ECE 250 - Project 3 – Trie data structure – Design Document

Easson Weisshaar, UW UsedID: etcweiss

November 20th

Overview of classes

Class:

node - A node data type that allows for a trie structure to be created

Private member variables:

letter: a char type variable that holds the represented letter of a node, **words_contained**: a integer type variable that stores the amount of words that a single node is included in, **pointers**: An array of node pointers of size 26, **leafNode**: A bool type variable that serves to act as the end of a word to help differentiate over-lapping words

Public member functions:

get_pointer: a function that takes a argument *int index* and returns a node pointer from the nodes **pointers** that correlates to the passed integer, **get_letter**: returns a char of the letter stored in the node, **get_wordCount**: returns an int of the words_contained in the node, **set_pointer**: takes in a node pointer argument and sets the passed node pointer into the correct pointers array index of the node called on, **increment/decrement_wordCount**: based on whether increment_wordCount or decrement_wordCount is called, it will add or minus 1 from the total word count respectively, **char_to_int**: takes in a char argument and returns the int correlating to it, i.e a = 0, b = 1…, **swapLeafBool**: changes the bool stored in the leafNode bool to its complimented form, **isLeaf**: returns the value of leafNode, **set_all_null**: sets all pointers in the pointers array to nullptr, **set_null**: given an index argument, sets the targeted *pointers* array index to nullptr, **isAllNull**: returns a bool based on whether or not the passed node's pointers are all nullptr, **recursive_print**: takes in a string argument that will print all the words contained in the node-called-on's subtree by calling itself until it reaches all leaf nodes, everytime it is called it adds a new letter onto the passed string. It determines whether to print a word if the leafNode bool returns true

Class:

trie - A class that is composed of the *node* class, it provides all the functionality and additional variables that allow a trie to operate

Private member variables:

wordCount: an int data type to track amount of words in trie, **root**: a node pointer that serves as the root node of the trie, stores a null char

Public member functions:

getRoot: returns the root node of the trie, **add_word/add_word_no_prints**: the add functions take in string argument and checks to see if the word exists or not and by calling the word_exists function. It then iterates through each letter and determine if there needs to be a new node create to accommodate the letter or if it needs to increment the words_contained of the node. The only difference between the two is that add_word_no_print doesn't have any print statements included for the purpose of inserting the entire corpus file, **word_exists**: function takes in a string argument and checks if each letter of the word is stored in the expected node *pointers* array, once it reaches the final letter it'll check if the node is a leafNode and if it is, the word is there, otherwise it is just contained by coincidence, **delete_word**: the function takes in a string argument and first checks if the word exists. If it does, it'll check if the word count is equal to 1 or if its greater than 1. In the situation it is greater it'll call decremenet_wordCount to "delete" the letter. If the word being deleted splits off into a unique subtree where it is the only letter stored, it'll call delete on the first instance of words_contained being equal to zero, which recursively deletes all the children of that node, **print**: calls recursive_print from the node class on the root node to print every word in the trie, **spellCheck**: given a word, the function will call word_exists to determine if the word is spelt correctly and will print "correct" in that case. Otherwise the function will check starting from the root node, if each letter exists in the correct order, the moment the expected value points to nullptr, the function will call recursive_print on the most recent present node in the word and pass the part of the word contained. This prints every word made-up of the first part of the checked word. In the case the first letter of a word points to nullptr, a newline is printed, **empty**: prints empty 1 if wordCount of trie is 0, prints empty 0 if wordCount is >0, **clear**: clears every node in the trie by calling delete on the root, additionally sets wordCount to zero and create a new root node, **size**: returns the wordCount of the tree, **charConversion**: takes in a char argument and returns the int correlating to it, i.e a = 0, b = 1…

Class:

illegal_exception - An exception class that inherits from the standard library exception class that allows custom exceptions

Private member variables:

N/A

Public member functions:

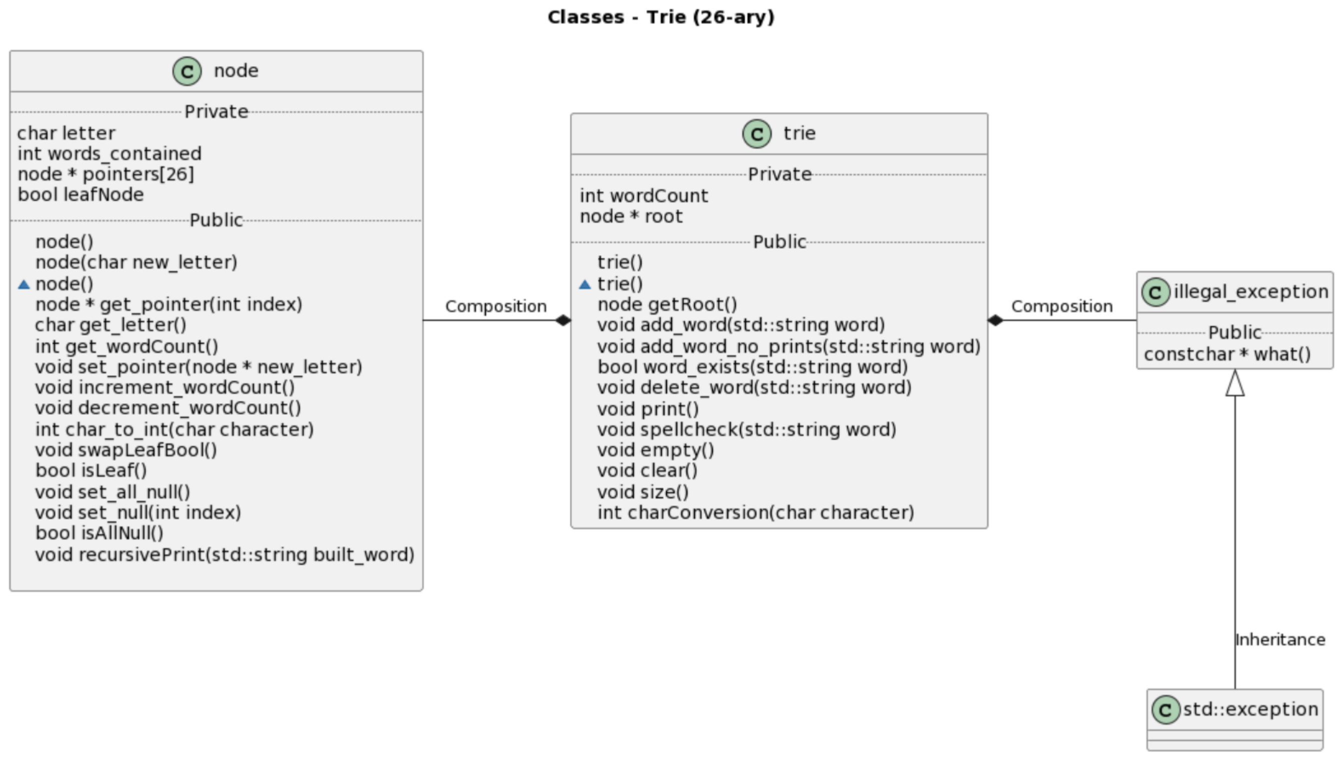
what: a function that when called, returns "illegal" argument, is used in insert, search, and delete to prevent any invalid characters from being inserted

ECE 250 - Project 3 – Trie data structure – Design Document

Easson Weisshaar, UW UsedID: etcweiss

November 20th

UML diagram



Constructor/Destructors

node:

- (default constructor) Given incorrect parameters it sets the letter stored to an empty char and the words contained to be zero, also initializes all of the pointers in the *pointers* array to nullptr
- The constructor takes in a char argument and sets the letter stored to that argument, it also sets the words contained to 1 and initializes all of the pointers in the *pointers* array to nullptr
- The destructor takes the called node and deletes all the pointers in the *pointers* array, this in turn recursively deletes all the children of those nodes

trie:

- Initializes the value of root to a new node that stores an empty char, also sets wordCount to 0
- The destructor calls clear on the root and then calls delete on the root

Test Cases

load:

- load the entire corpus and prints “success” whether corpus is loaded already or not
- able to reload corpus after cleaing

insert (i)

- add a word that creates all new nodes
- add a word that is coincidentally contained in another word
- add a word where a pre-existing word is coincidentally contained in the word getting added
- if a word already exists the function fails, or if a invalid character is contained in the word an exception is thrown
- adds single letters as words
- able to re-add previously deleted words

search (s)

- find words that exist that are either: contained in other words, overlapping with otherwords, and completely unique
- doesn't find words that don't exist or that are coincidentally contained in otherwords
- throws an exception when a word to be searched contains illegal characters

erase (e)

- deletes specified word if contained, otherwise functions fails if word doesn't exist, and throws exception for illegal arguments
- deletes only the word specified and not an over-lapping word that shares nodes
- deletes only a word that contains another word or a word containing the word to delete and leaving the other word
- can't re-delete an word after it's deleted

print (p)

- prints all words alphabetically with correct spacing and doesn't print any deleted words
- no output with empty trie

spellcheck

- prints all words alphabetically that share the same prefix with the word getting checked
- new-line is outputted if no matches are found

empty

- empty 1 if trie is empty, empty 0 if trie contains words

clear

- clears all words
- still prints "success" if clear is called on an empty trie

size

- returns proper size of trie after inserts, loads, clears, and deletes are called in any amount and order

exit

- exits properly when called
- exits with zero memory leaks

Performance considerations

To achieve a constant run time for the size and empty functions, I made use of a variable to track adds, deletes, and clears. This would allow the simple access of a number to determine the output. To achieve a linear runtime based on the number of characters in a word for search I would create a for loop that runs word (length - 1) times max, and in each iteration, the next letter is used and is checked is converted to an integer based on the letter (a=1…) and checks the associated pointer in the *pointers* array which prevent iterating through every pointer in the array. A linear runtime based on character amount is achieved for the add and erase functions by determining if the word exists (O(n)) and then executing instructions based on if there are overlaps in letters or not. If there is an overlap, either increment or decrement is called based on if add or erase is being used. If it is a singular node that needs to be made or deleted, then a new node is created or a node is deleted which deletes its children in turn. The end runtime is then O(n) + O(n) which is just O(n). Spellcheck, clear, and print all follow a similar algorithm to achieve a linear runtime based on the word count (O(N)). It is achieved through recursion. In print and clear, the root is called as the start node, and in print; recursive_print is called and in clear; the destructor for the node class is called. Both keep calling the functions used on each pointer in the *pointers* array that isn't nullptr until all nodes have been reached, this ultimately results in going through all the words and grows with the word count. Since spellcheck is essentially the print function but called on a child node in one of the root's subtrees, it will achieve the same runtime as print which is already O(N), additionally, spellcheck calls word_exists which adds the runtime of O(n), but still results in O(N) since it considers all words and not just the letters included in the spellchecked word.