```cpp
// FILE: DPQueue.cpp
// IMPLEMENTS: p_queue (see DPQueue.h for documentation.)
//
// INVARIANT for the p_queue class:
//   1. The number of items in the p_queue is stored in the member
//      variable used.
//   2. The items themselves are stored in a dynamic array (partially
//      filled in general) organized to follow the usual heap storage
//      rules.
//      2.1 The member variable heap stores the starting address
//          of the array (i.e., heap is the array's name). Thus,
//          the items in the p_queue are stored in the elements
//          heap[0] through heap[used - 1].
//      2.2 The member variable capacity stores the current size of
//          the dynamic array (i.e., capacity is the maximum number
//          of items the array currently can accommodate).
//          NOTE: The size of the dynamic array (thus capacity) can
//                be resized up or down where needed or appropriate
//                by calling resize(...).
// NOTE: Private helper functions are implemented at the bottom of
// this file along with their precondition/postcondition contracts.

#include <cassert>   // provides assert function
#include <iostream>  // provides cin, cout
#include <iomanip>   // provides setw
#include <cmath>     // provides log2
#include "DPQueue.h"

using namespace std;

namespace CS3358_FA2019_A7
{
   // EXTRA MEMBER FUNCTIONS FOR DEBUG PRINTING
   void p_queue::print_tree(const char message[], size_type i) const
   // Pre:  (none)
   // Post: If the message is non-empty, it has first been written to
   //       cout. After that, the portion of the heap with root at
   //       node i has been written to the screen. Each node's data
   //       is indented 4*d, where d is the depth of the node.
   //       NOTE: The default argument for message is the empty string,
   //             and the default argument for i is zero. For example,
   //             to print the entire tree of a p_queue p, with a
   //             message of "The tree:", you can call:
   //                p.print_tree("The tree:");
   //             This call uses the default argument i=0, which prints
   //             the whole tree.
   {
      const char NO_MESSAGE[] = "";
      size_type depth;

      if (message[0] != '\0')
         cout << message << endl;

      if (i >= used)
         cout << "(EMPTY)" << endl;
      else
      {
         depth = size_type( log( double(i+1) ) / log(2.0) + 0.1 );
         if (2*i + 2 < used)
            print_tree(NO_MESSAGE, 2*i + 2);
         cout << setw(depth*3) << "";
         cout << heap[i].data;
         cout << '(' << heap[i].priority << ')' << endl;
         if (2*i + 1 < used)
            print_tree(NO_MESSAGE, 2*i + 1);
      }
   }

   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;
79
80         if (used == 0)
81            cout << "(EMPTY)" << endl;
82         else
83            for (size_type i = 0; i < used; i++)
84               cout << heap[i].data << ' ';
85      }
86
87      // CONSTRUCTORS AND DESTRUCTOR
88   p_queue::p_queue(size_type initial_capacity) : capacity(initial_capacity),
89                                                  used(0)
90      {
91         // If provided capacity < 1, set to default
92         if(initial_capacity < 1){capacity = DEFAULT_CAPACITY;}
93
94         // Allocating new dynamic array with specified capacity
95         heap = new ItemType[capacity];
96      }
97
98      p_queue::p_queue(const p_queue& src) : capacity(src.capacity), used(src.used)
99      {
100        // Allocating new dynamic array with src.cpacity
101        heap = new ItemType[capacity];
102
103        // Deep copy of items in src
104        for(size_type index = 0; index < capacity; ++index)
105           heap[index] = src.heap[index];
106     }
107
108     // ~DESTRUCTOR
109     p_queue::~p_queue()
110     {
111        delete [] heap;
112        heap = 0;
113     }
114
115     // MODIFICATION MEMBER FUNCTIONS
116     p_queue& p_queue::operator=(const p_queue& rhs)
117     {
118        // If same (this) object return this
119        if (this == &rhs)
120           return *this;
121
122        // Temp array to store contents of rhs
123        ItemType *temp_heap = new ItemType[rhs.capacity];
124
125        // rsh content transfer
126        for (size_type index = 0; index < rhs.used; ++index)
127        {
128           temp_heap[index] = rhs.heap[index];
129        }
130
131        // Deallocating old dynamic array
132        delete [] heap;
133
134        // Assigning to new members
135        heap = temp_heap;
136        capacity = rhs.capacity;
137        used = rhs.used;
138        return *this;
```

```
139     }
140
141     void p_queue::push(const value_type& entry, size_type priority)
142     {
143        // Resizing if at capacity
144        if(used == capacity)
145        {
146           resize(size_type (1.5 * capacity)+1);
147        }
148
149        size_type index = used;
150
151        // Pushing new item and updating used
152        heap[used].data = entry;
153        heap[used].priority = priority;
154        ++used;
155
156        // While new item has higher priority than parent, swap
157        while(index !=0 && parent_priority(index) < heap[index].priority)
158        {
159           swap_with_parent(index);
160           index = parent_index(index);
161        }
162     }
163
164     void p_queue::pop()
165     {
166        // Checking precondition
167        assert(size() > 0);
168
169        // If only one item
170        if (used == 1)
171        {
172           --used;
173           return;
174        }
175
176        // Relocating end data to front
177        heap[0].data = heap[used-1].data;
178
179        // Relocating end priority to front
180        heap[0].priority = heap[used-1].priority;
181        --used;
182
183        // Tracking indeces
184        size_type parentIndx = 0,
185                  childIndx = 0;
186
187        // Swaping smaller parents with larger children
188        while (!is_leaf(parentIndx) && heap[parentIndx].priority
189              <= big_child_priority(parentIndx))
190        {
191           childIndx = big_child_index(parentIndx);
192           swap_with_parent(big_child_index(parentIndx));
193           parentIndx = childIndx;
194        }
195
196     }
197
198     // CONSTANT MEMBER FUNCTIONS
199
200     p_queue::size_type p_queue::size() const
201     {
202        return used;
203     }
204
205     p_queue::value_type p_queue::front() const
206     {
207        // Checking precondition
```

```
208         assert(size() > 0);
209
210         return heap[0].data;
211      }
212
213      bool p_queue::empty() const
214      {
215         return (used == 0);
216      }
217
218      // PRIVATE HELPER FUNCTIONS
219      void p_queue::resize(size_type new_capacity)
220      // Pre:  (none)
221      // Post: The size of the dynamic array pointed to by heap (thus
222      //       the capacity of the p_queue) has been resized up or down
223      //       to new_capacity, but never less than used (to prevent
224      //       loss of existing data).
