Задания к работе №3 по Фундаментальным алгоритмам.

1. Реализуйте функцию перевода числа из десятичной системы счисления в систему счисления с основанием $2^r, r = 1, \ldots, 5$. При реализации функции разрешается использовать битовые операции и операции обращения к памяти, запрещается использовать стандартные арифметические операции. Продемонстрируйте работу реализованной функции
2. Реализуйте библиотеку, предоставляющую пользователю функционал динамического массива (Vector) для произвольного типа элементов (используя макросы).

   Важно: в одной программе, где используется данная библиотека, может быть определён только один тип значений массива. Указатель на функцию копирования(поверхностного или глубокого) и удаления необходимо передавать в рамках функции create_vector и хранить в структуре Vector.

   Требуется реализовать следующие пункты

   Требуется реализовать следующие функции:

   a. Определение структуры Vector.
   b. create_vector — создание экземпляра типа Vector.
   c. erase_vector — удаление внутреннего содержимого экземпляра Vector.
   d. is_equal_vector — функция сравнения двух экземпляров Vector (лексикографический компаратор).
   e. copy_vector — копирование содержимого одного экземпляра Vector: в уже существующий экземпляр; в новый экземпляр, размещённый в динамической памяти.
   f. push_back_vector — добавление элемента в конец Vector.
   g. delete_at_vector — удаление элемента по индексу.
   h. get_at_vector — получение элемента по индексу.
   i. delete_vector — освобождение памяти, занимаемой экземпляром Vector.

```
typedef struct {
    VECTOR_TYPE *data;  // указатель на элементы     size_t
size;        // текущее количество элементов size_t capacity;
// вместимость (количество выделенных
элементов)
    VECTOR_TYPE (*CopyVoidPtr)(VECTOR_TYPE); void
    (*DeleteVoidPtr)(VECTOR_TYPE);
} Vector;
// Создание нового вектора
Vector create_vector(size_t initial_capacity, VECTOR_TYPE
(*CopyFunc)(VECTOR_TYPE), void (*DeleteFunc)(VECTOR_TYPE));
// Удаление внутреннего содержимого вектора (data, size=0, capacity=0)
void erase_vector(Vector *v);
// Сравнение двух векторов (лексикографически)
// возвращает 1 — равны, 0 — не равны
int is_equal_vector(const Vector *v1, const Vector *v2);
// Копирование содержимого одного вектора в другой (существующий)
void copy_vector(Vector *dest, const Vector *src);
// Создание нового вектора в динамической памяти и копирование
содержимого
Vector *copy_vector_new(const Vector *src); //
Добавление элемента в конец вектора
```

```
void push_back_vector(Vector *v, VECTOR_TYPE value);
// Удаление элемента по индексу
void delete_at_vector(Vector *v, size_t index);
// Получение элемента по индексу
VECTOR_TYPE get_at_vector(const Vector *v, size_t index);
// Освобождение памяти, занимаемой экземпляром вектора void
delete_vector(Vector *v);
```

3. Реализуйте библиотеку, предоставляющую пользователю функционал двусвязного списка (LinkedList) для типа Liver. Требуется реализовать следующие функции. На его основе решите эту задачу. В текстовом файле находится информация о жителях (тип структуры Liver) некоторого поселения: id жителя(неотрицательное значение, которое неизменно в течении работы программы), фамилия (непустая строка только из букв латинского алфавита), имя(непустая строка только из букв латинского алфавита), отчество (строка только из букв латинского алфавита; допускается пустая строка), дата рождения (в формате число, месяц, год), пол (символ ‘M’ - мужской символ ‘W’ - женский), средний доход за месяц (неотрицательное вещественное число). Напишите программу, которая считывает эту информацию из файла в односвязный упорядоченный список (в порядке увеличения возраста). Информация о каждом жителе должна храниться в объекте структуры Liver. Реализуйте возможности поиска жителя с заданными параметрами, изменение существующего жителя списка, удаления/ добавления информации о жителях и возможность выгрузки данных из списка в файл (путь к файлу запрашивайте у пользователя с консоли). Добавьте возможность отменить последние $N/2$ введённых модификаций, то есть аналог команды Undo; N - общее количество модификаций на текущий момент времени с момента чтения файла/последней отмены введённых модификаций
    a. Определение структур Node и LinkedList.
    b. create_list — создание пустого списка.
    c. erase_list — очистка содержимого списка (удаление всех элементов, но сохранение структуры).
    d. delete_list — удаление списка (освобождение всех ресурсов).
    e. push_back_list — добавление элемента в конец списка.
    f. push_front_list — добавление элемента в начало списка.
    g. pop_back_list — удаление элемента с конца списка и возврат его значения.
    h. pop_front_list — удаление элемента с начала списка и возврат его значения.
    i. insert_at_list — вставка элемента по индексу.
    j. delete_at_list — удаление элемента по индексу.
    k. get_at_list — получение элемента по индексу.
    l. is_equal_list — сравнение двух списков (лексикографически).
    m. Интерфейс стека (Stack):
        i. push_stack — поместить элемент на вершину стека (реализовать через push_back_list).
        ii. pop_stack — извлечь элемент с вершины стека (реализовать через pop_back_list).
        iii. peek_stack — получить элемент с вершины стека без удаления.

