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: Хасанов Д.Р.

Группа: М8О-213Б-21

Преподаватель: Ирбитский И.С.

Оценка:

Дата:

Москва, 2023

Содержание

[**Введение** 3](#_Toc136970883)

[**Описание реализованного решения** 5](#_Toc136970884)

[**Обработка запросов** 5](#_Toc136970885)

[**API базы данных** 6](#_Toc136970886)

[**Splay-дерево** 7](#_Toc136970887)

[**Splay** 8](#_Toc136970888)

[**Merge** 12](#_Toc136970889)

[**Вставка** 13](#_Toc136970890)

[**Удаление** 13](#_Toc136970891)

[**Поиск** 13](#_Toc136970892)

[**Аллокаторы** 13](#_Toc136970893)

[**Аллокатор на глобальной куче** 13](#_Toc136970894)

[**Освобождение в рассортированном списке** 14](#_Toc136970895)

[**Освобождение с дескрипторами границ** 15](#_Toc136970896)

[**Система двойников** 16](#_Toc136970897)

[**Методы поиска блока** 17](#_Toc136970898)

[**Руководство пользователя** 17](#_Toc136970899)

[**Демонстрация работы приложения** 18](#_Toc136970900)

[**Вывод** 23](#_Toc136970901)

[**Приложение** 24](#_Toc136970902)

[**Список используемых источников** 25](#_Toc136970903)

# Введение

Целью курсового проекта является реализация приложения на языке программирования C++ (стандарт C++14 и выше), позволяющего выполнять операции над коллекциями данных заданных типов 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 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 её ключу;
* чтение набора записей с ключами из диапазона [minbound… maxbound];
* обновление данных для записи по ключу;
* удаление существующей записи по ключу.

Во время работы приложения возможно выполнение также следующих операций:

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

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

В дополнение к основному заданию реализованы четыре аллокатора для возможности их кастомизации (для заданного пула схем) для размещения объектов данных:

* освобождение в рассортированном списке;
* освобождение с дескрипторами границ;
* система двойников;
* аллокатор на глобальной куче.

Ассоциативный контейнер представляет собой косое дерево, а коллекции данных представляют собой 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)).

**Описание реализованного решения**

Архитектура приложения представляет собой базу данных, хранящая в себе пулы схем, которые в свою очередь хранят коллекции данных. Каждый уровень (пул, схема, коллекция) представляет собой класс, предоставляющий API для работы с ним.

Пользователь посылает запрос в базу данных, которая обрабатывает его и возвращает уведомление о выполнении с нужными данными, если это необходимо.

**Обработка запросов**

Обработка запросов осуществлена с помощью поведенческого паттерна – цепочка обязанностей. Он позволяет избежать привязки отправителя запроса к его получателю, предоставляя возможность обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос по этой цепочке, пока он не будет обработан. Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который действует точно так же. У объекта, отправившего запрос, отсутствует информация об обработчике.

Основные условия для применения паттерна цепочка обязанностей:

* запрос может быть обработан более чем одним объектом, причём настоящий обработчик заранее неизвестен и должен быть найден автоматически;
* запрос должен быть отправлен одному из нескольких объектов, без явного указания, какому именно;
* набор объектов, способных обработать запрос, должен задаваться динамически.

Все три условия в приложении выполнены.

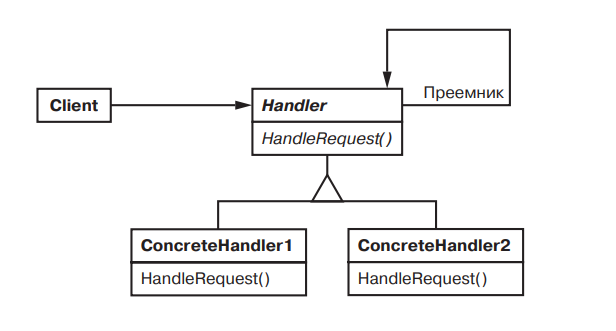


Рисунок 1. Типичная структура взаимодействия компонентов

В программе имеется класс handler, который хранит в себе список объектов команд, которые обрабатывают запрос. У этого класса имеются два метода:

1. can\_execute;
2. execute;

Метод can\_execute проверяет название команды, которая была отправлена запросом, и после этого либо начинает обработку аргументов, либо уведомляет handler о том, что он не отвечает за эту команду, после чего продолжается поиск нужного объекта в списке. Для обработки аргументов созданы 3 класса, которые отвечают за валидацию чисел, дат и времени, которые являются аргументами команды.

Метод execute взаимодействует с API базы данных, после получения сигнала о том, что отправленная команда, может быть исполнена данных объектом списка. [2]

**API базы данных**

База данных предоставляет API для работы с ней в виде набора методов таких как:

* add\_pool;
* add\_scheme;
* add\_collection;
* add\_data;
* remove\_pool;
* remove\_scheme;
* remove\_collection;
* remove\_data;
* get\_data;
* get\_data\_between\_keys;
* update\_data.

Метод add\_pool принимает название пула, аллокатор, который должен храниться в пуле, размер и метод выделения для аллокатора и добавляет его в базу данных.

Метод add\_scheme принимает называние пула, в который должна быть добавлена схема, и её название.

Метод add\_collection принимает название пула и схемы, в которую должна быть добавлена коллекцию данных, и название коллекции.

Метод add\_data принимает название пула, схемы, коллекции в которую должны быть добавлены данные, а также сами данные – ключ и значение.

