Project 08 - extra credit

● Graded

Student

Ishan Mukherjee

Total Points

10 / 50 pts

Autograder Score

10.0 / 50.0

Failed Tests

Test 2

Test 3

Passed Tests

Test 1

Autograder Results

Autograder Output

```
************************************************************
This is submission #1
Submitted @ 23:38 on 2024-3-16 (Chicago time)

Submission history:

Total # of valid submissions so far: 0
# of valid submissions since midnight: 0
# of minutes since last valid submission: -1
************************************************************
You have 0 submissions this 24-hr period.


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

This is submission #1 in current time period

You are allowed a total of 6 submissions per 24-hr time period.

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


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


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


** Test output (first 100 lines) **
[==========] Running 25 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 25 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_with_more_movies_and_dups
[       OK ] myset.set_with_more_movies_and_dups (0 ms)
[ RUN      ] myset.set_from_class_with_nine
```

```
[       OK ] myset.set_from_class_with_nine (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.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)
[ 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.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_test
[       OK ] myset.stress_test (10912 ms)
[ RUN      ] myset.stress_iterator
[       OK ] myset.stress_iterator (14189 ms)
[ RUN      ] myset.stress_to_pairs
[       OK ] myset.stress_to_pairs (1540 ms)
[----------] 25 tests from myset (26642 ms total)

[----------] Global test environment tear-down
[==========] 25 tests from 1 test suite ran. (26642 ms total)
[  PASSED  ] 25 tests.


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

```
./run_tests_no_vg: line 123:    74 Segmentation fault      (core dumped) timelimit -p -t "${TIMEOUT}" -T "${TIMI
**********************************************************************
** Test Number: 2 **


****************************************
** test case failed / crashed...        **
****************************************


** Test output (first 100 lines) **
[==========] Running 43 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 43 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_with_more_movies_and_dups
[       OK ] myset.set_with_more_movies_and_dups (0 ms)
[ RUN      ] myset.set_from_class_with_nine
[       OK ] myset.set_from_class_with_nine (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.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)
[ 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.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.erase_with_set_from_class
test02.cpp:669: Failure
Expected equality of these values:
  S.size()
    Which is: 1
  (int)V.size()
    Which is: 0
[  FAILED  ] myset.erase_with_set_from_class (0 ms)
[ RUN      ] myset.erase_root
test02.cpp:714: Failure
Expected equality of these values:
  S.size()
    Which is: 9
  (int)V.size()
    Which is: 8
[  FAILED  ] myset.erase_root (0 ms)
[ RUN      ] myset.erase_root_with_dups
test02.cpp:762: Failure
Expected equality of these values:
  S.size()
    Which is: 13
  (int)V.size()
    Which is: 10
[  FAILED  ] myset.erase_root_with_dups (0 ms)
[ RUN      ] myset.erase_interior_with_two_from_left


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


./run_valgrind: line 95:    91 Segmentation fault      (core dumped) timelimit -p -t "${TIMEOUT}" -T "${TIMEOU
************************************************************************
** Test Number: 3 **


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

```
*****************************************
** test case failed...                  **
*****************************************


*****************************************
** just in case test crashed, running    **
** valgrind to help pinpoint error...    **
*****************************************
./run_valgrind: line 227:    96 Segmentation fault      valgrind --tool=memcheck --leak-check=no --error-exitcod


==96== Memcheck, a memory error detector
==96== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==96== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==96== Command: ./test.exe
==96==
==96== Stack overflow in thread #1: can't grow stack to 0x1ffe601000
==96==
==96== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==96==  Access not within mapped region at address 0x1FFE601FF8
==96== Stack overflow in thread #1: can't grow stack to 0x1ffe601000
==96==    at 0x14428F: int* std::__copy_move<false, true, std::random_access_iterator_tag>::__copy_m<int>(int
==96==  If you believe this happened as a result of a stack
==96==  overflow in your program's main thread (unlikely but
==96==  possible), you can try to increase the size of the
==96==  main thread stack using the --main-stacksize= flag.
==96==  The main thread stack size used in this run was 10485760.
==96==
==96== HEAP SUMMARY:
==96==     in use at exit: 92,062 bytes in 153 blocks
==96==   total heap usage: 164,803 allocs, 164,650 frees, 844,792 bytes allocated
==96==
==96== For a detailed leak analysis, rerun with: --leak-check=full
==96==
==96== For lists of detected and suppressed errors, rerun with: -s
==96== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)


*****************************************************
** Test output (first 100 lines) **
[==========] Running 39 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 39 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_with_more_movies_and_dups
[       OK ] myset.set_with_more_movies_and_dups (0 ms)
[ RUN      ] myset.set_from_class_with_nine
[       OK ] myset.set_from_class_with_nine (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.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)
[ 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.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.erase_with_set_from_class
test03.cpp:669: Failure
Expected equality of these values:
  S.size()
    Which is: 1
  (int)V.size()
    Which is: 0
[  FAILED  ] myset.erase_with_set_from_class (0 ms)
```

```
[ RUN      ] myset.erase_root
test03.cpp:714: Failure
Expected equality of these values:
  S.size()
    Which is: 9
  (int)V.size()
    Which is: 8
[  FAILED  ] myset.erase_root (0 ms)
[ RUN      ] myset.erase_root_with_dups
test03.cpp:762: Failure
Expected equality of these values:
  S.size()
    Which is: 13
  (int)V.size()
    Which is: 10
[  FAILED  ] myset.erase_root_with_dups (0 ms)
[ RUN      ] myset.erase_interior_with_two_from_left


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

Go go go!

Test 1

Test 1: test01 (handling duplicates, stress tests, no erase) -- yay, output correct!

Test 2

Test 2: test02 (duplicates + erase, stress tests) -- program ran but output is incorrect.

Test 3

Test 3: test06 (valgrind check of all non-stress tests) -- program ran but output is incorrect.

Submitted Files

multiset.h

Download

```
/*multiset.h*/

//
// Implements a multiset with duplicates.
//
// Ishan Mukherjee
// CS 211
// Northwestern University
//
// Original template: Prof. Joe Hummel
// Northwestern University
// CS 211
//

