**Project 3 Design report**

Pan Liu 20830383

**Overview of the Classes**

In project 3, we implement a trie data structure which store word(lower cases letter) in a 26-ary tree. Also, we need to deal with the invalid input, if the input contains any characters other than lower-case English letter, the code will throw a illegal_argument exception. Thus, I have 3 classes to implement this data structure, they are node, trie and illegal exception.

In node class, we have private variables which are child, parent, letter, and isWordEnd. In each node, we have 26 child and 1 parent, each of them hold a value which store the letter. Besides, if the node is the last letter of the word we stored, isWordEnd will be true.

In trie class, we have private variable which are count and root. Variable count has the number of word. Variable root is empty which is the root node of the trie.

**bool is_find(std::string word)**: convert letter in the string to ASCII, in a loop do the search to check if the child of that index is NULL or not. If the last letter is end of word, then return true, else, return false.
**int is_empty(node *p)**: In the loop to check number of the node's child is not NULL.
**void empty()**: If size is 0, it is empty, if not, is not empty.
**int t_size()**: return the count of trie which is the size of trie.

**void addword(std::string word)**: first do the is_find function, if it is already inserted, return 'failure'. If it is not insert we do the loop, if the letter already exist, we search next letter, if the child node is NULL, we insert the letter in the node. At the end of letter, we set a terminal which boolean the isWordEnd to true.

**void print()**: Initial function of print then call the recursive function r_print.
**void r_print( node *p, std::string word)**: Using DFS, initializing a string. If the word is terminal, we push back the string then print. If the word has child, we push back in the string but not print. Then, we doing the loop to search every child of the node, if we find a child, then we do the recursion. If we reach the terminal and print, then we need to pop back to find another child.

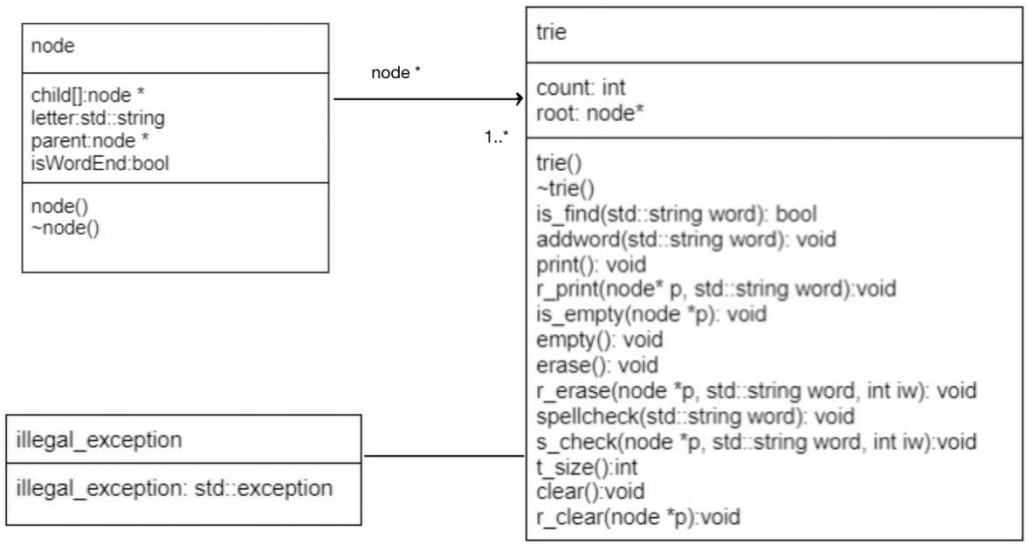
**void erase(std::string word)**: Initial function then call the recursive function r_erase.
**void r_erase(node *p, std::string word, int iw)**: Using DFS, using the recursion to checking the node is the terminal, if it is, check if the terminal has child. If the terminal has child, we just set the isWordEnd to false, else, we delete the terminal. If the node has no child, go to the parent and delete the node child which is origin node.

**void spellcheck(std::string word)**: Call the recursive function s_check.
**void s_check(node *p, std::string word, int iw)**: If the node is not reach the end of spellcheck word letter, doing the recursion with the index of child and iw +1. If the node reach to the end of word or the node has no child or the next letter is NULL in

the trie, we call the r_print function with input node p and sub-string of the word from 0 to the iw.

**void clear()**: call the recursive function r_clear.
**void r_clear(node *p)**: Using DFS, find every child of node in the trie. If the child is not NULL, do the recursion. When it reaches the end then delete the child.
In the illegal_exception class, we do nothing in the class, but, in the main function if we find any illegal argument, we will catch and throw that illegal argument.

**The UML**




**Design decisions**
There is a const int size equal to 26 which set 26 child const in the node.
**Node class**
Constructor: set the letter string to blank, isWordEnd equal false and from 0 to 26 initialize the every child to NULL.
Destructor: delete every child in the node.
**Trie class**
Constructor: count equals to 0 and new node root which is the root of the trie.
Destructor: Call the clear function to delete every node in the trie then delete root.
**Illegal_exception class**
Constructor: Do nothing
Destructor: Do nothing

**Test cases**
Test1: testing load, print, insert, size, empty and clear, by test load function then print see how it works. Then call the clear function to see if it works to delete all the node in the trie.
Test2: testing the insert and erase function, insert bunch of different words(andy andi

andrew and an) which contains same letter. Then erase several word and print to see if print is right.
Test3: testing the spellcheck function, first we load the text in to the trie. Then test spellcheck several times in different word to see if it is work.
Test4: testing illegal exception, we insert, erase and search word in illegal word(capital letter, number, symbol) to see the code will catch and throw the illegal argument.
Test5: Testing the memory leak. Using valgrind to test all function to see any memory leak and error.

**Performance consideration**

Load: depends on how many words we need to insert.
size: O(1) return the number of word in the trie(count)
empty: O(1) empty 0 or empty 1
i: check every letter in the word thus the running time is O(n)
s: check every letter in the word to check the letter thus the running time is O(n)
e: check every letter in the word to delete the node thus the running time is O(n)
p: using DFS to print every node in the trie. If we think print 1 word is O(1), thus the running time will be O(N) which N is the number of words in the trie.
spellcheck: best case: there is no any letter in the spellcheck for example spellcheck z in corpus thus it's O(1). worst case: go through all the words in the trie running time will be O(N) which N is the number of words in the trie
clear: best case, if the trie is empty, is take O(1) time. If the trie is not empty, the running time will be O(N) which N is the number of words in the trie.