Assignment 2 Grade Report (Page 1 of 1)

Name: *Maldonado_Robert* Due: 09/15/2021 (11:59 pm)
Received: on-time 09/ /2021 (:)

Remarks: compilation: compiled OK / ~~compiled with warning(s) / wouldn't compile~~
test w/ `a2test.in`: ~~OK~~ ~~different @ places (all are just different element-ordering → OK)~~
code check: (see *penalty tally* below) "Invalid next size" 5

Score: (100 − 17 (from below)) / 100 = 83 / 100

Not Heeding Style Guide (up to 30 points total penalty)

- *Naming Convention:* (penalty - up to 3 per type of violation)
  - not using meaningful name
  - not starting a variable name in lowercase
  - not using separating underscore(s) or camel style in multi-word name
  - name of constant not in all-uppercase or name of variable in all-uppercase
  - function not doing, or doing more than, what it's name suggests
- Other:
- *Readability Woes:* (penalty - depends on severity)
  - poor indentation
  - poor spacing
  - poor alignment
  - lines wrap/cut off
  - using `l` or `o` (looks like `1` or `0`) as variable name
  - not using `monospaced` font

General Shortcomings

- Code in hardcopy not gibing with code in softcopy. [depends on severity]
- Leaving behind irrelevant comments, debugging code, *etc.* [½ 1 1½ 2]
- Removing as-provided documentation (*esp.* class invariant) at top of *IntSet.cpp* [1]
- Declaring a pointer and leaving it uninitialized - dangling pointer. [½ 1]

Function Specific Penalties

- `void IntSet::resize(int new_capacity)`
  - Not (or not correctly) adjusting `new_capacity` where appropriate (to be in line w/ postcondition) [½ 1 1½ 2]
  - Not freeing up "old" dynamic memory (thus incurring *memory leak*) [2]
  - Out-of-bound (in general) traversing array(s) [2]
- `IntSet::IntSet(int initial_capacity = DEFAULT_CAPACITY)`
  - Not using *initializer list* [½]
  - Not (or not correctly) trapping invalid `initial_capacity` [½ 1]
  - Not observing *class invariant* [1]
  - Not setting `used` [2½]
- `IntSet::IntSet(const IntSet& src)`
  - Not using initializer list [½]
- `IntSet::~IntSet()`
- `IntSet& IntSet::operator=(const IntSet& rhs)`
  - Not trapping self-assignment [1½ if won't lead to problems on self-assignment, 5 otherwise]
  - Not freeing up "old" dynamic memory (thus incurring *memory leak*) [2]
- `bool IntSet::size() const` 2
- `bool IntSet::isEmpty() const` 2
  - Unnecessarily traversing/processing array (algorithmically correct or otherwise) [1½]
- `int IntSet::contains(int anInt) const`
  - Traversing *entire* array and using an algorithm not in line with *class invariant* [2]
  - Out-of-bound (in general) traversing array(s) [2]
  - Logic error (role reversal, *etc.*) [2]
- `bool IntSet::isSubsetOf(const IntSet& otherIntSet) const`
  - Unnecessarily traversing/processing array (algorithmically correct or otherwise) [1½]
  - Out-of-bound (in general) traversing array(s) [2]
  - Logic error (role reversal, *etc.*) [2]
- `IntSet IntSet::unionWith(const IntSet& otherIntSet) const`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4]
- `IntSet IntSet::intersect(const IntSet& otherIntSet) const`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4]
- `IntSet IntSet::subtract(const IntSet& otherIntSet) const`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4]
- `void IntSet::reset()`
  - Not observing *class invariant* [1]
  - Not setting `used` [2]
- `bool IntSet::add(int anInt)`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7 7½ 8]
- `bool IntSet::remove(int anInt)`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4]
- `bool operator==(const IntSet& is1, const IntSet& is2)`
  - Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4]

