**Содержание**

Введение……………………………………………………………..….2

Логирование в приложении…………………………………………....4

Выделение памяти. Аллокаторы.……………………………………...7

Красно-чёрное дерево…………………………………………..…….10

AVL-дерево………………………………………..………….………..17

База данных……………………………………...….…………………26

Хранение объектов строк……………………………………………..27

Руководство пользователя. Шаблоны реализованных команд…….30

Демонстрация функционала и работы приложения…………….......32

Вывод………………………………………………………...………...34

Список использованных источников………………………………...34

Приложение…………………………………………………………....34

**Введение.**

В рамках данного курсового проекта на языке программирования C++ было реализовано приложение, которое позволяет выполнять операции над коллекциями данных заданных типов (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, в котором каждый объект данных соответствует некоторому уникальному ключу. Для ассоциативного контейнера была выведена интерфейсная часть (в виде абстрактного класса C++) и реализован интерфейс. Взаимодействие с коллекцией объектов происходит посредством выполнения одной из операций над ней:

* добавление новой записи по ключу;
* чтение записи по её ключу;
* чтение набора записей с ключами из диапазона [minbound… maxbound];
* обновление данных для записи по ключу;
* удаление существующей записи по ключу.

Во время работы приложения возможно выполнение также следующих операций:

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

Поток команд, выполняемых в рамках работы приложения, поступает напрямую из консоли, или же из текстового файла, путь к которому пользователь может указать самостоятельно.

Данные о доставке содержат следующие поля:

* **id пользователя;**
* **id доставки;**
* Описание доставки;
* Фамилия пользователя;
* Имя пользователя;
* Отчество пользователя;
* Адрес электронной почты пользователя;
* Номер телефона пользователя;
* Адрес доставки;
* Комментарий пользователя;
* Дата/время доставки;

При том **id пользователя** и **id доставки** формируют уникальный ключ объекта данных.

**Логирование в приложении.**

Приложение содержит базовый класс logger. В свою очередь, этот класс содержит метод log и перечисление, состоящее из уровней логирования, которые в дальнейшем необходимо будет указывать при отправке каждого лога.

Листинг кода 1.

class logger

{

public:

enum class severity

{

trace,

debug,

information,

warning,

error,

critical

};

public:

virtual ~logger() noexcept = default;

public:

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

};

В свою очередь, производный от него класс logger\_concrete содержит реализацию метода log и локальную коллекцию потоков.

Листинг кода 2.

class logger\_concrete final : public logger

{

friend class logger\_builder\_concrete;

private:

std::map<std::string, std::pair<std::ofstream\*, logger::severity> > \_logger\_streams;

private:

static std::map<std::string, std::pair<std::ofstream\*, size\_t> > \_streams;

private:

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

public:

logger\_concrete(logger\_concrete const&) = delete;

logger\_concrete& operator=(logger\_concrete const&) = delete;

~logger\_concrete();

public:

logger const\* log(const std::string&, severity) const override;

};

Основную роль в реализации логирования играет класс logger\_builder, использующий порождающий паттерн проектирования Builder, который определяет интерфейс для построения отдельных частей продукта, а также производный класс logger\_builder\_concrete, который реализовывает его соответствующим образом.

Листинг кода 3.

class logger\_builder

{

public:

virtual logger\_builder\* add\_stream(std::string const&, logger::severity) = 0;

virtual logger\* construct() const = 0;

virtual logger\_builder\* config(std::string const& path) = 0;

public:

virtual ~logger\_builder() noexcept = default;

};

Листинг кода 4.

class logger\_builder\_concrete final : public logger\_builder

{

private:

std::map<std::string, logger::severity> \_construction\_info;

public:

logger\_builder\* add\_stream(std::string const&, logger::severity) override;

logger\* construct() const override;

logger\_builder\* config(std::string const& path) override;

};

Сам же логгер конфигурируется с помощью следующих команд:

add\_stream() – принимает на вход название файлового потока, уровень логирования.

construct() – возвращает указатель на внешний уровень настроенного логгера.

config() – принимает на вход файл с данными, обрабатывает его и вызывает construct().

**Выделение памяти. Аллокаторы.**

Выделение памяти для хранимых в коллекциях данных в программе происходит при помощи аллокаторов (на глобальной куче, c освобождением с дескрипторами границ). Пользователь может сам выбрать необходимый ему способ выделения памяти при создании пула.

**Аллокатор на глобальной куче.**

В данной реализации класс memory выступает в роли аллокатора. Он предоставляет нам методы для выделения и освобождения памяти. Поскольку сам класс memory является абстрактным, то мы также имеем и дочерний класс memory\_global\_heap, который реализует его методы.

Листинг кода 5.

class memory

{

public:

virtual void\* allocate(size\_t target\_size) const = 0;

virtual void deallocate(void\* target\_to\_deallocate) const = 0;

public:

virtual ~memory() = default;

};

Листинг кода 6.

class memory\_global\_heap final : public memory

{

private:

logger\* logger\_c;

public:

memory\_global\_heap(logger\* logger\_c = nullptr)

{

this->logger\_c = logger\_c;

};

public:

void\* allocate(size\_t target\_size) const override

{

auto\* result = ::operator new(sizeof(size\_t) + target\_size);

if (result == nullptr) {

throw logic\_error("Bad Alloc!");

}

\*reinterpret\_cast<size\_t\*>(result) = target\_size;

if (logger\_c != nullptr)

{

ostringstream optr;

optr << reinterpret\_cast<size\_t\*>(result) + sizeof(size\_t);

logger\_c->log(optr.str(), logger::severity::debug);

}

return reinterpret\_cast<void\*>(reinterpret\_cast<size\_t\*>(result) + 1);

}

void deallocate(void\* target\_to\_deallocate) const override

{

target\_to\_deallocate = reinterpret\_cast<size\_t\*>(target\_to\_deallocate) - 1;

if (logger\_c != nullptr)

{

string rstr = "[ ";

auto mem\_size = \*reinterpret\_cast<size\_t\*>(target\_to\_deallocate);

for (int i = 0; i < mem\_size; ++i) {

unsigned char r = \*(reinterpret\_cast<unsigned char\*>(target\_to\_deallocate) + sizeof(size\_t) + i);

rstr = rstr + to\_string(static\_cast<unsigned short>(r)) + " ";

}

rstr = "Deallocated block: " + rstr + "]";

ostringstream optr;

optr << reinterpret\_cast<size\_t\*>(target\_to\_deallocate) + sizeof(size\_t);

logger\_c->log(optr.str(), logger::severity::debug)

->log(rstr, logger::severity::debug);

}

::operator delete(target\_to\_deallocate);}};

**Освобождение с дескрипторами границ.**

Данный алгоритм базируется на следующем принципе: благодаря концепции двусвязного списка, который содержит занятые блоки, мы можем легко удалять элементы списка из произвольной его части.

Выделение в данном аллокаторе работает так: вначале мы ищем блок, размер которого удовлетворяет запрос пользователя. Если нас интересует первый подходящий, то мы ищем первый блок памяти, размер которого нам подходит, если лучший – то ищем такой блок, размер которого является самым маленьким среди подходящих, а если худший, то наоборот, самый большой. Для того, чтобы найти такой подходящий блок, нам нужно вычислять разницу между адресами уже занятых блоков и хранить указатели на предыдущий и следующий блок после текущего, своевременно их переставляя и совершая соответствующие проверки. После приобретения блоком статуса, занятого его можно вернуть пользователю.

