МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный исследовательский университет)

Институт №8

«Компьютерные науки и прикладная математика»

Кафедра 806

«Вычислительная математика и программирование»

Курсовой проект по дисциплине «Фундаментальные алгоритмы»

Тема: «Разработка алгоритмов системы хранения и управления данными на основе динамических структур данных»

Студент: Барабанов Клим Александрович

Группа: М8О-211Б-21

Преподаватель: Ирбитский И.С.

Оценка:___

Дата:_____

Москва, 2023

Содержание

Введение

Разработанное на языке программирования C++ приложение предоставляет возможность выполнения операций над коллекциями данных заданных типов (данные о встречах в рабочем календаре) 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. Для ассоциативного контейнера необходимо вынести интерфейсную часть (в виде абстрактного класса C++) и реализовать этот интерфейс. 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.заданного пула данных.

Поток команд, выполняемых в рамках работы приложения, поступает из файла или консоли.

Дополнительные задания, которые были сделаны:

•Возможность кастомизации (при создании) реализаций ассоциативных

контейнеров, репрезентирующих коллекции данных: АВЛ-дерево, красно-чёрное дерево, косое дерево.

•Механизм, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени.

•Возможность кастомизации (для заданного пула схем) аллокаторов для размещения объектов данных : первый + лучший + худший подходящий + освобождение в рассортированном списке, первый + лучший + худший подходящий + освобождение с дескрипторами границ, система двойников.

•Конфигурация логгера JSON файлом и соответственно логирование приложения в файловые потоки ввода

• Интерактивный диалог с пользователем.

•Механизм, позволяющий выполнять эффективный поиск по различным отношениям порядка на пространстве данных.

Данные о встречах в рабочем календаре содержат следующие данные (id встречи, вид встречи (ежедневная встреча/встреча по результатам отчётного периода/собеседование/корпоратив), формат проведения (очный/дистанционный), описание встречи, ссылка на встречу в дистанционном формате, ФИО создателя встречи (раздельные поля), дата встречи, время начала встречи, продолжительность встречи (в минутах), список приглашённых на встречу (в виде строки)). Id встречи является уникальным ключом.

1. Логирование

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

Базовый класс logger_builder:

```
class logger_builder
{
class logger_builder
{

public:

    virtual logger_builder *add_stream(std::string const &, logger::severity) = 0;

    virtual logger *construct() const = 0;
    virtual logger_builder *build(std::string const &way) = 0;
public:

    virtual ~logger_builder();
```

```
};
```

Производный класс logger_builder_concrete:

```
public:

   virtual logger_builder *add_stream(std::string const &, logger::severity) = 0;

   virtual logger *construct() const = 0;
   virtual logger_builder *build(std::string const &way) = 0;
public:

   virtual ~logger_builder();

};
```

1.1. 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:

   logger_concrete(std::map<std::string, logger::severity> const &);


public:

   logger_concrete(std::vector<std::pair<std::string, logger::severity>> const &streams);

   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;
};
```

1.2 logger_guard

Также чтобы каждый раз не проверять на пустоту логгер, была реализована обёртка для класса logger.

```
class logger_guard
{
public:
   logger* logger_with_guard(const std::string &target, logger::severity level) const;
   virtual ~logger_guard() = default;
   virtual logger *get_logger() const = 0;
}
```

1.3 Конфигурация логгера через json

Для того, чтобы конфигурировать логгер из файла была найдена и использована библиотека - nlohman.

Рис. 1 Структура файла config.json

Логгер создается и конфигурируется во время создания первого парсера, обработки команд:

```
parser::parser() : _next_parser(nullptr)
{
    json_builder *builder = new json_builder_concrete();
    _logger = builder->build("json/config.json");
}
```

2. Бинарное дерево поиска

Бинарное дерево - это иерархическая структура данных, состоящая из узлов, в которой каждый узел может иметь не более двух дочерних узлов, обычно называемых левым и правым дочерними узлами. Узлы бинарного дерева организованы таким образом, что каждый узел имеет вышестоящий узел, называемый родительским узлом, за исключением корневого узла, который является верхним уровнем и не имеет родительского узла.

Бинарное дерево поиска - это особый тип бинарного дерева, где каждый узел содержит значение (ключ) и удовлетворяет следующему свойству: для любого узла X все значения в левом поддереве меньше значения узла X, а все значения в правом поддереве больше значения узла X. Это свойство упорядочивает значения в дереве и позволяет эффективно выполнять операции поиска, вставки и удаления элементов, что делает бинарные деревья поиска важной структурой данных для организации и управления упорядоченной информацией.

Структура узла бинарного дерева поиска:

```
struct node
{
    tkey key;
    tvalue value;
    node *left_subtree;
    node *right_subtree;
```

