**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

Студент **Простев Тимофей Александрович**

Группа **ИУ7-33Б**

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

Студент _____ **Простев Т. А.**

Преподаватель _____ **Никульшина Т. А.**

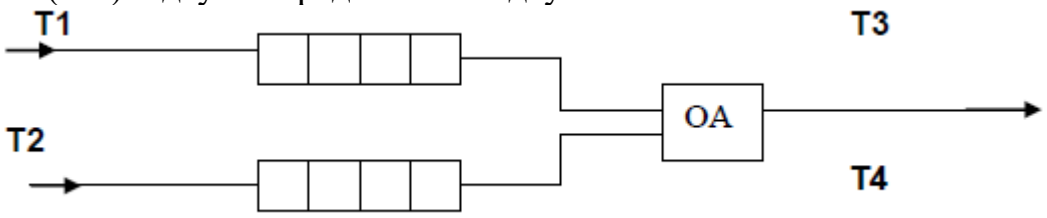
Преподаватель _____ **Барышникова М. Ю.**

Оценка _____

*2023 г.*

1. 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 **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 ждет первого освобождения ОА и далее поступает на обслуживание (система с **относительным** приоритетом).

Смоделировать процесс обслуживания первых 1000 заявок **1-го типа.** Выдать на экран после обслуживания каждых 100 заявок **1-го типа** информацию о текущей и средней длине каждой очереди, количестве вошедших и вышедших заявок и о среднем времени пребывания заявок в очереди. В конце процесса выдать 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.

2. Техническое задание.

Входные данные:

- Номер пункта меню: целое число от 1 до 8:
  Демонстрация работы очереди:
  - 1 – Добавить в очередь
  - 2 – Удалить из очереди
  - 3 – Вывести состояние очереди
  - 4 – Исследование операций над очередью
  Симуляция
  - 5 - Выполнить симуляцию
  - 6 – Выполнить симуляцию с выводом адресов
  - 7 – Исследование времени симуляции
  - 8 – Выход
- Добавление в очередь – целое число от $-2^{31}$ до $2^{31}-1$.
- Количество заявок 1-го типа, которые должны быть обработаны во время симуляции.
- Шаг вывода информации о симуляции.

Выходные данные:

- Число – результат удаления из очереди.
- Числа – состояние очереди.
- Текущие длины очередей.
- Средние длины текущих очередей.
- Количество обработанных заявок.
- Общее время моделирования.

Способ обращения к программе:

Запуск исполняемого файла. В рабочей директории выполнить команду ./app.exe.

Аварийные ситуации:

- Ввод некорректных данных.
- Ошибка выделения памяти.
- Переполнение очереди.

В случае аварийной ситуации выводится сообщение об ошибке.

3. Описание внутренних структур данных.

Структура очереди на статическом массиве:

```
struct queue {
    int data[QUEUE_MAX_SIZE];
    size_t head;
    size_t tail;
    size_t length;
};
```

Структура содержит 3 поля:

- `data` – значения очереди - статический массив типа `int`.
- `head` – номер «головного» элемента очереди в массиве – беззнаковое целое число типа `size_t`.
- `tail` - номер «хвостового» элемента очереди в массиве – беззнаковое целое число типа `size_t`.
- `length` – количество элементов в очереди– беззнаковое целое число типа `size_t`.

Структура узла списка:

```
typedef struct node node_t;
struct node {
    int data;
    node_t *next;
}
```

Структура содержит 2 поля:

- `data` – данные узла списка – целое число типа `int`.
- `next` – указатель на следующий узел списка.

Структура очереди на списке:

```
struct queue {
    node_t *head_parent;
    node_t *tail;
};
```

Структура содержит 2 поля:

- `head_parent` – указатель на родителя головного узла.
- `tail` – указатель на хвостовой узел.

Для работы со структурной используются функции:

`queue_t* queue_new(void);` - создание очереди и выделение памяти под неё.

`void queue_delete(queue_t *queue);` - освобождение памяти, выделенной под очередь.

`int queue_push(queue_t *queue, int value);` - добавить элемент в очередь.

`int queue_pop(int *value, queue_t *queue);` - удалить элемент из очереди.

`int queue_is_full(queue_t *queue);` - заполнена ли очередь.

`int queue_is_empty(queue_t *queue);` - пуста ли очередь.

`size_t queue_get_length(queue_t *queue);` - получить количество элементов в очерди.

`size_t queue_get_size(queue_t *queue);` - получить размер, занимаемый очередью в памяти.

`void queue_print(queue_t *queue);` - вывести состояние очереди на экран.

`int simulate(queue_t *queue_1, queue_t *queue_2, size_t requests_count, size_t print_step, int is_research, int print_adresses, size_t *queue_1_max_size, size_t *queue_2_max_size);` - произвести симуляцию.

4. Описание алгоритма.

Выводится меню программы. Пользователь вводит номер команды, после чего выполняется действие.

При добавлении элемента в очередь на статическом массиве происходит проверка на то, заполнена ли очередь. Если да, возвращается код ошибки. Если нет, данные записываются в элемент массива под номером головного элемента, после чего номер головного элемента увеличивается на 1. Если достигнут конец массива, номеру головного элемента присваивается значение 0.