Метод remove\_pool принимает название пула, который должен быть удалён из базы данных.

Метод remove\_scheme принимает название схемы и пула, из которого она должна быть удалена.

Метод remove\_collection принимает название коллекции, а также пула и схемы, из которой она должна быть удалена.

Метод remove\_data принимает название пула, схемы и коллекции, из которой должны быть удалены данные, а также ключ, по которому эти данные хранятся.

Метод get\_data принимает название пула, схемы и коллекции, в которой хранятся данные, и ключ, по которому ищутся данные.

Метод get\_data\_between\_keys принимает название пула, схемы и коллекции, в которой лежат данные, и два ключа, между которыми нужно собрать данные.

Метод update\_data принимает название пула, схемы, коллекции и ключ, для которого нужно изменить данные.

**Splay-дерево**

Ассоциативный контейнер для хранения пулов, схем и коллекций данных представляет собой шаблонное косое дереве, реализованное на основе бинарного дерева поиска.

Подобно красно-черным и  АВЛ-деревьям, splay-дерево (или косое дерево) также является самобалансирующимся бинарным деревом поиска. Основная идея splay-дерева состоит в том, чтобы помещать элемент, к которому недавно осуществлялся доступ, в корень дерева, что делает этот элемент, доступным за время порядка O(1) при повторном доступе. Вся суть заключается в том, чтобы использовать концепцию локальности ссылок (в среднестатистическом приложении 80% обращений приходятся на 20% элементов). Представьте себе ситуацию, когда у нас есть миллионы или даже миллиарды ключей, и лишь к некоторым из них обращаются регулярно, что весьма вероятно для многих типичных приложениях. Разберём операции над косым деревом.

**Splay**

Данная операция является главной в косом дереве, с её помощью узел, к которому недавно осуществлялся доступ, подымается в корень дерева из-за чего все основные операции над деревом, такие как вставка, удаление, поиск имеют амортизированную сложность O(log(n)).

Принцип работы этой операции состоит в том, что она выполняет малые и большие повороты относительно узлов дерева, в следствии чего нужный нам узел становиться корнем. Эту операцию можно разделить на 3 случая и ещё 3 будут зеркальными.

1. zig (zag);
2. zig-zig (zag-zag);
3. zig-zag (zag-zig).

Первый случай возникает если *p* — корень дерева с сыном *x*. В этой ситуации нужно совершить один поворот вокруг ребра (*x*, *p*), делая x корнем дерева. Данный случай является крайним и выполняется только один раз в конце, если изначальная глубина *x* была нечетной.

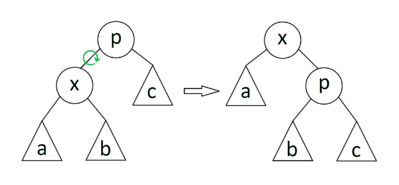


Рисунок 2. Операция zig (малый правый поворот)

Листинг 1. Реализация малого правого поворота на языке C++

node \*rightRotate(node \*current\_node)

{

node \*tmp = current\_node;

current\_node = current\_node->left\_subtree;

tmp->left\_subtree = current\_node->right\_subtree;

current\_node->right\_subtree = tmp;

return current\_node;

}

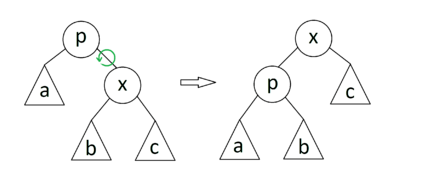


Рисунок 3. Операция zag (малый левый поворот)

Листинг 2. Реализация малого левого поворота на языке C++

node \*leftRotate(node \*current\_node)

{

node \*tmp = current\_node;

current\_node = current\_node->right\_subtree;

tmp->right\_subtree = current\_node->left\_subtree;

current\_node->left\_subtree = tmp;

return current\_node;

}

Второй случай возникает если *p* — не корень дерева, а *x* и *p* — оба левые или оба правые дети, то делаем поворот ребра (*p*, *g*), где *g* отец *p*, а затем поворот ребра (*x*, *p*).

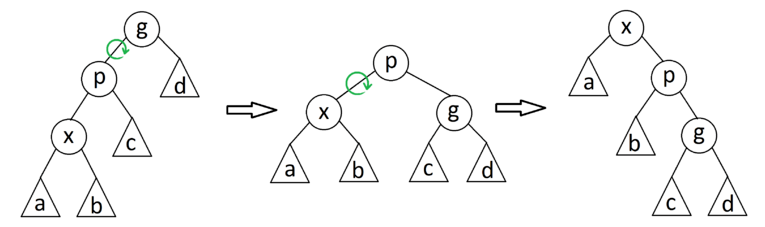


Рисунок 4. Операция zig-zig (большой правый поворот)

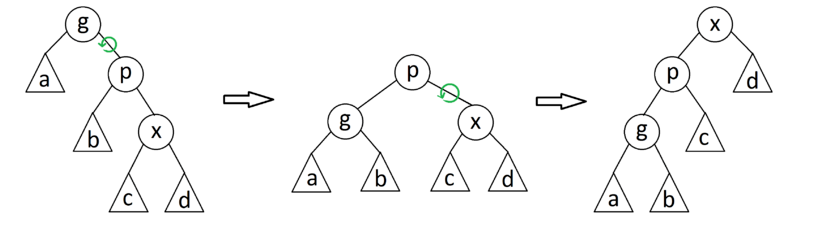


Рисунок 5. Операция zag-zag (большой левый поворот)

Последний случай возникает если *p* — не корень дерева и *x* — левый ребенок, а *p* — правый, или наоборот, то делаем поворот вокруг ребра (*x*, *p*), а затем поворот нового ребра (*x*, *g*), где *g* — бывший родитель *p*.

