**Refactoring:**

1. **Refactoring prefix helper method**

One of the more complex refactorings that I did was I altered my prefix helper method to include a rootNode as a parameter. Instead of checking if the current letter was the end of a word, I instead made use of the isWord function and called rootNode.isWord() with my prefix I was checking. The root node is simply the trie that originally called the allWordsStartingWithPrefix function. I chose to adjust this because I think it makes it more clear and easier to understand when I use isWord instead of randomly checking a variable labeled endOfWord. I also repeat less code this way, because I already have provided the functionality for checking if something is a word in the isWord() function, there is no need for me to repeat that code elsewhere. I also changed it from a for loop to a foreach loop to check each character in the branches map, instead of every index in the branches array.

Code before:

vector<string> Trie::prefixHelper(string prefix)

{

    vector<string> allWords;

    //If we have reached the end of the word, push it to the vector

    if ((\*this).wordFlag == true)

    {

        allWords.push\_back(prefix);

    }

    //Loop through all child tries and their arrays to find all words with prefix

    for (int i = 0; i < 26; i++)

    {

        if ((\*this).branches[i] != nullptr)

        {

            string newPrefix = prefix + (char(i + 'a'));

            vector<string> tempVector = ((\*(\*this).branches[i]).prefixHelper(newPrefix));

            allWords.insert(allWords.end(), tempVector.begin(), tempVector.end());

        }

    }

    return allWords;

}

Code after:

vector<string> Trie::prefixHelper(Trie rootNode, string prefix)

{

    vector<string> allWords;

    if (rootNode.isWord(prefix) == true )

    {

        allWords.push\_back(prefix);

    }

    for (auto currentChar : (\*this).branches )

    {

        string newPrefix = prefix + (currentChar.first);

        vector<string> tempVector = (currentChar.second).prefixHelper(rootNode, newPrefix);

        allWords.insert(allWords.end(), tempVector.begin(), tempVector.end());

    }

    return allWords;

}

1. **Removing unnecessary if/else statements**

The next refactoring I did was removing a lot of unnecessary if/else statements in my code, specifically in my addWord and isWord functions. I realized that I had a lot of if statements that were checking something then returning either true or false, and then having all the actual logic in the else statement. This seemed redundant, so I flipped the if statement and got rid of the else statements entirely. I also did a lot of triple checking for situations that would return the same true or false value (i.e. I had an if statement to return false for each situation: if the branch didn’t contain the letter, if the letter was contained but wasn’t the end of a word, and if it just wasn’t the base case entirely). I instead took those statements and got rid of the else statements associated, condensed the if statements down into only situations where I would return true, then returned false at the very end if I didn’t make it into those if statements. It made my code a lot more readable, and it condensed the code into only the necessary parts instead of wasting calculation time on checking if statements that weren’t strictly necessary.

Code before:

void Trie::addWord(string word)

{

    //Base case. If we have reached the end of the word set the word flag to true

    if (word.length() == 0)

    {

        (\*this).wordFlag = true;

    }

    else

    {

        int index = word[0] - 'a';

        //If the index contains a null pointer, create a new trie to be held in the array at that index

        if (((\*this).branches[index]) == nullptr)

        {

            Trie\* newTrie = new Trie();

            (\*this).branches[index] = newTrie;

            (\*newTrie).addWord(word.substr(1, word.length())); //Recursion, use the new trie to call addWord with the substring

        }

        else

        {

            (\*(\*this).branches[index]).addWord(word.substr(1, word.length())); //Recursion, use the new trie to call addWord with the substring

        }

    }

}

bool Trie::isWord(string word)

{

    for (int i = 0; i < (int) word.size(); i++)

    {

        //Return false if it contains invalid characters

        if (!(word[i]>=97 || word[i]<=122))

        {

            return false;

        }

    }

    //If the word is longer than a single character then recurse

    if (word.size() != 2)

    {

        int index = word[0] - 'a';

        if ((\*this).branches[index] == nullptr)

        {

            return false;

        }

        return (\*(\*this).branches[index]).isWord(word.substr(1, word.size()));

    }

    //Base case, if the word only has one character left then return the wordFlag

    else

    {

        int index = word[0] - 'a';

        if ((\*(\*this).branches[index]).wordFlag == true)

        {

            return true;

        }

        else

        {

            return false;

        }

    }

    return false;

}

Code after:

void Trie::addWord(string word)

{

    //Base case. If we have reached the end of the word set the word flag to true

    if (word.length() == 0)

    {

        (\*this).endOfWord = true;

    }

    else

    {

        char firstLetter = word[0];

        //If the map doesn't contain a trie at the key associated with the first character in the word, add a new trie

        if (!((\*this).branches.contains(firstLetter)))

        {

            (\*this).branches[firstLetter] = \*(new Trie());

        }

        string subword = word.substr(1, word.length());

        ((\*this).branches[firstLetter]).addWord(subword); //Recursion, use the new trie to call addWord with the substring

    }

}

bool Trie::isWord(string word)

{

    for (int i = 0; i < (int) word.size(); i++)

    {

        //Return false if it contains invalid characters

        if (!(word[i] >= 97 || word[i] <= 122))

        {

            return false;

        }

    }

    char firstLetter = word[0];

    //If the word is longer than a single character then recurse

    if (word.size() != 1)

    {

        if ((\*this).branches.contains(firstLetter))

        {

            return ((\*this).branches[firstLetter]).isWord(word.substr(1, word.size()));

        }

        return false;

    }

    //Base case, if the word only has one character left then return the endOfWord indicator

    else

    {

        //If the current trie contains that letter and if the letter indicates the end of a word, return true

        if ((\*this).branches.contains(firstLetter))

        {

            if (((\*this).branches[firstLetter]).endOfWord)

            {

                return true;

            }

        }

        return false;

    }

}