



**Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное 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»**

## **ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5 ПО ДИСЦИПЛИНЕ «ТИПЫ И СТРУКТУРЫ ДАННЫХ»**

*Обработка очередей*

**Вариант № 4**

Студент: **Бондарева В. А.**

Группа: **ИУ7-34Б**

Преподаватель: **Никульшина Т. А.**

**2025 г.**

## Условие задачи

Система массового обслуживания состоит из обслуживающего аппарата (ОА) и двух очередей заявок двух типов. Заявки 1-го и 2-го типов поступают в «хвосты» своих очередей по случайному закону с интервалами времени  $T_1$  и  $T_2$ , равномерно распределенными от 1 до 5 и от 0 до 3 единиц времени (е.в.) соответственно. В ОА они поступают из «головы» очереди по одной и обслуживаются также равномерно за времена  $T_3$  и  $T_4$ , распределенные от 0 до 4 е.в. и от 0 до 1 е.в. соответственно, после чего покидают систему. (Все времена – вещественного типа). В начале процесса в системе заявок нет. Заявка любого типа может войти в ОА, если:

- а) она вошла в пустую систему;
- б) перед ней обслуживалась заявка ее же типа;
- в) перед ней из ОА вышла заявка другого типа, оставив за собой пустую очередь (система с чередующимся приоритетом).

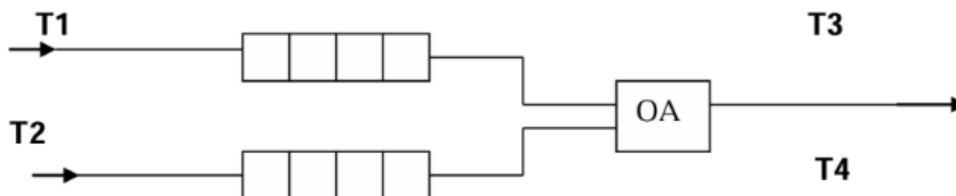


Рис. 1. Схема очередей заявок и обслуживающего аппарата.

Смоделировать процесс обслуживания первых 1000 заявок 1-го типа, выдавая после обслуживания каждых 100 заявок информацию о текущей и средней длине каждой очереди, а в конце процесса - общее время моделирования и количество вошедших в систему и вышедших из нее заявок обоих типов, время простоя ОА. По требованию пользователя выдать на экран адреса элементов очереди при удалении и добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

## Описание T3

### Исходные данные

Исходными данными программы являются:

1. Строка, содержащая единственное целое положительное число, являющееся пунктом меню.

Пользователю предоставляются следующие опции меню:

- |  |
|--|
| 0. Выход                                   |
| 1. Запустить симуляцию массивом            |
| 2. Запустить симуляцию списком             |
| 3. Оценить эффективность                   |
| 4. Посмотреть информацию о памяти (массив) |
| 5. Посмотреть информацию о памяти (список) |
| 6. Таблица сравнения push и pop            |

## 2. Временные интервалы T1, T2, T3 и T4.

### *Результат*

Результатом выполнения программы являются:

1. Вывод меню для выбора опции (главное меню и меню изменения временных интервалов).
2. Таблицы сравнения, такие как таблица сравнения времени выполнения операций push и pop для разных реализаций очереди, таблица свободных областей памяти, таблица эффективности симуляций, основанных на двух различных реализациях очереди.
3. Текущее состояние очереди.
4. Сообщения о статусе выполнения программы и ошибках.

### *Способ обращения к программе*

Пользователь обращается к программе при помощи исполняемого файла app.exe.

