**Modified Queue**

**ADT Queue Operations**

+isEmpty( ):boolean {query}

Determines whether Queue is empty

+enqueue(in newEntry:ItemType)

{exception QueueException}

Places newEntry at back of the Queue

Throws QueueException if Queue is full

+dequeue()

{exception QueueException}

Removes the front of the Queue

Throws QueueException if Queue is initially empty

+peekFront():ItemType {query}

{exception QueueException}

Returns a copy of the front of the Queue

Throws QueueException if Queue is empty

**QueueInterface.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* @file QueueInterface.h \*/

#ifndef \_QUEUE\_INTERFACE

#define \_QUEUE\_INTERFACE

template<class ItemType>

class QueueInterface

{

public:

/\*\* Sees whether this Queue is empty.

@return true if the Queue is empty, or false if not. \*/

virtual bool isEmpty() const = 0;

/\*\* Adds a new entry to the back of this Queue.

@post newEntry is at the back of the Queue.

@param newEntry The object to be added as a new entry.

@throw QueueException if Queue is full. \*/

virtual void enqueue(const ItemType& newEntry) = 0;

/\*\* Removes the front of this Queue.

@post The front of the Queue is removed.

@throw QueueException if Queue is initially empty. \*/

virtual void dequeue() = 0;

/\*\* Returns a copy of the front of this Queue.

@post A copy of the front of the Queue is returned, and

the Queue is unchanged.

@return A copy of the front of the Queue.

@throw QueueException if Queue is initially empty. \*/

virtual ItemType peekFront() const = 0;

};

#endif

**QueueException.h**

// Modified from Carrano, Data Abstraction & Problem Solving with C++, Fifth Edition,

// Pearson Education, 2007, page 348.

#ifndef \_QUEUE\_EXCEPTION

#define \_QUEUE\_EXCEPTION

#include <stdexcept>

#include <string>

using namespace std;

class QueueException : public logic\_error

{

public:

QueueException(const string& message = "")

: logic\_error(message.c\_str())

{}

};

#endif

**ArrayQueue.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT Queue: Array-based implementation.

Listing 7-1

@file ArrayQueue.h \*/

#ifndef \_ARRAY\_QUEUE

#define \_ARRAY\_QUEUE

#include "QueueInterface.h"

#include "QueueException.h"

const int MAX\_QUEUE = 100;

template<class ItemType>

class ArrayQueue : public QueueInterface<ItemType>

{

private:

ItemType items[MAX\_QUEUE]; // Array of Queue items

int front; // Index to front of Queue

int back; // Index to back of Queue

int count; // Number of items currently in the Queue

public:

ArrayQueue();

// Copy constructor and destructor are supplied by the compiler

bool isEmpty() const;

/\*\* @throw QueueException if Queue is full. \*/

void enqueue(const ItemType& newEntry);

/\*\* @throw QueueException if Queue is initially empty. \*/

void dequeue();

/\*\* @throw QueueException if Queue is empty. \*/

ItemType peekFront() const;

};

#include "ArrayQueue.cpp"

#endif

**ArrayQueue.cpp**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Listing 7-1

@file ArrayQueue.cpp \*/

template<class ItemType>

ArrayQueue<ItemType>::ArrayQueue() : front(0), back(MAX\_QUEUE - 1), count(0)

{

}

template<class ItemType>

bool ArrayQueue<ItemType>::isEmpty() const

{

return count == 0;

}

/\*\* @throw QueueException if Queue is full. \*/

template<class ItemType>

void ArrayQueue<ItemType>::enqueue(const ItemType& newEntry)

{

if (count >= MAX\_QUEUE)

throw QueueException("QueueException: Queue full on enqueue");

else

{

back = (back + 1) % MAX\_QUEUE;

items[back] = newEntry;

count++;

}

}

/\*\* @throw QueueException if Queue is initially empty. \*/

template<class ItemType>

void ArrayQueue<ItemType>::dequeue()

{

if (isEmpty())

throw QueueException("QueueException: Queue empty on dequeue");

else

{

front = (front + 1) % MAX\_QUEUE;

count--;

}

}

/\*\* @throw QueueException if Queue is empty. \*/

template<class ItemType>

ItemType ArrayQueue<ItemType>::peekFront() const

{

if (isEmpty())

throw QueueException("QueueException: Queue empty on peekFront");

else

return items[front];

}

**Node.h** and **Node.cpp**

See pages 136-137 of the text.

**LinkedQueue.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT Queue: Link-based implementation.

Listing 7-3.

@file LinkedQueue.h \*/

#ifndef \_LINKED\_QUEUE

#define \_LINKED\_QUEUE

#include "QueueInterface.h"

#include "Node.h"

#include "QueueException.h"

template<class ItemType>

class LinkedQueue : public QueueInterface<ItemType>

{

private:

// The queue is implemented as a chain of linked nodes that has

// two external pointers, a head pointer for front of the queue and

// a tail pointer for the back of the queue.