При добавлении элемента в очередь на списке создается новый узел списка. Если память под узел не выделилась, возвращается код ошибки. Если выделилась, данные записываются в новый узел. Если очередь пуста, указателю на узел хвоста присваивается указатель на новый узел. Иначе, указателю на родителя головного узла присваивается указатель на его следующий узел, указателю следующему его узла присваивается указатель на новый узел.

При удалении элемента из очереди на статическом массиве происходит проверка на пустоту очереди. Если очередь пуста, возвращается код ошибки. Если нет, в переменную-результат записываются данные, находящиеся под номером хвостового элемента в массиве, после чего номер хвостового элемента увеличивается на 1. Если достигнут конец массива, номеру хвостового элемента присваивается значение 0.

При удалении элемента из очереди на списке происходит проверка на пустоту очереди. Если очередь пуста, возвращается код ошибки. Если нет, в переменную-результат записываются данные, хранящиеся в хвостовом узле, после чего, указателю на хвостовой узел присваивается указатель на следующий его узел.

5. Оценка эффективности.

При сравнении эффективности по 100 раз производилось от 1000 и до 10000 вставок и удалений в очередь, после чего бралось среднее значение.

Время добавлении в очередь:

| | Время добавления, тики | |
|---|---|---|
| Количество операций | Очередь на массиве | Очередь на списке |
| 1000 | 3,0E+03 | 6,2E+03 |
| 2000 | 2,3E+03 | 6,2E+03 |
| 3000 | 2,4E+03 | 5,7E+03 |
| 4000 | 2,3E+03 | 5,8E+03 |
| 5000 | 2,2E+03 | 6,0E+03 |
| 6000 | 2,3E+03 | 5,6E+03 |
| 7000 | 2,5E+03 | 5,8E+03 |
| 8000 | 2,4E+03 | 5,7E+03 |
| 9000 | 2,4E+03 | 5,8E+03 |
| 10000 | 2,4E+03 | 5,6E+03 |

Время удаления из очереди:

| | Время удаления, тики | |
|---|---|---|
| Количество операций | Очередь на массиве | Очередь на списке |
| 1000 | 3,1E+03 | 4,7E+03 |
| 2000 | 2,3E+03 | 4,6E+03 |
| 3000 | 2,3E+03 | 4,4E+03 |
| 4000 | 2,3E+03 | 4,4E+03 |
| 5000 | 2,2E+03 | 4,5E+03 |
| 6000 | 2,3E+03 | 4,3E+03 |
| 7000 | 2,5E+03 | 4,4E+03 |
| 8000 | 2,4E+03 | 4,3E+03 |
| 9000 | 2,4E+03 | 4,3E+03 |
| 10000 | 2,4E+03 | 4,3E+03 |

Занимаемая память:

| | Размер, байты | |
|---|---|---|
| Количество операций | Очередь на массиве | Очередь на списке |
| 1000 | 40024 | 12016 |
| 2000 | 40024 | 24016 |
| 3000 | 40024 | 36016 |
| 4000 | 40024 | 48016 |
| 5000 | 40024 | 60016 |
| 6000 | 40024 | 72016 |
| 7000 | 40024 | 84016 |
| 8000 | 40024 | 96016 |
| 9000 | 40024 | 108020 |
| 10000 | 40024 | 120020 |

Сравнение времени выполнения операций над стеком и над очередью.

Добавление:

| | На массиве | | На списке | |
|---|---|---|---|---|
| Количество операций | Стек | Очередь | Стек | Очередь |
| 1000 | 9,6E+01 | 3,0E+03 | 8,4E+01 | 6,2E+03 |
| 2000 | 1,0E+02 | 2,3E+03 | 8,9E+01 | 6,2E+03 |
| 3000 | 9,1E+01 | 2,4E+03 | 8,8E+01 | 5,7E+03 |
| 4000 | 8,3E+01 | 2,3E+03 | 8,2E+01 | 5,8E+03 |
| 5000 | 8,0E+01 | 2,2E+03 | 8,0E+01 | 6,0E+03 |
| 6000 | 9,2E+01 | 2,3E+03 | 9,1E+01 | 5,6E+03 |
| 7000 | 7,8E+01 | 2,5E+03 | 7,9E+01 | 5,8E+03 |
| 8000 | 7,8E+01 | 2,4E+03 | 7,8E+01 | 5,7E+03 |
| 9000 | 7,9E+01 | 2,4E+03 | 7,8E+01 | 5,8E+03 |
| 10000 | 8,8E+01 | 2,4E+03 | 9,1E+01 | 5,6E+03 |

