```
File - F: \ SEM4-NSCC \ PROG\_2400 \ Assignments \ assignment-3-amkaminski \ src \ Node. has a signment \ Assignment \ As
      1 //
      2 // Header file describing 'Node' members and features
      4 // Created by am_ka on 2021-03-01.
      7 #ifndef ASSIGNMENT_3_NODE_H
      8 #define ASSIGNMENT_3_NODE_H
  10 #include <iostream>
  11 #include <iomanip>
  12
  13 using namespace std;
  14
  15 // forward declaration
  16 class Node;
  17
  18 typedef Node* NodePtr;
  19
  20 class Node {
  21 public:
  22
                               string data; // Value held
  23
  24
                              // Pointers to subsequent Nodes branching off
  25
                              // current Node
  26
                               NodePtr left;
                               NodePtr right;
  27
  28
  29
                              // Default Node Constructor
                               Node() : data(""), left(nullptr), right(nullptr
  30
               ) {}
  31 };
  32
  33 #endif //ASSIGNMENT_3_NODE_H
  34
```

```
File - F: \ SEM4-NSCC \ PROG\_2400 \ Assignments \ assignment-3-amkaminski \ STree.h
 1 //
 2 // Header file describing 'BSTree' members and
   features
 3 //
 4 // Created by am_ka on 2021-03-01.
 5 //
 7 #ifndef ASSIGNMENT_3_BSTREE_H
 8 #define ASSIGNMENT_3_BSTREE_H
10 #include "Node.h"
11 #include <vector>
12
13 // Console Output Color Values
14 const int RED_TEXT = 4;
15 const int WHITE_TEXT = 7;
16
17 class BSTree {
18 private:
       NodePtr root; // Head of the Binary Search Tree
19
20
       vector<string> allWords; // Vector to store the
   ordered data
21
22 public:
23
       BSTree() : root(nullptr) {} // Constructor
24
       virtual ~BSTree(); // Destructor
25
26
       void Delete(); // Invoke DeleteTree
27
       void DeleteTree(NodePtr node); // Remove Tree
   Contents
28
       void Insert(string word);
29
       void Insert(string word, NodePtr& node);
30
       void Remove(string word);
31
32
       void LoadTree(string filename); // Load File
   Contents into Tree
33
34
       void Balance(); // Invoke Recursive Balancing
   Function
35
       void Balance(vector<string> aList);
36
37
       void CheckFile(string filename); // Check File
   Contents against Tree
38
       bool CheckIfNodeExists(string key);
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\BSTree.h
        bool CheckIfNodeExists(NodePtr node, string key);
39
40
        void Color(int c); // Coloring Function
41
        string toLower(string word); // Bring a given
42
   string to all lower case
43
        void PrintTree(ostream& output, NodePtr& node,
44
   int indent);
45
        // Output Operator Friend Function
46
47
        friend ostream& operator<< (ostream& output,</pre>
   BSTree& tree);
48
49 };
50 #endif //ASSIGNMENT_3_BSTREE_H
51
```

```
\label{lem:file-F} File-F:\colored{File-File-SCC\colored} For all the colored constants and the colored colo
     1 #include <iostream>
     2 #include <fstream>
     3
     4 #include "Node.h"
     5 #include "BSTree.h"
     7 using namespace std;
     9 void TreeToFile(BSTree aTree); // File writing
          prototype
 10
 11 int main() {
 12
                        BSTree tree; // Declare tree
 13
 14
                        tree.LoadTree("dictionary.txt"); // Load file
           contents into tree
 15
 16
                        // Test 1
 17
                        cout << "\n\nDictionary Tree (Unbalanced)" <<</pre>
           endl;
                        cout << "----" << endl:
 18
 19
                        cout << tree << endl;</pre>
 20
 21
 22
                        // Test 2
 23
                        cout << "\n\nDictionary Tree (Balanced)" << endl;</pre>
                        cout << "----" << endl;
 24
 25
                        tree.Balance();
                        cout << tree << endl;</pre>
 26
 27
 28
                        // Test 3
                        cout << "\n\nSpell Check Test" << endl;</pre>
 29
                        cout << "----" << endl;
 30
                        tree.CheckFile("mispelled.txt");
 31
 32
                        TreeToFile(tree); // Write Tree to File
 33
 34
 35
 36
                        return 0;
 37
 38 } // End Main
 39
 40
 41 void TreeToFile(BSTree aTree) { // Redirect BST
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\main.cpp
41 output to a text file
42
43
        fstream treeFile;
        treeFile.open("..\\docs\\tree_file.txt", ios::out
44
   );
45
        string branch;
46
47
        // Back up stream buffer
        streambuf* stream_buffer_cout = cout.rdbuf();
48
49
        // Get stream buffer of file
50
51
        streambuf* stream_buffer_file = treeFile.rdbuf();
        cout.rdbuf(stream_buffer_file);
52
53
54
        cout << aTree;</pre>
55
        cout.rdbuf(stream_buffer_cout); // Redirect
56
   output to console
57
        treeFile.close();
58
59 } // End TreeToFile()
60
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\Node.cpp

1 //
2 // Created by am_ka on 2021-03-01.