В случае освобождения нам просто необходимо получить указатели на следующий и предшествующий блоки и поменять их указатели, проверив есть ли они вообще в списке.

**Красно-чёрное дерево.**

Красно-чёрное дерево является самобалансирующимся двоичным деревом поиска. Благодаря своей структуре, оно гарантирует логарифмический рост высоты дерева от числа узлов и достаточно быстрое выполнение операций дерева поиска (добавление удаление, поиск).

**Свойства(правила) красно-чёрного дерева:**

1. Корень дерева всегда является чёрным (за исключение поддеревьев).
2. Каждый узел является либо красным, либо чёрным. NIL-листья всегда чёрными.
3. У красного узла могут быть только чёрные потомки.
4. Для любого узла, путь от него до его листьев должен содержать одинаковое количество чёрных узлов. Это обеспечивает балансировку в высоте.

**Вставка в красно-чёрное дерево.**

При вставке нового элемента он красится в красный цвет. Заметим, что для выполнения первых двух свойств(условий) необходимо перекрашивать новые вершины в нужный цвет.

Теперь необходимо рассмотреть правила балансировки для выполнения оставшихся двух свойств.

**Первый случай: красный дядя.**

Если отец и дядя добавленного элемента красного цвета, то мы должны перекрасить отца и дядю в чёрный цвет, а деда, в свою очередь, в красный. Таким образом чёрная высота останется прежней. Не стоит забывать о том, что подобные изменения могут вызвать нарушение третьего свойства у деда, поэтому рекурсивно вызываем дальнейшую балансировку для него.

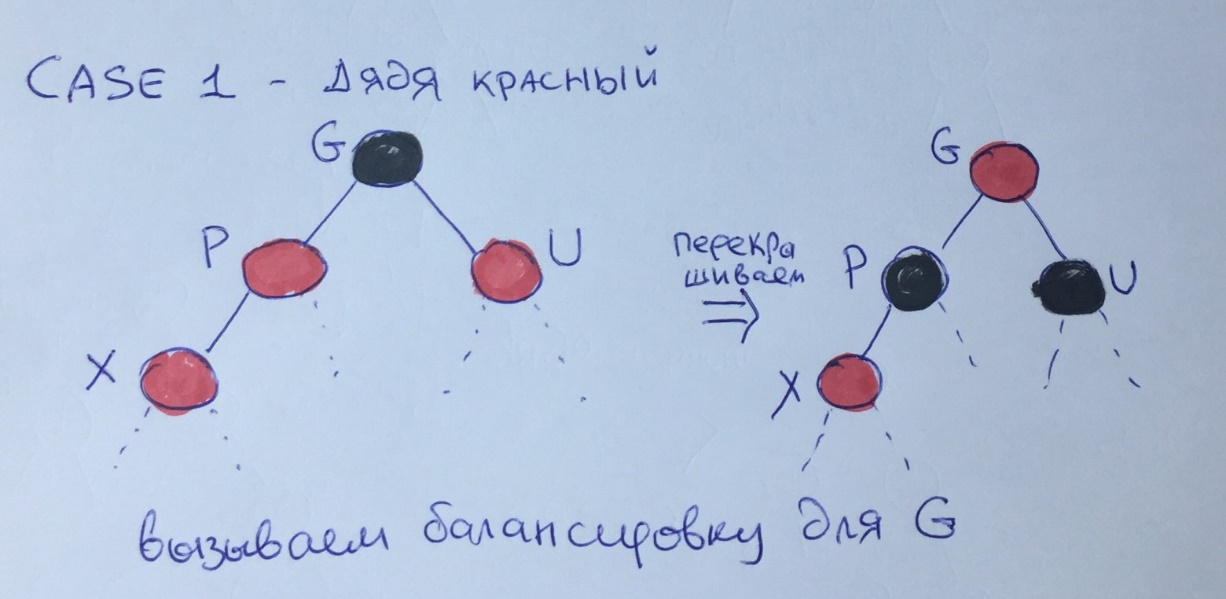


Рисунок 1

**Второй случай: чёрный дядя, папа и дед в разных сторонах.**

В таком случае нам необходимо свести ситуацию к третьему случаю, где папа и дед находятся по одну сторону от вставляемого элемента. Для этого выполним малый поворот от сына к его отцу, а затем вызовем балансировку (3 случай) для отца.

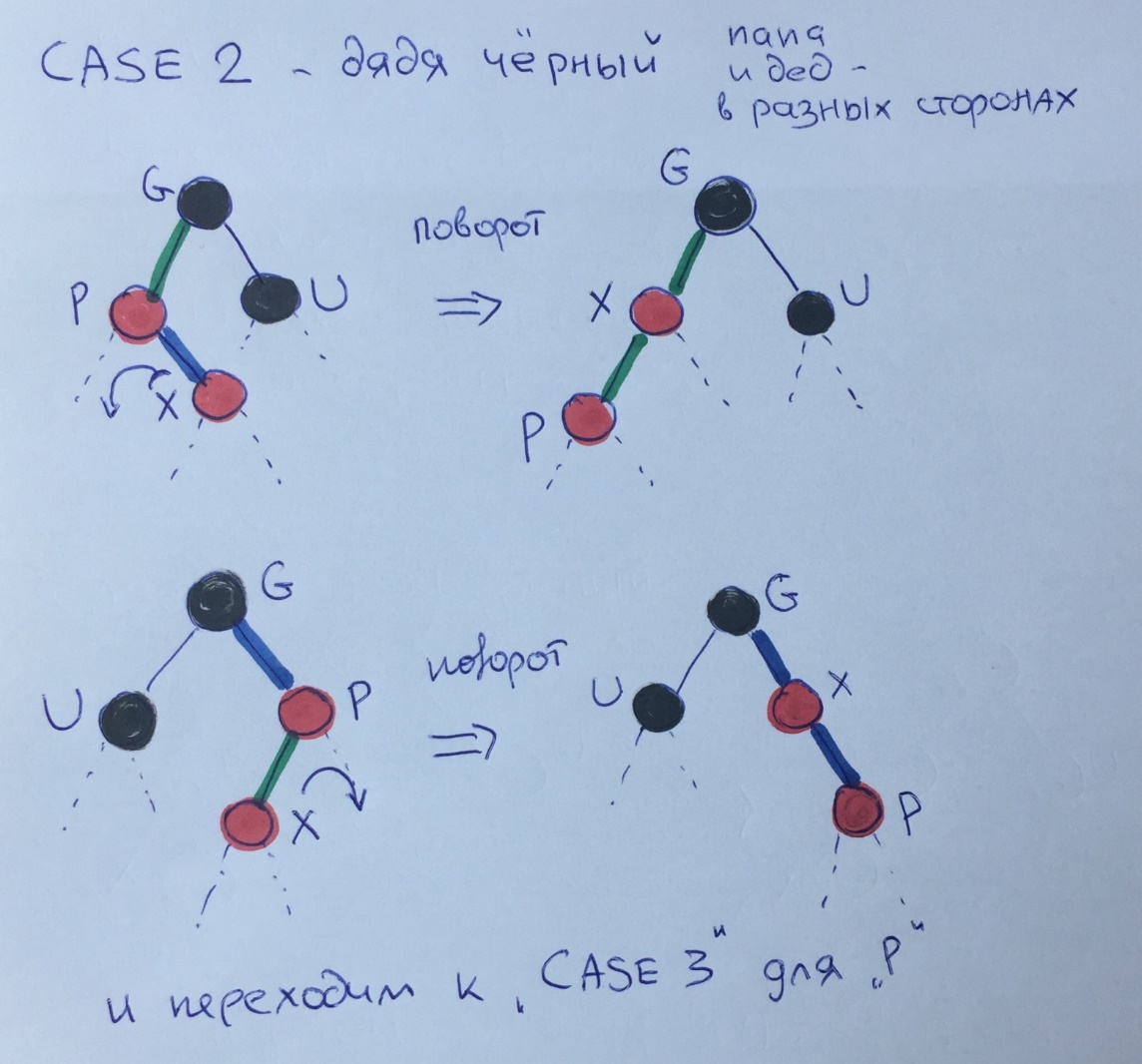


Рисунок 2

**Третий случай: чёрный дядя, папа и дед в одной стороне.**

Если папа и дед нашего элемента расположены по одну сторону от него, то мы должны совершить большой поворот от отца через деда к чёрному дяде, а затем перекрасить старого отца в чёрного и старого деда в красный.



