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-exitcoc


==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
    NODE* Left;
    NODE* Right;

  public:
    // constructor:
    NODE(TKey key)
      : isThreaded(false), Left(nullptr), Right(nullptr)
    {
      this->Keys.push_back(key);
    }

    // getters:
    TKey         get_RepresentativeKey() { return this->Keys[0]; }
    vector<TKey> get_Keys() { return this->Keys; }
    bool         get_isThreaded() { return this->isThreaded; }
    NODE*        get_Left() { return this->Left; }

    // NOTE: this ignores the thread, call to perform "normal" traversals
    NODE* get_Right() {
      if (this->isThreaded)
        return nullptr;
      else
        return this->Right;
    }

    // always gets the node to the right
    NODE* get_Thread() {
      return this->Right;
    }

    // setters:
    void set_isThreaded(bool threaded) { this->isThreaded = threaded; }
    void set_Left(NODE* left) { this->Left = left; }
    void set_Keys(vector<TKey> keys) { this->Keys = keys; }
    void set_Right(NODE* right) { this->Right = right; }
    void addToKeys(TKey key) { this->Keys.push_back(key); }
  };


  // ##################################################################
  //
  // set data members:
  //
  NODE* Root; // pointer to root node
  int Size;   // # of nodes in tree


  // ##################################################################
  //
  // set methods:
  //
public:
  //
  // default constructor:
  //
  multiset()
    : Root(nullptr), Size(0)
  { }

  //
  // copy constructor:
  //
private:
  void _copy(NODE* other)
  {
    if (other == nullptr)
      return;
    else {
      //
      // we make a copy using insert so that threads
      // are recreated properly in the copy:
      //
      vector<TKey> keys = other->get_Keys();
      for (TKey key : keys) {
        this->insert(key);
      }

      // code from the age of vanilla sets
      // this->insert(other->get_RepresentativeKey());

      _copy(other->get_Left());
      _copy(other->get_Right());
    }
  }

public:
  multiset(const multiset& other)
    : Root(nullptr), Size(0)
  {
    _copy(other.Root);
  }

  //
  // destructor:
  //
private:
  void _destroy(NODE* cur)
  {
    if (cur == nullptr)
      ;
    else {
      _destroy(cur->get_Left());
      _destroy(cur->get_Right());
      delete cur;
    }
  }

public:
  ~multiset()
  {
    //
    // NOTE: this is commented out UNTIL you are ready. The last
    // step is to uncomment this and check for memory leaks.
    //
    _destroy(this->Root);
  }

  //
  // size
  //
  // Returns # of elements in the multiset
  //
  int size()
  {
    return this->Size;
  }

  //
  // contains
  //
  // Returns true if multiset contains key, false if not
  //
private:
  bool _contains(NODE* cur, TKey key)
  {
    if (cur == nullptr)
      return false;
    else {

      if (key < cur->get_RepresentativeKey()) // search left:
        return _contains(cur->get_Left(), key);
      else if (cur->get_RepresentativeKey() < key) // search right:
        return _contains(cur->get_Right(), key);
      else // must be equal, found it!
        return true;
    }
  }
```

```

public:
  bool contains(TKey key)
  {
    return _contains(this->Root, key);
  }

  int count(TKey key) {
    NODE* cur = this->Root;

    while (cur != nullptr) {
      if (key < cur->get_RepresentativeKey()) {
        cur = cur->get_Left();
      }
      else if (cur->get_RepresentativeKey() < key) {
        cur = cur->get_Right();
      }
      else {
        return cur->get_Keys().size();
      }
    }
    return 0;
  }

  //
  // insert
  //
  // Inserts the given key into the multiset.
  //
  void insert(TKey key)
  {
    NODE* parent = nullptr;
    NODE* cur = this->Root;

    //
    // 1. Search for key, return if found:
    //
    while (cur != nullptr) {
      if (key < cur->get_RepresentativeKey()) { // left:
        parent = cur;
        cur = cur->get_Left();
      }
      else if (cur->get_RepresentativeKey() < key) { // right:
        parent = cur;
        cur = cur->get_Right();
      }
      else { // must be equal => already in tree
        cur->addToKeys(key);
        this->Size++;
        return;
      }
    }

