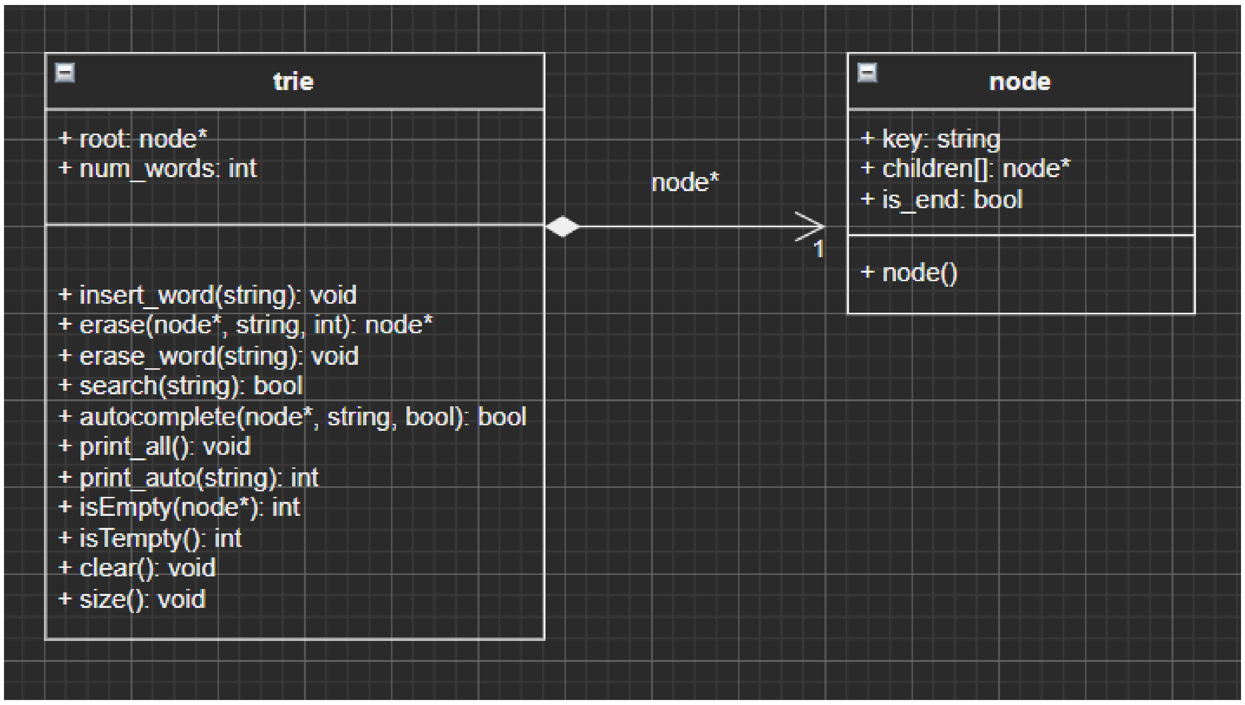
Project 2 Design Document

Zhiyuan Zhang z667zhan

1. Overview of classes

Two classes are implemented in this design, one for the trie data structure and the other for the nodes in the trie. Class ‘trie’ stores the root node and number of words in this trie structure, with functions to insert, erase and search for words in it. Class ‘node’ stores a bool value to check if it is the end of a word, and points to 26 children node representing ‘a’ to ‘z’.

2. UML class diagram



3. Design decisions:

<u>For trie class:</u>

For a trie data structure, when it is constructed the initial number of words is set to zero, and will increase or decrease based on the number of words inserted/erased. A root node is also created when initializing the trie data structure. Erasing/ searching a word in trie involve recursive functions, as it is the same process for each character in a word. Since it only involves 26 characters from ‘a’ to ‘z’, the size of children for each node can be set to a const 26.

For node class:

It has 26 children nodes, and a bool flag identifying whether it is the end of a word or not.

4. Test strategy:

For inserting/ erasing/ searching words, it should throw illegal exceptions if the input contains characters other than ‘a’ to ‘z’.

It should always return the correct size of trie; it should always return whether the trie is empty.

It should always set a node’s flag as end of word if it represents the end of a word, whether it is leaf or not.

For clearing, it should always remove all nodes from the trie.

For erase/ search nodes, it should always delete/ find the specific word if the word is inserted and not erased; it should not delete/ find a node if no words in the trie is the same as input.

For printing trie, it should always print all words in the trie starting from ‘a’; there should not be trailing blanks after the last printed word; it should not print anything if the trie is empty.

5. Performance analysis:

i(word): O(n) since each node only represents one character; it needs to go over or create n nodes, which are O(1), to complete the process.

e(word) & s(word): O(n) since each node only represents one character; it needs to go over n nodes in recurrence to delete a word or find a word.

print() & autocomplete(prefix*): For autocomplete, O(N) since it needs to check every word stored in trie that contains the input prefix; for print(), O(N) as well since it is the same process as autocomplete, with the only difference that the prefix is Null for print() function.

empty() & size(): O(1) since it only access the int that represents number of word.

clear(): O(N) since it needs to go over the entire trie to delete all words stored.