Содержание

Введение

Данное приложение, реализованное на языке программирования C++, позволяет выполнять операции над коллекциями данных заданного типа (данные о доставке) и контекстами их хранения (коллекциями данных).

Коллекция данных описывается набором строковых параметров (набор параметров однозначно идентифицирует коллекцию данных):

- Название пула схем данных, хранящего схемы данных;
- Название схемы данных, хранящей коллекции данных;
- Название коллекции данных.

Коллекция данных представляет собой ассоциативный контейнер (конкретная реализация определяется вариантом), в котором каждый объект данных соответствует некоторому уникальному ключу. Взаимодействие с коллекцией объектов происходит посредством выполнения одной из операций над ней:

- Добавление новой записи по ключу;
- Чтение записи по её ключу;
- Чтение набора записей с ключами из диапазона $[minbound... maxbound]$;
- Обновление данных для записи по ключу;
- Удаление существующей записи по ключу.

Во время работы приложения возможно выполнение также следующих операций:

- Добавление/удаление пулов данных;
- Добавление/удаление схем данных для заданного пула данных;
- Добавление/удаление коллекций данных для заданной схемы данных заданного пула данных.

Поток команд, выполняемых в рамках работы приложения, поступает из консоли. Также реализована возможность выполнения набора команд, подаваемых в качестве текстового файла.

Данные игрока MMO содержат следующие поля:

- **Id пользователя**
- Никнейм

- Игровая зона
- Статус (обычный игрок/премиум игрок/модератор чата/администратор)
- Значение внутриигровой валюты
- Значение внутриигровой премиальной валюты
- Количество очков опыта
- Дата регистрации
- Время, проведённое в игре (в минутах)

Полужирным шрифтом выделены поля, формирующие уникальный ключ объекта данных

1. Логирование

1.1. Порождающий паттерн проектирования Builder

Необходимость использования паттерна Builder в программе возникает в случаях, когда нужно создать несколько объектов с разными возможностями без существенного изменения кода путём поэтапного создания продукта.

Поэтапное создание - конструирование объекта по частям, а после вызов необходимого метода для создания конечного продукта.

Класс logger_builder определяет интерфейс для построения отдельных частей продукта, а соответствующие подклассы logger_builder_concrete реализуют его подходящим образом.

Логгер конфигурируется тремя командами класса logger_builder:

Add_stream - принимает на вход название файлового потока вывода и уровень логирования

Construct - отдает указатель на настроенный логгер на внешний уровень

Conduct - принимает на вход файл Json, содержащий данные в формате: “файловый поток”: “severity”, после выполняет обработку этого файла с помощью сторонней библиотеки nlohmann/json и вызывает Construct

Базовый класс logger_builder:

```
class logger_builder
{
public:
    virtual logger_builder *add_stream(std::string const &, logger::severity) = 0;
    virtual logger *conduct(std::string const &) = 0;
    virtual logger *construct() const = 0;
public:
    virtual ~logger_builder() = 0;
};
```

Производный класс logger_builder_concrete:

```
class logger_builder_concrete final : public logger_builder
{
private:
    std::map<std::string, logger::severity> _construction_info;
public:
    logger_builder *add_stream(std::string const &, logger::severity) override;
    logger *construct() const override;
    logger *conduct(std::string const &) override;

};
```

1.2. Реализация логгера

Базовый класс logger содержит метод log, а также уровни логирования, представленные перечислением.

Производный класс logger_concrete содержит локальную коллекцию потоков и реализацию функции log

Базовый класс logger:

```
class logger
{
public:
    enum class severity
    {
        trace,
        debug,
        information,
        warning,
        error,
        critical
    };
public:
    virtual ~logger();
public:
    virtual logger const *log(const std::string &, severity) const = 0;
};
```

Производный класс logger_concrete:

```
class logger_concrete final : public logger
{
    friend class logger_builder_concrete;
private:
    std::map<std::string, std::pair<std::ofstream *, logger::severity> > _logger_streams;
private:
    static std::map<std::string, std::pair<std::ofstream *, size_t> > _streams;
private:
    explicit logger_concrete(std::map<std::string, logger::severity> const &);
public:
    logger_concrete(logger_concrete const &) = delete;
    logger_concrete &operator=(logger_concrete const &) = delete;
    ~logger_concrete();
public:
    logger const *log(const std::string &, severity) const override;
};
```

1.3. Реализация удобного хранения - logger_holder

Класс logger_holder не являлся необходимым для решения поставленной задачи, однако он представляет собой удобную обёртку для класса logger, чтобы им можно было проще управлять. Он содержит методы для логирования на всех представленных в классе logger уровнях, причем в реализации этих методов присутствует проверка на пустоту логгера, что позволяет избежать постоянных проверок при логировании.

