**Содержание**

Введение……………………………………………………………..…...3

Логирование в приложении…………………………………………......5

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

Красно-чёрное дерево…………………………………………..…….....9

AVL-дерево………………………………………..………….………….16

База данных……………………………………...….…………………...25

Хранение объектов строк…………………………………………….....25

IPC: Unix message queues……………………………………………….26

Splay дерево……………………………………………………………...28

Руководство пользователя. Шаблоны реализованных команд……....30

Демонстрация функционала и работы приложения……………..........33

Вывод………………………………………………………...………......35

Список использованных источников………………………………......35

Приложение…………………………………………………………......36

**Введение.**

В рамках данного курсового проекта на языке программирования C++ необходимо реализовать 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 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;

Притом **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. Реализация класса logger.

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. Реализация класса logger\_concrete.

class logger\_concrete final : public logger

{

friend class logger\_builder\_concrete;

private:

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

private:

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

private:

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

public:

logger\_concrete(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. Реализация класса logger\_builder.

class logger\_builder

{

public:

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

virtual logger\* construct() const = 0;

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

public:

virtual ~logger\_builder() noexcept = default;

};

Листинг кода 4. Реализация класса logger\_builder\_concrete.

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

{

private:

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

public:

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

logger\* construct() const override;

logger\_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. Реализация класса memory.

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;

};

Реализация класса memory\_global\_heap доступна в **приложении 2**.

**Освобождение с дескрипторами границ.**

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

Выделение в данном аллокаторе работает так: вначале ищется блок, размер которого удовлетворяет запрос пользователя. Если необходим первый подходящий, то ищется первый блок памяти, размер которого подходит, если лучший – то ищется такой блок, размер которого является самым маленьким среди подходящих, а если худший, то наоборот, самый большой. Для того, чтобы найти такой подходящий блок, 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 просто необходимо получить указатели на следующий и предшествующий блоки и поменять их указатели, проверив есть ли они вообще в списке.

Кроме двух вышеупомянутых алгоритмов выделения памяти в реализованном приложении были использованы аллокаторы в рассортированном списке и с системой двойников, которые подробно описаны в книге Кнута Д.Э. “Искусство программирования. Том 1.” **[1]**

**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 нужно 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 вызывается дальнейшая балансировка.

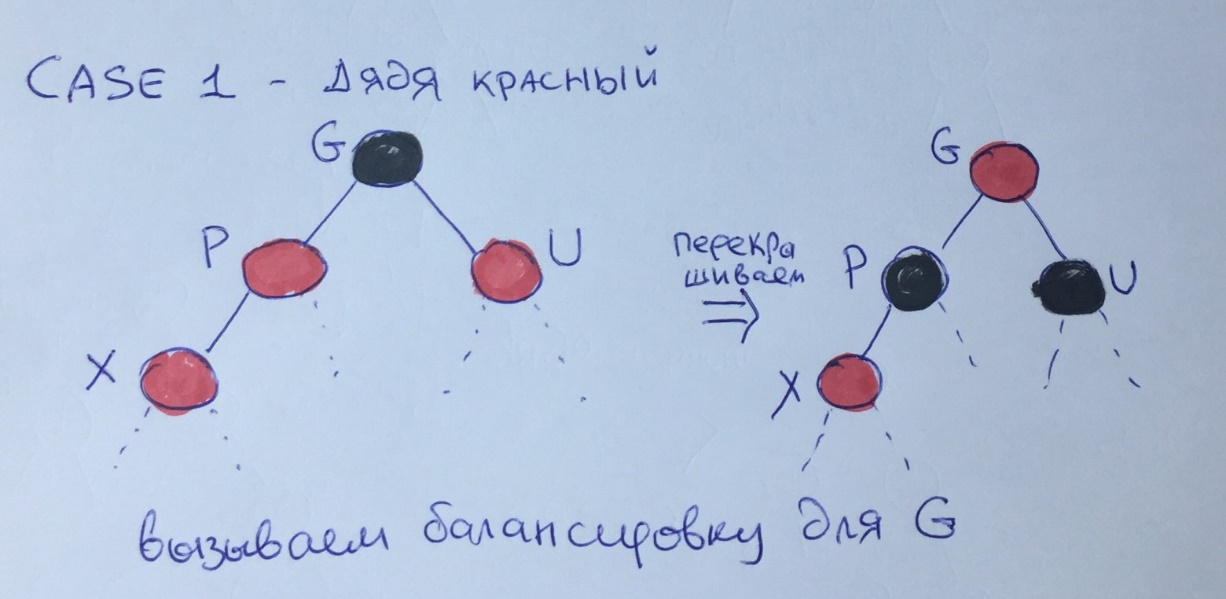


Рисунок 1. Демонстрация первого случая вставки.

