Project 08

Graded

22 Hours, 56 Minutes Late

Student
Ishan Mukherjee

Total Points
100 / 100 pts

Autograder Score
80.0 / 80.0

Passed Tests
Test 1
Test 2
Test 3
Test 4
Test 5
Test 6

Question 2

Manual review **20** / 20 pts

Additional test cases...

✔ **+ 20 pts** Excellent -- 10+ additional test cases involving both toPairs and foreach / iterators, stress test

+ 16 pts Good test cases, but missing e.g. stress test

+ 12 pts Some additional test cases, but we asked for 10 additional good test cases

+ 6 pts 1-2 additional tests cases, but very little

+ 0 pts No additional test cases beyond what was given

+ 0 pts WARNING: additional field(s) added to NODE class

+ 0 pts WARNING: additional fields added to set class

+ 0 pts WARNING: toPairs( ) not using thread

+ 0 pts WARNING: iterator class is not Ptr-based

Autograder Results

Autograder Output

```
**********************************************************
This is submission #7
Submitted @ 22:55 on 2024-3-9 (Chicago time)

Submission history:
 Submission #6: score=80, submitted @ 22:15 on 2024-3-9 (Chicago time)
 Submission #5: score=0, submitted @ 21:35 on 2024-3-9 (Chicago time)
 Submission #4: score=-1, submitted @ 18:18 on 2024-3-9 (Chicago time)
 Submission #3: score=0, submitted @ 23:59 on 2024-3-8 (Chicago time)
 Submission #2: score=-1, submitted @ 23:39 on 2024-3-8 (Chicago time)
 Submission #1: score=-1, submitted @ 23:38 on 2024-3-8 (Chicago time)

Total # of valid submissions so far: 3
# of valid submissions since midnight: 2
# of minutes since last valid submission: 40
**********************************************************
You have 2 submissions this 24-hr period.


**********************************************************************
** Number of Submissions This Time Period                    **
**********************************************************************

This is submission #3 in current time period

You are allowed a total of 4 submissions per 24-hr time period.

**********************************************************************


**********************************************************************
** Test Number: 1 **


**********************************************************************
** TEST CASE PASSED!                                        **
**********************************************************************


** Test output (first 100 lines) **
[==========] Running 13 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 13 tests from myset
[ RUN      ] myset.empty_set
[       OK ] myset.empty_set (0 ms)
[ RUN      ] myset.set_with_one
[       OK ] myset.set_with_one (0 ms)
[ RUN      ] myset.set_with_four_strings
```

```
[       OK ] myset.set_with_four_strings (0 ms)
[ RUN      ] myset.set_with_movies
[       OK ] myset.set_with_movies (0 ms)
[ RUN      ] myset.set_from_class_with_nine
[       OK ] myset.set_from_class_with_nine (0 ms)
[ RUN      ] myset.set_no_duplicates
[       OK ] myset.set_no_duplicates (0 ms)
[ RUN      ] myset.toVector
[       OK ] myset.toVector (0 ms)
[ RUN      ] myset.copy_empty
[       OK ] myset.copy_empty (0 ms)
[ RUN      ] myset.copy_constructor
[       OK ] myset.copy_constructor (0 ms)
[ RUN      ] myset.find_empty
[       OK ] myset.find_empty (0 ms)
[ RUN      ] myset.find_one
[       OK ] myset.find_one (0 ms)
[ RUN      ] myset.find_with_set_from_class
[       OK ] myset.find_with_set_from_class (0 ms)
[ RUN      ] myset.stress_test
[       OK ] myset.stress_test (2541 ms)
[----------] 13 tests from myset (2541 ms total)

[----------] Global test environment tear-down
[==========] 13 tests from 1 test suite ran. (2541 ms total)
[  PASSED  ] 13 tests.


** End of Test 1 **
**********************************************************************


**********************************************************************
** Test Number: 2 **


**********************************************************************
** TEST CASE PASSED!                                  **
**********************************************************************


** Test output (first 100 lines) **
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from myset
[ RUN      ] myset.to_pairs_empty
[       OK ] myset.to_pairs_empty (0 ms)
[ RUN      ] myset.to_pairs_one
[       OK ] myset.to_pairs_one (0 ms)
```

```
[ RUN      ] myset.to_pairs_two
[       OK ] myset.to_pairs_two (0 ms)
[ RUN      ] myset.to_pairs_four
[       OK ] myset.to_pairs_four (0 ms)
[ RUN      ] myset.to_pairs_with_set_from_class
[       OK ] myset.to_pairs_with_set_from_class (0 ms)
[----------] 5 tests from myset (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 5 tests.


** End of Test 2 **
************************************************************************


************************************************************************
** Test Number: 3 **


************************************************************************
** TEST CASE PASSED!                                  **
************************************************************************


** Test output (first 100 lines) **
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from myset
