```
// 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()
```

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

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

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

   return false;
}

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
{
   //tracks shared elements
   int sharedElm = 0;

   //returns true if IntSet is empty
   if (used == 0)
   {
        return true;
   }

   //returns false if invoking IntSet is larger than otherIntSet
   if (used > otherIntSet.used)
```

```
    {
         return false;
    }

    //traverse both arrays and itterates sharedElm
    for (int i = 0; i < used; i++)
    {
        for (int j = 0; j < otherIntSet.used; j++)
        {
            if (data[i] == otherIntSet.data[j])
            {
                sharedElm++;
            }
        }
    }

    //checks if all elemnts of IntSet are shared elements
    if (sharedElm == 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 tempIntSet;

     //populates the tempIntSet
     for (int i = 0; i < used; i++)
     {
         tempIntSet.data[i] = this->data[i];
         tempIntSet.used = this->used;
     }

```

```cpp
    //adds elements from otherIntSet to tempIntSet
    for (int i = 0; i < otherIntSet.used; i++)
    {
        if (!tempIntSet.contains(otherIntSet.data[i]))
        {
            tempIntSet.data[used] = otherIntSet.data[i];
            tempIntSet.used++;
        }

    }

    return tempIntSet;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
    IntSet tempIntSet;

    for (int i = 0; i < used; i++)
    {
        for (int j = 0; j < otherIntSet.used; j++)
        {
            if (data[i] == otherIntSet.data[j])
            {
                tempIntSet.data[tempIntSet.used] = otherIntSet.data[j];
                tempIntSet.used++;
            }
        }
    }

    return tempIntSet;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
    IntSet tempIntSet;

    //populates tempIntSet
    for (int i = 0; i < used; i++)
    {
        tempIntSet.data[i] = this->data[i];
        tempIntSet.used = this->used;
    }

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

    return tempIntSet;
}

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

bool IntSet::add(int anInt)
{
   //checks if this->data[] is full
   if (used == MAX_SIZE)
   {
       return false;
   }

   if (!this->contains(anInt))
   {
       data[used] = anInt;
       used++;
       return true;
   }

   return false;
}

bool IntSet::remove(int anInt)
{
    if (this->contains(anInt))
    {
        int index = 0;

        //find index of anInt
        for (int i = 0; i < used; i++)
        {
            if (data[i] == anInt)
            {
                index = i;
                break;
            }
        }

        //shift index of other elements -1
        for (index; index < used - 1; index++)
        {
            this->data[index] = data[index + 1];
        }

        used--;

        return true;
    }

    return false;
}

bool equal(const IntSet& is1, const IntSet& is2)
{
   IntSet tempIntSet;

   //checks if the IntSets are the same size
   if (is1.size() != is2.size())
   {
        return false;
   }

   tempIntSet = is1.subtract(is2);

   if (tempIntSet.size() == 0)
   {
       return true;
   }
   else
   {
       return false;
   }

}
```