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: Федулов Дмитрий Андреевич

Группа: М8О-211Б-21

Преподаватель: Ирбитский И.С.

Оценка:

Дата:

Москва, 2023

# Содержание

1. Введение 3
2. Описание реализованного решения 4
   1. Логирование 4
   2. Распределение памяти 5
   3. Структуры данных 7
   4. Реализация базы данных 9
3. Руководство пользователя 11
4. Вывод 15
5. Приложение 15
6. Источники 15

# Введение

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

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

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

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

* Добавление новой записи по ключу;
* Чтение записи по 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 консоли. Также реализована возможность выполнения набора команд, подаваемых в качестве текстового файла.

Помимо основного функционала реализованы дополнительные возможности:

* Возможность взаимодействия с приложением через интерактивный диалог
* Возможность кастомизации (при создании) реализаций ассоциативных контейнеров, репрезентирующих коллекции данных
* Возможность кастомизации (для заданного пула схем) аллокаторов для размещения объектов данных
* Логирование приложения (а также его компонентов) в консоль и файловые потоки вывода

Данные о прохождении пайплайна содержат следующие поля:

## 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)

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

# Описание реализованного решения

* 1. **Логирование**

В реализации логгера был использован порождающий паттерн проектирования Builder. Паттерн Builder отделяет алгоритм поэтапного конструирования сложного продукта (объекта) от его внешнего представления так, что с помощью одного и того же алгоритма можно получать разные представления этого продукта.

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

Логгер конфигурируется двумя командами класса logger\_builder:

* + - Add\_stream - принимает на вход название файлового потока вывода и значение уровня логирования
    - Construct - отдает указатель на сконфигурировнный логгер на внешний уровень

Класс логгера logger содержит коллекцию потоков вывода, ключом которой является название потока, а значением - пара из указателя на открытый поток и количества логгеров, владеющих потоком, перечисление уровней логирования, а также метод log, принимающий на вход 2 аргумента - название потока и уровень логирования.

Листинг 1. Абстрактный класс logger

class logger{  
public:  
 enum class severity{  
 *trace*,  
 *debug*,  
 *information*,  
 *warning*,  
 *error*,  
 *critical* };  
 virtual ~logger(){  
  
 };  
 virtual logger const\* log(const std::string& target, severity) const = 0;  
};

# Распределение памяти

Аллокатор или распределительпамяти в языке программирования C++ — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Аллокатор представляет собой объект управляющий памятью. Под аллокатор отведен блок памяти заданного размера, которым он управляет. Память аллокатора содержит системную информацию и блоки памяти, которые запросил выделить внешний объект. В каждом аллокаторе доступно выделение и освобождение памяти.

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

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

Каждый класс аллокатора представляет собой реализацию интерфейса класса abstract\_allocator:

Листинг 2. Абстрактный класс memory

class memory{  
public:  
 virtual void\* allocate(size\_t target\_size) const = 0;  
  
 virtual void deallocate(void\* result) const = 0;  
  
 virtual ~memory(){}  
};

Для реализации приложения было использовано несколько видов аллокаторов, которые отличаются алгоритмами выделения и освобождения памяти:

* Аллокатор с освобождением в рассортированном списке
* Аллокатор с освобождением с дескрипторами границ
* Аллокатор с алгоритмом системы двойников

Каждый аллокатор реализован по определенному алгоритму, который описан в книге Д.Э. Кнут “Искусство программирования”, Том 1. **[1]**

# Структуры данных

Хранение коллекций данных в приложение сделано на базе структуры данных “АВЛ дерево”. С точки зрения реализации, структура данных являются производными классами от класса бинарного дерева поиска, которое в свою очередь наследуется от абстрактного класса associative\_container:

Листинг 2. Абстрактный класс associative\_container.

template<  
 typename tkey,  
 typename tvalue>  
class associative\_container  
{  
  
public:  
 enum class bypass\_detour{  
 *prefix*,  
 *postfix*,  
 *infix* };  
  
 struct key\_value\_pair  
 {  
 tkey \_key;  
 tvalue \_value;  
 };  
  
public:  
  
 virtual const tvalue& get(const tkey &key) const = 0;  
  
 virtual void insert(  
 tkey const &key,  
 const tvalue &value) = 0;  
  
 void operator+=(  
 key\_value\_pair pair);  
  
 virtual bool find(  
 key\_value\_pair \*target\_key\_and\_result\_value) = 0;  
  
 bool operator[](  
 key\_value\_pair \*target\_key\_and\_result\_value);  
  
 virtual tvalue remove(  
 tkey const &key) = 0;  
  
  
  
 tvalue operator-=(  
 key\_value\_pair pair);  
  
 tvalue operator-=(  
 tkey const &key);  
  
 virtual void bypass\_tree(bypass\_detour detour) const = 0;  
  
 virtual ~associative\_container() = default;  
  