[ RUN      ] myset.to_pairs_with_edge_case_to_right
[       OK ] myset.to_pairs_with_edge_case_to_right (0 ms)
[ RUN      ] myset.to_pairs_with_edge_case_to_left
[       OK ] myset.to_pairs_with_edge_case_to_left (0 ms)
[ RUN      ] myset.stress_to_pairs
[       OK ] myset.stress_to_pairs (135 ms)
[----------] 3 tests from myset (135 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (135 ms total)
[  PASSED  ] 3 tests.


** End of Test 3 **
************************************************************************


************************************************************************
** Test Number: 4 **
```

```
**********************************************************************
** TEST CASE PASSED!                                  **
**********************************************************************


** Test output (first 100 lines) **
[==========] Running 3 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 3 tests from myset
[ RUN      ] myset.foreach_empty
[       OK ] myset.foreach_empty (0 ms)
[ RUN      ] myset.foreach_one_element
[       OK ] myset.foreach_one_element (0 ms)
[ RUN      ] myset.foreach_set_from_class
[       OK ] myset.foreach_set_from_class (0 ms)
[----------] 3 tests from myset (0 ms total)

[----------] Global test environment tear-down
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.


** End of Test 4 **
**********************************************************************


**********************************************************************
** Test Number: 5 **


**********************************************************************
** TEST CASE PASSED!                                  **
**********************************************************************


** Test output (first 100 lines) **
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from myset
[ RUN      ] myset.stress_iterator
[       OK ] myset.stress_iterator (2317 ms)
[----------] 1 test from myset (2317 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (2317 ms total)
[  PASSED  ] 1 test.
```

```
** End of Test 5 **
************************************************************************


************************************************************************
** Test Number: 6 **


************************************************************************
** VALGRIND TEST for errors AND memory leaks...                **
************************************************************************


************************************************************************
** TEST CASE PASSED, with no memory errors or leaks!           **
************************************************************************


** Test output (first 100 lines) **
[==========] Running 21 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 21 tests from myset
[ RUN      ] myset.empty_set
[       OK ] myset.empty_set (0 ms)
[ RUN      ] myset.set_with_one
[       OK ] myset.set_with_one (0 ms)
[ RUN      ] myset.set_with_four_strings
[       OK ] myset.set_with_four_strings (0 ms)
[ RUN      ] myset.set_with_movies
[       OK ] myset.set_with_movies (0 ms)
[ RUN      ] myset.set_from_class_with_nine
[       OK ] myset.set_from_class_with_nine (0 ms)
[ RUN      ] myset.set_no_duplicates
[       OK ] myset.set_no_duplicates (0 ms)
[ RUN      ] myset.toVector
[       OK ] myset.toVector (0 ms)
[ RUN      ] myset.copy_empty
[       OK ] myset.copy_empty (0 ms)
[ RUN      ] myset.copy_constructor
[       OK ] myset.copy_constructor (0 ms)
[ RUN      ] myset.find_empty
[       OK ] myset.find_empty (0 ms)
[ RUN      ] myset.find_one
[       OK ] myset.find_one (0 ms)
[ RUN      ] myset.find_with_set_from_class
[       OK ] myset.find_with_set_from_class (0 ms)
[ RUN      ] myset.stress_test
[       OK ] myset.stress_test (2527 ms)
```

```
[ RUN      ] myset.to_pairs_empty
[       OK ] myset.to_pairs_empty (0 ms)
[ RUN      ] myset.to_pairs_one
[       OK ] myset.to_pairs_one (0 ms)
[ RUN      ] myset.to_pairs_two
[       OK ] myset.to_pairs_two (0 ms)
[ RUN      ] myset.to_pairs_four
[       OK ] myset.to_pairs_four (0 ms)
[ RUN      ] myset.to_pairs_with_set_from_class
[       OK ] myset.to_pairs_with_set_from_class (0 ms)
[ RUN      ] myset.foreach_empty
[       OK ] myset.foreach_empty (0 ms)
[ RUN      ] myset.foreach_one_element
[       OK ] myset.foreach_one_element (0 ms)
[ RUN      ] myset.foreach_set_from_class
[       OK ] myset.foreach_set_from_class (0 ms)
[----------] 21 tests from myset (2527 ms total)

[----------] Global test environment tear-down
[==========] 21 tests from 1 test suite ran. (2527 ms total)
[  PASSED  ] 21 tests.


** End of Test 6 **
**********************************************************************
```

Excellent, perfect score!

Test 1

Test 1: test01 (13 tests that were given) -- yay, output correct!

Test 2

Test 2: test02 (toPairs) -- yay, output correct!

Test 3

Test 3: test03 (toPairs edge cases, stress) -- yay, output correct!

Test 4

Test 4: test04 (foreach) -- yay, output correct!

