```
// 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_FA2021_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
// Pre:  (none)
// Post: If the message is non-empty, it has first been written to
//       cout. After that, the contents of the array representing
//       the current heap has been written to cout in one line with
//       values separated one from another with a space.
//       NOTE: The default argument for message is the empty string.
{
   if (message[0] != '\0')
      cout << message << endl;

   if (used == 0)
      cout << "(EMPTY)" << endl;
   else
      for (size_type i = 0; i < used; i++)
         cout << heap[i].data << ' ';
}

// CONSTRUCTORS AND DESTRUCTOR

p_queue::p_queue(size_type initial_capacity) : used(0)
{
  heap = new ItemType[5];
}

p_queue::p_queue(const p_queue& src)
{
  *this = src;
}

p_queue::~p_queue()
{
  delete [] heap;
}

// MODIFICATION MEMBER FUNCTIONS
p_queue& p_queue::operator=(const p_queue& rhs)
{
    resize(rhs.capacity);
    used = rhs.used;
    for (size_type i = 0; i < rhs.used; i++)
    {
```

```
        heap[i] = rhs.heap[i];
    }
    return *this;
}

void p_queue::push(const value_type& entry, size_type priority)
{
    if (used == capacity)
    {
      resize(capacity * 2);
    }
    heap[used].data = entry;
    heap[used].priority = priority;
    size_type i = used;
    used++;
    while (i != 0 && parent_priority(i) < heap[i].priority)
    {
        swap_with_parent(i);
        i = parent_index(i);
    }
}

void p_queue::pop(){
    if (used == 1)
    {
        used--;
        return;
    }
    heap[0].data = heap[used-1].data;
    heap[0].priority = heap[used-1].priority;
    used--;
    size_type i = 0;
    size_type index;
    while (!is_leaf(i) && heap[i].priority <= big_child_priority(i))
    {
        index = big_child_index(i);
        swap_with_parent(index);
        i = index;
    }
}

// CONSTANT MEMBER FUNCTIONS

p_queue::size_type p_queue::size() const
{
  return used;
}

bool p_queue::empty() const
{
  return used==0;
}

p_queue::value_type p_queue::front() const
{
```

```
  return heap[0].data;
}

// PRIVATE HELPER FUNCTIONS
void p_queue::resize(size_type new_capacity)
// Pre:  (none)
// Post: The size of the dynamic array pointed to by heap (thus
//       the capacity of the p_queue) has been resized up or down
//       to new_capacity, but never less than used (to prevent
//       loss of existing data).
//       NOTE: All existing items in the p_queue are preserved and
//             used remains unchanged.
{
    if (new_capacity < used) new_capacity = used;
    ItemType *newHeap = new ItemType[new_capacity];
    for (size_type i = 0; i < used; i++)
    {
        newHeap[i].priority = heap[i].priority;
        newHeap[i].data = heap[i].data;
    }
    ItemType *temp = heap;
    heap = newHeap;
    delete temp;
    capacity = new_capacity;
}

bool p_queue::is_leaf(size_type i) const
// Pre:  (i < used)
// Post: If the item at heap[i] has no children, true has been
//       returned, otherwise false has been returned.
{
    assert(i < used);

    if ((i*2+1) >= used)
    {
      return true;
    }
    return false;
}

p_queue::size_type
p_queue::parent_index(size_type i) const
// Pre:  (i > 0) && (i < used)
// Post: The index of "the parent of the item at heap[i]" has
//       been returned.
{
    assert(i > 0 && i < used);

    return static_cast<size_type>((i-1)/2);
}

p_queue::size_type
p_queue::parent_priority(size_type i) const
// Pre:  (i > 0) && (i < used)
// Post: The priority of "the parent of the item at heap[i]" has
```

```
//       been returned.
{
    assert(i > 0 && i < used);

    return heap[parent_index(i)].priority;
}

p_queue::size_type
p_queue::big_child_index(size_type i) const
// Pre:  is_leaf(i) returns false
// Post: The index of "the bigger child of the item at heap[i]"
//       has been returned.
//       (The bigger child is the one whose priority is no smaller
//       than that of the other child, if there is one.)
{
    assert(!is_leaf(i));

    if (i == 0)
    {
        if (heap[1].priority >= heap[2].priority) return 1;
        return 2;
    }
    if (i*2+2 < used && heap[i*2+2].priority > heap[i*2+1].priority)
    {
      return i*2+2;
    }
    return i*2+1;
}

p_queue::size_type
p_queue::big_child_priority(size_type i) const
// Pre:  is_leaf(i) returns false
// Post: The priority of "the bigger child of the item at heap[i]"
//       has been returned.
//       (The bigger child is the one whose priority is no smaller
//       than that of the other child, if there is one.)
{
    assert(!is_leaf(i));

    return heap[big_child_index(i)].priority;
}

void p_queue::swap_with_parent(size_type i)
// Pre:  (i > 0) && (i < used)
// Post: The item at heap[i] has been swapped with its parent.
{
    assert(i > 0 && i < used);

    size_type index = parent_index(i);
    ItemType temp = heap[index];
    heap[index] = heap[i];
    heap[i] = temp;
}
}
```