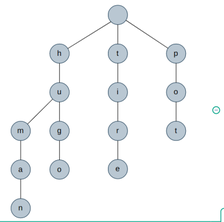
Trie

定义

Trie是一种多叉树，用于专门处理字符串而设计，其目的就在于实现字典的高效查询，且其空间利用率要优于散列表

Trie在进行字符串的查找时，其时间复杂度与字典中的条目数量无关，只与待查询的字符串长度相关，时间复杂度为O(m)，m为字符串的长度



和其它字典实现不同，Trie将单词拆分成一个一个的字母，每一个字母就是一个节点

左图中一共包含4个字母，其中"hugo"与"human"共享前两个字母。

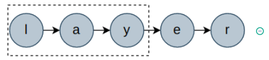
节点的设计

每个节点将会有若干个指向下一个节点的指针，如果只是单纯的小写英文字母，则只需要26个指针

节点指针的选择

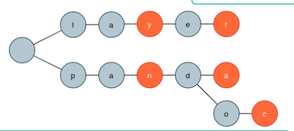
对于数量不定的指针存储，首先想到的就是使用数组进行存储，在查找下一个节点时可使用二分搜索进行查找，但是插入元素的效率较低

同时也可以使用散列表来进行存储，此时key为下一个节点的值，value为下一个节点，查找和删除元素的效率较高，但是需要一些额外的内存空间



在上图中进行单词查找时，会直接遍历到Trie的叶子节点来进行判断。但是，英文单词中有许多的单词前缀是重复的

如果完全遍历至叶子节点的话，将会错过实际存储的数据



所以，节点中还需要存储一个标识位，用于标记当前节点是否为某个单词的结尾

实现

```
class Node {
    public boolean isWordEnd;
    public TreeMap<Character, Node> next;
}
```

这里使用Map来保存节点字符和指针的对应关系

Trie的操作

添加一个单词

在添加一个单词时，首先需要将原单词拆分成一个个的字符，然后插入至Trie中

```
public void add(String word) {
    Node current = root;

    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);

        if (current.next.get(c) == null)
            current.next.put(c, new Node());

        current = current.next.get(c);
    }

    if (!current.isWordEnd)
        current.isWordEnd = true;
}
```

可以看到，使用Map来存储指针在编码实现时相当的简洁

完整单词的查询

```
public boolean contains(String word) {
    Node current = root;

    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);

        if (current.next.get(c) == null)
            return false;

        current = current.next.get(c);
    }

    return current.isWordEnd;
}
```

一个需要注意的问题，当循环结束时，是直接返回true，还是返回当前节点的结束标识位？

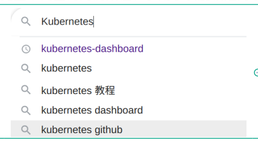
若直接返回true，那么是否需要将当前查询的结束节点的标识位设置为true?

返回节点的结束标识要更加合理一些，不过仍然需要视具体场景而定

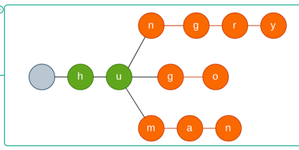
Trie的应用

Trie由于其高效的字符前缀、完整单词查询的高效性，多用于长度较短的字符串查询

实现搜索引擎的自动补齐



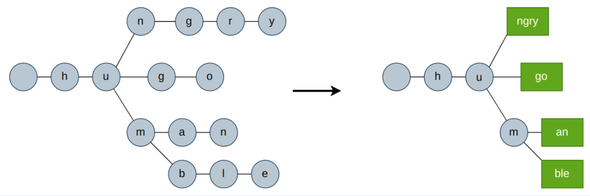
对于搜索中非常热门的词语或者是词组，完全可以构造出一个Trie，然后找出prefix为搜索词的所有单词，然后根据权重进行排序返回



IDE的自动补全功能，其实和搜索引擎的自动补齐是一样的，只不过应用于不同的场景

对空间的优化

在Trie的基本实现中，不管是使用数组还是使用Map来保存节点指针，都会造成比较大的空间开销



因此，产生了压缩Trie，对单链进行压缩，从而省去多余的节点存储开销