```
// 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
```

```
50  //                 class invariant. Thus, resize() should not
51  //                 be used within constructors unless it is at
52  //                 a point where the class invariant has already
53  //                 been made to hold true.
54  //     Post: The capacity (size of the dynamic array) of the
55  //           invoking IntSet is changed to new_capacity...
56  //           ...EXCEPT when new_capacity would not allow the
57  //           invoking IntSet to preserve current contents (i.e.,
58  //           value for new_capacity is invalid or too low for the
59  //           IntSet to represent the existing collection),...
60  //           ...IN WHICH CASE the capacity of the invoking IntSet
61  //           is set to "the minimum that is needed" (which is the
62  //           same as "exactly what is needed") to preserve current
63  //           contents...
64  //           ...BUT if "exactly what is needed" is 0 (i.e. existing
65  //           collection is empty) then the capacity should be
66  //           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
82      //set new_capacity to the minimum if needed
83      if (new_capacity < used)
84      {
85          new_capacity = used;
86      }
87
88      //prevent array of capacity 0
89      if (new_capacity == 0)
90      {
91          new_capacity == DEFAULT_CAPACITY;
92      }
93
94
95      int* newData = new int[new_capacity];
96
97      for (int i = 0; i < used; i++)
98      {
```

```
 99          newData[i] = data[i];
100      }
101
102      //deallocate data and replace it with newData
103      delete[] data;
104      data = newData;
105
106      capacity = new_capacity;
107  }
108
109  IntSet::IntSet(int initial_capacity)
110  {
111      used = 0;
112      capacity = initial_capacity;
113      data = new int[capacity];
114  }
115
116  IntSet::IntSet(const IntSet& src)
117  {
118      used = src.used;
119      capacity = src.capacity;
120      data = new int[capacity];
121
122      //initialize all elem of data[]
123      for (int i = 0; i < capacity; i++)
124      {
125          data[i] = src.data[i];
126      }
127  }
128
129
130  IntSet::~IntSet()
131  {
132      delete[] data;
133  }
134
135  IntSet& IntSet::operator=(const IntSet& rhs)
136  {
137      used = rhs.used;
138      capacity = rhs.capacity;
139
140      //Dealocate data array
141      delete[] data;
142
143      //Initialize new data array
144      data = new int[capacity];
145
146      //repopulate this->data[] with rhs.data[]
147      for (int i = 0; i < rhs.used; i++)
```

```
148     {
149         data[i] = rhs.data[i];
150     }
151 
152    return *this;
153 }
154 
155 int IntSet::size() const
156 {
157     return used;
158 }
159 
160 bool IntSet::isEmpty() const
161 {
162     if (used == 0)
163     {
164         return true;
165     }
166 
167     return false;
168 }
169 
170 bool IntSet::contains(int anInt) const
171 {
172     for (int i = 0; i < used; i++)
173     {
174         if (data[i] == anInt)
175         {
176             return true;
177         }
178     }
179     return false;
180 }
181 
182 bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
183 {
184     //tracks shared elements
185     int sharedElm = 0;
186 
187     //returns true if IntSet is empty
188     if (used == 0)
189     {
190         return true;
191     }
192 
193     //returns false if invoking IntSet is larger than otherIntSet
194     if (used > otherIntSet.used)
195     {
196         return false;
```

```
197     }
198
199     //traverse both arrays and itterates sharedElm
200     for (int i = 0; i < used; i++)
201     {
202         for (int j = 0; j < otherIntSet.used; j++)
203         {
204             if (data[i] == otherIntSet.data[j])
205             {
206                 sharedElm++;
207             }
208         }
209     }
210
211     //checks if all elemnts of IntSet are shared elements
212     if (sharedElm == used)
213     {
214         return true;
215     }
216     else
