|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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

ДИСЦИПЛИНА Типы и структуры данных

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5 «ОБРАБОТКА ОЧЕРЕДЕЙ».**

Студент Иванов Павел Александрович

Группа ИУ7 – 35Б

Вариант №2

Проверил

**Оглавление**

[**Цель работы** 1](#_Toc86699734)

[**Описание технического задания** 3](#_Toc86699735)

[**Описание структур данных** 5](#_Toc86699736)

[**Описание алгоритма** 7](#_Toc86699737)

[**Набор тестов** 8](#_Toc86699738)

[**Оценка эффективности** 9](#_Toc86699739)

[**Ответы на контрольные вопросы** 10](#_Toc86699740)

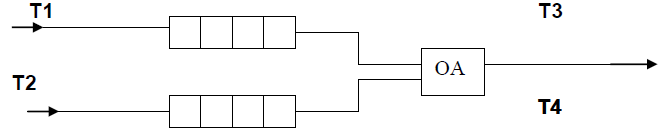
­

# **Цель работы**

Приобрести навыки работы с типом данных «очередь», представленным в виде одномерного массива и односвязного линейного списка, 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.

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 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.

Интерфейс программы должен быть понятен неподготовленному пользователю. При разработке интерфейса программы следует предусмотреть:

* возможность изменения времен прихода и обработки заявок с экрана;
* наличие пояснений при выводе результата;
* вывод процента погрешности работы программы.

Кроме того, нужно вывести на экран время выполнения программы при реализации очереди списком и массивом, а также требуемый при этом объем памяти.

При тестировании программы необходимо:

* проверить правильность работы программы при различном заполнении очередей, т.е., когда время моделирования определяется временем обработки заявок и когда определяется временем прихода заявок;
* отследить переполнение очереди, если очередь в программе ограничена.

При реализации очереди списком необходимо тщательно следить за освобождением памяти при удалении элемента из очереди. При этом по запросу пользователя должны выдаваться на экран адреса памяти, содержащие элементы очереди при добавлении или удалении элемента очереди. Следует проверять, происходит ли при этом фрагментация памяти (выделение непоследовательных адресов память).

# **Описание технического задания**

Входные данные:

*Целое число:*

Число от 0 до 4 — пункт меню или число заявок первого типа от 1 до 1 000 000.

*Вещественное число:*

Число от 0 до 10 000 — временная граница обработки или поступления заявки.

Выходные данные:

Числа – число обработанных заявок, длина и средняя длина очередей, число вошедших и вышедших заявок, среднее время, общее время выполнения программы, объем памяти, занимаемой структурами, в байтах.

Действие программы:

Выполнение операций с очередью, решение задачи с двумя очередями и одним обработчиком, над которым установлен относительный приоритет в пользу заявок первой очереди. Также сравнение эффективности операций при использовании различных структур данных.

*Меню программы:*

1 – Решение задачи с использованием массива

2 – Решение задачи с использованием списка

3 – Изменение исходных данных задачи

4 – Измерение времени и эффективности операций

Обращение к программе:

Запускается файл app.exe. Если его нет, из консоли следует вызывать make, и тогда запустить app.exe.

Аварийные ситуации:

* При очистке очистке очередь пустая

Результат: сообщение «Queue is already empty!»

* При добавлении возникло переполнение очереди

Результат: сообщение «Queue is already full!»

* Получение элемента из пустой очереди

Результат: сообщение «Queue is already empty!»

* Некорретный ввод пунта меню

Результат: сообщение «Wrong input!»

* Некорретный ввод числа заявок первой очереди

Результат: сообщение «Wrong input!»

* Некорретный ввод временных границ

Результат: сообщение «Wrong input!»

# **Описание структур данных**

Для хранения очереди — массива используется следующая структура:

typedef struct arr\_queue

{

int pin; // указатель на конец

int pout; // указатель на начало

int size; // размер массива

int count; // количество элементов в очереди

double \*arr; // массив, элементы коротого - время прихода

} arr\_queue;

Поля структуры:

int pin – указатель (индекс) на последний добавленный элемент;

int pout – указатель (индекс) на первый элемент в очереди;

int size – размер очереди;

int count – количество элементов в очереди;

double \*arr – сам массив, где хранятся элементы очереди.

