ECE250 Project 3 Design Document Matthew Li (m739li)

For this project, I designed 3 classes: Trie, Node, and illegal_exception. These classes are stored in their own separate header file in order to keep the code separated and organized.

Trie

Member Variables

root (Node *): Stores the root of the Trie.
size (int): Represents the total number of words currently in the Trie.

Member Functions

int findCombinations(Node *currentNode): The only private member function since this only gets called by count(). This function returns the number of words starting at the given node. This takes O(N) time where N is the number of words in the Trie because each word takes a maximum of O(M) operations where M is the number of in that word. Therefore, all words would take O(N*M) = O(N) time since there is a limit on M but no limit on N.
Trie(): Assigns the root of the Trie to a new instance of the Node class and assigns the size of the Trie to 0. The root node is never deleted until the Trie object is deleted.
~Trie(): Calls clear() with the root node, then deletes the root node. This deletes all nodes in the Trie including the root node, and the Trie has no other memory allocated to it so nothing else needs to be deleted.
Node *getRoot(): Returns the root node.
int getSize(): Returns the size of the Trie.
void setSize(int size): Sets the size of the Trie to the size given. This is only called after the clear command in order to set the size to 0.
bool insertWord(std::string word): Inserts the word into the Trie. Illegal argument was checked in the main function, and this function only gets called if the word is valid, but the word may still exist in the Trie. The illegal argument check can be done in O(n) time where n is the number of characters in the word because I am iterating through each character of the word. The actual insertion takes O(n) time as well because I attempt to find the word in the Trie; traversing the tree and checking if each character exists. In the worst case, iterating through the whole word takes O(n) time. If a new character is found, I allocate new Nodes to represent those characters which takes constant time. If the exact word was found with a terminal on the last character, a duplicate word was found, and this check takes constant time as well. Overall, O(n) + O(n) = O(2n) = O(n).
void count(std::string prefix): Counts the number of words in the Trie that start with the given prefix. Illegal argument was checked in the main function, and as explained earlier, this takes O(n) time where n is the number of characters in the prefix. Checking if the prefix exists in the Trie takes O(n) time as well since I traverse the Trie and check if each character exists. Worst case, the word exists and I iterate through the whole word. I then call findCombinations() with the address of the last node in the word, which takes O(N) time as explained earlier. Overall, since there is a limit on n but no limit on N, O(n) + O(n) + O(N) = O(N).
void erase(std::string word): Erases the word from the Trie. Illegal argument check takes O(n) time as explained earlier. Finding whether the word exists takes O(n) time since in the worst case, I have to iterate through all letters of the word. Checking if the last character is a terminal node can be done in constant time. I then check whether I can delete any nodes by backtracking that path I took to the leaf node. Starting at the leaf node, I check if that node has any children and whether it is a terminal node. If both are not satisfied, I can delete the node which takes O(1) time. In the worst case, I delete all nodes of the entire word, which takes O(n) time. Overall, O(n) + O(n) + O(n) = O(3n) = O(n).
void print(Node *currentNode, std::string currentString): Prints all words in the Trie. The call from main sends the root node and an empty string to this function. First it checks whether the current Node is a terminal node, in which case it outputs the current string. This can be done in constant time. Then it recursively calls itself with each of it's children and the current string appended with the character that the child represents. The entire recurrence takes O(N) time where N is the number of words in the Trie because each word takes a maximum of O(M) operations where M is the number of characters in that word. Therefore, all words would take O(N*M) = O(N) time since there is a limit on M but no limit on N.

void spellCheck(std::string word): Spellchecks the word or offers suggestions. I iterate through the given word until the child doesn't exist; in the worst case, I go through the whole word which takes O(n) time. On each iteration, I check if the entire word is spelt correctly, if the next character exists, or if the last letter has been reached. All of these can be checked in O(1) time. If the first is satisfied, I simply output "correct" and return to main. If the second is satisfied, I call print() with the current node and the string that the current node's path represents (excluding the current node). If the third is satisfied, I call print with the same parameters except the string includes the current node. In the worst case, I call print() after iterating through the entire word which overall takes O(n) + O(N) = O(N) time since there is a limit on n but no limit on N.
void clear(Node *currentNode): Deletes all words from the Trie. Iterates through the root and whenever a child is found, it recursively calls clear() with the child node, then deletes the child and the pointer to that child. The entire recurrence takes O(N) time where N is the number of words in the Trie because each word takes a maximum of O(M) operations where M is the number of characters in that word. Therefore, all words would take O(N*M) = O(N) time since there is a limit on M but no limit on N.

Node

Member Variables

Node letterArray:** Stores the 26 possible child nodes.
bool terminal: Stores whether the node is a terminal node or not.

Member Functions

Node(): Assigns letterArray to be an array of Node pointers with a size of 26, initialized to nullptr meaning there is no child Node stored there. Also assigns "terminal" an initial value of false.
~Node(): Each individual node gets deleted by the destructor of Trie, so only the letterArray array needs to be deleted.
bool getTerminal(): Returns the terminal value of the node.
void setTerminal(bool value): Sets the terminal value of the node to the value parameter.
Node *getLetterArray(int index): Returns the value of letterArray at the given index.
void setLetterArray(int index, Node *newNode): Sets the value of letterArray at the given index to newNode.

Illegal_exception

One member function called **void print():** Outputs "illegal argument" and returns to main(). Due to the simplicity of this class, the default constructor and destructor is used.

Other Runtime Analysis

Empty: Checks whether the size of the Trie is 0 and outputs the corresponding output. This is done in constant time since the Trie has a size member variable that gets updated accordingly.
Size: Outputs the size of the Trie. This is done in constant time since the Trie has a size member variable that gets updated accordingly.

UML Diagram