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

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

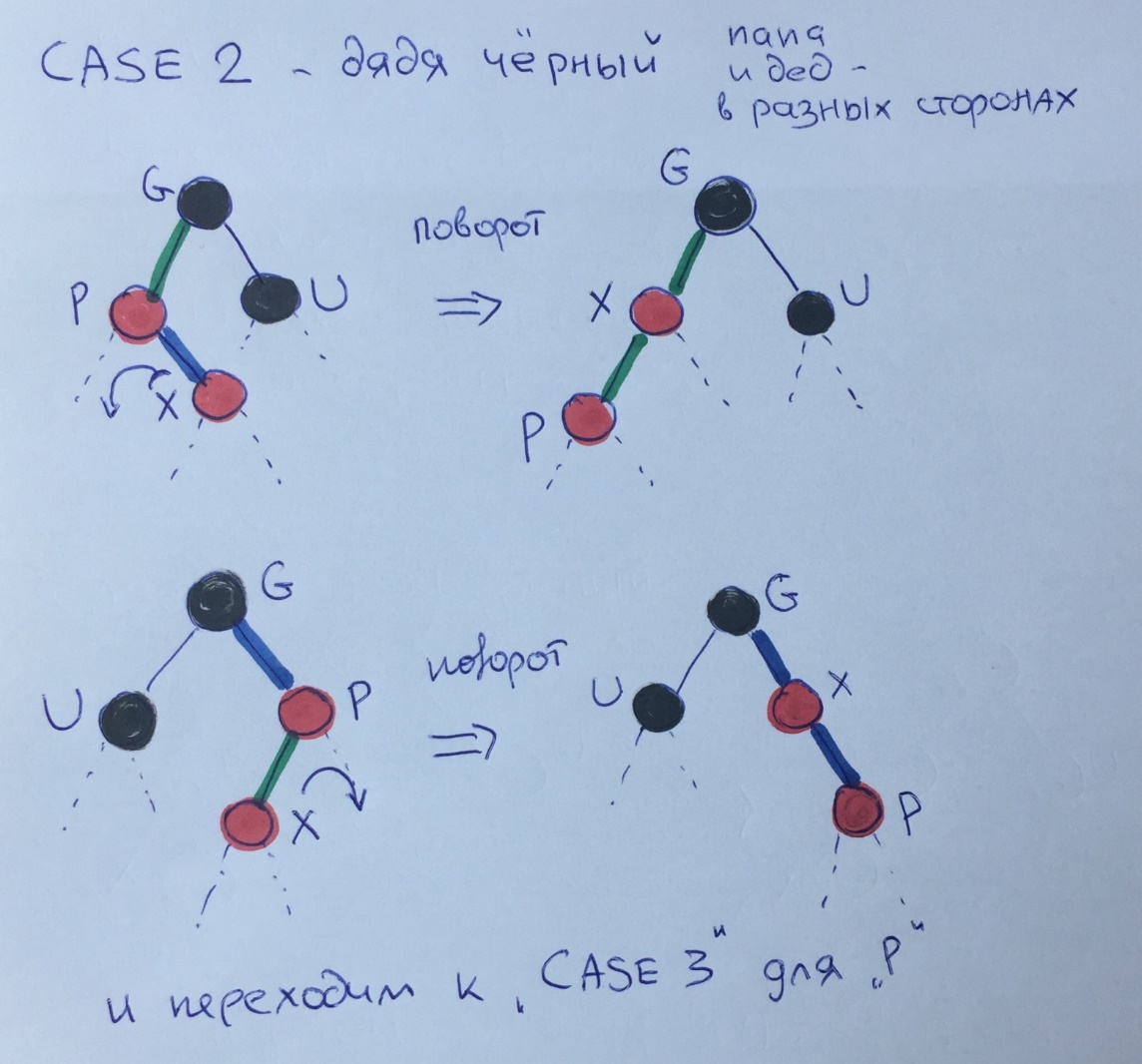


Рисунок 2. Демонстрация второго случая вставки.

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

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



Рисунок 3. Демонстрация третьего случая вставки.

**Удаление из красно-чёрного дерева.**

Для начала, необходимо рассмотреть самые базовые случаи:

1. Если рассматривается красный узел, у которого два ребёнка, то нужно всего лишь найти ближайший элемент, который меньше или больше удаляемого и поменять их местами, оставив цвет узлов прежним.
2. Если рассматривается чёрный узел, у которого два ребёнка, то необходимо сделать аналогичный обмен.
3. Если рассматривается чёрный узел 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 рассматривается красный узел без детей, то удаляется элемент, а вместо него ставится ссылка на NIL.

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

Наконец, достигнут самый сложный случай, где 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 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, возможно их перемещать, тем самым восстанавливая баланс для тех или иных элементов. В этом случае спускаем красный цвет от родителя к левому сыну. Как можно увидеть на рисунке, получилось выравнивание чёрной высоты для родителя.

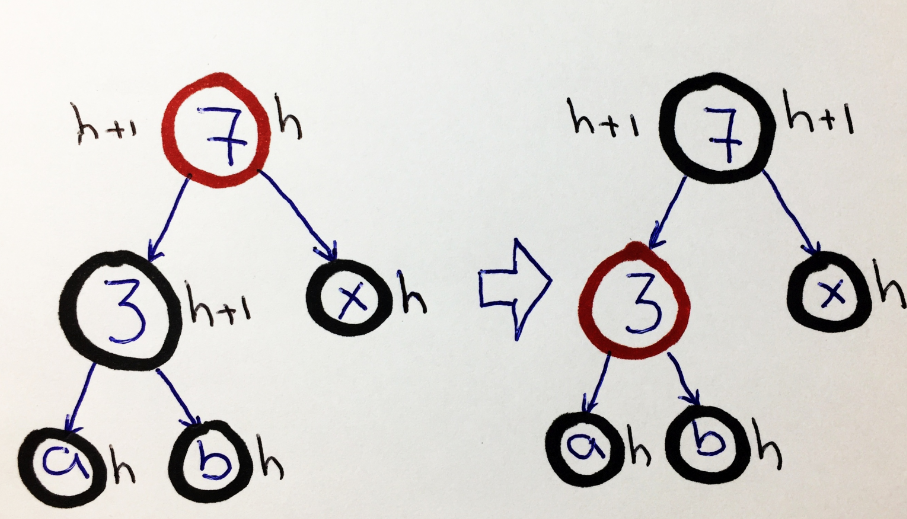


Рисунок 4. Демонстрация первого случая балансировки.

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

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

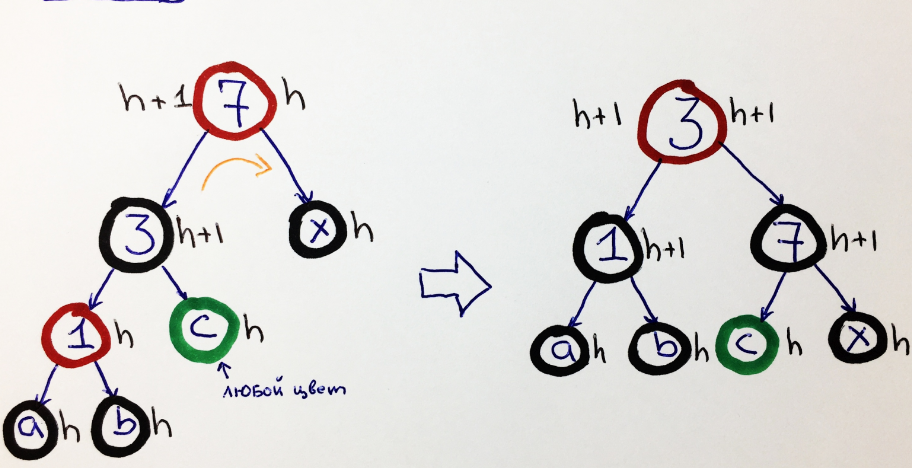


Рисунок 5. Демонстрация второго случая балансировки.

