

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**БРЯНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**Кафедра «Информатика и программное обеспечение»**

## **КУРСОВОЙ ПРОЕКТ**

### **Программа “Текстовый редактор”**

**Всего листов 26**

Преподаватель: -

«\_\_\_\_» 2025 г.

Студент гр. О-24-ИВТ-1-ПО-Б

«\_\_\_\_» \_\_\_\_\_ 2025 г.

**БРЯНСК 2025**

|   |           |
|---|-----------|
| <b>Введение .....</b>   | <b>3</b>  |
| <b>1. Анализ предметной области.....</b>  | <b>5</b>  |
| <b>1.1. Историческая справка .....</b>  | <b>5</b>  |
| 1.1.1. Текстовые редакторы .....  | 5         |
| 1.1.2. Деревья.....   | 5         |
| <b>1.2. Основные термины и определения.....</b>                                 | <b>5</b>  |
| <b>1.3. Типовые подходы к решению.....</b>                                      | <b>7</b>  |
| 1.3.1. Хранение в виде плоского массива строк. ....                             | 7         |
| 1.3.2. Список строк (линейный связный список).....                              | 7         |
| 1.3.3. Rope-структура («канат»).....  | 8         |
| 1.3.4. Произвольное бинарное дерево строк .....                                 | 8         |
| 1.3.5. СерIALIZАЦИЯ .....   | 8         |
| 1.3.6. Использование стандартных средств C++ vs. ручное управление памятью..... | 9         |
| <b>2. Конструкторская часть.....</b>  | <b>10</b> |
| <b>2.1. Общая структура проекта.....</b>  | <b>10</b> |
| 2.1.1. Корневая директория проекта .....  | 10        |
| 2.1.2. Исходный код приложения.....   | 11        |
| 2.1.3. Модульные и интеграционные тесты .....                                   | 11        |
| <b>2.2. Обобщённый алгоритм программы.....</b>                                  | <b>12</b> |
| <b>2.3. Технические решения.....</b>  | <b>13</b> |
| 2.3.1. Структура дерева .....   | 13        |
| 2.3.2. Сохранение в бинарный файл.....  | 15        |
| 2.3.3. Чтение из бинарного файла .....  | 16        |
| <b>3. Тестирование.....</b>   | <b>18</b> |
| <b>4. Итоги проделанной работы.....</b>   | <b>22</b> |
| <b>Список литературы.....</b>   | <b>23</b> |

# **ВВЕДЕНИЕ**

## **Актуальность:**

В условиях роста требований к эффективности обработки текстовых данных и необходимости обеспечения надёжного промежуточного хранения информации при редактировании, использование структурированных подходов к организации данных становится всё более значимым. Традиционные текстовые редакторы зачастую полагаются на стандартные библиотеки и контейнеры высокого уровня, что ограничивает контроль над памятью и форматом хранения. Реализация собственной структуры данных — в частности, двоичного дерева с переменными по длине записями строк — позволяет не только глубже понять принципы низкоуровневого управления памятью и сериализации, но и создать более гибкую основу для расширяемых систем редактирования. Актуальность работы обусловлена также учебной необходимостью освоения принципов работы с производными классами потоков ввода-вывода, двоичной сериализации и рекурсивных структур данных.

## **Цель работы:**

Разработка текстового редактора, использующего собственную структуру данных — двоичное дерево с переменной длиной строк — для промежуточного хранения содержимого редактируемого файла, с реализацией двоичного файла как класса, производного от `std::fstream`.

## **Задачи работы:**

Спроектировать и реализовать структуру данных «бинарное дерево», в которой:

терминальные узлы содержат строку в формате: целочисленный счётчик длины + последовательность символов без нуль-терминатора;

промежуточные узлы содержат указатели на левое и правое поддеревья и счётчик количества вершин в поддереве.

Реализовать класс двоичного файла, наследуемый от `std::fstream`, обеспечивающий:

запись структуры дерева в двоичный файл с указателем на корень в заголовке;

загрузку дерева из двоичного файла в память.

Обеспечить преобразование между текстовым файлом и двоичным представлением:

создание двоичного файла из заданного текстового;

сохранение содержимого дерева в виде текстового файла.

Разработать графический интерфейс пользователя с использованием GTK4, интегрированный с реализованной структурой данных.

Провести тестирование корректности операций ввода-вывода, сериализации и редактирования.

# 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1. Историческая справка

### 1.1.1. Текстовые редакторы

Текстовые редакторы. Первые текстовые редакторы появились в 1960-х годах как линейные редакторы, предназначенные для корректировки программ на перфокартах; одним из ранних примеров считается **qed** (1965). В 1970-х годах появились полноэкранные и первые графические редакторы, такие как **Electric Pencil**. Дальнейшее развитие привело к созданию мощных инструментов вроде **vi**, **Emacs**, а затем — современных редакторов и **IDE**, таких как **Visual Studio Code**. [1]

### 1.1.2. Деревья

Структуры данных «деревья». Дерево — одна из фундаментальных структур данных, эмулирующая иерархическую организацию в виде связанных узлов. Бинарные деревья поиска были формализованы в 1960-х годах (в частности, **Конвеем Бернерс-Ли** и **Дэвидом Уилером**) для эффективного хранения помеченных данных. Древовидные структуры широко применяются, например, в файловых системах, где реализуют понятную пользователю иерархию каталогов. [2]

## 1.2. Основные термины и определения

В рамках данного проекта центральное место занимают два взаимосвязанных понятия: **текстовый редактор** и **древовидная структура данных**. **Текстовый редактор** определяется как программное средство, предназначенное для создания, просмотра и модификации неформатированного текста. В отличие от текстовых процессоров, он не

сохраняет метаинформацию о стиле оформления, что делает его особенно подходящим для работы с исходным кодом, конфигурационными файлами и другими форматами, где важна точность представления данных.

Для промежуточного хранения редактируемого текста в проекте используется **бинарное дерево** — иерархическая структура данных, в которой каждый узел содержит не более двух дочерних узлов: **левый** и **правый**. Узлы дерева подразделяются на два типа. Терминальные узлы, или листья, не имеют потомков и в данном случае хранят отдельные строки текста в виде записи переменной длины: перед самой последовательностью символов располагается целочисленный счётчик, указывающий длину строки; завершающий нулевой символ (null-terminator) при этом не используется. Промежуточные узлы содержат указатели на левое и правое поддеревья, а также целочисленное поле — счётчик количества вершин в соответствующем поддереве, что может быть использовано, например, для балансировки или навигации.

Весь набор данных сериализуется в **двоичный файл** — файл, в котором информация представлена в виде последовательности байтов, отражающей точную внутреннюю структуру объектов в памяти, без преобразования в текстовое представление. Для организации доступа к такому файлу реализуется специализированный класс, производный от стандартного C++-класса **std::fstream**, обеспечивающего низкоуровневый контроль над операциями ввода-вывода.[3] Процесс **сериализации** подразумевает преобразование дерева в последовательность байтов с учётом относительных смещений узлов, чтобы обеспечить корректное восстановление структуры при последующем чтении.

### **1.3. Типовые подходы к решению**

Реализация текстового редактора с промежуточным хранением данных в виде структуры на диске может быть выполнена различными способами, в зависимости от целей: максимальная производительность, минимальное потребление памяти, простота реализации или учебная демонстрация принципов низкоуровневого управления данными. Ниже рассматриваются основные подходы, применимые к задаче хранения и редактирования текста.

#### *1.3.1. Хранение в виде плоского массива строк.*

Наиболее простой способ — загрузка всего текстового файла в память как последовательности строк, например, с использованием контейнера `std::vector<std::string>`. Такой подход обеспечивает быстрый произвольный доступ к любой строке и прост в реализации, однако плохо масштабируется при работе с большими файлами и не поддерживает эффективную вставку или удаление строк в середине документа без копирования значительной части данных. Кроме того, он не соответствует требованию использовать древовидную структуру данных.

