```cpp
//
// Header file describing 'Node' members and features
//
// Created by am_ka on 2021-03-01.
//

#ifndef ASSIGNMENT_3_NODE_H
#define ASSIGNMENT_3_NODE_H

#include <iostream>
#include <iomanip>

using namespace std;

// forward declaration
class Node;

typedef Node* NodePtr;

class Node {
public:
    string data; // Value held

    // Pointers to subsequent Nodes branching off
    // current Node
    NodePtr left;
    NodePtr right;

    // Default Node Constructor
    Node() : data(""), left(nullptr), right(nullptr) {}
};

#endif //ASSIGNMENT_3_NODE_H

```

```cpp
//
// Header file describing 'BSTree' members and features
//
// Created by am_ka on 2021-03-01.
//

#ifndef ASSIGNMENT_3_BSTREE_H
#define ASSIGNMENT_3_BSTREE_H

#include "Node.h"
#include <vector>

// Console Output Color Values
const int RED_TEXT = 4;
const int WHITE_TEXT = 7;

class BSTree {
private:
    NodePtr root; // Head of the Binary Search Tree
    vector<string> allWords; // Vector to store the ordered data

public:
    BSTree() : root(nullptr) {} // Constructor
    virtual ~BSTree(); // Destructor

    void Delete(); // Invoke DeleteTree
    void DeleteTree(NodePtr node); // Remove Tree Contents
    void Insert(string word);
    void Insert(string word, NodePtr& node);
    void Remove(string word);

    void LoadTree(string filename); // Load File Contents into Tree

    void Balance(); // Invoke Recursive Balancing Function
    void Balance(vector<string> aList);

    void CheckFile(string filename); // Check File Contents against Tree
    bool CheckIfNodeExists(string key);
    bool CheckIfNodeExists(NodePtr node, string key);

    void Color(int c); // Coloring Function
    string toLower(string word); // Bring a given string to all lower case

    void PrintTree(ostream& output, NodePtr& node, int indent);

    // Output Operator Friend Function
    friend ostream& operator<< (ostream& output, BSTree& tree);

};
#endif //ASSIGNMENT_3_BSTREE_H

```

```cpp
#include <iostream>
#include <fstream>

#include "Node.h"
#include "BSTree.h"

using namespace std;

void TreeToFile(BSTree aTree); // File writing prototype

int main() {

    BSTree tree; // Declare tree
    tree.LoadTree("dictionary.txt"); // Load file contents into tree

    // Test 1
    cout << "\n\nDictionary Tree (Unbalanced)" << endl;
    cout << "----------------------------" << endl;
    cout << tree << endl;


    // Test 2
    cout << "\n\nDictionary Tree (Balanced)" << endl;
    cout << "--------------------------" << endl;
    tree.Balance();
    cout << tree << endl;

    // Test 3
    cout << "\n\nSpell Check Test" << endl;
    cout << "----------------" << endl;
    tree.CheckFile("mispelled.txt");

    TreeToFile(tree); // Write Tree to File


    return 0;

} // End Main


void TreeToFile(BSTree aTree) { // Redirect BST
output to a text file

    fstream treeFile;
    treeFile.open("..\\docs\\tree_file.txt", ios::out);
    string branch;

    // Back up stream buffer
    streambuf* stream_buffer_cout = cout.rdbuf();

    // Get stream buffer of file
    streambuf* stream_buffer_file = treeFile.rdbuf();
    cout.rdbuf(stream_buffer_file);

    cout << aTree;

    cout.rdbuf(stream_buffer_cout); // Redirect output to console
    treeFile.close();

} // End TreeToFile()

```

```cpp
//
// Created by am_ka on 2021-03-01.
//

#include "Node.h"

```

```cpp
//
// cpp Source file defining 'BSTree' functionality
//
// Created by am_ka on 2021-03-01.
//

#include "BSTree.h"
#include "Node.h"
#include <vector>
#include <algorithm>
#include <cstring>
#include <fstream>
#include <winnt.h>
#include <afxres.h>


using namespace std;



BSTree::~BSTree() { // Destructor
    DeleteTree(root);
} // End Destructor()


void BSTree::Delete() { // Invoke DeleteTree()

    DeleteTree(root);
    delete root;
    root = nullptr;
}


void BSTree::DeleteTree(NodePtr node) { // Delete Tree Contents Recursively

    if (node != nullptr){

        DeleteTree(node->left);
        DeleteTree(node->right);

        delete node;
        node = nullptr; // Just to be Safe
    }
} // End DeleteTree()


// Invoke Recursive Invoke Function
void BSTree::Insert(string word) {

    Insert(word, root);

} // End Insert()


// Navigate the search tree Recursively to Insert New Element/Node
void BSTree::Insert(string word, NodePtr& node) {

    if (node == nullptr) {

        node = new Node();
        node->data = word;

        // If inserted word is not in the alphabetizing vector 'allWords', add it to vector
        if (!(find(allWords.begin(), allWords.end(), word) != allWords.end())){
            allWords.push_back(word);
            sort(allWords.begin(), allWords.end());
        }


    } else if (word < node->data) {

        Insert(word, node->left);

    } else if (word > node->data) {

        Insert(word, node->right);

    } else { // Word already exists within BS Tree

        cout << "WARNING: '"<< node->data << "' Already Exists in the Current Context" << endl;
    }
} // End Insert()

