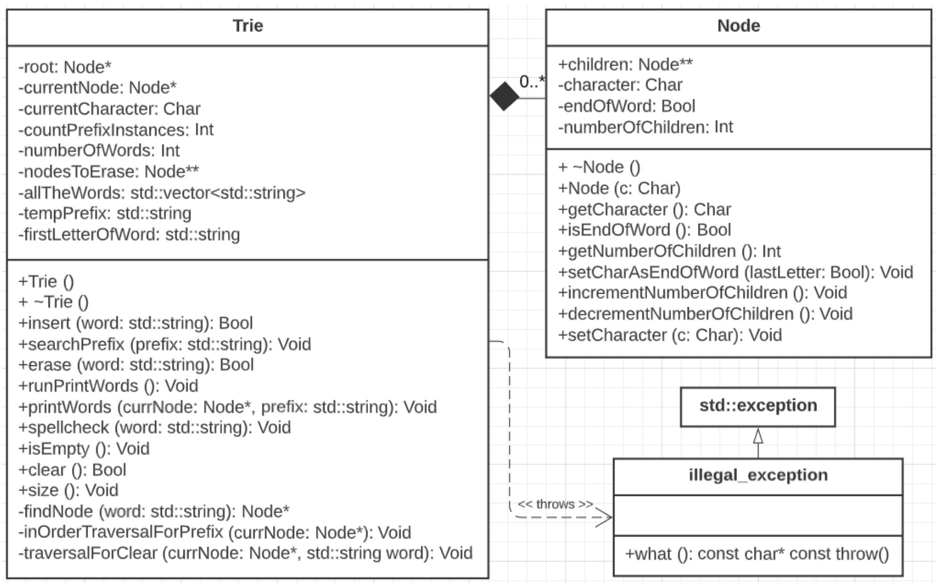
Design Document | ECE 250 Project 3: Spell Checker Using Trie ADT

Azizul Chowdhury | a48chowd@uwaterloo.ca | 20893907

UML Diagram


Trie
-root: Node*
-currentNode: Node*
-currentCharacter: Char
-countPrefixInstances: Int
-numberOfWords: Int
-nodesToErase: Node**
-allTheWords: std::vector<std::string>
-tempPrefix: std::string
-firstLetterOfWord: std::string
+Trie ()
+ ~Trie ()
+insert (word: std::string): Bool
+searchPrefix (prefix: std::string): Void
+erase (word: std::string): Bool
+runPrintWords (): Void
+printWords (currNode: Node*, prefix: std::string): Void
+spellcheck (word: std::string): Void
+isEmpty (): Void
+clear (): Bool
+size (): Void
-findNode (word: std::string): Node*
-inOrderTraversalForPrefix (currNode: Node*): Void
-traversalForClear (currNode: Node*, std::string word): Void
0..*
Node
+children: Node**
-character: Char
-endOfWord: Bool
-numberOfChildren: Int
+ ~Node ()
+Node (c: Char)
+getCharacter (): Char
+isEndOfWord (): Bool
+getNumberOfChildren (): Int
+setCharAsEndOfWord (lastLetter: Bool): Void
+incrementNumberOfChildren (): Void
+decrementNumberOfChildren (): Void
+setCharacter (c: Char): Void
std::exception
<< throws >>
illegal_exception
+what (): const char* const throw()

Constructors

`Trie` class:

- **`Trie ()`**: Default constructor for a Trie. Initializes the Trie with a root node and a counter, `numberOfWords`, which is initialized as zero.

`Node` class:

- **`Node (char c):`** Constructs a node for a given character in the uppercase English alphabet. The node stores its character. The member variable `isEndOfWord` is initialized as false; `numberOfChildren` is initialized as zero; and an array containing the children of the node is initialized, where each entry in the array is `nullptr`. During insertion, these entries are replaced with the addresses of the appropriate nodes.

Destructors

`~Trie ():` Calls the member function `clear()` to clear the entire Trie. The root node is also deallocated in this destructor.
`~Node ():` Deallocates the `children` array of the given node.

Member Functions

`Trie` class:

The following functions are **`public`** as they are called from outside the class (in the `main` function).

