



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

---

## Отчёт

### по лабораторной работе №5

Название «Обработка очередей»

---

Дисциплина «Типы и структуры данных»

---

Вариант 1

Студент ИУ7-31Б

---

(подпись, дата)

Корниенко К.Ю.

(Фамилия И.О.)

Преподаватель

(подпись, дата)

Барышникова М.Ю.

(Фамилия И.О.)

Москва, 2021

## Содержание

Введение . . . . .	3
1 Аналитический раздел . . . . .	4
1.1 Теоритические сведения . . . . .	4
1.2 Реализации очереди . . . . .	4
1.2.1 В виде массива . . . . .	4
1.2.2 В виде линейного списка . . . . .	4
1.3 Описание условия задачи . . . . .	4
1.3.1 Техническое задание . . . . .	5
1.4 Вывод . . . . .	5
2 Конструкторский раздел . . . . .	6
2.1 Описание структур данных . . . . .	6
2.2 Описание алгоритма . . . . .	9
2.3 Теоретический расчет работы ОА . . . . .	10
2.4 Вывод . . . . .	10
3 Технологический раздел . . . . .	11
3.1 Требования к ПО . . . . .	11
3.2 Реализация алгоритмов . . . . .	11
3.3 Тестовые данные . . . . .	19
3.4 Вывод . . . . .	19
4 Экспериментальный раздел . . . . .	20
4.1 Результаты моделирования . . . . .	20
4.2 Постановка эксперимента . . . . .	20
4.2.1 Технические характеристики . . . . .	20
4.2.2 Описание экспериментальных данных . . . . .	20
4.3 Результат эксперимента . . . . .	21
4.3.1 Время работы . . . . .	21
4.3.2 Затрачиваемая память . . . . .	21
4.4 Анализ фрагментации . . . . .	21
5 Контрольные вопросы . . . . .	25
Заключение . . . . .	27

## **Введение**

**Целью данной лабораторной работы является** приобретение навыков работы с типом данных «очередь», представленным в виде одномерного массива и односвязного линейного списка, проведение сравнительного анализа реализации алгоритмов включения и исключения элементов из очереди при использовании указанных структур данных, оценка эффективности программы по времени и по используемому объему памяти.

**Для достижения поставленной цели необходимо решить следующие задачи:**

- 1) Исследовать реализации очередей.
- 2) Описать используемые структуры данных.
- 3) Описать алгоритм обработки данных в обслуживающем аппарате.
- 4) Описать требования к ПО.
- 5) Протестировать разработанное ПО.
- 6) Сравнить реализации очереди через вектор и в виде линейного списка.

## 1 Аналитический раздел

В данном разделе представлена основная информация об очереди и реализациях очереди в виде массива и линейного односвязного списка.

### 1.1 Теоритические сведения

Очередь – это последовательный список переменной длины, включение элементов в который идет с одной стороны (с «хвоста»), а исключение – с другой стороны (с «головы»). Принцип работы очереди: первым пришел – первым вышел, т. е. *First In – First Out (FIFO)*. Для «хвоста» и «головы» очереди используют соответственно два указателя *Pin* и *Pout*, то есть, исключается элемент с адресом *Pout* и включается элемент по адресу *Pin*.

### 1.2 Реализации очереди

Моделировать простейшую линейную очередь можно как на основе вектора (одномерного массива), так и на основе динамического списка.

#### 1.2.1 В виде массива

При моделировании простейшей линейной очереди на основе одномерного массива выделяется последовательная область памяти из  $m$  мест по  $L$  байт, где  $L$  – размер поля данных для одного элемента размещаемого типа. В каждый текущий момент времени выделенная память может быть вся свободна, занята частично или занята полностью. В начале процесса очередь пуста, при этом:  $P_{in} = P_{out} = Q_1$ , где  $Q_1$  – адрес (индекс) первого элемента очереди, а количество элементов очереди ( $K$ ) равно нулю. При включении очередного элемента в очередь он располагается по адресу  $P_{in}$ , а сам указатель  $P_{in}$  перемещается на длину типа данных к началу следующего элемента.

#### 1.2.2 В виде линейного списка

Большинство проблем, возникающих при реализации очереди в виде массива, устраняется при реализации очереди на основе односвязного линейного списка, каждый элемент которого содержит информационное поле и поле с указателем «вперед» (на следующий элемент). В этом случае в статической памяти можно либо хранить адрес начала и конца очереди, либо – адрес начала очереди и количество элементов. Исходное состояние очереди:  $P_{out} = P_{in}$  - пустой указатель,  $K = 0$ .

### 1.3 Описание условия задачи

Большинство проблем, возникающих при реализации очереди в виде массива, устраняется при реализации очереди на основе односвязного линейного списка, каждый элемент которого содержит информационное поле и поле с указателем «вперед» (на следующий элемент). В этом случае в статической памяти можно либо хранить адрес начала и конца очереди, либо – адрес начала очереди и количество элементов. Исходное состояние очереди:  $P_{out} = P_{in}$  - пустой указатель,  $K = 0$ .