    //
    // 2. If not found, insert where we
    //    fell out of the tree:
    //
    NODE* n = new NODE(key);

    if (parent == nullptr) {
      //
      // tree is empty, insert at root:
      //
      this->Root = n;
    }
    else if (key < parent->get_RepresentativeKey()) {
      //
      // we are to the left of our parent:
      //
      parent->set_Left(n);
      n->set_isThreaded(true);
      n->set_Right(parent);
    }
    else {
      //
      // we are to the right of our parent:
      //
      // inherit parent's thread
      n->set_isThreaded(parent->get_isThreaded());
      n->set_Right(parent->get_Thread());
      // update parent's thread
      parent->set_isThreaded(false);
      parent->set_Right(n);
    }

    //
    // STEP 3: update size and return
    //
    this->Size++;
    return;
  }

  //
  // toPairs
  //
  // Returns pairs of elements: <element, threaded element>.
  // If a node is not threaded: <element, no_element value>.
  //
```

```
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,
  // in a vector.
  //
private:
  void _toVector(NODE* cur, std::vector<TKey>& V) {
    if (cur == nullptr)
      return;
    else {
      //
      // we want them in order, so go left, then
      // middle, then right:
      //
      _toVector(cur->get_Left(), V);

      // old code for no duplicates:
      // V.push_back(cur->get_Key()); (should be cur->get_RepresentativeKey() though)

      // append keys to V
      vector<TKey> keys = cur->get_Keys();
      V.insert(V.end(), keys.begin(), keys.end());
      _toVector(cur->get_Right(), V);
    }
  }

public:
  std::vector<TKey> toVector()
  {
    std::vector<TKey> V;

    _toVector(this->Root, V);

    return V;
  }


  // ##################################################################
  //
  // class iterator:
  //
private:
  class iterator
  {
  private:
    NODE* Ptr;
    int KeyIdx;

  public:

    iterator(NODE* ptr) : Ptr(ptr), KeyIdx(0) {}

```

```cpp
    //
    // !=
    //
    // Returns true if the given iterator is not equal to
    // this iterator.
    //
    bool operator!=(iterator other) {
      return (this->Ptr != other.Ptr || this->KeyIdx != other.KeyIdx);
    }

    //
    // ++
    //
    // Advances the iterator to the next ordered element of
    // the multiset; if the iterator cannot be advanced, ++ has
    // no effect.
    //
    void operator++() {
      // if vector of keys still not exhausted
      if (this->KeyIdx + 1 < (int) this->Ptr->get_Keys().size()) {
        this->KeyIdx++;
        return;
      }

      // else
      this->KeyIdx = 0;
      if (Ptr->get_isThreaded()) {
        this->Ptr = Ptr->get_Thread();
      } else {
        this->Ptr = leftmost(Ptr->get_Right());
      }
      return;
    }

    //
    // *
    //
    // Returns the key denoted by the iterator; this
    // code will throw an out_of_range exception if
    // the iterator does not denote an element of the
    // multiset.
    //
    TKey operator*()
    {
      if (this->Ptr == nullptr)
        throw std::out_of_range("multiset::iterator:operator*");

      return this->Ptr->get_Keys()[this->KeyIdx];
    }
    //
    // ==
    //
    // Returns true if the given iterator is equal to
    // this iterator.
    //
    bool operator==(iterator other)
    {
      return (this->Ptr == other.Ptr && this->KeyIdx == other.KeyIdx);
    }
  };

private:
  // helper function to return leftmost child of subtree
  static NODE* leftmost(NODE* root) {
    NODE* current = root;
    while (current && current->get_Left()) {
      current = current->get_Left();
    }
    return current;
  }
  static NODE* rightmost(NODE* root) {
    NODE* current = root;
    while (current && current->get_Right()) {
      current = current->get_Right();
    }
    return current;
  }

public:

  iterator begin() const {
    return iterator(leftmost(this->Root));
  }

  // ##################################################################
  //
  // find:
  //
  // If the multiset contains key, then an iterator denoting this
  // element is returned. If the multiset does not contain key,
  // then multiset.end() is returned.
  //
public:
  iterator find(TKey key)
  {
    NODE* cur = this->Root;

    while (cur != nullptr) {
      if (key < cur->get_RepresentativeKey()) { // search left:
        cur = cur->get_Left();
      }
      else if (cur->get_RepresentativeKey() < key) { // search right:
        cur = cur->get_Right();
      }
      else { // must be equal, found it!
        return iterator(cur);
      }
    }

