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 **Паншин Сергей Константинович**

Группа **ИУ7-33Б**

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

Студент _____ **Паншин С. К.**

Преподаватель _____ **Никульшина Т. А.**

Преподаватель _____ **Барышникова М. Ю.**

Оценка _____

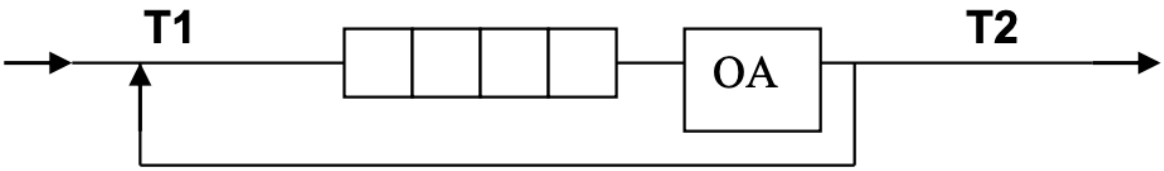
2023 г.

Условие задачи

Провести сравнительный анализ реализации алгоритмов включения и исключения элементов из очереди при использовании указанных структур данных, оценить эффективности программы по времени и по используемому объему памяти.

Техническое задание

Система массового обслуживания состоит из обслуживающего аппарата (ОА) и очереди заявок.



Заявки поступают в "хвост" очереди по случайному закону с интервалом времени T1, равномерно распределенным от 0 до 6 единиц времени (е.в.). В ОА они поступают из "головы" очереди по одной и обслуживаются также равновероятно за время T2 от 0 до 1 е.в., Каждая заявка после ОА вновь поступает в "хвост" очереди, совершая всего 5 циклов обслуживания, после чего покидает систему. (Все времена – вещественного типа) В начале процесса в системе заявок нет. Смоделировать процесс обслуживания до ухода из системы первых 1000 заявок, выдавая после обслуживания каждых 100 заявок информацию о текущей и средней длине очереди, а в конце процесса - общее время моделирования и количестве вошедших в систему и вышедших из нее заявок, количестве срабатываний ОА, время простоя аппарата. По требованию пользователя выдать на экран адресов элементов очереди при удалении и добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

Входные данные:

Пункт меню (число от 0 до 10):

```
Menu:
    1) Push in queue list
    2) Pop from queue list
    3) Print queue list
    4) Push in queue array
    5) Pop from queue array
    6) Print queue array
    7) Comparison of queue implementations
    8) View simulation parameters
    9) Simulation settings
    10) Start simulation
    0) Quit program
```

Также начало и конец интервала T1 и T2, и количество повторений обработки задачи.

Выходные данные:

После обслуживания каждых 100 заявок информацию о текущей и средней длине очереди, а в конце процесса - общее время моделирования и количестве вошедших в систему и вышедших из нее заявок, количестве срабатываний ОА, время простоя аппарата.

Возможные аварийные ситуации:

Некорректный ввод: пункта меню, не число, попытка очистить пустую очередь, попытка добавить элемент в полную очередь.

Способ обращения к программе

В папке с программой запустить команду *make run*.

Структуры данных

```
// данные задачи
typedef struct task
{
    int id;
    int count;
    double time_out;
} task_t;

// элемент массива
typedef struct array_element
{
    task_t task;
} array_element_t;

// очередь основанная на массиве
typedef struct queue_array
{
    array_element_t data[MAX_QUEUE_LEN];
    size_t size;
    size_t pin;
    size_t pout;
    int max_id;
} queue_array_t;

// элемент списка
typedef struct list_node
{
    task_t task;
    struct list_node *next;
} list_node_t;

// очередь основанная на списке
typedef struct queue_list
{
    list_node_t *head;
    list_node_t *tail;
    size_t size;
    int max_id;
    void **free_area;
    size_t free_area_size;
    size_t free_area_size_alloc;
} queue_list_t;

// очередь (внутри обе реализации)
typedef struct queue
{
    queue_array_t arr;
    queue_list_t list;
} queue_t;
```

