Trie Refactoring Justification Document

Author: Thu Ha.

Introduction: This document provides a detailed explanation of two complex refactorings performed on my Trie project.

1. **Improved memory management in destructor and copy constructor**
   Original Code:

```
Trie::~Trie() {
   // No need to explicitly delete `root`, it is managed automatically
}

Trie::Trie(const Trie& other) : root(other.root) {
   // Rely on the Node copy constructor to handle deep copy
}
```

   My original destructor did not explicitly manage the deletion of child nodes and the copy constructor relied on a shallow copy of the root node.

   My refactored code:

```
Trie::~Trie() {}

Trie::Trie(const Trie& other) {
   root = other.root;
}

Trie& Trie::operator=(Trie other) {
   std::swap(root, other.root);
   return *this;
}
```

   The updated code now manages the root node more explicitly, ensuring that all allocated resources are properly handled. In this code, I used std::swap in the assignment operator, it will ensure a strong exception safety guarantee as the swap operation is less likely to fail. This change helps to improve memory safety and stability.

2. **Refactor addWord function to improve readability**
   Original Code:

```
/// @brief Add a word to the trie
/// @param word
void Trie::addWord(const string& word) {
    Node* currNode = &root;
    for (char ch : word) {
        if (!currNode) {
            cerr << "Error: Null node encountered in addWord.\n";
            return;
        }
        currNode = currNode->addCharacter(ch);
    }
    if (currNode) {
        currNode->setEndOfWordFlag();
    }
}
```

In my original code, I put an unnecessary null check and did not effectively handle the logic for adding new nodes.

Refactored code:

```
/// @brief Add a word to the trie
void Trie::addWord(string word) {
    Node* currNode = &root;
    for (unsigned int i = 0; i < word.size(); i++)
    {
        if (!(currNode->hasCharacter(word[i])))
            currNode = currNode->addCharacter(word[i]);
        else
            currNode = currNode->getChildNode(word[i]);

        if (i == (word.size() - 1))
            currNode->setEndOfWordFlag();
    }
}
```

My refactored code improves readability by removing unnecessary null checks and using a more straightforward approach to traverse and add nodes. By using hasCharacter to check for the existence of a character and then either adding or retrieving child node, the function becomes easier to understand. Also, setting the endOfWord flag at the appropriate point in the loop enhances the clarity of the function’s intent.