1. Overview of Classes

a. Node class

- **Purpose and relation**: Serve as the node of a Trie and stores its parent, if it is an end of a word (EOW), whether it has been visited and its children. Used by the Trie class.
- **Public member function**:
  - **getEOW(), getVisited(), getParent()**: Returns the EOW, visited, and its parent respectively
  - **setEOW(bool inEOW), setParent(Node *inParent), setVisited(bool isVisited)**: sets the variables, isEOW, parent and visited respectively to the passed in the values
- **Constructor and Destructor**:
  - **Node()**: sets all data elements to 0 and any pointers to nullptr
  - **~Node()**: sets all data members to 0 and any pointers to nullptr. In addition, removes the pointer pointing to it from its parents (if it has a parent).
- **Private Member variables**: bool isEOW, bool visited, Node *parent
- **Public Member variables**: Node *children[26]

The Public member functions are all public as they are getters and setters and all the data members except children are private because there is no need for other functions to modify. Children is public as it is used a lot and its values are changed a lot so using getters and setters would result in a large amount of function calls.

b. Trie class

- **Purpose and relation**: Serve as a trie, stores the head of the trie as well as how many items are in the trie (N). It uses the Node class as its nodes.
- **Public member function**:
  - **insert(std::string inKey)**: Inserts the key into the trie if it doesn't exist and is valid. Inserts by iterating through the inKey and checking if each letter exists after another in the trie and adding a new node where necessary
  - **search(std::string inKey)**: Iterates through the inKey checking if each letter exists after another in the Trie. If not then the inKey is not in the trie and returns false
  - **clearVisited(Node *inNode)**: traverses the subtrie using DFS with head inNode and sets each node’s visited to 0 on a visit
  - **print(std:string out, std::string base, Node *inNode)**: Recursively iterates through the entire subtrie with head inNode using DFS and outputs each node if it is a word (isEOW = 1).
  - **printt()**: Calls print followed by clearVisited.
  - **erase(std::string inKey)**: Searches for the inKey and if it exists in the trie then erase traverse the trie and erases the word inside the trie
  - **empty()**: Checks if the trie is empty by checking if the head has any children
  - **spellcheck(std::string inWord)**: Searches for the word and if it doesn't exist then it iterates through inWord and traverses the trie if each character exists in the trie. When we reach a point where we have the largest possible substring between inWord and words then we print the subtrie

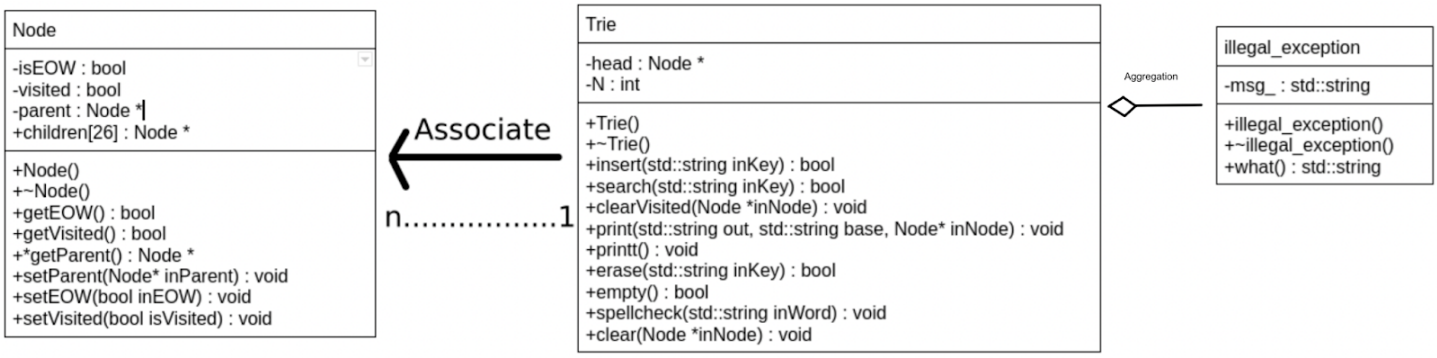
- **clear(Node *inNode)**: Traverses the trie using DFS from its head erasing each node.

- **Constructor and Destructor**:
  - **Trie()**: sets N to 0 and head to a new Node
  - **~Trie()**: clear the trie using clear(head) and then deletes the head
- **Private Member variables**: Node *head, int N

c. Illegal_exception class

- **Purpose and relation**: Serves as an exception class and is used whenever we have an illegal input. Has data member msg_ which is a string and is the output msg
- **Public member function**:
  - **what()**: returns msg_
- **Constructor and Destructor**:
  - **illegal_exception()**: sets msg_ to “illegal argument”
  - **~illegal_exception()**: does nothing as when msg_ leaves scope the string destructor is called

2. UML Class Diagram



3. Details on Class Decisions

- Const functions can be used for the getters as there is no reason to modify the object, and are meant to prevent accidental changes. Given that each getter was one line, I felt no need for it.