Требуется смоделировать процесс обслуживания первых 1000 заявок первого типа, выдавая после обслуживания каждых 100 заявок первого типа информацию о текущей и средней длине каждой очереди и о среднем времени пребывания заявок каждого типа в очереди. В конце процесса необходимо выдать на экран общее время моделирования, время простоя ОА, количество вошедших в систему и вышедших из нее заявок первого и второго типов.

### 1.3.1 Техническое задание

Система массового обслуживания состоит из обслуживающего аппарата (ОА) и очереди заявок двух типов. (рисунок 1.1)

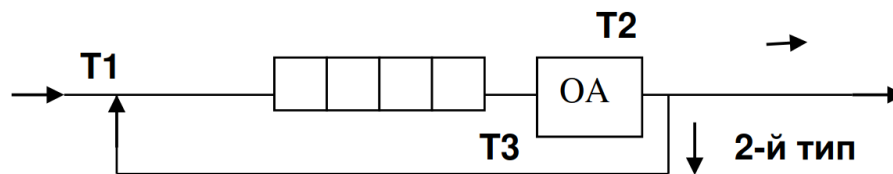


Рисунок 1.1 — Система массового обслуживания

Заявки 1-го типа поступают в "хвост" очереди по случайному закону с интервалом времени  $T_1$ , равномерно распределенным от 0 до 5 единиц времени (е.в.). В ОА они поступают из "головы" очереди по одной и обслуживаются также равновероятно за время  $T_2$  от 0 до 4 е.в., после чего покидают систему. Единственная заявка 2-го типа постоянно обращается в системе, обслуживаясь в ОА равновероятно за время  $T_3$  от 0 до 4 е.в. и возвращаясь в очередь не далее 4-й позиции от "головы". В начале процесса заявка 2-го типа входит в ОА, оставляя пустую очередь (все времена – вещественного типа). Смоделировать процесс обслуживания первых 1000 заявок 1-го типа. Выдавать после обслуживания каждых 100 заявок 1-го типа информацию о текущей и средней длине очереди, количестве вошедших и вышедших заявок и о среднем времени пребывания заявок в очереди. В конце процесса выдать общее время моделирования, время простоя аппарата, количество вошедших в систему и вышедших из нее заявок первого типа и количество обращений заявок второго типа. По требованию пользователя выдать на экран адреса элементов очереди при удалении и добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

### 1.4 Вывод

В данном разделе были рассмотрены основные теоритические сведения об очередях и реализациях очередей, а также сформулировано техническое задание.

## 2 Конструкторский раздел

В данном разделе будут описаны используемые структуры данных, приведен список функций для работы с ними, а также описан алгоритм моделирования работы ОА и теоритически рассчитаны результаты работы модели.

### 2.1 Описание структур данных

Ниже, на листинге 2.1, представлена структура очереди, реализованная на основе (кольцевого) массива, и функции для работы с данной структурой.

Листинг 2.1 — Структура очереди на массиве

```
1 typedef struct
2 {
3     uint32_t size;          // Размер очереди
4     uint32_t capacity;     // Вместимость очереди
5     request_t *begin;      // Начало массива, выделенного под очередь
6     request_t *first;      // "Хвост" очереди
7     request_t *last;       // "Голова" очереди
8 } qarr_t;
9
10 // Выделение памяти под пустую очередь
11 qarr_t qarr_create(uint32_t size);
12
13 // Очищение памяти под очередь
14 void qarr_destroy(qarr_t *queue);
15
16 // Вставка элемента в "хвост" очереди
17 int qarr_push_back(qarr_t *queue, request_t value);
18
19 // Вставка элемента в очередь
20 int qarr_insert(qarr_t *queue, uint32_t pos, request_t value);
21
22 // Удаление элемента из "головы" очереди
23 int qarr_pop_front(qarr_t *queue, request_t *value);
24
25 // Печать очереди на экран
26 void qarr_print(const qarr_t *queue);
```

Ниже, на листинге 2.2, представлена структура очереди, реализованная на основе линейного односвязного списка, и функции для работы с ней.

Листинг 2.2 — Структура очереди на линейном списке

```
1 typedef struct node
2 {
3     request_t data;         // Запрос ОА
4     struct node *next;     // Следующий узел
```

```

5 } node_t;
6
7
8 typedef struct
9 {
10     uint32_t size; // Размер очереди
11     node_t *first; // "Хвост" очереди
12     node_t *last; // "Голова" очереди
13 } qlist_t;
14
15
16 // Создание пустой очереди
17 qlist_t qlist_create(void);
18
19 // Очищение памяти под очередь
20 void qlist_destroy(qlist_t *queue);
21
22 // Вставка элемента в "хвост" очереди
23 int qlist_push_back(qlist_t *queue, request_t value);
24
25 // Вставка элемента в очередь
26 int qlist_insert(qlist_t *queue, uint32_t pos, request_t value);
27
28 // Удаление элемента из "головы" очереди
29 int qlist_pop_front(qlist_t *queue, request_t *value);
30
31 // Печать очереди на экран
32 void qlist_print(const qlist_t *queue);

```

Ниже, на листинге 2.3, представлена структура, являющаяся «оберткой» над структурами очередей, реализованных на основе массива и линейного односвязного списка, позволяющая работать с двумя структурами, используя один набор функций.

