Listing 1: BST.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>
#include <string>

/*-----------------------------------------------------------------------------
 * class definition
 *---------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 6 simple binary search tree.
 *
 * BST is a simple binary search tree. It allows insert, contains, and remove
 * methods, and supports similar administrative methods as earlier structures:
 * clear, isEmpty, size, overrides for assignment and stream insertion, etc.
 */
template <class T> class BST {
private:
    /**
     * @brief Node in the binary search tree.
     *
     * Node is a private inner class of BST. It represents a single node in the
     * tree. Each node has a payload of type T, and pointers to the left and
     * right subtrees below the node.
     */
    class Node {
    public:
        /**
         * @brief Construct a new Node object.
         *
         * Make a new node with the specified data value, and null left and
         * right pointers.
         *
         * @param val Value this node will hold.
         */
        Node(const T& val) : data(val), pLeft(0), pRight(0) { }

        /**
         * @brief Node payload.
         *
         * Type T payload of the node. Assumed to support assignment, equality
         * testing, less and greater than, copy constructor, and stream
         * insertion.
         */
        T data;

        /**
         * @brief Left tree pointer.
         *
         * Pointer to the node at the root of the left subtree under this node,
         * zero if there is no subtree.
         */
        Node *pLeft;

        /**
         * @brief Right tree pointer.
         *
         * Pointer to the node at the root of the right subtree under this
         * node, zero if there is no subtree.
         */
        Node *pRight;
    };

public:
    /**
     * @brief BST default constructor.
     *
     * Make an initially empty BST.
     */
    BST() : pRoot(0), n(0u) { }

```

```
75      /**
76       * @brief BST copy constructor.
77       *
78       * Make a new, deep-copy BST, just like the parameter tree.
79       *
80       * @param tree Reference to the BST to copy.
81       */
82      BST(const BST<T> &tree) { pRoot = copy(tree.pRoot); }
83
84      /**
85       * @brief Destructor.
86       *
87       * Free the memory used by this list.
88       */
89      ~BST() { clear(); }
90
91      /**
92       * @brief Clear the BST.
93       *
94       * Remove all the elements from this tree.
95       */
96      void clear();
97
98      /**
99       * @brief Search the BST for a value.
100      *
101      * @param val Value to search for.
102      * @return true if the value is in the BST, false otherwise.
103      */
104     bool contains(const T &val) const { return contains(val, pRoot); }
105
106     /**
107      * @brief Add a value to the BST.
108      *
109      * Adds a new value to the tree. If the value is already in the BST, no
110      * action is taken and the tree is not changed.
111      *
112      * @param val Value to insert into the tree.
113      *
114      * @return true if the value was inserted, false if it was a duplicate.
115      */
116     bool insert(const T &val);
117
118     /**
119      * @brief Determine if the BST is empty.
120      *
121      * Convenience method to test if the BST contains no elements.
122      *
123      * @return true if the list is empty, false otherwise.
124      */
125     bool isEmpty() { return n == 0u; }
126
127     /**
128      * @brief BST stream insertion override.
129      *
130      * Writes a pre-order traversal printing of the tree to the specified
131      * output stream.
132      *
133      * @param out std::ostream object to write tree contents to.
134      * @param tree BST object to print.
135      * @return std::ostream& Reference to the object written to.
136      */
137     friend std::ostream &operator<<(std::ostream &out, const BST<T> & tree) {
138         out << "[";
139         tree.preOrderPrint(out, tree.pRoot);
140         out << "]";
141         return out;
142     }
143
144     /**
145      * @brief BST assignment operator override.
146      *
147      * Makes this tree a new, deep-copy clone of the specified BST.
148      *
149      * @param tree BST to copy from.
150      * @return BST<T>& Reference to this object, for chaining.
```