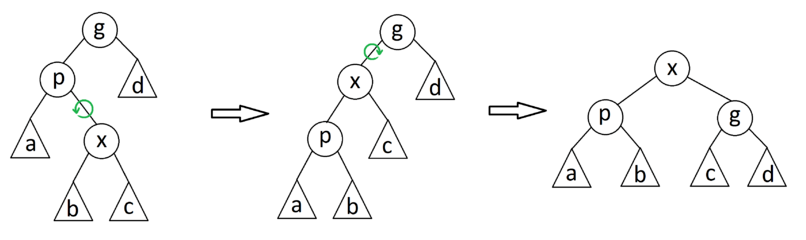


Рисунок 6. Операция zig-zag (малый левый и малый правый поворот)

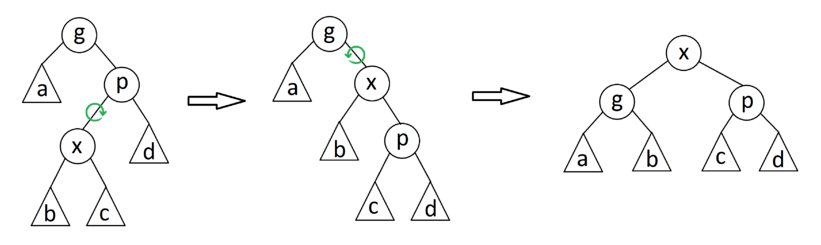


Рисунок 7. Операция zag-zig (малый правый и малый левый поворот)

Данная операция занимает O(d) времени, где d — длина пути от *x* до корня.

Листинг 3. Реализация операции splay на языке C++

node \*splay(node \*root, int key)

{

if (root == NULL || root->key == key)

return root;

if (root->key > key)

{

if (root->left == NULL) return root;

if (root->left->key > key)

{

root->left->left = splay(root->left->left, key);

root = rightRotate(root);

}

else if (root->left->key < key)

{

root->left->right = splay(root->left->right, key);

if (root->left->right != NULL)

root->left = leftRotate(root->left);

}

return (root->left == NULL)? root: rightRotate(root);

}

else

{

if (root->right == NULL) return root;

if (root->right->key > key)

{

root->right->left = splay(root->right->left, key);

if (root->right->left != NULL)

root->right = rightRotate(root->right);

}

else if (root->right->key < key)

{

root->right->right = splay(root->right->right, key);

root = leftRotate(root);

}

return (root->right == NULL)? root: leftRotate(root);

}

}

**Merge**

Операция merge вызывается при удалении узла из дерева. Она склеивает два поддерева удаляемого узла – левое и правое. Суть заключается в том, что нужно найти наибольший элемент в левом поддереве, поднять его в корень с помощью операции splay и правым поддеревом приклеить правого потомка удаляемого узла. Так как все ключи левого поддерева меньше всех ключей правого, то баланс не нарушится.

Листинг 4. Реализация операции merge на языке C++

node \*merge(node \*left\_subtree, node \*right\_subtree)

{

if (left\_subtree == nullptr && right\_subtree == nullptr)

{

return nullptr;

}

if (left\_subtree == nullptr && right\_subtree != nullptr)

{

return right\_subtree;

}

else if (left\_subtree != nullptr && right\_subtree == nullptr)

{

return left\_subtree;

}

else

{

node \*successor = left\_subtree;

while (successor->right\_subtree != nullptr)

{

successor = successor->right\_subtree;

}

left\_subtree = splay(successor);

left\_subtree->right\_subtre = right\_subtree;

return left\_subtree;

}

}

**Вставка**

Операция вставки в splay дерево очень похожа на вставку в бинарное дереве поиска, но с небольшими изменениями, после того как узел будет вставлен в нужное место, необходимо так же выполнить операцию splay, тем самым он окажется в корне дерева.

**Удаление**

Операция удаления в косом дерево работает следующим образом. Для начала нужно найти узел, который мы хотим удалить, после этого необходимо выполнить операцию splay от найденного узла и операцию merge от его левого и правого поддерева.

**Поиск**

Операция поиска в косом дереве аналогична операции поиска в бинарном дереве поиска за исключением того, что после нахождения узла, необходимо выполнить операцию splay от этого узла.

**Аллокаторы**

Память под данные, хранимые в коллекциях, выделяется в реализованных аллокаторах. Для этого при создании пула необходимо указать какой из аллокаторов будет использоваться для выделения памяти под данные.

Всего реализовано 4 различных аллокатора:

1. на глобальной куче;
2. с освобождением в рассортированном списке;
3. с освобождением с дескрипторами границ;
4. система двойников.

**Аллокатор на глобальной куче**

Аллокатор на глобальной куче представляет собой обёртку над встроенным выделением динамической памяти. В моей реализации это класс, имеющий методы allocate, deallocate, которые запрашивают память из глобальной кучи и отдают её пользователю и очищают выделенную память соответственно.

### **Освобождение в рассортированном списке**

Аллокатор с освобождением в рассортированном списке представляет собой класс, хранящий указатель на доступную ему область памяти, из которой он будет выделять место и возвращать на него указатель пользователю.