Листинг 2.3 — Универсальная структура очереди

```

1 /**
2  * Реализация очереди:
3  * ARRAY_QUEUE - через массив
4  * LIST_QUEUE  - через линейный односвязный список
5  */
6 typedef enum
7 {
8     ARRAY_QUEUE,
9     LIST_QUEUE
10 } qtype_t;
11
12
13 /**

```

```

14 * @brief Структура очереди
15 *
16 */
17 typedef struct
18 {
19     qtype_t type;          // Реализация очереди
20
21     union
22     {
23         qlist_t lst;       // Очередь на списке
24         qarr_t arr;        // Очередь на массиве
25     } imp;
26
27 } queue_t;
28
29 /**
30 * @brief Создание пустой очереди
31 *
32 * \param[in] type тип очереди (ARRAY_QUEUE или LIST_QUEUE)
33 * \param[in] size вместимость очереди (для ARRAY_QUEUE)
34 *
35 * \return Возвращается очередь: выделенная память под очередь
36 * на массиве или пустая очередь на списке
37 */
38 queue_t queue_create(qtype_t type, uint32_t size);
39
40 /**
41 * @brief Очищение памяти, занимаемой очередью
42 *
43 * @param queue очередь, память из-под которой следует очистить
44 */
45 void queue_destroy(queue_t *queue);
46
47 /**
48 * @brief Вставка элемента в "хвост" очереди
49 *
50 * @param queue очередь, в которую вставляется элемент
51 * @param value элемент для вставки
52 * @return код ошибки: 0 при успешном завершении.
53 */
54 int queue_push_back(queue_t *queue, request_t value);
55
56 /**
57 * @brief Вставка элемента в очередь
58 *
59 * @param queue очередь, в которую вставляется элемент
60 * @param pos позиция для вставки (отсчитывается от головы)

```



```

61  * @param value элемент для вставки
62  * @return код ошибки: 0 при успешном завершении.
63  */
64  int queue_insert(queue_t *queue, uint32_t pos, request_t value);
65
66  /**
67   * @brief Удаление элемента из "головы" очереди
68   *
69   * @param queue очередь, из которой удаляется элемент
70   * @param value значение удаляемого элемента
71   * @return код ошибки: 0 при успешном завершении.
72   */
73  int queue_pop_front(queue_t *queue, request_t *value);
74
75  /**
76   * @brief Печатает очередь в stdout
77   *
78   * @param queue очередь для печати
79   */
80  void queue_print(const queue_t *queue);
81
82  /**
83   * @brief Заполняет
84   *
85   * @param queue
86   * @param n
87   * @return int
88   */
89  int queue_fill(queue_t *queue, const int n);

```

## 2.2 Описание алгоритма

### Алгоритм моделирования:

- 1) Выбрать ближайшее событие (по времени) из следующих:
  - а) Приход заявки 1-го типа.
  - б) Завершение обработки текущей заявки аппаратом.
- 2) Реализовать выбранное событие.
- 3) Увеличить счётчик времени.
- 4) Уменьшить времена next\_request\_1\_time и oa\_time.
- 5) Рассчитать новое время до следующего события того же типа.

### 2.3 Теоретический расчет работы ОА

Ожидаемое время прихода заявок рассчитывается по формуле (2.1):

$$T_{in} = \langle T_1 \rangle \times N = \frac{t_2^{in} - t_1^{in}}{2} \times N \quad (2.1)$$

Подставляя в формулу (2.1) интервал прихода заявки второго типа – 0 ... 5, получаем:  $T_{in} = 2500$  е.в.

Ожидаемое время обработки заявок рассчитывается по формуле (2.2):

$$T_{out} = \langle T_2 \rangle \times N = \frac{t_2^{out} - t_1^{out}}{2} \times N \quad (2.2)$$

Подставляя в формулу (2.2) интервал обработки заявки: – 0 ... 4, получаем:  $T_{out} = 2000$  е.в.

### 2.4 Вывод

В данном разделе были описаны используемые структуры данных, приведен список функций для работы с ними, а также описан алгоритм моделирования работы ОА и теоритически рассчитаны результаты работы модели.

### 3 Технологический раздел

В данном разделе будут представлены листинги кодов реализации моделирования работы ОА, представлены требования к ПО и приведены тестовые данные.

#### 3.1 Требования к ПО

К разрабатываемому ПО предъявлены следующие требования:

- на вход программе подается режим работы ПО (пункт меню: 0 . . . 2);
  - если режим работы программы 0 – производится выход;
  - если режим работы программы 1 – производится моделирование работы ОА. На вход в данном режиме подается пункт меню (0 – выход):
    - 1) Запуск моделирования с параметрами по-умолчанию.
    - 2) Изменение времени прихода и обработки заявок с экрана, в случае чего на вход подаются новые времена прихода и обработки заявок.
  - если режим работы программы 2 – производится ручное тестирование работы очередей, на вход программе подается пункт меню (от 0 до 4, где 0 - выход):
    - 1) Вставка элемента в очередь.
    - 2) Удаление элемента из очереди.
    - 3) Вывод очереди.
- на выход программа выдает результаты моделирования (общее время моделирования, время простоя аппарата, количество вошедших в систему и вышедших из нее заявок и среднее время пребывания заявок в очереди, процент погрешности результата), если выбран режим работы (1), если выбран режим работы (2), то на выход выдаются затраты времени на вставку и удаление элемента из очереди для списка и очереди, а также выводятся на экран: очереди и адреса используемой списком памяти, с информацией о фрагментации.

