МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ”

(национальный исследовательский университет)

Курсовая работа по курсу “Фундаментальные алгоритмы”

студент: Соломатина Светлана Викторовна (_____)

группа: М8О-213Б-21

преподаватель: Романенков Александр Михайлович (_____)

дата: __/__/____

Москва, 2023

Введение

Задание курсовой работы:

На языке программирования C++ (стандарт C++14 и выше) реализуйте приложение, позволяющее выполнять операции над коллекциями данных заданных типов (типы обслуживаемых объектов данных определяются вариантом) 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 ее 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 файла, путь к которому подаётся в качестве аргумента командной строки. Формат команд в файле определите самостоятельно.

Дополнительные задания, реализованные в этой курсовой работе:

1. Реализуйте интерактивный диалог с пользователем. Пользователь при этом может вводить конкретные команды (формат ввода определите самостоятельно) и подавать на вход файлы с потоком команд.

2. Реализуйте механизм, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени (дата и время, для которых нужно вернуть актуальную версию данных, передаётся как параметр). Для реализации используйте поведенческие паттерны проектирования “Команда” и “Цепочка обязанностей”.

4. Обеспечьте хранение объектов строк, размещенных в объектах данных, на основе структурного паттерна проектирования “Приспособленец”. Дублирования объектов строк для разных объектов (независимо от контекста хранения) при этом запрещены. Доступ к строковому пулу обеспечьте на основе порождающего паттерна проектирования “Одиночка”.

6. Реализуйте возможность кастомизации (при создании) реализаций ассоциативных контейнеров, репрезентирующих коллекции данных: АВЛ-дерево, красно-чёрное дерево, косое дерево.

7. Реализуйте 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.

Вариант курсовой работы 27:

Тип данных:

3. Информация о прохождении контеста соискателем (id соискателя, ФИО соискателя (раздельные поля), дата рождения соискателя, ссылка на резюме соискателя, id закрепленного за соискателем HR-менеджера, id контеста, ЯП на котором реализуются задачи контеста, кол-во задач в контесте, кол-во решённых задач в контесте, было ли обнаружено списывание с микронаушником)

Контейнер: косое дерево

IPC: Windows shared memory + Windows semaphores

Описание реализованного приложения

Реализованное в ходе курсовой работы приложение представляет собой многокомпонентную программу, предназначенную для формирования, хранения, добавления, удаления и просмотра данных о прохождении контеста соискателем. Программа применяется для структурирования и упорядочивания данных, а также быстрого поиска необходимых записей.

Поток команд, выполняемых в начале приложения, подаётся из файла, путь к которому подаётся в качестве аргумента командной строки. После выполнения изложенных в файле команд приложение предлагает возможность интерактивного диалога с пользователем. При удовлетворительном ответе в стандартный поток вывода печатается справка о программе и выполняемом функционале приложения, предлагается ввод пользователем команд. Пользователь вводит конкретные команды, которые в ходе выполнения программой синтаксического разбора введенных лексем проверяются на корректность, и, в случае успеха, выполняются с обращением к объекту класса $data\_base$. Класс $data\_base$ реализует основную задачу курсовой работы – выполнение операций над коллекциями данных заданных типов и контекстами их хранения (коллекциями данных).

Описание внешних библиотек

Таблица 1. Описание внешних библиотек

Внешняя библиотека	Описание	В программе
<iostream>	Часть стандартной библиотеки C++. Представляет собой объектно-ориентированную иерархию классов, где используется и множественное, и виртуальное наследование. В библиотеке реализована поддержка для файлового ввода/вывода данных встроенных типов.	Чтение пользовательского ввода/вывода, ведение интерактивного диалога с пользователем, сообщение программой об ошибках/неточностях пользовательских запросов, сообщение о возникших исключительных ситуациях.
<tuple>	Создает последовательность элементов фиксированной длины.	Возвращение функциями, отвечающими за синтаксический разбор

	Шаблон класса описывает объект, в котором хранятся N объектов типов T_1,T_2..., T_N соответственно. $0 \leq N \leq N_{max}$.	пользовательского ввода, отдельных лексем, предназначенных для работы отдельных методов и функций.
<vector>	Эмулирует работу стандартного массива С, а также некоторые дополнительные возможности, вроде автоматического изменения размера вектора при вставке или удалении элементов. Все элементы вектора должны принадлежать одному типу.	Является типом возвращаемого значения метода $find\_in\_range$ класса $data\_base$, который находит в заданной коллекции записи с ключом k, $bound_{min} \leq k \leq bound_{m}$.
<fstream>	Класс потока ввода/вывода для работы с файлами. Объекты этого класса поддерживают объект filebuf в качестве своего внутреннего буфера потока, который выполняет операции ввода/вывода в файле, с которым они связаны.	Чтение команд из файла, подаваемого как аргумент командной строки программы.
<stack>	Контейнерный адаптер, который дает программисту функциональность стека, в частности, структуру данных LIFO (последним пришел — первым вышел). Шаблон класса выступает в качестве оболочки базового контейнера — предоставляется только	Используется при поиске, вставке и удалении элементов в деревьях.

	определенный набор функций.	
<cmath>	Объявляет набор функций для вычисления общих математических операций и преобразований.	Вычисление размера блоков памяти для аллокатора типа системы двойников, представленного классом $memory\_with\_buddy\_syst$.

Спецификация программы

Ниже представлены заголовочные файлы программы и их краткое описание.

Таблица 2. Спецификация программы

main.cpp	Производит открытие и чтение файла, путь к которому передан как аргумент командной строки, создает объекты классов $logger$ и $data\_base$, выполняет команды из файла, консольные команды.
data_base.h	Содержит класс $data\_base$, наследник классов $logger\_holder$ и $memory\_holder$.
db_user_communication.h	Содержит идентификаторы возможных команд, исключение $parse\_exception$, функции, предназначенные для синтаксического разбора пользовательского ввода, функции, предназначенные для обращения к объекту класса $data\_base$ с целью добавления, обновления, чтения, удаления записи по ключу.
key.h	Содержит классы key и $key\_comparer$.
db_value.h	Содержит идентификаторы полей значения записи, класс db_value.
db_vaue_builder.h	Содержит класс $db\_value\_builder$.
string_holder.h	Содержит класс $string\_holder$.

command.h	Содержит базовый виртуальный класс $command$ и его классы-наследники: $remove\_command$, $update\_command$, $add\_command$.
handler.h	Содержит базовый виртуальный класс $handler$ и его классы-наследники: $remove\_handler$, $update\_handler$, $add\_handler$.
logger.h	Содержит базовый виртуальный класс $logger$ и его класс-наследник $logger\_impl$.
logger_builder.h	Содержит базовый виртуальный класс $logger\_builder$ и его класс-наследник $logger\_builder\_impl$.
logger_holder.h	Содержит класс $logger\_holder$.
memory_base_class.h	Содержит класс $memory$, наследник $logger\_holder$.
memory_holder.h	Содержит класс $memory\_holder$.
memory_from_global_heap.h	Содержит класс $memory\_from\_global\_heap$, наследник класса $memory$.
memory_with_sorted_list_deallocation.h	Содержит класс $memory\_with\_sorted\_list\_deallocation$, наследника класса $memory$.
memory_with_boundary_tags.h	Содержит класс $memory\_with\_boundary\_tags$, наследника класса $memory$.
memory_with_buddy_system.h	Содержит класс $memory\_with\_buddy\_system$, наследника класса $memory$.
associative_container.h	Содержит базовый виртуальный класс $associative\_container$.
bs_tree.h	Содержит класс bs_tree, наследника классов $associative\_container$, $logger\_holder$, $memory\_holder$.

avl_tree.h	Содержит класс avl_tree, наследника класса bs_tree.
splay_tree.h	Содержит класс $splay\_tree$, наследника класса bs_tree.
rb_tree.h	Содержит класс rb_tree, наследника класса bs_tree.

Описание классов и структур

Класс $data\_base$. Класс-наследник классов $logger\_holder$ и $memory\_holder$. Содержит следующие приватные поля: объект класса $logger$, объект класса $memory$, словарь ($std::map$) аллокаторов, сформированных в процессе работы программы при создании кастомных объектов деревьев структуры, $\_database$ – структура, представляющая собой дерево пулов, каждый из которых – дерево схем, каждая из которых – дерево коллекций. В коллекциях хранятся пары объекта класса key и db_value.

Также класс $data\_base$ содержит классы исключений $db\_insert\_exception$, $db\_find\_exception$, $db\_remove\_exception$ и идентификаторы аллокаторов и деревьев, доступных для кастомизации при создании пула, схемы или коллекции. Ниже приведены методы, реализованные в классе $data\_base$.

Вспомогательные методы:

Листинг 1. Вспомогательные методы класса *data_base*

```
[[nodiscard]] associative_container<std::string, associative_container<std::string,
associative_container<key, db_value *> *> *> *
find_data_pool
(std::string const & pool_name) const;
```

```
[[nodiscard]] associative_container<std::string, associative_container<key, db_value *>
*> *
find_data_scheme
(std::string const & pool_name, std::string const & scheme_name) const;
```

```
[[nodiscard]] associative_container<key, db_value *> *
find_data_collection
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name) const;
```

$find\_data\_pool$: находит в структуре $\_database$ пул по ключу $pool\_name$. Генерирует исключение в случае неудачи.

$find\_data\_scheme$: находит пул по ключу $pool\_name$, вызывая метод $find\_data\_pool$, находит в этом пуле схему по ключу $scheme\_name$.

$find\_data\_collection$: находит схему по ключу $scheme\_name$, вызывая метод $find\_data\_scheme$, в этой схеме находит коллекцию данных по ключу $collection\_name$.

Методы, относящиеся к работе непосредственно со значениями:

Листинг 2. Методы, относящиеся к работе непосредственно со значениями класса $data\_base$

```
void add_to_collection
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
const key& _key, db_value * value) const;
```

```
void update_in_collection
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
const key& _key, std::map<db_value_fields, unsigned char *> upd_dict) const;
```

```
[[nodiscard]] db_value * find_among_collection
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
const key& _key) const;
```

```
[[nodiscard]] db_value * find_with_time
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
const key& _key, uint64_t time_parameter) const;
```

```
[[nodiscard]] std::vector<db_value *> find_in_range
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
key min_key, key max_key) const;
```

```
void delete_from_collection
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
const key& key) const;
```

$add\_to\_collection$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Находит значение типа db_value * по ключу key из этой коллекции, вызывая метод $associative\_container\ get$. Если этого ключа нет в коллекции, добавляет новую запись по ключу key, иначе проверяет цепочку $handler$ * $\_chain\_of\_resp$. Если последняя операция с этим значением – удаление, добавляет в $\_chain\_of\_resp$ новый $handler$ * – объект класса $add\_handler$. Если последняя операция – не удаление, генерирует исключение.

$update\_in\_collection$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Находит значение типа db_value * по ключу key из этой коллекции, вызывая метод $associative\_container\ get$. Если ключ не найден, генерирует исключение. Иначе проверяет цепочку $handler$ * $\_chain\_of\_resp$. Если последняя операция с этим значением – не удаление, добавляет в $\_chain\_of\_resp$ новый $handler$ * – объект класса $update\_handler$, иначе генерирует исключение.

$find\_among\_collection$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Находит значение типа db_value * по ключу

key из этой коллекции, вызывая метод $associative\_container\ get$. Если ключ не найден, генерирует исключение. Возвращает найденное методом get значение $db\_value\ *$.

$find\_with\_time$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Находит значение типа $db\_value\ *$ по ключу key из этой коллекции, вызывая метод $associative\_container\ get$. Если ключ не найден, генерирует исключение. Создает копию найденного значения $db\_value\ *\ found\_value\_copy$, вызывает метод $handle$ первого $handler\ *$ в $\_chain\_of\_resp$ найденного значения. Возвращает $found\_value\_copy$.

$find\_in\_range$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Создает $std::vector < db\_value\ *> \ to\_return\_vector$. С помощью инфиксного обхода дерева коллекции, находит значения, ключ k которых удовлетворяет условиям $bound_{min} \leq k \leq bound_{max}$. Возвращает to_return_vector.

$delete\_from\_collection$: находит коллекцию по ключу $collection\_name$, вызывая метод $find\_data\_collection$. Находит значение типа $db\_value\ *$ по ключу key из этой коллекции, вызывая метод $associative\_container\ get$. Если ключ не найден, генерирует исключение. Иначе проверяет цепочку $handler\ *\ \_chain\_of\_resp$. Если последняя операция с этим значением – не удаление, добавляет в $\_chain\_of\_resp$ новый $handler\ *$ – объект класса $remove\_handler$, иначе генерирует исключение.

Методы, относящиеся к структуре $\_database$:

Листинг 3. Методы, относящиеся к структуре $\_database$ класса $data\_base$

```
memory *
get_new_allocator_for_inner_trees
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
allocator_types_ allocator_type, size_t allocator_pool_size);
```

```
void add_to_structure
(std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
trees_types_ tree_type, allocator_types_ allocator_type, size_t allocator_pool_size);
```

```
void delete_collection
(const std::string & full_path, const std::string & collection_name,
associative_container<std::string,
       associative_container<key, db_value *> *> * parent_scheme);
```

```
void delete_scheme
(const std::string & full_path, const std::string & scheme_name,
associative_container<std::string,
    associative_container<std::string, associative_container<key, db_value *> *> *> *
parent_pool);
```

```
void delete_pool(const std::string & pool_name);
```

```
void delete_from_structure
```

```
(std::string const & pool_name, std::string const & scheme_name, std::string const & collection_name);
```

get_new_allocator_for_inner_trees: создает объект класса *memory * new_allocator*. В соответствии с параметром *allocator_type* и *allocator_pool_size* создает аллокатор требуемого типа с размером доступной памяти *allocator_pool_size*. Добавляет сгенерированный аллоктор в словарь объекта класса *data_base _all_trees_allocators*. Возвращает сгенерированный аллокатор.

add_to_structure: получает объект класса *memory * new_allocator*. При создании нового пула, схемы, коллекции создает соответствующее пулу, схеме, коллекции дерево в соответствии с параметром *tree_type*. Вставляет новый пул, схему или коллекцию соответственно в структуру *_database*, родительский для добавляемой схемы пул или родительскую для добавляемой коллекции схему. Если пул, схема, коллекция с ключом *pool_name*, *scheme_name* или *collection_name* уже есть в соответствующих родительских структурах данных, генерирует исключение.

delete_collection: удаляет коллекцию из родительской схемы, удаляет аллокатор коллекции.

delete_scheme: удаляет схему из родительского пула, собирает все аллокаторы коллекций этой схемы, удаляет схему и все аллокаторы ее коллекций, ее аллокатор.

delete_pool: удаляет пул из структуры *_database*, собирает аллокаторы всех дочерних подструктур, удаляет пул и все аллокаторы подструктур, аллокатор пула.

delete_from_structure: находит пул, схему или коллекцию, которую нужно удалить. Если записи с ключом *pool_name*, *scheme_name* или *collection_name* нет в в соответствующих родительских структурах данных, генерирует исключение. Иначе вызывает метод *delete_pool*,*delete_scheme* или *delete_collection* для этого пула, схемы или коллекции.

Правило пяти:

Листинг 4. Правило пяти класса *data_base*

```
explicit data_base(logger * this_db_logger = nullptr, memory * this_db_allocator = nullptr)
```

```
~data_base() override;
```

```
data_base(data_base const &obj) = delete;
```

```
data_base(data_base &&obj) noexcept = delete;
```

```
data_base &operator=(data_base const &obj) = delete;
```

```
data_base &operator=(data_base &&obj) noexcept = delete;
```

конструктор: с помощью списков инициализации инициализирует поля $\_logger$ и $\_allocator$ значениями $this\_db\_logger$ и $this\_db\_allocator$. Создает структуру $\_database$, определенную номером варианта (splay tree).

деструктор: если структура $\_database$ инициализирована, с помощью постфиксного обхода дерева структуры удаляет все пулы структуры, вызывая метод $delete\_from\_structure\_inner$. Удаляет объект структуры.

Контракт $logger\_holder$ и $memory\_holder$:

Листинг 5. Контракт $logger\_holder$ и $memory\_holder$ класса $data\_base$

```
[[nodiscard]] logger *get_logger() const noexcept override
[[nodiscard]] memory *get_memory() const noexcept override
```

get_logger: возвращает поле $\_logger$.

get_memory: возвращает поле $\_allocator$.

Класс key. Содержит 2 приватных поля $\_applicant\_id$ и $\_contest\_id$ и класс $create\_exception$. В классе key реализованы: перегрузка оператора << и правило пяти.

Листинг 6. Методы класса key

```
friend std::ostream &operator<<(std::ostream &out, const key &_key_);
key(std::ifstream* input_stream, bool is_cin);
key() = default;
key(int applicant_id, int contest_id);
key(key const &obj);
key(key &&obj) noexcept;
key &operator=(key const &obj);
key &operator=(key &&obj) noexcept;
```

Класс $key\_comparer$. Дружественный класс класса key. Содержит перегруженный оператор () – функтор. Служит для сравнения ключей (объектов класса key).

Листинг 7. Метод класса $key\_comparer$

```
int operator()(key x, key y);
```

Класс db_value. Содержит приватные поля $\_surname$, $\_name$, $\_patronymic$, $\_birthday$, $\_link\_to\_resume$, $\_hr\_id$, $\_programming\_language$, $\_task\_count$, $\_solved\_task\_count$, $\_copying$, репрезентирующие поля данных, определенные вариантом, $\_timestamp$ – дата в миллисекундах, когда был создан этот объект,

$_chain\_of\_resp$ – объект класса $handler$, репрезентирующий цепочку обязанностей, $_last\_handler$ – объект класса $handler$, последний объект в цепочке обязанностей этого значения. Объекты строк, размещенные в db_value хранятся в объекте класса $string\_holder$, не допуская дублирования. В классе db_value реализованы:

$make\_a\_copy$: создает и возвращает копию объекта db_value.

$get\_ptr\_from\_string\_holder$: вызывает метод get_string класса $string\_holder$, возвращает указатель на уникальный объект строки.

$remove\_string\_from\_string\_holder$: вызывает метод $remove\_string$ класса $string\_holder$.

конструктор: инициализирует не строковые поля переданными параметрами. Строковые поля запрашивает из объекта класса $string\_holder$ с помощью метода $get\_ptr\_from\_string\_holder$. Вычисляет время создания этого объекта класса db_value.

деструктор: Удаляет строковые поля из объекта класса $string\_holder$ с помощью метода $remove\_string\_from\_string\_holder$. Вызывает удаление цепочки обязанностей этого объекта.

Листинг 8. Методы класса *db_value*

```
friend std::ostream &operator<<(std::ostream &out, const db_value &value);
```

```
db_value * make_a_copy();
```

```
static std::string * get_ptr_from_string_holder(std::string const & s);
```

```
static void remove_string_from_string_holder(std::string const & s)
```

```
db_value(std::string const & surname, std::string const & name, std::string const &
patronymic,
        std::string const & birthday, std::string const & link_to_resume, unsigned
hr_id,
        std::string const & prog_lang, unsigned task_count, unsigned solved_task_count,
bool copying);
```

```
~db_value();
```

Класс $db\_value\_builder$. Реализация паттерна “Строитель”. Дружественный класс класса db_value. Содержит приватные поля $_surname$, $_name$, $_patronymic$, $_birthday$, $_link\_to\_resume$, $_hr\_id$, $_programming\_language$, $_task\_count$, $_solved\_task\_count$, $_copying$, репрезентирующие поля данных, определенные вариантом. Реализует методы:

$with\_surname$: инициализирует поле $_surname$ с помощью move-семантики.

$with\_name$: инициализирует поле $_name$ с помощью move-семантики.

$with\_patronymic$: инициализирует поле $_patronymic$ с помощью move-семантики.

with_birthday: инициализирует поле *_birthday* с помощью move-семантики.

with_link_to_resume: инициализирует поле *_link_to_resume* с помощью move-семантики.

with_hr_id: инициализирует поле *_hr_id*.

with_programming_language: инициализирует поле *_programming_language* с помощью move-семантики.

with_task_count: инициализирует поле *_task_count*.

with_solved_task_count: инициализирует поле *_solved_task_count*.

with_copying: инициализирует поле *_copying*.

check_if_string_of_value_is_empty: проверяет параметр s на пустоту.

build: возвращает объект класса *db_value* по собранным данным.

build_from_stream: принимает, проверяет на корректность пользовательский ввод значения полей объекта класса *db_value*. Возвращает объект класса *db_value* по собранным данным.

Листинг 9. Методы класса *db_value_builder*

```
db_value_builder * with_surname(std::string && surname);
db_value_builder * with_name(std::string && name);
db_value_builder * with_patronymic(std::string && patronymic);
db_value_builder * with_birthday(std::string && birthday);
db_value_builder * with_link_to_resume(std::string && link);
db_value_builder * with_hr_id(int hr_id);
db_value_builder * with_programming_language(std::string && programming_language);
db_value_builder * with_task_count(unsigned task_count);
db_value_builder * with_solved_task_count(unsigned solved_task_count);
db_value_builder * with_copying(bool did_copy);
[[nodiscard]] db_value *build() const;
db_value *build_from_stream(std::ifstream *input_stream, bool is_cin);
static void check_if_string_of_value_is_empty(std::string const& s)
```

Класс *string_holder*. Реализация паттерна “Приспособленец” и “Одиночка”. Содержит приватные поля *static string_holder * _instance* и *_pool*, класс *key_comparer* – компаратор строк. Реализует методы:

get_instance: возвращает указатель на статический объект класса *string_holder*.

get_string: ищет в структуре $\_pool$ значение по переданной строке. В случае удачи инкрементирует значение поля найденного значения, репрезентирующего количество ссылок на объект строки. Иначе добавляет новое значение строки в структуру $\_pool$.

$remove\_string$: ищет в структуре $\_pool$ значение по переданной строке. Если количество ссылок на объект этой строки – 1, удаляет запись из структуры $\_pool$.

Листинг 10. Методы класса $string\_holder$

```
static string_holder *get_instance();
std::string *get_string(std::string const &key);
void remove_string(std::string const &key);
```

Класс $command$. Виртуальный (базовый, описывающий контракт взаимодействия) класс. Реализация паттерна “Команда”. Перечисляет метод выполнения команды – $execute$.

Листинг 11. Методы класса $command$

```
virtual db_value ** execute(db_value **) = 0;
```

Класс $remove\_command$. Наследник класса $command$. Реализует метод выполнения команды удаления – $execute$: вызывает деструктор объекта класса db_value, возвращает указатель на указатель на этот объект.

Класс $update\_command$. Дружественный класс класса db_value. Наследник класса $command$. Содержит приватное поле $\_update\_dictionary$ – словарь <название поля><новое значение>. Реализует метод выполнения команды обновления значения – $execute$: удаляет старые значения полей объекта класса db_value, заменяет их на новые, переданные пользователем.

Класс $add\_command$. Дружественный класс класса db_value. Наследник класса $command$. Содержит приватное поле $\_add\_dictionary$ – словарь <название поля><значение>. Реализует метод выполнения команды добавления значения – $execute$: с помощью объекта класса $db\_value\_builder$ создает новый объект класса db_value на основе $\_add\_dictionary$.

Класс $handler$. Реализация паттерна “Цепочка обязанностей”. Содержит тип перечислений $handler\_types$, перечисляющий типы классов-наследников $handler$, класс-исключение $order\_exception$. Также содержит приватные поля $\_next\_handler$ – следующий за этим обработчик в цепочке обязанностей, $\_command$ – команда, которую выполняет этот обработчик, $timestamp$ - время в

миллисекундах, когда был создан этот обработчик, $\_type\_of\_handler$ – тип этого обработчика. Реализует методы:

$get\_handler\_type$: возвращает значение поля $\_type\_of\_handler$.

set_next: инициализирует значения поля $\_next\_handler$ параметром $handler$.

$handle$: в соответствии с параметром $time\_parameter$ и положением в цепочке обязанностей, вызывает метод $execute$ объекта $\_command$ и метод $handle$ объекта $\_next\_handler$, либо возвращает текущее состояние параметра $request$.

$delete\_chain\_of\_responsibility$: покомпонентно удаляет обработчики цепочки обязанностей.

Листинг 12. Методы класса *handler*

```
[[nodiscard]] handler_types get_handler_type() const;
handler *set_next(handler *handler);
db_value* handle(db_value ** request, uint64_t time_parameter);
void delete_chain_of_responsibility();
```

Класс $remove\_handler$. Наследник класса $handler$. Реализует конструктор $remove\_handler$, который инициализирует приватные поля $\_command$ командой $remove\_command$ и $\_type\_of\_handler$ как $handler\_types$:: $\_remove\_handler\_$.

Класс $update\_handler$. Наследник класса $handler$. Реализует конструктор $update\_handler$, который инициализирует приватные поля $\_command$ командой $update\_command$ и $\_type\_of\_handler$ как $handler\_types$:: $\_update\_handler\_$.

Класс $add\_handler$. Дружественный класс класса db_value. Наследник класса $handler$. Реализует конструктор класса $add\_handler$, который принимает на вход новое значение. Собирает из этого значения словарь – параметр для команды $add\_command$, которой инициализируется приватное поле $\_command$ объекта класса. $\_type\_of\_handler$ принимает значение $handler\_types$:: $\_add\_handler\_$.

Класс $logger$. Интерфейсный класс. Содержит тип перечислений $severity$, класс-исключение $severity\_exception$. Перечисленные методы отображены в листинге 13:

Листинг 13. Методы класса *logger*

```
static std::string severity_to_string_logger(severity s);
static logger::severity from_string_to_severity_parse(const std::string &s);
virtual logger const *log(std::string const &target, severity level) const = 0;
```

Класс $logger\_impl$. Наследник класса $logger$. Содержит приватные поля $this\_logger\_streams$ – словарь потоков этого логгера и $\_all\_loggers\_streams$ – словарь потоков всех логгеров. Переопределяет метод log: для каждого из потоков $this\_logger\_streams$ проверяется условие достаточно строгого $severity$, генерируется запись вида [timestamp][severity]{сообщение $target$}.

Класс $logger\_builder$. Реализация паттерна “Строитель”. Базовый класс. Перечисленные методы отображены в листинге 14.

Листинг 14. Методы класса $logger\_builder$

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

```
[[nodiscard]] virtual logger *build() const = 0;
```

```
virtual logger *config_from_json(const std::string &) = 0;
```

Класс $logger\_builder\_impl$. Дружественный класс класса $logger\_impl$. Содержит приватное поле $\_under\_construction\_logger\_setup$, представляющее собой словарь название файла – $severity$. Переопределяет методы класса $logger\_builder$:

$with\_stream$: добавляет в словарь $\_under\_construction\_logger\_setup$ запись <путь к файлу><$severity$>.

$build$: возвращает объект класса $logger\_impl$, созданного на основе $\_under\_construction\_logger\_setup$.

$config\_from\_json$: создает и возвращает объект класса $logger\_impl$, созданного после парсинга .json файла с помощью методов библиотеки $rapidjson$.

Класс $logger\_holder$. Представляет собой интерфейс-оболочку для использования объекта класса $logger$. Реализует методы логирования с заданной $severity$: $trace\_with\_guard$, $debug\_with\_guard$, $information\_with\_guard$, $warning\_with\_guard$, $error\_with\_guard$, $critical\_with\_guard$, вызывающие метод log_with_guard с соответствующими $severity$, который получает объект класса $logger$ и, при его наличии, вызывает его метод log. Перечисляет виртуальный метод get_logger.

Класс $memory$. Наследник класса $logger\_holder$. Базовый класс. Содержит тип перечислений $Allocation\_strategy$, класс-исключение $memory\_exception$, приватное поле $\_ptr\_to\_allocator\_metadata$, репрезентирующее начало блока доверенной аллокатору памяти. Реализует методы:

$dump\_occupied\_block\_before\_deallocate$ – логгирует неслужебное содержимое освобождаемого участка перед освобождением в виде коллекции значений байт.

$get\_ptr\_size\_of\_allocator\_pool$: возвращает указатель на размер доверенной аллокатору области памяти.

$get\_ptr\_logger\_of\_allocator$: возвращает указатель на указатель на объект класса $logger$ этого аллокатора.

$get\_ptr\_to\_ptr\_parent\_allocator$: возвращает указатель на родительский аллокатор этого аллокатора (объект класса $memory$).

$get\_ptr\_allocation\_mode$: возвращает указатель на стратегию аллокации этого аллокатора (объект типа перечислений $Allocation\_strategy$).

$get\_ptr\_to\_ptr\_to\_pool\_start$: возвращает указатель на первый пул аллокатора, где размещаются данные пользователя.

$address\_to\_hex$: возвращает адрес указателя в виде строки в шестнадцатеричной записи.

$operator\ +=$: вызывает метод объекта $memory\ allocate$.

$operator\ -=$: вызывает метод объекта $memory\ deallocate$.

Перечисленные методы класса $memory$ отображены в листинге 15.

Листинг 15. Методы класса memory

```
[[nodiscard]] virtual size_t get_allocator_service_block_size() const;
[[nodiscard]] virtual size_t* get_ptr_size_of_allocator_pool() const;
[[nodiscard]] virtual void *get_first_available_block_address() const;
[[nodiscard]] virtual void ** get_first_available_block_address_address() const;
[[nodiscard]] virtual size_t get_available_block_service_block_size() const;
virtual size_t get_available_block_size(void * memory_block) const;
virtual void * get_next_available_block_address(void * memory_block) const;
[[nodiscard]] virtual void ** get_first_occupied_block_address_address() const;
[[nodiscard]] virtual size_t get_occupied_block_service_block_size() const;
virtual size_t get_occupied_block_size(void * memory_block) const;
virtual size_t get_size_of_occupied_block_pool(void * const occupied_block) const = 0;
virtual void * get_next_occupied_block_address(void * memory_block) const;
virtual void * get_previous_occupied_block_address(void * memory_block) const;
virtual void *allocate(size_t target_size) const = 0;
virtual void deallocate(void const * const target_to_dealloc) const = 0;
```

Класс $memory\_holder$. Представляет собой интерфейс для использования объекта класса $memory$. Содержит методы для обращения к памяти $allocate\_with\_guard$ и $deallocate\_with\_guard$, которые получают объект класса $memory$, вызывая виртуальный метод get_memory, и, при наличии объекта, вызывают его метод $allocate$, иначе возвращают указатель на выделенный из глобальной кучи участок памяти.

Класс $memory\_from\_global\_heap$. Наследник класса $memory$. Представляет собой прослойку между пользователем и операторами ::new и ::delete, таким образом, что взаимодействие с объектом класса является подобным взаимодействию с ::allocate и ::deallocate. Его $\_ptr\_to\_allocator\_metadata$ указывает только на объект класса $logger$ этого аллокатора. Реализует и переопределяет методы:

$get\_ptr\_logger\_of\_allocator$: переопределяет метод наследуемого класса $memory$, возвращает указатель на указатель на объект класса $logger$.

$allocate$: возвращает указатель на блок памяти, выделенный из глобальной кучи с помощью оператора $::new$, сохраняет размер блока в в начале этого блока, как метаданные блока.

$deallocate$: вызывает метод $dump\_occupied\_block\_before\_deallocate$ класса $memory$, возвращает блок памяти в глобальную кучу с помощью метода $::delete$.

$get\_size\_of\_occupied\_block\_pool$: переопределяет метод наследуемого класса $memory$, возвращает указатель на размер этого занятого блока.

get_logger – контракт взаимодействия с наследуемым классом $logger\_holder$.

Класс $memory\_with\_sorted\_list\_deallocation$. Наследник класса $memory$. Представляет собой аллокатор с освобождением блоков в рассортированном списке. Структура, на которую указывает $\_ptr\_to\_allocator\_metadata$ состоит из размера доверенной аллокатору области памяти, указателя на объект класса $logger$ этого аллокатора, указателя на объект класса $memory$ — родительского для этого аллокатора аллокатора, $allocation\_mode$ – одного из значений $Allocation\_strategy$, указателя на первый свободный блок в доверенной области. Свободные блоки этого аллокатора представляют собой метаданные (размер свободного блока, включая метаданные, указатель на следующий свободный блок) и незанятую область, занятые блоки представляют собой метаданные (размер занятого блока, включая метаданные) и занятую область. Все свободные блоки аллокатора связаны последовательно в виде односвязного списка. Класс реализует и переопределяет методы:

$get\_allocator\_service\_block\_size$: возвращает размер метаданных аллокатора.

$get\_ptr\_to\_allocator\_trusted\_pool$: указывает на начало области памяти аллокатора, доступной для пользовательских манипуляций памятью.

$get\_first\_available\_block\_address$: возвращает указатель на первый свободный блок в доверенной аллоктору области памяти.

$get\_first\_available\_block\_address\_address$: возвращает указатель на указатель на первый свободный блок в доверенной аллоктору области памяти.

$get\_available\_block\_service\_block\_size$: возвращает размер метаданных свободного блока.

$get\_available\_block\_size$: возвращает указатель на размер свободного блока.

$get\_next\_available\_block\_address$: возвращает указатель на следующий свободный блок этого свободного блока.

$get\_occupied\_block\_service\_block\_size$: возвращает размер метаданных занятого блока.

$get\_size\_of\_occupied\_block\_pool$: возвращает размер занятого блока без учета метаданных.

$allocate$: проходит по списку свободных блоков аллокатора. В соответствии со стратегией выделения (best, worst или first), находит подходящий участок памяти. Добавляет остаток свободного блока к аллоцируемуму блоку, если его размер слишком мал, чтобы вместить служебные данные свободного блока. Корректирует односвязный список свободных блоков.

$deallocate$: вызывает метод $dump\_occupied\_block\_before\_deallocate$ класса $memory$, возвращает деаллоцируемый блок в список свободных блоков этого аллокатора.

get_logger – контракт взаимодействия с наследуемым классом $logger\_holder$.

Класс $memory\_with\_boundary\_tags$. Наследник класса $memory$. Представляет собой аллокатор с освобождением блоков с дескрипторами границ. Структура, на которую указывает $\_ptr\_to\_allocator\_metadata$ состоит из размера доверенной аллокатору области памяти, указателя на объект класса $logger$ этого аллокатора, указателя на объект класса $memory$ — родительского для этого аллокатора аллокатора, $allocation\_mode$ – одного из значений $Allocation\_strategy$, указателя на первый занятый блок в доверенной области. Свободные блоки этого аллокатора не хранят никаких метаданных; занятые представляют собой метаданные (размер блока с учётом метаданных и указатели на предыдущий и следующий занятые блоки) и занятый участок памяти. Все

занятые блоки аллокатора последовательно связаны в виде двусвязного списка. Класс реализует и переопределяет методы:

$get\_allocator\_service\_block\_size$: возвращает размер метаданных аллокатора.

$get\_ptr\_to\_allocator\_trusted\_pool$: указывает на начало области памяти аллокатора, доступной для пользовательских манипуляций памятью.

