Andrew Tran

Lab Experience 12

**QueueTemplate2.h – Queue Template**

#include <iostream>

#include <iomanip>

#include <string>

#include <vector>

using namespace std;

#ifndef CIRCULARQ\_H

#define CIRCULARQ\_H

class Student {

public:

Student() { }

void getName(string username)

{

name = username;

}

void getID(int userid)

{

id = userid;

}

void display(ostream &out) const {

out << "Student Name: " << name << "\tStudent Id: " << id

<< "\tAddress: " << this << endl;

}

private:

string name;

int id;

};

typedef Student \*StudentPointer;

template <typename T>

class CircularQ {

private:

class Node {

public:

T data;

Node \*next;

Node(T d)

{

data = d;

next = NULL;

}// end Node Constructor

};

public:

typedef Node \* NodePointer;

CircularQ() { last = NULL; } // Create an empty Q

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Default Constructor creates an empty Queue

Pre: None

Post: An empty Queue is created and the last pointer is initialized to NULL

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void display(ostream &out) const; // Helper method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Display the entire Queue

Pre: Ostream out is open.

Post: Outputs the contents of the Queue in FIFO order

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CircularQ(const T &origQ); // Copy constructor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copy Constructor: creates a copy of the original Queue

Pre: Original is the Queue to be copied and is received as a const reference

parameter

Post: A copy of the original has been constructed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

~CircularQ(); // Destructor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Destructor: deletes the Queue

Pre: None

Post: The queue is destroyed and empty queue is created

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CircularQ &operator=(const T &rhs);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Overloaded Assignment Operator: Mimics the assignment operator

Pre: Original is the Queue to be copied and is received as a const reference

parameter

Post: The current Queue is replaced by a copy of the Queue of rhs and

a reference to it is returned.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void enqueue(T item); // put an item into the Q

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

enqueue: Inserts an item into the front of the Queue.

Pre: Queue must exist and the parameter item must be initialized.

Post: The item is inserted into the Queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

T front();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

front: Returns the first item in the queue.

Pre: Queue must exist

Post: Either an error message will be displayed if the Queue is empty or

the front of the Queue is returned.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void dequeue();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dequeue: Deletes the item located in the front of the Queue.

Pre: Checks to see if the Queue is empty

Post: Displays an error message if the Queue is empty otherwise the element

at the front of the Queue is deleted.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool empty() { return last == NULL; }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

empty: Checks to see if the Queue is empty

Pre: A queue must be created

Post: Returns either true if the Queue is empty, false otherwise.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private:

NodePointer last;

};

template <typename T>

ostream &operator<<(ostream& out, const CircularQ<T> &q);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

operator<<: Overloaded operator which allows you to display the contents

of the entire Queue

Pre: ostream must be opened and the const parameter is passed by reference

Post: Calls the method display that displays the entire Queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <typename T>

void CircularQ<T>::display(ostream &out) const {

NodePointer first; // Need a pointer

if (last == NULL)

cout << "Queue is empty \n\n";

else {

first = last->next; // no, intialize to first item in the list

do {

out << first->data << " "; // output the item

first = first->next; // move along

} while (first != last->next); // have we traversed the entire list?

}// end else

}//display method

template <typename T>

CircularQ<T>::~CircularQ() {

NodePointer first = 0;

NodePointer ptr = first;

while (ptr != 0)

{

first = ptr->next;

delete ptr;

ptr = first;

}

}

template <typename T>

void CircularQ<T>::enqueue(T item) {

NodePointer newNode = new Node(item); // Need a node for the Q

if (last == NULL) { // empty Q

last = newNode; // start constructing the Q

last->next = last; // pointer to itself since one item

}

else {

newNode->next = last->next; // Chain it

last->next = newNode;

last = newNode;

}

}// end enqueue

template <typename T>

CircularQ<T>::CircularQ(const T &origQ) { // Copy constructor

if (origQ.last != NULL) { // empty?

NodePointer origLast = origQ.last, // No. Create pointers to traverse Q's

origFirst = origLast->next,

newNode;

last = new Node(origFirst->data); // Initialize the new list

last->next = last; // set up the correct pointers

origFirst = origFirst->next; // Move to the next item in the list to copy

while (origFirst != origLast->next) { // Start traversing

newNode = new Node(origFirst->data); // create a new new node

newNode->next = last->next; // Start chaining it together

last->next = newNode;

last = newNode;

origFirst = origFirst->next; // go to the next one

}// end while

}// end if

}//end copy constructor

template <typename T>

CircularQ<T> &CircularQ<T>::operator=(const T &rhs) {

if (this != &rhs) { // Self-referential Assignment?

this->~CircularQ(); // No. Destoy LHS

if (rhs.last != NULL) { // rhs empty?