    // if get here, not found
    return iterator(nullptr);
  }

  void erase(TKey key) {
    NODE* parent = nullptr;
    NODE* cur = this->Root;

    //
    // 1. Search for key, set cur to it if found:
    //
    while (cur != nullptr) {
      if (key < cur->get_RepresentativeKey()) { // left:
        parent = cur;
        cur = cur->get_Left();
      }
      else if (cur->get_RepresentativeKey() < key) { // right:
        parent = cur;
        cur = cur->get_Right();
      }
      else { // must be equal => already in tree
        break;
      }
    }

    //
    // 2. If found, delete cur:
    //

    if (parent == nullptr || cur == nullptr) {
      //
      // tree is empty or not found, return:
      //
      return;
    }

    // else
    this->Size -= cur->get_Keys().size();

    NODE* new_child = nullptr;

    if (!cur->get_Left() && !cur->get_Right()) {
      // cur has no children
      // child remains nullptr
    }
    else if (cur->get_Left() && !cur->get_Right()) {
      // left child but no right child
      NODE* rightmost_of_left_subtree = rightmost(cur->get_Left());
      // set thread to parent of cur
      rightmost_of_left_subtree->set_Right(cur->get_Thread());
      new_child = cur->get_Left();
    }
    else if (!cur->get_Left() && cur->get_Right()) {
      // right child but no left child
      new_child = cur->get_Right();
    }
    else {
      // both left and right children

      // find leftmost of right subtree (cur_r_st)
      NODE* leftmost_of_right_subtree = leftmost(cur->get_Left());
      cur->set_Keys(leftmost_of_right_subtree->get_Keys());
      erase(leftmost_of_right_subtree->get_RepresentativeKey());
    }

    if (key < parent->get_RepresentativeKey()) {
      // we are to the left of our parent

      parent->set_Left(new_child);
      delete cur;
    }
    else {
      // right of parent

      parent->set_Right(new_child);
      delete cur;
    }

    //
    // STEP 3: return
    //
    return;
  }

  //
  // end:
```

```
  //
  // Returns an iterator to the end of the iteration space,
  // i.e. to no element. In other words, if your iterator
  // == multiset.end(), then you are not pointing to an element.
  //
  iterator end()
  {
    return iterator(nullptr);
  }

};

```

tests.cpp

Download

```
/*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}
  };

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

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

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

  S.insert("Angola");

  ASSERT_EQ(S.size(), 1);

  auto pairs = S.toPairs("FIN");

  vector<pair<string, string>> expected = { {"Angola", "FIN"} };

  ASSERT_EQ(pairs.size(), expected.size());
  ASSERT_EQ(pairs[0], expected[0]);
}

TEST(mymultiset, toPairs_multiple_string_elements)
{
  multiset<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(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;

  for (int x : S)
  {
    elements.push_back(x);
  }

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

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

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

  std::vector<int> expected = {21, 22, 20, 19, 23, 21, 21, 23};
  sort(expected.begin(), expected.end());

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

  std::vector<int> elements;

  for (int x : S)
  {
    elements.push_back(x);
  }

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

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

TEST(mymultiset, foreach_with_single_string)
{
  multiset<string> S;
  S.insert("Hello");
  int count = 0;
  string value;

  for (const string& x : S)
  {
    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;
  for (int i = 0; i < N_STRESS; ++i) {
    S.insert(i);
  }

  // check multiset size
  ASSERT_EQ(S.size(), N_STRESS);

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

  ASSERT_EQ(count, N_STRESS);
}

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

  // 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, 42};
  auto gen = [&dist, &mersenne_engine](){
    return dist(mersenne_engine);
  };
  vector<int> expected(N_STRESS);
  generate(begin(expected), end(expected), gen);

  // populate multiset
  for (int i = 0; i < N_STRESS; ++i) {
    S.insert(expected[i]);
  }

  std::sort(expected.begin(), expected.end());

