Drew Lawson and Deveren Schultz
CS 3505 Software Practice 2
A4: Refactoring and Testing
September 28 - October 4

Refactoring Changes Justifying Document

Rule of 3 Methods

On the first day, we recognized that we did not need the rule of 3 methods, destructor, copy constructor and assignment operator. Because those are really only used for C style array objects. As well they help instigate interaction with items created by the ‘new’ keyword, and sent to the heap.
Now, we know that a Map needs to expand to hold new values on the heap, but still the rule of 3 methods were not necessary with how we could construct and now simply copy a Map
From there went about simply converting our code from using a C Array with 26 null pointers to a Map that we could freely insert into. Switching our methods to use the Map, and see if everything could stay more or less the same.

Major Problems with Old Logic

On October 2nd and October 3rd we ran into some infuriating problems with our testing phase. The previous implementation was a mess of simple loops, pointers, and pointer to value conversion errors. Right when we got it all working, our tests for addAWord would not seem to create layers to the Trie node structure. To be brief, it got so bad that during on Monday a TA helped us extensively, believed our code should work, and could not find the problem with Stack seemingly deleting values.

Switch to Only 1 Class

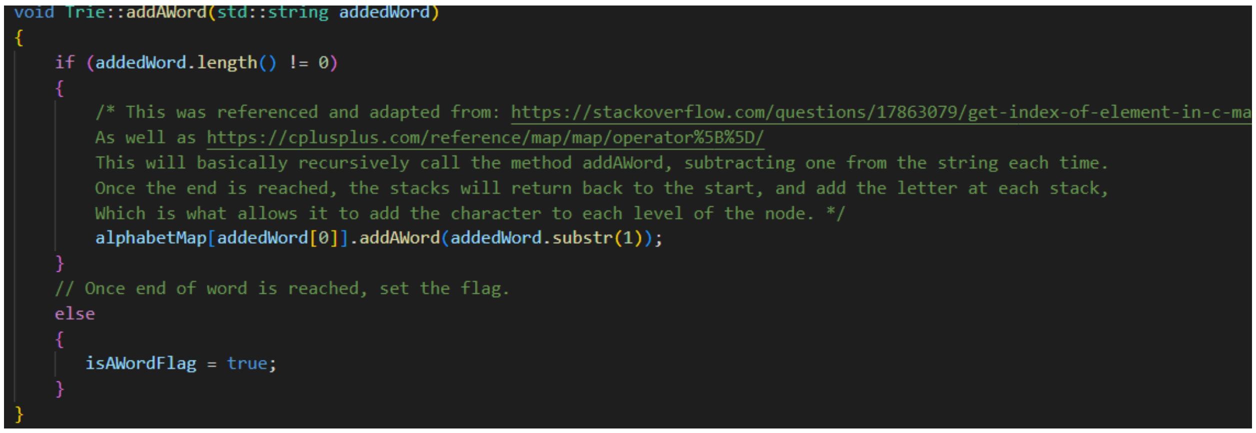
In the end we decided to start scraping our code and seriously simplifying things. That started with refactoring our code to only Class the Trie Class. Before we had it as the Trie Class and the node Class. We think what was happening before may have been linked to the constant invoking of the Node Class with Trie’s methods. And something or some stack was getting lost in between. Access permissions and whatnot. Or perhaps it was because access to the Map that was within the Node class was private and causing trouble with getting passed to Trie. We also updated our Makefile

Switch to Recursion

To add to that we also made major changes to the logic. We got fed up with the loops, and switched to recursion with helper methods. We looked up a bunch of C++ documentation and stackoverflow questions. It was a lot. We think we have a good understanding now, but could probably learn even more. We mostly take advantage of the operator[] for maps. By using this and calling a method recursively, we can use it to access the nodes in maps at lower levels of the Trie while still also getting the desired effects we need. The 2 helper methods we made are getAllWords() and wordSearch().