Для хранения очереди — списка используется следующая структура:

typedef struct list\_queue

{

int count;

list\_node \*pout; // первый элемент на выход

list\_node \*pin; // адрес последнего элемента

} list\_queue;

Поля структуры:

int count — число элементов в стеке;

list\_node \*pout — указатель на первый элемент в очереди;

list\_node \*pin – указатель на последний элемент в очереди.

Для хранения списка свободных адресов используется:

typedef struct free\_mem

{

size\_t data;

struct free\_mem \*next;

} free\_mem;

Поля структуры:

size\_t data – адрес

struct free\_mem \*next – следующий элемент списка

Для обработчика используется следующая структура данных, которая хранит всю необходимую информацию:

typedef struct handler\_t

{

    int type1\_processed;

    int type2\_processed;

    double time\_standby;

    request\_t in\_process;

    float time\_finish;

} handler;

Поля структуры:

int type1\_processed – число обработанных заявок первого типа;

int type2\_processed – число обработанных заявок второго типа;

double time\_standby – время «простоя» обработчика;

request\_t in\_process – тип обрабатываемой заявки;

float time\_finish – время, к которому заявка будет обработана.

Тип заявки – перечислимый тип:

typedef enum { NONE, ONE, TWO } request\_t;

NONE – заявки нет;

ONE – заявка из первой очереди;

TWO – заявка из второй очереди.

*Основные функции программы:*

// Инициализация очереди

int arr\_queue\_init(arr\_queue \*queue);

// Добавление элемента в конец очереди

int arr\_queue\_push(arr\_queue \*queue, double come\_time);

// Удаление элемента из очереди

int arr\_queue\_pop(arr\_queue \*queue, double \*ret\_time);

// Освобождение памяти под очередь

void arr\_queue\_free(arr\_queue \*queue);

// Очистка очереди

void arr\_queue\_clean(arr\_queue \*queue);

// Печать очереди

void arr\_queue\_print(arr\_queue \*queue);

// Инициализвция очереди

void list\_queue\_init(list\_queue \*queue);

// Создание ноды списка

int create\_node(list\_node \*\*node, double tmp);

// Добавление элемента в очередь

int list\_queue\_push(list\_queue \*queue, double tmp, free\_mem \*mem);

// Удаление элемента из очереди

int list\_queue\_pop(list\_queue \*queue, double \*tmp, free\_mem \*mem);

// Очистка очереди

void list\_queue\_clean(list\_queue \*queue, free\_mem \*mem);

// Печать очереди

void list\_queue\_print(list\_queue \*queue, free\_mem \*mem);

// Инициализия списка свободных областей

void free\_mem\_init(free\_mem \*mem);

// Добавление адреса в список свободных адресов

int free\_mem\_add(free\_mem \*mem, size\_t add);

// Удаление адреса из списка свободных адресов при его наличии

void free\_mem\_del(free\_mem \*mem, size\_t del);

// Вывод списка свободных адресов

void free\_mem\_print(free\_mem \*mem);

# **Описание алгоритма**

1. Ввод номера команды;
2. Если выбрано моделирование с помощью массива:
   1. Создается цикл, который идёт пока не обработается нужное число заявок первого типа. В это же время создается счетчик времени. По ходу времени в очереди и обработчик добавляются новые заявки.
   2. По окончании цикла выводится сравнение полуенных результатов с идеальными
3. Если выбрано моделирование с помощью списка:
   1. Создается цикл, который идёт пока не обработается нужное число заявок первого типа. В это же время создается счетчик времени. По ходу времени в очереди и обработчик добавляются новые заявки.
   2. По окончании цикла выводится сравнение полуенных результатов с идеальными
4. Если выбрано сравнение по времени, то выводится сравнительная таблица времени добавления и удаления элемента из очереди, а также объём занимаемой структурами памяти при разной наполненности очередей.

