Michael Sizemore and Kevin Macknight
Professor Johnson
CS 3505
2/22/2022

A4: Refactoring and Testing Justification Paper

During this project, we utilized many opportunities to refactor our code from A3 to clean up the logic and make the code of higher quality. Along with the required refactoring of the Trie class to utilize a map, we also implemented various changes to our code that made it easier to read, cleaner in logic, and reduced in repeated code. The first complex refactoring we wanted to highlight was our heavy truncating of the addAWord method. Our initial implementation of addAWord involved multiple if and else statements, which we used to cover the base case, and a case for if the branch does not exist. While we were unable to shorten the base case much shorter than it was before, we were able to significantly reduce the iterative part of the recursive method by utilizing the functionality of maps. Using maps in a way such as "branches[char]" creates a new default node if it's not present just by searching for it. By using this unique functionality in addAWord, we are able to elegantly move through the Trie while continuing to add nodes if they did not exist. While we found this to be a concise solution for this method, we avoided applying this logic elsewhere as addAWord is the only method that should be adding characters to the Trie.

Trie addAWord: Before

```
void Trie::addAWord(string givenString)
{
    if (givenString.length() == 0)// Base case for the recursive calls which sets the last node to true
    {
        formsAWord = true;
    }
    else if (letterPointers[indexOfLetter(givenString.at(0))])// If not nullptr goes to next recursive call
    {
        letterPointers[indexOfLetter(givenString.at(0))]->addAWord(givenString.substr(1));
    }
    else// If the node does not exist it is added into to Trie
    {
        Trie *newNode = new Trie();
        letterPointers[indexOfLetter(givenString.at(0))] = newNode;
        letterPointers[indexOfLetter(givenString.at(0))]->addAWord(givenString.substr(1));
    }
}
```

Trie addAWord: After

```
void Trie::addAWord(string givenString)
{
    if (givenString.length() == 0) // Base case for the recursive calls which sets the last node to true
    {
        formsAWord = true;
    }
    else
    {
        // Uses AlphabetTrieMap[char] to look for a branch as it will make a default Node if it does not exist
        AlphabetTrieMap[givenString.at(0)].addAWord(givenString.substr(1));
    }
}
```

For our allWordsStartingWithPrefix and isAWord methods, we decided that we could refactor them by adapting our traverseToString helper method, which would allow us to remove the unnecessary pointer calls that contributed to bloating our method. We did this by adapting our traverseToString method to return a Trie instead of a pointer to a Trie, which allowed us to cut out many of the cases we had to check for when using this method. This change ended up being really small and didn't impact the structure of this method, so instead of showing pictures of the minor changes in this method, we instead wanted to focus on how this small change allowed us to reduce code further. The pictures show how through this minor change to the helper method, we were able to reduce unneeded code in our other methods, such as in allWordsStartingWithPrefix, as we no longer needed to check for nullptrs.

Trie allWordStartingWithPrefix: Before updating traverseToString

```
vector<string> Trie::allWordsStartingWithPrefix(string givenPrefix)
{
    vector<string> validWords;
    Trie *prefixLocation = traverseToString(givenPrefix);
    if (prefixLocation)
    {
        prefixLocation->getAll(validWords, givenPrefix);
    }
    return validWords;
}
```

Trie allWordStartingWithPrefix: After updating traverseToString

```
vector<string> Trie::allWordsStartingWithPrefix(string givenPrefix)
{
    vector<string> validWordVector;
    traverseToString(givenPrefix).getAllWords(validWordVector, givenPrefix);
    return validWordVector;
}
```