Класс logger_holder:

```
class logger_holder

{

public:

    virtual ~logger_holder() noexcept = default;

public:

    logger_holder const *log_with_guard(

        std::string const &message,

        logger::severity severity) const;

    logger_holder const *trace_with_guard(

        std::string const &message) const;

    logger_holder const *debug_with_guard(

        std::string const &message) const;

    logger_holder const *information_with_guard(

        std::string const &message) const;

    logger_holder const *warning_with_guard(

        std::string const &message) const;

    logger_holder const *error_with_guard(

        std::string const &message) const;

    logger_holder const *critical_with_guard(

        std::string const &message) const;

protected:

    virtual logger *get_logger() const noexcept = 0;


};
```

2. Распределение памяти
2.1. Аллокатор на глобальной куче

В качестве аллокатора выступает класс memory, который является более удобной надстройкой над оператором new, использующим память из глобальной кучи.

Всего представлены два класса - базовый memory, являющийся абстрактным классом, и производный - memory_concrete, реализующий методы своего родителя.

Класс memory позволяет пользователю выделять и освобождать память, а также отслеживать её выделение и освобождение путем передачи в конструктор класса указателя на класс logger.

Базовый класс memory:

```
class memory{
    public:
        enum class method{
            first,
            best,
            worst
        };
        virtual ~memory() = default;
        virtual void * allocate(size_t target_size) = 0;
        virtual void deallocate(void const * const target_to_dealloc) const = 0;
        memory() = default;

        memory(memory const&) = delete;

        memory& operator=(memory const&) = delete;

        void* operator+=(size_t const&);

        void operator-=(void const * const object);

        virtual void set_logger(logger* &lg) noexcept = 0;
    private:
```

```
            virtual void _log_with_guard(const std::string& str, logger::severity
level) const = 0
      };
```

Класс потомок memory_concrete:

```
class memory_concrete: public memory{
private:
    logger* loggerr;
    void _log_with_guard(const std::string& str, logger::severity level) const override;
public:
    ~memory_concrete();
    memory_concrete(logger* = nullptr);

    memory_concrete(memory_concrete const&) = delete;

    memory_concrete& operator=(memory_concrete const&) = delete;

    void * allocate(size_t target_size) override;

    void set_logger(logger* &lg) noexcept override;

    void deallocate(void const * const target_to_dealloc) const override;
    template<typename T>
    std::string to_str(T const &object) const noexcept;
};
```

2.2. Вспомогательный класс memory_holder

Класс memory_holder, так же, как и logger_holder - представляет собой обёртку класса memory для более простого оформления кода и для снижения количества проверок в коде.

Класс memory_holder:

```
class memory_holder

{

public:

    virtual ~memory_holder() noexcept = default;

public:

    void *allocate_with_guard(

        size_t size) const;

    void deallocate_with_guard(

        void *block_pointer) const;

protected:

    virtual memory *get_memory() const noexcept = 0;


};
```

3. B-дерево

B-дерево - сильноветвящееся сбалансированное дерево поиска, позволяющее проводить поиск, добавление и удаление элементов за O(log(n)).

B-дерево с n узлами имеет высоту O(log(n)).

Количество детей узлов может быть от нескольких до тысяч (обычно степень ветвления B-дерева определяется характеристиками устройства (дисков), на котором производится работа с деревом). B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время O(log(n)).

3.1. Свойства дерева и структура узла

B-дерево обладает следующими свойствами:

- При создании дерева определяется его степень t, такая что t > 1
- Исходя из степени определяется максимальное и минимальное количество ключей в узлах (кроме корня) - 2*t – 1 и t – 1 соответственно. Для узлов максимальное количество указателей на поддеревья составляет максимально 2*t и минимально t, однако это правило на лист не распространяется.

