Project 3 Design Document

Bill Yao (b29yao)

1. Overview of classes

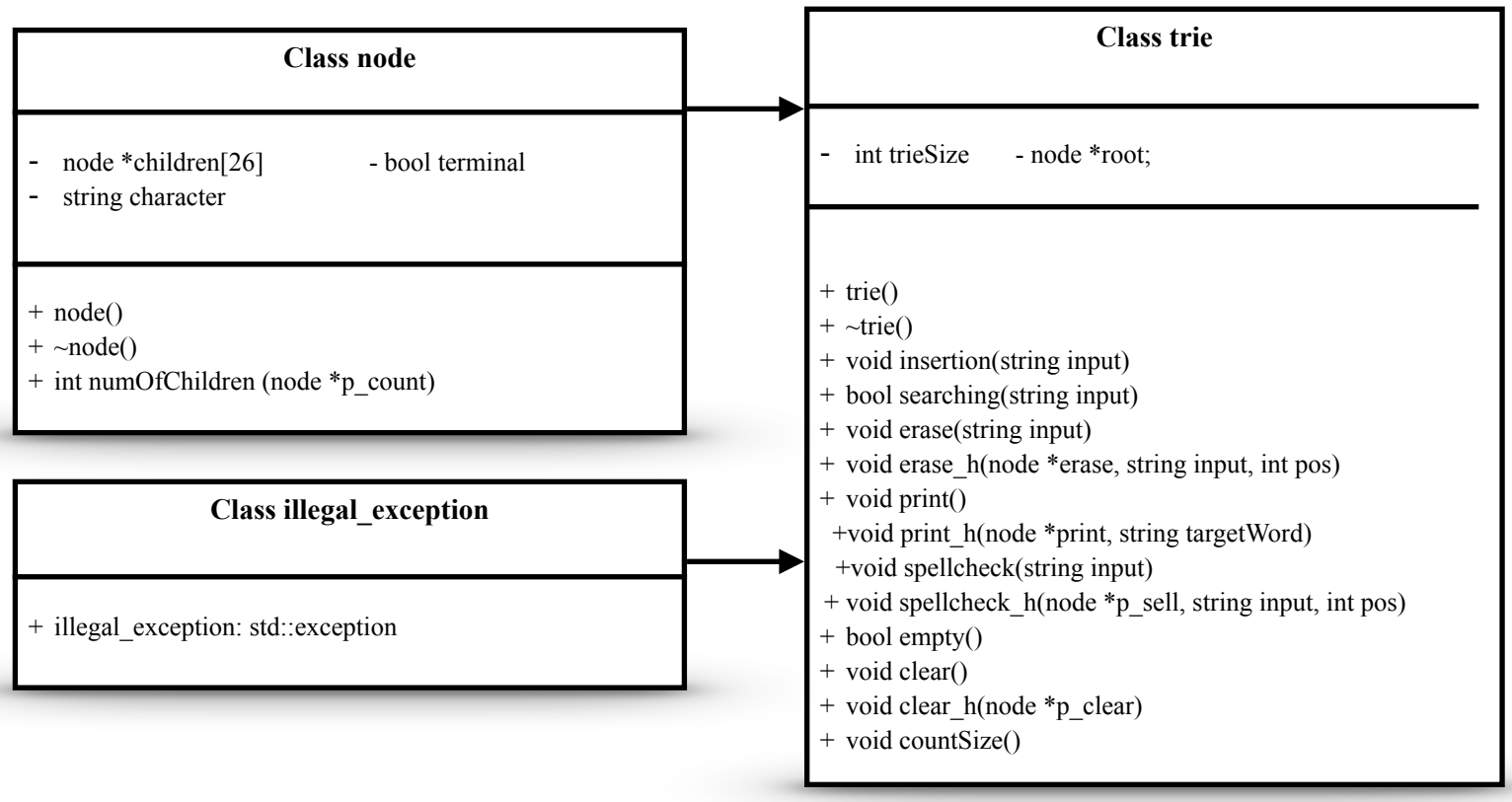
In this project, I have implemented three classes, node, illegal_exception and trie, in order to fulfill the functionalities.