// my note: this note still applies to std::multiset
//
// 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 multiset
{
public:
  // ########################################################################
  //
  // A node in the search tree:
  //
  class NODE {
  private:
    vector<TKey> Keys;
    bool  isThreaded : 1; // 1 bit
```

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

```
96    // set methods:
97    //
98  public:
99    //
100   // default constructor:
101   //
102   multiset()
103     : Root(nullptr), Size(0)
104   { }
105
106   //
107   // copy constructor:
108   //
109 private:
110   void _copy(NODE* other)
111   {
112     if (other == nullptr)
113       return;
114     else {
115       //
116       // we make a copy using insert so that threads
117       // are recreated properly in the copy:
118       //
119       vector<TKey> keys = other->get_Keys();
120       for (TKey key : keys) {
121         this->insert(key);
122       }
123
124       // code from the age of vanilla sets
125       // this->insert(other->get_RepresentativeKey());
126
127       _copy(other->get_Left());
128       _copy(other->get_Right());
129     }
130   }
131
132 public:
133   multiset(const multiset& other)
134     : Root(nullptr), Size(0)
135   {
136     _copy(other.Root);
137   }
138
139   //
140   // destructor:
141   //
142 private:
143   void _destroy(NODE* cur)
144   {
```

```
145     if (cur == nullptr)
146       ;
147     else {
148       _destroy(cur->get_Left());
149       _destroy(cur->get_Right());
150       delete cur;
151     }
152   }
153
154 public:
155   ~multiset()
156   {
157     //
158     // NOTE: this is commented out UNTIL you are ready. The last
159     // step is to uncomment this and check for memory leaks.
160     //
161     _destroy(this->Root);
162   }
163
164   //
165   // size
166   //
167   // Returns # of elements in the multiset
168   //
169   int size()
170   {
171     return this->Size;
172   }
173
174   //
175   // contains
176   //
177   // Returns true if multiset contains key, false if not
178   //
179 private:
180   bool _contains(NODE* cur, TKey key)
181   {
182     if (cur == nullptr)
183       return false;
184     else {
185
186       if (key < cur->get_RepresentativeKey()) // search left:
187         return _contains(cur->get_Left(), key);
188       else if (cur->get_RepresentativeKey() < key) // search right:
189         return _contains(cur->get_Right(), key);
190       else // must be equal, found it!
191         return true;
192     }
193   }
```

```
194
195 public:
196   bool contains(TKey key)
197   {
198     return _contains(this->Root, key);
199   }
200
201   int count(TKey key) {
202     NODE* cur = this->Root;
203
204     while (cur != nullptr) {
205       if (key < cur->get_RepresentativeKey()) {
206         cur = cur->get_Left();
207       }
208       else if (cur->get_RepresentativeKey() < key) {
209         cur = cur->get_Right();
210       }
211       else {
212         return cur->get_Keys().size();
213       }
214     }
215     return 0;
216   }
217
218   //
219   // insert
220   //
221   // Inserts the given key into the multiset.
222   //
223   void insert(TKey key)
224   {
225     NODE* parent = nullptr;
226     NODE* cur = this->Root;
227
228     //
229     // 1. Search for key, return if found:
230     //
231     while (cur != nullptr) {
232       if (key < cur->get_RepresentativeKey()) { // left:
233         parent = cur;
234         cur = cur->get_Left();
235       }
236       else if (cur->get_RepresentativeKey() < key) { // right:
237         parent = cur;
238         cur = cur->get_Right();
239       }
240       else { // must be equal => already in tree
241         cur->addToKeys(key);
242         this->Size++;
```

```
243         return;
244       }
245     }
246
247     //
248     // 2. If not found, insert where we
249     //    fell out of the tree:
250     //
251     NODE* n = new NODE(key);
252
253     if (parent == nullptr) {
254       //
255       // tree is empty, insert at root:
256       //
257       this->Root = n;
258     }
259     else if (key < parent->get_RepresentativeKey()) {
260       //
261       // we are to the left of our parent:
262       //
263       parent->set_Left(n);
264       n->set_isThreaded(true);
265       n->set_Right(parent);
266     }
267     else {
268       //
269       // we are to the right of our parent:
270       //
271       // inherit parent's thread
272       n->set_isThreaded(parent->get_isThreaded());
273       n->set_Right(parent->get_Thread());
274       // update parent's thread
275       parent->set_isThreaded(false);
276       parent->set_Right(n);
277     }
278
279     //
280     // STEP 3: update size and return
281     //
282     this->Size++;
283     return;
284   }
285
286   //
287   // toPairs
288   //
289   // Returns pairs of elements: <element, threaded element>.
290   // If a node is not threaded: <element, no_element value>.
291   //
```

```
private:
  void _toPairs(NODE* cur, std::vector<std::pair<TKey, TKey>>& P, TKey no_element) {
    if (cur == nullptr) {
      return;
    } else {
      //
      // we want them in order, so go left, then
      // middle, then right:
      //

      _toPairs(cur->get_Left(), P, no_element);

      // determine threadValue
      TKey threadValue;
      if (cur->get_isThreaded()) {
        threadValue = cur->get_Thread()->get_RepresentativeKey();
      } else {
        threadValue = no_element;
      }

      P.push_back(std::make_pair(cur->get_RepresentativeKey(), threadValue));

      _toPairs(cur->get_Right(), P, no_element);
    }
  }
public:
  std::vector<std::pair<TKey, TKey>> toPairs(TKey no_element)
  {
    std::vector<std::pair<TKey, TKey>> P;

    _toPairs(this->Root, P, no_element);

    return P;
  }

  //
  // []
  //
  // Returns true if multiset contains key, false if not.
  //
  bool operator[](TKey key)
  {
    return this->contains(key);
  }