```
    bool is_left_child;
    bool is_right_child;
};
```

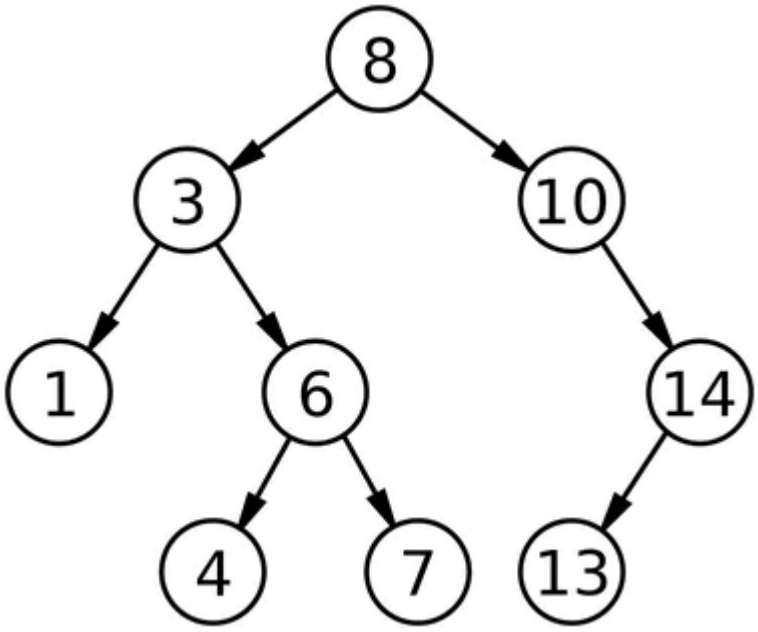

Рис.2. Пример бинарного дерева поиска

2.1 Сбалансированные деревья поиска

Сбалансированные деревья поиска - это специальный вид бинарных деревьев поиска, в которых происходит стремление к равномерному распределению элементов обеспечивается оптимальная высота дерева. Они поддерживают принцип равновесия между левыми и правыми поддеревьями каждого узла, что позволяет минимизировать глубину дерева и, следовательно, обеспечивает эффективное время выполнения операций поиска, вставки и удаления за O(logn).

Сбалансированные деревья поиска становятся необходимыми, когда необходимо гарантировать высокую производительность операций поиска и изменения даже в худших случаях. В отличие от несбалансированных деревьев, где плохо подобранная последовательность операций может привести к вырождению дерева в одноцветную цепочку, сбалансированные деревья стремятся поддерживать более устойчивую структуру.

Деревья, которые реализованы в данной курсовой работе, являются немного измененными бинарными деревьями поиска, поэтому в классе binary_search_tree реализованы шаблонные методы вставки, удаления, поиска элемента. Однако важно понимать, что все эти операции делятся на два этапа, потому что на их основе будут

реализовываться сбалансированные деревья поиска. Этапы:

1) Операция.

2) Последующие действия, необходимые после выполнения операции, которые будут отличаться для разных деревьев.

Шаблонные методы, которые были реализованы в binary_search_tree:

```
class insertion_template_method : private memory_guard, private logger_guard
{
private:
   binary_search_tree<tkey, tvalue, tkey_comparer> *_tree;

private:
   memory *get_memory() const override;
   logger *get_logger() const override;
   virtual size_t get_node_size() const;

public:
   explicit insertion_template_method(binary_search_tree<tkey, tvalue, tkey_comparer> *tree);
   void insert(tkey const &key,tvalue value);

private:
   void insert_inner(tkey const &key,
tvalue value,
node *&subtree_root_address,std::stack<binary_search_tree::node**>
&path_to_subtree_root_exclusive);

protected:
   virtual void after_insert_inner(tkey const &key,
node *&subtree_root_address,
std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);
};


class removing_template_method : private memory_guard, private logger_guard
{
private:
   binary_search_tree<tkey, tvalue, tkey_comparer> *_tree;

private:
   memory *get_memory() const override;
   logger *get_logger() const override;
   virtual void get_info_deleted_node(
           node *deleted_node,
           std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);

public:
   explicit removing_template_method(binary_search_tree<tkey, tvalue, tkey_comparer> *tree);
   tvalue remove(tkey const &key, node *&tree_root_address);
```

```
private:
   virtual tvalue remove_inner(tkey const &key,
            node *&subtree_root_address,
            std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);

protected:
   virtual void after_remove_inner(tkey const &key,
            node *&subtree_root_address,
            std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);
};

class find_template_method : private logger_guard
{
private:
   binary_search_tree<tkey, tvalue, tkey_comparer> *_tree;

public:
   bool find(typename associative_container<tkey, tvalue>::key_value_pair
*target_key_and_result_value, node *&subtree_address);
   explicit find_template_method(binary_search_tree<tkey, tvalue, tkey_comparer> *tree);

private:
   bool find_inner(typename associative_container<tkey, tvalue>::key_value_pair
*target_key_and_result_value,
            node *&subtree_address,
            std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);

protected:
   virtual void after_find_inner(typename associative_container<tkey, tvalue>::key_value_pair
*target_key_and_result_value,
            node *&subtree_address,
            std::stack<binary_search_tree::node**> &path_to_subtree_root_exclusive);
};
```

2.2 АВЛ дерево

АВЛ-дерево (Adelson-Velsky and Landis tree) - это сбалансированное двоичное дерево поиска, в котором для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Это свойство балансировки позволяет гарантировать логарифмическую сложность операций вставки, удаления и поиска элементов в дереве, что делает его эффективной структурой данных для работы с большими объемами информации. В АВЛ-дереве после каждой операции вставки или удаления происходит перебалансировка, чтобы удовлетворить условие баланса. Чтобы проверять высоту поддеревьев, в узле требуется хранить ещё и высоту:

```
struct avl_node : binary_search_tree<tkey, tvalue, tkey_comparer>::node
{
   size_t node_height = 0;
```

```
};
```

Чтобы получить высоту дерева, был реализован метод get_height:

```
int get_height(typename binary_search_tree<tkey, tvalue, tkey_comparer>::node *node)
{
   if (node == nullptr) {
       return 0;
   } else {
       return reinterpret_cast<avl_node*>(node)->node_height;
   }
}
```

Для вычисления баланс фактор, был реализован метод get_balance_factor, использующий метод get_height:

```
int get_balance_factor(typename binary_search_tree<tkey, tvalue, tkey_comparer>::node
*subtree_address) {
   return get_height(subtree_address->left_subtree) -
get_height(subtree_address->right_subtree);
}
```

Для обновления высоты узла в дереве был реализован метод update_height:

```
void update_height(typename binary_search_tree<tkey, tvalue, tkey_comparer>::node
**subtree_address)
{
   if (subtree_address != nullptr && *subtree_address != nullptr)
   {
       int height_left = 0;
       int height_right = 0;

       if (*subtree_address != nullptr) {
           height_left = get_height((*subtree_address)->left_subtree);
       }

       if (*subtree_address != nullptr) {
           height_right = get_height((*subtree_address)->right_subtree);
       }
       reinterpret_cast<avl_node*>(*subtree_address)->node_height = std::max(height_left,
height_right) + 1;
   }
}
```

Вставка и удаление в дерево происходит аналогично бинарному дерево, однако потребовалось доопределить функции after_insert_inner и after_remove_inner, теперь они вызывают метод балансировки дерева. Поиск изменять не потребовалось.

2.3 Косое дерево

Косое дерево, также известное как "Splay Tree", представляет собой сбалансированное двоичное дерево поиска, которое обладает уникальным свойством - оно реорганизует свою структуру в ответ на операции поиска, вставки и удаления.

Это дерево стремится поднимать недавно использованные или модифицированные узлы ближе к корню, что делает их более доступными для последующих операций и, в результате, повышает общую эффективность работы.

Косое дерево преследует цель минимизировать время доступа к часто используемым узлам путем проведения операции splay - это процесс перемещения выбранного узла в корень дерева. Операция splay реорганизует путь от корня до целевого узла, поднимая его ближе к корню, что в итоге приводит к улучшению времени доступа к этому узлу в будущем.

Реализация метода splay:

```
void splay(typename binary_search_tree<tkey, tvalue, tkey_comparer>::node *&subtree_address,
       std::stack<typename binary_search_tree<tkey, tvalue, tkey_comparer>::node**>
&path_to_subtree_root_exclusive)
{
   typename binary_search_tree<tkey, tvalue, tkey_comparer>::node *curr = subtree_address;
   typename binary_search_tree<tkey, tvalue, tkey_comparer>::node **parent = nullptr;
   typename binary_search_tree<tkey, tvalue, tkey_comparer>::node **grandfather = nullptr;
   typename binary_search_tree<tkey, tvalue, tkey_comparer>::node **parent_of_grandfather =
nullptr;

   while(!path_to_subtree_root_exclusive.empty())
   {
       if (!path_to_subtree_root_exclusive.empty())
       {
           parent = path_to_subtree_root_exclusive.top();
           path_to_subtree_root_exclusive.pop();

           if (!path_to_subtree_root_exclusive.empty())
           {
               grandfather = path_to_subtree_root_exclusive.top();
               path_to_subtree_root_exclusive.pop();

               if (!path_to_subtree_root_exclusive.empty())
               {
                   parent_of_grandfather = path_to_subtree_root_exclusive.top();
                   path_to_subtree_root_exclusive.pop();
               }
           }
       }

       if((*parent)->left_subtree == curr)
       {
           if (grandfather == nullptr)
           {
               this->right_rotate(parent, grandfather);
               curr = *parent;
           }
```

```
            else if(*parent == (*grandfather)->left_subtree)
            {
                this->right_rotate(grandfather, parent_of_grandfather);
                this->right_rotate(grandfather, parent_of_grandfather);
                curr = *grandfather;
            }
            else
            {
                this->right_rotate(parent, grandfather);
                this->left_rotate(grandfather, parent_of_grandfather);
                curr = *grandfather;
            }
        }
        else
        {
            if (grandfather == nullptr)
            {
                this->left_rotate(parent, grandfather);
                curr = *parent;
            }
            else
            {
                if ((*grandfather)->left_subtree == *parent)
                {
                    this->left_rotate(parent, grandfather);
                    this->right_rotate(grandfather, parent_of_grandfather);
                    curr = *grandfather;
                }
                else
                {
                    this->left_rotate(grandfather, parent_of_grandfather);
                    this->left_rotate(grandfather, parent_of_grandfather);
                    curr = *grandfather;
                }
            }
        }
        if (parent_of_grandfather != nullptr)
        {
            path_to_subtree_root_exclusive.push(parent_of_grandfather);
        }

        parent = nullptr;
        grandfather = nullptr;
        parent_of_grandfather = nullptr;

    }
}
```

2.4 Красно-чёрное дерево

Красно-чёрное дерево (RB-Tree) — это сбалансированное бинарное дерево поиска, которое используется для хранения и управления отсортированными данными. Каждый узел имеет цвет, который может быть красным (red) или черным (black). Красно-черные деревья обладают несколькими важными свойствами и инвариантами, которые обеспечивают их сбалансированность и эффективность вставки, удаления и поиска элементов:

•Корень чёрный

•Листья без данных чёрные

•У красного узла оба потомка будут чёрными

В структуру узла добавляется цвет:

```
struct red_black_node : public binary_search_tree<tkey, tvalue, tkey_comparer>::node
{
   color_node color;
};
```

Чтобы определить цвет узла, была написан метод get_color:

```
color_node get_color(red_black_node *current_node)
{
   return current_node == nullptr ? color_node::BLACK : current_node->color;
}
```

Во время вставки нового узла в красно-чёрное дерево, он добавляется на место одного из листьев, окрашивается в красный цвет и к нему прикрепляется два листа. Затем проверяется балансировку. Если отец нового элемента красный, то достаточно рассмотреть только два случая:

•"Дядя" этого узла тоже красный. Перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём ставим чёрный цвет.

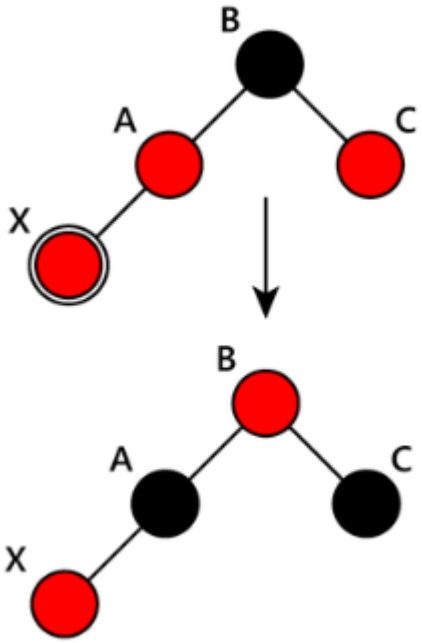

Рис. 3. Первый случай

•"Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком.

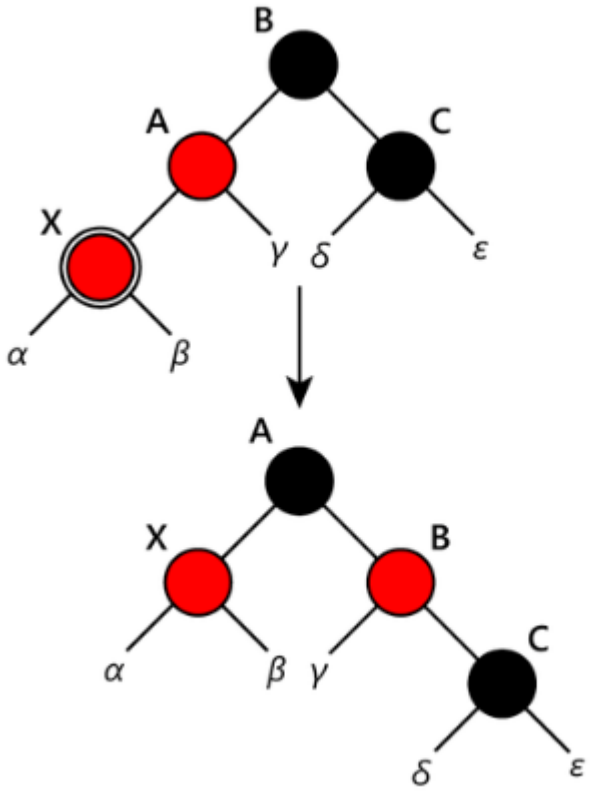

Рис.4 Второй случай

Для удаления узла используется метод из бинарного дерева поиска, а затем в самом красно-чёрном дереве происходит балансировка. Во время удаления может произойти три случая:

•Если у вершины нет детей, то изменяем указатель на неё у родителя на nullptr.

•Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.

•Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Ведь при удалении красной вершины свойства дерева не нарушаются, то и восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

•Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет.

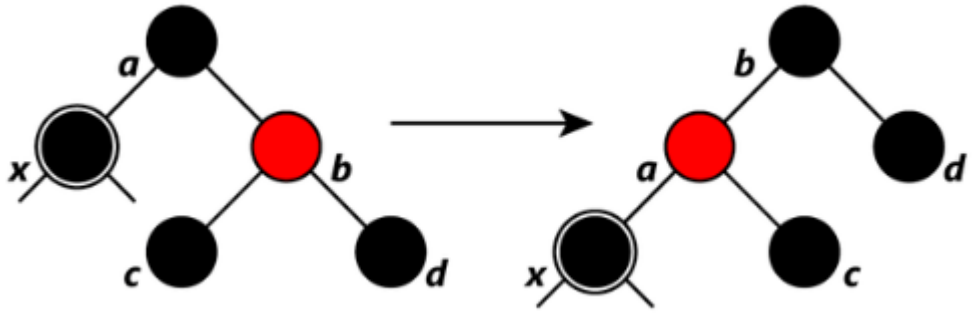

Рис. 5. Первый случай удаления

А если брат текущей вершины был чёрным, то получается 3 случая:

•Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины.

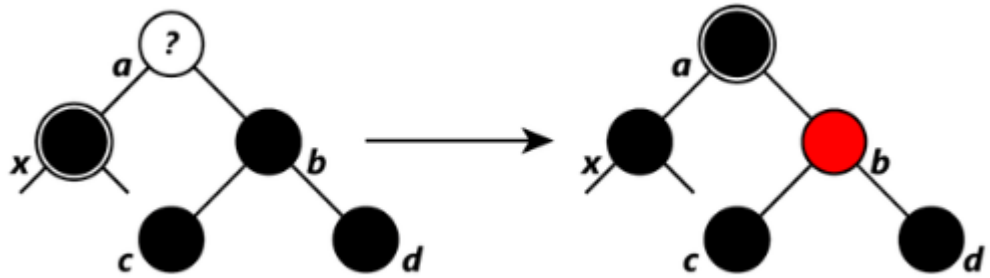

Рис. 6. Второй случай удаления

•Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение.

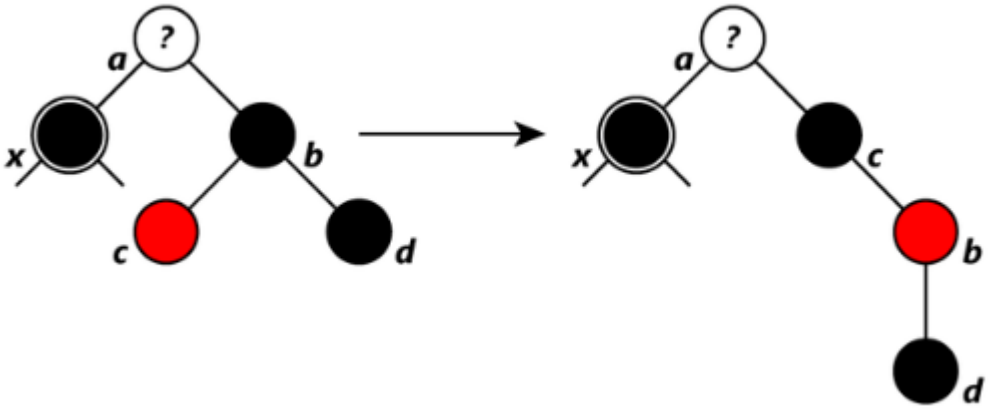

Рис. 7. Третий случай удаления

•Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение.

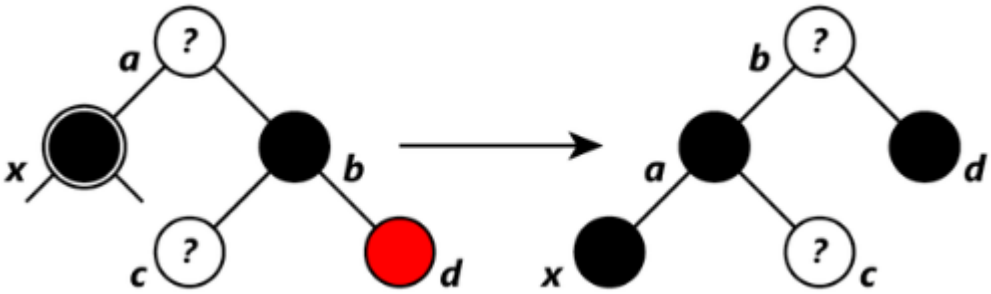

Рис. 8. Четвёртый случай удаления

3. Распределение памяти

Аллокатор - это объект, управляющий выделением и освобождением памяти. Ему выделяется блок памяти определенного размера, которым он управляет. Внутри аллокатора содержится как системная информация, так и блоки памяти, запрошенные внешними объектами.

В рамках каждого аллокатора предоставляются методы для выделения и освобождения памяти. Метод выделения зависит от конкретного аллокатора, но методы поиска подходящего блока памяти остаются одинаковыми:

•Метод "первого подходящего": осуществляется поиск первого блока памяти, размер которого соответствует размеру запрашиваемой памяти.

•Метод "худшего подходящего": выполняется поиск блока памяти, размер которого наименьший среди всех блоков, подходящих по размеру для запрашиваемой памяти.

•Метод "лучшего подходящего": проводится поиск блока памяти, размер которого наибольший среди всех блоков, подходящих по размеру для запрашиваемой памяти.

Каждый класс аллокатора представляет реализацию интерфейса класса "memory":