- **`bool insert (std::string word):`** Used to insert words into the Trie. If any character in the word is not an uppercase character in the English alphabet, an `illegal_exception` is thrown. If a given word already exists in the Trie, the function returns a failure (`false`). This is checked by calling the helper function `findNode()` which returns the last node comprising the word if the word is found, or `nullptr` if the word is not found. If the last node is returned, we check whether it terminates a word using the getter function `isEndOfWord()`. To insert a word, we begin at the root of the tree, and attempt to traverse down the tree by following the path of the characters that comprise the word. If at any point during this traversal, we find that an expected child node is `nullptr` (meaning, the character isn't in the Trie at that position), then a valid child node is inserted for that letter and the number of children of the current node is incremented. This process is repeated until the last letter of the word is inserted. The node for the last letter is marked as the end of the word using the setter function `setCharAsEndOfWord()`. The function returns `true` (success) if the word is successfully inserted.
- **`void searchPrefix (std::string prefix):`** Outputs a count of all the words in the Trie starting with the given prefix. `illegal_exception` is thrown if any character in the given prefix is not an uppercase character in the English alphabet. Just as with `insert()`, the helper function `findNode()` is used to determine whether the given prefix exists in the Trie. If the prefix exists in the Trie, the helper function `inOrderTraversalForPrefix()` is used to traverse the Trie and count all the words with the prefix. The count is stored in a member variable, `countPrefixInstances`, which is outputted once the traversal is complete.
- **`bool erase (std::string word):`** Used to erase a word from the Trie. `illegal_exception` is thrown if any character in the given prefix is not an uppercase character in the English alphabet. The helper function `findNode()` is used to determine whether the given word exists in the Trie. There are four cases in erasing words. The first case is when the word being erased is a prefix of another word in the Trie, e.g. if "BAKE" and "BAKERY" were both in the Trie and "BAKE" was being erased. In this case, we locate the node corresponding to the last letter in "BAKE" using `findNode()` and we use the setter function `setCharAsEndOfWord()` to stop marking it as the end of a word. There are three cases when the word being deleted isn't the prefix of another word. In all of these cases, we traverse down the tree, following the path of the word being erased, and accumulate all the nodes that comprise the word being erased into a new array. We then iterate backwards from the end of that array and attempt to delete all the nodes in that array, unless certain stopping conditions are met. The first stopping condition is if we reach a node which has more than one child – we don't want to delete letters that are shared by other words in the Trie, so we stop deleting. The second stopping condition is if we reach a node which is marked as the end of the word – then we must stop deleting, as otherwise we would delete another word. Finally, if neither of these scenarios are encountered, we would delete all the nodes in the array, as they only belong to the word being erased. The function returns `true` if the erasure was successful; `false` if the word isn't in the Trie to begin with (failure).

- **`void runPrintWords ():`** Used when the command `p` is executed. Calls `printWords()` with the root node as the argument for the `currNode` parameter, and an empty string as the argument for the `prefix` parameter. `printWords()` is used when spellcheck is executed as well, hence, it was appropriate to create a separate function for handling how `printWords()` should be used when the command `p` is executed.
- **`void printWords (Node *currNode, std::string prefix):`** Prints all the words in a given subtree (or in the entire Trie) recursively. When first called, the argument for `currNode` defines which node should be treated as the root – all words beneath this root are printed. An empty string is passed in as the `prefix` when this function is first called. There are two base cases: the first is when the current node is `nullptr`, in which case we return; the second is when the current node terminates a word, in which case we print the value accumulated in `prefix`, which would be the word whose last node is the current node. Besides the base case, we use a for loop to traverse through all the subtrees of the current node recursively. When we encounter a child of the current node, we accumulate the character associated with that child to the `prefix` variable. We then recursively call `printWords()` with that child node as the first argument, and the updated `prefix` as the second. When that function call returns, we remove the last letter in `prefix`.
- **`void spellcheck (std::string word):`** Used to spell-check the given word. Using the helper function `findNode()`, we check if the given word already exists in the Trie, in which case we print "correct". Using `findNode()`, we also check whether the first letter of the given word exists in the Trie – if it does not, there are no suggestions that can be given, hence a blank line is outputted. Otherwise, we traverse down the Trie and follow the path of the letters that comprise the given word. If at any point during this traversal, a letter of the word is missing from the Trie, we use `printWords()` to print all the branches of the last node whose letter is in the given word. If the given word is the prefix of other words in the Trie, we use `printWords()` to print all the words that have the given word as their prefix. If the given word was only one letter, we use `printWords()` to print all the words that start with that letter.
- **`void isEmpty ():`** Prints whether or not the Trie is empty. The member variable `numberOfWords` keeps track of how many words are in the Trie. If it is is equal to zero, "`empty 1`" is outputted; otherwise, "`empty 0`" is outputted (meaning the Trie is not empty).
- **`bool clear ():`** Used to clear the entire Trie. If the Trie isn't empty, the helper function `traversalForClear()` is used to accumulate all the words in the Trie into one vector. We then iterate through that vector, starting at its end and iterating backwards, and call the member function `erase()` on each entry in the vector to delete each word from the Trie. As each word is deleted, it is also removed from the vector. Function always returns true (success).
- **`void size ():`** Prints the number of words in the Trie, using the value stored in the member variable `numberOfWords`.

