

```
1 #ifndef ASSIGNMENT_3_BST_H
2 #define ASSIGNMENT_3_BST_H
3
4 #include "Node.h"
5
6 class BST {
7     using NodePtr = Node *;
8
9 private:
10     NodePtr m_root{nullptr};
11
12     NodePtr insert(std::string &, NodePtr &);
13
14     void print_tree(std::ostream &, NodePtr &,
15         int);
16 public:
17     int height(NodePtr &);
18
19     int max(int, int);
20
21     int balance_factor(NodePtr &);
22
23     NodePtr rotate_right(NodePtr &);
24
25     NodePtr rotate_left(NodePtr &);
26
27     void insert(std::string &);
28
29     static int compare(std::string &, NodePtr &);
30
31     bool find(std::string &);
32
33     friend std::ostream &operator<<(std::ostream
34         &, BST &);
35
36 #endif //ASSIGNMENT_3_BST_H
```

```
1 #ifndef ASSIGNMENT_3_NODE_H
2 #define ASSIGNMENT_3_NODE_H
3
4 #include <iomanip>
5
6 struct Node {
7     using NodePtr = Node *;
8
9     std::string m_data{"NULL"};
10    NodePtr m_left{nullptr};
11    NodePtr m_right{nullptr};
12 };
13
14 #endif //ASSIGNMENT_3_NODE_H
15
```

```
1 #include "BST.h"
2 #include <iostream>
3
4 using namespace std;
5
6 int BST::height(NodePtr &node) {
7     // Return height of subtree
8     if (node == nullptr)
9         return -1;
10    return max(height(node->m_left), height(node
->m_right)) + 1;
11 }
12
13 int BST::max(int a, int b) {
14     // Return greater value of two ints
15     return (a > b) ? a : b;
16 }
17
18 int BST::balance_factor(NodePtr &node) {
19     // Return balance factor of left and right
    subtrees of given node
20     if (node == nullptr)
21         return 0;
22     return height(node->m_left) - height(node->
m_right);
23 }
24
25 NodePtr BST::rotate_right(NodePtr &y) {
26     // Rotate left
27     NodePtr x = y->m_left;
28     NodePtr T2 = x->m_right;
29     // Rotate values
30     x->m_right = y;
31     y->m_left = T2;
32
33     return x;
34 }
35
```

```
36 Node::NodePtr BST::rotate_left(NodePtr &x) {
37     // Rotate right
38     NodePtr y = x->m_right;
39     NodePtr T2 = y->m_left;
40     // Rotate values
41     y->m_left = x;
42     x->m_right = T2;
43
44     return y;
45 }
46
47 void BST::insert(string &word) {
48     // Use recursive insert function/update root
    node
49     m_root = insert(word, m_root);
50 }
51
52 Node::NodePtr BST::insert(string &word, NodePtr &
    node) {
53     // Recursive insert function
54     if (node == nullptr) {
55         // Create new node and store data
56         node = new Node();
57         node->m_data = word;
58
59         // Return new node
60         return node;
61
62     } else if (word.compare(node->m_data) < 0) {
63         // Go left
64         node->m_left = insert(word, node->m_left
    );
65     } else if (word.compare(node->m_data) > 0) {
66         // Go right
67         node->m_right = insert(word, node->
    m_right);
68     } else {
69         return node;
```

```

70     }
71
72     // Run balancing
73     int bf = balance_factor(node);
74
75     // Left-Left rotate
76     if (bf > 1 && compare(word, node->m_left) <
0) {
77         return rotate_right(node);
78     }
79     //Right-Right rotate
80     if (bf < -1 && compare(word, node->m_right
) > 0) {
81         return rotate_left(node);
82     }
83     // Left-Right rotate
84     if (bf > 1 && compare(word, node->m_left) >
0) {
85         node->m_left = rotate_left(node->m_left
);
86         return rotate_right(node);
87     }
88     // Right-Left rotate
89     if (bf < -1 && compare(word, node->m_right
) < 0) {
90         node->m_right = rotate_right(node->
m_right);
91         return rotate_left(node);
92     }
93
94     // Return original node
95     return node;
96 }
97
98 bool BST::find(string &word) {
99     // Check if word is in dictionary
100     NodePtr node = m_root;
101     bool found = false;

```

```

102
103     // Look for desired node
104     while (node != nullptr) {
105         if (compare(word, node) < 0) {
106             // Continue left
107             node = node->m_left;
108         } else if (compare(word, node) > 0) {
109             // Continue right
110             node = node->m_right;
111         } else if (compare(word, node) == 0) {
112             // Found desired node
113             found = true;
114             break;
115         }
116     }
117     return found;
118 }
119
120 int BST::compare(string &word, NodePtr &node) {
121     // Use string compare() to determine string
122     comparison
123     return word.compare(node->m_data);
124 }
125
126 void BST::print_tree(ostream &output, NodePtr &
127     node, int indent) {
128     // Recursive printing function
129     if (node != nullptr) {
130         // Pass in right side first
131         print_tree(output, node->m_right, indent
132             + 4);
133         // Add current node to output
134         output << setw(indent + node->m_data.
135             length());
136         output << node->m_data << endl;
137         // Pass in left side
138         print_tree(output, node->m_left, indent
139             + 4);

