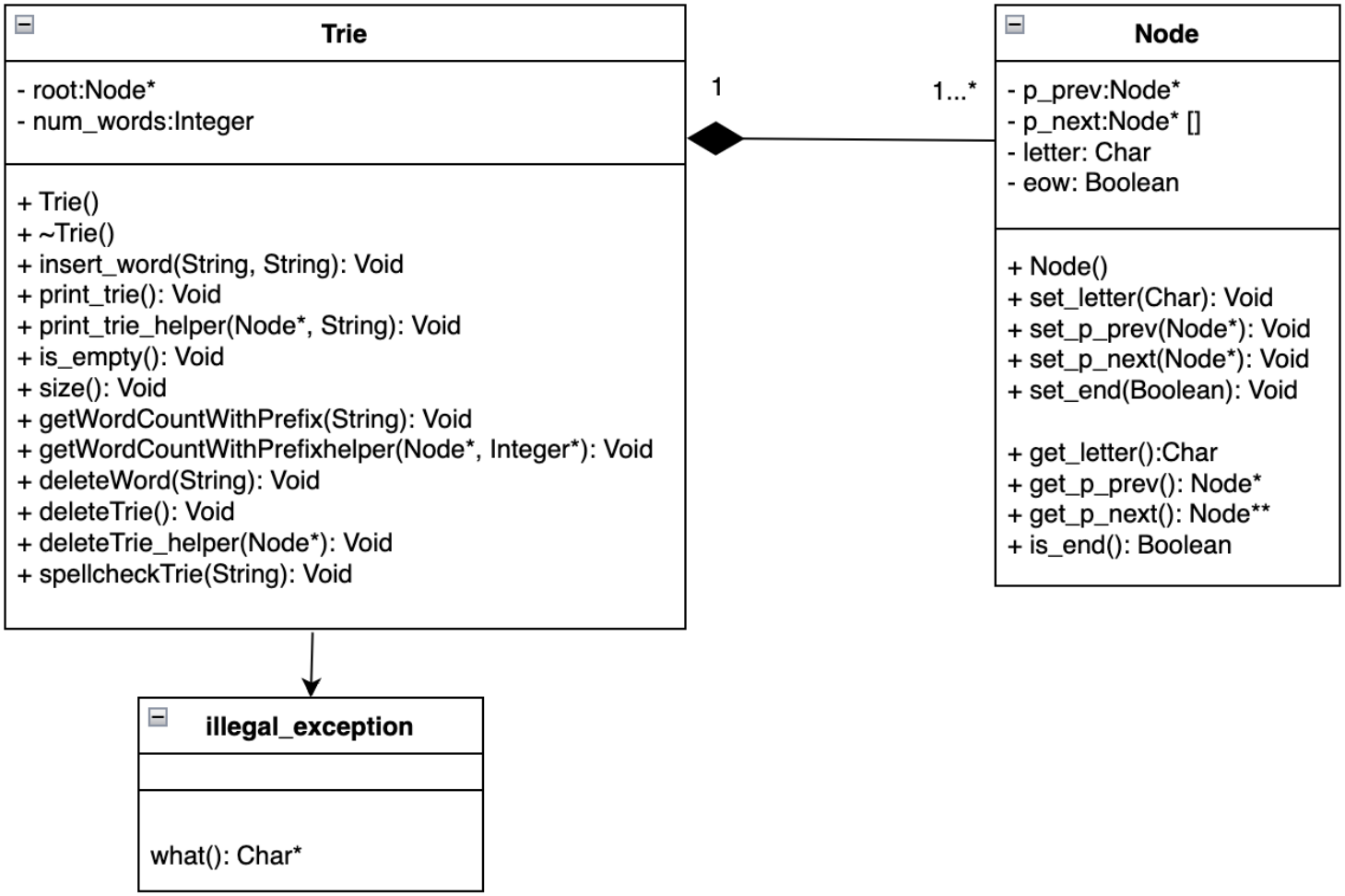
The trie data structure is implemented using the **Trie** class, and the Node class represents a single node in a Trie. Finally, there is a custom exception class called **illegal_exception** which inherits from the standard exception class.

UML Diagram:



Illegal_Exception Class - This class is used to throw exceptions when there is an illegal argument, such as entering a word without uppercase letters. It has a function what() which returns the string "illegal argument" so it can be returned in the program. Note: As I was unfamiliar with exception classes, I have adopted a variation of the implementation from https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/.

Node Class - There are 4 private member variables:

- **p_prev**: Stores a pointer to the parent of the node.
- **p_next**: Stores an array of pointer to children nodes. It is initialized to nullptr, indicating that there are no child nodes to a given node.
- **letter**: For ease, each node stores the letter or value of node.
- **eow**: It's a boolean which represents whether this node is the end of word or not.
  Node, the constructor, initializes the parent of the node as nullptr, letter as an empty character, and the end of the word is false. There is no destructor as it is not needed. Since all variable are private, there are appropriate getter and setter functions for each node.

Trie Class - There are 2 private member variables:

- **root**: Stores a pointer to the root node of the trie.
- **num_words**: Stores the number of words inserted into the trie.