217     {
218         return false;
219     }
220 }
221
222 void IntSet::DumpData(ostream& out) const
223 {  // already implemented ... DON'T change anything
224    if (used > 0)
225    {
226       out << data[0];
227       for (int i = 1; i < used; ++i)
228          out << "  " << data[i];
229    }
230 }
231
232 IntSet IntSet::unionWith(const IntSet& otherIntSet) const
233 {
234     //call copy constr
235     IntSet tempIntSet(*this);
236
237     //resize tempIntSet with worst case scenerio (all unique elem)
238     tempIntSet.resize(capacity + otherIntSet.capacity);
239
240     //adds elements from otherIntSet to tempIntSet
241     for (int i = 0; i < otherIntSet.used; i++)
242     {
243         if (!tempIntSet.contains(otherIntSet.data[i]))
244         {
245             tempIntSet.data[tempIntSet.used] = otherIntSet.data[i];
```

```
246             tempIntSet.used++;
247         }
248
249     }
250
251     return tempIntSet;
252 }
253
254 IntSet IntSet::intersect(const IntSet& otherIntSet) const
255 {
256     IntSet tempIntSet;
257
258     //resize to the smallest capacity of the two IntSets
259     if (capacity < otherIntSet.capacity)
260     {
261         tempIntSet.resize(capacity);
262     }
263     else
264     {
265         tempIntSet.resize(otherIntSet.capacity);
266     }
267
268     for (int i = 0; i < used; i++)
269     {
270         for (int j = 0; j < otherIntSet.used; j++)
271         {
272             if (data[i] == otherIntSet.data[j])
273             {
274                 tempIntSet.data[tempIntSet.used] = otherIntSet.data[j];
275                 tempIntSet.used++;
276             }
277         }
278     }
279
280     return tempIntSet;
281 }
282
283 IntSet IntSet::subtract(const IntSet& otherIntSet) const
284 {
285     //call copy constr
286     IntSet tempIntSet(*this);
287
288     for (int i = 0; i < used; i++)
289     {
290         for (int j = 0; j < otherIntSet.used; j++)
291         {
292             if (data[i] == otherIntSet.data[j])
293             {
294                 tempIntSet.remove(data[i]);
```

```
295             }
296         }
297     }
298
299     return tempIntSet;
300 }
301
302 void IntSet::reset()
303 {
304     used = 0;
305 }
306
307 bool IntSet::add(int anInt)
308 {
309     //check if anInt already exists in data[]
310     if (this->contains(anInt))
311     {
312         return false;
313     }
314
315     //check if data[] is full
316     if (used == capacity)
317     {
318         //Increase capacity
319         this->resize((capacity * 1.5) + 1);
320     }
321
322     //add elem
323     data[used] = anInt;
324     used++;
325
326     return true;
327 }
328
329 bool IntSet::remove(int anInt)
330 {
331     if (this->contains(anInt))
332     {
333         int index = 0;
334
335         //find index of anInt
336         for (int i = 0; i < used; i++)
337         {
338             if (data[i] == anInt)
339             {
340                 index = i;
341                 break;
342             }
343         }
```

```
344
345         //shift index of other elements -1
346         for (index; index < used - 1; index++)
347         {
348             this->data[index] = data[index + 1];
349         }
350
351         used--;
352
353         return true;
354     }
355
356     return false;
357 }
358
359 bool operator==(const IntSet& is1, const IntSet& is2)
360 {
361
362     //check if both sets are empty
363     if (is1.size() == 0 && is2.size() == 0)
364     {
365         return true;
366     }
367     else if (is1.size() != is2.size())
368     {
369         //check if both sets are the same size
370         return false;
371     }
372     else
373     {
374         IntSet temp1(is1);
375         IntSet temp2(is2);
376
377         //use temp IntSets to avoid changing originals
378         temp1 = temp1.subtract(temp2);
379
380         //if temp1.size == 0 after subtracting temp2, then they are equal
381         if (temp1.size() == 0)
382         {
383             return true;
384         }
385     }
386
387    return false;
388 }
389
```