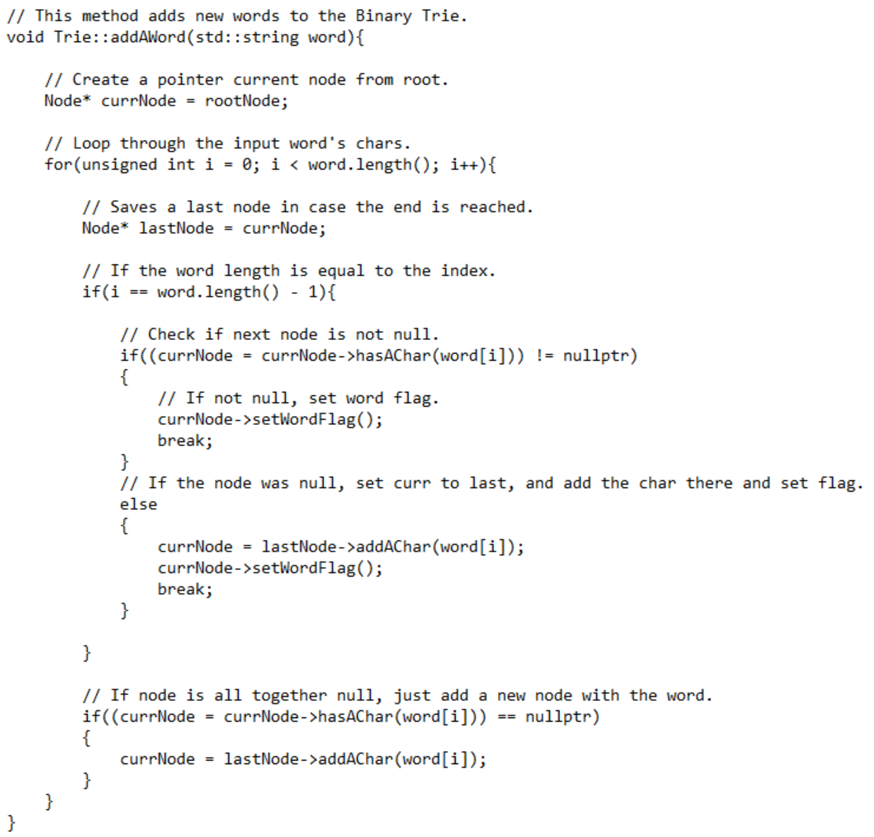
Please see below 2 examples of how using these helpers and refactoring made an insane night and day difference between the length and complexity of our New vs Old code:

Example 1 : Simplification of AddWord
New Version:



```
void Trie::addAWord(std::string addedWord)
{
    if (addedWord.length() != 0)
    {
        /* This was referenced and adapted from: https://stackoverflow.com/questions/17863079/get-index-of-element-in-c-ma
        As well as https://cplusplus.com/reference/map/map/operator%5B%5D/
        This will basically recursively call the method addAWord, subtracting one from the string each time.
        Once the end is reached, the stacks will return back to the start, and add the letter at each stack,
        Which is what allows it to add the character to each level of the node. */
        alphabetMap[addedWord[0]].addAWord(addedWord.substr(1));
    }
    // Once end of word is reached, set the flag.
    else
    {
       isAWordFlag = true;
    }
}
```

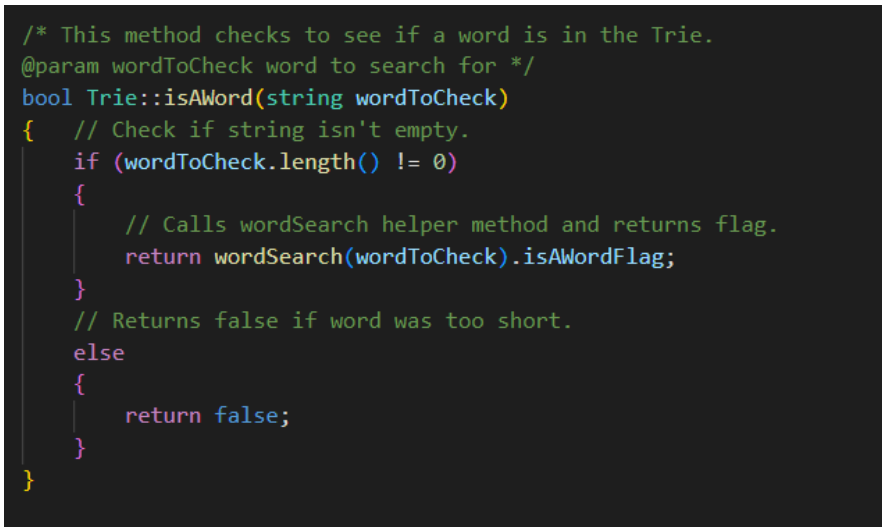
Old Version:



```
// This method adds new words to the Binary Trie.
void Trie::addAWord(std::string word){

    // Create a pointer current node from root.
    Node* currNode = rootNode;

    // Loop through the input word's chars.
    for(unsigned int i = 0; i < word.length(); i++){

        // Saves a last node in case the end is reached.
        Node* lastNode = currNode;

        // If the word length is equal to the index.
        if(i == word.length() - 1){

            // Check if next node is not null.
            if((currNode = currNode->hasAChar(word[i])) != nullptr)
            {
                // If not null, set word flag.
                currNode->setWordFlag();
                break;
            }
            // If the node was null, set curr to last, and add the char there and set flag.
            else
            {
                currNode = lastNode->addAChar(word[i]);
                currNode->setWordFlag();
                break;
            }

        }

        // If node is all together null, just add a new node with the word.
        if((currNode = currNode->hasAChar(word[i])) == nullptr)
        {
            currNode = lastNode->addAChar(word[i]);
        }
    }
}
```

