

```

1
2
3 #include "IntSet.h"
4 #include <iostream>
5 #include <cassert>
6 using namespace std;
7
8
9 void IntSet::resize(int new_capacity)
10 {
11     // Validating new capacity value
12     if (used != 0 && new_capacity < used )
13     {
14         capacity = used;
15     }
16     else if (new_capacity <= 0)
17     {
18         capacity = DEFAULT_CAPACITY;
19     }
20     else
21     {
22         capacity = new_capacity;
23     }
24
25     // Creating temp dynamic array with new capacity value.
26     int * temp_array = new int[capacity];
27
28     // Transferring old elements to new dynamic array.
29     for (int i = 0; i < used; i++)
30     {
31         temp_array[i] = data[i];
32     }
33
34     // Deallocating old array.
35     delete [] data;
36
37     // Assigning data to new array.
38     data = temp_array;
39
40     // Removing temp_array pointer because it is no longer needed
41     temp_array = NULL;
42     delete temp_array;
43 }
44
45
46 IntSet::IntSet(int initial_capacity) : capacity(initial_capacity), used(0)
47 {
48     // Initializing capacity to DEFAULT_CAPACITY if initial_capacity < 1
49     if (initial_capacity < 1)
50     {
51         capacity = DEFAULT_CAPACITY;
52     }
53
54     // Assingning 'data' to a new intance of a dynamic array of size 'capacity'
55     data = new int[capacity];
56 }
57
58
59 IntSet::IntSet(const IntSet& src) : capacity(src.capacity), used(src.used)
60 {
61     // Assingning 'data' to a new intance of a dynamic array of size 'capacity'
62     data = new int[capacity];
63
64     // Copying every element in src to data
65     for (int i = 0; i < src.used; i++)
66     {
67         data[i] = src.data[i];
68     }
69 }

```

```

70
71
72 IntSet::~~IntSet()
73 {
74     // Deallocating dynamic variables
75     delete [] data;
76     data = NULL;
77 }
78
79
80 IntSet& IntSet::operator=(const IntSet& rhs)
81 {
82     // Allocating space in temp array to hold elements in rhs
83     int * temp_array = new int[rhs.capacity];
84     for (int i = 0; i < rhs.used; i++)
85     {
86         temp_array[i] = rhs.data[i];
87     }
88
89     // Deleting current dynamic array pointed to by data
90     // and assigning data to temp_array
91     delete [] data;
92     data = temp_array;
93
94     // Copying over all properties from rhs to data
95     capacity = rhs.capacity;
96     used = rhs.used;
97
98     // Deleting temp_array pointer because is no longer needed
99     temp_array = NULL;
100    delete temp_array;
101
102    return *this;
103 }
104
105
106 int IntSet::size() const
107 {
108     // Returning # of distinct int values the invoking IntSet currently contains
109     // which is stored in the member variable used.
110     return used;
111 }
112
113
114 bool IntSet::isEmpty() const
115 {
116     // Returning true if used is equal to 0.
117     return (used == 0);
118 }
119
120
121 bool IntSet::contains(int anInt) const
122 {
123     if (used > 0)
124     {
125         for (int i = 0; i < used; i++)
126         {
127             // if (data[i] == anInt) // Differnt version
128             if (*(data + i) == anInt)
129             {
130                 return true;
131             }
132         }
133     }
134
135     return false;
136 }
137
138

```

```

139 bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
140 {
141     // An empty set is a subset of any set.
142     if (isEmpty())
143     {
144         return true;
145     }
146
147     // Check for all elements of invoking set, if any one element is not
148     // contained return false.
149     else
150     {
151         for (int i = 0; i < used; i++)
152         {
153             if (!(otherIntSet.contains(data[i])))
154             {
155                 return false;
156             }
157         }
158     }
159
160     return true;
161 }
162
163 void IntSet::DumpData(ostream& out) const
164 {
165     // already implemented ... DON'T change anything
166     if (used > 0)
167     {
168         out << data[0];
169         for (int i = 1; i < used; ++i)
170             out << " " << data[i];
171     }
172 }
173
174 IntSet IntSet::unionWith(const IntSet& otherIntSet) const
175 {
176     // Instanciating IntSet unionSet = *this to hold union elements of both
177     // this and otherIntSet
178     IntSet unionSet = *this;
179
180     // Copying over unique elements from otherIntSet since all
181     // elements from *this are already contained
182     for (int i = 0; i < otherIntSet.used; i++)
183     {
184         unionSet.add(otherIntSet.data[i]);
185     }
186
187     return unionSet;
188 }
189
190 IntSet IntSet::intersect(const IntSet& otherIntSet) const
191 {
192     // IntSet representing the intersection of the invoking IntSet
193     // and otherIntSet that will be returned
194     IntSet interSet = *this;
195
196     // Removing all elements not contained in otherIntSet
197     // from the interSet
198     for (int i = 0; i < used; i++)
199     {
200         if (!(otherIntSet.contains(interSet.data[i])))
201         {
202             interSet.remove(data[i]);
203         }
204     }
205
206     return interSet;
207 }

```

```

208
209
210 IntSet IntSet::subtract(const IntSet& otherIntSet) const
211 {
212     // IntSet representing the difference between the
213     // invoking IntSet and otherIntSet
214     IntSet subtractSet = *this;
215
216     // Removing all elems in otherIntSet that
217     // are contained in subtractSet
218     for (int i = 0; i < otherIntSet.used; i++)
219     {
220         // Calling remove() right away because the remove function
221         // does the contain() check already
222         subtractSet.remove(otherIntSet.data[i]);
223     }
224
225     return subtractSet;
226 }
227
228
229 void IntSet::reset()
230 {
231     // Deleting all array data, creating new empty array
232     // and resetting used to 0
233     delete [] data;
234     data = new int[DEFAULT_CAPACITY];
235     used = 0;
236     capacity = DEFAULT_CAPACITY;
237 }
238
239
240 bool IntSet::add(int anInt)
241 {
242     // Validating new value, if not new then false is returned
243     if (!contains(anInt))
244     {
245         // Validating that array is not full,
246         // if so then calling the resize function
247         if (used >= capacity)
248         {
249             // Resizing to atleast capacity + 1.
250             resize(int(1.5 * capacity) + 1);
251         }
252
253         // Adding new element
254         data[used] = anInt;
255         used++;
256
257         return true;
258     }
259     else
260     {
261         return false;
262     }
263 }
264
265
266 bool IntSet::remove(int anInt)
267 {
268     // Checking to see if anInt is contained in 'data'
269     // if contained, we proceed with removal
270     if (contains(anInt))
271     {
272         for (int i = 0; i < used; i++)
273         {
274             if (data[i] == anInt)
275             {
276                 // Sifting all elements to the left after the

```

```

277         // requested element was removed
278         for (int j = i; j < used - 1; j++)
279         {
280             data[j] = data[j+1];
281         }
282         --used;
283     }
284 }
285 return true;
286 }
287
288 return false;
289 }
290
291
292 bool operator==(const IntSet& is1, const IntSet& is2)
293 {
294     // If both sets are empty then they already equal to each other
295     // This saves a little bit of time
296     if (is1.isEmpty() && is2.isEmpty())
297     {
298         return true;
299     }
300     // if they are both subsets of eachother then they are equal
301     else if (is1.isSubsetOf(is2) && is2.isSubsetOf(is1))
302     {
303         return true;
304     }
305     else
306     {
307         return false;
308     }
309 }
310

```