Содержание

[**Введение** 4](#_Toc136392332)

[**1. Логирование** 6](#_Toc136392333)

[**1.1. Порождающий паттерн проектирования Builder** 6](#_Toc136392334)

[**1.2. Реализация логгера** 8](#_Toc136392335)

[**2. Распределение памяти** 9](#_Toc136392336)

[**2.1. Аллокатор с освобождением в рассортированном списке** 11](#_Toc136392337)

[**2.1.1. Выделение памяти** 11](#_Toc136392338)

[**2.1.2. Освобождение памяти** 11](#_Toc136392339)

[**2.1.3. Оценка сложности операций** 12](#_Toc136392340)

[**2.2. Аллокатор с освобождением с дескрипторами границ** 12](#_Toc136392341)

[**2.2.1. Выделение памяти** 12](#_Toc136392342)

[**2.2.2. Освобождение памяти** 14](#_Toc136392343)

[**2.2.3. Оценка сложности операций** 15](#_Toc136392344)

[**2.3. Аллокатор с алгоритмом системы двойников** 15](#_Toc136392345)

[**2.3.1. Выделение памяти** 16](#_Toc136392346)

[**2.3.2. Освобождение памяти** 17](#_Toc136392347)

[**2.3.3. Оценка сложности операций** 17](#_Toc136392348)

[**3. Бинарные деревья поиска** 18](#_Toc136392349)

[**3.1. Реализация сбалансированных деревьев** 19](#_Toc136392350)

[**3.2. АВЛ дерево** 23](#_Toc136392351)

[**3.1.1. Балансировка поддерева** 25](#_Toc136392352)

[**3.1.2. Вставка** 28](#_Toc136392353)

[**3.1.3. Удаление** 28](#_Toc136392354)

[**3.1.4. Поиск** 28](#_Toc136392355)

[**3.2. Красно-черное дерево** 28](#_Toc136392356)

[**3.2.1. Вставка** 30](#_Toc136392357)

[**3.2.2. Удаление** 32](#_Toc136392358)

[**3.2.3. Поиск** 35](#_Toc136392359)

[**3.3. Косое дерево** 35](#_Toc136392360)

[**3.3.1. Вставка** 38](#_Toc136392361)

[**3.3.2. Удаление** 38](#_Toc136392362)

[**3.3.3. Поиск** 39](#_Toc136392363)

[**4. Разработка и реализация приложения, управляющего хранилищем** 39](#_Toc136392364)

[**4.1. Основной функционал приложения** 39](#_Toc136392365)

[**4.2. Интерактивный диалог с пользователем** 39](#_Toc136392366)

[**4.3. Механизм, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени** 40](#_Toc136392367)

[**4.4. Поиск по различным порядкам отношений на пространстве данных** 43](#_Toc136392368)

[**4.5. Эффективное хранение строк с помощью паттерна Flyweight** 43](#_Toc136392369)

[**4.6. Кастомизация реализаций ассоциативных контейнеров, репрезентирующих коллекции данных** 45](#_Toc136392370)

[**4.7. Кастомизация аллокаторов для размещения объектов данных** 46](#_Toc136392371)

[**4.8. Логирование приложения** 46](#_Toc136392372)

[**5. Руководство пользователя** 47](#_Toc136392373)

[**6. Вывод** 53](#_Toc136392374)

[**7. Приложение** 53](#_Toc136392375)

**Введение**

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

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

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

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

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

Во время работы приложения возможно выполнение также следующих операций:

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

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

Помимо основного функционала реализованы дополнительные возможности:

* Механизм, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени
* Механизм, позволяющий выполнять эффективный поиск по различным отношениям порядка на пространстве данных
* Возможность кастомизации (при создании) реализаций ассоциативных контейнеров, репрезентирующих коллекции данных
* Возможность кастомизации (для заданного пула схем) аллокаторов для размещения объектов данных
* Логирование приложения в файловые потоки ввода, причем конфигурирование логгера обеспечивается на основе файла со структурой JSON

Данные о доставке содержат следующие поля:

* **Id пользователя**
* **Id доставки**
* Фамилия пользователя
* Имя пользователя
* Отчество пользователя
* Адрес электронной почты пользователя
* Номер телефона пользователя
* Комментарий пользователя
* Дата/время доставки

**Полужирным шрифтом** выделены поля, формирующие уникальный ключ объекта данных

# **1. Логирование**

## **1.1. Порождающий паттерн проектирования Builder**

Паттерн Builder отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

Поэтапное создание продукта означает его построение по частям. После того как построена последняя часть, продукт можно использовать.

Класс builder определяет интерфейс для построения отдельных частей продукта, а соответствующие подклассы builder\_concrete реализуют его подходящим образом.

Логгер конфигурируется двумя командами класса builder:

Add\_stream - принимает на вход название файлового потока вывода и значение уровня логирования

Build - отдает указатель на сконфигурировнный логгер на внешний уровень

Базовый класс builder:

class builder  
{  
  
public:  
  
 virtual builder\* add\_stream(

std::string const&,

logger::severity) = 0;  
  
virtual builder\* clear() = 0;  
  
virtual logger\* build() const = 0;  
  
virtual ~builder();

};

Производный класс builder\_concrete:

class builder\_concrete final : public builder  
{

private:

std::vector<std::pair<std::string, logger::severity>> \_streams;

public:

builder\* add\_stream(

std::string const& name\_stream,  
 logger::severity level) override;

builder\* clear() override;

logger\* build() const override;  
  
};

## **1.2. Реализация логгера**

Базовый класс logger содержит коллекцию потоков вывода, ключом которой является название потока, а значением - пара из указателя на открытый поток и количества логгеров, владеющих потоком, перечисление уровней логирования, а также виртуальную функцию log, принимающая на вход 2 аргумента - название потока и уровень логирования, и реализация которой остается на подклассе.

Подкласс logger\_concrete содержит локальную коллекцию потоков, аналогичную базовому классу logger и реализацию функции log

Базовый класс logger:

class logger  
{  
  
protected:  
  
 static std::map<std::string, std::pair<std::ofstream\*, size\_t>> \_collection\_streams;  
  
public:  
  
 enum class severity  
 {  
 TRACE,  
 DEBUG,  
 INFORMATION,  
 WARNING,  
 ERROR,  
 CRITICAL  
 };  
  
 virtual logger\* log(  
 const std::string&,  
 severity) = 0;  
  
 virtual ~logger();  
  
};

Производный класс logger\_concrete:

class logger\_concrete final : public logger  
{  
  
private:  
  
 std::map<std::string, std::pair<std::ofstream\*, severity>> \_streams\_log;  
  
public:  
  
 explicit logger\_concrete(  
 std::vector<std::pair<std::string, severity>> const& streams);  
  
 logger \*log(

const std::string& target,

severity level) override;  
  
 ~logger\_concrete() override;

};

# **2. Распределение памяти**

Аллокатор представляет собой объект управляющий памятью. Под аллокатор отведен блок памяти заданного размера, которым он управляет. Память аллокатора содержит системную информацию и блоки памяти, которые запросил выделить внешний объект. Каждый аллокатор реализован по определенному алгоритму, который описан в книге Д.Э. Кнут “Искусство программирования”.

