File - B:\school\CurrentSemester\DataStructures\Assignments\3.2\assignment-3-Veniotn\src\main.cpp

```cpp
#include <iostream>
#include "SpellChecker.h"

using namespace std;

int main(int argc, char **argv) {

    //check for correct number of arguments
    if (argc != 4) {
        cout << "Error: wrong number of parameters" << endl;
        return -1;
    }

    //holds our dictionary and misspelled files
    ifstream input_file;
    //holds our printed balanced tree
    ofstream output_file;
    //holds our search tree and spellchecking methods
    SpellChecker spell_checker = SpellChecker();

    //open the dictionary file
    input_file.open(argv[1]);
    //validate the file before continuing
    if (!spell_checker.validate_file(input_file, argv[1])) { return -1; }

    //create our search tree and fill it with the dictionary file
    input_file >> spell_checker.get_tree();
    input_file.close();

    //open the misspelled file
    input_file.open(argv[2]);
    //validate the file before continuing
    if (!spell_checker.validate_file(input_file, argv[2])) { return -1; }

    //spellcheck the file
    spell_checker.spell_check(input_file);
    input_file.close();

    //create output file in the desired location, no need to validate file type
    output_file.open(argv[3]);
    if (!output_file.fail()) {
        //print our balanced tree to the output file
        output_file << spell_checker.get_tree();
        output_file.close();
    }
    return 0;
}

```

```cpp
//
// Created by nick on 11/5/2023.
//

#ifndef ASSIGNMENT_3_SEARCHTREE_H
#define ASSIGNMENT_3_SEARCHTREE_H

#include <iostream>

//avl balancing methods and printing method found here:
//https://www.programiz.com/dsa/avl-tree

//individual search tree node
struct Node {
    std::string m_data{"/0"};
    Node *m_left{nullptr};
    Node *m_right{nullptr};
    int m_height{1};
};

class SearchTree {
private:

    //start of our search tree
    Node *m_root{nullptr};

    //recursive insert method that auto balances
    Node *insert(std::string word, Node *&node);

    //checks for inbalances and adjusts the tree accordingly
    Node *balance(Node *&node, std::string &word);

    //used to search through the list for a given word
    bool search(Node *&node, std::string word);

    // returns the height differences between each node
    int balance_factor(Node *node);

    //returns the nodes current height
    int height(Node *node);

    // updates the currents node height relative to its children
    void update_height(Node *node);

    //both of our rotation methods, called when tree isn't balanced
    Node *right_rotate(Node *node);

    Node *left_rotate(Node *node);

    //prints our tree to an output file
    std::ostream &print_tree(std::ostream &output, Node *root, std::string indent, bool last);

    //calls on the print tree method
    friend std::ostream &operator<<(std::ostream &output, SearchTree &tree);

    //used for importing our dictionary file
    friend void operator>>(std::istream &input, SearchTree &tree);


public:
    //both used in search tree object as an abstracted methods
    void insert(std::string word);

    bool search(std::string &word);
};


#endif //ASSIGNMENT_3_SEARCHTREE_H

```

```cpp
//
// Created by nick on 11/5/2023.
//
#include "SearchTree.h"

using namespace std;

void SearchTree::insert(string word) {
    m_root = insert(word, m_root);
}

Node *SearchTree::insert(string word, Node *&node) {

    //if the current node reference is null, create a new node
    if (node == nullptr) {
        //end of our recursive function
        node = new Node();
        node->m_data = word;
        return node;
    } else if (word > node->m_data) {
        //insert to the right of the current node
        node->m_right = insert(word, node->m_right);
    } else if (word < node->m_data) {
        //insert to the left of the current node
        node->m_left = insert(word, node->m_left);
    } else {
        //duplicate word, disregard
        cout << "Word" << node->m_data << "already exists" << endl;
        return node;
    }

    update_height(node);

    //balance the tree
    return balance(node, word);


}


Node *SearchTree::balance(Node *&node, string &word) {

    //grab the balance factor
    int bf = balance_factor(node);
    //check for imbalances in the tree
    if (bf > 1) {
        //left imbalance
        if (word < node->m_left->m_data) {
            return right_rotate(node);
        } else {
            //left right imbalance
            node->m_left = left_rotate(node->m_left);
            return right_rotate(node);
        }
    }
    if (bf < -1) {
        //right imbalance
        if (word < node->m_right->m_data) {
            node->m_right = right_rotate(node->m_right);
            return left_rotate(node);
        } else {
            return left_rotate(node);
        }
    }

    return node;
}

Node *SearchTree::left_rotate(Node *node) {
    //grab the first right child, to be moved to where the selected node is now
    Node *y = node->m_right;
    //grab its left child
    Node *t2 = y->m_left;
    //move the original node underneath the y node
    y->m_left = node;
    //reattach the t2 underneath our moved node which is under the root to the left
    node->m_right = t2;
    //update the heights of the effected nodes
    update_height(node);
    update_height(y);
    //return the subtrees new root
    return y;
}


Node *SearchTree::right_rotate(Node *node) {
    //grab the left child of the node we want to rotate
    Node *x = node->m_left;
    //grab its child which will be reattached to our rotated node
    Node *t2 = x->m_right;
    //move the selected node underneath x
    x->m_right = node;
    //reattach t2 to be under our rotated node
    node->m_left = t2;
    //update the heights of the effected nodes
    update_height(node);
    update_height(x);
    //return the subtrees new root
    return x;
}


bool SearchTree::search(std::string &word) {
    //use the inner search method to attempt to find the selected word
    return search(m_root, word);
}

bool SearchTree::search(Node *&node, string word) {

    if (node == nullptr) {
        //end of tree with no word found, end of recursion
        return false;
    }

    if (word == node->m_data) {
        //word found in tree, end of recursion
```

```cpp
        return true;
    } else if (word < node->m_data) {
        //if word is higher in the alphabet than the current node, check the nodes left child
        return search(node->m_left, word);
    } else {
        //if word is lower in the alphabet than the current node, check the nodes left child
        return search(node->m_right, word);
    }
}


int SearchTree::balance_factor(Node *node) {
    //if node exists, return the difference between its left and right children's heights
    if (node == nullptr) return 0;
    return height(node->m_left) - height(node->m_right);
}


void SearchTree::update_height(Node *node) {
    //set the new nodes m_height to 1 more than the highest child node
    if (node != nullptr) node->m_height = 1 + max(height(node->m_left), height(node->m_right));
}

int SearchTree::height(Node *node) {
    //if node is null return 0, else return its height
    return node == nullptr ? 0 : node->m_height;
}

ostream &SearchTree::print_tree(ostream &output, Node *root, string indent, bool last) {
    //recursive function to the end of the search tree, once it hits nullptr, recursion ends
    if (root != nullptr) {
        output << indent;
        if (last) {
            output << "R----";
            indent += "   ";
        } else {
            output << "L----";
            indent += "|  ";
        }
        output << root->m_data << root->m_height << endl;
        print_tree(output, root->m_left, indent, false);
        print_tree(output, root->m_right, indent, true);
    }

    return output;
}


void operator>>(std::istream &input, SearchTree &tree) {
    string word;
    //while the file is not empty, add each word to the dictionary tree
    while (!input.eof()) {
        getline(input, word);
        tree.insert(word);
    }
}


std::ostream &operator<<(std::ostream &output, SearchTree &tree) {
    //start the recursive print. returns the file output stream returned from the print method
    return tree.print_tree(output, tree.m_root, "", true);
}