Test 5

Test 5: test05 (foreach stress) -- yay, output correct!

Test 6

Test 6: test06 (valgrind check of non-stress tests) -- yay, output correct!

Submitted Files

set.h

Download

```
/*set.h*/

//
// Implements a set in the mathematical sense, with no duplicates.
//
// Ishan Mukherjee
//
// Original template: Prof. Joe Hummel
// Northwestern University
// CS 211
//

//
// NOTE: because our set has the same name as std::set
// in the C++ standard template library (STL), we cannot
// do the following:
//
//   using namespace std;
//
// This implies refernces to the STL will need to use
// the "std::" prefix, e.g. std::cout, std::endl, and
// std::vector.
//
#pragma once

#include <iostream>
#include <vector>
#include <utility>  // std::pair
#include <cassert>


template <typename TKey>
class set
{
public:
  // ###################################################################
  //
  // A node in the search tree:
  //
  class NODE {
  private:
    TKey  Key;
    bool  isThreaded : 1; // 1 bit
    NODE* Left;
    NODE* Right;

```

```
47    public:
48      // constructor:
49      NODE(TKey key)
50        : Key(key), isThreaded(false), Left(nullptr), Right(nullptr)
51      { }
52
53      // getters:
54      TKey  get_Key() { return this->Key; }
55      bool  get_isThreaded() { return this->isThreaded; }
56      NODE* get_Left() { return this->Left; }
57
58      // NOTE: this ignores the thread, call to perform "normal" traversals
59      NODE* get_Right() {
60        if (this->isThreaded)
61          return nullptr;
62        else
63          return this->Right;
64      }
65
66      NODE* get_Thread() {
67        return this->Right;
68      }
69
70      // setters:
71      void set_isThreaded(bool threaded) { this->isThreaded = threaded; }
72      void set_Left(NODE* left) { this->Left = left; }
73      void set_Right(NODE* right) { this->Right = right; }
74    };
75
76
77    // ########################################################################
78    //
79    // set data members:
80    //
81    NODE* Root; // pointer to root node
82    int Size;   // # of nodes in tree
83
84
85    // ########################################################################
86    //
87    // set methods:
88    //
89  public:
90    //
91    // default constructor:
92    //
93    set()
94      : Root(nullptr), Size(0)
95    { }
```

```
96
97    //
98    // copy constructor:
99    //
100 private:
101   void _copy(NODE* other)
102   {
103     if (other == nullptr)
104       return;
105     else {
106       //
107       // we make a copy using insert so that threads
108       // are recreated properly in the copy:
109       //
110       this->insert(other->get_Key());
111
112       _copy(other->get_Left());
113       _copy(other->get_Right());
114     }
115   }
116
117 public:
118   set(const set& other)
119     : Root(nullptr), Size(0)
120   {
121     _copy(other.Root);
122   }
123
124   //
125   // destructor:
126   //
127 private:
128   void _destroy(NODE* cur)
129   {
130     if (cur == nullptr)
131       ;
132     else {
133       _destroy(cur->get_Left());
134       _destroy(cur->get_Right());
135       delete cur;
136     }
137   }
138
139 public:
140   ~set()
141   {
142     //
143     // NOTE: this is commented out UNTIL you are ready. The last
144     // step is to uncomment this and check for memory leaks.
```

```
145     //
146     _destroy(this->Root);
147   }
148
149   //
150   // size
151   //
152   // Returns # of elements in the set
153   //
154   int size()
155   {
156     return this->Size;
157   }
158
159   //
160   // contains
161   //
162   // Returns true if set contains key, false if not
163   //
164 private:
165   bool _contains(NODE* cur, TKey key)
166   {
167     if (cur == nullptr)
168       return false;
169     else {
170
171       if (key < cur->get_Key()) // search left:
172         return _contains(cur->get_Left(), key);
173       else if (cur->get_Key() < key) // search right:
174         return _contains(cur->get_Right(), key);
175       else // must be equal, found it!
176         return true;
177     }
178   }
179
180 public:
181   bool contains(TKey key)
182   {
183     return _contains(this->Root, key);
184   }
185
186   //
187   // insert
188   //
189   // Inserts the given key into the set; if the key is already in
190   // the set then this function has no effect.
191   //
192   void insert(TKey key)
193   {
```

```
194     NODE* prev = nullptr;
195     NODE* cur = this->Root;
196
197     //
198     // 1. Search for key, return if found:
199     //
200     while (cur != nullptr) {
201       if (key < cur->get_Key()) { // left:
202         prev = cur;
203         cur = cur->get_Left();
204       }
205       else if (cur->get_Key() < key) { // right:
206         prev = cur;
207         cur = cur->get_Right();
208       }
209       else { // must be equal => already in tree
210         return;  // don't insert again
211       }
212     }
213
214     //
215     // 2. If not found, insert where we
216     //    fell out of the tree:
217     //
218     NODE* n = new NODE(key);
219
220     if (prev == nullptr) {
221       //
222       // tree is empty, insert at root:
223       //
224       this->Root = n;
225     }
226     else if (key < prev->get_Key()) {
227       //
228       // we are to the left of our parent:
229       //
230       prev->set_Left(n);
231       n->set_isThreaded(true);
232       n->set_Right(prev);
233     }
234     else {
235       //
236       // we are to the right of our parent:
237       //
238       n->set_isThreaded(prev->get_isThreaded());
239       n->set_Right(prev->get_Thread());
240       prev->set_isThreaded(false);
241       prev->set_Right(n);
242     }
```

```
243
244     //
245     // STEP 3: update size and return
246     //
247     this->Size++;
248     return;
249   }
250
251   //
252   // toPairs
253   //
254   // Returns pairs of elements: <element, threaded element>.
255   // If a node is not threaded: <element, no_element value>.
256   //
257 private:
258   void _toPairs(NODE* cur, std::vector<std::pair<TKey, TKey>>& P, TKey no_element) {
259     if (cur == nullptr) {
260       return;
261     } else {
262       //
263       // we want them in order, so go left, then
264       // middle, then right:
265       //
266
267       _toPairs(cur->get_Left(), P, no_element);
268
269       TKey threadValue;
270       if (cur->get_isThreaded()) {
271         threadValue = cur->get_Thread()->get_Key();
272       } else {
273         threadValue = no_element;
274       }
275       P.push_back(std::make_pair(cur->get_Key(), threadValue));
276
277       _toPairs(cur->get_Right(), P, no_element);
278     }
279   }
280 public:
281   std::vector<std::pair<TKey, TKey>> toPairs(TKey no_element)
282   {
283     std::vector<std::pair<TKey, TKey>> P;
284
285     _toPairs(this->Root, P, no_element);
286
287     return P;
288   }
289
290   //
291   // []
```

```
292   //
293   // Returns true if set contains key, false if not.
294   //
295   bool operator[](TKey key)
296   {
297     return this->contains(key);
298   }
299
300   //
301   // toVector
302   //
303   // Returns the elements of the set, in order,
304   // in a vector.
305   //
306 private:
307   void _toVector(NODE* cur, std::vector<TKey>& V) {
308     if (cur == nullptr)
309       return;
310     else {
311       //
312       // we want them in order, so go left, then
313       // middle, then right:
314       //
315       _toVector(cur->get_Left(), V);
316       V.push_back(cur->get_Key());
317       _toVector(cur->get_Right(), V);
318     }
319   }
320
321 public:
322   std::vector<TKey> toVector()
323   {
324     std::vector<TKey> V;
325
326     _toVector(this->Root, V);
327
328     return V;
329   }
330
331
332   // ##################################################################
333   //
334   // class iterator:
335   //
336 private:
337   class iterator
338   {
339   private:
340     NODE* Ptr;
```

```
341
342   public:
343
344     iterator(NODE* ptr) : Ptr(ptr) {}
345
346     //
347     // !=
348     //
349     // Returns true if the given iterator is not equal to
350     // this iterator.
351     //
352     bool operator!=(iterator other) {
353       return this->Ptr != other.Ptr;
354     }
355
356     //
357     // ++
358     //
359     // Advances the iterator to the next ordered element of
360     // the set; if the iterator cannot be advanced, ++ has
361     // no effect.
362     //
363     void operator++() {
364       if (Ptr->get_isThreaded()) {
365         Ptr = Ptr->get_Thread();
366       } else {
367         Ptr = leftmost(Ptr->get_Right());
368       }
369       return;
370     }
371
372     //
373     // *
374     //
375     // Returns the key denoted by the iterator; this
376     // code will throw an out_of_range exception if
377     // the iterator does not denote an element of the
378     // set.
379     //
380     TKey operator*()
381     {
382       if (this->Ptr == nullptr)
383         throw std::out_of_range("set::iterator:operator*");
384
385       return this->Ptr->get_Key();
386     }
387
388     //
389     // ==
```

```
390       //
391       // Returns true if the given iterator is equal to
392       // this iterator.
393       //
394       bool operator==(iterator other)
395       {
396         if (this->Ptr == other.Ptr)
397           return true;
398         else
399           return false;
400       }
401     };
402
403   public:
404     // helper function to return leftmost child of subtree
405     static NODE* leftmost(NODE* root) {
406       NODE* current = root;
407       while (current && current->get_Left() != nullptr) {
408         current = current->get_Left();
409       }
410       return current;
411     }
412
413     iterator begin() const {
414       return iterator(leftmost(this->Root));
415     }
416
417     // ##################################################################
418     //
419     // find:
420     //
421     // If the set contains key, then an iterator denoting this
422     // element is returned. If the set does not contain key,
423     // then set.end() is returned.
424     //
425   public:
426     iterator find(TKey key)
427     {
428       NODE* cur = this->Root;
429
430       while (cur != nullptr) {
431         if (key < cur->get_Key()) { // search left:
432           cur = cur->get_Left();
433         }
434         else if (cur->get_Key() < key) { // search right:
435           cur = cur->get_Right();
436         }
437         else { // must be equal, found it!
438           return iterator(cur);
```

```
439       }
440     }
441
442     // if get here, not found
443     return iterator(nullptr);
444   }
445
446   //
447   // end:
448   //
449   // Returns an iterator to the end of the iteration space,
450   // i.e. to no element. In other words, if your iterator
451   // == set.end(), then you are not pointing to an element.
452   //
453   iterator end()
454   {
455     return iterator(nullptr);
456   }
457
458 };
459
```