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

Так как имеются чёрные правнуки, то надо сделать внука красным и переместить в правое поддерево, выполнив малый правый поворот от дяди к отцу.

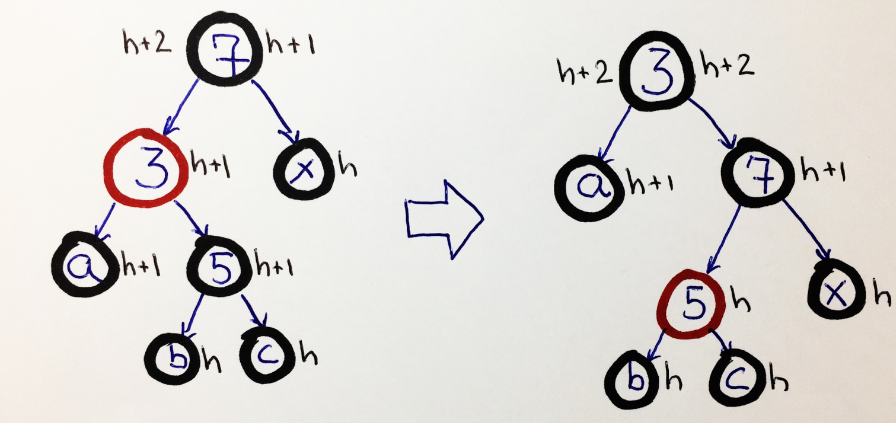


Рисунок 6. Демонстрация третьего случая балансировки.

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

Теперь 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.

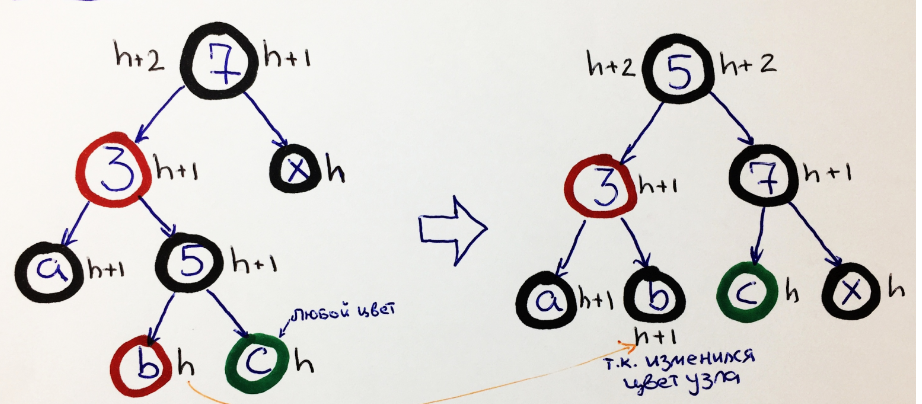


Рисунок 7. Демонстрация четвёртого случая балансировки.

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

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

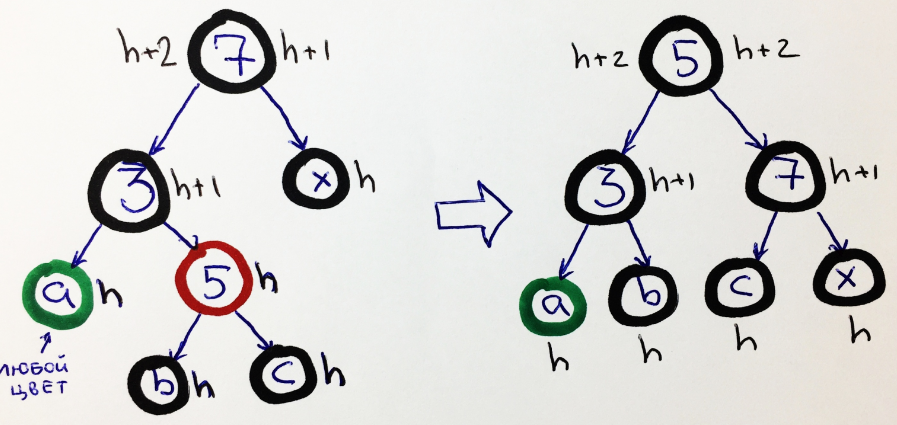


Рисунок 8. Демонстрация пятого случая балансировки.

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

Так как больше нельзя использовать красные узлы (их не осталось), то 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.

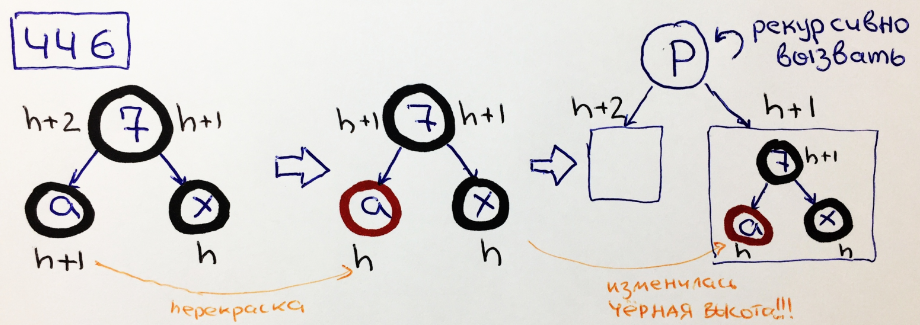


Рисунок 9. Демонстрация шестого случая балансировки.

**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 нужно обновлять высоту для каждого узла, находящегося в стеке во время его скрутки.

**Малые левый и правый повороты.**

Необходимо использовать малый левый поворот в том случае, если требуется укоротить ветку, которая перевешивает справа.

Рассмотрим алгоритм малого левого поворота на примере с рисунка.



Рисунок 10. Демонстрация исходного дерева.

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

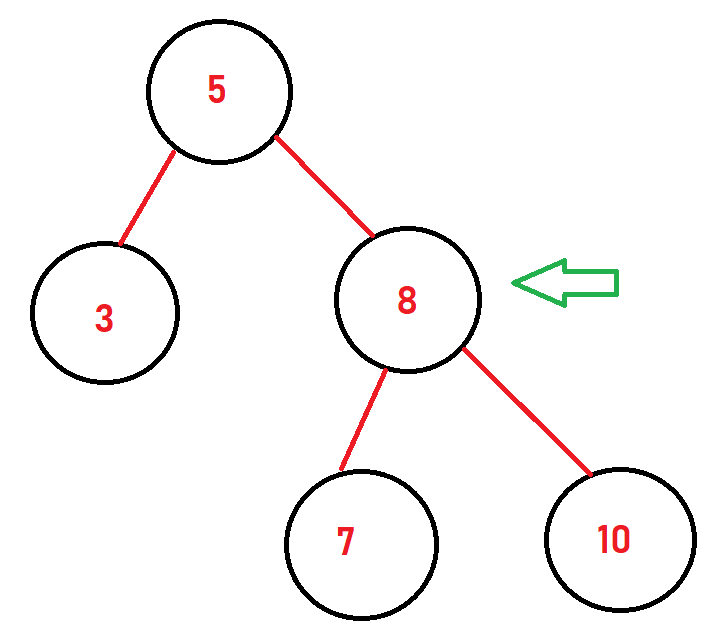


Рисунок 11. Создание указателя на правый дочерний узел.

Шаг 2: меняется у (5) указатель правого поддерева на левое поддерево дочернего узла (7). Это можно сделать, так как известно, что любое из значений правого поддерева для (5) будет не меньше него.

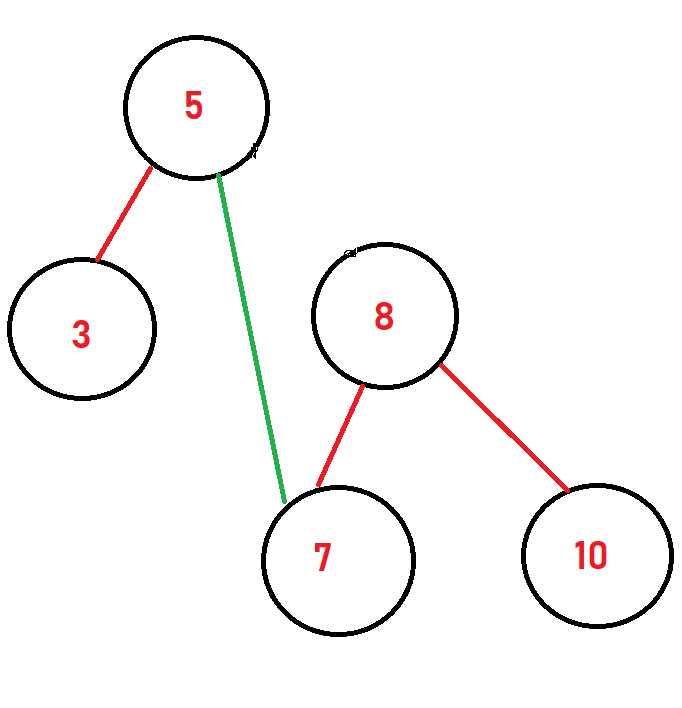


Рисунок 12. Смена указателя у старого корня.

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

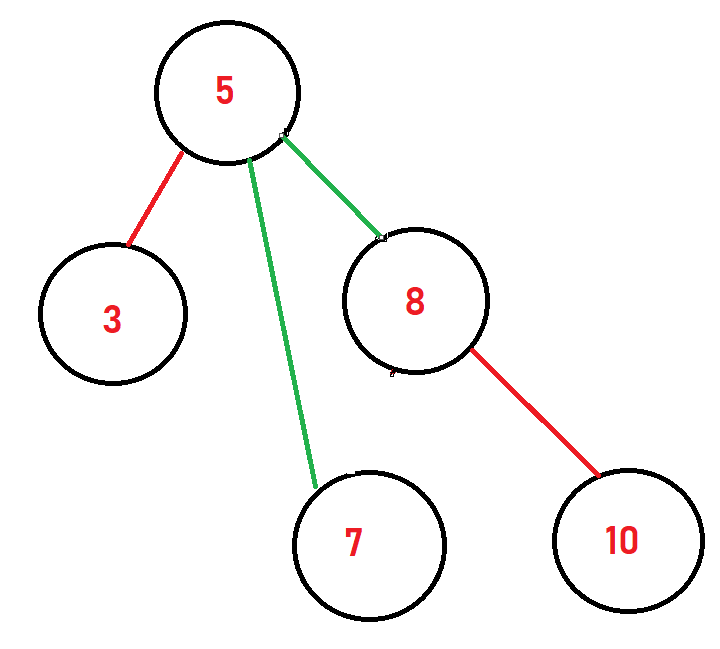


Рисунок 13. Смена указателя у нового корня.

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

Поворот завершён.

