// 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

// 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)

{

if(initial\_capacity < 1)

capacity = DEFAULT\_CAPACITY;

heap = new ItemType[capacity];

}

p\_queue::p\_queue(const p\_queue& src) : capacity(src.capacity), used(src.used)

{

heap = new ItemType[capacity];

for(size\_type i = 0; i < capacity; ++i)

heap[i] = src.heap[i];

}

p\_queue::~p\_queue()

{

delete [] heap;

heap = nullptr;

}

// MODIFICATION MEMBER FUNCTIONS

p\_queue& p\_queue::operator=(const p\_queue& rhs)

{

if (this == &rhs)

return \*this;

ItemType \*tempHeap = new ItemType[rhs.capacity];

for(size\_type i = 0; i < rhs.used; ++i)

tempHeap[i] = rhs.heap[i];

delete [] heap;

heap = tempHeap;

capacity = rhs.capacity;

used = rhs.used;

return \*this;

}

void p\_queue::push(const value\_type& entry, size\_type priority)

{

if(used == capacity)

resize(size\_type(1.05 \* capacity) + 1);

size\_type i = used;

heap[used].data = entry;

heap[used].priority = priority;

++used;

while(i != 0 && parent\_priority(i) < heap[i].priority)

{

swap\_with\_parent(i);

i = parent\_index(i);

}

}

void p\_queue::pop()

{

assert(size() > 0);

if(used == 1)

{

--used;

return;

}

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

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

--used;

size\_type iParent = 0;

size\_type iChild = 0;

while(!is\_leaf(iParent) && heap[iParent].priority <=

big\_child\_priority(iParent))

{

iChild = big\_child\_index(iParent);

swap\_with\_parent(big\_child\_index(iParent));

iParent = iChild;

}

}

// 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

{

assert(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.

{

if(used > new\_capacity)

new\_capacity = used;

ItemType\* tempHeap = new ItemType[new\_capacity];

for(size\_type i = 0; i < used; ++i)

tempHeap[i] = heap[i];

delete [] heap;

heap = tempHeap;

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(used > i);

return (2 \* i + 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.)

{

assert(!is\_leaf(i));

size\_type lhs\_i = (2 \* i) + 1;

size\_type rhs\_i = (2 \* i) + 2;

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;

else

return lhs\_i;

}

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);

assert(i < used);

size\_type parentI = parent\_index(i);

ItemType tItem = heap[parentI];

heap[parentI] = heap[i];

heap[i] = tItem;

}

}