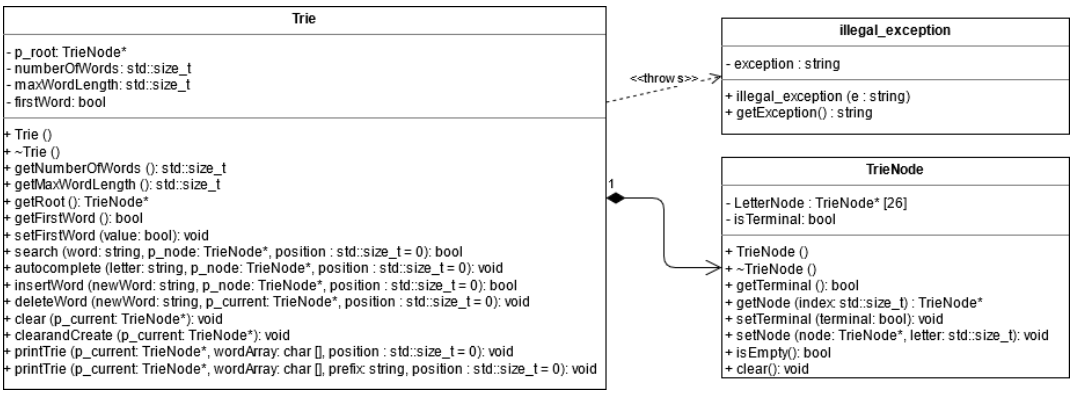
Design Document Project 2

Overview of Classes

Three classes were designed, a Trie class, a TrieNode class and an illegal_exception class. The Trie class was designed to create and manage (insert/erase/search/autocomplete/clear etc.) a trie (26-ary Tree) used to store words. The TrieNode class was designed to contain information about an individual node in the trie such whether the node represents the end of a word in the trie as well as which letter(s) the following node(s) is/are supposed to represent in the trie (ie. an array of pointers to the next TrieNode(s) which indicate which letter(s) can succeed the current letter if any). The illegal_exception class handles the illegal exceptions caused by invalid inputs to the insert, erase, and search commands.

The Trie class uses the TrieNode objects as nodes in the trie which store information about each character in the Trie and indicate which nodes are the end of a word. The Trie class can then manipulate the information (ie. terminal indicator and pointers to other nodes) in the trie using the public member functions from the TrieNode class. The illegal_exception class is used by the Trie class to throw exceptions when the input to its member functions is invalid.

UML Class Diagram



Design Decisions

Trie

The constructor used for this class dynamically allocates the root of the Trie and assigns a pointer (member variable) to indicate where the Trie begins (the entry point). Private member variables which store the number of words in the trie and the length of the longest word in the trie are initialized to 0, boolean member variable is initialized to true.

The destructor deallocates the memory for the trie which is allocated in the insert function (where words are inserted) by calling the *clear* function. The *clear* function traverses the trie and recursively deletes every node. The clear command uses a modified clear function which calls *clear* and then creates a new root. Using a single function as a part of both my destructor and clear trie function was ideal since both had to remove all non-root nodes from the trie, and the destructor additionally had to delete the root as well. No operators were overwritten. Passing by reference was not used since the parameters being passed where quite small (std::size_t, std::string, bool).

For the following member functions:

```
void setFirstWord(const bool value); bool search(const string word, TrieNode* p_node, std::size_t position=0);
void autocomplete(const string letter, TrieNode* p_node, std::size_t position=0);
bool insertWord(const string newWord, TrieNode* p_node, std::size_t position=0);
bool deleteWord(const string newWord, TrieNode* p_node, std::size_t position=0);
void printTrie(TrieNode* p_current, char wordArray [], const string prefix, std::size_t position=0);
```

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered within the function.

For the following member functions:

```
std::size_t getNumberOfWords() const; std::size_t getMaxWordLength() const; TrieNode* getRoot() const;
bool isEmpty() const; URLNode* getFront() const; URLNode* getBack() const; std::size_t getSize() const;
URLNode* findURL(const string find_URL) const;
```

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

For the following member functions:

```
void clear(TrieNode* p_current); clearandCreate(TrieNode* p_current); printTrie(TrieNode* p_current, char
wordArray [], std::size_t position=0);
```

The keyword *const* was **not** used at all since in each case the function needs to be able to **change** the value(s) of member variables and/or the value(s) of the parameter(s).

