Федеральное государственное автономное образовательное учреждение высшего образования
«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Queue

Выполнил:
студент ф-та ИТММ гр. 3821Б1ФИ3

Кулагин А.С.

Проверил:
зав. лабораторией СТиВВ

Лебедев И.Г.

Нижний Новгород
2022 г.

Содержание

Введение

Лабораторная работа направлена на практическое освоение динамической структуры данных Очередь. С этой целью в лабораторной работе изучаются различные варианты структуры хранения очереди и разрабатываются методы и программы решения задач с использованием очередей. В качестве области приложений выбрана тема эффективной организации выполнения потока заданий на вычислительных системах.

Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях. В качестве примера использования очереди предлагается задача разработки системы имитации однопроцессорной ЭВМ. Рассматриваемая в рамках лабораторной работы схема имитации является одной из наиболее простых моделей обслуживания заданий в вычислительной системе и обеспечивает тем самым лишь начальное ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

1. Постановка задачи

Имитационное моделирование реально существующих объектов и явлений – физических, химических, биологических, социальных процессов, живых и неживых систем, инженерных конструкций, конструируемых объектов – представляет собой построение математической модели, которая описывает изучаемое явление с достаточной точностью, и последующую реализацию разработанной модели на ЭВМ для проведения вычислительных экспериментов с целью изучения свойств моделируемых явлений. Использование имитационного моделирования позволяет проводить изучение исследуемых объектов и явлений без проведения реальных (натурных) экспериментов. Очередь (англ. queue), – схема запоминания информации, при которой каждый вновь поступающий ее элемент занимает крайнее положение (конец очереди). При выдаче информации из очереди выдается элемент, расположенный в очереди первым (начало очереди), а оставшиеся элементы продвигаются к началу; следовательно, элемент, поступивший первым, выдается первым.

2. Руководство пользователя

Проект queue содержит в себе класс очереди TQueue, TJobStream и TProc. Узнать о них можно в руководстве программиста. Если кратко, ниже приведен принцип работы очереди (файл sample_queue.cpp).

```
1  // ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
2  //
3  //
4  //
5  // Тестирование очереди
6
7  #include <iostream>
8  #include "tqueue.h"
9
10 int main(int argc, char* argv[])
11 {
12   TQueue<int> q(3);
13   q.Push(12);
14   std::cout << q.TopPop() << std::endl;
15   q.Push(12);
16   q.Push(13);
17   q.Push(14);
18   q.Pop();
19   q.Push(15);
20   std::cout << q.Top() << std::endl;
21   q.Pop();
22   q.Push(16);
23   std::cout << q.Top() << std::endl;
24   return 0;
25 }
```

Рисунок 1: Образец очереди.

Очевидно, в 13 строке на консоли будет 12, в 20 строке будет 13, в 23 строке будет 14.

3. Руководство программиста

3.1. Описание структуры программы

Проект queue содержит примеры в проектах sample_proc и sample_queue.

Проект queue содержит 4 класса.

Глобальная переменная size_t MAX_QUEUE_SIZE = 100000000 — максимальный размер массива.

3.1.1. Класс TQueueItem<typename T>.

3.1.1.1. Публичные поля

T value – значение элемента.

size_t priority – приоритет элемента.

3.1.2. Класс TJobStream

3.1.2.1. Публичные методы, конструкторы и деструкторы:

TJobStream(double JobIntens) – Конструктор инициализации интенсивности потока задач. JobIntense от 0 до 1.

double GetJobIntens(void) const noexcept – возвращает интенсивность потока задач

size_t GetNewJob(void) – сгенерировать работу с вероятностью JobIntens. 0, если не сгенерировалось, id≠0, если сгенерировалось.

size_t GetLastJobId(void) const noexcept – возвращает id последней сгенерированной работы. Если 0, то ещё не было сгенерировано ни одной работы.

3.1.2.2. Защищённые поля:

double JobIntense – интенсивность потока задач. От 0 до 1.

size_t LastJobId – id последней сгенерированной работы. Если 0, то ещё не было сгенерировано ни одной работы.

3.1.3. Класс TProc.