};  
  
template<  
 typename tkey,  
 typename tvalue>  
void associative\_container<tkey, tvalue>::operator+=(  
 key\_value\_pair pair)  
{  
 return insert(pair.\_key, std::move(pair.\_value));  
}  
  
template<  
 typename tkey,  
 typename tvalue>  
bool associative\_container<tkey, tvalue>::operator[](  
 key\_value\_pair \*target\_key\_and\_result\_value)  
{  
 return find(target\_key\_and\_result\_value);  
}  
  
template<  
 typename tkey,  
 typename tvalue>  
tvalue associative\_container<tkey, tvalue>::operator-=(  
 key\_value\_pair pair)  
{  
 return remove(pair.\_key);  
}  
  
template<  
 typename tkey,  
 typename tvalue>  
tvalue associative\_container<tkey, tvalue>::operator-=(  
 tkey const &key)  
{  
 return remove(key);  
}

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

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

**Сбалансированные деревья** были придуманы, чтобы обеспечить сложность выполнения операций вставки, удаления и поиска за O(log(n)). Из-за того, что обычное бинарное дерево поиска может выродиться в односвязный список, или не всегда может иметь высоту равную log(n) (n - количество узлов в дереве), то и сложность операций вставки, удаления и поиска не всегда равна логарифму. Эту проблему решают сбалансированные деревья.

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

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

# Реализация базы данных

База данных реализована как АВЛ дерево с указателями на другие АВЛ деревья (пулы), которые в свою очередь хранят на АВЛ деревья схем, в которых хранятся указатели на абстрактный класс associative\_container (для дальнейшей конфигурации типа дерева для хранения информации о прохождении сессии в университете):

Листинг 4. Объявление основного АВЛ дерева \_database.

AVL\_tree<std::string, AVL\_tree<std::string, AVL\_tree<std::string, associative\_container<session\_key, session\_value>\*, string\_comparer>\*, string\_comparer>\*, string\_comparer>\* \_database;

У класса database есть 1 публичных метода:

* + - dialogue() - Открывает интерактивный диалог и контролирует все ошибки выполнения. В диалоге предусмотрены все невалидные действия со стороны пользователя, а также обрабатываются все ошибки при работе с приложением.

# Реализованные команды приложения:

* Вставка записи по ключу:

Insert: (pool name) (scheme name) (collection name)

Key: (id session) (id student) (name subject) (reporting format)

Value: (name student) (surname student) (patronymic student) (exam date) (start date) (estimation) (name teacher) (surname teacher) (patronymic teacher)

* Чтение по ключу:

Get key information: (pool name) (scheme name) (collection name)

Key: (id session) (id student) (name subject) (reporting format)

* Чтение в диапазоне:

Read range: (pool name) (scheme name) (collection name)

[Key: (id session) (id student) (name subject) (reporting format)]

[Key: (id session) (id student) (name subject) (reporting format)]

(вместо указания конкретного значения ключа как границу, допускается оставить квадратные скобки пустыми, чтобы установить диапазон от начала/до конца коллекции данных)

* Удаление записи:

Remove: (pool name) (scheme name) (collection name)

key: (id session) (id student) (name subject) (reporting format)

* Создание пула:

Add pool: (pool\_name) [ {alloc\_name} {size(only if needed)} {alloc\_mode(only if exists)} ]

* Удаление пула:

Delete pool: (pool name)

* Создание схемы:

Add scheme: (pool name) (scheme name)

* Удаление схемы:

Delete scheme: (pool name) (scheme name)

* Создание коллекции:

Add collection: (pool name) (scheme name) (collection name)

* Удаление коллекции:

Delete collection: (pool name) (scheme name) (collection name)

# Руководство пользователя

Продемонстрируем работу и взаимодействие с приложением. Для начала выполним список команд из файла coursework\_commands.txt:

В main.cpp мы создаем указатель на объект типа logger и нашу базу данных. В нашей базе данных мы вызываем публичный метод dialogue(), который и осуществляет взаимодействие с пользователем.

Листинг 4

int main() {  
 logger\_builder\* builder = new logger\_builder\_concrete();  
 logger\* mylogger = builder->add\_stream("console", logger::severity::*warning*)->add\_stream("C:\\Users\\HP\\CLionProjects\\labs\_4\_sem\\courework\\debug.txt", logger::severity::*debug*)->construct();  
 auto\* data = new database(mylogger);  
 try{  
 data->dialogue();  
 }catch (const std::logic\_error& er){  
 std::cout<<er.what()<<std::endl;  
 }  
 delete data;  
 delete mylogger;  
 delete builder;  
 return 0;  
}