```
class memory
{

public:
   enum class METHODS {
       BEST,
       WORST,
       FIRST
   };

   void* operator +=(size_t const&);

   void operator -=(void *obj);
```

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

    [[nodiscard]] virtual void *allocate(size_t size) const = 0;

    virtual void deallocate(void * deallocated_block) const = 0;

    memory() = default;

    memory(memory const&) = delete;

    virtual ~memory();
};
```

3.1. Аллокатор с освобождением в рассортированном списке

Для выделения блока памяти из аллокатора достаточно пройти по списку свободных блоков и найти подходящий блок памяти с использованием одного из трех методов.

Чтобы освободить память, необходимо изменить указатель на следующий свободный блок для блока, за которым будет вставлен новый участок памяти, затем изменить указатель на следующий свободный блок во вставленном блоке и проверить что блоки не нужно склеивать, это делается для свободных блоков, которые располагаются рядом.

В управляемом аллокатором блоке памяти содержится следующая информация:

- Размер доступной памяти.
- Указатель на первый доступный свободный блок.
- Внешний аллокатор.
- Указатель на логгер.

Каждый свободный блок памяти, который представлен в виде списка внутри памяти, содержит следующие данные:

- Размер свободного блока.
- Указатель на следующий свободный блок.

3.2. Аллокатор с освобождением с дескрипторами границ

Теперь требуется не просматривать список свободных блоков, а анализировать список занятых блоков. В самый первый проход, когда имеется обширный свободный блок памяти, нужно переместить указатель на первый занятый блок, установить его размер и установить указатели на предыдущий и следующий блоки в nullptr.

После этого, для того чтобы выделить блок памяти, требуется осуществить обход списка занятых блоков, вычисляя размеры свободных блоков, и найти подходящий блок.

Чтобы удалить переданный блок, достаточно изменить у него на предыдущий и следующий занятый блок.

В блоке памяти содержится:

- Размер памяти
- Указатель на первый занятый блок
- Внешний аллокатор
- Указатель на логгер

Каждый занятый блок памяти содержит:

- Размер блока
- Указатель на предыдущий занятый блок
- Указатель на следующий занятый блок

3.3. Аллокатор с алгоритмом системы двойников

Алгоритм, лежащий в основе системы двойников, представляет собой следующий принцип: каждый доступный блок памяти имеет свой "двойник". Здесь важно отметить, что размер каждого свободного блока памяти является степенью двойки. В данном контексте, термин "двойник" обозначает блок памяти, который необязательно является свободным, но имеет или в перспективе будет иметь (при освобождении блока соседние свободные блоки объединяются в один) тот же размер, что и текущий блок.

В начале необходимо вычислить относительный адрес блока относительно начала памяти. Затем следует вычислить размер блока, возводя двойку в степень, предоставленную в системной информации для данного блока. Результат операции XOR между относительным адресом блока и его размером позволяет вычислить адрес "двойника". Прибавив этот результат к начальному адресу памяти, можно получить адрес блока-двойника.

Для добавления блока требуется осуществить перебор доступных свободных блоков в списке, найти подходящий, разделить блок пополам, и в случае если половина блока по-прежнему соответствует требуемому объему памяти, то следует

обновить указатели для новых блоков, образовавшихся в результате разделения текущего блока.

При освобождении памяти необходимо начать с вставки нового блока, аналогично добавлению узла в односвязный список, и при этом правильно настроить связи указателей. Затем производится обход до блока-двойника, где проверяется его доступность и соответствие размера с текущим блоком. Если условия выполняются, блоки объединяются в один путем переназначения указателей. Этот подход упрощает процесс слияния блоков, однако не сильно увеличивает скорость освобождения памяти.