Node<ItemType>\* backPtr;

Node<ItemType>\* frontPtr;

public:

LinkedQueue();

LinkedQueue(const LinkedQueue<ItemType>& aQueue);

virtual ~LinkedQueue();

bool isEmpty() const;

/\*\* @throw QueueException if Queue is full. \*/

void enqueue(const ItemType& newEntry);

/\*\* @throw QueueException if Queue is initially empty. \*/

void dequeue();

/\*\* @throw QueueException if Queue is empty. \*/

ItemType peekFront() const;

}; // end LinkedQueue

#include "LinkedQueue.cpp"

#endif

**LinkedQueue.cpp**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Listing 7-4.

@file LinkedQueue.cpp \*/

template<class ItemType>

LinkedQueue<ItemType>::LinkedQueue() : frontPtr(nullptr), backPtr(nullptr)

{

}

template<class ItemType>

LinkedQueue<ItemType>::LinkedQueue(const LinkedQueue& aQueue)

{

Node<ItemType>\* origChainPtr = aQueue->frontPtr;

if (origChainPtr == nullptr)

{

frontPtr = nullptr; // Original queue is empty

backPtr = nullptr;

}

else

{

// Copy first node

frontPtr = new Node<ItemType>();

frontPtr->setItem(origChainPtr->getItem());

origChainPtr = origChainPtr->getNext();

// Copy remaining nodes

Node<ItemType>\* newChainPtr = frontPtr;

while (origChainPtr != nullptr)

{

ItemType nextItem = origChainPtr->getItem();

Node<ItemType>\* newNodePtr =

new Node<ItemType>(nextItem);

newChainPtr->setNext(newNodePtr);

newChainPtr = newChainPtr->getNext();

origChainPtr = origChainPtr->getNext();

}

newChainPtr->setNext(nullptr);

backPtr = newChainPtr;

}

}

template<class ItemType>

LinkedQueue<ItemType>::~LinkedQueue()

{

while (!isEmpty())

dequeue();

}

template<class ItemType>

bool LinkedQueue<ItemType>::isEmpty() const

{

return frontPtr == nullptr;

}

/\*\* @throw QueueException if Queue is full. \*/

template<class ItemType>

void LinkedQueue<ItemType>::enqueue(const ItemType& newEntry)

{

try

{

Node<ItemType>\* newNodePtr = new Node<ItemType>(newEntry);

// Insert the new node

if (isEmpty())

frontPtr = newNodePtr; // Insertion into empty queue

else

backPtr->setNext(newNodePtr); // Insertion into nonempty queue

backPtr = newNodePtr; // New node is at back

}

catch (bad\_alloc e)

{

throw QueueException("QueueException: cannot allocate memory on enqueue");

}

}

/\*\* @throw QueueException if Queue is initially empty. \*/

template<class ItemType>

void LinkedQueue<ItemType>::dequeue()

{

if (isEmpty())

throw QueueException("QueueException: Queue empty on dequeue");

else

{

// Queue is not empty; delete front

Node<ItemType>\* nodeToDeletePtr = frontPtr;

if (frontPtr == backPtr)

{

// Special case: one node in queue

frontPtr = nullptr;

backPtr = nullptr;

}

else

frontPtr = frontPtr->getNext();

// Return deleted node to system

nodeToDeletePtr->setNext(nullptr);

delete nodeToDeletePtr;

nodeToDeletePtr = nullptr;

}

}

/\*\* @throw QueueException if Queue is empty. \*/

template<class ItemType>

ItemType LinkedQueue<ItemType>::peekFront() const

{

if (isEmpty())

throw QueueException("QueueException: Queue empty on dequeue");

else

return frontPtr->getItem();

}

**ListQueue.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT queue: ADT list implementation.

Listing 14-1.

@file ListQueue.h \*/

#ifndef \_LIST\_QUEUE

#define \_LIST\_QUEUE

#include "QueueInterface.h"

#include "LinkedList.h"

#include "QueueException.h"

template<class ItemType>

class ListQueue : public QueueInterface<ItemType>

{

private:

LinkedList<ItemType> theList;

public:

ListQueue();

ListQueue(const ListQueue& aQueue);

~ListQueue();

bool isEmpty() const;

/\*\* @throw QueueException if Queue is full. \*/

void enqueue(const ItemType& newEntry);

/\*\* @throw QueueException if Queue is initially empty. \*/

void dequeue();

/\*\* @throw QueueException if Queue is empty. \*/

ItemType peekFront() const;

};

#include "ListQueue.cpp"

#endif

**ListQueue.cpp**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT queue: ADT list implementation.

Listing 14-2.

@file ListQueue.cpp \*/

#include "ListQueue.h" // header file

template<class ItemType>

ListQueue<ItemType>::ListQueue()

{

}

template<class ItemType>

ListQueue<ItemType>::ListQueue(const ListQueue& aQueue) : theList(aQueue.theList)

{

}

template<class ItemType>

ListQueue<ItemType>::~ListQueue()

{

}

template<class ItemType>

bool ListQueue<ItemType>::isEmpty() const

{

return theList.isEmpty();

}

/\*\* @throw QueueException if Queue is full. \*/

template<class ItemType>

void ListQueue<ItemType>::enqueue(const ItemType& newEntry)

{

try

{

theList.insert(theList.getLength() + 1, newEntry);

}

catch (ListException)

{

throw QueueException("QueueException: cannot enqueue item");

}

}

/\*\* @throw QueueException if Queue is initially empty. \*/

template<class ItemType>

void ListQueue<ItemType>::dequeue()

{

if (theList.isEmpty())

throw QueueException ("QueueException: cannot dequeue from empty queue");

else

theList.remove(1);

}

/\*\* @throw QueueException if Queue is empty. \*/

template<class ItemType>

ItemType ListQueue<ItemType>::peekFront() const

{

if (theList.isEmpty())

throw QueueException ("QueueException: cannot peekFront from empty queue");

else

return theList.getEntry(1);

}