```
// 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
```

```
67  //           don't want to request dynamic arrays of size 0).
68  //           The collection represented by the invoking IntSet
69  //           remains unchanged.
70  //           If reallocation of dynamic array is unsuccessful, an
71  //           error message to the effect is displayed and the
72  //           program unconditionally terminated.
73
74  #include "IntSet.h"
75  #include <iostream>
76  #include <cassert>
77  using namespace std;
78
79  void IntSet::resize(int new_capacity)
80  {
81     if (new_capacity<used){
82      new_capacity=used;
83     }
84     if (new_capacity<1){
85      new_capacity=1;
86     }
87     capacity=new_capacity;
88     int* newData= new int[capacity];
89     for(int i; i<used; ++i){
90      newData[i]=data[i];
91     }
92     delete[]data;
93     data=newData;
94  }
95
96  IntSet::IntSet(int initial_capacity): capacity(initial_capacity), used(0)
97  {
98     if (initial_capacity<1){
99      capacity=DEFAULT_CAPACITY;
100    }
101    data=new int[initial_capacity];
102 }
103
104 IntSet::IntSet(const IntSet& src):capacity(src.capacity),used(src.used)
105 {
106    data = new int [capacity];
107    for (int i=0; i < used; ++i){
108     data[i]=src.data[i];
109    }
110 }
111
112
113 IntSet::~IntSet()
114 {
115    delete [] data;
116 }
117
118 IntSet& IntSet::operator=(const IntSet& rhs)
119 {
120    if (this != &rhs){
121     int* newData=new int [rhs.capacity];
122     for (int i; i < rhs.used; ++i){
123         newData[i]=rhs.data[i];
124         }
125     delete [] data;
126     data=newData;
127     capacity=rhs.capacity;
128     used=rhs.used;
129    }
130    return *this;
131 }
132
```

```
133 int IntSet::size() const
134 {
135    return used;
136 }
137
138 bool IntSet::isEmpty() const
139 {
140    if (used==0)
141     return true;
142    else
143     return false;
144 }
145
146 bool IntSet::contains(int anInt) const
147 {
148    for(int i = 0; i < used; i++)
149    {
150       if(data[i] == anInt)
151       {
152          return true;
153       }
154    }
155    return false;
156 }
157
158 bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
159 {
160    for(int i = 0; i < used; i++)
161    {
162       if(!otherIntSet.contains(data[i]))
163       {
164          return false;
165       }
166    }
167    return true;
168 }
169
170 void IntSet::DumpData(ostream& out) const
171 {  // already implemented ... DON'T change anything
172    if (used > 0)
173    {
174       out << data[0];
175       for (int i = 1; i < used; ++i)
176          out << "  " << data[i];
177    }
178 }
179
180 IntSet IntSet::unionWith(const IntSet& otherIntSet) const
181 {
182   //  for (int i=0; i <otherIntSet.used; ++i){
183    //     add(otherIntSet.data[i]);
184   //  }
185      IntSet a;
186
187       for(int i = 0; i < used; i++)
188       {
189          a.add(data[i]);
190       }
191
192       for(int i = 0; i < otherIntSet.size(); i++)
193       {
194          {
195             a.add(otherIntSet.data[i]);
196          }
197       }
198       return a;
```

```
}

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

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

   return b;

}

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

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

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

   return c;

}

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

bool IntSet::add(int anInt)
{
   if (contains(anInt)==0){
    if (used>capacity){
        resize(int(1.5*capacity)+1);
        }
    data[used]=anInt;
    used++;
    return true;
   }
   return false;
}

bool IntSet::remove(int anInt)
{
   if(contains(anInt)){
      for(int i = 0; i < size(); i++){
         if(data[i] == anInt){
            for(int j = i; j < size() - 1; j++){
               data[j] = data[j+1];
            }
            used--;
```

```
265             return true;
266          }
267       }
268    }
269    return false;
270 }
271
272 bool operator==(const IntSet& is1, const IntSet& is2)
273 {
274    if (is1.isSubsetOf(is2)&&is2.isSubsetOf(is1))
275     return true;
276    else
277     return false;
278 }
```