В каждом аллокаторе доступно выделение и освобождение памяти

Способ выделения зависит от аллокатора, но методы поиска подходящего для выделения блока памяти одинаковы (за исключением аллокатора с алгоритмом системы двойников):

* Метод первого подходящего - поиск первого блока памяти, размер которого удовлетворяет размеру запрашиваемой памяти
* Метод худшего подходящего - поиск блока памяти, размер которого наименьший среди всех блоков, удовлетворяющих размеру запрашиваемой памяти
* Метод лучшего подходящего - поиск блока памяти, размер которого наибольший среди всех блоков, удовлетворяющих размеру запрашиваемой памяти

Каждый класс аллокатора представляет собой реализацию интерфейса класса memory:

class memory  
{   
public:  
   
 enum class METHOD\_SUITABLE  
 {  
 FIRST\_SUITABLE,  
 BEST\_SUITABLE,  
 WORSE\_SUITABLE  
 };  
  
 memory(  
 memory const&) = delete;

void operator=(  
 memory const&) = delete;  
  
 memory() = default;  
   
virtual ~memory() = default;  
  
 virtual void \*allocate(  
 size\_t const&) const = 0;  
  
 void \*operator+=(  
 size\_t const &size) const;  
  
 virtual void deallocate(  
 void\*) const = 0;  
  
 void operator-=(  
 void\*) const;  
};

## **2.1. Аллокатор с освобождением в рассортированном списке**

В блоке памяти, управляемым аллокатором, содержится следующая информация:

* Размер памяти
* Указатель на первый свободный блок
* Внешний аллокатор
* Указатель на логгер

Каждый свободный блок памяти, который располагаются в памяти, в виде списка, содержит данные:

* Размер свободного блока
* Указатель на следующий свободный блок

### **2.1.1. Выделение памяти**

Для выделения блока памяти из-под аллокатора достаточно пройтись по всему списку свободных блоков и найти подходящий блок памяти по одному из 3 методов. Метод выделения памяти конфигурируется при создании аллокатора.

### **2.1.2. Освобождение памяти**

Освобождение памяти очень схоже с вставкой узла в односвязный список. Для этого нужно:

Переставить указатель на следующий свободный блок у блока, после которого будет вставлен новый (освобожденный) участок памяти

Переставить указатель на следующий свободный блок у вставленного блока

Проверить не стоят ли свободные блоки рядом и если стоят, то “склеить” их

### **2.1.3. Оценка сложности операций**

Т.к. выделение и освобождение блока памяти очень схоже с поиском и вставкой в односвязном списке соответственно, то сложность операций оценить не трудно:

Выделение - O(n)

Освобождение - O(n)

## **2.2. Аллокатор с освобождением с дескрипторами границ**

Аналогично (за исключением одной тонкости) аллокатору из пункта 3.1. (Аллокатор с освобождением в рассортированном списке) в блоке памяти содержится:

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

Каждый занятый блок памяти содержит:

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

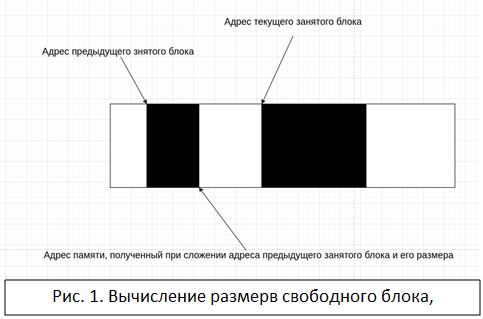
### **2.2.1. Выделение памяти**

Выделение памяти происходит несколько сложнее. Теперь мы проходимся не по списку свободных блоков, а по списку занятых блоков.

Но в самом начале, когда память представляет собой один большой свободный блок памяти, нет необходимости проходить весь список занятых блоков (т.к. он пока еще даже и не сформировался, потому что занятых блоков нет). Достаточно только переставить указатель на первый занятый блок, проставить размер и указатели на предыдущий и занятый блок (они будут равны nullptr)

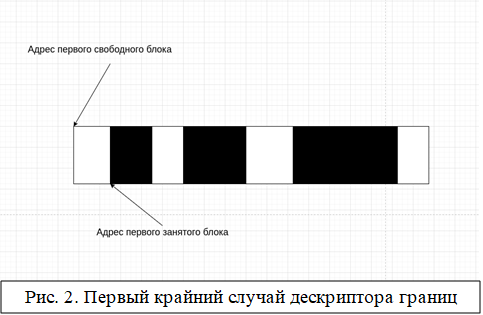
Для вычисления размера свободного блока памяти необходимо вычесть из адреса текущего занятого блока адрес предыдущего занятого блока, к которому прибавлен размер предыдущего занятого блока.

На рисунке показано размещение занятых блоков и как при помощи них вычислить размер свободного блока, заключенного между занятыми.

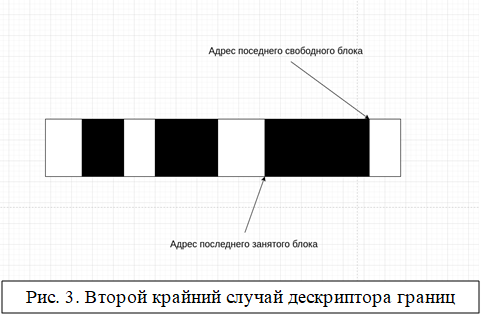


Чтобы выделить блок памяти необходимо пройтись по всему списку занятых блоков, вычисляя размеры свободных блоков, и найти подходящий. Помимо обычного прохода списка стоит учесть крайние случаи:

Указатель на первый занятый блок находится не в начале управляемого блока памяти (т.е. перед первым занятым блоком есть свободный блок)



Указатель на последний занятый блок находится не в конце управляемого блока памяти (т.е. после последнего занятого блока памяти есть свободный блок)



### **2.2.2. Освобождение памяти**

Освобождение блока памяти абсолютно аналогично удалению узла из двусвязного списка, но сам блок для удаления искать не надо. В памяти, указатель на который подается в функцию для освобождения памяти, уже располагаются указатели на предыдущий и следующий занятый блок. Достаточно лишь переставить у них нужные указатели.

