```
// FILE: DPQueue.cpp
 1
 2
   // IMPLEMENTS: p_queue (see DPQueue.h for documentation.)
   //
 3
   // INVARIANT for the p_queue class:
 4
   //
 5
        1. The number of items in the p_queue is stored in the member
   //
 6
            variable used.
         2. The items themselves are stored in a dynamic array (partially
 7
   //
   //
           filled in general) organized to follow the usual heap storage
 8
 9
   //
           rules.
10 //
            2.1 The member variable heap stores the starting address
11
   //
                of the array (i.e., heap is the array's name). Thus,
12 //
                the items in the p_queue are stored in the elements
13 //
                heap[0] through heap[used - 1].
14 //
           2.2 The member variable capacity stores the current size of
15 //
                the dynamic array (i.e., capacity is the maximum number
16 //
                of items the array currently can accommodate).
17
   //
                NOTE: The size of the dynamic array (thus capacity) can
18
   //
                      be resized up or down where needed or appropriate
19
   //
                      by calling resize(...).
20
   // NOTE: Private helper functions are implemented at the bottom of
   // this file along with their precondition/postcondition contracts.
2.2
23 #include <cassert>
                       // provides assert function
24 #include <iostream> // provides cin, cout
                         // provides setw
25 #include <iomanip>
                         // provides log2
26 #include <cmath>
   #include "DPQueue.h"
27
28
29 using namespace std;
30
31
   namespace CS3358_FA2021_A7
32
33
       // EXTRA MEMBER FUNCTIONS FOR DEBUG PRINTING
34
       void p_queue::print_tree(const char message[], size_type i) const
35
       // Pre: (none)
       // Post: If the message is non-empty, it has first been written to
36
37
                cout. After that, the portion of the heap with root at
       //
38
       //
                node i has been written to the screen. Each node's data
39
       //
               is indented 4*d, where d is the depth of the node.
       //
                NOTE: The default argument for message is the empty string,
40
41
       11
                      and the default argument for i is zero. For example,
42
       11
                      to print the entire tree of a p_queue p, with a
43
       //
                      message of "The tree:", you can call:
44
       //
                         p.print_tree("The tree:");
45
       11
                      This call uses the default argument i=0, which prints
46
       11
                      the whole tree.
47
48
          const char NO MESSAGE[] = "";
49
          size_type depth;
50
51
          if (message[0] != '\0')
52
             cout << message << endl;</pre>
53
54
          if (i >= used)
55
             cout << "(EMPTY)" << endl;</pre>
56
          else
57
58
             depth = size\_type(log(double(i+1)) / log(2.0) + 0.1);
59
             if (2*i + 2 < used)
60
                print_tree(NO_MESSAGE, 2*i + 2);
61
             cout << setw(depth*3) << "";</pre>
62
             cout << heap[i].data;</pre>
63
             cout << '(' << heap[i].priority << ')' << endl;</pre>
64
             if (2*i + 1 < used)
65
                print_tree(NO_MESSAGE, 2*i + 1);
66
```

```
67
 68
 69
        void p_queue::print_array(const char message[]) const
 70
        // Pre: (none)
 71
        // Post: If the message is non-empty, it has first been written to
 72
        //
                 cout. After that, the contents of the array representing
 73
        //
                 the current heap has been written to cout in one line with
 74
                 values separated one from another with a space.
        //
 75
        //
                 NOTE: The default argument for message is the empty string.
 76
 77
           if (message[0] != '\0')
 78
              cout << message << endl;</pre>
 79
           if (used == 0)
 80
              cout << "(EMPTY)" << endl;</pre>
 81
 82
           else
 83
              for (size_type i = 0; i < used; i++)</pre>
 84
                 cout << heap[i].data << ' ';</pre>
 85
 86
 87
        // CONSTRUCTORS AND DESTRUCTOR
 88
 89
        p_queue::p_queue(size_type initial_capacity) : capacity(initial_capacity), used(0
 90
 91
 92
           //adjusting the capacity for user input anything <=0 will be set to default
           if (initial_capacity < 1){</pre>
 93
 94
             capacity = DEFAULT_CAPACITY;
 95
 96
 97
           // allocating new dynamic array based on input
 98
           heap = new ItemType[capacity];
 99
100
101
        p_queue::p_queue(const p_queue& src)
102
           // creating a new dynamic array bsed on src
103
104
           heap = new ItemType[capacity];
105
106
           //copying each value over to the src heap
107
           for (size_type i = 0; i < capacity; ++i){</pre>
108
             heap[i] = src.heap[i];
109
110
111
112
        p_queue::~p_queue()
113
114
           delete heap;
115
           heap = 0;
116
117
118
        // MODIFICATION MEMBER FUNCTIONS
119
        p_queue& p_queue::operator=(const p_queue& rhs)
120
           //checking for self assignment
121
122
           if (this == &rhs){return *this;}
123
124
           //creating a temporary dynamic array
125
           ItemType *temp = new ItemType[rhs.capacity];
126
           //copying the contents of the array to the temp array
127
           for (size_type i = 0; i < rhs.used; ++i){</pre>
128
129
             temp[i] = rhs.heap[i];
130
131
```

```
132
           //de-allocate old memory
133
           delete [] temp;
134
           //reassign varibles to member varibles from rhs
135
136
           heap = temp;
137
           capacity = rhs.capacity;
           used = rhs.used;
138
139
           return *this;
140
141
142
        void p_queue::push(const value_type& entry, size_type priority)
143
           //checking to see if we need to resize the dynamic array
144
145
           if (used == capacity){
             resize(size_type(1.5 * capacity) + 1);
146
147
148
149
           size_type i = used;
