МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра программной инженерии

Работа допущена к защите

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Руководитель

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

**КУРСОВАЯ РАБОТА**

по дисциплине «Алгоритмы и структуры данных»

на тему: «Приоритетные очереди: возможные реализации и сравнение подходов»

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Щекотихин С.Е.

Шифр 170590

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 71ПГ

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Фролов А.И.

Оценка: «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_» Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Орел 2019

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 4

1 ОПИСАНИЕ И АНАЛИЗ ЗАДАЧИ 5

2 ПОДОХОДЫ К РЕАЛИЗАЦИИ ПРИОРИТЕТНЫХ ОЧЕРЕДЕЙ 6

2.1 Приоритетная очередь на неотсортированном связном списке 6

2.2 Приоритетная очередь на отсортированном массиве 6

2.3 Приоритетная очередь на пирамиде 6

3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 8

4 РЕАЛИЗАЦИЯ ПРОГРАММНОГО СРЕДСТВА 13

5 ОЦЕНКА СЛОЖНОСТИ ПО ВРЕМЕНИ 16

ЗАКЛЮЧЕНИЕ 19

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 20

ПРИЛОЖЕНИЕ А (обязательное) ИСХОДНЫЙ КОД 21

ВВЕДЕНИЕ

Очень часто возникает потребность в организации очередей. Но не простых очередей, где обслуживание проводится по принципу «первый пришел — первый вышел», а на основе приоритетов персон, находящихся в очереди. Такие очереди получили название приоритетных. Данный тип очередей применяется как в обслуживании процессов, так и при обработке различных писем, заявок и т.д.

Целью курсовой работы является реализация программного средства, позволяющего принимать и обрабатывать заявки, требующие различной длительности обслуживания.

Для выполнения поставленной цели необходимо выполнить следующие задачи:

* описать и проанализировать задачу;
* рассмотреть различные подходы к реализации приоритетных очередей;
* спроектировать и реализовать программное средство;
* дать оценку сложности по времени.

1 ОПИСАНИЕ И АНАЛИЗ ЗАДАЧИ

Необходимо реализовать программное средство, позволяющее принимать и обрабатывать заявки, характеризующиеся различной длительностью обслуживания. Наивысшим приоритетом обладают заявки, которые требуют наименьшего времени на обслуживание.

Проанализировав данную задачу, можно сделать вывод о том, что нам необходима очередь заявок. Причем не простая очередь, а очередь с приоритетами. Приоритетная очередь —  это абстрактный тип данных, наподобие стека, где у каждого элемента есть свой приоритет. Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди. Для очереди с приоритетами характерна функция, которая позволяет вычислить приоритет элемента. К примеру , где — элемент. Приоритет элемента, в общем случае, принадлежит множеству действительных чисел, а в более общем случае — некоторому линейному упорядоченному множеству.

Таким образом, необходимо реализовать программное средство, использующее очереди с приоритетами, которое позволяет добавлять и обрабатывать поступающие заявки.

2 ПОДОХОДЫ К РЕАЛИЗАЦИИ ПРИОРИТЕТНЫХ ОЧЕРЕДЕЙ

Существуют различные подходы к реализации приоритетной очереди. Рассмотрим и выберем из них наиболее эффективный.

2.1 Приоритетная очередь на неотсортированном связном списке

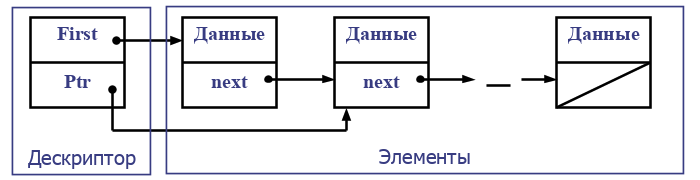
Приоритетную очередь можно реализовать с помощью связного списка. Связный список — последовательность, элементы которой связаны между собой указателями, хранящимися в самих элементах. Данная реализация является неоднозначной, поскольку с одной стороны, вставка нового элемента осуществляется за , а с другой стороны, поиск элемента с минимальным или максимальным приоритетом выполняется за . На рисунке 1 связный список представлен на физическом уровне.

Рисунок 1 — Связный список на физическом уровне

2.2 Приоритетная очередь на отсортированном массиве

Отсортированный массив — упорядоченная последовательность элементов одного типа, называемого базовым.

Реализация приоритетной очереди на отсортированном массиве позволяет нам находить элемент за , так как наибольший элемент является первым. Однако при вставке нового элемента в массив необходимо будет просматривать, в среднем, около половины элементов списка, то есть эффективность поиска будет равна .