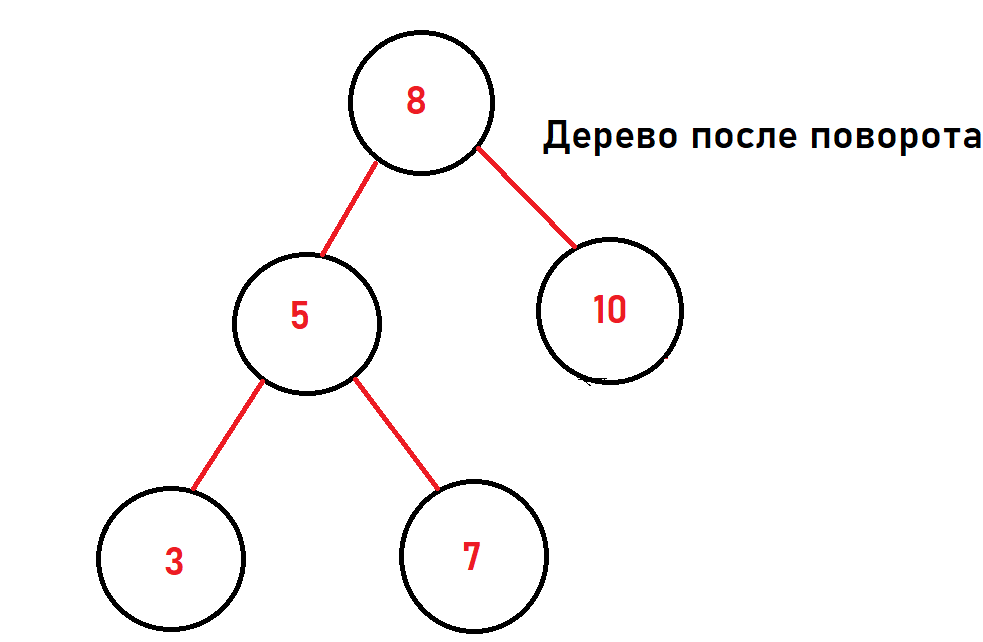


Рисунок 14. Демонстрация дерева после поворота.

В свою очередь, необходимо использовать малый правый поворот в том случае, если требуется 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. Совершается проход вниз по дереву, на каждом шагу сравнивается значение нового узла с текущими, и при нахождении подходящего места производится вставка.

Однако, так как в AVL-дереве высота играет большую роль, после вставки вызывается балансировка. Требуется сделать так, чтобы разница между высотами для каждого дочернего узла всегда была меньше или равна 1 по модулю.

**Дерево перегружено влево.**

Если рассматривается дерево, которое перегружено влево, то тут может возникнуть два случая.

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

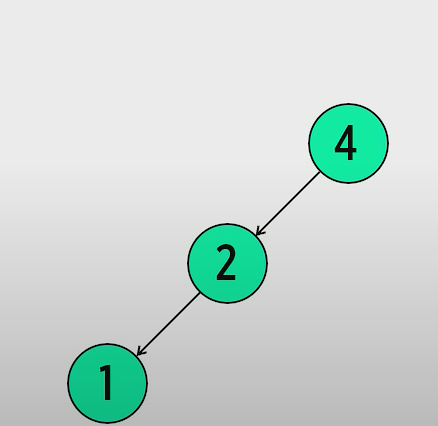
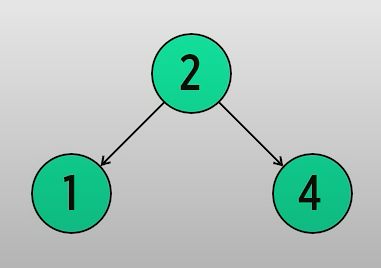
 

Рисунок 15. Дерево до поворота. Рисунок 16. Дерево после поворота.

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 нужно сделать малый правый поворот. Однако сделав его, можно увидеть, что теперь дерево стало перегружено вправо.



Рисунок 17. Дерево до поворота.

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

Для начала, выполняется малый поворот влево от (1), и выходит первый случай.

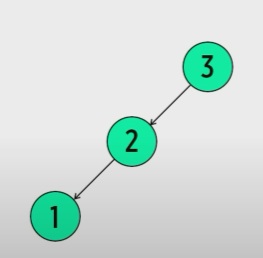


Рисунок 18. Дерево после первого поворота.

Затем, необходимо сделать малый правый поворот, в результате чего приходим к балансу.

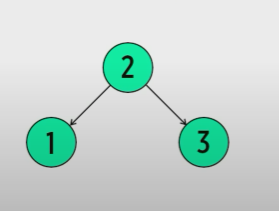


Рисунок 19. Дерево после второго поворота.

**Дерево перегружено вправо**.

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

Случай 1:

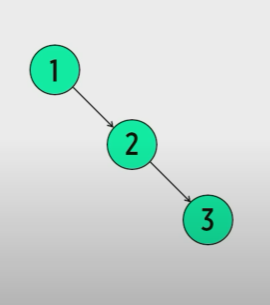
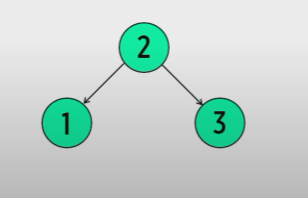
 

Рисунок 20. Дерево до поворота. Рисунок 21. Дерево после поворота.

Случай 2:

Требуется право-левый поворот.

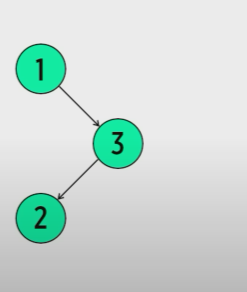


Рисунок 22. Дерево до поворота.

Необходимо сделать малый поворот вправо от (3).

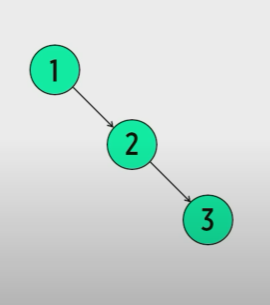


Рисунок 23. Дерево после первого поворота.

И затем малый поворот влево от (2).

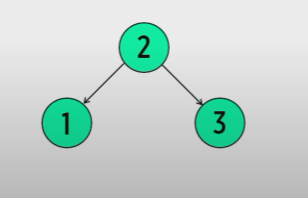


Рисунок 24. Дерево после второго поворота

**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 будет обновляться высота узлов и в случае необходимости их балансировка по раннее разобранным алгоритмам. Стоит отметить, что это будет выполнено не только для удаляемого узла, но и для того узла, на который удаляемый заменяется, в случае с двумя детьми, потому что его удаление может снова привести к изменению высот родительских узлов, поэтому происходит рекурсивный проход по каждому такому узлу вплоть до удаляемого, обновляются 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 текущей программы, они являются красно-чёрным деревом, коллекция может являться Splay/AVL-деревом). В дальнейшем будет описано, какие именно операции можно совершать относительно данной структуры.

Листинг кода 7. Объявление базы данных.

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 объявлено два метода:

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

void file\_parse\_options(const string& 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:

string& get\_flyweight(const string& str), который получает на вход строку: если такой строки не было, то он вставляет её в коллекцию, а если была, то возвращает её.

void check\_flyweight(const string& str), который помогает в удалении строк из коллекции и ведёт учёт количества той или иной строки.