3.1.3.1. Публичные методы, конструкторы и деструкторы:

TProc(double rate, size_t MaxJobs) – Конструктор инициализации производительности процессора rate (от 0 до 1) с вместимостью MaxJobs работ.

double GetRate(void) const noexcept – возвращает производительность процессора.

size_t IsProcBusy(void) const noexcept – показывает, занят ли процессор выполнением работ. 0, если не занят. Id работы, если занят.

bool IsProcFull(void) const noexcept – переполнена ли очередь работ в процессоре.

double GetRate(void) const noexcept – возвращает производительность процессора.

size_t RunNewJob(size_t JobId) – добавляет новую работу в очередь процессора. Если очередь заполнена, возвращает 0. Если попала «пустая» работа (работа с id 0), возвращает ~0. В противном случае возвращает JobId.

size_t DoJob(void) – если очередь пуста, возвращает 0. В противном случае пытается выполнить работу. Выполненная работа удаляется из очереди. Возвращает всегда первую работу очереди.

size_t GetJobsDone(void) const noexcept – возвращает значение поля GetJobsDone.

size_t GetOverFlowTimes(void) const noexcept – возвращает значение поля OverFlowTimes.

size_t GetNoJobsTimes(void) const noexcept – возвращает значение поля NoJobsTimes.

3.1.3.2. Защищённые поля:

double ProcRate – производительность процессора. От 0 до 1.

TQueue<size_t> Jobs – очередь работ

size_t JobsDone – счётчик сколько работ выполнено

size_t OverflowTimes – счётчик сколько раз пытались добавить в заполненную очередь работу

size_t NoJobsTimes – счётчик сколько было тактов простоя

3.2.Описание алгоритмов

Моделирование потока задач. Для моделирования момента появления нового задания можно использовать значение датчика случайных чисел. Если значение датчика меньше некоторого порогового значения q_1, $0 \leq q_1 \leq 1$, то считается, что на данном такте имитации в вычислительную систему поступает новое задание (тем самым параметр q1 можно интерпретировать как величину, регулирующую интенсивность потока заданий – новое задание генерируется в среднем один раз за $(1/q_1)$ тактов).

Моделирование момента завершения обработки очередного задания также может быть выполнено по описанной выше схеме. При помощи датчика случайных чисел формируется еще одно случайное значение, и если это значение меньше порогового значения q_2, $0 \leq q_2 \leq 1$,то принимается, что на данном такте имитации процессор завершил обслуживание очередного задания и готов приступить к обработке задания из очереди ожидания (тем самым параметр q_2 можно интерпретировать как величину, характеризующую производительность процессора вычислительной системы – каждое задание обслуживается в среднем за $(1/q_2)$ тактов).

Очередь есть динамическая структура, операции вставки и удаления переводят очередь из одного состояния в другое, при этом добавление новых элементов осуществляется в конец очереди, а извлечение – из начала очереди (дисциплина обслуживания «первым пришел – первым обслужен».

В этой лабораторной работе очередь имеет кольцевой буфер — структура хранения, получаемая из вектора расширением отношения следования парой $p(a_n,a_1)$.

Алгоритм добавления элемента в конец очереди аналогичен со стеком, но с учётом приоритета, путём сдвига всех элементов приоритета ниже, элемент добавляется над ними. Увеличить счётчик количества элементов на 1.

Алгоритм удаления первого элемента очереди заключается в перемещении указателя начала очереди и уменьшения количества элементов.

Схема имитации процесса поступления и обслуживания заданий в вычислительной системе состоит в следующем.

1. Каждое задание в системе представляется некоторым однозначным идентификатором (например, порядковым номером задания).

2. Для проведения расчетов фиксируется (или указывается в диалоге) число тактов работы системы.

3. На каждом такте опрашивается состояние потока задач и процессор.

4. Регистрация нового задания в вычислительной системе может быть сведена к запоминанию идентификатора задания в очередь ожидания процессора. Ситуацию переполнения очереди заданий следует понимать как нехватку ресурсов вычислительной системы для ввода нового задания (отказ от обслуживания).

5. Для моделирования процесса обработки заданий следует учитывать, что процессор может быть занят обслуживанием очередного задания, либо же может находиться в состоянии ожидания (простоя).