Рисунок 3

**Удаление из красно-чёрного дерева.**

Для начала, необходимо рассмотреть самые базовые случаи:

1. Если мы имеем дело с красным узлом, у которого два ребёнка, то нам всего лишь нужно найти ближайший элемент, который меньше или больше удаляемого и поменять их местами, оставив цвет узлов прежним.
2. Если мы имеем дело с чёрным узлом, у которого два ребёнка, то нам необходимо сделать аналогичный обмен.
3. Если мы имеем дело с чёрный узлом с одним ребёнком, то алгоритм таков: если у чёрного элемента нет одного ребёнка, значит вместо него находится чёрный NIL с чёрной высотой 1. Из этого следует то, что с другой стороны должен быть красный узел без детей. А значит, для удаления этого элемента переносим значение красного элемента в чёрный узел.
4. Если мы имеем красный узел без детей, то мы удаляем элемент, а вместо него ставим ссылку на NIL.

**Удаление чёрного узла без детей. Балансировка.**

Наконец, мы переходим к самому сложному случаю, где нам потребуется балансировка после удаления.

Дело в том, что после удаления чёрного элемента меняется чёрная высота поддерева и нужно выполнить балансировку для его родителя.

Важно заметить, что в этом случае у нас появляется 12 различных случаев (6, если удалённый элемент является правым сыном и 6, если левым). Они являются симметричными, поэтому рассмотрим нашу задачу на примере правого сына.

**Первый случай: красный родитель, левый ребёнок чёрный с чёрными внуками.**

Имея красный узлы, мы можем их перемещать, тем самым восстанавливая баланс для тех или иных элементов. В этом случае мы можем спустить красный цвет от родителя к левому сыну. Как можно увидеть на рисунке, мы добились выравнивания чёрной высоты для родителя.

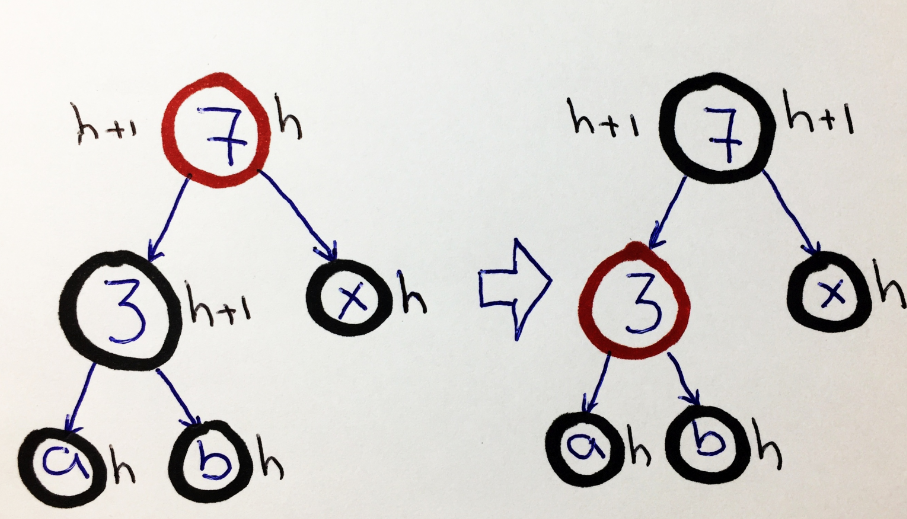


Рисунок 4

**Второй случай: красный родитель, левый ребёнок чёрный с левым красным внуком.**

Для начала, нам необходимо выполнить малый поворот направо от дяди к отцу. Таким манёвром мы увеличили высоту поддерева на 1. Сам дядя должен стать красным, а его новые сыновья – чёрными.

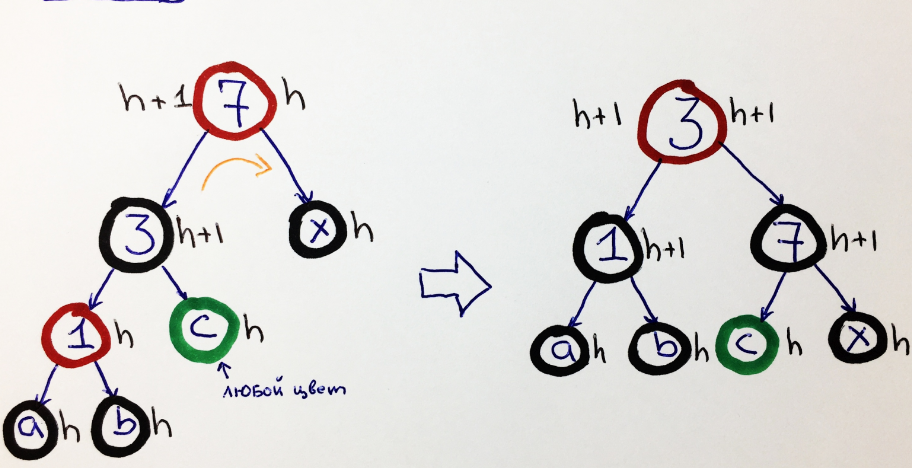


Рисунок 5

**Третий случай: Чёрный родитель, левый ребёнок красный, у правого внука чёрные правнуки.**

Так как у нас есть чёрные правнуки, то мы имеем право сделать внука красным и переместить в правое поддерево, выполнив малый правый поворот от дяди к отцу.

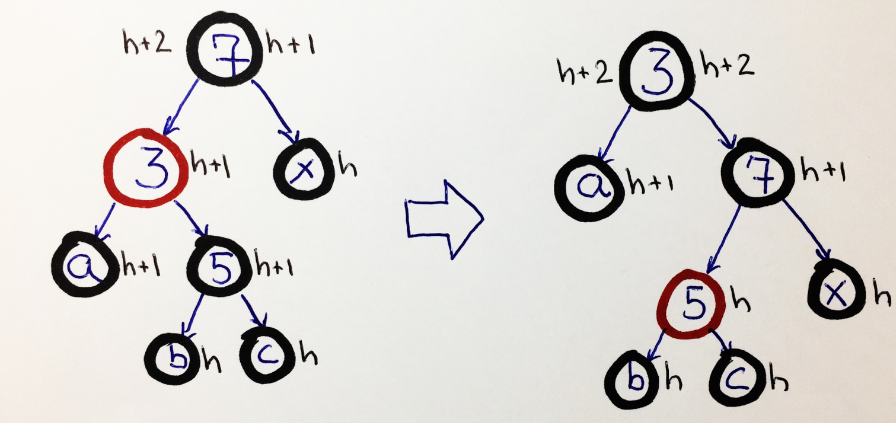


Рисунок 6

**Четвёртый случай: Чёрный родитель, левый сын красный, у правого внука левый правнук красный.**

Теперь нам необходимо выполнить два малых поворота: от правого внука к сыну, и затем от нового сына (старого правнука) к отцу. Узел, который изначально являлся левым правнуком, меняет свой цвет на чёрный.

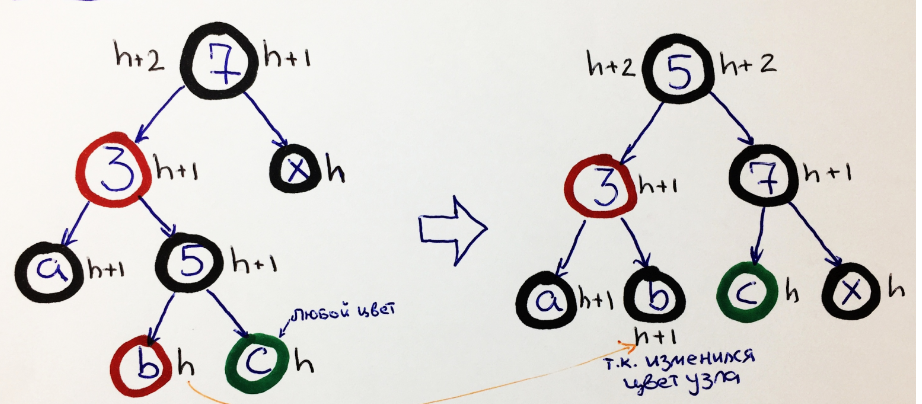


Рисунок 7

**Пятый случай: Чёрный родитель, левый сын чёрный с правым красным внуком.**

Наконец, мы перекрашиваем оставшегося красного внука в чёрный цвет и выполняем два малых поворота влево, как в четвёртом случае.

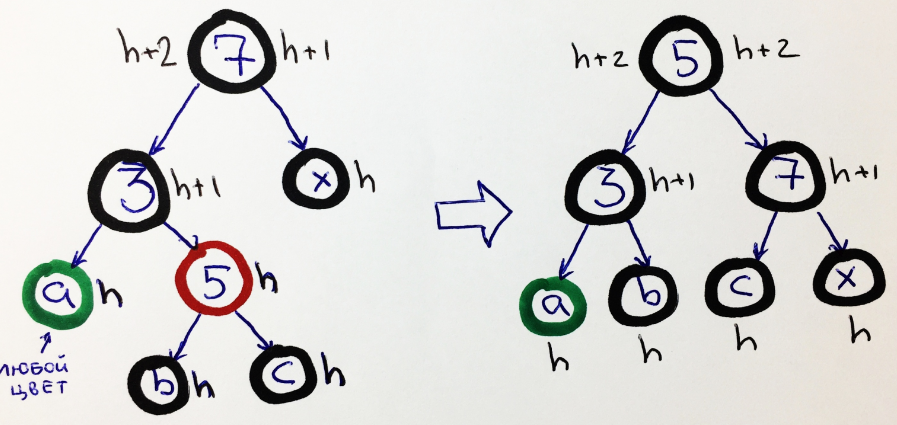


Рисунок 8

**Шестой случай: Чёрный родитель, чёрный левый сын с чёрными внуками.**

Так как мы больше не можем использовать красные узлы (их не осталось), то мы будем красить чёрный узел в красный, чтобы выровнять чёрную высоту родителя за счёт уменьшения чёрной высоты левого поддерева. Таким образом, чёрная высота структуры уменьшится на 1. Вызываем рекурсивно балансировку к предкам.

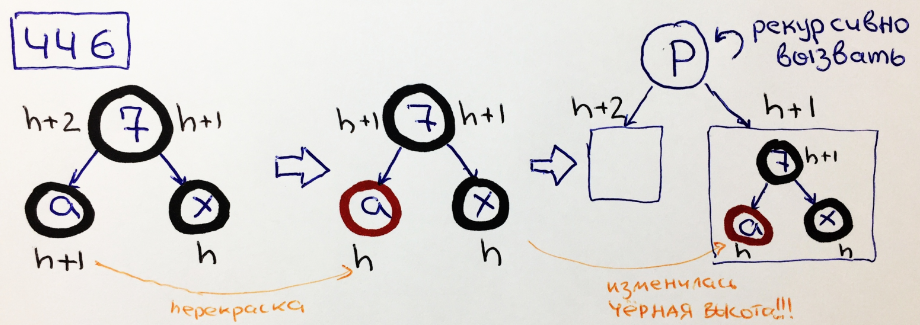


Рисунок 9

**АVL-дерево.**

AVL-дерево также является одной из модификаций бинарного дерева поиска.

Главная цель AVL-деревьев – поддерживать баланс в дереве путём автоматических операций поворотов, чтобы гарантировать, что высота дерева остаётся в логарифмических пределах от числа узлов.

**Свойства:**

1. Высота: разница в высоте между левым и правым поддеревьями для любого узла не может превышать 1. В противном случае выполняются операции поворотов для того, чтобы восстановить баланс.
2. Балансировка: после каждой вставки или удаления элемента выполняются операции поворотов.
3. Повороты: в AVL-деревьях используются четыре типа поворотов: левый поворот, правый поворот, лево-правый поворот и право-левый поворот.

**Балансировка дерева. Малые левый и правый повороты.**

Прежде чем переходить к операциям вставке и удаления, необходимо разобрать средства достижения сбалансированности дерева.

Балансировка применяется в том случае, если поддеревья одного узла различаются более чем на 1 уровень. Очевидно, что при добавлении или удалении узлов высота родителей в рекурсивной цепочке будет меняться, следовательно мы должны обновлять высоту для каждого узла, находящегося в стеке во время его скрутки.

**Малые левый и правый повороты.**

Нам необходимо использовать малый левый поворот в том случае, если мы хотим укоротить ветку, которая перевешивает справа.

Рассмотрим алгоритм малого левого поворота на примере с рисунка.



Рисунок 10

Шаг 1: создадим указатель на правый дочерний узел (8).

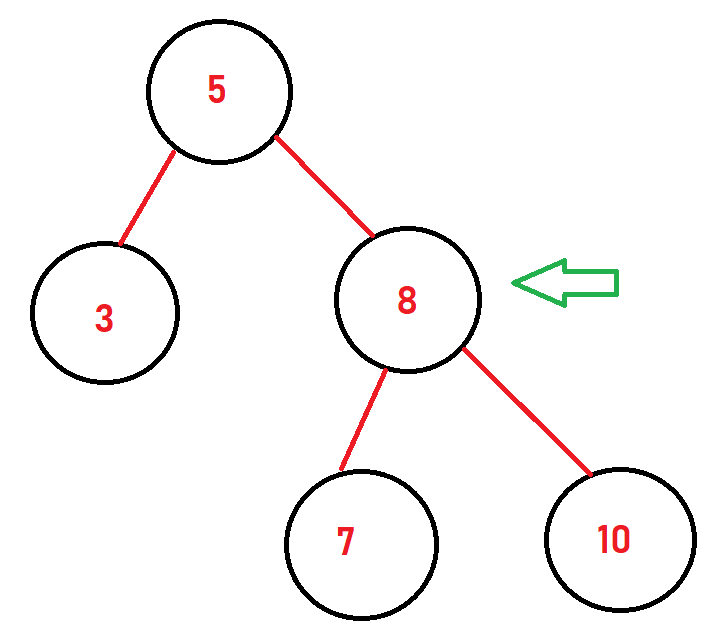


Рисунок 11

Шаг 2: меняем у (5) указатель правого поддерева на левое поддерево дочернего узла (7). Заметим, что мы можем это сделать, так как знаем, что любое из значений правого поддерева для (5) будет не меньше него.