$get\_first\_occupied\_block\_address\_address$: возвращает указатель на указатель на первый занятый блок этого аллокатора.

$get\_occupied\_block\_service\_block\_size$: возвращает размер метаданных занятого блока.

$get\_occupied\_block\_size$: возвращает размер занятого блока.

$get\_size\_of\_occupied\_block\_pool$: возвращает размер занятого блока без учета метаданных.

$get\_next\_occupied\_block\_address$: возвращает указатель на следующий занятый блок.

$get\_previous\_occupied\_block\_address$: возвращает указатель на предыдущий для этого занятый блок.

$allocate$: находит блок достаточного размера в соответствии со стратегией выделения (best, worst или first. Добавляет остаток свободного блока к аллоцируемуму блоку, если его размер слишком мал, чтобы вместить служебные данные занятого блока. Корректирует двусвязный список занятых блоков этого аллокатора.

$deallocate$: вызывает метод $dump\_occupied\_block\_before\_deallocate$ класса $memory$, удаляет деаллоцируемый блок из списка занятых блоков.

get_logger – контракт взаимодействия с наследуемым классом $logger\_holder$.

Класс $memory\_with\_buddy\_system$. Наследник класса $memory$. Представляет собой аллокатор с выделением и освобождением памяти при помощи алгоритма систему двойников. Согласно этому алгоритму, все блоки имеют размер 2^n, таким образом найти “двойника” – блок с тем же размером легко, достаточно использовать оператор побитового исключения. Структура, на которую указывает $\_ptr\_to\_allocator\_metadata$ состоит из степени n доверенной области аллокатора, указателя на объект класса $logger$ этого аллокатора, указателя на объект класса $memory$ — родительского для этого аллокатора аллокатора, указателя на первый свободный блок. Свободный блок представляет собой метаданные (занят ли участок, степень n этого блока, указатель на следующий свободный блок) и свободный участок памяти. Занятый блок

представляет собой метаданные (занят ли участок, степень n этого блока) и занятый участок памяти. Класс реализует и переопределяет методы:

$\_buddy\_system\_is\_block\_available$: возвращает указатель на переменную типа $bool$, отображающую, занят ли блок.

$\_buddy\_system\_get\_size\_of\_block$: возвращает указатель на степень n этого блока.

$\_buddy\_system\_get\_available\_block\_address\_field$: возвращает указатель на следующий свободный блок в этом аллокаторе.

$get\_ptr\_to\_buddy$: возвращает указатель на блок-двойник этого блока.

$get\_allocator\_service\_block\_size$: возвращает размер метаданных аллокатора.

$\_buddy\_system\_get\_ptr\_size\_of\_allocator\_pool$: возвращает указатель на степень n доверенной области памяти аллокатора.

$\_buddy\_system\_get\_ptr\_logger\_of\_allocator$: возвращает указатель на указатель на объект класса $logger$.

$\_buddy\_system\_get\_ptr\_to\_ptr\_parent\_allocator$: возвращает указатель на указатель на объект класса $memory$, родительский аллокатор этого аллокатора.

$\_buddy\_system\_get\_ptr\_to\_ptr\_to\_pool\_start$: возвращает указатель на первый свободный блок этого аллокатора.

$get\_ptr\_to\_allocator\_trusted\_pool$: возвращает указатель на указатель на начало области памяти, доступной для пользовательских манипуляций памятью.

$get\_ptr\_logger\_of\_allocator$: возвращает указатель на указатель на объект класса $logger$.

$get\_occupied\_block\_service\_block\_size$: возвращает размер метаданных занятого блока.

$get\_size\_of\_occupied\_block\_pool$: возвращает размер занятого блока без учета метаданных.

$get\_first\_available\_block\_address$: возвращает указатель на первый свободный блок этого аллокатора.

$get\_available\_block\_service\_block\_size$: возвращает размер метаданных свободного блока.

$get\_next\_available\_block\_address$: возвращает указатель на следующий свободный блок этого свободного блока.

$get\_number\_in\_bin\_pow$: возвращает 2^{pow}.

$get\_bin\_pow\_of\_number$: возвращает $log_2(number)$ – степень n блока.

$allocate$: находит свободный блок, минимальный по размеру. Удаляет этот блок из списка свободных блоков. Делит найденный блок надвое, пока размер блока удовлетворяет искомому, каждый двойник делимого блока добавляется в список свободных блоков.

$deallocate$: вызывает метод $dump\_occupied\_block\_before\_deallocate$ класса $memory$, добавляет блок в список свободных блоков, если это возможно, объединяет новый свободный блок с его двойником.

get_logger – контракт взаимодействия с наследуемым классом $logger\_holder$.

Класс $associative\_container$. Родовой интерфейс (кастомизируемые параметры родового интерфейса: тип ключа tkey, тип значения tvalue, тип компаратора на ключах tkey_comparer), предоставляющий функционал для работы с ассоциативным контейнером:

- $insert$ – вставка пары <ключ><значение> в ассоциативный контейнер.
- get – получает значение из ассоциативного контейнера по ключу.
- $remove$ – удаляет запись из ассоциативного контейнера по ключу.

Листинг 16. Методы класса $associative\_container$

```
virtual void insert(tkey const &key, tvalue &&value) = 0;
```

```
virtual tvalue const &get(tkey const &key) = 0;
```

```
virtual tvalue remove(tkey const &key) = 0;
```

Класс bs_tree. Наследник класса $associative\_container$, $memory\_holder$, $logger\_holder$. Шаблонный класс с кастомизируемыми параметрами: тип ключа tkey, тип значения tvalue, тип компаратора на ключах tkey_comparer. Распределение вложенных в объект дерева данных организовано через аллокатор, подаваемый объекту через конструктор. Реализованы классы префиксного ($prefix\_iterator$), инфиксного ($infix\_iterator$) и постфиксного ($postfix\_iterator$) итераторов для обхода (префиксного, инфиксного, постфиксного) дерева с возвратом из итератора ключа, значения, глубины (относительно корня; глубина корня дерева равна нулю) обходимого узла; для каждого узла дерева, в порядке, определяемом правилом обхода. Также реализованы защищенные методы малого левого и малого правого поворотов. Класс содержит структуру $node$, представляющую собой узел двоичного дерева и содержащую поля key, $value$, указатели на левое и правое поддерево. В классе определены классы-исключения $insert\_exception$, $find\_exception$, $remove\_exception$, $iterator\_exception$. Операции CRD реализованы на основе поведенческого паттерна проектирования “шаблонный метод” с помощью определения классов $template\_method\_basics$, $insertion\_template\_method$, $finding\_template\_method$, $removing\_template\_method$, предусмотрены хуки для выполнения в подклассах BST дополнительных операций до/после рекурсивного вызова относительно текущего узла дерева, реализовано правило пяти. Приватное поле класса – $\_root$ –

указатель на корень BST дерева. Реализует контракт класса $associative\_container$:

- $insert$ – вызывает метод $insert$ класса $insertion\_template\_method$.
- get – вызывает метод $find$ класса $finding\_template\_method$.
- $remove$ – вызывает метод $remove$ класса $removing\_template\_method$.

Класс $template\_method\_basics$. Дружественный класс класса bs_tree, наследник класса $logger\_holder$. Содержит приватное поле $\_target\_tree$ – указатель на дерево, для которого будут реализованы вставка, поиск и удаление. Включает в себя методы:

get_root_node: возвращает указатель на корень дерева $\_target\_tree$.

$find\_path$: возвращает пару <стек><узел>, где стек – путь от корня до этого узла.

$find\_parent$: возвращает родителя этого узла — элемент на вершине стека.

$find\_grandparent$: возвращает родителя родителя этого узла (деда) с помощью стека.

$rotate\_left$: малый поворот налево.

$rotate\_right$: малый поворот направо.

get_logger: контракт взаимодействия с наследуемым классом $logger\_holder$.

Класс $insertion\_template\_method$, наследник $template\_method\_basics$ и $memory\_holder$. Реализует методы:

$insert$: находит место для вставки узла с помощью метода $find\_path$ класса $template\_method\_basics$, выделяет память под новый узел и инициализирует его параметрами. Вызывает виртуальный метод $after\_insert\_inner$.

get_node_size: виртуальный метод, возвращает размер структуры $node$.

$initialize\_memory\_with\_node$: виртуальный метод, инициализирует выделенный участок памяти как объект структуры $node$.

get_memory: контракт взаимодействия с наследуемым классом $memory\_holder$.

Листинг 17. Методы класса *insertion_template_method*

```
void insert(tkey const &key, tvalue &&value);
```

```
[[nodiscard]] virtual size_t get_node_size() const;
```

```
virtual void initialize_memory_with_node(node *target_ptr) const;
```

```
virtual void after_insert_inner(std::stack<node **> &path, node **target_ptr);
```

```
[[nodiscard]] memory *get_memory() const noexcept override;
```

Класс $finding\_template\_method$, наследник $template\_method\_basics$. Реализует метод:

$find$: находит узел с помощью метода $find\_path$ класса $template\_method\_basics$, вызывает виртуальный метод $after\_find\_inner$, возвращает значение узла.

Листинг 18. Методы класса $finding\_template\_method$

```
tvalue const &find(tkey const &key);
```

```
virtual void after_find_inner(std::stack<node **> &path, node **&target_ptr);
```

Класс $removing\_template\_method$, наследник $template\_method\_basics$ и $memory\_holder$. Реализует методы:

$remove$: находит узел с помощью метода $find\_path$ класса $template\_method\_basics$, удаляет элемент из дерева в соответствии с количеством дочерних узлов и корректирует указатели объектов. Вызывает метод $cleanup\_node$, и, если удаляемый элемент не был последним в дереве, вызывает виртуальный метод $after\_remove$. Возвращает значение удаленного узла.

$swap$: меняет указатели местами.

$swap\_nodes$: меняет местами узлы с учетом их возможного близкого “родства”.

$cleanup\_node$: вызывает деструктор объекта $node$, деаллоцирует выделенный под него участок памяти.

get_memory: контракт взаимодействия с наследуемым классом $memory\_holder$.

Листинг 19. Методы класса $removing\_template\_method$

```
virtual tvalue remove(tkey const &key);
```

```
template<typename T>
void swap(T **left, T **right);
```

```
node** swap_nodes(node **one_node, node **another_node);
```

```
void cleanup_node(node **node_address);
```

```
virtual void swap_additional_data(node *one_node, node *another_node);
```

```
virtual void after_remove(std::stack<node **> &path) const;
```

```
memory *get_memory() const noexcept override;
```

Класс avl_tree. Наследник класса bs_tree. Представляет собой AVL-дерево, содержит структуру avl_node – наследник структуры $node$ с приватным полем $height$ – высотой узла. Баланс узла avl_node – разность значений $height$ его левого и правого поддеревьев. Класс включает в себя классы

$template\_methods\_avl$, наследника $template\_method\_basics$, $insertion\_avl\_tree$, наследника $insertion\_template\_method$, $removing\_avl\_tree$, наследника $removing\_template\_method$, реализует правило пяти.

Класс $template\_methods\_avl$. Реализует метод $do\_balance$: считает баланс текущего узла, балансы левых и правых поддеревьев этого узла. При балансе узлов b: $|b| = 2$, производит операции поворота в зависимости от баланса узлов:

- если баланс текущего узла $b = 2$
  - баланс левого поддерева $b_{left} \geq 0$, производит малый правый поворот
  - баланс левого поддерева $b_{left} < 0$, производит большой правый поворот.
- если баланс текущего узла $b = -2$
  - баланс правого поддерева $b_{right} \leq 0$, производит малый левый поворот
  - баланс правого поддерева $b_{right} > 0$, производит большой левый поворот.

Корректирует значения $height$ узлов и продолжает балансировку до корня дерева.

Класс $insertion\_avl\_tree$. Реализует и переопределяет методы:

get_node_size: возвращает размер avl_node.

$initialize\_memory\_with\_node$: инициализирует выделенный участок памяти как объект структуры avl_node.

$after\_insert\_inner$: обновляет значения $height$ родительского и прародительского (деда) узлов. Если есть дед, вызывает метод $do\_balance$ от узла деда.

Класс $removing\_avl\_tree$. Переопределяет методы:

$after\_remove$: обновляет значение $height$ родительского узла. Вызывает метод $do\_balance$ от родительского узла.

$swap\_additional\_data$: обменивает значения $height$ двух узлов.

Класс $splay\_tree$. Наследник класса bs_tree. Представляет собой splay-дерево (косое). Класс включает в себя классы $template\_method\_splay$, наследника $template\_method\_basics$, $insertion\_splay\_tree$, наследника $insertion\_template\_method$, $finding\_splay\_tree$, наследника класса $finding\_template\_method$, $removing\_splay\_tree$, наследника $removing\_template\_method$, реализует правило пяти.

Класс $template\_method\_splay$. Реализует метод $splay$: находит родителя текущего узла. Если родитель – корень дерева, производит операцию zig: выполняет малый левый или малый правый поворот в зависимости от положения текущего узла относительно родительского. Иначе находит деда текущего узла. Выполняет операции поворота в зависимости от положения узлов:

- если и родитель, и текущий узел – левые поддеревья, производит операцию $zig - zig$: малый правый поворот относительно родителя и деда и малый правый поворот относительно текущего и родительского узлов. Случай зеркален, если и родитель, и текущий узел – правые поддеревья.
- если родитель – левое поддерево, текущий узел – правое, производит операцию $zig - zag$: большой левый поворот. Случай зеркален, если родитель – правое поддерево, текущий узел – левое.

Операция продолжается, пока текущий узел не станет корневым узлом дерева.

Класс $insertion\_splay\_tree$. Переопределяет метод $after\_insert\_inner$: вызывает метод $splay$ от добавленного узла.

Класс $finding\_splay\_tree$. Переопределяет метод $after\_find\_inner$: вызывает метод $splay$ от найденного узла.

Класс $removing\_splay\_tree$. Переопределяет метод $after\_remove$: вызывает метод $splay$ от родителя удаленного узла.

Класс rb_tree. Наследник класса bs_tree. Представляет собой красно-черное дерево, свойства которого – одинаковая черная высота (количество черных узлов от корня до любого листа) и отсутствие у красного родителя красных дочерних узлов. Класс содержит структуру rb_node – наследник структуры $node$ с приватным полем $\_color$ – цветом узла. rb_node поддерживает операции чтения, изменения цвета узла. Класс включает в себя классы $insertion\_rb\_tree$, наследника $insertion\_template\_method$, $removing\_rb\_tree$, наследника $removing\_template\_method$, реализует правило пяти.

Класс $insertion\_rb\_tree$. Реализует и переопределяет методы:

get_node_size: возвращает размер rb_node.

$initialize\_memory\_with\_node$: инициализирует выделенный участок памяти как объект структуры rb_node.

$find\_uncle$: возвращает указатель на дядю узла – поддерево деда, отличное от родительского текущего узла.

$insertion\_do\_balance$: производит балансировку дерева после вставки. Если текущий узел – корень дерева, меняет цвет этого узла на черный. Иначе

находит родителя, деда и дядю этого узла. Балансировка требуется, только если родитель текущего узла – красный.

- дядя – красный: меняет цвет родителя и дяди на черный, деда – на красный. Вызывает метод $insertion\_do\_balance$ от деда.
- дядя – черный, родитель – левое поддерево:
  - текущий узел – правое поддерево: производит малый левый поворот относительно родительского и текущего узлов. Меняет текущий узел с родительским, после чего переходит к следующему пункту (текущий узел – левое поддерево).
  - текущий узел – левое поддерево: производит малый правый поворот относительно родителя и деда, окрашивает родителя в черный, а деда – в красный.
- дядя – черный, родитель – правое поддерево:
  - текущий узел – левое поддерево: производит малый правый поворот относительно родительского и текущего узлов. Меняет текущий узел с родительским, после чего переходит к следующему пункту (текущий узел – правое поддерево).
  - текущий узел – правое поддерево: производит малый левый поворот относительно родителя и деда, окрашивает родителя в черный, а деда – в красный.

$after\_insert\_inner$: меняет цвет добавленного узла на красный. Вызывает метод $insertion\_do\_balance$.

Класс $removing\_rb\_tree$. Реализует и переопределяет методы:

$swap\_additional\_data$: обменивает значения $\_color$ двух узлов

$remove$: метод выполняет удаление rb_node из дерева, в зависимости от количества детей удаляемого узла. При наличии двух потомков – находим максимально правый элемент в левом поддереве, обмениваем его с удаляемым вплоть до цвета узлов, переходим к случаю с одним и отсутствием потомков. Согласно свойствам красно-черного дерева, не может быть красного узла с одним потомком, поэтому при его наличии вызывается метод $cleanup\_node$ для удаляемого узла и корректируется ссылка из родительского для удаляемого узла на узел-потомок, последний окрашивается в черный. Если потомков у удаляемого нет, вызывается метод $cleanup\_node$ для него, и, если удаляемый узел – черный, вызывается метод $after\_remove$.

$after\_remove$: находит родителя и деда узла. Пусть удаляемый узел был левым поддеревом. Случай, когда удаляемый узел – правое поддерево зеркален.

- Если брат – красный, то производит малый левый поворот относительно брата и родителя, перекрашивает родителя в красный, брата – в черный. Переходит к следующему шагу.

- Если брат текущей вершины черный, то получает три случая:
    - Оба ребёнка у брата чёрные. Перекрашивает брата в красный цвет, отца в черный.
    - Если у брата левое поддерево – красное, производит малый правый поворот относительно брата и его левого поддерева, перекрашивает брата в черный и его правого сына в красный. Переходит к следующему шагу.
    - Если у брата правое поддерево – красное, выполняет малый левый поворот относительно брата и отца, перекрашивает брата в цвет отца, правое поддерево брата и отца – в черный.

Описание примененных паттернов

“Строитель” – порождающий паттерн, порождающий объекты. Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления. При его использовании алгоритм создания сложного объекта не зависит от того, из каких частей состоит объект и как они стыкуются между собой; процесс конструирования обеспечивает разные представления конструируемого объекта. Паттерн позволяет изменять внутреннее представление продукта, изолирует код, реализующий конструирование и представление, дает более тонкий контроль над процессом конструирования.

“Приспособленец” – структурный паттерн. Использует разделение для эффективной поддержки множества мелких объектов. Используется, когда в приложении используется большое количество объектов, из-за чего возникают высокие накладные расходы на их хранение, большую часть состояния объектов можно вынести вовне, многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов и приложение не зависит от идентичности объекта. Таким образом, применив паттерн “Приспособленец”, получим значительную экономию памяти.

“Одиночка” – порождающий паттерн. Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. “Одиночка” предоставляет контролируемый доступ к единственному экземпляру, уменьшает числа имен, допускает уточнение операций и представления, переменное число экземпляров и дает большую гибкость, чем у операций класса.

“Команда” – поведенческий паттерн. Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций. “Команда” разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить. Из простых команд можно собирать более составные, добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

“Цепочка обязанностей” – поведенческий паттерн. Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают. “Цепочка обязанностей” позволяет ослабить связанности, дает дополнительную гибкость при распределении обязанностей между объектами.

“Шаблонный метод” – поведенческий паттерн, определяющий основу алгоритма и позволяющий подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

“Итератор” – поведенческий паттерн. Предоставляет способ последовательного доступа ко всем элементам составного класса, не раскрывая его внутреннего представления. “Итератор” позволяет поддерживать различные виды обхода агрегата, упрощает интерфейс агрегата.

Руководство пользователя

После выполнения команд из файла, путь к которому подан на вход программе как аргумент командной строки, пользователю будет предложен интерактивный диалог с приложением, изложена справочная информация по программе.

Листинг 20. Справочная информация о программе

```
Hello! Would you like to get some closer interaction with my program? Print y for yes an
n for no
>>y
----- Course work help -----
Collection commands list:
Full path to collection: <pool_name>/<scheme_name>/<collection_name>
        - add
        - find
        - find dataset
        - find DD/MM/YYYY hh/mm/ss
        - update
        - delete
Structural and customization commands list:
Supported trees: BST (binary tree), AVL, SPLAY, RB (red-black tree)
Supported allocators: global, sorted_list best||worst||first, descriptors
best||worst||first, buddy_system
        - add <tree type> <allocator type> <size of allocator> <full path>
                one can note tree/allocator type only
                if one wants to note an allocator, it's needed to also node desired
length in bytes (not for global one)

        - delete <full path>
DB commands list:
        - help
        - delete DB
        - exit
```

Для всего пользовательского ввода одинаковы:

- формат полного пути к пулу/схеме/коллекции: <pool_name>/<scheme_name>/<collection_name>. При добавлении/удалении пула полный путь – <pool_name>, схемы – <pool_name>/<scheme_name>
- формат записи ключа: <applicant id>, <contest id>
- формат записи нового значения:
    - Фамилия Имя Отчество
    - дата рождения
    - ссылка на резюме соискателя
    - id HR-менеджера
    - язык программирования на котором реализуются задачи контеста
    - кол-во задач в контесте
    - кол-во решённых задач в контесте
    - было ли обнаружено списывание с микронаушником

Рассмотрим использование команд, взаимодействующих с коллекцией.

add. Пользователю будет предложено ввести ключ и значение, полный путь к коллекции, куда пользователь хочет добавить пару ключ-значение.

```
>>add
Enter key:
applicant id, contest id: >>7, 1
Enter value:
Surname Name Patronymic:              >>Solomatina Svetlana Viktorovna
Birthday <dd/mm/year>:                >>13/11/2003
Link to resume:                       >>
link.com
id of HR-manager:                     >>1
Programming language:                 >>c++
Task count: {whole number}            >>
21
Solved tasks count {whole number}:    >>9
Copying with a micro-earphone {bool}: >>false
Enter full path to collection: >>p1/s1/c1
Added a
 value to collection c1 successfully!
```

Рисунок 1. Добавление записи в $data\_base$

find. Пользователю будет предложено ввести ключ, по которому будет искаться значение, и полный путь к коллекции. Если значение по этому ключу найдено в этой коллекции, в консоль выводятся данные о записи, в том числе время создания записи в миллисекундах.

```
>>find
Enter key:
applicant id, contest id: >>0, 1
Enter full path to collection: >>p1/s1/c1
Volkova Anastasia Nikolaevna
15/07/2001
bs.com
8
c#
11
3
0
1685431313664

>>
```

Рисунок 2. Поиск записи в $data\_base$

find dataset. Пользователю будет предложено ввести ключи min_key и max_key, образующие диапазон значений $[min\_key, max\_key]$, и полный путь к коллекции. Будет производиться поиск ключей, удовлетворяющих этому диапазону в этой коллекции. Если такие ключи найдены, в консоль выводятся данные о соответствующих записях.

```
>>find dataset
Enter min and max keys
Enter key:
applicant id, contest id: >>1, 0
Enter key:
applicant id, contest id: >>2, 1
Enter full path to collection: >>p1/s1/c1
----- 1 value -----
Petrov Ilya Andreevich
```

Рисунок 3. Поиск набора записей в *data_base*

find DD/MM/YYYY hh/mm/ss. Пользователю будет предложено ввести ключ и полный путь к коллекции. Если значение по ключу найдено в этой коллекции, в консоль выводятся данные о значении этой записи на заданную дату.

```
>>find 30/05/2023 13/37/50
Enter key:
applicant id, contest id: >>0, 1
Enter full path to collection: >>p1/s1/c1
Ivanova Yana Vladimirovna
15/07/2023
```

Рисунок 4. Поиск записи по времени в *data_base*

update. Пользователю будет предложено ввести ключ и пары <имя поля>: <значение поля>, ввести полный путь к коллекции. На консоль будет выведено сообщение об удаче/неудаче выполнения команды.

```
>>update
Enter key:
applicant id, contest id: >>0, 1
Print field_name: new_value
Fields: surname, name, patronymic, birthday, link_to_resume, hr_id, p_language, tasks, solved, copying
To stop print exit
surname: Solomatina
link_to_resume: new_link.com
birthday: 13/11/2003
exit
Enter full path to collection: >>p1/s1/c1
Updated a value in collection c1 successfully!
>>
```

Рисунок 5. Обновление значения записи в *data_base*

delete/remove. Пользователю будет предложено ввести ключ и полный путь к коллекции. На консоль будет выведено сообщение об удаче/неудаче выполнения команды.

```
>>remove
Enter key:
applicant id, contest id: >>0, 1
Enter full path to collection: >>p1/s1/c1
Deleted value from collection c1 successfully!
```

Рисунок 6. Удаление записи из *data_base*

add <tree type> <allocator type> <size of allocator> <full path>. На консоль будет выведено сообщение об удаче/неудаче выполнения команды.

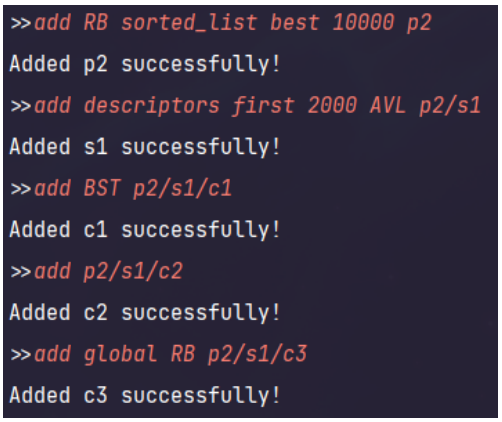

Рисунок 7. Добавление структуры в $data\_base$

delete <full path>. На консоль будет выведено сообщение об удаче/неудаче выполнения команды.

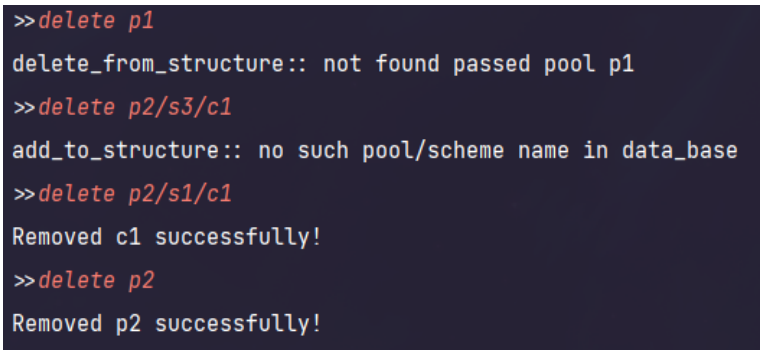

Рисунок 8. Удаление структуры из $data\_base$

help. Вызывает справку о программе.

delete DB. Удаляет данные в объекте класса $data\_base$.

exit. Удаляет объект класса $data\_base$ и выходит из программы.

Вывод

В результате выполнения курсовой работы было реализовано приложение, позволяющее выполнять операции над коллекциями данных заданных типов и контекстами их хранения. Реализован класс $data\_base$, репрезентирующий многоуровневую структуру, состоящую из пулов, схем, коллекций. Выполнено основное задание курсовой работы [0] и дополнительные задания [1], [2], [4], [6] (без реализации B и B^+ деревьев), [7]. todo: дополнить вывод

Приложение

Раздел 1. Класс *data_base*.

```
#ifndef DATA_BASE_H
#define DATA_BASE_H

#include <iostream>
#include <tuple>
#include <vector>

#include "../allocator/memory_holder.h"
#include "../allocator_from_global_heap/memory_from_global_heap.h"
#include
"../allocator_with_sorted_list_deallocation/memory_with_sorted_list_deallocation.h"
#include "../allocator_with_boundary_tags_deallocation/memory_with_boundary_tags.h"
#include "../allocator_with_buddy_system/memory_with_buddy_system.h"

#include "../logger/logger_holder.h"

#include "../binary_tree/associative_container.h"
#include "../binary_tree/bs_tree.h"
#include "../avl_tree/avl_tree.h"
#include "../splay_tree/splay_tree.h"
#include "../rb_tree/rb_tree.h"

#include "../chain_of_resp_and_command/handler.h"
#include "../db_key/key.h"

class data_base final :
       private memory_holder,
       private logger_holder
{
private:
   class string_comparer
   {
   public:
       int operator()(std::string const & x, std::string const & y) {
           return x.compare(y);
       }
   };

private:
   splay_tree<std::string,
       associative_container<std::string,
           associative_container<std::string,
               associative_container<key, db_value *> *
                                   > *
                               > *, string_comparer
                           > * _database;
   logger * _logger;
   memory* _allocator;
   std::map<std::string, memory *> _all_trees_allocators;

public:
   typedef enum trees_types {
       BST,
       AVL,
       SPLAY,
       RB,
       not_a_tree
```

```
    } trees_types_;

    typedef enum allocator_types {
        global,
        for_inner_use_sorted_list,
        sorted_list_best,
        sorted_list_worst,
        sorted_list_first,
        for_inner_use_descriptors,
        descriptors_best,
        descriptors_worst,
        descriptors_first,
        buddy_system,
        not_an_allocator
    } allocator_types_;

#pragma region exceptions
public:
    class db_insert_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit db_insert_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

    class db_find_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit db_find_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

    class db_remove_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit db_remove_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };
#pragma endregion
```

```
#pragma region Find structure
private:
   [[nodiscard]] associative_container<std::string, associative_container<std::string,
associative_container<key, db_value *> *> *> *
   find_data_pool
   (std::string const & pool_name) const;

   [[nodiscard]] associative_container<std::string, associative_container<key, db_value
*> *> *
   find_data_scheme
   (std::string const & pool_name, std::string const & scheme_name) const;

   [[nodiscard]] associative_container<key, db_value *> *
   find_data_collection
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name) const;
#pragma endregion

#pragma region Collection-related functions
#pragma region Insertion in collection
public:
   void add_to_collection
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    const key& _key, db_value * value) const;

#pragma endregion

#pragma region Updating a collection value
public:
   void update_in_collection
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    const key& _key, std::map<db_value_fields, unsigned char *> upd_dict) const;
#pragma endregion

#pragma region Finding among collection
public:
   [[nodiscard]] db_value * find_among_collection
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    const key& _key) const;

   [[nodiscard]] db_value * find_with_time
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    const key& _key, uint64_t time_parameter) const;

   [[nodiscard]] std::vector<db_value *> find_in_range
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    key min_key, key max_key) const;
#pragma endregion

#pragma region Deletion from collection
public:
   void delete_from_collection
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    const key& key) const;
#pragma endregion
#pragma endregion
```

```
#pragma region Structure functions
#pragma region Inserting in structure of data base
   // adding a collection: no string is empty
   // adding a scheme: collection string is empty
private:
   memory *
   get_new_allocator_for_inner_trees
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    allocator_types_ allocator_type, size_t allocator_pool_size);

public:
   void add_to_structure
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name,
    trees_types_ tree_type, allocator_types_ allocator_type, size_t
allocator_pool_size);
#pragma endregion

#pragma region Deletion from structure of data base
private:
   void delete_collection
           (const std::string & full_path, const std::string & collection_name,
associative_container<std::string,
                   associative_container<key, db_value *> *> * parent_scheme);

   void delete_scheme
           (const std::string & full_path, const std::string & scheme_name,
associative_container<std::string,
                   associative_container<std::string, associative_container<key,
db_value *> *> *> * parent_pool);

   void delete_pool(const std::string & pool_name);

public:
   void delete_from_structure
   (std::string const & pool_name, std::string const & scheme_name, std::string const &
collection_name);
#pragma endregion
#pragma endregion

#pragma region logger_holder and memory_holder contract
   [[nodiscard]] logger *get_logger() const noexcept override
   {
       return this->_logger;
   }

   [[nodiscard]] memory *get_memory() const noexcept override
   {
       return this->_allocator;
   }
#pragma endregion

#pragma region rool 5
public:
   explicit data_base(logger * this_db_logger = nullptr, memory * this_db_allocator =
nullptr)
   : _logger(this_db_logger), _allocator(this_db_allocator)
   {
       _database = new splay_tree<std::string,
                                  associative_container<std::string,
                                      associative_container<std::string,
```

```
                                                associative_container<key, db_value *> *> *>
*
                                                                                        ,
string_comparer>(this_db_logger, this_db_allocator);
   }

   ~data_base() override;

   // copy constructor
   data_base(data_base const &obj) = delete;

   // move constructor
   data_base(data_base &&obj) noexcept = delete;

   // copy assignment (оператор присваивания)
   data_base &operator=(data_base const &obj) = delete;

   // move assignment (оператор присваивания перемещением)
   data_base &operator=(data_base &&obj) noexcept = delete;

#pragma endregion
};

#endif //DATA_BASE_H
```