  //
  // toVector
  //
  // Returns the elements of the multiset, in order,
```

```
341   // in a vector.
342   //
343 private:
344   void _toVector(NODE* cur, std::vector<TKey>& V) {
345     if (cur == nullptr)
346       return;
347     else {
348       //
349       // we want them in order, so go left, then
350       // middle, then right:
351       //
352       _toVector(cur->get_Left(), V);
353 
354       // old code for no duplicates:
355       // V.push_back(cur->get_Key()); (should be cur->get_RepresentativeKey() though)
356 
357       // append keys to V
358       vector<TKey> keys = cur->get_Keys();
359       V.insert(V.end(), keys.begin(), keys.end());
360       _toVector(cur->get_Right(), V);
361     }
362   }
363 
364 public:
365   std::vector<TKey> toVector()
366   {
367     std::vector<TKey> V;
368 
369     _toVector(this->Root, V);
370 
371     return V;
372   }
373 
374 
375   // ##################################################################
376   //
377   // class iterator:
378   //
379 private:
380   class iterator
381   {
382   private:
383     NODE* Ptr;
384     int KeyIdx;
385 
386   public:
387 
388     iterator(NODE* ptr) : Ptr(ptr), KeyIdx(0) {}
389 
```

```
390     //
391     // !=
392     //
393     // Returns true if the given iterator is not equal to
394     // this iterator.
395     //
396     bool operator!=(iterator other) {
397       return (this->Ptr != other.Ptr || this->KeyIdx != other.KeyIdx);
398     }
399
400     //
401     // ++
402     //
403     // Advances the iterator to the next ordered element of
404     // the multiset; if the iterator cannot be advanced, ++ has
405     // no effect.
406     //
407     void operator++() {
408       // if vector of keys still not exhausted
409       if (this->KeyIdx + 1 < (int) this->Ptr->get_Keys().size()) {
410         this->KeyIdx++;
411         return;
412       }
413
414       // else
415       this->KeyIdx = 0;
416       if (Ptr->get_isThreaded()) {
417         this->Ptr = Ptr->get_Thread();
418       } else {
419         this->Ptr = leftmost(Ptr->get_Right());
420       }
421       return;
422     }
423
424     //
425     // *
426     //
427     // Returns the key denoted by the iterator; this
428     // code will throw an out_of_range exception if
429     // the iterator does not denote an element of the
430     // multiset.
431     //
432     TKey operator*()
433     {
434       if (this->Ptr == nullptr)
435         throw std::out_of_range("multiset::iterator:operator*");
436
437       return this->Ptr->get_Keys()[this->KeyIdx];
438     }
```

```
439
440     //
441     // ==
442     //
443     // Returns true if the given iterator is equal to
444     // this iterator.
445     //
446     bool operator==(iterator other)
447     {
448       return (this->Ptr == other.Ptr && this->KeyIdx == other.KeyIdx);
449     }
450   };
451
452 private:
453   // helper function to return leftmost child of subtree
454   static NODE* leftmost(NODE* root) {
455     NODE* current = root;
456     while (current && current->get_Left()) {
457       current = current->get_Left();
458     }
459     return current;
460   }
461   static NODE* rightmost(NODE* root) {
462     NODE* current = root;
463     while (current && current->get_Right()) {
464       current = current->get_Right();
465     }
466     return current;
467   }
468
469 public:
470
471   iterator begin() const {
472     return iterator(leftmost(this->Root));
473   }
474
475   // ##################################################################
476   //
477   // find:
478   //
479   // If the multiset contains key, then an iterator denoting this
480   // element is returned. If the multiset does not contain key,
481   // then multiset.end() is returned.
482   //
483 public:
484   iterator find(TKey key)
485   {
486     NODE* cur = this->Root;
487
```

```
488     while (cur != nullptr) {
489       if (key < cur->get_RepresentativeKey()) { // search left:
490         cur = cur->get_Left();
491       }
492       else if (cur->get_RepresentativeKey() < key) { // search right:
493         cur = cur->get_Right();
494       }
495       else { // must be equal, found it!
496         return iterator(cur);
497       }
498     }
499
500     // if get here, not found
501     return iterator(nullptr);
502   }
503
504   void erase(TKey key) {
505     NODE* parent = nullptr;
506     NODE* cur = this->Root;
507
508     //
509     // 1. Search for key, set cur to it if found:
510     //
511     while (cur != nullptr) {
512       if (key < cur->get_RepresentativeKey()) { // left:
513         parent = cur;
514         cur = cur->get_Left();
515       }
516       else if (cur->get_RepresentativeKey() < key) { // right:
517         parent = cur;
518         cur = cur->get_Right();
519       }
520       else { // must be equal => already in tree
521         break;
522       }
523     }
524
525     //
526     // 2. If found, delete cur:
527     //
528
529     if (parent == nullptr || cur == nullptr) {
530       //
531       // tree is empty or not found, return:
532       //
533       return;
534     }
535
536     // else
```

```
537     this->Size -= cur->get_Keys().size();
538
539     NODE* new_child = nullptr;
540
541     if (!cur->get_Left() && !cur->get_Right()) {
542       // cur has no children
543       // child remains nullptr
544     }
545     else if (cur->get_Left() && !cur->get_Right()) {
546       // left child but no right child
547       NODE* rightmost_of_left_subtree = rightmost(cur->get_Left());
548       // set thread to parent of cur
549       rightmost_of_left_subtree->set_Right(cur->get_Thread());
550       new_child = cur->get_Left();
551     }
552     else if (!cur->get_Left() && cur->get_Right()) {
553       // right child but no left child
554       new_child = cur->get_Right();
555     }
556     else {
557       // both left and right children
558
559       // find leftmost of right subtree (cur_r_st)
560       NODE* leftmost_of_right_subtree = leftmost(cur->get_Left());
561       cur->set_Keys(leftmost_of_right_subtree->get_Keys());
562       erase(leftmost_of_right_subtree->get_RepresentativeKey());
563     }
564
565     if (key < parent->get_RepresentativeKey()) {
566       // we are to the left of our parent
567
568       parent->set_Left(new_child);
569       delete cur;
570     }
571     else {
572       // right of parent
573
574       parent->set_Right(new_child);
575       delete cur;
576     }
577
578     //
579     // STEP 3: return
580     //
581     return;
582   }
583
584   //
585   // end:
```