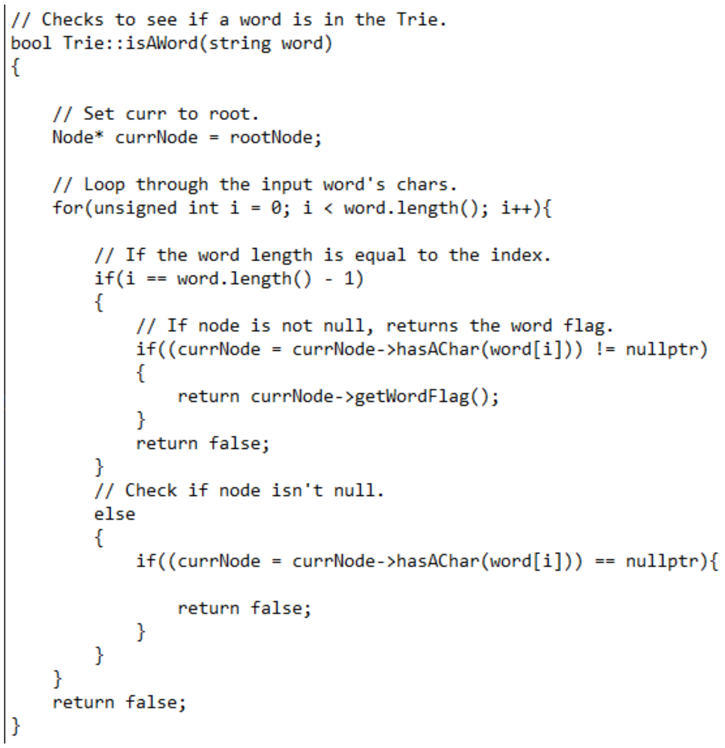
As can be observed, recursion takes the old logic of the code and makes it done in one line with the helper method. Instead of having to check every line to see if it's null, and then cycling through to check to see if the IsAWordFlag should be tripped in a fixed loop, we can now just recurse using a couple lines of code.

Example 2 : Simplification of IsAWord
New Version:



```
/* This method checks to see if a word is in the Trie.
@param wordToCheck word to search for */
bool Trie::isAWord(string wordToCheck)
{   // Check if string isn't empty.
    if (wordToCheck.length() != 0)
    {
        // Calls wordSearch helper method and returns flag.
        return wordSearch(wordToCheck).isAWordFlag;
    }
    // Returns false if word was too short.
    else
    {
        return false;
    }
}
```

Old Version:



```
// Checks to see if a word is in the Trie.
bool Trie::isAWord(string word)
{

    // Set curr to root.
    Node* currNode = rootNode;

    // Loop through the input word's chars.
    for(unsigned int i = 0; i < word.length(); i++){

        // If the word length is equal to the index.
        if(i == word.length() - 1)
        {
            // If node is not null, returns the word flag.
            if((currNode = currNode->hasAChar(word[i])) != nullptr)
            {
                return currNode->getWordFlag();
            }
            return false;
        }
        // Check if node isn't null.
        else
        {
            if((currNode = currNode->hasAChar(word[i])) == nullptr){

                return false;
            }
        }
    }
    return false;
}
```

Once again, no need to constantly cycle through the indexes. Now it can just call the helper method to recurse.

The end results are quite stunning in how short and simple they are. Even though it took so long, and was so frustrating at the moment, we don't regret the learning curve of this assignment. We feel we gained some solid understanding.