2.3 Приоритетная очередь на пирамиде

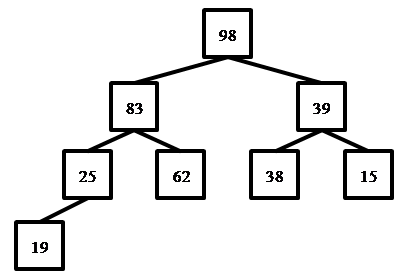
Пирамида — это структура данных типа дерево (бинарное), для элементов которого выполняется дополнительное условие (основное свойство пирамиды): значение элемента в родительском узле больше (более точно — не меньше) значений во всех узлах потомках. Глубина листьев пирамиды отличается не более чем на 1 слой. Общий вид пирамиды показан на рисунке 2.

Рисунок 2 — Структура данных — пирамида

Реализация приоритетной очереди на пирамиде позволяет добиться больших успехов в эффективности, чем на массиве и списке. Так, и вставка элемента в пирамиду будет осуществляться за , и поиск элемента в пирамиде будет осуществляться за .

Рассмотрев каждый из этих подходов, можно выяснить, что наиболее эффективной является реализация приоритетной очереди на основе пирамиды, так как другие виды реализаций позволяют быстро осуществлять только одну из главных операций приоритетных очередей: либо вставку элемента, либо поиск элемента.

3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

Рассмотрим структуру элемента очереди. В очередь на обработку будут поступать заявки, следовательно структура элемента очереди должна содержать поле, содержащее текст поступившей заявки. У каждой заявки есть время, необходимое для ее обработки. Каждой заявке при поступлении присваивается уникальный номер, необходимый для однозначной идентификации. Так же структура должна содержать поле, хранящее значение приоритета элемента. Приоритет элемента необходимо рассчитывать исходя из того, сколько ему необходимо времени на обслуживание. Чем меньше времени необходимо элементу на обслуживание, тем выше должен быть его приоритет. К примеру, можно задать исходное множество приоритетов, которые будут присваиваться по следующему: правилу пусть все заявки требующие меньше 10 секунд на обслуживание имеют приоритет 100, тогда заявки требующие от 10 до 20 секунд имеют приоритет 90 и так далее; или же использовать формулу , где — приоритет элемента, а — время, требуемое на обработку. Исходя из всего вышесказанного можно сделать вывод, что необходимо реализовать структуру типа «Заявка», являющуюся элементом приоритетной очереди, со следующими полями:

* «ID» — уникальный идентификатор;
* «Text» — содержимое заявки;
* «Time» — исходное время, необходимое для обработки;
* «CurrentTime» — оставшееся время, необходимое для обработки;
* «Priority» — приоритет заявки в очереди.

Поскольку реализуемая нами приоритетная очередь основана на пирамиде, рассмотрим как можно ее реализовать. Пирамиду можно реализовать как c помощью одномерного массива в последовательной памяти, так и с помощью указателей в связной памяти. Разницы между этими подходами немного. К примеру, в случае реализации на указателях у элемента пирамиды будет ссылка на своего родителя и правого и левого потомков, а при реализации на массиве индекс родителя и потомков можно рассчитать исходя из индекса самого элемента в массиве. Так как отличий между подходами немного, то для дальнейшей реализации приоритетной очереди будет использоваться пирамида, реализованная на массиве.

Итак, для реализации пирамиды необходимо разработать процедуры, позволяющие определять родителя и потомков для конкретного элемента: «Parent», «LeftChild» и «RightChild»; а так же процедуры позволяющие поддерживать основное свойство пирамиды: «Up» — поднять элемент вверх по пирамиде, и «Down» — опустить элемент вниз по пирамиде. Поддерживать основное свойство пирамиды необходимо в связи с тем, что при добавлении или удалении элемента его приоритет может быть выше чем у родителя, и тогда его надо поднять, или ниже чем у одного из потомков, и тогда его надо опустить. Алгоритмы работы данных процедур приведены на рисунках 3 и 4.