Реализация методов класса *data_base*.

```
   #include "data_base.h"

   #pragma region Find structure

   associative_container<std::string, associative_container<std::string,
      associative_container<key, db_value *> *> *> *
   data_base::find_data_pool(const std::string &pool_name) const
   {
      this->trace_with_guard("data_base::find_data_pool method started");
      if (pool_name.empty()) {
          this->warning_with_guard("data_base::find_data_pool pool name must not be an
      empty string")
                ->trace_with_guard("data_base::find_data_pool method finished");
          throw data_base::db_find_exception("find_data_pool:: pool name must not be an
      empty string");
      }

      associative_container<std::string,
              associative_container<std::string,
                      associative_container<key, db_value *> *> *> *data_pool;

      try {
          data_pool = this->_database->get(pool_name);
      }
      catch (bs_tree<std::string,
              associative_container<std::string,
                  associative_container<std::string,
```

```
                    associative_container<key, db_value *> *> *> *,
    data_base::string_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::find_data_pool data pool not found")
                ->trace_with_guard("data_base::find_data_pool method finished");
        throw data_base::db_find_exception("find_data_pool:: data pool not found");
    }

    this->trace_with_guard("data_base::find_data_pool method finished");
    return data_pool;
}

associative_container<std::string, associative_container<key, db_value *> *> *
data_base::find_data_scheme(const std::string &pool_name, const std::string
    &scheme_name) const
{
    this->trace_with_guard("data_base::find_data_scheme method started");
    if (scheme_name.empty()) {
        this->warning_with_guard("data_base::find_data_scheme scheme name must not be
    an empty string")
                ->trace_with_guard("data_base::find_data_scheme method finished");
        throw data_base::db_find_exception("find_data_scheme:: scheme name must not be
    an empty string");
    }

    associative_container<std::string,
            associative_container<std::string,
                    associative_container<key, db_value *> *> *> *data_pool =
    find_data_pool(pool_name);

    associative_container<std::string,
            associative_container<key, db_value *> *> *data_scheme;

    try {
        data_scheme = data_pool->get(scheme_name);
    }
    catch (typename bs_tree<std::string,
            associative_container<std::string,
                associative_container<key, db_value *> *> *,
    data_base::string_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::find_data_scheme data scheme not found")
                ->trace_with_guard("data_base::find_data_scheme method finished");
        throw data_base::db_find_exception("find_data_scheme:: data scheme not
    found");
    }

    this->trace_with_guard("data_base::find_data_scheme method finished");
    return data_scheme;
}

associative_container<key, db_value *> *
data_base::find_data_collection(const std::string &pool_name,
                                const std::string &scheme_name,
                                const std::string &collection_name) const
{
    this->trace_with_guard("data_base::find_data_collection method started");
```

```
    if (collection_name.empty()) {
        this->warning_with_guard("data_base::find_data_collection collection name must
    not be an empty string")
                ->trace_with_guard("data_base::find_data_collection method finished");
        throw data_base::db_find_exception("find_data_collection:: collection name
    must not be an empty string");
    }

    associative_container<std::string,
            associative_container<key, db_value *> *> *data_scheme =
    find_data_scheme(pool_name, scheme_name);

    associative_container<key, db_value *> *data_collection;

    try {
        data_collection = data_scheme->get(collection_name);
    }
    catch (typename bs_tree<std::string, associative_container<key, db_value *> *,
    data_base::string_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::find_data_collection data collection not
    found")
                ->trace_with_guard("data_base::find_data_collection method finished");
        throw data_base::db_find_exception("find_data_collection:: data collection not
    found");
    }

    this->trace_with_guard("data_base::find_data_collection method finished");
    return data_collection;
}

#pragma endregion

#pragma region Collection-related functions
#pragma region Insertion in collection

void
data_base::add_to_collection(const std::string &pool_name,
                             const std::string &scheme_name,
                             const std::string &collection_name, const key& _key,
                             db_value * value) const
{
    this->trace_with_guard("data_base::add_to_collection method started");
    associative_container<key, db_value *> *data_collection
            = find_data_collection(pool_name, scheme_name, collection_name);
    // handle if not found ?
    db_value * found_value;
    try {
        found_value = data_collection->get(_key);
        handler *add_handle = new add_handler(value);
        handler *last_handler = found_value->get_last_handler();
        if (last_handler == nullptr) {
            // there are no commands added
            this->warning_with_guard("data_base::add_to_collection add handler cannot
    be the first one in chain of responsibility")
```

```
                    ->trace_with_guard("data_base::add_to_collection method
    finished");
            throw handler::order_exception("add_to_collection:: add handler cannot be
    the first one in chain of responsibility");
        } else {
            // add command may be only after remove command
            if (last_handler->get_handler_type() ==
    handler::handler_types::_remove_handler_) {
                found_value->add_new_handler(add_handle);
            } else {
                this->warning_with_guard("data_base::add_to_collection add handler can
    be only after remove_handler in chain of responsibility")
                        ->trace_with_guard("data_base::add_to_collection method
    finished");
                throw handler::order_exception("add_to_collection:: add handler can be
    only after remove_handler in chain of responsibility");
            }
        }
    }
    catch (typename bs_tree<key, db_value *, key_comparer>::find_exception const &) {
        data_collection->insert(_key, std::move(value));
    }
    this->trace_with_guard("data_base::add_to_collection method finished");
}

#pragma endregion

#pragma region Updating a collection value
void
data_base::update_in_collection(const std::string &pool_name, const std::string
    &scheme_name,
                                const std::string &collection_name, const key& _key,
                                std::map<db_value_fields, unsigned char *> upd_dict)
    const
{
    this->trace_with_guard("data_base::update_in_collection method started");
    associative_container<key, db_value *> *data_collection
            = find_data_collection(pool_name, scheme_name, collection_name);
    // handle if not found ?
    db_value * found_value;
    try {
        found_value = data_collection->get(_key);
        handler * update_handle = new update_handler(std::move(upd_dict));
        handler * last_handler = found_value->get_last_handler();
        if (last_handler == nullptr) {
            found_value->add_new_handler(update_handle);
        } else {
            if (last_handler->get_handler_type() !=
    handler::handler_types::_remove_handler_) {
                found_value->add_new_handler(update_handle);
            } else {
                this->warning_with_guard("data_base::update_in_collection update
    handler cannot be after remove_handler in chain of responsibility")
                        ->trace_with_guard("data_base::update_in_collection method
    finished");
```

```
                throw handler::order_exception("update_in_collection:: update handler
    cannot be after remove_handler in chain of responsibility");
            }
        }
    }
    catch (typename bs_tree<key, db_value *, key_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::update_in_collection no element with passed
    _key found")
                ->trace_with_guard("data_base::update_in_collection method finished");
        throw data_base::db_find_exception("update_in_collection:: no element with
    passed _key found");
    }

    this->trace_with_guard("data_base::update_in_collection method finished");
}

#pragma endregion

#pragma region Finding among collection
db_value *
data_base::find_among_collection(const std::string &pool_name, const std::string
    &scheme_name,
                                 const std::string &collection_name, const key& _key)
    const
{
    this->trace_with_guard("data_base::find_among_collection method started");
    associative_container<key, db_value *> *data_collection
            = find_data_collection(pool_name, scheme_name, collection_name);

    db_value * found_value;
    try {
        found_value = data_collection->get(_key);
    }
    catch (typename bs_tree<key, db_value *, key_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::find_among_collection no value with passed
    _key found")
                ->trace_with_guard("data_base::find_among_collection method
    finished");
        throw data_base::db_find_exception("find_among_collection:: no value with
    passed _key found");
    }

    this->trace_with_guard("data_base::find_among_collection method finished");
    return found_value;
}

db_value *
data_base::find_with_time
(const std::string &pool_name, const std::string &scheme_name, const std::string
    &collection_name,
const key& _key, uint64_t time_parameter) const
{
    this->trace_with_guard("data_base::find_with_time method started");
    associative_container<key, db_value *> * collection =
    find_data_collection(pool_name, scheme_name, collection_name);
```

```
    db_value * found_value;
    try {
        found_value = collection->get(_key);
    }
    catch (typename bs_tree<key, db_value *, key_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::find_with_time cannot find a value with
    passed key")
                ->trace_with_guard("data_base::find_with_time method finished");
        throw data_base::db_find_exception("find_with_time:: cannot find a value with
    passed key");
    }

    db_value * found_value_copy = found_value->make_a_copy();
    handler * first_handler = found_value->get_first_handler();

    if (first_handler == nullptr) {
        return found_value_copy;
    }

    first_handler->handle(&found_value_copy, time_parameter);

    this->trace_with_guard("data_base::find_with_time method finished");
    return found_value_copy;
}

std::vector<db_value *>
data_base::find_in_range(const std::string &pool_name, const std::string
    &scheme_name,
                         const std::string &collection_name, key min_key, key max_key)
    const
{
    this->trace_with_guard("data_base::find_in_range method started");
    associative_container<key, db_value *> *data_collection
            = find_data_collection(pool_name, scheme_name, collection_name);

    key_comparer comparer;
    if (comparer(min_key, max_key) > 0) {
        key tmp = min_key;
        min_key = max_key;
        max_key = tmp;
    }
    // todo: if comparer(min_key, max_key) == 0, do simple finding

    std::vector<db_value *> to_return_vector;

    // done only for bst-like trees. B and B+ trees require another iterators
    auto * bst = reinterpret_cast<bs_tree<key, db_value *, key_comparer>
    *>(data_collection);
    auto end_iteration = bst->end_infix();
    bool in_range = false;

    for (auto it = bst->begin_infix(); it != end_iteration; ++it) {
        if (in_range) {
            if (comparer(max_key, std::get<1>(*it)) >= 0) {
```

```
                to_return_vector.push_back(std::get<2>(*it));
            } else {
                break;
            }
        }

        if (!in_range) {
            if (comparer(min_key, std::get<1>(*it)) <= 0) {
                in_range = true;
                to_return_vector.push_back(std::get<2>(*it));
            }
        }
    }

    this->trace_with_guard("data_base::find_in_range method finished");
    return to_return_vector;
}

#pragma endregion

#pragma region Deletion from collection
void
data_base::delete_from_collection(const std::string &pool_name, const std::string
    &scheme_name,
                                  const std::string &collection_name, const key& _key)
    const
{
    this->trace_with_guard("data_base::delete_from_collection method started");
    associative_container<key, db_value *> *data_collection
            = find_data_collection(pool_name, scheme_name, collection_name);

    try {
        db_value * found_value = data_collection->get(_key);
        handler * delete_handle = new remove_handler();
        handler * last_handler = found_value->get_last_handler();
        if (last_handler == nullptr) {
            found_value->add_new_handler(delete_handle);
        } else {
            if (last_handler->get_handler_type() !=
    handler::handler_types::_remove_handler_) {
                found_value->add_new_handler(delete_handle);
            } else {
                this->warning_with_guard("data_base::delete_from_collection delete
    handler cannot be after delete handler in chain of responsibility")
                        ->trace_with_guard("data_base::delete_from_collection method
    finished");
                throw handler::order_exception("update_in_collection:: delete handler
    cannot be after delete handler in chain of responsibility");
            }
        }
    }
    catch (typename bs_tree<key, db_value *, key_comparer>::find_exception const &) {
        this->debug_with_guard("data_base::delete_from_collection no element with
    passed _key found")
```

```
                ->trace_with_guard("data_base::delete_from_collection method
    finished");
        throw data_base::db_find_exception("delete_from_collection:: no element with
    passed _key found");
    }
    this->trace_with_guard("data_base::delete_from_collection method finished");
}

#pragma endregion

#pragma endregion

#pragma region Structure functions
#pragma region Inserting in structure of data base

memory *
data_base::get_new_allocator_for_inner_trees
        (std::string const & pool_name, std::string const & scheme_name, std::string
    const &collection_name,
         data_base::allocator_types_ allocator_type, size_t allocator_pool_size)
{
    this->trace_with_guard("data_base::get_new_allocator_for_inner_trees method
    started");
    memory * new_allocator = nullptr;
    switch (allocator_type) {
        case data_base::allocator_types_ ::global:
            new_allocator = new memory_from_global_heap(this->get_logger());
            break;
        case data_base::allocator_types_ ::sorted_list_best:
            new_allocator = new
    memory_with_sorted_list_deallocation(allocator_pool_size,
    memory::Allocation_strategy::best_fit, this->get_logger(), nullptr);
            break;
        case data_base::allocator_types_ ::sorted_list_worst:
            new_allocator = new
    memory_with_sorted_list_deallocation(allocator_pool_size,
    memory::Allocation_strategy::worst_fit, this->get_logger(), nullptr);
            break;
        case data_base::allocator_types_ ::sorted_list_first:
            new_allocator = new
    memory_with_sorted_list_deallocation(allocator_pool_size,
    memory::Allocation_strategy::first_fit, this->get_logger(), nullptr);
            break;
        case data_base::allocator_types_ ::descriptors_best:
            new_allocator = new memory_with_boundary_tags(allocator_pool_size,
    memory::Allocation_strategy::best_fit, this->get_logger(), nullptr);
            break;
        case data_base::allocator_types_ ::descriptors_worst:
            new_allocator = new memory_with_boundary_tags(allocator_pool_size,
    memory::Allocation_strategy::worst_fit, this->get_logger(), nullptr);
            break;
        case data_base::allocator_types_ ::descriptors_first:
            new_allocator = new memory_with_boundary_tags(allocator_pool_size,
    memory::Allocation_strategy::first_fit, this->get_logger(), nullptr);
            break;
```

```
        case data_base::allocator_types_ ::buddy_system:
            new_allocator = new
    memory_with_buddy_system((char)log2((double)allocator_pool_size),
    this->get_logger(), nullptr);
            break;
        default:
            break;
    }
    if (new_allocator != nullptr) {
        _all_trees_allocators[pool_name + "/" + scheme_name + "/" + collection_name] =
    new_allocator;
    }

    this->trace_with_guard("data_base::get_new_allocator_for_inner_trees method
    finished");
    return new_allocator;
}

void
data_base::add_to_structure(const std::string &pool_name, const std::string
    &scheme_name,
                            const std::string &collection_name,
                            data_base::trees_types_ tree_type,
                            data_base::allocator_types_ allocator_type,
                            size_t allocator_pool_size)
{
    this->trace_with_guard("data_base::add_to_structure method started");
    if (pool_name.empty()) {
        this->warning_with_guard("data_base::add_to_structure one should pass pool
    name for correct work of method")
                ->trace_with_guard("data_base::add_to_structure method finished");
        throw data_base::db_insert_exception("add_to_structure:: one should pass pool
    name for correct work of method");
    }

    memory * new_allocator = get_new_allocator_for_inner_trees(pool_name, scheme_name,
    collection_name, allocator_type, allocator_pool_size);

    // insert data pool in _database
    if (scheme_name.empty()) {
        associative_container<std::string,
                associative_container<std::string,
                        associative_container<key, db_value *> *> *> *data_pool =
    nullptr;

        switch (tree_type) {
            case data_base::trees_types_::BST:
                data_pool = new bs_tree<std::string,
                        associative_container<std::string,
                                associative_container<key, db_value *> *> *,
    string_comparer>(this->get_logger(), new_allocator);
                break;
            case data_base::trees_types_::AVL:
                data_pool = new avl_tree<std::string,
                        associative_container<std::string,
```

```
                            associative_container<key, db_value *> *> *,
string_comparer>(this->get_logger(), new_allocator);
            break;
        case data_base::trees_types_::RB:
            data_pool = new rb_tree<std::string,
                    associative_container<std::string,
                            associative_container<key, db_value *> *> *,
string_comparer>(this->get_logger(), new_allocator);
            break;
        default:
            data_pool = new splay_tree<std::string,
                    associative_container<std::string,
                            associative_container<key, db_value *> *> *,
string_comparer>(this->get_logger(), new_allocator);
            break;
    }

    try {
        _database->insert(pool_name, std::move(data_pool));
    }
    catch (typename bs_tree<std::string,
            associative_container<std::string,
                associative_container<std::string,
                    associative_container<key, db_value *> *> *> *,
string_comparer>::insert_exception const &) {
        delete data_pool; // ?? todo: check
        _all_trees_allocators.erase(pool_name + "/" + scheme_name + "/" +
collection_name);
        delete new_allocator;

        this->warning_with_guard("data_base::add_to_structure insert failed due to
non-unique pool name to insert")
                ->trace_with_guard("data_base::add_to_structure method finished");
        throw data_base::db_insert_exception("add_to_structure:: insert failed due
to non-unique pool name to insert");
    }
}
    // insert data scheme
else if (collection_name.empty()) {
    // find a pool
    associative_container<std::string,
            associative_container<std::string,
                    associative_container<key, db_value *> *> *> *data_pool;
    try {
        data_pool = find_data_pool(pool_name);
    }
    catch (data_base::db_find_exception const &) {
        this->debug_with_guard("data_base::add_to_structure no such pool name in
data_base")
                ->trace_with_guard("data_base::add_to_structure method finished");
        throw data_base::db_insert_exception("add_to_structure:: no such pool name
in data_base");
    }

    associative_container<std::string,
```

```
            associative_container<key, db_value *> *> * data_scheme = nullptr;

        switch (tree_type) {
            case data_base::trees_types_::BST:
                data_scheme = new bs_tree<std::string,
                                    associative_container<key, db_value *> *,
                                        string_comparer>(this->get_logger(),
    new_allocator);
                break;
            case data_base::trees_types_::AVL:
                data_scheme = new avl_tree<std::string,
                        associative_container<key, db_value *> *,
                        string_comparer>(this->get_logger(), new_allocator);
                break;
            case data_base::trees_types_::RB:
                data_scheme = new rb_tree<std::string,
                        associative_container<key, db_value *> *,
                        string_comparer>(this->get_logger(), new_allocator);
                break;
            default:
                data_scheme = new splay_tree<std::string,
                        associative_container<key, db_value *> *,
                        string_comparer>(this->get_logger(), new_allocator);
                break;
        }

        try {
            data_pool->insert(scheme_name, std::move(data_scheme));
        }
        catch (typename bs_tree<key, db_value *, key_comparer>::insert_exception const
    &) {
            delete data_scheme;
            _all_trees_allocators.erase(pool_name + "/" + scheme_name + "/" +
    collection_name);
            delete new_allocator;

            this->warning_with_guard("data_base::add_to_structure insert failed due to
    non-unique scheme name to insert")
                    ->trace_with_guard("data_base::add_to_structure method finished");
            throw data_base::db_insert_exception(
                    "add_to_structure:: insert failed due to non-unique scheme name to
    insert");
        }
    }
        // insert collection
    else {
        // find a scheme
        associative_container<std::string,
                associative_container<key, db_value *> *> * data_scheme;

        try {
            data_scheme = find_data_scheme(pool_name, scheme_name);
        }
        catch (data_base::db_find_exception const &) {
```

```
            _all_trees_allocators.erase(pool_name + "/" + scheme_name + "/" +
    collection_name);
            delete new_allocator;

            this->debug_with_guard("data_base::add_to_structure no such pool/scheme
    name in data_base")
                    ->trace_with_guard("data_base::add_to_structure method finished");
            throw data_base::db_insert_exception("add_to_structure:: no such
    pool/scheme name in data_base");
        }

        associative_container<key, db_value *> * data_collection = nullptr;

        switch (tree_type) {
            case data_base::trees_types_::BST:
                data_collection = new bs_tree<key, db_value *,
    key_comparer>(this->get_logger(), new_allocator);
                break;
            case data_base::trees_types_::AVL:
                data_collection = new avl_tree<key, db_value *,
    key_comparer>(this->get_logger(), new_allocator);
                break;
            case data_base::trees_types_::RB:
                data_collection = new rb_tree<key, db_value *,
    key_comparer>(this->get_logger(), new_allocator);
                break;
            default:
                data_collection = new splay_tree<key, db_value *,
    key_comparer>(this->get_logger(), new_allocator);
                break;
        }

        try {
            data_scheme->insert(collection_name, std::move(data_collection));
        }
        catch (typename bs_tree<key, db_value *, key_comparer>::insert_exception const
    &) {
            delete data_collection; // ?? todo: check
            _all_trees_allocators.erase(pool_name + "/" + scheme_name + "/" +
    collection_name);
            delete new_allocator;

            this->warning_with_guard("data_base::add_to_structure insert failed due to
    non-unique collection name to insert")
                    ->trace_with_guard("data_base::add_to_structure method finished");
            throw data_base::db_insert_exception(
                    "add_to_structure:: insert failed due to non-unique collection
    name to insert");
        }
    }
    this->trace_with_guard("data_base::add_to_structure method finished");
}

#pragma endregion
```

```
#pragma region Deleting from structure of data base

void data_base::delete_pool(const std::string & pool_name)
{
   associative_container<std::string,
           associative_container<std::string,
               associative_container<key, db_value *> *> *> * data_pool =
   _database->remove(pool_name);

   std::vector<memory *> sub_structures_allocators;

   auto * pool_to_d = reinterpret_cast<bs_tree<std::string,
                                              associative_container<std::string,
                                                  associative_container<key, db_value *>
   *> *, string_comparer> *>(data_pool);

   std::string sub_struct_full_name;

   auto iter_over_schemes_end = pool_to_d->end_postfix();
   for (auto iter_over_schemes = pool_to_d->begin_postfix(); iter_over_schemes !=
   iter_over_schemes_end; ++iter_over_schemes) {
       auto * sub_scheme = reinterpret_cast<bs_tree<std::string,
               associative_container<key, db_value *> *, string_comparer>
   *>(std::get<2>(*iter_over_schemes));

       auto iter_over_collections_end = sub_scheme->end_postfix();
       for (auto iter_over_collections = sub_scheme->begin_postfix();
   iter_over_collections != iter_over_collections_end; ++iter_over_collections) {
           sub_struct_full_name = pool_name + "/" + std::get<1>(*iter_over_schemes) +
   "/" + std::get<1>(*iter_over_collections);
           if (_all_trees_allocators.contains(sub_struct_full_name)) {

   sub_structures_allocators.push_back(_all_trees_allocators[sub_struct_full_name]);
               _all_trees_allocators.erase(sub_struct_full_name);
           }
       }

       sub_struct_full_name = pool_name + "/" + std::get<1>(*iter_over_schemes) +
   "/";
       if (_all_trees_allocators.contains(sub_struct_full_name)) {

   sub_structures_allocators.push_back(_all_trees_allocators[sub_struct_full_name]);
           _all_trees_allocators.erase(sub_struct_full_name);
       }
   }

   delete data_pool;

   size_t size_of_vector = sub_structures_allocators.size();

   unsigned i;
   for (i = 0; i < size_of_vector; i++) {
       delete sub_structures_allocators[i];
   }
```

```
    sub_struct_full_name = pool_name + "/" + "/";
    if (_all_trees_allocators.contains(sub_struct_full_name)) {
        memory * tmp = _all_trees_allocators[sub_struct_full_name];
        _all_trees_allocators.erase(sub_struct_full_name);
        delete tmp;
    }
}

// full path must include pool/scheme/
void data_base::delete_scheme(const std::string & full_path, const std::string &
    scheme_name, associative_container<std::string,
                              associative_container<std::string,
    associative_container<key, db_value *> *> *> * parent_pool)
{
    associative_container<std::string,
            associative_container<key, db_value *> *> * data_scheme =
    parent_pool->remove(scheme_name);

    std::vector<memory *> sub_structures_allocators;

    auto * scheme_to_d = reinterpret_cast<bs_tree<std::string,
            associative_container<key, db_value *> *, string_comparer>
    *>(data_scheme);

    std::string collection_full_name;

    auto iter_end = scheme_to_d->end_postfix();
    for (auto iter = scheme_to_d->begin_postfix(); iter != iter_end; ++iter) {
        collection_full_name = full_path + std::get<1>(*iter);
        if (_all_trees_allocators.contains(collection_full_name)) {

    sub_structures_allocators.push_back(_all_trees_allocators[collection_full_name]);
            _all_trees_allocators.erase(collection_full_name);
        }
    }

    delete data_scheme;

    size_t size_of_vector = sub_structures_allocators.size();

    unsigned i;
    for (i = 0; i < size_of_vector; i++) {
        delete sub_structures_allocators[i];
    }

    if (_all_trees_allocators.contains(full_path)) {
        memory * tmp = _all_trees_allocators[full_path];
        _all_trees_allocators.erase(full_path);
        delete tmp;
    }
}

void data_base::delete_collection(const std::string & full_path, const std::string &
    collection_name, associative_container<std::string,
        associative_container<key, db_value *> *> * parent_scheme)
```

```
{
    associative_container<key, db_value *> * data_collection =
    parent_scheme->remove(collection_name);

    delete data_collection;

    if (_all_trees_allocators.contains(full_path)) {
        memory * this_collection_allocator = _all_trees_allocators[full_path];
        delete this_collection_allocator;
        _all_trees_allocators.erase(full_path);
    }
}

void
data_base::delete_from_structure(const std::string &pool_name, const std::string
    &scheme_name,
                                 const std::string &collection_name)
{
    this->trace_with_guard("data_base::delete_from_structure method started");
    if (pool_name.empty()) {
        this->warning_with_guard("data_base::delete_from_structure one should pass
    pool name for correct work of method")
                ->trace_with_guard("data_base::delete_from_structure method
    finished");
        throw data_base::db_remove_exception(
                "delete_from_structure:: one should pass pool name for correct work of
    method");
    }

    // delete pool. should delete all schemes' collections' allocators, delete all
    schemes' allocators, delete pool's allocator
    if (scheme_name.empty()) {
        try {
            delete_pool(pool_name);
        }
        catch (bs_tree<std::string,
                    associative_container<std::string,
                        associative_container<std::string,
                            associative_container<key, db_value *> *> *> *,
    data_base::string_comparer>::remove_exception const &) {
            this->debug_with_guard("data_base::delete_from_structure not found passed
    pool " + pool_name)
                    ->trace_with_guard("data_base::delete_from_structure method
    finished");
            throw db_remove_exception("delete_from_structure:: not found passed pool "
    + pool_name);
        }
    }
    // deleting scheme
    else if (collection_name.empty()) {
        associative_container<std::string,
                associative_container<std::string,
                        associative_container<key, db_value *> *> *> *data_pool;
        try {
            data_pool = find_data_pool(pool_name);
```

```
    }
    catch (data_base::db_find_exception const &) {
        this->debug_with_guard("data_base::delete_from_structure no such pool name
in data_base")
                ->trace_with_guard("data_base::delete_from_structure method
finished");
        throw data_base::db_remove_exception("delete_from_structure:: no such pool
name in data_base");
    }

    try {
        delete_scheme(pool_name + "/" + scheme_name + "/", scheme_name,
data_pool);
    }
    catch (typename bs_tree<std::string,
                        associative_container<std::string,
                            associative_container<key, db_value *> *> *,
string_comparer>::remove_exception const &) {
        this->debug_with_guard("data_base::delete_from_structure no scheme with
name " + scheme_name + "in data base")
                ->trace_with_guard("data_base::delete_from_structure method
finished");
        throw data_base::db_remove_exception(
                "delete_from_structure:: no scheme with name " + scheme_name + "
in data base");
    }
}
// deleting collection
else {
    // find a scheme
    associative_container<std::string,
            associative_container<key, db_value *> *> * data_scheme;
    try {
        data_scheme = find_data_scheme(pool_name, scheme_name);
    }
    catch (data_base::db_find_exception const &) {
        this->debug_with_guard("data_base::delete_from_structure no such
pool/scheme name in data_base")
                ->trace_with_guard("data_base::delete_from_structure method
finished");
        throw data_base::db_remove_exception(
                "add_to_structure:: no such pool/scheme name in data_base");
    }

    try {
        delete_collection(pool_name + "/" + scheme_name + "/" + collection_name,
collection_name, data_scheme);
    }
    catch (typename bs_tree<std::string, associative_container<key, db_value *> *,
string_comparer>::remove_exception const &) {
        this->debug_with_guard("data_base::delete_from_structure no collection
with name " + collection_name + "in data base")
                ->trace_with_guard("data_base::delete_from_structure method
finished");
        throw data_base::db_remove_exception(
```

```
                    "delete_from_structure:: no collection with name " +
    collection_name + " in data base");
        }
    }

    this->trace_with_guard("data_base::delete_from_structure method finished");
}

data_base::~data_base() {
    if (_database == nullptr) {
        this->warning_with_guard("data_base::~data_base data_base has already been
    removed");
        return;
    }
    // todo: done only for bst-like trees

    this->trace_with_guard("data_base::~data_base() method started");
    std::string pool_name;

    for (const auto& [key, value] : _all_trees_allocators) {
        delete value;
    }

    delete _database;
    this->_database = nullptr;

    this->trace_with_guard("data_base::~data_base() method finished");
}

#pragma endregion
#pragma endregion
```

```
#ifndef KEY_H
#define KEY_H

#include <iostream>
#include <fstream>

class key
{
   int _applicant_id{};
   int _contest_id{};

   friend class key_comparer;
   friend std::ostream &operator<<(std::ostream &out, const key &_key_);

public:
   class create_exception final : public std::exception {
   private:
       std::string _message;

   public:
       explicit create_exception(std::string message)
               : _message(std::move(message)) {

       }

       [[nodiscard]] char const *what() const noexcept override {
           return _message.c_str();
       }
   };

public:
   key(std::ifstream* input_stream, bool is_cin)
   {
       std::string token, delimiter = ", ";
       std::string applicant_id, contest_id;
       size_t pos;
       unsigned delimiter_length = delimiter.length();

       if (is_cin) {
           std::cout << "Enter key:\napplicant id, contest id: >>";
           std::getline(std::cin, token);
       } else {
           std::getline((*input_stream), token);
       }

       if ((pos = token.find(delimiter)) != std::string::npos) {
           applicant_id = token.substr(0, pos);
           token.erase(0, pos + delimiter_length);
       } else {
           throw key::create_exception("key constructor: incorrect input for key");
       }

       if (token.empty()) {
```

```
            throw key::create_exception("key constructor: incorrect input for key");
        }

        contest_id = token;

        this->_applicant_id = stoi(applicant_id);
        this->_contest_id = stoi(contest_id);
        // to get the time as a string: dt = asctime(new_key.utc_time);
    }

    key() = default;

    key(int applicant_id, int contest_id)
    : _contest_id(contest_id), _applicant_id(applicant_id)
    {

    }

    // copy constructor
    key(key const &obj)
    : key(obj._applicant_id, obj._contest_id)
    {

    }

    // move constructor
    key(key &&obj) noexcept
    : key(obj._applicant_id, obj._contest_id)
    {
        obj._applicant_id = 0;
        obj._contest_id = 0;
    }

    // copy assignment (оператор присваивания)
    key &operator=(key const &obj)
    {
        if (this == &obj)
        {
            return *this;
        }

        _applicant_id = obj._applicant_id;
        _contest_id = obj._contest_id;

        return *this;
    }

    // move assignment (оператор присваивания перемещением)
    key &operator=(key &&obj) noexcept
    {
        if (this == &obj)
        {
            return *this;
        }
```

```
        _applicant_id = obj._applicant_id;
        obj._applicant_id = 0;

        _contest_id = obj._contest_id;
        obj._contest_id = 0;

        return *this;
    }

    ~key() = default;
};

inline std::ostream &operator<<(std::ostream &out, const key &_key_)
{
    out << _key_._applicant_id << ", " << _key_._contest_id << std::endl;
    return out;
}

class key_comparer
{
public:
    int operator()(key x, key y) {
        int applicant_id_comparison = x._applicant_id - y._applicant_id;
        if (applicant_id_comparison == 0) {
            return x._contest_id - y._contest_id;
        } else {
            return applicant_id_comparison;
        }
    }
};

#endif //KEY_H
```

```
#ifndef DB_VALUE_H
#define DB_VALUE_H

#include "string_holder.h"

typedef enum fields {
   _surname_,
   _name_,
   _patronymic_,
   _birthday_,
   _link_to_resume_,
   _hr_id_,
   _programming_language_,
   _task_count_,
   _solved_task_count_,
   _copying_
} db_value_fields;

class handler;

class db_value {
public:
   class create_exception final : public std::exception {
   private:
       std::string _message;

   public:
       explicit create_exception(std::string message)
               : _message(std::move(message)) {

       }

       [[nodiscard]] char const *what() const noexcept override {
           return _message.c_str();
       }
   };

private:
   std::string * _surname, * _name, * _patronymic;
   std::string * _birthday;
   std::string * _link_to_resume;
   int _hr_id;
   std::string * _programming_language;
   unsigned _task_count;
   unsigned _solved_task_count;
   bool _copying;

   uint64_t _timestamp;

   handler * _chain_of_resp;
   handler * _last_handler;

   friend class db_value_builder;
```

```
    friend class update_command;
    friend class add_command;
    friend class add_handler;

public:
    handler * get_first_handler();
    handler * get_last_handler();
    void add_new_handler(handler * handler_);

public:
    friend std::ostream &operator<<(std::ostream &out, const db_value &value);

    db_value * make_a_copy()
    {
        db_value * copy = new db_value((*(this->_surname)), (*(this->_name)),
    (*(this->_patronymic)),
                                       (*(this->_birthday)),
    (*(this->_link_to_resume)), this->_hr_id,
                                       (*(this->_programming_language)),
    this->_task_count, this->_solved_task_count,
                                       this->_copying);
        copy->_timestamp = this->_timestamp;
        copy->_chain_of_resp = nullptr;
        copy->_last_handler = nullptr;
        return copy;
    }

private:
    static std::string * get_ptr_from_string_holder(std::string const & s)
    {
        return string_holder::get_instance()->get_string(s);
    }

    static void remove_string_from_string_holder(std::string const & s)
    {
        string_holder::get_instance()->remove_string(s);
    }

    db_value(std::string const & surname, std::string const & name, std::string const
    & patronymic,
             std::string const & birthday, std::string const & link_to_resume,
    unsigned hr_id,
             std::string const & prog_lang, unsigned task_count, unsigned
    solved_task_count, bool copying)
            : _hr_id(hr_id), _task_count(task_count),
    _solved_task_count(solved_task_count), _copying(copying)
    {
        _surname = get_ptr_from_string_holder(surname);
        _name = get_ptr_from_string_holder(name);
        _patronymic = get_ptr_from_string_holder(patronymic);
        _birthday = get_ptr_from_string_holder(birthday);
        _link_to_resume = get_ptr_from_string_holder(link_to_resume);
        _programming_language = get_ptr_from_string_holder(prog_lang);

        _timestamp = duration_cast<std::chrono::milliseconds>
```

```
                (std::chrono::system_clock::now().time_since_epoch()).count();
        _chain_of_resp = nullptr;
        _last_handler = nullptr;
    }

public:
    ~db_value();
};

inline std::ostream &operator<<(std::ostream &out, const db_value &value) {
    out << (*(value._surname)) << " " << (*(value._name)) << " " <<
    (*(value._patronymic)) << std::endl;
    out << (*(value._birthday)) << std::endl;
    out << (*(value._link_to_resume)) << std::endl;
    out << value._hr_id << std::endl;
    out << (*(value._programming_language)) << std::endl;
    out << value._task_count << std::endl;
    out << value._solved_task_count << std::endl;
    out << value._copying << std::endl;
    out << value._timestamp << std::endl;
    return out;
}

#endif //DB_VALUE_H
```

Реализация методов класса *db_value*.

```
#include "../chain_of_resp_and_command/handler.h"

handler * db_value::get_first_handler()
{
    return this->_chain_of_resp;
}

handler * db_value::get_last_handler()
{
    return _last_handler;
}

void db_value::add_new_handler(handler * handler_)
{
    if (_chain_of_resp == nullptr) {
        _chain_of_resp = handler_;
    }

    if (_last_handler != nullptr) {
        _last_handler->set_next(handler_);
    }
    _last_handler = handler_;
}

db_value::~db_value()
{
    remove_string_from_string_holder((*_surname));
```

```
    remove_string_from_string_holder((*_name));
    remove_string_from_string_holder((*_patronymic));
    remove_string_from_string_holder((*_birthday));
    remove_string_from_string_holder((*_link_to_resume));
    remove_string_from_string_holder((*_programming_language));

    if (_chain_of_resp != nullptr) {
        _chain_of_resp->delete_chain_of_responsibility();
    }
}
```

Раздел 4. Класс *db_value_builder*.

```
#ifndef DB_VALUE_BUILDER_H
#define DB_VALUE_BUILDER_H

#include "db_value.h"

class db_value_builder {
   std::string _surname, _name, _patronymic;
   std::string _birthday;
   std::string _link_to_resume;
   int _hr_id;
   std::string _programming_language;
   unsigned _task_count;
   unsigned _solved_task_count;
   bool _copying;

public:
   db_value_builder * with_surname(std::string && surname);
   db_value_builder * with_name(std::string && name);
   db_value_builder * with_patronymic(std::string && patronymic);
   db_value_builder * with_birthday(std::string && birthday);
   db_value_builder * with_link_to_resume(std::string && link);
   db_value_builder * with_hr_id(int hr_id);
   db_value_builder * with_programming_language(std::string && programming_language);
   db_value_builder * with_task_count(unsigned task_count);
   db_value_builder * with_solved_task_count(unsigned solved_task_count);
   db_value_builder * with_copying(bool did_copy);

   [[nodiscard]] db_value *build() const;

private:
   static void check_if_string_of_value_is_empty(std::string const& s)
   {
       if (s.empty()) {
           throw db_value::create_exception("db_user_communication:: incorrect input
   while building a value");
       }
   }

public:
   db_value *build_from_stream(std::ifstream *input_stream, bool is_cin);

};

#endif //DB_VALUE_BUILDER_H
```

