**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»

# ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5 ПО ДИСЦИПЛИНЕ:
# ТИПЫ И СТРУКТУРЫ ДАННЫХ

# ОБРАБОТКА ОЧЕРЕДЕЙ

Студент Жаринов М. А.

Вариант 4

Группа **ИУ7-32Б**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

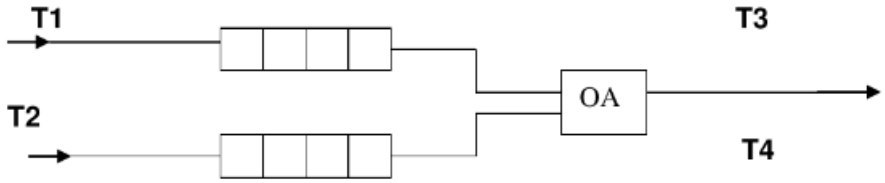
Студент _____ **Жаринов М. А.**

Преподаватель _____ **Силантьева А. В.**

**2024**

## Описание условия 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 Т1 и Т2, 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 равновероятно за времена Т3 и Т4, 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).

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. По требованию пользователя выдать на экран адреса элементов очереди при удалении и добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

Оценить эффективность программы (при различной реализации) по времени и по используемому объему памяти. Сравнить реализации алгоритмов включения и исключения элементов из очереди при использовании двух указанных структур данных

## Техническое задание

**Исходные данные:**

**Времена поступления и обработки заявок** 8 положительных вещественных чисел, записанных в форме IEEE 754: нижний и верхний предел времен T1-T4

**Выходные данные:**

Общее время симуляции, теоретически рассчитанное время, среднее количество заявок в очередях, количество поступивших и обработанных заявок, результаты сравнения способов реализации.

Данные о фрагментации памяти: массив освобожденных адресов памяти

**Реализуемые задачи:**

Пункт меню 0 – Выход из программы

Пункт меню 1 – Инициализация очереди на массиве

Пункт меню 2 – Инициализация очереди на списке

Пункт меню 3 – Запись элемента в очередь

Пункт меню 4 – Удалить элемент из очереди

Пункт меню 5 – Печать очереди

Пункт меню 6 – Печать списка освобожденных адресов (только для очереди на списке)

Пункт меню 7 – Изменение времен поступления и обработки

Пункт меню 8 – Запуск симуляции с очередью на массиве

Пункт меню 9 – Запуск симуляции с очередью на списке

Пункт меню 10 – Проведение сравнительного анализа различных методов реализации очереди

**Способ обращения к программе:**

Строка запуска программы ./app.exe

После запуска с помощью меню нужно инициализировать очередь, запустить симуляцию на одной из реализаций или запустить анализ методов реализации.

**Аварийные ситуации:**

1. EOF в вводе. Код ошибки 1
2. Недостаток свободного места в очереди при симуляции. Код ошибки 2

Все остальные ошибочные ситуации (некорректные элементы к записи, не инициализированная очередь, некорректный пункт меню) не являются аварийными и вызывают повторное приглашение к вводу или возвращение в меню.

## Внутренние структуры данных

Элементы очереди хранится в структуре `value_t`.

```
typedef union
{
    double time;
    struct list_node_t *ptr;
} value_t;
```

Поле `value.time-` вещественное число – время обработки заявки

Поле `value.ptr-` указатель на освобожденный элемент(используется в списке освобожденных адресов)

Вершины списка хранятся в структуре `list_node_t`.

```
typedef struct list_node_t
{
    value_t value;
    struct list_node_t *next_node;
} list_node_t;
```

Поле `value` –значение элемента

Поле `next_node` –указатель на следующий элемент

Очередь в виде массива хранятся в структуре `array_queue_t`.

```
typedef struct
{
    value_t values[QUEUE_DEPTH];
    int in;
    int out;
    int depth;
} array_queue_t;
```