Вот так выглядит свободный блок памяти:

- Бит занятости блока
- Степень двойки
- Указатель на предыдущий блок
- Указатель на следующий блок

4. Разработка и реализация приложения, управляющего хранилищем

4.1 Состав хранилища

Хранилище данных состоит из:

• Набора пулов, содержащих схемы данных

• Схем данных, содержащих коллекции данных

• Коллекции данных, содержащие объекты

Каждый из этих компонентов обладает уникальным именем и представляет собой ассоциативный контейнер. Пулы и схемы данных реализованы в виде АВЛ деревьев, а тип коллекции данных выбирается пользователем и может быть АВЛ-деревом, косым деревом или красно-чёрным деревом.

Коллекции:

```
using type_collection = associative_container<size_t, type_value*>;
using type_collection_avl = avl_tree<size_t, type_value*, compare_data_keys>;
using type_collection_rb = red_black_tree<size_t, type_value*, compare_data_keys>;
using type_collection_splay = splay_tree<size_t, type_value*, compare_data_keys>;
```

Схемы:

```
using type_scheme = associative_container<std::string, std::pair<type_collection*,
std::map<std::string, type_order_collection*>>*>;
```

```
using type_scheme_avl = avl_tree<std::string, std::pair<type_collection_avl*,
std::map<std::string, type_order_collection*>>*, compare_str_keys>;
using type_scheme_rb = red_black_tree<std::string, std::pair<type_collection_rb*,
std::map<std::string, type_order_collection*>>*, compare_str_keys>;
using type_scheme_splay = splay_tree<std::string, std::pair<type_collection_splay*,
std::map<std::string, type_order_collection*>>*, compare_str_keys>;
```

Пулы:

```
using type_pool = associative_container<std::string, std::pair<type_scheme*, memory*>>;
using type_pool_avl = avl_tree<std::string, std::pair<type_scheme_avl*, memory*>,
compare_str_keys>;
using type_pool_rb = red_black_tree<std::string, std::pair<type_scheme_avl*, memory*>,
compare_str_keys>;
using type_pool_splay = splay_tree<std::string, std::pair<type_scheme_avl*, memory*>,
compare_str_keys>;
```

Хранилища:

```
using type_data_base = associative_container<std::string, std::pair<type_pool*, memory*>>;
using type_data_base_avl = avl_tree<std::string, std::pair<type_pool_avl*, memory*>,
compare_str_keys>;
using type_data_base_rb = red_black_tree<std::string, std::pair<type_pool_avl*, memory*>,
compare_str_keys>;
using type_data_base_splay = splay_tree<std::string,std::pair<type_pool_avl*, memory*>,
compare_str_keys>;
```

4.2 Реализация механизма, позволяющего выполнять запросы к данным в рамках коллекции данных на заданный момент времени

Чтобы была возможность выполнять запросы к данным на заданный момент времени, было принято решение хранить все изменения объекта в контейнере std::map. Каждый элемент std::map имеет ключ - это момент времени изменения, и значение - это вектор пар, содержащих имя измененного поля и новое значение этого поля.

Когда пользователь хочет получить данные на текущий момент времени, система последовательно применяет все изменения, хранящиеся в std::map для данного объекта. Если пользователь указывает конкретный момент времени, система применяет изменения до тех пор, пока не найдется ключ, значение которого соответствует указанному моменту времени.

По заданию при реализации этого механизма требовалось использовать поведенческие паттерны проектирования “Команда” и “Цепочка обязанностей”.

Паттерн "Цепочка обязанностей" (Chain of Responsibility) - это поведенческий паттерн проектирования, который позволяет передавать запросы по цепочке объектов-обработчиков. Каждый объект-обработчик решает, может ли он обработать запрос, и если нет, передает запрос следующему объекту в цепочке.
Основные участники паттерна "Цепочка обязанностей":

1) Абстрактный обработчик: Это абстрактный класс или интерфейс, который объявляет метод обработки запроса и имеет ссылку на следующий обработчик в цепочке.

2)Конкретные обработчики: Эти классы реализуют интерфейс обработчика и предоставляют конкретную реализацию обработки запроса. Они также решают, должны ли они передавать запрос следующему обработчику в цепочке.

3)Клиент: Клиент создает объекты-обработчики и формирует цепочку, устанавливая связи между ними. Затем клиент отправляет запрос на обработку в начало цепочки.

Паттерн "Команда" (Command) - это один из поведенческих паттернов проектирования, который используется для инкапсуляции запросов или операций как объекты, позволяя параметризовать клиентов с указанием конкретных операций, управлять последовательностью запросов. То есть мы проходимся по всем парсерам, находим который необходим нам конкретно сейчас и выполняем запрос, используя паттерн "Команда". Базовый класс parser:

```
class parser
{
public:
   enum class TREE
   {
       RED_BLACK_TREE,
       SPLAY_TREE,
       AVL_TREE
   };

public:
   class command
   {
   public:
       virtual void execute(
               const std::vector<std::string> &params,
               std::pair<type_pool*, memory*> *pool,
               std::pair<type_scheme *, memory*> *scheme,
               std::pair<type_collection*, std::map<std::string, type_order_collection*>>
*collection,
               std::istream &stream,
               bool console) = 0;
       virtual ~command() = default;
   };

private:
   static type_data_base *_data_base;
   static logger *_logger;
```

```
protected:
   parser *_next_parser;

public:

   static std::string delete_spaces(const std::string &str);

   static std::vector<std::string> split_by_spaces(const std::string &str);

   parser();

   void set_next(parser *next_parser);

   void accept_request(const std::string &request);

   virtual void handle_request(const std::vector<std::string> &params, std::istream &stream,
bool console) = 0;

   static type_data_base *get_instance();

   static logger *get_logger();

   static type_data_base *allocate_data_base(TREE tree = TREE::AVL_TREE);

   static type_pool *allocate_pool(TREE tree = TREE::AVL_TREE);

   static type_scheme *allocate_scheme(TREE tree = TREE::AVL_TREE);

   static type_collection *allocate_collection(memory *memory_init, TREE tree =
TREE::AVL_TREE);

   virtual ~parser();
};
```

