```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_SP2023_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;
```

```
50
51          if (message[0] != '\0')
52             cout << message << endl;
53
54          if (i >= used)
55             cout << "(EMPTY)" << endl;
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) << "";
62             cout << heap[i].data;
63             cout << '(' << heap[i].priority << ')' << endl;
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;
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
89       p_queue::p_queue(size_type initial_capacity)
90       {
91           capacity = initial_capacity;
92           used = 0;
93           heap = new ItemType[capacity];
94       }
95
96       p_queue::p_queue(const p_queue& src)
97       {
98           capacity = src.capacity;
```

```
 99          used = src.capacity;
100          heap = new ItemType[capacity];
101
102          for (int i = 0; i < src.used; i++)
103          {
104              heap[i] = src.heap[i];
105          }
106      }
107
108      p_queue::~p_queue()
109      {
110          capacity = 0;
111          used = 0;
112          delete[] heap;
113      }
114
115      // MODIFICATION MEMBER FUNCTIONS
116      p_queue& p_queue::operator=(const p_queue& rhs)
117      {
118          if (this->capacity < rhs.capacity)
119          {
120              this->resize(rhs.capacity);
121          }
122
123          for (int i = 0; i < rhs.used; i++)
124          {
125              this->heap[i] = rhs.heap[i];
126          }
127
128          this->used = rhs.used;
129
130          return *this;
131      }
132
133      void p_queue::push(const value_type& entry, size_type priority)
134      {
135          if (used == capacity) // resize if neccesary
136          {
137              this->resize(capacity * 2);
138          }
139
140          ItemType temp;
141          size_type tempIndex = used;
142          temp.data = entry;
143          temp.priority = priority;
144          heap[tempIndex] = temp;
145
146          if (this->empty())
147          {
```

```
148             used++;
149             return;
150         }
151 
152         while (heap[tempIndex].priority > parent_priority(tempIndex))
153         {
154             if (tempIndex == 0)
155             {
156                 break;
157             }
158 
159             swap_with_parent(tempIndex);
160             tempIndex = parent_index(tempIndex);
161         }
162         used++;
163     }
164 
165     void p_queue::pop()
166     {
167         if (this->empty())
168         {
169             return;
170         }
171 
172         size_type tempIndex = 0;
173         size_type newIndex;
174 
175         heap[0] = heap[used - 1];
176 
177         while (heap[tempIndex].priority < big_child_priority(tempIndex))
178         {
179             newIndex = big_child_index(tempIndex);
180 
181             swap_with_parent(newIndex);
182 
183             tempIndex = newIndex;
184         }
185 
186         used--;
187     }
188 
189     // CONSTANT MEMBER FUNCTIONS
190 
191     p_queue::size_type p_queue::size() const
192     {
193         return used;
194     }
195 
196     bool p_queue::empty() const
```

```
197     {
198         if (used == 0)
199         {
200             return true;
201         }
202 
203         return false;
204     }
205 
206     p_queue::value_type p_queue::front() const
207     {
208         if (this->size() > 0)
209         {
210             value_type front = heap[0].data;
211             return front;
212         }
213     }
214 
215     // PRIVATE HELPER FUNCTIONS
216     void p_queue::resize(size_type new_capacity)
217     // Pre:  (none)
218     // Post: The size of the dynamic array pointed to by heap (thus
219     //       the capacity of the p_queue) has been resized up or down
220     //       to new_capacity, but never less than used (to prevent
221     //       loss of existing data).
222     //       NOTE: All existing items in the p_queue are preserved and
223     //             used remains unchanged.
224     {
225         if (new_capacity < used)
226         {
227             new_capacity = used;
228         }
229 
230         ItemType* temp = new ItemType[new_capacity];
231 
232         for (int i = 0; i < used; i++)
233         {
234             temp[i] = heap[i];
235         }
236 
237         delete[] heap;
238 
239         heap = temp;
240     }
241 
242     bool p_queue::is_leaf(size_type i) const
243     // Pre:  (i < used)
244     // Post: If the item at heap[i] has no children, true has been
245     //       returned, otherwise false has been returned.
```

```
246     {
247         if (((2 * i) + 1) < used) // check for left child
248         {
249             return false;
250         }
251 
252         if (((2 * i) + 2) < used) // check for right child
253         {
254             return false;
255         }
256 
257         return true;
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         return (i - 1) / 2;
267     }
268 
269     p_queue::size_type
270     p_queue::parent_priority(size_type i) const
271     // Pre:  (i > 0) && (i < used)
272     // Post: The priority of "the parent of the item at heap[i]" has
273     //       been returned.
274     {
275         size_type temp;
276 
277         temp = parent_index(i);
278 
279         return heap[temp].priority;
280     }
281 
282     p_queue::size_type
283     p_queue::big_child_index(size_type i) const
284     // Pre:  is_leaf(i) returns false
285     // Post: The index of "the bigger child of the item at heap[i]"
286     //       has been returned.
287     //       (The bigger child is the one whose priority is no smaller
288     //       than that of the other child, if there is one.)
289     {
290         size_type leftChild = (2 * i) + 1;
291         size_type rightChild = (2 * i) + 2;
292 
293         if (rightChild >= used) // no right child
294         {
```

```
295             return leftChild;
296         }
297 
298         if (heap[leftChild].priority > heap[rightChild].priority)
299         {
300             return leftChild;
301         }
302         else if (heap[leftChild].priority < heap[rightChild].priority)
303         {
304             return rightChild;
305         }
306         else // equal priority returns index of largest data
307         {
308             if (heap[leftChild].data > heap[rightChild].data)
309             {
310                 return leftChild;
311             }
312             else
313             {
314                 return rightChild;
315             }
316         }
317     }
318 
319     p_queue::size_type
320     p_queue::big_child_priority(size_type i) const
321     // Pre:  is_leaf(i) returns false
322     // Post: The priority of "the bigger child of the item at heap[i]"
323     //       has been returned.
324     //       (The bigger child is the one whose priority is no smaller
325     //       than that of the other child, if there is one.)
326     {
327         size_type temp = big_child_index(i);
328 
329         return heap[temp].priority;
330     }
331 
332     void p_queue::swap_with_parent(size_type i)
333         // Pre:  (i > 0) && (i < used)
334         // Post: The item at heap[i] has been swapped with its parent.
335     {
336         ItemType temp = heap[i];
337         size_type pIndex = parent_index(i);
338 
339         heap[i] = heap[pIndex];
340         heap[pIndex] = temp;
341     }
342 }
343 
```