  // code to remove duplicates (no longer applies)
  //
  // std::set<int> s;
  // unsigned size = sorted_expected.size();
  // for( unsigned i = 0; i < size; ++i ) s.insert( sorted_expected[i] );
  // sorted_expected.assign( s.begin(), s.end() );

  // check multiset size
  ASSERT_EQ(S.size(), (int) expected.size());

```

```cpp
  // check values
  int count = 0;
  for (int x : S) {
    ASSERT_EQ(x, expected[count]);
    count++;
  }
}

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

  // 61 is repeated (3 instances)
  // 11 is repeated (2 instances)
  vector<int> V = { 22, 11, 49, 61, 3, 19, 35, 11, 61, 30, 41, 61 };

  for (auto x : V)
    S.insert(x);

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

  vector<int> V2 = S.toVector();

  ASSERT_EQ(V2.size(), V.size());

  std::sort(V.begin(), V.end());

  //
  // V and V2 should have the same elements in
  // the same order:
  //
  auto iterV = V.begin();
  auto iterV2 = V2.begin();

  while (iterV != V.end()) {
    ASSERT_EQ(*iterV, *iterV2);

    iterV++;
    iterV2++;
  }
}

TEST(mymultiset, find_should_return_first_inserted_key)
{
  multiset<Movie> movies;

  movies.insert( Movie("The Matrix", 603, 1999) );
  movies.insert( Movie("Toy Story", 862, 1995) );
  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);
  }

  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_left_children_duplicates)
{
  multiset<int> S;

  vector<int> before = {10, 9, 9, 8, 7, 7, 7, 5, 5, 5, 5, 4, 3};
  vector<int> after = {10, 9, 9, 8, 7, 7, 7, 4, 3};

  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_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;

  vector<int> before = {10, 11, 12, 12, 12, 12, 15, 15, 15, 16, 17, 18};
  vector<int> after = {10, 11, 12, 12, 12, 12, 16, 17, 18};
  int to_erase = 15;

  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(to_erase);

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

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

  vector<int> before = {1, 4, 5, 3, 2};
  vector<int> after = {1, 2, 3, 4};
  int to_erase = 4;

  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(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");

  ASSERT_EQ(S.size(), 4);

  ASSERT_TRUE(S.contains("pear"));
  ASSERT_TRUE(S["banana"]);
  ASSERT_TRUE(S.contains("chocolate"));
  ASSERT_TRUE(S["apple"]);

  ASSERT_FALSE(S.contains("Apple"));
  ASSERT_FALSE(S["carmel"]);
  ASSERT_FALSE(S.contains("appl"));
  ASSERT_FALSE(S["chocolatee"]);
}

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

  ASSERT_EQ(S.size(), 0);

  Movie Sleepless("Sleepless in Seattle", 123, 123456789.00);
  S.insert(Sleepless);

  Movie Matrix("The Matrix", 456, 400000000.00);
  S.insert(Matrix);

  Movie AnimalHouse("Animal House", 789, 1000000000.00);
  S.insert(AnimalHouse);

  ASSERT_EQ(S.size(), 3);

  vector<Movie> V = S.toVector();

  ASSERT_EQ(V[0].Title, "Animal House");
  ASSERT_EQ(V[1].Title, "Sleepless in Seattle");
  ASSERT_EQ(V[2].Title, "The Matrix");
}

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

  vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };

  for (auto x : V)
    S.insert(x);

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

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

  ASSERT_FALSE(S.contains(0));
  ASSERT_FALSE(S[0]);
  ASSERT_FALSE(S.contains(2));
  ASSERT_FALSE(S[2]);
  ASSERT_FALSE(S.contains(4));
  ASSERT_FALSE(S[4]);
  ASSERT_FALSE(S.contains(29));
  ASSERT_FALSE(S[31]);
  ASSERT_FALSE(S.contains(40));
  ASSERT_FALSE(S[42]);
}

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

  vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };

  for (auto x : V)
    S.insert(x);

  // try to insert them all again:
  for (auto x : V)
    S.insert(x);

  // should be twice the size since all numbers reinserted
  ASSERT_EQ(S.size(), 2 * (int) V.size());

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

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

  vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };

  for (auto x : V)
    S.insert(x);

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

  vector<int> V2 = S.toVector();