Note: The following functions cannot be const at the function level (even though it would make sense if they were) since they update a member variable related to removing trailing spaces when printing.

```
void printTrie(TrieNode* p_current, char wordArray [], std::size_t position);
void printTrie(TrieNode* p_current, char wordArray [], const string prefix, std::size_t position);
void autocomplete(const string letter, TrieNode* p_node, std::size_t position);
```

TrieNode

The constructor for the TrieNode class assigns the terminal indicator to *false* and ensures that all the values in the array of pointers for the node are assigned *nullptr*. The destructor was assigns all the elements of the array to nullptr to ensure there are not dangling pointer. No memory needs to be deallocated since none was allocated in the constructor. No operators were overwritten. Passing by reference was not used since the parameters being passed where quite small (std::size_t, std::string, bool).

For the following member functions:

```
void setTerminal(const bool terminal); void setNode(TrieNode* node, const std::size_t letter);
```

The keyword *const* was used at the parameter level since in each case the parameter being passed should not be altered within the function.

For the following member functions:

```
bool getTerminal() const; TrieNode* getNode(std::size_t index) const;  isEmpty() const;
```

The keyword *const* was used at the function level since in each case the function should not change the value of any member variable.

Note: the clear() function is the only function in which the keyword const was not used at either the parameter or function level since it needs to be able to change the values of the member variables as well as the parameter.

Illegal_exception

The constructor for the *illegal_exception* class is passed a string when the exception is thrown and assigns it to the *exception* member variable. No destructor was required since there was no dynamic memory allocation in the constructor. No operators were overwritten. Passing by reference was not required for such a simple class.

The keyword const is used in the `string getException()` function as no member variables should be updated in this function. The keyword const is not used at the parameter level as `string getException()` is the only member function of this class.

Test Cases

To test if my program met the specified requirements, I created specific inputs to try and cover the many possible cases which would reveal problems with my code (stack dump, memory leak, dangling pointer, etc.)

Some specific cases which I tested for:

- Inserting a word which is already contained in the trie
- Inserting a word which already has a subset of itself in the trie (inserting *hello* when *he* is in the trie)
- Inserting a word which is a subset of a word already in the trie (inserting *he* when *hello* is in the trie)
- Deleting a word which does not exist in the trie
- Deleting a word from the trie which contains another word as a subset (deleting *hello* when *he* is also in the trie)
- Deleting a word from the trie which is a subset of another word in the trie (deleting *he* when *hello* is in the trie)
- Autocompleting using a prefix which itself is a word contained in the trie
- Autocompleting words using a prefix which does not match any words in the trie
- Clearing/Printing/Searching/Deleting/Autocompleting when there are no words contained in the trie
- Searching the trie for words which are subsets of in the trie but which are not words contained in the trie
- Searching/Deleting/Inserting words with invalid characters contained in the string
- Searching for a word which does not exits in the trie

Performance Considerations

Expected run time of O(n)

Operations *insert, erase,* and *search* each have an expected runtime of O(n) (n is the number of characters in word) since for each operation the number of nodes traversed is equal to the number of letters in the word. For example, for *search*, each character in the word, the representative node must be traversed until the next node does not exist (word doesn't exist) or the end of the word is reached. It is similar in *erase* and *insert*, the number of nodes traversed is equal to the number of characters in the word. For both *erase* and *insert* the *search* operation is used to ensure that when inserting the word does not already exist or when deleting the word does exist. In each case the runtime is O(n+n) which is just O(n). Hence the asymptotic upper bound for *insert, erase* and *search* is O(n).

Expected run time of O(N)

Operations *print*, *autocomplete*, and *clear* all have an expected runtime of O(N) where N is the number of letters contained in the trie. The maximum number of nodes traversed in each case is equal to the number of characters in the trie. For example, in the *print* operation each node must be traversed once since each character in the tree must belong to at least one word. The letters which are repeated are stored in a char array to avoid traversing a node more than once. *Clear* must traverse every node since it must remove them all. Finally, *autocomplete's* worst case is where every word in the tree shares the same prefix. In this case every node will be traversed once however in most cases the number of nodes traversed will be less than the number of characters in the trie. Therefore, *print*, *autocomplete*, and *clear* all have an asymptotic upper bound of O(N).

Expected run time of O(1)

Operations *empty* and *size* have an expected running time of O(1) since they only require accessing of a member variable. Size can be computed in a constant time since it is stored as a member variable and updated (incremented/decremented) whenever a word is inserted or deleted from the trie. The *empty* operation checks whether the size is 0 or not. This design choice allows *size* and *empty* operations have an asymptotic tight bound of O(1) instead of O(N) which would be the case if entire trie was traversed each time to calculate the number of words.