```

```
135     }
136 }
137
138 ostream &operator<<(ostream &output, BST &bst) {
139     // Use recursive print_tree function
140     bst.print_tree(output, bst.m_root, 0);
141     return output;
142 }
143
```

```
1 #include <iostream>
2 #include "SpellChecker.h"
3
4 using namespace std;
5
6 int main(int argc, char **argv) {
7
8     SpellChecker sc;
9
10    string input_file;
11    string output_file = "../output/output.txt";
12    string dictionary = "../docs/dictionary.txt";
13
14    switch (argc) {
15        case 4:
16            input_file = argv[1];
17            output_file = argv[2];
18            dictionary = argv[3];
19            break;
20        case 3:
21            input_file = argv[1];
22            output_file = argv[2];
23            break;
24        case 2:
25            input_file = argv[1];
26            break;
27        default:
28            // Prompt user for file name(s) if
29            not given
30            cout << "Incorrect number / improper
31            file types specified." << endl;
32
33            cout << "Please specify relative path
34            to input file: ";
35            getline(cin, input_file);
```



```
36         cout << "Please specify relative path
    to output file or '-d'"
37         << " for default output file: ";
38         getline(cin, input);
39
40         if (input != "-d") {
41             output_file = input;
42         }
43
44         cout << "Please specify relative path
    to dictionary file or '-d'"
45         << " for default dictionary file
    : ";
46         getline(cin, input);
47
48         if (input != "-d") {
49             dictionary = input;
50         }
51     }
52
53     // Run SpellCheck
54     sc.check_words(input_file, output_file,
    dictionary);
55
56     return 0;
57 }
```

```
1 #ifndef ASSIGNMENT_3_SPELLCHECKER_H
2 #define ASSIGNMENT_3_SPELLCHECKER_H
3
4 #include "BST.h"
5
6 class SpellChecker {
7 private:
8     BST m_bst;
9 public:
10     ~SpellChecker();
11
12     void check_words(std::string &, std::string
    &, std::string &);
13
14     void read_dictionary(std::string &);
15
16     std::string remove_chars(std::string &);
17
18     void save_tree(std::string &);
19
20     void display_tree();
21 };
22
23 #endif //ASSIGNMENT_3_SPELLCHECKER_H
```

```

1 #include "SpellChecker.h"
2 #include<iostream>
3 #include <fstream>
4
5 using namespace std;
6
7 void SpellChecker::check_words(string &input_file
8     , string &output_file,
9         string &dict_file
10    ) {
11
12    // Read in dictionary words
13    read_dictionary(dict_file);
14
15    fstream file;
16    string word;
17
18    try {
19        // Open text file
20        file.open(input_file.c_str());
21
22        if (file.is_open()) {
23            // Loop through words
24            while (file >> word) {
25                // Wash words of any special
26                chars & convert to lowercase
27                word = remove_chars(word);
28                // Check if word is empty after
29                wash
30
31                if (!word.empty()) {
32                    // Check if word is found in
33                    BST, output if not
34
35                    if (!m_bst.find(word)) {
36                        cout << word << endl;
37                    }
38                }
39            }
40        }
41    }
42
43 }

```

```

34     } catch (exception &e) {
35         cout << e.what();
36     }
37     file.close();
38
39     // Save BST to file
40     save_tree(output_file);
41
42     cout << endl << "BST saved to " <<
    output_file << "." << endl;
43 }
44
45 void SpellChecker::read_dictionary(string &
    file_name) {
46     fstream file;
47     string word;
48
49     try {
50         // Open dictionary file
51         file.open(file_name.c_str());
52         if (file.is_open()) {
53             // Populate BST with dictionary words
54             while (file >> word) {
55                 m_bst.insert(word);
56             }
57         }
58     } catch (exception &e) {
59         cout << e.what();
60     }
61     file.close();
62 }
63
64 string SpellChecker::remove_chars(string &word) {
65     // Loop through word and erase special chars
66     for (auto i = 0; i < word.size(); i++) {
67         if (word[i] < 'A' || word[i] > 'Z' &&
68             word[i] < 'a' ||
        word[i] > 'z') {

```

```
69         // Erase non alphabetic char
70         word.erase(i, 1);
71         i--;
72     }
73     // Convert chars to lowercase
74     word[i] = tolower(word[i]);
75 }
76 return word;
77 }
78
79 void SpellChecker::save_tree(string &file_name
    ) {
80     fstream file;
81
82     try {
83         file.open(file_name.c_str(), ios::out |
            ios::trunc);
84
85         if (file.is_open()) {
86             // Save BST to file
87             file << m_bst;
88         }
89     } catch (exception &e) {
90         cout << e.what();
91     }
92     file.close();
93 }
94
95 void SpellChecker::display_tree() {
96     // Display tree to console
97     cout << m_bst << endl;
98 }
99
100 SpellChecker::~SpellChecker() = default;
```