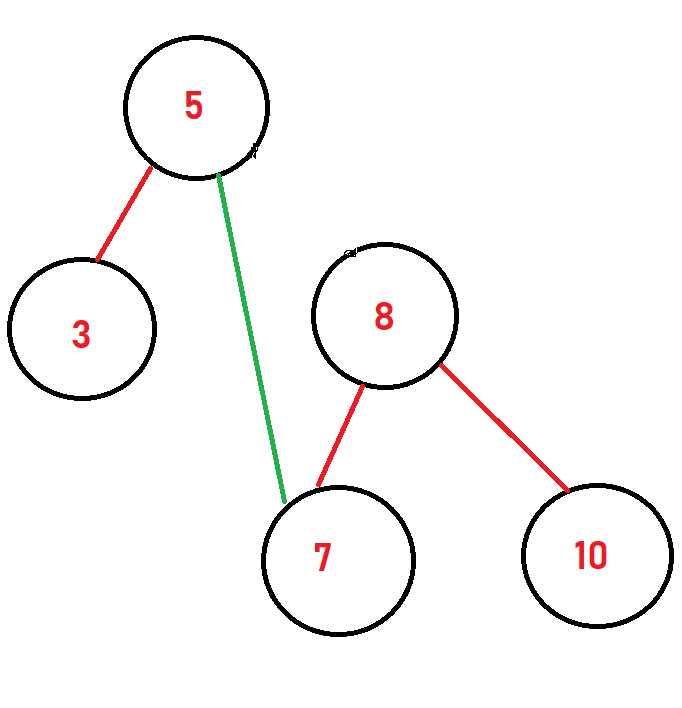


Рисунок 12

Шаг 3: меняем указатель левого поддерева нового корня (8) на старый корень (5).

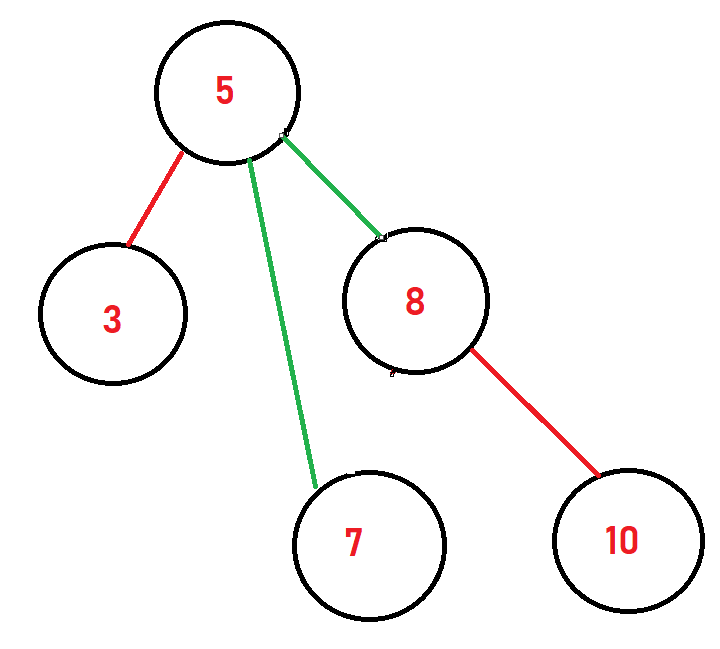


Рисунок 12

Теперь (8) – наш новый корень, (5) – его левое сын, а (7) – правый сын (5).

Поворот завершён.

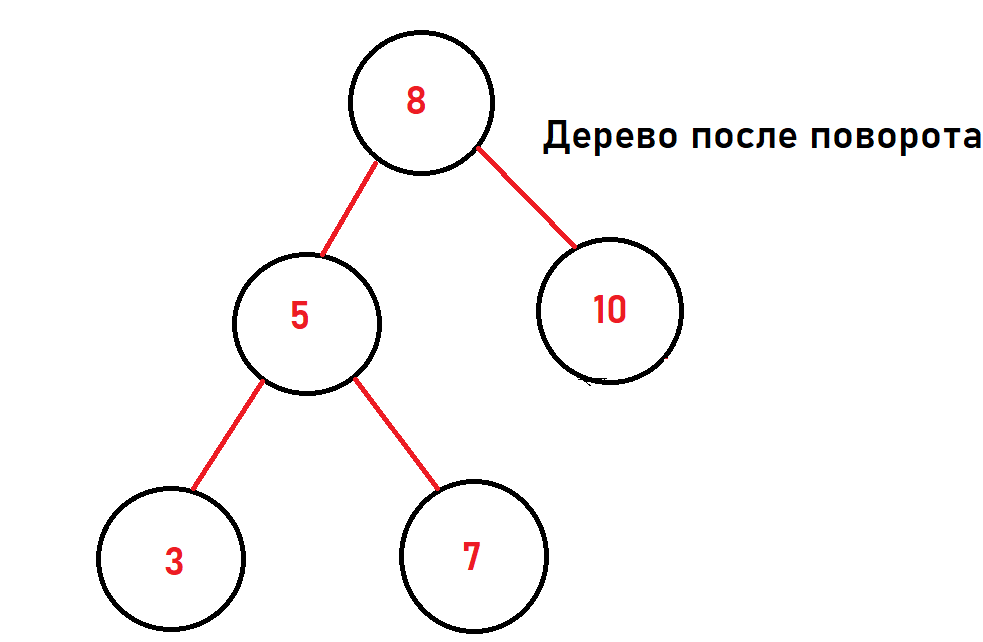


Рисунок 13

В свою очередь, нам необходимо использовать малый правый поворот в том случае, если мы хотим укоротить ветку, которая перевешивает справа.

Алгоритм малого правого поворота выглядит симметрично описанному выше.

**Вставка в AVL-дерево.**

Сама по себе вставка ничем не отличается от вставки в бинарное дерево поиска. Мы идём вниз по дереву, на каждом шагу сравнивая значение нового узла с текущими, и найдя нужное нам место производим вставку.

Однако, так как в AVL-дереве высота играет большую роль, после вставки мы должны вызвать балансировку. Наша задача сделать так, чтобы разница между высотами для каждого дочернего узла всегда была меньше или равна 1 по модулю.

**Дерево перегружено влево.**

Если мы имеем дело с деревом, которое перегружено влево, то тут может возникнуть два случая.

Первый случай является более простым, и для достижения сбалансированности нам необходимо сделать один малый поворот вправо.

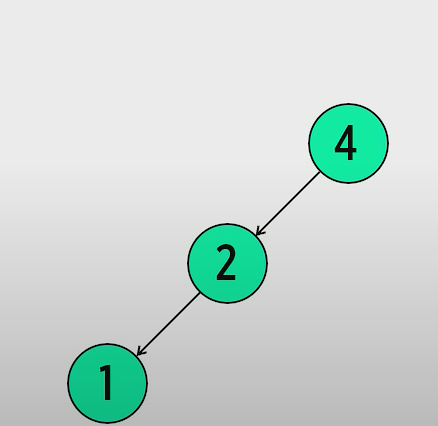
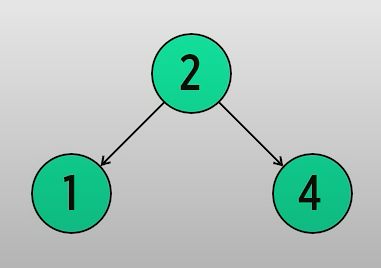
 

Рисунок 14 Рисунок 15

Однако, возможен и более сложный случай, когда обычный поворот вправо не сделает ситуацию лучше.

На следующем рисунке можно увидеть, что дерево перегружено влево, значит, по логике мы должны сделать малый правый поворот. Однако сделав его мы увидим, что теперь дерево стало перегружено вправо.



Рисунок 16

В таком случае нам необходимо сделать лево-правый поворот, состоящий из двух раннее описанных операций.

Для начала, выполняем малый поворот влево от (1), и получаем первый случай.

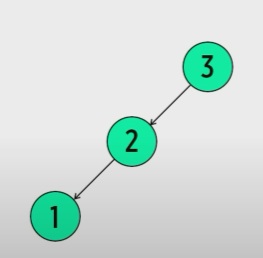


Рисунок 17

Затем, нам необходимо сделать малый правый поворот, в результате чего мы придём к балансу.

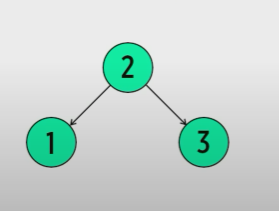


Рисунок 18

**Дерево перегружено вправо**.

Если же дерево перегружено вправо, то у нас возникают две абсолютно симметричные ситуации.

Случай 1:

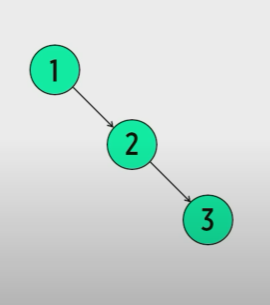
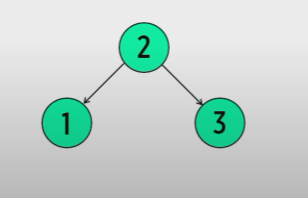
 

Рисунок 19 Рисунок 20

Случай 2:

Требуется право-левый поворот.

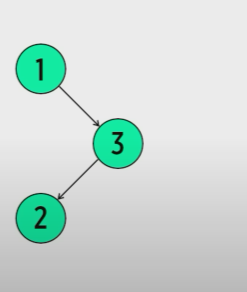


Рисунок 21

Необходимо сделать малый поворот вправо от (3).

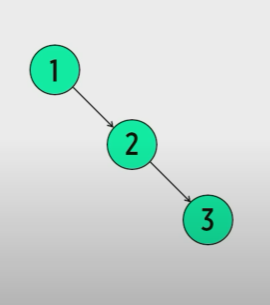
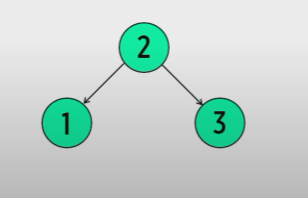


Рисунок 22

И затем малый поворот влево от (2).

 Рисунок 23

**Удаление из AVL-дерева.**

Удаление из AVL-дерева работает по таким же правилам, как и в бинарном дереве поиска. Только теперь после каждого удаления, поднимаясь вверх по цепочке во время скрутки стека мы будем обновлять высоты узлов и в случае необходимости сразу же их балансировать по раннее разобранным алгоритмам балансировки. Стоит отметить, что это будет выполнено не только для удаляемого узла, но и для того узла, на который мы удаляемый заменяем, в случае с двумя детьми, потому что его удаление может снова привести к изменению высот родительских узлов, поэтому мы рекурсивно должны пройтись по каждому такому узлу вплоть до удаляемого, обновляя все высоты.

**База данных.**

База данных представляет собой набор пулов, которые содержат схемы данных, которые содержат коллекции данных, содержащие объекты. Важно отметить, что каждый пул/схема/коллекция являются ассоциативным контейнером (в случае моей программы, они являются красно-чёрным деревом, коллекция может являться АВЛ-деревом). В дальнейшем будет описано, какие именно операции можно совершать относительно данной структуры.

Листинг кода 7.

red\_black\_tree<string, // главное дерево

red\_black\_tree<string, //дерево пуллов

red\_black\_tree<string, //дерево схем

associative\_container<delivery\_data, consumer\_info>\*, //коллекция

comparison\_string>\*,

comparison\_string>\*,

comparison\_string>\* \_database;

В свою очередь, в классе database было объявлено два метода:

dialogue\_with\_user() – реализованный интерактивный диалог с пользователем. Пользователь при этом может вводить конкретные команды и подавать на вход файл с потоком команд.

file\_parse\_options(read\_path) – позволяет исполнять команды, которые пользователь записал в файле в соответствии с предоставленными шаблонами.

**Хранение объектов строк.**

Изначальный алгоритм хранения строк не подразумевает тот факт, что данные о доставке пользователей могут дублироваться. Поэтому, необходимо обеспечить доступ к строковому пулу на основе порождающего паттерна проектирования “Одиночка”.

Само хранение строк, размещённых в объектах данных, осуществляется при помощи структурного паттерна проектирования “Приспособленец”.

Основными методами здесь являются get\_flyweight(),который получает на вход строку: если такой строки не было, то он вставляет её в коллекцию, а если была, то возвращает её и check\_flyweight, который помогает нам в удалении строк из коллекции и ведёт учёт количества той или иной строки.

Листинг кода 8.

class string\_flyweight final

{

private:

class string\_comparer

{

public:

string\_comparer() = default;

int operator()(const string& s\_c1, const string& s\_c2) const {

return s\_c1.compare(s\_c2);

}

};

string\_flyweight() :

\_alloc(new allocator\_border\_descriptors(500000)), \_strings\_pool(new red\_black\_tree<string, size\_t, string\_comparer>(\_alloc))

{};

memory\* \_alloc;

red\_black\_tree<string, size\_t, string\_comparer>\* \_strings\_pool;

public:

string\_flyweight(const string\_flyweight& other) = delete;

string\_flyweight(string\_flyweight&& other) = delete;

string\_flyweight& operator=(const string\_flyweight& other) = delete;

string\_flyweight& operator=(string\_flyweight&& other) = delete;

static string\_flyweight& get\_instance() {

static string\_flyweight \_instance;

return \_instance;

}

string& get\_flyweight(const string& str) {

if (!\_strings\_pool->find\_key(str)) {

\_strings\_pool->insert(str, 1);

}

else {

size\_t k = \_strings\_pool->get(str);

k++;

\_strings\_pool->update(str, k);

}

return \_strings\_pool->get\_key\_ref(str);

}

void check\_flyweight(const string& str) {

if (\_strings\_pool->get(str) <= 1) {

\_strings\_pool->remove(str);

}

else {

size\_t k = \_strings\_pool->get(str);

k--;

\_strings\_pool->update(str, k);

}

}

~string\_flyweight() {

if (\_strings\_pool != nullptr) {

delete \_strings\_pool;

}

if (\_alloc != nullptr) {

delete \_alloc;

}

}};

**Руководство пользователя. Шаблоны реализованных команд.**

1. Добавление новой записи по ключу.

insert: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

key: [consumer\_id,delivery\_id]

value: [delivery\_description] [consumer\_name] [consumer\_surname] [consumer\_patronymic] [consumer\_email] [consumer\_phone\_number] [consumer\_adress] [consumer\_comments] [delivery\_date\_time]

1. Чтение записи по её ключу.

read key: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

key: [consumer\_id,delivery\_id]

1. Чтение набора записей с ключами из диапазона [𝑚𝑖𝑛𝑏𝑜𝑢𝑛𝑑... 𝑚𝑎𝑥𝑏𝑜𝑢𝑛𝑑].

read range: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

keys: [consumer\_id, delivery\_id] [consumer\_id, delivery\_id]

1. Обновление данных для записи по ключу.

update key: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

key: [consumer\_id,delivery\_id]

value: [delivery\_description] [consumer\_name] [consumer\_surname] [consumer\_patronymic] [consumer\_email] [consumer\_phone\_number] [consumer\_adress] [consumer\_comments] [delivery\_date\_time]

1. Удаление существующей записи по ключу.

remove: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

key: [consumer\_id,delivery\_id]

1. Добавить пул данных.

add pool: [nameof\_pool] ['global heap' / 'border descriptors',size(for border descriptors),'best'/'worst'/'first'(for border descriptors)]

1. Удалить пул данных.

delete pool: [nameof\_pool]

1. Добавить схему данных.

add scheme: [nameof\_pool] [nameof\_scheme]

1. Удалить схему данных.

delete scheme: [nameof\_pool] [nameof\_scheme]

1. Добавить коллекцию.

add collection: [nameof\_pool] [nameof\_scheme] [nameof\_collection] ['avl tree'/'red black tree']

1. Удалить коллекцию

delete collection: [nameof\_pool] [nameof\_scheme] [nameof\_collection]

1. Сбросить базу данных (Данный шаблон работает только для реализации с файлом, для ввода через консоль предусмотрена непосредственная реализация в диалоге с пользователем).

reboot database:

**Демонстрация функционала и работы приложения.**

На вход пользователем подаётся некоторый текстовый файл, содержащий все вышеперечисленный команды.

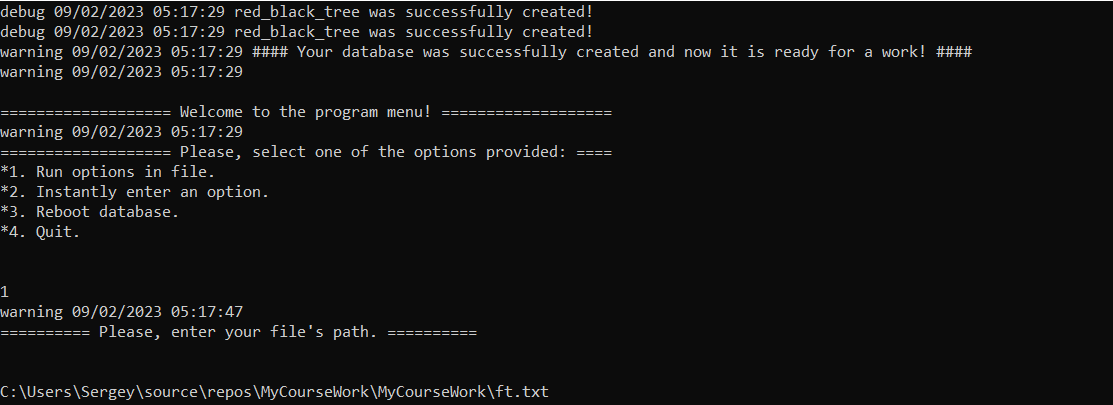


Рисунок 24

Результат работы.

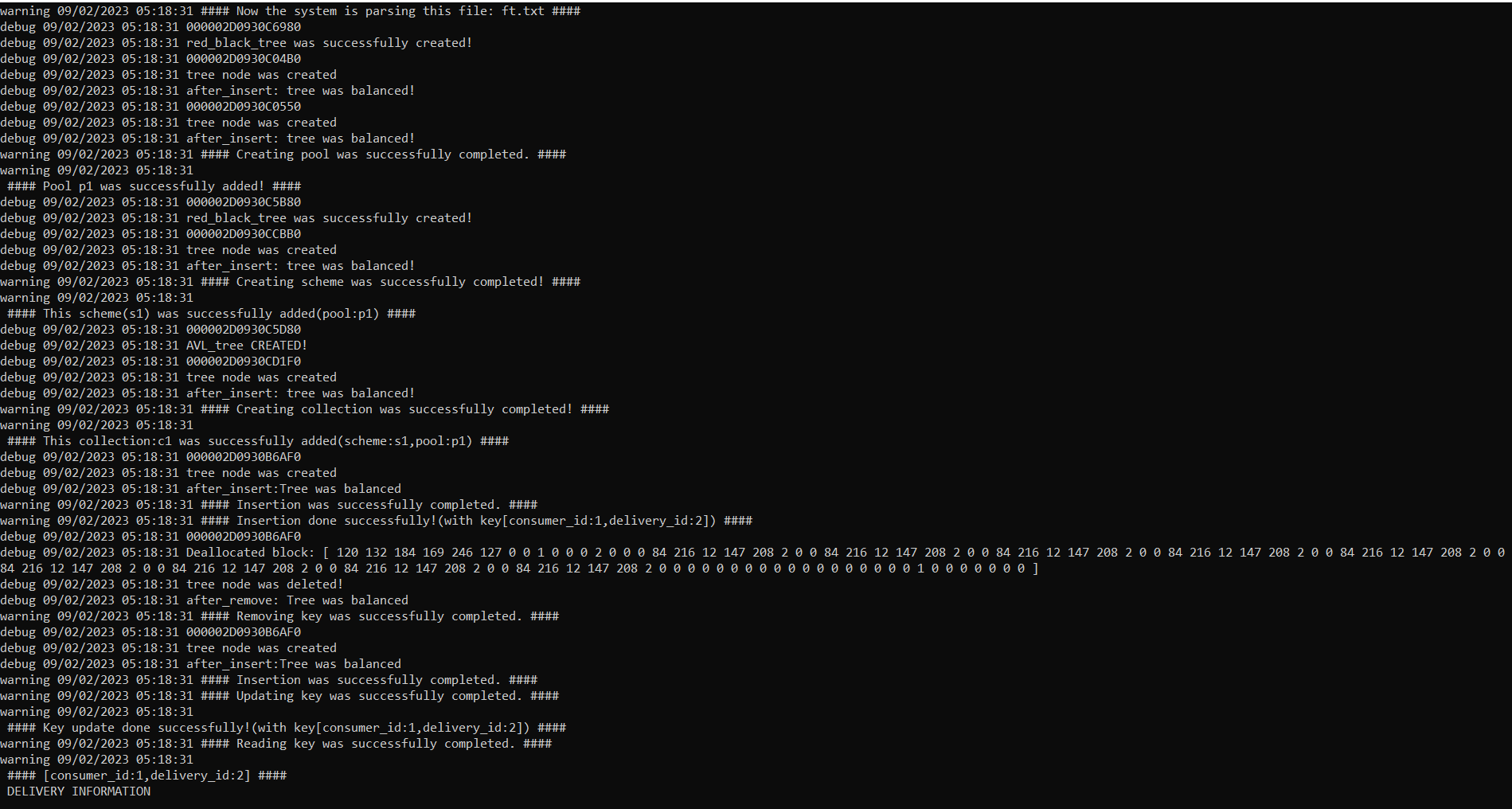


Рисунок 25

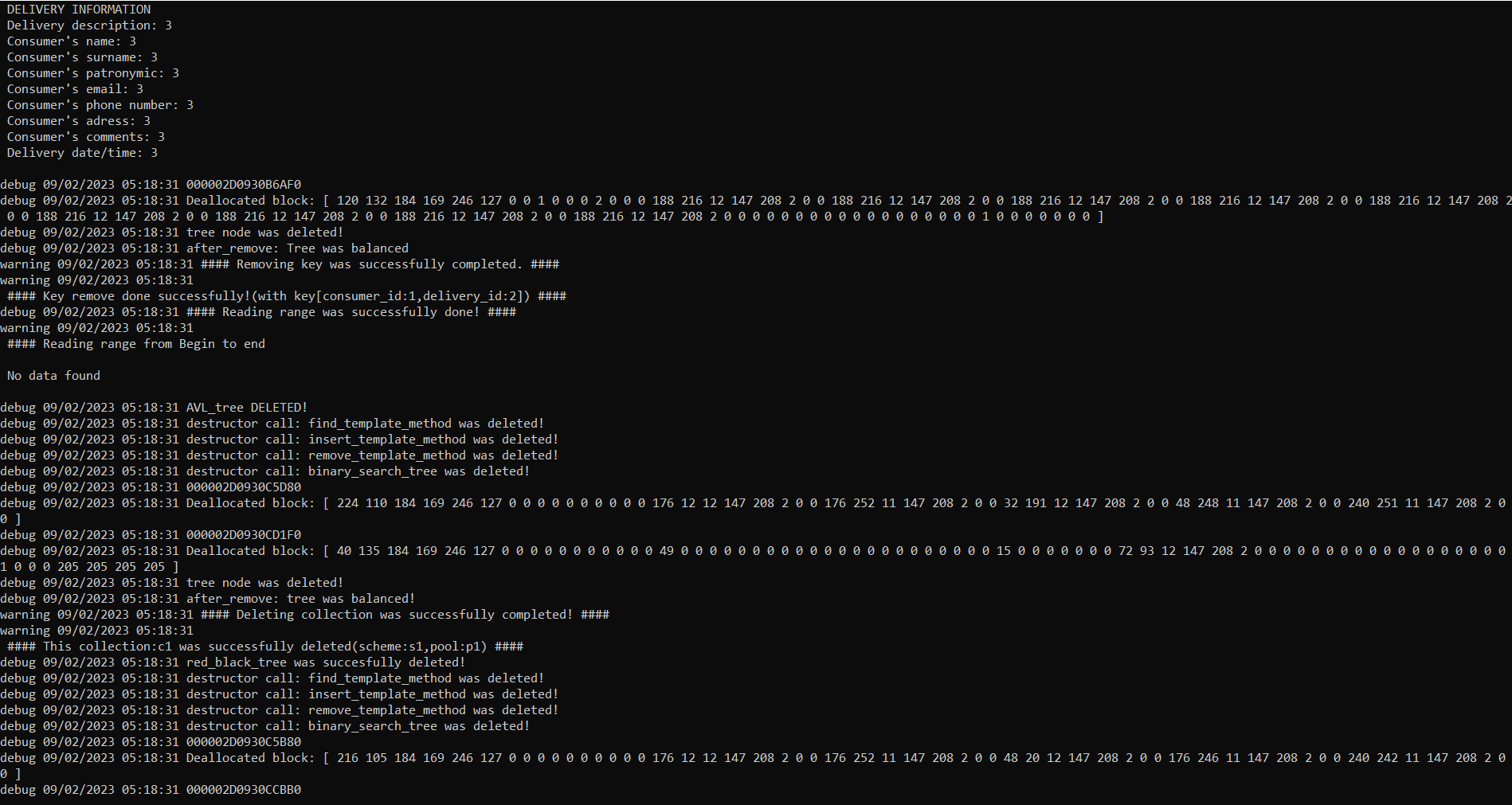


Рисунок 26

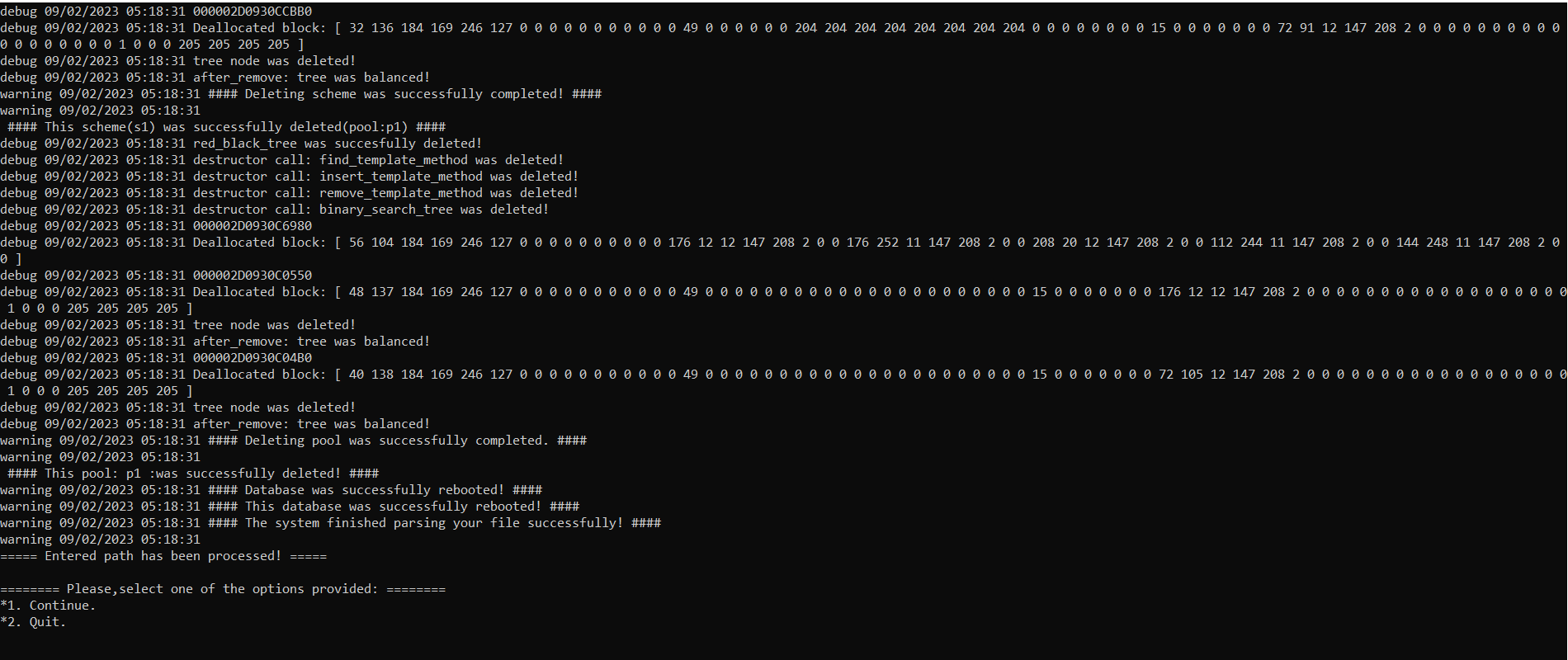


Рисунок 27

**Вывод.**

В результате данного курсового проекта мною было получено приложение, моделирующее работу базы данных. В роли хранилища реализованных структур выступает красно-чёрное дерево. Помимо этого, мною были реализованы два аллокатора, логгер, AVL-дерево, интерактивный диалог с пользователем, а также реализовал класс string\_flyweight для недопущения повторного хранения объектов строк в разных областях памяти.

**Список использованных источников.**

1. Кормен, Лейзерсон, Ривест, Штайн - Алгоритмы построение и анализ.

2. Кнут - Искусство программирования, т. 3

**Приложение.**

Основной код приложения доступен по ссылке в следующем репозитории:

https://github.com/dolgofff/CourseWork