Рисунок 3 — Блок-схема алгоритма работы процедуры «Up»

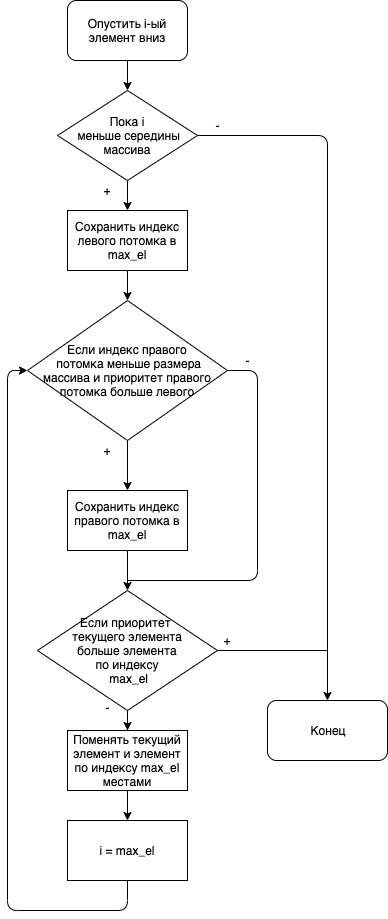
Для реализации приоритетной очереди необходимо реализовать следующие процедуры:

Рисунок 4 — Блок-схема алгоритма работы процедуры «Down»

* «Insert» — вставка элемента в очередь;
* «Priority» — расчет приоритета элемента;
* «Max» — найти максимальный элемент очереди;
* «Delete» — удаление элемента из очереди.

Рассмотрим алгоритмы работы данных процедур.

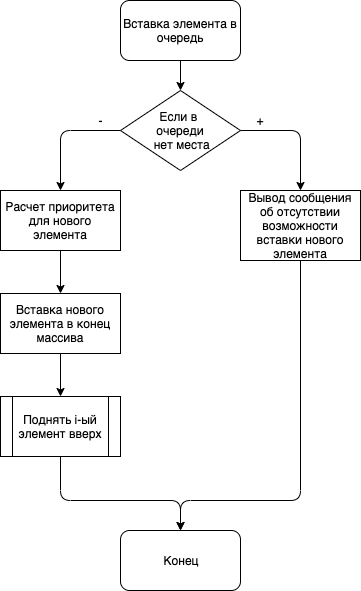
Поскольку проектируемая нами приоритетная очередь реализуется на пирамиде, то это значительно облегчает поиск элемента с максимальным приоритетом — процедура «Max», так как этим элементом будет являться корень пирамиды (исходит из основного свойства пирамиды). За основу процедуры «Priority» будет взята формула, рассмотренная ранее — , где — приоритет элемента , а — время, необходимое элементу на обслуживание. Одной из основных процедур приоритетной очереди является вставка — «Insert». При вставке элемента в очередь необходимо рассчитать его приоритет с помощью процедуры «Priority», вставить его в конец массива и вызвать процедуру «Up», так как необходимо восстановить основное свойство пирамиды, если оно было нарушено. Алгоритм работы данной процедуры представлен на рисунке 5.

Рисунок 5 — Блок-схема алгоритма работы процедуры «Insert»

После того как элемент был обработан, в нашем случае закончилось время элемента, необходимое на обработку, он должен быть удален из очереди. За это отвечает вторая, не менее важная чем вставка, процедура — «Delete». При удалении элемента из пирамиды проще всего поменять корень, который нам необходимо удалить, и последний элемент местами. Затем удалить последний элемент массива и вызвать процедуру «Down» для корня, так как могло быть нарушено основное свойство пирамиды. На рисунке 6 приведен алгоритм работы данной процедуры.

Рисунок 6 — Блок-схема алгоритма работы процедуры «Delete»

4 РЕАЛИЗАЦИЯ ПРОГРАММНОГО СРЕДСТВА

В качестве языка для реализации программного средства был выбран язык C++. С++ — компилируемый, статически типизированный язык программирования. Поддерживает процедурную и объектно-ориентированную парадигмы программирования. С++ сочетает в себе свойства как высокоуровневых, так и низкоуровневых языков. Для реализации программного средства данный язык был выбран благодаря следующим достоинствам:

* высокая производительность;
* поддержка объектно-ориентированного программирования;
* возможность низкоуровневой работы с памятью.

