ГУАП

КАФЕДРА № 14

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

ассистент		П. В. Шпигун
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 4

ДВОИЧНЫЕ ДЕРЕВЬЯ ПОИСКА

по курсу: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 1446 22.12.2025 А. С. Пырву

подпись, дата — инициалы, фамилия

Санкт-Петербург 2025

Цель работы: Изучить основные алгоритмы работы с двоичными деревьями поиска (ДДП) и освоить их на практике. Проверить работу алгоритмов на различных наборах данных.

Постановка задачи: Разработать программу в соответствии с заданием к лабораторной работе. Данные для построения двоичных деревьев вводить из заранее созданных файлов.
По входным данным из файла построить двоичное дерево поиска.
Во всех работах разработать функцию удаления дерева.
Вариант задания выбирается по номеру студента в списке подгруппы.
Для построения ДДП использовать либо алгоритм построения рандомизированного дерева или алгоритм построения АВЛ-дерева (алгоритмы имеются в лекциях)
Для отображения результатов работы разработанной программы использовать
Функции их файла printtree.c

Вариант 21: Построить дерево из набора слов на русском языке. Удалить все узлы со словами, имеющими четное количество гласных букв

Формализация:

Константы:

WORD_MAX = 100 — максимальная длина слова.
LINE_MAX = 1024 — максимальная длина строки.

Структуры данных:

tree_t — узел двоичного дерева, содержащий слово и указатели на левого и правого потомков.
Trunk — вспомогательная структура для красивого вывода дерева в консоль.

Реализованные функции:

tree_create — создание нового узла дерева.
tree_add — добавление слова в дерево.
tree_attach — прикрепление поддерева к существующему узлу.
need_delete — проверка, нужно ли удалять слово (чётное количество гласных).
tree_find_to_delete — поиск узла для удаления.
tree_delete_words — удаление всех подходящих слов из дерева.
tree_print_dot_recursive, tree_print_dot — сохранение дерева в DOT-формат.
printTree, showTrunks — вывод дерева в консоль.
tree_free — освобождение памяти.

Использованные библиотеки:
```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <stdbool.h>
#include <wchar.h>
#include <wctype.h>
```

Листинг программы:

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <stdbool.h>
#include <wchar.h>
#include <wctype.h>

#define WORD_MAX 100
#define LINE_MAX 1024

typedef struct tree tree_t;

struct tree
{
    wchar_t word[WORD_MAX];
    struct tree *left;
    struct tree *right;
};

static tree_t *tree_create(const wchar_t *word)
{
    tree_t *node = (tree_t *)malloc(sizeof(tree_t));
    if (!node)
    {
        return NULL;
    }

    wcsncpy(node->word, word, WORD_MAX - 1);
    node->word[WORD_MAX - 1] = L'\0';
    node->left = NULL;
    node->right = NULL;

    return node;
}

static void tree_add(tree_t **tree, const wchar_t *word)
{
    while (*tree != NULL)
    {
        if (wcscmp(word, (*tree)->word) < 0)
        {
            tree = &(*tree)->left;
        }
        else
        {
            tree = &(*tree)->right;
        }
    }

    *tree = tree_create(word);
}

static void tree_attach(tree_t **tree, tree_t *node)
{
    if (node == NULL)
    {
        return;
    }

    while (*tree != NULL)
```

```
    {
        if (wcscmp(node->word, (*tree)->word) < 0)
        {
            tree = &(*tree)->left;
        }
        else
        {
            tree = &(*tree)->right;
        }
    }

    *tree = node;
}

static bool need_delete(const wchar_t *word)
{
    int k = 0;

    for (const wchar_t *c = word; *c != L'\0'; c++)
    {
        wchar_t ch = towlower((wint_t)*c);

        if (wcschr(L"уёеыаоэяию", ch) != NULL)
        {
            k++;
        }
    }

    return (k % 2) == 0;
}

static tree_t *tree_find_to_delete(tree_t *tree, tree_t **parent)
{
    if (tree == NULL)
    {
        return NULL;
    }

    if (need_delete(tree->word))
    {
        return tree;
    }
    else
    {
        *parent = tree;
        tree_t *v = tree_find_to_delete(tree->left, parent);

        if (v != NULL)
        {
            return v;
        }

        *parent = tree;
        return tree_find_to_delete(tree->right, parent);
    }
}

static void tree_delete_words(tree_t **tree)
{
    tree_t *parent = NULL;
    tree_t *to_delete = tree_find_to_delete(*tree, &parent);

    while (to_delete != NULL)
    {
        tree_t *del = to_delete;
```

```
        if (parent == NULL)
        {
            tree_t *right = (*tree)->right;
            *tree = (*tree)->left;
            tree_attach(tree, right);
        }
        else
        {
            tree_t *right = to_delete->right;

            if (parent->left == to_delete)
            {
                parent->left = to_delete->left;
                tree_attach(&parent->left, right);
            }
            else
            {
                parent->right = to_delete->left;
                tree_attach(&parent->right, right);
            }
        }

        free(del);

        parent = NULL;
        to_delete = tree_find_to_delete(*tree, &parent);
    }
}

static void tree_print_dot_recursive(tree_t *tree, FILE *fp, int *counter, int
*null_counter)
{
    if (tree == NULL)
    {
        return;
    }