Поле values –массив элементов

Поле depth –количество элементов в очереди

Поле in – индекс следующего вставляемого элемента

Поле out – индекс следующего удаляемого элемента

Макроконстанта QUEUE_DEPTH равна 2000

Очередь в виде списка хранятся в структуре list_queue_t.

```
typedef struct
{
    list_node_t *head;
    list_node_t *tail;
    int depth;
} list_queue_t;
```

Поле head –указатель на вершину списка

Поле tail –указатель на конец списка

Поле depth –глубина очереди

Для удобства работы используется абстрактный тип для использования очереди независимо от его реализации debug_queue_t.

```
typedef struct
{
    void *queue;
    bool (*push)(void *queue, value_t value);
    bool (*pop_mem)(void *queue, value_t *value, list_queue_t
*freed_memory);
    bool (*print)(void *queue);
    list_queue_t *freed_memory;
    void (*print_freed)(list_queue_t *freed_memory);
    void (*free)(void *queue, list_queue_t *freed_memory);
} debug_queue_t;
```

Поле queue – указатель на очередь

Поле push – указатель на функцию записи элемента в очередь

Поле pop_mem – указатель на функцию удаления элемента из очереди с сохранением освобожденного адреса

Поле print – указатель на функцию печати очереди

Поле `freed_memory` – указатель на список-очередь освобожденных адресов

Поле `print_freed` – указатель на функцию печати списка освобожденных адресов

Поле `free` – указатель на функцию освобождения очереди

Также для корректного замера скорости работа создана упрощенная структура очереди без лишних функций и сохранения адресов освобожденной памяти `abstract_queue_t`.

```
typedef struct
{
    void *queue;
    bool (*push)(void *queue, value_t value);
    bool (*pop)(void *queue, value_t *value);
    void (*free)(void *queue);
    int (*get_len)(void *queue);
} abstract_queue_t;
```

Поле `queue` – указатель на очередь

Поле `push` – указатель на функцию записи элемента в очередь

Поле `pop` – указатель на функцию удаления элемента из очереди

Поле `free` – указатель на функцию освобождения очереди

Поле `get_len` – указатель на функцию получения длины очереди

## Описание алгоритма

Программа запрашивает номер элемента в меню, считывает его и выполняет соответствующее действие:

Пункт меню 0 – Выход из программы

Пункт меню 3 – Запись элемента в очередь

1. Программа запрашивает и считывает положительное вещественное число
2. Программа записывает элемент в очередь

Пункт меню 4 – Удаление элемента из очереди

Пункт меню 5 – Печать очереди

Пункт меню 6 – Печать списка освобожденных адресов (только для очереди на списке)

Пункт меню 7 – Изменение времен поступления и обработки

1. Программа запрашивает и считывает 8 положительных вещественных чисел
2. Программа проверяет, что верхняя граница больше нижней и сохраняет значения времен
3. Программа проверяет и печатает предупреждение, в случае если вероятна бесконечная работа симуляции

Пункт меню 8 – Запуск симуляции с очередью на массиве

1. Программа инициализирует 2 очереди на массиве
2. Программа запускает симуляцию с использованием этих очередей

Пункт меню 9 – Запуск симуляции с очередью на списке

1. Программа инициализирует 2 очереди на списке
2. Программа запускает симуляцию с использованием этих очередей

Пункт меню 10 – Проведение сравнительного анализа различных методов реализации очереди

1. Программа запускает симуляции с реализациями очереди на списке и на массиве 500 раз
2. Программа находит среднее время моделирования с двумя методами реализации очереди, выигрыш при использовании очереди на массиве,

используемый размер очереди на списке, выигрыш памяти при использовании очереди на списке и печатает результаты.

