

```
1 // FILE: IntSet.cpp - header file for IntSet class
2 //      Implementation file for the IntStore class
3 //      (See IntSet.h for documentation.)
4 // INVARIANT for the IntSet class:
5 // (1) Distinct int values of the IntSet are stored in a 1-D,
6 //     compile-time array whose size is IntSet::MAX_SIZE;
7 //     the member variable data references the array.
8 // (2) The distinct int value with earliest membership is stored
9 //     in data[0], the distinct int value with the 2nd-earliest
10 //    membership is stored in data[1], and so on.
11 //    Note: No "prior membership" information is tracked; i.e.,
12 //          if an int value that was previously a member (but its
13 //          earlier membership ended due to removal) becomes a
14 //          member again, the timing of its membership (relative
15 //          to other existing members) is the same as if that int
16 //          value was never a member before.
17 //    Note: Re-introduction of an int value that is already an
18 //          existing member (such as through the add operation)
19 //          has no effect on the "membership timing" of that int
20 //          value.
21 // (4) The # of distinct int values the IntSet currently contains
22 //     is stored in the member variable used.
23 // (5) Except when the IntSet is empty (used == 0), ALL elements
24 //     of data from data[0] until data[used - 1] contain relevant
25 //     distinct int values; i.e., all relevant distinct int values
26 //     appear together (no "holes" among them) starting from the
27 //     beginning of the data array.
28 // (6) We DON'T care what is stored in any of the array elements
29 //     from data[used] through data[IntSet::MAX_SIZE - 1].
30 //    Note: This applies also when the IntSet is empty (used == 0)
31 //          in which case we DON'T care what is stored in any of
32 //          the data array elements.
33 //    Note: A distinct int value in the IntSet can be any of the
34 //          values an int can represent (from the most negative
35 //          through 0 to the most positive), so there is no
36 //          particular int value that can be used to indicate an
37 //          irrelevant value. But there's no need for such an
38 //          "indicator value" since all relevant distinct int
39 //          values appear together starting from the beginning of
40 //          the data array and used (if properly initialized and
41 //          maintained) should tell which elements of the data
42 //          array are actually relevant.
43
44 #include "IntSet.h"
45 #include <iostream>
46 #include <cassert>
47 using namespace std;
48
49 IntSet::IntSet()
```

```
50 {
51     for (int i = 0; i < MAX_SIZE; i++)
52     {
53         data[i] = 0;
54     }
55     used = 0;
56 }
57
58 int IntSet::size() const
59 {
60     return used;
61 }
62
63 bool IntSet::isEmpty() const
64 {
65     if (used == 0)
66     {
67         return true;
68     }
69
70     return false;
71 }
72
73 bool IntSet::contains(int anInt) const
74 {
75
76     for (int i = 0; i < used; i++)
77     {
78         if (data[i] == anInt)
79         {
80             return true;
81         }
82     }
83     return false;
84 }
85
86 bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
87 {
88     //tracks shared elements
89     int sharedElm = 0;
90
91     //returns true if IntSet is empty
92     if (used == 0)
93     {
94         return true;
95     }
96
97     //returns false if invoking IntSet is larger than otherIntSet
98     if (used > otherIntSet.used)
```

```
99     {
100         return false;
101     }
102
103     //traverse both arrays and itterates sharedElm
104     for (int i = 0; i < used; i++)
105     {
106         for (int j = 0; j < otherIntSet.used; j++)
107         {
108             if (data[i] == otherIntSet.data[j])
109             {
110                 sharedElm++;
111             }
112         }
113     }
114
115     //checks if all elemnts of IntSet are shared elements
116     if (sharedElm == used)
117     {
118         return true;
119     }
120     else
121     {
122         return false;
123     }
124 }
125
126 void IntSet::DumpData(ostream& out) const
127 { // already implemented ... DON'T change anything
128     if (used > 0)
129     {
130         out << data[0];
131         for (int i = 1; i < used; ++i)
132             out << " " << data[i];
133     }
134 }
135
136
137 IntSet IntSet::unionWith(const IntSet& otherIntSet) const
138 {
139     IntSet tempIntSet;
140
141     //populates the tempIntSet
142     for (int i = 0; i < used; i++)
143     {
144         tempIntSet.data[i] = this->data[i];
145         tempIntSet.used = this->used;
146     }
147 }
```

```
148     //adds elements from otherIntSet to tempIntSet
149     for (int i = 0; i < otherIntSet.used; i++)
150     {
151         if (!tempIntSet.contains(otherIntSet.data[i]))
152         {
153             tempIntSet.data[used] = otherIntSet.data[i];
154             tempIntSet.used++;
155         }
156     }
157 }
158
159 return tempIntSet;
160 }
161
162 IntSet IntSet::intersect(const IntSet& otherIntSet) const
163 {
164     IntSet tempIntSet;
165
166     for (int i = 0; i < used; i++)
167     {
168         for (int j = 0; j < otherIntSet.used; j++)
169         {
170             if (data[i] == otherIntSet.data[j])
171             {
172                 tempIntSet.data[tempIntSet.used] = otherIntSet.data[j];
173                 tempIntSet.used++;
174             }
175         }
176     }
177
178     return tempIntSet;
179 }
180
181 IntSet IntSet::subtract(const IntSet& otherIntSet) const
182 {
183     IntSet tempIntSet;
184
185     //populates tempIntSet
186     for (int i = 0; i < used; i++)
187     {
188         tempIntSet.data[i] = this->data[i];
189         tempIntSet.used = this->used;
190     }
191
192     for (int i = 0; i < used; i++)
193     {
194         for (int j = 0; j < otherIntSet.used; j++)
195         {
196             if (data[i] == otherIntSet.data[j])
```

```
197         {
198             tempIntSet.remove(data[i]);
199         }
200     }
201 }
202
203 return tempIntSet;
204 }
205
206 void IntSet::reset()
207 {
208     used = 0;
209 }
210
211 bool IntSet::add(int anInt)
212 {
213     //checks if this->data[] is full
214     if (used == MAX_SIZE)
215     {
216         return false;
217     }
218
219     if (!this->contains(anInt))
220     {
221         data[used] = anInt;
222         used++;
223         return true;
224     }
225
226     return false;
227 }
228
229 bool IntSet::remove(int anInt)
230 {
231     if (this->contains(anInt))
232     {
233         int index = 0;
234
235         //find index of anInt
236         for (int i = 0; i < used; i++)
237         {
238             if (data[i] == anInt)
239             {
240                 index = i;
241                 break;
242             }
243         }
244
245         //shift index of other elements -1
```

```
246         for (index; index < used - 1; index++)
247         {
248             this->data[index] = data[index + 1];
249         }
250
251         used--;
252
253         return true;
254     }
255
256     return false;
257 }
258
259 bool equal(const IntSet& is1, const IntSet& is2)
260 {
261     IntSet tempIntSet;
262
263     //checks if the IntSets are the same size
264     if (is1.size() != is2.size())
265     {
266         return false;
267     }
268
269     tempIntSet = is1.subtract(is2);
270
271     if (tempIntSet.size() == 0)
272     {
273         return true;
274     }
275     else
276     {
277         return false;
278     }
279 }
280 }
281
```