Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6
ПО ДИСЦИПЛИНЕ «ТИПЫ И СТРУКТУРЫ ДАННЫХ»

Обработка деревьев

Вариант № 6

Студент: Бондарева **В. А.**

Группа: ИУ**7-34Б**

Преподаватель: **Никульшина Т. А.**

2025 г.

Условие задачи

Построить частотный словарь из слов текстового файла в виде дерева двоичного поиска. Вывести его на экран в виде дерева. Осуществить поиск указанного слова в дереве и в файле. Если слова нет, то (по желанию пользователя) добавить его в дерево и, соответственно, в файл. Сравнить время поиска слова в дереве и в файле. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность алгоритмов сортировки и поиска значения в зависимости от высоты деревьев и степени их ветвления (при равном количестве узлов).

Описание ТЗ

Исходные данные

Исходными данными программы являются:

1. Строка, содержащая единственное целое положительное число, являющееся пунктом меню.
   Пользователю предоставляются следующие опции меню:

```
0. Выход
1. Считать дерево из файла
2. Вывести дерево
3. Добавить узел дерева
4. Удалить узел дерева
5. Очистить все дерево
6. Найти слово в файле
7. Найти слово в дереве
8. Сравнить время поиска слова дерево/файл
9. Сравнить эффективность
```

Листинг 1. Опции меню.

2. Строка, содержащая имя файла, над которым необходимо произвести какие-либо операции.
3. Строка, содержащая слово, над которым необходимо произвести какие-либо операции, например, добавить узел с этим словом в дерево.

Результат

Результатом выполнения программы являются:

1. Вывод меню для выбора опции (главное меню или диалоговое меню с предложением добавить узел в дерево).
2. Таблицы сравнений, такие как таблица сравнения памяти и времени поиска слова в файле и в дереве, таблица сравнения эффективности.
3. Текущее состояние дерева.
4. Сообщения о статусе выполнения программы и ошибках.

Способ обращения к программе

Пользователь обращается к программе при помощи исполняемого файла app.exe.

Возможные аварийные ситуации и ошибки пользователя

Все возможные аварийные ситуации о ошибки пользователя описаны типом status_t, включающим в себя коды возврата программы:

1. ERR_IO: ошибка ввода/вывода данных программы.
2. ERR_RANGE: неверно указано количество чего-либо.
3. ERR_MEM: ошибка при работе с памятью.
4. ERR_ARGS: ошибка при работе с аргументами функции.
5. ERR_NOT_FOUND: ошибка при отсутствии необходимого элемента.
6. ERR_FILE: ошибка при работе с файлом.
7. ERR_EMPTY_TREE: ошибка при попытке работы с пустым деревом.

Описание внутренних структур данных

```
typedef struct tree_node
{
    char *word;
    size_t counted;
    struct tree_node *left;
    struct tree_node *right;
} tree_node_t;
```

Листинг 2. Внутренние структуры данных.

В данной лабораторной работе единственной необходимой структурой является структура бинарного дерева поиска tree_node_t. Поле word – динамический массив типа char, т.е. слово, которое хранит узел дерева, поле counted – это число, которое описывает сколько раз это слово встретилось в файле или сколько раз его добавил в дерево пользователь. Поля left и right – указатели на левое и правое поддерево соответственно.

В листинге №3 представлены сигнатуры основных функций.

```
// функции для работы с бинарным деревом
status_t insert_tree_node(tree_node_t **root, const char *word);
status_t delete_tree_node(tree_node_t **root, const char *word);
status_t clear_tree(tree_node_t **root);
status_t find_word_in_tree(tree_node_t *root, tree_node_t **target_root,
const char *word);
status_t print_pretty_tree(tree_node_t *root);
size_t calculate_tree_memory(tree_node_t *root);

// функции для работы с файлом
status_t read_tree_from_file(tree_node_t **root, char *filename);
status_t find_word_in_file(char *filename, char *word, ssize_t *word_num);
status_t insert_word_to_file(const char *filename, char *target_word);
```

```
// функции сравнения
status_t compare_find_operation(void);
status_t compare_sort_operation(void);
```

Листинг 3. Сигнатуры основных функций.