3 //
4
5 #include "Node.h"
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
 1 //
 2 // cpp Source file defining 'BSTree' functionality
 3 //
 4 // Created by am_ka on 2021-03-01.
 5 //
 6
 7 #include "BSTree.h"
 8 #include "Node.h"
 9 #include <vector>
10 #include <algorithm>
11 #include <cstring>
12 #include <fstream>
13 #include <winnt.h>
14 #include <afxres.h>
15
16
17 using namespace std;
18
19
20
21 BSTree::~BSTree() { // Destructor
        DeleteTree(root);
23 } // End Destructor()
24
25
26 void BSTree::Delete() { // Invoke DeleteTree()
27
28
        DeleteTree(root);
29
        delete root;
30
        root = nullptr;
31 }
32
33
34 void BSTree::DeleteTree(NodePtr node) { // Delete
   Tree Contents Recursively
35
36
        if (node != nullptr){
37
38
            DeleteTree(node->left);
39
            DeleteTree(node->right);
40
41
            delete node;
            node = nullptr; // Just to be Safe
42
43
        }
```

```
File - F:\SEM4-NSCC\PROG 2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
44 } // End DeleteTree()
45
46
47 // Invoke Recursive Invoke Function
48 void BSTree::Insert(string word) {
49
50
        Insert(word, root);
51
52 } // End Insert()
53
54
55 // Navigate the search tree Recursively to Insert New
    Element/Node
56 void BSTree::Insert(string word, NodePtr& node) {
57
        if (node == nullptr) {
58
59
60
            node = new Node();
61
            node->data = word;
62
63
            // If inserted word is not in the
   alphabetizing vector 'allWords', add it to vector
            if (!(find(allWords.begin(), allWords.end(),
64
   word) != allWords.end())){
65
                allWords.push_back(word);
66
                sort(allWords.begin(), allWords.end());
            }
67
68
69
        } else if (word < node->data) {
70
71
72
            Insert(word, node->left);
73
        } else if (word > node->data) {
74
75
            Insert(word, node->right);
76
77
        } else { // Word already exists within BS Tree
78
79
            cout << "WARNING: '"<< node->data << "'
80
   Already Exists in the Current Context" << endl;
81
        }
82 } // End Insert()
83
```

```
84
 85 void BSTree::Remove(string word) { // Remove Element
     From BS Tree
 86
 87
        NodePtr node = root;
 88
        NodePtr parent = nullptr;
 89
 90
        while (node != nullptr) {
            if (word < node->data) { // Go Left
 91
 92
                parent = node;
 93
                node = node->left;
 94
            } else if (word > node->data) { // Go Right
 95
                parent = node;
 96
                node = node->right;
 97
            } else {
 98
                break;
 99
            }
        }
100
101
        if (node == nullptr) { // Tree Search Failed
102
            cout << "'Remove()' Could Not Locate " <<
103
    word << endl;</pre>
        }
104
105
        // If child has two children, use right most
106
    Node
107
        // of left tree as successor
        if (node->left != nullptr && node->right !=
108
    nullptr) {
109
110
            NodePtr successor = node->left; // start at
    left of tree
111
112
            // Keep going right as far as possible
113
            parent = node;
114
            while(successor->right != nullptr) {
115
                parent = successor;
116
                successor = successor->right;
            }
117
118
119
            // Swap data with successor and successor is
     now the one to delete
            node->data = successor->data;
120
121
            node = successor;
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
         }
122
123
124
         // now the node to delete must have only one or
    no children
125
126
         // assume there is a left child
127
         NodePtr subtree = node->left;
128
129
         // if no left child, maybe a right child
130
         if (subtree == nullptr) {
131
             subtree = node->right;
132
         }
133
134
         // connect any children to new parents
135
         if (parent == nullptr) {
136
             // must be the root node
137
             root = subtree;
138
         } else if (parent->left == node) {
139
             // deleting a left node of a parent
140
             parent->left = subtree;
141
         } else if (parent->right == node) {
142
             //deleting a right node of a parent
143
             parent->right = subtree;
         }
144
145
146
         // Iterate through allWords vector
147
         for (unsigned i = 0; i < allWords.size(); i++)</pre>
148
         {
             if( word == allWords.at(i))
149
150
             {
151
                 allWords.erase(allWords.begin() + i);
152
             }
         }
153
154
155
         delete node; // FINALLY
156
157 } // End Remove()
158
159
160 // Read text file and load individual words into
    BSTree
161 void BSTree::LoadTree(string filename) {
162
163
         string row;
```

```
File - F:\SEM4-NSCC\PROG 2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
164
         ifstream dictFile;
         dictFile.open("..\\docs\\" + filename);
165
166
         if (!dictFile.fail()) {
167
             while (getline(dictFile, row)){
                  Insert(row);
168
             }
169
         }
170
171
172
         dictFile.close();
173
174 } // End LoadTree()
175
176
177 void BSTree::Balance() { // Invoke Recursive Bαlance
      Function
178
179
         Delete();
         Balance(allWords);
180
181
182 } // End Balance()
183
184