Рассмотрим как были реализованы основные структуры данных и процедуры программного средства. Для реализации структуры типа «Заявка» был разработан класс «Request» (файл request.h), часть исходного кода которого представлена ниже.

class Request {

private:

int id;

int priority;

std::string text;

int time;

int remainTime;

public:

Request(std::string text, int time, int priority) {

this->id = rand() % 1000 + 1;

this->text = text;

this->priority = priority;

this->time = time;

this->remainTime = time;

}

int Tick() {

sleep(1);

this->remainTime--;

return this->remainTime;

};

Все заявки хранятся в абстрактном типе данных — приоритетная очередь, которая реализована с помощью класса «PriorityQueue» (файл priority\_queue.h). Данный класс имеет поле «requests» — указатель на массив со всеми заявками, а также поле «size» — текущий размер очереди. Реализации основных процедур приоритетной очереди — вставки и удаления, представлены ниже.

bool Insert(Request request) {

if (this->size + 1 > this->sz) return false;

\*(this->requests + size++) = request;

this->up(this->size - 1);

return true;

}

bool Delete() {

if (this->size == 0) return false;

std::swap(\*(this->requests + 0), \*(this->requests + --this->size));

this->down(0);

return true;

}

Поскольку приоритетная очередь реализована с помощью структуры данных — пирамида, то была необходимость реализовать процедуры «up» и «down», позволяющие поддерживать основное свойство пирамиды. Исходный код операции «up» представлен ниже.

void up(int pos) {

while (pos != 0 &&

(this->requests + pos)->GetPriority() > (this->requests + this->parent(pos))->GetPriority()) {

std::swap(\*(this->requests + pos), \*(this->requests + this->parent(pos)));

pos = this->parent(pos);

}

}

С исходным кодом процедуры «down» можно ознакомиться в приложении А.

Для нахождения родительского и дочерних элементов в массиве были разработаны функции, принимающие на вход позицию элемента в массиве и возвращающее индекс соответствующего элемента. Ниже приведена их реализация.

static int parent(int pos) { return (pos - 1) / 2; }

static int leftChild(int pos) { return 2 \* pos + 1; }

static int rightChild(int pos) { return 2 \* pos + 2; }

Рассмотрим реализацию пользовательского интерфейса. Пользовательский интерфейс приложения реализован в консоли. После запуска приложения перед пользователем предстает меню со следующими пунктами:

1. Добавить заявку в очередь;
2. Запустить обработку заявок;
3. Запуск тесты;
4. Выход.

Первые два пункта позволяют пользователю сымитировать работу конвейера по обработке заявок. С помощью третьего пункта можно проверить эффективность работы процедур вставки и удаления в приоритетную очередь. Ниже приведён пример обработки 2 заявок.

На обработке 808

Осталось 5 4 3 2 1 0

На обработке 250

Осталось 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Все заявки обработаны

Для тестов был создан массив с размерами приоритетной очереди и разработана процедура заполнения очереди тестовыми данными. Ниже представлена их реализация.

int sizes[] = { 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120 };

void initData(PriorityQueue \*&priorityQueue, int size) {

Request \*requests = new Request[size];

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

\*(requests + i) = Request("", 10, i);

}

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

priorityQueue->Insert(\*(requests + i));

}

}

5 ОЦЕНКА СЛОЖНОСТИ ПО ВРЕМЕНИ

Оценка сложности алгоритмов — это трудоемкость (то, сколько времени требует алгоритм на выполнение) и требуемый ему объем выделяемой памяти.

В приоритетной очереди вся сложность сводится к вставке элемента в набор данных и удалению элемента из этого набора. При выборе структуры данных для реализации приоритетной очереди были рассмотрены отсортированный массив, неотсортированный связный список и пирамида. Так вставка элемента в список — , в массив — , в пирамиду — ; удаление элемента из списка — , из массива — , из пирамиды — . Рассмотрим данные оценки эффективностей более подробно.

— константное время. Время выполнения программы не зависит от количества элементов.

— линейное время. Время выполнения программы линейно зависит от количества элементов.

— логарифмическое время. Время выполнения программы увеличивается с ростом количества элементов.

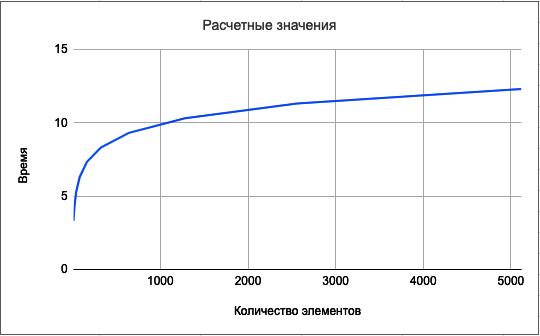
На рисунке 7 приведен график зависимости времени выполнения алгоритмов вставки и удаления элемента от количества элементов в очереди.

Рисунок 7 — График зависимости времени выполнения алгоритмов от количества элементов в очереди

Для проверки эффективности реализации при выполнении тестов засекалось время, необходимое на вставку нового элемента в очередь и удаление элемента из очереди. В ходе тестов были получены результаты, представленные в таблице 1.