Алгоритм

Помимо основного алгоритма программы (main.c) основополагающими алгоритмами для реализации заданного функционала являются:

- алгоритм добавления элемента в дерево
- алгоритм удаления элемента из дерева
- алгоритм поиска элемента в дереве
- алгоритм чтения дерева из файла
- алгоритм сравнения времени поиска слова в файле и в дереве
- алгоритм сравнения сортировки последовательности слов в дереве и в файле

Рассмотрим каждый алгоритм.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Происходит рекурсивный спуск по дереву: пока не найдено место для вставки нового слова, это слово сравнивается со словом в текущем узле.
3. Если слова совпадают, то счетчик counted просто увеличивается на единицу.
4. Если текущее слово больше или меньше искомого, то функция рекурсивно вызывает саму себя, передавая адрес указателя на соответствующего потомка.
5. Если место для вставки найдено (текущий узел пуст), и в процессе выполнения алгоритма не было обнаружено ни одно совпадение слов, то данный узел инициализируется и ему присваивается искомое слово.

Листинг 4. Алгоритм добавления элемента в дерево.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Искомое слово сравнивается со словом в текущем узле. Если оно лексикографически меньше, то осуществляется рекурсивный переход к поиску в левом поддереве, иначе – в правом.
3. Если искомое слово равно слову в текущем узле (совпадение найдено), то:
    a. Если у текущего узла нет потомков, то он освобождается.
    b. Если у текущего узла нет левого потомка (или правого), то указатель на существующего единственного потомка сохраняется во временную переменную, память, которая занята текущим узлом, освобождается, а удаляемый узел «заменяется» на своего существующего и ранее сохраненного потомка.
    c. Если у текущего узла два потомка: и левый, и правый, то сначала необходимо найти узел с наименьшим значением относительно удаляемого узла (самый левый в правом поддереве), таким образом иерархия дерева не нарушается. После того как необходимый узел-преемник найден, указателю на удаляемый узел присваивается значение указателя на данного преемника. Если преемник был непосредственно правым потомком удаляемого узла, то правое поддерево преемника «пришивается» к его родителю (удаляемому узлу) как новое правое

поддерево. Если преемник был глубже в правом поддереве, то его правое поддерево «пришивается» к левому указателю его родителя. После настоящий узел-преемник освобождается.

Листинг 5. Алгоритм удаления элемента из дерева.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Осуществляется итеративный обход дерева: цикл работает, пока текущий узел не NULL, код возврата корректен или не найден необходимый узел.
3. На каждой итерации слово из текущего узла сравнивается с искомым словом. Если слова совпали, то необходимый узел найден, происходит выход из цикла.
4. Если искомое слово меньше, поиск продолжается в левом поддереве, иначе – в правом.

Листинг 6. Алгоритм поиска элемента в дереве.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Перед чтением содержимого файла текущее дерево очищается от старых данных.
3. Осуществляется открытие файла в режиме чтения.
4. Функция входит в цикл, который считывает из файла последовательности символов, разделенные пробельными символами (слова). Каждое успешно прочитанное слово передается в функцию для вставки слова в дерево. Цикл продолжается до тех пор, пока в файле есть слова.

Листинг 7. Алгоритм чтения дерева из файла.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Для каждого типа дерева (сбалансированное, случайное и вырожденное), которые представлены в виде трех текстовых файлов с фиксированными именами:
    a. Загружаются слова из файла в массив для тестирования.
    b. На основе массива слов строится двоичное дерево.
    c. Измеряется время поиска каждого слова в массиве в построенном дереве.
    d. Вычисляется среднее время поиска для одного слова.
3. После получения данных для всех трех видов деревьев результат измерений выводится в виде таблицы.

Листинг 8. Алгоритм сравнения времени поиска слова в файле и в дереве.

1. Инициализируется код возврата и проверяется корректность входных данных.
2. Для каждого типа дерева (сбалансированное, случайное и вырожденное), которые представлены в виде трех текстовых файлов с фиксированными именами:
    a. Строится двоичное дерево.
    b. Выполняется инфиксный обход (1000 итераций) для получения статистических результатов.
    c. Измеряется общее время всех обходов и вычисляется среднее время одного обхода.
3. После получения среднего времени обхода для всех трёх типов деревьев результат измерений выводится в виде таблицы.

Листинг 9. Алгоритм сравнения сортировки последовательности слов в дереве и в файле.