#### 3.2 Реализация алгоритмов

Ниже, на листинге 3.1, представлены функции и структуры для работы модели.

Листинг 3.1 — Структуры и функции для моделирования

```
1  #include "utils/types.h"
2  #include "queue/queue.h"
3  #include <stddef.h>
4
5  /**
6   * @brief Структура с параметрами моделирования
7   */
8  typedef struct
9  {
10     time_intv_t t1;
```

```

11     time_intv_t t2;
12     time_intv_t t3;
13 } params_t;
14
15 /**
16  * @brief Структура с результатами моделирования
17  */
18 typedef struct
19 {
20     int status;
21     double total_time;
22     double push_time;
23     double dispatch_time;
24     size_t requests_processed;
25     size_t request2_cycles;
26
27     size_t queue_size;
28     double avg_qsize_enumerator;
29     double avg_qsize_denominator;
30     size_t requests_entered;
31     size_t requests_leaved;
32
33     double average_wait_time;
34     double oa_downtime;
35 } model_result_t;
36
37 /**
38  * @brief Временные настройки по-умолчанию
39  *
40  * @return структура с параметрами
41  */
42 params_t model_default_params(void);
43
44 /**
45  * @brief Сбрасывает состояние моделируемого аппарата.
46  * Очищает очередь, сбрасывает накопленные данные
47  *
48  * @param capacity - вместимость рабочей очереди
49  * @return SUCCESS - успешная инициализация модели,
50  *         MEM_ERR - ошибка при выделении памяти.
51  */
52 int model_reset(size_t capacity);
53
54 void print_results(model_result_t result, size_t requests_count);
55
56 /**
57  * @brief Моделирование работы обслуживающего аппарата

```

```

58 *
59 * @param requests_amount - число заявок 1го типа, которые требуется обслужить
60 * @param params - параметры моделирования
61 * @return структура с результатами моделирования
62 */
63 model_result_t model_run(size_t requests_amount, params_t params);

```

Ниже, на листинге 3.2, представлена реализации алгоритма работы модели.

Листинг 3.2 — Реализация алгоритма работы модели

```

1  #include <stdbool.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "utils/errors.h"
5  #include "model.h"
6
7  #define MAX(a,b) (((a)>(b))?(a):(b))
8
9  typedef struct
10 {
11     queue_t queue_lst;
12     queue_t queue_arr;
13
14     model_result_t results;
15
16     // прошедшее время с момента начала моделирования
17     double curr_time;
18
19     // сколько времени осталось до реализации следующих событий
20     double request_time; // приход заявки 1го типа
21     double oa_time;      // окончание обработки заявки в ОА
22
23     // текущая обрабатываемая ОА заявка
24     request_t dispatching_request;
25 } model_t;
26
27 static model_t model;
28
29 params_t model_default_params(void)
30 {
31     params_t params;
32
33     params.t1.min = 0.0;
34     params.t1.max = 5.0;
35     params.t2.min = 0.0;
36     params.t2.max = 4.0;
37     params.t3.min = 0.0;

```

```

38     params.t3.max = 4.0;
39
40     return params;
41 }
42
43 static void _model_results_reset(model_result_t* results)
44 {
45     results->status = SUCCESS;
46     results->total_time = 0.0;
47     results->push_time = 0.0;
48     results->dispatch_time = 0.0;
49     results->requests_processed = 0;
50     results->request2_cycles = 0;
51
52     results->queue_size = 0;
53     results->avg_qsize_enumerator = 0.0;
54     results->avg_qsize_denominator = 0.0;
55
56     results->requests_entered = 0;
57     results->requests_leaved = 0;
58
59     results->average_wait_time = 0.0;
60     results->oa_downtime = 0.0;
61 }
62
63 int model_reset(size_t capacity)
64 {
65     queue_destroy(&model.queue_arr);
66     queue_destroy(&model.queue_lst);
67     _model_results_reset(&model.results);
68
69     model.queue_arr = queue_create(ARRAY_QUEUE, capacity);
70     model.queue_lst = queue_create(LIST_QUEUE, capacity);
71
72     model.curr_time = 0.0;
73     model.request_time = 0.0;
74     model.oa_time = 0.0;
75
76     return model.queue_arr.imp.arr.capacity == 0 ? MEM_ERR : SUCCESS;
77 }
78
79 void print_results(model_result_t result, size_t requests_count)
80 {
81     printf("Результаты моделирования после обработки %ld заявок:\n",
82           requests_count + 1);
83     printf("Время прихода: %.2lf е.в.\n", result.push_time);
84     printf("Время обслуживания: %.2lf е.в.\n", result.dispatch_time);

