```
#ifndef ASSIGNMENT_3_BST_H
#define ASSIGNMENT_3_BST_H

#include "Node.h"

class BST {
    using NodePtr = Node *;

private:
    NodePtr m_root{nullptr};

    NodePtr insert(std::string &, NodePtr &);

    void print_tree(std::ostream &, NodePtr &, int);

public:
    int height(NodePtr &);

    int max(int, int);

    int balance_factor(NodePtr &);

    NodePtr rotate_right(NodePtr &);

    NodePtr rotate_left(NodePtr &);

    void insert(std::string &);

    static int compare(std::string &, NodePtr &);

    bool find(std::string &);

    friend std::ostream &operator<<(std::ostream &, BST &);
};

#endif //ASSIGNMENT_3_BST_H
```

```
#ifndef ASSIGNMENT_3_NODE_H
#define ASSIGNMENT_3_NODE_H

#include <iomanip>

struct Node {
    using NodePtr = Node *;

    std::string m_data{"NULL"};
    NodePtr m_left{nullptr};
    NodePtr m_right{nullptr};
};

#endif //ASSIGNMENT_3_NODE_H

```

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

using namespace std;

int BST::height(NodePtr &node) {
    // Return height of subtree
    if (node == nullptr)
        return -1;
    return max(height(node->m_left), height(node->m_right)) + 1;
}

int BST::max(int a, int b) {
    // Return greater value of two ints
    return (a > b) ? a : b;
}

int BST::balance_factor(NodePtr &node) {
    // Return balance factor of left and right subtrees of given node
    if (node == nullptr)
        return 0;
    return height(node->m_left) - height(node->m_right);
}

Node::NodePtr BST::rotate_right(NodePtr &y) {
    // Rotate left
    NodePtr x = y->m_left;
    NodePtr T2 = x->m_right;
    // Rotate values
    x->m_right = y;
    y->m_left = T2;

    return x;
}

```

```cpp
Node::NodePtr BST::rotate_left(NodePtr &x) {
    // Rotate right
    NodePtr y = x->m_right;
    NodePtr T2 = y->m_left;
    // Rotate values
    y->m_left = x;
    x->m_right = T2;

    return y;
}

void BST::insert(string &word) {
    // Use recursive insert function/update root node
    m_root = insert(word, m_root);
}

Node::NodePtr BST::insert(string &word, NodePtr &node) {
    // Recursive insert function
    if (node == nullptr) {
        // Create new node and store data
        node = new Node();
        node->m_data = word;

        // Return new node
        return node;

    } else if (word.compare(node->m_data) < 0) {
        // Go left
        node->m_left = insert(word, node->m_left);
    } else if (word.compare(node->m_data) > 0) {
        // Go right
        node->m_right = insert(word, node->m_right);
    } else {
        return node;
    }

    // Run balancing
    int bf = balance_factor(node);

    // Left-Left rotate
    if (bf > 1 && compare(word, node->m_left) < 0) {
        return rotate_right(node);
    }
    //Right-Right rotate
    if (bf < -1 && compare(word, node->m_right) > 0) {
        return rotate_left(node);
    }
    // Left-Right rotate
    if (bf > 1 && compare(word, node->m_left) > 0) {
        node->m_left = rotate_left(node->m_left);
        return rotate_right(node);
    }
    // Right-Left rotate
    if (bf < -1 && compare(word, node->m_right) < 0) {
        node->m_right = rotate_right(node->m_right);
        return rotate_left(node);
    }

    // Return original node
    return node;
}

bool BST::find(string &word) {
    // Check if word is in dictionary
    NodePtr node = m_root;
    bool found = false;