Реализация методов класса *db_value_builder*.

```
#include "db_value_builder.h"

db_value_builder *db_value_builder::with_surname(std::string &&surname) {
   _surname = std::move(surname);
   return this;
```

```
}

db_value_builder *db_value_builder::with_name(std::string &&name) {
   _name = std::move(name);
   return this;
}

db_value_builder *db_value_builder::with_patronymic(std::string &&patronymic) {
   _patronymic = std::move(patronymic);
   return this;
}

db_value_builder *db_value_builder::with_birthday(std::string &&birthday) {
   _birthday = std::move(birthday);
   return this;
}

db_value_builder *db_value_builder::with_link_to_resume(std::string &&link) {
   _link_to_resume = std::move(link);
   return this;
}

db_value_builder *db_value_builder::with_hr_id(int hr_id)
{
   _hr_id = hr_id;
   return this;
}

db_value_builder *db_value_builder::with_programming_language(std::string
   &&programming_language) {
   _programming_language = std::move(programming_language);
   return this;
}

db_value_builder *db_value_builder::with_task_count(unsigned int task_count) {
   _task_count = task_count;
   return this;
}

db_value_builder *db_value_builder::with_solved_task_count(unsigned int
   solved_task_count) {
   _solved_task_count = solved_task_count;
   return this;
}

db_value_builder *db_value_builder::with_copying(bool did_copy) {
   _copying = did_copy;
   return this;
}

// todo: check if it works
db_value* db_value_builder::build() const {
   return new db_value(_surname, _name, _patronymic,
                   _birthday,
                   _link_to_resume,
```

```
                    _hr_id,
                    _programming_language,
                    _task_count, _solved_task_count,
                    _copying);
}

db_value* db_value_builder::build_from_stream(std::ifstream *input_stream, bool
   is_cin) {
   std::string token, delimiter = " ";
   size_t pos;
   unsigned delimiter_length = delimiter.length();

   if (is_cin) {
       std::cout << "Enter value:" << std::endl;
       std::cout << "Surname Name Patronymic:              >>";
       std::getline(std::cin, token);
   } else {
       std::getline((*input_stream), token);
   }

   if ((pos = token.find(delimiter)) != std::string::npos) {
       this->with_surname(std::move(token.substr(0, pos)));
       token.erase(0, pos + delimiter_length);
   } else {
       throw db_value::create_exception("db_value_builder::build_from_stream::
   incorrect input while building a value");
   }

   if ((pos = token.find(delimiter)) != std::string::npos) {
       this->with_name(std::move(token.substr(0, pos)));
       token.erase(0, pos + delimiter_length);
   } else {
       throw db_value::create_exception("db_value_builder::build_from_stream::
   incorrect input while building a value");
   }

   if (token.empty()) {
       throw db_value::create_exception("db_value_builder::build_from_stream::
   incorrect input while building a value");
   }
   this->with_patronymic(std::move(token));

   if (is_cin) {
       std::cout << "Birthday <dd/mm/year>:                >>";
       std::getline(std::cin, token);
   } else {
       std::getline((*input_stream), token);
   }
   check_if_string_of_value_is_empty(token);
   this->with_birthday(std::move(token));

   if (is_cin) {
       std::cout << "Link to resume:                       >>";
       std::getline(std::cin, token);
   } else {
```

```
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);
    this->with_link_to_resume(std::move(token));

    if (is_cin) {
        std::cout << "id of HR-manager:                    >>";
        std::getline(std::cin, token);
    } else {
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);
    this->with_hr_id(stoi(token));

    if (is_cin) {
        std::cout << "Programming language:                >>";
        std::getline(std::cin, token);
    } else {
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);
    this->with_programming_language(std::move(token));

    if (is_cin) {
        std::cout << "Task count: {whole number}           >>";
        std::getline(std::cin, token);
    } else {
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);
    this->with_task_count(stoi(token));

    if (is_cin) {
        std::cout << "Solved tasks count {whole number}:   >>";
        std::getline(std::cin, token);
    } else {
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);
    this->with_solved_task_count(stoi(token));

    if (is_cin) {
        std::cout << "Copying with a micro-earphone {bool}: >>";
        std::getline(std::cin, token);
    } else {
        std::getline((*input_stream), token);
    }
    check_if_string_of_value_is_empty(token);

    if (token == "true" || token == "1") {
        this->with_copying(true);
    } else if (token == "false" || token == "0") {
        this->with_copying(false);
    } else {
```

```
        throw db_value::create_exception("db_value_builder::build_from_stream::
    incorrect input while building a value {bool must be true/false || 1/0}");
    }
    return (this->build());
}
```

```
#ifndef STRING_HOLDER_H
#define STRING_HOLDER_H

#include "../binary_tree/associative_container.h"
#include "../avl_tree/avl_tree.h"

class string_holder
{
public:
   class string_comparer
   {
   public:
       int operator()(std::string const & x, std::string const & y) {
           return x.compare(y);
       }
   };
private:
   static string_holder *_instance;
public:
   static string_holder *get_instance()
   {
       if (_instance == nullptr)
       {
           _instance = new string_holder;
       }

       return _instance;
   }

private:

   associative_container<std::string, std::pair<std::string *, unsigned *>> *_pool;

private:

   string_holder()
   {
       _pool = new avl_tree<std::string, std::pair<std::string *, unsigned *>,
    string_comparer>;
   }

public:

   string_holder(string_holder const &) = delete;

   string_holder(string_holder &&) = delete;

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

   string_holder &operator=(string_holder &&) = delete;

   ~string_holder() noexcept
   {
```

```
        delete _pool;
    }

public:

    std::string *get_string(std::string const &key)
    {
        std::pair<std::string *, unsigned *> find_result;
        try
        {
            find_result = _pool->get(key);
        }
        catch (bs_tree<std::string, std::pair<std::string *, unsigned *>,
    string_comparer>::find_exception const &ex)
        {
            auto * string_to_insert = new std::string(key);
            unsigned * used = new unsigned();
            (*used) = 1;
            _pool->insert(key, std::move(std::pair<std::string *, unsigned
    *>(string_to_insert, used)));
            return _pool->get(key).first;
        }
        (*(find_result.second))++;
        return find_result.first;
    }

    void remove_string(std::string const &key)
    {
        std::pair<std::string *, unsigned *> string_and_used;
        try
        {
            string_and_used = _pool->get(key);
        }
        catch (bs_tree<std::string, std::string *, string_comparer>::find_exception
    const &ex)
        {
            // it cannot be (?)
            return;
        }

        if ((*(string_and_used.second)) == 1) {
            _pool->remove(key);
        }
    }

};

inline string_holder *string_holder::_instance = nullptr;

#endif //STRING_HOLDER_H
```

Раздел 6. Классы *command*, *add_command*, *remove_command*, *update_command*.

```
#ifndef COURSE_WORK_COMMAND_H
#define COURSE_WORK_COMMAND_H

#include "../db_value/db_value.h"
#include "../db_value/db_value_builder.h"

class command {
public:
   virtual ~command() = default;
   virtual db_value ** execute(db_value **) = 0;
};

class remove_command final : public command
{
public:
   db_value ** execute(db_value ** initial_value_copy) override
   {
       std::cout << "remove command" << std::endl;
       (*initial_value_copy)->~db_value();
       (*initial_value_copy) = nullptr;
       return initial_value_copy;
   }
};

class update_command final : public command {
   std::map<db_value_fields, unsigned char *> _update_dictionary;
public:
   explicit update_command(std::map<db_value_fields, unsigned char *> dictionary)
   : _update_dictionary(std::move(dictionary))
   {

   }

   ~update_command() override {
       // do delete void * for all items in dictionary
       for (const auto & iter : _update_dictionary) {
           delete iter.second;
       }
   }

   db_value ** execute(db_value ** initial_value_copy) override
   {
       std::cout << "update command" << std::endl;

       for (const auto & iter : _update_dictionary) {
           switch (iter.first) {
               case db_value_fields::_surname_:

    string_holder::get_instance()->remove_string((*((*initial_value_copy)->_surname))
    );
```

```
                (*initial_value_copy)->_surname =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_name_:

string_holder::get_instance()->remove_string((*((*initial_value_copy)->_name)));
                (*initial_value_copy)->_name =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_patronymic_:

string_holder::get_instance()->remove_string((*((*initial_value_copy)->_patronymi
c)));
                (*initial_value_copy)->_patronymic =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_birthday_:

string_holder::get_instance()->remove_string((*((*initial_value_copy)->_birthday)
));
                (*initial_value_copy)->_birthday =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_link_to_resume_:

string_holder::get_instance()->remove_string((*((*initial_value_copy)->_link_to_r
esume)));
                (*initial_value_copy)->_link_to_resume =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_hr_id_:
                (*initial_value_copy)->_hr_id = (*(reinterpret_cast<int
*>(iter.second)));
                break;
            case db_value_fields::_programming_language_:

string_holder::get_instance()->remove_string((*((*initial_value_copy)->_programmi
ng_language)));
                (*initial_value_copy)->_programming_language =
string_holder::get_instance()->get_string((*(reinterpret_cast<std::string
*>(iter.second))));
                break;
            case db_value_fields::_task_count_:
                (*initial_value_copy)->_task_count = (*(reinterpret_cast<unsigned
*>(iter.second)));
                break;
            case db_value_fields::_solved_task_count_:
                (*initial_value_copy)->_solved_task_count =
(*(reinterpret_cast<unsigned *>(iter.second)));
                break;
```

```
                case db_value_fields::_copying_:
                    (*initial_value_copy)->_copying = (*(reinterpret_cast<bool
    *>(iter.second)));
                    break;
            }
        }

        return initial_value_copy;
    }
};

class add_command final : public command
{
    std::map<db_value_fields, unsigned char *> _add_dictionary;
    // по факту, то же самое, что update command, но тут именно все поля новые, и т.к.
    add можно только после удаления, то нужно создать новое db_value;
public:
    explicit add_command(std::map<db_value_fields, unsigned char *> dictionary)
    : _add_dictionary(std::move(dictionary))
    {

    }

    ~add_command() override {
        for (const auto & iter : _add_dictionary) {
            delete iter.second;
        }
    }
    db_value ** execute(db_value ** initial_value_copy) override
    {
        std::cout << "add command" << std::endl;

        auto * dbValueBuilder = new db_value_builder();

        for (auto const & iter : _add_dictionary) {
            switch (iter.first) {
                case db_value_fields::_surname_:

    dbValueBuilder->with_surname(std::move(*(reinterpret_cast<std::string
    *>(iter.second))));
                    break;
                case db_value_fields::_name_:
                    dbValueBuilder->with_name(std::move(*(reinterpret_cast<std::string
    *>(iter.second))));
                    break;
                case db_value_fields::_patronymic_:

    dbValueBuilder->with_patronymic(std::move(*(reinterpret_cast<std::string
    *>(iter.second))));
                    break;
                case db_value_fields::_birthday_:

    dbValueBuilder->with_birthday(std::move(*(reinterpret_cast<std::string
    *>(iter.second))));
                    break;
```

```
                case db_value_fields::_link_to_resume_:

    dbValueBuilder->with_link_to_resume(std::move(*(reinterpret_cast<std::string
    *>(iter.second))));
                    break;
                case db_value_fields::_hr_id_:
                    dbValueBuilder->with_hr_id((*(reinterpret_cast<int
    *>(iter.second))));
                    break;
                case db_value_fields::_programming_language_:

    dbValueBuilder->with_programming_language(std::move(*(reinterpret_cast<std::strin
    g *>(iter.second))));
                    break;
                case db_value_fields::_task_count_:
                    dbValueBuilder->with_task_count((*(reinterpret_cast<unsigned
    *>(iter.second))));
                    break;
                case db_value_fields::_solved_task_count_:

    dbValueBuilder->with_solved_task_count((*(reinterpret_cast<unsigned
    *>(iter.second))));
                    break;
                case db_value_fields::_copying_:
                    dbValueBuilder->with_copying((*(reinterpret_cast<bool
    *>(iter.second))));
                    break;
            }
        }
        (*initial_value_copy) = dbValueBuilder->build();
        delete dbValueBuilder;
        return initial_value_copy;
    }
};

#endif //COURSE_WORK_COMMAND_H
```

Раздел 7. Классы *handler*, *add_handler*, *remove_handler*, *update_handler*.

```
#ifndef HANDLER_H
#define HANDLER_H

#include "command.h"

class handler {
public:
    enum handler_types {
        _add_handler_,
        _update_handler_,
        _remove_handler_
    };
```

```
    class order_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit order_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

protected:
    handler * _next_handler;
    command * _command;
    uint64_t timestamp;
    handler_types _type_of_handler;

public:
    [[nodiscard]] handler_types get_handler_type() const
    {
        return this->_type_of_handler;
    }

    handler *set_next(handler *handler)
    {
        this->_next_handler = handler;
        return handler;
    }

public:
    db_value* handle(db_value ** request, uint64_t time_parameter)
    {
        // todo: если у значения, заданного в дереве timestamp < time_parameter,
    сгенерить исключение
        if (this->timestamp <= time_parameter) {
            request = _command->execute(request);
        } else {
            return (*request);
        }

        if (this->_next_handler != nullptr) {
            return this->_next_handler->handle(request, time_parameter);
        }

        return (*request);
    }

public:
    explicit handler()
            : _next_handler(nullptr), _command(nullptr)
    {
```

```
        timestamp = duration_cast<std::chrono::milliseconds>
                (std::chrono::system_clock::now().time_since_epoch()).count();
    }

    void delete_chain_of_responsibility()
    {
        handler * next = this->_next_handler;
        this->~handler();
        if (next != nullptr) {
            next->delete_chain_of_responsibility();
        }
    }

    ~handler()
    {
        delete _command;
    }
};

class add_handler final : public handler
{
public:
    explicit add_handler(db_value * new_value)
    {
        // собираем словарь для команды add
        std::map<db_value_fields, unsigned char *> command_input;

        command_input[db_value_fields::_surname_] =
                reinterpret_cast<unsigned char *>(new
    std::string((*(new_value->_surname))));
        command_input[db_value_fields::_name_] =
                reinterpret_cast<unsigned char *>(new
    std::string((*(new_value->_name))));
        command_input[db_value_fields::_patronymic_] =
                reinterpret_cast<unsigned char *>(new
    std::string(*(new_value->_patronymic)));
        command_input[db_value_fields::_birthday_] =
                reinterpret_cast<unsigned char *>(new
    std::string(*(new_value->_birthday)));
        command_input[db_value_fields::_link_to_resume_] =
                reinterpret_cast<unsigned char *>(new
    std::string(*(new_value->_link_to_resume)));
        command_input[db_value_fields::_hr_id_] =
                reinterpret_cast<unsigned char *>(new int(new_value->_hr_id));
        command_input[db_value_fields::_programming_language_] =
                reinterpret_cast<unsigned char *>(new
    std::string(*(new_value->_programming_language)));
        command_input[db_value_fields::_task_count_] =
                reinterpret_cast<unsigned char *>(new
    unsigned(new_value->_task_count));
        command_input[db_value_fields::_solved_task_count_] =
                reinterpret_cast<unsigned char *>(new
    unsigned(new_value->_solved_task_count));
        command_input[db_value_fields::_copying_] =
                reinterpret_cast<unsigned char *>(new bool(new_value->_copying));
```

```
        _command = new add_command(command_input);
        _type_of_handler = handler_types::_add_handler_;
    }
};

class update_handler final : public handler
{
public:
    explicit update_handler(std::map<db_value_fields, unsigned char *> command_input)
    {
        _command = new update_command(std::move(command_input));
        _type_of_handler = handler_types::_update_handler_;
    }
};

class remove_handler final : public handler
{
public:
    remove_handler()
    {
        _command = new remove_command();
        _type_of_handler = handler_types::_remove_handler_;
    }
};

#endif //HANDLER_H
```

```
#ifndef LOGGER_BASE_CLASS_H
#define LOGGER_BASE_CLASS_H

#include <iostream>
#include <exception>
#include
   "C:\fund-algths\lab5\vcpkg\installed\x86-windows\include\rapidjson\document.h"
#include
   "C:\fund-algths\lab5\vcpkg\installed\x86-windows\include\rapidjson\istreamwrapper
   .h"

#include <fstream>
#include <map>
#include <string>
#include <chrono>

class logger_builder;

class logger {
public:
   enum class severity {
       trace = 0,
       debug,
       information,
       warning,
       error,
       critical
   };

   class severity_exception final : public std::exception {
   private:
       std::string _message;

   public:
       explicit severity_exception(std::string message)
               : _message(std::move(message)) {
       }

       [[nodiscard]] char const *what() const noexcept override {
           return _message.c_str();
       }

   };

   static std::string severity_to_string_logger(severity s);

   static logger::severity from_string_to_severity_parse(const std::string &s);

   virtual logger const *log(std::string const &target, severity level) const = 0;

   virtual ~logger() noexcept = default;
};
```

```
class logger_impl : public logger {
private:
   std::map<std::string, severity> this_logger_streams;
   static std::map<std::string, std::pair<std::ofstream *, size_t> >
   _all_loggers_streams;

   explicit logger_impl(std::map<std::string, severity> const
   &info_to_construct_from);

public:
   logger_impl(logger_impl const &log) = delete;        // copying
   logger_impl &operator=(logger_impl const &) = delete; // assignment

   friend class logger_builder_impl;

   // static logger_builder builder();
   ~logger_impl();

   logger const *log(std::string const &target, severity level) const override;
};

#endif //LOGGER_BASE_CLASS_H
```

Реализация методов классов *logger* и *logger_impl*.

```
#include "logger.h"

std::string logger::severity_to_string_logger(severity s) {
   switch (s) {
       case severity::trace:
           return "[ trace ] ";
       case severity::debug:
           return "[ debug ] ";
       case severity::information:
           return "[ information ] ";
       case severity::warning:
           return "[ warning ] ";
       case severity::error:
           return "[ error ] ";
       case severity::critical:
           return "[ critical ] ";
       default:
           throw severity_exception("invalid severity value used");
   }
}

logger::severity logger::from_string_to_severity_parse(const std::string &s) {
   if (s == "trace") {
       return logger::severity::trace;
   }
   if (s == "debug") {
       return logger::severity::debug;
   }
   if (s == "information") {
```

```
        return logger::severity::information;
    }
    if (s == "warning") {
        return logger::severity::warning;
    }
    if (s == "error") {
        return logger::severity::error;
    }
    if (s == "critical") {
        return logger::severity::critical;
    } else {
        throw logger::severity_exception("invalid severity string passed");
    }
}

logger_impl::~logger_impl() {
    std::pair<std::ofstream *, size_t> stream_pair;
    for (auto &this_logger_stream: this_logger_streams) {
        if ((stream_pair =
    logger_impl::_all_loggers_streams[this_logger_stream.first]).second == 1) {
            if (stream_pair.first != nullptr) {
                stream_pair.first->flush();
                stream_pair.first->close();
            }
            logger_impl::_all_loggers_streams.erase(this_logger_stream.first);
//              this_logger_streams.erase(this_logger_stream.first);
        } else {
            logger_impl::_all_loggers_streams[this_logger_stream.first].second--;
        }
    }
}

logger_impl::logger_impl(std::map<std::string, severity> const
    &info_to_construct_from)
        : this_logger_streams(info_to_construct_from) {
    std::pair<std::ofstream *, size_t> stream_pair;
    for (auto &this_logger_stream: this_logger_streams) {
        auto iterator = _all_loggers_streams.find(this_logger_stream.first);
        if (iterator == _all_loggers_streams.end()) {
            std::ofstream *new_stream = nullptr;
            if (this_logger_stream.first != "console") {
                new_stream = new std::ofstream();
                new_stream->open(this_logger_stream.first);
                // todo: if new_stream is not opened, throw a message
                *new_stream << "<--- " <<
    severity_to_string_logger(this_logger_stream.second) << " --->" << std::endl;
            }
            _all_loggers_streams.insert(std::make_pair(this_logger_stream.first,
    std::make_pair(new_stream, 1)));
        } else {
            iterator->second.second++;
        }
    }
}
```

```
logger const *logger_impl::log(std::string const &target, severity level) const {
   for (const auto &this_loggers_stream: this_logger_streams) {
       if (this_loggers_stream.second <= level) {
           time_t raw_time;
           struct tm *time_info;
           char curr_time[80];

           time(&raw_time);
           time_info = localtime(&raw_time);
           strftime(curr_time, 80, "[ %d/%m/%Y %H:%M:%S ]", time_info);

           if (this_loggers_stream.first == "console") {
               std::cout << curr_time << severity_to_string_logger(level) << target
   << std::endl;
           } else {
               (*logger_impl::_all_loggers_streams[this_loggers_stream.first].first)
   << curr_time

   << severity_to_string_logger(

   level) << target

   << std::endl;
           }
       }
   }
   return this;
}

std::map<std::string, std::pair<std::ofstream *, size_t>>
   logger_impl::_all_loggers_streams =
       std::map<std::string, std::pair<std::ofstream *, size_t>>();
```

Раздел 9. Классы *logger_builder* и *logger_builder_impl*.

```
#ifndef LAB5_LOGGER_BUILDER_H
#define LAB5_LOGGER_BUILDER_H

#include "logger.h"

class logger_builder {
public:
   virtual logger_builder *with_stream(std::string const &, logger::severity) = 0;

   [[nodiscard]] virtual logger *build() const = 0;

   virtual ~logger_builder() {}

   virtual logger *config_from_json(const std::string &) = 0;
};

class logger_builder_impl final : public logger_builder {
private:
   std::map<std::string, logger::severity> _under_construction_logger_setup;
public:
   logger_builder *with_stream(const std::string &stream, logger::severity severity)
   override;

   [[nodiscard]] logger *build() const override;

   logger *config_from_json(const std::string &filename) override;
};

#endif //LAB5_LOGGER_BUILDER_H
```

Реализация методов классов *logger_builder* и *logger_builder_impl*.

```
#include "logger_builder.h"

logger_builder *logger_builder_impl::with_stream(const std::string &stream,
   logger::severity severity) {
   _under_construction_logger_setup[stream] = severity;
   return this;
}

logger *logger_builder_impl::build() const {
   return new logger_impl(_under_construction_logger_setup);
}

logger *logger_builder_impl::config_from_json(const std::string &filename) {
   std::ifstream config_file_ifs(filename);
   if (!(config_file_ifs.is_open())) {
       throw std::runtime_error("Configuration file not opened");
   }

   rapidjson::IStreamWrapper config_file_isw(config_file_ifs);
   rapidjson::Document doc;
```

```
    doc.ParseStream(config_file_isw);

    if (doc.HasParseError()) {
        config_file_ifs.close();
        throw std::runtime_error("Error parsing configuration file");
    }

    logger *log;
    const rapidjson::Value &streams = doc["streams"];
    rapidjson::Value::ConstValueIterator iter;
    for (iter = streams.Begin(); iter != streams.End(); ++iter) {
        this->with_stream(iter->GetObject()["name"].GetString(),

    logger::from_string_to_severity_parse(iter->GetObject()["severity"].GetString()))
    ;
    }

    log = this->build();
    config_file_ifs.close();
    return log;
}
```

```
#ifndef LOGGER_HOLDER_H
#define LOGGER_HOLDER_H

#include "logger.h"

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;
};

logger_holder const *logger_holder::log_with_guard(
       std::string const &message,
       logger::severity severity) const
{
   auto *got_logger = get_logger();
   if (got_logger != nullptr)
   {
       got_logger->log(message, severity);
   }

   return this;
}

logger_holder const *logger_holder::trace_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::trace);
}
```

```
logger_holder const *logger_holder::debug_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::debug);
}

logger_holder const *logger_holder::information_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::information);
}

logger_holder const *logger_holder::warning_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::warning);
}

logger_holder const *logger_holder::error_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::error);
}

logger_holder const *logger_holder::critical_with_guard(
       std::string const &message) const
{
   return log_with_guard(message, logger::severity::critical);
}

#endif //LOGGER_HOLDER_H
```

```
#ifndef MEMORY_BASE_CLASS_H
#define MEMORY_BASE_CLASS_H

#include "../logger/logger_builder.h"
#include "../logger/logger_holder.h"

/* Common structure for allocators:
* size_t          -- size of allocator pool
* logger*
* memory*         -- parent allocator (if present)
* allocation_mode -- first/best/worst
* void *          -- ptr to (available/occupied block from pool)
*
* Common for all blocks:
*  - if size_t size is used, it is a size of a whole block including service part
*
* */

class memory : protected logger_holder
{
public:
   enum Allocation_strategy {
       first_fit,
       best_fit,
       worst_fit,
   };

   class memory_exception final : public std::exception
   {
   private:
       std::string _message;

   public:
       explicit memory_exception(std::string const & message)
               : _message(message)
       {

       }

       char const * what() const noexcept override
       {
           return _message.c_str();
       }

   };

protected:
   void *_ptr_to_allocator_metadata;
   void dump_occupied_block_before_deallocate(void *const current_block_address)
   const;

#pragma region Allocator properties
   [[nodiscard]] virtual size_t get_allocator_service_block_size() const;
```

```
    [[nodiscard]] virtual size_t* get_ptr_size_of_allocator_pool() const;
    [[nodiscard]] virtual logger** get_ptr_logger_of_allocator() const;
    [[nodiscard]] virtual memory** get_ptr_to_ptr_parent_allocator() const;
    [[nodiscard]] virtual Allocation_strategy* get_ptr_allocation_mode() const;
    [[nodiscard]] virtual void ** get_ptr_to_ptr_to_pool_start() const;
    [[nodiscard]] virtual void * get_ptr_to_allocator_trusted_pool() const;
#pragma endregion

#pragma region Available block methods
    [[nodiscard]] virtual void *get_first_available_block_address() const; // get the
    address of the first available block in allocator
    [[nodiscard]] virtual void ** get_first_available_block_address_address() const;
    // get an address field of the first block available in allocator

    [[nodiscard]] virtual size_t get_available_block_service_block_size() const;
    virtual size_t get_available_block_size(void * memory_block) const;
    virtual void * get_next_available_block_address(void * memory_block) const;
#pragma endregion

#pragma region Occupied block methods
    [[nodiscard]] virtual void ** get_first_occupied_block_address_address() const;

    [[nodiscard]] virtual size_t get_occupied_block_service_block_size() const;
    virtual size_t get_occupied_block_size(void * memory_block) const;
    virtual size_t get_size_of_occupied_block_pool(void * const occupied_block) const
    = 0;

    virtual void * get_next_occupied_block_address(void * memory_block) const;
    virtual void * get_previous_occupied_block_address(void * memory_block) const;
#pragma endregion

    std::string address_to_hex(void const * ptr) const;

public:
    virtual ~memory() noexcept = default;

    virtual void *allocate(size_t target_size) const = 0;

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

    friend void * operator+=(memory const &allocator, size_t target_size);
    friend void operator-=(memory const &allocator, void const * const
    target_to_dealloc);
};

#pragma region Reach && change allocator properties
size_t* memory::get_ptr_size_of_allocator_pool() const {
    return reinterpret_cast<size_t *>(_ptr_to_allocator_metadata);
}

logger **memory::get_ptr_logger_of_allocator() const {
    return reinterpret_cast<logger **>(get_ptr_size_of_allocator_pool() + 1);
}

memory **memory::get_ptr_to_ptr_parent_allocator() const {
```

```
    return reinterpret_cast<memory**>(get_ptr_logger_of_allocator() + 1);
}

memory::Allocation_strategy* memory::get_ptr_allocation_mode() const {
    return reinterpret_cast<memory::Allocation_strategy
    *>(get_ptr_to_ptr_parent_allocator() + 1);
}

void **memory::get_ptr_to_ptr_to_pool_start() const {
    return reinterpret_cast<void **>(get_ptr_allocation_mode() + 1);
}
#pragma endregion

std::string memory::address_to_hex(const void * ptr) const {
    char address_buf[(sizeof(void const * const) << 3) + 3];
    sprintf(address_buf, "0x%p", ptr);
    return std::string { address_buf };
}

void memory::dump_occupied_block_before_deallocate(void *const current_block_address)
    const {
    if (*get_ptr_logger_of_allocator() == nullptr) {
        return;
    }

    auto const current_block_size =
    get_size_of_occupied_block_pool(current_block_address);
    auto const * dump_iterator = reinterpret_cast<unsigned char
    *>(current_block_address);
    std::string result;

    for (auto i = 0; i < current_block_size; i++) {
        result += std::to_string(static_cast<unsigned short>(*dump_iterator++));

        if (i != current_block_size - 1) {
            result += ' ';
        }
    }

    this->trace_with_guard("Block dump: [" + result + "]");
}

void *operator+=(const memory &allocator, size_t target_size) {
    return allocator.allocate(target_size);
}

void operator-=(const memory &allocator, const void *const target_to_dealloc) {
    allocator.deallocate(target_to_dealloc);
}

#pragma region virtual in-class methods implemention
void *memory::get_first_available_block_address() const {
    throw memory::memory_exception("Method get_first_available_block_address not
    implemented");
}
```

```
void **memory::get_first_available_block_address_address() const {
   throw memory::memory_exception("Method get_first_available_block_address_address
   not implemented");
}

size_t memory::get_available_block_service_block_size() const {
   throw memory::memory_exception("Method get_available_block_service_block_size not
   implemented");
}

size_t memory::get_available_block_size(void * memory_block) const
{
   throw memory::memory_exception("Method get_available_block_size not implemented");
}

void * memory::get_next_available_block_address(void * memory_block) const
{
   throw memory::memory_exception("Method get_next_available_block_address not
   implemented");
}

void ** memory::get_first_occupied_block_address_address() const
{
   throw memory::memory_exception("Method get_first_occupied_block_address_address
   not implemented");
}

size_t memory::get_occupied_block_service_block_size() const
{
   throw memory::memory_exception("Method get_occupied_block_service_block_size not
   implemented");
}

size_t memory::get_occupied_block_size(void * memory_block) const
{
   throw memory::memory_exception("Method get_occupied_block_size not implemented");
}

void * memory::get_next_occupied_block_address(void * memory_block) const
{
   throw memory::memory_exception("Method get_next_occupied_block_address not
   implemented");
}

void * memory::get_previous_occupied_block_address(void * memory_block) const
{
   throw memory::memory_exception("Method get_previous_occupied_block_address not
   implemented");
}

size_t memory::get_allocator_service_block_size() const {
   throw memory::memory_exception("Method get_allocator_service_block_size not
   implemented");
}
```

```
void *memory::get_ptr_to_allocator_trusted_pool() const {
   throw memory::memory_exception("Method get_ptr_to_allocator_trusted_pool not
   implemented");
}

#pragma endregion

#endif //MEMORY_BASE_CLASS_H
```

```
#ifndef MEMORY_HOLDER_H
#define MEMORY_HOLDER_H

#include "memory_base_class.h"

class memory_holder {
public:
   virtual ~memory_holder() noexcept = default;

public:
   void *allocate_with_guard(size_t block_size) const;

   void deallocate_with_guard(void * block_pointer) const;

protected:
   virtual memory *get_memory() const noexcept = 0;
};

void *memory_holder::allocate_with_guard(size_t block_size) const {
   auto * allocator = get_memory();

   return allocator == nullptr ?
       ::operator new(block_size)
       : allocator->allocate(block_size);
}

void memory_holder::deallocate_with_guard(void *block_pointer) const {
   auto *allocator = get_memory();

   if (allocator == nullptr) {
       ::operator delete(block_pointer);
   } else {
           allocator->deallocate(block_pointer);
   }
}

#endif //MEMORY_HOLDER_H
```

```
#ifndef MEMORY_FROM_GLOBAL_HEAP_H
#define MEMORY_FROM_GLOBAL_HEAP_H

#include "../allocator/memory_base_class.h"

class memory_from_global_heap final : public memory
{
   logger** get_ptr_logger_of_allocator() const override;

public:
   explicit memory_from_global_heap(logger * gh_allocator_logger);

   ~memory_from_global_heap() override;

   memory_from_global_heap(memory_from_global_heap const &) = delete;

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

   void *allocate(size_t target_size) const override;

   void deallocate(void const * const target_to_dealloc) const override;

   size_t get_size_of_occupied_block_pool(void * const occupied_block) const
   override;

private:

   logger *get_logger() const noexcept override;
};

void *memory_from_global_heap::allocate(size_t target_size) const {
   this->trace_with_guard("memory_from_global_heap::allocate method execution
   started");

   try
   {
       auto *block_pointer = ::operator new(target_size + sizeof(size_t));
       *reinterpret_cast<size_t *>(block_pointer) = target_size;

       this->information_with_guard("memory block with _size = " +
   std::to_string(target_size) + " was allocated successfully")
           ->debug_with_guard("Allocated block address: " +
   address_to_hex(block_pointer))
           ->trace_with_guard("memory_from_global_heap::allocate method execution
   finished");

       return reinterpret_cast<void *>(reinterpret_cast<size_t *>(block_pointer) +
   1);
   }
   catch (std::bad_alloc const &) {
       this->error_with_guard("memory block with _size = " +
   std::to_string(target_size) + " cannot be allocated")
```

```
            ->trace_with_guard("memory_from_global_heap::allocate method execution
    finished");
        throw memory::memory_exception("A block cannot be allocated in global heap
    allocator");
    }
}

void memory_from_global_heap::deallocate(const void *const target_to_dealloc) const {
    this->trace_with_guard("memory_from_global_heap::deallocate method execution
    started");

    dump_occupied_block_before_deallocate(const_cast<void *>(target_to_dealloc));
    void * tmp = reinterpret_cast<size_t *>(const_cast<void *>(target_to_dealloc)) -
    1;

    std::string address = address_to_hex(tmp);

    ::operator delete(tmp);

    this->information_with_guard("memory block with address: " + address + " was
    deallocated successfully")
        ->trace_with_guard("memory_from_global_heap::deallocate method execution
    finished");
}

// ptr to metadata start
size_t memory_from_global_heap::get_size_of_occupied_block_pool(void *const
    occupied_block) const {
    return *(reinterpret_cast<size_t *>(occupied_block) - 1);
}

memory_from_global_heap::memory_from_global_heap(logger * gh_allocator_logger)
{
    try {
        _ptr_to_allocator_metadata = ::operator new(sizeof(logger *));
    }
    catch (std::bad_alloc const &ex) {
        throw memory::memory_exception("An object of global heap allocator cannot be
    constructed");
    }
    *get_ptr_logger_of_allocator() = gh_allocator_logger;
    this->trace_with_guard("An object of global heap allocator was constructed");
}

memory_from_global_heap::~memory_from_global_heap() {
    this->trace_with_guard("An object of global heap allocator was destructed");
    ::operator delete(_ptr_to_allocator_metadata);
}

logger **memory_from_global_heap::get_ptr_logger_of_allocator() const {
    return reinterpret_cast<logger **>(_ptr_to_allocator_metadata);
}

logger *memory_from_global_heap::get_logger() const noexcept {
    return *get_ptr_logger_of_allocator();
```

```
}

#endif //MEMORY_FROM_GLOBAL_HEAP_H
```