Идея заключается в том, что свободные блоки будут хранить указатель на следующий свободный блок и свой размер, в результате у нас получиться своего рода односвязный рассортированный список, что и отражает его название. В занятых блоках мы храним только размер самого блока, так как указатели на занятые блоки нам не понадобятся.

Выделение памяти в таком аллокаторе происходит следующим образом. Пользователь запрашивает необходимый ему размер памяти, после чего проходя по нашему списку свободных блоков каждый раз проверяется подходит ли блок по размеру. После нахождения нужного блока начинается процесс обновления списка доступных блоков, в зависимости от того, останется ли место в блоке после изъятия из него места нужного пользователю, тут у нас появляется 2 случая. Первый случай возникает, когда размер блока равен размеру, запрашиваемому пользователем, здесь нужно просто переставить указатель у предыдущего блока с текущего на следующий после него. Второй случай происходит, если после выделения останется место в текущем блоке, в таком случае нужно отделить занятую часть блока от свободной, уменьшить размер блока на размер выделяемой памяти и переставить указатель у предыдущего блока на оставшуюся часть. Так же стоит учесть, что если найденный для выделения блок был первым в списке, то необходимо в сервисной части аллокатора указатель на первый свободный блок переставить на следующий после выделяемого. Когда все указатели переставлены можно установить размер выделяемого блока и отдать его пользователю.

Теперь перейдём к освобождению блока. Сначала нужно найти адреса следующего и предыдущего блока, для этого следует проходя по списки свободных блоков, сравнивать адрес освобождаемого блока и блока из списка, если адрес текущего блока станет больше адреса освобождаемого, значит мы нашли нужный нам блок. Теперь нужно правильно расставить указатели, здесь появляются два случая. Первый случай – это когда освобождаемый блок стоит сразу перед доступным блоком, необходимо слить два этих блока в один, для этого увеличим размер освобождаемого блока на размер стоящего после него блока и указатель на следующий свободный блок переставить через один. Второй случай возникает, если между найденным блоком и освобождаемым есть ещё блоки, тогда нужно у освобождаемого блока указатель на следующий после него установить на найденный блок. Аналогично нужно рассмотреть случаи с предыдущим блоком.

Данный алгоритм, определенно, несколько прямолинеен. Однако всего лишь небольшое изменение стратегии может существенно повысить скорость его работы.

Алгоритм выделения может использоваться как для больших, так и для малых значений. Предположим, что нас интересуют, в первую очередь, большие значения. Может получиться так, что при выделении в свободном блоке останется очень мало место, он настолько мал, что практически бесполезен и только засоряет список. В таком случае лучше выделить немного больше памяти, чем запросил пользователь, для повышения скорости алгоритма. [1]

**Освобождение с дескрипторами границ**

Рассмотрим другой подход к выделению памяти. Идея этого алгоритма состоит в поддержании двусвязного списка занятых блоков, так что элементы списка могут быть легко удалены из произвольной части списка. В своей реализации данного аллокатора, я храню в занятых блоках их размер и указатели на следующий и предыдущий занятые блоки, получается своего рода двусвязный список, а свободные не хранят никакой информации. Этот подход позволит ускорить алгоритм освобождения блоков.

Начнём с выделения, в данном аллокаторе оно несколько сложнее, чем в предыдущем. Сначала мы ищем блок, размер которого подойдёт под запрос пользователя. Так как у нас список из занятых блоков, этот алгоритм несколько усложняется, теперь для нахождения подходящего блока нужно вычислять разницу между адресами занятых блоков. Для добавления найденного блока в список занятых нужно указатели у предыдущего и следующего после текущего блока переставить на него, а его указатели установить на них. Конечно, нужно не забыть проверить есть ли предыдущий и следующий блоки. Когда блок добавлен в список занятых, его можно отдать пользователю.

Освобождение в этом аллокаторе очень простое, для этого нужно получить предыдущий и следующий блоки для освобождаемого, так как у нас есть список занятых блоков, сделать это не составит никаких трудностей. После этого переставляем их указатели друг на друга и алгоритм освобождения закончен. Так же не стоит забывать проверить есть ли эти блоки в списке.

Принципиально новое свойство этого метода освобождения памяти заключается в освобождении блока за, по сути, фиксированное время. [1]

**Система двойников**

Изучим теперь другой подход к динамическому выделению памяти для использования на двоичных компьютерах. В этом методе применяется по одному дополнительному биту занятости на каждый блок, кроме того, все блоки должны иметь размер 1, 2, 4, 8, 16 и так далее. Если размер блока не равна байт для некоторого целого *k*, выбирается следующая более высокая степень 2 и часть выделенной памяти при этом не используется.

Суть этого метода заключается в организации отдельных списков доступных блоков каждого размера , 0 ≤ *k* ≤ *m*. Весь пул распределяемого пространства памяти состоит из байт, адреса которых, предположим, находятся в диапазоне от 0 до - 1. Изначально весь блок из байт свободен. Позже, при требовании блока из байт и отсутствии свободного блока такого размера больший доступный блок разбивается на две равные части, в конечном итоге появляется блок необходимого размера . Когда один блок разделяется на два (каждый половинного размера по сравнению с исходным), эти блоки называются двойниками. Позже, когда оба двойника вновь становятся свободны, они объединяются в один большой блок. Таким образом, процесс выделения и освобождения может осуществляться бесконечно, если только в какой-то момент вся доступная память не окажется занятой.