Таблица 1 — Результаты, полученные в ходе тестирования.

|  |  |  |
| --- | --- | --- |
| Количество элементов | Время на вставку, с | Время на удаление, с |
| 10 | 0,0002 | 0,0001 |
| 20 | 0,0002 | 0,0001 |
| 40 | 0,0003 | 0,0002 |
| 80 | 0,0004 | 0,0002 |
| 160 | 0,0005 | 0,0003 |
| 320 | 0,0006 | 0,0003 |
| 640 | 0,0006 | 0,0003 |
| 1280 | 0,0007 | 0,0004 |
| 2560 | 0,0008 | 0,0004 |
| 5120 | 0,0008 | 0,0005 |

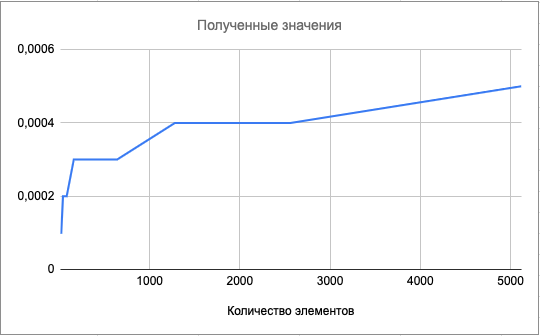
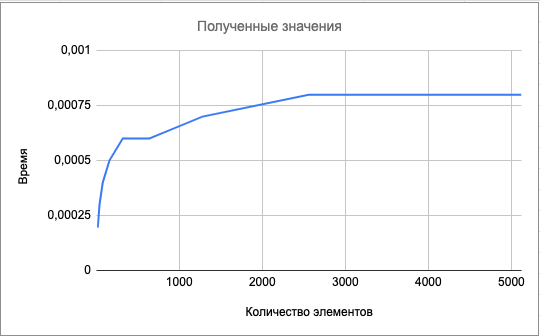
 На основе полученных данных построим графики зависимости трудоёмкости алгоритмов от количества элементов в очереди и изобразим их на рисунках 8 и 9 соответственно для алгоритма вставки и алгоритма удаления.

Рисунок 9 — График зависимости времени выполнения алгоритма удаления от количества элементов в очереди

Рисунок 8 — График зависимости времени выполнения алгоритма вставки от количества элементов в очереди

Проанализировав и сравнив полученные графики с графиком зависимости времени выполнения алгоритмов от количества элементов в очереди, можно сделать вывод о том, что сложность реализованных алгоритмов вставки и удаления элемента близка к , а благодаря высокой производительности языка С++ получается добиться значительного прироста в скорости, что в итоге сказывается на уменьшении времени, требуемого на выполнение.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было разработано программное средство для обработки заявок.

Для достижения цели была описана и проанализирована задача, рассмотрены различные подходы к реализации приоритетных очередей, спроектировано и реализовано программное средство, а также дана оценка оценка сложности по времени.

Так как каждая поставленные задачи были выполнены, курсовую работу можно считать завершенной.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ахо, А. Структуры данных и алгоритмы [Текст]/ А. Ахо,Д. Хопкрофт, Д. Ульман. — М.: Издательский дом «Вильямс», 2000. — 384 с.
2. Вирт, Н. Алгоритмы и структуры данных [Текст]/ Н. Вирт. –М.: Мир, 1989. – 360 с.
3. Кнут, Д. Искусство программирования для ЭВМ; в 3 т.; Основные алгоритмы [Текст]/ Д. Кнут. – М.: Мир, 1976. – 734 c.

ПРИЛОЖЕНИЕ А

(обязательное)

ИСХОДНЫЙ КОД

Файл request.h

#ifndef CODE\_REQUEST\_H

#define CODE\_REQUEST\_H

#include <string>

#include <ctime>

#include <cstdlib>

#include <unistd.h>

#include <sstream>

class Request {

private:

int id;

int priority;

std::string text;

int time;

int remainTime;

public:

Request() {};

Request(std::string text, int time, int priority) {

this->id = rand() % 1000 + 1;

this->text = text;

this->priority = priority;

this->time = time;

this->remainTime = time;

}

int Tick() {

sleep(1);

this->remainTime--;

return this->remainTime;

}

int GetPriority() { return this->priority; }

int Id() { return this->id; }

int RemainTime() { return this->remainTime; }

};