tests.cpp

Download

```cpp
/*tests.c*/

//
// Google test cases for our set class.
//
// Initial template: Prof. Joe Hummel
// Northwestern University
// CS 211
//

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <random>
#include <set>  // for comparing answers

using std::string;
using std::vector;
using std::pair;

#include "set.h"
#include "gtest/gtest.h"

// my tests

TEST(myset, toPairs_empty_set)
{
  set<int> S;
  auto pairs = S.toPairs(-1);
  ASSERT_TRUE(pairs.empty());
}

TEST(myset, toPairs_set_with_one_element)
{
  set<int> S;
  S.insert(123);

  auto pairs = S.toPairs(-1);

  vector<pair<int, int>> expected = { {123, -1} };

  ASSERT_EQ(pairs.size(), (unsigned long) 1);
  ASSERT_EQ(pairs, expected);
}

```

```
47
48  TEST(myset, toPairs_with_multiple_elements)
49  {
50    set<int> S;
51
52    S.insert(30);
53    S.insert(15);
54    S.insert(50);
55    S.insert(8);
56    S.insert(25);
57    S.insert(70);
58    S.insert(20);
59    S.insert(28);
60    S.insert(60);
61
62    ASSERT_EQ(S.size(), 9);
63
64    auto pairs = S.toPairs(-1);
65
66    vector<pair<int, int>> expected = {
67      {8, 15}, {15, -1}, {20, 25}, {25, -1}, {28, 30},
68      {30, -1}, {50, -1}, {60, 70}, {70, -1}
69    };
70
71    ASSERT_EQ(pairs.size(), expected.size());
72
73    for (size_t i = 0, n = expected.size(); i < n; ++i)
74    {
75      ASSERT_EQ(pairs[i], expected[i]);
76    }
77  }
78
79  TEST(myset, toPairs_single_string_element)
80  {
81    set<string> S;
82
83    S.insert("Angola");
84
85    ASSERT_EQ(S.size(), 1);
86
87    auto pairs = S.toPairs("FIN");
88
89    vector<pair<string, string>> expected = { {"Angola", "FIN"} };
90
91    ASSERT_EQ(pairs.size(), expected.size());
92    ASSERT_EQ(pairs[0], expected[0]);
93  }
94
95  TEST(myset, toPairs_multiple_string_elements)
```

```
{
  set<string> S;

  S.insert("Congo");
  S.insert("Brazil");
  S.insert("Djibouti");
  S.insert("Angola");
  S.insert("Eritrea");

  ASSERT_EQ(S.size(), 5);

  auto pairs = S.toPairs(" ");

  vector<pair<string, string>> expected = {
    {"Angola", "Brazil"},
    {"Brazil", "Congo"},
    {"Congo", " "},
    {"Djibouti", " "},
    {"Eritrea", " "}
  };

  ASSERT_EQ(pairs.size(), expected.size());

  for (size_t i = 0; i < expected.size(); ++i)
  {
    ASSERT_EQ(pairs[i], expected[i]);
  }
}


TEST(myset, foreach_with_empty_set)
{
  set<long> S;
  int count = 0;

  for (long x : S)
  {
    count++;
  }

  ASSERT_EQ(count, 0);
}

TEST(myset, foreach_with_single_element_set)
{
  set<int> S;
  S.insert(42);
  int count = 0;
  int val = 0;
```

```
145
146   for (int x : S)
147   {
148     val = x;
149     count++;
150   }
151
152   ASSERT_EQ(count, 1);
153   ASSERT_EQ(val, 42);
154 }
155
156 TEST(myset, foreach_with_increasing_multiple_elements_set)
157 {
158   set<int> S;
159   S.insert(10);
160   S.insert(20);
161   S.insert(30);
162
163   std::vector<int> elements;
164
165   for (int x : S)
166   {
167     elements.push_back(x);
168   }
169
170   std::vector<int> expected = {10, 20, 30};
171
172   ASSERT_EQ(elements.size(), expected.size());
173
174   for (size_t i = 0; i < expected.size(); ++i)
175   {
176     ASSERT_EQ(elements[i], expected[i]);
177   }
178 }
179
180 TEST(myset, foreach_with_unordered_multiple_elements_set)
181 {
182   set<int> S;
183
184   std::vector<int> expected = {33, 21, 19, 35, 21};
185   std::vector<int> sorted_expected = expected;
186   // removing duplicates
187   std::set<int> s;
188   unsigned size = sorted_expected.size();
189   for( unsigned i = 0; i < size; ++i ) s.insert( sorted_expected[i] );
190   sorted_expected.assign( s.begin(), s.end() );
191
192   for (int i : expected) {
193     S.insert(i);
```

```
194   }
195
196   std::vector<int> elements;
197