The following functions are **`private`** as they are helper functions only used from within the class.

- **`Node *findNode (std::string word):`** Traverses down the Trie, starting at the root, by following the path of the letters that comprise the given word. If at any point during this traversal, an expected letter of the word is missing from the Trie, `nullptr` is returned, as the given word is not in the Trie. If the traversal reaches the end of the word, the node with the last letter of the word is returned.
- **`void inOrderTraversalForPrefix (Node *currNode):`** An In-Order Traversal used to count the number of words that start with the given prefix. This function is implemented recursively. The base case is when the current node is not a valid node, in which case we return. In the first case, we attempt to traverse the left subtree of the current node by checking if the left-most child is a valid node (or in other words, whether the letter 'A' is a child of the current node). If this is true, we recursively call `inOrderTraversalForPrefix()` with the left-most child of the current node as the argument. If this isn't true, we move onto the second case, which is where we check if the current node is the end of a word, in which case we increment the member variable `countPrefixInstances`. If the current node is not the end of a word, we move onto the third case, where we traverse all the right subtrees of the current node by recursively calling `inOrderTraversalForPrefix()` for each child node.
- **`void traversalForClear (Node *currNode, std::string prefix):`** This function recursively traverses the entire Trie and accumulates all the words in the Trie into a vector. There are two base cases: the first is if the current node is `nullptr`, in which case we return; the second is if the current node terminates a word, then we add the value stored in the `prefix` string to the end of the vector. Besides the base case, we use a for loop to traverse through all the subtrees of the current node recursively. When we encounter a child of the current node, we accumulate the character associated with that child to the `prefix` variable. We then recursively call `traversalForClear()` with that child node as the first argument, and the updated `prefix` as the second. When that function call returns, we remove the last letter in `prefix`.

`Node` class:

All member functions are **`public`** as they are getter functions called from outside the class.

- **`char getCharacter ():`** Returns the character of the given Node.
- **`bool isEndOfWord ():`** Indicates whether or not a given Node terminates a word. Returns true if it does; false if it doesn't.
- **`int getNumberOfChildren ():`** Returns the number of children the given Node has.
- **`void setCharAsEndOfWord (bool lastLetter):`** Used to mark a Node as the end of a word.
- **`void incrementNumberOfChildren ():`** Used to increment the number of children a Node has. Used when new words are inserted.
- **`void decrementNumberOfChildren ():`** Used to decrement the number of children a Node has. Used when a word is erased from the Trie.
- **`void setCharacter (char c):`** Used to set the letter associated with the Node.

`illegal_exception` class:

This class inherits from the `std::exception` class. The following function is **`public`** as it is called from outside the class:

- **`const char *what() const throw():`** An overwrite of the `what()` function from the `std::exception` class. Returns the string "`illegal argument`" when an exception is thrown.

Member Variables

For both the `Trie` and `Node` classes, all the member variables are **`private`** to ensure safety, as they do not need to be modified outside their respective class. The only exception to this is the `Node **children` member variable of the Node class, which is **`public`**. All the Node pointers stored in this array have to be modified using getter and setter functions anyway, so it was convenient to make this array public to make its indices accessible from the Trie class. The `illegal_exception` class has no member variables of its own beyond what it inherits from the `std::exception` class.

Runtime

For the following runtimes, "n" represents the number of characters in a word and "N" represents the number of words in the Trie.

Helper functions:

- **`Node *findNode (std::string word):`** Traverses the Trie, following the path of the nodes that comprise the given word. The traversal only visits one node for each character in the string, and the total number of nodes visited is equal to the number of letters in the word (at worst case, which is if the word exists in the Trie). Thus the runtime is **O(n)**.
- **`void inOrderTraversalForPrefix (Node *currNode)`** ***and*** **`void traversalForClear (Node *currNode, std::string prefix):`** Both of these functions implement traversals where each node in the Trie is only visited once. Equivalently, each word in the Trie is only visited once. Thus the runtime is **O(N)**.