Узлы дерева хранят следующую информацию:

- Массив arr ключей, такой что max(size(arr)) = 2*t – 1 и min(size(arr)) = t - 1. Все ключи в узле упорядочены слева - направо.
- Массив sub указателей на поддеревья, такой что size(sub) = 2*t. Кроме того индекс поддерева относительно ключа определяет, что в поддерево, имеющее индекс i в массиве sub, совпадающий с индексом текущего ключа, будут помещены ключи строго меньшие его, а в поддереве с индексом i + 1 будет помещены ключи строго большие текущего.
- Количество size текущих ключей в узле. Будем говорить, что узел является полным, если он имеет ровно 2*t – 1 ключ
- Булевое значение leaf, определяющее является ли узел листом - т.е. не имеет поддеревьев (необязательный параметр). Все листья расположены на одной и той же глубине, равной глубине дерева.

Простейшее B-дерево получается при t = 2. При этом каждый внутренний узел может иметь 2, 3 или 4 дочерних узла, и мы получаем так называемое 2-3-4-дерево, однако обычно на практике используют гораздо большие значения t.

Высота Б-дерева в худшем случае равна log(n), где n - количество ключей.

3.2. Основные операции с B-деревьями

Поиск:

Поиск в Б-дереве во многом схож с поиском в бинарном дереве поиска, но с тем отличием, что в бинарном дереве поиска предстояло выбрать один из двух путей, здесь же предстоит выбрать из большего числа количества альтернатив, если точнее, то в каждом узле нам предстоит выбрать одну из ветвей корня.

Начиная с корня поиск просматривает ключи в узле и сравнивает их с искомым значением, в случае, когда был найден узел больший искомого - необходимо переместиться в левое поддерево и продолжить поиск там. Вычислительная сложность составляет O(t*h)

Создание пустого дерева:

Создание пустого дерева достаточно простая операция. Необходимо выделить память на узел, на массив ключей arr, массив указателей на поддеревья sub, а также установить size = 0 и leaf = true (если есть).

Вставка ключа в B-дерево:

Вставка ключа в B-дерево существенно сложнее вставки в бинарное дерево поиска. Для начала, как и в бинарном дереве, необходимо найти позицию для вставки, куда будет помещён новый ключ. Главное различие с бинарным деревом поиска - в б-дереве не надо при вставке просто создавать новый узел и вставлять туда значение, иначе это нарушит баланс. Вместо этого вставка будет происходить в уже существующий лист. Поскольку, руководствуясь правилами, вставка в заполненный лист невозможна, необходимо ввести новую операцию - разбиение заполненного узла Y на два, каждый из которых содержит по t – 1 ключей. При этом медиана, или же средний ключ, перемещается в родительский по отношению к у узел, где становится точкой - разделителем для двух новых узлов. Однако в случае, если узел - родитель тоже заполнен, то и его следует разбить. Такой процесс может продолжаться до самого корня. При поиске позиции для вставки ключа мы будет разбивать каждый узел, который окажется полным, таким образом гарантируется, что в случае разбиения какого-либо узла его родитель не будет заполнен.

Разбиение узла:

Процедура разбиения получает в качестве входного параметра незаполненный узел X и индекс i, такой что X.sub[i] - заполненный дочерний узел X. Функция разбивает дочерний узел на два и соответствующим образом обновляет поля X, внося в него информацию о новом узле. В случае, если необходимо разбить корень, первым делом стоит сделать корень дерева дочерним узлом нового узла, после чего вызываем функцию разбиения.

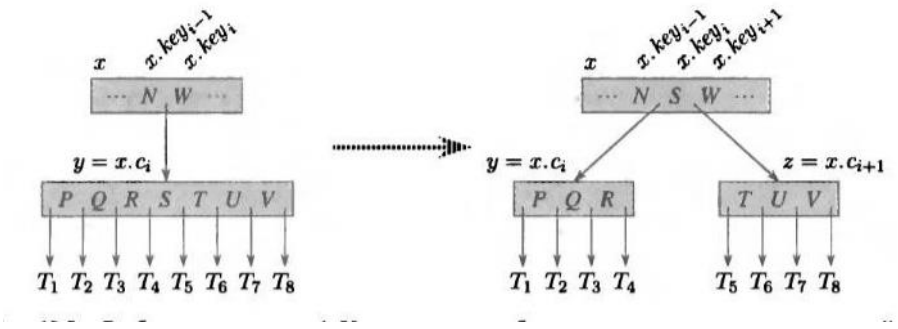

Рисунок 1. Разбиение узла

Удаление узла:

Удаление узла представляет собой более сложную задачу, чем вставка. Сложность связана с тем, что ключ может быть не только в листе, но и во внутреннем узле непосредственно, что может потребовать дополнительной перестройки дочерних узлов. Как и со вставкой, во время удаления необходимо соблюдать свойства Б-дерева. Если в операции вставки необходимо было следить, чтобы узел не переполнился, то в операции удаления необходимо отслеживать что количество ключей в узле не меньше минимального количества (за исключением корня, там может храниться и 1 ключ), то есть t – 1.

Для проведения операции удаления необходимо рассмотреть определенные случаи, которые могут произойти во время выполнения функции:

1) Если ключ K находится в узле X и X является листом, то просто удаляем ключ K из X.
2) Если ключ K находится в узле X, а X - внутренний узел, то необходимо провести следующие операции:
    a) Если дочерний по отношению к X узел Y, предшествующий ключу K в узле X, содержит по меньшей мере t ключей, то необходимо найти предшественника K – K', в поддереве, корнем которого будет Y. После необходимо заменить K на K'.
    b) Если Y имеет менее t ключей, то симметрично обращаемся к дочернему по отношению к X узлу Z, который следует за ключом K в узле X. Если Z содержит не менее t ключей, то находим K' - следующий за K ключ в поддереве, корнем которого является Z. Заменяем K в X ключом K'.
    c) Иначе, если и Y, и Z содержат по t – 1 ключей, вносим ключ K и все ключи Z в Y. Удаляем из X ключ K и указатель на Z. Освобождаем Z и удаляем K из Y.
3) Если ключ K отсутствует во внутреннем узле X, тогда находим корень X.sub[i] такой, что в нем есть K. В случае, когда найденный узел X.sub[i] содержит только t – 1 ключей, то необходимо выполнить п. 3(a) или п. 3(b), чтобы гарантировать, что мы переходим в узел как минимум с t ключами. После удаляем K из X.sub[i].
    a) Если X.sub[i] имеет только t – 1 ключей, но при этом один из его непосредственных соседей (дочерний по отношению к X узел,

отделенный от рассматриваемого ровно одним ключом разделителем) содержит как минимум t ключей, то необходимо передать в X.sub[i] ключ - разделитель между данным узлом и его непосредственным соседом из X, на его место поместим ключ из соседнего узла и перенесем соответствующий указатель из соседнего узла в X.sub[i].

b) Если и X.sub[i] и оба его непосредственных соседа содержат по t – 1 ключей, то необходимо объединить X.sub[i] с одним из его соседей, при этом ключ - разделитель этих узлов переносится в слитый узел и становится его медианой.

Учитывая то, что большинство ключей Б-дерева хранятся в листьях, можно ожидать, что чаще всего удаление будет производиться именно из них.