### *Возможные аварийные ситуации и ошибки пользователя*

Все возможные аварийные ситуации и ошибки пользователя описаны типом status\_t, включающим в себя коды возврата программы:

1. ERR\_IO: ошибка ввода/вывода данных программы.
2. ERR\_INVALID\_POINTER: ошибка при работе с указателем.
3. ERR\_RANGE: неверно указано количество чего-либо.
4. ERR\_OVERFLOW: произошло переполнение очереди.
5. ERR\_POP\_EMPTY: ошибка удаления элемента из пустой очереди.
6. ERR\_MEM: ошибка при работе с памятью.
7. ERR\_CRITICAL\_RECORDS\_ATTITUDE: критическая разница между поступившими и обработанными заявками в рамках одной очереди.

## Описание внутренних структур данных

```
typedef enum
{
    TYPE_1,
    TYPE_2
} request_class_t;

typedef struct
{
    request_class_t request_class;
```

```

    double arrival_time;
} request_t;

typedef struct
{
    double min_time;
    double max_time;
} time_range_t;

typedef struct
{
    request_t data[MAX_QUEUE_SIZE];
    size_t pin; // индекс для добавления (хвост)
    size_t pout; // индекс для извлечения (голова)
    size_t size; // текущее количество элементов
    size_t total_enqueued; // было добавлено элементов
    size_t total_dequeued; // было извлечено
} arr_queue_t;

typedef struct node
{
    request_t request;
    struct node *next;
} node_t;

typedef struct
{
    node_t *pin;
    node_t *pout;
    size_t curr_size;
} list_queue_t;

typedef struct
{
    size_t request_in_count; // кол-во заявок, которые вошли в систему
    size_t request_out_count; // кол-во заявок, которые вышли из системы
    size_t failed_request_count; // кол-во потерянных заявок
    size_t function_call_count; // кол-во обращений к ОА
    double total_length; // суммарная длина очереди на момент извлечения
    // каждой заявки
    double total_wait_time; // суммарное время ожидания всех заявок в
    // очереди
} simulation_log_t;

```

Листинг 2. Внутренние структуры данных.

Структура `request_class_t` описывает класс заявки (заявка может быть первого или второго типа), а структура `request_t` описывает саму заявку: она включает в себя тип заявки и время ее прибытия.

Структура `time_range_t` задает интервал времени, включая в себя минимальное время и максимальное время промежутка.

Основными структурами являются `arr_queue_t` и `list_queue_t`. Описывающая массивную очередь `arr_queue_t` хранит следующие поля:

1. `data[]` – массив заявок.
2. `pin` – индекс для добавления нового элемента.

3. `pout` – индекс для удаления элемента.
4. `size` – текущий размер массива.
5. `total_enqueued` – количество добавленных в очередь элементов.
6. `total_dequeued` – количество извлеченных из очереди элементов.

Другая структура `list_queue_t` содержит поле `curr_size`, хранящее текущий размер очереди-списка, а также указатели `node_t` (`pin` и `pout`) на узлы списка («голова» и «хвост» соответственно). В свою очередь каждый узел списка `node_t` хранит заявку (значение в поле `request`), а также указатель на следующий узел списка `next`.

Также с помощью структуры `simulation_log_t` в симуляции хранится информация об очередях, а именно количество заявок, которые вошли в систему, количество заявок, которые вышли из системы, количество потерянных заявок (в случае с переполнением очереди), количество обращений к обслуживающему аппарату, суммарная длина очереди на момент извлечения каждой заявки, суммарное время ожидания всех заявок в очереди. Данная информация необходима при сборе статистики о симуляции и формирования данных для таблицы сравнения.

В листинге №3 представлены сигнатуры основных функций.

```
// основные функции симуляции
status_t simulate_service_unit_by_arr(bool verbose_mode);
status_t simulate_service_unit_by_list(bool verbose_mode, size_t
*max_total_len_two_queues);
status_t change_simulation_configurations(void);

// основные функции для работы с массивом
status_t push_arr(arr_queue_t *arr_queue, request_t *request);
status_t pop_arr(arr_queue_t *arr_queue, request_t *popped_request);
status_t print_arr(arr_queue_t *arr_queue);

// основные функции для работы со списком
status_t push_list(list_queue_t *list_queue, request_t *request);
status_t pop_list(list_queue_t *list_queue, request_t *popped_request);
status_t allocate_list_node(node_t **new_node, const request_t *request);
status_t destroy_list_queue(list_queue_t *list_queue);

// основные функции необходимых измерений
status_t measure_efficiency(void);
status_t compare_push_and_pop(void);
```

Листинг 3. Сигнатуры основных функций.

