Российская федерация

Ханты-Мансийский автономный округ – Югра

Департамент образования и науки

Сургутский государственный университет ХМАО

Политехнический институт

Кафедра автоматики и компьютерных систем

**Пояснительная записка**

К курсовому проекту

по дисциплине «Объектно-ориентированное программирование»

Выполнил: Бельтюков М. О.

студент группы 609-22

Проверил: Карамышев Э.Р.

Ассистент кафедры автоматики и компьютерных систем

Сургут

2024 г.

# Задание

Вариант 14 – Дерево. Требуется реализовать класс для работы с произвольными деревьями.

# Аннотация

## Цель и задачи

Необходимо реализовать программу на языке программирования “C++”, благодаря которой можно будет взаимодействовать с произвольным деревом. Таким образом можно выделить следующие задачи:

1. Вставка и удаление узла с заданным ключом
2. Обход всех узлов дерева, поиск узла по ключу, доступ к родительскому и дочерним узлам для заданного узла
3. Сравнение двух деревьев без учета ключей (только структура) и с учетом значений ключей
4. Выделение ветви как отдельного дерева и включение дерева как ветви в другое дерево
5. Определение ширины и высоты дерева, количества узлов в дереве, максимального количество потомков одного узла
6. Одинарные и двойные повороты влево и вправо, произвольная перестановка узлов и ветвей
7. Сохранение дерева в файл и загрузка из файла

## Результаты

Реализован класс для работы с AVL-деревом.

Программа выполняет операции с AVL-деревом, хранит файлы сохранения и позволяет загрузить данные из файла.

# Оглавление

[Задание 2](#_Toc156177865)

[Аннотация 3](#_Toc156177866)

[Цель и задачи 3](#_Toc156177867)

[Результаты 3](#_Toc156177868)

[Введение 6](#_Toc156177869)

[1 Анализ 7](#_Toc156177870)

[1.1 Обзор аналогов 8](#_Toc156177871)

[1.1.1 Аналог 1 - Декартово дерево 8](#_Toc156177872)

[1.1.2 Аналог 2 – АВЛ дерево 9](#_Toc156177873)

[1.1.3 Аналог 3 – красно-чёрное дерево 11](#_Toc156177874)

[1.2 Обобщение частной задачи 13](#_Toc156177875)

[2 Проектирование 15](#_Toc156177876)

[2.1 Формальное описание задачи 15](#_Toc156177877)

[2.2 Поведенческая модель программы 15](#_Toc156177878)

[2.3 Структура файлов 16](#_Toc156177879)

[2.4 Представление данных в памяти при их обработке 17](#_Toc156177880)

[2.5 Основные алгоритмы 18](#_Toc156177881)

[2.5.1 Добавление узла 19](#_Toc156177882)

[2.5.2 Удаление узла 19](#_Toc156177883)

[2.5.3 Вывод дерева 20](#_Toc156177884)

[2.5.4 Доступ к узлу по ключу 21](#_Toc156177885)

[2.5.5 Сложение деревьев 21](#_Toc156177886)

[2.5.6 Сравнение деревьев с учётом ключей 21](#_Toc156177887)

[2.5.7 Сравнение деревьев без учёта ключей 21](#_Toc156177888)

[2.5.8 Определение высоты. 22](#_Toc156177889)

[3 Разработка программного продукта 23](#_Toc156177890)

[3.1 Методы класса дерево 23](#_Toc156177891)

[3.1.1 static методы, реализованные вне класса дерево 23](#_Toc156177892)

[3.1.2 non-static методы, реализованные вне класса дерево 24](#_Toc156177893)

[3.1.2 inline методы класса дерево 31](#_Toc156177894)

[3.2 Методы класса узел 31](#_Toc156177895)

[3.2.1 Методы, реализованные вне класса узел 31](#_Toc156177896)

[3.2.2 inline методы класса узел 32](#_Toc156177897)

[4 Тестирование 35](#_Toc156177898)

[4.1 Проверка работы меню 35](#_Toc156177899)

[4.1 Проверка работы загрузки и остальных методов 35](#_Toc156177900)

[Заключение 38](#_Toc156177901)

[5 Источники 39](#_Toc156177902)

[6 Приложения 40](#_Toc156177903)

[Приложения А. Листинги 40](#_Toc156177904)

[Приложения Б. Рисунки 47](#_Toc156177905)

# Введение

Древовидная структура данных — это иерархическая структура, которая используется для представления и организации данных в удобном для навигации и поиска виде. Она представляет собой набор узлов, соединенных ребрами, и имеет иерархические отношения между узлами.

Самый верхний узел дерева называется корнем, а узлы, расположенные под ним, - дочерними узлами. Каждый узел может иметь несколько дочерних узлов, а эти дочерние узлы могут иметь свои собственные дочерние узлы, образуя рекурсивную структуру.

Эта структура данных представляет собой специализированный метод организации и хранения данных в компьютере для их более эффективного использования. Она состоит из центрального узла, структурных узлов и подузлов, которые соединены ребрами. Можно также сказать, что древовидная структура данных имеет корни, ветви и листья, связанные между собой. [1]

Древовидная структура данных применяется:

* Для организации и управления файлами и каталогами в файловой системе. Каждый файл и каталог представлен в виде узла дерева, а отношения "родитель-ребенок" указывают на иерархическую структуру файловой системы.
* В сервере типа DNS (сервер доменных имен)
* В алгоритмах машинного обучения
* В компьютерной графике
* Для индексации в базе данных. [2]

# 1 Анализ

Свойства древовидной структуры данных:

* Количество ребер — ребро можно определить как связь между двумя узлами. Если дерево имеет N узлов, то оно будет иметь (N-1) ребер. От каждого узла к любому другому узлу дерева ведет только один путь.
* Глубина узла — глубина узла определяется как длина пути от корня до этого узла. Каждое ребро добавляет к пути 1 единицу длины. Таким образом, ее можно также определить как количество ребер на пути от корня дерева до узла.
* Высота узла — высота узла может быть определена как длина самого длинного пути от узла до листового узла дерева.
* Высота дерева — это длина самого длинного пути от корня дерева до листового узла дерева.
* Степенью узла называется общее количество поддеревьев, присоединенных к данному узлу. Степень листового узла должна быть равна 0. Степень дерева — это максимальная степень узла среди всех узлов дерева.

Преимущества древовидной структуры данных:

* Деревья обеспечивают эффективный поиск в зависимости от типа дерева: среднее время поиска составляет O(log n) для сбалансированных деревьев типа AVL.
* Деревья обеспечивают иерархическое представление данных, что облегчает организацию и навигацию по большим объемам информации.
* Рекурсивная природа деревьев делает их удобными для обхода и манипулирования с помощью рекурсивных алгоритмов.

Недостатки древовидной структуры данных:

* Несбалансированные деревья, то есть высота дерева смещена в одну сторону, что может привести к неэффективному поиску.
* Деревья требуют больше места в памяти, чем некоторые другие структуры данных, такие как массивы и связные списки, особенно если дерево очень большое. [1]

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.

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. [3]

Далее будут рассмотрены 3 вида бинарных деревьев поиска, из которых будет выбрана одна структура как курсовой проект.

## 1.1 Обзор аналогов

### Аналог 1 - Декартово дерево

Декартово дерево определяется и строится так:

* Нанесём на плоскость набор из n точек. Их координату “x” назовём ключом – значение узла дерева, а координату “y” приоритетом – значение, которое задаётся для каждого узла случайным образом.
* Выберем самую верхнюю точку (с наибольшим y, а если таких несколько — любую) и назовём её корнем.
* От всех вершин, лежащих слева (с меньшим x) от корня, рекурсивно запустим этот же процесс. Если слева была хоть одна вершина, то присоединим корень левой части в качестве левого сына текущего корня.
* Аналогично, запустимся от правой части и добавим корню правого сына.

Заметим, что если все y и x различны, то дерево строится однозначно.

Если нарисовать получившуюся структуру на плоскости, то получится действительно дерево — по традиции, корнем вверх: [4]

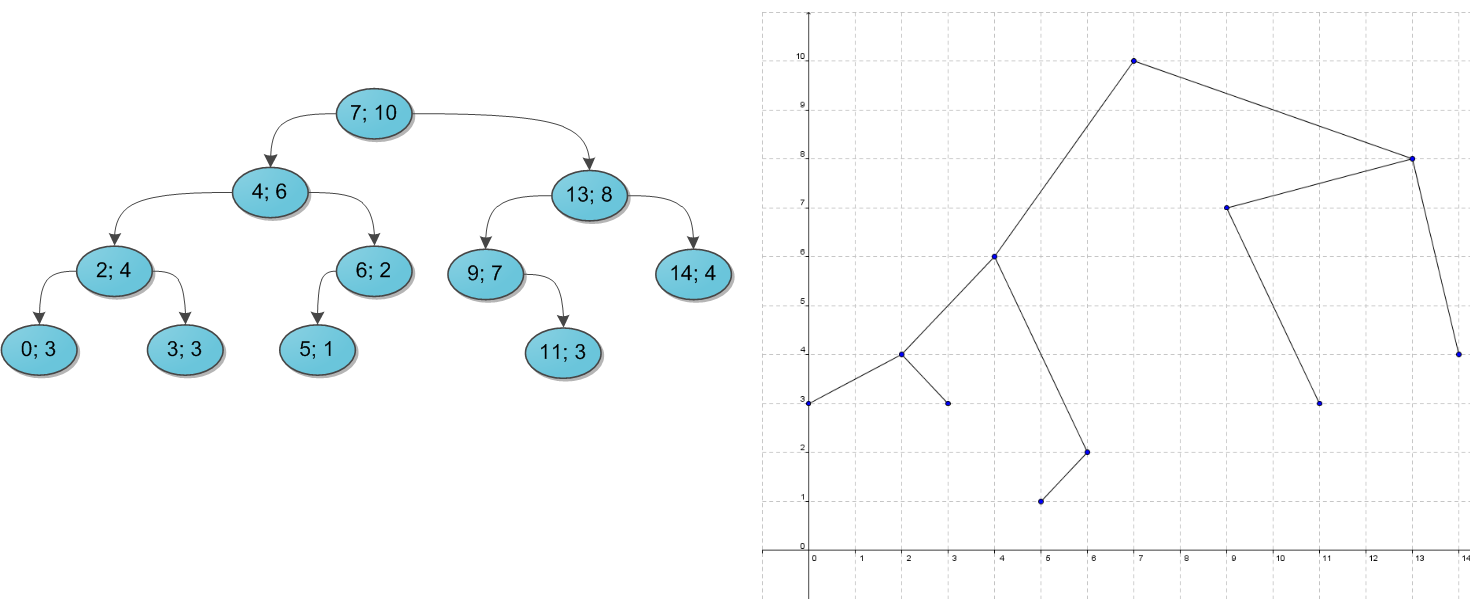


Рисунок 1 – Построение декартова дерева

С небольшими модификациями, декартово дерево умеет всё то же, что и любое бинарное дерево поиска, например:

* Добавить число x в множество;
* Определить, есть ли в множестве число x;
* Найти первое число, не меньшее x — аналогично lower\_bound;
* Найти количество чисел в промежутке [l,r][l,r].