a. Node class

- **bool getEOW()** - returns isEOW
- **bool getVisited()** - returns visited
- **Node *getParent()** - returns parent
- **void setParent(Node *inParent)** - Requires a Node pointer parameter. Sets parent to inParent
- **void setEOW(bool inEOW)** - Requires a bool parameter. sets isEOW to inEOW
- **void setVisited(bool inVisited)** - Requires a bool parameter. sets visited to inVisited
- **Node()** - Sets all pointers to nullptr and all other data elements to 0
- **~Node()** - If the node has a pointer, set the parent’s child which points to itself to nullptr by iterating across the children of the parent until we find the node (using a for loop). Then set all other pointers to nullptr and all other data elements to 0

b. Trie class

- **bool insert(std::string inKey)** - Requires a string parameter. Searches if inKey exists in the trie. If it does, return 0 if not then iterate through the characters of inKey. For each iteration check if inKey's i^{th} character (i = # of iterations) is a child, if not then we create a new Node and move to the next depth (the newly created Node), else we travel to that Node.
- **bool search(std::string inKey)** - Requires a string parameter. Searches if inKey exists in the trie by iterating through inKey and checking if it exists by attempting to travel to the next node per each iteration. If it is unable to travel before inKey is fully iterated across then it returns 0. Once we finish iterating, we check if the Node is an EOW by checking its data member isEOW. If not, we return 0. After all of this, we know that the Node is an EOW and then we can return 1
- **void clearVisited(Node *inNode**) - Traverses the subtrie with head inNode recursively using DFS and sets each Node's data member visited to 0
- **void print(std:string out, std::string base, Node *inNode)** - Requires 2 string and 1 Node pointer parameters. Traverse the subtrie with head inNode recursively using DFS and marking visited nodes and adding the visited Nodes to the string out. If we reach a leaf Node then we pop_back out. If reach an EOW then we print out base+out
- **void printt()** - Calls print after clearVisited
- **bool erase(std::string inKey)** - Requires a string parameter. Calls search to determine if inKey is in the parameter.
- **bool empty()** - Checks if the Trie is empty by iterating through the head's children and determining if they are all nullptr. If they are all nullptr then return 1, else return 0.
- **void spellcheck(std::string inWord)** - Requires a string parameter. Iterate through the string inWord to traverse the trie by determining if we are able to travel to the Node. Once we cannot traverse anymore we call print and print the sub-trie. Then we call clearVisited
- **void clear(Node *inNode)** - Requires a Node pointer parameter. Uses DFS to traverse the trie continuously and delete every node.
- **Node *getHead()** - returns head
- **int getN()** - returns N

4. Test Cases

- General Case
  - Test inserting by inserting a word, deleting it and inserting it again
  - Test search by inserting many words that share a substring and search for one of those words
  - Test erase by inserting a word, erasing it and searching to see if we can find it
  - Test print by loading in the corpus and printing it to determine if some words are not in order
- Edge case
  - Test spellcheck by spellchecking a string with length larger than the depth of the tree. Then I tested the same situation but when the part of the string that is in the Trie is a full word (e.g. the trie has test, and testaa and then I spellcheck testing). This is to check my conditions when printing. Lastly to fully check searching I printed after spellcheck to make sure that I fully removed all the visited flags I set when printing for spell check

- Test deleting by adding two words where one word includes the other word (e.g. test and testing) and repeat deleting either the smaller or larger string. This is to test to make sure erase stopping conditions are correct. In addition I checked this case with valgrind to double check.
- To make sure clear works and the destructor is working I loaded the corpus and inserted works and exited without erasing anything. Then I checked using valgrind to see if I lost any memory. I repeated this test with a variety of trie structures such as balanced trie and linear trie.
- To test illegal arguments I tried inserting, searching and erasing words that had special characters or capital letters.

- Memory Tests
  - I tested the above casses all in valgrind and found no memory leaks

5. Performance Decisions

- **insert**: Searches the trie for the word which is O(n). Then iterates through the trie n times adding the Nodes that correlate to the word where needed which is n*O(1) as a Node constructor is O(1) and checking for children is O(1). Thus inserting is **O(n) + n*O(1) = O(n)**
- **search**: Iterate through each character of the string and check if it exists in the Trie, if not then return that it is not in trie. Since we iterate through the entire string (of length n) and check if it exists (which is O(1)), searching is **n*O(1) = O(n)**
- **erase**: First we search which is O(n), then if it exists we iterate through the Trie till we reach the end of the word which is O(n). Then we traverse back up the trie and check if we can delete the node (check if not part of another word). Checking in this case takes O(1) and since the word is n length, then traversing back up the trie takes n*O(1) = O(n). Thus erase is **O(n) + O(n) + O(n) = O(n)**
- **print**: Uses DFS to traverse the trie and outputs each word. Since DFS is O(N), print is **O(N)**
- **spellcheck**: First we search which is O(n). Then we print the sub-trie which is O(N). Thus, spellcheck is **O(n) + O(N) = O(N)**. Since N >= n in this case, we cannot have a word longer than height of the trie in the trie.
- **empty**: Iterates through each child of the head node (26 children), and checks if it has a child. Thus, empty is **26*O(1) = O(1)**
- **clear**: Use DFS to traverse the trie which is O(N) (traverses each node), if it has no children then we delete it. Deleting happens in O(1) and since each child is eventually a leaf, clear is **N*O(1) = O(N)**
- **size**: Prints out the integer data member N, hence is **O(1)**