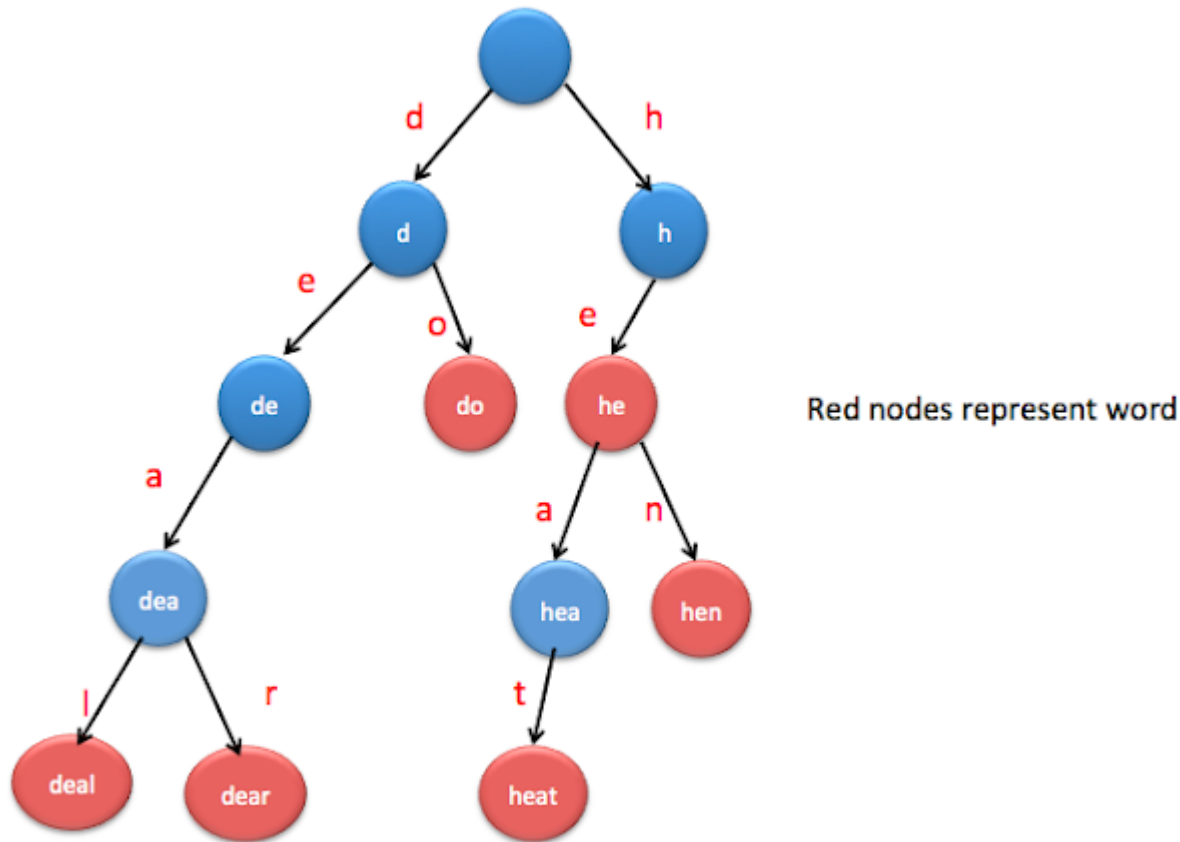


A2: Trie Trees

What are they?

Trie Trees are a tree data structure, sometimes called prefix trees. They are commonly used as a dictionary or hashmap (you'll learn more about this in 263), facilitating the search of a key. The nodes in the tree contain a value, and are found by traversing the edges in a depth-first search (you'll learn more about this in 263), which are linked by individual characters in the key.

Example



Trie Data structure

Keep in mind we won't store non-words in our nodes, this is just an example

Use Cases

Aside from serving as a dictionary (key-value pairs), trie trees have a few other applications

String Sorting

Inorder traversal of the trie tree would obtain you a sorted list of all words in the tree from a-z, likewise a postOrder traversal would obtain you a z-a ordering. Note this is with the assumption that a and z are your left and right most subtrees respectively.

Auto-complete

By traversing the tree with the given key, one can simply check all subtrees in the key's last character, as you would do when sorting, to get all possible words starting with the key as prefix

Auto-correct

Although not as complex as autocorrect in your phone or other applications, traversing the tree until you find a character with no valid subtree, then checking all the valid subtrees to see if they contain the remainder of the key can help you correct misspelled words.

Your Tasks

NOTE: we'll only use lowercase words

You will be implementing the functionality of a `trieTree` as described above.

Like ADTs we've seen before your exact implementation may differ as long as the expected outcome is the same, for example, you may choose between using a recursive solution or an iterative one.

Make sure you read the doc below each method thoroughly before starting to make sure you implement things properly

The `__init__` function and `__str__` are provided for you, and should not be changed. the `__str__` function will display a graphical representation of the `TrieTree` in your terminal/console.

Additionally, a script to load a lot of words into your trie tree is provided, you can use and augment this to help you test your implementation more.

Task 1: insert

Your first task is to the `insert` function. Given a word, insert it in the `TrieTree`. If your tree has subtrees matching part of the word, you should make sure not to augment these, as you could lose other words from your tree. If there are no subtrees for your words prefix, then you should create the children nodes appropriately and add them to your subtree dictionary. Some basic `docTests` are provided, but make sure to test out your implementation thoroughly to make sure all edge cases work properly.

Task 2: `__contains__`

Now that you can insert words, we want to be able to know if a given word is in our tree. Remember, only nodes with valid words have a value, so you can traverse the tree using the character of each node, and utilize this fact to evaluate whether a word exists in your tree.

Task 3: `__delitem__`

Now that you can insert words, and verify that they are there, let's implement a method to remove words. At first thought you may think to simply delete the value of node that contains the word, however, consider how that would impact your code's performance on other search methods. Because of this you will need to prune any useless branches after you delete a word from the tree.

Task 4: sort

Sorting is the first real use case of TrieTrees, by performing an Inorder Traversal of the TrieTree - with the assumption that the subtrees go left to right alphabetically - you can obtain a set of all your words in increasing order. Likewise a reversed inorder traversal will return a decreasingly ordered set.

You might want to review your tree traversals if you're unsure here.

Task 5: autoComplete

Now given a prefix of a word, return the first N words in your TrieTree that begin with that prefix, sorted alphabetically.

Hint: Consider the sort method and how it can be utilized to obtain words matching a prefix.

Task 6: autoCorrect

This implementation of autoCorrect is somewhat similar to autoComplete. Given a word, you will traverse your tree until you: **A** find the word, or **B** find no possible path forward.

In the case of **A** you simply need to return a list containing that word.

In the case of **B** you must look through your sub-children and traverse them to find possible words, with up to *errorMax* errors.

As an example, if your tree only contains **dab** and you autoCorrect **dod** with a max of 2 errors, it should find dab, however, if your max error is 1, it would not find anything.

More examples are in the docs.

Task 7: merge

This one is pretty straight forward, implement a function to merge 2 dictionaries together. Try to not use extra memory.