**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 «Работа с очередью»

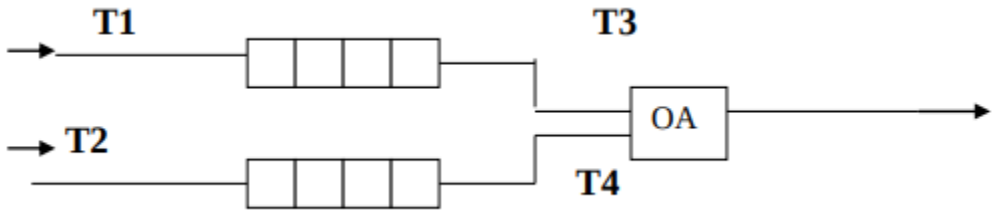
Студент Романов Семен Константинович

Группа ИУ7 – 35Б

Проверено Барышникова Марина Юрьевна

## ***Описание 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 **T1** и **T2**, 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 **T3** и **T4**, 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 2-го типа может войти в ОА, если в системе нет заявок 1-го типа**. Если в момент обслуживания заявки 2-го типа в пустую очередь входит заявка 1-го типа, то она немедленно поступает на обслуживание; обработка заявки 2- го типа прерывается, и она возвращается в "хвост" своей очереди (*система с абсолютным приоритетом и повторным обслуживанием*).

Смоделировать процесс обслуживания первых 1000 заявок 1-го типа, выдавая после обслуживания каждых 100 заявок 1-го типа

информацию о **текущей** и **средней длине каждой очереди**. В конце процесса выдать **общее время моделирования** и **количестве вошедших в систему** и **вышедших из нее заявок обоих типов, среднем времени пребывания заявок в очереди, количестве «выброшенных» заявок второго типа**. Обеспечить по требованию пользователя выдачу на экран адресов элементов очереди при удалении и ОА добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

# Техническое задание:

### Входные данные:

1. **Целое число (номер команды):** целое число от 0 до 9 (см. Функции программы)
2. **Числовые обозначения:** количество чисел в очереди, интервалы времени

### Выходные данные:

1. **Результаты работы с очередью:**
    a. Каждые 100 обработанных элементов: информация о текущей и средней длине каждой очереди.
    b. В конце процесса: общее время моделирования и количество вошедших в систему и вышедших из нее заявок обоих типов, среднее время пребывания заявок в очереди, количество «выброшенных» заявок второго типа
2. **Эффективность работы команды:** Затраченное время и память

**Способ обращения к программе:** запускается из терминала командой ./app.exe

## Функции программы:

1. Вывести результаты работы ОА на основе массива
2. Вывести характеристику работы ОА на основе массива
3. Вывести результаты работы ОА на основе списка
4. Вывести характеристику работы ОА на основе списка
0. Выход

## Аварийные ситуации:

1. Неверный тип данных при вводе числовых значений: Вывод сообщения “Invalid Input”
2. Неверная команда: Сообщение “Unknown command”
3. Переполнение массива: Сообщение “Невозможно добавить элемент. Очередь переполнена”

# Структуры данных:

## Для очереди на основе связанного списка:

```
typedef struct node
{
    float work_time;
    struct node *next;
} node_t;
```

**Поля структуры:**

- **Float work_time:** время, за которое будет обработана заявка
- **Struct node *next:** указатель на следующее значение очереди (в случае головного элемента – указатель NULL)

```
typedef struct List
{
    node_t *head;
    node_t *bottom;
    int count;
    inter_t to_arrive, to_process;
} list_t;
```

**Поля структуры:**

- **Node_t *head:** указатель на “голову” очереди
- **Node_t *bottom:** указатель на “низ” очереди
- **Int count:** количество элементов в очереди
- **Inter_t to_arrive:** временной промежуток прибытия заявки
- **Inter_t to_process:** временной промежуток обработки заявки

**Адреса свободных областей (на основе связанного списка):**

```
typedef struct node_mem
{
    size_t address;
    struct frmem *next;
} mem_t;
```

**Поля структуры:**

- **Long address:** значение освободившегося аддресса
- **Struct frmem *next:** указатель на следующее расположение аддресса (в случае последнего элемента – указатель NULL)

## Для очереди на основе массива:

```
typedef struct
{
    float *array;
    float *head;
    float *bottom;
    int size;
    int count;
    inter_t to_arrive, to_process;
} arr_t;
```

**Поля структуры:**

- **Float *array:** указатель на начало массива
- **Float *head:** указатель на “голову” очереди
- **Float *bottom:** указатель на “голову” очереди
- **Int size:** размер массива
- **Int count:** количество элементов в очереди
- **Inter_t to_arrive:** временной промежуток прибытия заявки
- **Inter_t to_process:** временной промежуток обработки заявки

## Алгоритм

1. На вход подается команда от 0 до 4
2. При симуляции обработки очереди с помощи массива, создаются два указателя: на голову и хвост очереди. В случае добавления элемента, элемент добавляется в голову, после чего указатель перемещается вперед, в случае же удаления, элемент считывается с хвоста, после чего указатель перемещается вперед.

В случае, когда количество элементов равно размеру массива, новый элемент не добавляется

В случае, если один из указателей уходит за массив, то он возвращается на его начало (Кольцевая обработка)

3. При симуляции обработки очереди с помощи списка, создаются два указателя: на голову и хвост очереди. В случае добавления элемента, выделяется память под новый элемент, по его адресу записывается значение элемента после чего голова очереди начинает ссылаться на этот элемент, делая его новой головой списка (сама голова всегда ссылается на NULL). В случае же удаления, элемент считывается с хвоста, после чего указатель перемещается по записанному адресу, а память выделенная под “нод” освобождается.