Ключевой факт, лежащий в основе практической ценности этого метода, состоит в том, что если известен адрес блока и его размер, то известен и адрес его двойника. Для того чтобы понять, почему это так, сначала заметим, что во время работы алгоритма адрес блока размером кратен . Другими словами, адрес в двоичной записи содержит справа как минимум *k* нулей. Данное наблюдение легко выводится по индукции. Адрес двойника легко вычисляется с помощью операции исключающее или.

Кроме бита занятости, имеющегося в каждом блоке, в свободных блоках есть два указателя – на предыдущий и на следующий свободный блок.

Перейдём к алгоритму выделения. Сначала в списке свободных блоков ищется подходящий под запрашиваемый размер блок, после чего пока его размер подходит под требования, делим его пополам, переставляя указатели и меняя размер блоков. В конце проставляем бит занятости и отдаём блок пользователю.

Алгоритм освобождения делится на два случая. Первый случай возникает, если двойник освобождаемого блока свободен, в этом случае нужно объединять двойников пока это возможно, а второй случай – когда двойник занят, тогда нужно найти будущие предыдущий и следующий блоки для освобождаемого блока в списке, после чего переставить у них и освобождаемого блока указатели. [1]

**Методы поиска блока**

Во всех аллокаторах есть возможность поиска тремя способами:

1. метод первого подходящего;
2. метод лучшего подходящего;
3. метод худшего полходящего.

Первый метод очень прост, как только найден блок, размер которого нам подходит, то он заканчивает поиск.

Идея метода лучшего подходящего состоит в том, что он обходит весь список блоков, и каждый раз сравнивает размер текущего и предыдущего найденных блоков, заменяя найденный блок при условии, что его размер всё ещё подходит, но он меньше размера предыдущего найденного блока.

Метод худшего подходящего схож с предыдущим методом, отличие заключается в том, что он ищет блок, размер которого будет больше предыдущего найденного.

**Руководство пользователя**

Для работы приложения на вход как аргумент командной строки нужно подать файл с записанными командами, которые должны обработаться.

Формат команд таков, что сначала пишется команда, которая должна выполниться, а после через пробел аргументы этой команды. Представлены 11 команд со следующим набором аргументов:

1. ADD\_POOL pool\_name allocator\_type memory\_size allocate\_mod
2. ADD\_SCHEME pool\_name scheme\_name
3. ADD\_COLLECTION pool\_name scheme\_name collection\_name
4. ADD\_DATA pool\_name scheme\_name collection\_name id\_session id\_student format subject surname name patronymic data time mark
5. REMOVE\_POOL pool\_name
6. REMOVE\_SCHEME pool\_name scheme\_name
7. REMOVE\_COLLECTION pool\_name scheme\_name collection\_name
8. REMOVE\_DATA pool\_name scheme\_name collection\_name id\_session id\_student format subject
9. GET\_DATA pool\_name scheme\_name collection\_name id\_session id\_student format subject
10. GET\_RANGE pool\_name scheme\_name collection\_name id\_session\_1 id\_student\_1 format\_1 subject\_1 id\_session\_2 id \_student\_2 format\_2 subject\_2
11. UPDATE\_DATA pool\_name scheme\_name collection\_name id\_session id\_student format subject surname name patronymic date time mark

Названия пулов, схем и коллекций могут быть любыми, кроме пустых строк, id сессии и id студента должны быть целыми неотрицательными числами, формат отчётности имеет 3 разновидности – course\_work, exam и test, если был выбран формат – test, то значение mark должно быть в диапазоне от 0 до 1, в двух других случаях в диапазоне от 2 до 5. Дата должна иметь вид дд.мм.гггг или же дд/мм/гггг, а время чч:мм. Аллокатор имеет 4 вариации – global\_heap, sorted\_list, border\_descriptors, buddies\_system. При выборе global\_heap размер и метод выделения указывать не нужно, в остальных случаях нужно указать размер и 1 из 3 методов – first\_match, the\_best\_match, the\_worst\_match.

**Демонстрация работы приложения**

Для начала создадим и удалим пул с названием pool\_1 с аллокатором с освобождением в рассортированном списке размером 1000000 и методом выделения – первого подходящего.

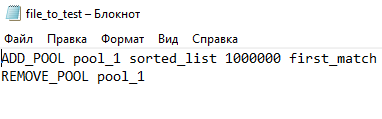


Рисунок 8. Содержимое файла для чтения команд

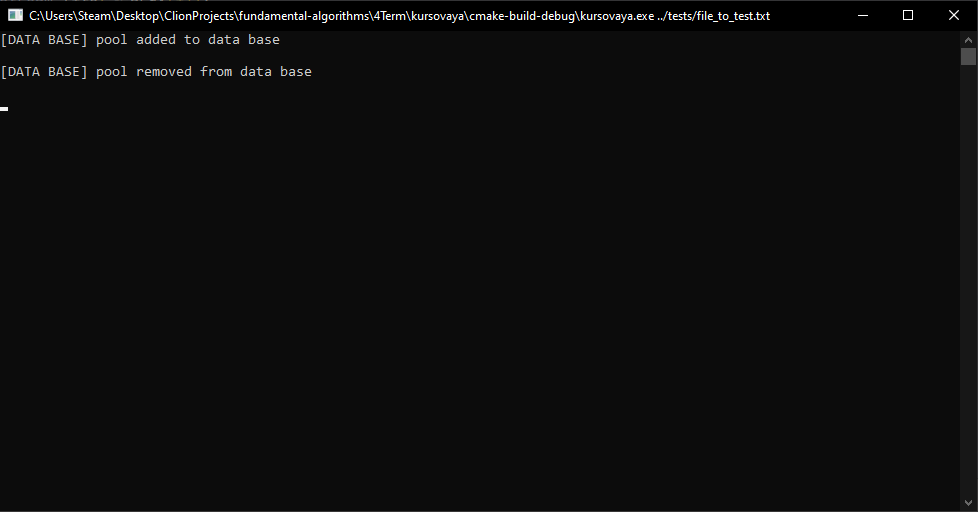


Рисунок 9. Вывод результата работы в консоль

Теперь создадим такой же пул добавим в него схему, в неё добавим коллекцию.

