

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Московский Авиационный Институт
(Национальный исследовательский университет)

Институт №8
«Компьютерные науки и прикладная математика»
Кафедра 806
«Вычислительная математика и программирование»

Курсовой проект по дисциплине «Фундаментальные алгоритмы»
Тема: «Разработка алгоритмов системы хранения и управления данными
на основе динамических структур данных»

Студент: Якухин Данила Олегович

Группа: М8О-213Б-21

Преподаватель: Ирбитский И.С.

Оценка: ____

Дата: _____

Москва, 2023

Содержание

Введение	4
1. Управление памятью	6
1.1 Memory	6
1.2 Аллокатор	6
1.2.1 Аллокатор с освобождением в рассортированном списке.....	7
1.2.2 Аллокатор с освобождением с дескрипторами границ	8
1.2.3 Аллокатор с алгоритмом системы двойников	9
2. Бинарное дерево поиска.....	9
2.1 Ассоциативный контейнер.....	10
2.2 Класс Binary_search_tree	11
3. Красно-чёрное дерево	12
3.1 Вставка	13
3.2 Удаление.....	15
4. АВЛ дерево	17
4.1 Вставка	18
4.2 Удаление.....	18
4.3 Балансировка	18
5. Косое дерево	20
5.1 Вставка	20
5.2 Удаление.....	20
5.4 Операция Splay	20
5.5 Операция Merge.....	22
6. Разработка и реализация приложения, управляющего хранилищем	22
6.1 Состав хранилища данных	22
6.2 Реализация механизма, позволяющего выполнять запросы к данным в рамках коллекции данных на заданный момент времени	25
6.3 Реализация механизма поиска по различным порядкам отношений на пространстве данных.....	27
6.4 Эффективное хранение строк.....	28
7. Руководство пользователя	29

8. Вывод	34
9. Список использованных источников	34
10. Приложение	34

Введение

Задание курсового проекта:

На языке программирования C++ (стандарт C++14 и выше) реализуйте приложение, позволяющее выполнять операции над коллекциями данных заданных типов (типы обслуживаемых объектов данных определяются вариантом) и контекстами их хранения (коллекциями данных). Коллекция данных описывается набором строковых параметров (набор параметров однозначно идентифицирует коллекцию данных):

- название пула схем данных, хранящего схемы данных;
- название схемы данных, хранящей коллекции данных;
- название коллекции данных.

Коллекция данных представляет собой ассоциативный контейнер (конкретная реализация определяется вариантом), в котором каждый объект данных соответствует некоторому уникальному ключу. Для ассоциативного контейнера необходимо вынести интерфейсную часть (в виде абстрактного класса C++) и реализовать этот интерфейс. Взаимодействие с коллекцией объектов происходит посредством выполнения одной из операций над ней:

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

Во время работы приложения возможно выполнение также следующих операций:

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

Поток команд, выполняемых в рамках работы приложения, поступает из файла, путь к которому подаётся в качестве аргумента командной строки. Формат команд в файле определите самостоятельно.

Дополнительные задания, реализованные в этой курсовой работе:

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

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

Вариант курсовой работы 19:

Данные о прохождении сессии в университете (**id сессии**, **id студента**, ФИО студента (раздельные поля)), **формат отчётности (курсовая работа/зачёт/экзамен)**, **название предмета**, дата проведения зачёта/экзамена, время начала проведения зачёта/экзамена, полученная оценка (в диапазоне 2..5 для экзамена или курсовой работы, 0..1 для зачёта), ФИО преподавателя (раздельные поля))

1. Управление памятью

1.1 Memory

Листинг 1. Класс `memory`

```
class memory
{
public:

    enum class ALLOCATION_METHOD
    {
        FIRST_SUITABLE,
        BEST_SUITABLE,
        WORSE_SUITABLE
    };

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

    virtual void deallocate(void*) const = 0;

    void *operator+=(size_t const &size) const;

    void operator-=(void*) const;

    memory(memory const&) = delete;

    void operator=(memory const&) = delete;

    memory() = default;

    virtual ~memory() = default;

};
```

Класс *memory_holder* представляет собой интерфейс для использования объекта класса *memory*. Содержит методы для обращения к памяти *guard_allocate* и *guard_deallocate*, которые получают объект класса *memory*, вызывая виртуальный метод *get_memory*, и, при наличии объекта, вызывают его метод *allocate*, иначе возвращают указатель на выделенный из глобальной кучи участок памяти.

1.2 Аллокатор

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.

В каждом аллокаторе доступно выделение и освобождение памяти, используются различные алгоритмы:

- метод освобождения в рассортированном списке
- метод освобождения с дескрипторами границ
- метод системы двойников

Способ выделения памяти 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

1.2.1 Аллокатор с освобождением в рассортированном списке

Класс *allocator_list*. Наследник класса *memory*. Представляет собой аллокатор с освобождением блоков в рассортированном списке.

Аллокатор с методом выделения в рассортированном списке - это специализированная структура данных, которая позволяет эффективно выделять и освобождать память для объектов, хранящихся в упорядоченном (рассортированном) списке. Он оптимизирован для ситуаций, когда объекты добавляются и удаляются из списка, и их память нужно выделять и освобождать так, чтобы сохранить порядок элементов.

В блоке памяти, управляемым аллокатором, содержится следующая информация:

- Размер памяти
- Указатель на логгер
- Внешний аллокатор

- Указатель на первый свободный блок

В свободных блоках памяти содержится следующая информация:

- Размер блока, включая служебную часть
- Указатель на следующий свободный блок

Все свободные блоки аллокатора связаны последовательно в виде односвязного списка.

В занятых блоках памяти содержится только информация о размер блока, включая служебную часть.

Вызов конструктора *allocator_list* состоит из 4 параметров: размера области памяти, которую необходимо выделить, указателя на объект класса *memory* - родительского для этого аллокатора, указателя на объект класса *logger* этого аллокатора и *method_allocation* - одного из 3 значений метода выделения памяти *ALLOCATION_METHOD*.

1.2.2 Аллокатор с освобождением с дескрипторами границ

Класс *allocator_descriptor*. Наследник класса *memory*. Представляет собой аллокатор с освобождением с дескрипторами границ.

Аллокатор с освобождением с дескрипторами границ - это механизм управления памятью, который выделяет и освобождает блоки памяти с использованием специальных дескрипторов, содержащих информацию о размере и структуре каждого блока. Эти дескрипторы границ позволяют аллокатору эффективно и безопасно определять границы блоков, что способствует избеганию фрагментации памяти и обеспечивает безопасность работы с памятью. Этот подход особенно полезен в случаях, когда необходимо выделять и освобождать память для объектов переменного размера, а также для управления многими блоками памяти с высокой эффективностью.