For node class, every single node we have allocated it with 26 potential children node, so that we can accommodate all the possible arrangements of characters in the word. Also, each node would store a character as well as a boolean value “terminal” that signals whether a word stops at this node. The boolean value helps us to delete words in the erase function when we need to erase nodes that is being used in other existing words trie. Regarding the constructor and destructor, we would like to initialize all member variables with a default value, and every children of a node should be set to Null in the first place. When we call the destructor of a node, every child of that node should be deleted. At the very end, we have a “numOfChildren” function that determines the number of children given a node. This helps us determines various conditions in other member functions in trie.

For trie class, we have a member variable trieSize that return the number of words currently in the trie. A constructor in the trie is used to initialize all the variables and a destructor that calls the clear function to delete all the nodes in the trie. A more detailed explanation of member functions is in the design decision.

For illegal_exception class, whenever the input is not valid, it prints out an error message.

2. UML class diagram



3. Design Decisions

For node class and trie class, they use basic pointer to store information at different node and dynamically allocate/destruct nodes as we implement different functions. There are no operators that we used to override, nor did we use any pass by reference on any parameters since they are constantly taking in new data as the pointer switches.

void insertion(string input) / void searching(string input): we get the index of each letter in the word, and traverse through the trie to put correct letter in the correct node, or extract the letter from the node.

void erase(string input): initialize function and call the recursive function erase_h.

void erase_h(node *erase, string input, int pos): use recursion to check if the node is a terminal. If it is, then we check if it has any children. If the terminal has children, then we set the node boolean value to be “not terminal”. Otherwise, we go to the last terminal letter and delete what is falling behind.

void print(): initialize function and call the recursive function print_h.

void print_h(node *print, string targetWord): use recursion to check the node. If the node is terminal, we push back the string and then print. If the node have children, we push back the string but do not print, and then we go through other children of the node. If we eventually reach the terminal, then we do pop_back to find another child.

void spellcheck(string input): initialize function and call the recursive function spellcheck_h

void spellcheck_h(node *p_spell, string input, int pos): we first check if the given word’s first letter is in the trie. If not, then we print a new line. Otherwise, we use recursion to check the node. If the node’s next index children is null and position is equal to the input word length, then we call the print_h with input node p_sepll starting at the last position index.

void clear(): initialize function and call the recursive function clear_h

void clear_h(node *p_clear): find every single node’s children using while loop. If the children node is not null, then we do the recursion check. When eventually the children node is null, then we delete the children.

void empty() / void countSize(): we have a member variable trieSize that determines if a function is empty and return the size of trie.

4. Testing Strategy

Functions like empty, size, load and exit can be tested together with other functions such as (printing, insertion), by checking the output of these functions.

For insertion function, every time we insert a new word into the trie, we use the print function to check if it is already in the trie. Also we need to check if the printed output is in alphabetical order and determine if the insertion is correctly placed in the trie. Eventually we throw a

For searching function, it is pretty easy to test. Every time we input a new word, we search it and check it is found; every time we erase a new word, we search it and check if it is gone.

For erase function, we can add different words that have overlapping letters in it (such as and, andy, Andrew) and delete each of them in different order. We want to check if two erase cases are both considered (erase entire word, erase bool value).

For testing spellcheck, we would check the word that is already in the trie, and word that has no first letter in the trie. Also, we want to give a partially correct word and see if it can print the correct words.

5. Performance considerations

Here we only discuss the functions that have expected running time.

For insertion function, since we use while loop to extract every character in the word and assign them to a fixed node in the trie, therefore the running time should be **O(n)**. Since if we increase the size of the input word by k, the required running time is improved with kn.

For searching function, it is almost the same procedure as insertion function, by which we traverse through the trie in a while loop with given character index. Therefore the running time should be **O(n)**.

For erase function, we first check every character in the word and determine the termination node. Then we can recursively do the erase. The running time is **O(n)**.

For print function, we would have to print every node in the trie. To print a word, the expected running time is O(1), thus if there are n words in the trie, we need **O(n)** running time to do printing.

For spellcheck function, the best case running time is when the first letter is not in the trie, or it is spelled correctly, however, the worst case occurs when we have to go through every node and determine the character where we start printing. In this case, the running time is **O(n)**.

For clear function, we need to go through every node to empty the node. Therefore, running time is **O(n)**.

For empty function, we have a trieSize variable in trie class that return an integer of the size of the trie. If the variable is 0, then we print “empty 1”, otherwise, we print “empty 0”. The expected running time is **O(1)** since we can directly access the member variable.

For size function, we would directly print the trieSize member variable value. The expected running time is **O(1)** since we can directly access the member variable.