```
586   //
587   // Returns an iterator to the end of the iteration space,
588   // i.e. to no element. In other words, if your iterator
589   // == multiset.end(), then you are not pointing to an element.
590   //
591   iterator end()
592   {
593     return iterator(nullptr);
594   }
595
596 };
597
```

tests.cpp

Download

```cpp
/*tests.c*/

//
// Google test cases for our multiset class.
//
// Ishan Mukherjee
// CS 211
// Northwestern University
//
// 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 "multiset.h"
#include "gtest/gtest.h"

// example class to test support for nontrivial objects
// moved from instructor's tests to top of tests file

class Movie
{
public:
  string Title;
  int    ID;
  double Revenue;

  Movie(string title, int id, double revenue)
    : Title(title), ID(id), Revenue(revenue)
  { }

  bool operator<(const Movie& other)
  {
    if (this->Title < other.Title)
      return true;
```

```
    else
      return false;
  }
};

// my tests

TEST(mymultiset, toPairs_empty_multiset)
{
  multiset<int> S;
  auto pairs = S.toPairs(-1);
  ASSERT_TRUE(pairs.empty());
}

TEST(mymultiset, toPairs_multiset_with_one_element)
{
  multiset<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);
}

TEST(mymultiset, toPairs_with_multiple_elements)
{
  multiset<int> S;

  S.insert(30);
  S.insert(15);
  S.insert(50);
  S.insert(8);
  S.insert(25);
  S.insert(70);
  S.insert(20);
  S.insert(28);
  S.insert(60);

  ASSERT_EQ(S.size(), 9);

  auto pairs = S.toPairs(-1);

  vector<pair<int, int>> expected = {
    {8, 15}, {15, -1}, {20, 25}, {25, -1}, {28, 30},
    {30, -1}, {50, -1}, {60, 70}, {70, -1}
  };
```

```
96
97    ASSERT_EQ(pairs.size(), expected.size());
98
99    for (size_t i = 0, n = expected.size(); i < n; ++i)
100   {
101     ASSERT_EQ(pairs[i], expected[i]);
102   }
103 }
104
105 TEST(mymultiset, toPairs_single_string_element)
106 {
107   multiset<string> S;
108
109   S.insert("Angola");
110
111   ASSERT_EQ(S.size(), 1);
112
113   auto pairs = S.toPairs("FIN");
114
115   vector<pair<string, string>> expected = { {"Angola", "FIN"} };
116
117   ASSERT_EQ(pairs.size(), expected.size());
118   ASSERT_EQ(pairs[0], expected[0]);
119 }
120
121 TEST(mymultiset, toPairs_multiple_string_elements)
122 {
123   multiset<string> S;
124
125   S.insert("Congo");
126   S.insert("Brazil");
127   S.insert("Djibouti");
128   S.insert("Angola");
129   S.insert("Eritrea");
130
131   ASSERT_EQ(S.size(), 5);
132
133   auto pairs = S.toPairs(" ");
134
135   vector<pair<string, string>> expected = {
136     {"Angola", "Brazil"},
137     {"Brazil", "Congo"},
138     {"Congo", " "},
139     {"Djibouti", " "},
140     {"Eritrea", " "}
141   };
142
143   ASSERT_EQ(pairs.size(), expected.size());
144
```

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

TEST(mymultiset, foreach_with_empty_multiset)
{
  multiset<long> S;
  int count = 0;

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

  ASSERT_EQ(count, 0);
}

TEST(mymultiset, foreach_with_single_element_multiset)
{
  multiset<int> S;
  S.insert(42);
  int count = 0;
  int val = 0;

  for (int x : S)
  {
    val = x;
    count++;
  }

  ASSERT_EQ(count, 1);
  ASSERT_EQ(val, 42);
}

TEST(mymultiset, foreach_with_increasing_ints_no_duplicates)
{
  multiset<int> S;

  std::vector<int> expected = {0, 10, 20, 30};
  sort(expected.begin(), expected.end());

  for (int i : expected) {
    S.insert(i);
  }

  std::vector<int> elements;

