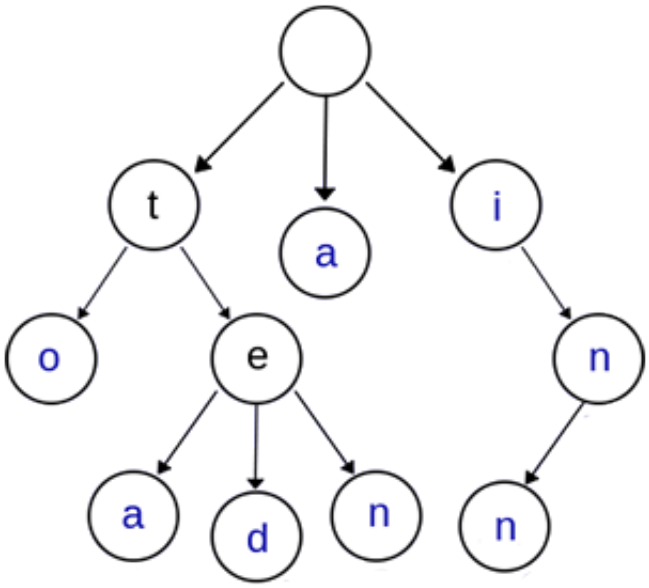
Trie树是一种非常重要的数据结构，它在信息检索，字符串匹配等领域有广泛的应用，同时，它也是很多算法和复杂数据结构的基础，如后缀树，AC自动机等，因此，掌握Trie树这种数据结构，对于一名IT人员，显得非常基础且必要！

一、什么是Trie树

Trie树，又叫字典树、前缀树（Prefix Tree）、单词查找树或键树，是一种多叉树结构。

字典树（Trie）可以保存一些字符串->值的对应关系。基本上，它跟 Java 的 HashMap 功能相同，都是 key-value 映射，只不过 Trie 的 key 只能是字符串。是一种哈希树的变种

如下图：



上图是一棵Trie树，表示了关键字集合{“a”,"to","tea","ted","ten","i","in","inn"}。从上图可以归纳出Trie树的基本性质：

1. 根节点不包含字符，除节结点外的每一个节点都包含一个字符。
2. 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符互不相同。

通常在实现的时候，会在节点结构中设置一个标志，用来标记该结点处是否构成一个单词（关键字）。

可以看出，Trie树的关键字一般都是字符串，而且Trie树把每个关键字保存在一条路径上，而不是

一个结点中。另外，两个有公共前缀的关键字，在Trie树中前缀部分的路径相同，所以Trie树又叫做前缀树（Prefix Tree）。

二、Trie的优缺点

Trie的核心思想是空间换时间。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。

2.1 优点

Trie的强大之处就在于它的时间复杂度，插入和查询的效率很高，都为$O(K)$，其中 K 是待插入/查询的字符串的长度，而与Trie中保存了多少个元素无关。

关于查询，会有人说 hash 表时间复杂度是$O(1)$不是更快？但是，哈希搜索的效率通常取决于 hash 函数的好坏，若一个坏的 hash 函数导致很多的冲突，效率并不一定比Trie树高。

而Trie树中不同的关键字就不会产生冲突。它只有在允许一个关键字关联多个值的情况下才有类似hash碰撞发生。

此外，Trie树不用求 hash 值，对短字符串有更快的速度。因为通常，求hash值也是需要遍历字符串的。

Trie树可以对关键字按字典序排序。

2.2 缺点

当然，当 hash 函数很好时，Trie树的查找效率会低于哈希搜索。

其次因为Trie的核心思想是空间换时间，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。所以它的空间消耗比较大。

三、Trie树的应用

典型应用是用于统计和排序大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。

3.1 字符串检索

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。

检索/查询功能是Trie树最原始的功能。给定一组字符串，查找某个字符串是否出现过，思路就是从根节点开始一个一个字符进行比较：

- 如果沿路比较，发现不同的字符，则表示该字符串在集合中不存在。
- 如果所有的字符全部比较完并且全部相同，还需判断最后一个节点的标志位（标记该节点是否代表一个关键字）。

```
struct trie_node
{
    bool isKey;   // 标记该节点是否代表一个关键字
    trie_node *children[26]; // 各个子节点
};
```