Реализация класса string\_flyweight доступна в **приложении 3**.

**IPC: Unix message queues.**

Unix message queues – это один из механизмов межпроцессного взаимодействия. Он предоставляет способ передачи данных между процессами, которые могут быть расположены как на одной машине, так и на разных. Очереди сообщений позволяют процессам обмениваться данными в виде отдельных блоков (сообщений).

Для очередей сообщений ведётся учёт ссылок. Очередь, помеченная для удаления, уничтожается только после того, как будет закрыта всеми процессами, которые её используют. Такие сообщения имеют приоритет, обеспечивающий строгий порядок передачи и получения данных.

Каждая очередь сообщений имеет определённый набор атрибутов. Часть из них могут быть установлены во время создания или открытия очереди с помощью вызова mq\_open(). Для просмотра и изменения этих атрибутов предусмотрены две функции: mq\_getattr() и mq\_setattr().

Функция mq\_open() создаёт новую или открывает существующую очередь сообщений:

mqd\_t mq\_open(const char\*name, int oflag, mode\_t mode, struct mq\_attr)

Аргумент name идентифицирует очередь сообщений, аргумент oflag представляет собой битовую маску, управляющую различными аспектами работы функции.

Необходимо использовать флаг O\_CREAT, который создаёт очередь, и флаг O\_RDWR, определяющий тип доступа к очереди (открывается для чтения и записи).

Аргумент mode представляет собой битовую маску, которая определяет права доступа к новой очереди сообщений.

Аргумент attr представляет собой структуру mq\_attr, определяющую атрибуты новой очереди сообщений.

Функция mq\_close() закрывает дескриптор очереди сообщений mqdes.

int mq\_close(mqd\_t mqdes)

Закрывая очередь сообщений, мы её не удаляем, для этого предусмотрена функция mq\_unlink().

int mq\_unlink(const char\* name)

Дескриптор очереди сообщений принадлежит конкретному процессу и ссылается на запись в общесистемной таблице открытых дескрипторов этого типа. Данная запись в свою очередь указывает на объект очереди.

Для того, чтобы послать сообщение на сервер, необходимо использовать следующий метод:

void send\_message(string data)

Для того, чтобы прочитать ответ с сервера, необходимо использовать метод:

string read\_message()

Для того, чтобы принять сообщение, отправленное клиентом, необходимо использовать метод:

void listen(function<void(string)> cb);

Более подробное описание данного алгоритма приведено в книге Керриска М. “Linux API. Исчерпывающее руководство”. **[4]**

**Splay дерево.**

Splay дерево представляет из себя модификацию бинарного дерева поиска. Структура этого дерева позволяет находить быстрее те данные, к которым обращались недавно.

Так как узел этого дерева ничем не отличается от привычного бинарного узла, необходимо рассмотреть лишь такие процедуры как splay и merge.

Операция поиска в splay дереве выполняется также, как и в бинарном дереве поиска. Однако, после неё необходимо вызвать метод splay.

Таким образом, если поиск завершился успешно, то найденный узел поднимается вверх на позицию корня. В ином случае позицию корня занимает последний узел, к которому был получен доступ.

Процедура splay предполагает реализацию нескольких различных случаев, получившихся в результате осуществления доступа к некоторому узлу.

1. Если узел является корневым, то нам просто необходимо вернуть текущий корень.
2. Случай Zig: Узел является прямым потомком корня.

Рассматриваемый узел является либо левым потомком корня, либо правым потомком своего родителя. В зависимости от ситуации, необходимо выполнить малый левый или малый правый поворот.

1. Узел имеет и родителя, и прародителя.
   1. Случаи Zig-zig и Zag-zag:

Узел является левым потомком родительского элемента, и родитель является левым потомком прародителя или же узел является правым потомком своего родителя, который тоже является правым потомком своего прародителя. В этих случаях необходимо осуществить большой левый или большой правый поворот.

* 1. Случаи Zig-zag и Zag-zig:

Узел является левым потомком по отношению к родительскому элементу, а родитель правым потомком прародителя или же узел является правым потомком своего родительского элемента, а родитель является левым потомком прародителя. Ситуацию помогают разрешить лево-правый и право-левый повороты, рассмотренные ранее в контексте AVL дерева.

Таким образом, если необходимо добавить новый элемент в дерево, то нужно произвести вставку, а затем вызвать метод splay.

В случае удаления, напротив, сначала нужно вызвать splay, а затем уже произвести удаление.

Связано это с тем, что операция удаления требует нахождения удаляемого элемента на позиции корня дерева.

После того, как элемент оказался на вершине дерева, требуется вызвать процедуру merge, которая объединяет левое и правое поддерево удаляемого узла. Пускай X – левое поддерево, а Y – правое поддерево.

Для слияния X и Y, в которых все ключи X меньше ключей Y, необходимо вызвать splay для максимального элемента X, тогда у корня X не будет правого ребёнка, на позицию которого нужно поставить поддерево Y.

**Руководство пользователя. Шаблоны реализованных команд.**

Выбирая ту, или иную опцию, пользователь получает шаблон, в соответствии с которым необходимо ввести эту опцию.

nameof\_pool – строка, название пула.

nameof\_scheme – строка, название схемы.

nameof\_collection – строка, название коллекции.

consumer\_id, delivery\_id – два числа, формирующие уникальный ключ (id пользователя и id доставки).

delivery\_description – строка, описание доставки.

consumer\_name, consumer\_surname, consumer\_patronymic – строки, ФИО пользователя.

consumer\_email – строка, электронная почта пользователя.

consumer\_phone\_number – строка, номер телефона пользователя.

consumer\_address – строка, адрес пользователя.

consumer\_comments – строка, комментарии пользователя.

delivery\_date\_time – строка, дата/время доставки.

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\_address] [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\_address] [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 серверу пользователем подаётся некоторый текстовый файл, содержащий все вышеперечисленные команды.