185 // Recursively Balance BS Tree
186 void BSTree::Balance(vector<string> aList) {
187
188
         // Grab the Central Value of the aList Vector
189
         unsigned middle = aList.size() / 2;
190
         string pivot = aList[middle];
191
192
         Insert(pivot); // Insert the Central Value into
    Tree
193
194
        // If the Vector is larger than 2 values, split
    it into two
195
         // and feed the new Vectors into new invocations
      of Balance()
         if (aList.size() > 2) {
196
197
198
             vector<string> left;
199
             vector<string> right;
200
201
             // Load larger vector contents into
202
             // the two smaller vectors
             for (unsigned i = 0; i < aList.size(); i++)</pre>
203
                            Page 5 of 9
```

```
File - F:\SEM4-NSCC\PROG 2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
204
             {
                  if(pivot != aList.at(i))
205
                  {
206
                      if (i < middle) {</pre>
207
208
                          left.push_back(aList.at(i));
209
210
                      } else {
211
212
213
                           right.push_back(aList.at(i));
214
                      }
                  }
215
             }
216
217
218
             // Invoke this Function on the new Vectors
             Balance(left);
219
220
             Balance(right);
221
         } else if (aList.size() == 2) {
222
223
224
             // If aList has only 2 values, one has
     already been
225
             // inserted, so insert the last one
226
             Insert(aList.at(0));
227
         }
228
229 } // End Balance()
230
231
232 // Check a text file against the contents of the BS
     Tree
233 void BSTree::CheckFile(string filename) {
234
235
         string word;
236
         ifstream aFile;
237
         vector<string>tokens;
         aFile.open("..\\docs\\" + filename);
238
239
         // Tokenize each word in the file and add it to
240
      'tokens'
241
         if (!aFile.fail()) {
242
             while (getline(aFile, word, ' ')){
243
244
                  // Remove any non alphanumeric
                            Page 6 of 9
```

```
244 characters from word
245
                 for (string::iterator i = word.begin();
    i != word.end(); i++) {
                     if(!isalpha(word.at(i - word.begin
246
    ())))
247
                     {
                         word.erase(i);
248
249
                          i--;
                     }
250
251
252
                 tokens.push_back(word);
            }
253
254
255
            for (int i = 0; i < tokens.size(); i++) {</pre>
256
                 // If token exists in tree, output
257
    normally to console
                 if (CheckIfNodeExists(toLower(tokens[i
258
    ])))
259
                 {
                     cout << tokens[i] << ' ';
260
261
262
                 } else { // Output token as red if not
    in tree
263
                     Color(RED_TEXT);
264
265
                     cout << tokens[i] << ' ';
266
                     Color(WHITE_TEXT);
                 }
267
268
                 if (i % 5 == 0) { // Carriage return
269
    after 5 words
270
271
                     cout << "\n";
                 }
272
273
274
            }
275
276
277
            cout << "\n";
278
        }
279
        aFile.close();
280
281
```

```
File - F:\SEM4-NSCC\PROG 2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
282 } // End CheckFile()
283
284
285 // Methods to recursively search through BS Tree,
286 // to match key (a word) against each value
287 bool BSTree::CheckIfNodeExists(string key) {
288
289
         return CheckIfNodeExists(root, key);
290 }
291
292 bool BSTree::CheckIfNodeExists(NodePtr node, string
    key) {
293
294
         if (node == nullptr) {
295
296
             return false;
         }
297
298
299
         if (node->data == key) {
300
301
             return true;
302
         }
303
304
         bool leftCheck = CheckIfNodeExists(node->left,
    key);
         if (leftCheck){
305
306
307
             return true;
         }
308
309
310
         bool rightCheck = CheckIfNodeExists(node->right
     , key);
311
312
         return rightCheck;
313
314 } // End CheckIfNodeExists()
315
316
317 // Method to color console output
318 void BSTree::Color(int c) {
319
320
         HANDLE hConsole;
         hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
321
322
         SetConsoleTextAttribute(hConsole, c);
                            Page 8 of 9
```

```
File - F:\SEM4-NSCC\PROG_2400\Assignments\assignment-3-amkaminski\src\BSTree.cpp
323
         return;
324
325 } // End Color()
326
327
328 // Convert a string to all lower case characters
329 string BSTree::toLower(string word){
330
         transform(word.begin(), word.end(), word.begin
331
     (), ::tolower);
         return word;
332
333
334 } // End toLower()
335
336
337 // Recursively print out Tree Contents
338 void BSTree::PrintTree(ostream& output, NodePtr&
    node, int indent) {
339
         if (node != nullptr) {
             PrintTree(output, node->right, indent + 5);
340
341
             output << setw(indent) << node->data << endl</pre>
342
             PrintTree(output, node->left, indent + 5);
343
344 } // End PrintTree()
345
346
347 // Define Output Operator Functionality
348 ostream& operator<<(ostream& output, BSTree& bst) {
349
         bst.PrintTree(output, bst.root, 5);
350
         return output;
351 }
352
353
```