В свободных блоках памяти не содержится никакой информации.

В занятых блоках памяти содержится следующая информация:

- Размер блока, включая служебную часть
- Указатель на следующий и предыдущий блок

Вызов конструктора *allocator_descriptor* состоит из 4 параметров: размера области памяти, которую необходимо выделить, указателя на объект класса *memory*

- родительского для этого аллокатора, указателя на объект класса `logger` этого аллокатора и `method_allocation` - одного из 3 значений метода выделения памяти `ALLOCATION_METHOD`.

1.2.3 Аллокатор с алгоритмом системы двойников

Класс `allocator_buddies`. Наследник класса `memory`. Представляет собой аллокатор с алгоритмом системы двойников.

Аллокатор с освобождением с алгоритмом системы двойников - это метод управления памятью, который использует подход двойников для эффективного выделения и освобождения памяти. В этом методе блоки памяти разделяются на две части при выделении памяти и объединяются обратно при освобождении, что помогает минимизировать фрагментацию памяти. Этот аллокатор позволяет эффективно использовать память, особенно в программах, где происходит множество операций выделения и освобождения памяти разного размера, и где важна оптимизация производительности и эффективного использования ресурсов.

Вызов конструктора `allocator_buddies` состоит из 3 параметров: размера области памяти, которую необходимо выделить, указателя на объект класса `memory` - родительского для этого аллокатора, указателя на объект класса `logger` этого аллокатора.

2. Бинарное дерево 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 NULL) называются листьями.

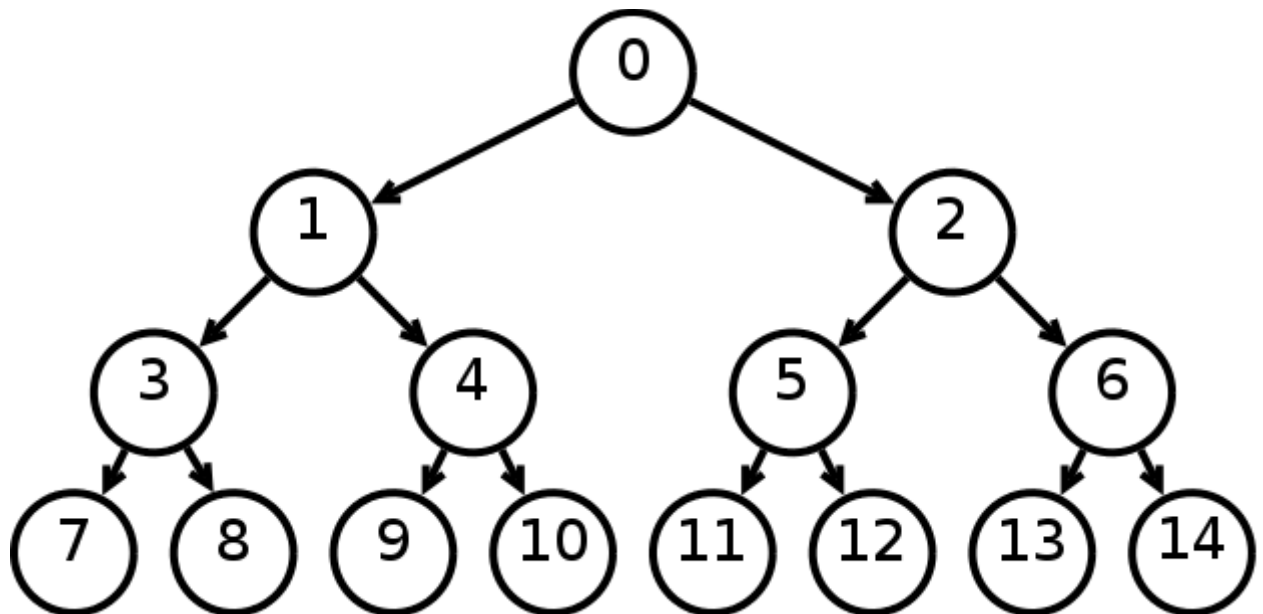


Рис. 1. Пример бинарного дерева поиска.

Бинарное дерево поиска обладает следующим свойством: 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 реализации важно обеспечить сбалансированность бинарного дерева поиска, так как это играет важную роль в эффективности операций вставки, удаления и поиска элементов.

2.1 Ассоциативный контейнер

Листинг 2. Класс `associative_container`

```
template<
    typename tkey,
    typename tvalue>
class associative_container
{
public:

    struct key_value_struct
    {
        tkey _key;
        tvalue _value;
    };

    virtual bool find(key_value_struct *target_key_and_result_value) = 0;

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

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

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

    tvalue get(tkey const &key);

    virtual std::vector<tvalue> get_range(
        tkey const &lower_bound,
        tkey const &upper_bound) const = 0;

    bool in(tkey const &key);

    bool operator[](key_value_struct *target_key_and_result_value);
```

```
void operator+=(key_value_struct pair);

tvalue operator-=(key_value_struct pair);

tvalue operator-=(tkey const &key);

virtual ~associative_container() = default;
};
```

Файл "associative_container.h" содержит определение абстрактного класса *associative_container*, который представляет абстрактный ассоциативный контейнер. Он используется для хранения данных в форме пар "ключ-значение", где ключи уникальны и используются для доступа к соответствующим значениям.

Этот абстрактный класс определяет общий интерфейс для операций, которые можно выполнять с ассоциативным контейнером, такими как поиск, вставка, удаление, обновление и другие. Он также предоставляет методы для получения значений по ключу и для проверки наличия ключа в контейнере.

2.2 Класс *Binary_search_tree*

Класс *binary_search_tree* наследуется от абстрактного класса *associative_container*, класса *memory_holder* и *logger_holder*. Внутри реализации бинарного дерева поиска будут использоваться и реализованы методы, определенные в *associative_container*. А классы *memory_holder* и *logger_holder* представляют собой удобную оболочку для работы с логгером и аллокатором. Они используются для выделения памяти в методах бинарного дерева поиска и логирования информации о действиях в файл.

Все деревья, используемые в данной программе, представляют собой разновидности сбалансированных деревьев аналогичные бинарному дереву поиска, но со своими отличительными свойствами и дополнительными методами. Поэтому в классе *binary_search_tree* реализован с использованием паттерна "Шаблонный метод" (Template Method). Смысл этого паттерна заключается в определении общего алгоритма поведения в базовом классе, который затем переопределяется в подклассах. В данном случае, шаблонный метод применяется к операциям вставки, удаления и поиска элементов в дереве. Каждая из этих операций разделяется на два этапа:

1. Основная операция: Этот этап выполняется одинаково для всех представленных деревьев и реализуется в базовом классе *binary_search_tree*. Он представляет собой основной алгоритм выполнения операции.
2. Последующие действия: После выполнения основной операции, подклассы могут определить дополнительные действия, которые необходимо выполнить, такие как балансировка дерева. Эти дополнительные действия могут различаться в зависимости от свойств и реализации конкретного дерева.

