Conner Fisk & Ethan Bowles
CS 3505 - A4: Refactoring and Testing

Justify Document

Highlight 1

Old:

```
//Constructors
///Destuctor that deletes the contents of the backing array
Trie::~Trie(){}

/// @brief Copy constructor that copies the contents of the provided Trie's backing array to this obj
/// @param other - trie to be copied
Trie::Trie(const Trie &other)
{
    endOfWord = other.endOfWord;
    for (int i = 0; i < 26; i++) {
        if(other.alphabetMap.contains(alphabet[i])){
            Trie* copy = new Trie(other.alphabetMap.at(alphabet[i]));
            alphabetMap.insert(std::make_pair(alphabet[i], *copy));
            delete copy;
        }
    }
}

// Operators
/// @brief Assignment overload to copy contents safely without messing up either trie
/// @param other - trie to be assigned to
Trie& Trie::operator=(Trie other)
{
    std::swap(endOfWord, other.endOfWord);
    for (int i = 0; i < 26; i++) {
        if(other.alphabetMap.contains(alphabet[i])){
            alphabetMap[alphabet[i]] = Trie();
            std::swap(alphabetMap.at(alphabet[i]), other.alphabetMap.at(alphabet[i]));
        }
    }
    return *this;
}
```

New:

All of pictured code deleted.

Explanation:

Initially when we implemented the map to our Trie class, we filled in the Rule of Three methods with the necessary changes. For the copy constructor, we created a new Trie, added it to the map, then deleted the Trie as it was now held in the map. For the copy constructor, we used the same std::swap function as we had in our original Trie implementation, but just swapped the values of the maps. Our destructor was left blank as the map already has its own destructor. This is where the idea of our refactoring came into place. Not only does map (and bool & string, the other instance variables we have) have its own working destructor, it also has its own copy constructor and assignment operator overload. Due to this, and not calling “new Trie” at all in our Trie class, we simply deleted all of our Rule of Three methods from the Trie class. The code now uses the map’s, string’s, and bool’s Rule of Three methods instead.

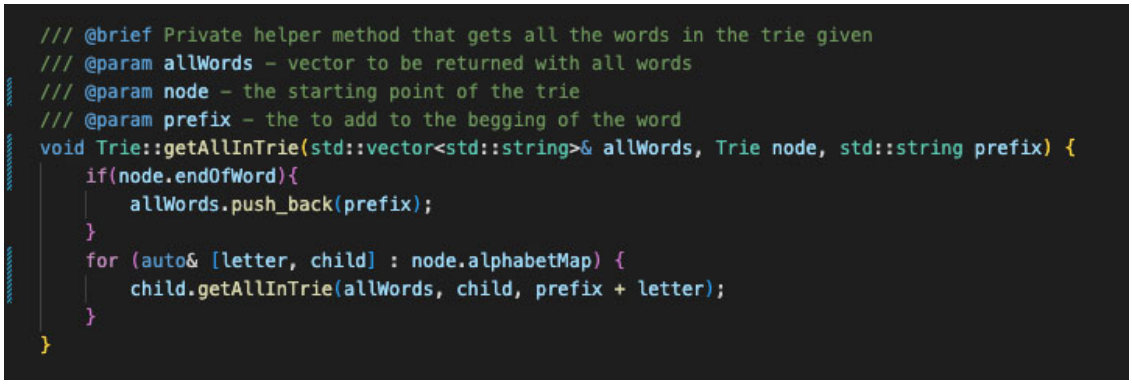
Highlight 2

In A3 there was redundant code in the isAWord and allWordsBeginningWithPrefix methods. They both had to navigate down to a certain point given by the user and then do something with that information. In order to reduce code repetition and create clarity we added a helper method called navigateToEndOfWord which takes in the word as a parameter and navigates down the tree until the correct trie node is reached. It will either return this node or it will return a nullptr if the node does not exist. This is used in the allWordsBeginningWithPrefix method to navigate to the end of the prefix and the isAWord method to navigate to the end of the word being checked. We also added another recursive helper method used in allWordsBeginningWithPrefix called getAllInTrie that gets all the words beginning at a certain tree node and adds a provided prefix to them. This returns a vector of all the words beginning with that prefix. This method allows us to not use recursion in the allWordsBeginningWithPrefix method making the code easier to read.

