#include <iostream>

#include <chrono>

#include <limits>

#include "linkedlist.h"

using namespace std;

struct CircArrayQueue

{

//core variables

int back;

int front;

int length;

int\* Array;

//member functions

int peek();

int size();

void enQueue(int value);

int deQueue();

//Let's make a constructor. This will initialize our main array.

CircArrayQueue(int size = 10)

{

//define a standard size

int number = 10;

//increase as needed

while (size > number) {

number \*= 2;

}

//set our other variables

front = back = -1;

length = size;

//create the main array

Array = new int[number];

}

//Time for a destructor

~CircArrayQueue() {

delete[] Array;

}

//Gave up on the copy constructor

/\*CircArrayQueue& operator =(CircArrayQueue& newArray) {

for (int i = 0; i < length; i++) {

int num = this->deQueue();

newArray.enQueue(num);

}

return newArray;

}\*/

};

//First, I did the size function because it was easy. It just returns length.

int CircArrayQueue::size()

{

return length;

}

//enqueue adds a value to the "queue"

void CircArrayQueue::enQueue(int value)

{

//check if the array is full here by using the mod function. This works because the array is circular, so even if front is double of length, it still works.

if ((back == (front - 1) % (length - 1)) or (front == 0 and back == length - 1))

{

int\* new\_array = new int [length \* 2];

for (int x = 0; x < length; x++) {

new\_array[x] = Array[x];

}

length \*= 2;

Array = new\_array;

}

//Check if it's empty

else if (front == -1)

{

front = 0;

back = 0;

Array[back] = value;

}

//Other cases

else if (back == length - 1 and front != 0)

{

back = 0;

Array[back] = value;

}

else

{

back++;

Array[back] = value;

}

}

int CircArrayQueue::peek()

{

//Cant return anything if it's empty right?

if (front == -1)

{

return NULL;

}

//But you can if it's not, so:

return Array[front];

}

//time to remove stuff

int CircArrayQueue::deQueue()

{

//can't remove from an empty array

if (front == -1)

{

return NULL;

}

//set a number so we can return at the end

int data = Array[front];

Array[front] = -1;

//Is it empty after we remove?

if (front == back)

{

front = -1;

back = -1;

}

//Is it just barely too full?

else if (front == length - 1)

front = 0;

//Otherwise

else

front++;

//return

return data;

}

//--------------------------------------------------Now for the Linked List implementation ------------------------------------------------

void LinkedListQueue::enQueue(int score) {

\_marker = new MyNode(score);

if (empty()) {

\_headPtr = \_marker;

}

else {

\_tailPtr->setNext(\_marker);

}

\_tailPtr = \_marker;

free(\_marker);

\_size++;

}

int LinkedListQueue::deQueue()

{

int x = 0;

struct MyNode\* ptr;

if (\_headPtr == NULL)

{

cout << "This is an empty list!" << endl;

return NULL;

}

else if (\_headPtr->next() == \_headPtr)

{

x = \_headPtr->score();

\_headPtr = NULL;

free(\_headPtr);

}

else

{

ptr = \_headPtr;

if (ptr->next() != \_headPtr)

{

ptr = ptr->next();

}

x = ptr->score();

ptr = \_headPtr;

free(ptr);

}

return x;

}

int LinkedListQueue::peek()

{

int x = 0;

struct MyNode\* ptr;

if (\_headPtr == NULL)

{

cout << "This is an empty list!" << endl;

return NULL;

}

else if (\_headPtr->next() == \_headPtr)

x = \_headPtr->score();

else

{

ptr = \_headPtr;

if (ptr->next() != \_headPtr)

{

ptr = ptr->next();

}

x = ptr->score();

free(ptr);

}

return x;

}

int MyNode::score(){

return \_score;

}

LinkedListQueue::LinkedListQueue() {

\_size = 0;

\_headPtr = nullptr;

\_tailPtr = nullptr;

\_prevPtr = nullptr;

\_marker = nullptr;

}

LinkedListQueue::~LinkedListQueue() {

\_marker = \_headPtr;

while (\_marker != NULL) {

\_prevPtr = \_marker->next();

delete \_marker;

\_marker = \_prevPtr;

}

\_headPtr = NULL;

\_tailPtr = NULL;

\_size = 0;

}

size\_t LinkedListQueue::size() {

return \_size;

}

bool LinkedListQueue::empty() {

return \_headPtr == nullptr;

}

MyNode\* LinkedListQueue::head() {

return \_headPtr;

}

MyNode::MyNode(int score) {

\_score = score;

\_nextPtr = nullptr;

}

void MyNode::setNext(MyNode\* other) {

\_nextPtr = other;

}

MyNode\* MyNode::next() {

return \_nextPtr;

}

int main()

{

chrono::high\_resolution\_clock::time\_point t1 = chrono::high\_resolution\_clock::now();

LinkedListQueue q();

cout << q.enQueue(25);

chrono::high\_resolution\_clock::time\_point t2 = chrono::high\_resolution\_clock::now();

chrono::duration<double> time\_span = chrono::duration\_cast<chrono::duration<double>>(t2 - t1);

std::cout << "It took me " << time\_span.count() << " seconds.";

std::cout << std::endl;

return 0;

}