Trie的使用意义：

运算时间从O(mlogn)提升至O(m)，n为字典的二分树大小，logn约为15~20。

以“你是一个好人”为例，二分树需要搜索“你”，“你是”，“你是一”。。。共6次搜索，每次搜索耗时logn，所以为以“你”开头的一句话需要mlogn的搜索时间。而使用trie从根节点搜索完“你”之后会返回“你”节点，其子节点包含“你是”，从“你”节点搜索“你是”耗时O(1)，之后返回“你是”节点，所以共耗时O(m)。

如果没有“你是一个好人”节点存在，只有“你”和“你是”节点存在，trie在搜索至“你是”节点后会知道不会有更长得节点存在，从而不会继续搜索，但是二分树则需要继续搜索。

Trie的中node结构：

每个节点都包含一个子节点列表，父节点（为了字典插入时更省时间，因为字典中下一个词很可能与上一个词在同一个父节点中，但如果并不是，那就从父节点的父节点中插入，以此类推至根节点），字符串值和字符串频率。每个子节点的字符串长度-1的值是其父节点的字符串值。

重要方程：

AddNode(node)

输入：需要加入的节点

输出：节点是否成功加入或修改trie

考虑情况：

前缀若不同则直接返回False

这个字符串是否应该属于此节点的子节点（是否是此节点字符串的长度+1），若属于，则遍历子节点查询是否有相同字符串存在，若有，则直接将要加入字节的频率加至已存在的字节频率中，若没有，则将要加入的节点直接加入子节点列表，并更新其父节点。

若不属于，则查看此节点是否有子节点与字符串有相同前缀，若有，则交给这个子节点处理（子节点.AddNode(node)），并返回其处理结果（肯定为True，因为前缀相同）。若没有，则建立一个此字节串相同前缀+1的中继子节点，词频为0，并交给这个中继节点处理此字符串（如字符串为abc，此节点为a,此节点没有ab节点,则创立ab节点并加入a的子节点，然后ab.AddNode(abc)）。

注意：

由于utf-8的汉字由多个字节组成，而每个子节点与母节点只相差一个字节，所以一个汉字会需要通过向子节点的多层子节点查找而得到，iterSearchChild实现这个多层查找的功能。它会通过输入字符串，从指定节点的子节点中，找到和输入字符串完全相同的节点，并返回，如果其子节点中不含有这个字符串，则返回-1.

训练集中字符串的添加没有从字典中相同父节点的规律，所以训练集的字符串添加较之字典会缓慢很多。

Node中仅由查找和添加方程，因为暂无删除的需求