Code Refactoring

Name: Harrison Doppelt & Victor Valdez Landa

Date: 02/18/2025

There were 2 complex refactors that we implemented in our project

The first refactor was adding a ‘traverseTrie’ helper method. This method used to consist of having to traverse a Trie twice in two methods, in isWord() and in allWordsStartingWithPrefix(), so we implemented a helper method to traverse it for us and lessen the amount of code in both methods, in the snips below shows the green for added and red for deleted code.

```
+ const Trie* Trie::traverseTrie(const string& searchPrefix) const {
+     const Trie* currentNode = this;
+
+     for (char letter : searchPrefix) {
+
+         if (!isValidChar(letter)) {
+             return nullptr;
+         }
+
+         auto childIterator = currentNode->children.find(letter);
+
+         if (childIterator == currentNode->children.end()) {
+             return nullptr;
+         }
+
+         currentNode = &childIterator->second;
+     }
+
+     return currentNode;
```

```
94      -     // Start from the root node
95      -     const Trie* currentNode = this;
    73  +     const Trie* lastNode = traverseTrie(searchPrefix);
96  74
97      -     for (char letter : searchPrefix) {
98      -
99      -         if (!isValidChar(letter)) {
100     -             return {};
101     -         }
102     -
103     -         auto childIterator = currentNode->children.find(letter);
104     -
105     -         if (childIterator == currentNode->children.end()) {
106     -             return {};
107     -         }
108     -
109     -         currentNode = &childIterator->second;
    75  +     if (!lastNode) {
    76  +         return {};
110 77      }
```

```
65  65      bool Trie::isWord(const string& word) const {
66      -
67      -     if (word.empty()) {
68      -         return false;
69      -     }
70      -
71      -     const Trie* currentNode = this;
72      -
73      -     for (char letter : word) {
74      -
75      -         if (!isValidCharacter(letter)) {
76      -             return false;
77      -         }
78      -
79      -         auto childIterator = currentNode->children.find(letter);
80      -
81      -         if (childIterator == currentNode->children.end()) {
82      -             return false;
83      -         }
84      -
85      -         currentNode = &childIterator->second;
86      -     }
87      -
88      -     return currentNode->isEndOfWord;
    66  +     const Trie* lastNode = traverseTrie(word);
    67  +     return lastNode != nullptr && lastNode->isEndOfWord;
```

Since both methods need to traverse a Trie and both used the same logic, we thought we needed to simplify it and make it a whole method on its own, it seemed repetitive to keep the same code and both required to traverse the Trie in a recursive way, this is how we did it. So both methods are now shorter and more concise and we named it traverseTrie as a good name to signify what it does, which it traverses the Trie and returns the last node or nullptr depending if there is a valid word at the end of the Trie. Now in the prefix method now we can use this traverse helper method to get the last node of the Trie, which is a different way to implement the prefix method since at first, we had to confirm all the prerequisites for empty words and null pointers, this method does it all at once, which in the traversing and looking up letters we take advantage of the map object type which we can use the find() methods to our advantage to help with lookups in the method which will be used now in isWord and allWordsStartingWithPrefix. This is the first complex refactoring we have implemented to help shorten and use more efficient methods from other data types like maps in these methods

The second refactor we did was adding ‘isValidChar’ helper method. This helper method is used to remove any unnecessary and repetitive checks in methods that need to check whether or not a character is valid or not, in many of our methods we had to do this check, but we noticed how it could look like a spacer between what the methods had to do like in the prefix method we only care about getting a vector of words, etc. So, we made this method to make our code look shorter and concise. Here are before and after shots of the code.

```
+
+ bool Trie::isValidChar(char letter) {
+     return letter >= 'a' && letter <= 'z';
  }
```

```
@@ -72,7 +72,7 @@ bool Trie::isWord(const string& word) const {
72  72
73  73          for (char letter : word) {
74  74
75      -           if (letter < 'a' || letter > 'z') {
    75  +           if (!isValidChar(letter)) {
76  76                  return false;
77  77              }
78  78
@@ -96,14 +96,14 @@ vector<string> Trie::allWordsStartingWithPrefix(const string& searchPrefix) cons
96  96
97  97          for (char letter : searchPrefix) {
98  98
99      -           // Reject uppercase and non-alphabetic characters
100     -           if (letter < 'a' || letter > 'z') {
101     -               return {};      // Return an empty list
    99  +           if (!isValidChar(letter)) {
    100 +               return {};
102 101             }
```

As you can see the red was eliminated and the green was just the call to the helper method, as said before this shortened the code for the isWord and allWordsStartingWithPrefix and was named isValidChar to be as short and descriptive as possible stating what the method does, which checks the char for the validity of characters being put into the parameters being between ‘a – z’.

These are our two complex refactoring’s that made our implementations shorter and more concise took advantage of the data types of maps and used shorter and more efficient logic and helpers to make the code more readable for many code readers of any level.