  ASSERT_EQ(V2.size(), V.size());

  std::sort(V.begin(), V.end());

  //
  // V and V2 should have the same elements in
  // the same order:
  //
  auto iterV = V.begin();
  auto iterV2 = V2.begin();

  while (iterV != V.end()) {
    ASSERT_EQ(*iterV, *iterV2);

    iterV++;
    iterV2++;
  }
}

TEST(mymultiset, copy_empty)
{
  multiset<int> S1;

  {
    //
    // create a new scope, which will trigger destructor:
    //
    multiset<int> S2 = S1;  // this will call copy constructor:

    S1.insert(123);  // this should have no impact on S2:
    S1.insert(100);
    S1.insert(150);

    ASSERT_EQ(S2.size(), 0);

    vector<int> V2 = S2.toVector();

    ASSERT_EQ((int) V2.size(), 0);
  }
}

TEST(mymultiset, copy_constructor)
{
  multiset<int> S1;

  vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };

  for (auto x : V)
    S1.insert(x);

  ASSERT_EQ(S1.size(), (int) V.size());

  {
    //
    // create a new scope, which will trigger destructor:
    //
    multiset<int> S2 = S1;  // this will call copy constructor:

    S1.insert(123);  // this should have no impact on S2:
    S1.insert(100);
    S1.insert(150);

    ASSERT_EQ(S2.size(), (int) V.size());

    vector<int> V2 = S2.toVector();

    ASSERT_EQ(V2.size(), V.size());

    std::sort(V.begin(), V.end());

    //
    // V and V2 should have the same elements in
    // the same order:
    //
    auto iterV = V.begin();
    auto iterV2 = V2.begin();

    while (iterV != V.end()) {
      ASSERT_EQ(*iterV, *iterV2);

      iterV++;
      iterV2++;
    }

    S2.insert(1000);  // this should have no impact on S1:
    S2.insert(2000);
    S2.insert(3000);
    S2.insert(4000);
    S2.insert(5000);

    V.push_back(123);
    V.push_back(100);
    V.push_back(150);
  }

  //
  // the copy was just destroyed, the original multiset
  // should still be the same as it was earlier:
  //
  ASSERT_EQ(S1.size(), (int) V.size());

  vector<int> V2 = S1.toVector();

  ASSERT_EQ(V2.size(), V.size());

  std::sort(V.begin(), V.end());

  //
  // V and V2 should have the same elements in
  // the same order:
  //
  auto iterV = V.begin();
  auto iterV2 = V2.begin();

  while (iterV != V.end()) {
    ASSERT_EQ(*iterV, *iterV2);

    iterV++;
    iterV2++;
  }
}

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

  auto iter = S.find(22);
  ASSERT_TRUE(iter == S.end());
}

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

  S.insert(1234);

  auto iter = S.find(123);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(1234);
  if (iter == S.end()) {  // this should not happen:
    ASSERT_TRUE(false); // fail:
  }

  ASSERT_EQ(*iter, 1234);

  iter = S.find(1235);
  ASSERT_TRUE(iter == S.end());
}

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

  vector<int> V = { 22, 11, 49, 3, 19, 35, 61, 30, 41 };

  for (auto x : V)
    S.insert(x);

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

  //
  // make sure we can find each of the values we inserted:
  //
  for (auto x : V) {
    auto iter = S.find(x);

    if (iter == S.end()) {  // this should not happen:
      ASSERT_TRUE(false); // fail:
    }

    ASSERT_EQ(*iter, x);
  }

  //
  // these searches should all fail:
  //
  auto iter = S.find(0);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(-1);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(1);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(4);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(34);
  ASSERT_TRUE(iter == S.end());

  iter = S.find(36);
  ASSERT_TRUE(iter == S.end());
```

```
}

// instructor's stress test
// modified to use my N_STRESS instead of given N
// this makes quick testing easier

TEST(mymultiset, stress_test)
{
  multiset<long long> S;
  // edited to create a multiset, not a set
  std::multiset<long long> C;

  //
  // setup random number generator so tree will
  // be relatively balanced given insertion of
  // random numbers:
  //
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<long long> distrib(1, N_STRESS * 100);  // inclusive

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

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

    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_STRESS);

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

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

  //
  // now let's compare our multiset 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++;
  }
}
```