6. В случае освобождения процессора предпринимается попытка его загрузки. Для этого извлекается задание из очереди ожидания.

7. Простой процессора возникает в случае, когда при завершении обработки очередного задания очередь ожидающих заданий оказывается пустой.

4. Эксперименты

Таблица 1: Работа системы имитации однопроцессорной ЭВМ

Номер эксперимента	Количество тактов	Размер очереди	Интенсивность потока задач	Производительность процессора	Создано задач	Выполнено задач	Количество перепол нений задач	Количество тактов простоя	Сколько в среднем тактов за задачу
1	10	3	0,5	0,5	5	2	0	3	5
2	10	3	0,5	0,2	4	2	0	4	5
3	10	3	0,2	0,5	2	1	0	8	10

Теоретические оценки времени работы арифметических операций совпадают с полученными программой.

5. Заключение

Удалось написать очередь и систему имитации однопроцессорной ЭВМ. Удалось протестировать и вывести следующие характеристики системы имитации однопроцессорной ЭВМ:

- Количество поступивших в вычислительную систему заданий в течение всего процесса имитации;
- Количество отказов в обслуживании заданий из-за переполнения очереди - целесообразно считать процент отказов как отношение количества не поставленных в очередь заданий к общему количеству сгенерированных заданий, умноженное на 100%;
- Среднее количество тактов выполнения задания;
- Количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания – целесообразно считать процент простоя процессора как отношение количества тактов простоя процессора к общему количеству тактов имитации, умноженное на 100%.

6. Литература

1. Лекции Сысоева Александра Владимировича, 2022.
2. Практики и лабораторные Лебедева Ильи Геннадьевича, 2022.
3. Лабораторный практикум. Составители: Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с.