## Алгоритм

Помимо основного алгоритма программы (`main.c`) основополагающими алгоритмами для реализации заданного функционала являются:

- алгоритмы ввода данных
- алгоритм добавления элемента в очередь

- алгоритм удаления элемента из очереди
- алгоритм извлечения элемента из очереди
- алгоритм симуляции работы обслуживающего аппарата

Рассмотрим каждый алгоритм.

1. Инициализируется код возврата.
2. Проверяется корректность входных данных.
3. Предлагается ввести нижнюю границу временного интервала.
4. Предлагается ввести верхнюю границу временного интервала.
5. Очищается буфер ввода.
6. Проверяется корректность введенных значений, после чего обновляются соответствующие значение по указателям.
7. Возвращается статус выполнения функции.

Листинг 4. Алгоритм ввода временного интервала.

1. Инициализируется код возврата.
2. Проверяется корректность входных данных. В случае с массивной реализацией проверяется также переполнение очереди.
3. Если входные данные корректны, в случае с реализацией списком выделяется память под новый узел.
4. Заявка добавляется в очередь, размер очереди обновляется, обновляются указатели `pin` (и `roul`, в случае если очередь пуста).
5. Возвращается статус выполнения функции.

Листинг 5. Алгоритм добавления элемента в очередь.

1. Инициализируется код возврата.
2. Проверяется корректность входных данных.
3. Если входные данные корректны, из очереди извлекается значение, соответствующее `roul`.
4. Указатель `roul` обновляется, «переводится» на «предыдущий» элемент.
5. В случае с реализацией списком извлеченный узел освобождается.
6. Длина очереди уменьшается.
7. Возвращается статус выполнения функции.

Листинг 6. Алгоритм удаления элемента из очереди.

1. Инициализируются код возврата, две очереди для заявок первого и второго типа, а также другие различные переменные, необходимые для работы симуляции, такие как:
  - a. `current_time = 0.0` (текущее модельное время)
  - b. `current_service_end_time = INFINITY` (время окончания обслуживания)
  - c. `next_arrival_type1, next_arrival_type2` (время следующего поступления заявок)
2. Начинается цикл симуляции: пока не обработано требуемое количество заявок типа 1 и нет ошибок.
3. Определяется следующее событие путем сравнения времени поступления заявок, а также времени окончания обслуживания текущей заявки.
4. Если следующее событие – поступление заявки типа 1, создается новая заявка с временем прибытия `current_time`, заявка добавляется в первую очередь (в случае с массивной реализацией проверяется переполнение очереди). Если на момент добавление такой

заявки обслуживающий аппарат свободен, то устанавливается новый срок окончания обслуживания уже для этой заявки. В случае успешного выполнения счетчик поступивших заявок первого типа увеличивается, next_arrival_type1 обновляется.
5. Если следующее событие – поступление заявки типа 2, все действия выполняются аналогично п. 4.
6. Если следующее событие – окончание обслуживания, выбирается следующая заявка для обработки (чередующийся приоритет). Если обе очереди пусты, учитывается время простоя системы.
7. Каждые 100 обработанных заявок выводится промежуточная статистика и проверяются критические соотношения длин очередей.
8. Когда цикл завершается, выводится итоговая статистика и освобождается память (динамическая).

Листинг 7. Алгоритм симуляции обслуживающего аппарата.

## Тесты

Так как в данной лабораторной работе изменяемыми параметрами симуляции являются только временные интервалы поступления и обработки заявок, то понятие «ПОЗИТИВНЫХ» и «НЕГАТИВНЫХ» тестов смещается в сторону успешности выполнения самой симуляции.

