```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_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) : capacity(initial_capacity), used(0)
{

   //adjusting the capacity for user input anything <=0 will be set to default
   if (initial_capacity < 1){
     capacity = DEFAULT_CAPACITY;
   }

   // allocating new dynamic array based on input
   heap = new ItemType[capacity];
}

p_queue::p_queue(const p_queue& src)
{
   // creating a new dynamic array bsed on src
   heap = new ItemType[capacity];

   //copying each value over to the src heap
   for (size_type i = 0; i < capacity; ++i){
     heap[i] = src.heap[i];
   }
}

p_queue::~p_queue()
{
   delete heap;
   heap = 0;
}

// MODIFICATION MEMBER FUNCTIONS
p_queue& p_queue::operator=(const p_queue& rhs)
{
   //checking for self assignment
   if (this == &rhs){return *this;}

   //creating a temporary dynamic array
   ItemType *temp = new ItemType[rhs.capacity];

   //copying the contents of the array to the temp array
   for (size_type i = 0; i < rhs.used; ++i){
     temp[i] = rhs.heap[i];
   }

   //de-allocate old memory
   delete [] temp;

   //reassign varibles to member varibles from rhs
   heap = temp;
   capacity = rhs.capacity;
   used = rhs.used;
   return *this;
}

void p_queue::push(const value_type& entry, size_type priority)
{
   //checking to see if we need to resize the dynamic array
   if (used == capacity){
     resize(size_type(1.5 * capacity) + 1);
   }

   size_type i = used;

   //copy the new items into the heap and increment used
   heap[used].data = entry;
   heap[used].priority = priority;
   ++used;

   //while the new entry has higher priority than the parent swap it
   while(i != 0 && parent_priority(i) < heap[i].priority){
     swap_with_parent(i);
     i = parent_index(i);
   }
}

void p_queue::pop()
{
   if (size() > 0);

   //making a base case
   if (used == 1){
     --used;
     return;
   }

   //moving end the data to the front
   heap[0].data = heap[used - 1].data;

   //moving end priority to the front
   heap[0].priority = heap[used - 1].priority;
   --used;

   //creating helper indexes
   size_type i_parent = 0;
   size_type i_child = 0;

   //swapping all parents with children that are larger
   while(!is_leaf(i_parent) && heap[i_parent].priority <= big_child_priority(
i_parent)){
     i_child = big_child_index(i_parent);
     swap_with_parent(big_child_index(i_parent));
     i_parent = i_child;
   }
}

// CONSTANT MEMBER FUNCTIONS

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

```
}

bool p_queue::empty() const
{
   if (used == 0)
     return true;
   else
     return false;
}

p_queue::value_type p_queue::front() const
{
   if (size() > 0){
     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.
{
   //checking if new capacity is less than used if so set equal to used
   if (new_capacity < used) new_capacity = used;

   //creating a temporary item to heap of new capacity
   ItemType* temp = new ItemType [new_capacity];

   //copying the info int heap
   for (size_type i = 0; i < used; ++i){
     temp[i] = heap[i];
   }
   delete [] heap;
   heap = 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);
   return (((i * 2) + 1) >= used);
}

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);
    assert(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);
       assert(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.)
   {
      if (!(is_leaf(i)));

      size_type lhs_i = (i * 2) + 1; //index for lhs child
      size_type rhs_i = (i * 2) + 2; //index for rhs child

      if (i == 0){
        if (heap[1].priority >= heap[2].priority){
            return 1;
        }
        else return 2;
      }
      if (rhs_i < used && heap[rhs_i].priority > heap[lhs_i].priority){
        return rhs_i; //2 children
      }
      else return lhs_i; //1 child

   }

   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.)
   {
      if (!(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.
   {
      if ( i > 0);
      if ( i < used);

      //find the parent index
      size_type parent_i = parent_index(i);

      //grab parent item
      ItemType temp = heap[parent_i];

      //set parent to child item
      heap[parent_i] = heap[i];

      //set child to parent item
      heap[i] = temp;
   }
}
```