225      //       NOTE: All existing items in the p_queue are preserved and
226      //             used remains unchanged.
227      {
228         // Checking to see if new_capacity less than used
229         // if so, then set new_capacity to used
230         if(new_capacity < used)
231         {
232            new_capacity = used;
233         }
234
235         //Creating temp heap to store new_capacity heap
236         ItemType* temp_heap = new ItemType[new_capacity];
237
238         // Deep copy of items
239         for(size_type index = 0; index < used; ++index)
240         {
241            temp_heap[index] = heap[index];
242         }
243         // Deallocating memory
244         delete [] heap;
245         heap = temp_heap;
246         capacity = new_capacity;
247      }
248
249      bool p_queue::is_leaf(size_type i) const
250      // Pre:  (i < used)
251      // Post: If the item at heap[i] has no children, true has been
252      //       returned, otherwise false has been returned.
253      {
254         // Checking precondition
255         assert(i <used);
256
257         return (((i*2)+1) >= used);
258      }
259
260      p_queue::size_type
261      p_queue::parent_index(size_type i) const
262      // Pre:  (i > 0) && (i < used)
263      // Post: The index of "the parent of the item at heap[i]" has
264      //       been returned.
265      {
266         // Checking preconditions
267         assert(i > 0);
268         assert(i < used);
269
270         return static_cast<size_type>((i-1)/2);
271      }
272
273      p_queue::size_type
274      p_queue::parent_priority(size_type i) const
275      // Pre:  (i > 0) && (i < used)
276      // Post: The priority of "the parent of the item at heap[i]" has
```

```
277      //       been returned.
278      {
279         // Checking preconditions
280         assert(i > 0);
281         assert(i < used);
282         return heap[parent_index(i)].priority;
283      }
284
285      p_queue::size_type
286      p_queue::big_child_index(size_type i) const
287      // Pre:  is_leaf(i) returns false
288      // Post: The index of "the bigger child of the item at heap[i]"
289      //       has been returned.
290      //       (The bigger child is the one whose priority is no smaller
291      //       than that of the other child, if there is one.)
292      {
293         // checking precondition
294         assert(!(is_leaf(i)));
295
296         size_type LHSC_index = (i * 2) + 1; // Index of LHS child
297         size_type RHSC_index = (i * 2) + 2; // Index of RHS child
298
299         if(i == 0)
300         {
301            if(heap[1].priority >= heap[2].priority)
302            {
303               return 1;
304            }
305            else
306            {
307               return 2;
308            }
309         }
310         if (RHSC_index < used && heap[RHSC_index].priority > heap[LHSC_index].priority)
311         {
312            return RHSC_index;  // Two children present
313         }
314         else
315         {
316            return LHSC_index;  // One child present
317         }
318      }
319
320      p_queue::size_type
321      p_queue::big_child_priority(size_type i) const
322      // Pre:  is_leaf(i) returns false
323      // Post: The priority of "the bigger child of the item at heap[i]"
324      //       has been returned.
325      //       (The bigger child is the one whose priority is no smaller
326      //       than that of the other child, if there is one.)
327      {
328         // Checking precondition
329         assert(!(is_leaf(i)));
330
331         return heap[big_child_index(i)].priority;
332      }
333
334      void p_queue::swap_with_parent(size_type i)
335      // Pre:  (i > 0) && (i < used)
336      // Post: The item at heap[i] has been swapped with its parent.
337      {
338         // Checking preconditions
339         assert(i > 0);
340         assert(i < used);
341
342         // Finding parent index
343         size_type parentIndx = parent_index(i);
344
345         // Copying parent item
```

```
346         ItemType temp_item = heap[parentIndx];
347 
348         // Swaping parent item with child[i] item
349         heap[parentIndx] = heap[i];
350 
351         // Swaping child item with parent's item
352         heap[i] = temp_item;
353     }
354 }
355 
```