Таким образом, в классе *binary_search_tree* используется 3 внутренних шаблонных класса:

1. *Bin_find_template_method* реализует поиск элемента по ключу в дереве. Дополнительно имеет метод *after_find_concrete*, который впоследствии будет определяться и использоваться в Splay дереве.
2. *Bin_insertion_template_method* реализует вставку элемента в дерево по ключу. Дополнительно имеет метод *after_insert_concrete*, который впоследствии будет определяться и использоваться в Splay, AVL и Red-Black дереве.
3. *Bin_removing_template_method* реализует удаление элемента в дерево по ключу. Дополнительно имеет метод *after_remove_concrete*, который впоследствии будет определяться и использоваться в Splay, AVL и Red-Black дереве.

3. Красно-чёрное дерево

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов (чёрная высота).
5. Чёрный узел может иметь чёрного родителя

Структура узла красно-черного дерева:

Листинг 3. Структура узла красно-черного дерева.

```
protected:

    enum class color_node
    {
        RED,
        BLACK
    };

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

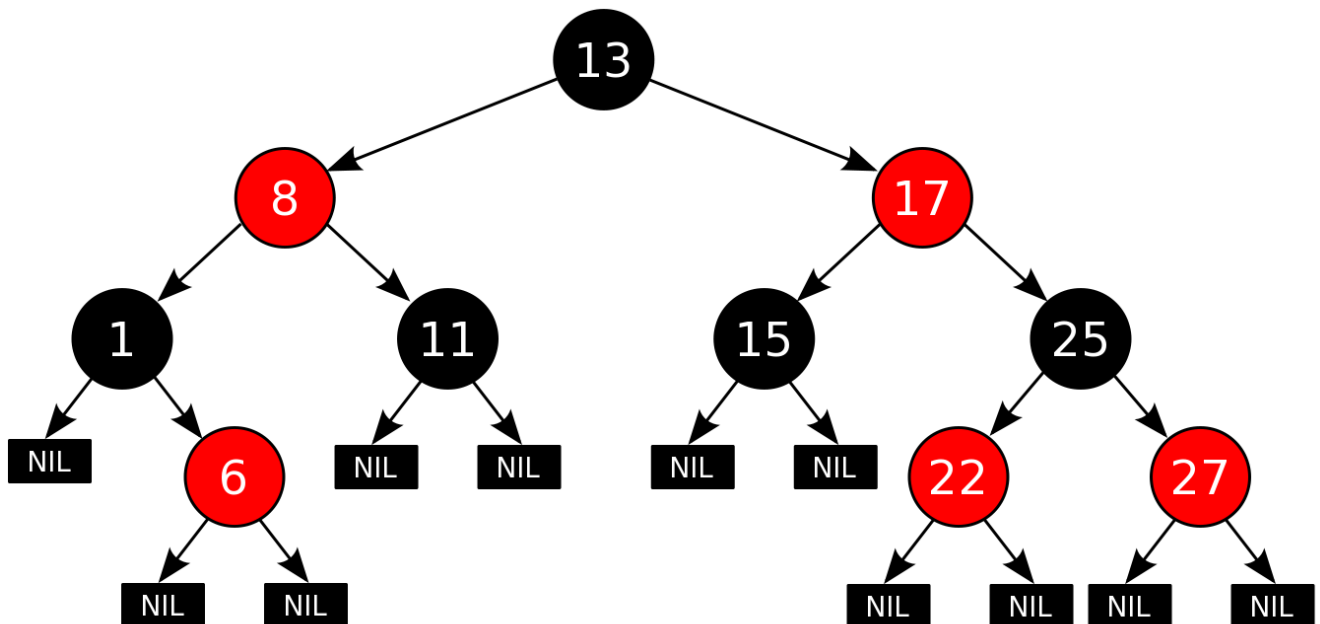


Рис. 2. Пример красно-черного дерева.

3.1 Вставка

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет null (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку цветов и черной высоты. Если отец нового элемента черный, то никакое из свойств дерева не

нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

1. "Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2.

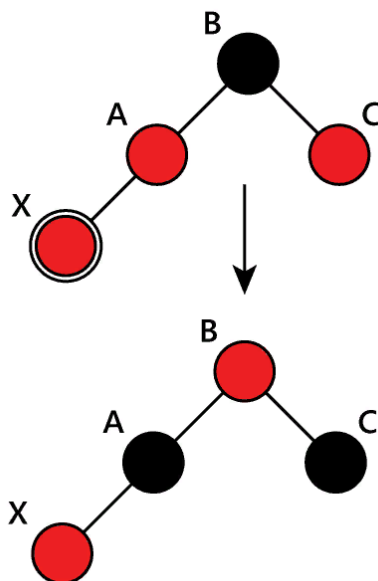


Рис. 3. Случай вставки 1.

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. Таким образом, свойство 3 и постоянство черной высоты сохраняются.

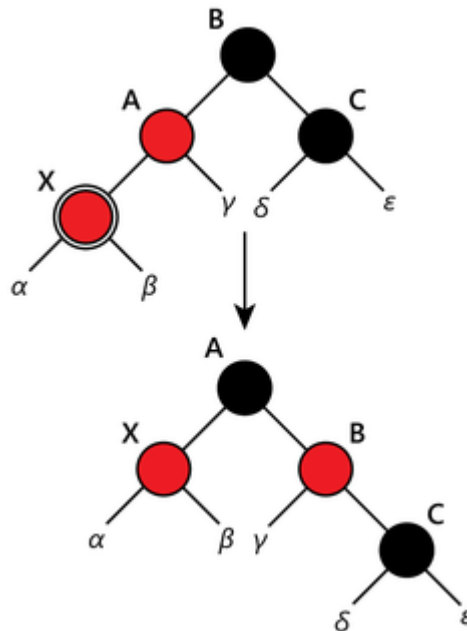


Рис 4. Случай вставки 2.

3.2 Удаление

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

- Если у вершины нет детей, то изменяем указатель на неё у родителя на null
- Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
- Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

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

- Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов,

сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

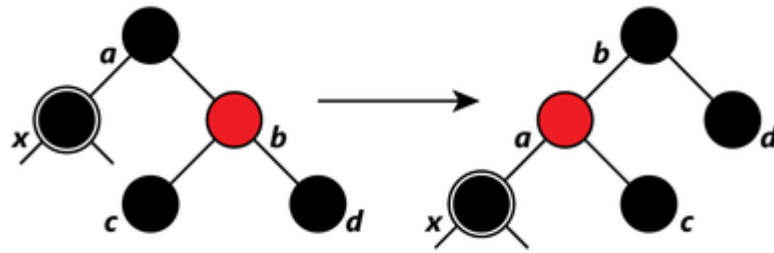


Рис 5. Случай удаления 1.

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

1. Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b , но добавит один к числу чёрных узлов на путях, проходящих через x , восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

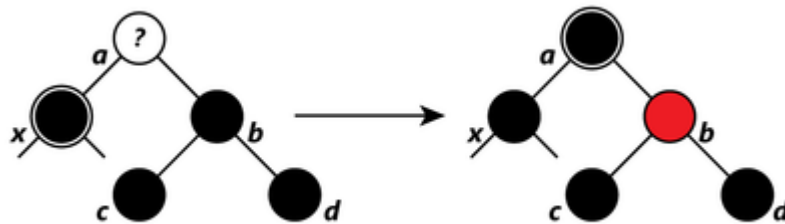


Рис 6. Случай удаления 2.

2. Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x , ни его отец не влияют на эту трансформацию.

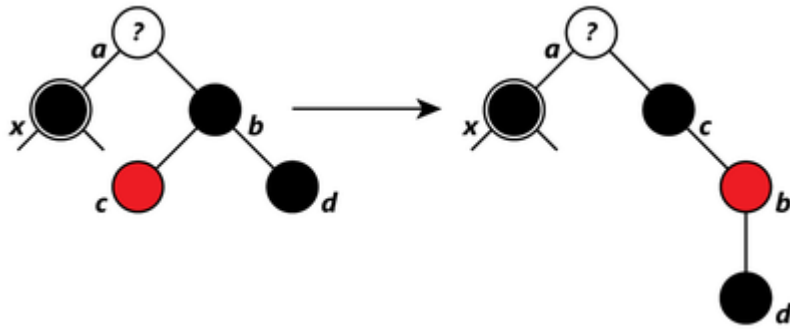


Рис 7. Случай удаления 3.

3. Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 33 и 44 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.

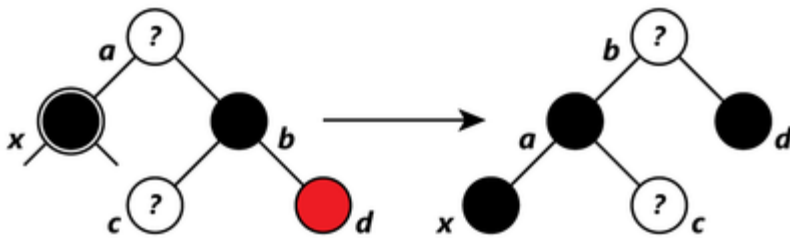