```

```

84     printf("Кол-во обработанных заявок: %lu\n", result.requests_processed);
85     printf("Кол-во обращений заявки 2го типа: %lu\n", result.request2_cycles);
86     printf("Время простоя %.2lf е.в.\n", result.push_time -
            result.dispatch_time);
87 }
88
89 static inline double _random_time(time_intv_t interval)
90 {
91     const int resolution = 1001;
92     return interval.min + (rand() % resolution) * (interval.max - interval.min)
            / (resolution - 1);
93 }
94
95 // Отправка новой заявки для обработки в хвост очереди.
96 // Здесь уместно фиксировать реальное время обработки очередей
97 static int _model_send_request(model_t *model, request_t request)
98 {
99     int status = queue_push_back(&model->queue_arr, request);
100     if (status == SUCCESS)
101         status = queue_push_back(&model->queue_lst, request);
102     return status;
103 }
104
105 // Извлечение заявки из головы очереди.
106 // Здесь уместно фиксировать реальное время обработки очередей
107 static bool _model_receive_request(model_t *model, request_t *request)
108 {
109     int status = queue_pop_front(&model->queue_arr, request);
110     if (status == SUCCESS)
111         status = queue_pop_front(&model->queue_lst, request);
112     return status == SUCCESS;
113 }
114
115 // вставляет заявку не далее 4 позиции с головы
116 static int _model_insert_request(model_t *model, request_t request)
117 {
118     uint32_t pos = rand() % 4;
119     int status = queue_insert(&model->queue_arr, pos, request);
120     if (status == SUCCESS)
121         status = queue_insert(&model->queue_lst, pos, request);
122     return status;
123 }
124
125 model_result_t model_run(size_t requests_amount, params_t params)
126 {
127     if (model.curr_time == 0.0)
128     {

```

```

129     model.dispatching_request = (request_t){ .id = 0, .type = 2 };
130     model.oa_time = _random_time(params.t3);
131     model.request_time = _random_time(params.t1);
132 }
133
134 int eps_printed = 0;
135
136 int in_tasks = 0;
137
138 int status;
139 for (size_t reqs = 1; reqs <= requests_amount;)
140 {
141     /* Алгоритм работы моделирования:
142     1. Выбрать ближайшее событие (по времени) из следующих:
143     1.1. Приход заявки 1го типа
144     1.2. Завершение обработки текущей заявки аппаратом
145     2. Реализовать выбранное событие
146     3. Увеличить счётчик времени
147     4. Уменьшить времена next_request_1_time и oa_time
148     5. Рассчитать новое время до следующего события того же типа
149     */
150
151     if (model.request_time < model.oa_time || model.oa_time < 0.0) // если но
        вая заявка пришла раньше, чем обработалась текущая
152     {
153         double time_delta = model.request_time;
154
155         status = _model_send_request(&model, (request_t){ .id = reqs, .type
            = 1 });
156         if (status != SUCCESS)
157             break;
158
159         model.request_time = _random_time(params.t1); // Расчёт нового времен
            и до следующего события того же типа
160         model.oa_time -= time_delta;
161         model.curr_time += time_delta;
162
163         if (in_tasks < requests_amount)
164         {
165             model.results.push_time += model.request_time;
166             in_tasks++;
167         }
168     }
169     else // если текущая заявка обработалась раньше, чем пришла новая
170     {
171         double time_delta = model.oa_time;
172

```



```

173 // Запустить заявку снова в очередь, если она 2го типа
174 if (model.dispatching_request.type == 2)
175 {
176     status = _model_insert_request(&model,
177                                     model.dispatching_request);
178     if (status != SUCCESS)
179         break;
180
181     model.results.request2_cycles++;
182 }
183
184 if (_model_receive_request(&model, &model.dispatching_request))
185 {
186     // Действительно взяли заявку на обслуживание
187     bool type1 = model.dispatching_request.type == 1;
188     model.oa_time = _random_time(type1 ? params.t2 : params.t3);
189     if (type1)
190         model.results.dispatch_time += model.oa_time;
191
192     // Увеличиваем счётчик заявок
193     if ((reqs + 1) % 100 == 0)
194     {
195         model.results.status = status;
196         model.results.total_time = model.curr_time;
197         model.results.requests_processed = requests_amount +
198             model.results.request2_cycles;
199         model.results.queue_size = model.queue_arr.imp.arr.size;
200         print_results(model.results, reqs);
201     }
202
203     if (type1 || reqs == 999)
204         reqs++;
205
206     if (reqs == 1000 && !eps_printed)
207     {
208         eps_printed = 1;
209
210         double in_exp = (params.t1.max + params.t1.min) / 2 *
211             (model.results.requests_processed -
212              model.results.request2_cycles);
213         double in_exp_eps = abs(model.results.push_time - in_exp) /
214             MAX(model.results.push_time, in_exp) * 100;
215
216         double out_exp = (params.t2.max + params.t2.min) / 2 *
217             (model.results.requests_processed -
218              model.results.request2_cycles);