3.2 词频统计

Trie树常被搜索引擎系统用于文本词频统计 。

```
struct trie_node
{
    int count;   // 记录该节点代表的单词的个数
    trie_node *children[26]; // 各个子节点
};
```

思路：为了实现词频统计，我们修改了节点结构，用一个整型变量count来计数。对每一个关键字执行插入操作，若已存在，计数加1，若不存在，插入后count置1。

3.3 排序

Trie树可以对大量字符串按字典序进行排序，思路也很简单：
给定N个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出。用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可。

3.4 前缀匹配

例如：找出一个字符串集合中所有以ab开头的字符串。我们只需要用所有字符串构造一个trie树，然后输出以$a->b->$开头的路径上的关键字即可。

trie树前缀匹配常用于搜索提示。如当输入一个网址，可以自动搜索出可能的选择。当没有完全匹配的搜索结果，可以返回前缀最相似的可能

3.5 最长公共前缀

查找一组字符串的最长公共前缀，只需要将这组字符串构建成Trie树，然后从跟节点开始遍历，直到出现多个节点为止（即出现分叉）。

举例说明：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；

关于并查集，Tarjan算法，RMQ问题，网上有很多资料。

3.6 作为辅助结构

如后缀树，AC自动机等。

3.7 应用实例

1. 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。
    - 之前在此文：海量数据处理面试题集锦与Bit-map详解中给出的参考答案：用trie树统计每个词出现的次数，时间复杂度是$O(n*le)$（le表示单词的平均长度），然后是找出出现最频繁的前10个词。也可以用堆来实现（具体的操作可参考第三章、寻找最小的k个数），时间复杂度是$O(n*lg10)$。所以总的时间复杂度，是$O(n*le)$与$O(n*lg10)$中较大的哪一个。
2. 有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。
3. 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。请怎么设计和实现?
4. 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给

出思想，给出时间复杂度分析。

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。
    - (1) 请描述你解决这个问题的思路；
    - (2) 请给出主要的处理流程，算法，以及算法的复杂度。

四、Trie树的实现

Trie树的插入、删除、查找的操作都是一样的，只需要简单的对树进行一遍遍历即可，时间复杂度：O（n）（n是字符串的长度）。

trie树每一层的节点数是26^i级别的。所以为了节省空间，对于Tried树的实现可以使用数组和链表两种方式。空间的花费，不会超过单词数×单词长度。

1. 数组：由于我们知道一个Tried树节点的子节点的数量是固定26个（针对不同情况会不同，比如兼容数字，则是36等），所以可以使用固定长度的数组来保存节点的子节点
    - 优点：在对子节点进行查找时速度快
    - 缺点：浪费空间，不管子节点有多少个，总是需要分配26个空间
2. 链表：使用链表的话我们需要在每个子节点中保存其兄弟节点的链接，当我们在一个节点的子节点中查找是否存在一个字符时，需要先找到其子节点，然后顺着子节点的链表从左往右进行遍历
    - 优点：节省空间，有多少个子节点就占用多少空间，不会造成空间浪费
    - 缺点：对子节点进行查找相对较慢，需要进行链表遍历，同时实现也较数组麻烦