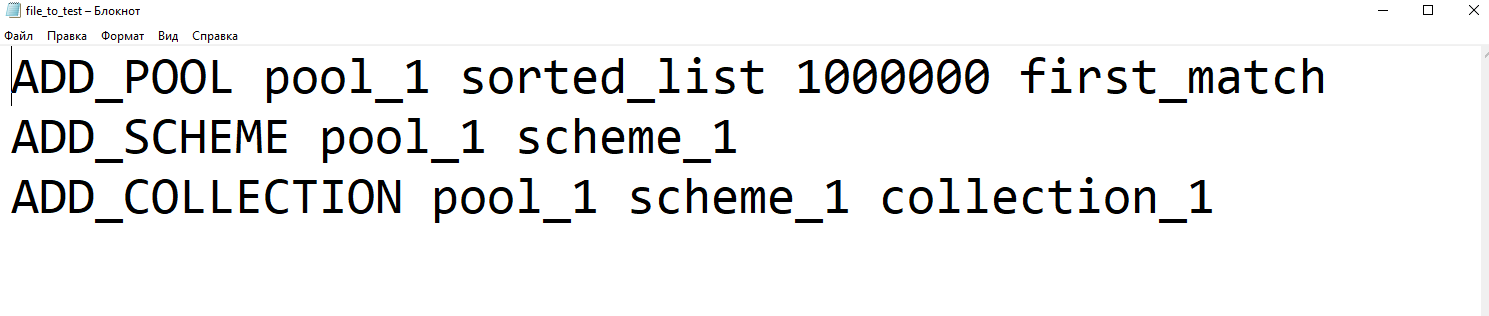


Рисунок 10. Содержимое файла для чтения команд

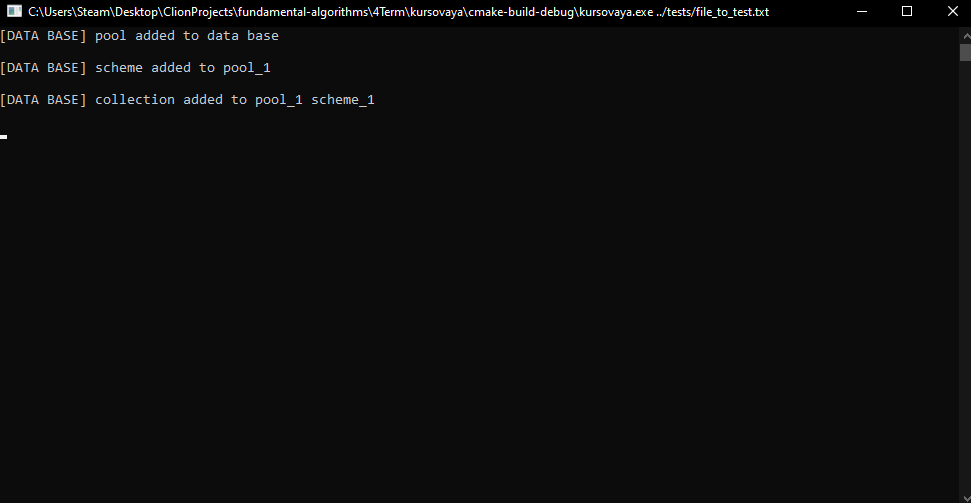


Рисунок 11. Вывод результата работы в консоль

Проверим что произойдёт, если мы попытаемся получить данные, неверно указав имя пула.