### **2.2.3. Оценка сложности операций**

Т.к. операции схожи с поиском и удалением узла из двусвязного списка, то легко оценить сложность таких операций

Выделение - O(n)

Освобождение - O(1) (из-за того что нет надобности искать предыдущий и следующий занятый блок, т.к. указатели на них есть в памяти, которую запросили освободить)

## **2.3. Аллокатор с алгоритмом системы двойников**

Алгоритм системы двойников заключается в следующей особенности - каждый свободный блок содержит двойника, причем размер каждого свободного блока является числом, которое можно представить в виде степени двойки. Двойник - это необязательно свободный блок, который имеет или будет иметь (при освобождении блока свободные блоки стоящие рядом соединяются в единый) такой же размер, как и у текущего.

Вычисление адреса двойника описано в книге Д.Э. Кнут “Искусство программирования”, но способ описанный в книге, можно модифицировать с помощью операции XOR.

* Для начала нужно вычислить адрес блока относительно начала памяти
* Вычислить размер блока, возведя двойку в степень, которая располагается в системной информации блока
* Вычислить результат операции XOR относительного адреса блока и его размера
* К началу памяти прибавить результат операции XOR

Каждый свободный блок памяти содержит следующую информацию:

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

### **2.3.1. Выделение памяти**

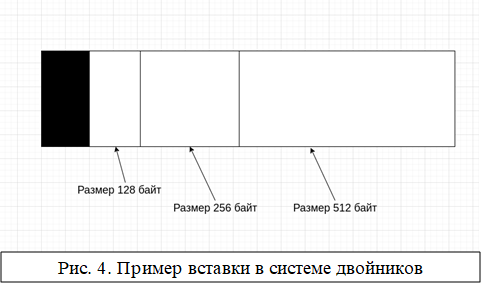
Необходимо пройтись по списку свободных блоков, найти подходящий и выполнить деление блока пополам, если половина блока все еще удовлетворяет запрашиваемому размеру памяти, и расставить указатели новых блоков, которые образовались в ходе деления текущего.

Например:

Память имеет размер в 1024 байт

Необходимо выделить 100 байт

Образуется такая картина:



Из одного блока размером 1024 байт образовалось 3 новых свободных блоков размерами 512, 256 и 128 байт. При этом, когда делился блок размером 256 байт, один из новых блоков размером 128 байт отдался на внешний уровень.

### **2.3.2. Освобождение памяти**

При освобождении памяти, для начала нужно вставить новый блок, аналогично вставке узла в односвязном списке, при этом расставив указатели определенным образом. Далее, идем к блоку-двойнику и проверяем свободен ли он и совпадает ли размер его памяти с текущим блоком, если да, то соединить их в один единый блок. Такой алгоритм несколько облегчает слияние блоков, но не дает прироста в скорости освобождения.

### **2.3.3. Оценка сложности операций**

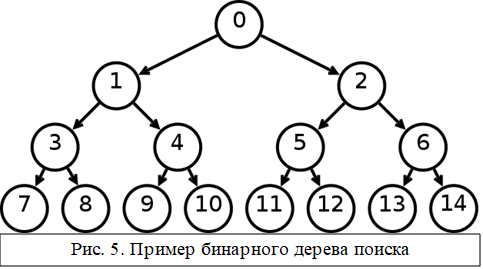
Выделение - O(n) (схоже с поиском блока в односвязном списке)

Освобождение - O(n) (схоже со вставкой блока в двусвязном списке)

# **3. Бинарные деревья поиска**

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны nullptr) называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.



## **3.1. Реализация сбалансированных деревьев**

Сбалансированные деревья были придуманы, чтобы обеспечить сложность выполнения операций вставки, удаления и поиска за O(log(n)). Из-за того, что обычное бинарное дерево поиска может выродиться в односвязный список, или не всегда может иметь высоту равную log(n) (n - количество узлов в дереве), то и сложность операций вставки, удаления и поиска не всегда равна логарифму. Эту проблему решают сбалансированные деревья. С помощью определенных алгоритмов, которые будут описаны далее, обеспечивается логарифмическая высота дерева, что позволяет гарантированно выполнять все операции за O(log(n)).

Т.к. все деревья представленные в данном приложении являются некой модификацией бинарного дерева поиска, то их реализацию можно оформить через паттерн Template Method, суть которого заключается в определении общего алгоритма поведения подклассов. В данном примере шаблонный метод применяется к операциям вставки, удаления и поиска элемента по уникальному ключу в дереве.

Вставка, удаление и поиск разделены на два этапа:

1. Сама операция, которая во всех представленный деревьях выполняется одинаково
2. Действия, которые необходимо выполнить после операции (например, балансировка). Эти действия определяются уже самим подклассом

Для реализации необходимо создать класс бинарного дерева, шаблонные методы которого выглядят следующим образом:

InsertionTemplate Method

class bin\_insertion\_template\_method : private memory\_holder, private logger\_holder  
{  
  
private:  
  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*\_tree;  
  
private:  
  
 memory \*get\_memory() const override;  
  
 logger \*get\_logger() const override;  
  
 virtual size\_t get\_node\_size() const;  
  
public:  
  
 explicit bin\_insertion\_template\_method(  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*tree);  
  
 void insert(  
 tkey const &key,  
 tvalue value);  
  
private:  
  
 void insert\_inner(  
 tkey const &key,  
 tvalue value,  
 bin\_node \*&subtree\_root\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
  
protected:  
  
 virtual void after\_insert\_inner(  
 tkey const &key,  
 bin\_node \*&subtree\_root\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
};

Функция insert\_inner на вход принимает указатель на поддерево, откуда надо начать поиск места для вставки (по дефолту это указатель на корень дерева) и стек для сбора пути к вставленному узлу.

