Code Refactoring

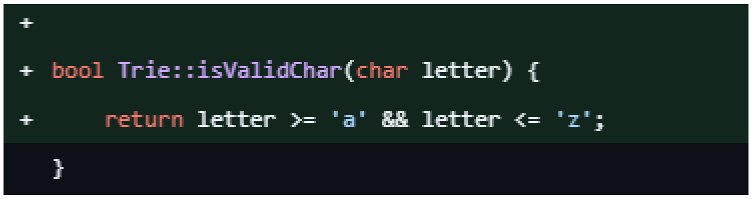
Name: Harrison Doppelt & Victor Valdez Landa

Date: 02/18/2025

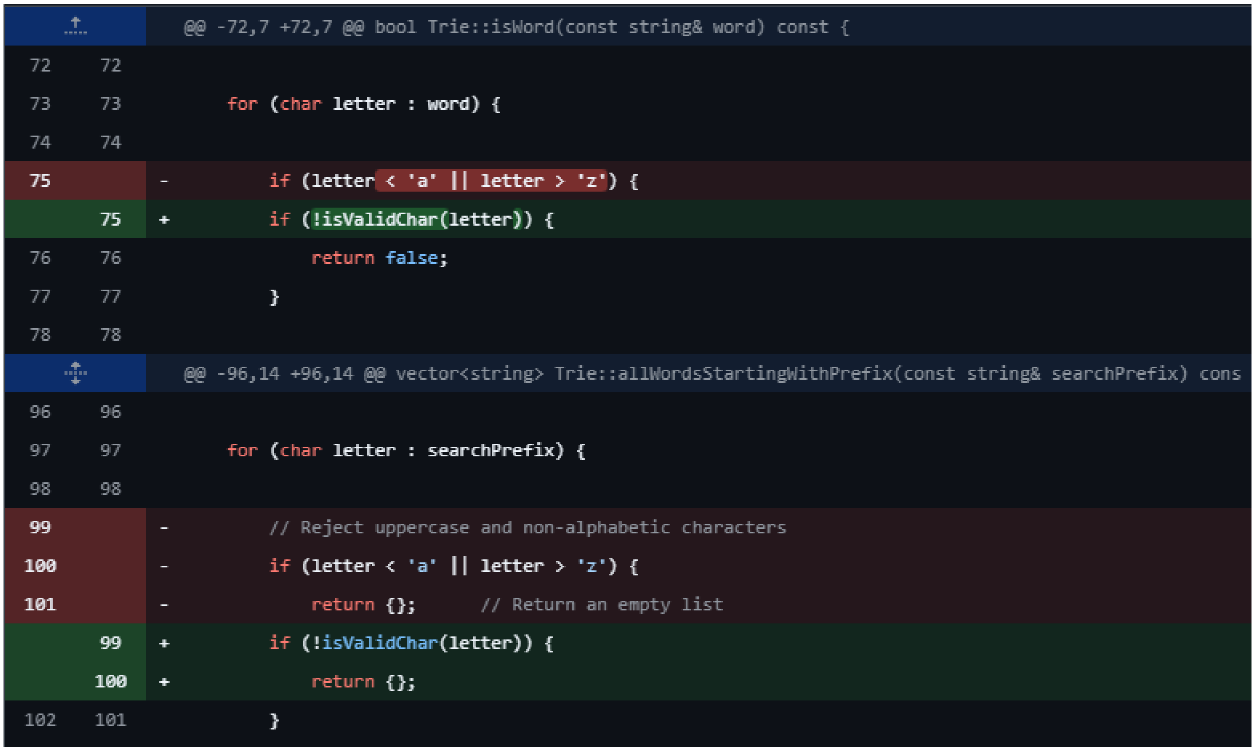
Our team has made a lot of refactors on this project, but there are two specific ones that we would like to discuss.

The first refactor we did was adding a ‘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. Two of our methods we had to do this check, isWord and allWordsStartingWithPrefix, 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. Below are the before and after screenshots of the code for the modified methods, as well as the helper method.

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’.