Рис 8. Случай удаления 4.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

4. AVL дерево

AVL дерево - сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Поэтому для соблюдения сбалансированности в каждом узле дерева хранится высота, которая считается как максимальная высота от любого потомка узла. С помощью высоты можно посчитать фактор сбалансированности, равный разности между высотой левого и правого поддеревьев этого узла. Листинг 4. Структура узла AVL дерева.

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

4.1 Вставка

Вставка узла в АВЛ дерево аналогична вставке в бинарное дерево поиска. Однако после вставки необходимо рекурсивно пройти по пути вставки узла и обновить высоты. В случае, если фактор сбалансированности будет нарушен, запускается балансировка для соответствующего узла.

4.2 Удаление

Удаление узла в АВЛ дереве аналогично вставке. То есть после удаления узла необходимо рекурсивно пройти по пути вставки узла и обновить высоты. В случае, если фактор сбалансированности будет нарушен, запускается балансировка для соответствующего узла.

4.3 Балансировка

В процессе вставки или удаления узлов в АВЛ дереве возможно возникновение ситуации, когда фактор сбалансированности некоторых узлов оказывается равными 2 или -2, т.е. возникает разбалансировка поддерева. Для выправления ситуации применяются левые и правые повороты вокруг тех или иных узлов дерева.

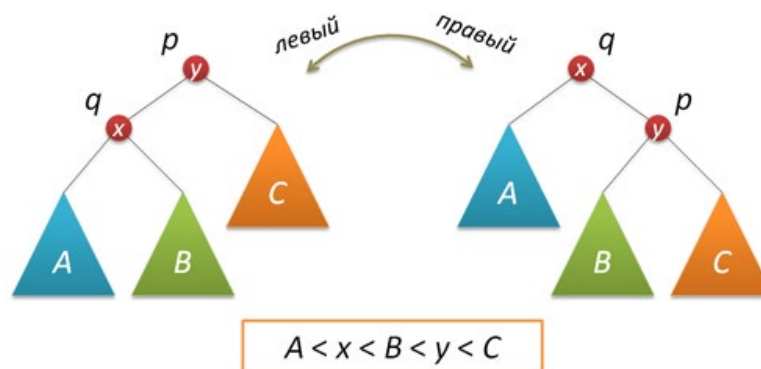


Рис 9. Левый и правый повороты поддерева.

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 .

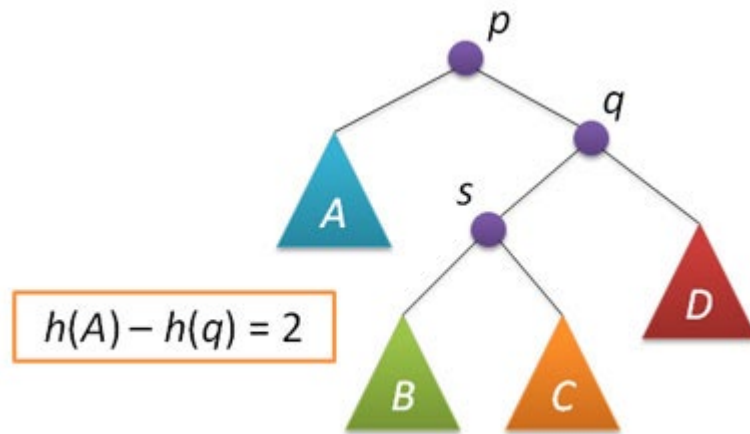


Рис. 10. Ситуация разбалансировки

Анализ возможных случаев в рамках данной ситуации показывает, что для исправления разбалансировки 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: $h(s) \leq h(D)$.

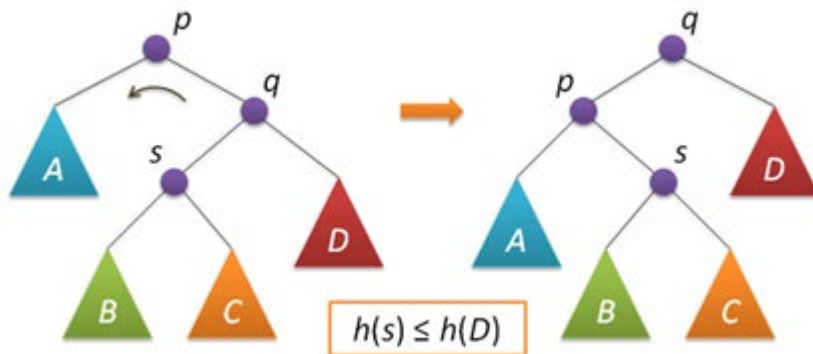


Рис. 11. Ситуация 1.

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 .

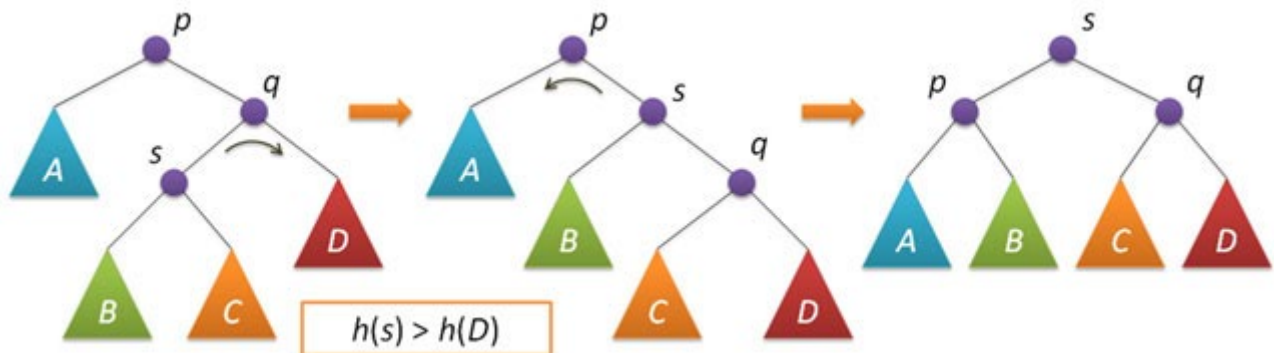


Рис. 12. Ситуация 2.

5. Косое дерево

Splay дерево (косое дерево) - это самобалансирующееся бинарное дерево поиска, в котором не нужно хранить никакой дополнительной информации, что делает его эффективным по памяти. Главной особенностью Splay-дерева является то, что оно не стремится поддерживать строгий баланс как АВЛ деревья или Красно-Черные деревья, но вместо этого уделяет внимание недавно использованным элементам, чтобы обеспечить быстрый доступ к ним.

После каждой операции доступа (поиск, вставка, удаление), элемент, с которым производилась операция, перемещается в корень дерева. Это обеспечивает "локальную" балансировку, при которой часто используемые элементы оказываются ближе к корню, что ускоряет последующий доступ к ним.

5.1 Вставка

Вставка узла в Splay дерево аналогична вставке в бинарное дерево поиска. Однако после вставки вызывается метод `splay` для вставленного узла.

5.2 Удаление

Удаление узла в Splay дерево аналогично удалению в бинарном дереве поиска. Однако после удаления дерево условно разрезается на 2 части и вызывается метод `merge` для правого и левого поддеревьев удаленного узла.