    // Look for desired node
    while (node != nullptr) {
        if (compare(word, node) < 0) {
            // Continue left
            node = node->m_left;
        } else if (compare(word, node) > 0) {
            // Continue right
            node = node->m_right;
        } else if (compare(word, node) == 0) {
            // Found desired node
            found = true;
            break;
        }
    }
    return found;
}

int BST::compare(string &word, NodePtr &node) {
    // Use string compare() to determine string comparison
    return word.compare(node->m_data);
}

void BST::print_tree(ostream &output, NodePtr &node, int indent) {
    // Recursive printing function
    if (node != nullptr) {
        // Pass in right side first
        print_tree(output, node->m_right, indent + 4);
        // Add current node to output
        output << setw(indent + node->m_data.length());
        output << node->m_data << endl;
        // Pass in left side
        print_tree(output, node->m_left, indent + 4);
    }
}

ostream &operator<<(ostream &output, BST &bst) {
    // Use recursive print_tree function
    bst.print_tree(output, bst.m_root, 0);
    return output;
}

```

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

using namespace std;

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

    SpellChecker sc;

    string input_file;
    string output_file = "../output/output.txt";
    string dictionary = "../docs/dictionary.txt";

    switch (argc) {
        case 4:
            input_file = argv[1];
            output_file = argv[2];
            dictionary = argv[3];
            break;
        case 3:
            input_file = argv[1];
            output_file = argv[2];
            break;
        case 2:
            input_file = argv[1];
            break;
        default:
            // Prompt user for file name(s) if not given
            cout << "Incorrect number / improper file types specified." << endl;

            cout << "Please specify relative path to input file: ";
            getline(cin, input_file);

            string input;

            cout << "Please specify relative path to output file or '-d'"
                 << " for default output file: ";
            getline(cin, input);

            if (input != "-d") {
                output_file = input;
            }

            cout << "Please specify relative path to dictionary file or '-d'"
                 << " for default dictionary file: ";
            getline(cin, input);

            if (input != "-d") {
                dictionary = input;
            }
    }

    // Run SpellCheck
    sc.check_words(input_file, output_file, dictionary);

    return 0;
}
```

```
#ifndef ASSIGNMENT_3_SPELLCHECKER_H
#define ASSIGNMENT_3_SPELLCHECKER_H

#include "BST.h"

class SpellChecker {
private:
    BST m_bst;
public:
    ~SpellChecker();

    void check_words(std::string &, std::string &, std::string &);

    void read_dictionary(std::string &);

    std::string remove_chars(std::string &);

    void save_tree(std::string &);

    void display_tree();
};

#endif //ASSIGNMENT_3_SPELLCHECKER_H
```

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

using namespace std;

void SpellChecker::check_words(string &input_file, string &output_file,
                               string &dict_file) {

    // Read in dictionary words
    read_dictionary(dict_file);

    fstream file;
    string word;

    try {
        // Open text file
        file.open(input_file.c_str());

        if (file.is_open()) {
            // Loop through words
            while (file >> word) {
                // Wash words of any special chars & convert to lowercase
                word = remove_chars(word);
                // Check if word is empty after wash
                if (!word.empty()) {
                    // Check if word is found in BST, output if not
                    if (!m_bst.find(word)) {
                        cout << word << endl;
                    }
                }
            }
        }
    } catch (exception &e) {
        cout << e.what();
    }
    file.close();

    // Save BST to file
    save_tree(output_file);

    cout << endl << "BST saved to " << output_file << "." << endl;
}

void SpellChecker::read_dictionary(string & file_name) {
    fstream file;
    string word;

    try {
        // Open dictionary file
        file.open(file_name.c_str());
        if (file.is_open()) {
            // Populate BST with dictionary words
            while (file >> word) {
                m_bst.insert(word);
            }
        }
    } catch (exception &e) {
        cout << e.what();
    }
    file.close();
}

string SpellChecker::remove_chars(string &word) {
    // Loop through word and erase special chars
    for (auto i = 0; i < word.size(); i++) {
        if (word[i] < 'A' || word[i] > 'Z' &&
                             word[i] < 'a' || word[i] > 'z') {
```

```
            // Erase non alphabetic char
            word.erase(i, 1);
            i--;
        }
        // Convert chars to lowercase
        word[i] = tolower(word[i]);
    }
    return word;
}

void SpellChecker::save_tree(string &file_name) {
    fstream file;

    try {
        file.open(file_name.c_str(), ios::out | ios::trunc);

        if (file.is_open()) {
            // Save BST to file
            file << m_bst;
        }
    } catch (exception &e) {
        cout << e.what();
    }
    file.close();
}

void SpellChecker::display_tree() {
    // Display tree to console
    cout << m_bst << endl;
}

SpellChecker::~SpellChecker() = default;
```