3. Программа полностью заполняет очередь на списке и на массиве, а затем опустошает, замеряя среднее время добавления и удаления, и печатает эти времена и выигрыш времени при работе над очередью на массиве

Алгоритм моделирования обслуживающего аппарата

1. Запускается цикл с условием выхода по обработке 1000 заявок 1 типа
2. Находится время до ближайшего из 3 событий – прибытие в 1 очередь заявки, прибытие во 2 очередь заявки, выход из ОА заявки (при ее наличии в нем).
3. Из переменных, хранящих время до ключевых событий, вычитается найденное значение
4. Исходя из состояния ОА, найденное значение прибавляется к времени работы или простоя
5. Если время до добавления заявки в соотв. очередь равно 0, то в нее добавляется новая заявка
6. Если время до конца обработки 0, то увеличивается количество обработанных заявок на 1, и флаг наличия заявки в аппарате устанавливается в 0
7. Если в автомате нет заявки (в том числе и после выхода заявки на данной итерации цикла), то исходя из типа прошлой заявки в ОА и состояния очередей на момент ее попадания в ОА, вычисляется из какой очереди на данный момента должна попасть в ОА заявка, и данная заявка удаляется из очереди, ее время обработки записывается в время до окончания работы ОА

## Алгоритм расчета теоретического времени

Если время обработки заявок первого типа больше чем интервал между их поступлением, то время считается как время обработки 1000 заявок первого типа

Иначе время считается как максимальное между временем прихода 1000 заявок 1 типа и временем обработки 1000 заявок первого типа и того кол-ва заявок 2 типа, которые успеют прийти за это время

## Пример теоретического расчета

Пусть даны времена из условия.

Сначала найдем среднее ожидаемое время поступления и обработки

T1 = (1+5)/2 = 3

T2 = (0+3)/2 = 1.5

T3 = (0+4)/2 = 2

T4 = (0+1)/2 = 0.5

T3 меньше чем T1, поэтому применяем общий алгоритм (обрабатываются обе очереди)

Время поступления 1000 заявок первого типа = T1 * 1000 = 3000

Время обработки 1000 заявок первого типа  = 2 * 1000 = 2000

За время поступления 1000 заявок первого типа придет (3000/T2) = 2000 заявок второго типа, которые будут обрабатываться 2000*T4 = 1000 е.в.

Суммарное время обработки = 2000+1000 = 3000

Максимальное из 3000 и 3000 это 3000

Значит, теоретическое время обработки составляет 3000 единиц времени

## Основные функции

Функция для обработки выбранного пункта меню. Принимает номер выбранного пункта меню, времена добавления и обработки заявок и указатель на очередь. Возвращает код ошибки

```
int process_menu(menu_item_t menu_item, params_t *params, debug_queue_t *debug_queue);
```

Функция записи элемента в очередь-массив. Принимает указатель на очередь, элемент для записи. Возвращает флаг успешности выполнения

```
bool push_array(void *array_queue, value_t value);
```

Функция удаления элемента из очереди-массива. Принимает указатель на очередь, указатель на элемент для чтения. Возвращает флаг успешности выполнения

```
bool pop_array(void *array_queue, value_t *value);
```

Функция печати очереди-массива. Принимает указатель на очередь

```
bool print_array(void *array_queue);
```

Функция освобождения очереди-массива. Принимает указатель на очередь

```
void free_array(void *array_queue);
```

Функция инициализации абстрактной очереди так, чтобы она был очередью-массивом. Принимает указатель на абстрактную очередь

```
void init_array_debug(debug_queue_t *debug_queue);
```

Функция записи элемента в очередь-список. Принимает указатель на очередь, элемент для записи. Возвращает флаг успешности выполнения

```
bool push_list(void *list_queue, value_t value);
```

Функция удаления элемента из очереди-списка. Принимает указатель на очередь, указатель на элемент для чтения. Возвращает флаг успешности выполнения

```
bool pop_list(void *list_queue, value_t *value);
```

Функция печати очереди-списка. Принимает указатель на очередь

```
bool print_list(void *list_queue);
```

Функция освобождения очереди-списка. Принимает указатель на очередь

```
void free_list(void *list_queue);
```