5.4 Операция Splay

Операция `splay` в Splay-дереве - это процесс перемещения определенного узла дерева в корень дерева с помощью серии вращений и переключений. Целью операции `splay` является перераспределение элементов дерева так, чтобы наиболее недавно использованный элемент оказался в корне, что повышает эффективность последующего доступа к этому элементу. Операция `splay` выполняется после каждой операции доступа (поиска, вставки, удаления).

В результате осуществления доступа к узлу возможны следующие случаи:

1. Узел является корневым. Мы просто возвращаем корень, больше ничего не делаем, так как узел, к которому осуществляется доступ, уже является корневым.
2. Узел является дочерним по отношению к корню (у узла нет прародителя). Если узел является левым потомком корня, то мы делаем правый поворот, а если правым потомком, то левый поворот.

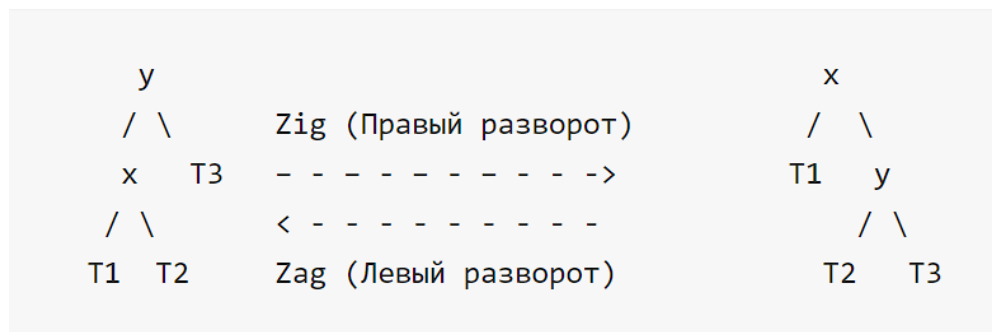


Рис. 13. Случай 2.

3. У узла есть и родитель, и прародитель. Возможны следующие варианты:
- а) Узел является левым потомком родительского элемента, и родитель также является левым потомком прародителя (два разворота вправо) или узел является правым потомком своего родительского элемента, и родитель также является правым потомком своего прародителя (два разворота влево).

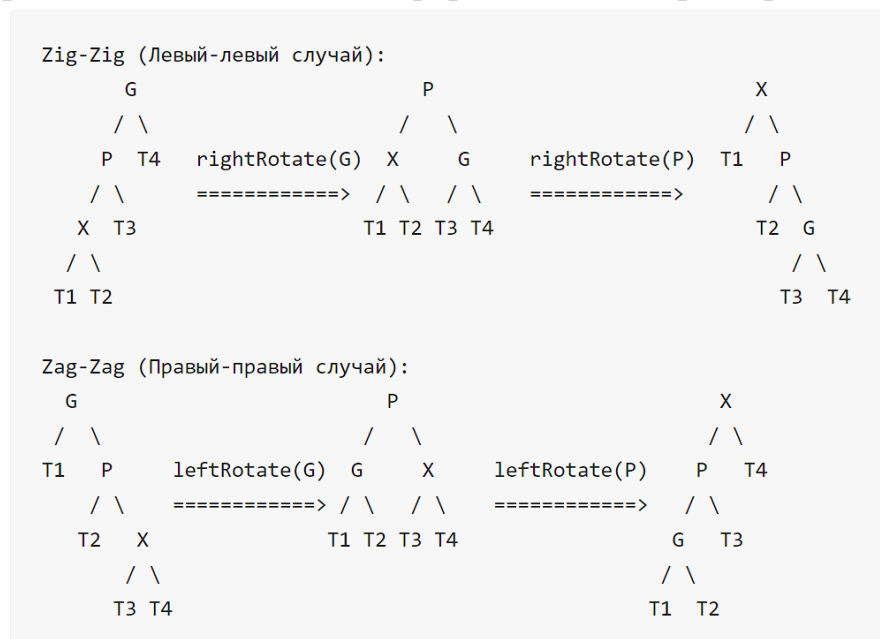


Рис. 14. Случай 3 а).

- б) Узел является левым потомком по отношению к родительскому элементу, а родитель является правым потомком прародителя (разворот влево с последующим разворотом вправо) ИЛИ узел является правым потомком своего родительского элемента, а родитель является левым потомком прародителя (разворот вправо с последующим разворотом влево).

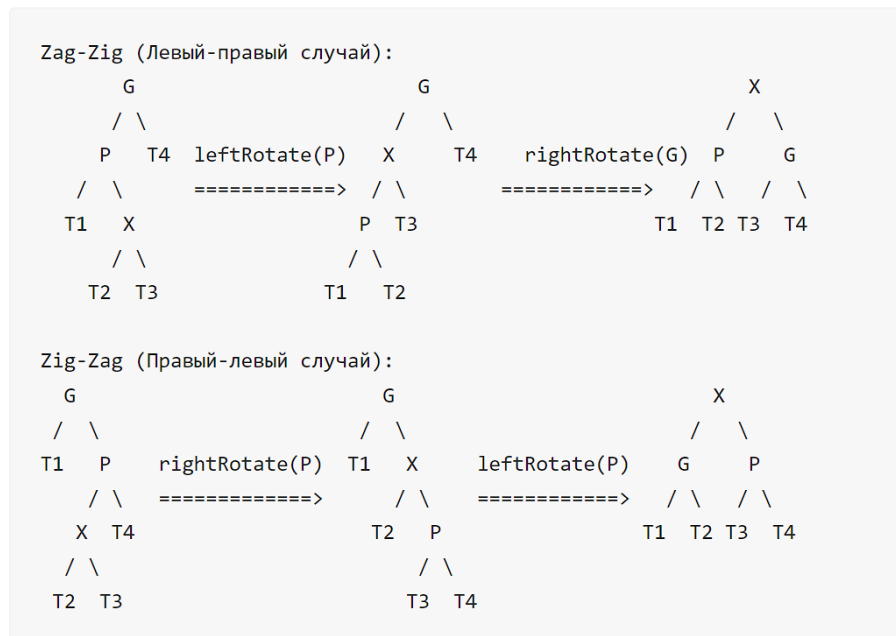


Рис. 15. Случай 3 б).

5.5 Операция Merge

Операция merge (слияние) в Splay дереве представляет собой процесс объединения двух Splay деревьев в одно. Она вызывается после удаления узла из дерева.

Функция merge получает на вход два дерева: левое `left_tree` и правое `right_tree`. Для корректной работы, ключи дерева `left_tree` должны быть меньше ключей дерева `right_tree`. Запускаем splay от самого большого элемента в дереве `left` (пусть это элемент `x`). После этого корень `left_tree` содержит элемент `x`, при этом у него нет правого ребёнка. Делаем `right_tree` правым поддеревом `x` и возвращаем полученное дерево.

6. Разработка и реализация приложения, управляющего хранилищем

6.1 Состав хранилища данных

Хранилище данных состоит из:

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

Каждый из этих компонентов обладает уникальным именем и представляет собой разновидность ассоциативного контейнера. Пул и схема данных основаны на структуре красно-черного дерева, а коллекция может быть реализована как АВЛ, Косое или Красно-Черное дерево в зависимости от выбора пользователя.

Листинг 5. Хранилище данных.