# **Набор тестов**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тестовый случай | Пользовательский ввод | Результат |
| 1 | Введён неверный пункт меню | a | Wrong input! |
| 2 | Введён неверный пункт меню | 5 | Wrong input! |
| 3 | Введено неверное количество заявок первого типа | 0 | Wrong input! |
| 4 | Введено неверное количество заявок первого типа | -10 | Wrong input! |
| 5 | Введено неверное количество заявок первого типа | 1000001 | Wrong input! |
| 6 | Ввод неверного промежутка времени | -1 5 | Wrong input! |
| 7 | Ввод неверного промежутка времени | 0 0 | Wrong input! |
| 8 | Ввод неверного промежутка времени | 0.0 10000.01 | Wrong input! |
| 9 | Ввод неверного промежутка времени | 10.5 10.5 | Wrong input! |
| 10 | Ввод неверного промежутка времени | 10 -5 | Wrong input! |
| 11 | Ввод неверного промежутка времени | 5.0 3.0 | Wrong input! |
| 12 | Выход из программы | 0 | Программа завершилась |
| 13 | Запуск программы, использующей очередь на массиве | 1 | Обработано установленное число заявок, вывдено сравнение реального и ожидаемого времени |
| 14 | Запуск программы, использующей очередь на списке | 2 | Обработано установленное число заявок, вывдено сравнение реального и ожидаемого времени |
| 15 | Вывод результатов сравнения эффективности | 4 | Выведена сравнительная таблица операций по времени и по памяти |

# **Оценка эффективности**

Оценка эффективно­­сти проводится по операциям добавления элемента и извлечения элемента. Измеряется число тактов, усреднённое за 1000 прогонов:

Операция добавления:

|  |  |  |
| --- | --- | --- |
| Число элементов | Стек — массив | Стек — список |
| 10 | 40 | 488 |
| 100 | 34 | 1259 |
| 1000 | 31 | 6774 |

Операция извлечения:

|  |  |  |
| --- | --- | --- |
| Число элементов | Стек — массив | Стек — список |
| 10 | 30 | 320 |
| 100 | 31 | 2964 |
| 1000 | 35 | 23613 |

Память:

|  |  |  |
| --- | --- | --- |
| Число элементов | Стек — массив, байт | Стек — список, байт |
| 10 | 24016 | 240 |
| 100 | 24016 | 2400 |
| 1000 | 24016 | 24000 |

# **Ответы на контрольные вопросы**

1. *Что такое стек?*

Стек — последовательный список с переменной длинной, действия с которым (включение и исключение элементов) происходят только с одной его стороны — с его вершины. Эта структура данных используется по принципу Last In — First Out (LIFO).

*2. Каким образом и сколько памяти выделяется под хранение стека при различной его реализации?*

При реализации с помощью массива память под стек может выделяться как динамически, так и статически (в любом случае выделится непрерывный участок памяти ограниченного размера). Так, выделяется кусок памяти размера sizeof(тип элемента) \* (количество элементов в стеке). В первом случае память выделяется в куче, во втором — на стеке.

При реализации связным списком память выделяется каждый раз, когда добавляется новый элемент. Таким образом, число элементов в стеке ограничено только объёмом оперативной памяти.

1. *Каким образом освобождается память при удалении элемента стека при различной реализации стека?*

В случае с массивом, если память статическая, то происходит возврат текущего значения и переход указателя вниз на один элемент (память не освобождается). Если же память динамическая, то способ освобождения зависит от реализации: можно заново выделить память нового размера, освободив старую, причем заранее скопировав данные оттуда в новую. А можно работать так же, как со статическим, и освободить все ресурсы в конце программы.

В случае со списком, при удалении элемента нужно вернуть значение текущей ячейки, записать нулевой указатель в предыдущую ячейку и освободить память из-под текущей.

1. *Что происходит с элементами стека при его просмотре?*

Все элементы стираются, так как чтобы увидеть последний элемент (предыдущие просто недоступны), надо его достать.

1. *Каким образом эффективнее реализовывать стек? От чего это зависит?*

По результатам оценки эффективности видно, что стек в виде массива использовать гораздо эффективнее по времени. По памяти же эффективнее использовать список, если мы не знаем максимальное количество элементов в стеке (либо это количество сильно варьируется): тогда придется выделять большой кусок памяти, который может и не понадобиться, но он будет занимать излишнюю память.

Вывод

В ходе данной работы я познакомился со структурой данных «стек», разобрался с операциями, которые можно реализовывать на этой структуре и сравнил реализацию стека списком и массивом.

Получилось, что если важно время работы, то выгоднее использовать стек на массиве. Стек на списке же удобнее использовать когда мы не знаем максимальное количество элементов в стеке (либо это количество сильно варьируется): тогда придется выделять большой кусок памяти, который может и не понадобиться, но он будет занимать излишнюю память.