#endif //CODE\_REQUEST\_H

Файл priority\_queue.h

#ifndef CODE\_PRIORITY\_QUEUE\_H

#define CODE\_PRIORITY\_QUEUE\_H

#include "request.h"

class PriorityQueue {

private:

Request \*requests;

int size = 0;

int sz = 0;

// Процедуры для нахождения родителя и потомков элемента в массиве

static int parent(int pos) { return (pos - 1) / 2; }

static int leftChild(int pos) { return 2 \* pos + 1; }

static int rightChild(int pos) { return 2 \* pos + 2; }

// Процедуры по работе с пирамидой

void up(int pos) {

while (pos != 0 &&

(this->requests + pos)->GetPriority() > (this->requests + this->parent(pos))->GetPriority()) {

std::swap(\*(this->requests + pos), \*(this->requests + this->parent(pos)));

pos = this->parent(pos);

}

}

void down(int pos) {

while (pos < this->size / 2) {

int max\_offset = this->leftChild(pos);

if (this->rightChild(pos) < size && (this->requests + this->rightChild(pos))->GetPriority() > (this->requests + max\_offset)->GetPriority()) {

max\_offset = this->rightChild(pos);

}

if ((this->requests + pos)->GetPriority() > (this->requests + max\_offset)->GetPriority()) {

return;

}

std::swap(\*(this->requests + pos), \*(this->requests + max\_offset));

pos = max\_offset;

}

}

public:

PriorityQueue(int size) {

this->requests = new Request[size];

this->sz = size;

this->size = 0;

}

bool Insert(Request request) {

if (this->size + 1 > this->sz) return false;

\*(this->requests + size++) = request;

this->up(this->size - 1);

return true;

}

Request\* Max() {

return (this->requests + 0);

}

bool Delete() {

if (this->size == 0) return false;

std::swap(\*(this->requests + 0), \*(this->requests + --this->size));

this->down(0);

return true;

}

int Size() { return this->size; }

static int CalcPriority(int time) { time = (time == 0) ? 1 : time; return 1000 / time; }

};

#endif //CODE\_PRIORITY\_QUEUE\_H

Файл main.cpp

#include <iostream>

#include "priority\_queue.h"

int sizes[] = { 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120 };

void initData(PriorityQueue \*&priorityQueue, int size) {

Request \*requests = new Request[size];

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

\*(requests + i) = Request("", 10, i);

}

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

priorityQueue->Insert(\*(requests + i));

}

}

int main() {

PriorityQueue \*pQ = new PriorityQueue(128);

while (true) {

int command;

std::cout << "1 - Добавить заявку в очередь" << std::endl;

std::cout << "2 - Запустить обработку заявок" << std::endl;

std::cout << "3 - Запустить тесты" << std::endl;

std::cout << std::endl << "0 - Выход" << std::endl;

std::cout << std::endl << "Введите команду: ";

std::cin >> command;

switch (command) {

case 1: {

std::string text;

int time;

std::cout << std::endl << "Текст заявки: ";

std::cin >> text;

std::cout << std::endl << "Время на обработку: ";

std::cin >> time;

pQ->Insert(Request(text, time, pQ->CalcPriority(time)));

break;

}

case 2: {

while (pQ->Size() > 0) {

std::cout << "На обработке " << pQ->Max()->Id() << std::endl;

std::cout << "Осталось ";

while (pQ->Max()->RemainTime() > 0) {

pQ->Max()->Tick();

std::cout << pQ->Max()->RemainTime() << " ";

}

std::cout << std::endl;

pQ->Delete();

}

std::cout << "Все заявки обработаны" << std::endl;

break;

}

case 3: {

std::cout << "Тесты запущены" << std::endl;

for (int testNumber = 0; testNumber < sizeof(sizes) / sizeof(int); ++testNumber) {

PriorityQueue \*priorityQueue = new PriorityQueue(sizes[testNumber]);

std::cout << std::endl << "Test number: " << testNumber << " ";

initData(priorityQueue, sizes[testNumber] - 1);

clock\_t begin\_time = clock();

priorityQueue->Insert(Request("", 3, 1000));

std::cout << "Insert time: " << (float(clock() - begin\_time) / **CLOCKS\_PER\_SEC**) \* 100;

begin\_time = clock();

priorityQueue->Delete();

std::cout << ", Delete time: " << (float(clock() - begin\_time) / **CLOCKS\_PER\_SEC**) \* 100;

}

std::cout << std::endl << "Тесты завершены" << std::endl;

break;

}

case 0: {

exit(0);

}

default: break;

}

}

return 0;

}