```

```
194   for (int x : S)
195   {
196     elements.push_back(x);
197   }
198
199   ASSERT_EQ(elements.size(), expected.size());
200
201   for (size_t i = 0; i < expected.size(); ++i)
202   {
203     ASSERT_EQ(elements[i], expected[i]);
204   }
205 }
206
207 TEST(mymultiset, foreach_with_unordered_ints_duplicates)
208 {
209   multiset<int> S;
210
211   std::vector<int> expected = {21, 22, 20, 19, 23, 21, 21, 23};
212   sort(expected.begin(), expected.end());
213
214   for (int i : expected) {
215     S.insert(i);
216   }
217
218   std::vector<int> elements;
219
220   for (int x : S)
221   {
222     elements.push_back(x);
223   }
224
225   ASSERT_EQ(S.size(), (int) expected.size());
226   ASSERT_EQ(elements.size(), expected.size());
227
228   for (size_t i = 0; i < expected.size(); ++i)
229   {
230     ASSERT_EQ(elements[i], expected[i]);
231   }
232 }
233
234 TEST(mymultiset, foreach_with_single_string)
235 {
236   multiset<string> S;
237   S.insert("Hello");
238   int count = 0;
239   string value;
240
241   for (const string& x : S)
242   {
```

```
    value = x;
    count++;
  }

  ASSERT_EQ(count, 1);
  ASSERT_EQ(value, "Hello");
}

TEST(mymultiset, foreach_with_increasing_strings_no_duplicates)
{
  multiset<string> S;

  S.insert("apple");
  S.insert("banana");
  S.insert("cherry");

  std::vector<string> elements;

  for (const string& x : S)
  {
    elements.push_back(x);
  }

  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(mymultiset, count_with_unordered_strings_no_duplicates)
{
  multiset<string> S;

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

  for (string s : expected) {
    S.insert(s);
  }

  for (string s : expected)
  {
    ASSERT_EQ(S.count(s), 1);
  }

  ASSERT_EQ(S.count("dragon fruit"), 0);
```

```
}

TEST(mymultiset, count_with_unordered_strings_duplicates)
{
  multiset<string> S;

  std::vector<string> expected = {"fig", "grape", "cherry", "banana", "elderberry"};

  for (string s : expected) {
    S.insert(s);
  }

  int dup1_count = 3;
  string dup1 = "apple";
  int dup2_count = 4;
  string dup2 = "honeydew";

  for (int i = 0; i < dup1_count; i++) {
    S.insert(dup1);
  }

  for (int i = 0; i < dup2_count; i++) {
    S.insert(dup2);
  }

  for (string s : expected)
  {
    ASSERT_EQ(S.count(s), 1);
  }
  ASSERT_EQ(S.count(dup1), dup1_count);
  ASSERT_EQ(S.count(dup2), dup2_count);

  ASSERT_EQ(S.count("dragon fruit"), 0);
}

TEST(mymultiset, count_with_movies_no_duplicates)
{
  multiset<Movie> S;

  Movie Sleepless("Sleepless in Seattle", 123, 123456789.00);
  Movie Matrix("The Matrix", 456, 400000000.00);
  Movie AnimalHouse("Animal House", 789, 1000000000.00);
  vector<Movie> expected = {Sleepless, Matrix, AnimalHouse};

  for (Movie m : expected) {
    S.insert(m);
  }

  for (Movie m : expected)
```

```
  {
    ASSERT_EQ(S.count(m), 1);
  }

  ASSERT_EQ(S.count(Movie("Goodfellas", 101, 0)), 0);
  ASSERT_EQ(S.count(Movie("The Matrix", 456, 0)), 1); // revenue doesn't matter for equality
  ASSERT_EQ(S.count(Movie("The Matrix", 789, 400000000.00)), 1); // ID doesn't matter for equality
  ASSERT_EQ(S.count(Movie("The Matrix", 0, 0)), 1); // neither revenue nor ID matters for equality
  ASSERT_EQ(S.count(Movie("the matrix", 456, 400000000.00)), 0); // case matter for equality
}

TEST(mymultiset, count_with_movies_duplicates)
{
  multiset<Movie> S;

  Movie Sleepless("Sleepless in Seattle", 123, 123456789.00);
  Movie Matrix("The Matrix", 456, 400000000.00);
  Movie AnimalHouse("Animal House", 789, 1000000000.00);
  vector<Movie> expected = {Sleepless, Matrix, AnimalHouse};

  int half_dup_count = 4;

  for (int i = 0; i < half_dup_count; i++) {
    S.insert(Movie("Napoleon Dynamite", i, i*100000));
  }

  for (Movie m : expected) {
    S.insert(m);
  }

  for (Movie m : expected)
  {
    ASSERT_EQ(S.count(m), 1);
  }

  for (int i = 0; i < half_dup_count; i++) {
    S.insert(Movie("Napoleon Dynamite", 2*i, 2*i*100000));
  }

  ASSERT_EQ(S.count(Movie("Napoleon Dynamite", 0, 0)), 2* half_dup_count);
}

// stress tests

const long long N_STRESS = 30;

TEST(mymultiset, foreach_stress_test_with_strictly_increasing_nums)
{
  multiset<int> S;
```

```
390   for (int i = 0; i < N_STRESS; ++i) {
391     S.insert(i);
392   }
393
394   // check multiset size
395   ASSERT_EQ(S.size(), N_STRESS);
396
397   // check values
398   int count = 0;
399   for (int x : S) {
400     ASSERT_EQ(x, count);
401     count++;
402   }
403
404   ASSERT_EQ(count, N_STRESS);
405 }
406
407 TEST(mymultiset, foreach_stress_test_with_random_nums)
408 {
409   multiset<int> S;
410
411   // populate a vector with random numbers
412   // code below is from https://stackoverflow.com/a/23143753
413   std::random_device rnd_device;
414   std::mt19937 mersenne_engine {rnd_device()};
415   std::uniform_int_distribution<int> dist {1, 42};
416   auto gen = [&dist, &mersenne_engine](){
417     return dist(mersenne_engine);
418   };
419   vector<int> expected(N_STRESS);
420   generate(begin(expected), end(expected), gen);
421
422   // populate multiset
423   for (int i = 0; i < N_STRESS; ++i) {
424     S.insert(expected[i]);
425   }
426
427   std::sort(expected.begin(), expected.end());
428
429   // code to remove duplicates (no longer applies)
430   //
431   // std::set<int> s;
432   // unsigned size = sorted_expected.size();
433   // for( unsigned i = 0; i < size; ++i ) s.insert( sorted_expected[i] );
434   // sorted_expected.assign( s.begin(), s.end() );
435
436   // check multiset size
437   ASSERT_EQ(S.size(), (int) expected.size());
438
```

```
439   // check values
440   int count = 0;
441   for (int x : S) {
442     ASSERT_EQ(x, expected[count]);
443     count++;
444   }
445 }
446
447 TEST(mymultiset, toVector_with_duplicates)
448 {
449   multiset<int> S;
450
451   // 61 is repeated (3 instances)
452   // 11 is repeated (2 instances)
453   vector<int> V = { 22, 11, 49, 61, 3, 19, 35, 11, 61, 30, 41, 61 };
454
455   for (auto x : V)
456     S.insert(x);
457
458   ASSERT_EQ(S.size(), (int) V.size());
459
460   vector<int> V2 = S.toVector();
461
462   ASSERT_EQ(V2.size(), V.size());
463
464   std::sort(V.begin(), V.end());
465
466   //
467   // V and V2 should have the same elements in
468   // the same order:
469   //
470   auto iterV = V.begin();
471   auto iterV2 = V2.begin();
472
473   while (iterV != V.end()) {
474     ASSERT_EQ(*iterV, *iterV2);
475
476     iterV++;
477     iterV2++;
478   }
479 }
480
481 TEST(mymultiset, find_should_return_first_inserted_key)
482 {
483   multiset<Movie> movies;
484
485   movies.insert( Movie("The Matrix", 603, 1999) );
486   movies.insert( Movie("Toy Story", 862, 1995) );
