# 1 Class Design

`Trie` has public methods corresponding to each of the program commands, and the data store representing the root node of the trie.
To safetly deconstruct `Trie`, `deleteNode` is called at the root of `Trie`.
Private member variables are `root` and `classificationCount`. `root` is private to ensure the ordering of subtrees is encapsulated by `Trie`. `classificationCount` is private to ensure the state of all classifications is encapsulated by `Trie`.

`TrieNode` has member methods and variables to represent a trie node.
**Constructor overloading:** primary constructor with no arguments is reserved for initializing the root node. Secondary constructor with argument `val` is used to initialize a non-root node with initial string value.

For each node, its children are stored in a vector of `TrieNode` objects. Due to the vector data structure's direct compatibility with iterators and automatic memory management, this simplifies the recursive implementation of `deleteNode` while avoiding unsafe memory access.

# 2 Function Design

**TrieNode class**
associated trie node methods - `addChild, findChild, removeChild`
params - `const string &childValue`
`addChild` and `findChild` return `TrieNode*`, the pointer to the child node that was added or found. `removeChild` returns void.

**Trie class**
associated program command methods - `insert, classify, erase, print, printEmpty, clear, printSize`
params - data types corresponding to I/O requirements table
all methods return `void` except for `insert`, which returns enum `ReturnCodes`. `insert` is used in the LOAD command to load classes into the trie line-by-line.
`deleteNode`
params - `TrieNode *node`
returns - `void`
Deletes the subtree rooted at that node, then deletes the node.
`pathStringBuilder`
params - `TrieNode *node, string path`
returns - `void`
Recursive function responsible for printing all classifications in the trie, rooted at `node`.
`classificationLabelBuilder`
params - `const vector<TrieNode *> childClassifications`
returns - `string`
Builds the comma-separated candidate labels using each of the values from `childClassifications`. Returns the completed string.

**illegal_exception class**
associated argument checker methods - `validateNoIllegalClassification, validateNoIllegalInputString`
returns - `bool`

# 3 Runtime

**Proof of O(n) for INSERT and ERASE, for n classes in the classification**
Proof of `O(n)` for trie traversal given a classification:
For each class in the classification, start at the root node and traverse down to the corresponding child node in `Trie`. This loop is done in `O(n)`. The lookup operation to check if the child exists is done in `O(1)`, since each node can have up to only 15 children.
`INSERT`: if the current node does not have any children, add a new child to that node and resume traversal.
`ERASE`: uses two variables to keep track of the child and parent node. Traversal is done so that the parent can safely delete the child.
Thus, the runtime for both methods is `O(n)`.

**Proof of O(N) for CLASSIFY, for N classes in the trie**
`Trie` implements an unbalanced tree. The worst case scenario is when only a single classification is inserted, and the tree degenerates into a linear list with N elements. Traversing to the end of the list to complete the classification takes `O(N)`.
The dominating runtime is the worst-case runtime, which is `O(N)`.

**Proof of O(N) for PRINT, for N classifications in the trie**
Upon successful insertion, the number of nodes and the number of classifications are both incremented by 1. To prove `O(N)`, we can assume N is also the number of nodes in the trie.
Consider the worst-case when the tree degenerates into a linear list with N elements. `pathStringBuilder` does comparison operations to build the comma separated classification. This is done in `O(1)`.
**Recursive call:** for the worst-case, every node has one child. For each node, `pathStringBuilder` is called on a subproblem of

size $N-1$.
The recurrence relation is $T(N) = O(1) + T(N-1)$, where $T(N-1)$ is the time complexity for the $N-1$ remaining nodes.
Thus, by **A.** the running time of `pathStringBuilder` is `O(N)`.

