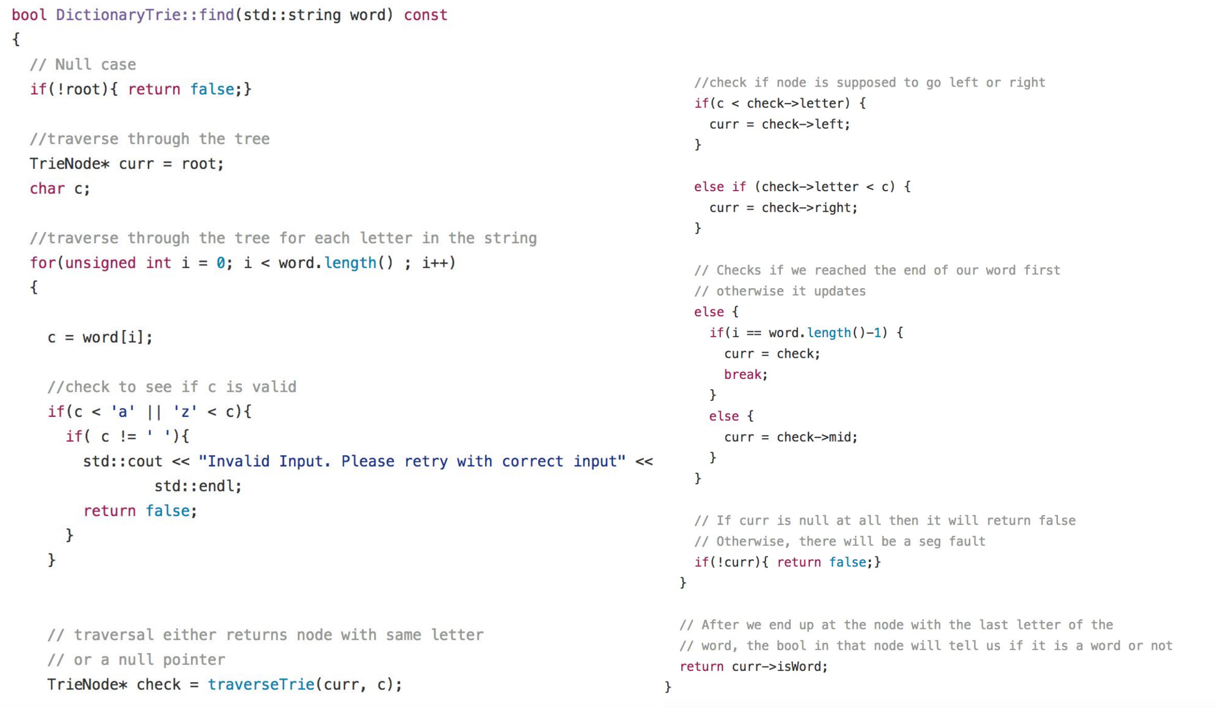
Peter Phan
Cs100wdh
A13042904

Chosen refactoring: I decided to refactor how I searching through the Trie.

In find() and predictNumCompletetions() I noticed that they both had a similar large for loop looking through the trie. So, I created a helper method that will implement this for loop for me. And, as I was able to implement this for loop in a new method called getNode(), my code looks much cleaner. Of course, some problems I had was that I wanted to do the same with insert(), but I depended on the index of the loop in this function to correctly add new nodes. My helper only returned the Node, so I was only able to clean up find() and predictNumCompletetions()

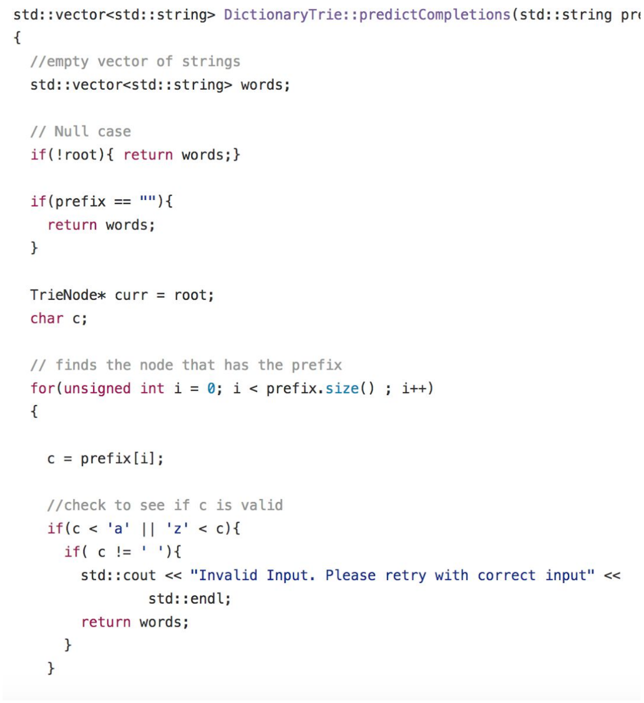
Summary:
What I did: created a helper function
Improvement: cleaner code, easier to read
Downfall: wasn't able to use this helper function for every method