> java

```java
import java.util.ArrayList;
import java.util.List;
/**
 * 单词查找树
 */
class Trie {
    /** 单词查找树根节点，根节点为一个空的节点 */
    private Vertex root = new Vertex();
    /** 单词查找树的节点(内部类) */
    private class Vertex {
```

```
        /** 单词出现次数统计 */
        int wordCount;
        /** 以某个前缀开头的单词，它的出现次数 */
        int prefixCount;
        /** 子节点用数组表示 */
        Vertex[] vertexs = new Vertex[26];
        /**
         * 树节点的构造函数
         */
        public Vertex() {
            wordCount = 0;
            prefixCount = 0;
        }
    }
    /**
     * 单词查找树构造函数
     */
    public Trie() {
    }
    /**
     * 向单词查找树添加一个新单词
     *
     * @param word
     *             单词
     */
    public void addWord(String word) {
        addWord(root, word.toLowerCase());
    }
    /**
     * 向单词查找树添加一个新单词
     *
     * @param root
     *             单词查找树节点
     * @param word
     *             单词
     */
    private void addWord(Vertex vertex, String word) {
        if (word.length() == 0) {
            vertex.wordCount++;
        } else if (word.length() > 0) {
            vertex.prefixCount++;
            char c = word.charAt(0);
            int index = c - 'a';
            if (null == vertex.vertexs[index]) {
                vertex.vertexs[index] = new Vertex();
            }
            addWord(vertex.vertexs[index], word.substring(1));
        }
```

```
    }
    /**
     * 统计某个单词出现次数
     *
     * @param word
     *             单词
     * @return 出现次数
     */
    public int countWord(String word) {
        return countWord(root, word);
    }
    /**
     * 统计某个单词出现次数
     *
     * @param root
     *             单词查找树节点
     * @param word
     *             单词
     * @return 出现次数
     */
    private int countWord(Vertex vertex, String word) {
        if (word.length() == 0) {
            return vertex.wordCount;
        } else {
            char c = word.charAt(0);
            int index = c - 'a';
            if (null == vertex.vertexs[index]) {
                return 0;
            } else {
                return countWord(vertex.vertexs[index], word.substring(1));
            }
        }
    }
    /**
     * 统计以某个前缀开始的单词，它的出现次数
     *
     * @param word
     *             前缀
     * @return 出现次数
     */
    public int countPrefix(String word) {
        return countPrefix(root, word);
    }
    /**
     * 统计以某个前缀开始的单词，它的出现次数(前缀本身不算在内)
     *
     * @param root
     *             单词查找树节点
```

```
     * @param word
     *            前缀
     * @return 出现次数
     */
    private int countPrefix(Vertex vertex, String prefixSegment) {
        if (prefixSegment.length() == 0) {
            return vertex.prefixCount;
        } else {
            char c = prefixSegment.charAt(0);
            int index = c - 'a';
            if (null == vertex.vertexs[index]) {
                return 0;
            } else {
                return countPrefix(vertex.vertexs[index], prefixSegment.substring(1
));
            }
        }
    }

    /**
     * 调用深度递归算法得到所有单词
     * @return 单词集合
     */
    public List<String> listAllWords() {
        List<String> allWords = new ArrayList<String>();
        return depthSearchWords(allWords, root, "");
    }
    /**
     * 递归生成所有单词
     * @param allWords 单词集合
     * @param vertex 单词查找树的节点
     * @param wordSegment 单词片段
     * @return 单词集合
     */
    private List<String> depthSearchWords(List<String> allWords, Vertex vertex,
            String wordSegment) {
        Vertex[] vertexs = vertex.vertexs;
        for (int i = 0; i < vertexs.length; i++) {
            if (null != vertexs[i]) {
                if (vertexs[i].wordCount > 0) {
                    allWords.add(wordSegment + (char)(i + 'a'));
                    if(vertexs[i].prefixCount > 0){
                        depthSearchWords(allWords, vertexs[i], wordSegment + (char)
(i + 'a'));
                    }
                } else {
                    depthSearchWords(allWords, vertexs[i], wordSegment + (char)(i +
 'a'));
```

```
                }
            }
        }
        return allWords;
    }
}
public class Main {
    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.addWord("abc");
        trie.addWord("abcd");
        trie.addWord("abcde");
        trie.addWord("abcdef");
        System.out.println(trie.countPrefix("abc"));
        System.out.println(trie.countWord("abc"));
        System.out.println(trie.listAllWords());
    }
}
```