7. Приложения

7.1. Приложение 1

```
// ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
//
//
//
//

#ifndef __TQueue_H__
#define __TQueue_H__

#include <iostream>

const size_t MAX_QUEUE_SIZE = 100000000;

template <typename T>
class TQueueItem
{
public:
	T value;
	size_t priority;
};

template <typename T>
class TQueue
{
protected:
	TQueueItem<T>* pMem;
	size_t MemSize;
	size_t Hi;
	size_t Li;
	size_t DataCount;
	const static size_t unzero = ~0;
public:
	TQueue(size_t sz = 10);
	TQueue(const TQueue& q);
	TQueue(TQueue&& q) noexcept;
	TQueue& operator=(const TQueue& q);
```

```
	TQueue& operator=(TQueue&& q) noexcept;
	~TQueue();

	bool IsEmpty(void) const noexcept;
	bool IsFull(void) const noexcept;

	void Free(void) noexcept;
	void Push(const T& e, size_t priority = 0);
	T Top(void) const;
	void Pop(void);
	T TopPop(void);

	bool operator==(const TQueue& q) const noexcept;
	bool operator!=(const TQueue& q) const noexcept;
};

template<typename T>
inline TQueue<T>::TQueue(size_t sz) : Hi(0), Li(0), pMem(nullptr), DataCount(0)
{
	if (sz == 0)
		throw std::out_of_range("Queue size should be greater than zero");
	if (sz > MAX_QUEUE_SIZE)
		throw	std::out_of_range("Queue	size	should	be	less	than
MAX_QUEUE_SIZE");
	MemSize = sz;
	pMem = new TQueueItem<T>[MemSize];
}

template<typename T>
inline TQueue<T>::TQueue(const TQueue<T>& q)
{
	if (q.pMem == nullptr)
	{
		MemSize = 0;
		pMem = nullptr;
		Free();
	}
	else
	{
		MemSize = q.MemSize;
		Hi = q.Hi;
		Li = q.Li;
```

```
            DataCount = q.DataCount;
            pMem = new TQueueItem<T>[MemSize];
            if (!(q.IsEmpty()))
            {
                  size_t j = Hi;
                  for (size_t i = 0; i < DataCount; i++)
                  {
                        pMem[j] = q.pMem[j];
                        j--;
                        if (j == unzero)
                              j = MemSize - 1;
                  }
            }
      }
}

template<typename T>
inline TQueue<T>::TQueue(TQueue<T>&& q) noexcept
{
      this->operator=(q);
}

template<typename T>
inline TQueue<T>& TQueue<T>::operator=(const TQueue<T>& q)
{
      if (this == &q)
            return *this;
      if (MemSize != q.MemSize)
      {
            TQueueItem<T>* tmp = new TQueueItem<T>[q.MemSize];
            delete[] pMem;
            MemSize = q.MemSize;
            pMem = tmp;
      }
      Hi = q.Hi;
      Li = q.Li;
      DataCount = q.DataCount;
      if (!(q.IsEmpty()))
      {
            size_t j = Hi;
            for (size_t i = 0; i < DataCount; i++)
            {
```

```
                    pMem[j] = q.pMem[j];
                    j--;
                    if (j == unzero)
                          j = MemSize - 1;
             }
      }
      return *this;
}

template<typename T>
inline TQueue<T>& TQueue<T>::operator=(TQueue<T>&& q) noexcept
{
      pMem = nullptr;
      MemSize = 0;
      Free();
      std::swap(pMem, q.pMem);
      std::swap(MemSize, q.MemSize);
      std::swap(Hi, q.Hi);
      std::swap(Li, q.Li);
      std::swap(DataCount, q.DataCount);
      return *this;
}

template<typename T>
inline TQueue<T>::~TQueue()
{
      delete[] pMem;
      MemSize = 0;
      Free();
}

template<typename T>
inline bool TQueue<T>::IsEmpty(void) const noexcept
{
      return DataCount == 0;
}

template<typename T>
inline bool TQueue<T>::IsFull(void) const noexcept
{
      return DataCount == MemSize;
}
```

```
template<typename T>
inline void TQueue<T>::Free(void) noexcept
{
	Hi = 0;
	Li = 0;
	DataCount = 0;
}

template<typename T>
inline void TQueue<T>::Push(const T& e, size_t priority)
{

	if (IsFull())
		throw std::out_of_range("Queue is full");
	size_t i = Li, j = Li - 1;
	while (i != Hi)
	{
		i = (i != ~0) ? i : MemSize - 1;
		j = (j != ~0) ? j : MemSize - 1;
		if (priority <= pMem[j].priority)
			break;
		TQueueItem<T> tmp = pMem[i];
		pMem[i] = pMem[j];
		pMem[j] = tmp;
		i--;
		j--;
	}
	pMem[i].value = e;
	pMem[i].priority = priority;
	Li++;
	Li = (Li < MemSize) ? Li : 0; // : Li - MemSize; // Но Li - MemSize всегда 0.
	DataCount++;
}

template<typename T>
inline T TQueue<T>::Top(void) const
{
	if (IsEmpty())
		throw "Queue is empty";
	return pMem[Hi].value;
}
```

```
template<typename T>
inline void TQueue<T>::Pop(void)
{
	if (IsEmpty())
		throw "Queue is empty";
	//Hi = (Hi + 1) % MemSize;
	Hi++;
	Hi = (Hi < MemSize) ? Hi : 0; // : Hi - MemSize; // Но Hi - MemSize всегда 0.
	DataCount--;
}

template<typename T>
inline T TQueue<T>::TopPop(void)
{
	T tmp = Top();
	Pop();
	return tmp;
}

template<typename T>
inline bool TQueue<T>::operator==(const TQueue& q) const noexcept
{
	if (DataCount != q.DataCount)
		return false;
	size_t j = Hi, k = q.Hi;
	for (size_t i = 0; i < DataCount; i++)
	{
		if (pMem[j].value != q.pMem[k].value || pMem[j].priority != q.pMem[j].priority)
			return false;
		j--;
		if (j == unzero)
			j = MemSize - 1;
		k--;
		if (k == unzero)
			k = q.MemSize - 1;
	}
	return true;
}

template<typename T>
```

```
inline bool TQueue<T>::operator!=(const TQueue& q) const noexcept
{
	return !(this->operator==(q));
}

#endif
```

tqueue.h

```
#include "tqueue.h"
```

tqueue.cpp

```
// ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
//
//
//
//

#ifndef __TJobStream_H__
#define __TJobStream_H__

#include <cstdlib>

class TJobStream
{
private:
	double JobIntense;
	size_t LastJobId;
public:
	TJobStream(double JobIntens);
	double GetJobIntens(void) const noexcept;
	size_t GetNewJob(void);
	size_t GetLastJobId(void) const noexcept;
};

#endif
```