Раздел 14. Класс *memory_with_sorted_list_deallocation*.

```
#ifndef MEMORY_WITH_SORTED_LIST_DEALLOCATION_H
#define MEMORY_WITH_SORTED_LIST_DEALLOCATION_H

#include "../allocator/memory_base_class.h"

class memory_with_sorted_list_deallocation final : public memory
{
private:
#pragma region Allocator properties
   size_t get_allocator_service_block_size() const override;
   void * get_ptr_to_allocator_trusted_pool() const override;
#pragma endregion

#pragma region Available block methods
   void * get_first_available_block_address() const override;
   void ** get_first_available_block_address_address() const override;

   size_t get_available_block_service_block_size() const override;
   size_t get_available_block_size(void * memory_block) const override;
   void * get_next_available_block_address(void * memory_block) const override;

#pragma endregion

#pragma region Occupied block methods

   size_t get_occupied_block_service_block_size() const override;
   size_t get_size_of_occupied_block_pool(void * const occupied_block) const
   override;

#pragma endregion

public:
   memory_with_sorted_list_deallocation(memory_with_sorted_list_deallocation const&)
   = delete;            // copying
   memory_with_sorted_list_deallocation&
   operator=(memory_with_sorted_list_deallocation const&) = delete; // assignment

   memory_with_sorted_list_deallocation(size_t size, memory::Allocation_strategy
   mode, logger*, memory *);
   ~memory_with_sorted_list_deallocation() override;

   void *allocate(size_t target_size) const override;
   void deallocate(void const * const target_to_dealloc) const override;

private:
   logger *get_logger() const noexcept override;
};

#pragma region Allocator properties
size_t memory_with_sorted_list_deallocation::get_allocator_service_block_size() const
   {
   return sizeof(size_t) + sizeof(logger *) + sizeof(memory *) + sizeof(void *) +
   sizeof(memory::Allocation_strategy);
```

```
}

void * memory_with_sorted_list_deallocation::get_ptr_to_allocator_trusted_pool()
   const {
   return reinterpret_cast<void *>(get_ptr_to_ptr_to_pool_start() + 1);
}

#pragma endregion

#pragma region Available block methods
void *memory_with_sorted_list_deallocation::get_first_available_block_address() const
{
   return *get_ptr_to_ptr_to_pool_start();
}

void
   **memory_with_sorted_list_deallocation::get_first_available_block_address_address
   () const {
   return get_ptr_to_ptr_to_pool_start();
}

size_t memory_with_sorted_list_deallocation::get_available_block_service_block_size()
   const
{
   return (sizeof(void *) + sizeof(size_t));
}

size_t memory_with_sorted_list_deallocation::get_available_block_size(void
   *memory_block) const
{
   return *reinterpret_cast<size_t *>(memory_block);
}

void *memory_with_sorted_list_deallocation::get_next_available_block_address(void
   *memory_block) const {
   return *reinterpret_cast<void **>(reinterpret_cast<unsigned char *>(memory_block)
   + sizeof(size_t));
}
#pragma endregion

#pragma region Occupied block methods
size_t memory_with_sorted_list_deallocation::get_occupied_block_service_block_size()
   const {
   return sizeof(size_t);
}

size_t memory_with_sorted_list_deallocation::get_size_of_occupied_block_pool(void
   *const occupied_block) const {
   return *(reinterpret_cast<size_t *>(occupied_block) - 1) -
   get_occupied_block_service_block_size();
}
#pragma endregion

memory_with_sorted_list_deallocation::memory_with_sorted_list_deallocation(
       size_t size,
```

```
        memory::Allocation_strategy mode,
        logger * logger,
        memory * parent_allocator)
{
    size_t size_with_service_size = size + get_allocator_service_block_size();
    size_t available_block_service_block_size =
    get_available_block_service_block_size();

    if (size < available_block_service_block_size) {
        size_with_service_size += available_block_service_block_size;
        this->debug_with_guard("Requested " + std::to_string(size) + " bytes, but
    reserved "
                               + std::to_string(size_with_service_size) + " bytes for
    correct work of allocator");
    }

    if (parent_allocator != nullptr) {
        _ptr_to_allocator_metadata =
    parent_allocator->allocate(size_with_service_size);
    } else {
        try {
            _ptr_to_allocator_metadata = ::operator new(size_with_service_size);
        }
        catch (std::bad_alloc const &ex) {
            throw memory::memory_exception("An allocator with sorted list deallocation
    cannot be allocated");
        }
    }

    auto * size_of_allocator_pool = reinterpret_cast<size_t
    *>(_ptr_to_allocator_metadata);
    *size_of_allocator_pool = size;

    auto * this_allocator_logger = reinterpret_cast<class logger
    **>(size_of_allocator_pool + 1);
    *this_allocator_logger = logger;

    auto * this_allocator_parent_allocator = reinterpret_cast<memory
    **>(this_allocator_logger + 1);
    *this_allocator_parent_allocator = parent_allocator;

    auto * allocation_mode = reinterpret_cast<memory::Allocation_strategy
    *>(this_allocator_parent_allocator + 1);
    *allocation_mode = mode;

    auto * ptr_to_pool_start = reinterpret_cast<void **>(allocation_mode + 1);
    *ptr_to_pool_start = reinterpret_cast<void *>(reinterpret_cast<char
    *>(ptr_to_pool_start) + sizeof(void *));

    auto * first_available_block = *ptr_to_pool_start;
    auto * first_available_block_size = reinterpret_cast<size_t
    *>(first_available_block);
    *first_available_block_size = size;
    auto * next_to_first_available_block = reinterpret_cast<void
    **>(first_available_block_size + 1);
```

```
    *next_to_first_available_block = nullptr;

    this->trace_with_guard("memory_with_sorted_list_deallocation allocator was
    constructed");
}

memory_with_sorted_list_deallocation::~memory_with_sorted_list_deallocation() {
    this->trace_with_guard("memory_with_sorted_list_deallocation allocator was
    destructed");

    auto * size_of_allocator_pool = reinterpret_cast<size_t
    *>(_ptr_to_allocator_metadata);
    auto * this_allocator_logger = reinterpret_cast<logger **>(size_of_allocator_pool
    + 1);
    auto * this_allocator_parent_allocator = *reinterpret_cast<memory
    **>(this_allocator_logger + 1);

    if (this_allocator_parent_allocator) {
        this_allocator_parent_allocator->deallocate(_ptr_to_allocator_metadata);
    } else {
        ::operator delete(_ptr_to_allocator_metadata);
    }
}

void *memory_with_sorted_list_deallocation::allocate(size_t target_size) const {
    this->trace_with_guard("memory_with_sorted_list_deallocation::allocate method
    execution started");

    void *previous_block = nullptr, *current_block =
    get_first_available_block_address();
    void *target_block = nullptr, *previous_to_target_block = nullptr,
    *next_to_target_block = nullptr;
    auto const available_block_service_block_size =
    get_available_block_service_block_size();
    auto const occupied_block_service_block_size =
    get_occupied_block_service_block_size();
    auto const allocation_mode = *get_ptr_allocation_mode();

    while (current_block != nullptr)
    {
        auto current_block_size = get_available_block_size(current_block);
        auto next_block = get_next_available_block_address(current_block);

        if (current_block_size - occupied_block_service_block_size >= target_size)
        {
            if (allocation_mode == memory::Allocation_strategy::first_fit ||
                allocation_mode == memory::Allocation_strategy::best_fit &&
                (target_block == nullptr || current_block_size <
    get_available_block_size(target_block)) ||
                allocation_mode == memory::Allocation_strategy::worst_fit &&
                (target_block == nullptr || current_block_size >
    get_available_block_size(target_block)))
            {
                previous_to_target_block = previous_block;
                target_block = current_block;
```

```
                next_to_target_block = next_block;
            }

            if (allocation_mode == memory::Allocation_strategy::first_fit) {
                break;
            }
        }
        previous_block = current_block;
        current_block = next_block;
    }

    // if the memory block cannot be allocated
    if (target_block == nullptr)
    {
        auto const warning_message = "There is no memory available to allocate";
        this->warning_with_guard(warning_message)
            ->trace_with_guard("memory_with_sorted_list_deallocation::allocate method
    execution failed");

        throw memory::memory_exception("A block in allocator with sorted list
    deallocation cannot be allocated");
    }

    size_t leftover = get_available_block_size(target_block) -
    occupied_block_service_block_size - target_size;
    // by allocating memory block will be divided so the leftover memory cannot
    include (void *) + size_t for available block structure
    if (leftover <= available_block_service_block_size && leftover > 0)
    {
        auto requested_block_size_override = target_size + leftover;

        this->debug_with_guard("Requested " + std::to_string(target_size) + " bytes,
    but reserved "
                               + std::to_string(requested_block_size_override) + "
    bytes for correct work of allocator");

        target_size = requested_block_size_override;
    }

    void * updated_next_block_to_previous_block;

    // size of target block is precisely equal to available block size + void* (size_t
    is needed for occupied block service)
    if (leftover <= available_block_service_block_size) {
        updated_next_block_to_previous_block = next_to_target_block;
    }
    else
    {
        updated_next_block_to_previous_block = reinterpret_cast<void
    *>(reinterpret_cast<unsigned char *>(target_block) +

    get_occupied_block_service_block_size() + target_size);
        auto * const target_block_leftover_size =
    reinterpret_cast<size_t*>(updated_next_block_to_previous_block);
```

```
        // available block size is size of full available block including service part
        *target_block_leftover_size = get_available_block_size(target_block)
                                    - get_occupied_block_service_block_size() -
    target_size;
        auto * const target_block_leftover_next_block_address = reinterpret_cast<void
    **>(target_block_leftover_size + 1);

        // | target_block | leftover | available | --> merge (leftover, available)
        if (reinterpret_cast<void *>(reinterpret_cast<char *>(target_block) +
    get_available_block_size(target_block))
            == next_to_target_block)
        {
            *target_block_leftover_size +=
    get_available_block_size(next_to_target_block);
            *target_block_leftover_next_block_address =
    get_next_available_block_address(next_to_target_block);
        }
            // | target_block | leftover | occupied |
        else {
            *target_block_leftover_next_block_address = next_to_target_block;
        }
    }

    /*
     * if previous_to_target_block == nullptr, then mem_start is updated with
    target_block.next
     * else previous to target_block.next address is updated with target_block.next
    */
    previous_to_target_block == nullptr ?
            *reinterpret_cast<void **>(get_first_available_block_address_address()) =
    updated_next_block_to_previous_block :
            *reinterpret_cast<void **>(reinterpret_cast<size_t
    *>(previous_to_target_block) + 1)
            = updated_next_block_to_previous_block;

    auto *target_block_size_address = reinterpret_cast<size_t *>(target_block);
    *target_block_size_address = target_size +
    get_occupied_block_service_block_size();

    std::string target_block_address = address_to_hex(reinterpret_cast<void *>(
            reinterpret_cast<char *>(target_block) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())));

    this->information_with_guard("Block of size = " + std::to_string(target_size) + "
    was allocated. " +
                                 "Address: " + target_block_address)
        ->trace_with_guard("memory_with_sorted_list_deallocation::allocate method
    execution finished");

    auto * to_return = reinterpret_cast<void *>(target_block_size_address + 1);
    return to_return;
}

void memory_with_sorted_list_deallocation::deallocate(const void *const
    target_to_dealloc) const
```

```
{
    this->trace_with_guard("memory_with_sorted_list_deallocation::deallocate method
    execution started");

    if (!target_to_dealloc) {
        this->warning_with_guard("Target to deallocate should not be nullptr")
            ->trace_with_guard("memory_with_sorted_list_deallocation::deallocate
    method execution finished");
        return;
    }

    dump_occupied_block_before_deallocate(const_cast<void *>(target_to_dealloc));

    auto * tmp = reinterpret_cast<void *>(reinterpret_cast<size_t *>(const_cast<void
    *>(target_to_dealloc)) - 1);

    std::string target_to_dealloc_address = address_to_hex(reinterpret_cast<void *>(
                    reinterpret_cast<char *>(tmp) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())
           ));

    this->information_with_guard("memory block with address: " +
    target_to_dealloc_address + " was deallocated successfully");

    void *next_to_current_block = nullptr, *current_block =
    get_first_available_block_address();

    // current block is first block to become available
    if (current_block == nullptr) {
        // change _mem_start
        *get_first_available_block_address_address() = tmp;

        // make a target_to_deallocate.next = nullptr;
        *reinterpret_cast<void **>(reinterpret_cast<size_t *>(tmp) + 1) = nullptr;
    }

    // there is a block available previous (or before the block?) to block to be freed
    else {
        // the first available block is after target_to_dealloc
        if (current_block > tmp) {
            // make a target_to_deallocate.next = _mem_start block (previously the
    first available block)
            *reinterpret_cast<void **>(reinterpret_cast<size_t *>(tmp) + 1) =
    current_block;
            // change _mem_start
            *get_first_available_block_address_address() = tmp;
        }
        // the _mem_start is before target_to_dealloc && there might be blocks before
    target_to_dealloc
        else {
            while (current_block != nullptr) {
                next_to_current_block =
    get_next_available_block_address(current_block);
```

```
                // find_pair a position, where current block < target_to_dealloc <
    current_block.next ?= nullptr
                if (next_to_current_block == nullptr || next_to_current_block > tmp) {
                    *reinterpret_cast<void **>(reinterpret_cast<size_t *>(tmp) + 1) =
    next_to_current_block;

                    *reinterpret_cast<void **>(reinterpret_cast<size_t
    *>(current_block) + 1) = tmp;
                }
                current_block = next_to_current_block;
            }
        }
    }

    this->trace_with_guard("memory_with_sorted_list_deallocation::deallocate method
    execution finished");
}

logger *memory_with_sorted_list_deallocation::get_logger() const noexcept {
    return *get_ptr_logger_of_allocator();
}

#endif //MEMORY_WITH_SORTED_LIST_DEALLOCATION_H
```

```
#ifndef MEMORY_WITH_BOUNDARY_TAGS_H
#define MEMORY_WITH_BOUNDARY_TAGS_H

#include "../allocator/memory_base_class.h"

/* structure of occupied memory block:
       size_t size;
       void *next, *prev;
  available blocks do not store any metadata
*/

class memory_with_boundary_tags final : public memory
{
private:
#pragma region Allocator properties
   size_t get_allocator_service_block_size() const override;
   void * get_ptr_to_allocator_trusted_pool() const override;
#pragma endregion

#pragma region Occupied block methods
   void ** get_first_occupied_block_address_address() const override;

   size_t get_occupied_block_service_block_size() const override;
   size_t get_occupied_block_size(void * memory_block) const override;
   size_t get_size_of_occupied_block_pool(void * const occupied_block) const
   override;

   void * get_next_occupied_block_address(void * memory_block) const override;
   void * get_previous_occupied_block_address(void * memory_block) const override;
#pragma endregion

public:
   memory_with_boundary_tags(size_t size, memory::Allocation_strategy mode, logger*
   logger, memory * parent_allocator);
   ~memory_with_boundary_tags();

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

   void *allocate(size_t target_size) const override;
   void deallocate(void const * const target_to_dealloc) const override;

private:
   logger *get_logger() const noexcept override;
};

#pragma region Allocator properties
size_t memory_with_boundary_tags::get_allocator_service_block_size() const {
   return sizeof(size_t) + sizeof(logger *) + sizeof(memory *) + sizeof(void *) +
   sizeof(memory::Allocation_strategy);
}

void * memory_with_boundary_tags::get_ptr_to_allocator_trusted_pool() const {
```

```
    return reinterpret_cast<void *>(get_ptr_to_ptr_to_pool_start() + 1);
}
#pragma endregion

#pragma region Occupied block methods

void **memory_with_boundary_tags::get_first_occupied_block_address_address() const {
    return get_ptr_to_ptr_to_pool_start();
}

size_t memory_with_boundary_tags::get_occupied_block_service_block_size() const {
    return (sizeof(void *) * 2 + sizeof(size_t));
}

size_t memory_with_boundary_tags::get_occupied_block_size(void *memory_block) const {
    return *reinterpret_cast<size_t *>(memory_block);
}

size_t memory_with_boundary_tags::get_size_of_occupied_block_pool(void *const
    occupied_block) const {
    return get_occupied_block_size(reinterpret_cast<size_t *>(reinterpret_cast<void
    **>(occupied_block) - 2) - 1)
         - get_occupied_block_service_block_size();
}

void *memory_with_boundary_tags::get_next_occupied_block_address(void *memory_block)
    const {
    return *reinterpret_cast<void **>(reinterpret_cast<size_t *>(memory_block) + 1);
}

void *memory_with_boundary_tags::get_previous_occupied_block_address(void
    *memory_block) const {
    return *reinterpret_cast<void **>((reinterpret_cast<size_t *>(memory_block) + 1) +
    1);
}
#pragma endregion

memory_with_boundary_tags::memory_with_boundary_tags(size_t size,
                                                     memory::Allocation_strategy mode,
                                                     logger * logger,
                                                     memory * parent_allocator)
{
    size_t size_with_service_size = size + get_allocator_service_block_size(),
    occupied_block_service_block_size = get_occupied_block_service_block_size();

    if (size <= occupied_block_service_block_size) {
        size_with_service_size += occupied_block_service_block_size;
        this->debug_with_guard("Requested " + std::to_string(size) + " bytes, but
    reserved "
                               + std::to_string(size_with_service_size) + " bytes for
    correct work of allocator");
    }

    if (parent_allocator != nullptr) {
```

```
        _ptr_to_allocator_metadata =
    parent_allocator->allocate(size_with_service_size);
    } else {
        try {
            _ptr_to_allocator_metadata = ::operator new(size_with_service_size);
        }
        catch (std::bad_alloc const &) {
            throw memory::memory_exception("A constructor of allocator with boundary
    tags cannot be constructed");
        }
    }

    auto * size_of_allocator_pool = reinterpret_cast<size_t
    *>(_ptr_to_allocator_metadata);
    *size_of_allocator_pool = size;

    auto * this_allocator_logger = reinterpret_cast<class logger
    **>(size_of_allocator_pool + 1);
    *this_allocator_logger = logger;

    auto * this_allocator_parent_allocator = reinterpret_cast<memory
    **>(this_allocator_logger + 1);
    *this_allocator_parent_allocator = parent_allocator;

    auto * allocation_mode = reinterpret_cast<memory::Allocation_strategy
    *>(this_allocator_parent_allocator + 1);
    *allocation_mode = mode;

    auto * ptr_to_pool_start = reinterpret_cast<void **>(allocation_mode + 1);
    *ptr_to_pool_start = nullptr;

    this->trace_with_guard("memory_with_boundary_tags allocator was constructed");
}

memory_with_boundary_tags::~memory_with_boundary_tags() {
    this->trace_with_guard("memory_with_sorted_list_deallocation allocator was
    destructed");

    auto * size_of_allocator_pool = reinterpret_cast<size_t
    *>(_ptr_to_allocator_metadata);
    auto * this_allocator_logger = reinterpret_cast<logger **>(size_of_allocator_pool
    + 1);
    auto * this_allocator_parent_allocator = *reinterpret_cast<memory
    **>(this_allocator_logger + 1);

    if (this_allocator_parent_allocator) {
        this_allocator_parent_allocator->deallocate(_ptr_to_allocator_metadata);
    } else {
        ::operator delete(_ptr_to_allocator_metadata);
    }
}

void *memory_with_boundary_tags::allocate(size_t target_size) const {
    this->trace_with_guard("memory_with_boundary_tags::allocate method execution
    started");
```

```
    size_t size_of_allocator_pool = *get_ptr_size_of_allocator_pool();
    void *previous_block = nullptr, *current_block = *get_ptr_to_ptr_to_pool_start(),
    *target_block = nullptr;
    void * next_to_target_block = nullptr, * prev_to_target_block = nullptr;

    void *start_of_allocator_pool = get_ptr_to_allocator_trusted_pool(),
    * end_of_allocator_pool = reinterpret_cast<void *>(reinterpret_cast<char
    *>(start_of_allocator_pool) + size_of_allocator_pool);
    auto const occupied_block_service_block_size =
    get_occupied_block_service_block_size();
    auto const allocation_mode = *get_ptr_allocation_mode();
    size_t probable_target_block_size = 0, target_block_size = 0,
           size_needed = target_size + occupied_block_service_block_size;


#pragma region Finding a block of appropriate size
    // not even one block was allocated in allocator
    if (current_block == nullptr && size_of_allocator_pool >= size_needed) {
       target_block = start_of_allocator_pool;
    }
    // at least one block was allocated
    else {
        do {
            // Get size of available block
            // <start> | target_block(?) | current block | ...
            if (previous_block == nullptr && current_block != start_of_allocator_pool)
    {
                probable_target_block_size = reinterpret_cast<char *>(current_block) -
    reinterpret_cast<char *>(start_of_allocator_pool);
            }
            // == nullptr: ... | previous_block | target_block(?) | <end>
            // != nullptr: ... | previous_block | target_block(?) | current_block |
    ...
            else if (previous_block != nullptr) {
                current_block == nullptr ?
                    probable_target_block_size = reinterpret_cast<char
    *>(end_of_allocator_pool)
                        - (reinterpret_cast<char *>(previous_block) +
    get_occupied_block_size(previous_block))
                        :
                    probable_target_block_size = reinterpret_cast<char
    *>(current_block)
                        - (reinterpret_cast<char *>(previous_block) +
    get_occupied_block_size(previous_block));
            }

            if (probable_target_block_size >= size_needed)
            {
                if (allocation_mode == memory::Allocation_strategy::first_fit ||
                    allocation_mode == memory::Allocation_strategy::best_fit &&
                    (target_block == nullptr || probable_target_block_size <
    target_block_size || target_block_size == 0) ||
                    allocation_mode == memory::Allocation_strategy::worst_fit &&
```

```
                    (target_block == nullptr || probable_target_block_size >
    target_block_size || target_block_size == 0))
                {
                    if (previous_block) {
                        target_block = reinterpret_cast<void *>(reinterpret_cast<char
    *>(previous_block) +

    get_occupied_block_size(previous_block));
                    } else {
                        target_block = start_of_allocator_pool;
                    }
                    target_block_size = probable_target_block_size;
                    next_to_target_block = current_block;
                    prev_to_target_block = previous_block;
                }

                if (allocation_mode == memory::Allocation_strategy::first_fit)
                    break;
            }

            if (current_block != nullptr) {
                previous_block = current_block;
                current_block = get_next_occupied_block_address(current_block);
            } else {
                break;
            }
        } while (previous_block != nullptr);
    }
#pragma endregion

    if (target_block == nullptr) {
        this->warning_with_guard("There is no memory available to allocate")
            ->trace_with_guard("memory_with_boundary_tags::allocate method execution
    finished");

        throw memory::memory_exception("A block in allocator with boundary tags cannot
    be allocated");
    }

    size_t leftover = 0;
    // | prev | target | current | ...
    // | metadata | target | current | ...
    if (next_to_target_block) {
        leftover = reinterpret_cast<char *>(next_to_target_block) -
    (reinterpret_cast<char *>(target_block) + size_needed);
    }
    // | previous | target | -> nullptr
    if (!next_to_target_block && prev_to_target_block) {
        leftover = reinterpret_cast<char *>(end_of_allocator_pool) -
    (reinterpret_cast<char *>(target_block) + size_needed);
    }

    // by allocating memory block will be divided so the leftover memory cannot
    include (void *) + size_t for available block structure
    if (leftover <= occupied_block_service_block_size && leftover > 0)
```

```
    {
        auto target_size_override = target_size + leftover;

        this->debug_with_guard("Requested " + std::to_string(target_size) + " bytes,
    but reserved "
                               + std::to_string(target_size_override) + " bytes for
    correct work of allocator");

        size_needed += leftover;
    }

#pragma region Inserting target block in list of occupied blocks
    // ... | previous_block | target_block | current_block | ...
    // target_block.prev = previous_block;
    *reinterpret_cast<void **>((reinterpret_cast<size_t *>(target_block) + 1) + 1) =
    prev_to_target_block;
    // target_block is not right after metadata
    if (prev_to_target_block != nullptr) {
        // target_block.prev.next = target_block;
        *reinterpret_cast<void **>(reinterpret_cast<size_t *>(prev_to_target_block) +
    1) = target_block;
    } else {
        *get_first_occupied_block_address_address() = target_block;
    }

    // target_block.next = current_block
    *reinterpret_cast<void **>(reinterpret_cast<size_t *>(target_block) + 1) =
    next_to_target_block;
    // target_block is not the 'last' block allocated
    if (next_to_target_block != nullptr) {
        // target_block.next.prev = target_block;
        *reinterpret_cast<void **>((reinterpret_cast<size_t *>(next_to_target_block) +
    1) + 1) = target_block;
    }
#pragma endregion

    *reinterpret_cast<size_t *>(target_block) = size_needed;

    std::string target_block_address = address_to_hex(reinterpret_cast<void *>(
            reinterpret_cast<char *>(target_block) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())
            ));

    this->information_with_guard("Block of size = " + std::to_string(target_size) + "
    was allocated. " +
                                 "Address: " + target_block_address)
        ->trace_with_guard("memory_with_boundary_tags::allocate method execution
    finished");

    void * to_return = reinterpret_cast<void *>(reinterpret_cast<void
    **>(reinterpret_cast<size_t *>(target_block) + 1) + 2);
    return to_return;
}
```

```
void memory_with_boundary_tags::deallocate(const void *const target_to_dealloc) const
    {
    this->trace_with_guard("memory_with_boundary_tags::deallocate method execution
    started");

    if (!target_to_dealloc) {
        this->warning_with_guard("Target to deallocate should not be nullptr")
            ->trace_with_guard("memory_with_boundary_tags::deallocate method execution
    finished");
        return;
    }

    dump_occupied_block_before_deallocate(const_cast<void *>(target_to_dealloc));

    // make target_to_dealloc to point to the beginning of the block
    auto * tmp = reinterpret_cast<void *>(reinterpret_cast<size_t *>(
            reinterpret_cast<void **>(const_cast<void *>(target_to_dealloc)) - 2) -
    1);

    std::string target_to_dealloc_address = address_to_hex(reinterpret_cast<void *>(
                   reinterpret_cast<char *>(tmp) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())
           ));

    this->information_with_guard("memory block with address: " +
    target_to_dealloc_address + " was deallocated successfully");

    void *next_to_target_to_deallocate = get_next_occupied_block_address(tmp),
         *prev_to_target_to_deallocate = get_previous_occupied_block_address(tmp);

    // the target_to_deallocate is a block which _mem_start points to
    if (prev_to_target_to_deallocate == nullptr) {
        *get_first_occupied_block_address_address() = next_to_target_to_deallocate;
    }
    // the target_to_deallocate is NOT a block which _mem_start points to
    else {
        // target_to_deallocate.prev.next = target_to_deallocate.next;
        *reinterpret_cast<void **>(reinterpret_cast<size_t
    *>(prev_to_target_to_deallocate) + 1) = next_to_target_to_deallocate;
    }

    // the target_to_deallocate is not the last block occupied
    if (next_to_target_to_deallocate != nullptr) {
        // target_to_deallocate.next.prev = target_to_deallocate.prev;
        *reinterpret_cast<void **>((reinterpret_cast<size_t
    *>(next_to_target_to_deallocate) + 1) + 1) = prev_to_target_to_deallocate;
    }

    *reinterpret_cast<void **>((reinterpret_cast<size_t *>(tmp) + 1 ) + 1) = nullptr;
    *reinterpret_cast<void **>(reinterpret_cast<size_t *>(tmp) + 1) = nullptr;
    *reinterpret_cast<size_t *>(tmp) = 0;

    this->trace_with_guard("memory_with_boundary_tags::deallocate method execution
    finished");
}
```

```
logger *memory_with_boundary_tags::get_logger() const noexcept {
   return *get_ptr_logger_of_allocator();
}

#endif //MEMORY_WITH_BOUNDARY_TAGS_H
```

```
#ifndef MEMORY_WITH_BUDDY_SYSTEM_H
#define MEMORY_WITH_BUDDY_SYSTEM_H

#include "../allocator/memory_base_class.h"
#include <cmath>

/* Structure:
* allocator:
*      char power
*      logger*
*      memory* parent_allocator
*      void * pool start
*
* available_block:
*      bool is_available
*      size_t power
*      void * ptr to next
*
* occupied block:
*      bool is_available
*      size_t power
* */

class memory_with_buddy_system final : public memory
{
private:
#pragma region Allocator properties
   size_t get_allocator_service_block_size() const override;
   char * _buddy_system_get_ptr_size_of_allocator_pool() const;
   logger ** _buddy_system_get_ptr_logger_of_allocator() const;
   memory ** _buddy_system_get_ptr_to_ptr_parent_allocator() const;
   void ** _buddy_system_get_ptr_to_ptr_to_pool_start() const;
   void * get_ptr_to_allocator_trusted_pool() const override;
   [[nodiscard]] logger** get_ptr_logger_of_allocator() const override;

#pragma endregion

#pragma region Buddy system block properties
   bool * _buddy_system_is_block_available(void * block) const;
   char * _buddy_system_get_size_of_block(void * block) const;
   void ** _buddy_system_get_available_block_address_field(void * block) const;
   void * get_ptr_to_buddy(void * block, void * ptr_to_pool_start) const;
#pragma endregion

#pragma region Available block methods
   void *get_first_available_block_address() const override; // get the address of
   the first available block in allocator

   size_t get_available_block_service_block_size() const override;
   void * get_next_available_block_address(void * memory_block) const override;
#pragma endregion

#pragma region Occupied block methods
```

```
    size_t get_occupied_block_service_block_size() const override;
    size_t get_size_of_occupied_block_pool(void * const occupied_block) const
    override;
#pragma endregion

    size_t get_number_in_bin_pow(char power) const;
    char get_bin_pow_of_number(size_t number) const;

public:
    memory_with_buddy_system(memory_with_buddy_system const&) = delete;
    memory_with_buddy_system& operator= (memory_with_buddy_system const&) = delete;
    memory_with_buddy_system(char pow, logger* logger, memory* parent_allocator);
    ~memory_with_buddy_system() override;

    void *allocate(size_t target_size) const override;

    void deallocate(void const * const target_to_dealloc) const override;

private:
    logger *get_logger() const noexcept override;
};

#pragma region Allocator properties
size_t memory_with_buddy_system::get_allocator_service_block_size() const {
    return sizeof(char) + sizeof(logger *) + sizeof(memory *) + sizeof(void *);
}

char *memory_with_buddy_system::_buddy_system_get_ptr_size_of_allocator_pool() const
    {
    return reinterpret_cast<char *>(_ptr_to_allocator_metadata);
}

logger **memory_with_buddy_system::_buddy_system_get_ptr_logger_of_allocator() const
    {
    return reinterpret_cast<logger **>(_buddy_system_get_ptr_size_of_allocator_pool()
    + 1);
}

memory **memory_with_buddy_system::_buddy_system_get_ptr_to_ptr_parent_allocator()
    const {
    return reinterpret_cast<memory **>(_buddy_system_get_ptr_logger_of_allocator() +
    1);
}

void **memory_with_buddy_system::_buddy_system_get_ptr_to_ptr_to_pool_start() const {
    return reinterpret_cast<void **>(_buddy_system_get_ptr_to_ptr_parent_allocator() +
    1);
}

void *memory_with_buddy_system::get_ptr_to_allocator_trusted_pool() const {
    return reinterpret_cast<void *>(_buddy_system_get_ptr_to_ptr_to_pool_start() + 1);
}

logger **memory_with_buddy_system::get_ptr_logger_of_allocator() const {
    return _buddy_system_get_ptr_logger_of_allocator();
```

```
}
#pragma endregion

#pragma region Buddy system block properties
bool * memory_with_buddy_system::_buddy_system_is_block_available(void *block) const
    {
    return reinterpret_cast<bool *>(block);
}

char * memory_with_buddy_system::_buddy_system_get_size_of_block(void *block) const {
    return reinterpret_cast<char *>(_buddy_system_is_block_available(block) + 1);
}

void **memory_with_buddy_system::_buddy_system_get_available_block_address_field(void
    * block) const {
    return reinterpret_cast<void **>(_buddy_system_get_size_of_block(block) + 1);
}

// [ buddy   ][ our block ]
void *memory_with_buddy_system::get_ptr_to_buddy(void * block, void *
    ptr_to_pool_start) const {
    auto * block_relatively_pool = reinterpret_cast<void *>(reinterpret_cast<char
    *>(block) - reinterpret_cast<char *>(ptr_to_pool_start));

    size_t shift =
            reinterpret_cast<size_t>(block_relatively_pool) ^
    get_number_in_bin_pow(*_buddy_system_get_size_of_block(block));

    return reinterpret_cast<void *>(shift + reinterpret_cast<char
    *>(ptr_to_pool_start));
}

#pragma endregion

#pragma region Available block methods
void *memory_with_buddy_system::get_first_available_block_address() const {
    return *_buddy_system_get_ptr_to_ptr_to_pool_start();
}

size_t memory_with_buddy_system::get_available_block_service_block_size() const {
    return sizeof(bool) + sizeof(char) + sizeof(void *);
}

void *memory_with_buddy_system::get_next_available_block_address(void *memory_block)
    const {
    return *reinterpret_cast<void **>(
            reinterpret_cast<char *>(
                    reinterpret_cast<bool *>(memory_block) + 1) + 1);
}
#pragma endregion

#pragma region Occupied block methods
size_t memory_with_buddy_system::get_occupied_block_service_block_size() const {
    return sizeof(bool) + sizeof(char);
}
```

```
size_t memory_with_buddy_system::get_size_of_occupied_block_pool(void *const
   occupied_block) const {
   void * beginning_of_block = reinterpret_cast<void *>(reinterpret_cast<bool *>(
           reinterpret_cast<char *>(occupied_block) - 1) - 1);
   return get_number_in_bin_pow(*_buddy_system_get_size_of_block(beginning_of_block))
          - get_occupied_block_service_block_size();
}
#pragma endregion

size_t memory_with_buddy_system::get_number_in_bin_pow(char power) const {
   return 1 << power;
}

char memory_with_buddy_system::get_bin_pow_of_number(size_t number) const {
   return log2(number);
}

memory_with_buddy_system::memory_with_buddy_system(
       char pow,
       logger *logger = nullptr,
       memory *parent_allocator = nullptr)
{
   // can't even make a one available block (service block is needed)
   char available_block_service_block_size_power =
   get_bin_pow_of_number(get_available_block_service_block_size());
   if (pow < available_block_service_block_size_power) {
       this->debug_with_guard("Power of allocator will be set to " +
   std::to_string(available_block_service_block_size_power + 1)
                              + " for correct work of allocator");
       pow = available_block_service_block_size_power + 1;
   }

   // 2^pow >= available block service block
   size_t size_with_service_block = get_number_in_bin_pow(pow) +
   get_allocator_service_block_size();

   if (parent_allocator) {
       _ptr_to_allocator_metadata =
   parent_allocator->allocate(size_with_service_block);
   } else {
       try {
           _ptr_to_allocator_metadata = ::operator new(size_with_service_block);
       }
       catch (std::bad_alloc const &) {
           throw memory::memory_exception("An object of buddy system allocator cannot
   be constructed");
       }
   }

   auto * allocator_pool_pow = reinterpret_cast<char *>(_ptr_to_allocator_metadata);
   *allocator_pool_pow = pow;

   auto * allocator_logger = reinterpret_cast<class logger **>(allocator_pool_pow +
   1);
```

```
    *allocator_logger = logger;

    auto * allocator_parent_allocator = reinterpret_cast<memory **>(allocator_logger +
    1);
    *allocator_parent_allocator = parent_allocator;

    auto * ptr_to_memory_pool = reinterpret_cast<void **>(allocator_parent_allocator +
    1);
    *ptr_to_memory_pool = reinterpret_cast<void *>(reinterpret_cast<char
    *>(ptr_to_memory_pool) + sizeof(void *));

    void* first_block = *ptr_to_memory_pool;
    bool * is_available_first_block = reinterpret_cast<bool *>(first_block);
    *is_available_first_block = true;

    auto * power_first_block = reinterpret_cast<char *>(is_available_first_block + 1);
    *power_first_block = pow;

    void ** next_to_first_block = reinterpret_cast<void **>(power_first_block + 1);
    *next_to_first_block = nullptr;

    this->trace_with_guard("memory_with_buddy_system allocator was constructed");
}

memory_with_buddy_system::~memory_with_buddy_system()
{
    this->trace_with_guard("memory_with_sorted_list_deallocation allocator was
    destructed");

    auto * allocator_pool_pow = reinterpret_cast<char *>(_ptr_to_allocator_metadata);

    auto * allocator_logger = reinterpret_cast<logger **>(allocator_pool_pow + 1);

    auto * parent_allocator = *reinterpret_cast<memory **>(allocator_logger + 1);

    if (parent_allocator) {
        parent_allocator->deallocate(_ptr_to_allocator_metadata);
    } else {
        ::operator delete(_ptr_to_allocator_metadata);
    }
}

void *memory_with_buddy_system::allocate(size_t target_size) const {
    this->trace_with_guard("memory_with_buddy_system::allocate method execution
    started");

    if (!target_size) {
        this->warning_with_guard("Size of allocated block cannot be 0")
            ->trace_with_guard("memory_with_buddy_system::allocate method execution
    finished");
        return nullptr;
    }

    char power_needed = get_bin_pow_of_number(target_size +
    get_occupied_block_service_block_size()),
```

```
                pool_power = *_buddy_system_get_ptr_size_of_allocator_pool(),
                current_block_power = 0, target_block_power = 0;

    if (power_needed > pool_power) {
        this->warning_with_guard("No memory available to allocate, allocator pool is
    less than memory requested")
            ->trace_with_guard("memory_with_buddy_system::allocate method execution
    finished");

        throw memory::memory_exception("A block in buddy system allocator cannot be
    allocated");
    }

    void * current_block = get_first_available_block_address(), *
    previous_to_current_block = nullptr,
         * previous_to_target_block = nullptr, * target_block = nullptr;

    while (current_block != nullptr) {
        current_block_power = *_buddy_system_get_size_of_block(current_block);
        if (current_block_power >= power_needed && (target_block_power == 0 ||
    target_block_power > current_block_power))
        {
            target_block = current_block;
            target_block_power = current_block_power;
            previous_to_target_block = previous_to_current_block;
        }

        previous_to_current_block = current_block;
        current_block = get_next_available_block_address(current_block);
    }

    if (!target_block) {
        this->warning_with_guard("There is no memory available to allocate")
            ->trace_with_guard("memory_with_buddy_system::allocate method execution
    finished");

        throw memory::memory_exception("A block in buddy system allocator cannot be
    allocated");
    }

    // delete target_block from list of available blocks
    void * next_to_target_block = get_next_available_block_address(target_block);
    if (previous_to_target_block) {
        *_buddy_system_get_available_block_address_field(previous_to_target_block) =
    next_to_target_block;
    }

    if (target_block == *_buddy_system_get_ptr_to_ptr_to_pool_start()) {
        *_buddy_system_get_ptr_to_ptr_to_pool_start() = next_to_target_block;
    }

    *_buddy_system_get_available_block_address_field(target_block) = nullptr;
    *_buddy_system_is_block_available(target_block) = false;

    // divide a block until target_block_power != (read: is bigger) power_needed
```

```
    void * buddy_to_target_block = nullptr, * allocator_pool_start =
    get_ptr_to_allocator_trusted_pool();
    char minimum_power = 0, available_service_bin_power =
    get_bin_pow_of_number(get_available_block_service_block_size());
    power_needed > available_service_bin_power ? minimum_power = power_needed :
    minimum_power = available_service_bin_power;

    while (target_block_power > minimum_power) {
        target_block_power -= 1;
        *_buddy_system_get_size_of_block(target_block) = target_block_power;

        // return buddy to available blocks list
        // | prev | target block might be divided | next | -> | prev | target block |
    buddy to target block | next |
        buddy_to_target_block = get_ptr_to_buddy(target_block, allocator_pool_start);
        *_buddy_system_is_block_available(buddy_to_target_block) = true;
        *_buddy_system_get_size_of_block(buddy_to_target_block) = target_block_power;

        if (previous_to_target_block) {
            *_buddy_system_get_available_block_address_field(previous_to_target_block)
    = buddy_to_target_block;
        } else {