    int current_id = (*counter)++;

    fwprintf(fp, L"    node%d [label=\"%ls\"];\n", current_id, tree->word);

    if (tree->left != NULL)
    {
        int left_id = *counter;
        tree_print_dot_recursive(tree->left, fp, counter, null_counter);
        fwprintf(fp, L"    node%d -> node%d;\n", current_id, left_id);
    }
    else
    {
        int null_id = (*null_counter)++;
        fwprintf(fp, L"    null%d [shape=point];\n", null_id);
        fwprintf(fp, L"    node%d -> null%d;\n", current_id, null_id);
    }

    if (tree->right != NULL)
    {
        int right_id = *counter;
        tree_print_dot_recursive(tree->right, fp, counter, null_counter);
        fwprintf(fp, L"    node%d -> node%d;\n", current_id, right_id);
    }
    else
    {
        int null_id = (*null_counter)++;
        fwprintf(fp, L"    null%d [shape=point];\n", null_id);
```

```
        fwprintf(fp, L"    node%d -> null%d;\n", current_id, null_id);
    }
}

static void tree_print_dot(tree_t *tree, const char *filename)
{
    FILE *fp = fopen(filename, "w");
    if (fp == NULL)
    {
        fprintf(stderr, "ошибка открытия файла %s\n", filename);
        return;
    }

    fwprintf(fp, L"digraph BST {\n");
    fwprintf(fp, L"    node [shape=circle, style=filled, fillcolor=lightblue];\n");

    int counter = 0;
    int null_counter = 0;

    tree_print_dot_recursive(tree, fp, &counter, &null_counter);

    fwprintf(fp, L"}\n");
    fclose(fp);

    wprintf(L"DOT файл сохранен в %s\n", filename);
}

typedef struct Trunk Trunk;

struct Trunk
{
    Trunk *prev;
    const wchar_t *str;
};

static void showTrunks(Trunk *p)
{
    if (p == NULL)
    {
        return;
    }

    showTrunks(p->prev);
    wprintf(L"%ls", p->str);
}

static void printTree(tree_t *root, Trunk *prev, bool isLeft)
{
    if (root == NULL)
    {
        return;
    }

    const wchar_t *prev_str = L"     ";

    Trunk *trunk = (Trunk *)malloc(sizeof(Trunk));
    if (trunk == NULL)
    {
        return;
    }

    trunk->prev = prev;
    trunk->str = prev_str;

    printTree(root->right, trunk, true);
```

```
    if (!prev)
    {
        trunk->str = L"-----";
    }
    else if (isLeft)
    {
        trunk->str = L".----";
        prev_str = L"    |";
    }
    else
    {
        trunk->str = L"`----";
        prev->str = prev_str;
    }

    showTrunks(trunk);
    wprintf(L" %ls\n", root->word);

    if (prev)
        prev->str = prev_str;

    trunk->str = L"    |";

    printTree(root->left, trunk, false);

    free(trunk);
}

static void tree_free(tree_t *tree)
{
    if (!tree)
    {
        return;
    }

    tree_free(tree->left);
    tree_free(tree->right);
    free(tree);
}

int main(void)
{
    setlocale(LC_ALL, "Russian");

    tree_t *tree = NULL;

    FILE *fp = fopen("words.txt", "r");
    if (!fp)
    {
        perror("words.txt");
        return 1;
    }

    wchar_t buffer[LINE_MAX];

    while (fgetws(buffer, LINE_MAX, fp) != NULL)
    {
        buffer[wcscspn(buffer, L"\n\r")] = L'\0';

        if (buffer[0] == L'\0')
            continue;

        tree_add(&tree, buffer);
    }
```

```
    fclose(fp);

    wprintf(L"До удаления\n");
    printTree(tree, NULL, false);
    tree_print_dot(tree, "tree.dot");

    tree_delete_words(&tree);

    wprintf(L"\nПосле удаления:\n");
    printTree(tree, NULL, false);
    tree_print_dot(tree, "tree_new.dot");

    tree_free(tree);

    return 0;
}
```

Листинг 1. Код программы

Тестирование:

```
$ ./app.exe
До удаления
     .---- Яблоко
    |    |     .---- Шашлык
    |    |    |    |              .---- Чашка
    |    |    |    |          .---- Стол
    |    |    |    |     .---- Солнце
    |    |    |     `---- Слово
    |    |    |         |                   .---- Река
    |    |    |         |                  |    |     .---- Песня
    |    |    |         |                  |     `---- Окно
    |    |    |         |              .---- Облако
    |    |    |         |          .---- Музыка
    |    |    |         |         |    |     .---- Море
    |    |    |         |         |     `---- Лес
    |    |    |         |     .---- Компьютер
    |    |    |          `---- Книга
    |     `---- Еда
    |          `---- Дорога
    |               `---- Дождь
----- Дерево
     `---- Город

После удаления:
     .---- Яблоко
    |    |                   .---- Стол
    |    |               .---- Облако
    |    |          .---- Музыка
    |    |         |     `---- Лес
    |    |     .---- Компьютер
    |     `---- Дорога
    |          `---- Дождь
----- Дерево
$
```

Рисунок 1. Результат удаления слов с чётным количеством гласных букв из дерева до и после.

Рисунок 2. Тестовый пример с другим набором слов.