Class Design

For this project, I used 2 main classes: TrieNode and Trie. I separated them because each node needs to manage its own children and data, while the trie class handles the overall operations like insert and search. I could have put everything in one class but splitting them made the code cleaner and easier to debug.

TrieNode: stores the class name, a boolean for whether it's terminal, and an array of 15 child pointers. I chose a fixed array of 15 instead of a vector because the design doc says no class will have more than 15 subclasses, so there's no point in using dynamic sizing. Using an array makes it faster to access children by index too. The terminal flag is important because it tells us if this node represents an actual classification or just an intermediate node in the path.

Trie: the main class that has the root node and tracks the total count of classifications. I keep a count variable so the SIZE command is O(1) instead of having to traverse the whole tree every time. The trie handles all the operations like inserting, searching, classifying, and printing.

The big challenge was figuring out the "internal nodes can't be terminal" rule from the doc. At first I had a bug where if you inserted "living" and then "living,mammal", it would print both "living" and "living,mammal". But the doc says an internal node (one with children) can't be terminal. So I had to make sure that when printing, I only print nodes that are terminal AND have no children. Tricky tricky, took me a while and a couple test runs to get right.

UML Diagram

Trie	TrieNode
- root: TrieNode* - classificationCount: int	- children: TrieNode*[15] - isTerminal: bool - className: string
+ Trie() + ~Trie() + load(filename: string): void + insert(classification: string): bool + classify(input: string): string + erase(classification: string): bool + print(): void + isEmpty(): bool + clear(): void + size(): int - printHelper(node: TrieNode*, current: string, results: vector<string>&): void - clearHelper(node: TrieNode*): void	+ TrieNode() + TrieNode(name: string) + ~TrieNode() + getChild(index: int): TrieNode* + setChild(index: int, child: TrieNode*): void + getTerminal(): bool + setTerminal(terminal: bool): void + getClassName(): string + findChildIndex(name: string): int + hasChildren(): bool

Trie contains TrieNode

illegal_exception

Empty class for illegal arguments (uppercase letters)

Function Design

insert(classification: string): This function parses the comma-separated classification string into individual class names, then traverses down the trie creating nodes as needed. The tricky part was handling the terminal flag correctly. I traverse to where the classification should be, create any missing nodes along the way, then mark the final node as terminal. Returns false if the classification already exists or if there's no space for more children.

classify(input: string): This one was interesting because it uses the language model classifier. It starts at root and keeps going down the tree. At each level, it collects all the child class names, sends them to the classifier, gets back the best match, and moves to that child. It stops when there are no more children to check. The classifier part is handled by the provided ece250_socket library so I just had to call labelText() with the right parameters.

erase(classification: string): Deletes a classification by traversing to it, marking it as non-terminal, then cleaning up any orphaned nodes. The cleanup part was tricky, I had to work backwards from the deleted node up to the root, deleting any nodes that have no children and aren't terminal. If I don't do this cleanup, you get memory leaks that were shown by valgrind giving me a headache and useless nodes sitting around.

print(): Uses a recursive helper function to traverse the entire trie and collect all terminal classifications. The key thing here is in printHelper(), I only add a classification to results if the node is terminal AND has no children. This implements the "internal nodes can't be terminal" rule. Then it prints everything with underscores between classifications and a trailing underscore at the end (based on the print command note).

findChildIndex(name: string): Goes through all 15 children looking for one with a matching class name. Returns the index if found, -1 otherwise. This is used a lot during traversal so it needs to be simple and reliable.

Runtime Analysis

Assumptions: For any classification path, we have at most n classes (like "living,from plants,food" has n=3). The number of children at any level is at most 15, which is O(1).

INSERT: Parsing the string with getline() takes O(n) where n is the number of classes. Then we traverse down the trie, and at each level we call findChildIndex() which loops through at most 15 children (O(1)). We do this n times for n levels, so traversal is O(n * 1) = O(n). Creating a new node and marking it terminal are both O(1) operations. Total is O(n).

CLASSIFY: At each level, we collect child names by looping through 15 children (O(1)), then call the classifier (doc says don't count this), then traverse to the next level. We do this for at most n levels. So it's O(n * 1) = O(n). The doc says to ignore the classifier runtime which is good because it's probably terrible.

ERASE: We parse the string in O(n), traverse to the node in O(n) using findChildIndex at each level, mark it non-terminal in O(1), then cleanup. The cleanup loop goes through the path

backwards which is at most n nodes, and for each one we loop through 15 children to find and delete it (O(1) per node). So cleanup is O(n). Total is O(n).

PRINT: This one uses recursion to visit every node in the trie. In the worst case we have T total classifications and each one has an average depth of d. The recursion visits each node once, and at each node we check up to 15 children. If there are N total nodes in the trie, we visit N nodes and do O(1) work at each. The doc says O(n) where n is the number of classes, so if we consider that printing each classification involves building its string (which is at most d comma-separated names), and we have T classifications, it's O(T * d). But since the doc just says O(n), I think they mean n = total number of class names in all classifications combined, which matches O(T * d).

SIZE & EMPTY: Both are O(1) because I keep a counter variable that tracks the number of classifications. SIZE just returns it and EMPTY checks if it's zero.

Citation: Used ChatGPT (chat.openai.com) for help with understanding istringstream for parsing comma-separated strings, recursive tree deletion in destructors, and exception handling with try-catch blocks. Accessed November 2025.