```

```

212         double out_exp_eps = abs(model.results.dispatch_time -
213             out_exp) / MAX(model.results.dispatch_time, out_exp) *
214             100;
215
216         printf("Ожидаемое время прихода -- %.2lf е.в., погрешность
217             -- %.2lf%%\n", in_exp, in_exp_eps);
218         printf("Ожидаемое время обслуживания: %.2lf е.в., погрешность
219             -- %.2lf%%\n", out_exp, out_exp_eps);
220     }
221 }
222 else
223 {
224     // Очередь пуста - обслуживающий аппарат простаивает
225     model.oa_time = -1.0;
226 }
227 }
228
229 model.request_time -= time_delta;
230 model.curr_time += time_delta;
231 }
232
233 model.results.status = status;
234 model.results.total_time = model.curr_time;
235 model.results.requests_processed = requests_amount +
236     model.results.request2_cycles;
237 model.results.queue_size = model.queue_arr.imp.arr.size;
238 return model.results;
239 }

```

### 3.3 Тестовые данные

Ниже, в таблице 3.1, представлены функциональные тесты.

Таблица 3.1 — Функциональные тесты

№	Описание теста	Входные данные	Выходные данные
1	Некорректная команда	a	Сообщение о неверной команде. Возврат в меню
2	Запуск моделирования с настройками по-умолчанию	1 1	Вывод информации о процессе моделирования
3	Изменение параметров по-умолчанию	1 2 1.0 2.0 3 3	Вывод информации о процессе моделирования с заданными параметрами
4	Установка неверного промежутка времени при изменении параметров	1 2 20 18	Вывод сообщения о неверном вводе, возврат в меню
5	Вставка элемента в очередь в ручном режиме	2 1 733	Вывод затраченного на вставку времени
6	Удаление элемента из пустой очереди	2 3	Сообщение об ошибке: удаление из пустой очереди
7	Превышение максимального числа элементов в очередях	1 2 1 2 2 ... 2 11	Сообщение об ошибке: превышение максимального числа элементов в очереди

### 3.4 Вывод

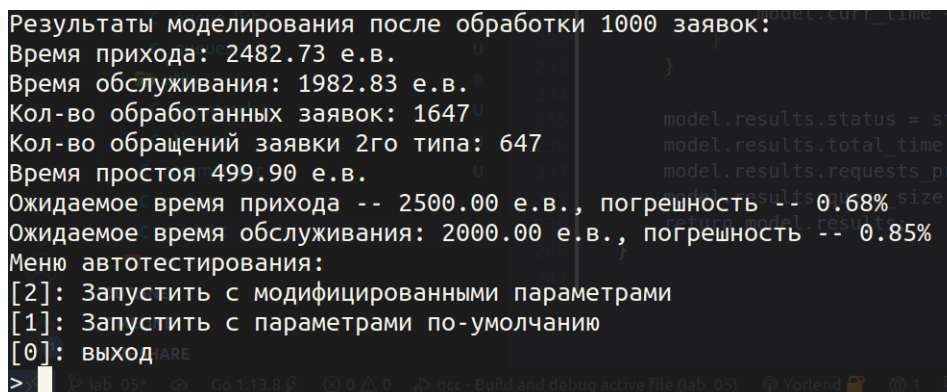
В данном разделе были представлены листинги кодов реализации моделирования работы ОА, представлены требования к ПО и приведены тестовые данные. Функциональные тесты пройдены успешно.

## 4 Экспериментальный раздел

В данном разделе будет проведено сравнение теоритических результатов и программного моделирования работы ОА, а также будет произведено сравнение работы очередей, реализованных на основе линейного односвязного списка и массива.

### 4.1 Результаты моделирования

Ниже, на рисунке 4.1, представлено сравнение теоритических результатов и программного моделирования работы Обслуживающего Аппарата.



```
Результаты моделирования после обработки 1000 заявок:
Время прихода: 2482.73 е.в.
Время обслуживания: 1982.83 е.в.
Кол-во обработанных заявок: 1647
Кол-во обращений заявки 2го типа: 647
Время простоя 499.90 е.в.
Ожидаемое время прихода -- 2500.00 е.в., погрешность -- 0.68%
Ожидаемое время обслуживания: 2000.00 е.в., погрешность -- 0.85%
Меню автотестирования:
[2]: Запустить с модифицированными параметрами
[1]: Запустить с параметрами по-умолчанию
[0]: выход ARE
>
```

Рисунок 4.1 — Результаты работы модели с параметрами по-умолчанию

### 4.2 Постановка эксперимента

В данном эксперименте проводится сравнительный анализ эффективности работы реализаций очереди на основе кольцевого массива и линейного односвязного списка. Анализ проводится по времени работы и по объему затрачиваемой памяти.

#### 4.2.1 Технические характеристики

Ниже представлены технические характеристики ПО, на котором производится тестирование ПО.

- операционная система: Ubuntu 20.04 Linux 64-bit;
- оперативная память: 16 GB;
- AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx @ 8x 2.1GHz;

#### 4.2.2 Описание экспериментальных данных

Тестирование произведено на основе очередей, максимальный размер которых 50 элементов. Для тестирования временной эффективности будет произведено 10 замеров вставки-удаления, после чего результат будет усреднен.

