|  |  |
| --- | --- |
|  | **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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

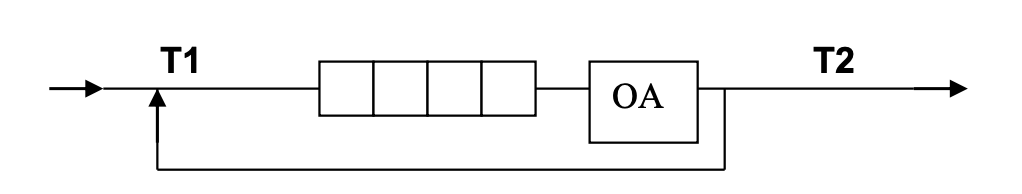
*2023 г.*

**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 Т1, 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 Т2 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) Сomparison of queue implementations

8) View simulation parameters

9) Simulation settings

10) Start simulation

0) Quit program

Также начало и конец интервала Т1 и Т2, 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 Т1 \* на кол-во необходимое для выхода больше, чем среднее отрезка Т2 \* кол-во повторения обработки элемента \* кол-во необходимое для выхода.

То

Ожидаемое время = среднее отрезка Т1 \* на кол-во необходимое для выхода

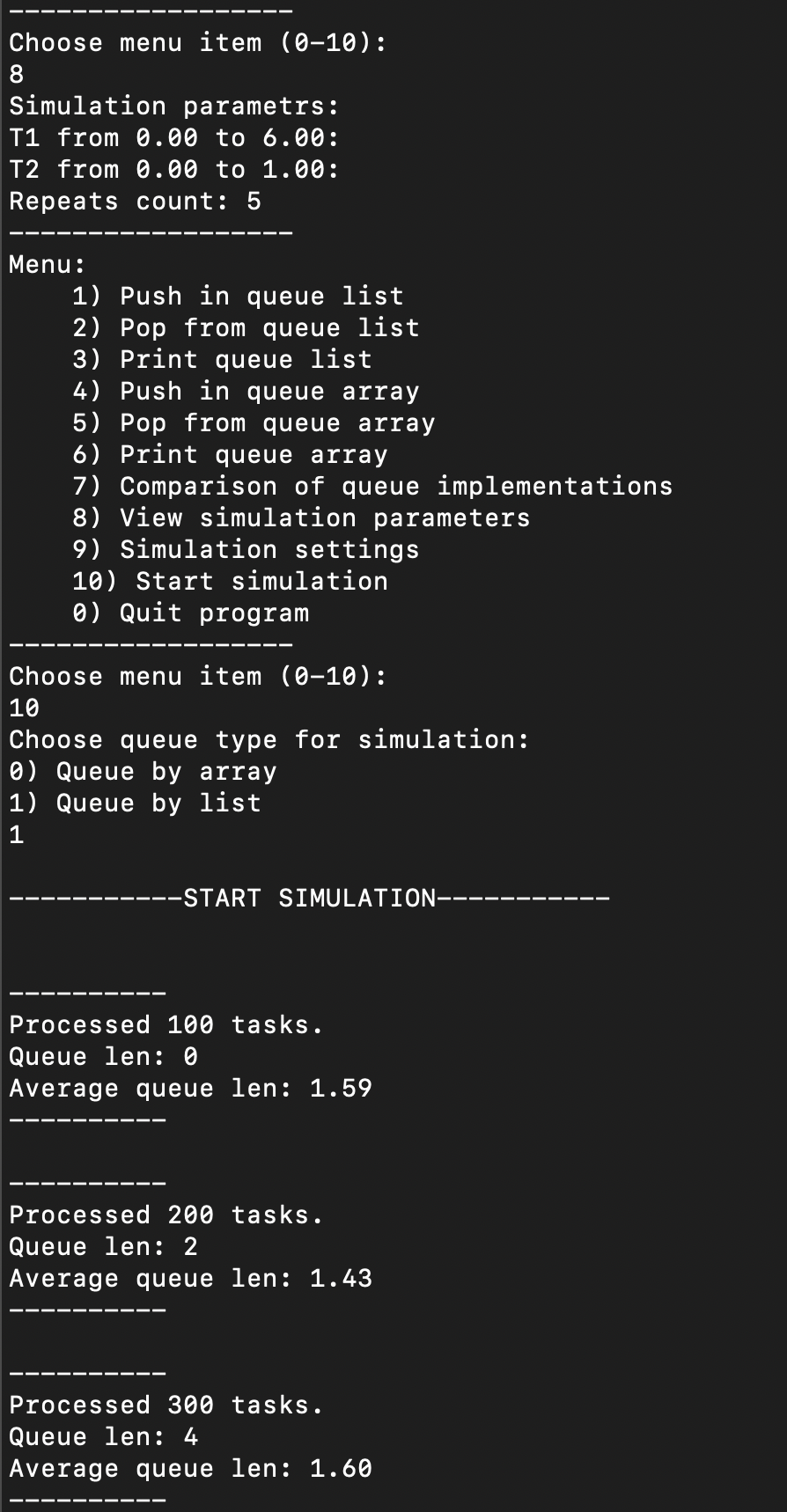
Иначе

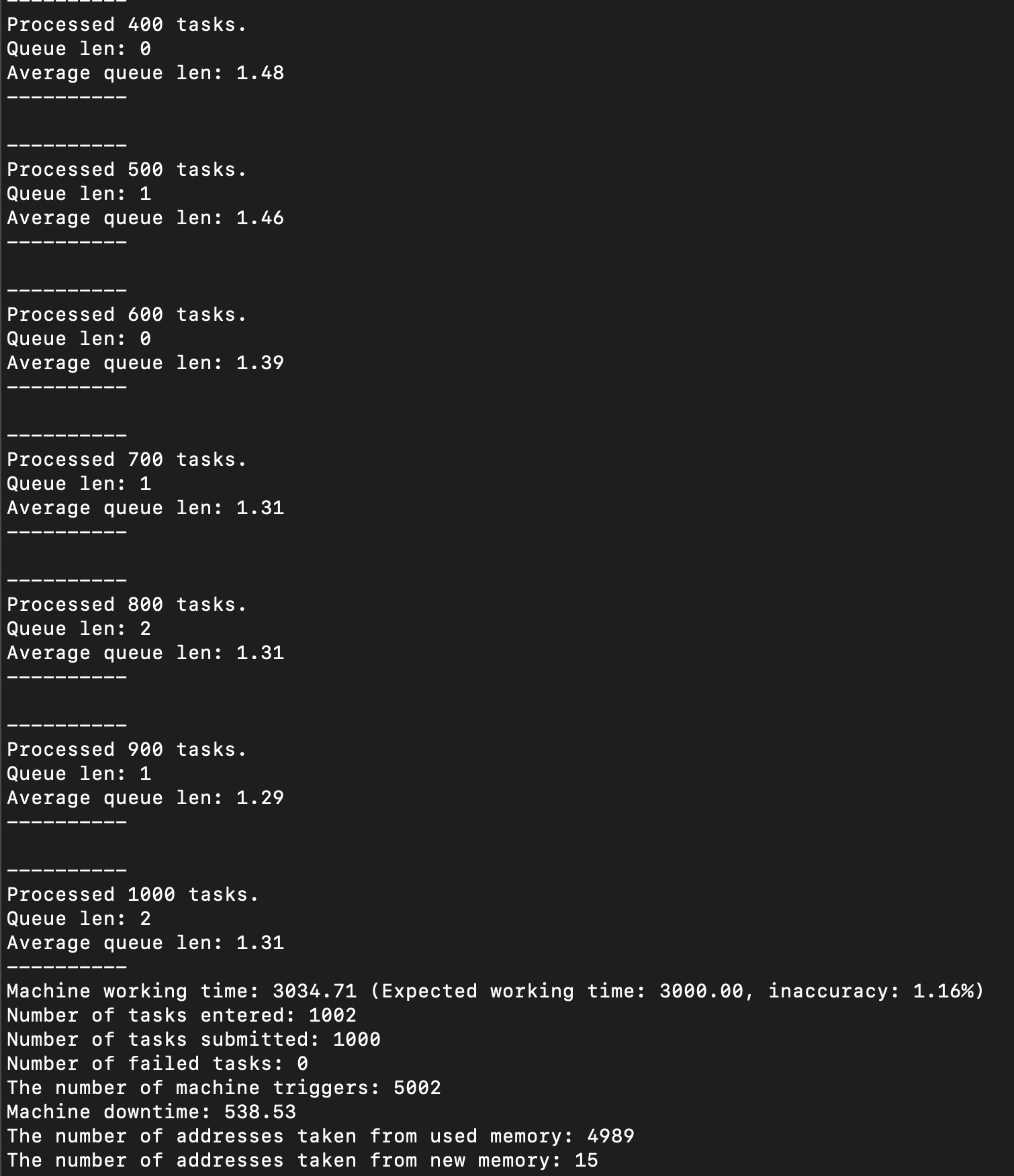
Ожидаемое время = среднее отрезка Т2 \* на кол-во вызовов ОА

Для Т1 = [0; 6], Т2 = [0; 1] и количества повторения = 5 получаем ожидаемое время:

Ожидаемое время = 3000

**Пример работы**



****

**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.