NodePointer rhsLast = rhs.last, // Create pointers for traversal

rhsFirst = rhsLast->next,

newNode;

last = new Node(rhsFirst->data); // Initialize the new LHS

last->next = last;

rhsFirst = rhsFirst->next; // go to the next RHS item

while (rhsFirst != rhsLast->next) { // Start traversing

newNode = new Node(rhsFirst->data); // Start the process of duplication

newNode->next = last->next;

last->next = newNode;

last = newNode;

rhsFirst = rhsFirst->next; // Go the next item in the RHS

}// end while

}// end if

}

return \*this; // Allow assignment chaining

}// end assignment operator

template <typename T>

T CircularQ<T>::front() {

T garbage = -9999999999999999;

if (empty()) {

cout << "Queue is empty returning garbage" << endl;

return garbage;

}

NodePointer first = last->next;

return (first->data);

}// return front of Q

template <typename T>

void CircularQ<T>::dequeue() {

if (empty())

cout << "Error ----- Cannot delete an item from an empty Queue"

<< endl << endl;

else {

NodePointer first = last->next;

if (first == last)

last = NULL; // Queue will be empty

else

last->next = first->next;

delete first; // eliminate the node

}

}// end dequeue

/\*----------------------------------------------------------------------------

Overloaded << operator

Preconditions: requires a constant reference and a Queue of type T

Postconditions: returns the ostream (for chaining)

description: this function is overloaded for outputing a queue with <<

----------------------------------------------------------------------------\*/

template <typename T>

ostream & operator<<(ostream &out, CircularQ<T> &s)

{

s.display(out);

return out;

}

/\* ------------------------------------------------------------------------

join

Preconditions: requires 2 queue values

Postconditions: appends queue2 to the end of queue1

description: this function joins 2 queues into 1.

------------------------------------------------------------------------\*/

template <typename T>

CircularQ<T> join(CircularQ<T> q1, CircularQ<T> q2)

{

CircularQ<T> q1Copy(q1), q2Copy(q2);

CircularQ<T> jQueue;

while (!q1Copy.empty())

{

jQueue.enqueue(q1Copy.front());

q1Copy.dequeue();

}

while (!q2Copy.empty())

{

jQueue.enqueue(q2Copy.front());

q2Copy.dequeue();

}

cout << "qdub: " << jQueue << endl << endl;

return jQueue;

}

/\*----------------------------------------------------------------------------

Overloaded << operator

Preconditions: requires a constant reference and a reference of type Student

Postconditions: none

description: this function is overloaded for outputing an object of type

Student.

----------------------------------------------------------------------------\*/

ostream & operator<<(ostream &out, Student &s) {

s.display(out);

return out;

}

/\*----------------------------------------------------------------------------

Overloaded << operator

Preconditions: requires a constant reference and a reference of a pointer to

a Student object.

Postconditions: none

description: this function is overloaded for outputing pointers to Students

----------------------------------------------------------------------------\*/

ostream & operator<<(ostream &out, StudentPointer &s)

{

s->display(out);

return out;

}

#endif

**Driver.cpp – DRIVER**

#include <iostream>

#include "QueueTemplate2.h"

using namespace std;

int main() {

double a;

Student s;

CircularQ<double> qdub; //qdub1 and qdub2 joined together

CircularQ<double> qdub1; //contains doubles 0.0,1.1,2.2,...,9.9

CircularQ<double> qdub2; //contains 5 doubles

CircularQ<CircularQ<double>> qq; //a queue of queues qdub1, qdub2, and qdub

CircularQ<string> qstr;

CircularQ<Student> qstu;

CircularQ<StudentPointer> qptr;

char answer;

string name;

int id;

StudentPointer stuPtr = 0;

cout << "Beginning Part 2" << endl;

cout << "Feature 1 - qdub1 and qdub2" << endl << endl;

for (int i = 0; i < 10; i++)

{

a = i\*1.1;

qdub1.enqueue(a);

}

cout << "qdub1: " << qdub1 << endl;

for (int i = 0; i < 5; i++)

{

cout << "Input a number for qdub2: " << endl;

cin >> a;

qdub2.enqueue(a);

}

cout << "qdub2: " << qdub2 << endl << endl;

cout << "Feature 2 - joining qdub1 and qdub2" << endl << endl;

//begin joining qdub1 and qdub2

qdub = join(qdub1, qdub2);

cout << "Feature 3 - Student objects" << endl << endl;;

for (;;)

