Design Document Project 3 ECE 250

Class Design

Node Class

Private Variables: std::string classification, std::vector<Node*> children, bool terminalFlag

The Node class represents a single node in a trie structure, with the responsibility of storing classification labels, maintaining child nodes, and marking whether it is a terminal node or not. The use of std::vector for children allows for dynamic resizing, providing flexibility in the number of child nodes. This is more memory efficient over alternative choices like using a fixed size array.

Public Methods: The constructor initializes a node with a classification label and sets the terminalFlag to false. The destructor deletes all child nodes to prevent memory leaks. Key methods include getClassification(), getChildren(), and isTerminal(), with modifiers like addChild(), removeChild(), and setTerminal(). The getNumChildren() method returns the number of child nodes. This class ensures efficient trie operations while maintaining integrity during insertion, deletion, and traversal. By keeping these variabls private we ensure the tree is not changed unless done in a controlled way through the functions created.

Trie Class

Private Variables: Node* root, int classificationCount

Private Methods: DFS_PrintPaths, removeSingleLeadingSpace, containsUppercase

removeSingleLeadingSpace and containsUppercase are private functions since they are only ever called within other functions. This is also beneficial to make them private since it controls the ability of the user to change the value of input strings from their original state which can affect how the tree is structured.

Public Methods:

The Load and Insert functions both split a classification string and traverse the trie to add nodes. Load adds classifications, while Insert also handles invalid input exceptions. Both functions create new nodes when necessary, but Insert marks the end node as terminal only if it's a new insertion, this ensures that a insertion that is not at the end of the path is not marked as terminal accidentally and ensures that the tree is properly updated. Splitting the string inside the functions rather than in a helper or main keeps the logic self-contained, reduces complexity of the main function, and ensures better encapsulation of trie insertion operations.

The Erase function removes a classification by traversing the trie and locating the node, ensuring it is terminal before deleting it. A distinct design choice is the manual traversal of child nodes through a loop using two pointers. This allows for control over node deletion and updating the parent-child relationships. Another notable design decision is marking the parent node as terminal if it has no remaining children, which maintains the integrity of the trie structure and prevents orphaned, non-terminal nodes.

The PrintAllPaths function uses recursion with depth-first search (DFS) style implementation to traverse the trie and print each path from root to terminal node. DFS ensures all paths are explored without needing to manage multiple queues, making it more memory-efficient than other options such as breadth-first search (BFS). The vector is used to store the path as it grows during traversal, providing dynamic resizing without extra memory cost. This method is more efficient than iterative approaches, as it avoids unnecessary data structure manipulation while maintaining a clear traversal of the trie. The DFS function

itself is private since it should only be called from the print paths function to ensure that the proper checks are done first.

The Classify function traverses the trie to classify the input string by matching it to the appropriate classification path. It uses a while loop, iterating through the node children to match classifications, and ensure no uppercase letters are present. If a valid classification is found, it prints the path of the classification.

Illegal_Exception Class: used to catch any uppercase letters in a string, works alongside a helper function in the tree class that detects if there is an uppercase based on ascii values.

Runtime Analysis

PRINT:

- The runtime of PrintAllPaths is determined by the recursive DFS_PrintPaths function. The PrintAllPaths function itself has an O(1) runtime since it performs only constant-time operations before calling the recursive function, DFS_PrintPaths.
- Once in the recursive function, the first check is to see if the current node is terminal, if it is not, all the children of the node are traversed to find a match, then search is recursively called on the matching child. For one traversal of a node the runtime is O(c) where c is the number of children which can be approximated to O(1).
- This traversal can be repeated at worst case n times, once for classification in the trie.
- If the current node is terminal, the path is printed which is a time O(n), where n is as bad as the number of classifications in the tree. Therefore, the **runtime is O(n)** for the function.

INSERT:

- The splitline part of the code has a runtime of O(n) since it reads each element in the classification once.
- Then we traverse the children of a node, since the number of children per node is capped at 15, this operation has a constant-time cost of O(1). Repeating this search for up to n labels results in O(n) for the traversal process where n is the number of elements in the classification.
- All other elements have constant runtime O(1).
- So, we can conclude the runtime components are O(n) + O(n) and so, the **overall runtime simplifies to O(n).**

EMPTY: Has a **runtime of O(1)** since it only performs constant time operations such as checking the size of the vector. This is because std::vector internally maintains the size of the vector as a separate data member, which is updated whenever elements are added or removed. Therefore, accessing this stored size does not require iterating through the elements, it's a simple retrieval of a stored value.

SIZE: Has a **runtime of O(1)** since it only performs a constant time operation to print out a stored value.

CLEAR: The runtime of the Clear function is determined by the deletion of all nodes in the trie, triggered by delete root, which has a complexity of O(N). Resetting the root, updating the classification count, and printing are all O(1) operations. Therefore, the **overall runtime of the Clear function is O(N)**

CLASSIFY:

Time Complexity for Each Component:

Removing space, checking for uppercase: we assume to be O(1) as specified in the lab

Traversing the Trie for the first time: O(c) where c is the maximum number of children (approximated to constant, O(1) since children capped at 15). This is because it checks all the children for one node at a level and records them in a string(insertion into path is O(1)) to be passed to the LLM.

Traversing the Trie again: O(c), check all children of the current node to ensure the output of the LLM matches one of the valid options.

Repeat: we repeat these traversals of children nodes up to a maximum of N times, and such is O(N). And such can be up to O(N), as N is the number of classifications in the classification.

In conclusion: the function runs in O(N +N) since c is constant due to it being capped at 15.So, this **simplifies to O(N)**.

ERASE:

Checking for Uppercase and Removing Leading Space: we assume to be O(1) as specified in the lab.

Traversing the Trie: The traversal works to Locate the node corresponding to the classification path.

- Splitting the string: O(n), where n is the number of elements in the string classification.
- Traversing each level: O(c) per level, where c is the maximum number of children (approximate to O(1) since children capped at constant 15).
- Total traversal: O(d * c=1), where d is the depth of the trie.
- Since c is constant and we approximate depth to be constant, this simplifies to O(1).

Removing the Node: Updates the parent's child list and deletes the node.

- Loops through the parent node's children vector to find and remove the child corresponding to currentNode. O(c) ~= O(1)
- Uses the removeChild function, which involves traversing the vector of children and potentially resizing it. O(c) ~= O(1)

In conclusion: **Total runtime is O(n),** where n is the length of the classification string.