Функция after\_insert\_inner на вход принимает указатель на поддерево, куда был вставлен узел, а также стек, содержащий путь до вставленного узла

Аналогично строится шаблонный метод для поиска и удаления:

Find Template Method

class bin\_find\_template\_method  
{  
  
private:  
  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*\_tree;  
  
public:  
  
 bool find(  
 typename associative\_container<tkey, tvalue>::key\_value\_pair \*target\_key\_and\_result\_value,  
 bin\_node \*&subtree\_address);  
  
 explicit bin\_find\_template\_method(  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*tree);  
  
private:  
  
 bool find\_inner(  
 typename associative\_container<tkey, tvalue>::key\_value\_pair \*target\_key\_and\_result\_value,  
 bin\_node \*&subtree\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
  
protected:  
  
 virtual void after\_find\_inner(  
 typename associative\_container<tkey, tvalue>::key\_value\_pair \*target\_key\_and\_result\_value,  
 bin\_node \*&subtree\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);

};

Removing Template Method

class bin\_removing\_template\_method : private memory\_holder, private logger\_holder  
{  
  
private:  
  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*\_tree;  
  
private:  
  
 memory \*get\_memory() const override;  
  
 logger \*get\_logger() const override;  
  
 virtual void get\_info\_deleted\_node(  
 bin\_node \*deleted\_node,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
  
public:  
  
 explicit bin\_removing\_template\_method(  
 binary\_search\_tree<tkey, tvalue, tkey\_comparer> \*tree);  
  
 tvalue remove(  
 tkey const &key,  
 bin\_node \*&tree\_root\_address);  
  
private:  
  
 virtual tvalue remove\_inner(  
 tkey const &key,  
 bin\_node \*&subtree\_root\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
  
protected:  
  
 virtual void after\_remove\_inner(  
 tkey const &key,  
 bin\_node \*&subtree\_root\_address,  
 std::stack<binary\_search\_tree::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive);  
  
};

Помимо этого определены классы logger\_holder и memory\_holder, которые являются простой оберткой над логгером и аллокатором. Они нужны, чтобы не проверять в коде реализации дерева на nullptr указатели на логгер и аллокатор. Если они равны nullptr, то логи выводится в консоль, а выделение памяти происходит из глобальной кучи.

## **3.2. АВЛ дерево**

АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Но для вставки и удаления потребуются операции, выполняющие балансировку поддерева.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

Т.е. нам необходимо в узле хранить еще и высоту поддерева. Для этого можно пронаследоваться от узла бинарного дерева поиска и доопределить дополнительное поле:

Структура узла бинарного дерева поиска

struct bin\_node  
{  
 tkey key;  
 tvalue value;  
 bin\_node \*left\_subtree;  
 bin\_node \*right\_subtree;  
 bool is\_left\_child;  
 bool is\_right\_child;  
}

Структура узла АВЛ дерева

struct avl\_node : binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node  
{  
 size\_t height;  
};

Определим вспомогательные функции, связанные с высотой:

Первая является оберткой для поля height, она возвращает высоту дерева

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
int avl\_tree<tkey, tvalue, tkey\_comparer>::get\_height(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*node)  
{  
 return node == nullptr ? 0 : (reinterpret\_cast<avl\_node\*>(node))->height;  
}

Вторая восстанавливает корректное значение поля height заданного узла

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
void avl\_tree<tkey, tvalue, tkey\_comparer>::fix\_height(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*\*subtree\_address)  
{  
 if (subtree\_address != nullptr &&  
 \*subtree\_address != nullptr)  
 {  
 int height\_left = (\*subtree\_address != nullptr) ? (get\_height((\*subtree\_address)->left\_subtree)) : 0;  
 int height\_right = (\*subtree\_address != nullptr) ? (get\_height((\*subtree\_address)->right\_subtree)) : 0;  
  
 reinterpret\_cast<avl\_node\*>(\*subtree\_address)->height = std::max(height\_left, height\_right) + 1;  
 }  
}

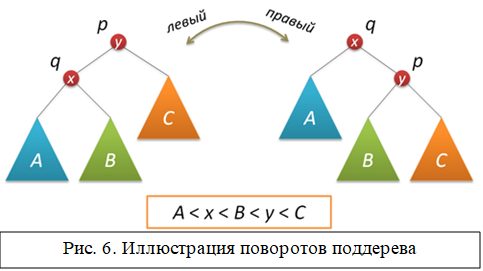
Третья вычисляет разницу высот между поддеревьями

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
int avl\_tree<tkey, tvalue, tkey\_comparer>::balance\_factor(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*subtree\_address)  
{  
 int height\_left = get\_height(subtree\_address->left\_subtree);  
 int height\_right = get\_height(subtree\_address->right\_subtree);  
  
 return height\_left - height\_right;  
}

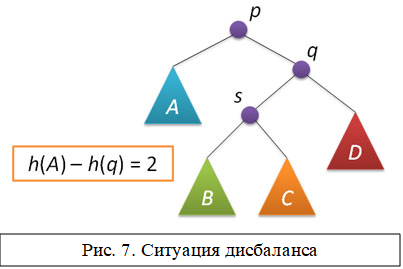
### **3.1.1. Балансировка поддерева**

В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда разница высот некоторых узлов оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются повороты вокруг тех или иных узлов дерева.

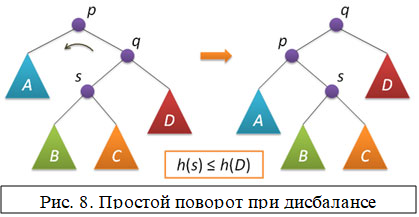
Простой поворот вправо (влево) производит следующую трансформацию дерева:



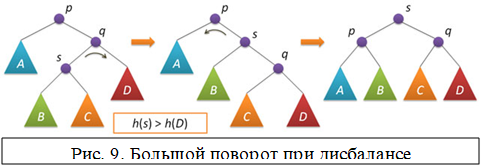
Рассмотрим теперь ситуацию дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.



Для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D)



Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



В данной реализации балансировка дерева происходит простым путем: в функции after\_insert\_inner или after\_remove\_inner поступает стек, который содержит путь до заданного узла, далее, необходимо пройтись по всему стеку, применяя операцию балансировки для каждого узла, лежащего в стеке.

Таким образом, в реализации АВЛ дерева доопределяется лишь один шаг - это операция, которую необходимо выполнить после вставки или удаления. А операция поиска не требует никаких модификаций.

### **3.1.2. Вставка**

После вставки для каждого узла, находящего в стеке, содержащий путь к вставленному узлу, вызывается функция балансировки. Таким образом, доопределяется только функция after\_insert\_inner

### **3.1.3. Удаление**

Аналогично вставке: вызывается функция, которая балансирует каждый узел, находящийся в стеке

### **3.1.4. Поиск**

При поиске никаких модификаций не требуется, он аналогичен поиску в обычном бинарном дереве поиска

## **3.2. Красно-черное дерево**

Красно-черное дерево - это тоже сбалансированное бинарное дерево поиска. Оно кардинально отличается от АВЛ дерева и имеет ряд своих особенностей:

Узел может быть либо красным, либо чёрным и имеет двух потомков;

Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;

Все листья, не содержащие данных — чёрные.

Оба потомка каждого красного узла — чёрные.

Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Аналогично узлу АВЛ дерева, строится узел и красно-черного дерева.

Структура узла красно-черного дерева:

enum class color\_node  
{  
 RED,  
 BLACK  
};  
  
struct red\_black\_node : public binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node  
{  
 color\_node color;  
};

Вспомогательная функция, возвращающая цвет узла:

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
typename red\_black\_tree<tkey, tvalue, tkey\_comparer>::color\_node red\_black\_tree<tkey, tvalue, tkey\_comparer>::get\_color(  
 red\_black\_node \*current\_node)  
{  
 return current\_node == nullptr ? color\_node::BLACK : current\_node->color;  
}

А также вспомогательная функция, собирающая информацию об удаляемом узле, которая понадобится при удалении.

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
void red\_black\_tree<tkey, tvalue, tkey\_comparer>::red\_black\_removing\_template\_method::get\_info\_deleted\_node(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*deleted\_node,  
 std::stack<typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive)  
{  
 \_information\_deleted\_node = std::make\_unique<red\_black\_node>();  
  
 if (path\_to\_subtree\_root\_exclusive.empty())  
 {  
 \_information\_deleted\_node->is\_left\_child = false;  
 \_information\_deleted\_node->is\_right\_child = false;  
 }  
 else  
 {  
 if ((\*path\_to\_subtree\_root\_exclusive.top())->right\_subtree == deleted\_node)  
 {  
 \_information\_deleted\_node->is\_left\_child = false;  
 \_information\_deleted\_node->is\_right\_child = true;  
 }  
  
 if ((\*path\_to\_subtree\_root\_exclusive.top())->left\_subtree == deleted\_node)  
 {  
 \_information\_deleted\_node->is\_left\_child = true;  
 \_information\_deleted\_node->is\_right\_child = false;  
 }  
 }  
  
 \_information\_deleted\_node->key = deleted\_node->key;  
 \_information\_deleted\_node->value = deleted\_node->value;  
 \_information\_deleted\_node->left\_subtree = deleted\_node->left\_subtree;  
 \_information\_deleted\_node->right\_subtree = deleted\_node->right\_subtree;  
 \_information\_deleted\_node->color = reinterpret\_cast<red\_black\_node\*>(deleted\_node)->color;  
}

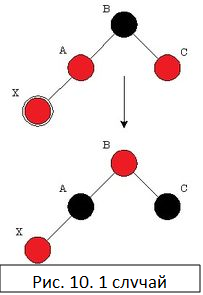
### **3.2.1. Вставка**

Новый узел в красно-чёрное дерево добавляется на место одного из листьев, окрашивается в красный цвет и к нему прикрепляется два листа (так как листья являются абстракцией, не содержащей данных, их добавление не требует дополнительной операции). Что происходит дальше, зависит от цвета близлежащих узлов. Заметим, что:

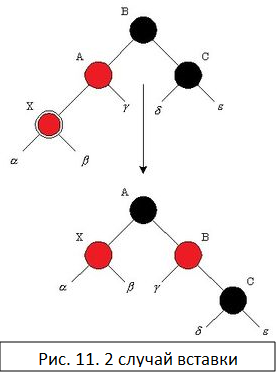
* Свойство 3 (Все листья чёрные) выполняется всегда.
* Свойство 4 (Оба потомка любого красного узла — чёрные) может нарушиться только при добавлении красного узла, при перекрашивании чёрного узла в красный или при повороте.
* Свойство 5 (Все пути от любого узла до листовых узлов содержат одинаковое число чёрных узлов) может нарушиться только при добавлении чёрного узла, перекрашивании красного узла в чёрный (или наоборот), или при повороте.

Вставляем новый узел с nullptr-потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента красный, то достаточно рассмотреть только два случая:

1. "Дядя" этого узла тоже красный. Тогда просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" - в красный. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет.



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



### **3.2.2. Удаление**

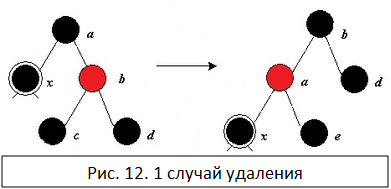
Во-первых в данной реализации для удаления узла потребуется вспомогательная функция, которая вбирает в себя информацию об удаленном узле. Т.к. удаление сначала происходит в функции бинарного дерева поиска, а балансировка определяется красно-черным деревом.

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на nullptr.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

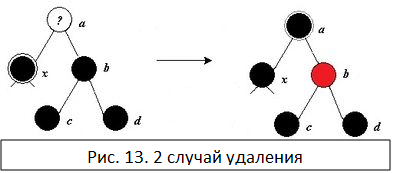
Проверим балансировку дерева. Т.к. при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

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

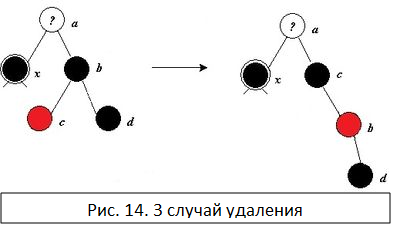


2. Если брат текущей вершины был чёрным, то получаем три случая:

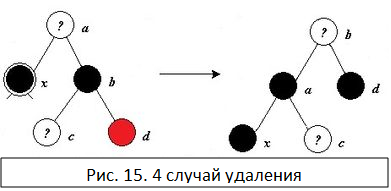
Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины.



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



В же у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца - в чёрный, делаем вращение и выходим из алгоритма.



Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. При удалении выполняется не более трёх вращений.

### **3.2.3. Поиск**

Поиск ничем не отличается от поиска в бинарном дереве, так что данный метод не требует никаких модификаций.

## **3.3. Косое дерево**

Косое дерево является двоичным деревом поиска, в котором поддерживается свойство сбалансированности. Это дерево принадлежит классу «саморегулирующихся деревьев», которые поддерживают необходимый баланс ветвления дерева, чтобы обеспечить выполнение операций поиска, добавления и удаления за логарифмическое время от числа хранимых элементов. Это реализуется без использования каких-либо дополнительных полей в узлах дерева (как, например, в Красно-чёрных деревьях или АВЛ-деревьях, где в вершинах хранится, соответственно, цвет вершины и глубина поддерева). Вместо этого «расширяющие операции» (splay operation), частью которых являются вращения, выполняются при каждом обращении к дереву.

