```
// 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,
//     compile-time array whose size is IntSet::MAX_SIZE;
//     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[IntSet::MAX_SIZE - 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.

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

IntSet::IntSet() : used(0)
{
```

```
}

int IntSet::size() const
{
   return used;
}

bool IntSet::isEmpty() const
{
   return used == 0;
}

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
{
   if(isEmpty())
   {
      return true;
   }
   else
   {
      for(int i = 0; i < used; i++)
      {
         if (!otherIntSet.contains(data[i]))
         {
            return false;
         }
      }
      return true;
   }
}

void IntSet::DumpData(ostream& out) const
{
   if (used > 0)
   {
      out << data[0];
      for (int i = 1; i < used; ++i)
```

```
         out << "  " << data[i];
   }
}

IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
   assert(used + (otherIntSet.subtract(*this)).size() <= MAX_SIZE);

   IntSet returnSet = *this;
   for(int i = 0; i < otherIntSet.size(); i++)
   {
      returnSet.add(otherIntSet.data[i]);
   }
   return returnSet;

}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
   IntSet returnSet = *this;

   for (int i = 0; i < used; i++)
   {
      if (!otherIntSet.contains(data[i]))
      {
         returnSet.remove(data[i]);
      }
   }
   return returnSet;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
   IntSet returnSet = *this;

   for(int i = 0; i < used; i++)
   {
      if (otherIntSet.contains(data[i]))
      {
         returnSet.remove(data[i]);
      }
   }
   return returnSet;
}

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

```
bool IntSet::add(int anInt)
{
   assert(contains(anInt) ? size() <= MAX_SIZE : size() < MAX_SIZE);

   if(contains(anInt))
   {
      return false;
   }
   else
   {
      data[used] = anInt;
      used++;
      return true;
   }
}

bool IntSet::remove(int anInt)
{
   if(!contains(anInt))
   {
      return false;
   }
   else
   {
      for(int i = 0; i < used; i++)
      {
         if(data[i] == anInt)
         {
            for(int j = i; j < used-1; j++)
            {
               data[j] = data[j+1];
            }
            used--;
            return true;
         }
      }
   }
   return false;
}

bool equal(const IntSet& is1, const IntSet& is2)
{
  return is1.isSubsetOf(is2) && is2.isSubsetOf(is1);
}
```