487   movies.insert( Movie("Hamlet", 10, 1948) );
```

```
  movies.insert( Movie("Dracula", 5678, 1979) );
  movies.insert( Movie("Dracula", 1234, 1931) );
  movies.insert( Movie("Hamlet", 8, 2009) );
  movies.insert( Movie("Hamlet", 9, 1964) );

  auto iter = movies.find(Movie("Hercules", 0, 0));
  ASSERT_TRUE(iter == movies.end());

  iter = movies.find(Movie("Toy Story", 0, 0));
  ASSERT_TRUE((*iter).ID == 862);
  ASSERT_TRUE((*iter).Revenue == 1995);

  iter = movies.find(Movie("Dracula", 0, 0));
  ASSERT_TRUE((*iter).ID == 5678);
  ASSERT_TRUE((*iter).Revenue == 1979);

  iter = movies.find(Movie("Hamlet", 0, 0));
  ASSERT_TRUE((*iter).ID == 10);
  ASSERT_TRUE((*iter).Revenue == 1948);
}

TEST(mymultiset, erase_nonexistent_key)
{
  multiset<int> S;

  vector<int> before = {10, 15, 16, 11, 4, 13};
  vector<int> after = {10, 15, 16, 11, 4, 13};

  for (int x : before) {
    S.insert(x);
  }

  sort(before.begin(), before.end());

  ASSERT_EQ(S.size(), (int) before.size());
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, before[count]);
    count++;
  }

  sort(after.begin(), after.end());

  S.erase(21);
  S.erase(-1);
  S.erase(0);
  S.erase(5);

  ASSERT_EQ(S.size(), (int) after.size());
```

```
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_left_with_no_children)
{
  multiset<int> S;

  vector<int> before = {10, 9, 9, 8, 7, 7, 7, 5};
  vector<int> after = {10, 9, 9, 8, 7, 7, 7};

  for (int x : before) {
    S.insert(x);
  }

  sort(before.begin(), before.end());

  ASSERT_EQ(S.size(), (int) before.size());
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, before[count]);
    count++;
  }

  sort(after.begin(), after.end());

  S.erase(5);

  ASSERT_EQ(S.size(), (int) after.size());
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_left_with_no_children_duplicates)
{
  multiset<int> S;

  vector<int> before = {10, 9, 9, 8, 7, 7, 7, 5, 5, 5, 5};
  vector<int> after = {10, 9, 9, 8, 7, 7, 7};

  for (int x : before) {
    S.insert(x);
  }
```

```
586
587   sort(before.begin(), before.end());
588
589   ASSERT_EQ(S.size(), (int) before.size());
590   int count = 0;
591   for (int x : S) {
592     ASSERT_EQ(x, before[count]);
593     count++;
594   }
595
596   sort(after.begin(), after.end());
597
598   S.erase(5);
599
600   ASSERT_EQ(S.size(), (int) after.size());
601   count = 0;
602   for (int x : S) {
603     ASSERT_EQ(x, after[count]);
604     count++;
605   }
606 }
607
608 TEST(mymultiset, erase_left_with_left_children_duplicates)
609 {
610   multiset<int> S;
611
612   vector<int> before = {10, 9, 9, 8, 7, 7, 7, 5, 5, 5, 5, 4, 3};
613   vector<int> after = {10, 9, 9, 8, 7, 7, 7, 4, 3};
614
615   for (int x : before) {
616     S.insert(x);
617   }
618
619   sort(before.begin(), before.end());
620
621   ASSERT_EQ(S.size(), (int) before.size());
622   int count = 0;
623   for (int x : S) {
624     ASSERT_EQ(x, before[count]);
625     count++;
626   }
627
628   sort(after.begin(), after.end());
629
630   S.erase(5);
631
632   ASSERT_EQ(S.size(), (int) after.size());
633   count = 0;
634   for (int x : S) {
```

```
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_right_with_no_children)
{
  multiset<int> S;

  vector<int> before = {10, 11, 12, 12, 12, 12, 14, 15};
  vector<int> after = {10, 11, 12, 12, 12, 12, 14};

  for (int x : before) {
    S.insert(x);
  }

  sort(before.begin(), before.end());

  ASSERT_EQ(S.size(), (int) before.size());
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, before[count]);
    count++;
  }

  S.erase(15);

  ASSERT_EQ(S.size(), (int) after.size());
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_right_with_no_children_duplicates)
{
  multiset<int> S;

  vector<int> before = {10, 11, 12, 12, 12, 12, 14, 15, 15, 15};
  vector<int> after = {10, 11, 12, 12, 12, 12, 14};

  for (int x : before) {
    S.insert(x);
  }

  sort(before.begin(), before.end());

  ASSERT_EQ(S.size(), (int) before.size());
```

```
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, before[count]);
    count++;
  }

  S.erase(15);

  ASSERT_EQ(S.size(), (int) after.size());
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_right_with_left_children_duplicates)
{
  multiset<int> S;

  vector<int> before = {10, 11, 12, 12, 12, 12, 15, 15, 15, 14};
  vector<int> after = {10, 11, 12, 12, 12, 12, 14};

  for (int x : before) {
    S.insert(x);
  }

  sort(before.begin(), before.end());

  ASSERT_EQ(S.size(), (int) before.size());
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, before[count]);
    count++;
  }

  S.erase(15);

  ASSERT_EQ(S.size(), (int) after.size());
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

TEST(mymultiset, erase_right_with_right_children_duplicates)
{
  multiset<int> S;
```

```
733
734   vector<int> before = {10, 11, 12, 12, 12, 12, 15, 15, 15, 16, 17, 18};
735   vector<int> after = {10, 11, 12, 12, 12, 12, 16, 17, 18};
736   int to_erase = 15;
737
738   for (int x : before) {
739     S.insert(x);
740   }
741
742   sort(before.begin(), before.end());
743
744   ASSERT_EQ(S.size(), (int) before.size());
745   int count = 0;
746   for (int x : S) {
747     ASSERT_EQ(x, before[count]);
748     count++;
749   }
750
751   S.erase(to_erase);
752
753   ASSERT_EQ(S.size(), (int) after.size());
754   count = 0;
755   for (int x : S) {
756     ASSERT_EQ(x, after[count]);
757     count++;
758   }
759 }
760
761 TEST(mymultiset, erase_right_with_both_children_duplicates)
762 {
763   multiset<int> S;
764
765   vector<int> before = {1, 4, 5, 3, 2};
766   vector<int> after = {1, 2, 3, 4};
767   int to_erase = 4;
768
769   for (int x : before) {
770     S.insert(x);
771   }
772
773   sort(before.begin(), before.end());
774
775   ASSERT_EQ(S.size(), (int) before.size());
776   int count = 0;
777   for (int x : S) {
778     ASSERT_EQ(x, before[count]);
779     count++;
780   }
781
```

```
  S.erase(to_erase);

  ASSERT_EQ(S.size(), (int) after.size());
  count = 0;
  for (int x : S) {
    ASSERT_EQ(x, after[count]);
    count++;
  }
}

// instructor's tests

TEST(mymultiset, empty_multiset)
{
  multiset<int> S;

  ASSERT_EQ(S.size(), 0);
}

TEST(mymultiset, multiset_with_one)
{
  multiset<int> S;

  ASSERT_EQ(S.size(), 0);

  S.insert(123);

  ASSERT_EQ(S.size(), 1);

  ASSERT_TRUE(S.contains(123));
  ASSERT_TRUE(S[123]);

  ASSERT_FALSE(S.contains(100));
  ASSERT_FALSE(S[100]);
  ASSERT_FALSE(S.contains(200));
  ASSERT_FALSE(S[200]);
}

TEST(mymultiset, multiset_with_four_strings)
{
  multiset<string> S;

  ASSERT_EQ(S.size(), 0);

  S.insert("banana");
  S.insert("apple");
  S.insert("chocolate");
  S.insert("pear");

