# Term Project _ Option 2: Dictionary Lookup using Tries
# CS 2336.004
# Due Date: April 30th, 2017

In this project option, you are going to implement dictionary lookup using Tries (a special rooted tree).

**Trie**

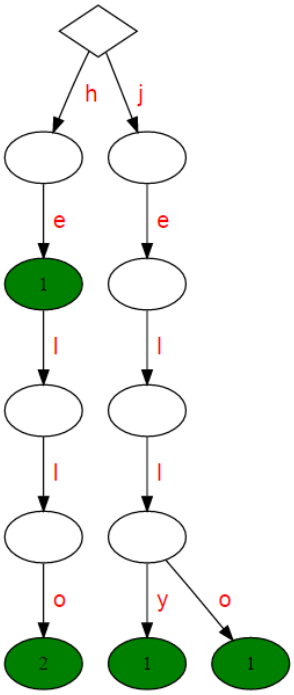
A trie is a special rooted tree that allows us to store a set of words by sharing their common prefixes.

Example:

Consider the following list of five words:

hello, jello, he, hello, jelly

The trie data structure for the list above is shown below:




A **trie node** can be of three types:

1. Root of the trie (shown as a diamond)
2. Internal non-terminal node of a trie (white ovals)
3. Word-ending node of a trie (green ovals).

Note that a word-ending node need not be a leaf. Also, such nodes are associated with a number that represents the frequency of the corresponding word.

Each trie edge is labelled with an alphabet.

**Reading the words in the dictionary.**

Starting from the root (diamond node), as we traverse a path along the trie, the letters along a path form a word. Whenever we encounter a word-ending node, we can stop and read off a valid word in the dictionary.

For instance, we note the correspondence between the list of words:

hello, hello, jello, he, jelly

***Trie Data Structure: Specification***

A trie (or a prefix tree) data structure allows us to store a set of strings from a dictionary along with a frequency count for each strings. The following operations should be supported:

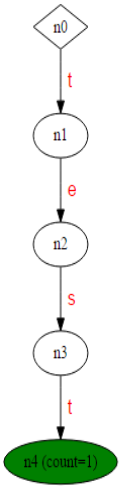
| Operation | Description | Return Type |
|---|---|---|
| **addWord**(w) | Insert a word w into the trie | None |
| **lookupWord**(w) | Return the frequency of w in the trie (0 if not present) | **int** |
| **autoComplete**(w) | Return a list of words in the trie along with their frequencies so that w is a prefix of each word in the list | **list** of (**str**,**int**) |

**Example**

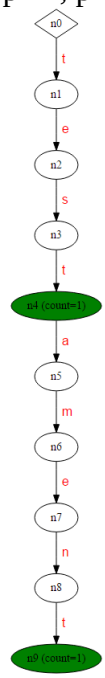
We will illustrate the process of constructing a trie using an example.
At the beginning the empty trie is simply a root node: N0

Suppose we add the strings in the following order:

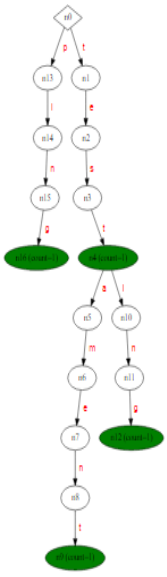
Test, testament, testing, ping, pink, pine, pint, testing, pinetree




1: **addWord**('test')

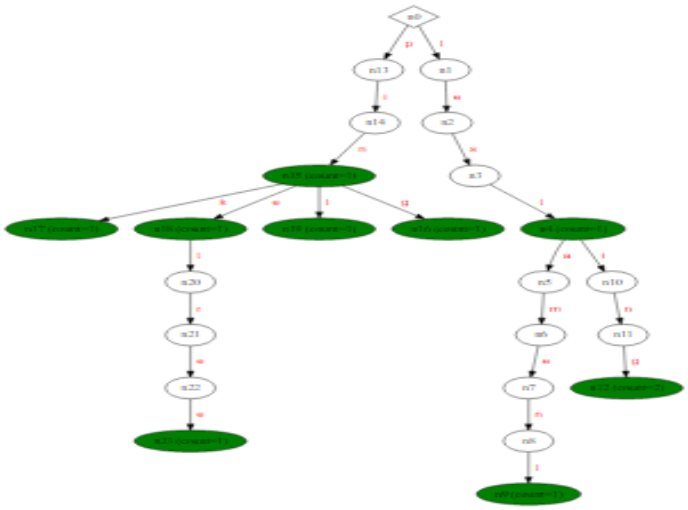



2: **addWord**('testament')



3: **addword**('testing')
4: **addWord**('ping')

The final trie after adding all workds looks like:



**Lookup of a Word**
The operation **lookupWord** allows us to count the frequency of a word in a trie. If a word does not belong to a trie, then the frequency is 0.

**Autocomplete**
The operation **autoComplete** provides a list of "completions" of a given string along with their frequencies.

**Trie Node**
A trie node structure has the following attributes:

- **isRoot** A Boolean flag denoting if the node is a root.
- **isWordEnd** A Boolean flag denoting if the node is the word ending node.
- **count** A count field for frequency count for a word ending node.
- **next**: A dictionary that maps from each of the alphabets 'a' - 'z' to the child node corresponding to the alphabet.

Please write a class for Trie, implement the functions addWord, lookupWord, and autoComplete.
Also write a test class which takes a TXT file as input, within the TXT files there is a list of words (dictionary file) and your test class will add all words in the TXT file, and the user can lookup a word, or given a prefix, the test class can give auto-completion.