            *_buddy_system_get_ptr_to_ptr_to_pool_start() = buddy_to_target_block;
        }
        *_buddy_system_get_available_block_address_field(buddy_to_target_block) =
    next_to_target_block;
        next_to_target_block = buddy_to_target_block;
    }

    std::string target_block_address = address_to_hex(reinterpret_cast<void *>(
            reinterpret_cast<char *>(target_block) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())));

    this->information_with_guard("Block of size = " +
    std::to_string(get_number_in_bin_pow(target_block_power))
                                 + " was allocated." + " Address: " +
    target_block_address)
        ->trace_with_guard("memory_with_buddy_system::allocate method execution
    started");

    void * to_return = reinterpret_cast<void *>(reinterpret_cast<char
    *>(reinterpret_cast<bool *>(target_block) + 1) + 1);
    return to_return;
}

void memory_with_buddy_system::deallocate(const void *const target_to_dealloc) const
{
    this->trace_with_guard("memory_with_buddy_system::deallocate method execution
    started");

    if (!target_to_dealloc) {
        this->warning_with_guard("Target to deallocate should not be nullptr")
            ->trace_with_guard("memory_with_buddy_system::deallocate method execution
    finished");
        return;
```

```
    }

    dump_occupied_block_before_deallocate(const_cast<void *>(target_to_dealloc));

    // make target_to_dealloc to point to the beginning of the block
    auto * new_available_block = reinterpret_cast<void *>(reinterpret_cast<bool *>(
            reinterpret_cast<char *>(const_cast<void *>(target_to_dealloc)) - 1) - 1);

    std::string target_to_dealloc_address = address_to_hex(reinterpret_cast<void *>(
            reinterpret_cast<char *>(new_available_block) - reinterpret_cast<char
    *>(get_ptr_to_allocator_trusted_pool())));

    this->information_with_guard("memory block with address: " +
    target_to_dealloc_address + " was deallocated successfully");

    *_buddy_system_is_block_available(new_available_block) = true;

    void * to_delete = nullptr, * allocator_pool_start =
    get_ptr_to_allocator_trusted_pool();
    void * buddy_block = get_ptr_to_buddy(new_available_block, allocator_pool_start),
         * next_to_new_available_block = nullptr;

    char new_available_block_power =
    *_buddy_system_get_size_of_block(new_available_block),
         allocator_pool_power = *_buddy_system_get_ptr_size_of_allocator_pool();

    while (new_available_block_power != allocator_pool_power
           && *_buddy_system_is_block_available(buddy_block)
           && *_buddy_system_get_size_of_block(buddy_block) ==
    new_available_block_power)
    {
        new_available_block_power++;
        next_to_new_available_block = get_next_available_block_address(buddy_block);
        // | new_available_block | buddy_block |
        if (new_available_block < buddy_block) {
            to_delete = buddy_block;
        } else {
            // | buddy_block | new_available_block |
            to_delete = new_available_block;
            new_available_block = buddy_block;
        }
        *_buddy_system_get_available_block_address_field(to_delete) = nullptr;
        // *_buddy_system_get_size_of_block(new_available_block) =
    new_available_block_power;
        *_buddy_system_get_size_of_block(new_available_block) =
    new_available_block_power;
        buddy_block = get_ptr_to_buddy(new_available_block, allocator_pool_start);
    }
    *_buddy_system_is_block_available(new_available_block) = true;
    *_buddy_system_get_size_of_block(new_available_block) = new_available_block_power;
    *_buddy_system_get_available_block_address_field(new_available_block) =
    next_to_new_available_block;

    // change link from previous available block to new available block
```

```
    void * previous_to_new_available_block =
    *_buddy_system_get_ptr_to_ptr_to_pool_start();
    void * next_to_previous_block = nullptr;

    // make new_available_block a first block in the list
    if (previous_to_new_available_block > new_available_block) {
        *_buddy_system_get_ptr_to_ptr_to_pool_start() = new_available_block;
    }
    // new_available_block is not a first block in the list
    else if (previous_to_new_available_block != new_available_block &&
    new_available_block_power != allocator_pool_power) {
        while (previous_to_new_available_block) {
            next_to_previous_block =
    get_next_available_block_address(previous_to_new_available_block);
            if (previous_to_new_available_block < new_available_block &&
    next_to_previous_block >= new_available_block)
            {

    *_buddy_system_get_available_block_address_field(previous_to_new_available_block)
    = new_available_block;
            }
            previous_to_new_available_block = next_to_previous_block;
        }
    }

    this->trace_with_guard("memory_with_buddy_system::deallocate method execution
    finished");
}

logger *memory_with_buddy_system::get_logger() const noexcept {
    return *get_ptr_logger_of_allocator();
}

#endif //MEMORY_WITH_BUDDY_SYSTEM_H
```

```
#ifndef ASSOCIATIVE_CONTAINER_H
#define ASSOCIATIVE_CONTAINER_H

template<typename tkey, typename tvalue>
class associative_container
{
public:

   virtual ~associative_container() = default;

   virtual void insert(tkey const &key, tvalue &&value) = 0;

   virtual tvalue const &get(tkey const &key) = 0;

   virtual tvalue remove(tkey const &key) = 0;
};

#endif //ASSOCIATIVE_CONTAINER_H
```

```
#ifndef BS_TREE_H
#define BS_TREE_H

#include <stack>
#include "associative_container.h"
#include "../logger/logger.h"
#include "../logger/logger_holder.h"
#include "../allocator/memory_base_class.h"
#include "../allocator/memory_holder.h"

template<typename tkey, typename tvalue, typename tkey_comparer>
class bs_tree:
       public associative_container<tkey, tvalue>,
       protected memory_holder,
       protected logger_holder
{

public:

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

   public:

       node() = default;

       virtual ~node() noexcept = default;
   };

public:
   class insert_exception final : public std::exception {
   private:
       std::string _message;

   public:
       explicit insert_exception(std::string message)
               : _message(std::move(message)) {

       }

       [[nodiscard]] char const *what() const noexcept override {
           return _message.c_str();
       }
   };
public:
   class find_exception final : public std::exception {
   private:
       std::string _message;
```

```
    public:
        explicit find_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

    class remove_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit remove_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

    class iterator_exception final : public std::exception {
    private:
        std::string _message;

    public:
        explicit iterator_exception(std::string message)
                : _message(std::move(message)) {

        }

        [[nodiscard]] char const *what() const noexcept override {
            return _message.c_str();
        }
    };

#pragma region iterators
#pragma region preorder iterator
public:
    class prefix_iterator final
    {
    friend class bs_tree<tkey, tvalue, tkey_comparer>;

    private:
        node *_current_node;
        std::stack<node *> _path;

    public:
        explicit prefix_iterator(node *current_node)
        {
```

```
            this->_current_node = current_node;
        }

        bool operator==(prefix_iterator const &other) const
        {
            return (_current_node == other._current_node);
        }

        bool operator!=(prefix_iterator const &other) const
        {
            return (_current_node != other._current_node);
        }

        prefix_iterator& operator++()
        {
            if (_current_node == nullptr) {
                throw iterator_exception("iterator iterator is out of range");
            }

            if (_current_node->left_subtree != nullptr) {
                _path.push(_current_node);
                _current_node = _current_node->left_subtree;
            }
            else if (_current_node->right_subtree != nullptr) {
                _path.push(_current_node);
                _current_node = _current_node->right_subtree;
            }
            else {
                if (_path.empty() == true) {
                    _current_node = nullptr;
                }
                else if (_path.top()->left_subtree == _current_node) {
                    while (true) {
                        if (_path.empty() == true) {
                            _current_node = nullptr;
                            break;
                        }

                        // parent element has a right subtree
                        if (_path.top()->right_subtree != nullptr) {
                            _current_node = _path.top()->right_subtree;
                            break;
                        } else {
                            while (true) {
                                _current_node = _path.top();
                                _path.pop();

                                if (_path.empty() == true) {
                                    _current_node = nullptr;
                                    break;
                                }

                                if (_path.top()->left_subtree == _current_node &&
    _path.top()->right_subtree != nullptr) {
                                    _current_node = _path.top()->right_subtree;
```

```
                                    break;
                                }
                            }
                        }

                    }
                }
                else if (_path.top()->right_subtree == _current_node) {
                    while (true) {
                        _current_node = _path.top();
                        _path.pop();

                        if (_path.empty() == true) {
                            _current_node = nullptr;
                            break;
                        }

                        if (_path.top()->left_subtree == _current_node &&
    _path.top()->right_subtree != nullptr) {
                            _current_node = _path.top()->right_subtree;
                            break;
                        }
                    }
                }
            }

            return *this;
        }

        prefix_iterator operator++(int not_used)
        {
            auto result = *this;
            ++*this;
            return *this;
        }

        std::tuple<unsigned int, tkey const&, tvalue const&> operator*() const
        {
            return std::tuple<unsigned int, const tkey &, const tvalue
    &>(_path.size(), _current_node->key, _current_node->value);
        }

    };
#pragma endregion

#pragma region inorder iterator
public:
    class infix_iterator final
    {
        friend class bs_tree<tkey, tvalue, tkey_comparer>;

    private:
        node *_current_node;
        std::stack<node *> _path;
```

```
public:
    explicit infix_iterator(node *current_node)
            : _current_node(current_node)
    {
        if (_current_node == nullptr) {
            return;
        }

        while (_current_node->left_subtree != nullptr) {
            _path.push(_current_node);
            _current_node = _current_node->left_subtree;
        }
    }

    bool operator==(infix_iterator const &other) const
    {
        return (_current_node == other._current_node);
    }

    bool operator!=(infix_iterator const &other) const
    {
        return (_current_node != other._current_node);
    }

    infix_iterator& operator++()
    {
        // левое--корень--правое
        if (_current_node == nullptr) {
            throw iterator_exception("infix_iterator iterator is out of range");
        }

        if (_current_node->right_subtree != nullptr) {
            _path.push(_current_node);
            _current_node = _current_node->right_subtree;
            while (_current_node->left_subtree != nullptr) {
                _path.push(_current_node);
                _current_node = _current_node->left_subtree;
            }
        }
        else if (_path.empty()) {
            _current_node = nullptr;
        }
        else if (_path.top()->left_subtree == _current_node) {
            _current_node = _path.top();
            _path.pop();
        }
        else if (_path.top()->right_subtree == _current_node) {
            while (_path.top()->right_subtree == _current_node) {
                _current_node = _path.top();
                _path.pop();

                if (_path.empty()) {
                    _current_node = nullptr;
                    break;
                }
```

```
                }

                if (!_path.empty()) {
                    _current_node = _path.top();
                    _path.pop();
                }
            }

            return *this;
        }

        infix_iterator operator++(int not_used)
        {
            auto result = *this;
            ++*this;
            return *this;
        }

        std::tuple<unsigned int, tkey const&, tvalue const&> operator*() const
        {
            return std::tuple<unsigned int, const tkey &, const tvalue
    &>(_path.size(), _current_node->key, _current_node->value);
        }
    };

#pragma endregion

#pragma region postorder iterator
public:
    class postfix_iterator final
    {
        friend class bs_tree<tkey, tvalue, tkey_comparer>;

    private:
        node *_current_node;
        std::stack<node *> _path;
    public:
        explicit postfix_iterator(node *current_node)
                : _current_node(current_node)
        {
            // лево-право-текущий
            if (_current_node == nullptr) {
                return;
            }

            while (_current_node->right_subtree || _current_node->left_subtree) {
                _path.push(_current_node);
                _current_node = _current_node->left_subtree != nullptr ?
                                _current_node->left_subtree :
                                _current_node->right_subtree;
            }
        }

        bool operator==(postfix_iterator const &other) const
        {
```

```
            return (_current_node == other._current_node);
        }

        bool operator!=(postfix_iterator const &other) const
        {
            return (_current_node != other._current_node);
        }

        postfix_iterator &operator++()
        {
            // если можно идти налево, идём, если можно направо, идём
            if (_current_node == nullptr) {
                throw iterator_exception("postfix_iterator iterator is out of range");
            }

            if (_path.empty() == true) {
                _current_node = nullptr;
            }
                // поднимаемся слева
            else if (_path.top()->left_subtree == _current_node) {
                if (_path.top()->right_subtree != nullptr) {
                    _current_node = _path.top()->right_subtree;
                    while (_current_node->right_subtree ||
    _current_node->left_subtree) {
                        _path.push(_current_node);
                        _current_node = _current_node->left_subtree != nullptr ?
                                        _current_node->left_subtree :
                                        _current_node->right_subtree;
                    }
                } else {
                    _current_node = _path.top();
                    _path.pop();
                }
            }
                // поднимаемся справа
            else {
                _current_node = _path.top();
                _path.pop();
            }
            return *this;
        }

        postfix_iterator operator++(int not_used)
        {
            auto result = *this;
            ++*this;
            return *this;
        }

        std::tuple<unsigned int, tkey const&, tvalue const&> operator*() const
        {
            return std::tuple<unsigned int, const tkey &, const tvalue
    &>(_path.size(), _current_node->key, _current_node->value);
        }
```

```
    };
#pragma endregion

public:
    prefix_iterator begin_prefix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::prefix_iterator(_root);
    }

    prefix_iterator end_prefix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::prefix_iterator(nullptr);
    }

    infix_iterator begin_infix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::infix_iterator(_root);
    }

    infix_iterator end_infix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::infix_iterator(nullptr);
    }

    postfix_iterator begin_postfix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::postfix_iterator(_root);
    }

    postfix_iterator end_postfix() const noexcept
    {
        return bs_tree<tkey, tvalue, tkey_comparer>::postfix_iterator(nullptr);
    }

#pragma endregion

protected:
#pragma region template methods
    class template_method_basics:
            protected logger_holder
    {
        friend class bs_tree<tkey, tvalue, tkey_comparer>;

    protected:

    bs_tree<tkey, tvalue, tkey_comparer> *_target_tree;
    node * get_root_node()
    {
        return _target_tree->_root;
    }

    public:

        explicit template_method_basics(
                bs_tree<tkey, tvalue, tkey_comparer> *target_tree)
```

```
                : _target_tree(target_tree)
        {

        }

    public:

        std::pair<std::stack<node **>, node **> find_path(tkey const &key) const
        {
            std::stack<node **> path;

            if (_target_tree->_root == nullptr)
            {
                return { path, &_target_tree->_root };
            }

            auto **iterator = &_target_tree->_root;
            tkey_comparer comparer;

            while ((*iterator) != nullptr)
            {
                auto comparison_result = comparer(key, (*iterator)->key);
                if (comparison_result == 0)
                {
                    return std::pair<std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **>,
                                typename bs_tree<tkey, tvalue,
    tkey_comparer>::node**>(path, iterator);
                }

                path.push(iterator);
                iterator = comparison_result < 0
                           ? &(*iterator)->left_subtree
                           : &(*iterator)->right_subtree;
            }

            return { path, iterator };
        }

        node** find_parent(std::stack<node **> &path, node **target_ptr) const
        {
            return path.empty() ? nullptr : path.top();
        }

        node** find_grandparent(std::stack<node **> &path, node **target_ptr) const
        {
            if (path.empty())
                return nullptr;
            auto ** parent = path.top();
            path.pop();
            if (path.empty()) {
                path.push(parent);
                return nullptr;
            }
            auto ** grandparent = path.top();
```

```
            path.push(parent);
            return grandparent;
        }

    public:
        virtual void rotate_fix_additional_data(node * target_ptr) const
        {

        }

    public:
        node** rotate_left(std::stack<node **> &path, node **target_ptr) const
        {
            node ** parent = find_parent(path, target_ptr);
            path.pop();

            node * left_to_target_ptr = (*target_ptr)->left_subtree;

            (*target_ptr)->left_subtree = (*parent);
            (*parent) = (*target_ptr);

            (*target_ptr)->left_subtree->right_subtree = left_to_target_ptr;

            this->rotate_fix_additional_data(*parent);
            return parent;
        }

        node** rotate_right(std::stack<node **> &path, node **target_ptr) const
        {
            node ** parent = find_parent(path, target_ptr);
            path.pop();

            node * right_to_target_ptr = (*target_ptr)->right_subtree;

            (*target_ptr)->right_subtree = (*parent);
            (*parent) = (*target_ptr);

            (*target_ptr)->right_subtree->left_subtree = right_to_target_ptr;

            this->rotate_fix_additional_data(*parent);
            return parent;
        }

    private:

        [[nodiscard]] logger *get_logger() const noexcept override
        {
            return _target_tree->get_logger();
        }

    };

#pragma region insertion template method
    class insertion_template_method :
            public template_method_basics,
```

```
        private memory_holder
{

public:

    explicit insertion_template_method(
            bs_tree<tkey, tvalue, tkey_comparer> *target_tree)
            : template_method_basics(target_tree)
    {

    }

public:
    void insert(tkey const &key, tvalue &&value)
    {
        this->trace_with_guard("bs_tree::insertion_template_method::insert method
started");
        auto path_and_target = this->find_path(key);
        auto path = path_and_target.first;
        auto **target_ptr = path_and_target.second;

        if (*target_ptr != nullptr)
        {
            this->debug_with_guard("bs_tree::insertion_template_method::insert
passed key is not unique")
                ->trace_with_guard("bs_tree::insertion_template_method::insert
method finished");
            throw insert_exception("bs_tree::insertion_template_method::insert
passed key is not unique");
        }

        *target_ptr = reinterpret_cast<node
*>(allocate_with_guard(get_node_size()));
        initialize_memory_with_node(*target_ptr);

        (*target_ptr)->key = key;
        (*target_ptr)->value = std::move(value);
        (*target_ptr)->left_subtree = nullptr;
        (*target_ptr)->right_subtree = nullptr;

        after_insert_inner(path, target_ptr);
        this->trace_with_guard("bs_tree::insertion_template_method::insert method
finished");
    }

protected:

    [[nodiscard]] virtual size_t get_node_size() const
    {
        return sizeof(node);
    }

    virtual void initialize_memory_with_node(node *target_ptr) const
    {
        new(target_ptr) node;
```

```
        }

        virtual void after_insert_inner(std::stack<node **> &path, node **target_ptr)
        {

        }

    private:

        [[nodiscard]] memory *get_memory() const noexcept override
        {
            return this->_target_tree->_allocator;
        }

    };
#pragma endregion

#pragma region finding template method
    class finding_template_method :
            public template_method_basics
    {

    public:
        explicit finding_template_method(bs_tree<tkey, tvalue, tkey_comparer>
    *target_tree)
                : template_method_basics(target_tree)
        {

        }

    public:
        tvalue const &find(tkey const &key)
        {
            this->trace_with_guard("bs_tree::finding_template_method::find method
    started");
            auto path_and_target = this->find_path(key);
            auto path = path_and_target.first;
            auto **target_ptr = path_and_target.second;

            if (*target_ptr == nullptr)
            {
                this->debug_with_guard("bs_tree::finding_template_method::find no
    value with passed key in tree")
                    ->trace_with_guard("bs_tree::finding_template_method::find method
    finished");
                throw find_exception("bs_tree::finding_template_method::find no value
    with passed key in tree");
            }

            after_find_inner(path, target_ptr);

            this->trace_with_guard("bs_tree::finding_template_method::find method
    finished");
            return (*target_ptr)->value;
        }
```

```
    protected:

        virtual void after_find_inner(std::stack<node **> &path, node **&target_ptr)
        {
            // TODO: nothing to do here in BST context...
        }

    };

#pragma endregion

#pragma region removing template method
    class removing_template_method:
            public template_method_basics,
            private memory_holder
    {

    public:

        explicit removing_template_method(
                bs_tree<tkey, tvalue, tkey_comparer> *target_tree)
                : template_method_basics(target_tree)
        {

        }

    public:

        virtual tvalue remove(tkey const &key)
        {
            this->trace_with_guard("bs_tree::removing_template_method::remove method
    started");
            auto path_and_target = this->find_path(key);
            auto path = path_and_target.first;
            auto **target_ptr = path_and_target.second;

            if (*target_ptr == nullptr)
            {
                this->debug_with_guard("bs_tree::removing_template_method::remove no
    value with passed key in tree")
                    ->trace_with_guard("bs_tree::removing_template_method::remove
    method finished");
                throw remove_exception("bs_tree::removing_template_method::remove no
    value with passed key in tree");
            }

            // здесь могла быть move-семантика
            tvalue result = (*target_ptr)->value;

            if ((*target_ptr)->left_subtree != nullptr &&
                (*target_ptr)->right_subtree != nullptr)
            {
                this->debug_with_guard("bs_tree::removing_template_method::remove
    deleting an element with 2 children");
```

```
            auto **element_to_swap_with = &(*target_ptr)->left_subtree;
            path.push(target_ptr);

            bool element_to_swap_with_has_non_null_right_subtree = false;
            std::stack<node **> swap_stack;

            if ((*element_to_swap_with)->right_subtree != nullptr) {
                element_to_swap_with_has_non_null_right_subtree = true;
                path.push(element_to_swap_with);
                element_to_swap_with = &((*element_to_swap_with)->right_subtree);

                bool local_element_to_swap_with_has_non_null_right_subtree =
false;
                do
                {
                    if ((*element_to_swap_with)->right_subtree != nullptr) {
                        local_element_to_swap_with_has_non_null_right_subtree =
true;
                        swap_stack.push(element_to_swap_with);
                        element_to_swap_with =
&((*element_to_swap_with)->right_subtree);
                    } else {
                        local_element_to_swap_with_has_non_null_right_subtree =
false;
                    }
                } while (local_element_to_swap_with_has_non_null_right_subtree);
            }

            target_ptr = swap_nodes(element_to_swap_with, target_ptr);

            if (element_to_swap_with_has_non_null_right_subtree) {
                path.pop();
                path.push(&((*(path.top()))->left_subtree));

                std::stack<node **> reverse_swap_stack;
                while (!(swap_stack.empty())) {
                    reverse_swap_stack.push(swap_stack.top());
                    swap_stack.pop();
                }

                while (!(reverse_swap_stack.empty())) {
                    path.push(reverse_swap_stack.top());
                    reverse_swap_stack.pop();
                }
            }
        }

        if ((*target_ptr)->left_subtree == nullptr &&
            (*target_ptr)->right_subtree == nullptr)
        {
            this->debug_with_guard("bs_tree::removing_template_method::remove
deleting an element with no children");
            cleanup_node(target_ptr);
        }
        else if ((*target_ptr)->left_subtree != nullptr)
```

```
        {
            this->debug_with_guard("bs_tree::removing_template_method::remove
deleting an element with 1 left child");
            auto *target_left_subtree = (*target_ptr)->left_subtree;
            cleanup_node(target_ptr);
            *target_ptr = target_left_subtree;
        }
        else
        {
            this->debug_with_guard("bs_tree::removing_template_method::remove
deleting an element with 1 right child");
            auto *target_right_subtree = (*target_ptr)->right_subtree;
            cleanup_node(target_ptr);
            *target_ptr = target_right_subtree;
        }

        if (!(path.empty())) {
            after_remove(path);
        }

        this->trace_with_guard("bs_tree::removing_template_method::remove method
finished");
        return result;
    }

protected:

    template<typename T>
    void swap(T **left, T **right)
    {
        T *temp = *left;
        *left = *right;
        *right = temp;
    }

    node** swap_nodes(node **one_node, node **another_node)
    {
        // близкородственный свап до хорошего не доводит
        if ((*another_node)->left_subtree == (*one_node)) {
            node * grandfather = (*another_node);
            node * grandchild = (*one_node)->left_subtree;
            (*another_node) = (*one_node);
            (*another_node)->left_subtree = grandfather;
            (*another_node)->right_subtree = grandfather->right_subtree;
            (*another_node)->left_subtree->left_subtree = grandchild;
            (*another_node)->left_subtree->right_subtree = nullptr;
            return (&(*another_node)->left_subtree);
        } else {
            swap(&((*one_node)->left_subtree), &((*another_node)->left_subtree));
            swap(&((*one_node)->right_subtree),
&((*another_node)->right_subtree));

            swap_additional_data(*one_node, *another_node);

            swap(one_node, another_node);
```

```
                return one_node;
            }
        }

        void cleanup_node(node **node_address)
        {
            (*node_address)->~node();
            deallocate_with_guard(reinterpret_cast<void *>(*node_address));

            *node_address = nullptr;
        }

    protected:

        virtual void swap_additional_data(node *one_node, node *another_node)
        {

        }

        virtual void after_remove(std::stack<node **> &path) const
        {

        }

    private:

        memory *get_memory() const noexcept override
        {
            return this->_target_tree->_allocator;
        }

    };

#pragma endregion
#pragma endregion

    node *_root;
protected:
#pragma region rool 5
    // constructor
    bs_tree(
            logger *logger,
            memory *allocator,
            insertion_template_method *insertion,
            finding_template_method *finding,
            removing_template_method *removing)
            : _logger(logger),
              _allocator(allocator),
              _insertion(insertion),
              _finding(finding),
              _removing(removing),
              _root(nullptr)
    {
        this->trace_with_guard("bs_tree constructor was called");
    }
```

```
protected:
   logger *_logger;
   memory *_allocator;
   insertion_template_method *_insertion;
   finding_template_method *_finding;
   removing_template_method *_removing;

public:
   // copy constructor
   bs_tree(bs_tree<tkey, tvalue, tkey_comparer> const &obj)
           : bs_tree(obj._logger, obj._allocator)
   {
       this->trace_with_guard("bs_tree copy constructor was called");
       _root = copy(obj._root);
   }

   // move constructor
   bs_tree(bs_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
           : bs_tree(obj._insertion,
                     obj._finding,
                     obj._removing,
                     obj._allocator,
                     obj._logger)
   {
       this->trace_with_guard("bs_tree move constructor was called");
       _root = obj._root;
       obj._root = nullptr;

       _insertion->_target_tree = this;
       obj._insertion = nullptr;

       _finding->_target_tree = this;
       obj._finding = nullptr;

       _removing->_target_tree = this;
       obj._removing = nullptr;

       obj._allocator = nullptr;

       obj._logger = nullptr;
   }

   // copy assignment (оператор присваивания)
   bs_tree &operator=(bs_tree<tkey, tvalue, tkey_comparer> const &obj)
   {
       this->trace_with_guard("bs_tree copy assignment constructor was called");
       if (this == &obj)
       {
           return *this;
       }

       clearup(_root);

       _allocator = obj._allocator;
```

```
        _logger = obj._logger;

        _root = copy(obj._root);

        return *this;
    }

    // move assignment (оператор присваивания перемещением)
    bs_tree &operator=(bs_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    {
        this->trace_with_guard("bs_tree move assignment constructor was called");
        if (this == &obj)
        {
            return *this;
        }

        clearup(_root);
        _root = obj._root;
        obj._root = nullptr;

        delete obj._insertion;
        obj._insertion = nullptr;

        delete obj._finding;
        obj._finding = nullptr;

        delete obj._removing;
        obj._removing = nullptr;

        _allocator = obj._allocator;
        obj._allocator = nullptr;

        _logger = obj._logger;
        obj._logger = nullptr;

        return *this;
    }

    // destructor
    ~bs_tree()
    {
        this->trace_with_guard("bs_tree destructor was called");
        delete _insertion;
        delete _finding;
        delete _removing;

        clearup(_root);
    }

protected:
    virtual void clearup(node *element)
    {
        if (element == nullptr)
        {
            return;
```

```
        }

        clearup(element->left_subtree);
        clearup(element->right_subtree);

        element->~node();
        deallocate_with_guard(element);
    }

    virtual node *copy(node *from)
    {
        if (from == nullptr)
        {
            return nullptr;
        }

        node *result = reinterpret_cast<node *>(allocate_with_guard(sizeof(node)));
        new (result) node(*from);

        result->left_subtree = copy(from->left_subtree);
        result->right_subtree = copy(from->right_subtree);

        return result;
    }

public:
    // constructor
    explicit bs_tree(
            logger *logger = nullptr,
            memory *allocator = nullptr)
            : bs_tree(logger,
                      allocator,
                      new insertion_template_method(this),
                      new finding_template_method(this),
                      new removing_template_method(this))
    {

    }
#pragma endregion

public:

    void insert(
            tkey const &key,
            tvalue &&value) override
    {
        _insertion->insert(key, std::move(value));
    }

    tvalue const &get(
            tkey const &key) override
    {
        return _finding->find(key);
    }
```

```
    tvalue remove(
            tkey const &key) override
    {
        return _removing->remove(key);
    }

private:

    [[nodiscard]] memory *get_memory() const noexcept override
    {
        return _allocator;
    }

private:

    [[nodiscard]] logger *get_logger() const noexcept override
    {
        return _logger;
    }

};

#endif //BS_TREE_H
```

```
#ifndef AVL_TREE_H
#define AVL_TREE_H

#include "../binary_tree/bs_tree.h"

template<
       typename tkey,
       typename tvalue,
       typename tkey_comparer>
class avl_tree final
       : public bs_tree<tkey, tvalue, tkey_comparer>
{
protected:
   struct avl_node
           : bs_tree<tkey, tvalue, tkey_comparer>::node
   {
   protected:
       unsigned height;

   public:
       unsigned int get_height()
       {
           return (this == nullptr ? 0 : height);
       }

       void set_height(unsigned new_h)
       {
           this->height = new_h;
       }

       int get_balance()
       {
           return (this == nullptr ? 0 : reinterpret_cast<avl_node
   *>(this->left_subtree)->get_height() -
                                         reinterpret_cast<avl_node
   *>(this->right_subtree)->get_height());
       }

   public:
       unsigned get_max_height_of_two_nodes(avl_node * another)
       {
           unsigned this_h = this->get_height(), another_h = another->get_height();
           return (this_h > another_h ? this_h : another_h);
       }

       void update_height()
       {
           height = reinterpret_cast<avl_node
   *>(this->left_subtree)->get_max_height_of_two_nodes(reinterpret_cast<avl_node
   *>(this->right_subtree)) + 1;
       }
   };
```

```
protected:
#pragma region template methods avl tree
   class template_methods_avl :
       public bs_tree<tkey, tvalue, tkey_comparer>::template_method_basics
   {
   public:
       void do_balance(std::stack<typename bs_tree<tkey, tvalue, tkey_comparer>::node
   **> &path, avl_node **target_ptr)
       {
           avl_node ** current_node = target_ptr;
           int balance, left_subtree_balance, right_subtree_balance;

           do {
               balance = (*current_node)->get_balance();

               left_subtree_balance = reinterpret_cast<avl_node
   *>((*current_node)->left_subtree)->get_balance();
               right_subtree_balance = reinterpret_cast<avl_node
   *>((*current_node)->right_subtree)->get_balance();

               if (balance == 2) { // == 2
                   path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
   tkey_comparer>::node **>(current_node));
                   // returned by rotations values are current top node
                   if (left_subtree_balance >= 0) {
                       this->rotate_right(path, &((*current_node)->left_subtree));
                   } else {
                       path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
   tkey_comparer>::node **>(&((*current_node)->left_subtree)));
                       this->rotate_left(path,
   &((*current_node)->left_subtree->right_subtree));
                       reinterpret_cast<avl_node
   *>((*current_node)->left_subtree->left_subtree)->update_height();
                       reinterpret_cast<avl_node
   *>((*current_node)->left_subtree)->update_height();

                       this->rotate_right(path, &((*current_node)->left_subtree));
                   }
                   reinterpret_cast<avl_node
   *>((*current_node)->right_subtree)->update_height();
                   reinterpret_cast<avl_node *>((*current_node))->update_height();

               } else if (balance == -2) {
                   path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
   tkey_comparer>::node **>(current_node));
                   if (right_subtree_balance <= 0) {
                       this->rotate_left(path, &((*current_node)->right_subtree));
                   } else {
                       path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
   tkey_comparer>::node **>(&((*current_node)->right_subtree)));
                       this->rotate_right(path,
   &((*current_node)->right_subtree->left_subtree));
                       reinterpret_cast<avl_node
   *>((*current_node)->right_subtree->right_subtree)->update_height();
```

```
                        reinterpret_cast<avl_node
    *>((*current_node)->right_subtree)->update_height();

                        this->rotate_left(path, &((*current_node)->right_subtree));
                    }
                    reinterpret_cast<avl_node
    *>((*current_node)->left_subtree)->update_height();
                    reinterpret_cast<avl_node *>((*current_node))->update_height();
                }

                if (path.empty()) {
                    break;
                }
                current_node = reinterpret_cast<avl_node **>(path.top());
                path.pop();
                (*current_node)->update_height();
            } while (true);
        }

    public:
        explicit template_methods_avl(
                avl_tree<tkey, tvalue, tkey_comparer> *target_tree)
                : bs_tree<tkey, tvalue,
    tkey_comparer>::template_method_basics(target_tree)
        {

        }
    };

#pragma region insertion avl tree
    class insertion_avl_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::insertion_template_method
    {
        size_t get_node_size() const override
        {
            return sizeof(avl_node);
        }

        void initialize_memory_with_node(
                typename bs_tree<tkey, tvalue, tkey_comparer>::node *target_ptr) const
    override
        {
            new(reinterpret_cast<avl_node *>(target_ptr)) avl_node;
        }

        void after_insert_inner(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path, typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **target_ptr) override
        {
            this->trace_with_guard("avl_tree::insertion_avl_tree::after_insert_inner
    method started");
            // 1) insert additional data to this node
            reinterpret_cast<avl_node *>((*target_ptr))->set_height(1);
```

```
            avl_node ** parent = reinterpret_cast<avl_node **>(this->find_parent(path,
    target_ptr));
            if (parent == nullptr) {
                return;
            }
            (*parent)->update_height();

    //            2) go to grandparent of this target_ptr (if it exists)
            auto ** grandparent = reinterpret_cast<avl_node
    **>(this->find_grandparent(path, target_ptr));
            if (grandparent == nullptr) {
                return;
            }
            (*grandparent)->update_height();
            path.pop();
            path.pop();

    //            3) do_balance
            this->trace_with_guard("avl_tree::template_methods_avl::do_balance method
    started");
            reinterpret_cast<template_methods_avl *>(this)->do_balance(path,
    grandparent);
            this->trace_with_guard("avl_tree::template_methods_avl::do_balance method
    finished")
                ->trace_with_guard("avl_tree::insertion_avl_tree::after_insert_inner
    method finished");
        }

    public:
        explicit insertion_avl_tree(avl_tree<tkey, tvalue, tkey_comparer> *
    target_tree)
                : bs_tree<tkey, tvalue,
    tkey_comparer>::insertion_template_method(target_tree)
        {

        }
    };

    #pragma endregion

    #pragma region removing avl tree
    class removing_avl_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::removing_template_method
    {
        void after_remove(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path) const override
        {
            this->trace_with_guard("avl_tree::removing_avl_tree::after_remove method
    started");
            // path содержит все узлы до удалённого, верхний узел в стеке -- родитель
    удалённого
            // 1) прийти к родителю
            if (path.empty()) {
                return;
            }
```

```
            avl_node ** parent = reinterpret_cast<avl_node **>(path.top());
            (*parent)->update_height();
            path.pop();

            // 2) do_balance
            this->trace_with_guard("avl_tree::template_methods_avl::do_balance method
    started");
            reinterpret_cast<template_methods_avl *>(const_cast<removing_avl_tree
    *>(this))->do_balance(path, parent);
            this->trace_with_guard("avl_tree::template_methods_avl::do_balance method
    finished")
                ->trace_with_guard("avl_tree::removing_avl_tree::after_remove method
    started");
        }

        void swap_additional_data(
                typename bs_tree<tkey, tvalue, tkey_comparer>::node *one_node,
                typename bs_tree<tkey, tvalue, tkey_comparer>::node *another_node)
    override
        {
            avl_node * one = reinterpret_cast<avl_node *>(one_node), *second =
    reinterpret_cast<avl_node *>(another_node);
