Approach Rationale

This project implements a trie data structure using a fixed N-ary tree (N=15) where each node can store a classification level and point to more specific subclassifications.

Of the following **3 possible implementations** for this project:

1 - Linked List Structure
2 - Dynamic Array Vector Structure
3 - Hash Table at Each Node

Implementation 2 was chosen for the following reasons:

1. Provides O(1) access to any and all child nodes as required for this project, this is achieved through direct indexing
2. Vector-based implementations also allow for easy iteration used to iterate through children for classification in this project
3. Simplest memory management of the 3 implementations, especially compared to linked lists which require more complex traversal and additional memory overall
4. More efficient usage than hashtables for smaller N values such as 15 in this case

Implementation 2 was found to be the best fit as it is clearly the most simple and efficient choice for the requirements of this project.

Class Design

1. **Class Trie**

   This class is the main trie data structure implementation that handles the main operations. It manages classification storage and retrieval as well as interaction with the language model classifier. It makes use of helper functions that split comma-separated classifications, validate inputs, and assemble complete paths.

   This class handles errors by validating inputs and exception throwing for illegal arguments.

   Private:

1. root: Root node pointer
   Core data structure must be protected, access should be through public interface
2. classificationCount: Counter
   Must stay synchronized with tree operations, modified only during insert/erase
3. containsUppercase(): Helper function
   For internal validation only, access not needed for public
4. gatherClassifications(): Helper function
   Internal traversal helper, only used by print() method

   Public:

   constructor - destructor - split() - insert() - classify() - erase() - clear() - empty() - size() - print()

2. **Class TrieNode**

   This class manages and represents individual nodes in the trie structure. It manages up to 15 child nodes and tracks terminal status to identify complete classifications. This class does not allow copying to prevent memory related issues and makes each node responsible for deleting its children.

   Private:

1. children: Vector of child pointers.
   Direct access could break tree structure, modifications must be controlled through interface methods, and to prevent invalid child assignments.
2. classification: String stored at node
   Should be immutable after node creation
3. isTerminal: Terminal status flag
   Must be managed consistently with tree operations, changes controlled through setTerminal() only

   Public:

   constructor - destructor - isEndNode() - getClassification() - getChild() - setTerminal() - setChild() - hasChildren()

3. **Class illegal_exception**

   An exception class that implements a simple exception handler for dealing with illegal input arguments (inputs containing uppercase letters). It has one responsibility, it handles only uppercase letter violations. Uses const char* for message storage to minimize memory overhead.

Function Design

Function: std::vector<std::string> Trie::split(const std::string& str, char delimiter)

Parameters: str, char delimiter
Return Type: vector of substrings (tokens)
Rationale: Splits input string by extracting substrings between delimiters and adds them to the result vector, also handles empty strings. Function is used by INSERT, CLASSIFY, and ERASE to process classification paths.

Function: void Trie::gatherClassifications(TrieNode* node, std::string currentPath, std::vector<std::string>& result)

Parameters: node, currentPath, result
Return Type: void (modifies result vector)
Rationale: This function traverses the trie to collect the complete classifications for PRINT. It builds a new path by adding the child of the current node if the child exists. It adds currentPath to the result vector if the current node reached is terminal and has no children.

Function: bool Trie::insert(const std::string& classification)

Parameters: classification

Return Type: bool (true=success)

Rationale: This function calls split() on the input if it is valid and begins searching existing children for a matching classification. If a match is not found, it finds the first empty slot and inserts, then marks the final node as terminal and updates classificationCount.

Function: std::string Trie::classify(const std::string& input)

Parameters: input (text to be classified)

Return Type: string (comma-separated classification))

Rationale: This function validates input then collects classifications at current level and calls the LLM with input and classifications. It then finds the matching node in trie for LLM classification and adds the node classification to the result.

Function: void Trie::clear()

Parameters: n/a

Return Type: void

Rationale: This function deletes all children starting at root, creates a new empty root and resets classificationCount to 0.

Function: std::string Trie::print()

Parameters: n/a

Return Type: string

Rationale: This function calls gatherClassifications() to collect all classifications needed for print and builds the string to print using underscores between classifications.

Function: bool Trie::erase(const std::string& classification)

Parameters: classification (path to erase)

Return Type: bool (true=success)

Rationale: This function splits the valid input into levels and then traverses trie. It keeps track of the parent node and last index while finding the target node to erase. Once found, it validates that its terminal, sets to non-terminal and removes from parent if node has no children then updates count.

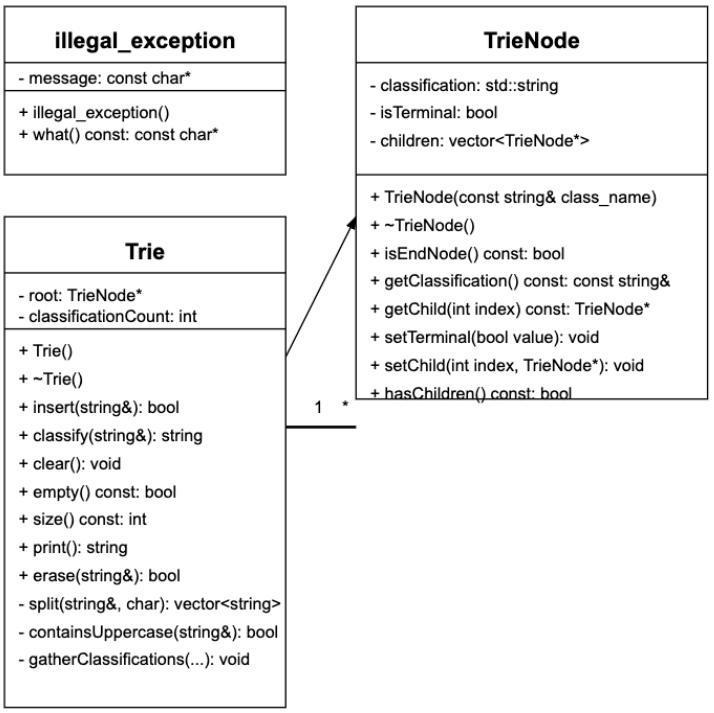
Function: TrieNode::TrieNode(const std::string& class_name)

Parameters: class_name

Return Type: n/a

Rationale: Initializes a new trie node with all necessary components. It stores the classification string as well as initializes the terminal status to false since new nodes are not yet complete, this can be changed and will later be used to differentiate between complete and partial classifications.

UML Diagram



Runtime Analysis

1. split function: O(n) where n is string length
   Single pass through string of length n, vector push_back operations are amortized O(1)
   Total: O(n)
2. insert: O(n) where n is number of classification levels
   For each n: Check children (O(1) * max 15 = O(1)) AND Create new node if needed (O(1))
   Total: O(n)
3. classify: O(N) where N is total nodes in trie
   At each level: Gather child classifications (O(1) * max 15 = O(1)) AND Process LLM result (O(1)) AND Follow path (O(1)), where **Max depth = O(N) in worst case**
   Total: O(N)
4. erase: O(n) where n is classification levels
   Find target node: O(n) AND Remove node and cleanup: O(1) AND Only traverse given path
   Total: O(n)
5. print: O(N) where N is total nodes
   Visit every node once: O(N) AND String operations at each node: O(1)
   Total: O(N)
6. clear: O(N) where N is total nodes
   Must visit and delete each node: O(N)
   Total: O(N)
7. empty and size: O(1)
   Return a stored value: O(1)
   Total: O(1)