Вспомогательные операции, которые понадобятся для реализации дерева:

Splay - операция поднимающая заданный узел в корень дерева путем поворотов

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
void splay\_tree<tkey, tvalue, tkey\_comparer>::splay(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*&subtree\_address,  
 std::stack<typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node\*\*> &path\_to\_subtree\_root\_exclusive)  
{  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*\*parent = nullptr;  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*\*grand\_parent = nullptr;  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*\*great\_grand\_parent = nullptr;  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*current\_node = subtree\_address;  
  
 while (!path\_to\_subtree\_root\_exclusive.empty())  
 {  
 parent = path\_to\_subtree\_root\_exclusive.top();  
 path\_to\_subtree\_root\_exclusive.pop();  
  
 if (!path\_to\_subtree\_root\_exclusive.empty())  
 {  
 grand\_parent = path\_to\_subtree\_root\_exclusive.top();  
 path\_to\_subtree\_root\_exclusive.pop();  
  
 if (!path\_to\_subtree\_root\_exclusive.empty())  
 {  
 great\_grand\_parent = path\_to\_subtree\_root\_exclusive.top();  
 path\_to\_subtree\_root\_exclusive.pop();  
 }  
 }  
  
 if ((\*parent)->left\_subtree == current\_node)  
 {  
 if (grand\_parent == nullptr)  
 {  
 this->right\_rotate(parent, grand\_parent);  
 current\_node = \*parent;  
 }  
 else  
 {  
 if ((\*grand\_parent)->right\_subtree == \*parent)  
 {  
 this->right\_rotate(parent, grand\_parent);  
 this->left\_rotate(grand\_parent, great\_grand\_parent);  
 current\_node = \*grand\_parent;  
 }  
 else  
 {  
 this->right\_rotate(grand\_parent, great\_grand\_parent);  
 this->right\_rotate(grand\_parent, great\_grand\_parent);  
 current\_node = \*grand\_parent;  
 }  
 }  
 }  
 else  
 {  
 if (grand\_parent == nullptr)  
 {  
 this->left\_rotate(parent, grand\_parent);  
 current\_node = \*parent;  
 }  
 else  
 {  
 if ((\*grand\_parent)->left\_subtree == \*parent)  
 {  
 this->left\_rotate(parent, grand\_parent);  
 this->right\_rotate(grand\_parent, great\_grand\_parent);  
 current\_node = \*grand\_parent;  
 }  
 else  
 {  
 this->left\_rotate(grand\_parent, great\_grand\_parent);  
 this->left\_rotate(grand\_parent, great\_grand\_parent);  
 current\_node = \*grand\_parent;  
 }  
 }  
 }  
  
 if (great\_grand\_parent != nullptr)  
 {  
 path\_to\_subtree\_root\_exclusive.push(great\_grand\_parent);  
 }  
  
 parent = nullptr;  
 grand\_parent = nullptr;  
 great\_grand\_parent = nullptr;  
 }  
}

Операция merge - сливает два поддерева, путем поднятия максимального узла в правом поддереве в корень и присоединения левого поддерева в качестве правого дочернего узла правого поддерева

template<  
 typename tkey,  
 typename tvalue,  
 typename tkey\_comparer>  
typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*splay\_tree<tkey, tvalue, tkey\_comparer>::merge(  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*&tree\_first,  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*&tree\_second)  
{  
 if (tree\_first == nullptr && tree\_second != nullptr)  
 {  
 return tree\_second;  
 }  
  
 if (tree\_first != nullptr && tree\_second == nullptr)  
 {  
 return tree\_first;  
 }  
  
 if (tree\_first != nullptr && tree\_second != nullptr)  
 {  
 std::stack<typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node\*\*> path;  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*max\_node = tree\_first;  
 typename binary\_search\_tree<tkey, tvalue, tkey\_comparer>::bin\_node \*\*parent = &tree\_first;  
  
 while (max\_node->right\_subtree != nullptr)  
 {  
 path.push(parent);  
 parent = &(max\_node->right\_subtree);  
 max\_node = max\_node->right\_subtree;  
 }  
  
 splay(max\_node, path);  
  
 tree\_first->right\_subtree = tree\_second;  
  
 return tree\_first;  
 }  
  
 return nullptr;  
}

### **3.3.1. Вставка**

Аналогично поиску: сначала производится вставка, как в бинарном дереве, а потом вызывается функция splay для вставленного узла

### **3.3.2. Удаление**

При удалении сначала вызывается функция splay для удаляемого узла, а потом функция merge для левого и правого поддерева удаляемого узла

### **3.3.3. Поиск**

Поиск выполняется как и в обычном бинарном дереве, за исключением того, что для найденного узла вызывается функция splay.

# **4. Разработка и реализация приложения, управляющего хранилищем**

## **4.1. Основной функционал приложения**

Хранилище данных состоит из:

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

Каждый пул, схема данных и коллекция имеет имя и представляют из себя ассоциативный контейнер. Пул и схема данных являются красно-черным деревом, а коллекция может являться АВЛ деревом, косым деревом, красно-черным деревом (в зависимости от выбора пользователя). С каждой коллекцией, пулом и схемой данных можно совершать определенные во введении операции.

## **4.2. Интерактивный диалог с пользователем**

Каждый запрос определяется тем, как он совершен - с консоли или из файла. В следствие этого определен и различный синтаксис для взаимодействия с хранилищем данных. Для того, чтобы запустить файл, можно ввести команду RUN. Введя эту команду, в функции внутри обработчиков будет подаваться параметр console, который и отвечает за поведение программы в зависимости от того, откуда вводится команда - с консоли или из файла.

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

Данный механизм реализован с помощью паттерна Chain of Responsibility и паттерна Command.

Паттерн Chain of Responsibility (цепочка обязанностей) относится к паттернам поведения объектов. Паттерн предназначен для создания цепочки объектов-получателей (обработчиков), которые получают и обрабатывают некоторый запрос от клиента. Клиент (клиентский объект) посылает запрос, который может быть обработан одним из объектов-получателей. Этот запрос рассматривается последовательно до его обработки Паттерн имеет следующие особенности реализации:

создается иерархия классов, содержащих ссылки (указатели) на своего преемника (successor)

на основе иерархии строится цепочка обработчиков. Здесь создаются объекты-обработчики, ссылающиеся на своего преемника (successor)

создается объект-клиент, который будет отправлять запрос. Этот объект-клиент ссылается на «ближайший» объект-обработчик. Термин «ближайший» означает, что объекты-обработчики могут ссылаться друг на друга с учетом некоторой иерархии классов. Эта иерархия строится по принципу «от конкретного к общему»

* объект-клиент посылает запрос путем вызова соответствующего метода (операции)
* Взаимодействие между клиентом и объектами-обработчиками можно сформулировать следующими шагами:
* создать объекты, которые обрабатывают запрос и сформировать их в цепочку;
* создать клиента, который указывает на первый объект в цепи;
* инициировать запрос клиентом для его просмотра объектами-обработчиками на предмет обработки. Запрос продвигается по цепочке пока некоторый объект не обработает его.

А паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Соответственно базовый класс handler должен содержать в себе указатель, на следующий обработчик, указатель на хранилище данных, доступ к которому осуществляется с помощью паттерна Singleton, и указатель на логгер:

class handler  
{  
  
public:  
  
 enum class TREE  
 {  
 RED\_BLACK\_TREE,  
 AVL\_TREE,  
 SPLAY\_TREE  
 };  
  
private:  
  
 static type\_data\_base \*\_data\_base;  
 static logger \*\_logger;  
  
protected:  
  
 handler \*\_next\_handler;  
  
public:  
  
 class command  
 {  
  
 public:  
  
 virtual void execute(  
 const std::vector<std::string> &params,  
 std::pair<type\_pool\*, memory\*> \*pool,  
 std::pair<type\_scheme \*, memory\*> \*scheme,  
 std::pair<type\_collection\*, std::map<std::string, type\_order\_collection\*>> \*collection,  
 std::istream &stream,  
 bool console) = 0;  
  
 virtual ~command() = default;  
  
 };  
  
  
public:  
  
 handler();  
  
 void set\_next(  
 handler \*next\_handler);  
  
 void accept\_request(  
 const std::string &request);  
  
 virtual void handle\_request(  
 const std::vector<std::string> &params,  
 std::istream &stream,  
 bool console) = 0;  
  
 static type\_data\_base \*get\_instance();  
  
 virtual ~handler();  
  
};

Таким образом, при обработке запроса клиента, запрос пройдется по всем обработчикам, найдет нужный и с помощью паттерна Command выполнит запрос.

Сам механизм, который позволяет обращаться к данным на заданный момент времени реализован с помощью хранения в std::map всех изменений объекта, где ключом является - время изменения, а значением - вектор пар, состоящих из имени изменяемого поля и самого значения, на которое его поменяли. Соответственно, когда пользователь хочет обратиться к данным на текущий момент времени, к данным последовательно применяются все изменения, находящиеся в std::map для этого объекта, иначе изменения применяются до тех пор, пока не найдется ключ, значение которого не удовлетворяет заданному моменту времени.

## **4.4. Поиск по различным порядкам отношений на пространстве данных**

Для поиска по различным порядкам отношений необходимо хранить несколько коллекций, где значением узлов является указатель на объект данных, находящийся в основной коллекции (у которой ключом которой является пара id пользователя и id доставки), а ключом - определенное значение объекта данных, например: комментарий, дата доставки, имя пользователя и т.д.

Т.е. каждая схема данных будет содержать пару, состоящую из указателя на основную коллекцию и std::map, содержащую название поля, на основе которого взят ключ коллекции и саму коллекцию.

using type\_scheme = associative\_container  
 <  
 std::string,  
 std::pair<type\_collection\*, std::map<std::string, type\_order\_collection\*>>\*  
 >;

## **4.5. Эффективное хранение строк с помощью паттерна Flyweight**

Легковес – это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

В данной реализации для каждой строки, которая является значением поля создается легковес. Управляет легковесами класс string\_flyweight\_factory, который содержит пул легковесов.

Леговес:

class string\_flyweight  
{  
  
private:  
  
 std::string \_value;  
  
public:  
  
 const std::string &get\_value() const;  
  
 void set\_value(  
 const std::string &value);  
};

Как он работает:

Например, какому-то новому объекту в коллекции присваивается значение поля, которое уже есть. Чтобы избежать хранения в памяти одинаковых строк, создается легковес и управляющий ими объект. Вместо присвоения значению поля строки, которая уже есть, ему присваивается указатель на легковес, находящийся в пуле. А если такой строки в пуле нет, то создаться новый легковес. Таким образом, избегается хранение одинаковых строк, что экономит оперативную память.

Класс, управляющий легковесами

class string\_flyweight\_factory  
{  
  
private:  
  
 std::unordered\_map<std::string, std::shared\_ptr<string\_flyweight>> \_string\_flyweights;  
  
public:  
  
 static string\_flyweight\_factory &get\_instance();  
  
 std::shared\_ptr<string\_flyweight> get\_string\_flyweight(  
 const std::string &value);

};

## **4.6. Кастомизация реализаций ассоциативных контейнеров, репрезентирующих коллекции данных**

С кастомизацией все проще: в файле или консоли указывается тип дерева, который пользователь хочет выбрать для коллекции объектов данных. В зависимости от того, что он написал в консоли или в файле, определяется значение перечислительного типа:

enum class TREE  
{  
 RED\_BLACK\_TREE,  
 AVL\_TREE,  
 SPLAY\_TREE  
};

Также реализованы вспомогательные функции, которые возвращают указатель на нужное дерево в зависимости от переданного в качестве параметра enum class TREE:

type\_collection \*handler::allocate\_collection(  
 memory \*memory\_init,  
 TREE tree)  
{  
 switch (tree)  
 {  
 case TREE::RED\_BLACK\_TREE:  
 return reinterpret\_cast<type\_collection\*>(new type\_collection\_rb(memory\_init));  
 case TREE::AVL\_TREE:  
 return reinterpret\_cast<type\_collection\*>(new type\_collection\_avl(memory\_init));  
 case TREE::SPLAY\_TREE:  
 return reinterpret\_cast<type\_collection\*>(new type\_collection\_splay(memory\_init));  
 default:  
 throw std::invalid\_argument("ERROR: Invalid number of type tree");  
 }  
}

## **4.7. Кастомизация аллокаторов для размещения объектов данных**

Аналогично кастомизации дерева для коллекции, выполняется и выбор аллокатора, который происходит при создании пула. Для того, чтобы сохранить указатель на созданный аллокатор при создании пула, он хранится в паре, вместе со схемой и пулами. Далее, когда дело доходит до создания коллекции, указатель на созданный аллокатор передается в конструктор дерева.

Т.е. типы данных для хранлища и пула выглядят так:

using type\_pool = associative\_container  
 <  
 std::string,  
 std::pair<type\_scheme\*, memory\*>  
 >;  
  
using type\_data\_base = associative\_container  
 <  
 std::string,  
 std::pair<type\_pool\*, memory\*>  
 >;

## **4.8. Логирование приложения**

При создании первого обработчика событий, который содержит в качестве указателя на следующий обработчик - начало цепочки, создается логгер, с помощью которого пишется системная информация в потоки вывода. Причем логгер конфигурируется с помощью файла типа JSON и паттерна builder. Для конфигурации логгера используется сторонняя библиотека nlohman.

handler::handler() :  
 \_next\_handler(nullptr)  
{  
 json\_builder \*builder = new json\_builder\_concrete();  
 \_logger = builder->build("json/config.json");  
}

# **5. Руководство пользователя**

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

NAME

REMOVE\_POOL - remove pool

SYNOPSIS

console: REMOVE\_POOL [pool\_name]

file: REMOVE\_POOL [pool\_name]

INPUT\_EXAMPLE\_CONSOLE:

REMOVE\_POOL pool

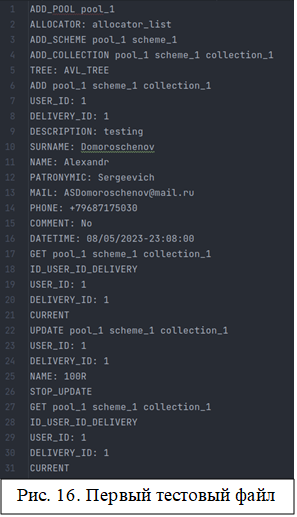
INPUT\_EXAMPLE\_FILE:

REMOVE\_POOL pool

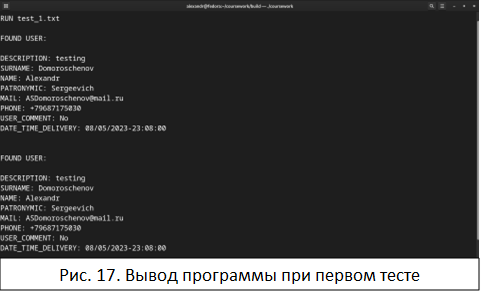
DESCRIPTION

Removes a pool that stores data schemas

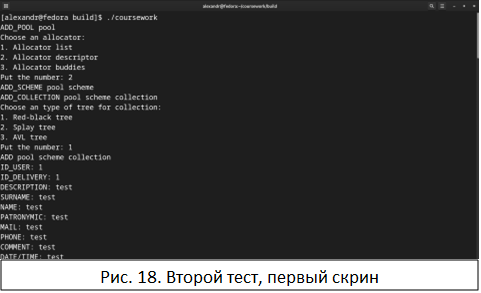
Например, давайте запустим текстовый файл, который создает пул, схему и коллекцию, затем добавляет нового пользователя, выводит его, обновляет информацию об имени и снова выводит:

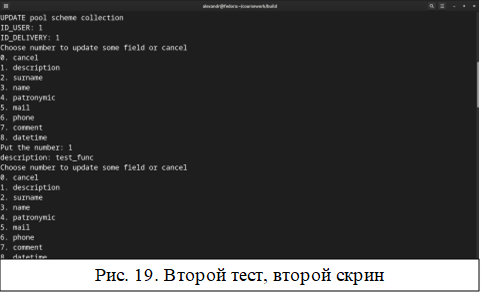


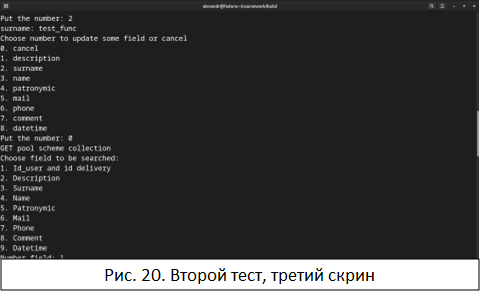
Вывод программы:

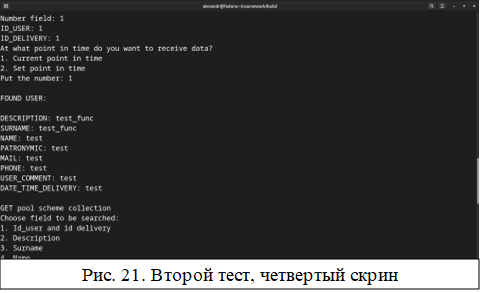


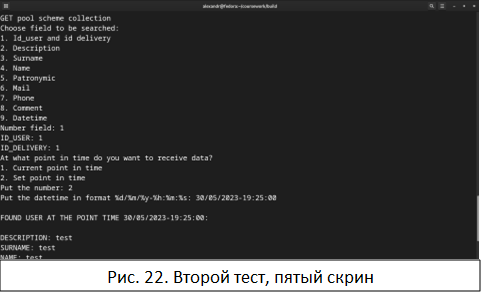
Или например опробуем функционал, позволяющий обращаться к данным на заданный момент времени:

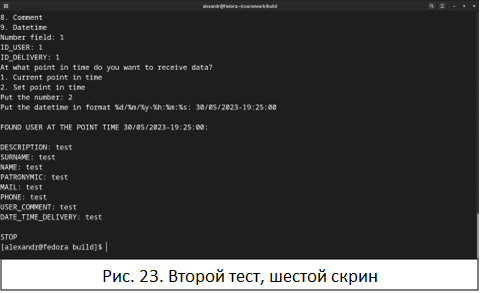








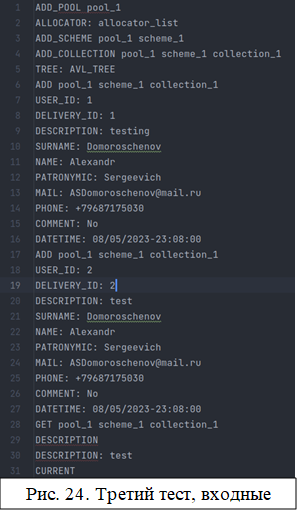




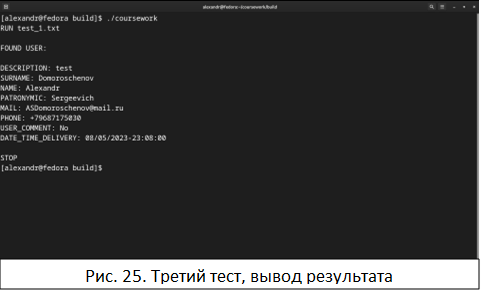
Как видно на скриншотах, мы попробовали обратиться к данным на текущий момент времени, получив значение полей описания и фамилии в виде test\_func, а при обращении к данным на момент времени, когда обновление еще не было совершено, получили ожидаемый результат.

Опробуем функционал для поиска по различным порядкам отношений.

Запустим файл вот с такими командами:



В коллекции содержатся 2 пользователя с разными ключами и полями описания. У первого пользователя описание - testing, а у второго - test. Попробуем найти пользователя у которого описание имеет значение test:



# **6. Вывод**

Таким образом, было создано приложение, управляющее хранилищем данных, которое имеет несколько уровней хранения (пул, схема, коллекция). Были использованы описанные аллокаторы, деревья, логгер, а также паттерны Flyweight, Builder, Singleton, Chain of Responsobilities, Command. Данное приложение позволяет управлять и манипулировать хранящимися данными, также оно имеет широкий функционал, которое обеспечивает гибкость и эффективность управления данными. Создание данного приложения потребовало от меня знаний об устройстве баз данных, эффективном управлении памяти и ресурсами, а также знаний о распределении памяти, стандартах языка C++ и сбалансированных деревьях.

# **7. Приложение**

Весь код можно увидеть, перейдя по ссылке на github: <https://github.com/ASDomoroschenov/relational-database-management-system>