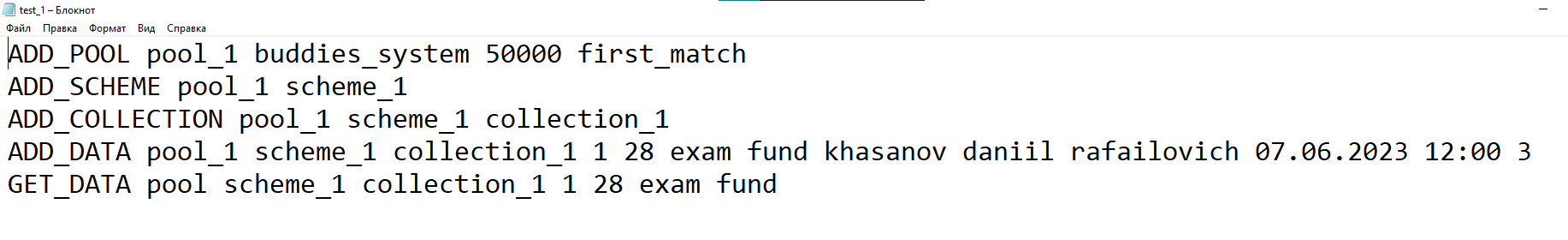


Рисунок 12. Содержимое файла для чтения команд

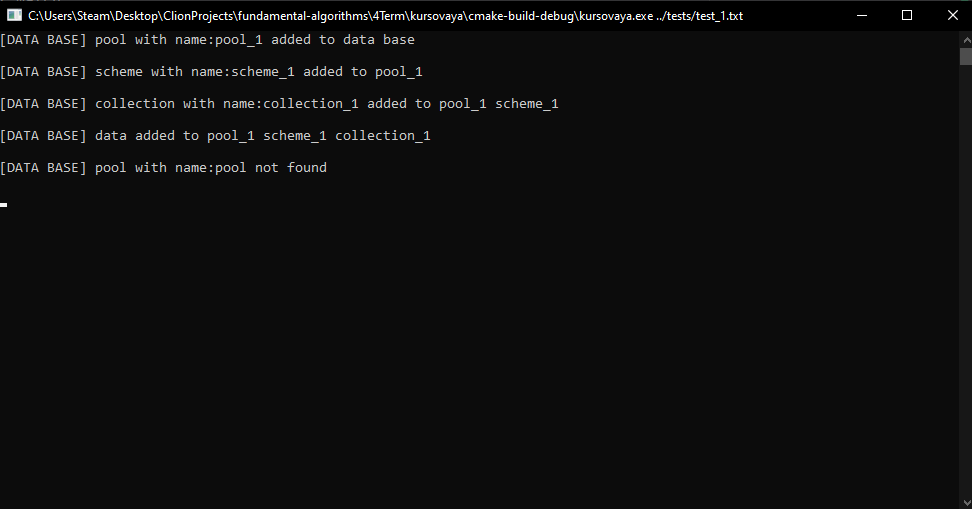


Рисунок 13. Вывод результата работы в консоль

Создадим несколько пулов с разными аллокаторами, добавим в ним схемы, коллекции и данные и прочитаем их.