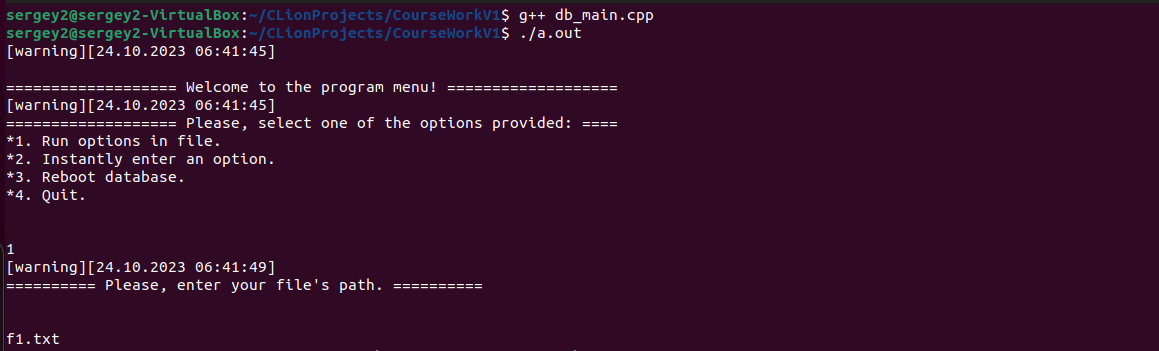


Рисунок 25. Демонстрация подачи пользователем в клиент текстового файла ft.txt.

Результат работы.

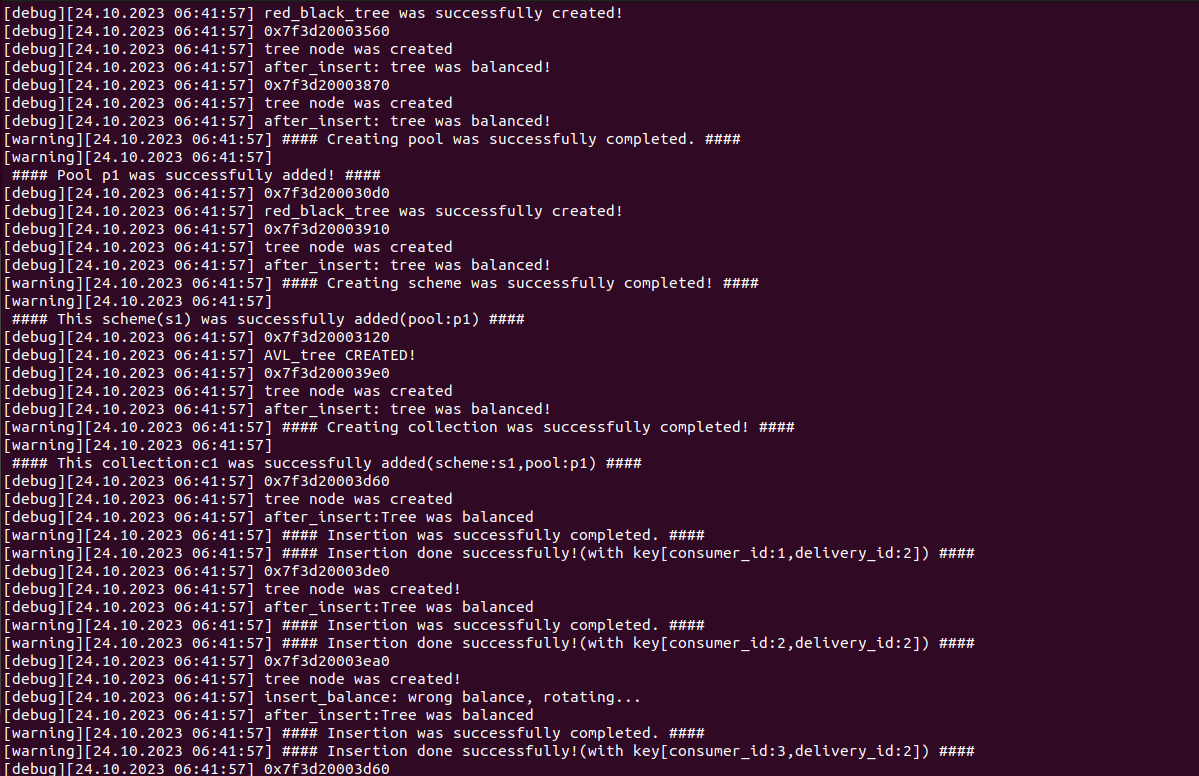


Рисунок 26. Демонстрация ответа сервера с поданным на вход файлом ft.txt.