Тесты

Позитивные тесты

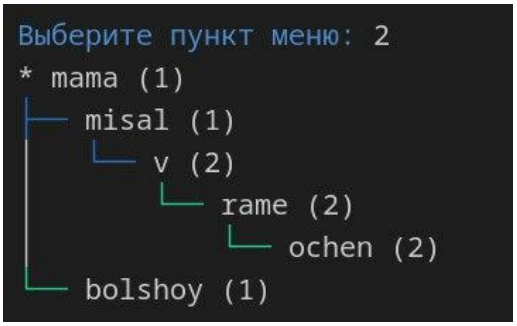
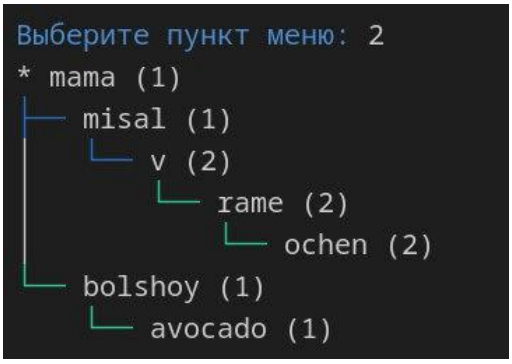
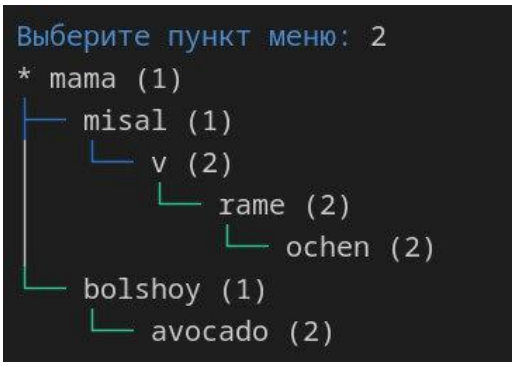
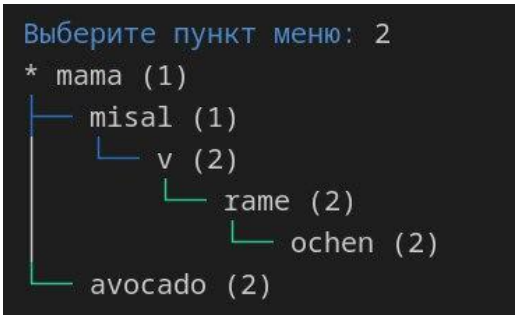
Номер теста	Описание	Ожидаемый результат
1	Попытка считывания дерева из валидного файла.	Дерево успешно считано из файла.
2	Вывод дерева.	
3	Добавление нового узла дерева.	
4	Добавление существующего узла дерева.	
5	Удаление существующего узла дерева.	

Таблица 1. Позитивные тесты.

Негативные тесты

Номер теста	Описание	Ожидаемый результат
1	Попытка работы с несуществующим файлом.	Выберите пункт меню: 8 Введите имя файла: lmk **Произошла ошибка при работе с файлом**
2	Попытка вывести пустое дерево.	Выберите пункт меню: 2 **Произошла ошибка: невозможно выполнить для пустого дерева**
3	Попытка удаления несуществующего элемента в дереве.	Выберите пункт меню: 4 Введите слово: avocado **Произошла ошибка при поиске чего-нибудь: мы ничего не нашли :(**
4	Попытка поиска несуществующего элемента в дереве.	Выберите пункт меню: 7 Введите слово для поиска: avocado **Произошла ошибка при работе с аргументами функции**

Таблица 2. Негативные тесты.

Оценка эффективности

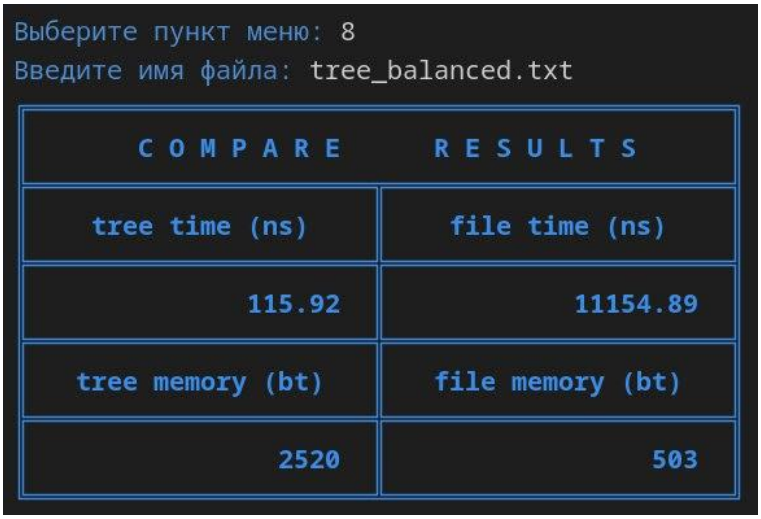


Рис. 1. Сравнение времени поиска слова в дереве или в файле для сбалансированного дерева.

