```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;

      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;
       heap = new ItemType[capacity];
   }

   p_queue::p_queue(const p_queue& src)
   {
       capacity = src.capacity;
```

```cpp
        used = src.capacity;
        heap = new ItemType[capacity];

        for (int i = 0; i < src.used; i++)
        {
            heap[i] = src.heap[i];
        }
    }

    p_queue::~p_queue()
    {
        capacity = 0;
        used = 0;
        delete[] heap;
    }

    // MODIFICATION MEMBER FUNCTIONS
    p_queue& p_queue::operator=(const p_queue& rhs)
    {
        if (this->capacity < rhs.capacity)
        {
            this->resize(rhs.capacity);
        }

        for (int i = 0; i < rhs.used; i++)
        {
            this->heap[i] = rhs.heap[i];
        }

        this->used = rhs.used;

        return *this;
    }

    void p_queue::push(const value_type& entry, size_type priority)
    {
        if (used == capacity) // resize if neccesary
        {
            this->resize(capacity * 2);
        }

        ItemType temp;
        size_type tempIndex = used;
        temp.data = entry;
        temp.priority = priority;
        heap[tempIndex] = temp;

        if (this->empty())
        {
```

```
        used++;
        return;
    }

    while (heap[tempIndex].priority > parent_priority(tempIndex))
    {
        if (tempIndex == 0)
        {
            break;
        }

        swap_with_parent(tempIndex);
        tempIndex = parent_index(tempIndex);
    }
    used++;
}

void p_queue::pop()
{
    if (this->empty())
    {
        return;
    }

    size_type tempIndex = 0;
    size_type newIndex;

    heap[0] = heap[used - 1];

    while (heap[tempIndex].priority < big_child_priority(tempIndex))
    {
        newIndex = big_child_index(tempIndex);

        swap_with_parent(newIndex);

        tempIndex = newIndex;
    }

    used--;
}

// CONSTANT MEMBER FUNCTIONS

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

bool p_queue::empty() const
```

```cpp
{
    if (used == 0)
    {
        return true;
    }

    return false;
}

p_queue::value_type p_queue::front() const
{
    if (this->size() > 0)
    {
        value_type front = heap[0].data;
        return front;
    }
}

// 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* temp = new ItemType[new_capacity];

    for (int i = 0; i < used; i++)
    {
        temp[i] = heap[i];
    }

    delete[] heap;

    heap = temp;
}

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.
```

```
{
    if (((2 * i) + 1) < used) // check for left child
    {
        return false;
    }

    if (((2 * i) + 2) < used) // check for right child
    {
        return false;
    }

    return true;
}

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.
{
    return (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.
{
    size_type temp;

    temp = parent_index(i);

    return heap[temp].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.)
{
    size_type leftChild = (2 * i) + 1;
    size_type rightChild = (2 * i) + 2;

    if (rightChild >= used) // no right child
    {
            return leftChild;
        }

        if (heap[leftChild].priority > heap[rightChild].priority)
        {
            return leftChild;
        }
        else if (heap[leftChild].priority < heap[rightChild].priority)
        {
            return rightChild;
        }
        else // equal priority returns index of largest data
        {
            if (heap[leftChild].data > heap[rightChild].data)
            {
                return leftChild;
            }
            else
            {
                return rightChild;
            }
        }
    }

    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.)
    {
        size_type temp = big_child_index(i);

        return heap[temp].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.
    {
        ItemType temp = heap[i];
        size_type pIndex = parent_index(i);

        heap[i] = heap[pIndex];
        heap[pIndex] = temp;
    }
}
```