```
151       */
152      BST<T> &operator=(const BST<T> &tree);
153
154      /**
155       * @brief Remove an element from the BST.
156       *
157       * Remove the specified value from the BST. If the value was not in the
158       * tree to begin with, no action is taken and the tree is not modified.
159       *
160       * @param val Value to remove from the BST.
161       *
162       * @return true if the value was removed from the tree, false if it was not
163       * in the tree initially.
164       */
165      bool remove(const T &val);
166
167      /**
168       * @brief Get the size of this BST.
169       *
170       * @return size_t Number of elements in the tree.
171       */
172      size_t size() { return n; }
173
174  private:
175      /**
176       * @brief Pointer to the root node of the tree.
177       *
178       */
179      Node *pRoot;
180
181      /**
182       * @brief Number of elements in the tree.
183       *
184       */
185      size_t n;
186
187      /**
188       * @brief Clear helper method.
189       *
190       * Private, recursive clear method.
191       *
192       * @param pCurr Pointer to the root of the subtree to remove.
193       */
194      void clear(Node *pCurr);
195
196      /**
197       * @brief Contains helper method.
198       *
199       * Private, recursive contains method.
200       *
201       * @param val Value to search the tree for.
202       * @param pCurr Pointer to the root of the subtree to search.
203       * @return true if the subtree contains the value, false otherwise.
204       */
205      bool contains(const T &val, Node *pCurr) const;
206
207      /**
208       * @brief Copy helper method.
209       *
210       * Private, recursive copy method.
211       *
212       * @param pOtherCurr Pointer to the root of the subtree *in the other tree*
213       * @return Node* Pointer to the node added *to this tree*
214       */
215      Node *copy(Node *pOtherCurr);
216
217      /**
218       * @brief Stream insertion helper method.
219       *
220       * Private, recursive helper method for stream insertion.
221       *
222       * @param out Output stream to print to.
223       * @param pCurr Pointer to the root of the subtree to print.
224       */
225      void preOrderPrint(std::ostream &out, Node *pCurr) const;
226
```

```
227 };
228
229 //--------------------------------------------------------------------------
230 // function implementations
231 //--------------------------------------------------------------------------
232
233 // doctest unit test for copy constructor
234 TEST_CASE("testing␣BST␣copy␣constructor") {
235     BST<int> tree1;
236     int values[10] = {80, 9, 3, 34, 33, 63, 81, 55, 86, 9};
237     for(int i = 0; i < 10; i++) {
238         tree1.insert(values[i]);
239     }
240
241     // create a copy of tree1
242     BST<int> tree2(tree1);
243
244     // "print" both to strings, see if they are equal
245     std::ostringstream oss1, oss2;
246
247     oss1 << tree1;
248     oss2 << tree2;
249
250     CHECK(oss1.str() == oss2.str());
251 }
252
253 /*
254  * Public clear function implementation.
255  */
256 template <class T>
257 void BST<T>::clear() {
258     clear(pRoot);
259     n = 0u;
260     pRoot = 0;
261 }
262
263 // doctest unit test for clear method
264 TEST_CASE("testing␣BST<T>::clear()") {
265     BST<int> tree1;
266     int values[10] = {80, 9, 3, 34, 33, 63, 81, 55, 86, 9};
267     for(int i = 0; i < 10; i++) {
268         tree1.insert(values[i]);
269     }
270
271     tree1.clear();
272
273     // did the clear empty the tree?
274     CHECK(0u == tree1.size());
275
276     // are there 0 elements in the print?
277     std::ostringstream oss;
278     oss << tree1;
279
280     CHECK(oss.str() == "[]");
281 }
282
283 /*
284  * Private recursive clear function implementation.
285  */
286 template <class T>
287 void BST<T>::clear(Node *pCurr) {
288     if (pCurr != 0) {
289         // post-order traversal is needed here - delete left and right
290         // subtrees first, then the current node
291         clear(pCurr->pLeft);
292         clear(pCurr->pRight);
293         delete pCurr;
294     }
295 }
296
297 /*
298  * Private recursive contains function implementation.
299  */
300 template <class T>
301 bool BST<T>::contains(const T &val, Node *pCurr) const {
302     if(pCurr == 0) {
```