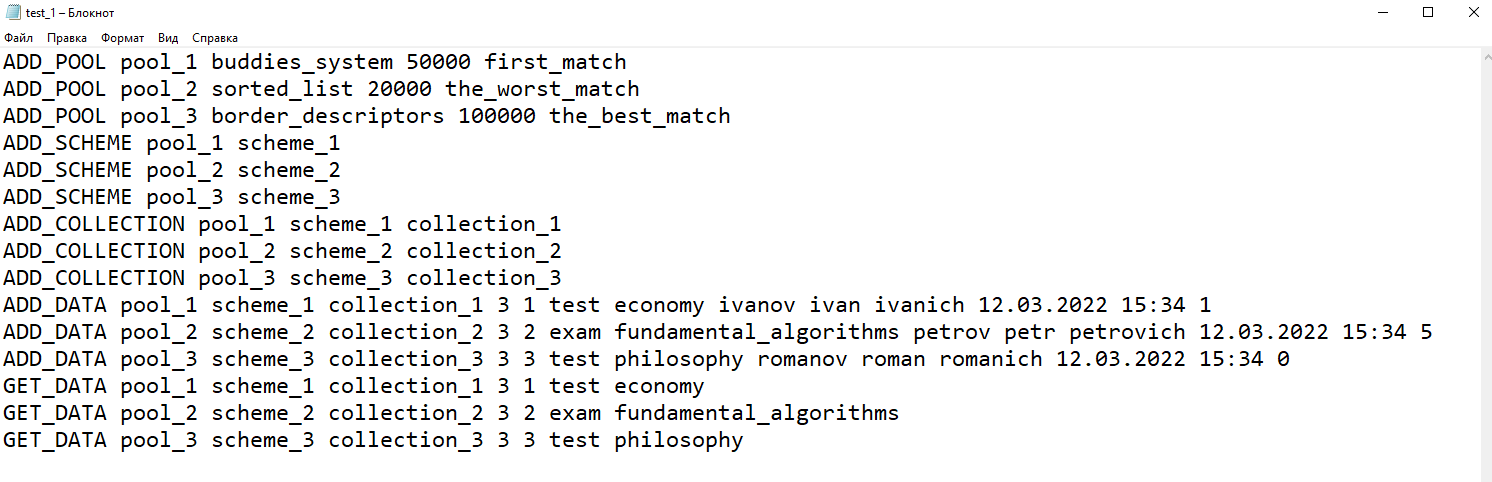


Рисунок 14. Содержимое файла для чтения команд

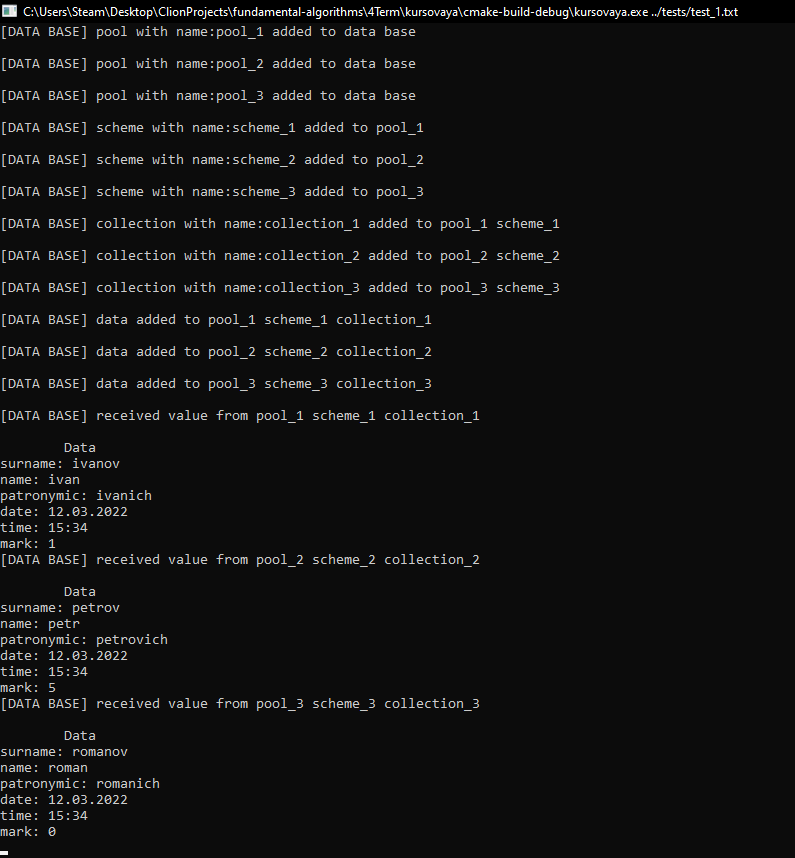


Рисунок 15. Вывод результата работы в консоль

Напоследок, добавим несколько данным в одну коллекцию, выведем данные, обновим данные по ключу и снова выведем значения.

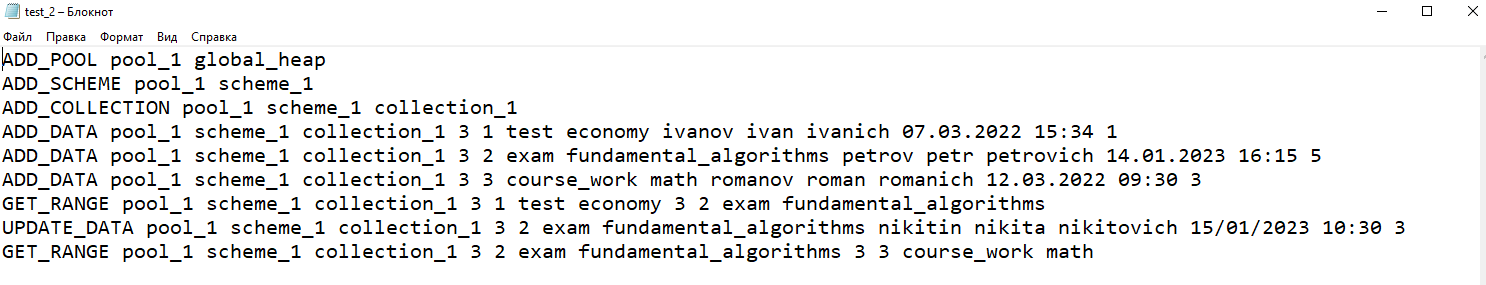


Рисунок 16. Содержимое файла для чтения команд

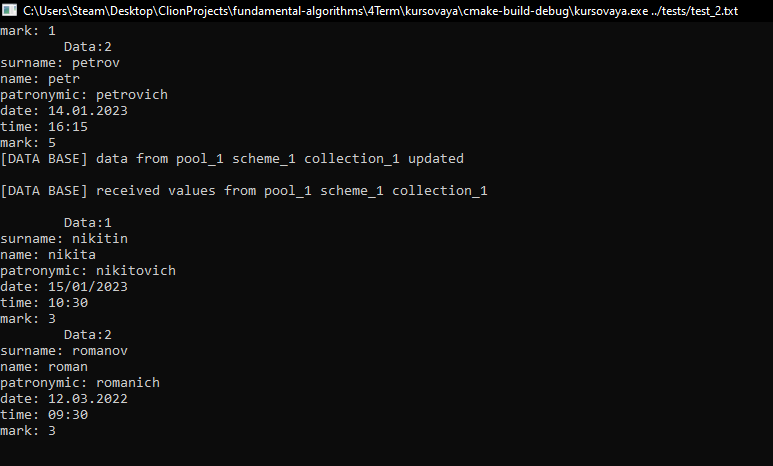


Рисунок 17. Вывод результата работы в консоль

**Вывод**

В данной курсовой реализовано приложение, позволяющее выполнять операции над коллекциями данных заданного типа и контекстами их хранения. В итоге получился аналог базы данных, которая хранит пулы схем с коллекциями данных. Она способна обрабатывать и выполнять запросы, которые подаются ей из файла.

Использование косого дерева в качестве контейнера для базы данных очень эффективно, так как часто будет осуществляется доступ к одним и тем же данным и благодаря его свойствам, все операции будут осуществляться за O(log(n)).

Реализация паттерна – цепочка обязанностей сильно облегчила обработку команд, так как за каждую команду отвечают разные объекты.

Использование собственных аллокаторов даёт возможность выбирать какой метод выделения использовать в той или иной ситуации.

Самым неэффективным оказался аллокатор с освобождением в рассортированном списке. Сложность алгоритмов выделения и освобождения в нём O(n), что не является хорошим показателем.

Самым же эффективным является аллокатор с алгоритмом освобождения с дескрипторами границ. Его плюсом является освобождение памяти за время O(1).

**Приложение**

Всю реализацию созданного мной приложения можно найти на github https://github.com/ReverseTM/fundamental-algorithms/tree/main/4Term/kursovaya

**Список используемых источников**

1. Дональд Эрвин Кнут. Искусство программирования том1. 2019, Компьютерные науки, 978-5-907144-23-1.
2. Э.Гамма, Р.Хелм, Р.Джонсон, Дж. Влиссидес. Паттерны объектно-ориентированного программирования. 2020, Питер. Серия Библиотека программиста, 978-5-4461-1595-2.