TJobStream.h

```
#include "TJobStream.h"

TJobStream::TJobStream(double JobIntense)
{
	if (JobIntense >= 0.0 && JobIntense <= 1.0)
	{
		this->JobIntense = JobIntense;
		LastJobId = 0;
	}
	else
		throw "Job Intens must be between 0.0 and 1.0";
}

double TJobStream::GetJobIntens(void) const noexcept
{
	return JobIntense;
}

size_t TJobStream::GetNewJob(void)
{
	if (static_cast<double>(std::rand()) / static_cast<double>(RAND_MAX) <=
JobIntense)
```

```
	{
		LastJobId++;
		return LastJobId - 1;
	}
	else
		return 0;
}

size_t TJobStream::GetLastJobId(void) const noexcept
{
	return LastJobId;
}
```

TJobStream.cpp

```
// ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
//
//
//
//

#ifndef __TProc_H__
#define __TProc_H__

#include "tqueue.h"
#include <cstdlib>

class TProc
{
private:
	double ProcRate;
	TQueue<size_t> Jobs;
	size_t JobsDone;
	size_t OverflowTimes;
	size_t NoJobsTimes;
public:
	TProc(double Rate, size_t MaxJobs);
	double GetRate(void) const noexcept;
	size_t IsProcBusy(void) const noexcept;
	bool IsProcFull(void) const noexcept;
	size_t RunNewJob(size_t JobId);
	size_t DoJob(void);
	size_t GetJobsDone(void) const noexcept;
	size_t GetOverflowTimes(void) const noexcept;
	size_t GetNoJobsTimes(void) const noexcept;
};

#endif
```

TProc.h

```
#include "TProc.h"

TProc::TProc(double Rate, size_t MaxJobs) : Jobs(MaxJobs)
{
	if (Rate >= 0.0 && Rate <= 1.0)
	{
		ProcRate = Rate;
		JobsDone = 0;
		OverflowTimes = 0;
		NoJobsTimes = 0;
	}
	else
		throw "Proc rate must be between 0.0 and 1.0";
}
```

```
double TProc::GetRate(void) const noexcept
{
	return ProcRate;
}

size_t TProc::IsProcBusy(void) const noexcept
{
	if (Jobs.IsEmpty())
		return 0;
	else
		return Jobs.Top();
}

bool TProc::IsProcFull(void) const noexcept
{
	return Jobs.IsFull();
}

size_t TProc::RunNewJob(size_t JobId)
{

	if (JobId > 0)
	{
		if (IsProcFull())
		{
			OverflowTimes++;
			return 0;
		}
		else
		{
			Jobs.Push(JobId);
			return JobId;
		}
	}
	else
		return ~0;
}

size_t TProc::DoJob(void)
{
	if (!IsProcBusy())
	{
		NoJobsTimes++;
		return 0;
	}
	else if (static_cast<double>(std::rand()) / static_cast<double>(RAND_MAX) <=
ProcRate && IsProcBusy())
	{
		size_t LastJob = Jobs.TopPop();
		JobsDone++;
		return LastJob;
	}
	else
		return Jobs.Top();
}

size_t TProc::GetJobsDone(void) const noexcept
{
	return JobsDone;
}

size_t TProc::GetOverflowTimes(void) const noexcept
{
	return OverflowTimes;
```

```
}

size_t TProc::GetNoJobsTimes(void) const noexcept
{
	return NoJobsTimes;
}
```

