```
#ifndef ASSIGNMENT_3_BST_H
#define ASSIGNMENT_3_BST_H

#include "Node.h"

typedef Node* NodePtr;

class BST {

private:
    Node *root;

public:
    BST();
    virtual ~BST();
    int Max(int num1, int num2);
    int Height(Node *node);
    int GetBalance(Node *node);

    Node *RotateRight(Node *y);
    Node *RotateLeft(Node *x);
    Node* Insert(Node* node, string word);
    void InsertNode(string &word);
    Node* Search(Node* root, string word);
    bool SearchResult(string &word);

    friend bool operator < (string first,string second);
    friend bool operator > (string first,string second);
    friend ostream& operator<<(ostream& output, BST& bst);

    void PrintTree(ostream& output, NodePtr& node, int indent, ofstream& outFile);
    string GetWord(Node *node);
    string ConvertToLowerCase(string &str);

    void ReadDictionary(BST &bst, string &filename);
    string ReadFileToCheck(string &filename);
    void CheckError(BST &bst, string &filename);

};


#endif //ASSIGNMENT_3_BST_H
```

```h
#ifndef ASSIGNMENT_3_NODE_H
#define ASSIGNMENT_3_NODE_H

#include <string>

using namespace std;

class Node {

public:
    string word;
    Node *left;
    Node *right;
    int height;

    Node();
    virtual ~Node();
};

#endif //ASSIGNMENT_3_NODE_H
```

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

using namespace std;
typedef Node* NodePtr;



BST::BST() {

    root = NULL;
}

BST::~BST() {

    delete root;
}

//Function to find greater number.
int BST::Max(int firstNum, int secondNum) {

    return (firstNum > secondNum) ? firstNum : secondNum;
}

//function to find height of a branch
int BST::Height(Node *node) {

    if (node == NULL) {
        return 0;
    }

    return node->height;
}

//Function for getting word
string BST::GetWord(Node *node) {

    if (node == NULL) {
        return 0;
    }

    return node->word;
}

//Function to check if the tree is balanced or not
int BST::GetBalance(Node *node) {

    if (node == NULL) {
        return 0;
    }
    // if the diff is -1 or 0 or 1, the tree is balanced
    return Height(node->left) - Height(node->right);
}

//Function to convert uppercase letter to lowercase
string BST::ConvertToLowerCase(string &str) {

    string newString;

    for(int i = 0; i < str.length(); i++) {
        newString += tolower(str[i]);
    }

    return newString;
}
```

```
//Function to right rotate the imbalanced tree
Node* BST::RotateRight(Node *node) {

    Node *subTree = node->left;
    Node *leaf = subTree->right;

    // perform rotation
    subTree->right = node;
    node->left = leaf;

    // update heights
    node->height = Max(Height(node->left),
                       Height(node->right)) + 1;
    subTree->height = Max(Height(subTree->left),
                          Height(subTree->right)) + 1;

    // return new root
    return subTree;
}

//Function to left rotate the imbalanced tree
Node* BST::RotateLeft(Node *node) {

    Node *subTree = node->right;
    Node *leaf = subTree->left;

    // perform rotation
    subTree->left = node;
    node->right = leaf;

    // update heights
    node->height = Max(Height(node->left),
                       Height(node->right)) + 1;
    subTree->height = Max(Height(subTree->left),
                          Height(subTree->right)) + 1;

    // return new root
    return subTree;
}

// operator to compare sequential letters of two strings, is less than or not
bool operator < (string firstStr, string secondStr) {

    int i = 0;
    int n = firstStr.length() < secondStr.length() ? firstStr.length() : secondStr.length();

    while(i < n) {
        if(tolower(firstStr[i]) != tolower(secondStr[i])) {
            return tolower(firstStr[i]) < tolower(secondStr[i]);
        }
        i++;
    }

    return firstStr.length() < secondStr.length();
}

// operator to compare sequential letters of two strings, is greater than or not
bool operator > (string firstStr, string secondStr) {

    int i = 0;
    int n = firstStr.length() < secondStr.length() ? firstStr.length() : secondStr.length();

    while(i < n) {
        if(tolower(firstStr[i]) != tolower(secondStr[i])) {
            return tolower(firstStr[i]) > tolower(secondStr[i]);
        }
        i++;
    }

    return firstStr.length() > secondStr.length();
}
```

```
// Function to call the recursive function to insert a word in the tree
// rooted with given node
void BST::InsertNode(string &word) {

    root = Insert(root,word);
}

//Recursive function to insert a word in the tree rooted with node and
//returns the new root of the subtree.
Node* BST::Insert(Node* node, string word) {

    // insert the word into the tree
    if (node == NULL) {
        Node *newNode = new Node();
        newNode->word = word;
        return newNode;
    }
    // determine where to insert, left ot right to the node
    if (word < GetWord(node)) {
        node->left = Insert(node->left, word);
    }
    else if (word > GetWord(node)) {
        node->right = Insert(node->right, word);
    }
    else {
        // equal words are not allowed to insert
        return node;
    }