Other Issues

```
// FILE: IntSet.cpp - header file for IntSet class
//       Implementation file for the IntStore class
//       (See IntSet.h for documentation.)
// INVARIANT for the IntSet class:
// (1) Distinct int values of the IntSet are stored in a 1-D,
//     dynamic array whose size is stored in member variable
//     capacity; the member variable data references the array.
// (2) The distinct int value with earliest membership is stored
//     in data[0], the distinct int value with the 2nd-earliest
//     membership is stored in data[1], and so on.
//     Note: No "prior membership" information is tracked; i.e.,
//           if an int value that was previously a member (but its
//           earlier membership ended due to removal) becomes a
//           member again, the timing of its membership (relative
//           to other existing members) is the same as if that int
//           value was never a member before.
//     Note: Re-introduction of an int value that is already an
//           existing member (such as through the add operation)
//           has no effect on the "membership timing" of that int
//           value.
// (4) The # of distinct int values the IntSet currently contains
//     is stored in the member variable used.
// (5) Except when the IntSet is empty (used == 0), ALL elements
//     of data from data[0] until data[used - 1] contain relevant
//     distinct int values; i.e., all relevant distinct int values
//     appear together (no "holes" among them) starting from the
//     beginning of the data array.
// (6) We DON'T care what is stored in any of the array elements
//     from data[used] through data[capacity - 1].
//     Note: This applies also when the IntSet is empry (used == 0)
//           in which case we DON'T care what is stored in any of
//           the data array elements.
//     Note: A distinct int value in the IntSet can be any of the
//           values an int can represent (from the most negative
//           through 0 to the most positive), so there is no
//           particular int value that can be used to indicate an
//           irrelevant value. But there's no need for such an
//           "indicator value" since all relevant distinct int
//           values appear together starting from the beginning of
//           the data array and used (if properly initialized and
//           maintained) should tell which elements of the data
//           array are actually relevant.
//
// DOCUMENTATION for private member (helper) function:
//   void resize(int new_capacity)
//     Pre:  (none)
//           Note: Recall that one of the things a constructor
//                 has to do is to make sure that the object
//                 created BEGINS to be consistent with the
//                 class invariant. Thus, resize() should not
//                 be used within constructors unless it is at
//                 a point where the class invariant has already
//                 been made to hold true.
//     Post: The capacity (size of the dynamic array) of the
//           invoking IntSet is changed to new_capacity...
```

```
//            ...EXCEPT when new_capacity would not allow the
//            invoking IntSet to preserve current contents (i.e.,
//            value for new_capacity is invalid or too low for the
//            IntSet to represent the existing collection),...
//            ...IN WHICH CASE the capacity of the invoking IntSet
//            is set to "the minimum that is needed" (which is the
//            same as "exactly what is needed") to preserve current
//            contents...
//            ...BUT if "exactly what is needed" is 0 (i.e. existing
//            collection is empty) then the capacity should be
//            further adjusted to 1 or DEFAULT_CAPACITY (since we
//            don't want to request dynamic arrays of size 0).
//            The collection represented by the invoking IntSet
//            remains unchanged.
//            If reallocation of dynamic array is unsuccessful, an
//            error message to the effect is displayed and the
//            program unconditionally terminated.

#include "IntSet.h"
#include <iostream>
#include <cassert>
using namespace std;

void IntSet::resize(int new_capacity)
{
    if (new_capacity < used)
    {
      new_capacity = used;
    }
    if (new_capacity < 1)
    {
      new_capacity = 1;
    }

    capacity = new_capacity;
    int * newData = new int[capacity];

    for (int i = 0; i < used; ++i)
    {
      newData[i] = data[i];
    }

    delete [] data;
    data = newData;
}

IntSet::IntSet(int initial_capacity) : capacity(initial_capacity), used(0)
{
  if (capacity < 1)
  {
     capacity = DEFAULT_CAPACITY;
  }
  data = new int[capacity];
}
```

```
IntSet::IntSet(const IntSet& src) : capacity(src.capacity), used(src.used)
{
  data = new int[capacity];
  for(int i = 0; i < used; i++)
  {
    data[i] = src.data[i];
  }
}


IntSet::~IntSet()
{
   delete[] data;
}

IntSet& IntSet::operator=(const IntSet& rhs)
{
  if (this != &rhs)
  {
     int* newData = new int[rhs.capacity];
     for (int i = 0; i < rhs.used; ++i)
     {
        newData[i] = rhs.data[i];
     }
     delete [] data;
     data = newData;
     capacity = rhs.capacity;
     used = rhs.used;
  }
  return *this;
}

int IntSet::size() const
{
  int items = 0;
  for(int i = 0; i < DEFAULT_CAPACITY; i++)
  {
       if(data[i] > 0 )
       {
         items++;
       }
  }
  return items;
}

bool IntSet::isEmpty() const
{
    int items = 0;
    for(int i = 0; i < DEFAULT_CAPACITY; i++)
    {
        if(data[i] > 0 )
        {
          items++;
        }
    }
```

not to invariant(2)

```
   if(items > 0)
   {
     return false;
   }
   else
   {
     return true;
   }
}

bool IntSet::contains(int anInt) const
{
  for(int i = 0; i < used; i++)
  {
      if(data[i] == anInt)
      {
        return true;
      }
  }
  return false;
}

bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
{
    int counter = 0;
    IntSet newSet = (*this);
    for(int i = 0; i < newSet.used; i++)
    {
      if(otherIntSet.contains(newSet.data[i])==true)
      {
        counter ++;
      }
    }
    if(counter == used)
    {
      return true;
    }
    else
      return false;
}

void IntSet::DumpData(ostream& out) const
{  // already implemented ... DON'T change anything
   if (used > 0)
   {
      out << data[0];
      for (int i = 1; i < used; ++i)
         out << "  " << data[i];
   }
}

IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
  IntSet newSet = (*this);
  for(int j = 0; j < otherIntSet.used; j++)
```

```
  {
    newSet.data[newSet.used] = otherIntSet.data[j];
    newSet.used++;
  }
  return newSet;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
  IntSet newSet = (*this);
  for(int j = 0; j < size(); j++)
  {
     if(otherIntSet.contains(data[j]) != true)
     {
       newSet.remove(data[j]);
     }
  }
  return newSet;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
   IntSet set = (*this);
   for(int i = 0; i < used; i++)
   {
     if(otherIntSet.contains(data[i]) == true)
     {
        set.remove(data[i]);
     }
  }
  return set;
}

void IntSet::reset()
{
   used = 0;
}

bool IntSet::add(int anInt)
{
    if(contains(anInt) == false)
    {
      if(used > capacity)
      {
        resize(int(1.5*capacity) + 1);
      }
      data[used] = anInt;
      used++;
      return true;
    }
    return false;
}

bool IntSet::remove(int anInt)
{
```

what if not enough capacity

don't want to wait until capacity exceeded to resize

```
    if(contains(anInt) == true)
    {
      for(int i = 0; i < used; i++)
      {
        if(data[i] == anInt)
        {
          data[i] = data[i+1];
          data[i+1] = anInt;
        }
      }
      used--;
      return true;
    }
    else
    {
      return false;
    }
}

bool operator==(const IntSet& is1, const IntSet& is2)
{
  if(is1.isSubsetOf(is2) == true && is2.isSubsetOf(is1) == true)
  {
    return true;
  }
  return false;
}
```

need to shift some items to left (by 1 spot)