```

```
//
// Created by nick on 11/16/2023.
//

#ifndef ASSIGNMENT_3_SPELLCHECKER_H
#define ASSIGNMENT_3_SPELLCHECKER_H

#include "SearchTree.h"
#include <regex>
#include <sstream>
#include <fstream>

class SpellChecker {

private:
    //file name pattern
    const std::regex FILE_NAME_REGEX = std::regex(R"(^(?:.*[\\/:])?[^\\/:*?"<>|]+\.txt$)");
    //bst that holds the dictionary
    SearchTree dictionary_tree;

    //removes all special characters from a word before it gets checked for spelling errors
    void clean_word(std::string &word);

    //confirms that the current string is a word and not blank space
    bool is_word(char &first_letter);

public:
    //getter for our dictionary tree
    SearchTree &get_tree();

    //confirms a valid txt file is being passed
    bool validate_file_name(std::string &file_name);

    //validates the file and calls on validate file name method
    bool validate_file(std::ifstream &file, std::string file_name);

    //main spell checking method ran on our misspelled input file
    void spell_check(std::istream &input_file);

};

#endif //ASSIGNMENT_3_SPELLCHECKER_H

```

```cpp
//
// Created by nick on 11/16/2023.
//

#include "SpellChecker.h"

using namespace std;


SearchTree &SpellChecker::get_tree() {
    return dictionary_tree;
}

void SpellChecker::spell_check(istream &input_file) {
    //search through the file and check every word
    string text_line, word;
    int misspelled_count = 0;

    //print header message to console
    cout << "\nMisspelled words: " << endl;

    while (!input_file.eof()) {
        //grab the current line
        getline(input_file, text_line);

        //use string stream to grab one word at a time from the current line
        istringstream iss(text_line);

        //grab one word from the line at a time
        while (iss >> word) {
            //get rid of special characters around the word
            clean_word(word);

            //if the word cannot be found, and it's not a number, print it to the console
            //and increment the misspelled count
            if (!dictionary_tree.search(word) && is_word(word[0])) {
                cout << word << endl;
                misspelled_count++;
            }
        }
    }
    //tell the user how many words are incorrect
    cout << "\nTotal number of misspelled words: " << misspelled_count << endl;
}


//removes special characters from a word
void SpellChecker::clean_word(string &word) {
    string cleaned;
    //iterate through each character in the word
    for (char character: word) {
        //if the char is a valid letter add it to the temp string
        if (isalpha(character)) {
            //lower case every letter
            character = tolower(character);
            cleaned += character;
        }
    }
    //if the string isn't null "return" our cleaned word
    if (cleaned != "\0") {
        word = cleaned;
    }
}

bool SpellChecker::is_word(char &first_letter) {
    //if the first letter of any given string isn't a letter it's either a digit or special character
    return isalpha(first_letter);
}

bool SpellChecker::validate_file_name(string &file_name) {
    //validate file name against the regex pattern
    return regex_match(file_name, FILE_NAME_REGEX);
}


bool SpellChecker::validate_file(std::ifstream &file, std::string file_name) {
    //if the name is invalid or the file fails to open, prompt the user and return false
    if (!validate_file_name(file_name) || file.fail()) {
        cout << "Error opening: " << file_name << " ! .txt files only." << endl;
        return false;
    }
    return true;
}

```