Удаление:

| | На массиве | | На списке | |
|---|---|---|---|---|
| Количество операций | Стек | Очередь | Стек | Очередь |
| 1000 | 7,7E+01 | 3,1E+03 | 4,1E+02 | 4,7E+03 |
| 2000 | 8,5E+01 | 2,3E+03 | 4,5E+02 | 4,6E+03 |
| 3000 | 8,3E+01 | 2,3E+03 | 3,6E+02 | 4,4E+03 |
| 4000 | 7,9E+01 | 2,3E+03 | 3,0E+02 | 4,4E+03 |
| 5000 | 7,8E+01 | 2,2E+03 | 2,1E+02 | 4,5E+03 |
| 6000 | 8,4E+01 | 2,3E+03 | 3,8E+02 | 4,3E+03 |
| 7000 | 7,8E+01 | 2,5E+03 | 2,5E+02 | 4,4E+03 |
| 8000 | 7,7E+01 | 2,4E+03 | 2,0E+02 | 4,3E+03 |
| 9000 | 7,6E+01 | 2,4E+03 | 2,4E+02 | 4,3E+03 |
| 10000 | 8,2E+01 | 2,4E+03 | 3,5E+02 | 4,3E+03 |

Время и занимаемая память при симуляции обработки очередей:

| | Время выполнения, тики | | Занимаемая память, байты | |
|---|---|---|---|---|
| Количество запросов | На массиве | На списке | На массиве | На списке |
| 1000 | 1,6E+06 | 1,4E+07 | 84050 | 9677 |
| 2000 | 3,0E+06 | 5,6E+07 | 84050 | 27182 |
| 3000 | 4,3E+06 | 1,7E+08 | 84050 | 38410 |
| 4000 | 6,0E+06 | 7,0E+08 | 84050 | 55826 |
| 5000 | 6,8E+06 | 5,0E+08 | 84050 | 68989 |
| 6000 | 8,3E+06 | 8,4E+08 | 84050 | 84189 |
| 7000 | 9,5E+06 | 1,2E+09 | 84050 | 96184 |
| 8000 | 1,1E+07 | 1,6E+09 | 84050 | 111530 |
| 9000 | 1,2E+07 | 2,3E+09 | 84050 | 121120 |
| 10000 | 1,4E+07 | 2,8E+09 | 84050 | 134500 |

6. Вывод.

Очередь, реализованная при помощи статического массива и заполняющаяся циклически выигрывает по времени добавления и удаления очередь, реализованную на списке. Однако очередь на массиве ограничена по памяти, в отличие от очереди на списке. Ели до момента запуска программы известно максимальный требуемый размер очереди, выгоднее использовать очередь на массиве, иначе, более эффективно будет реализовать очередь на списке.

Скорость добавления данных в очередь и стек примерно одинаковы. Добавление в очередь на статическом массиве может происходить дольше, чем в стек на статическом массиве из-за математических операций, выполнение которых нужно для организации циклического массива.

Во время симуляции обработки очередей, очередь, реализованная при помощи списка, проигрывает по времени выполнения симуляции очереди, реализованной при помощи статического массива. Однако при проведении симуляции до 6000 заявок, она выигрывала по памяти.

7. Контрольные вопросы.

1. Что такое FIFO и LIFO?

FIFO – принцип - первый вошел – первый вышел. LIFO – последний вошел – первый вышел. Примерами таких принципов являются типы данных стек (LIFO) и очередь (FIFO).

2. Каким образом и какой объем памяти выделяется под хранения очереди при различной её реализации?

При реализации очереди при помощи статического массива, выделяется память единожды под все элементы массива. При реализации очереди при помощи списка, память выделяется при каждом добавлении значения под новый узел.

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

При реализации очереди при помощи статического массива память освобождается только при удалении всей очереди. Объём выделенной памяти не уменьшается при удалении элемента. При реализации очереди при помощи списка, при удалении элемента память освобождается из-под одного узла.

4. Что происходит с элементами очереди при её просмотре?

Хвостовой элемент очереди выходит из неё при просмотре.

5. От чего зависит эффективность физической реализации очереди?

Эффективность зависит способа реализации. При использовании статического массива, действия происходят быстрее, чем при использовании динамической памяти.

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

Очередь, реализованная при помощи статического массива, выполняет операции быстрее, чем очередь, реализованная при помощи списка.

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

Фрагментация памяти – это разбиение памяти на куски, которые располагаются не вплотную друг к другу. Занятые и свободные области памяти чередуются. Фрагментация происходит в куче.

8. Для чего нужен алгоритм «близнецов»?

Алгоритм близнецов нужен для эффективной работы с динамической памятью. Когда требуется выделить блок памяти, система ищет блок заданного размера. Если найденный блок больше, чем требуемый, он разделяется на две равные части. Если блок уже занят, то система ищет следующий блок нужного размера. Этот процесс повторяется до тех пор, пока не будет найден подходящий блок или необходимый размер блока станет минимальным.

9. Какие дисциплины выделения памяти вы знаете?

Выделение первого подходящего блока памяти, выделение самого подходящего блока памяти.

10. На что необходимо обратить внимание при тестировании программы?

При работе со статической памятью необходимо отслеживать переполнение. При работе с динамической памятью необходимо отслеживать успешность выделения памяти.

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

При выделении памяти, в таблице адресов отмечается, что память используется, при освобождении отметка удаляется из таблицы.