Trie, the constructor, initializes root to be an instance of the Node class. Additionally, since no words have been added to the trie, the total number of words is 0.
~Trie, the destructor, runs the **deleteTrie** function. At the end, we delete the root node and set it to nullptr.

insert_word takes in two parameters

1. type string which stores the word to be inserted in the trie and,
2. type string which indicates whether any status messages are required i.e. success/failure

- Before inserting word into the trie, it checks if every letter is in uppercase. If not, it throws an illegal exception.
- Next, we iterate through each letter in the word and check if the letter is a node in the trie. If it already exists, we move to the next node or otherwise, we create a node and add it to the trie.
- Once we reach the last letter, we check if this word already existed in the trie. This is achieved by checking if the last letter is marked as the end of a word. If it is not the end of a word, the function sets the **is_end** variable to true and increases the number of words in the trie.
  The load command uses insert_word, with type **LOAD** so no extra output messages are printed.
  The insert command uses insert_word, with type **INSERT** so extra output messages are provided.
  Since we have two for loops that iterate n times where n is the number of character in word, the runtime is 2 ** O(n) = O(2n) = O(n).

is_empty returns the output for the **empty** command. It checks to see if the number of words in the Trie is 0, and returns the appropriate response. This is achieved in constant time since we are simply comparing the size of the tree with 0.

size returns the output for the **size** command. It returns the variable **num_words** which stores the existing number of words in the Trie. This is achieved in constant time since we are just returning a value.

deleteTrie and **deleteTrie_helper** is a recursive algorithm that deletes all nodes from the Trie. In **deleteTrie**, we set the total number of words to 0, set the current pointer to the root node and pass all of it's children nodes through the **deleteTrie_helper** function. **deleteTrie_helper** continues to recursively finds children nodes and deletes them. Since we need to traverse through every node and delete them, the runtime is O(n) where n represnts the number of nodes or alternatively, letters in the word. Since we assume n << N, this has a runtime of O(N).

deleteWord takes in one parameter of type string which stores the word to be deleted in the trie.

- Before deleting the word from the trie, it checks if every letter is in uppercase. If not, it throws an illegal exception.
- If this trie is empty, we exit early out of the function since this word is not in the trie.
- First, we iterate through each letter in the word and confirm that it exists in the trie. Once we reach the last letter, we confirm that this is marked as the end of the word in the trie. This process is to confirm that the word was previously inserted into the trie, and there's an early return if there are any discrepancy. Hence, this has a runtime of O(n) where n is the number of character in word.
- There are two cases I consider when deleting a word:
  - From the node with the last letter of the word, are there any children nodes? If yes, I unmark this as the end of an word. The runtime of this choice is O(1)
  - In case the current node has no child nodes, we traverse the word in reverse order and remove each character from the trie. If we encounter a character that marks the end of a word, we stop deleting nodes at that point.
  In the worst case, we iterate twice over the length of the word, resulting in a runtime of 2 ** O(n) = O(2n) = O(n)

printTrie and **print_trie_helper** is a recursive algorithm that prints all words from the Trie. The end of a word is marked by setting the **is_end** variable true for a node. If there are no words in the Trie, we print an empty newline. We initialize word to be an empty string and stores the letters of one word in the tree, building itself as we traverse the trie.

From the root, if we find a non-null child node, we append its letter to the current word and pass both the node and updated word to the **print_trie_helper** function. The print_trie_helper function continues this recursive approach of finding children nodes and appending the letter to the word. This algorithm uses DFS, where it travels to a leaf node before backtracking. When travelling back, we remove the letter from the word to accurately represent our traversal path in the trie. While traversing the trie, if we see the is_end variable set to true, we print the current version of the word. In this method, we traverse through all nodes in the trie once. Each node represents a letter in the word, which is represented by n. We can conclude that this has a runtime of O(n). Since we assume n << N, this has a runtime of O(N).

spellcheckTrie is a recursive algorithm that takes in one parameter of type string that represents the word we are searching for in the trie. We iterate through the word and in each iteration, we check if the letter is in the trie.

- If the letter is in this traversal path, we add it to a **prefix** variable which stores the letters that were in the trie.If not, we pass the prefix and node to the **print_trie_helper** function to recursively print out the words.
- If all letters of the word are in the trie, we check if **is_end** is set to true, as this indicates an exact match.
  In either possible option, the worst case runtime occurs if we call **print_trie_helper**, which as mentioned before is O(N) where N is the number of words in the trie.

getWordCountWithPrefix and **getWordCountWithPrefixhelper** are recursive algorithms to count the number of words with the given prefix. First, **getWordCountWithPrefix** is called with the prefix to search for in the trie. We iterate through the given prefix, confirming each letter is in the trie. If the letter is not found, we exit early since there is no word with this prefix. When we have confirmed the prefix exists in the trie, we pass it through **getWordCountWithPrefixhelper**. It has the same algorithm as **print_trie_helper** but instead of printing letters, it counts the number of leaf nodes it encounters. Hence, it has the same runtime as **print_trie_helper** which is O(N).