### 4.3 Результат эксперимента

#### 4.3.1 Время работы

В результате подсчета эффективности по времени для очередей на основе массива и списка, были получены следующие данные, отображенные в таблице 4.1.

Таблица 4.1 — Время выполнения

Реализация очереди	Вставка, тики	Удаление, тики
Линейный список	2853	2433
Кольцевой массив	988	242

Можно увидеть, что реализация очереди на основе линейного списка на 65% эффективнее по вставке, и на 90% эффективнее по удалению элемента.

#### 4.3.2 Затрачиваемая память

В результате подсчета затрачиваемой под очереди памяти, были получены следующие результаты (таблица 4.2).

Таблица 4.2 — Затраты памяти

Макс. размер	реал. размер	Объем памяти на кольцевом массиве, байт	Объем памяти списка, байт	относительная разница %
10	0	32	24	79
10	8	112	152	-36
100	15	832	264	68
100	50	832	824	1

Исходя из приведенных выше данных, можно сделать вывод, что реализация очереди на линейном списке выгоднее реализации на кольцевом массиве при проценте фактической заполненности очереди менее 50%. В общем же случае массив занимает меньший объем памяти, чем линейный односвязный список.

### 4.4 Анализ фрагментации

Рассмотрим очереди, с максимальным числом элементов 5. Заполним их значениями от 1 до 5 (рисунок 4.2)

```

QUEUE SIZE: 5
id: 1 -- type: 1
id: 2 -- type: 1
id: 3 -- type: 1
id: 4 -- type: 1
id: 5 -- type: 1
Память, затраченная на хранение очереди списком: 104 байт.
На хранение одного элемента 16 байт.
Память, затраченная на хранение очереди массивом: 72 байт.
На хранение одного элемента 8 байт.
Список адресов:
Адрес: 0x55f712342de0, состояние: используется
Адрес: 0x55f712342e20, состояние: используется
Адрес: 0x55f712342e60, состояние: используется
Адрес: 0x55f712342ea0, состояние: используется
Адрес: 0x55f712342ee0, состояние: используется
Меню ручного тестирования:
[3]: Вывести очереди
[2]: Удалить элемент из очереди
[1]: Вставить элемент в очередь
[0]: выход
>

```

Рисунок 4.2 — Заполненные очереди

Удалим 2 элемента и вставим их снова (рисунок 4.3)

```

QUEUE SIZE: 3
id: 3 -- type: 1
id: 4 -- type: 1
id: 5 -- type: 1
Память, затраченная на хранение очереди списком: 72 байт.
На хранение одного элемента 16 байт.
Память, затраченная на хранение очереди массивом: 72 байт.
На хранение одного элемента 8 байт.
Список адресов:
Адрес: 0x560ac8567de0, состояние: не используется
Адрес: 0x560ac8567e20, состояние: не используется
Адрес: 0x560ac8567e60, состояние: используется
Адрес: 0x560ac8567ea0, состояние: используется
Адрес: 0x560ac8567ee0, состояние: используется
Меню ручного тестирования:
[3]: Вывести очереди
[2]: Удалить элемент из очереди
[1]: Вставить элемент в очередь
[0]: выход
>

```

Рисунок 4.3 — Очереди из тремя элементами и двумя удаленными

Вставим эти элементы заново (рисунок 4.4)

```
QUEUE SIZE: 5
id: 3 -- type: 1
id: 4 -- type: 1
id: 5 -- type: 1
id: 1 -- type: 1
id: 2 -- type: 1
Память, затраченная на хранение очереди списком: 104 байт.
На хранение одного элемента 16 байт.
Память, затраченная на хранение очереди массивом: 72 байт.
На хранение одного элемента 8 байт.
Список адресов:
Адрес: 0x560ac8567de0, состояние: переиспользуется
Адрес: 0x560ac8567e20, состояние: переиспользуется
Адрес: 0x560ac8567e60, состояние: используется
Адрес: 0x560ac8567ea0, состояние: используется
Адрес: 0x560ac8567ee0, состояние: используется
Меню ручного тестирования:
[3]: Вывести очереди
[2]: Удалить элемент из очереди
[1]: Вставить элемент в очередь
[0]: выход
>
```

Рисунок 4.4 — Переиспользование областей памяти

Очистим и переинициализируем очереди (рисунок 4.5)

```
QUEUE SIZE: 2
id: 2 -- type: 1
id: 32 -- type: 1
Память, затраченная на хранение очереди списком: 56 байт.
На хранение одного элемента 16 байт.
Память, затраченная на хранение очереди массивом: 72 байт.
На хранение одного элемента 8 байт.
Список адресов:
Адрес: 0x560ac8567da0, состояние: используется
Адрес: 0x560ac8567de0, состояние: не используется
Адрес: 0x560ac8567e20, состояние: не используется
Адрес: 0x560ac8567e60, состояние: переиспользуется
Адрес: 0x560ac8567ea0, состояние: не используется
Адрес: 0x560ac8567ee0, состояние: не используется
Меню ручного тестирования:
[3]: Вывести очереди
[2]: Удалить элемент из очереди
[1]: Вставить элемент в очередь
[0]: выход
>
```