Primary functions:

- **`bool insert (std::string word)`** *`(Associated with "i")`***`:`** This operation begins with a linear search (O(n)) to verify that the given word only contains uppercase letters from the English alphabet. It then calls the helper function `findNode()`, whose runtime is O(n), to check whether the word already exists in the Trie. Finally, if the word isn't in the Trie, it is inserted by traversing down the Trie, following the path where the word should be, and inserting nodes for each character in the word at the appropriate locations. Since one node is inserted per character, the runtime for insertion is O(n).
  - Thus, the overall runtime for this function is given by O(n) + O(n) + O(n) = **O(n)**.
- **`void searchPrefix (std::string prefix)`** *`(Associated with "c")`***`:`** This operation begins with a linear search (O(n)) to verify that the given prefix only contains uppercase letters from the English alphabet. `findNode()`, whose runtime is O(n), is used to check whether the prefix already exists in the Trie. If the prefix exists in the Trie, the helper function `inOrderTraversalForPrefix()`, whose runtime is O(N), is used to count the number of instances of this prefix.
  - Thus, the overall runtime for this function is given by O(n) + O(n) + O(N) = **O(N)**.
- **`bool erase (std::string word)`** *`(Associated with "e")`***`:`** This operation begins with a linear search (O(n)) to verify that the given word only contains uppercase letters from the English alphabet. `findNode()`, whose runtime is O(n), is called to obtain the last node of of the given word; this is used to check whether the word is in the Trie and whether the word is the prefix of another word. A for loop is used to traverse down the Trie, following the path of the given word, and accumulate all the nodes that comprise the word into an array – the runtime of this is O(n) as only the nodes of the word are visited, and they are all only visited once. Another for loop is used to iterate backwards from the end of that array, to delete all the nodes (until certain stopping conditions are met); the runtime of this is also O(n).
  - Thus, the overall runtime for this function is given by O(n) + O(n) + O(n) + O(n) = **O(n).**
- **`void printWords (Node *currNode, std::string prefix):`** *`(Associated with "p")`***`:`** This function is the main component of the "p" command as `runPrintWords()` is a driver function that runs in constant time. In this function, the entire Trie is traversed recursively, and every node is visited once. Equivalently, every word in the Trie is visited once and only once. Thus the runtime is **O(N)**.
- **`void spellcheck (std::string word):`** *`(Associated with "spellcheck")`***`:`** This function begins by calling `findNode()`, whose runtime is O(n), to obtain the last node of the given word – this is used to check whether the word exists in the Trie. A for loop is used to traverse down the Trie, following the path of the nodes that comprise the given word, until either the end of the word is reached, or until an expected letter of the word is missing from the Trie. This traversal is O(n) as each node comprising the word is only visited once. At the end of each traversal, `printWords()`, whose runtime is O(N), is called to print out suggested words when the word is missing from the Trie.
  - Thus, the overall runtime for this function is given by O(n) + O(n) + O(N) = **O(N)**.
- **`bool clear ()`** *`(Associated with "clear")`***`:`** If the Trie isn't empty, this function calls `traversalForClear()`, whose runtime is O(N), to accumulate all the words in the Trie into a vector. It then iterates through that vector and deletes every word from the Trie by calling `erase()` on each word. Since the runtime of erase() is O(n) and since N words are deleted, the runtime of deleting all the words is N*O(n) .
  - Thus, the overall runtime for this function is given by O(N) + N*O(n) = **O(N)**.
- **`void isEmpty ()`** *`(Associated with "empty")`***`:`** Checks if `numberOfWords = 0` and prints a statement accordingly. Runtime is **O(1)**.
- **`void size ()`** *`(Associated with "size")`***`:`** Prints the value of a member variable, `numberOfWords`. Runtime is **O(1)**.

Design Decisions

Initially, when each Node was constructed, I was filling its array of children nodes with 'invalid nodes' (nodes marked as 'invalid' using a boolean member variable). I later realized that this was a bad decision as deconstructing the Trie and avoiding memory leaks was next to impossible. Thus, I treated un-inserted nodes as null pointers instead, which made it easier to mitigate memory leaks. For traversing the Trie, I used an In-Order Traversal for the "c" command, and for other commands, I used a regular traversal where all the subtrees are visited. I was interested in learning how to apply an In-Order Traversal, as it is something we learned in class, so for that reason I used it for one of the operations.