```
using type_collection = associative_container
    <
        Key,
        type_value*
    >;

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

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*>
    >;

using type_collection_rb = red_black_tree
    <
        Key,
        type_value*,
        compare_data_keys
    >;

using type_scheme_rb = red_black_tree
    <
        std::string,
        std::pair<type_collection_rb*, std::map<std::string,
type_order_collection*>>*,
        compare_str_keys
    >;

using type_pool_rb = red_black_tree
    <
        std::string,
        std::pair<type_scheme_rb*, memory*>,
```

```

        compare_str_keys
    >;

using type_data_base_rb = red_black_tree
    <
        std::string,
        std::pair<type_pool_rb*, memory*>,
        compare_str_keys
    >;

using type_collection_avl = avl_tree
    <
        Key,
        type_value*,
        compare_data_keys
    >;

using type_scheme_avl = avl_tree
    <
        std::string,
        std::pair<type_collection_avl*, std::map<std::string,
type_order_collection*>>*,
        compare_str_keys
    >;

using type_pool_avl = avl_tree
    <
        std::string,
        std::pair<type_scheme_rb*, memory*>,
        compare_str_keys
    >;

using type_data_base_avl = avl_tree
    <
        std::string,
        std::pair<type_pool_rb*, memory*>,
        compare_str_keys
    >;

using type_collection_splay = splay_tree
    <
        Key,
        type_value*,
        compare_data_keys
    >;

using type_scheme_splay = splay_tree
    <
        std::string,

```



```

        std::pair<type_collection_splay*, std::map<std::string,
type_order_collection*>>*,
        compare_str_keys
    >;

using type_pool_splay = splay_tree
    <
        std::string,
        std::pair<type_scheme_rb*, memory*>,
        compare_str_keys
    >;

using type_data_base_splay = splay_tree
    <
        std::string,
        std::pair<type_pool_rb *, memory*>,
        compare_str_keys
    >;

```

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

Для реализации данного механизма было необходимо использовать поведенческие паттерны проектирования “Команда” и “Цепочка обязанностей”. Они подразумевают отправление команд от пользователей и их последующую обработку с помощью обработчиков.

Принцип работы:

1. Создаются экземпляры классов, представляющих различные обработчики действий. Каждый обработчик реализует метод для обработки конкретного действия или команды.
2. Устанавливаются связи между обработчиками, формируя цепочку обязанностей

В результате связывания обработчиков друг с другом, создается цепочка обработчиков, где каждый обработчик знает о следующем обработчике в цепочке. Это позволяет последовательно передавать запросы через всю цепочку до тех пор, пока один из обработчиков не сможет обработать запрос или пока цепочка не закончится.

Листинг 6. Класс handler.

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

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

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();

static logger *get_logger();

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

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

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

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

virtual ~handler();

};

```

Для того, чтобы была возможность выполнять запросы к данным на заданный момент времени, используется контейнер `std::map`. В виде ключа в `map` хранится время изменения (или создания) объекта, а в качестве данных выступает вектор пар, состоящий из имени изменяемого поля и нового значения поля. 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.