```

```
831   ASSERT_EQ(S.size(), 4);
832
833   ASSERT_TRUE(S.contains("pear"));
834   ASSERT_TRUE(S["banana"]);
835   ASSERT_TRUE(S.contains("chocolate"));
836   ASSERT_TRUE(S["apple"]);
837
838   ASSERT_FALSE(S.contains("Apple"));
839   ASSERT_FALSE(S["carmel"]);
840   ASSERT_FALSE(S.contains("appl"));
841   ASSERT_FALSE(S["chocolatee"]);
842 }
843
844 TEST(mymultiset, multiset_with_movies)
845 {
846   multiset<Movie> S;
847
848   ASSERT_EQ(S.size(), 0);
849
850   Movie Sleepless("Sleepless in Seattle", 123, 123456789.00);
851   S.insert(Sleepless);
852
853   Movie Matrix("The Matrix", 456, 400000000.00);
854   S.insert(Matrix);
855
856   Movie AnimalHouse("Animal House", 789, 1000000000.00);
857   S.insert(AnimalHouse);
858
859   ASSERT_EQ(S.size(), 3);
860
861   vector<Movie> V = S.toVector();
862
863   ASSERT_EQ(V[0].Title, "Animal House");
864   ASSERT_EQ(V[1].Title, "Sleepless in Seattle");
865   ASSERT_EQ(V[2].Title, "The Matrix");
866 }
867
868 TEST(mymultiset, multiset_from_class_with_nine)
869 {
870   multiset<int> S;
871
872   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
873
874   for (auto x : V)
875     S.insert(x);
876
877   ASSERT_EQ(S.size(), (int) V.size());
878
879   for (auto x : V) {
```

```
880     ASSERT_TRUE(S.contains(x));
881     ASSERT_TRUE(S[x]);
882   }
883
884   ASSERT_FALSE(S.contains(0));
885   ASSERT_FALSE(S[0]);
886   ASSERT_FALSE(S.contains(2));
887   ASSERT_FALSE(S[2]);
888   ASSERT_FALSE(S.contains(4));
889   ASSERT_FALSE(S[4]);
890   ASSERT_FALSE(S.contains(29));
891   ASSERT_FALSE(S[31]);
892   ASSERT_FALSE(S.contains(40));
893   ASSERT_FALSE(S[42]);
894 }
895
896 TEST(mymultiset, multiset_no_duplicates)
897 {
898   multiset<int> S;
899
900   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
901
902   for (auto x : V)
903     S.insert(x);
904
905   // try to insert them all again:
906   for (auto x : V)
907     S.insert(x);
908
909   // should be twice the size since all numbers reinserted
910   ASSERT_EQ(S.size(), 2 * (int) V.size());
911
912   for (auto x : V) {
913     ASSERT_TRUE(S.contains(x));
914     ASSERT_TRUE(S[x]);
915   }
916 }
917
918 TEST(mymultiset, toVector)
919 {
920   multiset<int> S;
921
922   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
923
924   for (auto x : V)
925     S.insert(x);
926
927   ASSERT_EQ(S.size(), (int) V.size());
928
```

```
929   vector<int> V2 = S.toVector();
930
931   ASSERT_EQ(V2.size(), V.size());
932
933   std::sort(V.begin(), V.end());
934
935   //
936   // V and V2 should have the same elements in
937   // the same order:
938   //
939   auto iterV = V.begin();
940   auto iterV2 = V2.begin();
941
942   while (iterV != V.end()) {
943     ASSERT_EQ(*iterV, *iterV2);
944
945     iterV++;
946     iterV2++;
947   }
948 }
949
950 TEST(mymultiset, copy_empty)
951 {
952   multiset<int> S1;
953
954   {
955     //
956     // create a new scope, which will trigger destructor:
957     //
958     multiset<int> S2 = S1;  // this will call copy constructor:
959
960     S1.insert(123);  // this should have no impact on S2:
961     S1.insert(100);
962     S1.insert(150);
963
964     ASSERT_EQ(S2.size(), 0);
965
966     vector<int> V2 = S2.toVector();
967
968     ASSERT_EQ((int) V2.size(), 0);
969   }
970 }
971
972 TEST(mymultiset, copy_constructor)
973 {
974   multiset<int> S1;
975
976   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
977
```

```
978   for (auto x : V)
979     S1.insert(x);
980
981   ASSERT_EQ(S1.size(), (int) V.size());
982
983   {
984     //
985     // create a new scope, which will trigger destructor:
986     //
987     multiset<int> S2 = S1;  // this will call copy constructor:
988
989     S1.insert(123);  // this should have no impact on S2:
990     S1.insert(100);
991     S1.insert(150);
992
993     ASSERT_EQ(S2.size(), (int) V.size());
994
995     vector<int> V2 = S2.toVector();
996
997     ASSERT_EQ(V2.size(), V.size());
998
999     std::sort(V.begin(), V.end());
1000
1001    //
1002    // V and V2 should have the same elements in
1003    // the same order:
1004    //
1005    auto iterV = V.begin();
1006    auto iterV2 = V2.begin();
1007
1008    while (iterV != V.end()) {