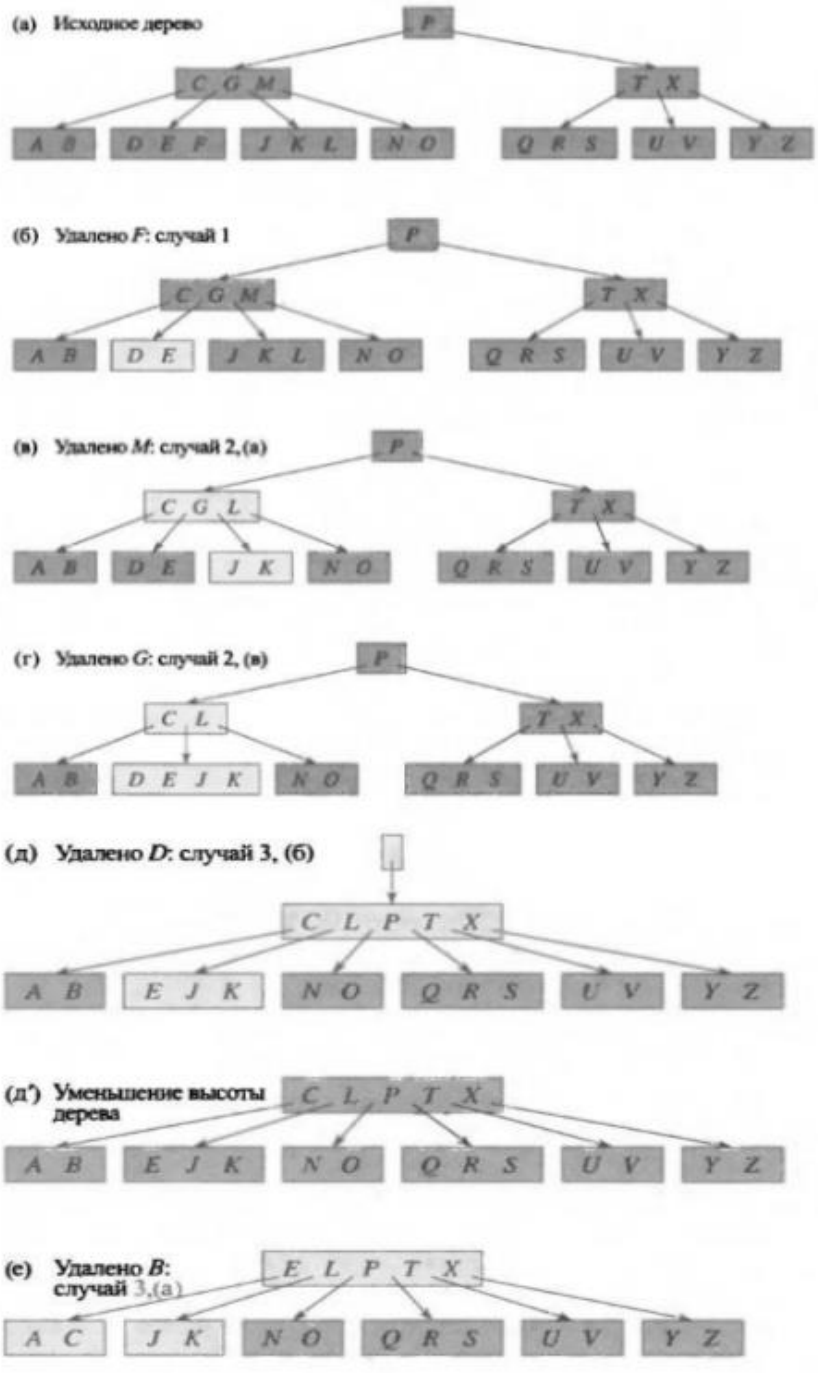

Рисунок 2. Случаи удаления из Б-дерева

4. Разработка и реализация приложения, управляющего хранилищем

4.1. Основной функционал приложения

Хранилище данных состоит из:

- Набора пулов, содержащих схемы данных
- Схем данных, содержащих коллекции данных
- Коллекции данных, содержащие объекты

Каждый пул, схема данных и коллекция имеет имя и представляют из себя ассоциативный контейнер. Пул, схема данных и коллекция являются B-деревом. С каждой коллекцией, пулом и схемой данных можно совершать определенные во введении операции.

Пример работы программы для набора данных:

```
CREATE_POOL poll_1
CREATE_SCHEMA schem_1 poll_1
CREATE_COLLECTION coll_1 schem_1 poll_1
INSERT_DATA coll_1 schem_1 poll_1 123 astra prem 111 12 12/08/2002 1234
SET_DATA coll_1 schem_1 poll_1 123 mega admin 112 23 12/08/2002 2000
GET_DATA coll_1 schem_1 poll_1 123
DELETE_DATA coll_1 schem_1 poll_1 123
DELETE_COLLECTION coll_1 schem_1 poll_1
DELETE_SCHEMA schem_1 poll_1
DELETE_POOL poll_1
```