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

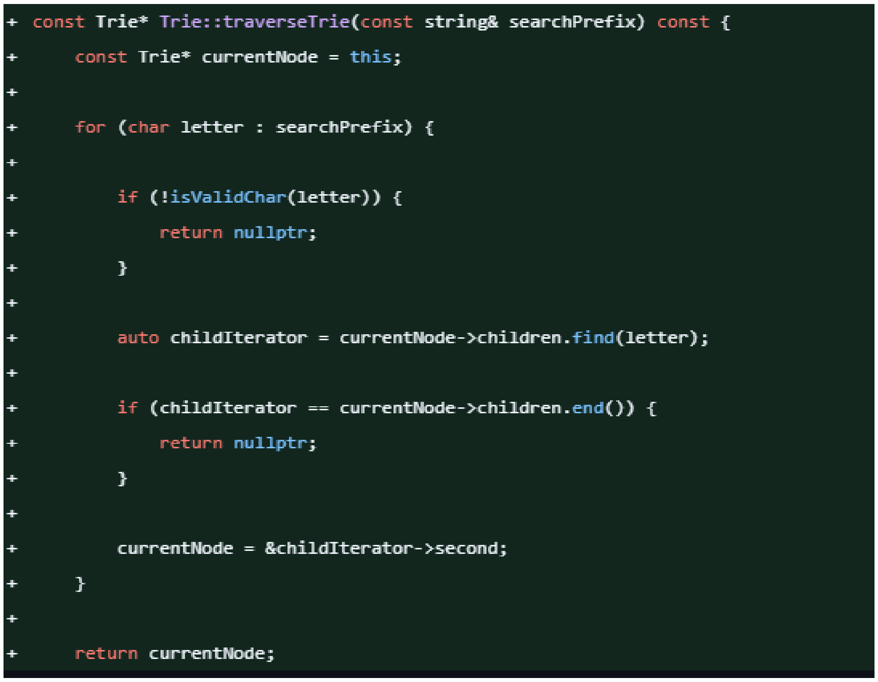


```
@@ -72,7 +72,7 @@ bool Trie::isWord(const string& word) const {

      for (char letter : word) {

-         if (letter < 'a' || letter > 'z') {
+         if (!isValidChar(letter)) {
              return false;
          }

@@ -96,14 +96,14 @@ vector<string> Trie::allWordsStartingWithPrefix(const string& searchPrefix) cons

      for (char letter : searchPrefix) {

-         // Reject uppercase and non-alphabetic characters
-         if (letter < 'a' || letter > 'z') {
-             return {};      // Return an empty list
+         if (!isValidChar(letter)) {
+             return {};
          }
```

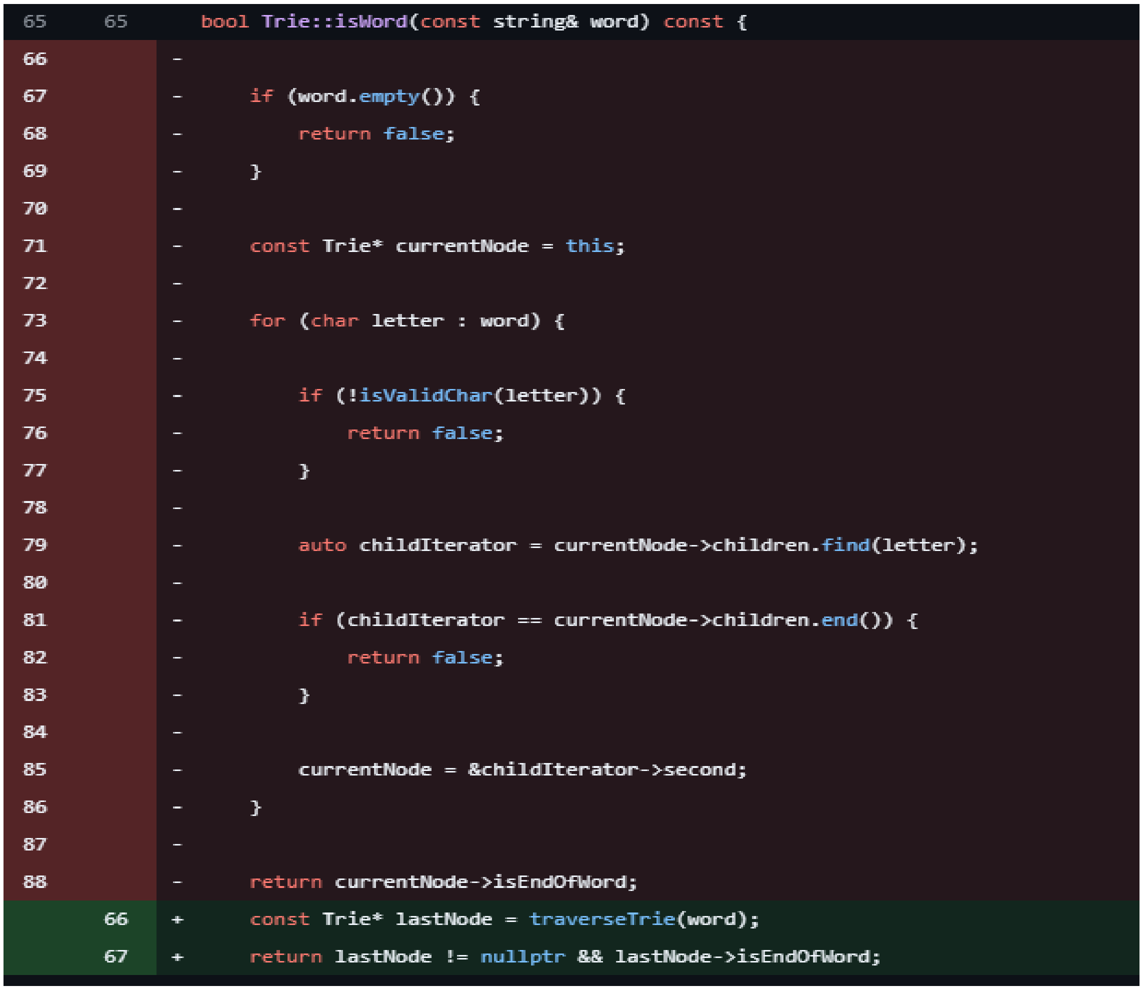
The second refactor was adding a ‘traverseTrie’ helper method. Before this implementation, our program was traversing a Trie twice in two methods, isWord() and allWordsStartingWithPrefix(). We realized this was an opportunity to simplify the logic and readability, so we implemented a helper method to traverse it for us in both methods. The screenshots display our before and after code for both of these methods, as well as the helper method. Since both methods need to traverse the Trie and used the same logic, we saw an opportunity to simplify it and make it a whole method on its own.