Выполняемая команда меню	Ожидаемый результат
Выход	Успешный выход из программы
Запустить симуляцию массивом	Успешная симуляция на основе массива
Запустить симуляцию списком	Успешная симуляция на основе списка
Оценить эффективность	Успешный расчёт и вывод таблицы эффективности
Посмотреть информацию о памяти (массив)	Успешный вывод таблицы фрагментации памяти для массива (фрагментация отсутствует)
Посмотреть информацию о памяти (список)	Успешный вывод таблицы фрагментации памяти для списка (возможна фрагментация)
Таблица сравнения push и pop	Успешный вывод сравнительной таблицы push и pop
Настроить параметры симуляции	Окно настройки параметров симуляции

Таблица 1. Ожидаемый результат выполнения команд меню.

Ситуацию, при которой программа завершится аварийно можно создать, определенным образом настроив параметры симуляции.

В случае с массивной реализацией такой ситуацией является переполнение очереди (рисунки 2 и 3).

C U R R E N T   C O N F I G S		
Интервал поступления T1:	{ 1.00 , 5.00 }	ед.в.
Интервал поступления T2:	{ 0.00 , 3.00 }	ед.в.
Интервал обслуживания T1:	{ 10.00 , 15.00 }	ед.в.
Интервал обслуживания T2:	{ 0.00 , 1.00 }	ед.в.

Рис. 2. Временные интервалы, при которых возникает переполнение очереди-массива.

INTERIM RESULTS						
requests (type1) processed	queue name	requests in	requests out	requests lost	current length	average length
100	queue_1	423	100	0	323	162.34
100	queue_2	837	2	0	835	0.00

Произошло переполнение чего-нибудь

Рис. 3. Ошибка переполнения очереди-массива.

В случае реализации на основе списка аварийная ситуация может возникнуть, если первая очередь, обработка элементов которой является обязательным условием для выхода из симуляции, не обрабатывается обслуживающим аппаратом или обрабатывается несравнимо медленнее, чем вторая очередь (рисунки 4 и 5).

C U R R E N T   C O N F I G S		
Интервал поступления T1:	{ 1.00 , 2.00 }	ед.в.
Интервал поступления T2:	{ 0.00 , 1.00 }	ед.в.
Интервал обслуживания T1:	{ 1000.00 , 4000.00 }	ед.в.
Интервал обслуживания T2:	{ 0.00 , 1.00 }	ед.в.

Рис. 4. Временные интервалы, при которых возникает ошибка симуляции очереди-списка.

INTERIM RESULTS						
requests (type1) processed	queue name	requests in	requests out	requests lost	current length	average length
Критическое отношение обработанных/необработанных заявок. Симуляция остановлена						

Рис. 5. Ошибка симуляции очереди-списка.

## Оценка эффективности

INTERIM RESULTS						
requests (type1) processed	queue name	requests in	requests out	requests lost	current length	average length
100	queue_1	101	100	0	1	0.72
100	queue_2	186	182	0	4	3.56
200	queue_1	201	200	0	1	1.56
200	queue_2	391	351	0	40	6.53
300	queue_1	303	300	0	3	3.13
300	queue_2	595	553	0	42	11.61
400	queue_1	401	400	0	1	3.44
400	queue_2	808	776	0	32	13.39
500	queue_1	503	500	0	3	3.66
500	queue_2	1011	972	0	39	15.58
600	queue_1	600	600	0	0	3.42
600	queue_2	1204	1175	0	29	15.06
700	queue_1	701	700	0	1	3.43
700	queue_2	1420	1420	0	0	14.65
800	queue_1	806	800	0	6	3.47
800	queue_2	1633	1604	0	29	14.96
900	queue_1	910	900	0	10	3.99
900	queue_2	1838	1823	0	15	17.45
1000	queue_1	1014	1000	0	14	4.40
1000	queue_2	2041	2032	0	9	19.19

Рис. 6. Данные симуляции на основе массивной очереди.

