Austin Topham and Gwenaelle Horrocks

Assignment 5 Refactoring Justifications

While we didn't have any major things to refactor outside of changing the array to a map, we did change a couple things.

1. **addWord**

We changed the addWord function to work correctly when an empty string is added. Before, the function would check to see if the string's length was zero, and if it was, set the validWord field to true, and return. This meant that empty strings would be marked as valid words and would return true when isWord("") is called. To fix this bug, we changed addWord function so that if a string's length was zero, we return but do not change the validWord field. We also added an if statement that checks to see if the string's length is one, and if it is, mark the next letter's validWord field to true. We also changed the variable name "alphabet" to "dictionary," since this is a more accurate name.

Old code:

```
void Trie::addWord(string word) {

   // when word is empty, the end of the word has been reached and the current trie can be
marked as a valid word
   if (word.length() == 0) {
       this->validWord = true;
       return;
   }

   // get the index of the first letter in word, set nextTrie equal to the trie in this trie's
index
   int index = word[0] - 'a';
   Trie* nextTrie = this->alphabet[index];

   if (!nextTrie) {
       nextTrie = new Trie();
       this->alphabet[index] = nextTrie;
   }

   // recursively call addWord with word minus the first letter
   nextTrie->addWord(word.substr(1));
}
```

New code:

```
void Trie::addWord(string word) {

   // when word is empty, the end of the word has been reached and the current trie can be
marked as a valid word
   if (word.length() == 0) {
```

```
        return;
    }

    // get the index of the first letter in word, set nextTrie equal to the trie in this trie's
index
    char letter = word[0];
    //Trie* nextTrie = this->dictionary[letter];

    if (!(dictionary.contains(letter))) {
        Trie newTrie;
        dictionary[letter] = newTrie;
    }

    if (word.length() == 1) {
        dictionary[letter].validWord = true;
        return;
    }

    // recursively call addWord with word minus the first letter
    dictionary[letter].addWord(word.substr(1));
}
```

2. allWordsStartingWithPrefix

We also changed the iterative loop in the allWordsStartingWithPrefix function to only use one Trie variable instead of two. Before, we had currentTrie and nextTrie to keep track of the current letter and next letter of the prefix we were looping over. We realized that the old way of doing it was using an unnecessary variable, since the same functionality can be accomplished with just one variable. This saves a little bit of storage and makes it easier to follow.

Old code:

```
vector<string> Trie::allWordsStartingWithPrefix(string prefix) {
    vector<string> prefixWords;
    Trie* currentTrie = this;    Trie* nextTrie;

    // iteratively navigate down the trie to the end of the prefix
    for (size_t i = 0; i < prefix.length(); i++) {
        int prefixIndex = prefix[i] - 'a';
        nextTrie = currentTrie->alphabet[prefixIndex];

        if (prefixIndex < 0 || prefixIndex >= 26) {
            return prefixWords;
        }

        if (!nextTrie) {
            return prefixWords;
        }
```

```
        currentTrie = nextTrie;
    }

    // at this point, currentTrie represents the node corresponding to the prefix
    // collect all words starting from currentTrie
    currentTrie->prefixRecursive(prefix, prefixWords);

    return prefixWords;
}
```

New code:

```
vector<string> Trie::allWordsStartingWithPrefix(string prefix) {
    vector<string> prefixWords;
    Trie currentTrie = *this;

    // iteratively navigate down the trie to the end of the prefix
    for (size_t i = 0; i < prefix.length(); i++) {
        char letter = prefix[i];

        if (letter < 'a' || letter > 'z') {
            return prefixWords;
        }

        if (!currentTrie.dictionary.contains(letter)) {
            return prefixWords;
        }

        currentTrie = currentTrie.dictionary[letter];
    }
    // at this point, currentTrie represents the node corresponding to the prefix
    // collect all words starting from currentTrie
    currentTrie.prefixRecursive(prefix, prefixWords);

    return prefixWords;
}
```