            unsigned height_tmp = one->get_height();
            one->set_height(second->get_height());
            second->set_height(height_tmp);
        }

    public:
        explicit removing_avl_tree(avl_tree<tkey, tvalue, tkey_comparer> *
    target_tree)
            : bs_tree<tkey, tvalue,
    tkey_comparer>::removing_template_method(target_tree)
        {

        }
    };

#pragma endregion
#pragma endregion

#pragma region rule 5
public:
    explicit avl_tree(
            logger *_logger = nullptr,
            memory *_allocator = nullptr)
            : bs_tree<tkey, tvalue, tkey_comparer>(
                _logger,
                _allocator,
                new insertion_avl_tree(this),
                new typename bs_tree<tkey, tvalue,
    tkey_comparer>::finding_template_method(this),
                new removing_avl_tree(this))
    {
    }
```

```
private:
    // constructor
    avl_tree(
            logger *logger,
            memory *allocator,
            insertion_avl_tree *insertion,
            typename bs_tree<tkey, tvalue, tkey_comparer>::finding_template_method
    *finding,
            removing_avl_tree *removing)
    {
        this->_logger(logger);
        this->_allocator(allocator);
        this->_insertion(insertion);
        this->_finding(finding);
        this->_removing(removing);
        this->_root(nullptr);

        this->trace_with_guard("avl_tree constructor was called");
    }

public:
    // copy constructor
    avl_tree(avl_tree<tkey, tvalue, tkey_comparer> const &obj)
            : avl_tree(obj._logger, obj._allocator)
    {
        this->trace_with_guard("avl_tree copy constructor was called");
        this->_root = this->copy(obj._root);
    }

    // move constructor
    avl_tree(avl_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    : avl_tree(obj._insertion,
               obj._finding,
               obj._removing,
               obj._allocator,
               obj._logger)
    {
        this->trace_with_guard("avl_tree move constructor was called");

        this->_root = obj._root;
        obj._root = nullptr;

        this->_insertion->_target_tree = this;
        obj._insertion = nullptr;

        this->_finding->_target_tree = this;
        obj._finding = nullptr;

        this->_removing->_target_tree = this;
        obj._removing = nullptr;

        obj._allocator = nullptr;

        obj._logger = nullptr;
    }
```

```
// copy assignment (оператор присваивания)
avl_tree &operator=(avl_tree<tkey, tvalue, tkey_comparer> const &obj)
{
    this->trace_with_guard("avl_tree copy assignment constructor was called");

    if (this == &obj)
    {
        return *this;
    }

    this->clearup(this->_root);

    this->_allocator = obj._allocator;
    this->_logger = obj._logger;
    this->_root = this->copy(obj._root);

    return *this;
}

// move assignment (оператор присваивания перемещением)
avl_tree &operator=(avl_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
{
    this->trace_with_guard("avl_tree move assignment constructor was called");

    if (this == &obj)
    {
        return *this;
    }

    this->clearup(this->_root);
    this->_root = obj._root;
    obj._root = nullptr;

    delete obj._insertion;
    obj._insertion = nullptr;

    delete obj._finding;
    obj._finding = nullptr;

    delete obj._removing;
    obj._removing = nullptr;

    this->_allocator = obj._allocator;
    obj._allocator = nullptr;

    this->_logger = obj._logger;
    obj._logger = nullptr;

    return *this;
}

// destructor
~avl_tree()
{
```

```
        this->trace_with_guard("avl_tree destructor was called");
    }

private:
    typename bs_tree<tkey, tvalue, tkey_comparer>::node *copy(typename bs_tree<tkey,
    tvalue, tkey_comparer>::node *from) override
    {
        if (from == nullptr)
        {
            return nullptr;
        }

        avl_node *result = reinterpret_cast<avl_node
    *>(this->allocate_with_guard(sizeof(avl_node)));
        new (result) avl_node(*reinterpret_cast<avl_node *>(from));
        result->set_height(reinterpret_cast<avl_node *>(from)->get_height());

        result->left_subtree = copy(from->left_subtree);
        result->right_subtree = copy(from->right_subtree);

        return result;
    }

#pragma endregion

};

#endif //AVL_TREE_H
```

```
#ifndef SPLAY_TREE_H
#define SPLAY_TREE_H

#include "../binary_tree/bs_tree.h"

template<
       typename tkey,
       typename tvalue,
       typename tkey_comparer>
class splay_tree final
       : public bs_tree<tkey, tvalue, tkey_comparer>
{
public:
#pragma region template methods splay tree

   class template_method_splay final
           : public bs_tree<tkey, tvalue, tkey_comparer>::template_method_basics
   {
   public:
       void splay(std::stack<typename bs_tree<tkey, tvalue, tkey_comparer>::node **>
   &path, typename bs_tree<tkey, tvalue, tkey_comparer>::node **&target_ptr) const
       {
           typename bs_tree<tkey, tvalue, tkey_comparer>::node ** parent,
   **grandparent = nullptr;
           typename bs_tree<tkey, tvalue, tkey_comparer>::node *tree_root =
   (reinterpret_cast<splay_tree<tkey, tvalue, tkey_comparer>
   *>(this->_target_tree))->_root;
           typename bs_tree<tkey, tvalue, tkey_comparer>::node ** current_node =
   target_ptr;

           while ((*current_node) != tree_root) {
               parent = this->find_parent(path, current_node);

               if (tree_root == (*parent)) {
                   // zig
                   // rotate right/left: target_ptr becomes new _root if its
   grandparent is nullptr
                   if ((*parent)->left_subtree == (*current_node)) {
                       this->rotate_right(path, current_node);
                   } else {
                       this->rotate_left(path, current_node);
                   }
                   target_ptr = parent;
//                    (*target_ptr) = (*parent);
                   break;
               }
               else {
                   grandparent = this->find_grandparent(path, current_node);
                   if ((*parent)->left_subtree == (*current_node) &&
   (*grandparent)->left_subtree == (*parent)) {
                       // zig-zig: rotate_right(parent) + rotate_right(target_ptr)
                       // stack: parent -> grandparent -> ...
                       path.pop();
```

```
                        parent = this->rotate_right(path, parent);
                        path.push(parent);
                        // stack: parent -> ...
                        current_node = this->rotate_right(path, current_node);
                        // stack: ...
                    }
                    else if ((*parent)->right_subtree == (*current_node) &&
    (*grandparent)->right_subtree == (*parent))
                    {
                        // zig-zig: rotate_left(parent) + rotate_left(target_ptr)
                        // stack: parent -> grandparent -> ...
                        path.pop();
                        parent = this->rotate_left(path, parent);
                        path.push(parent);
                        // stack: parent -> ...
                        current_node = this->rotate_left(path, current_node);
                        // stack: ...
                    }
                    else if ((*parent)->right_subtree == (*current_node) &&
    (*grandparent)->left_subtree == (*parent)) {
                        // zig-zag: rotate_left(target_ptr) + rotate_right(target_ptr)
                        current_node = this->rotate_left(path, current_node);
                        current_node = this->rotate_right(path, current_node);
                    } else {
                        // zig-zag: rotate_right(target_ptr) + rotate_left(target_ptr)
                        current_node = this->rotate_right(path, current_node);
                        current_node = this->rotate_left(path, current_node);
                    }
                }

                tree_root = reinterpret_cast<splay_tree<tkey, tvalue, tkey_comparer>
    *>(this->_target_tree)->_root;
            }
        }

    public:
        explicit template_method_splay(
                splay_tree<tkey, tvalue, tkey_comparer> *target_tree)
        : bs_tree<tkey, tvalue, tkey_comparer>::template_method_basics(target_tree)
                {

                }

            virtual ~template_method_splay() = default;
    };

  #pragma region insertion splay tree
    class insertion_splay_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::insertion_template_method
    {
        void after_insert_inner(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path, typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **target_ptr) override
        {
```

```
    this->trace_with_guard("splay_tree::insertion_splay_tree::after_insert_inner::spl
    ay method started");
            reinterpret_cast<template_method_splay *>(this)->splay(path, target_ptr);

    this->trace_with_guard("splay_tree::insertion_splay_tree::after_insert_inner::spl
    ay method finished");
        }

    public:
        explicit insertion_splay_tree(splay_tree<tkey, tvalue, tkey_comparer>
    *target_tree)
        : bs_tree<tkey, tvalue, tkey_comparer>::insertion_template_method(target_tree)
                {

                }
    };

#pragma endregion

#pragma region finding splay tree
    class finding_splay_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::finding_template_method
    {
        void after_find_inner(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path, typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **&target_ptr) override
        {

    this->trace_with_guard("splay_tree::finding_splay_tree::after_find_inner::splay
    method started");
            reinterpret_cast<template_method_splay *>(this)->splay(path, target_ptr);

    this->trace_with_guard("splay_tree::finding_splay_tree::after_find_inner::splay
    method finished");
        }

    public:
        explicit finding_splay_tree(splay_tree<tkey, tvalue, tkey_comparer>
    *target_tree)
            : bs_tree<tkey, tvalue,
    tkey_comparer>::finding_template_method(target_tree)
        {

        }
    };

#pragma endregion

#pragma region removing splay tree
    class removing_splay_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::removing_template_method
    {
        void after_remove(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path) const override
```

```
        {

    this->trace_with_guard("splay_tree::removing_splay_tree::after_remove::splay
    method started");
            typename bs_tree<tkey, tvalue, tkey_comparer>::node **
    parent_to_deleted_node = path.top();

            if (path.empty() == false) {
                path.pop();
                (reinterpret_cast<template_method_splay
    *>(const_cast<removing_splay_tree *>(this)))->splay(path,
    parent_to_deleted_node);
            }

    this->trace_with_guard("splay_tree::removing_splay_tree::after_remove::splay
    method started");
        }

    public:
        explicit removing_splay_tree(splay_tree<tkey, tvalue, tkey_comparer>
    *target_tree)
            : bs_tree<tkey, tvalue,
    tkey_comparer>::removing_template_method(target_tree)
        {

        }
    };

#pragma endregion
#pragma endregion

#pragma region rule 5
public:
    explicit splay_tree(logger *_logger = nullptr, memory *_allocator = nullptr)
            : bs_tree<tkey, tvalue, tkey_comparer>(
                    _logger,
                    _allocator,
                    new insertion_splay_tree(this),
                    new finding_splay_tree(this),
                    new removing_splay_tree(this))
    {

    }

private:
    // constructor
    splay_tree(
            logger *logger,
            memory *allocator,
            insertion_splay_tree *insertion,
            finding_splay_tree *finding,
            removing_splay_tree *removing)
    {
        this->_logger(logger);
        this->_allocator(allocator);
```

```
        this->_insertion(insertion);
        this->_finding(finding);
        this->_removing(removing);
        this->_root(nullptr);

        this->trace_with_guard("splay_tree constructor was called");
    }

public:
    // copy constructor
    splay_tree(splay_tree<tkey, tvalue, tkey_comparer> const &obj)
            : splay_tree(obj._logger, obj._allocator)
    {
        this->trace_with_guard("splay_tree copy constructor was called");
        this->_root = this->copy(obj._root);
    }

    // move constructor
    splay_tree(splay_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    : splay_tree(obj._insertion,
                 obj._finding,
                 obj._removing,
                 obj._allocator,
                 obj._logger)
    {
        this->trace_with_guard("splay_tree move constructor was called");

        this->_root = obj._root;
        obj._root = nullptr;

        this->_insertion->_target_tree = this;
        obj._insertion = nullptr;

        this->_finding->_target_tree = this;
        obj._finding = nullptr;

        this->_removing->_target_tree = this;
        obj._removing = nullptr;

        obj._allocator = nullptr;

        obj._logger = nullptr;
    }

    // copy assignment (оператор присваивания)
    splay_tree &operator=(splay_tree<tkey, tvalue, tkey_comparer> const &obj)
    {
        this->trace_with_guard("splay_tree copy assignment constructor was called");

        if (this == &obj)
        {
            return *this;
        }

        this->clearup(this->_root);
```

```
        this->_allocator = obj._allocator;
        this->_logger = obj._logger;
        this->_root = this->copy(obj._root);

        return *this;
    }

    // move assignment (оператор присваивания перемещением)
    splay_tree &operator=(splay_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    {
        this->trace_with_guard("splay_tree move assignment constructor was called");
        if (this == &obj)
        {
            return *this;
        }

        clearup(this->_root);
        this->_root = obj._root;
        obj._root = nullptr;

        delete obj._insertion;
        obj._insertion = nullptr;

        delete obj._finding;
        obj._finding = nullptr;

        delete obj._removing;
        obj._removing = nullptr;

        this->_allocator = obj._allocator;
        obj._allocator = nullptr;

        this->_logger = obj._logger;
        obj._logger = nullptr;

        return *this;
    }

    // destructor
    ~splay_tree()
    {
        this->trace_with_guard("splay_tree destructor was called");

    }
#pragma endregion
};

#endif //SPLAY_TREE_H
```

```
#ifndef RB_TREE_H
#define RB_TREE_H
#define RED true
#define BLACK false

#include "../binary_tree/bs_tree.h"

template<
       typename tkey,
       typename tvalue,
       typename tkey_comparer>
class rb_tree
       : public bs_tree<tkey, tvalue, tkey_comparer>
{
protected:
   struct rb_node
       : public bs_tree<tkey, tvalue, tkey_comparer>::node
   {
   private:
       // red == 1, black = 0
       bool _color;

   public:
       bool is_red()
       {
           return _color;
       }

       bool is_black()
       {
           return (!_color);
       }

       bool get_color()
       {
           return (this == nullptr ? BLACK : _color);
       }

       void change_color(bool new_color)
       {
           _color = new_color;
       }
   };

protected:
#pragma region template methods rb tree
#pragma region insertion to rb tree
   class insertion_rb_tree final :
           public bs_tree<tkey, tvalue, tkey_comparer>::insertion_template_method
   {
   private:
       size_t get_node_size() const override
       {
```

```
            return sizeof(rb_node);
        }

        void initialize_memory_with_node(typename bs_tree<tkey, tvalue,
    tkey_comparer>::node *target_ptr) const override
        {
            new(reinterpret_cast<rb_node *>(target_ptr)) rb_node;
        }

        rb_node** find_uncle(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path, rb_node** current_node)
        {
            typename bs_tree<tkey, tvalue, tkey_comparer>::node ** parent =
    path.top();
            path.pop();
            if (path.empty()) {
                path.push(parent);
                return nullptr;
            }
            typename bs_tree<tkey, tvalue, tkey_comparer>::node ** grandparent =
    path.top();
            rb_node ** uncle = ((*grandparent)->left_subtree == (*parent) ?
                    reinterpret_cast<rb_node **>(&((*grandparent)->right_subtree))
                    :
                    reinterpret_cast<rb_node **>(&((*grandparent)->left_subtree))
                    );

            path.push(parent);
            return ((*uncle) == nullptr ? nullptr : uncle);
        }

        void insertion_do_balance(std::stack<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **> &path, typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **target_ptr)
        {
            rb_node ** current_node = reinterpret_cast<rb_node **>(target_ptr);

            // if current node is root make it black
            if (path.empty()) {
                (*current_node)->change_color(BLACK);
                return;
            }

            rb_node ** parent = reinterpret_cast<rb_node **>(this->find_parent(path,
    target_ptr));
            rb_node ** grandparent = reinterpret_cast<rb_node
    **>(this->find_grandparent(path, target_ptr));

            if (parent == nullptr || grandparent == nullptr) {
                return;
            }

            // perform balance if parent is red
            if ((*parent)->is_black()) {
                return;
```

```
        }

        rb_node ** uncle = find_uncle(path, current_node);
        // uncle might be a null terminated leaf, thus its color will be black
accordingly to rb tree properties
        bool uncle_color = (uncle == nullptr ? BLACK : (*uncle)->get_color());

        if (uncle_color == RED) {
            (*parent)->change_color(BLACK);
            if (uncle != nullptr) {
                (*uncle)->change_color(BLACK);
            }
            (*grandparent)->change_color(RED);

            // make path correct by popping current node's parent and grandparent
out of it
            path.pop();
            path.pop();
            insertion_do_balance(path, reinterpret_cast<typename bs_tree<tkey,
tvalue, tkey_comparer>::node **>(grandparent));
        }
        else {
            if ((*grandparent)->left_subtree == (*parent)) {
                // left-right case
                if ((*parent)->right_subtree == (*current_node)) {
                    // do left rotation(current_node), make links correct, do
left-left case
                    // actually, the rotation returns current node, but for
left-right case, which consists of one rotation + left-left case, we pretend
actual parent is newly inserted and actual inserted node is a parent; grandparent
and uncle not changed
                    parent = reinterpret_cast<rb_node **>(this->rotate_left(path,
reinterpret_cast<typename bs_tree<tkey, tvalue, tkey_comparer>::node
**>(current_node)));
                    path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(parent));
                    current_node = reinterpret_cast<rb_node
**>(&((*parent)->left_subtree));
                }
                // left-left case
                if ((*parent)->left_subtree == (*current_node)) {
                    // pop 1 value, do right_rotation(parent),push parent(wish),
make parent black and grandparent red
                    path.pop();
                    parent = reinterpret_cast<rb_node **>(this->rotate_right(path,
reinterpret_cast<typename bs_tree<tkey, tvalue, tkey_comparer>::node
**>(parent)));
                    grandparent = reinterpret_cast<rb_node
**>(&((*parent)->right_subtree));

                    (*parent)->change_color(BLACK);
                    (*grandparent)->change_color(RED);
                }
            }
            else {
```

```
                // right-left case
                if ((*parent)->left_subtree == (*current_node)) {
                    // do right_rotation(current_node), make links correct, do
right-right case
                    // actually, the rotation returns current node, but for
left-right case, which consists of one rotation + left-left case, we pretend
actual parent is newly inserted and actual inserted node is a parent; grandparent
and uncle not changed
                    parent = reinterpret_cast<rb_node **>(this->rotate_right(path,
reinterpret_cast<typename bs_tree<tkey, tvalue, tkey_comparer>::node
**>(current_node)));
                    path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(parent));
                    current_node = reinterpret_cast<rb_node
**>(&((*parent)->right_subtree));
                }
                // right-right case
                if ((*parent)->right_subtree == (*current_node)) {
                    // pop 1 value, do left_rotation(parent), push parent(wish),
make parent black and grandparent red
                    path.pop();
                    parent = reinterpret_cast<rb_node **>(this->rotate_left(path,
reinterpret_cast<typename bs_tree<tkey, tvalue, tkey_comparer>::node
**>(parent)));
                    grandparent = reinterpret_cast<rb_node
**>(&((*parent)->left_subtree));

                    (*parent)->change_color(BLACK);
                    (*grandparent)->change_color(RED);
                }
            }
        }
    }

    void after_insert_inner(std::stack<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **> &path, typename bs_tree<tkey, tvalue,
tkey_comparer>::node **target_ptr) override
    {
        this->trace_with_guard("rb_tree::insertion_rb_tree::after_insert_inner
method started");
        rb_node ** current_node = reinterpret_cast<rb_node **>(target_ptr);
        (*current_node)->change_color(RED);

        this->trace_with_guard("rb_tree::insertion_rb_tree::insertion_do_balance
method started");
        insertion_do_balance(path, target_ptr);
        this->trace_with_guard("rb_tree::insertion_rb_tree::insertion_do_balance
method finished")
            ->trace_with_guard("rb_tree::insertion_rb_tree::after_insert_inner
method finished");
    }

public:
    explicit insertion_rb_tree(rb_tree<tkey, tvalue, tkey_comparer> * target_tree)
```

```
                : bs_tree<tkey, tvalue,
    tkey_comparer>::insertion_template_method(target_tree)
        {

        }
    };

#pragma endregion

#pragma region removing rb tree
    class removing_rb_tree final :
            public bs_tree<tkey, tvalue, tkey_comparer>::removing_template_method
    {
        void swap_additional_data(
                typename bs_tree<tkey, tvalue, tkey_comparer>::node *one_node,
                typename bs_tree<tkey, tvalue, tkey_comparer>::node *another_node)
    override
        {
            auto *first  = reinterpret_cast<rb_node *>(one_node);
            auto *second = reinterpret_cast<rb_node *>(another_node);
            bool first_color = first->get_color();
            first->change_color(second->get_color());
            second->change_color(first_color);
        }

        tvalue remove(tkey const &key) override
        {
            this->trace_with_guard("rb_tree::removing_rb_tree::remove method
    started");

            auto path_and_target = this->find_path(key);
            auto path = path_and_target.first;
            rb_node **target_ptr = reinterpret_cast<rb_node
    **>(path_and_target.second);

            if (*target_ptr == nullptr)
            {
                this->debug_with_guard("rb_tree::removing_rb_tree::remove no value
    with passed key in tree")
                    ->trace_with_guard("rb_tree::removing_rb_tree::remove method
    finished");
                throw typename bs_tree<tkey, tvalue,
    tkey_comparer>::remove_exception("rb_tree::removing_rb_tree::remove::no value
    with passed key in tree");
            }

            tvalue result = (*target_ptr)->value;
            bool target_ptr_color = (*target_ptr)->get_color();

            // deleting element with 2 children (color does not matter)
            if ((*target_ptr)->left_subtree != nullptr &&
                (*target_ptr)->right_subtree != nullptr)
            {
                if (target_ptr_color == RED) {
```

```
                this->debug_with_guard("rb_tree::removing_rb_tree::remove deleting
an RED element with 2 children");
            } else {
                this->debug_with_guard("rb_tree::removing_rb_tree::remove deleting
an BLACK element with 2 children");
            }

            typename bs_tree<tkey, tvalue, tkey_comparer>::node
**element_to_swap_with = &((*target_ptr)->left_subtree);
            path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(target_ptr));

            bool element_to_swap_with_has_non_null_right_subtree = false;
            std::stack<typename bs_tree<tkey, tvalue, tkey_comparer>::node **>
swap_stack;

            if ((*element_to_swap_with)->right_subtree != nullptr) {
                element_to_swap_with_has_non_null_right_subtree = true;
                path.push(element_to_swap_with);
                element_to_swap_with = &((*element_to_swap_with)->right_subtree);

                bool local_element_to_swap_with_has_non_null_right_subtree =
false;
                do
                {
                    if ((*element_to_swap_with)->right_subtree != nullptr) {
                        local_element_to_swap_with_has_non_null_right_subtree =
true;
                        swap_stack.push(element_to_swap_with);
                        element_to_swap_with =
&((*element_to_swap_with)->right_subtree);
                    } else {
                        local_element_to_swap_with_has_non_null_right_subtree =
false;
                    }
                } while (local_element_to_swap_with_has_non_null_right_subtree);
            }

            target_ptr = reinterpret_cast<rb_node
**>(this->swap_nodes(element_to_swap_with, reinterpret_cast<typename
bs_tree<tkey, tvalue, tkey_comparer>::node **>(target_ptr)));

            if (element_to_swap_with_has_non_null_right_subtree) {
                path.pop();
                path.push(&((*(path.top()))->left_subtree));

                std::stack<typename bs_tree<tkey, tvalue, tkey_comparer>::node **>
reverse_swap_stack;
                while (!(swap_stack.empty())) {
                    reverse_swap_stack.push(swap_stack.top());
                    swap_stack.pop();
                }

                while (!(reverse_swap_stack.empty())) {
                    path.push(reverse_swap_stack.top());
```

```
                    reverse_swap_stack.pop();
                }
            }
        }

        // deleting an element with no children
        if ((*target_ptr)->left_subtree == nullptr && (*target_ptr)->right_subtree
== nullptr)
        {
            this->debug_with_guard("rb_tree::removing_rb_tree::remove deleting an
element with no children");
            this->cleanup_node(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(target_ptr));
            if (target_ptr_color == BLACK) {
                after_remove(path);
                if (this->get_root_node() != nullptr) {
                    reinterpret_cast<rb_node
*>(this->get_root_node())->change_color(BLACK);
                }
            }
        }
        // deleting an element with 1 child. There cannot be a red element to
delete with one child
        else if ((*target_ptr)->left_subtree != nullptr)
        {
            this->debug_with_guard("rb_tree::removing_rb_tree::remove deleting an
element with 1 left child");
            auto *target_left_subtree = (*target_ptr)->left_subtree;
            this->cleanup_node(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(target_ptr));
            *target_ptr = reinterpret_cast<rb_node *>(target_left_subtree);
            (*target_ptr)->change_color(BLACK);
        }
        else
        {
            this->debug_with_guard("rb_tree::removing_rb_tree::remove deleting an
element with 1 right child");
            auto *target_right_subtree = (*target_ptr)->right_subtree;
            this->cleanup_node(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(target_ptr));
            *target_ptr = reinterpret_cast<rb_node *>(target_right_subtree);
            (*target_ptr)->change_color(BLACK);
        }

        this->trace_with_guard("rb_tree::removing_template_method::remove method
finished");
        return result;
    }

    void after_remove(std::stack<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **> &path) const override
    {
        this->trace_with_guard("rb_tree::removing_rb_tree::after_remove method
started");
```

```
        // path содержит все узлы до удалённого, верхний узел в стеке -- родитель
удалённого
        // warning! color of root may change. Change color if needed
        if (path.empty()) {
            return;
        }
        rb_node ** parent = reinterpret_cast<rb_node **>(path.top());
        rb_node **brother_to_deleted =
                reinterpret_cast<rb_node **>(((*parent)->left_subtree == nullptr ?
&((*parent)->right_subtree) : &((*parent)->left_subtree)));

        if ((*brother_to_deleted) == nullptr) {
            return;
        }

        rb_node **brother_right_child, **brother_left_child;
        bool brother_right_child_color, brother_left_child_color;

        // deleted node was a left child of parent
        if ((*parent)->left_subtree == nullptr) {
            // case 1) if brother is red, do rotation(brother, parent), make
parent red and brother black. redo links: brother has changed, so find_pair new
brother. Go to next case (brother is now black)
            if ((*brother_to_deleted)->is_red()) {
                brother_to_deleted = reinterpret_cast<rb_node
**>(this->rotate_left(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted)));
                parent = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                (*parent)->change_color(RED);
                (*brother_to_deleted)->change_color(BLACK);
                path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted));
                path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(parent));
                brother_to_deleted = reinterpret_cast<rb_node
**>(&((*parent)->right_subtree));
            }

            // case 2) if brother is black we have 3 sub-cases:
            // for another cases we need to find_pair brother's children and their
colors
            if ((*brother_to_deleted) != nullptr) {
                brother_left_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                brother_right_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->right_subtree));
                brother_left_child_color = (*brother_left_child)->get_color();
                brother_right_child_color = (*brother_right_child)->get_color();
            } else {
                brother_left_child_color = BLACK;
                brother_right_child_color = BLACK;
            }
```

```
            // case 2.1) children of brother are both black. Make brother red and
parent black.
            if (brother_left_child_color == BLACK && brother_right_child_color ==
BLACK) {
                if ((*brother_to_deleted) != nullptr) {
                    (*brother_to_deleted)->change_color(RED);
                }
                (*parent)->change_color(BLACK);
            }
            else {
                // case 2.2) brother's right child is black and left child is red.
Do rotation(brother, his left child), make brother red, his left child black. Go
to next case with correct links
                if (brother_left_child_color == RED) {
                    path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted));
                    brother_to_deleted = reinterpret_cast<rb_node
**>(this->rotate_right(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_left_child)));
                    (*brother_to_deleted)->change_color(BLACK);
                    reinterpret_cast<rb_node
*>((*brother_to_deleted)->right_subtree)->change_color(RED);
                    brother_left_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                    brother_right_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->right_subtree));
                    brother_left_child_color = (*brother_left_child)->get_color();
                    brother_right_child_color = RED;
                }

                // case 2.3) (deleted node is left child) brother's right child is
red. Make brother be of color of our parent, brother's right child, our parent
black. Do rotation (parent, brother)
                if (brother_right_child_color == RED) {
                    bool parent_color = (*parent)->get_color();
                    brother_to_deleted = reinterpret_cast<rb_node
**>(this->rotate_left(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted)));
                    parent = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                    (*brother_to_deleted)->change_color(parent_color);
                    (*brother_right_child)->change_color(BLACK);
                    (*parent)->change_color(BLACK);
                }
            }
        }
        // cases are simply mirrored if deleted node was a right child
        else {
            // case 1) if brother is red, do rotation(brother, parent), make
parent red and brother black. redo links: brother has changed, so find_pair new
brother. Go to next case (brother is now black)
            if ((*brother_to_deleted)->is_red()) {
                brother_to_deleted = reinterpret_cast<rb_node
**>(this->rotate_right(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted)));
```

```
                parent = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->right_subtree));
                (*parent)->change_color(RED);
                (*brother_to_deleted)->change_color(BLACK);
                path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted));
                path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(parent));
                brother_to_deleted = reinterpret_cast<rb_node
**>(&((*parent)->left_subtree));
            }

            // case 2) if brother is black we have 3 sub-cases:
            // for another cases we need to find_pair brother's children and their
colors
            if ((*brother_to_deleted) != nullptr) {
                brother_left_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                brother_right_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->right_subtree));
                brother_left_child_color = (*brother_left_child)->get_color();
                brother_right_child_color = (*brother_right_child)->get_color();
            } else {
                brother_left_child_color = BLACK;
                brother_right_child_color = BLACK;
            }

            // case 2.1) children of brother are both black. Make brother red and
parent black.
            if (brother_left_child_color == BLACK && brother_right_child_color ==
BLACK) {
                if ((*brother_to_deleted) != nullptr) {
                    (*brother_to_deleted)->change_color(RED);
                }
                (*parent)->change_color(BLACK);
            }
            else {
                // case 2.2) brother's right child is black and left child is red.
Do rotation(brother, his left child), make brother red, his left child black. Go
to next case with correct links
                if (brother_right_child_color == RED) {
                    path.push(reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_to_deleted));
                    brother_to_deleted = reinterpret_cast<rb_node
**>(this->rotate_left(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
tkey_comparer>::node **>(brother_right_child)));
                    (*brother_to_deleted)->change_color(BLACK);
                    reinterpret_cast<rb_node
*>((*brother_to_deleted)->left_subtree)->change_color(RED);
                    brother_left_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->left_subtree));
                    brother_right_child = reinterpret_cast<rb_node
**>(&((*brother_to_deleted)->right_subtree));
                    brother_left_child_color = RED;
```

```
                        brother_right_child_color =
    (*brother_right_child)->get_color();
                    }

                    // case 2.3) (deleted node is left child) brother's right child is
    red. Make brother be of color of our parent, brother's right child, our parent
    black. Do rotation (parent, brother)
                    if (brother_left_child_color == RED) {
                        bool parent_color = (*parent)->get_color();
                        brother_to_deleted = reinterpret_cast<rb_node
    **>(this->rotate_right(path, reinterpret_cast<typename bs_tree<tkey, tvalue,
    tkey_comparer>::node **>(brother_to_deleted)));
                        parent = reinterpret_cast<rb_node
    **>(&((*brother_to_deleted)->right_subtree));
                        (*brother_to_deleted)->change_color(parent_color);
                        (*brother_left_child)->change_color(BLACK);
                        (*parent)->change_color(BLACK);
                    }
                }
            }
            this->trace_with_guard("rb_tree::removing_rb_tree::after_remove method
    finished");
        }

    public:
        explicit removing_rb_tree(rb_tree<tkey, tvalue, tkey_comparer> * target_tree)
                : bs_tree<tkey, tvalue,
    tkey_comparer>::removing_template_method(target_tree)
        {

        }
    };

#pragma endregion

#pragma endregion

public:
    explicit rb_tree(
            logger *_logger = nullptr,
            memory *_allocator = nullptr)
    : bs_tree<tkey, tvalue, tkey_comparer>(
            _logger,
            _allocator,
            new insertion_rb_tree(this),
            new typename bs_tree<tkey, tvalue,
    tkey_comparer>::finding_template_method(this),
            new removing_rb_tree(this))
    {

    }

private:
    // constructor
    rb_tree(
```

```
            logger *logger,
            memory *allocator,
            insertion_rb_tree *insertion,
            typename bs_tree<tkey, tvalue, tkey_comparer>::finding_template_method
    *finding,
            removing_rb_tree *removing)
    {
        this->_logger(logger);
        this->_allocator(allocator);
        this->_insertion(insertion);
        this->_finding(finding);
        this->_removing(removing);
        this->_root(nullptr);

        this->trace_with_guard("rb_tree constructor was called");
    }

public:
    // copy constructor
    rb_tree(rb_tree<tkey, tvalue, tkey_comparer> const &obj)
    : rb_tree(obj._logger, obj._allocator)
    {
        this->trace_with_guard("rb_tree copy constructor was called");
        this->_root = this->copy(obj._root);
    }

    // move constructor
    rb_tree(rb_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    : rb_tree(obj._insertion,
              obj._finding,
              obj._removing,
              obj._allocator,
              obj._logger)
    {
        this->trace_with_guard("rb_tree move constructor was called");

        this->_root = obj._root;
        obj._root = nullptr;

        this->_insertion->_target_tree = this;
        obj._insertion = nullptr;

        this->_finding->_target_tree = this;
        obj._finding = nullptr;

        this->_removing->_target_tree = this;
        obj._removing = nullptr;

        obj._allocator = nullptr;

        obj._logger = nullptr;
    }

    // copy assignment (оператор присваивания)
    rb_tree &operator=(rb_tree<tkey, tvalue, tkey_comparer> const &obj)
```

```
    {
        this->trace_with_guard("rb_tree copy assignment constructor was called");

        if (this == &obj)
        {
            return *this;
        }

        this->clearup(this->_root);

        this->_allocator = obj._allocator;
        this->_logger = obj._logger;
        this->_root = this->copy(obj._root);

        return *this;
    }

    // move assignment (оператор присваивания перемещением)
    rb_tree &operator=(rb_tree<tkey, tvalue, tkey_comparer> &&obj) noexcept
    {
        this->trace_with_guard("rb_tree move assignment constructor was called");

        if (this == &obj)
        {
            return *this;
        }

        this->clearup(this->_root);
        this->_root = obj._root;
        obj._root = nullptr;

        delete obj._insertion;
        obj._insertion = nullptr;

        delete obj._finding;
        obj._finding = nullptr;

        delete obj._removing;
        obj._removing = nullptr;

        this->_allocator = obj._allocator;
        obj._allocator = nullptr;

        this->_logger = obj._logger;
        obj._logger = nullptr;

        return *this;
    }

    // destructor
    ~rb_tree()
    {
        this->trace_with_guard("rb_tree destructor was called");
     }
```

```
private:
   typename bs_tree<tkey, tvalue, tkey_comparer>::node *copy(typename bs_tree<tkey,
   tvalue, tkey_comparer>::node *from) override
   {
       if (from == nullptr)
       {
           return nullptr;
       }

       rb_node *result = reinterpret_cast<rb_node
   *>(this->allocate_with_guard(sizeof(rb_node)));
       new (result) rb_node(*reinterpret_cast<rb_node *>(from));
       result->change_color(reinterpret_cast<rb_node *>(from)->get_color());

       result->left_subtree = copy(from->left_subtree);
       result->right_subtree = copy(from->right_subtree);

       return result;
   }
};


#endif //RB_TREE_H
```

```
#ifndef DB_USER_COMMUNICATION_H
#define DB_USER_COMMUNICATION_H

#include "db_key/key.h"
#include "db_value/db_value_builder.h"
#include "db/data_base.h"

void help();

#pragma region Parsing
typedef enum commands {
   _add_,
   _find_,
   _update_,
   _delete_,
   _save_,
   _upload_,
   _help_,
   _exit_,
   _not_a_command_
} commands_;

class parse_exception final : public std::exception {
private:
   std::string _message;

public:
   explicit parse_exception(std::string message)
           : _message(std::move(message)) {

   }

   [[nodiscard]] char const *what() const noexcept override {
       return _message.c_str();
   }
};

commands_
get_command(std::string const &user_input);

std::pair<commands_, std::string>
parse_user_input(std::string const &user_input);

std::tuple<std::string, std::string, std::string>
parse_path(std::string &input_string);

std::tuple<std::string, std::string, std::string>
get_path_from_user_input(std::ifstream *input_stream, bool is_cin, bool is_path);

#pragma endregion

#pragma region Add command
```

```
void
do_add_command(data_base *db, std::string &input_str_leftover, std::ifstream *
   input_stream, bool is_cin);

#pragma endregion

#pragma region Find command

typedef struct time_str
{
   short YY, MM, DD;
   short hh, mm, ss;
} struct_time;

std::tuple<db_value *, std::vector<db_value *>, db_value *>
do_find_command
(data_base * db, std::string &input_str_leftover, std::ifstream * input_stream, bool
   is_cin);

#pragma endregion

#pragma region Update command

void do_update_command(data_base * db, std::ifstream *input_stream, bool is_cin);

#pragma endregion

#pragma region Delete command

void delete_db(data_base *db);

void do_delete_command(data_base * db, std::string & path_inner, std::ifstream
   *input_stream, bool is_cin);

#pragma endregion

#endif //DB_USER_COMMUNICATION_H
```

