1.0 Class Design

1.1 TrieNode

TrieNode represents the individual nodes in the Trie structure, holding data about a particular class in the hierarchy and references to child nodes that represent subclasses.

1.1.1 Data

The pointers to the children of a TrieNode are stored in a vector of size 15. A vector was chosen due to the easy-to-use API and reasonable time complexity. Since we are told that the max number of subclasses if 15, a vector is suitable since it's operations will take constant time, but if it was the case that a node could have N children, then using a hash table might make more sense to improve the runtime of finding and deleting children. Each node also stores whether it is a terminal node, the state of which is used by the Trie to keep track of how many complete classifications it contains

1.1.2 Methods

The methods of the TrieNode mainly involve getting various attributes or helper functions to get information about the node including if it has children or if it's children contains a certain node. Most of the methods are either simple getters and setters or work by iterating over their child vector and returning the requested information.

1.2 Trie

Trie manages the hierarchical structure for classification. It provides an interface to insert classifications, classify input phrases, erase classifications, and traverse the Trie for various operations.

1.2.1 Data

The Trie only stores a reference to the root node and then uses the reference stored withing the nodes to traverse the tree. The Trie also stores it's size which is a count of the number of classifications. This count is incremented and decremented whenever a node changes from being a non-terminal to terminal and vice versa. The size variable can also quickly be used to determine if a tree is empty or not.

1.2.2 Methods

Contains methods to carry out the functionality required by the assignment including, inserting, erasing, classifying, printing and checking the size and emptiness state of the Trie. The inserting, erasing methods involve iterating over the classes in the classification, and then finding matching child nodes in the children of each node and then traversing to them. Classifying involved going to a node, running LabelText with the labels of the children and the input and then traversing to whichever child is returned by the LLM.

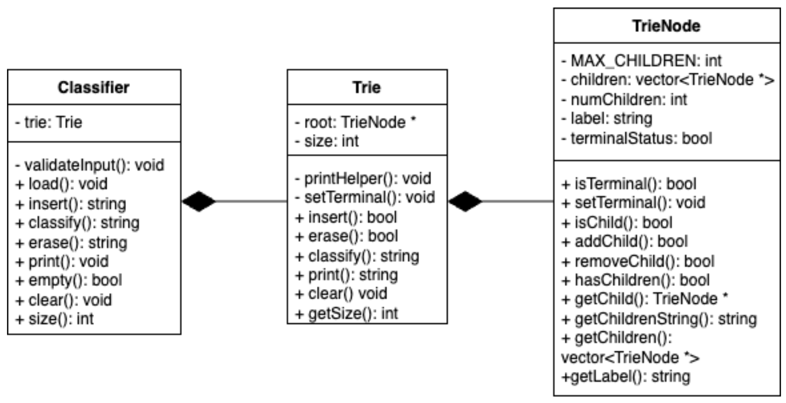
1.3 Classifier

In essence, this is really just a shim class that sits between the user and the Trie. I could have included all of the functionality for the project into the Trie but it did not seem like a proper separation of concerns. The classifiy provides and API which mostly just calls Trie methods but also performs input validation on then user inputs and interprets the Trie responses and converts them to print statements. The classifier class handles all of the printing outside of main. The only data the Classifier stores is a pointer to the Trie structure.

1.3.1 Methods

Other than the methods which are duplicated in the Trie, the classifier class also contains a load method which reads through a file line by line and then calls Trie.insert() on each line of the file to load the file into the Trie structure. The Classifier also has and empty method which calls the getSize methods from the Trie and does a logical comparison to 0 to determine if the tree is empty.

2.0 UML Diagram



4.0 Runtime

4.1 INSERT

To insert an n class classification, Trie.insert(), will have to traverse n nodes. Traversing each node involves searching the children vector for a child with a label that matches the next class, which in the worst case is O(15). If any of the intermediate classes are missing, or once the terminal class is reached, they are added to the tree with TrieNode.addChild() which appends to the children vector which is an O(1) operation, in the worst case, this happens n times. The worst case time complexity for INSERT is then O(n* (1 + 15)) = O(n)

4.2 CLASSIFY

To classify an input, Trie.classify() will traverse through, in the worst case, N nodes and at each node it will call LabelText on the input with the current children as the possible classes. Getting the labels of the current children is O(15) in the worst case. The worst-case time complexity of CLASSIFY is then O(N*(15+15)) = O(N)

4.3 ERASE

To erase and n class classification, Trie.erase() will have to traverse n nodes. Once it has reached the terminal node of the classification, it will have to delete the node by calling TrieNode.removeChild(), which since the node is a terminal node, guarantees deletion will be O(1). To properly delete the node, the pointer in the children vector of the parent node also needs to be set to nullptr, which takes O(15) in the worst case. The worst case time complexity of ERASE is then O(n*15 + 1 +15) = O(n).

4.4 PRINT

To print a Trie, Trie.print() recursively calls Trie.printHelper() on the children of each node starting from the root. printHelper collects the labels of the nodes that called it and when it reaches a terminal node, appends the string of traversed nodes to the vector of traversal strings and then returns, both operations are O(1). For each classification in the tree, in the worst case printHelper is called NxC times where C is the number of classes in the classification. Once all the calls to printHelper have returned, print() iterates over the vector of traversal strings and prints them out. In the worst case, there are N traversal strings to be printed out. The worst case time complexity for PRINT is then O(N*C*(1+1) + N) = O(N).

4.5 EMPTY and SIZE

The size of the Trie, or the number of classifications, is tracked by incrementing and decrementing the size attribute whenever a Trie node becomes a new terminal node or ceases to be one respectively. Thus returning the size of the Trie is O(1) since Trie.getSize() just return the attribute value. Trie.empty() is also O(1) since it just makes a call and returns the Boolean comparison to 0. The worst case runtime for EMPTY and SIZE then is O(1).

4.6 CLEAR

Trie.clear() calls delete on the root which calls the TrieNode destructor which calls delete on each of the children of the TrieNode. For a Trie with N nodes, N memory deallocations occur which each take O(1) time. Then worst-case time complexity for CLEAR then is O(N*1) = O(N)