Find:

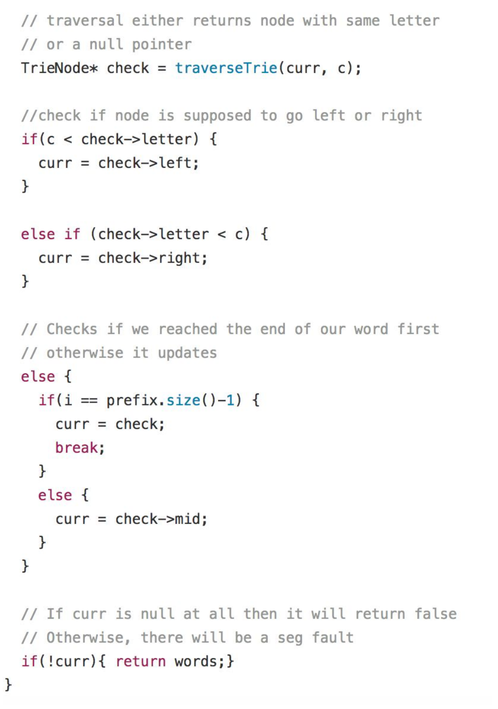


```
bool DictionaryTrie::find(std::string word) const
{
  // Null case
  if(!root){ return false;}

  //traverse through the tree
  TrieNode* curr = root;
  char c;

  //traverse through the tree for each letter in the string
  for(unsigned int i = 0; i < word.length() ; i++)
  {

    c = word[i];

    //check to see if c is valid
    if(c < 'a' || 'z' < c){
      if( c != ' '){
        std::cout << "Invalid Input. Please retry with correct input" <<
                std::endl;
        return false;
      }
    }


    // traversal either returns node with same letter
    // or a null pointer
    TrieNode* check = traverseTrie(curr, c);

    //check if node is supposed to go left or right
    if(c < check->letter) {
      curr = check->left;
    }

    else if (check->letter < c) {
      curr = check->right;
    }

    // Checks if we reached the end of our word first
    // otherwise it updates
    else {
      if(i == word.length()-1) {
        curr = check;
        break;
      }
      else {
        curr = check->mid;
      }
    }

    // If curr is null at all then it will return false
    // Otherwise, there will be a seg fault
    if(!curr){ return false;}
  }

  // After we end up at the node with the last letter of the
  // word, the bool in that node will tell us if it is a word or not
  return curr->isWord;
}
```

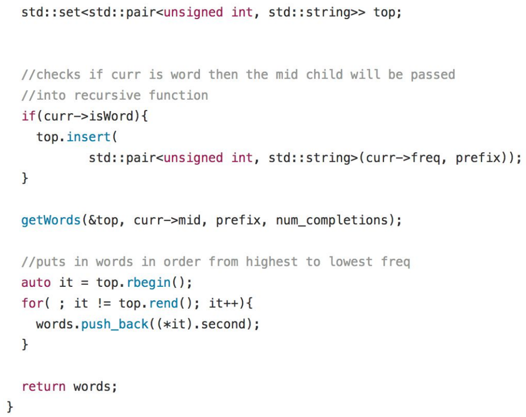
PredictNumCompletions



```
std::vector<std::string> DictionaryTrie::predictCompletions(std::string pr
{
  //empty vector of strings
  std::vector<std::string> words;

  // Null case
  if(!root){ return words;}

  if(prefix == ""){
    return words;
  }

  TrieNode* curr = root;
  char c;

  // finds the node that has the prefix
  for(unsigned int i = 0; i < prefix.size() ; i++)
  {

    c = prefix[i];

    //check to see if c is valid
    if(c < 'a' || 'z' < c){
      if( c != ' '){
        std::cout << "Invalid Input. Please retry with correct input" <<
                std::endl;
        return words;
      }
    }
```



```
  // traversal either returns node with same letter
  // or a null pointer
  TrieNode* check = traverseTrie(curr, c);

  //check if node is supposed to go left or right
  if(c < check->letter) {
    curr = check->left;
  }

  else if (check->letter < c) {
    curr = check->right;
  }

  // Checks if we reached the end of our word first
  // otherwise it updates
  else {
    if(i == prefix.size()-1) {
      curr = check;
      break;
    }
    else {
      curr = check->mid;
    }
  }

  // If curr is null at all then it will return false
  // Otherwise, there will be a seg fault
  if(!curr){ return words;}
}
```



```
  std::set<std::pair<unsigned int, std::string>> top;


  //checks if curr is word then the mid child will be passed
  //into recursive function
  if(curr->isWord){
    top.insert(
           std::pair<unsigned int, std::string>(curr->freq, prefix));
  }

  getWords(&top, curr->mid, prefix, num_completions);

  //puts in words in order from highest to lowest freq
  auto it = top.rbegin();
  for( ; it != top.rend(); it++){
    words.push_back((*it).second);
  }

  return words;
}
```

Comment: If we compare these former function implementation to the older ones, we can tell that the newer implementations are much more easier to read and look at