**Proof of O(1) for EMPTY, SIZE**
The associated method contains one print operation, which is done in `O(1)`.

**Proof of O(N) for CLEAR, for N nodes in the trie**
Consider the worst-case when the tree degenerates into a linear list with N elements. `deleteNode` is called at the root of the trie.
**Recursive call:** for the worst-case, every node has one child. For each node, `deleteNode` is called on a subproblem of size $N-1$.
The recurrence relation is $T(N) = O(1) + T(N-1)$, where $T(N-1)$ is the time complexity for the $N-1$ remaining nodes.
Thus, by **A.** the running time of `deleteNode` called at root is `O(N)`.

**A. Solving the recurrence relation $T(N) = T(N-1) + O(1)$ by repeated substitution**
Define the base case, $T(0) = O(1)$
For 3 substutions, $T(N) = T(N-1) + O(1) = T(N-2) + 2O(1) = T(N-3) + 3O(1)$
After k substitutions, $T(N) = T(N-k) + kO(1)$
Substitutions are continued until the base case, where $k = N$
$T(N) = T(0) + NO(1) = O(1) + NO(1)$
By asymptotic analysis and big-Oh notation, $T(N) = O(N)$
Thus, the reccurence relation has a running time of `O(N)`.

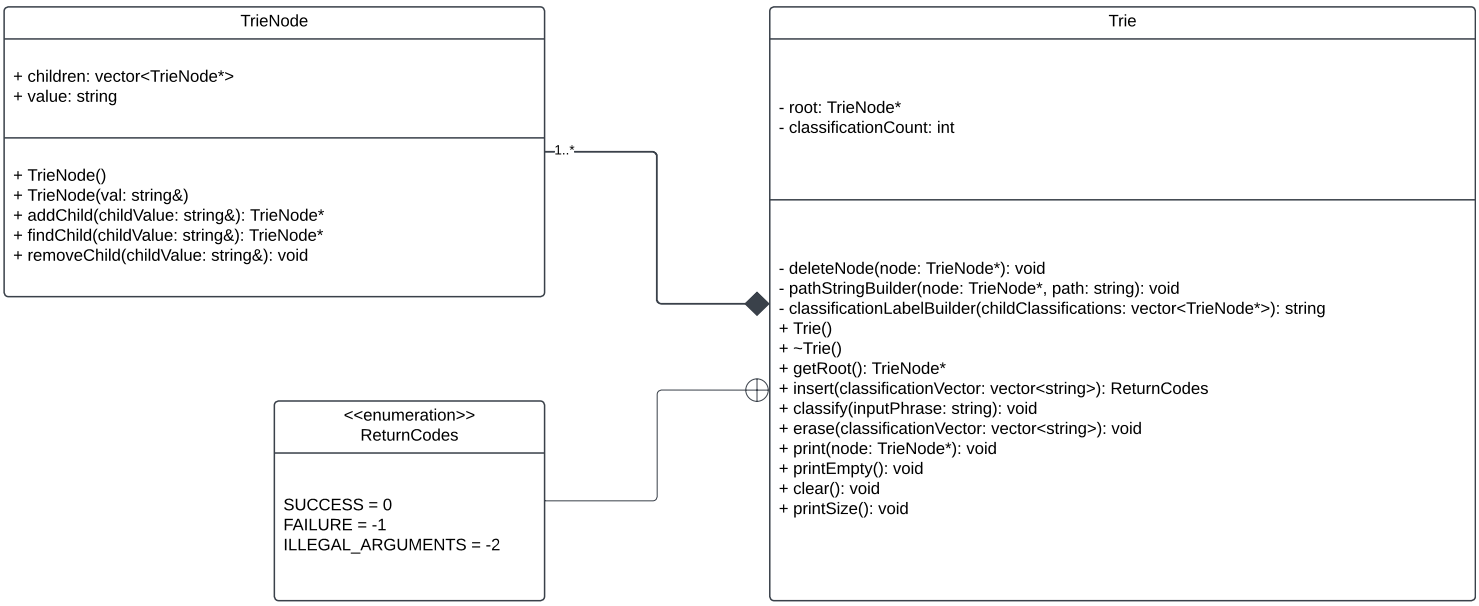


Figure 1: UML Class Diagram