#### *1.3.2. Список строк (линейный связный список).*

Использование связного списка (`std::list` или собственной реализации) позволяет вставлять и удалять строки за константное время при наличии итератора на позицию. Однако произвольный доступ к строке по индексу требует линейного обхода, что снижает удобство при навигации в длинных документах. Этот метод также не использует дерево и не демонстрирует иерархическую организацию данных.

### *1.3.3. Rope-структура («канат»)*

Rope — специализированная древовидная структура данных, разработанная именно для эффективного редактирования длинных текстовых последовательностей.[4] В rope листья содержат подстроки текста, а внутренние узлы хранят суммарную длину поддерева слева, что позволяет быстро выполнять операции по индексу (например, «получить символ на позиции  $n$ »). Rope поддерживает вставку, удаление и конкатенацию за  $O(\log n)$ , что делает её подходящей для редакторов с большим объёмом текста. Однако её реализация требует сложной логики балансировки и управления памятью, что выходит за рамки учебной задачи, но остаётся близкой концептуально.

### *1.3.4. Произвольное бинарное дерево строк*

В рамках учебного проекта допустимо использование несбалансированного бинарного дерева, в котором каждая строка помещается в отдельный лист, а порядок строк задаётся, например, in-order обходом. Такой подход демонстрирует базовые принципы рекурсивных структур данных и сериализации, но не оптимален с точки зрения доступа. Тем не менее, он полностью удовлетворяет условию задачи: использование дерева с переменной длиной строк и хранение в двоичном файле.

### *1.3.5. СерIALIZАЦИЯ*

При записи дерева в файл возможны два основных подхода. Первый — хранение каждого узла в фиксированном формате, включая смещения (offsets) дочерних узлов относительно начала файла. Это позволяет восстанавливать дерево без загрузки всего файла в память и соответствует низкоуровневому стилю задания. Второй — рекурсивная сериализация всех узлов в порядке обхода (например, preorder), что упрощает чтение/запись, но

требует полной загрузки структуры в память перед использованием. Выбор первого метода предпочтителен в контексте учебной задачи, так как он подчёркивает контроль над двоичным форматом.

#### *1.3.6. Использование стандартных средств C++ vs. ручное управление памятью.*

Многие современные реализации полагаются на std::string, std::shared\_ptr и стандартные контейнеры, что упрощает код, но скрывает детали размещения данных в памяти и на диске. Условие задачи, напротив, подразумевает минимизацию зависимости от стандартной библиотеки и акцент на ручной сериализации, что соответствует классическому подходу к обучению системному программированию.

Таким образом, предложенный в проекте подход — использование собственного бинарного дерева с переменной длиной строк и сериализацией через смещения в двоичный файл, инкапсулированный в классе, производном от std::fstream, — представляет собой обоснованный учебный компромисс между сложностью промышленных решений (например, rope) и простотой линейных структур, одновременно демонстрируя ключевые принципы низкоуровневой работы с данными.

## 2. КОНСТРУКТОРСКАЯ ЧАСТЬ

Проект организован по классической модульной схеме, включающей корневые файлы сборки, исходный код приложения, тесты и конфигурацию инструментов разработки. Ниже приведено детальное описание каждого элемента дерева каталогов.

### 2.1. Общая структура проекта

Структура каталога проекта включает следующие основные элементы:

- корневые файлы конфигурации сборки и окружения;
- каталог исходных кодов приложения;
- каталог модульных и интеграционных тестов;
- инфраструктурные файлы для среды Nix.

#### 2.1.1. Корневая директория проекта

**CMakeLists.txt** — основной сценарий сборки. Определяет параметры проекта, стандарт C++, правила сборки исполняемого файла, подключение поддиректорий и остальные глобальные настройки.

**Makefile** — обёртка над CMake, предназначенная для удобного запуска сборки с помощью команды make. Обычно включает цели для конфигурации, сборки, очистки и запуска.

**compile\_commands.json** — автоматически генерируемый файл, содержащий команды компиляции всех исходников проекта. Используется IDE и системами статического анализа для корректной работы подсветки, автодополнения и линтеров.

**flake.nix / flake.lock** — файлы, определяющие репродуцирую-среду через Nix Flakes. Позволяют собирать проект в идентичной окружении на любых машинах без установки зависимостей вручную.

### *2.1.2. Исходный код приложения*

За исходный код программы отвечает каталог — **src/**. Там хранятся все ключевые заголовочные файлы и их реализация в **.cpp** файлах, а также важный **CMake-файл** определяющий их сборку.

**main.cpp** — точка входа программы. Инициализирует графический интерфейс и основные компоненты редактора.

**EditorWindow.h / EditorWindow.cpp** — определение главного окна приложения. Содержит логику управления интерфейсом, компоновку GTK4-виджетов и основные действия редактора.

**CustomTextView.h / CustomTextView.cpp** — реализация кастомного текстового виджета на базе GTK4. Отвечает за отображение текста, обработку событий, синхронизацию с бинарным деревом и пользовательское взаимодействие.

**Tree.h / Tree.cpp** — класс структуры бинарного дерева. Содержит операции над узлами, обходы дерева, расчётные функции и логические связи с редактором.

**BinaryTreeFile.h / BinaryTreeFile.cpp** — модуль для работы со структурой дерева в бинарном файле. Реализует чтение, запись, сериализацию и десериализацию узлов.

**CMakeLists.txt (внутри src)** — локальный сценарий сборки модуля. Определяет библиотеку/объектные файлы и связывает их с главным исполняемым файлом.

### *2.1.3. Модульные и интеграционные тесты*

Каталог **tests/** — отвечает за автотесты над основными классами программы и тест над окружением для разработки и тестирование программы.

**test1.cpp, test2.cpp** — модульные тесты для проверки корректности алгоритмов дерева, логики работы с бинарными файлами и отдельных функциональных блоков.

**test-gtk.cpp** — тест, направленные на базовую проверку GTK-компонентов, включая создание окна, рендеринг и корректное взаимодействие с системой. Служит для проверки насколько окружение готово к запуска исходных файлов, в случае проблем основная программа никак не сможет скомпилироваться.

**gen\_file.cpp** — скрипт для создания текстового файла желаемого размера с целью тестирования производительности редактора.

**CMakeLists.txt (внутри tests)** — определяет цели сборки тестов, подключает тестовые файлы, обеспечивает интеграцию с системой тестирования.

## 2.2. Обобщённый алгоритм программы

Программа представляет собой текстовый редактор с графическим интерфейсом на GTK4, в котором текст временно хранится в двоичном файле, реализованном через класс, производный от `fstream`. Основной особенностью является использование бинарного дерева для организации данных: конечные вершины содержат строки переменной длины, представленные целочисленным счётчиком и последовательностью символов без нулевого терминатора, а промежуточные вершины включают указатели на левое и правое поддеревья и количество вершин в них. В начале файла хранится указатель на корень дерева, что обеспечивает эффективный доступ к содержимому.

Программа поддерживает создание и заполнение двоичного файла на основе текстового документа, редактирование текста через удобный графический интерфейс, а также сохранение данных как в двоичный, так и в текстовый файл. Такое решение позволяет сочетать наглядное

редактирование с эффективной внутренней структурой хранения данных, демонстрируя работу с двоичными файлами и деревьями в контексте текстового редактора. С внешним видом программы можно ознакомиться на Рис. 1

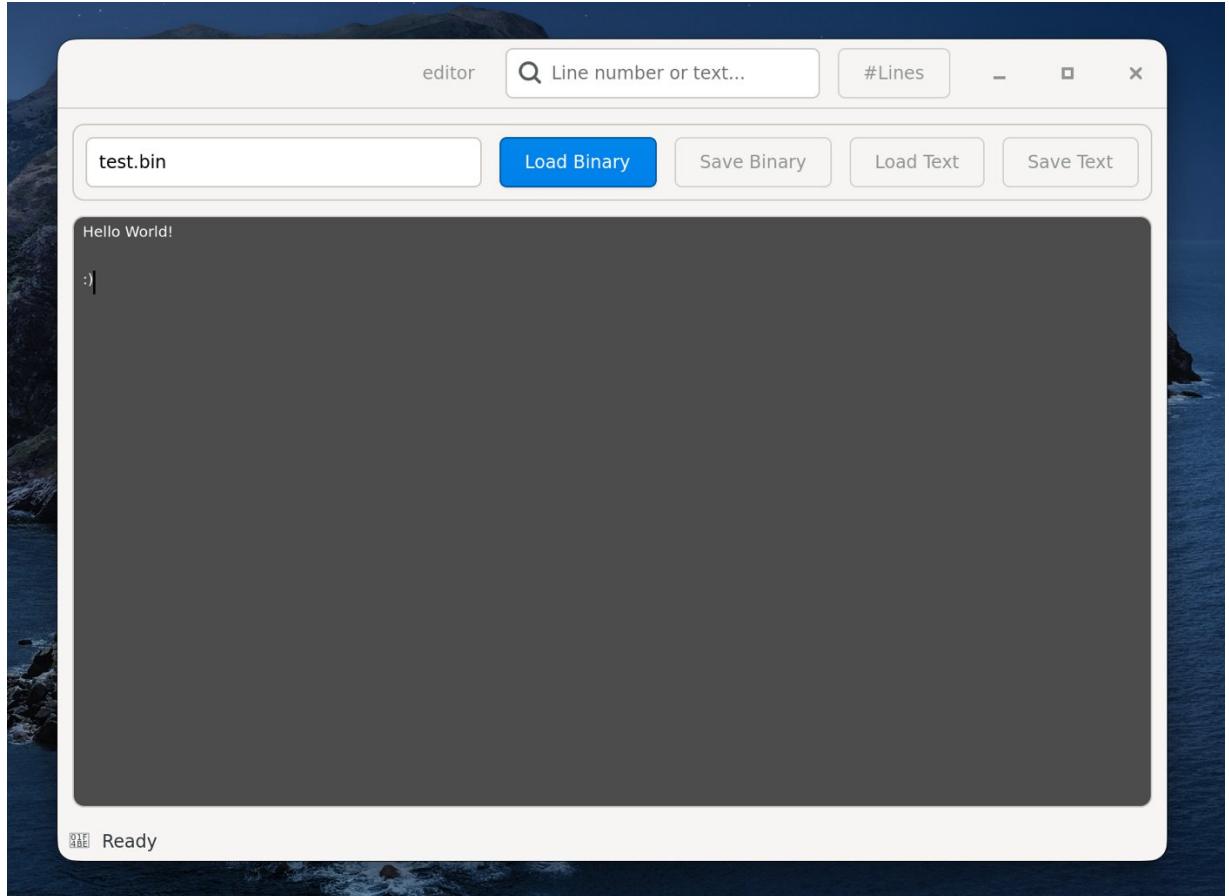


Рис. 1 Внешний вид программы.

## 2.3. Технические решения

### 2.3.1. Структура дерева

Разработаны структуры данных для двоичного файла, содержащего дерево, как того требует задание.

Базовая структура Node — абстрактный класс, определяющий интерфейс для узлов дерева. Содержит виртуальные методы для получения типа узла, длины данных и количества строк.

Структура `LeafNode` (лист) соответствует конечной вершине по заданию. Хранит строку переменной длины через указатель `data` и её метаданные: `length` (счетчик символов) и `lineCount`. Управление памятью реализовано через запрет копирования и поддержку семантики перемещения.

Структура `InternalNode` (внутренний узел) соответствует промежуточной вершине. Содержит указатели на левое и правое поддеревья (`left`, `right`), а также кэшированные суммарные значения размера (`totalLength`) и количества строк (`totalLineCount`) своего поддерева. Метод `recalc()` обновляет эти значения.

Данные структуры формируют дерево, пригодное для промежуточного хранения текста в требуемом бинарном формате.

### Листинг 1 — Структуры дерева

```
struct Node {
    virtual NodeType getType() const = 0;

    // Быстрый доступ к статистике
    virtual int getLength() const = 0; // Вес в байтах
    virtual int getLineCount() const = 0; // Вес в строках (\n)

    virtual ~Node() = default;
};

struct LeafNode : public Node {
    int length;
    int lineCount; // Количество строк-1 (\n)
    char* data; // Указатель на строку в памяти (кучи)

    LeafNode(const char* str, int len);
    ~LeafNode() override;

    // Запрет копирования (от утечек)
    LeafNode(const LeafNode&) = delete;
    // Запрет присваивания копированием (от утечек)
    LeafNode& operator=(const LeafNode&) = delete;

    // Реализуем перемещающий конструктор и перемещающее
    присваивание:
    LeafNode(LeafNode&& other) noexcept;
    LeafNode& operator=(LeafNode&& other) noexcept;
```

```

NodeType getType() const override;
int getLength() const override;
int getLineCount() const override;
};

struct InternalNode : public Node {
    Node* left;
    Node* right;

    // Суммы детей
    int totalLength;
    int totalLineCount;

    InternalNode(Node* l, Node* r);
    ~InternalNode() override = default;

    NodeType getType() const override;
    int getLength() const override;
    int getLineCount() const override;

    void recalc(); // пересчитать totalLength и totalLineCount
};

```

### 2.3.2. Сохранение в бинарный файл

Процесс записи дерева в файл (`saveTree`) начинается с подготовки файла: он открывается для двоичной записи, полностью очищается, и в его начало помещается заголовок. Этот заголовок — это своеобразная визитная карточка файла: специальная сигнатура, версия формата и зарезервированное место для указателя на корень дерева. Далее начинается рекурсивный обход самого дерева. Система проходит от листьев к корню, сохраняя каждый узел в конец файла. Листовые узлы, хранящие строки, записываются целиком: сначала служебная информация (длина текста и количество строк), а затем сами символы. Внутренние узлы, являющиеся скелетом дерева, записываются иначе: в них сохраняются не данные, а адреса — смещения в файле, где находятся их левое и правое поддеревья. После того как всё дерево полностью записано и известен точный адрес корневого узла, программа возвращается к началу файла и записывает этот адрес в заранее оставленное для него место в заголовке. На этом сохранение завершено.

## Листинг 2 — Сохранения дерева в бинарный файл

```
void BinaryTreeFile::saveTree(const Tree& tree) {
    if (m_filename.empty()) {
        throw BinaryTreeFileError("No file opened for saving
(filename missing)");
    }

    if (is_open()) close();

    // Открываем файл в режиме truncate для очистки
    open(m_filename.c_str(), std::ios::binary | std::ios::in |
std::ios::out | std::ios::trunc);
    if (!is_open()) {
        std::ofstream out(m_filename.c_str(), std::ios::binary |
std::ios::trunc);
        if (!out)
            throw BinaryTreeFileError("Cannot open file for
writing");
        out.close();
        open(m_filename.c_str(), std::ios::binary | std::ios::in |
std::ios::out);
        if (!is_open())
            throw BinaryTreeFileError("Cannot reopen file for
writing");
    }

    // Заголовок: magic(4) + version(4) + rootOffset(8) (всего
16 байт)
    seekp(0, std::ios::beg);
    write(FILE_MAGIC, 4);
    write_le_uint32(FILE_VERSION);
    // Плэйсхолдер для смещения корня (8 байт)
    write_le_int64(OFFSET_NONE);

    // Пишем узлы (post-order), получаем смещение корня
    std::int64_t rootOffset =
writeNodeRecursive(tree.getRoot());

    // Обновляем реальный rootOffset в заголовок
    seekp(4 + 4, std::ios::beg); // magic(4) + version(4)
    write_le_int64(rootOffset);
    flush();
}
```

### 2.3.3. Чтение из бинарного файла

Обратный процесс — загрузка дерева из файла (**loadTree**) — читает файл в противоположном порядке. Сначала проверяется заголовок: если сигнатура и версия совпадают, система извлекает из него главную отправную точку — смещение корневого узла. Имея этот адрес, метод начинает рекурсивно восстанавливать дерево. Он переходит по указанному смещению, определяет тип узла и в зависимости от этого либо загружает строку данных (для листа), либо считывает два новых смещения на потомков (для внутреннего узла) и продолжает погружение. Таким образом, последовательно перемещаясь по сохранённым адресам, метод в точности реконструирует исходную древовидную структуру в памяти, собирая её из разрозненных частей, как по карте.

Листинг 3 — Чтение дерева из бинарного файла

```
void BinaryTreeFile::loadTree(Tree& tree) {
    if (!is_open()){
        throw BinaryTreeFileError("file not open");
        return;
    }

    tree.clear();

    seekg(0, std::ios::end);
    auto fileSize = static_cast<std::int64_t>(tellg());
    if (fileSize < 16) return; // Минимальный размер заголовка

    seekg(0, std::ios::beg);
    char magic[4]; // NOSONAR
    read(magic, 4);
    if (std::memcmp(magic, FILE_MAGIC, 4) != 0)
        throw BinaryTreeFileError("Bad file magic - not a tree
file");

    if (read_le_uint32() != FILE_VERSION)
        throw BinaryTreeFileError("Unsupported file version");

    std::int64_t rootOffset = read_le_int64();
    if (rootOffset == OFFSET_NONE) {
        tree.setRoot(nullptr);
        return;
    }
}
```

```
    Node* newRoot = readNodeRecursive(rootOffset, fileSize);
    tree.setRoot(newRoot);
}
```

### 2.3.4. Создание дерева из строки

Алгоритм осуществляет преобразование линейного текстового буфера в сбалансированную древовидную структуру данных, оптимизированную для эффективного выполнения операций редактирования текста. Процесс построения основан на рекурсивной стратегии разделения исходного текста на сегменты до ‘\n’, размер которых не превышает установленного ограничения в 4096 байт.

При необходимости разделения текстового фрагмента, превышающего максимальный размер листа, алгоритм выполняет поиск оптимальной точки разреза в окрестности середины сегмента, отдавая приоритет границам строк, обозначенным символами новой строки. Данный подход обеспечивает минимальную фрагментацию логических единиц текста. В случае отсутствия подходящих границ строк применяется разделение по точной середине сегмента.

Рекурсивное построение продолжается до полного преобразования всего текста в иерархическую структуру, где листовые узлы содержат непосредственно текстовые данные, а внутренние узлы агрегируют метаинформацию о своих поддеревьях. В процессе формирования внутренних узлов автоматически вычисляются показатели длины и количества строк, что впоследствии обеспечивает константное время доступа к этим характеристикам.

Результатом работы алгоритма является сбалансированное дерево с логарифмической глубиной, обеспечивающее высокую производительность при выполнении операций поиска, вставки и удаления текстовых фрагментов, что делает данную структуру особенно эффективной для работы с объемными текстовыми документами. Алгоритм запечатлен на Рис. 2.

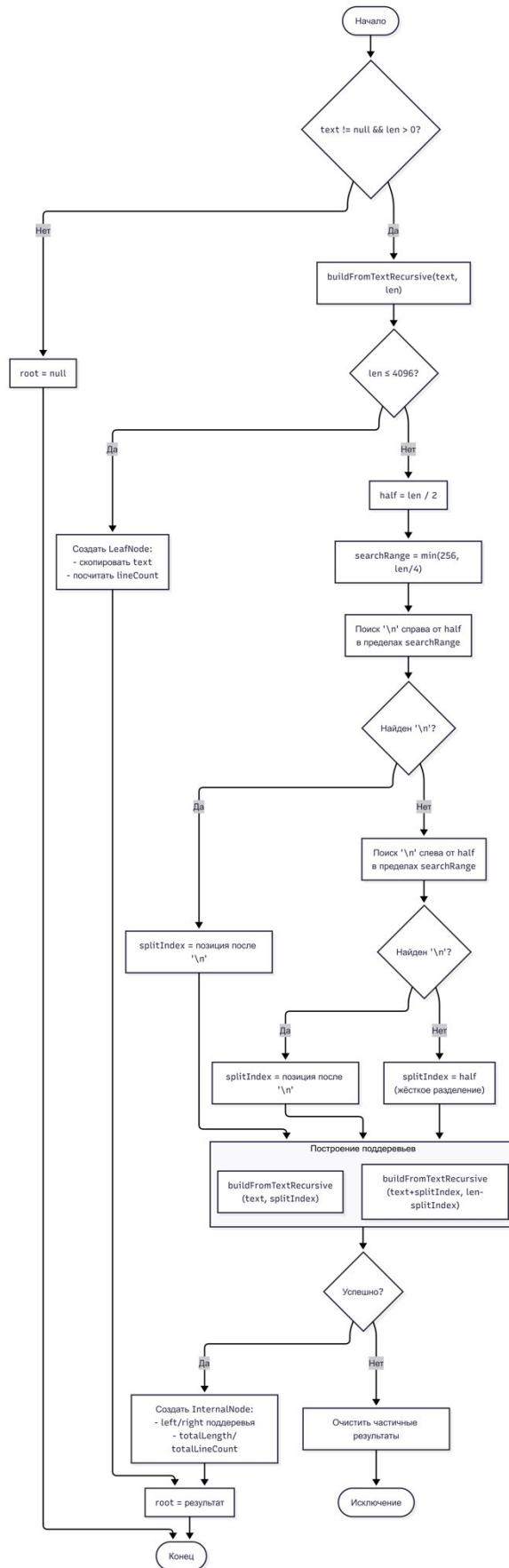


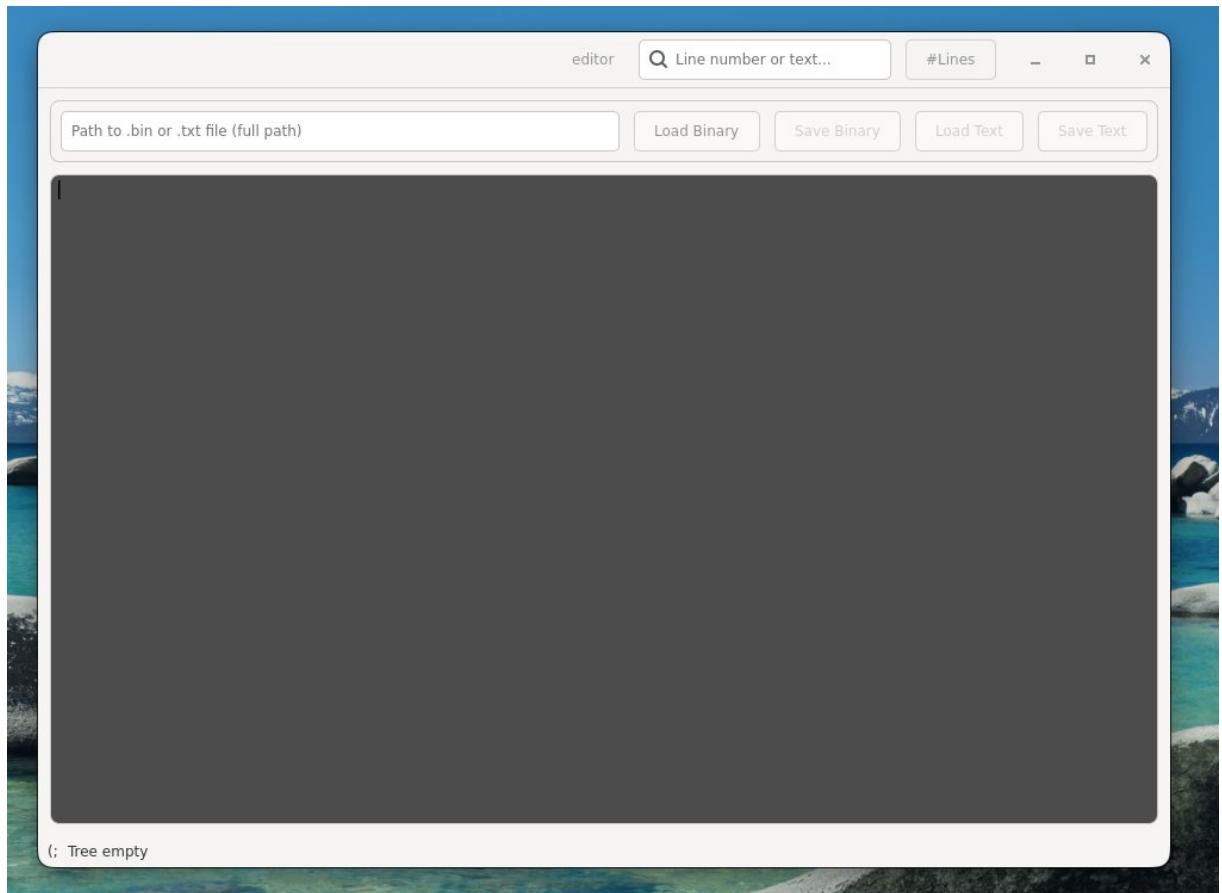
Рис. 2 Алгоритм создания дерева из строки.

### 3. ТЕСТИРОВАНИЕ

Проверим работу программы в соответствии с заданными требованиями.

При открытии программы должно появится окно в котором будет поле для ввода текста.

При запуске программы видим, что, действительно, открылось окно с текстовым полем которое готово для написания текста (См Рис. 3)



*Рис. 3 Открытие программы.*

После этого можем набрать какой то текст используя символы как латиницы так и кириллицы, программа должна спокойно обработать ввод с клавиатуры и отобразить все символы корректно (См. Рис. 4)

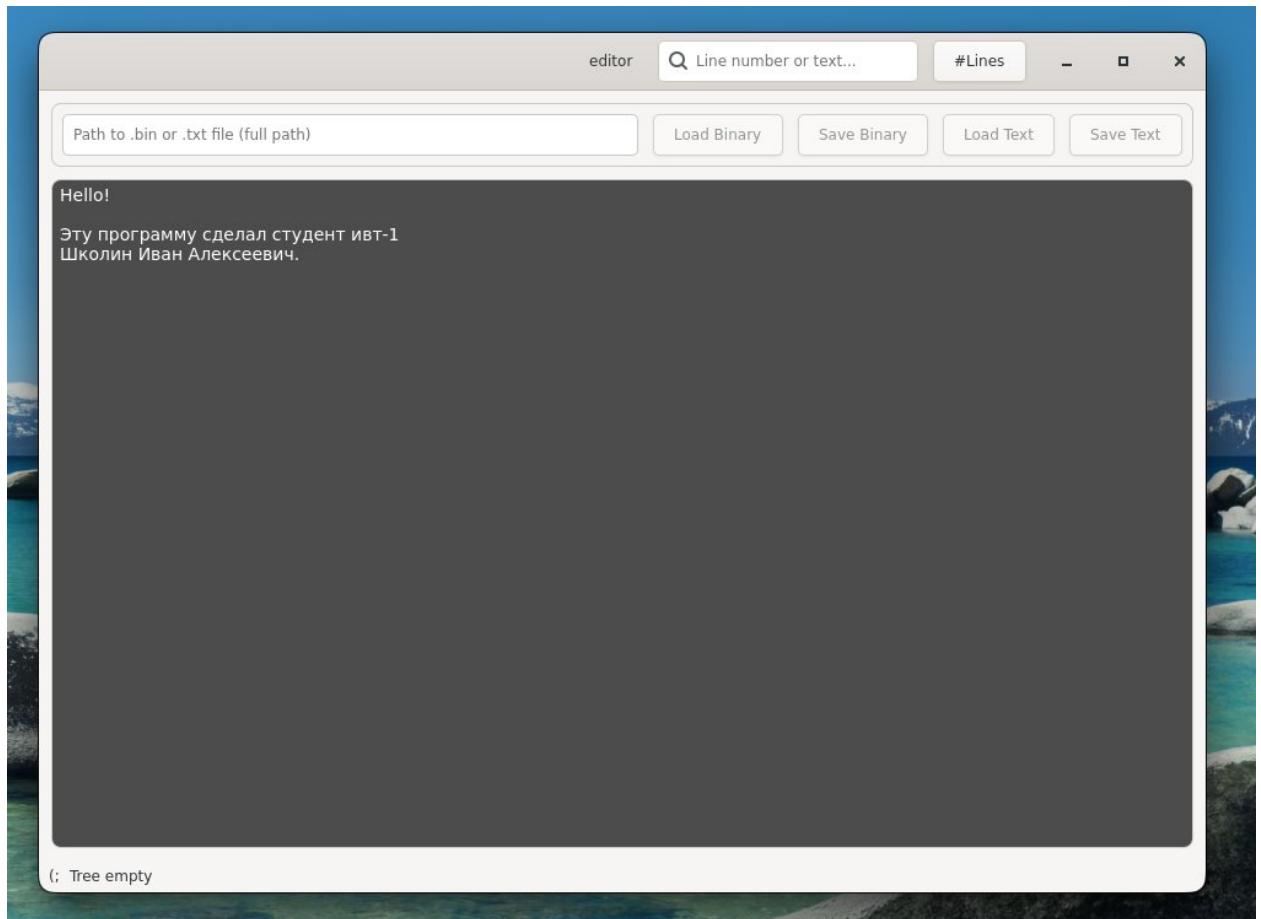


Рис. 4 Тестирование ввода текста.

После можно проверить как программа сохраняет и загружает файл, сохраним бинарный файл и посмотрим его содержимой с помощью vscode (См. Рис. 5) Также мы ожидаем четкую иерархию бинарного файла где в самом начале будет название структуры ‘TREE’, а дальше уже сериализованные части структуры, редактор vscode вряд ли сможет их отобразить, а дальше обычный UTF-8 текст, который скорее всего можно будет прочитать.

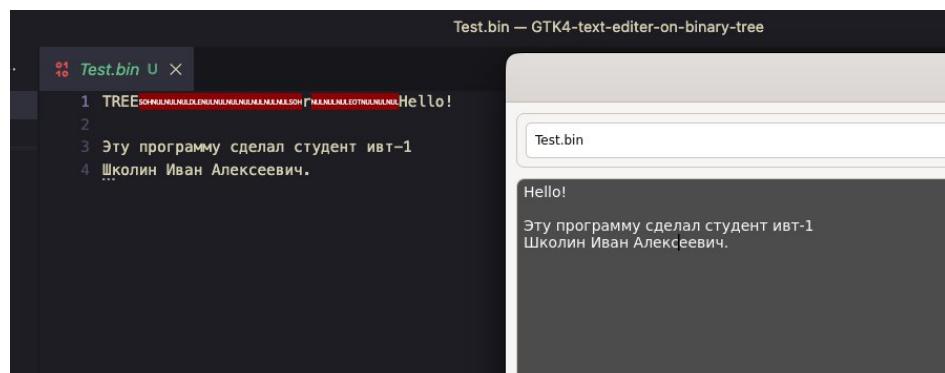
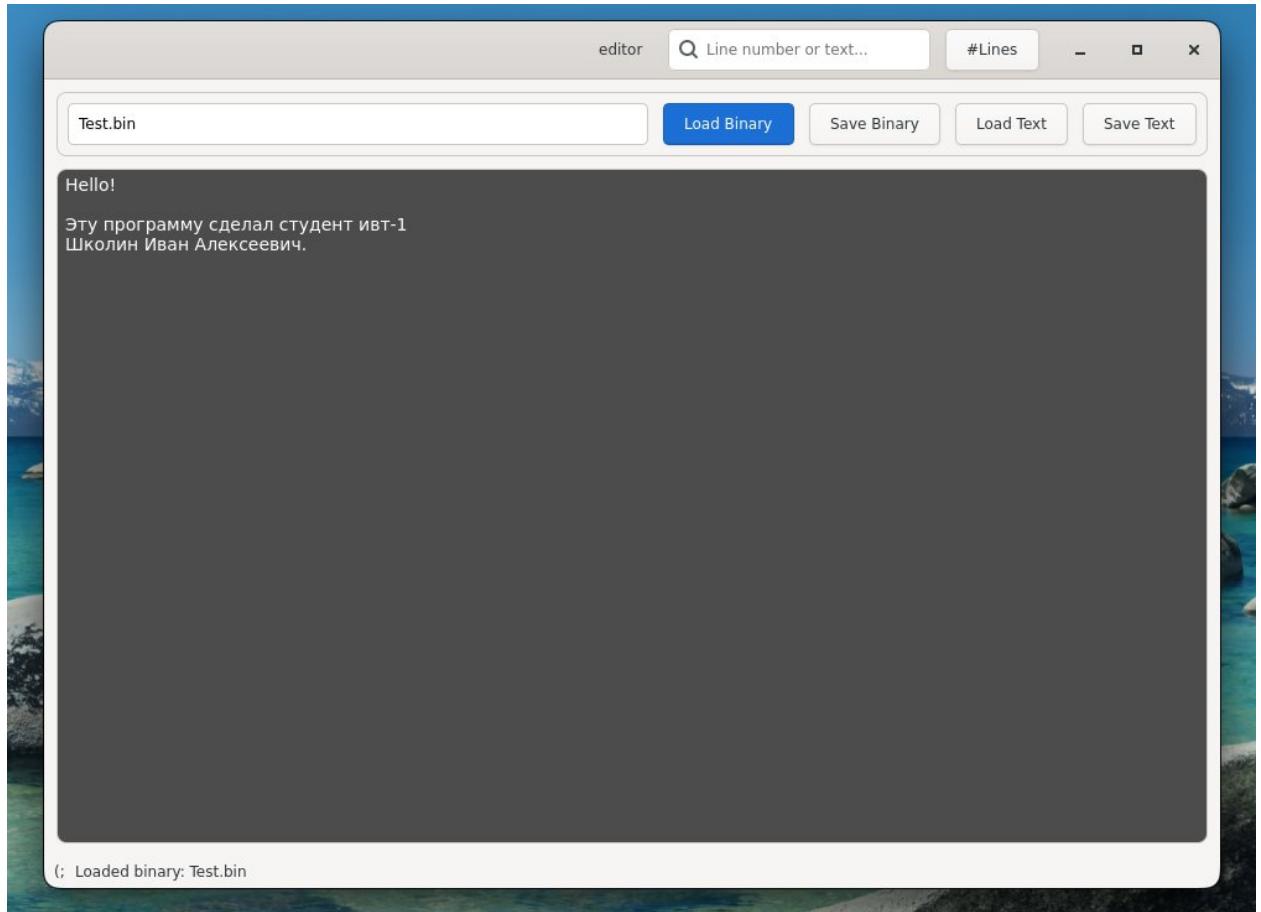


Рис. 5 Проверка сохранение в бинарный файл текста.

Теперь закроем редактор и проверим считает ли он из бинарного файла текст (См. Рис. 6)



*Рис. 6 Чтение из бинарного файла.*

Теперь попробуем найти часть текста со строкой `Иван` (См. Рис. 7)

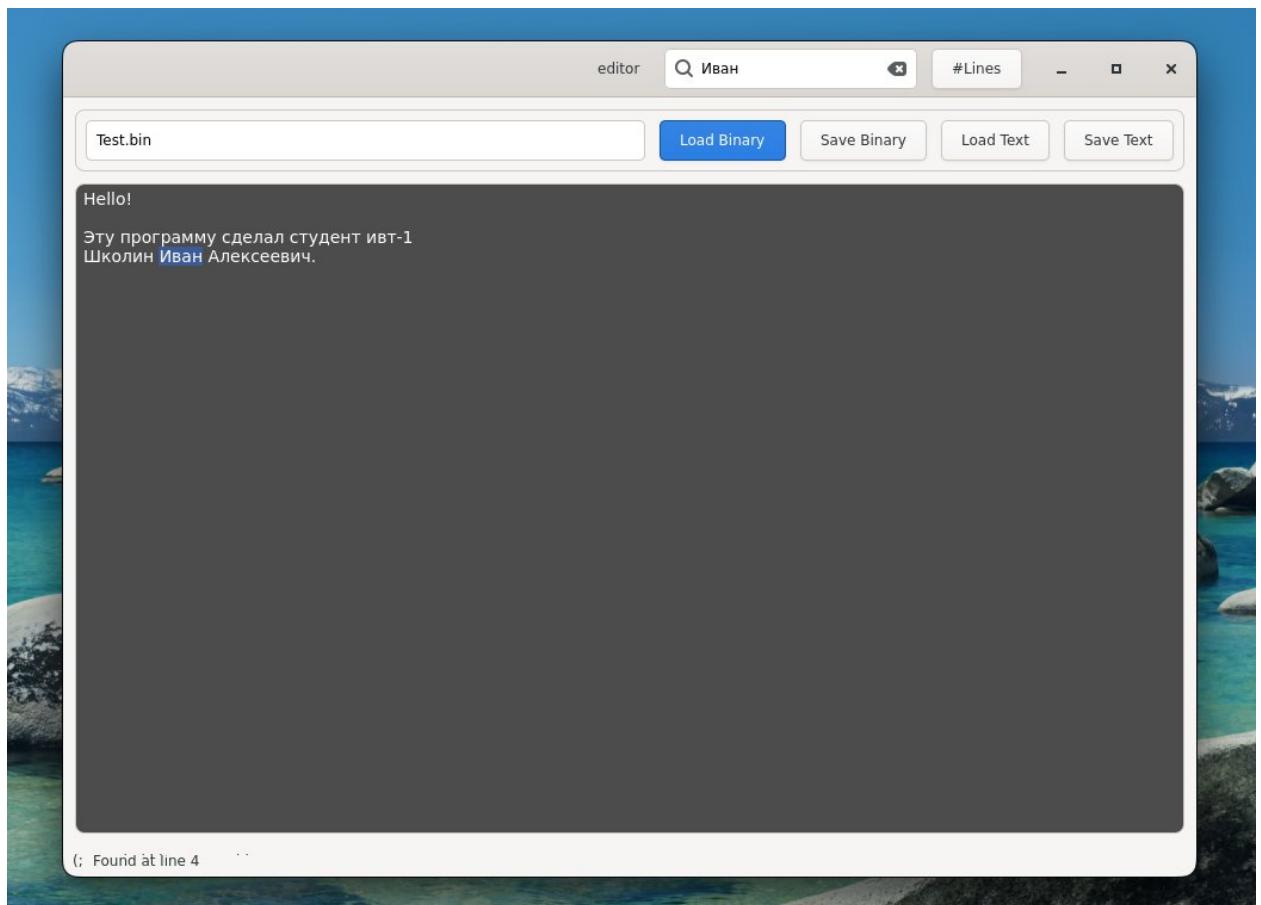


Рис. 7 Поиск по тексту.

Также проведем стресс-тест программы и попробуем открыть файл размером 500кб, который сгенерировал скрипт **gen\_file.cpp** (из `tests/`) редактор должен корректно его открыть используя функционал загрузки из текста и не переполнить буфер системы (См Рис. 8)

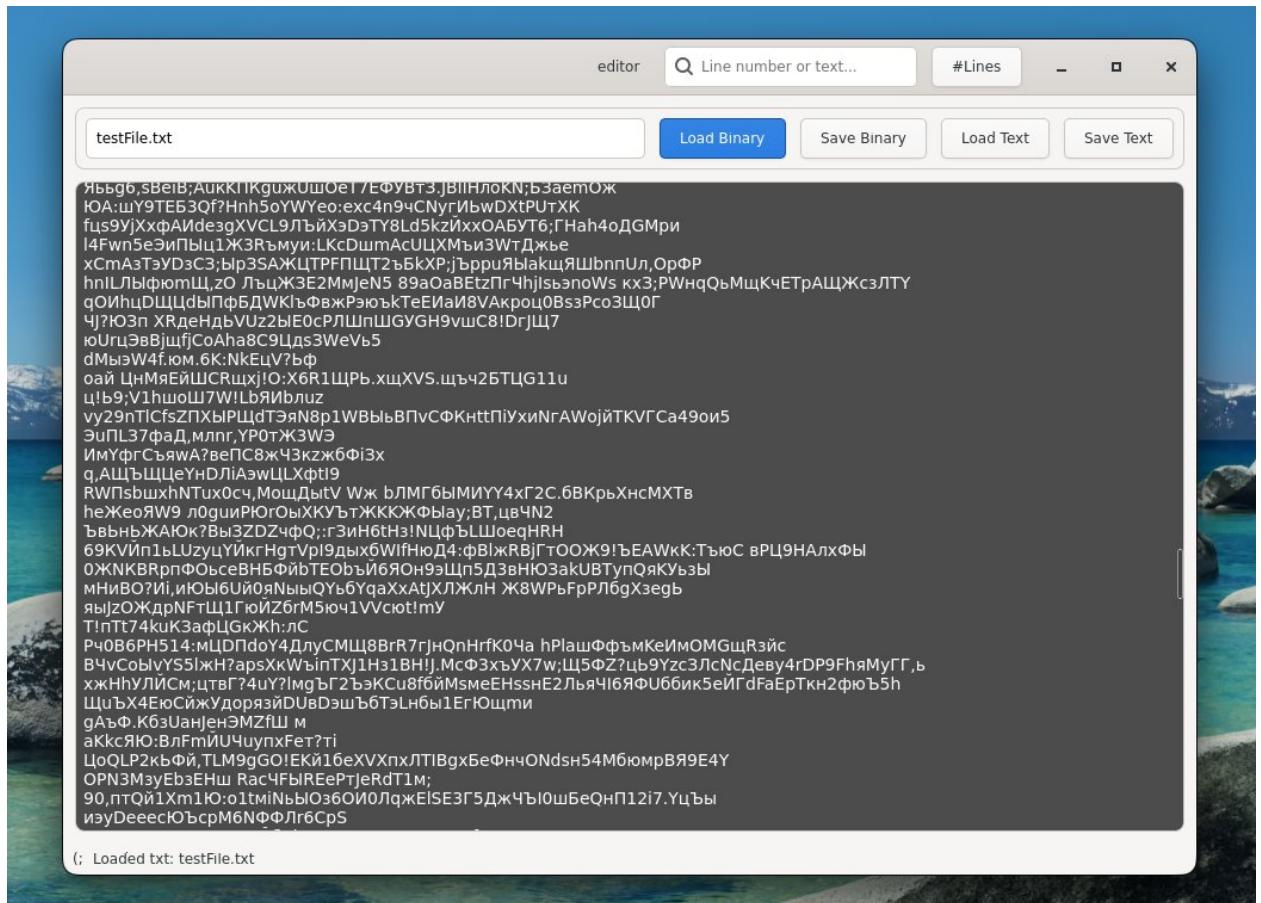


Рис. 8 Открытия файла размером 500КБ.

Попробуем сохранить наш огромный файл в бинарный формат (См Рис. 9), Ожидаем что редактор справится с этой задачей не переполнив буфер и успешно запишет на диск данные.

testFile.bin — GTK4-text-editer-on-binary-tree

1 **TREE** 5 **0PLr**, ThkЧ45VлйVwhknUEdUTTmьуTOЖOгAьPZe, эBMZgYбjэTD4вЛЙ8zf0fntHIEVЯE, bhe, Э  
2 Q5; xK9XHM5GtЬчиwKhnftM0ЭcsxЬгqяИMdxTfVшcV  
3 К7ыЕqи3ИЭyоtНЩBфUoзiйuBХDbdVTo  
4 nз хаюPnRi05NSWlG ,heч; u5Q1C0Q6ooBxГBQTУY;  
5 йшoтDхDуUспеcCTYiCtKyJ; оhзBзnзnзvрPb  
6 иMфРwуaшaз857 wEzExhлfвE; cPшhншЧaоg  
7 УкчкTзMтlfwzUJшAуxдfгyHsv; g eaБryCfo7  
8 LS.M.Pn;; TiсDзDиnZLJyGшhа5Dтpo  
9 юsCPxhу. LрZcl9tУiЧyQoAysMjM; yФoЧfR0lн4  
10 ъпAоKhGучumмbЕkд602AтxfoAфAгрж  
11 ОEуя3v; WwqM5kшLзIeDыbTшSH1gшw. тшшHлXф  
12 KоКшяШiaли?x6E55Nryи6xтcимя0Эж: Fph  
13 вMjяFуqЧq8w; gПeМpe: гzДB5x; чy  
14 дyб0тfewbaГapерPteLDCкdУs3Bex; ,иMб8  
15 4FysBМiгo txСeжiзfз3яяyзHxly3o4uS55tюi  
16 HSуcRbЬhe2tY Cpt8KptLy04eAьшj тЧKкnoиль  
17 kb56TшitБ, урJ2deAьГF! AишeГжwоцExшk66  
18 INм3zБNиNM7xt40n? Tфhни6o. 2q? nMЧ4RuTcaж  
19 utДao9n0K9E0fYiEoDwMФimyBшwOnQyдBv  
20 KNoXwрoкc90tB4ADrB  
21 Hи09LEX7у; D5xшH; h3зшЦшкN  
22 ГdrрRiaBw017gcpB5s1bтшNyиg5: 0:АзgMto  
23 ?BзAиfCaoEPt0x9s0уAй7j AжLtxвoC, yCDM:  
24 ?AGBne8! gHTцЬшyзяyS0y14MxhKp3z. Wbшyшt  
25 Чg r8w. fhРgсf8nRwуSpn0hе2yEgiбiИd  
26 xTouPm0FжTП1lfxAяBбesACaш5C3кц  
27 ESIшTк5xHtLe0tBдCиHwBзJ3, uсKd0L5L3Bъy  
28 щfj AEKсBкExh2шLшBзCуiЧtч. x. иuHиB60ш  
29 VEzG04Ns4cau! J? nMарKкA40 ? yчnчsC4xHb6l!w  
30 Y5xjnvLшuGшu9lCqkДffxMs?d7: XjБ Kdшm0xTb  
31 тжшBvQU? 2qthiR8geeeqJ40w01sBtф, XjDmA9  
32 Хlрion0vBvИH ИшuPv0ХTиNуЦdoушш0, FAMHdub  
33 9Ф0viц1DhрGкNp, eHnt  
34 тAЕoк! eTKT; рФ09Mш0iJш6 LM7T3MлаxФшFкx  
35 уgmдАвTМХМО1зарAшCкбdCфxуРphjCуbRУCYя  
36 зAоlХf9шшuо, CгtшgHnYшBw. oСiжzяяavd. xBдяеBцLjоxб0жFрь  
37 АютG84vQn0X? DqHnK00XrEmz7 8utчUdeяkш? Z8pCрРММcTmшsшypxtBdя

editor  Line number or text... #Line

testFile.bin

Рис. 9 Сохранения файла размером 500КБ в бинарный файл.

И наконец попробуем считать огромный файл не допустив переполнения буфера и прочих ошибок (См Рис. 10)

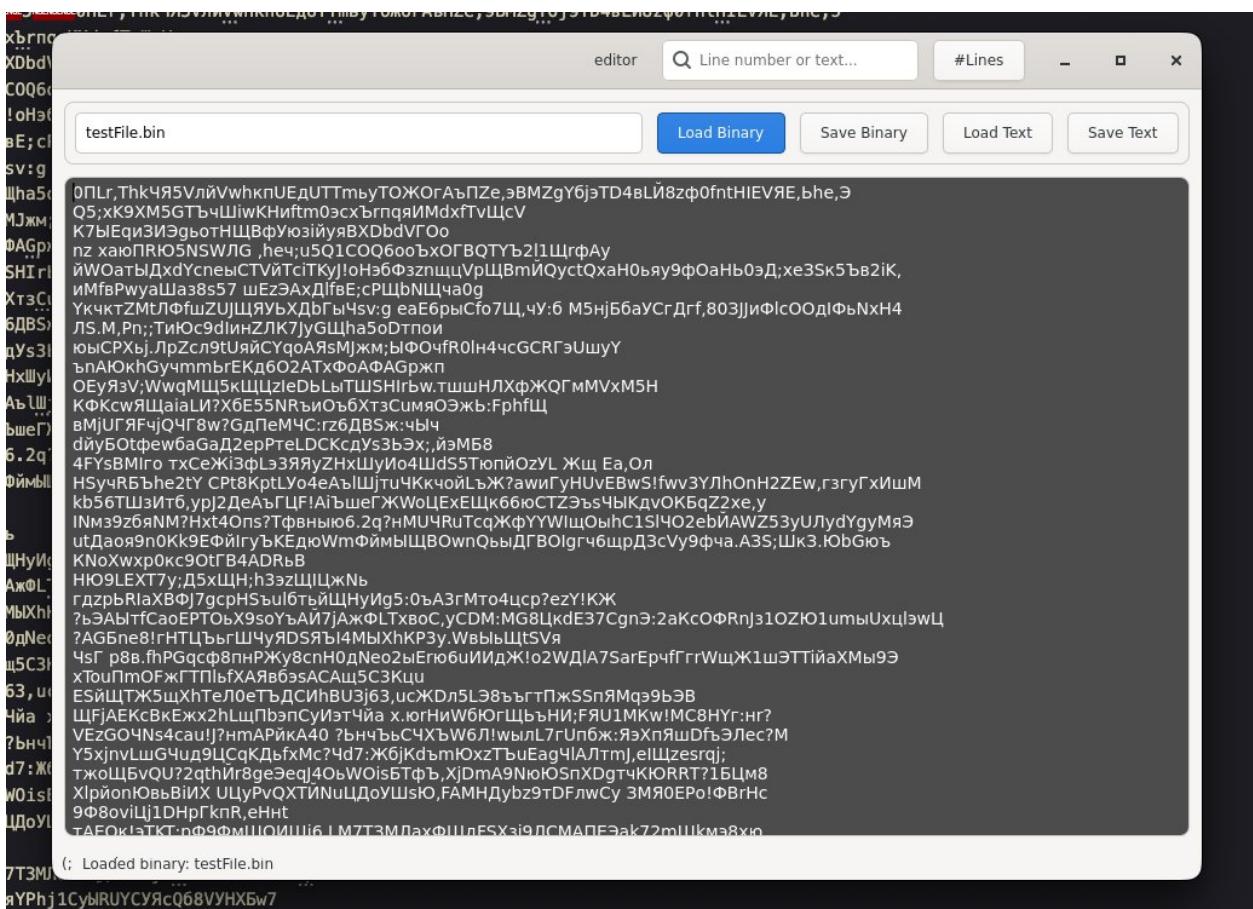


Рис. 10 Открытие бинарного файла весом в 500КБ.

Редактор корректно справился со всеми тестами и не допустил крашей, а также умеренно уместился в оперативно памяти, не занимая больше 150мб оперативно памяти при открытии крупного файла (в простое занимает 50мб), не испытав утечек памяти и переполнения буфера при открытии огромного файла и его сохранении в бинарный файл.

## **4. ИТОГИ ПРОДЕЛАННОЙ РАБОТЫ**

В рамках настоящей курсовой работы было реализовано программное обеспечение текстового редактора, ключевой особенностью которого является использование бинарного дерева в качестве внутренней структуры для хранения и обработки текстовых данных. Цель работы — создание наглядного приложения, демонстрирующего практическое применение сложных структур данных и алгоритмов сериализации в контексте разработки пользовательских интерфейсов — была успешно достигнута.

Основным результатом работы является законченное кроссплатформенное приложение, построенное по модульному принципу. Проект включает в себя логическое ядро, реализующее заданную структуру бинарного дерева и операции с ним, подсистему работы с бинарными файлами специального формата, а также графический интерфейс, обеспечивающий интуитивное взаимодействие с пользователем. Для обеспечения надежности и переносимости была настроена автоматизированная система сборки и репродуцируемое окружение разработки, а также создан комплекс модульных и интеграционных тестов

Разработанный редактор наглядно иллюстрирует, как теоретические концепции организации данных и работы с файлами могут быть интегрированы в современное приложение с графическим интерфейсом. Практическая ценность работы подтверждается полным циклом реализованных функций: от загрузки и парсинга текстового файла в бинарное дерево до интерактивного редактирования и сохранения результатов.

Перспективы развития проекта могут включать следующие направления:

- Добавление системы отмены и повтора действий, основанной на сохранении снимков состояния дерева.
- Развитие интерфейса: поддержка вкладок, настройка темы оформления, расширенные возможности форматирования текста.
- Поддержка шифрования бинарного файла.

Таким образом, данный проект не только представляет собой законченное решение в рамках учебной задачи, но и формирует основательную базу для дальнейшего изучения и совершенствования, демонстрируя тесную связь между фундаментальными алгоритмами, системным программированием и разработкой удобных пользовательских приложений.

## **СПИСОК ЛИТЕРАТУРЫ**

1. **Raymond, E. S.** The Art of Unix Programming / E. S. Raymond. – Boston : Addison-Wesley, 2003. – 544 p. – ISBN 0-13-142901-9.
2. **Knuth, D. E.** The Art of Computer Programming. Vol. 1: Fundamental Algorithms / D. E. Knuth. – 3rd ed. – Reading, Massachusetts : Addison-Wesley, 1997. – 650 p. – ISBN 0-201-89683-4.
3. **Stroustrup, B.** *The C++ Programming Language* / B. Stroustrup. – 4th ed. – Boston : Addison-Wesley, 2013. – 1360 p. – ISBN 978-0-321-56384-2.
4. **Boehm, H.-J.; Atkinson, R.; Plass, M.** Ropes: an Alternative to Strings // Software: Practice and Experience. – 1995. – Vol. 25, № 12. – P. 1315–1330. – DOI: 10.1002/spe.4380251203