```

```

void BSTree::Remove(string word) { // Remove Element From BS Tree

    NodePtr node = root;
    NodePtr parent = nullptr;

    while (node != nullptr) {
        if (word < node->data) { // Go Left
            parent = node;
            node = node->left;
        } else if (word > node->data) { // Go Right
            parent = node;
            node = node->right;
        } else {
            break;
        }
    }

    if (node == nullptr) { // Tree Search Failed
        cout << "'Remove()' Could Not Locate " << word << endl;
    }

    // If child has two children, use right most Node
    // of left tree as successor
    if (node->left != nullptr && node->right != nullptr) {

        NodePtr successor = node->left; // start at left of tree

        // Keep going right as far as possible
        parent = node;
        while(successor->right != nullptr) {
            parent = successor;
            successor = successor->right;
        }

        // Swap data with successor and successor is now the one to delete
        node->data = successor->data;
        node = successor;
```

```cpp
    }

    // now the node to delete must have only one or no children

    // assume there is a left child
    NodePtr subtree = node->left;

    // if no left child, maybe a right child
    if (subtree == nullptr) {
        subtree = node->right;
    }

    // connect any children to new parents
    if (parent == nullptr) {
        // must be the root node
        root = subtree;
    } else if (parent->left == node) {
        // deleting a left node of a parent
        parent->left = subtree;
    } else if (parent->right == node) {
        //deleting a right node of a parent
        parent->right = subtree;
    }

    // Iterate through allWords vector
    for (unsigned i = 0; i < allWords.size(); i++)
    {
        if( word == allWords.at(i))
        {
            allWords.erase(allWords.begin() + i);
        }
    }

    delete node; // FINALLY

} // End Remove()


// Read text file and load individual words into BSTree
void BSTree::LoadTree(string filename) {

    string row;
    ifstream dictFile;
    dictFile.open("..\\docs\\" + filename);
    if (!dictFile.fail()) {
        while (getline(dictFile, row)){
            Insert(row);
        }
    }

    dictFile.close();

} // End LoadTree()


void BSTree::Balance() { // Invoke Recursive Balance Function

    Delete();
    Balance(allWords);

} // End Balance()


// Recursively Balance BS Tree
void BSTree::Balance(vector<string> aList) {

    // Grab the Central Value of the aList Vector
    unsigned middle = aList.size() / 2;
    string pivot = aList[middle];

    Insert(pivot); // Insert the Central Value into Tree

    // If the Vector is larger than 2 values, split it into two
    // and feed the new Vectors into new invocations of Balance()
    if (aList.size() > 2) {

        vector<string> left;
        vector<string> right;

        // Load larger vector contents into
        // the two smaller vectors
        for (unsigned i = 0; i < aList.size(); i++)
        {
            if(pivot != aList.at(i))
            {
                if (i < middle) {

                    left.push_back(aList.at(i));

                } else {

                    right.push_back(aList.at(i));
                }
            }
        }

        // Invoke this Function on the new Vectors
        Balance(left);
        Balance(right);

    } else if (aList.size() == 2) {

        // If aList has only 2 values, one has already been
        // inserted, so insert the last one
        Insert(aList.at(0));
    }

} // End Balance()


// Check a text file against the contents of the BS Tree
void BSTree::CheckFile(string filename) {

    string word;
    ifstream aFile;
    vector<string>tokens;
    aFile.open("..\\docs\\" + filename);

    // Tokenize each word in the file and add it to 'tokens'
    if (!aFile.fail()) {
        while (getline(aFile, word, ' ')){

            // Remove any non alphanumeric
characters from word
      for (string::iterator i = word.begin(); i != word.end(); i++) {
        if(!isalpha(word.at(i - word.begin())))
        {
          word.erase(i);
          i--;
        }
      }
      tokens.push_back(word);
    }

    for (int i = 0; i < tokens.size(); i++) {

      // If token exists in tree, output normally to console
      if (CheckIfNodeExists(toLower(tokens[i])))
      {
        cout << tokens[i] << ' ';

      } else { // Output token as red if not in tree

        Color(RED_TEXT);
        cout << tokens[i] << ' ';
        Color(WHITE_TEXT);
      }

      if (i % 5 == 0) { // Carriage return after 5 words

        cout << "\n";
      }


    }

    cout << "\n";
  }

  aFile.close();

```

```cpp
} // End CheckFile()


// Methods to recursively search through BS Tree,
// to match key (a word) against each value
bool BSTree::CheckIfNodeExists(string key) {

    return CheckIfNodeExists(root, key);
}

bool BSTree::CheckIfNodeExists(NodePtr node, string key) {

    if (node == nullptr) {

        return false;
    }

    if (node->data == key) {

        return true;
    }

    bool leftCheck = CheckIfNodeExists(node->left, key);
    if (leftCheck){

        return true;
    }

    bool rightCheck = CheckIfNodeExists(node->right, key);

    return rightCheck;

} // End CheckIfNodeExists()


// Method to color console output
void BSTree::Color(int c) {

    HANDLE hConsole;
    hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsole, c);
    return;

} // End Color()


// Convert a string to all lower case characters
string BSTree::toLower(string word){

    transform(word.begin(), word.end(), word.begin(), ::tolower);
    return word;

} // End toLower()


// Recursively print out Tree Contents
void BSTree::PrintTree(ostream& output, NodePtr& node, int indent) {
    if (node != nullptr) {
        PrintTree(output, node->right, indent + 5);
        output << setw(indent) << node->data << endl;
        PrintTree(output, node->left, indent + 5);
    }
} // End PrintTree()


// Define Output Operator Functionality
ostream& operator<<(ostream& output, BSTree& bst) {
    bst.PrintTree(output, bst.root, 5);
    return output;
}


```