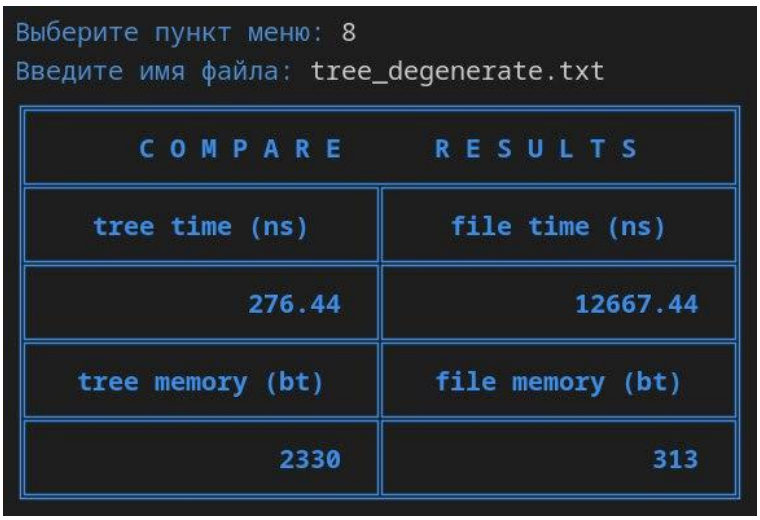


Рис. 2. Сравнение времени поиска слова в дереве или в файле для вырожденного дерева.

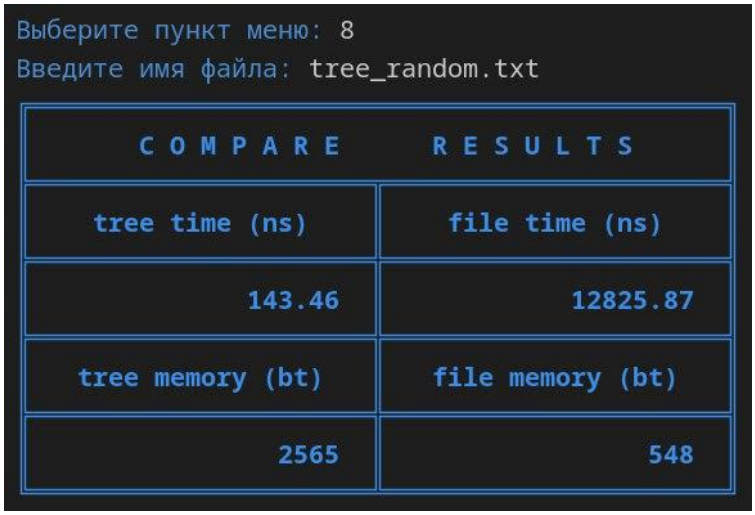
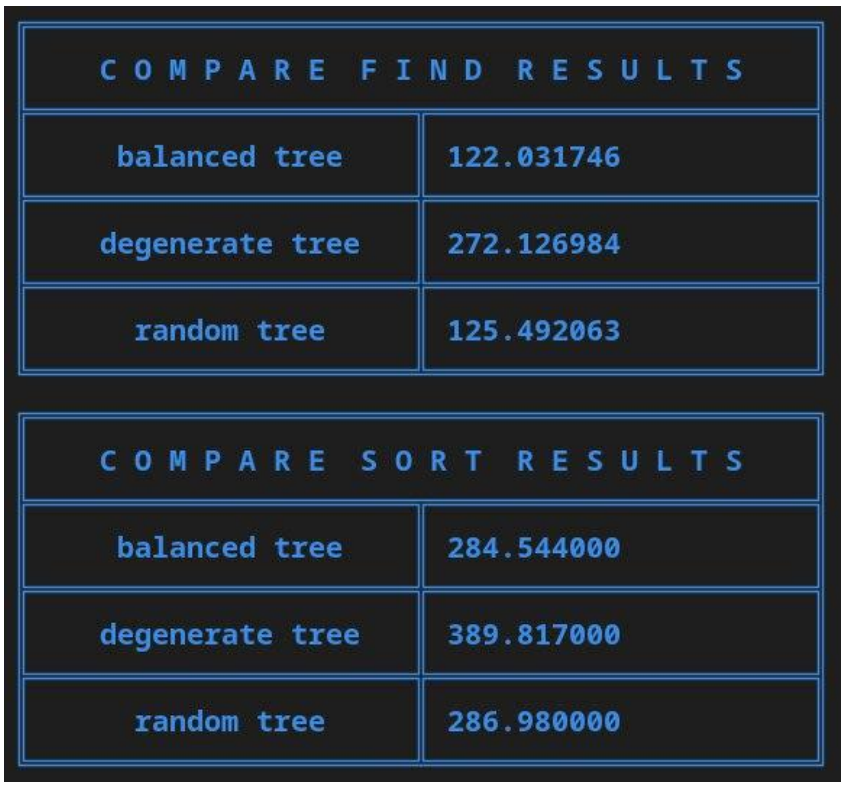


Рис. 3. Сравнение времени поиска слова в дереве или в файле для случайного дерева.