Как и во всех сбалансированных деревьях поиска, все операции работают за высоту дерева: O(log n). [4]

### Аналог 2 – АВЛ дерево

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

Используются 4 типа вращений:

* Малое левое вращение. Данное вращение используется тогда, когда разница высот L-поддерева и b-поддерева равна 2 и высота С <= высоты R.



Рисунок 2 – Малое левое вращение АВЛ дерева

* Большое левое вращение. Данное вращение используется тогда, когда разница высот L-поддерева и b-поддерева равна 2 и

высота c-поддерева > высоты R.



Рисунок 3 – Большое левое вращение АВЛ дерева

* Малое правое вращение. Данное вращение используется тогда, когда разница высот R-поддерева и b-поддерева равна 2 и

высота С <= высоты L.



Рисунок 4 – Малое правое вращение АВЛ дерева

* Большое правое вращение Данное вращение используется тогда, когда разница высот R-поддерева и b-поддерева равна 2 и

высота c-поддерева > высоты L.



Рисунок 5 – Большое правое вращение АВЛ дерева

[5]

### 1.1.3 Аналог 3 – красно-чёрное дерево

Красно-чёрное дерево (англ. red-black tree, RB tree) — один из видов самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

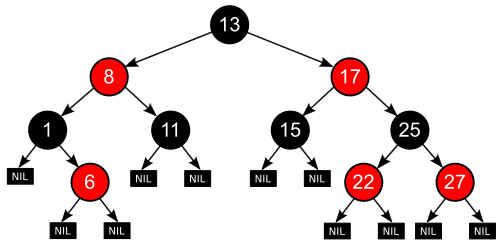


Рисунок 6 – пример красно-чёрного дерева

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвета. При этом:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этим ограничениям путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям в худшем случае быть более эффективными, чем обычные двоичные деревья поиска.

Чтобы понять, как это работает, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева T число чёрных узлов от корня до листа равно B. Тогда кратчайший возможный путь до любого листа содержит B узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря п.4 в дереве не может быть двух красных узлов подряд, а согласно пп. 2 и 3, путь начинается и кончается чёрным узлом. Поэтому самый длинный возможный путь состоит из 2B-1 узлов, попеременно красных и чёрных.

Если разрешить нелистовому узлу иметь меньше двух потомков, а листовым — содержать данные, дерево сохраняет основные свойства, но алгоритмы работы с ним усложнятся. Поэтому в статье рассматриваются только «фиктивные листовые узлы», которые не содержат данных и просто служат для указания, где дерево заканчивается. Эти узлы могут быть опущены в некоторых иллюстрациях. Из п.5, также следует, что потомками красного узла могут быть либо два чёрных промежуточных узла, либо два чёрных листа, а с учётом п.3 и 4 — что если у чёрного узла один из потомков — листовой узел, то вторым должен быть либо тоже листовой, либо вышеописанная конструкция из одного красного и двух листовых. [6]

## 1.2 Обобщение частной задачи

После рассмотрения аналогов было выбрано АВЛ-дерево.

Написать класс для работы с АВЛ-деревом

Задачи:

* Возможность сохранять деревья в разные файлы и загружать их путём диалога с пользователем с помощью меню.
* Интерактивный интерфейс с помощью меню. В меню можно заполнить параметры дерева и сохранить данные параметры в файл сохранения. Также можно загрузить из файла параметры дерева, сохранить параметры в другой файл, который не будет подтягивать основная программа.
* Реализацию нужно выполнить под Windows с использованием языка программирования “c++” и IDE “dev cpp”
* Присваивание деревьев
* Вставка и удаление узла с заданным ключом
* Обход всех узлов дерева, поиск узла по ключу, доступ к родительскому и дочерним узлам для заданного узла
* Сравнение двух деревьев без учета ключей (только структура) и с учетом значений ключей
* Сложение деревьев
* Определение высоты дерева, количества узлов в дереве
* Сохранение дерева в файл и загрузка из файла

# 2 Проектирование

## 2.1 Формальное описание задачи

Программа по входным параметрам должна реализовывать АВЛ-дерево. Также программа должна хранить несколько файлов с входными параметрами.

## 2.2 Поведенческая модель программы

Диалог с пользователем происходит при помощи файла course\_work.exe – Файл открывает консоль и предлагаются варианты действий на выбор.

Меню реализовано в 2 этапа – 1й этап для выбора дерева, 2й этап для работы с деревом.

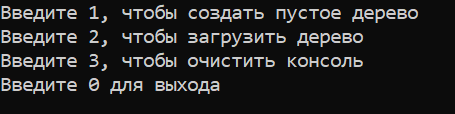


Рисунок 7 – первый этап меню

Далее после нажатия цифры “1” Откроется меню для редактирования дерева

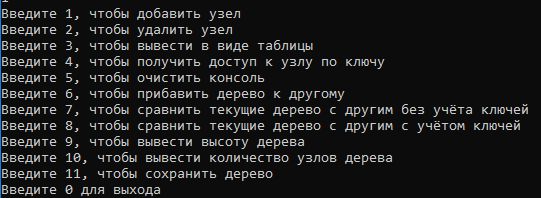


Рисунок 8 – второй этап меню

После нажатия цифры “2” программа запросит ввод названия файла, из которого будет загружено дерево

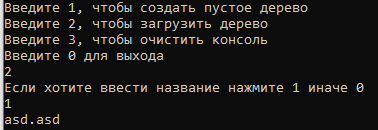


Рисунок 9 – запрос на ввод названия для загрузки

В случае ввода цифры “0” дерево сохраниться в файл save.txt, который как и все сохранения находится в папке “save”

## 2.3 Структура файлов

Сохранение и загрузка файлов происходит через папку “saves” с любым расширением. Дерево сохраняется как массив чисел через пробел, из-за чего пользователь может ввести данные в файл внутри папки “saves” и загрузить по названию файла. Также пользователь может просто ввести нужные данные в файл “save.txt” – стандартный файл сохранения, из которого по умолчанию загружается дерево.

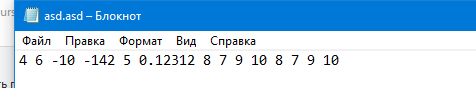


Рисунок 10 – Пример заполнения файла для загрузки дерева

## 2.4 Представление данных в памяти при их обработке

Структура классов для работы с АВЛ деревьями.

1. class Node{
2. private:
3. char diff;
4. char height;
5. bin\_tree \*owner;
6. double key;
7. public:
8. Node \*left;
9. Node \*right;
10. Node \* parent;
11. bool visited;
12. Node(double key){
13. left = NULL; right = NULL; visited = 0; parent = NULL;
14. this->key = key; owner = NULL; diff = 0; height = 1;
15. }
17. Node(double key, bin\_tree \* owner){
18. left = NULL; parent = NULL;
19. right = NULL; visited = 0; this->key = key;
20. this->owner = owner; diff = 0; height = 1;
21. }
22. ~Node();
23. }

Листинг 1 – класс узел дерева

Для класса узел переменные left и right обязательны, без них невозможно построить структуру данных “Двоичное дерево поиска”.

Переменная diff отвечает за разницу двух поддеревьев узла. Является приватной, т.к. величина пересчитывается при добавлении или удалении узла и, следовательно, её нельзя изменять. Тип char был выбран т.к. максимально возможная разница = 2, а минимально возможная = -2.

Переменная height также, как и diff пересчитывается после добавления или удаления узла. Тип char был выбран т.к. высота h < 1.44 log2(n + 2) и не сложно посчитать, что при h = 128 количество элементов будет приблизительно = 5.73e+26. Приблизительный вес объекта класса 71 байт. Таким образом чтобы превысить предел типа char необходимо приблизительно 22332214687140 петабайт памяти. В данный момент компьютеров с таким объёмом памяти не существует.

Переменная owner отвечает за привязку узла к дереву.

Переменная key хранит в себе значение узла. Тип double был выбран для возможности использовать дробные числа.

Переменная parent хранит в себе указатель на узел предок.

Переменная visited является вспомогательной для метода обхода в глубину – в случае если в узел уже “зашёл” алгоритм visited меняется на true. Далее после прохода все узлы “обнуляются” т.е. visited становится = false.

Реализованы 2 конструктора класса – с указанием только ключа и с указанием ключа и владельца. Создание узла без ключа не несёт смысла, поэтому нет конструктора, который будет создавать узел без ключа.

1. class bin\_tree {
2. public:
3. Node \*root;
4. int count;
5. bin\_tree() {root = NULL; count = 0; }
6. bin\_tree(Node \* rt) : root(rt), count(1){}
7. ~bin\_tree(){};
8. };

Листинг 2 – класс двоичное дерево

Переменная root отвечает за указание корня дерева.

Переменная count хранит в себе информацию о количестве узлов в дереве.

Реализованы 2 конструктора – один создаёт пустое дерево без корня, другой же создаёт дерево с корнем.

## 2.5 Основные алгоритмы

### 2.5.1 Добавление узла

Вставка в АВЛ дерево происходит в 2 этапа:

* Вставка узла в двоичное дерево поиска
* Балансировка АВЛ-дерева

Вставка реализуется по следующему алгоритму:

* Вызываем функцию в корне
* Если значение, которое надо вставить, меньше значения узла, где мы находимся, идем налево — вызываем ту же функцию для левого потомка;
* Если значение равно или больше тому, где мы сейчас, идем направо — вызываем функцию для правого потомка;
* Если мы попали в точку, где еще нет узла, создаем там узел и помещаем туда новое значение;
* Вызываем функцию балансировки. Так как вставка рекурсивная, балансировка выполнится начиная с последнего открытого экземпляра функции вставки и заканчивая первым. [7]

Балансировка реализуется по следующему алгоритму:

* Для начала обозначим разницу высот правого и левого поддеревьев узла за diff
* Если diff = 2 и diff правого потомка узла меньше “0” необходимо совершить Большое левое вращение
* Если diff = 2 и второе условие не выполняется необходимо совершить малое левое вращение
* Если diff = -2 и diff левого потомка узла больше 0 необходимо совершить большое правое вращение
* Если diff = -2 и второе условие не выполняется необходимо совершить малое правое вращение

### 2.5.2 Удаление узла

Сначала необходимо найти удаляемый элемент, затем перейти в его правого потомка, если он есть, и найти там минимальный узел:

* По логике структуры бинарного дерева поиска у минимального узла может быть максимум один потомок — справа. Поэтому функция, «удаляющая» минимальный узел, возвращает его правого потомка — он пригодится при замене;
* Минимальный узел не удаляется безвозвратно, а подставляется на место удаляемого — опять же происходит замена ссылок;
* Слева к минимальному узлу присоединяется левый потомок удаляемого узла, справа — то, что осталось от правого потомка после вычленения минимального
* Происходит балансировка. [9]

### 2.5.3 Вывод дерева

Дерево будет выводиться в виде таблицы, где каждый столбец будет описывать узел с помощью строк, в которых будет информация о двух потомках. Узлы будут выводится в порядке, который рассчитывается с помощью алгоритма обхода графа, где каждая вершина посещается лишь раз – поиска в глубину.

Поиском в глубину или эйлеровым обходом называется рекурсивный алгоритм обхода корневого дерева или графа, начинающий в корневой вершине и рекурсивно обходящий весь граф, посещая каждую вершину ровно один раз. [8]

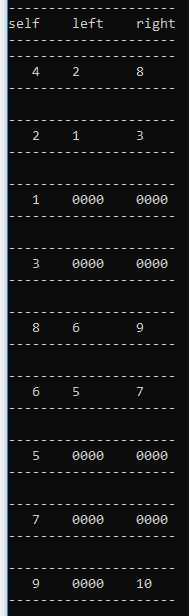


Рисунок 11 – вывод АВЛ дерева

### 2.5.4 Доступ к узлу по ключу

Для поиска элемента по ключу нам нужно выполнять следующий алгоритм:

* Ключ сравнивается с ключом в рассматриваемом узле начиная с корня дерева.
* Если заданный ключ больше ключа в узле функция вызывается для правого потомка рассматриваемого узла
* Если заданный ключ меньше ключа в узле функция вызывается для левого потомка
* Если заданный ключ равен ключу в узле, выводим свой ключ и ключи левого и правого потомков.
* Если у узла нет потомков, то в дереве нет такого ключа

### 2.5.5 Сложение деревьев

Для сложения деревьев необходимо:

* Так как деревья хранятся в файлах как массив чисел – можно не выделять память для класса дерево, а просто вставить все элементы из файла по порядку

### 2.5.6 Сравнение деревьев с учётом ключей

Сравнение будет произведено по 3 пунктам:

1. Максимальный ключ – Находится в конце правой ветки начиная с корня дерева
2. Минимальный ключ – находится в конце левой ветки начиная с корня дерева
3. Среднее значение ключей – проходимся с помощью алгоритма обхода в глубину по всем ключам сохраняя их сумму – количество автоматически подсчитывается и выдаётся за O(1)

### 2.5.7 Сравнение деревьев без учёта ключей

Для сравнения деревьев без учёта ключей необходимо:

* Загрузить дерево как ещё один объект класса “дерево”
* Сравнить количество элементов и высоту деревьев.

### 2.5.8 Определение высоты.

Для определения глубины (высоты) дерева необходимо знать высоту потомков корня дерева:

* Выбирается максимальная глубина левого и правого потомка
* К полученному значению добавляется “1”
* Глубина определяется автоматически, при добавлении узлов

# 3 Разработка программного продукта

## 3.1 Методы класса дерево

### 3.1.1 static методы, реализованные вне класса дерево

1. static void out\_menu\_tree();
2. static void out\_menu();
3. static void menu();
4. static bin\_tree \* load();

Листинг 3 – static-методы класса дерево

Метод out\_menu\_tree необходим для вывода информации о меню для отдельного дерева (рисунок 8). Является static-методам т.к. этот метод необходим только для вывода фиксированной информации, не зависящей от объекта класса. Реализация метода представлена в листинге А.1

Метод out\_menu необходим для вывода информации о полном меню (Рисунок 7). Также, как и предыдущий метод является static-методам т.к. этот метод необходим только для вывода фиксированной информации, не зависящей от объекта класса.

1. void bin\_tree::out\_menu(){
2. std::cout << "Введите 1, чтобы создать пустое дерево\n";
4. std::cout << "Введите 2, чтобы загрузить дерево\n";
6. std::cout << "Введите 3, чтобы очистить консоль\n";
8. std::cout << "Введите 0 для выхода\n";
9. }

Листинг 4 – реализация метода out\_menu

Метод load позволяет ввести путь к загружаемому дереву, далее возвращает загруженное дерево. Данный метод является static-методом т.к. реализация работы с двумя деревьями выполнена так, что в метод класса передаётся объект класса дерево, поэтому удобнее сделать загрузку как возвращаемое значение, а не как запись в пустое дерево. Реализация представлена в приложении А.2.

1. cmp\_struct(bin\_tree::load());

Листинг 5 – пример работы с методом load

### 3.1.2 non-static методы, реализованные вне класса дерево

1. void menu\_tree();
2. void dfs(Node \* a, char flag, string path="", double \* sum = NULL);
3. void dfs\_support(Node \* a);
4. void out();
5. void insert(Node \* node, double key);
6. Node \* find(Node \* node, double key);
7. Node \* remove(Node \* node, double key);
8. Node \* rotate\_left(Node \* node);
9. Node \* rotate\_right(Node \* node);
10. Node \* balance(Node \* node);
11. void save();
12. void sum\_tree();
13. Node \* find\_min(Node \* node = NULL);
14. Node \* remove\_min(Node \* node = NULL);
15. Node \* find\_max(Node \* node = NULL);
16. double find\_avg();

Листинг 6 – non-static методы, реализованные вне класса дерево

Метод menu необходим для начала работы программы.

1. void bin\_tree::menu(){
2. int menu = 1, i=0;
4. bin\_tree \* tree;
5. while (menu){
6. out\_menu();
7. input\_check(&menu);
8. switch(menu){
9. case 1: /\* пустое дерево\*/
10. tree = new bin\_tree();
11. tree->menu\_tree();
12. delete tree;
13. break;
14. case 2: /\* Загрузить дерево \*/
15. bin\_tree::load()->menu\_tree();
16. break;
18. case 3:
19. system("cls");
20. break;
21. default:
22. std::cout << "Неправильный ввод\n";
24. }
25. }
26. }

Листинг 7 – реализация метода menu

Метод menu\_tree позволяет вызвать меню для редактирования, получения информации и сравнения с другими деревьями. Реализация представлена в приложении А.3

Метод dfs необходим для операций, связанных с обходом дерева. Параметр node необходим для задания узла, с которого будет начинаться поиск, параметр flag необходим для выбора действий, которые будут производиться с каждым узлом при обходе, остальные параметры являются служебными для flag. В случае, если flag = 1 - каждый узел дерева выводиться, в случае если flag = 2 – происходит сохранение значения каждого узла, в случае,

если flag = 3 – каждый узел суммируется с переменной sum, которая изначала равна нулю. Реализация метода представлена в приложении А.4

1. if (!a->visited){
2. switch (flag){
3. case 1:
4. a->out();
5. break;
6. case 2:
7. out1 << a->get\_key() << " ";
8. break;
9. case 3:
10. \*sum += a->get\_key();
11. break;
12. }

Листинг 8 – Работа метода dfs с параметром flag

Метод dfs\_support необходим для обнуления переменных visited после работы метода dfs.

1. void bin\_tree::dfs\_support(Node \* a){
2. static int cnt = 0;
3. if (a->visited){
4. a->visited = 0;
5. cnt++;
6. }
7. if (cnt != count && a->left)
8. dfs\_support(a->left);
9. if(cnt != this->count && a->right)
10. dfs\_support(a->right);
11. }

Листинг 9 – реализация метода dfs\_support

Метод out позволяет вывести информацию о дереве - значение ключа каждого узла, левого и правого потомков, а также предка узла.

1. void bin\_tree::out(){
2. cout.width(4);
3. cout << "---------------------------------\n";
4. cout << "self\t" << "left\t" << "right\t" << "parent\n";
5. cout << "---------------------------------\n";
6. dfs(this->root, 1);
7. cout << "Количество узлов = " << count << "\n";
8. }

Листинг 10 – реализация метода out

Метод insert позволяет вставить элемент в дерево. Параметром node нужно указать корень дерева, параметр key необходим для задания значения ключа. Реализация представлена в приложении А.5

Метод find позволяет получить доступ к узлу по ключу.

1. Node \* bin\_tree::find(Node \* node, double key){
2. if (fabs(node->get\_key() - key) <= 0.000001)
3. return node;
4. else if (node->get\_key() < key){
5. if (!node->right)
6. return NULL;
7. find(node->right, key);
8. }
9. else{
10. if (!node->left)
11. return NULL;
12. find(node->left, key);
13. }
14. }

Листинг 11 – реализация метода find

Метод remove позволяет удалить ключ из дерева и вернуть удалëнный узел. Реализация представлена в приложении А.6

Методы rotate\_right, rotate\_left позволяют совершить поворот, необходимый для балансировки дерева.

1. Node \* bin\_tree::rotate\_left(Node \* node){
2. Node \* b = node->right;
3. node->right = b->left;
4. if (node->right)
5. b->left->parent = node;
6. b->left = node;
7. b->parent = node->parent;
8. b->set\_diff();
9. node->set\_diff();
10. if (node == root){
11. root = b;
12. b->parent = NULL;
13. }else if (node->parent->right == node){
14. node->parent->right = b;
15. }else if (node->parent->left == node)
16. node->parent->left = b;
17. node->parent = b;
18. node->calc\_height();
19. b->calc\_height();
20. b->set\_diff();
21. node->set\_diff();
22. return b;
23. }

Листинг 12 – реализация метода rotate\_right

1. Node \* bin\_tree::rotate\_right(Node \* node){
2. Node \* b = node->left;
3. node->left = b->right;
4. if (node->left)
5. b->right->parent = node;
6. b->right = node;
7. b->parent = node->parent;
8. b->set\_diff();
9. node->set\_diff();
10. if (node == root){
11. root = b;
12. b->parent = NULL;
13. }else if (node->parent->right == node){
14. node->parent->right = b;
15. }else if (node->parent->left == node)
16. node->parent->left = b;
17. node->parent = b;
18. node->calc\_height();
19. b->calc\_height();
20. b->set\_diff();
21. node->set\_diff();
22. return b;
23. }

Листинг 13 – реализация метода rotate\_right

Метод balance выполняет балансировку для узла node.

1. Node \* bin\_tree::balance(Node \* node){
2. node->set\_diff();
3. if (node->get\_diff() == 2){
4. node->right->set\_diff();
5. if (node->right->get\_diff() < 0)
6. node->right = rotate\_right(node);
7. return rotate\_left(node);
8. }else if (node->get\_diff() == -2){
9. node->left->set\_diff();
10. if (node->left->get\_diff() > 0)
11. node->left = rotate\_left(root);
12. return rotate\_right(node);
13. }
14. return node;
15. }

Листинг 14 – реализация метода balance

Метод save позволяет пользователь указать путь к сохранению дерева и сохранить данные в файл.

1. void bin\_tree::save(){
2. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
3. int i=0;
4. string path;
5. input\_check(&i);
6. if (i){
7. while (std::cin >> path, std::cin.fail()){
8. cin.clear();
9. cin.ignore(500, '\n');
10. cout << "Неправильный ввод\n";
11. }
12. path = "saves/" + path;
13. }else
14. path = "saves/save.txt";
15. out1.open(path);
16. if (out1.is\_open()){
17. dfs(root, 2, path);
18. out1.close();
19. }
20. }

Листинг 15 – реализация метода save

Метод sum\_tree позволяет сложить дерево с деревом, которое пользователю необходимо будет загрузить. Входных параметров нет, т.к. нет необходимости создавать новый объект класса bin\_tree.

1. void bin\_tree::sum\_tree(){
2. cout << "Загрузите дерево, с которым будет сложение\n";
3. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
4. int i=0;
5. string path;
6. input\_check(&i);
7. if (i){
8. while (std::cin >> path, std::cin.fail()){
9. cin.clear();
10. cin.ignore(500, '\n');
11. cout << "Неправильный ввод\n";
12. }
13. path = "saves/" + path;
14. }else
15. path = "saves/save.txt";
16. double key;
17. ifstream in(path);
18. if (in.is\_open()){
19. while(in >> key){
20. insert(root, key);
21. }
22. }
23. in.close();
24. }

Листинг 16 – реализация метода sum\_tree

Метод find\_min позволяет найти минимальный элемент начиная с узла node.

1. Node \* bin\_tree::find\_min(Node \* node){
2. Node \* temp;
4. if (node)
5. temp = node;
6. else
7. temp = root;
9. while (temp->left)
10. temp = temp->left;
12. return temp;
13. }

Листинг 17 – реализация метода find\_min

Метод remove\_min позволяет удалить минимальный элемент начиная с узла node.

1. Node \* bin\_tree::remove\_min(Node \* node){
2. if (node){
3. if (!node->left)
4. return node->right;
6. node->left = remove\_min(node->left);
7. return balance(node);
8. }else{
9. if (!root->left)
10. return root->right;
12. root->left = remove\_min(root->left);
13. return balance(root);
14. }
16. }

Листинг 18 – реализация метода remove\_min

Метод find\_max позволяет найти максимальный элемент начиная с узла node.

1. Node \* bin\_tree::find\_max(Node \* node){
2. Node \* temp;
4. if (node)
5. temp = node;
6. else
7. temp = root;
9. while (temp->right)
10. temp = temp->right;
12. return temp;
13. }

Листинг 19 – реализация метода find\_max

Метод find\_avg позволяет найти среднее значение ключей в дереве.

1. double bin\_tree::find\_avg(){
2. double sum;
3. dfs(root, 3, "", &sum);
4. return sum / count;
5. }

Листинг 20 – реализация метода find\_avg

### 3.1.2 Методы, реализованные в классе дерево

1. void cmp\_struct(bin\_tree \* tree){
2. cout << "Высота загруженного дерева = " << (int)(tree->root->get\_height())
3. << "\t" << "Высота исходного дерева = " << (int)(root->get\_height()) << "\n";
4. cout << "Количество узлов загруженного дерева = " << tree->count
5. << "\t" << "Количество узлов исходного дерева = " << count << "\n";
6. }
7. void cmp\_keys(bin\_tree \* tree){
8. cout << "Максимальный узел загруженного дерева = " << tree->find\_max()->get\_key()
9. << "\t" << "Максимальный узел исходного дерева = " << find\_max()->get\_key() << "\n";
10. cout << "Минимальный узел загруженного дерева = " << tree->find\_min(tree->root)->get\_key()
11. << "\t" << "Минимальный узел исходного дерева = " << find\_min(root)->get\_key() << "\n";
12. cout << "Среднее значение по узлам у загруженного дерева = " << tree->find\_avg()
13. << "\t" << "Среднее значение по узлам у исходного дерева = " << find\_avg() << "\n";
14. }

Листинг 21 – Методы, реализованные в классе дерево

Метод cmp\_struct позволяет сравнить объекта класса дерево с переданным параметром tree без учёта ключей

Метод cmp\_keys позволяет сравнить объекта класса дерево с переданным параметром tree с учёта ключей

## 3.2 Методы класса узел

### 3.2.1 Методы, реализованные вне класса узел

1. void out();

Листинг 22 – Метод, реализованный вне класса узел

Метод out позволяет вывести информация об одном узле.

1. void Node::out(){
2. cout << "---------------------------------\n";
3. cout.width(4);
4. if (!this)
5. cout << "Pointer error\n";
6. else if (this->left && this->right)
7. cout << this->get\_key() << "\t" << this->left->get\_key() << "\t" << this->right->get\_key() << "\t" << (parent ? parent->get\_key() : 0) << "\n";
8. else if (this->left)
9. cout << this->get\_key() << "\t" << this->left->get\_key() << "\t" << "0000" << "\t" << (parent ? parent->get\_key() : 0)<< "\n";
10. else if (this->right)
11. cout << this->get\_key() << "\t" << "0000" << "\t" << this->right->get\_key() << "\t" << (parent ? parent->get\_key() : 0)<< "\n";
12. else
13. cout << this->get\_key() << "\t" << "0000" << "\t" << "0000"<< "\t" << (parent ? parent->get\_key() : 0) <<"\n";
14. cout << "---------------------------------\n\n";
15. }

Листинг 23 – реализация метода out класса узел

### 3.2.2 Методы, реализованные в классе узел

Все методы, реализованные в классе узел представлены в приложении А.7

Метод get\_diff позволяет получить разницу правого и левого поддеревьев данного узла.

1. char get\_diff(){
2. if (right && left)
3. return right->height - left->height;
4. else if (left)
5. return 0 - left->height;
6. else if(right)
7. return right->height - 0;
8. else
9. return 0;
10. }

Листинг 24 – реализация метода get\_diff

Метод set\_diff позволяет рассчитать актуальную разницу правого и левого поддеревьев.

1. char set\_diff(){
2. if (right && left)
3. diff = right->height - left->height;
4. else if (left)
5. diff = 0 - left->height;
6. else if(right)
7. diff = right->height - 0;
8. else
9. diff = 0;
10. }

Листинг 25 – реализация метода set\_diff

Метод calc\_height позволяет рассчитать актуальную высоту для узла.

1. void calc\_height(){
2. if (right && left)
3. height = right->height>left->height? right->height + 1 : left->height + 1;
4. else if (left)
5. height = left->height+1;
6. else if(right)
7. height = right->height+1;
8. else
9. height = 1;
10. }

Листинг 26 – реализация метода calc\_height

Метод get\_height позволяет получить высоту узла.

1. char get\_height(){ return height; }

Листинг 27 – реализация метода get\_height

Метод get\_key позволяет получить значение ключа данного узла.

1. double get\_key(){ return key; }

Листинг 28 – реализация метода get\_height

Метод get\_owner позволяет получить адрес дерева, к которому принадлежит данный узел.

1. bin\_tree \* get\_owner(){ return owner; }

Листинг 29 – Реализация метода get\_owner

# 4 Тестирование

## 4.1 Проверка работы меню

Для начала необходимо открыть файл avl\_tree.exe.

Далее необходимо следуя инструкциям на экране создать пустое дерево, выйти и попытаться ввести неправильные значения

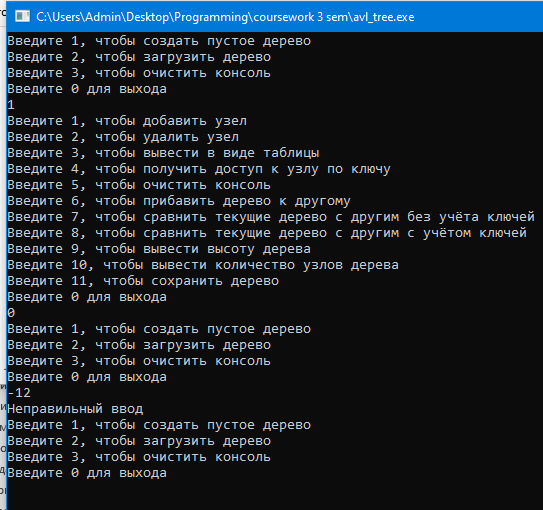


Рисунок 12 – проверка работоспособности метода menu

## 4.1 Проверка работы загрузки и остальных методов

Для начала необходимо заполнить файл save.txt.

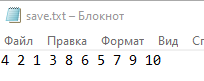


Рисунок 13 – заполнение файла save.txt

Далее необходимо создавать файлы asd.asd для тестирования возможности загрузки произвольного набора данных и large.txt для тестирования утечки памяти и возможности загрузки большого количества информации (1e7 элементов). Для заполнения фала large.txt создам ещё одну программу – test.cpp, программа будет находится в папке test.

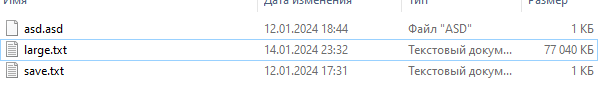


Рисунок 14 – созданные и заполненные файлы в папке saves

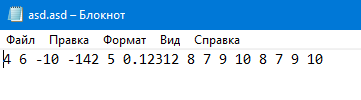


Рисунок 15 – содержимое файла asd.asd

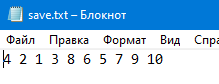


Рисунок 16 – содержимое файла save.txt

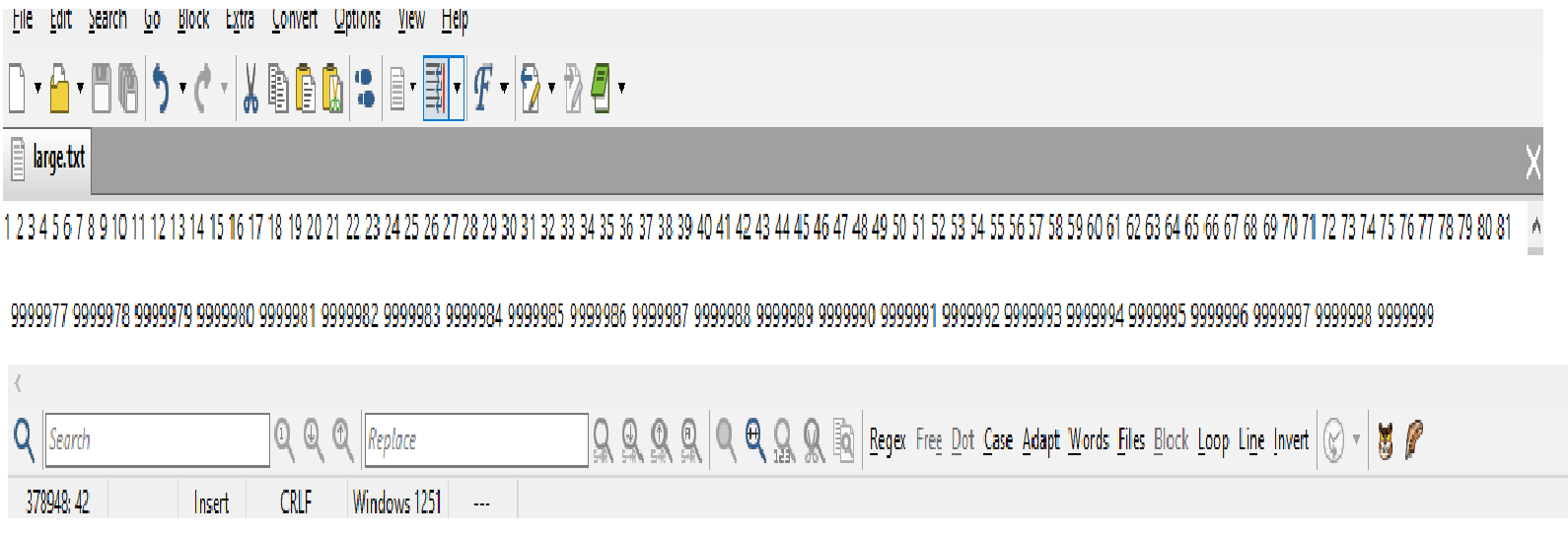


Рисунок 17 – начало и конец файла large.txt

1. #include <fstream>
2. #include <iostream>
3. using namespace std;
4. int main(){
5. ofstream out;
6. out.open("../saves/large.txt");
7. for (int i=1; i<10000000;i++){
8. out << i << " ";
9. }
10. }

Листинг 30 – программа, заполняющая файл large.txt

Далее попробуем загрузить данные из файлов asd.asd и large.txt, вывести информацию о них и тоже самое проделать с файлом save (стандартным). Процесс тестирование представлен в приложениях Б.1, Б.2, Б.3.1 и Б.3.2

Все действия, связанные с файлами asd.asd и save.txt, происходили мгновенно, но загрузка из файла large.txt заняла около минуты, при попытке вывести данные в консоль, из-за пределов консоли windows не получилось завершить операцию до конца, время ожидания работы программы примерно 30 минут, после этого было принято решение перезапустить программу. Удаление произошло, и вставка произошли моментально. Доступ к узлу по ключу и сравнение деревьев без учёта ключей произошли моментально. Но сравнение с учётом ключей не получилось и причиной тому стал алгоритм подсчёта суммы ключей, чтобы решить данную проблему необходимо пожертвовать памятью и добавить в класс переменную тип double и динамически (при добавлении или удалении узла) считать значение суммы.

## Заключение

Реализована программа на языке программирования “c++”, программа работает с файлами, отвечая за сохранение и загрузку данных из файлов. Программа позволяет просматривать и редактировать информацию о выбранной структуре данных.

Программа позволяет сравнивать и взаимодействовать с другим объектом класса.

Программу можно изменить – класс станет больше весить, но будет получена возможность получить среднее значение по ключам.

# 5 Источники

1. GeeksforGeeks, Introduction to Tree – Data Structure and Algorithm Tutorials (Введение в деревья – структура данных и алгоритмы) - https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/

2. GeeksforGeeks, Applications of tree data structure (Применения древовидной структуры данных) - <https://www.geeksforgeeks.org/applications-of-tree-data-structure/>

3. Хабр, Бинарные деревья поиска и рекурсия – это просто –

<https://habr.com/ru/articles/267855/>

4. Алгортмика - Декартово дерево -

<https://ru.algorithmica.org/cs/tree-structures/treap/>

5. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ = Introduction to algorithms. — 2-е изд.

6. *Вирт Н.* Алгоритмы и структуры данных. — М.: Мир, 1989

7. skillfactory – АВЛ-дерево -

<https://blog.skillfactory.ru/glossary/avl-derevo/>

8. Алгоритмика – Поиск в глубину -

<https://ru.algorithmica.org/cs/graph-traversals/dfs/>

9. Хабр – АВЛ деревья –

https://habr.com/ru/articles/150732/

# 6 Приложения

## Приложения А. Листинги

1. void bin\_tree::out\_menu\_tree(){
2. std::cout << "Введите 1, чтобы добавить узел\n";
4. std::cout << "Введите 2, чтобы удалить узел\n";// Вовремя теста Insert-a всё сломалось
6. std::cout << "Введите 3, чтобы вывести в виде таблицы\n";
8. std::cout << "Введите 4, чтобы получить доступ к узлу по ключу\n"; // Выводится узел и его дочерние узлы и его родительский ключ
10. std::cout << "Введите 5, чтобы очистить консоль\n";
12. std::cout << "Введите 6, чтобы прибавить дерево к другому\n";// изи - загрузить дерево как массив и просто вставить все элементы
14. std::cout << "Введите 7, чтобы сравнить текущие дерево с другим без учёта ключей\n";// Нужна глубина
16. std::cout << "Введите 8, чтобы сравнить текущие дерево с другим с учётом ключей\n";// сначала переделать удаление
18. std::cout << "Введите 9, чтобы вывести высоту дерева\n"; // Ширина - максимальное кол-во узлов одинаковой высоты (можно обновлять при insert)
20. std::cout << "Введите 10, чтобы вывести количество узлов дерева\n";
22. std::cout << "Введите 11, чтобы сохранить дерево\n";
24. std::cout << "Введите 0 для выхода\n";
25. }

Листинг А.1 – реализация метода out\_menu

1. bin\_tree \* bin\_tree::load(){
2. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
3. int i=0;
4. string path;
5. input\_check(&i);
6. if (i){
8. while (std::cin >> path, std::cin.fail()){
9. cin.clear();
10. cin.ignore(500, '\n');
11. cout << "Неправильный ввод\n";
12. }

15. path = "saves/" + path;
16. }else{
17. path = "saves/save.txt";
18. }
20. bin\_tree \* tree = NULL;
21. tree = new bin\_tree();
22. double key;
23. ifstream in(path);
24. if (in.is\_open()){
26. while(in >> key){
27. tree->insert(tree->root, key);
28. }
30. }
31. in.close();
32. return tree;
33. }

Листинг А.2 – реализация метода load

1. void bin\_tree::menu\_tree(){
2. int i=0;
3. int menu = 1;
4. double key = 0;
5. Node \* node = NULL;
7. while (menu){
8. out\_menu\_tree();
10. input\_check(&menu);
12. switch(menu){
13. case 1: /\* Добавить узел\*/
14. std::cout << "Введите ключ\n";
15. input\_check(&key);
16. this->insert(this->root, key);
17. break;
18. case 2: /\* Удалить узел \*/
19. std::cout << "Введите ключ по которому хотите удалить узел\n";
20. input\_check(&key);
21. this->remove(root, key);
22. break;
23. case 3: /\* Вывод\*/
24. this->out();
25. break;
26. case 4: /\* Узел по ключу \*/
27. std::cout << "Введите ключ\n";
28. input\_check(&key);
29. std::cout << "---------------------\n";
30. std::cout << "self\t" << "left\t" << "right\n";
31. std::cout << "---------------------\n";
32. this->find(this->root, key)->out();
33. break;
34. case 5:
35. system("cls");
36. break;
37. case 6:
38. sum\_tree();
39. break;
40. case 7:
41. cmp\_struct(bin\_tree::load());
42. break;
43. case 8:
44. cmp\_keys(bin\_tree::load());
45. break;
46. case 9:
47. std::cout << "Высота = " << (int)this->root->get\_height() << "\n";
48. break;
49. case 10:
50. std::cout << "В дереве "<< count << " узлов\n";
51. break;
52. case 11:
53. this->save();
54. break;
55. case 0:
56. break;
57. default:
58. std::cout << find\_avg();
59. std::cout << "Неправильный ввод\n";
60. break;
61. }
62. }
63. }

Листинг А.3 – реализация метода menu\_tree

1. void bin\_tree::dfs(Node \* a, char flag, string path, double \* sum){
2. if (a){
3. dfs\_support(root);
4. static int cnt = 0;
5. if (!a->visited){
6. switch (flag){
7. case 1:
8. a->out();
9. break;
10. case 2:
11. out1 << a->get\_key() << " ";
12. break;
13. case 3:
14. \*sum += a->get\_key();
15. break;
16. }
18. a->visited = 1;
19. cnt += 1;
20. }
21. if (cnt != this->count && a->left)
22. dfs(a->left, flag, path, sum);
23. if(cnt != this->count && a->right)
24. dfs(a->right, flag, path, sum);
25. dfs\_support(root);

28. }
29. }

Листинг А.4 – Реализация метода dfs

1. void bin\_tree::insert(Node \* node, double key){
2. if (node){
3. Node \* temp = node;
4. if (fabs(node->get\_key() - key) <= 0.00001){
5. return;
6. }else if (node->get\_key() < key){
7. if (node->right){
8. temp = node;
9. insert(node->right, key);
10. node->calc\_height();
11. this->balance(node);
12. }
14. else{
15. Node \* new\_node = new Node(key, this);
16. new\_node->parent = temp;
17. node->right = new\_node;
18. count++; // + ýëåìåíò
19. node->calc\_height();
20. return;
21. }
22. }else{
23. if (node->left){
24. temp = node;
25. insert(node->left, key);
26. node->calc\_height();
27. this->balance(node);
28. }
29. else{
30. Node \* new\_node = new Node(key, this);
31. new\_node->parent = temp;
32. node->left = new\_node;
33. count++; // + ýëåìåíò
34. node->calc\_height();
35. return;
36. }
37. }
38. }else{
39. root = new Node(key);
40. count++;
41. return;
42. }
43. }

Листинг А.5 – Реализация метода insert

1. Node \* bin\_tree::remove(Node \* node, double key){
2. if (!node) return 0;
3. if (key < node->get\_key())
4. node->left = remove(node->left, key);
5. else if (key > node->get\_key()){
6. node->right = remove(node->right, key);
7. }else if (fabs(node->get\_key() - key) < 0.0001 && node != root){
8. Node \* l = node->left;
9. Node \* r = node->right;
10. if (l)
11. l->parent = node->parent;
12. if (r)
13. r->parent = node->parent;
14. delete node;
15. count--;
16. if(l)
17. l->calc\_height();
18. if(r)
19. r->calc\_height();
21. if (!r) return l;
22. Node \* min = find\_min(r);
23. min->right = remove\_min(r);
24. min->left = l;
25. min->calc\_height();
26. return balance(min);
27. }else if (fabs(node->get\_key() - key) < 0.0001){
28. Node \* l = node->left;
29. Node \* r = node->right;
30. if (!l && !r){
31. count--;
32. delete root;
33. root = NULL;
34. return NULL;
35. }
36. if (l)
37. l->parent = node->parent;
38. if (r)
39. r->parent = node->parent;
41. delete node;
42. count--;
43. if(l)
44. l->calc\_height();
45. else if(r)
46. r->calc\_height();
47. else if(l && r)
48. l->calc\_height(), r->calc\_height();
50. Node \* min = find\_min(r);
51. min->right = remove\_min(r);
52. min->left = l;
53. root = min;
55. if (l){
56. min->left->calc\_height();
57. }if (min->right){
58. min->right->calc\_height();
59. }
60. min->calc\_height();
61. return balance(min);
62. }
63. return balance(node);
64. }

Листинг А.6 – Реализация метода remove

1. char get\_diff(){
2. if (right && left)
3. return right->height - left->height;
4. else if (left)
5. return 0 - left->height;
6. else if(right)
7. return right->height - 0;
8. else
9. return 0;
10. }
12. char set\_diff(){
13. if (right && left)
14. diff = right->height - left->height;
15. else if (left)
16. diff = 0 - left->height;
17. else if(right)
18. diff = right->height - 0;
19. else
20. diff = 0;
21. }
23. void calc\_height(){
24. if (right && left)
25. height = right->height>left->height? right->height + 1 : left->height + 1;
26. else if (left)
27. height = left->height+1;
28. else if(right)
29. height = right->height+1;
30. else
31. height = 1;
32. }
33. char get\_height(){ return height; }
34. double get\_key(){ return key; }
35. bin\_tree \* get\_owner(){ return owner; }

Листинг А.7 – Методы, реализованные в классе узел

1. #include <iostream>
2. #include <algorithm>
3. #include <vector>
4. #include <string>
5. #include <fstream>
6. #include "tree.h"
7. using namespace std;
8. ofstream out1;
9. Node \* bin\_tree::rotate\_left(Node \* node){
10. Node \* b = node->right;
11. node->right = b->left;
12. if (node->right)
13. b->left->parent = node;
14. b->left = node;
15. b->parent = node->parent;
16. b->set\_diff();
17. node->set\_diff();
18. if (node == root){
19. root = b;
20. b->parent = NULL;
21. }else if (node->parent->right == node){
22. node->parent->right = b;
23. }else if (node->parent->left == node)
24. node->parent->left = b;
25. node->parent = b;
26. node->calc\_height();
27. b->calc\_height();
28. b->set\_diff();
29. node->set\_diff();
30. return b;
31. }
32. Node \* bin\_tree::rotate\_right(Node \* node){
33. Node \* b = node->left;
34. node->left = b->right;
35. if (node->left)
36. b->right->parent = node;
37. b->right = node;
38. b->parent = node->parent;
39. b->set\_diff();
40. node->set\_diff();
41. if (node == root){
42. root = b;
43. b->parent = NULL;
44. }else if (node->parent->right == node){
45. node->parent->right = b;
46. }else if (node->parent->left == node)
47. node->parent->left = b;
48. node->parent = b;
49. node->calc\_height();
50. b->calc\_height();
51. b->set\_diff();
52. node->set\_diff();
53. return b;
54. }
55. Node \* bin\_tree::balance(Node \* node){
56. node->set\_diff();
57. if (node->get\_diff() == 2){
58. node->right->set\_diff();
59. if (node->right->get\_diff() < 0)
60. node->right = rotate\_right(node);
61. return rotate\_left(node);
62. }else if (node->get\_diff() == -2){
63. node->left->set\_diff();
64. if (node->left->get\_diff() > 0)
65. node->left = rotate\_left(root);
66. return rotate\_right(node);
67. }
68. return node;
69. }
70. // Старт с корня
71. void bin\_tree::insert(Node \* node, double key){
72. if (node){
73. Node \* temp = node;
74. if (fabs(node->get\_key() - key) <= 0.00001){
75. return;
76. }else if (node->get\_key() < key){
77. if (node->right){
78. temp = node;
79. insert(node->right, key);
80. node->calc\_height();
81. this->balance(node);
82. }
84. else{
85. Node \* new\_node = new Node(key, this);
86. new\_node->parent = temp;
87. node->right = new\_node;
88. count++; // + элемент
89. node->calc\_height();
90. return;
91. }
92. }else{
93. if (node->left){
94. temp = node;
95. insert(node->left, key);
96. node->calc\_height();
97. this->balance(node);
98. }
99. else{
100. Node \* new\_node = new Node(key, this);
101. new\_node->parent = temp;
102. node->left = new\_node;
103. count++; // + элемент
104. node->calc\_height();
105. return;
106. }
107. }
108. }else{
109. root = new Node(key);
110. count++;
111. return;
112. }
113. }
114. Node \* bin\_tree::find(Node \* node, double key){
115. if (fabs(node->get\_key() - key) <= 0.000001)
116. return node;
117. else if (node->get\_key() < key){
118. if (!node->right)
119. return NULL;
120. find(node->right, key);
121. }
122. else{
123. if (!node->left)
124. return NULL;
125. find(node->left, key);
126. }
127. }
128. Node \* bin\_tree::find\_min(Node \* node){
129. Node \* temp;
130. if (node)
131. temp = node;
132. else
133. temp = root;
134. while (temp->left)
135. temp = temp->left;
136. return temp;
137. }
138. Node \* bin\_tree::find\_max(Node \* node){
139. Node \* temp;
141. if (node)
142. temp = node;
143. else
144. temp = root;
146. while (temp->right)
147. temp = temp->right;
149. return temp;
150. }
151. Node \* bin\_tree::remove\_min(Node \* node){
152. if (node){
153. if (!node->left)
154. return node->right;
156. node->left = remove\_min(node->left);
157. return balance(node);
158. }else{
159. if (!root->left)
160. return root->right;
162. root->left = remove\_min(root->left);
163. return balance(root);
164. }
166. }
167. // Начинать с корня
168. Node \* bin\_tree::remove(Node \* node, double key){
169. if (!node) return 0;
170. if (key < node->get\_key())
171. node->left = remove(node->left, key);
172. else if (key > node->get\_key()){
173. node->right = remove(node->right, key);
174. }else if (fabs(node->get\_key() - key) < 0.0001 && node != root){
175. Node \* l = node->left;
176. Node \* r = node->right;
177. if (l)
178. l->parent = node->parent;
179. if (r)
180. r->parent = node->parent;
181. delete node;
182. count--;
183. if(l)
184. l->calc\_height();
185. if(r)
186. r->calc\_height();
188. if (!r) return l;
189. Node \* min = find\_min(r);
190. min->right = remove\_min(r);
191. min->left = l;
192. min->calc\_height();
193. return balance(min);
194. }else if (fabs(node->get\_key() - key) < 0.0001){
195. Node \* l = node->left;
196. Node \* r = node->right;
197. if (!l && !r){
198. count--;
199. delete root;
200. root = NULL;
201. return NULL;
202. }
203. if (l)
204. l->parent = node->parent;
205. if (r)
206. r->parent = node->parent;
208. delete node;
209. count--;
210. if(l)
211. l->calc\_height();
212. else if(r)
213. r->calc\_height();
214. else if(l && r)
215. l->calc\_height(), r->calc\_height();
217. Node \* min = find\_min(r);
218. min->right = remove\_min(r);
219. min->left = l;
220. root = min;
222. if (l){
223. min->left->calc\_height();
224. }if (min->right){
225. min->right->calc\_height();
226. }
227. min->calc\_height();
228. return balance(min);
229. }
230. return balance(node);
231. }
232. // сброс visited после dfs
233. void bin\_tree::dfs\_support(Node \* a){
234. static int cnt = 0;
235. if (a->visited){
236. a->visited = 0;
237. cnt++;
238. }
239. if (cnt != count && a->left)
240. dfs\_support(a->left);
241. if(cnt != this->count && a->right)
242. dfs\_support(a->right);
243. }
244. // обход всех узлов дерева и применение на них функции f
245. void bin\_tree::dfs(Node \* a, char flag, string path, double \* sum){
246. if (a){
247. dfs\_support(root);
248. static int cnt = 0;
249. if (!a->visited){
250. switch (flag){
251. case 1:
252. a->out();
253. break;
254. case 2:
255. out1 << a->get\_key() << " ";
256. break;
257. case 3:
258. \*sum += a->get\_key();
259. break;
260. }
262. a->visited = 1;
263. cnt += 1;
264. }
265. if (cnt != this->count && a->left)
266. dfs(a->left, flag, path, sum);
267. if(cnt != this->count && a->right)
268. dfs(a->right, flag, path, sum);
269. dfs\_support(root);

272. }
273. }
274. double bin\_tree::find\_avg(){
275. double sum;
276. dfs(root, 3, "", &sum);
277. return sum / count;
278. }
279. void Node::out(){
280. cout << "---------------------------------\n";
281. cout.width(4);
282. if (!this)
283. cout << "Pointer error\n";
284. else if (this->left && this->right)
285. cout << this->get\_key() << "\t" << this->left->get\_key() << "\t" << this->right->get\_key() << "\t" << (parent ? parent->get\_key() : 0) << "\n";
286. else if (this->left)
287. cout << this->get\_key() << "\t" << this->left->get\_key() << "\t" << "0000" << "\t" << (parent ? parent->get\_key() : 0)<< "\n";
288. else if (this->right)
289. cout << this->get\_key() << "\t" << "0000" << "\t" << this->right->get\_key() << "\t" << (parent ? parent->get\_key() : 0)<< "\n";
290. else
291. cout << this->get\_key() << "\t" << "0000" << "\t" << "0000"<< "\t" << (parent ? parent->get\_key() : 0) <<"\n";
292. cout << "---------------------------------\n\n";
293. }
294. void bin\_tree::out(){
295. cout.width(4);
296. cout << "---------------------------------\n";
297. cout << "self\t" << "left\t" << "right\t" << "parent\n";
298. cout << "---------------------------------\n";
299. dfs(this->root, 1);
300. cout << "Количество узлов = " << count << "\n";
301. }
302. void bin\_tree::save(){
303. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
304. int i=0;
305. string path;
306. input\_check(&i);
307. if (i){
308. while (std::cin >> path, std::cin.fail()){
309. cin.clear();
310. cin.ignore(500, '\n');
311. cout << "Неправильный ввод\n";
312. }
313. path = "saves/" + path;
314. }else
315. path = "saves/save.txt";
316. out1.open(path);
317. if (out1.is\_open()){
318. dfs(root, 2, path);
319. out1.close();
320. }
321. }
322. // Возвращает указатель на загруженный объект
323. bin\_tree \* bin\_tree::load(){
324. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
325. int i=0;
326. string path;
327. input\_check(&i);
328. if (i){
330. while (std::cin >> path, std::cin.fail()){
331. cin.clear();
332. cin.ignore(500, '\n');
333. cout << "Неправильный ввод\n";
334. }

337. path = "saves/" + path;
338. }else{
339. path = "saves/save.txt";
340. }
342. bin\_tree \* tree = NULL;
343. tree = new bin\_tree();
344. double key;
345. ifstream in(path);
346. if (in.is\_open()){
348. while(in >> key){
349. tree->insert(tree->root, key);
350. }
352. }
353. in.close();
354. return tree;
355. }
356. void bin\_tree::sum\_tree(){
357. cout << "Загрузите дерево, с которым будет сложение\n";
358. cout << "Если хотите ввести название нажмите 1 иначе 0\n";
359. int i=0;
360. string path;
361. input\_check(&i);
362. if (i){
363. while (std::cin >> path, std::cin.fail()){
364. cin.clear();
365. cin.ignore(500, '\n');
366. cout << "Неправильный ввод\n";
367. }
368. path = "saves/" + path;
369. }else
370. path = "saves/save.txt";
371. double key;
372. ifstream in(path);
373. if (in.is\_open()){
374. while(in >> key){
375. insert(root, key);
376. }
377. }
378. in.close();
379. }
380. /\*
381. 1 тест
382. 1
383. 1
384. 1
385. 1
386. 2
387. 1
388. 3
389. 1
390. 4
391. 1
392. 5
393. 1
394. 6
395. 1
396. 7
397. 1
398. 8
399. 1
400. 9
401. 1
402. 10
403. 2 тест
404. 1
405. 1
406. 10
407. 1
408. 9
409. 1
410. 8
411. 1
412. 7
413. 1
414. 6
415. 1
416. 5
417. 1
418. 4
419. 1
420. 3
421. 1
422. 2
423. 1
424. 1
425. 3 тест
426. 1
427. 1
428. 2
429. 1
430. 4
431. 1
432. 6
433. 1
434. 8
435. 1
436. 10
437. 1
438. 1
439. 1
440. 3
441. 1
442. 5
443. 1
444. 7
445. 1
446. 9
447. \*/
448. #include <iostream>
449. #include <cstdlib>
450. #include "tree.h"
451. /\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/
452. int main(int argc, char\*\* argv) {
453. system("chcp 1251 > nul");
454. bin\_tree::menu();
455. system("pause");
456. return 0;
457. }
458. #include <iostream>
459. #include <cstdlib>
460. #include "tree.h"
461. void bin\_tree::out\_menu\_tree(){
462. std::cout << "Введите 1, чтобы добавить узел\n";
464. std::cout << "Введите 2, чтобы удалить узел\n";// Вовремя теста Insert-a всё сломалось
466. std::cout << "Введите 3, чтобы вывести в виде таблицы\n";
468. std::cout << "Введите 4, чтобы получить доступ к узлу по ключу\n"; // Выводится узел и его дочерние узлы и его родительский ключ
470. std::cout << "Введите 5, чтобы очистить консоль\n";
472. std::cout << "Введите 6, чтобы прибавить дерево к другому\n";// изи - загрузить дерево как массив и просто вставить все элементы
474. std::cout << "Введите 7, чтобы сравнить текущие дерево с другим без учёта ключей\n";// Нужна глубина
476. std::cout << "Введите 8, чтобы сравнить текущие дерево с другим с учётом ключей\n";// сначала переделать удаление
478. std::cout << "Введите 9, чтобы вывести высоту дерева\n"; // Ширина - максимальное кол-во узлов одинаковой высоты (можно обновлять при insert)
480. std::cout << "Введите 10, чтобы вывести количество узлов дерева\n";
482. std::cout << "Введите 11, чтобы сохранить дерево\n";
484. std::cout << "Введите 0 для выхода\n";
485. }
486. template<class Type>
487. void input\_check(Type \*t){
488. while (std::cin >> \*t, std::cin.fail()){
489. std::cin.clear();
490. std::cin.ignore(500, '\n');
491. std::cout << "Неправильный ввод\n";
492. }
493. }
494. void bin\_tree::menu\_tree(){
495. int i=0;
496. int menu = 1;
497. double key = 0;
498. Node \* node = NULL;
500. while (menu){
501. out\_menu\_tree();
503. input\_check(&menu);
505. switch(menu){
506. case 1: /\* Добавить узел\*/
507. std::cout << "Введите ключ\n";
508. input\_check(&key);
509. this->insert(this->root, key);
510. break;
511. case 2: /\* Удалить узел \*/
512. std::cout << "Введите ключ по которому хотите удалить узел\n";
513. input\_check(&key);
514. this->remove(root, key);
515. break;
516. case 3: /\* Вывод\*/
517. this->out();
518. break;
519. case 4: /\* Узел по ключу \*/
520. std::cout << "Введите ключ\n";
521. input\_check(&key);
522. std::cout << "---------------------\n";
523. std::cout << "self\t" << "left\t" << "right\n";
524. std::cout << "---------------------\n";
525. this->find(this->root, key)->out();
526. break;
527. case 5:
528. system("cls");
529. break;
530. case 6:
531. sum\_tree();
532. break;
533. case 7:
534. cmp\_struct(bin\_tree::load());
535. break;
536. case 8:
537. cmp\_keys(bin\_tree::load());
538. break;
539. case 9:
540. std::cout << "Высота = " << (int)this->root->get\_height() << "\n";
541. break;
542. case 10:
543. std::cout << "В дереве "<< count << " узлов\n";
544. break;
545. case 11:
546. this->save();
547. break;
548. case 0:
549. break;
550. default:
551. std::cout << "Неправильный ввод\n";
552. break;
553. }
555. }
556. }
557. void bin\_tree::out\_menu(){
558. std::cout << "Введите 1, чтобы создать пустое дерево\n";
560. std::cout << "Введите 2, чтобы загрузить дерево\n";
562. std::cout << "Введите 3, чтобы очистить консоль\n";
564. std::cout << "Введите 0 для выхода\n";
565. }
566. void bin\_tree::menu(){
567. int menu = 1, i=0;
569. bin\_tree \* tree;
570. while (menu){
571. out\_menu();
572. input\_check(&menu);
573. switch(menu){
574. case 1: /\* пустое дерево\*/
575. tree = new bin\_tree();
576. tree->menu\_tree();
577. delete tree;
578. break;
579. case 2: /\* Загрузить дерево \*/
580. bin\_tree::load()->menu\_tree();
581. break;
583. case 3:
584. system("cls");
585. break;
586. default:
587. std::cout << "Неправильный ввод\n";
589. }
590. }
591. }
592. #include <string>
593. using namespace std;
594. class Node;
595. class bin\_tree;
596. class Node{
597. private:
598. char diff; // Разница высот левого и правого поддерева
599. char height; // Расчитывается, получить всегда можно
600. bin\_tree \*owner; // Задаётся в конструкторе, получить можно всегда
601. double key; // Только задавать, изменить нельзя
602. public:
603. Node \*left;
604. Node \*right;
605. Node \* parent;
606. bool visited;// Для обхода
608. char get\_diff(){
609. if (right && left)
610. return right->height - left->height;
611. else if (left)
612. return 0 - left->height;
613. else if(right)
614. return right->height - 0;
615. else
616. return 0;
617. }
619. char set\_diff(){
620. if (right && left)
621. diff = right->height - left->height;
622. else if (left)
623. diff = 0 - left->height;
624. else if(right)
625. diff = right->height - 0;
626. else
627. diff = 0;
628. }
630. void calc\_height(){
631. if (right && left)
632. height = right->height>left->height? right->height + 1 : left->height + 1;
633. else if (left)
634. height = left->height+1;
635. else if(right)
636. height = right->height+1;
637. else
638. height = 1;
639. }
640. char get\_height(){ return height; }
641. double get\_key(){ return key; }
642. bin\_tree \* get\_owner(){ return owner; }
644. void out();
645. Node(double key){
646. left = NULL; right = NULL; visited = 0; parent = NULL;
647. this->key = key; owner = NULL; diff = 0; height = 1;
648. }
650. Node(double key, bin\_tree \* owner){
651. left = NULL; parent = NULL;
652. right = NULL; visited = 0; this->key = key;
653. this->owner = owner; diff = 0; height = 1;
654. }
655. ~Node(){};
656. };
657. class bin\_tree {
658. public:
659. Node \*root;
660. int count; // кол-во узлов, увеличивается при добавлении в дерево, уменьшается при удалении
662. static void out\_menu\_tree();
663. static void out\_menu();
664. static void menu();
665. static bin\_tree \* load();
667. void menu\_tree();
668. void dfs(Node \* a, char flag, string path="", double \* sum = NULL);
669. void dfs\_support(Node \* a);
670. void out();
671. void insert(Node \* node, double key);
672. Node \* find(Node \* node, double key); // Доступ к узлу по ключу
673. Node \* remove(Node \* node, double key);
674. Node \* rotate\_left(Node \* node);
675. Node \* rotate\_right(Node \* node);
676. Node \* balance(Node \* node);
677. void save();
678. void sum\_tree();
679. Node \* find\_min(Node \* node = NULL);
680. Node \* remove\_min(Node \* node = NULL);
681. Node \* find\_max(Node \* node = NULL);
682. double find\_avg();
684. // Сравнение без учёта ключей
685. void cmp\_struct(bin\_tree \* tree){
686. cout << "Высота загруженного дерева = " << (int)(tree->root->get\_height())
687. << "\t" << "Высота исходного дерева = " << (int)(root->get\_height()) << "\n";
688. cout << "Количество узлов загруженного дерева = " << tree->count
689. << "\t" << "Количество узлов исходного дерева = " << count << "\n";
690. }
692. // Сравнение с учётом ключей
693. void cmp\_keys(bin\_tree \* tree){
694. cout << "Максимальный узел загруженного дерева = " << tree->find\_max()->get\_key()
695. << "\t" << "Максимальный узел исходного дерева = " << find\_max()->get\_key() << "\n";
696. cout << "Минимальный узел загруженного дерева = " << tree->find\_min(tree->root)->get\_key()
697. << "\t" << "Минимальный узел исходного дерева = " << find\_min(root)->get\_key() << "\n";
698. cout << "Среднее значение по узлам у загруженного дерева = " << tree->find\_avg()
699. << "\t" << "Среднее значение по узлам у исходного дерева = " << find\_avg() << "\n";
700. }

703. bin\_tree() : root(NULL), count(0) {}
704. bin\_tree(Node \* rt) : root(rt), count(1){}
705. ~bin\_tree(){};
706. };
707. template<class Type>
708. void input\_check(Type \*t);

## Приложения Б. Рисунки

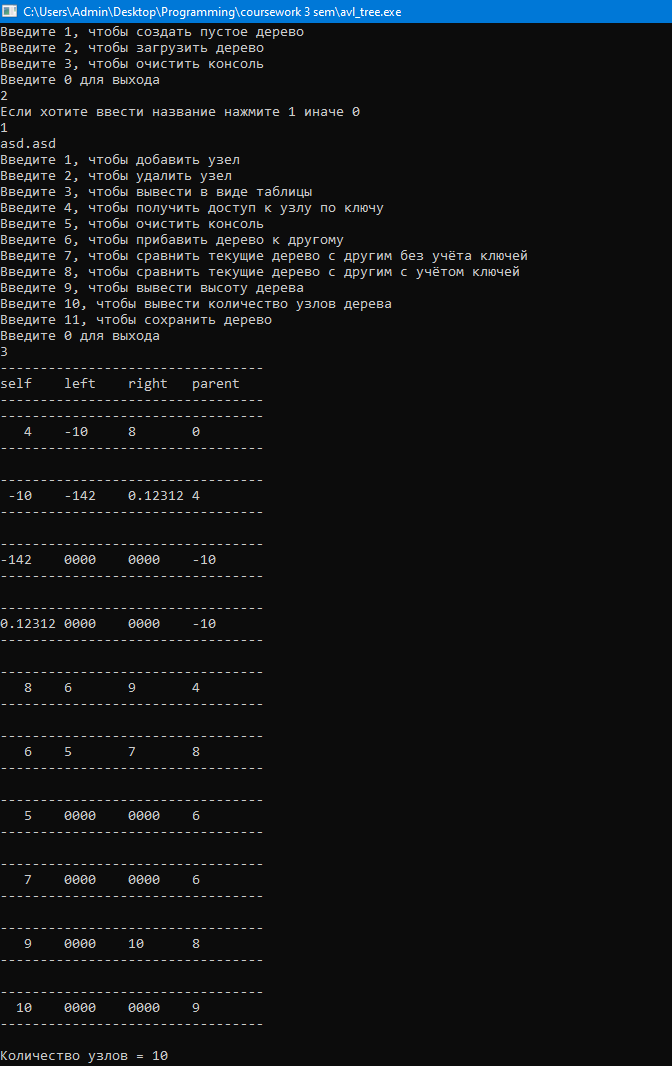


Рисунок Б.1 – тестирование загрузки с названием файла

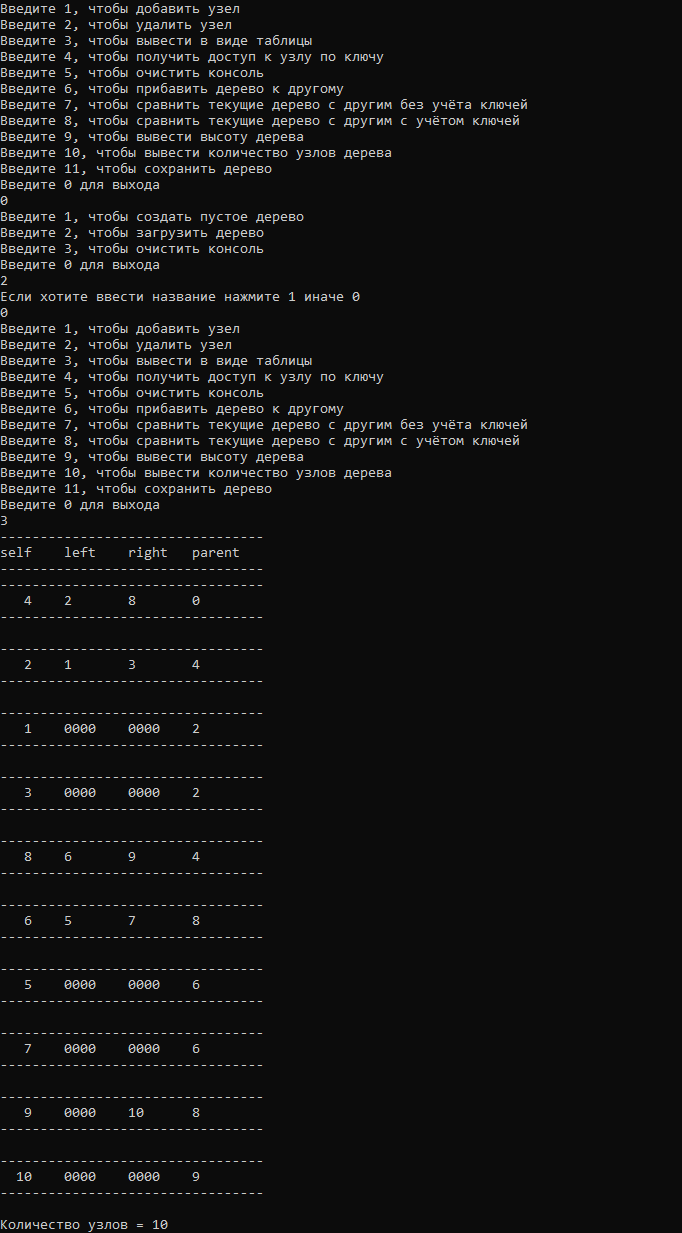


Рисунок Б.2 – тестирование загрузки из стандартного файла

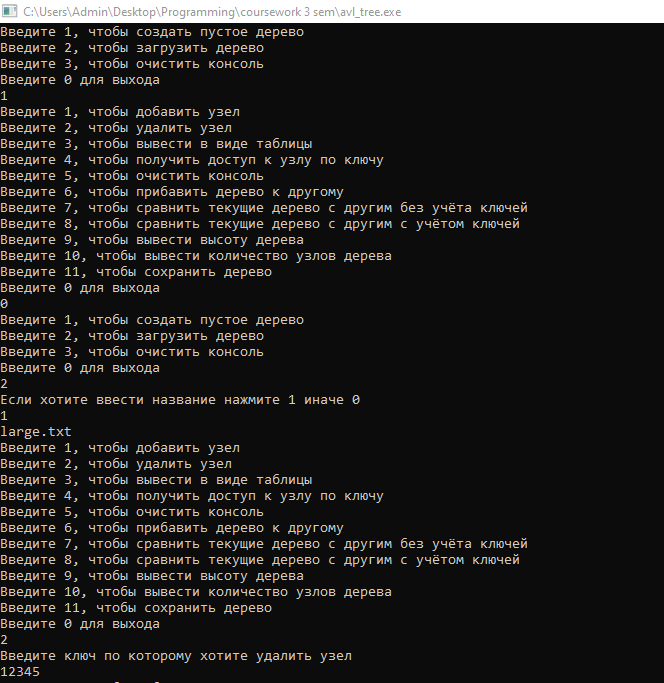


Рисунок Б.3.1 – тестирование загрузки больших данных

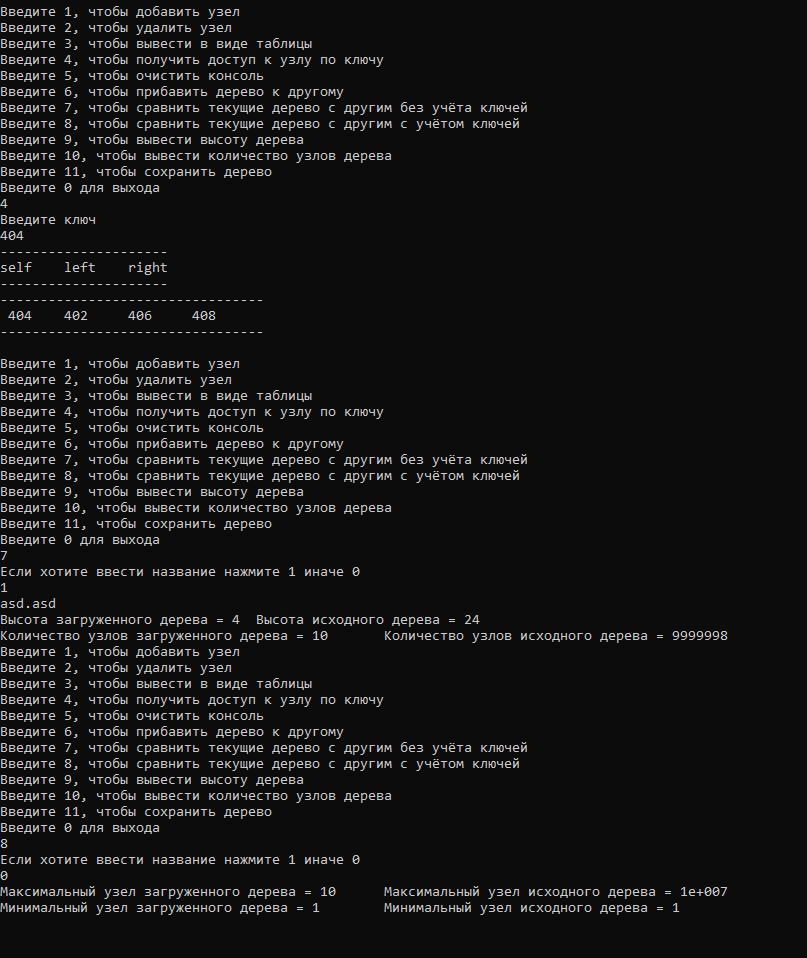


Рисунок Б.3.2 – тестирование загрузки больших данных