Рисунок 4.5 — Пустые очищенные очереди

Ниже, на рисунке 4.6, можно заметить, что один из вставленных элементов переиспользовал ранее освобожденную память, а другой занял ранее неиспользуемую память.

```
QUEUE SIZE: 2
id: 2 -- type: 1
id: 32 -- type: 1
Память, затраченная на хранение очереди списком: 56 байт.
На хранение одного элемента 16 байт.
Память, затраченная на хранение очереди массивом: 72 байт.
На хранение одного элемента 8 байт.
Список адресов:
Адрес: 0x560ac8567da0, состояние: используется
Адрес: 0x560ac8567de0, состояние: не используется
Адрес: 0x560ac8567e20, состояние: не используется
Адрес: 0x560ac8567e60, состояние: переиспользуется
Адрес: 0x560ac8567ea0, состояние: не используется
Адрес: 0x560ac8567ee0, состояние: не используется
Меню ручного тестирования:
[3]: Вывести очереди
[2]: Удалить элемент из очереди
[1]: Вставить элемент в очередь
[0]: выход
>
```

Рисунок 4.6 — Фрагментация

В результате чего наблюдаем фрагментацию памяти. Выделение цельного блока заданной длины не может быть произведено.



## 5 Контрольные вопросы

### 1) Что FIFO и LIFO?

Очередь работает по принципу FIFO – первый пришёл – первый вышел. Стек работает по принципу LIFO – последний пришёл – первый вышел.

2) Каким образом и какой объём памяти выделяется под хранение очереди при различной её реализации?

При реализации очереди на массиве память выделяется единожды в момент инициализации.

При реализации стека на связном списке память выделяется каждый раз при добавлении нового элемента в стек.

3) Каким образом освобождается память при удалении элемента из очереди при её различной реализации?

При реализации очереди на массиве память очищается только по окончании работы с очередью.

При реализации стека на связном списке память очищается каждый раз при удалении элемента из очереди.

### 4) Что происходит с элементами очереди при её просмотре?

В классической реализации очереди просмотр осуществляется поэлементным удалением элементов из одного конца, запоминанием их и вставкой в другой конец.

### 5) От чего зависит эффективность физической реализации очереди?

Эффективность физической реализации очереди зависит от способа реализации очереди.

6) Каковы достоинства и недостатки различных реализаций очереди в зависимости выполняемых над ними операций?

Достоинства и недостатки реализации на массиве:

- + быстрая выполнения операций вставки и удаления;
- нужно знать максимальный размер очереди заранее;
- ± эффективна по памяти только при большой заполненности.

Достоинства и недостатки реализации на списке:

- + размер ограничен лишь оперативной памятью;
- операции выполняются медленнее, чем на массиве;
- ± эффективна по памяти при небольшой заполненности.

### 7) Что такое фрагментация памяти, в какой части ОП она возникает?

Фрагментация памяти – явление, при котором занятые участки памяти перемешаны со свободными участками. Оно приводит к ситуациям, когда несмотря на то, что физический объём свободной памяти достаточен для выделения блока заданной длины, выделение не может быть осуществлено из-за отсутствия цельного свободного блока памяти заданного размера. Фрагментация возникает в куче (heap).

### 8) Для чего нужен алгоритм «близнецов»?

Метод близнецов необходим для эффективного выделения памяти.

### 9) Какие дисциплины выделения памяти вы знаете?

Дисциплины: самый подходящий, первый подходящий и метод близнецов.

10) На что необходимо обратить внимание при тестировании программы?

При тестировании программы необходимо учесть как можно больше (а в лучшем случае все) классов эквивалентности для входных данных, убедиться, что 100% написанного кода было выполнено, и результат выполнения совпадает с ожидаемым. Также следует вести учет потребления программой ресурсов, выделяемых операционной системой. Все ресурсы по окончании работы с ними должны быть возвращены системе в полном объеме.

11) Каким образом физически выделяется и освобождается памяти при динамических запросах?

При запросе на выделение памяти менеджер памяти проходит по списку блоков памяти в поисках блока требуемого размера, далее если свободный блок слишком большой, он разбивает его на два, возвращая указатель на блок требуемой длины. Если же блок требуемого размера не был найден (например вследствие фрагментации памяти) то менеджер памяти завершает работу с неудачей. При освобождении памяти менеджер памяти помечает блок как свободный и при необходимости объединяет соседние свободные блоки в один.

## **Заключение**

Решение о выборе той или иной реализации очереди сильно зависит от выполняемых задач. Для представленной в данной лабораторной работе задачи (моделирование работы системы ОА) у каждой реализации есть свои преимущества и свои недостатки.

Для реализации очереди на основе кольцевого массива характерна быстрота выполняемых операций (вставки и удаления). Это решение эффективно по времени, но зачастую проигрывает по памяти, при малой заполненности очереди (менее 50%). Главной особенностью реализации очереди на списке является удобство работы и тот факт, что максимальное количество элементов ограничено лишь объемом оперативной памяти.