На основании данных рисунков 1, 2, 3 можно сделать вывод о скорости поиска слова в дереве и в файле.

Для сбалансированного дерева поиск с помощью двоичного дерева работает на 11038.97 наносекунд быстрее, то есть быстрее на 9623%. Для вырожденного дерева эти показатели равняются 12391 наносекунд и 4582%, а для случайного – 12682.41 наносекунд и 8940%. В среднем абсолютная разница между поиском слова в двоичном дереве и поиском слова в файле составляет 12037.46 наносекунд, а относительная – 7715%.

Однако в контексте работы с памятью файл оказался более эффективным: в случае сбалансированного дерева файл занимает в 5.01 раз меньше, в случае вырожденного – в 7.44 раз, в случае случайного – в 4.68 раз. В среднем файл занимает в 5.71 раз меньше памяти, чем двоичное дерево.



COMPARE FIND RESULTS	
balanced tree	122.031746
degenerate tree	272.126984
random tree	125.492063

COMPARE SORT RESULTS	
balanced tree	284.544000
degenerate tree	389.817000
random tree	286.980000

Рис. 4. Сравнение времени поиска слова и сортировки для трех видов деревьев.

На рисунке 4 представлены данные о времени поиска слова и сортировки для трех видов деревьев: сбалансированного, вырожденного и случайного. Исходя из них можно сделать вывод о том, что быстрее всего сортировка и поиск работает для сбалансированного и случайного деревьев, в то время как вырожденное дерево показывает худший результат: поиск в вырожденном дереве работает в 2.22 раз медленнее, а сортировка – в 1.36 раз.

Выводы

Двоичное дерево поиска демонстрирует абсолютное превосходство по скорости выполнения операции поиска по сравнению с поиском в файле, обеспечивая ускорение в среднем в 77 раз. Максимальная эффективность достигается при работе со сбалансированным деревом, что подтверждает теоретическую логарифмическую сложность алгоритма. Однако с точки зрения экономии памяти файловое хранение оказывается более эффективным, занимая в среднем в 5.7 раза меньше памяти. При сравнении различных структур деревьев наилучшие результаты по времени поиска и сортировки показывают сбалансированное и случайное деревья, в то время как вырожденное дерево, по сути являющееся линейным списком, работает значительно медленнее - поиск в нём происходит в 2.2 раза дольше. Таким образом, выбор структуры данных зависит от требуемой скорости выполнения операций и объёмом потребляемой памяти, где бинарное дерево поиска является оптимальным решением для задач, требующих высокой скорости доступа к данным, а файловое хранение - для случаев с ограниченными ресурсами памяти.

Контрольные вопросы

1. *Что такое дерево? Как выделяется память под представление деревьев?*
   Дерево – структура данных, состоящая из узлов, один из которых является корневым, а остальные узлы организованы в виде иерархии поддеревьев и связаны друг с другом ребрами. Каждый узел имеет ноль или более потомков (в случае бинарного дерева максимальное количество потомков равно двум). Каждый узел дерева содержит информацию (значение) и указатели на своих потомков (если потомков нет, то указатели нулевые).
   Память под представление поддеревьев обычно выделяется динамически для каждого нового добавленного узла.
2. *Какие бывают типы деревьев?*
   - Двоичное дерево: каждый узел имеет не более двух потомков
   - Дерево двоичного поиска: упорядоченное двоичное дерево, узлы упорядочены так, что для каждого узла все узлы в его левом поддереве меньше его, а в правом – больше
   - N-арное дерево: каждый узел может иметь произвольное количество потомков
   - AVL дерево (Адельсон-Вельский и Ландис): сбалансированное двоичное дерево поиска
3. *Какие стандартные операции возможны над деревьями?*
   Стандартные операции над деревьями включают в себя:
   - добавление нового узла в дерево
   - удаление существующего узла из дерева
   - поиск узла с определенным значением
   - обход дерева, посещение всех узлов дерева в определенном порядке
   - вывод дерева (в определенном формате)
4. *Что такое дерево двоичного поиска?*
   Дерево двоичного поиска – бинарное дерево, в котором каждый узел имеет не более двух потомков. При этом для каждого узла все узлы в левом поддереве меньше текущего узла, а все узлы в правом поддереве больше текущего узла, что делает дерево двоичного поиска эффективной структурой данных для поиска, вставки и удаления элементов, так как оно обеспечивает логарифмическую сложность этих операций в общем случае.