TProc.cpp

```
// ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
//
//
//
// Тестирование очереди

#include <iostream>
#include "tqueue.h"

int main(int argc, char* argv[])
{
	TQueue<int> q(3);
	q.Push(12);
	std::cout << q.TopPop() << std::endl;
	q.Push(12);
	q.Push(13);
	q.Push(14);
	q.Pop();
	q.Push(15);
	std::cout << q.Top() << std::endl;
	q.Pop();
	q.Push(16);
	std::cout << q.Top() << std::endl;
	return 0;
}
```

sample_queue.cpp

```
// ННГУ, ИИТММ, Курс "Алгоритмы и структуры данных"
//
//
//
// Тестирование вычислительной системы

#include <iostream>
#include "TProc.h"
#include "TJobStream.h"


int main(int argc, char* argv[])
{
	std::srand(std::time(nullptr));
	const size_t tests = 3;
	TProc proc[tests] = { TProc(0.5, 3), TProc(0.2, 3), TProc(0.5, 3) };
	TJobStream stream[tests] = { TJobStream(0.5), TJobStream(0.5),
TJobStream(0.2) };
	size_t tacts[tests] = { 10, 10, 10 };
	TQueue<size_t> jobsqueue(10);

	for (size_t i = 0; i < tests; i++)
	{
		std::cout << tacts[i] << " tacts, queue size is 3, JobsIntense is " <<
stream[i].GetJobIntens() << ", proc rate is " << proc[i].GetRate() << '.' <<
std::endl;
		for (size_t j = 0; j < tacts[i]; j++)
```

```
            {
                  size_t temp_job = stream[i].GetNewJob();
                  if (temp_job > 0)
                        jobsqueue.Push(temp_job);
                  if (!jobsqueue.IsEmpty())
                        if (proc[i].RunNewJob(jobsqueue.Top()))
                              jobsqueue.Pop();
                  proc[i].DoJob();
            }
            jobsqueue.Free();
            std::cout << "Jobs Generated " << stream[i].GetLastJobId() << ", Proc
Jobs Done " << proc[i].GetJobsDone() << ", Proc Jobs Overflow " <<
proc[i].GetOverflowTimes() << ", Proc No Jobs Tacts " << proc[i].GetNoJobsTimes() <<
",       average       "       <<       (static_cast<double>(tacts[i])       /
static_cast<double>(proc[i].GetJobsDone())) << " tacts done per job." << std::endl <<
std::endl;
      }
      return 0;
}
```