Интерфейсы структур:

```
// инициализация элемента массива
// на вход: указатель на элемент
void array_el_init(array_element_t *el);

// инициализация очереди на основе массива
// на вход: указатель на очередь
void queue_array_init(queue_array_t *queue);

// добавление элемента в очередь на основе массива
// на вход: указатель на очередь и указатель на элемент
// на выход: код ошибки или код что все хорошо
int queue_array_push(queue_array_t *queue, array_element_t *el);

// удаление элемента из очереди на основе массива
// на вход: указатель на очередь и указатель на элемент
// на выход: код ошибки или код что все хорошо
int queue_array_pop(queue_array_t *queue, task_t *task);

// печать очереди на основе массива
// на вход: указатель на очередь
void queue_array_print(queue_array_t *queue);

// инициализация узла
// на вход: указатель на указатель узла
// на выход: код ошибки или код что все хорошо
int list_node_init(list_node_t **node);

// освобождение памяти из-под узла
// на вход: указатель узел
void list_node_free(list_node_t *node);

// копирование узла
// на вход: указатель на узел в который копирует и на исходный
void list_node_copy(list_node_t *dst, list_node_t *src);

// инициализация очереди на основе списка
// на вход: указатель на очередь
// на выход: код ошибки или код что все хорошо
int queue_list_init(queue_list_t *queue);

// добавление элемента в очередь на основе списка
// на вход: указатель на очередь и указатель на узел
// на выход: код ошибки или код что все хорошо
int queue_list_push(queue_list_t *queue, list_node_t *node);

// удаление элемента из очереди на основе списка
// на вход: указатель на очередь и указатель на то куда положить
данные выходящего узла
// на выход: код ошибки или код что все хорошо
int queue_list_pop(queue_list_t *queue, task_t *task);
```

```
// печать очереди на основе списка
// на вход: указатель на очередь
void queue_list_print(queue_list_t *queue);

// освобождение памяти из-под очереди на основе списка
// на вход: указатель на очередь
void queue_list_free(queue_list_t *queue);


// инициализация массива освобождаемых адресов
// на вход: указатель на очередь и необходимый размер
// на выход: код ошибки или код что все хорошо
int init_free_area(queue_list_t *queue, size_t size);

// изменения размера массива освобождаемых адресов
// на вход: указатель на очередь и новый размер
// на выход: код ошибки или код что все хорошо
int realloc_free_area(queue_list_t *queue, size_t new_size);

// добавления адреса в массив освобождаемых адресов
// на вход: указатель на очередь и добавляемый адрес
// на выход: код ошибки или код что все хорошо
int add_free_area(queue_list_t *queue, void *ptr);

// освобождение памяти из-под массива освобождаемых адресов
// на вход: указатель на очередь
void free_free_area(queue_list_t *queue);

// проверка использует ли последний элемент адрес и удаляет его из
списка свободных если так
// на вход: указатель на очередь
// на выход: возвращает истину/ложь (1/0)
int check_if_mem_reused(queue_list_t *queue);

// инициализация двух видов очереди
// на вход: указатель на очередь
// на выход: код ошибки или код что все хорошо
int queue_init(queue_t *queue);

// освобождение памяти из-под двух видов очереди
// на вход: указатель на очередь
void queue_free(queue_t *queue);


// Получение длины очереди
// на вход: указатель на очередь и флаг реализации
// на выход: длина очереди
size_t queue_len(queue_t *queue, int is_queue_list);
```

```
// Добавления элемента в очередь
// на вход: указатель на очередь, флаг реализации и указатель на данные для добавления
// на выход: код ошибки или код что все хорошо
int queue_push(queue_t *queue, int is_queue_list, task_t task);

// Добавления элемента в очередь
// на вход: указатель на очередь, флаг реализации и указатель куда положить выходящие данные
// на выход: код ошибки или код что все хорошо
int queue_pop(queue_t *queue, int is_queue_list, task_t *task);
```

Теоретический расчет

Количество необходимое для выхода = 1000 в нашем случае

Если среднее отрезка T1 * на кол-во необходимое для выхода больше, чем среднее отрезка T2 * кол-во повторения обработки элемента * кол-во необходимое для выхода.

То

Ожидаемое время = среднее отрезка T1 * на кол-во необходимое для выхода

Иначе

Ожидаемое время = среднее отрезка T2 * на кол-во вызовов ОА

Для T1 = [0; 6], T2 = [0; 1] и количества повторения = 5 получаем ожидаемое время:

Ожидаемое время = 3000

Пример работы

```
------------------
Choose menu item (0-10):
8
Simulation parametrs:
T1 from 0.00 to 6.00:
T2 from 0.00 to 1.00:
Repeats count: 5
------------------
Menu:
    1) Push in queue list
    2) Pop from queue list
    3) Print queue list
    4) Push in queue array
    5) Pop from queue array
    6) Print queue array
    7) Comparison of queue implementations
    8) View simulation parameters
    9) Simulation settings
    10) Start simulation
    0) Quit program
------------------
Choose menu item (0-10):
10
Choose queue type for simulation:
0) Queue by array
1) Queue by list
1

-----------START SIMULATION-----------


----------
Processed 100 tasks.
Queue len: 0
Average queue len: 1.59
----------

----------
Processed 200 tasks.
Queue len: 2
Average queue len: 1.43
----------

----------
Processed 300 tasks.
Queue len: 4
Average queue len: 1.60
----------
```

```
----------
Processed 400 tasks.
Queue len: 0
Average queue len: 1.48
----------

----------
Processed 500 tasks.
Queue len: 1
Average queue len: 1.46
----------

----------
Processed 600 tasks.
Queue len: 0
Average queue len: 1.39
----------

----------
Processed 700 tasks.
Queue len: 1
Average queue len: 1.31
----------

----------
Processed 800 tasks.
Queue len: 2
Average queue len: 1.31
----------

----------
Processed 900 tasks.
Queue len: 1
Average queue len: 1.29
----------

----------
Processed 1000 tasks.
Queue len: 2
Average queue len: 1.31
----------
Machine working time: 3034.71 (Expected working time: 3000.00, inaccuracy: 1.16%)
Number of tasks entered: 1002
Number of tasks submitted: 1000
Number of failed tasks: 0
The number of machine triggers: 5002
Machine downtime: 538.53
The number of addresses taken from used memory: 4989
The number of addresses taken from new memory: 15
```

Замеры

Производится 50 итераций при замерах.

Время выполнения добавления элемента в очередь в наносекундах:

Размер очереди	Очередь в виде массива	Очередь в виде списка
10	140	220
25	220	560
50	380	1000
100	740	2060
250	1800	4820
500	3600	9020
1000	6220	22080

Время выполнения удаления элемента из очереди в наносекундах:

Размер очереди	Очередь в виде массива	Очередь в виде списка
10	60	240
25	280	560
50	400	1180
100	780	1940
250	1140	4060
500	3100	6900
1000	5000	13760

Память, занимаемая очередью разных размеров:

Размер очереди	Очередь в виде массива (занимаемая объектами)	Очередь в виде массива (выделенная)	Очередь в виде списка
10	160	16000	240
25	400	16000	600
50	800	16000	1200
100	1600	16000	2400
250	4000	16000	6000
500	8000	16000	12000
1000	16000	16000	24000

Выводы по проделанной работе

Очередь, реализованная связанным списком, проигрывает как по памяти, так и по времени обработки. Таким образом, можно сделать вывод, что если нужно реализовать такую структуру данных как очередь, то лучше использовать массив, а не связанный список. Но, когда заранее неизвестен размер очереди, то стоит использовать связанные списки, так как в отличии от статического массива, списки ограничены в размерах только размером оперативной памяти.

Контрольные вопросы

1. Что такое FIFO и LIFO?

FIFO – First In First Out (очередь)

LIFO – Last In First Out (стек)

Это две абстрактных структуры данных.

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. Память выделяется на стеке.

При хранении списком: кол-во элементов * (размер одного элемента + 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 массивом память не освобождается, а просто меняется указатель на конец очереди.

При хранении списком, память из-под удаляемого элемента освобождается.

4. Что происходит с элементами очереди при ее просмотре?

Элементы удаляются из очереди.

5. От чего зависит эффективность физической реализации очереди?

При реализации очереди в виде кольцевого статического массива, может возникнуть переполнение памяти. Быстрее работают операции добавления и удаления элементов.
При реализации в виде списка легче удалять и добавлять элементы, переполнение памяти может возникнуть только если закончится оперативная память, однако может возникнуть фрагментация памяти.
Если изначально знать размер очереди и тип данных, то лучше воспользоваться массивом. Не зная размер — списком.
Если важна скорость выполнения, то лучше использовать массив, так как все операции с массивом выполняются быстрее, но очередь ограничена по памяти (так как массив статический).

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

Очередь в виде списка проигрывает во всем реализации массива в виде кольцевого, кроме того, что ее размер ограничен только размером доступной оперативной памяти. В случае реализации массива не в циклическом в виде, то будет затрачиваться большое время на сдвиг хвоста.

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

Фрагментация памяти — разбиение памяти на куски, которые лежат не рядом друг с другом. В куче.

8. Для чего нужен алгоритм «близнецов».

Метод близнецов обеспечивает очень высокую скорость динамического выделения и освобождения памяти.

9. Какие дисциплины выделения памяти вы знаете?

Две основные дисциплины сводятся к принципам "самый подходящий" и "первый подходящий". По дисциплине "самый подходящий" выделяется тот свободный участок, размер которого равен запрошенному или превышает его на минимальную величину. По дисциплине "первый подходящий" выделяется первый же найденный свободный участок, размер которого не меньше запрошенного. При применении любой дисциплины, если размер выбранного для выделения участка превышает запрос, выделяется запрошенный объем памяти, а остаток образует свободный блок меньшего размера.

10. На что необходимо обратить внимание при тестировании программы?

Корректное освобождение памяти

11. 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 пользователем памяти.