It seemed repetitive to have repeated code for both methods, and both needed to traverse the Trie in a recursive way, so we decided this would be a good modification to make. Now, both methods are 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 the prefix method can use this 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, and 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.

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 programmers of any level.

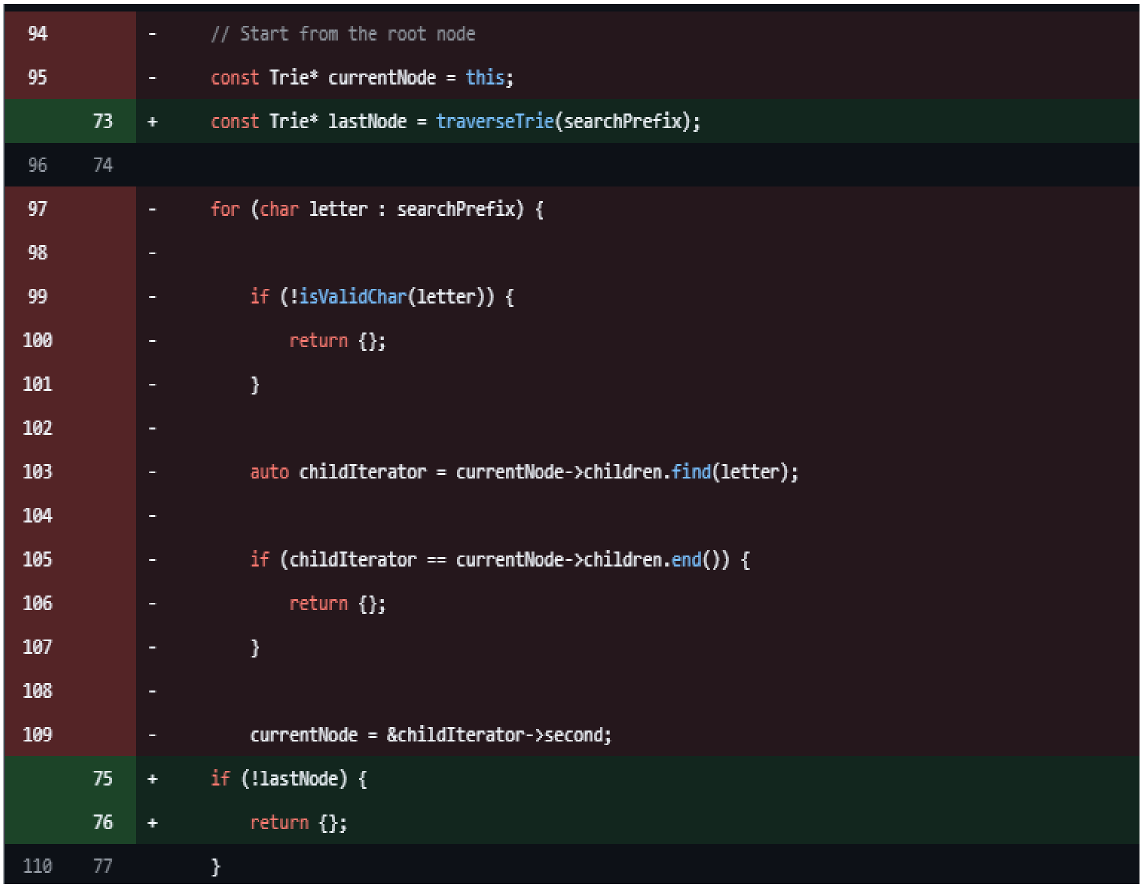


```
+ const Trie* Trie::traverseTrie(const string& searchPrefix) const {
+     const Trie* currentNode = this;
+
+     for (char letter : searchPrefix) {
+
+         if (!isValidChar(letter)) {
+             return nullptr;
+         }
+
+         auto childIterator = currentNode->children.find(letter);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



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

allWordsStartingWithPrefix()



```
-     // Start from the root node
-     const Trie* currentNode = this;
+     const Trie* lastNode = traverseTrie(searchPrefix);

-     for (char letter : searchPrefix) {
-
-         if (!isValidChar(letter)) {
-             return {};
-         }
-
-         auto childIterator = currentNode->children.find(letter);
-
-         if (childIterator == currentNode->children.end()) {
-             return {};
-         }
-
-         currentNode = &childIterator->second;
+     if (!lastNode) {
+         return {};
      }
```