198   for (int x : S)
199   {
200     elements.push_back(x);
201   }
202
203   ASSERT_EQ(elements.size(), sorted_expected.size());
204
205   for (size_t i = 0; i < sorted_expected.size(); ++i)
206   {
207     ASSERT_EQ(elements[i], sorted_expected[i]);
208   }
209 }
210
211 TEST(myset, foreach_with_single_string_element)
212 {
213   set<string> S;
214   S.insert("Hello");
215   int count = 0;
216   string value;
217
218   for (const string& x : S)
219   {
220     value = x;
221     count++;
222   }
223
224   ASSERT_EQ(count, 1);
225   ASSERT_EQ(value, "Hello");
226 }
227
228 TEST(myset, foreach_with_multiple_string_elements)
229 {
230   set<string> S;
231
232   S.insert("apple");
233   S.insert("banana");
234   S.insert("cherry");
235
236   std::vector<string> elements;
237
238   for (const string& x : S)
239   {
240     elements.push_back(x);
241   }
242
```

```
  std::vector<string> expected = {"apple", "banana", "cherry"};

  ASSERT_EQ(elements.size(), expected.size());

  for (size_t i = 0; i < expected.size(); ++i)
  {
    ASSERT_EQ(elements[i], expected[i]);
  }
}

TEST(myset, foreach_stress_test_with_strictly_increasing_nums)
{
  set<int> S;
  const int N_ELEMS = 1000000;
  for (int i = 0; i < N_ELEMS; ++i) {
    S.insert(i);
  }

  // check set size
  ASSERT_EQ(S.size(), N_ELEMS);

  // check values
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, count);
    count++;
  }

  ASSERT_EQ(count, N_ELEMS);
}

TEST(myset, foreach_stress_test_with_random_nums)
{
  set<int> S;
  const int N_ELEMS = 1000000;

  // populate a vector with random numbers
  // code below is from https://stackoverflow.com/a/23143753
  std::random_device rnd_device;
  std::mt19937 mersenne_engine {rnd_device()};
  std::uniform_int_distribution<int> dist {1, 52};
  auto gen = [&dist, &mersenne_engine](){
    return dist(mersenne_engine);
  };
  vector<int> vec(N_ELEMS);
  generate(begin(vec), end(vec), gen);

  // popualate set
  for (int i = 0; i < N_ELEMS; ++i) {
```

```
292     S.insert(vec[i]);
293   }
294
295   // create sorted vector
296   std::vector<int> sorted_expected = vec;
297   // removing duplicates
298   std::set<int> s;
299   unsigned size = sorted_expected.size();
300   for( unsigned i = 0; i < size; ++i ) s.insert( sorted_expected[i] );
301   sorted_expected.assign( s.begin(), s.end() );
302
303   // check set size
304   ASSERT_EQ(S.size(), sorted_expected.size());
305
306   // check values
307   int count = 0;
308   for (int x : S) {
309     ASSERT_EQ(x, sorted_expected[count]);
310     count++;
311   }
312 }
313
314 // intructor's tests
315
316 TEST(myset, empty_set)
317 {
318   set<int> S;
319
320   ASSERT_EQ(S.size(), 0);
321 }
322
323 TEST(myset, set_with_one)
324 {
325   set<int> S;
326
327   ASSERT_EQ(S.size(), 0);
328
329   S.insert(123);
330
331   ASSERT_EQ(S.size(), 1);
332
333   ASSERT_TRUE(S.contains(123));
334   ASSERT_TRUE(S[123]);
335
336   ASSERT_FALSE(S.contains(100));
337   ASSERT_FALSE(S[100]);
338   ASSERT_FALSE(S.contains(200));
339   ASSERT_FALSE(S[200]);
340 }
```

```
341
342 TEST(myset, set_with_four_strings)
343 {
344   set<string> S;
345
346   ASSERT_EQ(S.size(), 0);
347
348   S.insert("banana");
349   S.insert("apple");
350   S.insert("chocolate");
351   S.insert("pear");
352
353   ASSERT_EQ(S.size(), 4);
354
355   ASSERT_TRUE(S.contains("pear"));
356   ASSERT_TRUE(S["banana"]);
357   ASSERT_TRUE(S.contains("chocolate"));
358   ASSERT_TRUE(S["apple"]);
359
360   ASSERT_FALSE(S.contains("Apple"));
361   ASSERT_FALSE(S["carmel"]);
362   ASSERT_FALSE(S.contains("appl"));
363   ASSERT_FALSE(S["chocolatee"]);
364 }
365
366 class Movie
367 {
368 public:
369   string Title;
370   int    ID;
371   double Revenue;
372
373   Movie(string title, int id, double revenue)