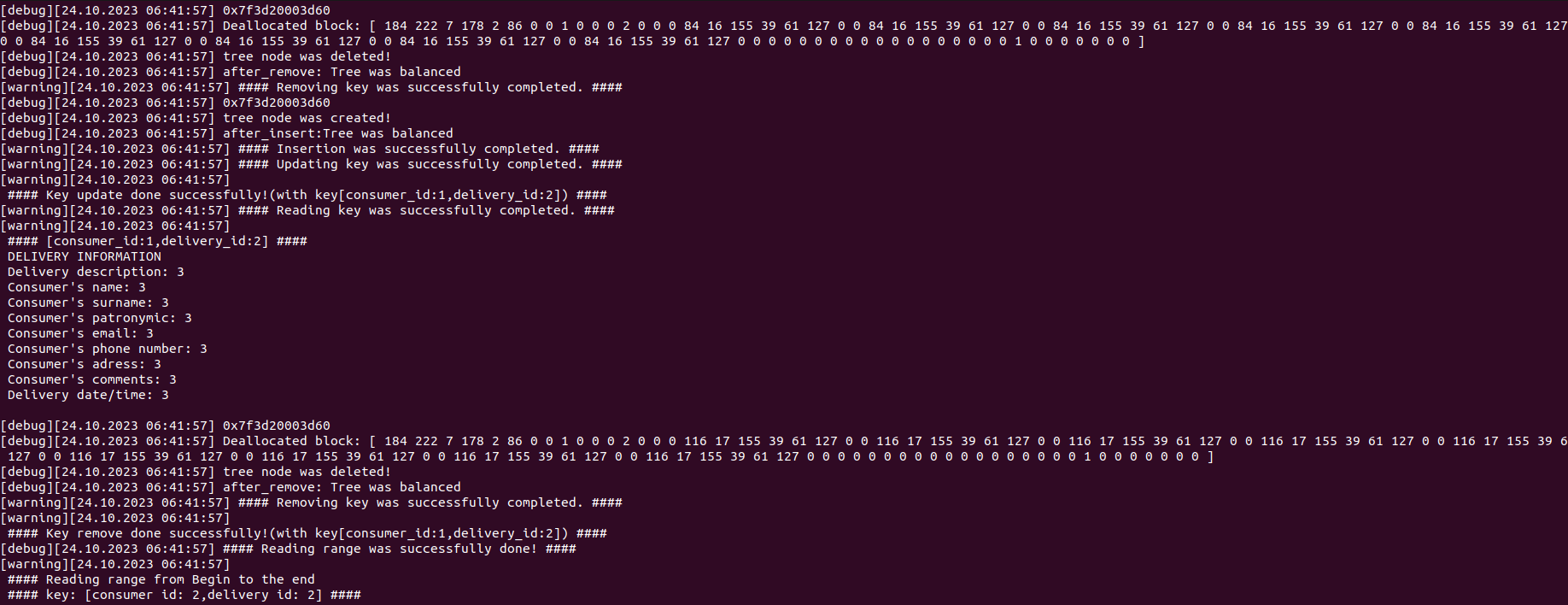


Рисунок 27. Демонстрация ответа сервера с поданным на вход файлом ft.txt.

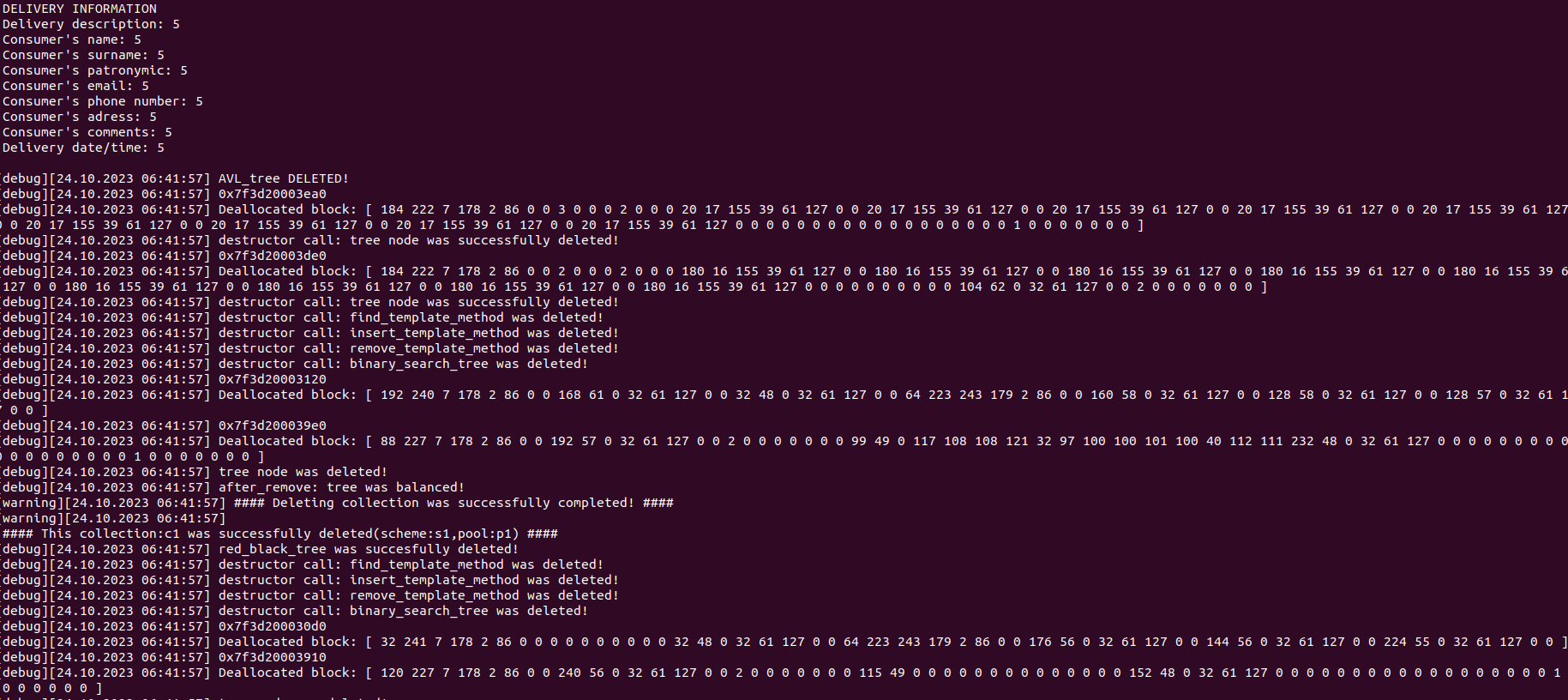


Рисунок 28. Демонстрация ответа сервера с поданным на вход файлом ft.txt.

**Вывод.**

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

**Список использованных источников.**

1. Кнут Д.Э. “Искусство программирования. Том 1”.

2. Электронный ресурс “Хабр”. Статья “Удаление узлов из красно-чёрного дерева”. <https://habr.com/ru/companies/otus/articles/521034>

3. Электронный ресурс “Хабр”. Статья “Балансировка красно-чёрных деревьев – Три случая”. <https://habr.com/ru/companies/otus/articles/472040>

4. Керриск М. “Linux API. Исчерпывающее руководство”.

**Приложение.**

1. Основной код приложения доступен по ссылке в следующем репозитории:

<https://github.com/dolgofff/CourseWork>

В случае, если у вас не вышло получить доступ к курсовому проекту по предоставленной ссылке, прошу обратиться по следующему адресу электронной почты: [michaltriedto@gmail.com](mailto:michaltriedto@gmail.com).

1. Реализация класса memory\_global\_heap.

Листинг кода 7. Реализация класса memory\_global\_heap.

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);

}

};

3. Реализация класса string\_flyweight.

Листинг кода 8. Реализация класса string\_flyweight.

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;

}}};