На рисунке 6 представлены данные симуляции работы ОА в массивной реализации, на их основе можно проверить правильность работы системы по входу и по выходу, рассчитав соответствующие погрешности.

CURRENT CONFIGS			
Интервал поступления T1:	{ 1.00 , 5.00 }	ед.в.	
Интервал поступления T2:	{ 0.00 , 3.00 }	ед.в.	
Интервал обслуживания T1:	{ 0.00 , 4.00 }	ед.в.	
Интервал обслуживания T2:	{ 0.00 , 1.00 }	ед.в.	

Рис. 7. Временные интервалы поступления и обработки заявок.

Для расчета погрешности по входу необходимо определить предполагаемое количество вошедших заявок, для чего нужно разделить общее время моделирования на среднее время прихода одной заявки. Общее время симуляции для обеих реализаций представлено в таблице 2, общее время для случая массивной очереди составляет 3047.62 е.в.

$$\frac{3047.62}{3} = 1015.87 \text{ е. в.}$$

Теоретический расчет показывает, что пришло 1015 заявок, в то время как на самом деле заявок первого типа поступило 1014, следовательно, погрешность по входу составляет:

$$\left| 100\% * \frac{1014 - 1015.87}{1015.87} \right| = 0.18\%$$

Чтобы рассчитать погрешность по выходу необходимо разницу фактического времени моделирования и теоретического времени моделирования разделить на теоретическое времени моделирования. Теоретическое время моделирования для массивной реализации равняется сумме теоретического времени работы ОА и времени его простоя:

$$3032 + 9.95 = 3041.95 \text{ (е. в. )}$$

Так как фактическое время работы ОА составило 3047.62, погрешность по выходу будет равняться:

$$\left| 100\% * \frac{3047.62 - 3041.95}{3041.95} \right| = 0.19\%$$

INTERIM RESULTS						
requests (type1) processed	queue name	requests in	requests out	requests lost	current length	average length
100	queue_1	105	100	0	5	4.06
100	queue_2	216	212	0	4	17.79
200	queue_1	204	200	0	4	3.25
200	queue_2	382	364	0	18	13.67
300	queue_1	303	300	0	3	3.27
300	queue_2	576	576	0	0	13.39
400	queue_1	401	400	0	1	3.90
400	queue_2	773	710	0	63	15.27
500	queue_1	504	500	0	4	4.33
500	queue_2	995	995	0	0	17.73
600	queue_1	600	600	0	0	3.71
600	queue_2	1186	1172	0	14	15.54
700	queue_1	702	700	0	2	3.49
700	queue_2	1408	1408	0	0	14.53
800	queue_1	802	800	0	2	3.17
800	queue_2	1592	1589	0	3	13.41
900	queue_1	904	900	0	4	3.27
900	queue_2	1777	1716	0	61	13.80
1000	queue_1	1013	1000	0	13	3.94
1000	queue_2	1988	1948	0	40	16.95

Рис. 8. Данные симуляции на основе очереди-списка.

SIMULATION SUMMARY			SIMULATION SUMMARY		
requests type	requests in	requests out	requests type	requests in	requests out
queue type 1	1014	1000	queue type 1	1013	1000
queue type 2	2041	2032	queue type 2	1988	1948
total simulation time		3047.62	total simulation time		2984.31
system downtime		9.95	system downtime		14.18

Таблица 2. Результаты симуляции на основе двух различных реализации очереди.

Аналогично алгоритму расчета погрешности для заявок первого типа и симуляции на основе массивной очереди можно рассчитать погрешность для заявок

второго типа, а также для заявок первого и второго типов в случае реализации очереди с помощью очереди-списка. Значения погрешностей для каждого такого случая представлены в таблицах 3 и 4:

	Очередь 1-ого типа	Очередь 2-ого типа
Погрешность на вход	0.18%	0.45%
Погрешность на выход	0.19%	0.19%

Таблица 3. Погрешности на вход и на выход для двух типов очереди на основе массива.

	Очередь 1-ого типа	Очередь 2-ого типа
Погрешность на вход	1.83%	0.08%
Погрешность на выход	0.75%	0.75%

Таблица 4. Погрешности на вход и на выход для двух типов очереди на основе списка.

Погрешности измерений, проведенных в процессе выполнения лабораторной работы, не превышают критическое значение 2-3%.

indx	added addresses	removed addresses
1	0x5214bf0	0x5214bf0
2	0x5214bd0	0x5214bd0
3	0x5214bb0	0x5214bb0
4	0x5214b90	0x5214b90
5	0x5214b70	0x5214b70
6	0x5214b50	0x5214b50
7	0x5214b30	0x5214b30
8	0x521c810	0x521c810
9	0x521c730	0x521c730
10	0x521c750	0x521c750

Рис. 9. Фрагментация памяти на основе удаления и добавления выборки из 10 элементов в очередь-список.

На основе данных рисунка 9 можно сделать вывод, что при удалении и добавлении элементов в очередь-список фрагментации не возникает, то есть добавленные элементы списка сохраняются в области памяти, которая была ранее освобождена при удалении другого элемента списка.

	push operation	pop operation
array	58.11	55.48
list	67.21	48.19

Рис. 10. Средняя скорость выполнения 1000 операций push и pop в очереди-массиве и очереди-списке (наносекунды).

На рисунке 10 продемонстрированы временные преимущества выполнения операций удаления и добавления элементов у очереди-массива относительно очереди-списка: очередь-массив выполняет операции «push» и «pop» значительно быстрее, чем очередь-список. Это обусловлено отсутствием необходимости выделения памяти и добавления нового узла в массивной реализации. Таким образом, массивная очередь более эффективна для реализаций, в которых часто приходится производить операции добавления и удаления элементов очереди.

EFFICIENCY TABLE		
queue type	time (ns)	memory (bytes)
array	176696.45	16040
list	216577.63	4440

Рис. 11. Таблица сравнения эффективности двух реализаций очереди по памяти (байты) и по скорости (наносекунды).

Исходя из данных, представленных на рисунке 11, можно произвести сравнение эффективности работы симуляции на основе двух различных реализаций очереди. Вычисление среднего времени выполнения симуляции на выборке из 1000 запусков симуляции для массива и для списка показало относительно незначительное преимущество массивной реализации. В таблице 5 представлены временные результаты работы симуляции для нескольких запусков, на их основе можно сделать

вывод, что в среднем массивная симуляция работает на 13.66% процентов быстрее, чем списочная.

Массив	Список
182968.03	214829.69
181090.40	203964.14
173563.67	201684.77
173246.61	200853.87
172761.57	202202.67

Таблица 5. Среднее время работы симуляции на основе 1000 итераций (наносекунды).

Сравнение двух реализаций симуляции работы ОА показало, что очередь-список занимает в среднем (данные для нескольких запусков расчета эффективности представлены в таблице 6) в 3.26 раз меньше, чем массивная очередь. Это обусловлено средней длиной фактической очереди в момент работы симуляции: чем больше заполненность массива заявок относительно его максимальной длины, тем менее эффективен становится список в контексте затрат на память. Чтобы найти процент заполнения массива, при котором очередь-список будет менее эффективен в контексте расхода памяти, необходимо сравнить зависимость объема занимаемой памяти от количества элементов  $n$  для двух структур данных.

Массив	Список
16040	4440
16040	4296
16040	5028
16040	5448
16040	5604

Таблица 6. Максимально большой занимаемый объем памяти в процессе симуляции для двух реализаций симуляции (байты).

