```
// Austin Keith Faulkner: a_f408
// Date: 9/18/2019
//
// 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 new_capacity is less than used, set new_capacity to used.
   if (new_capacity < used)
      new_capacity = used;

   // If new_capacity is less than 1, then set new_capacity to
   // DEFAULT_CAPACITY, which is 1.
   if (new_capacity < DEFAULT_CAPACITY)
      new_capacity = DEFAULT_CAPACITY;

   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 is 0 or negative, set capacity to DEFAULT_CAPACITY.
   if (capacity < DEFAULT_CAPACITY)
      capacity = DEFAULT_CAPACITY;

   // Dynamically allocate an array data.
   data = new int[capacity];
}

IntSet::IntSet(const IntSet& src)
   : capacity(src.capacity), used(src.used)
{
   data = new int[capacity];

   // Perform a deep copy of the IntSet array.
   for (int i = 0; i < used; ++i)
      data[i] = src.data[i];
}

IntSet::~IntSet()
{
   // Delete contents of dynamic, run-time array and prevent memory
   // leaks.
   delete [] data;

   // Prevent the array data from being a stale pointer
   data = nullptr;
}

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 { return used; }
```

```
bool IntSet::isEmpty() const
{
   // If used == 0, then the set is empty; and so, return true.
   // Otherwise, return false: the set is not empty.
   if (used == 0)
      return true;
   else
      return false;
}

bool IntSet::contains(int anInt) const
{
   // It the set is not empty, loop through the used elements and
   // determine whether anInt is in the set. If so, return true;
   // Otherwise, return false.
   if (!isEmpty())
   {
      for (int i = 0; i < used; ++i)
      {
         if (anInt == data[i])
            return true;
      }
   }
   return false;
}

bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
{
   IntSet intSet = *this;  // local IntSet initialized to a
                           // copy of the invoking IntSet

   if(intSet.isEmpty())
   {
      // Check to see if the invoking IntSet
      // is empty. If it is, then it is a subset of any IntSet.
      // Therefore, return true.
      return true;
   }
   else
   {
      // Check otherIntSet against intSet data, up to intSet size.
      // Determine whether or not the otherIntSet contains all the
      // elements of intSet. If it does not, then intSet is not a
      // subset of the otherIntSet: return false; otherwise, return
      // true: otherIntSet contains intSet.
      for (int i = 0; i < intSet.used; ++i)
      {
         if(!otherIntSet.contains(intSet.data[i]))
            return false;
      }
   }
   return true;
```

```
}

void IntSet::DumpData(ostream& out) const
{
   // Display the element data for an IntSet
   if (used > 0)
   {
      out << data[0];
      for (int i = 1; i < used; ++i)
         out << "  " << data[i];
   }
}

IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
   IntSet intSetUnion = *this; // local IntSet initialized to a copy
                               // of the invoking IntSet

   // Up to otherIntSet's size, if the IntSet intSetUnion does not
   // contain the otherIntSet's elements, then add them to the IntSet
   // intSetUnion. Return the IntSet intSetUnion.
   for (int i = 0; i < otherIntSet.used; ++i)
   {
      if (!intSetUnion.contains(otherIntSet.data[i]))
         intSetUnion.add(otherIntSet.data[i]);
   }
   return intSetUnion;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
   IntSet intSetIntersect = *this; // local IntSet initialized to a
                                   // copy of the invoking IntSet

   // If otherIntSet does not contain elements of data[i] up to used,
   // then remove the same elements from IntSet intSetIntersect.
   for (int i = 0; i < used; ++i)
   {
      if (!otherIntSet.contains(data[i]))
         intSetIntersect.remove(data[i]);
   }
   return intSetIntersect;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
   IntSet intSetDifference = *this;  // local IntSet initialized to a
                                     // copy of the invoking IntSet

   // Loop through otherIntSet up to its size checking whether of not
   // an element in otherIntSet is in IntSet intSetDifference. If so,
   // remove the shared element from IntSet intSetDifference.
   for (int i = 0; i < otherIntSet.used; ++i)
   {
      if (intSetDifference.contains(otherIntSet.data[i]))
         intSetDifference.remove(otherIntSet.data[i]);
   }
   return intSetDifference;
}

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

bool IntSet::add(int anInt)
{
   // Sets containing multiples of the same element are equal to a set
   // containing only one of the same element. So, below we only check
   // if anInt is not in the set. If used is greater than
   // capacity, resize the capacity to one and half plus 1 its size.
   // If anInt is not in the set, add anInt and increment used by
   // one. Return true; else, return false.
   if (!contains(anInt))
   {
      if (used > capacity)
         resize(int(1.5 * capacity) + 1);

      data[used] = anInt;
      ++used;
      return true;
   }
   return false;
}

bool IntSet::remove(int anInt)
{
   // If the set contains anInt, loop through the set; find the
   // anInt; remove it/shift the array data to the left (closing
   // the gap if there is one). Then decrement used to keep the proper
   // set size and return true. Otherwise, return false.
   if(contains(anInt))
   {
      for(int i = 0; i < used; ++i)
      {
         if(data[i] == anInt)
         {
            for(int j = i + 1; j < used; ++j)
            {
               data[j - 1] = data[j];
            }
            --used;
            return true;
         }
      }
   }
   return false;
```

```
}

bool operator==(const IntSet& is1, const IntSet& is2)
{
   // If IntSet is2 is a subset of IntSet is1 and IntSet is1 is a
   // subset of is2, then the two sets are equal. If equal,
   // return true. Otherwise, return false, indicating the two IntSets
   // are not equal.
   if (is2.IntSet::isSubsetOf(is1) && is1.IntSet::isSubsetOf(is2))
      return true;
   else
      return false;
}
```