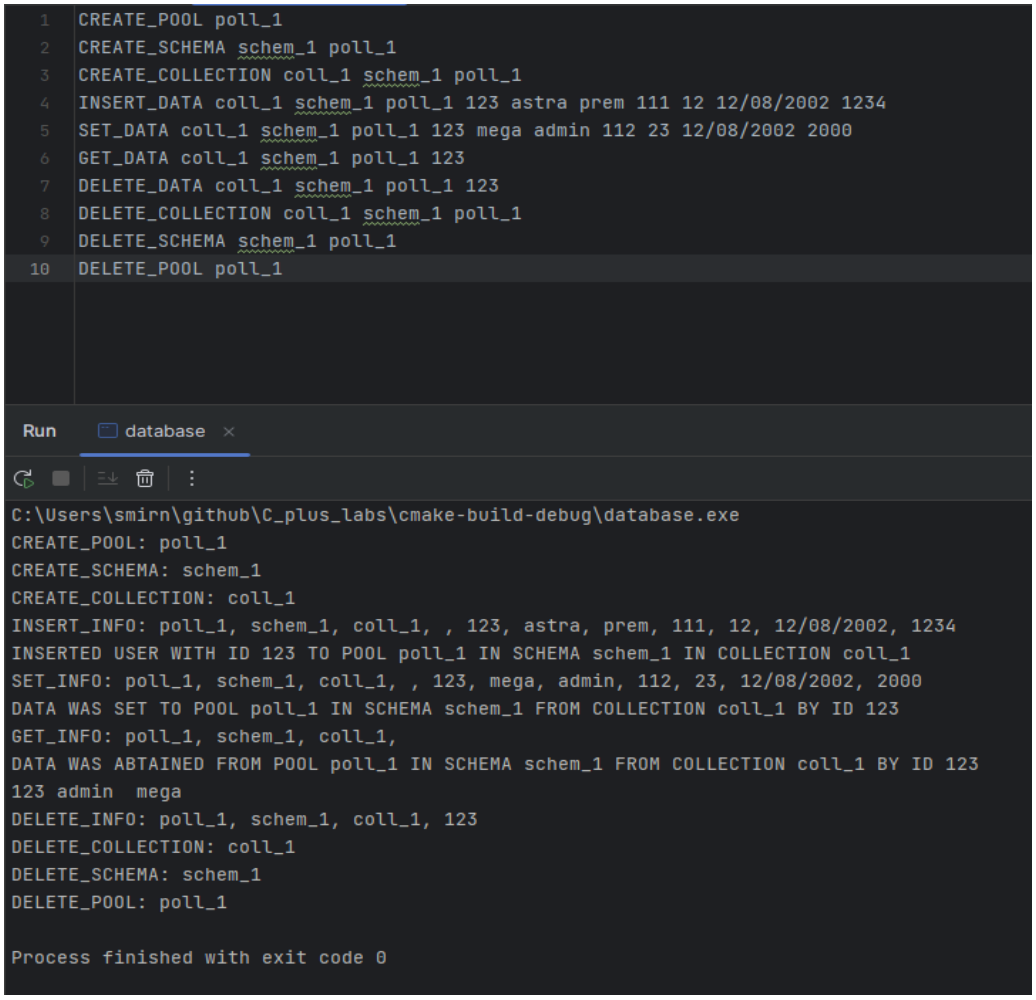

Рисунок 4. Пример работы программы по имитации базы данных

5. Руководство пользователя

В папке с проектом есть файл reference.txt, описывающий синтаксис команд в таком формате:

```
CREATE_PULL pull_name

CREATE_SCHEMA pull_name schema

CREATE_COLLECTION pull_name schema collection

INSERT_DATA pul_name schema_name collection_name id nickname play_zone status steel gold exp reg_date minutes_in

SET_DATA pul_name schema_name collection_name id(as key) nickname play_zone status steel gold exp reg_date minutes_in

GET_DATA pul_name schema_name collection_name id(as key)

GET_FROM_TO pul_name schema_name collection_name id(start) id(end)

DELETE_PULL pull_name

DELETE_SCHEMA pull_name schema
```

```
DELETE_COLLECTION pull_name schema collection
DELETE_DATA pull_name schema collection id
```

6. Вывод

В итоге сделано приложение, позволяющее имитировать работу базы данных, имеющее в качестве хранилища пуллов, пуллов, схем и коллекций B-дерево. Также реализованы аллокатор и логгер, помогающие выделять память и отслеживать изменения, протекающие в процессе выполнения работы соответственно.

7. Список использованных источников

1. Кормен, Лейзерсон, Ривест, Штайн - Алгоритмы построение и анализ.
2. Кнут - Искусство программирования, т. 3

8. Приложение

Весь код вместе с примерами файлов и настроенным CMakeLists.txt можно найти в репозитории: https://github.com/Bybulda/b-tree_db_c-/