Реализация перечисленных в методов *db_user_communication*

```
#include "db_user_communication.h"

void help() {
   std::cout << "----- Course work help -----" << std::endl;
   std::cout << "Collection commands list:" << std::endl;
   std::cout << "Full path to collection:
   <pool_name>/<scheme_name>/<collection_name>" << std::endl;
   std::cout << "\t- add" << std::endl;
   std::cout << "\t- find" << std::endl;
   std::cout << "\t- find dataset" << std::endl;
   std::cout << "\t- find DD/MM/YYYY hh/mm/ss" << std::endl;
   std::cout << "\t- update" << std::endl;
   std::cout << "\t- delete" << std::endl;
   std::cout << "Structural and customization commands list:" << std::endl;
```

```
    std::cout << "Supported trees: BST (binary tree), AVL, SPLAY, RB (red-black tree)"
    << std::endl;
    std::cout
            << "Supported allocators: global, sorted_list best||worst||first,
    descriptors best||worst||first, buddy_system"
            << std::endl;
    std::cout << "\t- add <tree type> <allocator type> <size of allocator> <full
    path>" << std::endl;
    std::cout << "\t\tone can note tree/allocator type only" << std::endl;
    std::cout
            << "\t\tif one wants to note an allocator, it's needed to also node
    desired length in bytes (not for global one)"
            << std::endl;
    std::cout << "\t- delete <full path>" << std::endl;
    std::cout << "DB commands list:" << std::endl;
    std::cout << "\t- help" << std::endl;
    std::cout << "\t- delete DB" << std::endl;
    std::cout << "\t- exit" << std::endl;
}

#pragma region Parsing
commands_ get_command(std::string const &user_input) {
    if (user_input == "add") {
        return commands_::_add_;
    } else if (user_input == "find") {
        return commands_::_find_;
    } else if (user_input == "update") {
        return commands_::_update_;
    } else if (user_input == "delete" || user_input == "remove") {
        return commands_::_delete_;
    } else if (user_input == "save") {
        return commands_::_save_;
    } else if (user_input == "upload") {
        return commands_::_upload_;
    } else if (user_input == "help") {
        return commands_::_help_;
    } else if (user_input == "exit") {
        return commands_::_exit_;
    }
    return commands_::_not_a_command_;
}

// returns a command, non-command info
std::pair<commands_, std::string>
parse_user_input(std::string const &user_input) {
    commands_ command = commands_::_not_a_command_;
    std::string token, s = user_input;
    size_t pos;

    // finding a command
    if ((pos = s.find(' ')) != std::string::npos) {
        command = get_command(s.substr(0, pos));
        s.erase(0, pos + 1);
    } else {
        s.erase(remove_if(s.begin(), s.end(), isspace), s.end());
```

```
        command = get_command(s);
        s.erase();
    }

    // a path is what is left from s
    return {command, s};
}

std::tuple<std::string, std::string, std::string>
parse_path(std::string &input_string) {
    input_string.erase(remove_if(input_string.begin(), input_string.end(), isspace),
    input_string.end());
    if (input_string.empty()) {
        throw parse_exception("parse_path:: incorrect path passed (is empty)");
    }

    std::string pool_name, scheme_name, collection_name;
    std::string delimiter = "/";
    unsigned delimiter_length = delimiter.size();
    size_t pos;

    if ((pos = input_string.find(delimiter)) != std::string::npos) {
        pool_name = input_string.substr(0, pos);
        input_string.erase(0, pos + delimiter_length);

        if ((pos = input_string.find(delimiter)) != std::string::npos) {
            scheme_name = input_string.substr(0, pos);
            input_string.erase(0, pos + delimiter_length);

            collection_name = input_string;
        } else {
            scheme_name = input_string;
        }
    } else {
        pool_name = input_string;
    }
    return {pool_name, scheme_name, collection_name};
}

// returns pool/scheme/collection
std::tuple<std::string, std::string, std::string>
get_path_from_user_input(std::ifstream *input_stream, bool is_cin, bool is_path) {
    std::string path_inner;

    if (is_cin) {
        (is_path
         ? std::cout << "Enter full path to collection: >>"
         : std::cout << "Enter full path to filename: >>");
    }

    (is_cin
     ? std::getline(std::cin, path_inner)
     : std::getline((*input_stream), path_inner));

    return parse_path(path_inner);
```

```
}

#pragma endregion

#pragma region Add command

data_base::trees_types_ get_tree_type(std::string const &user_input) {
   if (user_input == "BST" || user_input == "bst") {
       return data_base::trees_types_::BST;
   } else if (user_input == "AVL" || user_input == "avl") {
       return data_base::trees_types_::AVL;
   } else if (user_input == "SPLAY" || user_input == "splay") {
       return data_base::trees_types_::SPLAY;
   } else if (user_input == "RB" || user_input == "rb") {
       return data_base::trees_types_::RB;
   }
   return data_base::trees_types_::not_a_tree;
}

data_base::allocator_types_ get_allocator_type(std::string const &user_input) {
   if (user_input == "global") {
       return data_base::allocator_types_::global;
   } else if (user_input == "sorted_list") {
       return data_base::allocator_types_::for_inner_use_sorted_list;
   } else if (user_input == "descriptors") {
       return data_base::allocator_types_::for_inner_use_descriptors;
   } else if (user_input == "buddy_system") {
       return data_base::allocator_types_::buddy_system;
   }
   return data_base::allocator_types_::not_an_allocator;
}

std::pair<data_base::allocator_types_, size_t>
define_allocator_type
       (data_base::allocator_types_ allocator_type,
        std::string &s, std::string &token, std::string &delimiter,
        size_t pos, unsigned delimiter_length) {
   if (allocator_type == data_base::allocator_types_::for_inner_use_sorted_list ||
       allocator_type == data_base::allocator_types_::for_inner_use_descriptors) {
//        s.erase(0, pos + delimiter_length);
       if ((pos = s.find(delimiter)) != std::string::npos) {
           token = s.substr(0, pos);
           if (token == "best") {
               allocator_type = (allocator_type ==

   data_base::allocator_types_::for_inner_use_sorted_list
                                ? data_base::allocator_types_::sorted_list_best
                                : data_base::allocator_types_::descriptors_best);
           } else if (token == "worst") {
               allocator_type = (allocator_type ==

   data_base::allocator_types_::for_inner_use_sorted_list
                                ? data_base::allocator_types_::sorted_list_worst
                                : data_base::allocator_types_::descriptors_worst);
           } else if (token == "first") {
```

```
                allocator_type = (allocator_type ==

    data_base::allocator_types_::for_inner_use_sorted_list
                                ? data_base::allocator_types_::sorted_list_first
                                : data_base::allocator_types_::descriptors_first);
            }
        } else {
            allocator_type = data_base::allocator_types_::not_an_allocator;
        }
    }

    size_t allocator_pool_size = 0;

    if (allocator_type != data_base::allocator_types_::global &&
        allocator_type != data_base::allocator_types_::not_an_allocator) {
        // need to read a size of allocator
        s.erase(0, pos + delimiter_length);
        if ((pos = s.find(delimiter)) != std::string::npos) {
            token = s.substr(0, pos);
            try {
                allocator_pool_size = std::stoull(token);
                s.erase(0, pos + delimiter_length);
            }
            catch (std::invalid_argument const &) {
                throw parse_exception("define_allocator_type:: passed size of
    allocator must be of type size_t");
            }
        }
    }

    return {allocator_type, allocator_pool_size};
}

std::tuple<data_base::trees_types_, data_base::allocator_types_, size_t>
parse_for_add_command(std::string &input_str_leftover) {
    data_base::trees_types_ tree_type = data_base::trees_types_::not_a_tree;
    data_base::allocator_types_ allocator_type =
    data_base::allocator_types_::not_an_allocator;

    size_t allocator_pool_size = 0, pos;
    std::string token, delimiter = " ";

    unsigned delimiter_length = delimiter.length();
    if ((pos = input_str_leftover.find(delimiter)) != std::string::npos) {
        token = input_str_leftover.substr(0, pos);
        input_str_leftover.erase(0, pos + delimiter_length);

        tree_type = get_tree_type(token);
        allocator_type = get_allocator_type(token);

        if (allocator_type != data_base::not_an_allocator) {
            auto allocator_type_and_size = define_allocator_type(allocator_type,
    input_str_leftover, token, delimiter, pos,
                                                                 delimiter_length);
            allocator_type = allocator_type_and_size.first;
```

```
            allocator_pool_size = allocator_type_and_size.second;
        }

        if ((pos = input_str_leftover.find(delimiter)) != std::string::npos) {
            token = input_str_leftover.substr(0, pos);
            input_str_leftover.erase(0, pos + delimiter_length);

            if (tree_type == data_base::not_a_tree) {
                tree_type = get_tree_type(token);
            }

            if (allocator_type == data_base::not_an_allocator) {
                allocator_type = get_allocator_type(token);
                auto allocator_type_and_size = define_allocator_type(allocator_type,
    input_str_leftover, token, delimiter,
                                                                     pos,
                                                                     delimiter_length);
                allocator_type = allocator_type_and_size.first;
                allocator_pool_size = allocator_type_and_size.second;
            }
        }
    }

    return {tree_type, allocator_type, allocator_pool_size};
}

void
do_add_command
(data_base *db, std::string &input_str_leftover, std::ifstream *input_stream, bool
    is_cin)
{
    if (input_str_leftover.empty()) {
        // adding a value
        key tmp_key(input_stream, is_cin);

        auto *dbValueBuilder = new db_value_builder();
        db_value *tmp_value = dbValueBuilder->build_from_stream(input_stream, is_cin);
        delete dbValueBuilder;

        auto path_parse_result = get_path_from_user_input(input_stream, is_cin, true);

        db->add_to_collection(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                              std::get<2>(path_parse_result),
                              tmp_key, tmp_value);
        if (is_cin) {
            std::cout << "Added a value to collection " <<
    std::get<2>(path_parse_result) << " successfully!" << std::endl;
        }
    } else {
        // adding to structure
        std::tuple<data_base::trees_types_, data_base::allocator_types_, size_t>
    parse_result
                = parse_for_add_command(input_str_leftover);
```

```
        data_base::trees_types_ tree_type = std::get<0>(parse_result);
        data_base::allocator_types_ allocator_type = std::get<1>(parse_result);
        size_t allocator_pool_size = std::get<2>(parse_result);

        std::tuple<std::string, std::string, std::string> struct_parse_path_result =
    parse_path(input_str_leftover);


        db->add_to_structure(std::get<0>(struct_parse_path_result),
                             std::get<1>(struct_parse_path_result),
    std::get<2>(struct_parse_path_result),
                             tree_type, allocator_type, allocator_pool_size);
        if (is_cin) {
            std::cout << "Added " << input_str_leftover << " successfully!" <<
    std::endl;
        }
    }
}

#pragma endregion

#pragma region Find command

short get_part_of_data_from_input_str(std::string &str, std::string &delimiter,
    unsigned delimiter_length) {
    short to_return = 0;
    size_t pos;
    if ((pos = str.find(delimiter)) != std::string::npos) {
        try {
            to_return = std::stoi(str.substr(0, pos));
        }
        catch (std::invalid_argument const &) {
            throw parse_exception("get_part_of_data_from_input_str:: incorrect value
    for data passed. Only digits");
        }
        str.erase(0, pos + delimiter_length);
    } else {
        try {
            to_return = std::stoi(str.substr(0, pos));
        }
        catch (std::invalid_argument const &) {
            throw parse_exception("get_part_of_data_from_input_str:: incorrect value
    for data passed. Only digits");
        }
        str.erase();
    }
    return to_return;
}

uint64_t convert_time_str_to_ms(time_str data) {
    std::tm tmp{};
    tmp.tm_sec = data.ss;
    tmp.tm_min = data.mm; // -1 ?
    tmp.tm_hour = data.hh;
    tmp.tm_year = data.YY - 1900;
```

```
    tmp.tm_mon = data.MM;
    tmp.tm_mday = data.DD;
    uint64_t milli = std::mktime(&tmp);
    milli = milli * 1000;
    return milli;
}

uint64_t parse_time_from_input_str(std::string &input_stream) {
    time_str to_return{};

    std::string lexeme_delimiter = " ", part_delimiter = "/";
    std::string day_month_year, hour_min_sec;
    unsigned delimiter_length = 1;
    size_t pos;

    if ((pos = input_stream.find(lexeme_delimiter)) != std::string::npos) {
        day_month_year = input_stream.substr(0, pos);
        input_stream.erase(0, pos + delimiter_length);
    } else {
        throw parse_exception("parse_time_from_input_str:: should pass full data
    timestamp");
    }

    to_return.DD = get_part_of_data_from_input_str(day_month_year, part_delimiter,
    delimiter_length);
    to_return.MM = get_part_of_data_from_input_str(day_month_year, part_delimiter,
    delimiter_length);
    to_return.YY = get_part_of_data_from_input_str(day_month_year, part_delimiter,
    delimiter_length);

    to_return.hh = get_part_of_data_from_input_str(input_stream, part_delimiter,
    delimiter_length);
    to_return.mm = get_part_of_data_from_input_str(input_stream, part_delimiter,
    delimiter_length);
    to_return.ss = get_part_of_data_from_input_str(input_stream, part_delimiter,
    delimiter_length);

    if (to_return.hh > 23 || to_return.hh < 0 ||
        to_return.mm > 60 || to_return.mm < 0 ||
        to_return.ss > 60 || to_return.ss < 0)
    {
        throw parse_exception("parse_time_from_input_str:: incorrect data passed");
    }

    return convert_time_str_to_ms(to_return);
}

std::tuple<db_value *, std::vector<db_value *>, db_value *>
do_find_command
(data_base *db, std::string &input_str_leftover, std::ifstream *input_stream, bool
    is_cin)
{
    std::vector<db_value *> to_return_vector;
    db_value *found_with_time = nullptr, *simply_found = nullptr;
```

```
    std::string token, delimiter = " ";
//    unsigned delimiter_length = delimiter.length();
//    size_t pos;

    if (input_str_leftover.find(delimiter) != std::string::npos) {
        // with time
        uint64_t time_stamp = parse_time_from_input_str(input_str_leftover);

        key tmp_key(input_stream, is_cin);

        auto path_parse_result = get_path_from_user_input(input_stream, is_cin, true);

        found_with_time = db->find_with_time(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                                             std::get<2>(path_parse_result),
                                             tmp_key, time_stamp);
    } else if (input_str_leftover == "dataset") {
        // find in range
        if (is_cin) {
            std::cout << "Enter min and max keys" << std::endl;
        }
        key min(input_stream, is_cin);
        key max(input_stream, is_cin);

        auto path_parse_result = get_path_from_user_input(input_stream, is_cin, true);

        to_return_vector = db->find_in_range(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                                             std::get<2>(path_parse_result),
                                             min, max);
    } else if (input_str_leftover.empty()) {
        key tmp_key(input_stream, is_cin);
        std::tuple<std::string, std::string, std::string> path_parse_result =
    get_path_from_user_input(input_stream,

    is_cin, true);

        simply_found = db->find_among_collection(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                                                 std::get<2>(path_parse_result),
                                                 tmp_key);
    } else {
        throw parse_exception("do_find_command:: Incorrect command entered");
    }

    return {found_with_time, to_return_vector, simply_found};
}

#pragma endregion

#pragma region Update command

std::pair<key, std::map<db_value_fields, unsigned char *>>
get_update_key_and_dictionary(std::ifstream *input_stream, bool is_cin) {
    key to_return_key(input_stream, is_cin);
```

```
    std::map<db_value_fields, unsigned char *> to_return_dict;

    std::string token, field_name, delimiter = ":";
    db_value_fields dbValueField;
    unsigned delimiter_length = delimiter.length(), iterations = 15;
    size_t pos;
    if (is_cin) {
        std::cout << "Print field_name: new_value" << std::endl
                  << "Fields: surname, name, patronymic, birthday, link_to_resume,
    hr_id, p_language, tasks, solved, copying"
                  << std::endl
                  << "To stop print exit" << std::endl;
    }

    while (iterations) {
        iterations--;

        if (is_cin) {
            std::getline(std::cin, token);
        } else {
            std::getline((*input_stream), token);
        }

        // delete all spaces, after this: surname:Surname
        token.erase(remove_if(token.begin(), token.end(), isspace), token.end());

        if (token == "exit") {
            break;
        }

        if ((pos = token.find(delimiter)) != std::string::npos) {
            field_name = token.substr(0, pos);
            token.erase(0, pos + delimiter_length);
        } else {
            throw parse_exception(
                    "get_update_key_and_dictionary:: incorrect input. Must be
    <field_name>:<new_field_value>");
        }

        if (field_name == "surname") {
            dbValueField = db_value_fields::_surname_;
        } else if (field_name == "name") {
            dbValueField = db_value_fields::_name_;
        } else if (field_name == "patronymic") {
            dbValueField = db_value_fields::_patronymic_;
        } else if (field_name == "birthday") {
            dbValueField = db_value_fields::_birthday_;
        } else if (field_name == "link_to_resume") {
            dbValueField = db_value_fields::_link_to_resume_;
        } else if (field_name == "hr_id") {
            dbValueField = db_value_fields::_hr_id_;
        } else if (field_name == "p_language") {
            dbValueField = db_value_fields::_programming_language_;
        } else if (field_name == "tasks") {
            dbValueField = db_value_fields::_task_count_;
```

```
        } else if (field_name == "solved") {
            dbValueField = db_value_fields::_solved_task_count_;
        } else if (field_name == "copying") {
            dbValueField = db_value_fields::_copying_;
        } else {
            throw parse_exception("get_update_key_and_dictionary:: incorrect field
    name");
        }

        if (to_return_dict.contains(dbValueField)) {
            delete to_return_dict[dbValueField];
        }

        if (dbValueField == db_value_fields::_surname_ || dbValueField ==
    db_value_fields::_name_ ||
            dbValueField == db_value_fields::_patronymic_ ||
            dbValueField == db_value_fields::_birthday_ || dbValueField ==
    db_value_fields::_link_to_resume_ ||
            dbValueField == db_value_fields::_programming_language_) {
            to_return_dict[dbValueField] = reinterpret_cast<unsigned char *>(new
    std::string(token));
        } else if (dbValueField == db_value_fields::_hr_id_) {
            // try catch?
            try {
                to_return_dict[dbValueField] = reinterpret_cast<unsigned char *>(new
    int(std::stoi(token)));
            }
            catch (std::invalid_argument const &) {
                throw parse_exception("get_update_key_and_dictionary:: hr id value
    must be of type int");
            }
        } else if (dbValueField == db_value_fields::_task_count_ ||
                   dbValueField == db_value_fields::_solved_task_count_) {
            try {
                to_return_dict[dbValueField] = reinterpret_cast<unsigned char *>(new
    unsigned(std::stoi(token)));
            }
            catch (std::invalid_argument const &) {
                throw parse_exception("get_update_key_and_dictionary:: task count
    value must be of type unsigned");
            }
        } else if (dbValueField == db_value_fields::_copying_) {
            if (token == "true" || token == "1") {
                to_return_dict[dbValueField] = reinterpret_cast<unsigned char *>(new
    bool(true));
            } else if (token == "false" || token == "0") {
                to_return_dict[dbValueField] = reinterpret_cast<unsigned char *>(new
    bool(false));
            } else {
                throw parse_exception(
                        "get_update_key_and_dictionary:: incorrect copying value. It
    must be true/false or 1/0");
            }
        }
    }
```

```
    return {to_return_key, to_return_dict};
}

void do_update_command(data_base *db, std::ifstream *input_stream, bool is_cin) {
    std::pair<key, std::map<db_value_fields, unsigned char *>> key_and_dict =
    get_update_key_and_dictionary(
            input_stream, is_cin);

    auto path_parse_result = get_path_from_user_input(input_stream, is_cin, true);
    db->update_in_collection(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                             std::get<2>(path_parse_result),
                             key_and_dict.first, key_and_dict.second);
    if (is_cin) {
        std::cout << "Updated a value in collection " <<
    std::get<2>(path_parse_result) << " successfully!" << std::endl;
    }
}

#pragma endregion

#pragma region Delete command

void delete_db(data_base *db) {
    db->~data_base();
}

void do_delete_command(data_base *db, std::string &path_inner, std::ifstream
    *input_stream,
                       bool is_cin) {
    path_inner.erase(remove_if(path_inner.begin(), path_inner.end(), isspace),
    path_inner.end());

    // a key from a collection (return a value)
    if (path_inner.empty()) {
        key tmp_key(input_stream, is_cin);
        auto path_parse_result = get_path_from_user_input(input_stream, is_cin, true);

        db->delete_from_collection(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                                   std::get<2>(path_parse_result),
                                   tmp_key);
        if (is_cin) {
            std::cout << "Deleted value from collection " <<
    std::get<2>(path_parse_result) << " successfully!" << std::endl;
        }
    }
        // delete the whole database
    else if (path_inner == "DB") {
        delete_db(db);
        if (is_cin) {
            std::cout << "Database was deleted successfully!" << std::endl;
        }
    }
        // delete pool/scheme/collection
```

```
    else {
        auto path_parse_result = parse_path(const_cast<std::string &>(path_inner));

        db->delete_from_structure(std::get<0>(path_parse_result),
    std::get<1>(path_parse_result),
                                  std::get<2>(path_parse_result));

        std::cout << "Removed " << path_inner << " successfully!" << std::endl;
    }
}

#pragma endregion
```

```
#include "db_user_communication.h"


void db_test(data_base * db, std::ifstream *input_stream, bool is_cin) {
   if (is_cin) {
       help();
   }
   bool not_exited = true;
   std::string user_input;

   while (not_exited) {
       if (is_cin) {
           std::cout << ">>";
           std::getline(std::cin, user_input);
       } else {
           std::getline((*input_stream), user_input);
       }

       auto command_and_leftover_string = parse_user_input(user_input);
       commands_ command = command_and_leftover_string.first;
       std::string leftover = command_and_leftover_string.second;

       switch (command) {
           case commands_::_add_:
               try {
                   do_add_command(db, leftover, input_stream, is_cin);
               }
               catch (parse_exception const & except) {
                   if (is_cin) {
                       std::cout << except.what() << std::endl;
                   }
               }
               catch (db_value::create_exception const & except) {
                   if (is_cin) {
                       std::cout << except.what() << std::endl;
                   }
               }
               catch (key::create_exception const & except) {
                   if (is_cin) {
                       std::cout << except.what() << std::endl;
                   }
               }
               catch (handler::order_exception const & except) {
                   if (is_cin) {
                       std::cout << except.what() << std::endl;
                   }
               }
               catch (data_base::db_insert_exception const & except) {
                   if (is_cin) {
                       std::cout << except.what() << std::endl;
                   }
               }
               catch (data_base::db_find_exception const & except) {
```

```
                    if (is_cin) {
                        std::cout << except.what() << std::endl;
                    }
                }
                break;
            case commands_::_find_:
                try {
                    std::tuple<db_value *, std::vector<db_value *>, db_value *> found
                            = do_find_command(db, leftover, input_stream, is_cin);

                    db_value * found_with_time = std::get<0>(found);
                    std::vector<db_value *> db_value_vector_in_range =
    std::get<1>(found);
                    db_value * simpy_found = std::get<2>(found);

                    if (found_with_time != nullptr) {
                        if (is_cin) {
                            std::cout << (*found_with_time) << std::endl;
                        }
                        delete found_with_time;
                    }
                    else if (simpy_found != nullptr) {
                        if (is_cin) {
                            std::cout << (*simpy_found) << std::endl;
                        }
                    }
                    else if (!(db_value_vector_in_range.empty())) {
                        if (is_cin) {
                            unsigned i, size_of_vector =
    db_value_vector_in_range.size();
                            for (i = 0; i < size_of_vector; i++) {
                                std::cout << "----- " << i + 1 << " value " << "-----"
    << std::endl;
                                std::cout << (*(db_value_vector_in_range[i])) <<
    std::endl;
                            }
                        }
                    } else {
                        if (is_cin) {
                            std::cout << "No values were found" << std::endl;
                        }
                    }
                }
                catch (parse_exception const & exception) {
                    if (is_cin) {
                        std::cout << exception.what() << std::endl;
                    }
                }
                catch (key::create_exception const & exception)
                {
                    if (is_cin) {
                        std::cout << exception.what() << std::endl;
                    }
                }
                catch (data_base::db_find_exception const & exception) {
```

```
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            break;
        case commands_::_update_:
            try {
                do_update_command(db, input_stream, is_cin);
            }
            catch (key::create_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (parse_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (handler::order_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (data_base::db_find_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            break;
        case commands_::_delete_:
            try {
                do_delete_command(db, leftover, input_stream, is_cin);
            }
            catch (parse_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (data_base::db_find_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (handler::order_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            catch (data_base::db_remove_exception const & exception) {
                if (is_cin) {
                    std::cout << exception.what() << std::endl;
                }
            }
            break;
```

```
            case commands_::_help_:
                help();
                break;
            case commands_::_exit_:
                if (is_cin) {
                    delete_db(db);
                    std::cout << "Exited successfully!" << std::endl;
                }
                not_exited = false;
                break;
            default:
                if (is_cin) {
                    std::cout << "Wrong command passed, try again!" << std::endl;
                }
                break;
        }
    }
}

int main(int argc, char **argv)
{
    if (argc != 2) {
        std::cout << "You should point a path to file with execution commands as a
    command line argument for proper work of program" << std::endl;
        return 0;
    }

    auto * file = new std::ifstream(argv[1]);
    if ( !(*file) ) {
        std::cout << "Could not open a file " << argv[1] << std::endl;
        return 0;
    }

    logger_builder * loggerBuilder = new logger_builder_impl();
    logger * logg = loggerBuilder->with_stream("trace.txt", logger::severity::trace)
                                 ->with_stream("warning.txt",
    logger::severity::warning)
                                 ->build();
    delete loggerBuilder;

    data_base db(logg);

    db_test(&db, file, false);
    file->close();

    std::cout << "Hello! Would you like to get some closer interaction with my
    program? Print y for yes an n for no\n>>";
    std::string answer_to_important_question;
    std::getline(std::cin, answer_to_important_question);

    if (answer_to_important_question == "y") {
        db_test(&db, nullptr, true);
    } else {
        delete_db(&db);
    }
```

```
    return 0;
}
```