{

cout << "Enter the name of your student " << endl;

cin >> name;

s.getName(name);

cout << "Enter the ID of your student " << endl;

cin >> id;

s.getID(id);

qstu.enqueue(s);

cout << "qstu: " << qstu << endl << endl;

cout << "Do you want to enter another Student? n to quit" << endl;

cin >> answer;

if (answer == 'n')

{

break;

}

}

for (;;)

{

stuPtr = new Student();

cout << "Feature 4 - Student pointers" << endl << endl;

cout << "Enter the name of your student " << endl;

cin >> name;

stuPtr->getName(name);

cout << "Enter the ID of your student " << endl;

cin >> id;

stuPtr->getID(id);

qptr.enqueue(stuPtr);

cout << "qptr: " << qptr << endl << endl;

answer = 0;

cout << "Do you want to enter another Student? n to quit" << endl;

cin >> answer;

if (answer == 'n')

{

break;

}

}

delete stuPtr;

cout << "Feature 5 - Queues of queues" << endl << endl;

//begin queue of queue of qdub, qdub1, and qdub2

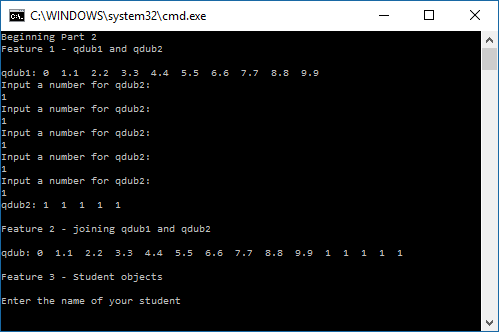
//qq.enqueue(qdub2);

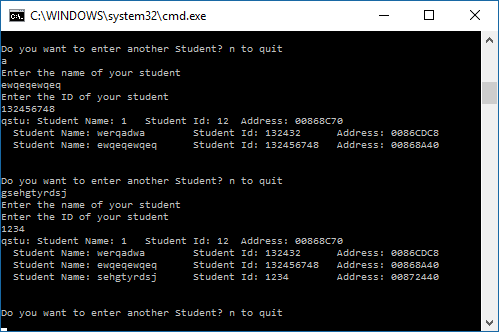
//qq.enqueue(qdub1);

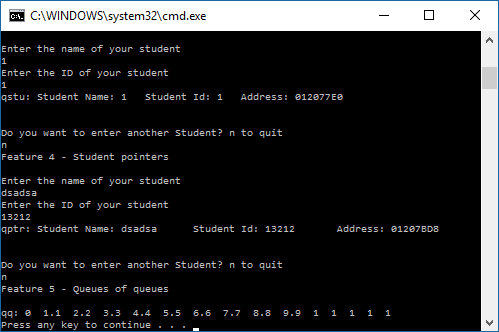
qq.enqueue(qdub);

cout << "qq: " << qq << endl;

}







**fibonacci.cpp – Fibonacci**

#include "Timer.h"

#include <iostream>

using namespace std;

unsigned iterFibonacci(unsigned n);

/\*---------------------------------------------------------

An iterative (nonrecursive) Fibonacci number calculator

Precondition: n >= 1.

Postcondition: n-th Fibonacci number is returned.

------------------------------------------------------------\*/

unsigned recFibonacci(unsigned n);

/\*---------------------------------------------------------

A recursive) Fibonacci number calculator

Precondition: n >= 1.

Postcondition: n-th Fibonacci number is returned.

------------------------------------------------------------\*/

int main()

{

Timer t;

unsigned x;

cout << "Please enter a positive integer: ";

cin >> x;

// Now print the result of the iterative version of the function

t.start();

cout << "Iterative fib(" << x << ") = " << iterFibonacci(x) << endl;

t.stop();

cout << "Time taken: " << t << endl;

// Now print the result of the recursive version of the function

t.start();

cout << "Recursive fib(" << x << ") = " << recFibonacci(x) << endl;

t.stop();

cout << "Time taken: " << t << endl;

}

//--- Definition of iterFibonacci()

unsigned iterFibonacci(unsigned n)

{

int

nextFib = 1, // the next Fibonacci number to be calculated

previousFib = 1, // the Fibonacci number before it

beforePreviousFib; // the Fibonacci number before that one

for (int i = 3; i <= n; i++)

{

// First, update the previous and before previous values

beforePreviousFib = previousFib;

previousFib = nextFib;

// Then compute the next Fibonacci value

nextFib = previousFib + beforePreviousFib;

}

return nextFib;

}

//--- Definition of recFibonacci()

unsigned recFibonacci(unsigned n)

{

if (n == 0)

return 0;

if (n == 1)

return 1;

return recFibonacci(n - 1) + recFibonacci(n - 2);

}