150
151
           //copy the new items into the heap and increment used
152
           heap[used].data = entry;
           heap[used].priority = priority;
153
154
           ++used;
155
156
           //while the new entry has higher priority than the parent swap it
157
           while(i != 0 && parent_priority(i) < heap[i].priority){</pre>
             swap_with_parent(i);
158
159
             i = parent_index(i);
160
161
        }
162
163
        void p_queue::pop()
164
165
           if (size() > 0);
166
167
           //making a base case
           if (used == 1){
168
169
             --used;
170
             return;
171
172
173
           //moving end the data to the front
174
           heap[0].data = heap[used - 1].data;
175
176
           //moving end priority to the front
177
           heap[0].priority = heap[used - 1].priority;
178
           --used;
179
180
           //creating helper indexes
181
           size_type i_parent = 0;
182
           size_type i_child = 0;
183
184
           //swapping all parents with children that are larger
185
           while(!is_leaf(i_parent) && heap[i_parent].priority <= big_child_priority(</pre>
i_parent)){
186
             i_child = big_child_index(i_parent);
             swap_with_parent(big_child_index(i_parent));
187
188
             i_parent = i_child;
189
190
        }
191
192
        // CONSTANT MEMBER FUNCTIONS
193
194
        p_queue::size_type p_queue::size() const
195
196
           return used;
```

```
197
198
199
        bool p_queue::empty() const
200
201
           if (used == 0)
202
             return true;
203
           else
204
             return false;
205
206
207
        p_queue::value_type p_queue::front() const
208
209
           if (size() > 0){
             return heap[0].data;
210
211
212
213
214
        // PRIVATE HELPER FUNCTIONS
215
        void p_queue::resize(size_type new_capacity)
216
        // Pre: (none)
217
        // Post: The size of the dynamic array pointed to by heap (thus
218
                 the capacity of the p queue) has been resized up or down
        //
219
        11
                 to new capacity, but never less than used (to prevent
220
        11
                 loss of existing data).
        //
221
                 NOTE: All existing items in the p_queue are preserved and
222
        11
                       used remains unchanged.
223
224
           //checking if new capacity is less than used if so set equal to used
225
           if (new_capacity < used) new_capacity = used;</pre>
226
           //creating a temporary item to heap of new capacity
2.2.7
           ItemType* temp = new ItemType [new_capacity];
228
229
230
           //copying the info int heap
231
           for (size_type i = 0; i < used; ++i){</pre>
232
             temp[i] = heap[i];
233
           delete [] heap;
234
235
           heap = temp;
236
           capacity = new_capacity;
237
238
239
        bool p_queue::is_leaf(size_type i) const
240
        // Pre: (i < used)
        // Post: If the item at heap[i] has no children, true has been
241
242
                 returned, otherwise false has been returned.
        //
243
244
           assert(i < used);</pre>
245
           return (((i * 2) + 1) >= used);
246
247
248
        p_queue::size_type
249
        p_queue::parent_index(size_type i) const
250
        // Pre: (i > 0) && (i < used)
251
        // Post: The index of "the parent of the item at heap[i]" has
252
        //
                 been returned.
253
254
            assert(i > 0);
255
            assert(i < used);</pre>
256
            return static_cast <size_type>((i-1)/2);
257
        }
258
259
        p_queue::size_type
        p_queue::parent_priority(size_type i) const
260
261
        // Pre: (i > 0) && (i < used)
262
        // Post: The priority of "the parent of the item at heap[i]" has
```

```
263
        //
                 been returned.
264
265
            assert(i > 0);
266
            assert(i < used);</pre>
267
            return heap [parent_index(i)].priority;
268
269
270
        p_queue::size_type
271
        p_queue::big_child_index(size_type i) const
        // Pre: is_leaf(i) returns false
272
273
        // Post: The index of "the bigger child of the item at heap[i]"
274
        //
                 has been returned.
275
        //
                 (The bigger child is the one whose priority is no smaller
276
                 than that of the other child, if there is one.)
        //
        {
277
278
           if (!(is_leaf(i)));
279
280
           size_type lhs_i = (i * 2) + 1; //index for lhs child
           size_type rhs_i = (i * 2) + 2; //index for rhs child
281
282
283
           if (i == 0){
284
             if (heap[1].priority >= heap[2].priority){
285
                 return 1;
286
287
             else return 2;
288
289
           if (rhs_i < used && heap[rhs_i].priority > heap[lhs_i].priority){
290
             return rhs_i; //2 children
291
292
           else return lhs_i; //1 child
293
294
295
296
        p_queue::size_type
297
        p_queue::big_child_priority(size_type i) const
298
        // Pre: is_leaf(i) returns false
299
        // Post: The priority of "the bigger child of the item at heap[i]"
300
                 has been returned.
        //
301
        11
                 (The bigger child is the one whose priority is no smaller
302
        11
                 than that of the other child, if there is one.)
303
           if (!(is_leaf(i)));
304
305
306
           return heap[big_child_index(i)].priority;
307
308
309
        void p queue::swap with parent(size type i)
310
        // Pre: (i > 0) && (i < used)
311
        // Post: The item at heap[i] has been swapped with its parent.
312
313
           if ( i > 0);
           if ( i < used);
314
315
316
           //find the parent index
317
           size_type parent_i = parent_index(i);
318
319
           //grab parent item
320
           ItemType temp = heap[parent_i];
321
322
           //set parent to child item
323
           heap[parent_i] = heap[i];
324
325
           //set child to parent item
326
           heap[i] = temp;
327
    }
328
```