```
303         // empty tree does not contain val
304         return false;
305     } else if(pCurr->data == val) {
306         // found it? return true
307         return true;
308     } else if(val < pCurr->data) {
309         // less than curr? go left
310         return contains(val, pCurr->pLeft);
311     } else {
312         // only other option is to go right
313         return contains(val, pCurr->pRight);
314     }
315 }
316
317 // doctest unit test for contains method
318 TEST_CASE("testing␣BST<T>::contains()") {
319     BST<char> tree;
320     std::string values ="I␣love␣data␣structures!";
321     for(size_t i = 0u; i < values.size(); i++) {
322         tree.insert(values[i]);
323     }
324
325     // verify tree contains all the characters we added
326     for(size_t i = 0u; i < values.size(); i++) {
327         CHECK(tree.contains(values[i]));
328     }
329
330     // verify it does not contain some others
331     CHECK(!tree.contains('Q'));
332     CHECK(!tree.contains('.'));
333     CHECK(!tree.contains('1'));
334
335 }
336
337 /*
338  * Private recursive copy function implementation.
339  */
340 template <class T>
341 typename BST<T>::Node *BST<T>::copy(Node* pOtherCurr) {
342     if(pOtherCurr != 0) {
343         Node *pCurr = new Node(pOtherCurr->data);
344         n++;
345         pCurr->pLeft = copy(pOtherCurr->pLeft);
346         pCurr->pRight = copy(pOtherCurr->pRight);
347
348         return pCurr;
349     } else {
350         return 0;
351     }
352 }
353
354 // copy is private, so it is tested indirectly in copy constructor and
355 // assignment operator tests
356
357 /*
358  * Insert function implementation.
359  */
360 template <class T>
361 bool BST<T>::insert(const T& val) {
362     // set up pointers to current node and its parent.
363     Node *pCurr = pRoot, *pParent = 0;
364
365     // search for the value's location in the tree. This code illustrates
366     // how to search non-recursively, while the contains method uses
367     // recursion. It is easier to keep track of the parent pointer if we use
368     // iteration.
369     while(pCurr != 0) {
370         pParent = pCurr;
371         if(val < pCurr->data) {
372             // search in left subtree
373             pCurr = pCurr->pLeft;
374         } else if(val > pCurr->data) {
375             // search in right subtree
376             pCurr = pCurr->pRight;
377         } else {
378             // val == pCurr->data means we are inserting a duplicate
```

```
379             return false;
380         }
381     } // while
382
383     // now, pCurr is the (empty) spot where the value should be inserted,
384     // and pParent is the parent node. Find out which side we are adding on,
385     // and add the new node. Special case is a previously empty tree.
386     Node *pN = new Node(val);
387     if(pRoot == 0) {
388         pRoot = pN;
389     } else {
390         if(val < pParent->data) {
391             pParent->pLeft = pN;
392         } else {
393             pParent->pRight = pN;
394         }
395     }
396
397     n++;
398     return true;
399 }
400
401 // doctest unit test for the insert method
402 TEST_CASE("testing BST<T>::insert()") {
403     BST<int> tree;
404
405     CHECK(0 == tree.size());
406     CHECK(true == tree.insert(1));
407     CHECK(1 == tree.size());
408     CHECK(tree.contains(1));
409
410     CHECK(true == tree.insert(-2));
411     CHECK(2 == tree.size());
412     CHECK(tree.contains(-2));
413
414     CHECK(false == tree.insert(-2));
415     CHECK(2 == tree.size());
416     CHECK(tree.contains(-2));
417
418     CHECK(true == tree.insert(2));
419     CHECK(3 == tree.size());
420     CHECK(tree.contains(2));
421 }
422
423 // doctest unit test for isEmpty
424 TEST_CASE("testing BST<T>::isEmpty()") {
425     BST<int> tree;
426
427     CHECK(tree.isEmpty());
428     tree.insert(1);
429     CHECK(!tree.isEmpty());
430     tree.clear();
431     CHECK(tree.isEmpty());
432 }
433
434 /*
435  * Assignment operator override.
436  */
437 template <class T>
438 BST<T> &BST<T>::operator=(const BST<T> &tree) {
439     // clear existing data, if any
440     clear();
441
442     // copy the other tree's data
443     pRoot = copy(tree.pRoot);
444
445     return *this;
446 }
447
448 // doctest unit test for assignment operator
449 TEST_CASE("testing BST<T>::operator=()") {
450     BST<int> tree1;
451     int values[10] = {80, 9, 3, 34, 33, 63, 81, 55, 86, 9};
452     for(int i = 0; i < 10; i++) {
453         tree1.insert(values[i]);
454     }
```