1009      ASSERT_EQ(*iterV, *iterV2);
1010
1011      iterV++;
1012      iterV2++;
1013    }
1014
1015    S2.insert(1000);  // this should have no impact on S1:
1016    S2.insert(2000);
1017    S2.insert(3000);
1018    S2.insert(4000);
1019    S2.insert(5000);
1020
1021    V.push_back(123);
1022    V.push_back(100);
1023    V.push_back(150);
1024  }
1025
1026  //
```

```
1027   // the copy was just destroyed, the original multiset
1028   // should still be the same as it was earlier:
1029   //
1030   ASSERT_EQ(S1.size(), (int) V.size());
1031
1032   vector<int> V2 = S1.toVector();
1033
1034   ASSERT_EQ(V2.size(), V.size());
1035
1036   std::sort(V.begin(), V.end());
1037
1038   //
1039   // V and V2 should have the same elements in
1040   // the same order:
1041   //
1042   auto iterV = V.begin();
1043   auto iterV2 = V2.begin();
1044
1045   while (iterV != V.end()) {
1046     ASSERT_EQ(*iterV, *iterV2);
1047
1048     iterV++;
1049     iterV2++;
1050   }
1051 }
1052
1053 TEST(mymultiset, find_empty)
1054 {
1055   multiset<int> S;
1056
1057   auto iter = S.find(22);
1058   ASSERT_TRUE(iter == S.end());
1059 }
1060
1061 TEST(mymultiset, find_one)
1062 {
1063   multiset<int> S;
1064
1065   S.insert(1234);
1066
1067   auto iter = S.find(123);
1068   ASSERT_TRUE(iter == S.end());
1069
1070   iter = S.find(1234);
1071   if (iter == S.end()) {  // this should not happen:
1072     ASSERT_TRUE(false); // fail:
1073   }
1074
1075   ASSERT_EQ(*iter, 1234);
```

```
1076
1077   iter = S.find(1235);
1078   ASSERT_TRUE(iter == S.end());
1079 }
1080
1081 TEST(mymultiset, find_with_multiset_from_class)
1082 {
1083   multiset<int> S;
1084
1085   vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };
1086
1087   for (auto x : V)
1088     S.insert(x);
1089
1090   ASSERT_EQ(S.size(), (int) V.size());
1091
1092   //
1093   // make sure we can find each of the values we inserted:
1094   //
1095   for (auto x : V) {
1096     auto iter = S.find(x);
1097
1098     if (iter == S.end()) {  // this should not happen:
1099       ASSERT_TRUE(false); // fail:
1100     }
1101
1102     ASSERT_EQ(*iter, x);
1103   }
1104
1105   //
1106   // these searches should all fail:
1107   //
1108   auto iter = S.find(0);
1109   ASSERT_TRUE(iter == S.end());
1110
1111   iter = S.find(-1);
1112   ASSERT_TRUE(iter == S.end());
1113
1114   iter = S.find(1);
1115   ASSERT_TRUE(iter == S.end());
1116
1117   iter = S.find(4);
1118   ASSERT_TRUE(iter == S.end());
1119
1120   iter = S.find(34);
1121   ASSERT_TRUE(iter == S.end());
1122
1123   iter = S.find(36);
1124   ASSERT_TRUE(iter == S.end());
```

```
1125 }
1126
1127 // instructor's stress test
1128 // modified to use my N_STRESS instead of given N
1129 // this makes quick testing easier
1130
1131 TEST(mymultiset, stress_test)
1132 {
1133   multiset<long long> S;
1134   // edited to create a multiset, not a set
1135   std::multiset<long long> C;
1136
1137   //
1138   // setup random number generator so tree will
1139   // be relatively balanced given insertion of
1140   // random numbers:
1141   //
1142   std::random_device rd;
1143   std::mt19937 gen(rd());
1144   std::uniform_int_distribution<long long> distrib(1, N_STRESS * 100);  // inclusive
1145
1146   vector<long long> V;  // collect a few values for searching:
1147   int count = 0;
1148
1149   while (S.size() != N_STRESS) {
1150
1151     long long x = distrib(gen);
1152
1153     S.insert(x);
1154     C.insert(x);
1155
1156     count++;
1157     if (count == 1000) { // save every 1,000th value:
1158
1159       V.push_back(x);
1160       count = 0;
1161     }
1162   }
1163
1164   ASSERT_EQ(S.size(), N_STRESS);
1165
1166   for (auto x : V) {
1167     ASSERT_TRUE(S.contains(x));
1168   }
1169
1170   ASSERT_FALSE(S.contains(0));
1171   ASSERT_FALSE(S.contains(-1));
1172
1173   //
```

```
1174    // now let's compare our multiset to C++ set:
1175    //
1176    V.clear();
1177    V = S.toVector();
1178
1179    ASSERT_EQ(V.size(), C.size());
1180    ASSERT_EQ(S.size(), (int) C.size());
1181
1182    int i = 0;
1183
1184    for (auto x : C) {
1185      ASSERT_EQ(V[i], x);
1186      i++;
1187    }
1188  }
```