4.3 Механизм, позволяющий выполнять эффективный поиск по различным отношениям порядка на пространстве данных

В схеме в качестве значения узла у нас несколько коллекций, у одной коллекции компаратор по основному ключу, у другой коллекции компаратор по другому полю, у третьей коллекции по другому полю и так далее. Все коллекции содержат указатель на одни и те же данные, но компаратор у них другой и это позволяет производить поиска по различным порядкам отношений.

```
using type_order_collection = avl_tree<std::string, type_value*, compare_str_keys>;
using type_scheme = associative_container<std::string, std::pair<type_collection*,
std::map<std::string, type_order_collection*>>*>;
```

4.4 Эффективное хранение строк

При реализации использовался Паттерн проектирования "Приспособленец" (Flyweight) .
.Приспособленец является структурным паттерном, который используется для оптимизации работы с большим количеством мелких объектов, разделяя общее состояние между ними. Этот паттерн позволяет уменьшить использование памяти и улучшить производительность при работе с множеством объектов, имеющих схожее или одинаковое внутреннее состояние.

Главная идея паттерна "Приспособленец" заключается в том, чтобы вынести общие данные из отдельных объектов и предоставить им доступ к этим данным через внешний объект-приспособленец (Flyweight). Таким образом, экономится память, так как общие данные разделяются между множеством объектов, и они могут быть переиспользованы.

В самом классе написаны два метода для получения и установки значения:

```
class flyweight_for_string
{
private:
   std::string _value;
public:
   void set_value(const std::string &value);
   const std::string &get_value() const;
};
```

Чтобы управлять всеми приспособленцами был написан класс string_flyweight_factory, который их хранит, используя std::unordered_map.

Теперь не требуется хранить в памяти одинаковые строки, потому что если будет присваиваться значение поля, которое уже есть, создастся легковес. И будет достаточно присвоить указатель на легковес.

Как теперь будет выглядеть meeting_data:

```
#include "../flyweight/flyweight_for_string.h"
class meeting_data
{

private:

   std::shared_ptr<flyweight_for_string> _id;
   std::shared_ptr<flyweight_for_string> _type;
   std::shared_ptr<flyweight_for_string> _format;
   std::shared_ptr<flyweight_for_string> _description;
   std::shared_ptr<flyweight_for_string> _link;
   std::shared_ptr<flyweight_for_string> _surname;
   std::shared_ptr<flyweight_for_string> _name;
   std::shared_ptr<flyweight_for_string> _patronymic;
```

```
    std::shared_ptr<flyweight_for_string> _date;
    std::shared_ptr<flyweight_for_string> _start_time;
    std::shared_ptr<flyweight_for_string> _duration;
    std::shared_ptr<flyweight_for_string> _members;

public:
    std::shared_ptr<flyweight_for_string> get_id() const;
    void set_id(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_type() const;
    void set_type(const std::string &type);

    std::shared_ptr<flyweight_for_string> get_format() const;
    void set_format(const std::string &format);

    std::shared_ptr<flyweight_for_string> get_description() const;
    void set_description(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_link() const;
    void set_link(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_surname() const;
    void set_surname(const std::string &surname);

    std::shared_ptr<flyweight_for_string> get_name() const;
    void set_name(const std::string &name);

    std::shared_ptr<flyweight_for_string> get_patronymic() const;
    void set_patronymic(const std::string &patronymic);

    std::shared_ptr<flyweight_for_string> get_date() const;
    void set_date(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_start_time() const;
    void set_start_time(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_duration() const;
    void set_duration(const std::string &description);

    std::shared_ptr<flyweight_for_string> get_members() const;
    void set_members(const std::string &description);

};
```

5. Как пользоваться приложением

Запросы можно вводить из файла или из консоли. Если требуется запустить файл с командами вводится команда: READ file_name. При работе в консоли можно сразу

вводить нужные команды. Возможные команды:

- CREATE_POOL pool_name
- CREATE_SCHEME pool_name scheme_name
- CREATE_COLLECTION pool_name scheme_name collection_name
- INSERT_DATA pool_name scheme_name collection_name
- GET_DATA pool_name scheme_name collection_name
- GET_RANGE pool_name scheme_name collection_name
- UPDATE_OBJECT pool_name scheme_name collection_name
- FINISH_UPDATE
- DELETE_OBJECT pool_name scheme_name collection_name
- DELETE_POOL pool_name
- DELETE_SCHEME pool_name scheme_name
- DELETE_COLLECTION pool_name scheme_name collection_name
- FINISH

Примеры работы программы:

```
CREATE_POOL pool_name
What kind of allocator do you want?
1. Allocator list
2. Allocator descriptor
3. Allocator buddies
Enter the number: 1
CREATE_SCHEME pool_name scheme_name
CREATE_COLLECTION pool_name scheme_name collection_name
Choose an type of tree for collection:
1. Red-black tree
2. Splay tree
3. AVL tree
Put the number: 3
INSERT_DATA pool_name scheme_name collection_name
ID: 100
TYPE: interview
FORMAT: remote
DESCRIPTION: lalala
LINK: -
SURNAME: Barabanov
NAME: Klim
PATRONYMIC: Aleksandrovich
DATE: 01/09/2023
Start time: 10:10
Duration: 120
Members: 3
GET_DATA pool_name scheme_name collection_name
Choose field to be searched:
1. Id
2. Type
3. Format
4. Description
5. Link
```