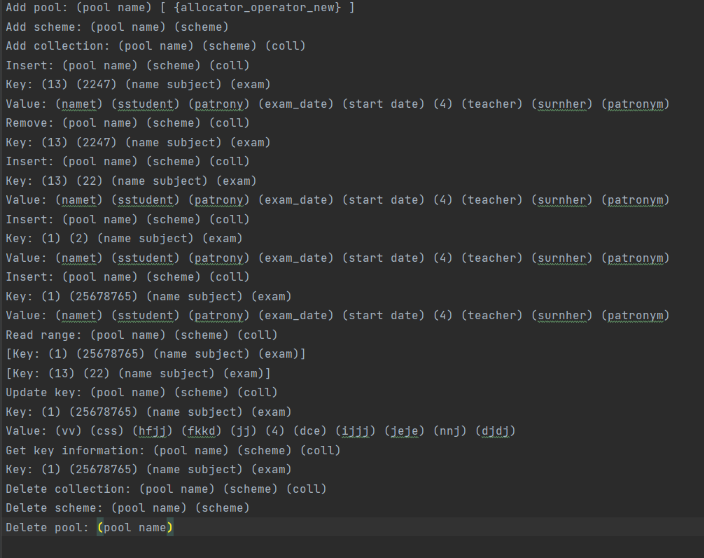
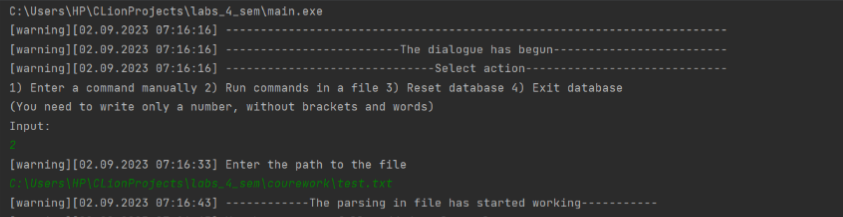


Рисунок 1. Команды в файле.

Вызовем метод parsing\_in\_file, передав в качестве аргумента путь к данному файлу (задается аргументом командной строки):

.

Рисунок 2. Начало работы и ввод файла из которого будет считывание

В результате получаем вывод в консоли результата выполнения команд:

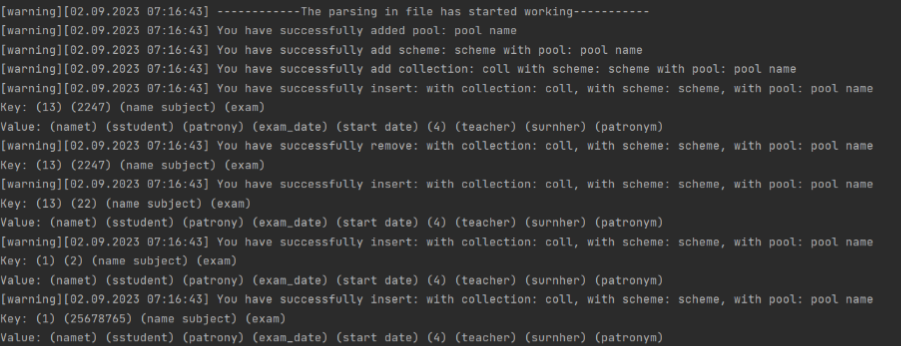


Рисунок 3. Вывод в консоль. 1 часть

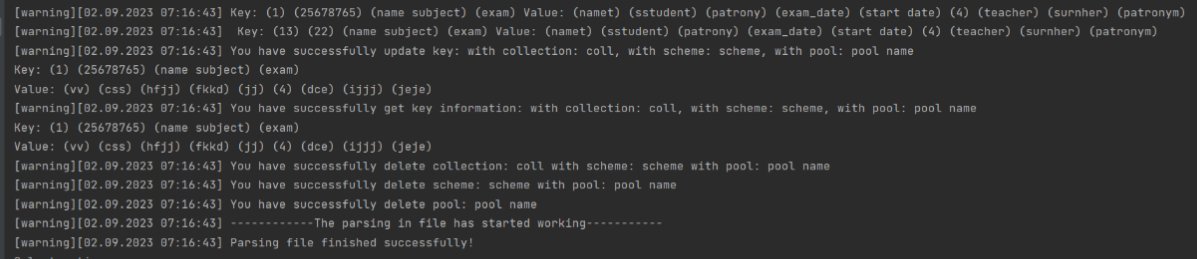


Рисунок 4. Вывод в консоль. 2 часть

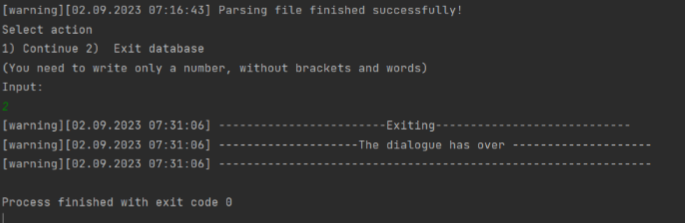


Рисунок 5. Завершение работы

# Вывод

Таким образом, реализовано приложение, которое позволяет хранить информацию о прохождении сессии в университете в виде структурированной коллекции данных (пулы, схемы, коллекции).Для этого мной были реализованы также и прикладные программы, такие как логгер, аллокаторы, бинарное дерево поиска и АВЛ дерево. Приложение способно удалять и вставлять как пулы, схемы и коллекции, так и нужные нам данные, а также выполнять другие функции. Создание данного приложения требует знаний об устройстве баз данных, эффективном управлении памяти и ресурсами, а также знаний о распределении памяти, стандартах языка C++ и сбалансированных деревьях.

# Приложение

Весь код приложения и его компонент доступен в репозитории на GitHub:

https://github.com/Swiftcr7/labs\_4\_sem

# Источники

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