   При добавлении и освобождении элементов происходит их проверка на переиспользование адресов.

4. Пользователь выполняет действия с таблицей, пока не введет 0 (Выход)

## Теоретические и практические результаты работы программы

*Количество элементов: N = 1000*

*Интервалы прибытия: T1 = 1 – 5(avg. 2.5), T2 = 0 – 3(avg. 1.5)*

*Интервалы обработки: T3 = 0 – 4 (avg. 2), T4 = 0 – 1 (avg. 0.5)*

*Теоретическое время работы: T = max(T1 , T3) * N; T = 2.5 *1000 = 2500*

*Количество вошедших заявок первого типа: T/T1 = 2500/2.5 = 1000*

*Количество обработанных заявок первого типа: 1000 (по условию)*

*На практике:*

```
Рабочее время автомата: 2952.566162 (ожидаемое рабочее время: 3000.000000, погрешность
: 1.581126%)

Число вошедших заявок: 2973
Число вошедших заявок первого типа: 1000
Число вошедших заявок второго типа: 1973

Число вышедших заявок: 2515
Число вышедших заявок первого типа: 1000
Число вышедших заявок второго типа: 1535

Число возвратившихся заявок: 553
Число срабатываний автомата: 3088
Время простоя автомата: 6.738000
```

# Тесты

| N | Test | Input | Output |
|---|---|---|---|
| 1 | Некорректная комманда | 11 | “Unknown command” |
| 2 | Некорректная комманда | Abc | “Invalid Input” |
| 3 | Некорректный ввод интервала | -1 5 | “Invalid Input” |
| 4 | Добавление в заполненную очередь | | “Невозможно добавить элемент, очередь переполнена” |

## Оценка эффективности:

Все время измеряется в тактах

## Время добавления элементов:

| Size | Array | List |
|---|---|---|
| 10 | 30 | 83 |
| 100 | 344 | 696 |
| 1000 | 2225 | 4802 |

## Время удаления элементов:

| Size | Array | List |
|---|---|---|
| 10 | 21 | 60 |
| 100 | 383 | 564 |
| 1000 | 2542 | 3991 |

## Занимаемая память в байтах:

Примечание: в случае заполненности массива из 10 элементов только на 20% (2 элемента), то памяти затрачено будет все еще как на 10, в то время как список будет занимать память ровно на 2 элемента (2 * 16).

Все размеры в байтах.

| Size | Количество элементов от размера массива | Array | List |
|---|---|---|---|
| 10 | 20% | 40 + 16 | 32 |
| | 40% | | 64 |
| | 60% | | 96 |
| | 80% | | 120 |
| | 100% | | 160 |
| 100 | 20% | 400 + 16 | 320 |
| | 40% | | 640 |
| | 60% | | 960 |

| | | | |
|---|---|---|---|
| | 80% | | 1200 |
| | 100% | | 1600 |
| 1000 | 20% | 4000 + 16 | 3200 |
| | 40% | | 6400 |
| | 60% | | 9600 |
| | 80% | | 12000 |
| | 100% | | 16000 |

Таким образом, список является более эффективным по памяти решением, чем массив, только при условии заполненности массива не более чем на 25%

# Контрольные вопросы

## 1. Что такое очередь?

Очередь - структура данных, для которой выполняется правило FIFO, то есть первым зашёл - первым вышел. Вход находится с одной стороны очереди, выход - с другой.

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

При хранении кольцевым массивом: кол-во элементов * размер одного элемента очереди. Память выделяется на стеке при компиляции, если массив статический. Либо память выделяется в куче, если массив динамический. При хранении списком: кол-во элементов * (размер одного элемента очереди + указатель на следующий элемент). Память выделяется в куче для каждого элемента отдельно.

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

При хранении кольцевым массивом память не освобождается, а просто меняется указатель на конец очереди. При хранении списком, память под удаляемый кусок освобождается.

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

Эти элементы удаляются из очереди.

## 5. Каким образом эффективнее реализовывать очередь. От чего это зависит?

Зная максимальный размер очереди, лучше всего использовать кольцевой статический массив. Не зная максимальный размер, стоит использовать связанный список, так как такую очередь можно будет переполнить только если закончится оперативная память.

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

При использовании линейного списка тратится больше времени на обработку операций с очередью, а так же может возникнуть фрагментация памяти. При реализации статическим кольцевым массивом, очередь всегда ограничена по размеру, но операции выполняются быстрее, нежели на списке.

## 7. Что такое фрагментация памяти?

Фрагментация памяти - разбиение памяти на куски, которые лежат не рядом друг с другом. Можно сказать, что это чередование свободных и занятых кусков памяти.

## 8. На что необходимо обратить внимание при тестировании программы?

При тестировании программы необходимо обратить внимание на эффективное выделение и корректное освобождение динамической памяти.

Помимо этого стоит обратить внимание на корректность реализации кольцевого массива, чтобы не произошло записи в невыделенную область памяти. Еще стоит обратить внимание на возникновение фрагментации памяти.

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

При запросе памяти, ОС находит подходящий блок памяти и записывает его в «таблицу» занятой памяти. При освобождении, ОС удаляет этот блок памяти из «таблицы» занятой пользователями памяти.

## Вывод

Очередь, реализованная связанным списком, внушительно проигрывает как по времени, так и по памяти в данной реализации в случае полной заполненности массива, однако при заполненности меньше 25%, список является более эффективным по памяти решением. Таким образом, можно сделать вывод, что если нужно реализовать такую структуру данных как очередь, то лучше использовать массив, а не связанный список. Однако всегда нужно держать в голове тот факт, что массив обладает ограничениями, в то время как связанный список – нет.