Рис. 9. Первый тест, первый скрин

```
6. Surname
7. Name
8. Patronymic
9. Date
10. Start time
11. Duration
12. Members
Number field: 1
ID: 100
At what point in time do you want to get the data?
1. At right now
2. At a given point in time
Enter the number: 1

FOUND MEETING:

TYPE: interview
FORMAT: remote
DESCRIPTION: lalala
LINK: -
SURNAME: Barabanov
NAME: Klim
PATRONYMIC: Aleksandrovich
DATE: 01/09/2023
START_TIME: 10:10
DURATION: 120
MEMBERS: 3

DELETE_COLLECTION pool_name scheme_name collection_name
DELETE_SCHEME pool_name scheme_name
DELETE_POOL pool_name
FINISH

Process finished with exit code 0
```

Рис. 10. Первый тест второй скрин.

test.txt

```
CREATE_POOL pool
ALLOCATOR: allocator_list
CREATE_SCHEME pool scheme
CREATE_COLLECTION pool scheme collection
TREE: RED_BLACK_TREE

INSERT_DATA pool scheme collection
ID: 11
TYPE: interview
FORMAT: remote
DESCRIPTION: 1
LINK: 1
SURNAME: Ivanov
NAME: Ilya
PATRONYMIC: Ivanovich
DATE: 10/09/2023
START_TIME: 1
DURATION: 69
MEMBERS: 11

INSERT_DATA pool scheme collection
ID: 22
TYPE: corporate
FORMAT: remote
DESCRIPTION: 2
LINK: https/asdfasdfggggg
SURNAME:
NAME: 2
PATRONYMIC: 2
DATE: 10/09/2023
START_TIME: 11:00
DURATION: 255
MEMBERS: 2
```

Рис. 11. Второй тест, первый скрин

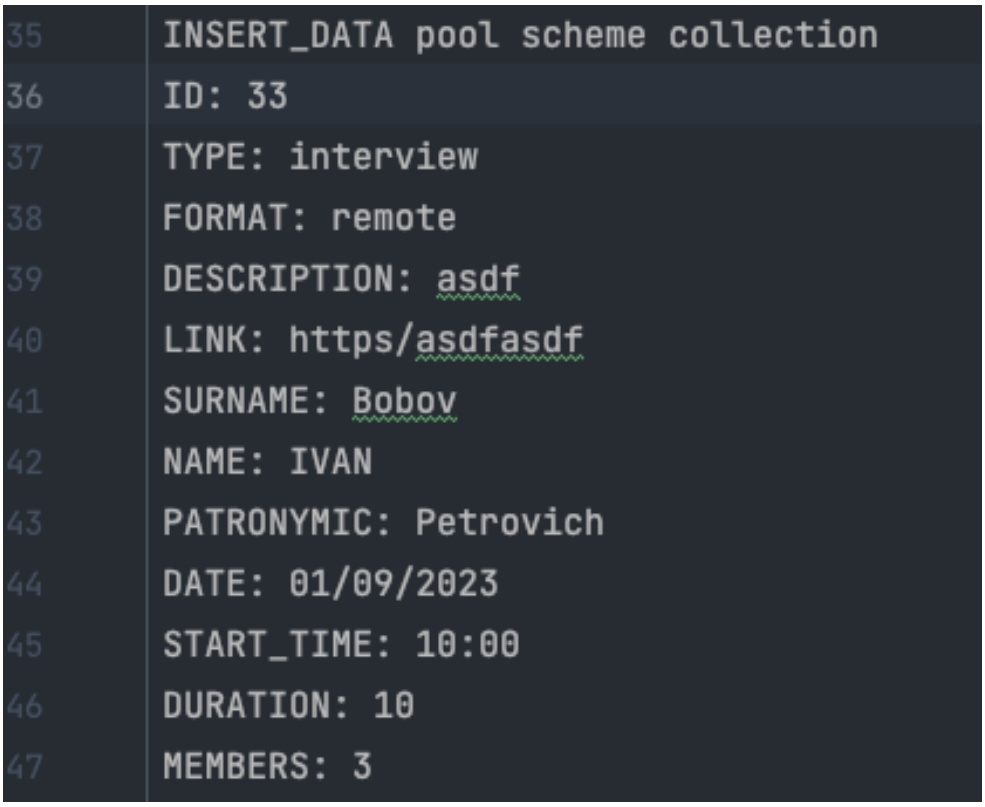

Рис. 12. Второй тест, второй скрин

```
READ test.txt
GET_RANGE pool scheme collection
Choose field to be searched:
1. Id
2. Type
3. Format
4. Description
5. Link
6. Surname
7. Name
8. Patronymic
9. Date
10. Start time
11. Duration
12. Members
Number field: 1
PUT LOWER BOUND:
ID: 15
PUT UPPER BOUND:
ID: 30
At what point in time do you want to receive data?
1. Current point in time
2. Set point in time
Put the number: 1
TYPE: corporate
FORMAT: remote
DESCRIPTION: 2
LINK: https/asdfasdfggggg
SURNAME:
NAME: 2
PATRONYMIC: 2
DATE: 10/09/2023
START_TIME: 11:00
DURATION: 255
MEMBERS: 255
```

Рис. 13. Второй тест, третий скрин

6. Вывод

Было реализовано приложение, которое позволяет имитировать работу базы данных и имеет несколько уровней хранения - пул, схема, коллекция. При его создании потребовалось реализовать деревья, аллокаторы, логгер, а также применить несколько паттернов. Полученные знания являются неоценимыми и будут служить мне надежной основой в дальнейших проектах.

7. Список использованных источников

1. Кормен, Лейзерсон, Ривест, Штайн - Алгоритмы построение и анализ.
2. Кнут - Искусство программирования, т. 3
3. Эрик Фримен, Элизабет Робсон - Паттерны проектирования

8. Приложение

Ссылка - https://github.com/Madjesk/data_base_fundamental_algorithms