sample_proc.cpp

```
#include "tqueue.h"

#include <gtest.h>

TEST(TQueue, can_create_queue_with_positive_length)
{
      ASSERT_NO_THROW(TQueue<int> q(5));
}

TEST(TQueue, cant_create_too_large_queue)
{
      ASSERT_ANY_THROW(TQueue<int> q(MAX_QUEUE_SIZE + 1));
}

TEST(TQueue, throws_when_create_queue_with_negative_length)
{
      ASSERT_ANY_THROW(TQueue<int> q(-5));
}

TEST(TQueue, can_create_copied_queue)
{
      TQueue<int> s1(10);
      ASSERT_NO_THROW(TQueue<int> q2(s1));
}

TEST(TQueue, copied_queue_is_equal_to_source_one)
{
      TQueue<int> q1(10);
      q1.Push(2);
      q1.Push(5);
      q1.Push(11);
      TQueue<int> q2(q1);
      EXPECT_EQ(true, q1 == q2);
}

TEST(TQueue, copied_queue_has_its_own_memory)
{
      TQueue<int> q1(10);
      for (int i = 0; i < 3; i++)
            q1.Push(i);
      TQueue<int> q2(q1);
      q2.Pop();
      q2.Push(123);
```

```
	EXPECT_EQ(true, q1 != q2);
}

TEST(TQueue, can_push_and_top_element)
{
	TQueue<int> q(5);
	q.Push(5);
	q.Push(6);
	EXPECT_EQ(5, q.Top());
}

TEST(TQueue, can_pop_element)
{
	TQueue<int> q(5);
	q.Push(5);
	q.Push(6);
	q.Pop();
	EXPECT_EQ(6, q.Top());
}

TEST(TQueue, can_top_pop_element)
{
	TQueue<int> q(5);
	q.Push(5);
	q.Push(6);
	EXPECT_EQ(5, q.TopPop());
	EXPECT_EQ(6, q.TopPop());
}

TEST(TQueue, knows_if_empty)
{
	TQueue<int> q(5);
	EXPECT_EQ(true, q.IsEmpty());
	q.Push(11);
	EXPECT_EQ(false, q.IsEmpty());
	q.Pop();
	EXPECT_EQ(true, q.IsEmpty());
}

TEST(TQueue, knows_if_full)
{
	TQueue<int> q(5);
	EXPECT_EQ(false, q.IsFull());
	q.Push(0);
	EXPECT_EQ(false, q.IsFull());
	for (int i = 1; i < 5; i++)
		q.Push(i * 2);
	EXPECT_EQ(true, q.IsFull());
}

TEST(TQueue, throws_when_push_when_full)
{
	TQueue<int> q(5);
	for (int i = 0; i < 5; i++)
		q.Push(i * 2);
	ASSERT_ANY_THROW(q.Push(0));
}

TEST(TQueue, throws_when_pop_when_empty)
{
	TQueue<int> q(5);
	q.Push(0);
	q.Pop();
	ASSERT_ANY_THROW(q.Pop());
}
```

```
TEST(TQueue, can_assign_queue_to_itself)
{
	TQueue<int> q(5);
	ASSERT_NO_THROW(q = q);
}

TEST(TQueue, can_assign_queue_of_equal_size)
{
	TQueue<int> q1(5), q2(5);
	q1.Push(15);
	q1.Push(16);
	q2 = q1;
	q2.Pop();
	q2.Pop();
	q2.Push(16);
	q2.Push(15);
	EXPECT_NE(15, q2.TopPop());
	EXPECT_NE(16, q2.Top());
}

TEST(TQueue, can_assign_queue_of_different_size)
{
	TQueue<int> q1(2), q2(1);
	q1.Push(100);
	q1.Push(200);
	q2.Push(0);
	q2 = q1;
	EXPECT_EQ(100, q2.TopPop());
	EXPECT_EQ(200, q2.TopPop());
}

TEST(TQueue, compare_equal_queues_return_true)
{
	TQueue<int> q1(5), q2(5);
	q1.Push(5);
	q1.Push(6);
	q1.Push(7);

	q2.Push(5);
	q2.Push(6);
	q2.Push(7);
	EXPECT_EQ(true, q1 == q2);
}

TEST(TQueue, compare_queue_with_itself_return_true)
{
	TQueue<int> q(5);
	q.Push(0);
	EXPECT_EQ(true, q == q);
}

TEST(TQueue, queues_with_different_size_are_not_equal)
{
	TQueue<int> q1(5), q2(5);
	q1.Push(5);
	q1.Push(6);
	q2.Push(5);
	EXPECT_EQ(true, q1 != q2);
}

TEST(TQueue, same_queues_but_with_different_memory_size_are_equal)
{
	TQueue<int> q1(5), q2(10);
	q1.Push(5);
```

```
	q1.Push(6);
	q2.Push(5);
	q2.Push(6);
	EXPECT_EQ(true, q1 == q2);
}

TEST(TQueue, test_queue_cercular_buffer)
{
	TQueue<int> q(4);
	q.Push(12);
	q.Push(13);
	q.Push(14);
	q.Push(15);
	EXPECT_EQ(true, q.IsFull());
	q.Pop();
	EXPECT_EQ(false, q.IsFull());
	ASSERT_NO_THROW(q.Push(16));
	q.Pop();
	ASSERT_NO_THROW(q.Push(17));
	EXPECT_EQ(14, q.TopPop());
	EXPECT_EQ(15, q.TopPop());
	EXPECT_EQ(16, q.TopPop());
	EXPECT_EQ(17, q.TopPop());
	EXPECT_EQ(true, q.IsEmpty());
}

TEST(TQueue, simple_queue_priority)
{
	TQueue<int> q(4);
	q.Push(12);
	q.Push(13, 1);
	EXPECT_EQ(13, q.Top());
	q.Push(14, 2);
	EXPECT_EQ(14, q.Top());
	q.Push(15, 1);
	EXPECT_EQ(14, q.Top());
	q.Pop();
	EXPECT_EQ(13, q.Top());
	q.Pop();
	EXPECT_EQ(15, q.Top());
	q.Push(99, 99);
	EXPECT_EQ(99, q.Top());
}
```

test_queue.cpp