Память очереди-массива фиксирована и не зависит от  $n$ :

$$M_{arr}(n) = \text{sizeof}(\text{request\_t}) * \text{MAX\_QUEUE\_SIZE} + 5 * 8$$

$$M_{arr}(n) = 16 * \text{MAX\_QUEUE\_SIZE} + 40$$

Память, занимаемая списком, линейно зависит от  $n$ :

$$M_{list}(n) = sizeof(node\_t) * n + sizeof(list\_queue\_t)$$

$$M_{list} = 24 * n + 24$$

Приравняем полученный объем, чтобы найти «границу равенства» по памяти от  $n$  элементов:

$$M_{arr}(n) = M_{list}(n)$$

$$24 * n + 24 = 16 * MAX\_QUEUE\_SIZE + 40$$

$$24 * n = 16 * MAX\_QUEUE\_SIZE + 16$$

$$n = \frac{16 * MAX\_QUEUE\_SIZE + 16}{24}$$

$$n = \frac{2}{3} * MAX\_QUEUE\_SIZE + \frac{2}{3}$$

Чтобы вычислить итоговое процентное соотношение, необходимо найти отношение  $n$  к  $MAX\_QUEUE\_SIZE$ :

$$fullness = \frac{n}{MAX\_QUEUE\_SIZE} * 100\%$$

$$fullness = \frac{\frac{2}{3} * MAX\_QUEUE\_SIZE + \frac{2}{3}}{MAX\_QUEUE\_SIZE} * 100\%$$

$$fullness = \left( \frac{2}{3} + \frac{2}{3 * MAX\_QUEUE\_SIZE} \right) * 100\%$$

Так как при больших значениях  $MAX\_QUEUE\_SIZE$  дробь справа незначительно мала, то

$$fullness \approx 66. (6)\%$$

В данной реализации массив будет менее эффективен чем список в контексте требуемого объема памяти, пока будет заполнен менее чем на 67%.

### *Выводы*

В процессе выполнения лабораторной работы было установлено, что каждая из рассмотренных реализаций очереди обладает характерными преимуществами и ограничениями: массивная очередь демонстрирует значительное преимущество в скорости операций добавления и удаления элементов (в среднем на 13–18% быстрее), что объясняется отсутствием затрат на динамическое выделение памяти. Очереди на основе массива предпочтительно использовать в реализациях, требующих частого добавления или удаления элементов из очереди. Однако размер массива строго

ограничен, что может привести к неэффективному использованию ресурсов, особенно при малой заполненности. С другой стороны, очередь-список оказывается более экономичной по памяти в условиях большого процента заполнения, занимая в 3–4 раза меньше места при малой и средней длине очереди, но проигрывает в производительности из-за дополнительных расходов на работу с выделением и освобождением памяти.

Таким образом, если ключевым требованием является скорость работы программы, следует использовать массивную очередь, особенно при высокой и стабильной нагрузке. Если же необходима гибкость размера очереди, то более подходящей будет реализация на списке. Полученные погрешности моделирования не превысили 3%, что свидетельствует о корректности реализованных алгоритмов.

### **Контрольные вопросы**

#### *1. Что такое FIFO и LIFO?*

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, то есть First In – First Out (FIFO).

Стек – это последовательный список с переменной длиной, в котором включение и исключение элементов происходит только с одной стороны – с его вершины. Стек функционирует по принципу: последним пришел – первым ушел, Last In – First Out (LIFO).

#### *2. Каким образом и какой объем памяти выделяется под хранение очереди при различной ее реализации?*

При реализации очереди на основе массива очередь занимает фиксированный объем памяти, равный сумме размеров массива и дополнительных информативных полей. При списковой реализации память выделяется динамически под каждый новый узел (элемент списка), что позволяет избежать ограничения размера очереди, однако требует дополнительной памяти для хранения указателей и приводит к фрагментации.