```
455
456     // create a copy of tree1
457     BST<int> tree2;
458
459     // add some extra before assignment; these should be removed before the
460     // copy
461     tree2.insert(5);
462     tree2.insert(7);
463     tree2 = tree1;
464
465     // "print" both to strings, see if they are equal
466     std::ostringstream oss1, oss2;
467
468     oss1 << tree1;
469     oss2 << tree2;
470
471     CHECK(oss1.str() == oss2.str());
472 }
473
474 /*
475  * Private print function implementation.
476  */
477 template <class T>
478 void BST<T>::preOrderPrint(std::ostream &out, Node *pCurr) const {
479     // this recursive method prints values using a preorder traversal --
480     // print current value, then print the left subtree, then print the
481     // right subtree
482     if(pCurr != 0) {
483         out << pCurr->data << "␣";
484
485         preOrderPrint(out, pCurr->pLeft);
486         preOrderPrint(out, pCurr->pRight);
487     }
488 }
489
490 // doctest unit test for stream insertion and preOrderPrint
491 TEST_CASE("testing␣BST<T>::operator<<") {
492     BST<char> tree;
493
494     std::string racecar = "racecar";
495     for(size_t i = 0; i < racecar.size(); i++) {
496         tree.insert(racecar[i]);
497     }
498
499     // print to a string and see if it looks right
500     std::ostringstream oss;
501
502     oss << tree;
503
504     CHECK(oss.str() == "[r␣a␣c␣e␣]");
505 }
506
507 /*
508  * Remove function implementation.
509  */
510 template <class T>
511 bool BST<T>::remove(const T& val) {
512     // establish pointers to the current node and its parent
513     Node *pCurr = pRoot, *pParent = 0;
514
515     // loop until we find an empty space (val not in tree), or the node
516     // containing val. Update so that pCurr points to the space or node,
517     // and pParent points to the parent of pCurr.
518     while(pCurr != 0 && pCurr->data != val) {
519         pParent = pCurr;
520         if(val < pCurr->data) {
521             // search left subtree
522             pCurr = pCurr->pLeft;
523         } else {
524             // search right subtree
525             pCurr = pCurr->pRight;
526         }
527     }
528
529     // empty space? we do nothing!
530     if(pCurr == 0) {
```

```
531         return false;
532     }
533 
534     // otherwise, what we do depends on number of children the node has.
535     // If node containing val has two children, we make some modifications
536     // so that the node to delete will have zero or one child.
537     if(pCurr->pLeft != 0 && pCurr->pRight != 0) {
538         // find inorder successor of this node -- smallest value in the right
539         // subtree. Start at the root of right subtree...
540         Node* pInorderSuccessor = pCurr->pRight;
541         Node* pIoSParent = pCurr;
542 
543         // ... then go as far left as possible in the right subtree
544         while(pInorderSuccessor->pLeft != 0) {
545             pIoSParent = pInorderSuccessor;
546             pInorderSuccessor = pInorderSuccessor->pLeft;
547         }
548 
549         // move data from inorder successor to current node
550         pCurr->data = pInorderSuccessor->data;
551 
552         // update pCurr and pParent to refer to the inorder successor node and
553         // its parent. Since we went as far left as possible in this subtree,
554         // pCurr's node will have at most one child, and the remaining code
555         // will remove the node (now containing the value we wanted to remove
556         // in the first place) without doing anything else special.
557         pCurr = pInorderSuccessor;
558         pParent = pIoSParent;
559     }
560 
561     // now pCurr points to a node with zero or one child. pCurr
562     // is the node to delete. First, get a pointer to the node to replace
563     // pCurr's node with. If there are zero children, pReplacement will be 0.
564     Node *pReplacement;
565     if(pCurr->pLeft != 0) {
566         pReplacement = pCurr->pLeft;
567     } else {
568         pReplacement = pCurr->pRight;
569     }
570 
571     // move the replacement node into pCurr's position in the tree
572     if(pCurr == pRoot) {
573         // special case for deleting current root node
574         pRoot = pReplacement;
575     } else {
576         // attach replacement node to proper child of the parent
577         if(pCurr == pParent->pLeft) {
578             pParent->pLeft = pReplacement;
579         } else {
580             pParent->pRight = pReplacement;
581         }
582     }
583 
584     // finally! delete the current node
585     delete pCurr;
586     n--;
587     return true;
588 }
589 
590 // doctest unit test for remove method
591 TEST_CASE("testing␣BST<T>::remove()") {
592     BST<char> tree;
593     std::string data = "datastructures";
594     for(size_t i = 0; i < data.size(); i++) {
595         tree.insert(data[i]);
596     }
597 
598     // check original values
599     std::ostringstream oss;
600     oss << tree;
601     CHECK(oss.str() == "[d␣a␣c␣t␣s␣r␣e␣u␣]");
602 
603     // remove value not in the tree
604     CHECK(false == tree.remove('q'));
605     std::ostringstream oss1;
606     oss1 << tree;
```

