Nandhini Ramanathan and Ishan Sharma
February 17, 2025

Trie Class Refactoring Document

1. One of the more complex refactorings we did was use for each loops, instead of for loops, wherever possible. This made the code more concise and avoided unnecessary index-based looping which we did when our arrays were of fixed size instead of using a map. The methods we did this were in the copy constructor, isValidWord helper method, and especially the getWords helper method (recursively called in the allWordsStartingWithPrefix method).

   The copy constructor before manually iterated over an indexed array and explicitly allocated new Trie nodes, which allocated new memory that had to be deleted in the destructor. In our new code, the updated copy constructor now uses a for each loop to traverse the 'branches' map, directly inserting existing Trie pointers without using new, so the destructor is empty.

Copy Constructor Before:

```
for (int i = 0; i < 26; i++) {
        if (other.branches[i]) {
            branches[i] = new Trie(*(other.branches[i]));
        } else {
            branches[i] = nullptr;
        }
}
```

Copy Constructor After:

```
for (auto branch : other.branches) {
        branches.insert(branch);
}
```

   The getWords helper method before used a fixed-size array with a for loop to iterate over children nodes, even if they were empty unnecessarily. Since 'branches' is now a map that only stores existing relationships (no empty ones), the for each loop now iterates over key value pairs in the map. This change does not iterate over a fixed length, directly accesses only existing children, and the object does not get copied since we are passing pairs by reference, ensures they are not changed by using const, and is more readable by using the keyword auto, improving performance and readability.

GetWords Before:

```
for (int i = 0; i < 26; i++) {
    char characterAtIndex = char('a' + i);
    if (node->branches[i]) {
        getWords(node->branches[i], prefix + characterAtIndex, words);
    }
}
```

GetWords After:

```
for (const auto& branchEntry : node->branches) {
    getWords(&branchEntry.second, prefix + branchEntry.first, words);
}
```

2. Another refactoring we did was adding an isValidWord helper method. In the old program, there were numerous places where we were calling the same code to test whether a word contained only valid characters. Here is the isWord method as an example.

isWord before:

```
// Is Word Driver
bool Trie::isWord(const string& word) const {
    for (char character : word) {
        if (character < 'a' || character > 'z') {
            return false;
        }
    }

    return isWordRecursive(word);
}
```

This code was also being called in allWordsStartingWithPrefix so we decided to just make a helper method. This opens the door so that if in the future we feel that changing the program to allow for all characters a-z and A-Z or allowing all alphanumeric characters, we would only have to update the code in one area instead of having to do so in all the various method calls.

isWord after:

```
// Is Word Driver
bool Trie::isWord(const string& word) const {
    if (!isValidWord(word)) {
        return false;
    }

    return isWordRecursive(word);
}
```

This new version is also more self-documenting. Someone can easily see in the new version that if the word isn't valid, it returns false. The older implementation was less clear as it is hard to infer why it's returning false inside of this for loop.