Функция инициализации абстрактной очереди так, чтобы она был очередью-списком. Принимает указатель на абстрактную очередь

```
void init_list_debug(debug_queue_t *debug_queue);
```

Функция прогона симуляции. Принимает указатель на структуру времен-условий, указатели на 2 пустые очереди, флаг необходимости печати информации, изменяет по указателю максимальную суммарную длину очередей.

```
int run_simulation(params_t *params, abstract_queue_t *queue_1,
abstract_queue_t *queue_2, bool verbose, int *max_queue_len);
```

Функция анализа разных реализаций очереди. Принимает указатель на структуру времен-условий

```
void run_analysis(params_t *params);
```

**Набор тестов**

**Обработка пунктов меню**

| Описание теста | Вход | Результат |
|---|---|---|
| Отрицательное число | -1 | Для выбора пункта меню введите целое число от 0 до 10 |
| Число больше 10 | 11 | Для выбора пункта меню введите целое число от 0 до 10 |
| Пустой ввод | | Для выбора пункта меню введите целое число от 0 до 10 |
| Максимальный пункт меню | 10 | [сведения о скорости симуляций] |
| Минимальный пункт меню | 0 | [выход из программы] |
| Символ вместо числа | a | Для выбора пункта меню введите целое число от 0 до 10 |
| Выбор пунктов меню, которые требуют инициализированную очередь, при неинициализированной очереди | 3 | Нельзя производить данное действие над неинициализированной очередью! |
| Удаления элемента из очереди при пустой очереди | 4 | Очередь пуста! |
| Запись в очередь при полной очереди | 3<br>1.2 | Очередь переполнена! |

**Запись элемента в очередь**

| Описание теста | Ввод | Вывод |
|---|---|---|
| Ввод целого числа | 3 | В очередь успешно добавлено число |
| Ввод вещественного числа | .5e5 | В очередь успешно добавлено число |
| Ввод отрицательного числа | -3 | Введено некорректное значение! |

| | | |
|---|---|---|
| Ввод постороннего символа | - | Введено некорректное значение! |
| Ввод числа с посторонним символом | 3e | Введено некорректное значение! |
| Ввод числа с 2 знаками | ++3 | Введено некорректное значение! |
| Пустой ввод | | Введено пустое значение! |

**Симуляция ОА**

| Описание теста | Значения времен | Вывод |
|---|---|---|
| Времена по умолчанию | 1 5 0 3 0 4 0 1<br>Время добавления 1: 3<br>Время добавления 2: 1.5<br>Время обработки 1: 2<br>Время обработки 2: 0.5<br>Теоретическое время: 3000 (см. Пример теоретического расчета) | Симуляция заняла всего: 3036.53 е.в.<br>Теоретический расчет времени: 3000.00 е.в.<br>Погрешность расчета: 1.20% |
| Время обработки заявок первого типа больше времени их поступления | 0 1 0 3 0 2 0 2<br>Время добавления 1: 0.5<br>Время добавления 2: 1.5<br>Время обработки 1: 1<br>Время обработки 2: 1<br>Очередь 1 не отдает приоритет<br>Теоретическое время – время обработки 1000 заявок первого типа, равно 1 * 1000 = 1000 | Симуляция заняла всего: 1001.80 е.в.<br>Теоретический расчет времени: 1000.00 е.в.<br>Погрешность расчета: 0.18% |

| | | |
|---|---|---|
| Время определяется временем прихода заявок 1 типа | 0 10 0 20 0 5 3 5<br>Время добавления 1: 5<br>Время добавления 2: 10<br>Время обработки 1: 2.5<br>Время обработки 2: 4<br>Теоретическое время – время поступления 1000 заявок первого типа, равно 5 * 1000 = 5000 | Симуляция заняла всего: 5082.17 е.в.<br>Теоретический расчет времени: 5000.00 е.в.<br>Погрешность расчета: 1.62% |
| Нестабильная работа: время обработки заявок лишь немного меньше времени поступления | 0 5 0 5 0 4 0 4<br>Время добавления 1: 2.5<br>Время добавления 2: 2.5<br>Время обработки 1: 2<br>Время обработки 2: 2<br>Обе очереди обрабатываются, время работы равно 2 * (2.5 / 2.5 * 1000) + 2 * 1000 = 4000 | Симуляция заняла всего: 3456.21 е.в.<br>Теоретический расчет времени: 4000.00 е.в.<br>Погрешность расчета: 15.73% (Погрешность случайная, вплоть до 50%) |
| Время поступления заявок 2 типа меньше чем их обработки | 0 2 0 4 0 1 0 7<br>Время добавления 1: 1<br>Время добавления 2: 2<br>Время обработки 1: 0.5<br>Время обработки 2: 3.5<br>Очередь 2 никогда не отдаст приоритет, если бы приоритет отсутствовал, то время работы было бы равно 3.5 * (1 / 2 * 1000) + 0.5 * 1000 = 2250 | Бесконечная работа программы для списка, переполнение для массива |

## Результаты сравнения

Для анализа я измерил средние скорости моделирования при разных реализациях и получил следующие результаты (размер статической очереди 2000)

Среднее время моделирования списками: 507 мкс

Среднее время моделирования массивами: 409 мкс

Выигрыш времени от очереди на массиве: 19.33%

Размер очереди на массиве: 16016 байт

Размер очереди на списке: 4472 байт

Выигрыш памяти от очереди на списке: 72.08%

Таким образом, при использовании очереди, реализованной на массиве, время моделирования на стандартных настройках уменьшается на 20 процентов, а при использовании очереди на списке максимальное использование меньше на 70%

При использовании очереди не на максимальную глубину, очередь на списке занимает гораздо меньше памяти, так как она выделяет память только под реально нужное количество элементов. Однако при минимально возможном размере (без переполнения на стандартных настройках) статической очереди в 400 элементов получим следующие результаты по памяти

Размер очереди на массиве: 3216 байт

Размер очереди на списке: 4472 байт

Выигрыш памяти от очереди на списке: -39.05%

Таким образом, в этом случае очередь на списке занимает на 40% больше памяти, так как он хранит не только сами элементы, но и указатели на каждый элемент.

Также я измерил среднее время добавления и удаления элементов с очередями на списке и на массиве и получил следующие результаты

Среднее время добавления элемента в очередь-список: 62 нс

Среднее время добавления элемента в очередь-массив: 13 нс

Выигрыш времени при добавлении в очередь на массиве: 79.03%

Среднее время удаления элемента из очереди-списка: 21 нс

Среднее время удаления элемента из очереди-массива: 10 нс

Выигрыш времени при удалении из очереди на массиве: 52.38%

Таким образом, основные операции над очередью на массиве выполняются на 50-80% быстрее, чем над очередью на списке, благодаря использованию статической памяти и отсутствии необходимости в дополнительных действиях (освобождение или выделение памяти).

То, что выигрыш при моделировании при использовании массива сильно меньше, чем выигрыш во времени исполнении основных операций, объясняется тем, что существенную часть времени работы моделирования составляет вызов функций получения случайного числа, проверка условий и вещественная арифметика.

## Вывод

При решении задач, которые требуют только возможности записи элемента в конец и чтения элемента с начала, таких как симулирование очередей, целесообразно вместо обычной структуры данных, такой как массив или список, использовать абстрактный тип данных – очередь, так как он позволяет менять его реализацию без изменения основного кода программы.

Способ, которым реализовывать очередь эффективнее, зависит от того, какие данные планируется хранить и что важнее – быстродействие или память. Если важнее быстродействие или если элементы очереди имеет размер сравнимый или меньший чем размер указателя и известен максимальный размер очереди, то эффективнее использовать статический массив.

Если важнее меньшее использование памяти или максимальная глубина очереди не ограничена, то имеет смысл использовать список

При использовании очереди, реализованной на массиве, время симуляции ОА уменьшается примерно на 20 процентов, а время добавления элемента в очередь снижается на 80%, а удаления элемента из очереди – на 50%.

При использовании очереди не на максимальную глубину, очередь на списке занимает гораздо меньше памяти, так как она выделяет память только под реально нужное количество элементов. Однако при использовании очереди на максимальную глубину очередь на списке занимает на 50% больше памяти, так как она хранит не только сами элементы, но и указатели на каждый элемент.

Также, при использовании очереди на списке, память обычно не фрагментируется – если в цикле производить добавление и удаление элемента из очереди, то он почти всегда будет записываться по одному и тому же адресу.

## Ответы на контрольные вопросы

1. Что такое FIFO и LIFO?

FIFO и LIFO – принципы организации структур данных, в основном очередей и стеков. LIFO - последним пришел – первым ушел, FIFO – первым пришел – первым ушел.

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

При реализации очереди через статический массив всегда выделяется (макс. кол-во элементов)*(размер элемента) = 16КБ, а при реализации через список динамически выделяется нужное количество памяти (реальное кол-во элементов)*(размер элемента + размер указателя) (на стандартных настройках 5Кб)

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

При реализации очереди через статический массив при удалении память не освобождается, а при реализации через список освобождается память из под вершины.

4. Что происходит с элементами очереди при ее просмотре?

Чтобы ”честно” прочитать элемент очереди, нужно сначала удалить его из очереди, а потом записать в конец, и переписать так всю очередь

5. От чего зависит эффективность физической реализации очереди?

Эффективность физической реализации очереди зависит от того, какое количество действий выполняется при добавлении и вставке, как это количество зависит от количества элементов в очереди, и от такого, используется ли динамическая память.

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

Достоинства массива со сдвигами в том, что его реализация самая простая.

Достоинство статического кольцевого массива в том, что он позволяет достичь наилучшей скорости работы, однако требует чуть более сложного программного кода.

Достоинство списка в практически неограниченном размере очереди, однако это приводит к замедлению работы из-за использования динамической памяти.

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

Фрагментация памяти — это явление, при котором память компьютера разбивается на небольшие, несмежные блоки, что приводит к неэффективному использованию доступной памяти. Фрагментация возникает в куче.

8. Для чего нужен алгоритм «близнецов».

Алгоритм близнецов нужен для минимизации фрагментации памяти

9. Какие дисциплины выделения памяти вы знаете?

First Fit (Первый подходящий) — выделяет первый достаточно большой блок памяти.

Best Fit (Лучший подходящий) — выделяет наименьший блок памяти подходящего размера.

Worst Fit (Худший подходящий) — выделяет самый большой доступный блок памяти.

10. На что необходимо обратить внимание при тестировании программы?

При тестировании программы нужно обратить внимание на работу симуляции при начальных условиях, приводящих программу в разные ситуации (пустые очереди, заполняющиеся очереди, нестабильная работа)

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

При динамических запросах память выделяется и освобождается следующим образом:

1. Выделение памяти:

- Запрос: Программа запрашивает определенное количество памяти.

- Поиск: Система управления памятью (например, менеджер кучи) ищет свободный блок памяти достаточного размера.

- Выделение: Если подходящий блок найден, он отмечается как занятый и возвращается программе.

- Разделение: Если найденный блок больше запрашиваемого, он может быть разделен на два блока: один для выделения, другой остается свободным.

2. Освобождение памяти:

- Освобождение: Программа уведомляет систему управления памятью, что блок памяти больше не используется.

- Объединение: Система управления памятью может попытаться объединить освобожденный блок с соседними свободными блоками, чтобы уменьшить фрагментацию.

- Обновление: Свободный блок добавляется в список свободных блоков для последующего использования.