n. Интерфейс очереди (Queue)
    i. enqueue — добавить элемент в очередь (реализовать через push_back_list).
    ii. dequeue — извлечь элемент из очереди (реализовать через pop_front_list).
    iii. peek_queue — получить первый элемент очереди без удаления.

```
// Узел двусвязного списка
typedef struct Node {
double* data;
    struct Node *prev;
struct Node *next;
} Node;
// Двусвязный список
typedef struct {
Node *head;     Node
*tail;     size_t
size;
} LinkedList;
// ----------------- БАЗОВЫЕ ОПЕРАЦИИ ----------------
// Создание пустого списка
LinkedList create_list(void);
// Очистка содержимого списка (удаление всех элементов) void
erase_list(LinkedList *list);
// Полное удаление списка (освобождение ресурсов)
void delete_list(LinkedList *list); //
Добавление элемента в конец списка
void push_back_list(LinkedList *list, LIST_TYPE value);
// Добавление элемента в начало списка
void push_front_list(LinkedList *list, LIST_TYPE value);
// Удаление элемента с конца списка double
pop_back_list(LinkedList *list); //
Удаление элемента с начала списка double
pop_front_list(LinkedList *list);
// Вставка элемента по индексу
void insert_at_list(LinkedList *list, size_t index, LIST_TYPE value);
// Удаление элемента по индексу
void delete_at_list(LinkedList *list, size_t index);
// Получение элемента по индексу
double get_at_list(const LinkedList *list, size_t index);
// Сравнение двух списков (лексикографически) //
возвращает 1 — равны, 0 — не равны
int is_equal_list(const LinkedList *l1, const LinkedList *l2);
// ----------------- СТЕК ----------------- //
Поместить элемент на вершину стека
void push_stack(LinkedList *stack, LIST_TYPE value);
// Извлечь элемент с вершины стека double
pop_stack(LinkedList *stack);
// Получить элемент с вершины стека без удаления
double peek_stack(const LinkedList *stack); // -
---------------- ОЧЕРЕДЬ -----------------
// Добавить элемент в очередь
void enqueue(LinkedList *queue, LIST_TYPE value);
// Извлечь элемент из очереди double
dequeue(LinkedList *queue);
// Получить первый элемент очереди без удаления double
peek_queue(const LinkedList *queue);
```

4. Реализуйте библиотеку, предоставляющую пользователю функционал **бинарной кучи (Heap)** для типа. Реализуйте приложение, выполняющее моделирование почтового сервиса. Программа моделирует работу почтового сервиса, который включает взаимодействие между почтовыми отделениями и письмами. В каждом отделении может храниться ограниченное количество писем любого состояния. На вход программе вторым аргументом командной строки подаётся путь к файлу маппингов (связей) между почтовыми отделениями в виде пар: id первого отделения, id второго отделения.
5. Система должна поддерживать следующие команды:
    a. Добавление почтового отделения: Команда добавляет новое почтовое отделение с уникальным ID, максимальной вместимостью писем n, и списком id отделений, имеющих связь с новым отделением. В отделении может храниться не более n писем одновременно.
    b. Удаление почтового отделения: Команда удаляет почтовое отделение по его ID. Письма, не связанные с этим отделением, отправляются в другие отделения.

Письма, связанные с удаляемым отделением, помечаются как "Не доставлено" и удаляются из системы.

c. Добавление письма: Команда добавляет письмо с полями тип, приоритет, id отделения отправителя, id отделения получателя, а также строкой технических данных. Каждому письму присваивается уникальный ID, который генерируется с помощью static переменной, чтобы обеспечить последовательное увеличение значения ID.

d. Пометить письмо как недоставленное

e. Попытка взять письмо: Команда позволяет попытаться забрать письмо по его ID в месте назначения. Если письмо успешно доставлено, оно удаляется из системы.

f. Получение списка всех писем: Команда выводит список всех писем в том числе доставленных и недоставленных в указанный выходной файл.

g. Выход из программы.