    // update the height of the parent node
    node->height = 1 + Max(Height(node->left),Height(node->right));

    // get the balance value to check whether the node is balance or not
    // the node is balanced if int balance value = {-1, 0, 1}, otherwise imbalance
    int balance = GetBalance(node);

    // if the node becomes imbalance, then there are 4 cases
    // Left Left Case
    if (balance > 1 && word < GetWord(node->left)) {
        return RotateRight(node);
    }

    // Right Right Case
    if (balance < -1 && word > GetWord(node->right)) {
        return RotateLeft(node);
    }

    // Left Right Case
    if (balance > 1 && word > GetWord(node->left)) {
        node->left = RotateLeft(node->left);
        return RotateRight(node);
    }

    // Right Left Case
    if (balance < -1 && word < GetWord(node->right)) {
        node->right = RotateRight(node->right);
        return RotateLeft(node);
    }

    return node;
}

//Function to search a word from the tree.
bool BST::SearchResult(string &word) {

    return Search(root, ConvertToLowerCase(word)) != NULL;
}
```

```cpp
//Recursive function to search a word from the tree
Node* BST::Search(Node* root, string word) {

    if (root == NULL || root->word == word) {
        return root;
    }

    // word is greater than root's word
    if (root->word < word) {
        return Search(root->right, word);
    }

    // word is less than root's word
    return Search(root->left, word);
}

// print the balanced tree to the console and write to a file
void BST::PrintTree(ostream& output, NodePtr& node, int indent, ofstream& outFile) {

    if (node != nullptr) {
        PrintTree(output, node->right, indent + 16, outFile);
        output << setw(indent) << node->word << endl;
        outFile << setw(indent) << node->word << endl;
        PrintTree(output, node->left, indent + 16, outFile);
    }
}

// output the balance tree to the console and write to a file
ostream& operator<<(ostream& output, BST& bst) {

    ofstream outputFile;
    string outPath = "..\\output\\BSTOutput.txt";
    outputFile.open(outPath);

    if(!outputFile) {
        cout << "Could not create or write to the file" << endl;
    }
    else {
        bst.PrintTree(output, bst.root, 0, outputFile);
    }

    return output;
}


// read the file content and insert the content into a binary tree
void BST::ReadDictionary(BST &bst, string &filename) {

    string word;
    ifstream dictionary(filename);

    if (dictionary.is_open()) {
        while (getline(dictionary, word)) {
            bst.InsertNode(word);
        }
        dictionary.close();
    }
    else {
        cout << "Could not open file to read" << endl;
    }
}

// read the file content and store into a string
string BST::ReadFileToCheck(string &filename) {

    string document;
    string documentSegment;
    ifstream file(filename);

    if (file.is_open()) {
        while (getline(file, documentSegment)) {
```

```
            document += documentSegment;
        }
        file.close();
    }
    else {
        cout << "Could not open file to read" << endl;
    }

    return document;
}

// check the tree for the words in the file and if the words from the file
// are not found in the tree print the words to the console
void BST::CheckError(BST &bst, string &filename) {

    string fileData;
    fileData = ReadFileToCheck(filename);
    string words[1000];
    int j = 0;

    string newWord = "";

    for(int i = 0; i < fileData.length(); i++) {
        if (fileData[i] >= 'A' && fileData[i] <= 'Z') {
            newWord += fileData[i];
        }
        else if(fileData[i] >= 'a' && fileData[i] <= 'z') {
            newWord += fileData[i];
        }
        else if(!newWord.empty()) {
            words[j] = newWord;
            newWord = "";
            j++;
        }
    }

    int totalError = 0;

    // search for every words of the words array the tree, if not found print the word
    for(int i = 0; i < j; i++) {
        if(!bst.SearchResult(words[i])) {
            if(totalError == 0) {
                cout << "Errors found in your file" << endl;
            }
            cout << words[i] << endl;
            totalError++;
        }
    }
    if(totalError == 0) {
        cout << "No error found." << endl;
    }
}
```

File - C:\NSCC\2nd Year\2nd Semester\PROG2400 - Data Structure\assignment-3-Paraga2mp\src\main.cpp

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

using namespace std;

int main() {

    BST bst;
    string dictionary = "..\\tests\\dictionary.txt";
    bst.ReadDictionary(bst,dictionary);

    string filename = "..\\tests\\test.txt";
    string filename2 = "..\\docs\\sample.txt";

    cout << "File: " << filename << endl;
    bst.CheckError(bst,filename);
    cout << endl;

    cout << "File: " << filename2 << endl;
    bst.CheckError(bst,filename2);
    cout << endl;

    cout << bst;


    return 0;
}
```

```cpp
#include "Node.h"

Node::Node() {

    this->word = "";
    this->left = NULL;
    this->right = NULL;
    this->height = 1;
}

Node::~Node() {

    delete left;
    delete right;
}
```