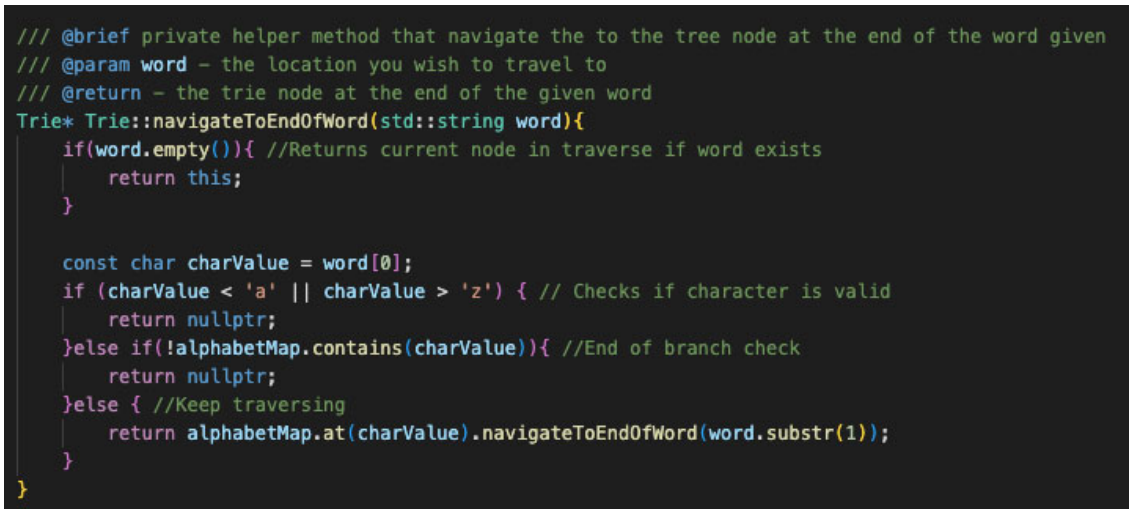
Old:

This code was implemented straight away so there are no picture of the before product

After:



```
/// @brief Private helper method that gets all the words in the trie given
/// @param allWords - vector to be returned with all words
/// @param node - the starting point of the trie
/// @param prefix - the to add to the begging of the word
void Trie::getAllInTrie(std::vector<std::string>& allWords, Trie node, std::string prefix) {
    if(node.endOfWord){
        allWords.push_back(prefix);
    }
    for (auto& [letter, child] : node.alphabetMap) {
        child.getAllInTrie(allWords, child, prefix + letter);
    }
}
```



```
/// @brief private helper method that navigate the to the tree node at the end of the word given
/// @param word - the location you wish to travel to
/// @return - the trie node at the end of the given word
Trie* Trie::navigateToEndOfWord(std::string word){
    if(word.empty()){ //Returns current node in traverse if word exists
        return this;
    }

    const char charValue = word[0];
    if (charValue < 'a' || charValue > 'z') { // Checks if character is valid
        return nullptr;
    }else if(!alphabetMap.contains(charValue)){ //End of branch check
        return nullptr;
    }else { //Keep traversing
        return alphabetMap.at(charValue).navigateToEndOfWord(word.substr(1));
    }
}
```

Highlight 3

In order to fully implement the map for A4 we needed a way to access character values quickly to use them as keys. Our solution to this was making a std::string of every lower case letter and accessing this through array calls. While this worked to access our map, certain parts of our code were cluttered with alphabet array calls. For example, the three locations were the navigateToEndOfWord helper, the getAllInTrie helper method, and the addWord method. In the navigateToEndOfWord method and addAWord method we changed the conditional statements logic to run on character values instead of integer values. Furthermore we also decided to store the letter and use this in our method calls for both methods. In the getAllInTrie method we changed the way we looped through the map by having a loop through the pairs instead of each letter and checking if it is contained. Overall all of these changes enabled us to completely get rid of the alphabet string in the header file, making our code cleaner and more optimized.