374     : Title(title), ID(id), Revenue(revenue)
375   { }
376
377   bool operator<(const Movie& other)
378   {
379     if (this->Title < other.Title)
380       return true;
381     else
382       return false;
383   }
384 };
385
386 TEST(myset, set_with_movies)
387 {
388   set<Movie> S;
389
```

```
390   ASSERT_EQ(S.size(), 0);
391
392   Movie Sleepless("Sleepless in Seattle", 123, 123456789.00);
393   S.insert(Sleepless);
394
395   Movie Matrix("The Matrix", 456, 400000000.00);
396   S.insert(Matrix);
397
398   Movie AnimalHouse("Animal House", 789, 1000000000.00);
399   S.insert(AnimalHouse);
400
401   ASSERT_EQ(S.size(), 3);
402
403   vector<Movie> V = S.toVector();
404
405   ASSERT_EQ(V[0].Title, "Animal House");
406   ASSERT_EQ(V[1].Title, "Sleepless in Seattle");
407   ASSERT_EQ(V[2].Title, "The Matrix");
408 }
409
410 TEST(myset, set_from_class_with_nine)
411 {
412   set<int> S;
413
414   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
415
416   for (auto x : V)
417     S.insert(x);
418
419   ASSERT_EQ(S.size(), (int) V.size());
420
421   for (auto x : V) {
422     ASSERT_TRUE(S.contains(x));
423     ASSERT_TRUE(S[x]);
424   }
425
426   ASSERT_FALSE(S.contains(0));
427   ASSERT_FALSE(S[0]);
428   ASSERT_FALSE(S.contains(2));
429   ASSERT_FALSE(S[2]);
430   ASSERT_FALSE(S.contains(4));
431   ASSERT_FALSE(S[4]);
432   ASSERT_FALSE(S.contains(29));
433   ASSERT_FALSE(S[31]);
434   ASSERT_FALSE(S.contains(40));
435   ASSERT_FALSE(S[42]);
436 }
437
438 TEST(myset, set_no_duplicates)
```

```
439 {
440   set<int> S;
441
442   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
443
444   for (auto x : V)
445     S.insert(x);
446
447   // try to insert them all again:
448   for (auto x : V)
449     S.insert(x);
450
451   ASSERT_EQ(S.size(), (int) V.size());
452
453   for (auto x : V) {
454     ASSERT_TRUE(S.contains(x));
455     ASSERT_TRUE(S[x]);
456   }
457 }
458
459 TEST(myset, toVector)
460 {
461   set<int> S;
462
463   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
464
465   for (auto x : V)
466     S.insert(x);
467
468   ASSERT_EQ(S.size(), (int) V.size());
469
470   vector<int> V2 = S.toVector();
471
472   ASSERT_EQ(V2.size(), V.size());
473
474   std::sort(V.begin(), V.end());
475
476   //
477   // V and V2 should have the same elements in
478   // the same order:
479   //
480   auto iterV = V.begin();
481   auto iterV2 = V2.begin();
482
483   while (iterV != V.end()) {
484     ASSERT_EQ(*iterV, *iterV2);
485
486     iterV++;
487     iterV2++;
```

```
488   }
489 }
490
491 TEST(myset, copy_empty)
492 {
493   set<int> S1;
494
495   {
496     //
497     // create a new scope, which will trigger destructor:
498     //
499     set<int> S2 = S1;  // this will call copy constructor:
500
501     S1.insert(123);  // this should have no impact on S2:
502     S1.insert(100);
503     S1.insert(150);
504
505     ASSERT_EQ(S2.size(), 0);
506
507     vector<int> V2 = S2.toVector();
508
509     ASSERT_EQ((int) V2.size(), 0);
510   }
511 }
512
513 TEST(myset, copy_constructor)
514 {
515   set<int> S1;
516
517   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
518
519   for (auto x : V)
520     S1.insert(x);
521
522   ASSERT_EQ(S1.size(), (int) V.size());
523
524   {
525     //
526     // create a new scope, which will trigger destructor:
527     //
528     set<int> S2 = S1;  // this will call copy constructor:
529
530     S1.insert(123);  // this should have no impact on S2:
531     S1.insert(100);
532     S1.insert(150);
533
534     ASSERT_EQ(S2.size(), (int) V.size());
535
536     vector<int> V2 = S2.toVector();
```

```
537
538     ASSERT_EQ(V2.size(), V.size());
539
540     std::sort(V.begin(), V.end());
541
542     //