6.3 Реализация механизма поиска по различным порядкам отношений на пространстве данных.

Для поиска по различным порядкам отношений необходимо хранить в схеме несколько коллекций, где значением узлов является указатель на объект данных, находящийся в основной коллекции, а ключом - определенное значение объекта данных, например: имя студента, оценка, время и т.д. То есть, по сути, все

коллекции содержат указатель на одни и те же данные, но с разными компараторами, что позволяет им производить поиск по различным порядкам отношений.

Листинг 7. Коллекция и схема.

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

6.4 Эффективное хранение строк

Для реализации эффективного хранения строк используется структурный паттерн Легковес (Flyweight). Паттерн Легковес - это структурный паттерн проектирования, который используется для оптимизации работы с большим количеством мелких объектов, снижая потребление памяти и улучшая производительность. Главная идея "Легковеса" заключается в том, чтобы разделить объекты на две части: внутреннее состояние, которое является общим для множества объектов, и внешнее состояние, которое может варьироваться для каждого объекта.

Листинг 8. Класс String_flyweight.

```
class string_flyweight
{
private:
    std::string _value;

public:
    const std::string &get_value() const;

    void set_value(const std::string &value);
};
```

Для управления легковесами используется класс `string_flyweight_factory`, который содержит пул легковесов. С помощью него достаточно иметь только указатель на легковес, больше не нужно хранить одинаковые строки.

Листинг 9. Класс `String_flyweight_factory`.

```
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);
};
```

Таким образом, мы храним только умные указатели на данные полей.

Листинг 10. Указатели на данные.

```
private:
    std::shared_ptr<string_flyweight> _surname_student;
    std::shared_ptr<string_flyweight> _name_student;
    std::shared_ptr<string_flyweight> _patronymic_student;
    std::shared_ptr<string_flyweight> _date_event;
    std::shared_ptr<string_flyweight> _time_event;
    std::shared_ptr<string_flyweight> _mark;
    std::shared_ptr<string_flyweight> _surname_teacher;
    std::shared_ptr<string_flyweight> _name_teacher;
    std::shared_ptr<string_flyweight> _patronymic_teacher;
```

7. Руководство пользователя

Команды можно вводить из консоли или считать из файла. Для запуска из файла необходимо ввести команду `START [filename]`.

Доступные команды:

1. `ADD_POOL ADD_POOL [pool_name]`
2. `ADD_SCHEME [pool_name] [scheme_name]`
3. `ADD_COLLECTION [pool_name] [scheme_name] [collection_name]`
4. `ADD [pool_name] [scheme_name] [collection_name]`
5. `GET [pool_name] [scheme_name] [collection_name]`
6. `GET_RANGE [pool_name] [scheme_name] [collection_name]`

7. UPDATE [pool_name] [scheme_name] [collection_name]
8. REMOVE [pool_name] [scheme_name] [collection_name]
9. REMOVE_COLLECTION [pool_name] [scheme_name] [collection_name]
10. REMOVE_SCHEME [pool_name] [scheme_name]
11. REMOVE_POOL [pool_name]
12. FINISH

Примеры работы программы:

Тест 1:

```
Print the commands:
ADD_POOL pool_1
Select an allocator:
1. Allocator list
2. Allocator descriptor
3. Allocator buddies
Print the number: 1
ADD_SCHEME pool_1 scheme_1
ADD_COLLECTION pool_1 scheme_1 collection_1
Select the type of tree for the collection:
1. Red-black tree
2. Splay tree
3. AVL tree
Print the number: 3

ADD pool_1 scheme_1 collection_1
SESSION_ID: 1
STUDENT_ID: 1
REPORTING_FORM: Exam
NAME_SUBJECT: Math
SURNAME_STUDENT: Ivanov
NAME_STUDENT: Ivan
PATRONYMIC_STUDENT: Ivanovich
DATE: 31/08/2023
TIME: 12:30:00
MARK: 4
SURNAME_TEACHER: Popov
NAME_TEACHER: Sergey
PATRONYMIC_TEACHER: Sergeevich
```

Рис. 16. Фото 1.

```

GET pool_1 scheme_1 collection_1
Choose field to be searched:
1. Id_session, id_student, reporting_form and name_subject
2. Surname student
3. Name student
4. Patronymic student
5. Date
6. Time
7. Mark
8. Surname teacher
9. Name teacher
10. Patronymic teacher
Number field: 1
SESSION_ID: 1
STUDENT_ID: 1
REPORTING_FORM: Exam
NAME_SUBJECT: Math
At what point in time do you want to receive data?
1. Current point in time
2. Set point in time
Print the number: 1