Old:

```
/// @brief Private helper method that gets all the words in the trie given
/// @param allWords - vector to be returned with all words
/// @param trie - the starting point of the trie
/// @param prefix - the to add to the begging of the word
void Trie::getAllInTrie(std::vector<std::string>& allWords, Trie* trie, std::string prefix) {
    if(trie->endOfWord){
        allWords.push_back(prefix);
    }
    for(int i = 0; i < 26; i++){

        if (trie->alphabetMap.contains(alphabet[i])){
            trie->alphabetMap.at(alphabet[i]).getAllInTrie(allWords, &(trie->alphabetMap.at(alphabet[i])), prefix + std::string (1, char(i+97)));
        }
    }
}

/// @brief private helper method that navigate the to the tree node at the end of the word given
/// @param word - the location you wish to travel to
/// @return - the trie node at the end of the given word
Trie* Trie::navigateToEndOfWord(std::string word){
    if(word.length() == 0){ //Returns if str is a word
        return this;
    }

    int charValue = word.at(0) - 97;
    if(charValue < 0 || charValue > 26){ //Valid Char Check
        return nullptr;
    }else if(!alphabetMap.contains(alphabet[charValue])){ //End of branch check
        return nullptr;
    }else { //Keep traversing
        return alphabetMap.at(alphabet[charValue]).navigateToEndOfWord(word.substr(1));
    }
}

 // Methods
 /// @brief method that adds a word to the tree recusivly in the correct location
 /// @param wordToAdd - the word to add
 void Trie::addAWord(std::string wordToAdd)
 {
     if(wordToAdd.length() > 0){ // Go until end of word adding
         int charValue = wordToAdd.at(0) - 97;
         Trie *copy = new Trie();
         if(!alphabetMap.contains(alphabet[charValue])){alphabetMap.insert(std::make_pair(alphabet[charValue], *copy));}
         delete copy;
         alphabetMap.at(alphabet[charValue]).addAWord(wordToAdd.substr(1));
     }else{ //End reached
         endOfWord = true;
     }
 }
```

After:

```
// Methods
/// @brief method that adds a word to the tree recusivly in the correct location
/// @param wordToAdd - the word to add
void Trie::addAWord(std::string wordToAdd)
{
    if (!wordToAdd.empty()) {
        const char letter = wordToAdd[0];
        if (!alphabetMap.contains(letter)) {
            alphabetMap.insert(std::make_pair(letter, Trie()));
        }
        alphabetMap.at(letter).addAWord(wordToAdd.substr(1));
    } else {
        endOfWord = true;
    }
}
```

```
/// @brief private helper method that navigate the to the tree node at the end of the word given
/// @param word - the location you wish to travel to
/// @return - the trie node at the end of the given word
Trie* Trie::navigateToEndOfWord(std::string word){
    if(word.empty()){ //Returns current node in traverse if word exists
        return this;
    }

    const char charValue = word[0];
    if (charValue < 'a' || charValue > 'z') { // Checks if character is valid
        return nullptr;
    }else if(!alphabetMap.contains(charValue)){ //End of branch check
        return nullptr;
    }else { //Keep traversing
        return alphabetMap.at(charValue).navigateToEndOfWord(word.substr(1));
    }
}
```

```
/// @brief Private helper method that gets all the words in the trie given
/// @param allWords - vector to be returned with all words
/// @param node - the starting point of the trie
/// @param prefix - the to add to the begging of the word
void Trie::getAllInTrie(std::vector<std::string>& allWords, Trie node, std::string prefix) {
    if(node.endOfWord){
        allWords.push_back(prefix);
    }
    for (auto& [letter, child] : node.alphabetMap) {
        child.getAllInTrie(allWords, child, prefix + letter);
    }
}
```