Работа с письмами: Каждые 0.2 секунды самое приоритетное письмо в каждом отделении передаётся в одно из связанных с ним отделений, в направлении к месту назначения. Для хранения писем внутри отделений необходимо использовать структуру данных косую приоритетную очередь. Письма с более высоким приоритетом обрабатываются раньше. Письмо передаётся от одного отделения к другому, пока не достигнет пункта назначения или не будет помечено как недоставленное. Свойства писем: Каждое письмо имеет следующие свойства: ID (уникальный идентификатор, задаётся автоматически с помощью статической переменной), Тип (например, обычное, срочное), Состояние (доставлено, не доставлено, в процессе отправки) Приоритет (целое число, чем выше, тем важнее письмо), ID отделения отправителя, ID отделения получателя, Технические данные (произвольная строка). Если письмо не может быть доставлено в конечное отделение по причине недоступности или полной загруженности получателя, письмо отправляется в ближайшее доступное отделение, и система сохраняет такую "петлю" перенаправления до тех пор, пока письмо не станет возможным

доставить в исходное место назначения. Выходные данные: реализуйте пользовательский интерфейс в командной строке «для домохозяек», по взаимодействию с вашей программой, а также обеспечьте логирование в файл передвижение всех писем и изменения состояний писем и отделений. (название генерируется либо псевдослучайно, либо передаётся третьим аргументом командной строки).

Требуется реализовать следующие функции:

h. Определение структуры Heap.
i. create_heap — создание кучи с заданной начальной вместимостью.
j. delete_heap — удаление кучи и освобождение памяти.
k. is_empty_heap — проверка, пуста ли куча.
l. size_heap — возвращает текущее количество элементов в куче.
m. peek_heap — получение элемента с наивысшим приоритетом (минимального), без его удаления.
n. push_heap — добавление элемента в кучу.
o. pop_heap — удаление элемента с наивысшим приоритетом (минимального) из кучи и возврат его значения.
p. build_heap — построение кучи из массива.
q. is_equal_heap — сравнение двух куч (лексикографически по массиву внутреннего представления). (Пример определения бинарной кучи)

```
// Бинарная куча (минимальная по умолчанию) typedef
struct {
    int *data;       // массив элементов     size_t
size;     // текущее количество элементов
size_t capacity; // вместимость
} Heap;
// ----------------- БАЗОВЫЕ ОПЕРАЦИИ -----------------
// Создание кучи с заданной начальной вместимостью
Heap create_heap(size_t initial_capacity);
// Удаление кучи и освобождение памяти
void delete_heap(Heap *h);
// Проверка, пуста ли куча (1 — пуста, 0 — нет)
int is_empty_heap(const Heap *h); // Возврат
текущего количества элементов size_t
size_heap(const Heap *h);
// Получение элемента с наивысшим приоритетом (минимального) без
удаления
int peek_heap(const Heap *h); //
Добавление элемента в кучу void
push_heap(Heap *h, int value);
// Удаление элемента с наивысшим приоритетом (минимального) и
возврат его
int pop_heap(Heap *h); //
Построение кучи из массива
Heap build_heap(const int *array, size_t n);
// Сравнение двух куч (лексикографически по массиву внутреннего
представления)
// возвращает 1 — равны, 0 — не равны
int is_equal_heap(const Heap *h1, const Heap *h2);
```

6. Напишите программу на языке C, которая проверяет правильность расстановки скобок в строке. На вход подаётся строка, содержащая любые символы (буквы,

цифры, пробелы, знаки препинания и т.д.). Проверке подлежат только скобки:круглые ( ), квадратные [ ], фигурные { }, угловые < >. Программа должна определить, является ли строка корректно сбалансированной по скобкам: Каждая открывающая скобка должна иметь соответствующую закрывающую. int check_brackets(const char *str)

7. Разработать консольное приложение на языке C, реализующее простой интерпретатор для математических выражений с поддержкой переменных, арифметических операций и функции вывода.
    a. **Переменные**
        i. Имена переменных — одна заглавная буква латинского алфавита (A–Z). ii. Значения переменных — целые числа.
    b. **Операции**
        i. Присваивание: A = 5
        ii. Арифметические операции: +, -, *, /
        iii. Быстрое возведение в степень: ^
        iv. Функция вывода: print(X)
        i. **Синтаксис выражений**
            i. Каждая команда записывается с новой строки.
            ii. Допускаются пробелы.
            iii. Результат выполнения print выводится в стандартный вывод.
        ii. **Логирование (трассировка)**
        i. После обработки каждой строки создаётся запись в файл трассировки.
        ii. В лог выводится:
            1. исходная команда,
            2. значение всех инициализированных переменных (A–Z), 3. описание выполненной операции.

Пример входного файла:

```
A = 2
B = 3
C = A + B * 2 D = C ^ B print(D)
```

Пример файла трассировки:

```
[1] A = 2         | A=2 | Assignment
[2] B = 3         | A=2, B=3 | Assignment
[3] C = A + B*2   | A=2, B=3, C=8 | Arithmetic operation
[4] D = C ^ B     | A=2, B=3, C=8, D=512 | Arithmetic operation
[5] print(D)      | A=2, B=3, C=8, D=512 | Print D
```