#### *3. Каким образом освобождается память при удалении элемента из очереди при ее различной реализации?*

В массиве удаление элемента происходит с помощью сдвига указателей (pin и rout), при данной реализации память при удалении элемента освобождать не надо. В списке память явно освобождается вызовом free: память возвращается в кучу, но это требует больше времени и может приводить к фрагментации.

*4. Что происходит с элементами очереди при ее просмотре?*

При просмотре очереди все ее элементы удаляются.

*5. От чего зависит эффективность физической реализации очереди?*

Эффективность реализации зависит от частоты операций добавления и удаления, размера элементов, требований к памяти и производительности - массив эффективен при частом доступе и известном размере, список эффективен при динамическом изменении размера и вставках/удалениях в произвольных позициях.

*6. Каковы достоинства и недостатки различных реализаций очереди в зависимости от выполняемых над ней операций?*

Массив обеспечивает быстрый доступ, но ограничен по размеру и неэффективен при частых вставках/удалениях. Список не имеет ограничений по размеру и эффективен при динамических изменениях, но требует больше памяти для указателей и больше времени для выполнения операций удаления и добавления элементов ввиду того, что операции выделения и освобождения памяти выполняются медленно.

*7. Что такое фрагментация памяти, и в какой части ОП она возникает?*

При моделировании очереди освобожденная при удалении элементов память может не использоваться для новых элементов. Это приводит к чередованию занятых и свободных участков памяти, то есть вызывает фрагментацию. Фрагментация возникает в «куче».

*8. Для чего нужен алгоритм «близнецов»?*

Алгоритм близнецов - это метод управления динамической памятью, который организует её в блоки размером степени двойки. При запросе памяти система находит подходящий блок, при необходимости разделяя большие блоки пополам на «близнецов». При освобождении, если оба близнеца свободны, они сливаются в один блок большего размера. Алгоритм близнецов обеспечивает высокую скорость работы и уменьшает фрагментацию, но может приводить к потерям памяти из-за округления размеров до степеней двойки.

*9. Какие дисциплины выделения памяти вы знаете?*

Существуют следующие дисциплины выделения памяти:

- First fit (Первый Подходящий): выделяет первый попавшийся блок памяти, который подходит по размеру;
- Best fit (Лучший Подходящий): выделяет блок памяти, минимизирует фрагментацию, однако работает медленнее, чем First fit;

- Worst fit (Худший Подходящий): выделяет самый большой из доступных блоков памяти, эффективен для больших запросов;
- Next fit (Следующий Подходящий): аналогичен First fit, но начинается поиск с того места, где закончил предыдущий запрос;
- Quick fit (Быстрый подходящий): использует несколько списков фиксированных размеров для быстрого доступа к блокам нужного размера;
- Алгоритм «близнецов» (см. пункт 8);

*10. На что необходимо обратить внимание при тестировании программы?*

При тестировании программы необходимо обеспечить расхождение не более 2-3% между теоретически ожидаемыми и практически полученными временными характеристиками. Требуется проверить корректность работы при разных сценариях заполнения очередей: когда время моделирования лимитируется продолжительностью обслуживания заявок и когда оно определяется интенсивностью их поступления. Важно контролировать обработку переполнения в случае ограниченной емкости очереди. При использовании списковой реализации необходимо строго следить за освобождением памяти при удалении элементов и анализировать возможное возникновение фрагментации памяти в процессе работы программы.

*11. Каким образом физически выделяется и освобождается память при динамических запросах?*

При динамических запросах память нужного размера выделяется через malloc/calloc из кучи с помощью менеджера памяти. Менеджер памяти соответствующим образом помечает области памяти: при освобождении конкретной памяти она помечается как «свободная», а при выделении как «занятая». При необходимости выделения новой памяти менеджер будет искать ее именно среди свободных областей. Выделенная память может быть неинициализированной (в случае работы функции malloc) или проинициализированной нулями (calloc). Освобождение free возвращает блок в кучу, операционная система помечает соответствующий блок памяти как свободный.