543     // V and V2 should have the same elements in
544     // the same order:
545     //
546     auto iterV = V.begin();
547     auto iterV2 = V2.begin();
548
549     while (iterV != V.end()) {
550       ASSERT_EQ(*iterV, *iterV2);
551
552       iterV++;
553       iterV2++;
554     }
555
556     S2.insert(1000);  // this should have no impact on S1:
557     S2.insert(2000);
558     S2.insert(3000);
559     S2.insert(4000);
560     S2.insert(5000);
561
562     V.push_back(123);
563     V.push_back(100);
564     V.push_back(150);
565   }
566
567   //
568   // the copy was just destroyed, the original set
569   // should still be the same as it was earlier:
570   //
571   ASSERT_EQ(S1.size(), (int) V.size());
572
573   vector<int> V2 = S1.toVector();
574
575   ASSERT_EQ(V2.size(), V.size());
576
577   std::sort(V.begin(), V.end());
578
579   //
580   // V and V2 should have the same elements in
581   // the same order:
582   //
583   auto iterV = V.begin();
584   auto iterV2 = V2.begin();
585
```

```
586   while (iterV != V.end()) {
587     ASSERT_EQ(*iterV, *iterV2);
588
589     iterV++;
590     iterV2++;
591   }
592 }
593
594 TEST(myset, find_empty)
595 {
596   set<int> S;
597
598   auto iter = S.find(22);
599   ASSERT_TRUE(iter == S.end());
600 }
601
602 TEST(myset, find_one)
603 {
604   set<int> S;
605
606   S.insert(1234);
607
608   auto iter = S.find(123);
609   ASSERT_TRUE(iter == S.end());
610
611   iter = S.find(1234);
612   if (iter == S.end()) {  // this should not happen:
613     ASSERT_TRUE(false); // fail:
614   }
615
616   ASSERT_EQ(*iter, 1234);
617
618   iter = S.find(1235);
619   ASSERT_TRUE(iter == S.end());
620 }
621
622 TEST(myset, find_with_set_from_class)
623 {
624   set<int> S;
625
626   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
627
628   for (auto x : V)
629     S.insert(x);
630
631   ASSERT_EQ(S.size(), (int) V.size());
632
633   //
634   // make sure we can find each of the values we inserted:
```

```
635   //
636   for (auto x : V) {
637     auto iter = S.find(x);
638
639     if (iter == S.end()) {  // this should not happen:
640       ASSERT_TRUE(false); // fail:
641     }
642
643     ASSERT_EQ(*iter, x);
644   }
645
646   //
647   // these searches should all fail:
648   //
649   auto iter = S.find(0);
650   ASSERT_TRUE(iter == S.end());
651
652   iter = S.find(-1);
653   ASSERT_TRUE(iter == S.end());
654
655   iter = S.find(1);
656   ASSERT_TRUE(iter == S.end());
657
658   iter = S.find(4);
659   ASSERT_TRUE(iter == S.end());
660
661   iter = S.find(34);
662   ASSERT_TRUE(iter == S.end());
663
664   iter = S.find(36);
665   ASSERT_TRUE(iter == S.end());
666 }
667
668 TEST(myset, stress_test)
669 {
670   set<long long> S;
671   std::set<long long> C;
672
673   long long N = 1000000;
674
675   //
676   // setup random number generator so tree will
677   // be relatively balanced given insertion of
678   // random numbers:
679   //
680   std::random_device rd;
681   std::mt19937 gen(rd());
682   std::uniform_int_distribution<long long> distrib(1, N * 100);  // inclusive
683
```

```
  vector<long long> V;  // collect a few values for searching:
  int count = 0;

  while (S.size() != N) {

    long long x = distrib(gen);

    S.insert(x);
    C.insert(x);

    count++;
    if (count == 1000) { // save every 1,000th value:

      V.push_back(x);
      count = 0;
    }
  }

  ASSERT_EQ(S.size(), N);

  for (auto x : V) {
    ASSERT_TRUE(S.contains(x));
  }

  ASSERT_FALSE(S.contains(0));
  ASSERT_FALSE(S.contains(-1));

  //
  // now let's compare our set to C++ set:
  //
  V.clear();
  V = S.toVector();

  ASSERT_EQ(V.size(), C.size());
  ASSERT_EQ(S.size(), (int) C.size());

  int i = 0;

  for (auto x : C) {
    ASSERT_EQ(V[i], x);
    i++;
  }
}
```