```
607     CHECK(oss1.str() == "[d␣a␣c␣t␣s␣r␣e␣u␣]");
608     CHECK(8 == tree.size());
609
610     // remove node with no children
611     CHECK(true == tree.remove('e'));
612     std::ostringstream oss2;
613     oss2 << tree;
614     CHECK(oss2.str() == "[d␣a␣c␣t␣s␣r␣u␣]");
615     CHECK(7 == tree.size());
616
617     //remove node w/ one child
618     CHECK(true == tree.remove('a'));
619     std::ostringstream oss3;
620     oss3 << tree;
621     CHECK(oss3.str() == "[d␣c␣t␣s␣r␣u␣]");
622     CHECK(6 == tree.size());
623
624     // remove node with 2 children
625     CHECK(true == tree.remove('t'));
626     std::ostringstream oss4;
627     oss4 << tree;
628     CHECK(oss4.str() == "[d␣c␣u␣s␣r␣]");
629     CHECK(5 == tree.size());
630
631     // remove root
632     CHECK(true == tree.remove('d'));
633     std::ostringstream oss5;
634     oss5 << tree;
635     CHECK(oss5.str() == "[r␣c␣u␣s␣]");
636     CHECK(4 == tree.size());
637
638     BST<int> tree2;
639
640     // remove from empty tree
641     CHECK(false == tree2.remove(13));
642     std::ostringstream oss6;
643     oss6 << tree2;
644     CHECK(oss6.str() == "[]");
645     CHECK(0 == tree2.size());
646
647     // remove root from tree with only one subtree
648     for(int i = 0; i < 5; i++) {
649         tree2.insert(i);
650     }
651     CHECK(true == tree2.remove(0));
652     std::ostringstream oss7;
653     oss7 << tree2;
654     CHECK(oss7.str() == "[1␣2␣3␣4␣]");
655     CHECK(4 == tree2.size());
656 }
657
658 // doctest unit test for size method
659 TEST_CASE("testing␣BST<T>::size()") {
660     BST<int> tree;
661
662     CHECK(0 == tree.size());
663     tree.insert(1);
664     CHECK(1 == tree.size());
665     tree.insert(2);
666     CHECK(2 == tree.size());
667     tree.clear();
668     CHECK(0 == tree.size());
669 }
```

Listing 2: SpellCheck.cpp

```
1  #include <algorithm>
2  #include <cctype>
3  #include <cstdlib>
4  #include <fstream>
5  #include <iostream>
6  #include <string>
7  #include "../1-BST/BST.hpp"
8
9  /**
10  * @brief See if a character is alphabetic.
```

```
 *
 * Checks to see if a character is alphabetic.
 *
 * @param c Character to check.
 *
 * @return true if the character meets the stated requirements, false otherwise.
 */
bool isAlpha(char c) {
    return (c >= 'a' && c <='z');
}

/**
 * @brief Convert a string to all lowercase.
 *
 * @param word std::string to convert.
 *
 * @return all-lowercase version of the string.
 */
std::string toLower(std::string word) {
    std::transform(word.begin(), word.end(), word.begin(),
        [](unsigned char c){ return std::tolower(c); });
    return word;
}

/**
 * @brief Remove all non-alphabetic characters from a string.
 *
 * @param word std::string to conver.
 *
 * @return word, but with all non-alphabetic characters removed.
 */
std::string stripNonAlpha(std::string word) {
    std::string newString = "";
    for(size_t i = 0; i < word.size(); i++) {
        if(isAlpha(word[i])) {
            newString += word[i];
        }
    }
    return newString;
}

/**
 * Application entry point.
 */
int main() {
    // read dictionary words into a BST
    BST<std::string> dictionary;
    std::ifstream dictionaryFile("dictionary.txt");
    std::string word;
    while(dictionaryFile >> word) {
        dictionary.insert(word);
    }
    dictionaryFile.close();

    // read file to check from standard input
    while(std::cin >> word) {
        word = stripNonAlpha(toLower(word));
        // dump all words not in the dictionary to standard output
        if(word != "" && !dictionary.contains(word)) {
            std::cout << word << std::endl;
        }
    }

    return EXIT_SUCCESS;
}
```