```

Рис. 17. Фото 2.

```

FOUND USER:

SURNAME_STUDENT: Ivanov
NAME_STUDENT: Ivan
PATRONYMIC_STUDENT: Ivanovich
DATE: 31/08/2023
TIME: 12:30:00
MARK: 4
SURNAME_TEACHER: Popov
NAME_TEACHER: Sergey
PATRONYMIC_TEACHER: Sergeevich

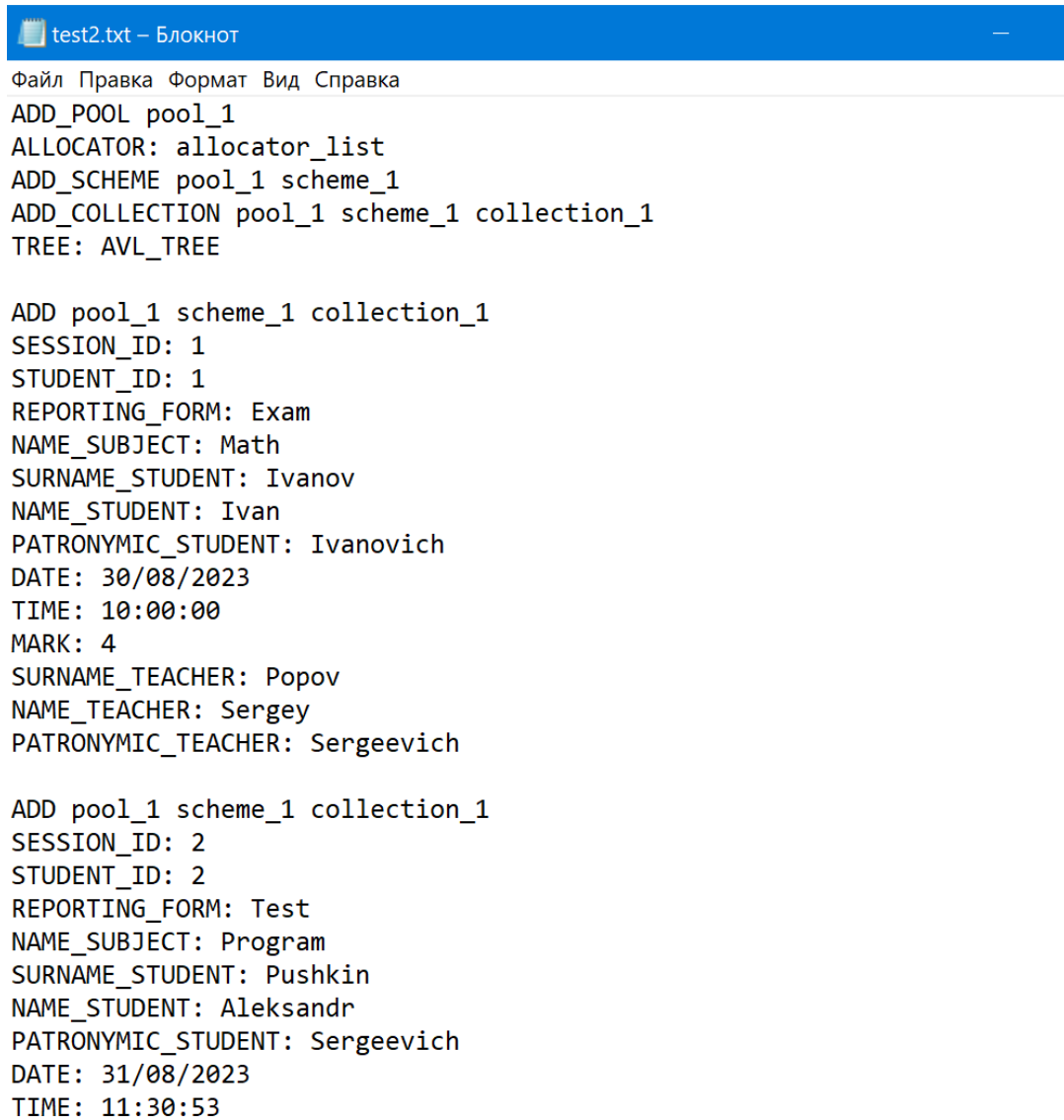
REMOVE_COLLECTION pool_1 scheme_1 collection_1
REMOVE_SCHEME pool_1 scheme_1
REMOVE_POOL pool_1
FINISH

Process finished with exit code 0
|

```

Рис. 18. Фото 3.

Тест 2:



```
test2.txt – Блокнот
Файл Правка Формат Вид Справка
ADD_POOL pool_1
ALLOCATOR: allocator_list
ADD_SCHEME pool_1 scheme_1
ADD_COLLECTION pool_1 scheme_1 collection_1
TREE: AVL_TREE

ADD pool_1 scheme_1 collection_1
SESSION_ID: 1
STUDENT_ID: 1
REPORTING_FORM: Exam
NAME_SUBJECT: Math
SURNAME_STUDENT: Ivanov
NAME_STUDENT: Ivan
PATRONYMIC_STUDENT: Ivanovich
DATE: 30/08/2023
TIME: 10:00:00
MARK: 4
SURNAME_TEACHER: Popov
NAME_TEACHER: Sergey
PATRONYMIC_TEACHER: Sergeevich

ADD pool_1 scheme_1 collection_1
SESSION_ID: 2
STUDENT_ID: 2
REPORTING_FORM: Test
NAME_SUBJECT: Program
SURNAME_STUDENT: Pushkin
NAME_STUDENT: Aleksandr
PATRONYMIC_STUDENT: Sergeevich
DATE: 31/08/2023
TIME: 11:30:53
```

Рис. 19. Фото 4


```
test2.txt – Блокнот
Файл Правка Формат Вид Справка
TIME: 11:30:53
MARK: 1
SURNAME_TEACHER: Smirnov
NAME_TEACHER: Artem
PATRONYMIC_TEACHER: Pavlovich

ADD pool_1 scheme_1 collection_1
SESSION_ID: 3
STUDENT_ID: 3
REPORTING_FORM: Coursework
NAME_SUBJECT: Physics
SURNAME_STUDENT: Sokolov
NAME_STUDENT: Dmitriy
PATRONYMIC_STUDENT: Ivanovich
DATE: 31/08/2023
TIME: 11:30:53
MARK: 4
SURNAME_TEACHER: Volkov
NAME_TEACHER: Nikita
PATRONYMIC_TEACHER: Pavlovich

GET_RANGE pool_1 scheme_1 collection_1
ID_SESSION_ID_STUDENT_REPORTING_FORM_NAME_SUBJECT
SESSION_ID: 1
STUDENT_ID: 1
REPORTING_FORM: Exam
NAME_SUBJECT: Math
SESSION_ID: 2
STUDENT_ID: 3
REPORTING_FORM: Coursework
NAME_SUBJECT: Program
CURRENT
```

Рис. 20. Фото 5.

```
Print the commands:
START test2.txt

FOUND USERS:

SURNAME_STUDENT: Ivanov
NAME_STUDENT: Ivan
PATRONYMIC_STUDENT: Ivanovich
DATE: 30/08/2023
TIME: 10:00:00
MARK: 4
SURNAME_TEACHER: Popov
NAME_TEACHER: Sergey
PATRONYMIC_TEACHER: Sergeevich

SURNAME_STUDENT: Pushkin
NAME_STUDENT: Aleksandr
PATRONYMIC_STUDENT: Sergeevich
DATE: 31/08/2023
TIME: 11:30:53
MARK: 1
SURNAME_TEACHER: Smirnov
NAME_TEACHER: Artem
PATRONYMIC_TEACHER: Pavlovich

FINISH

Process finished with exit code 0
```

Рис. 21. Фото 6.

8. Вывод

В результате выполнения курсового проекта было реализовано приложение, имеющее несколько уровней хранения - пул схема и коллекция, и позволяющее выполнять операции над коллекциями данных заданных типов и контекстами их хранения. Создание данного приложения потребовало от меня знаний об устройстве баз данных, сбалансированных деревьях, эффективном управлении памятью и ресурсами, а также знаний о распределении памяти, стандартах языка C++. Полученные знания помогут мне в дальнейших проектах.

9. Список использованных источников

1. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес - Приёмы объектно-ориентированного проектирования. Паттерны проектирования
2. Кнут - Искусство программирования, т. 3

10. Приложение

Ссылка на весь код приложения в github: