МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации

**

**ЛАБОРАТОРНАЯ РАБОТА №1**

**по дисциплине: «Программирование»**

**на тему: «Абстрактные структуры данных»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБx-425», «АВТФ»,  *ФИО*  «09» октября 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент кафедры ЗИ*  *Исаев Глеб Андреевич*  «09» октября 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2025

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc210830876)

[Ход выполнения 5](#_Toc210830877)

[Реализация заданий 1 и 2 5](#_Toc210830878)

[Код на C++: 5](#_Toc210830879)

[Код на Rust: 62](#_Toc210830880)

[Результат выполнения программы на C++: 102](#_Toc210830881)

[Реузльтат выполнение программы на Rust: 105](#_Toc210830882)

[Реализация задания 3 108](#_Toc210830883)

[Реализация задания 4 109](#_Toc210830884)

[Реализация задания 5 111](#_Toc210830885)

# ВВЕДЕНИЕ

**Цели и задачи работы:** изучение алгоритмов формирования и работы с абстрактными структурами данных.

**Задание к работе**: самостоятельно решить задачи в соответствии с индивидуальным вариантом.

**Методика выполнения работы:**

1. Разработать алгоритм решения задачи по индивидуальному заданию.

2. Написать и отладить программу решения задачи.

3. Протестировать работу программы на различных исходных данных.

4. По запросу преподавателя быть готовым модифицировать алгоритм и добавить операцию работы с данными.

Задания:

1.Реализовать структуры данных с базовым набором операций: a. Массив. Операции: создание, добавление элемента (по индексу и в конец массива), получение элемента по индексу, удаление элемента по индексу, замена элемента по индексу, длина массива, чтение. b. Односвязный список. Операции: добавление и удаление элемента (4 способа (до после голова хвост)), чтение (разное, несколько способов), удаление элемента по значению, поиск элемента по значению. c. Двусвязный список. Операции: добавление и удаление элемента (4 способа (до после голова хвост)), чтение (разное, несколько способов), удаление элемента по значению, поиск элемента по значению. d. Стек. Операции: добавление и удаление элемента (push и pop), чтение. e. Очередь. Операции: добавление и удаление элемента (push и pop), чтение. f. Деревья – Full Binary Tree.

2.Реализовать интерфейс работы со структурами данных. Предусмотреть считывание из файла и запись в файл при внесении изменений в состав элементов. Часть операций для реализации представлено в таблице. Дополнить список операций согласно п.1 и сохраняя стиль написания команды (первая буква – принадлежность структуре данных). Исключение: Операция PRINT выводит любую структуру данных на экран.

3.Обосновать сложность выполнения каждой операции с позиции BigO нотации. Составить таблицу BigO по операциям и структурам в отчете. 4.Представить достоинства и недостатки каждой структуры данных, описать сферу применения (примеры).

5.В отчете в контрольном примере представить визуальные формы, подтверждающие соответствие результатам работы программы.

# Ход выполнения

## Реализация заданий 1 и 2

### Код на C++:

#pragma once

#include <iostream>

using namespace std;

// Шаблонная структура для массива

template<typename T>

struct Array {

T\* data = nullptr; // Динамический массив значений

bool\* is\_set = nullptr; // Флаги для отслеживания установленных элементов

int capacity = 0; // Текущая вместимость массива

int max\_index = 0; // Максимальный индекс, где есть установленный элемент

~Array() {

delete[] data;

delete[] is\_set;

}

};

// Создание массива с заданной вместимостью

template<typename T>

Array<T> create\_array\_ar(int new\_capacity) {

Array<T> new\_arr;

if (new\_capacity > 0) {

new\_arr.capacity = new\_capacity;

new\_arr.data = new T[new\_capacity](); // Инициализация по умолчанию (0 для int)

new\_arr.is\_set = new bool[new\_capacity](); // Инициализация false

}

return new\_arr;

}

// Вспомогательная функция для расширения массива

template<typename T>

void resize\_array\_ar(Array<T>& arr, int new\_capacity) {

// Устанавливаем новую вместимость (не меньше текущей)

if (new\_capacity < arr.capacity) {

new\_capacity = arr.capacity;

}

T\* new\_data = new T[new\_capacity](); // Новый массив с инициализацией

bool\* new\_is\_set = new bool[new\_capacity](); // Новый массив флагов

// Копируем существующие данные

for (int i = 0; i < arr.max\_index; i++) {

new\_data[i] = arr.data[i];

new\_is\_set[i] = arr.is\_set[i];

}

// Освобождаем старую память

delete[] arr.data;

delete[] arr.is\_set;

// Обновляем поля

arr.data = new\_data;

arr.is\_set = new\_is\_set;

arr.capacity = new\_capacity;

}

// Добавление элемента в конец массива

template<typename T>

void add\_element\_end\_ar(Array<T>& arr, T new\_element) {

if (arr.max\_index >= arr.capacity) {

// Расширяем массив, удваивая вместимость или устанавливая минимальную

int new\_capacity = (arr.capacity == 0) ? 4 : arr.capacity \* 2;

resize\_array\_ar(arr, new\_capacity);

}

arr.data[arr.max\_index] = new\_element;

arr.is\_set[arr.max\_index] = true;

arr.max\_index++;

}

// Добавление элемента по индексу (1-based)

template<typename T>

void add\_element\_index\_ar(Array<T>& arr, int index, T new\_element) {

if (index < 1) {

cerr << "Неправильное значение индекса" << endl;

return;

}

// Проверяем, нужен ли resize

if (index > arr.capacity) {

// Устанавливаем новую вместимость как max(index, capacity \* 2)

int new\_capacity = (arr.capacity == 0) ? index : max(index, arr.capacity \* 2);

resize\_array\_ar(arr, new\_capacity);

}

arr.data[index - 1] = new\_element;

arr.is\_set[index - 1] = true;

if (index > arr.max\_index) {

arr.max\_index = index; // Обновляем max\_index

}

}

// Удаление элемента по индексу (1-based)

template<typename T>

void delete\_element\_index\_ar(Array<T>& arr, int index) {

if (index < 1 || index > arr.max\_index) {

cerr << "Неправильное значение индекса" << endl;

return;

}

if (!arr.is\_set[index - 1]) {

cerr << "Элемент по индексу " << index << " не установлен" << endl;

return;

}

arr.is\_set[index - 1] = false;

arr.data[index - 1] = T(); // Сбрасываем значение

// Обновляем max\_index, если удалили последний элемент

if (index == arr.max\_index) {

while (arr.max\_index > 0 && !arr.is\_set[arr.max\_index - 1]) {

arr.max\_index--;

}

}

}

// Изменение элемента по индексу (1-based)

template<typename T>

void change\_element\_index\_ar(Array<T>& arr, int index, T new\_value) {

if (index < 1 || index > arr.max\_index) {

cerr << "Неправильное значение индекса" << endl;

return;

}

if (!arr.is\_set[index - 1]) {

cerr << "Элемент по индексу " << index << " не установлен" << endl;

return;

}

arr.data[index - 1] = new\_value;

arr.is\_set[index - 1] = true;

}

// Получение элемента по индексу (1-based)

template<typename T>

T get\_element\_index\_ar(const Array<T>& arr, int index) {

if (index < 1 || index > arr.max\_index) {

cerr << "Неправильное значение индекса" << endl;

return;

}

if (!arr.is\_set[index - 1]) {

cerr << "Элемент по индексу " << index << " не установлен" << endl;

return T();

}

return arr.data[index - 1];

}

// Вывод массива

template<typename T>

void print\_array\_ar(const Array<T>& arr) {

if (arr.max\_index == 0) {

cout << "Массив пуст" << endl;

return;

}

cout << "Элементы массива: ";

for (int i = 0; i < arr.max\_index; i++) {

if (arr.is\_set[i]) {

cout << arr.data[i] << " ";

} else {

cout << "0 ";

}

}

cout << endl;

}

#pragma once

#include "Queue.hpp"

#include "Stack.hpp"

#include "Array.hpp"

#include "DoublyList.hpp"

#include "ForwardList.hpp"

#include "SystemWork.hpp"

#include "FullBinaryTree.hpp"

using namespace std;

void cmd\_Mpush(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "MPUSH требует имя и значение\n"; return; }

string name = toks.data[1];

// build value from tokens[2..]

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_M, name);

Array<string>\* arr;

if (idx == -1) {

arr = new Array<string>(create\_array\_ar<string>(4));

add\_named\_container<Array<string>\*>(names\_M, data\_M, name, arr);

} else arr = data\_M.data[idx];

add\_element\_end\_ar<string>(\*arr, value);

cout << "OK\n";

save\_db(g\_file\_path);

}

void cmd\_Mget(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "MGET требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_M, name);

if (idx == -1) { cerr << "Массив не найден\n"; return; }

Array<string>\* arr = data\_M.data[idx];

if (arr->max\_index < id || !arr->is\_set[id-1]) { cerr << "Индекс вне диапазона\n"; return; }

cout << arr->data[id-1] << "\n";

}

void cmd\_Mdel(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "MDEL требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_M, name);

if (idx == -1) { cerr << "Массив не найден\n"; return; }

Array<string>\* arr = data\_M.data[idx];

if (arr->max\_index < id || !arr->is\_set[id-1]) { cerr << "Индекс вне диапазона\n"; return; }

delete\_element\_index\_ar<string>(\*arr, id);

cout << "OK\n";

save\_db(g\_file\_path);

}

// FPUSH (back)

void cmd\_Fpush(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "FPUSH требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_F, name);

ForwardList<string>\* fl;

if (idx == -1) {

fl = new ForwardList<string>(create\_forwardList<string>());

add\_named\_container<ForwardList<string>\*>(names\_F, data\_F, name, fl);

} else fl = data\_F.data[idx];

add\_element\_back\_fl<string>(value, \*fl);

cout << "OK\n";

save\_db(g\_file\_path);

}

void cmd\_Fget(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "FGET требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_F, name);

if (idx == -1) { cerr << "Список не найден\n"; return; }

ForwardList<string>\* fl = data\_F.data[idx];

Node\_Fl<string>\* cur = fl->head;

int pos = 1;

while (cur != nullptr && pos < id) { cur = cur->next; ++pos; }

if (cur == nullptr) { cerr << "Индекс вне диапазона\n"; return; }

cout << \*(cur->data) << "\n";

}

void cmd\_Fdel(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "FDEL требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_F, name);

if (idx == -1) { cerr << "Список не найден\n"; return; }

ForwardList<string>\* fl = data\_F.data[idx];

if (fl->head == nullptr) { cerr << "Список пуст\n"; return; }

if (id == 1) {

Node\_Fl<string>\* todel = fl->head;

fl->head = todel->next;

delete todel->data;

delete todel;

cout << "OK\n";

save\_db(g\_file\_path);

return;

}

Node\_Fl<string>\* cur = fl->head;

int pos = 1;

while (cur != nullptr && pos < id-1) { cur = cur->next; ++pos; }

if (cur == nullptr || cur->next == nullptr) { cerr << "Индекс вне диапазона\n"; return; }

Node\_Fl<string>\* todel = cur->next;

cur->next = todel->next;

delete todel->data;

delete todel;

cout << "OK\n";

save\_db(g\_file\_path);

}

// LPUSH (back)

void cmd\_Lpush(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "LPUSH требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_L, name);

DoublyList<string>\* dl;

if (idx == -1) {

dl = new DoublyList<string>(create\_doublyList<string>());

add\_named\_container<DoublyList<string>\*>(names\_L, data\_L, name, dl);

} else dl = data\_L.data[idx];

add\_element\_back\_dl<string>(value, \*dl);

cout << "OK\n";

save\_db(g\_file\_path);

}

void cmd\_Lget(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "LGET требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_L, name);

if (idx == -1) { cerr << "Список не найден\n"; return; }

DoublyList<string>\* dl = data\_L.data[idx];

Node\_Dl<string>\* cur = dl->head;

int pos = 1;

while (cur != nullptr && pos < id) { cur = cur->next; ++pos; }

if (cur == nullptr) { cerr << "Индекс вне диапазона\n"; return; }

cout << \*(cur->data) << "\n";

}

void cmd\_Ldel(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "LDEL требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_L, name);

if (idx == -1) { cerr << "Список не найден\n"; return; }

DoublyList<string>\* dl = data\_L.data[idx];

if (dl->head == nullptr) { cerr << "Список пуст\n"; return; }

Node\_Dl<string>\* cur = dl->head;

int pos = 1;

while (cur != nullptr && pos < id) { cur = cur->next; ++pos; }

if (cur == nullptr) { cerr << "Индекс вне диапазона\n"; return; }

// unlink

if (cur->prev != nullptr) cur->prev->next = cur->next; else dl->head = cur->next;

if (cur->next != nullptr) cur->next->prev = cur->prev; else dl->tail = cur->prev;

delete cur->data;

delete cur;

cout << "OK\n";

save\_db(g\_file\_path);

}

// QPUSH / QPOP / QGET

void cmd\_Qpush(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "QPUSH требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_Q, name);

Queue<string>\* q;

if (idx == -1) {

q = new Queue<string>(create\_queue<string>());

add\_named\_container<Queue<string>\*>(names\_Q, data\_Q, name, q);

} else q = data\_Q.data[idx];

push\_queue<string>(\*q, value);

cout << "OK\n";

save\_db(g\_file\_path);

}

void cmd\_Qpop(const Array<string>& toks) {

if (toks.max\_index < 2) { cerr << "QPOP требует имя\n"; return; }

string name = toks.data[1];

int idx = find\_name\_index(names\_Q, name);

if (idx == -1) { cout << "-> (queue empty)\n"; return; }

Queue<string>\* q = data\_Q.data[idx];

if (q->size == 0) { cout << "-> (queue empty)\n"; return; }

string v = pop\_queue<string>(\*q);

cout << "-> " << v << "\n";

save\_db(g\_file\_path);

}

void cmd\_Qget(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "QGET требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_Q, name);

if (idx == -1) { cerr << "Очередь не найдена\n"; return; }

Queue<string>\* q = data\_Q.data[idx];

if (q->size < id) { cerr << "Индекс вне диапазона\n"; return; }

int at = (q->front + (id - 1)) % q->capacity;

cout << q->data[at] << "\n";

}

// SPUSH / SPOP / SGET

void cmd\_Spush(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "SPUSH требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_S, name);

Stack<string>\* s;

if (idx == -1) {

s = new Stack<string>(create\_stack<string>());

add\_named\_container<Stack<string>\*>(names\_S, data\_S, name, s);

} else s = data\_S.data[idx];

push\_stack<string>(\*s, value);

cout << "OK\n";

save\_db(g\_file\_path);

}

void cmd\_Spop(const Array<string>& toks) {

if (toks.max\_index < 2) {

cerr << "SPOP требует имя\n";

return;

}

string name = toks.data[1];

int idx = find\_name\_index(names\_S, name);

if (idx == -1) {

cout << "-> (stack empty)\n";

return;

}

Stack<string>\* s = data\_S.data[idx];

if (s->top\_index == 0) {

cout << "-> (stack empty)\n";

return;

}

string v = pop\_stack(\*s);

cout << "-> " << v << "\n";

save\_db(g\_file\_path);

}

void cmd\_Sget(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "SGET требует имя и индекс\n"; return; }

string name = toks.data[1];

int id = parse\_index(toks.data[2]);

if (id < 1) { cerr << "Неправильный индекс\n"; return; }

int idx = find\_name\_index(names\_S, name);

if (idx == -1) { cerr << "Стек не найден\n"; return; }

Stack<string>\* s = data\_S.data[idx];

if (s->top\_index < id) { cerr << "Индекс вне диапазона\n"; return; }

// определяем: индекс 1 == первый элемент (нижний). Чтобы вернуть по такому порядку:

cout << s->data[id-1] << "\n";

}

// === ХРАНИЛИЩЕ ДЕРЕВЬЕВ ===

const int MAX\_TREES = 16;

string tree\_names[MAX\_TREES];

FullBinaryTree<int>\* trees[MAX\_TREES];

int tree\_count = 0;

// === ВСПОМОГАТЕЛЬНАЯ ФУНКЦИЯ ===

int find\_tree\_index(const string& name) {

for (int i = 0; i < tree\_count; ++i)

if (tree\_names[i] == name)

return i;

return -1;

}

// === TINSERT ===

void cmd\_Tinsert(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "TINSERT требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_T, name);

FullBinaryTree<string>\* tree;

if (idx == -1) {

tree = new FullBinaryTree<string>(create\_fbt<string>());

add\_named\_container<FullBinaryTree<string>\*>(names\_T, data\_T, name, tree);

} else tree = data\_T.data[idx];

add\_element\_fbt(\*tree, value);

cout << "OK\n";

save\_db(g\_file\_path);

}

// TGET tree\_name

void cmd\_Tget(const Array<string>& toks) {

if (toks.max\_index < 2) { cerr << "TGET требует имя\n"; return; }

string name = toks.data[1];

int idx = find\_name\_index(names\_T, name);

if (idx == -1) { cerr << "Дерево не найдено\n"; return; }

FullBinaryTree<string>\* tree = data\_T.data[idx];

cout << "TREE " << name << ": ";

print\_fbt(tree->root);

}

// TFULL tree\_name

void cmd\_Tfull(const Array<string>& toks) {

if (toks.max\_index < 2) { cerr << "TFULL требует имя\n"; return; }

string name = toks.data[1];

int idx = find\_name\_index(names\_T, name);

if (idx == -1) { cerr << "Дерево не найдено\n"; return; }

FullBinaryTree<string>\* tree = data\_T.data[idx];

cout << "Full? " << (is\_full\_fbt(\*tree) ? "Да" : "Нет") << "\n";

}

// ISMEMBER tree\_name value

void cmd\_Tismember(const Array<string>& toks) {

if (toks.max\_index < 3) { cerr << "ISMEMBER требует имя и значение\n"; return; }

string name = toks.data[1];

string value = toks.data[2];

for (int i = 3; i < toks.max\_index; ++i) value += " " + toks.data[i];

int idx = find\_name\_index(names\_T, name);

if (idx == -1) { cerr << "Дерево не найдено\n"; return; }

FullBinaryTree<string>\* tree = data\_T.data[idx];

cout << (find\_element\_fbt(tree->root, value) ? "1\n" : "0\n");

}

void cmd\_PRINT(const Array<string>& toks) {

if (toks.max\_index < 2) {

cerr << "PRINT требует имя\n";

return;

}

string name = toks.data[1];

// массив

int idx = find\_name\_index(names\_M, name);

if (idx != -1) {

print\_array\_ar(\*data\_M.data[idx]);

return;

}

// односвязный список

idx = find\_name\_index(names\_F, name);

if (idx != -1) {

ForwardList<string>\* fl = data\_F.data[idx];

Node\_Fl<string>\* cur = fl->head;

while (cur != nullptr) {

cout << \*(cur->data) << " ";

cur = cur->next;

}

cout << "\n";

return;

}

// двусвязный список

idx = find\_name\_index(names\_L, name);

if (idx != -1) {

DoublyList<string>\* dl = data\_L.data[idx];

Node\_Dl<string>\* cur = dl->head;

while (cur != nullptr) {

cout << \*(cur->data) << " ";

cur = cur->next;

}

cout << "\n";

return;

}

// очередь

idx = find\_name\_index(names\_Q, name);

if (idx != -1) {

Queue<string>\* q = data\_Q.data[idx];

for (int i = 0; i < q->size; i++) {

int at = (q->front + i) % q->capacity;

cout << q->data[at] << " ";

}

cout << "\n";

return;

}

// стек

idx = find\_name\_index(names\_S, name);

if (idx != -1) {

Stack<string>\* s = data\_S.data[idx];

for (int i = 0; i < s->top\_index; i++) {

cout << s->data[i] << " ";

}

cout << "\n";

return;

}

cerr << "Контейнер с именем " << name << " не найден\n";

}

#pragma once

#include <iostream>

using namespace std;

template<typename T>

struct Node\_Dl {

T\* data = nullptr;

Node\_Dl<T>\* next = nullptr;

Node\_Dl<T>\* prev = nullptr;

};

template<typename T>

struct DoublyList {

Node\_Dl<T>\* head = nullptr;

Node\_Dl<T>\* tail = nullptr;

};

template<typename T>

DoublyList<T> create\_doublyList() {

DoublyList<T> new\_dl;

new\_dl.head = nullptr;

new\_dl.tail = nullptr;

return new\_dl;

}

template<typename T>

void add\_element\_front\_dl(T new\_element, DoublyList<T>& dl) {

Node\_Dl<T>\* new\_node = new Node\_Dl<T>;

new\_node->data = new T(new\_element);

new\_node->next = dl.head;

new\_node->prev = nullptr;

if (dl.head != nullptr) {

dl.head->prev = new\_node;

}

dl.head = new\_node;

if (dl.tail == nullptr) {

dl.tail = new\_node;

}

}

template<typename T>

void add\_element\_back\_dl(T new\_element, DoublyList<T>& dl) {

Node\_Dl<T>\* new\_node = new Node\_Dl<T>;

new\_node->data = new T(new\_element);

new\_node->next = nullptr;

new\_node->prev = dl.tail;

if (dl.head == nullptr) {

dl.head = new\_node;

dl.tail = new\_node;

} else {

dl.tail->next = new\_node;

dl.tail = new\_node;

}

}

template<typename T>

void add\_element\_before\_dl(T new\_element, DoublyList<T>& dl, Node\_Dl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

if (dl.head == nullptr) {

cerr << "Ошибка: список пуст" << endl;

return;

}

Node\_Dl<T>\* current = dl.head;

bool found = false;

while (current != nullptr) {

if (current == node) {

found = true;

break;

}

current = current->next;

}

if (!found) {

cerr << "Ошибка: узел не найден в списке" << endl;

return;

}

Node\_Dl<T>\* new\_node = new Node\_Dl<T>;

new\_node->data = new T(new\_element);

new\_node->next = node;

new\_node->prev = node->prev;

if (node->prev != nullptr) {

node->prev->next = new\_node;

} else {

dl.head = new\_node;

}

node->prev = new\_node;

}

template<typename T>

void add\_element\_after\_dl(T new\_element, DoublyList<T>& dl, Node\_Dl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

if (dl.head == nullptr) {

cerr << "Ошибка: список пуст" << endl;

return;

}

Node\_Dl<T>\* current = dl.head;

bool found = false;

while (current != nullptr) {

if (current == node) {

found = true;

break;

}

current = current->next;

}

if (!found) {

cerr << "Ошибка: узел не найден в списке" << endl;

return;

}

Node\_Dl<T>\* new\_node = new Node\_Dl<T>;

new\_node->data = new T(new\_element);

new\_node->next = node->next;

new\_node->prev = node;

if (node->next != nullptr) {

node->next->prev = new\_node;

} else {

dl.tail = new\_node;

}

node->next = new\_node;

}

template<typename T>

Node\_Dl<T>\* find\_element\_dl(T element, DoublyList<T>& dl) {

Node\_Dl<T>\* current = dl.head;

while (current != nullptr) {

if (\*(current->data) == element) {

return current;

}

current = current->next;

}

cerr << "Ошибка: узел не найден" << endl;

return nullptr;

}

template<typename T>

void delete\_element\_dl(DoublyList<T>& dl, Node\_Dl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

if (dl.head == nullptr) {

cerr << "Ошибка: список пуст" << endl;

return;

}

Node\_Dl<T>\* current = dl.head;

bool found = false;

while (current != nullptr) {

if (current == node) {

found = true;

break;

}

current = current->next;

}

if (!found) {

cerr << "Ошибка: узел не найден в списке" << endl;

return;

}

if (node->prev != nullptr) {

node->prev->next = node->next;

} else {

dl.head = node->next;

}

if (node->next != nullptr) {

node->next->prev = node->prev;

} else {

dl.tail = node->prev;

}

delete node->data;

delete node;

}

template<typename T>

void print\_dlist\_while\_dl(DoublyList<T>& dl) {

Node\_Dl<T>\* current = dl.head;

if (current == nullptr) {

cout << "Список пуст" << endl;

return;

}

while (current != nullptr) {

cout << "Значение: " << \*(current->data) << "; адрес: " << current << "; адрес следующего узла: " << current->next << "; адрес предыдущего узла: " << current->prev << endl;

current = current->next;

}

}

template<typename T>

void print\_dlist\_for\_dl(DoublyList<T>& dl) {

Node\_Dl<T>\* current = dl.head;

if (current == nullptr) {

cout << "Список пуст" << endl;

return;

}

for (current; current != nullptr; current = current->next) {

cout << "Значение: " << \*(current->data) << "; адрес: " << current << "; адрес следующего узла: " << current->next << "; адрес предыдущего узла: " << current->prev << endl;

}

}

#pragma once

#include <iostream>

using namespace std;

template<typename T>

struct Node\_Fl {

T\* data = nullptr;

Node\_Fl\* next = nullptr;

};

template<typename T>

struct ForwardList {

Node\_Fl<T>\* head;

};

template<typename T>

ForwardList<T> create\_forwardList() {

ForwardList<T> new\_fl;

new\_fl.head = nullptr;

return new\_fl;

}

template<typename T>

void add\_element\_front\_fl(T new\_element, ForwardList<T>& fl) {

Node\_Fl<T>\* new\_node = new Node\_Fl<T>;

new\_node->data = new T(new\_element);

new\_node->next = fl.head;

fl.head = new\_node;

}

template<typename T>

void add\_element\_back\_fl(T new\_element, ForwardList<T>& fl) {

Node\_Fl<T>\* new\_node = new Node\_Fl<T>;

new\_node->data = new T(new\_element);

new\_node->next = nullptr;

if (fl.head == nullptr) {

fl.head = new\_node;

} else {

Node\_Fl<T>\* current = fl.head;

while (current->next != nullptr) {

current = current->next;

}

current->next = new\_node;

}

}

template<typename T>

void add\_element\_before\_fl(T new\_element, ForwardList<T>& fl, Node\_Fl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

if (fl.head == nullptr) {

cerr << "Ошибка: список пуст" << endl;

return;

}

if (node == fl.head) {

Node\_Fl<T>\* new\_node = new Node\_Fl<T>;

new\_node->data = new T(new\_element);

new\_node->next = fl.head;

fl.head = new\_node;

return;

} else {

Node\_Fl<T>\* current =fl.head;

while (current != nullptr && current->next != node) {

current = current->next;

}

if (current == nullptr) {

cerr << "Ошибка: узел не найден" << endl;

return;

}

Node\_Fl<T>\* new\_node = new Node\_Fl<T>;

new\_node->data = new T(new\_element);

new\_node->next = current->next;

current->next = new\_node;

}

}

template<typename T>

void add\_element\_after\_fl(T new\_element, ForwardList<T>& fl, Node\_Fl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

if (fl.head == nullptr) {

cerr << "Ошибка: список пуст" << endl;

return;

}

Node\_Fl<T>\* new\_node = new Node\_Fl<T>;

new\_node->data = new T(new\_element);

new\_node->next = node->next;

node->next = new\_node;

}

template<typename T>

Node\_Fl<T>\* find\_element\_fl(T element, ForwardList<T>& fl) {

Node\_Fl<T>\* current = fl.head;

while (current != nullptr) {

if (\*(current->data) == element) {

return current;

}

current = current->next;

}

cerr << "Ошибка: узел не найден" << endl;

return nullptr;

}

template<typename T>

void delete\_element\_fl(ForwardList<T>& fl, Node\_Fl<T>\* node) {

if (node == nullptr) {

cerr << "Ошибка: передан нулевой указатель на узел" << endl;

return;

}

Node\_Fl<T>\* current = fl.head;

while (current != nullptr && current->next != node) {

current = current->next;

}

if (current == nullptr) {

cerr << "Ошибка: узел не найден" << endl;

return;

}

current->next = node->next;

delete node->data;

delete node;

}

template<typename T>

void print\_flist\_while\_fl(ForwardList<T>& fl) {

Node\_Fl<T>\* current = fl.head;

if (current == nullptr) {

cout << "Список пуст" << endl;

return;

}

while (current != nullptr) {

cout << "Значение: " << \*(current->data) << "; адрес: " << current << "; адрес следующего узла: " << current->next << endl;

current = current->next;

}

}

template<typename T>

void print\_flist\_for\_fl(ForwardList<T>& fl) {

Node\_Fl<T>\* current = fl.head;

if (current == nullptr) {

cout << "Список пуст" << endl;

return;

}

for (Node\_Fl<T>\* current = fl.head; current != nullptr; current = current->next) {

cout << "Значение: " << \*(current->data) << "; адрес: " << current << "; адрес следующего узла: " << current->next << endl;

}

}

#pragma once

#include "Queue.hpp"

// Структура узла и дерева

template<typename T>

struct NodeFBT {

T data;

NodeFBT\* left;

NodeFBT\* right;

};

template<typename T>

struct FullBinaryTree {

NodeFBT<T>\* root;

};

template<typename T>

FullBinaryTree<T> create\_fbt() {

FullBinaryTree<T> tree;

tree.root = nullptr;

return tree;

}

template<typename T>

NodeFBT<T>\* create\_node\_fbt(const T& value) {

NodeFBT<T>\* node = new NodeFBT<T>;

node->data = value;

node->left = node->right = nullptr;

return node;

}

template<typename T>

void add\_element\_fbt(FullBinaryTree<T>& tree, const T& value) {

NodeFBT<T>\* newNode = create\_node\_fbt(value);

if (!tree.root) {

tree.root = newNode;

return;

}

Queue<NodeFBT<T>\*> q = create\_queue<NodeFBT<T>\*>();

push\_queue(q, tree.root);

while (q.size > 0) {

NodeFBT<T>\* cur = pop\_queue(q);

if (!cur->left) {

cur->left = newNode;

return;

} else {

push\_queue(q, cur->left);

}

if (!cur->right) {

cur->right = newNode;

return;

} else {

push\_queue(q, cur->right);

}

}

}

template<typename T>

bool is\_full\_fbt(const FullBinaryTree<T>& tree) {

if (!tree.root) return true;

Queue<NodeFBT<T>\*> q = create\_queue<NodeFBT<T>\*>();

push\_queue(q, tree.root);

bool mustBeLeaf = false; // флаг: после первого неполного узла все должны быть листьями

while (q.size > 0) {

NodeFBT<T>\* cur = pop\_queue(q);

if (mustBeLeaf) {

// Если мы должны встретить только листья, а текущий узел имеет детей

if (cur->left || cur->right) return false;

}

if (cur->left && cur->right) {

// Оба ребенка есть, добавляем их в очередь

push\_queue(q, cur->left);

push\_queue(q, cur->right);

} else if (cur->left && !cur->right) {

// Только левый ребенок — дерево не полное

return false;

} else if (!cur->left && cur->right) {

// Только правый ребенок — дерево не полное

return false;

} else {

// Узел без детей — все последующие должны быть листьями

mustBeLeaf = true;

}

}

return true;

}

template<typename T>

bool find\_element\_fbt(NodeFBT<T>\* root, const T& value) {

if (!root) return false;

Queue<NodeFBT<T>\*> q = create\_queue<NodeFBT<T>\*>();

push\_queue(q, root);

while (q.size > 0) {

NodeFBT<T>\* cur = pop\_queue(q);

if (cur->data == value) return true;

if (cur->left) push\_queue(q, cur->left);

if (cur->right) push\_queue(q, cur->right);

}

return false;

}

template<typename T>

void print\_fbt(NodeFBT<T>\* root) {

if (!root) return;

Queue<NodeFBT<T>\*> q = create\_queue<NodeFBT<T>\*>();

push\_queue(q, root);

while (q.size > 0) {

NodeFBT<T>\* cur = pop\_queue(q);

cout << cur->data << " ";

if (cur->left) push\_queue(q, cur->left);

if (cur->right) push\_queue(q, cur->right);

}

cout << "\n";

}

#include "SystemWork.hpp"

// Массивы и контейнеры

Array<string> names\_M = create\_array\_ar<string>(4);

Array<Array<string>\*> data\_M = create\_array\_ar<Array<string>\*>(4);

Array<string> names\_F = create\_array\_ar<string>(4);

Array<ForwardList<string>\*> data\_F = create\_array\_ar<ForwardList<string>\*>(4);

Array<string> names\_L = create\_array\_ar<string>(4);

Array<DoublyList<string>\*> data\_L = create\_array\_ar<DoublyList<string>\*>(4);

Array<string> names\_Q = create\_array\_ar<string>(4);

Array<Queue<string>\*> data\_Q = create\_array\_ar<Queue<string>\*>(4);

Array<string> names\_S = create\_array\_ar<string>(4);

Array<Stack<string>\*> data\_S = create\_array\_ar<Stack<string>\*>(4);

Array<string> names\_T = create\_array\_ar<string>(4);

Array<FullBinaryTree<string>\*> data\_T = create\_array\_ar<FullBinaryTree<string>\*>(4);

// Путь к файлу

string g\_file\_path;

#pragma once

#include <iostream>

using namespace std;

template<typename T>

struct Queue {

T\* data = nullptr;

int front = 0;

int rear = 0;

int capacity = 0;

int size = 0;

};

template<typename T>

Queue<T> create\_queue() {

Queue<T> new\_queue;

new\_queue.data = nullptr;

new\_queue.front = 0;

new\_queue.rear = 0;

new\_queue.capacity = 0;

new\_queue.size = 0;

return new\_queue;

}

template<typename T>

void resize\_queue(Queue<T>& queue) {

int new\_capacity = queue.capacity;

if (new\_capacity == 0) {

new\_capacity = (queue.capacity + 1) \* 2;

} else {

new\_capacity = queue.capacity \* 2;

}

T\* new\_data = new T[new\_capacity]();

int new\_index = 0;

for (int i = 0; i < queue.size; i++) {

int index = (queue.front + i) % queue.capacity;

new\_data[new\_index++] = queue.data[index];

}

delete[] queue.data;

queue.data = new\_data;

queue.capacity = new\_capacity;

queue.front = 0;

queue.rear = queue.size;

}

template<typename T>

void push\_queue(Queue<T>& queue, T new\_element) {

if (queue.size == queue.capacity) {

resize\_queue(queue);

}

queue.data[queue.rear] = new\_element;

queue.rear = (queue.rear + 1) % queue.capacity;

queue.size++;

}

template<typename T>

T pop\_queue(Queue<T>& queue) {

if (queue.size == 0) {

cerr << "Ошибка: очередь пуста" << endl;

return T();

}

T value = queue.data[queue.front];

queue.front = (queue.front + 1) % queue.capacity;

queue.size--;

return value;

}

template<typename T>

void print\_queue(Queue<T>& queue) {

for (int i = 0; i < queue.size; i++) {

int index = (queue.front + i) % queue.capacity;

cout << queue.data[index] << " ";

}

cout << endl;

}

#pragma once

#include <iostream>

using namespace std;

template<typename T>

struct Stack {

T\* data = nullptr;

int top\_index = 0;

int capacity = 0;

};

template<typename T>

Stack<T> create\_stack() {

Stack<T> new\_stack;

new\_stack.data = nullptr;

new\_stack.top\_index = 0;

new\_stack.capacity = 0;

return new\_stack;

}

template<typename T>

void resize\_stack(Stack<T>& stack) {

int new\_capacity = (stack.capacity == 0) ? 4 : stack.capacity \* 2;

T\* new\_data = new T[new\_capacity]();

for (int i = 0; i < stack.top\_index; i++) new\_data[i] = stack.data[i];

delete[] stack.data;

stack.data = new\_data;

stack.capacity = new\_capacity;

}

template<typename T>

void push\_stack(Stack<T>& stack, T new\_element) {

if (stack.top\_index >= stack.capacity) resize\_stack(stack);

stack.data[stack.top\_index] = new\_element;

stack.top\_index++;

}

template<typename T>

T pop\_stack(Stack<T>& stack) {

if (stack.top\_index == 0) {

cerr << "Ошибка: стек пуст" << endl;

return T();

}

stack.top\_index--;

return stack.data[stack.top\_index];

}

template<typename T>

void print\_stack(Stack<T>& stack) {

for (int i = 0; i < stack.top\_index; i++) cout << stack.data[i] << " ";

cout << endl;

}

#pragma once

#include "Stack.hpp"

#include "Array.hpp"

#include "DoublyList.hpp"

#include "ForwardList.hpp"

#include "FullBinaryTree.hpp"

#include "Globals.cpp"

#include "Queue.hpp"

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

// Все контейнеры

extern Array<string> names\_M;

extern Array<Array<string>\*> data\_M;

extern Array<string> names\_F;

extern Array<ForwardList<string>\*> data\_F;

extern Array<string> names\_L;

extern Array<DoublyList<string>\*> data\_L;

extern Array<string> names\_Q;

extern Array<Queue<string>\*> data\_Q;

extern Array<string> names\_S;

extern Array<Stack<string>\*> data\_S;

extern Array<string> names\_T;

extern Array<FullBinaryTree<string>\*> data\_T;

extern string g\_file\_path;

// Добавление контейнера по имени

template<typename PtrT>

void add\_named\_container(Array<string>& names\_arr, Array<PtrT>& data\_arr, const string& name, PtrT ptr) {

add\_element\_end\_ar<string>(names\_arr, name);

add\_element\_end\_ar<PtrT>(data\_arr, ptr);

}

// Токенизация строки

Array<string> tokenize\_to\_array(const string& line) {

Array<string> toks = create\_array\_ar<string>(4);

string token;

stringstream ss(line);

while (ss >> token) add\_element\_end\_ar<string>(toks, token);

return toks;

}

// --- Загрузка базы ---

void load\_db(const string& path) {

ifstream in(path);

if (!in.is\_open()) return;

string line;

while (getline(in, line)) {

if (line.empty()) continue;

Array<string> toks = tokenize\_to\_array(line);

if (toks.max\_index < 2) continue;

string type = toks.data[0];

string name = toks.data[1];

if (type == "M") {

Array<string>\* arr = new Array<string>(create\_array\_ar<string>(max(4, toks.max\_index-2)));

for (int i=2; i<toks.max\_index; ++i) add\_element\_end\_ar<string>(\*arr, toks.data[i]);

add\_named\_container(names\_M, data\_M, name, arr);

} else if (type == "F") {

ForwardList<string>\* fl = new ForwardList<string>(create\_forwardList<string>());

for (int i=2; i<toks.max\_index; ++i) add\_element\_back\_fl(toks.data[i], \*fl);

add\_named\_container(names\_F, data\_F, name, fl);

} else if (type == "L") {

DoublyList<string>\* dl = new DoublyList<string>(create\_doublyList<string>());

for (int i=2; i<toks.max\_index; ++i) add\_element\_back\_dl(toks.data[i], \*dl);

add\_named\_container(names\_L, data\_L, name, dl);

} else if (type == "Q") {

Queue<string>\* q = new Queue<string>(create\_queue<string>());

for (int i=2; i<toks.max\_index; ++i) push\_queue(\*q, toks.data[i]);

add\_named\_container(names\_Q, data\_Q, name, q);

} else if (type == "S") {

Stack<string>\* s = new Stack<string>(create\_stack<string>());

for (int i=2; i<toks.max\_index; ++i) push\_stack(\*s, toks.data[i]);

add\_named\_container(names\_S, data\_S, name, s);

} else if (type == "T") {

FullBinaryTree<string>\* tree = new FullBinaryTree<string>(create\_fbt<string>());

for (int i=2; i<toks.max\_index; ++i)

add\_element\_fbt(\*tree, toks.data[i]);

add\_named\_container(names\_T, data\_T, name, tree);

}

}

in.close();

}

// --- Сохранение базы ---

void save\_db(const string& path) {

string tmp = path + ".tmp";

ofstream out(tmp);

if (!out.is\_open()) { cerr << "Не удалось открыть файл\n"; return; }

// Массивы

for (int i=0; i<names\_M.max\_index; ++i) {

if (!names\_M.is\_set[i]) continue;

out << "M " << names\_M.data[i];

Array<string>\* arr = data\_M.data[i];

for (int j=0; j<arr->max\_index; ++j) if (arr->is\_set[j]) out << " " << arr->data[j];

out << "\n";

}

// Односвязные списки

for (int i=0; i<names\_F.max\_index; ++i) {

if (!names\_F.is\_set[i]) continue;

out << "F " << names\_F.data[i];

ForwardList<string>\* fl = data\_F.data[i];

Node\_Fl<string>\* cur = fl->head;

while(cur) { out << " " << \*(cur->data); cur = cur->next; }

out << "\n";

}

// Двусвязные списки

for (int i=0; i<names\_L.max\_index; ++i) {

if (!names\_L.is\_set[i]) continue;

out << "L " << names\_L.data[i];

DoublyList<string>\* dl = data\_L.data[i];

Node\_Dl<string>\* cur = dl->head;

while(cur) { out << " " << \*(cur->data); cur = cur->next; }

out << "\n";

}

// Очереди

for (int i=0; i<names\_Q.max\_index; ++i) {

if (!names\_Q.is\_set[i]) continue;

out << "Q " << names\_Q.data[i];

Queue<string>\* q = data\_Q.data[i];

for (int k=0; k<q->size; ++k) {

int at = (q->front + k) % q->capacity;

out << " " << q->data[at];

}

out << "\n";

}

// Стеки

for (int i=0; i<names\_S.max\_index; ++i) {

if (!names\_S.is\_set[i]) continue;

out << "S " << names\_S.data[i];

Stack<string>\* s = data\_S.data[i];

for (int k=0; k<s->top\_index; ++k) out << " " << s->data[k];

out << "\n";

}

// Деревья (FullBinaryTree)

for (int i = 0; i < names\_T.max\_index; ++i) {

if (!names\_T.is\_set[i]) continue;

out << "T " << names\_T.data[i];

FullBinaryTree<string>\* tree = data\_T.data[i];

if (!tree->root) { out << "\n"; continue; }

// BFS обход с пользовательской очередью

Queue<NodeFBT<string>\*> q = create\_queue<NodeFBT<string>\*>();

push\_queue(q, tree->root);

while (q.size > 0) {

NodeFBT<string>\* cur = pop\_queue(q);

out << " " << cur->data;

if (cur->left) push\_queue(q, cur->left);

if (cur->right) push\_queue(q, cur->right);

}

out << "\n";

}

out.close();

remove(path.c\_str());

rename(tmp.c\_str(), path.c\_str());

}

int parse\_index(const string& s) {

try {

long x = stol(s);

if (x < 1) return -1;

return (int)x;

} catch (...) {

return -1;

}

}

int find\_name\_index(Array<string>& names, const string& name) {

for (int i = 0; i < names.max\_index; ++i) {

if (names.is\_set[i] && names.data[i] == name) return i;

}

return -1;

}

#pragma once

#include "Stack.hpp"

#include "Array.hpp"

#include "DoublyList.hpp"

#include "ForwardList.hpp"

#include "FullBinaryTree.hpp"

#include "Globals.cpp"

#include "Queue.hpp"

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

// Все контейнеры

extern Array<string> names\_M;

extern Array<Array<string>\*> data\_M;

extern Array<string> names\_F;

extern Array<ForwardList<string>\*> data\_F;

extern Array<string> names\_L;

extern Array<DoublyList<string>\*> data\_L;

extern Array<string> names\_Q;

extern Array<Queue<string>\*> data\_Q;

extern Array<string> names\_S;

extern Array<Stack<string>\*> data\_S;

extern Array<string> names\_T;

extern Array<FullBinaryTree<string>\*> data\_T;

extern string g\_file\_path;

// Добавление контейнера по имени

template<typename PtrT>

void add\_named\_container(Array<string>& names\_arr, Array<PtrT>& data\_arr, const string& name, PtrT ptr) {

add\_element\_end\_ar<string>(names\_arr, name);

add\_element\_end\_ar<PtrT>(data\_arr, ptr);

}

// Токенизация строки

Array<string> tokenize\_to\_array(const string& line) {

Array<string> toks = create\_array\_ar<string>(4);

string token;

stringstream ss(line);

while (ss >> token) add\_element\_end\_ar<string>(toks, token);

return toks;

}

// --- Загрузка базы ---

void load\_db(const string& path) {

ifstream in(path);

if (!in.is\_open()) return;

string line;

while (getline(in, line)) {

if (line.empty()) continue;

Array<string> toks = tokenize\_to\_array(line);

if (toks.max\_index < 2) continue;

string type = toks.data[0];

string name = toks.data[1];

if (type == "M") {

Array<string>\* arr = new Array<string>(create\_array\_ar<string>(max(4, toks.max\_index-2)));

for (int i=2; i<toks.max\_index; ++i) add\_element\_end\_ar<string>(\*arr, toks.data[i]);

add\_named\_container(names\_M, data\_M, name, arr);

} else if (type == "F") {

ForwardList<string>\* fl = new ForwardList<string>(create\_forwardList<string>());

for (int i=2; i<toks.max\_index; ++i) add\_element\_back\_fl(toks.data[i], \*fl);

add\_named\_container(names\_F, data\_F, name, fl);

} else if (type == "L") {

DoublyList<string>\* dl = new DoublyList<string>(create\_doublyList<string>());

for (int i=2; i<toks.max\_index; ++i) add\_element\_back\_dl(toks.data[i], \*dl);

add\_named\_container(names\_L, data\_L, name, dl);

} else if (type == "Q") {

Queue<string>\* q = new Queue<string>(create\_queue<string>());

for (int i=2; i<toks.max\_index; ++i) push\_queue(\*q, toks.data[i]);

add\_named\_container(names\_Q, data\_Q, name, q);

} else if (type == "S") {

Stack<string>\* s = new Stack<string>(create\_stack<string>());

for (int i=2; i<toks.max\_index; ++i) push\_stack(\*s, toks.data[i]);

add\_named\_container(names\_S, data\_S, name, s);

} else if (type == "T") {

FullBinaryTree<string>\* tree = new FullBinaryTree<string>(create\_fbt<string>());

for (int i=2; i<toks.max\_index; ++i)

add\_element\_fbt(\*tree, toks.data[i]);

add\_named\_container(names\_T, data\_T, name, tree);

}

}

in.close();

}

// --- Сохранение базы ---

void save\_db(const string& path) {

string tmp = path + ".tmp";

ofstream out(tmp);

if (!out.is\_open()) { cerr << "Не удалось открыть файл\n"; return; }

// Массивы

for (int i=0; i<names\_M.max\_index; ++i) {

if (!names\_M.is\_set[i]) continue;

out << "M " << names\_M.data[i];

Array<string>\* arr = data\_M.data[i];

for (int j=0; j<arr->max\_index; ++j) if (arr->is\_set[j]) out << " " << arr->data[j];

out << "\n";

}

// Односвязные списки

for (int i=0; i<names\_F.max\_index; ++i) {

if (!names\_F.is\_set[i]) continue;

out << "F " << names\_F.data[i];

ForwardList<string>\* fl = data\_F.data[i];

Node\_Fl<string>\* cur = fl->head;

while(cur) { out << " " << \*(cur->data); cur = cur->next; }

out << "\n";

}

// Двусвязные списки

for (int i=0; i<names\_L.max\_index; ++i) {

if (!names\_L.is\_set[i]) continue;

out << "L " << names\_L.data[i];

DoublyList<string>\* dl = data\_L.data[i];

Node\_Dl<string>\* cur = dl->head;

while(cur) { out << " " << \*(cur->data); cur = cur->next; }

out << "\n";

}

// Очереди

for (int i=0; i<names\_Q.max\_index; ++i) {

if (!names\_Q.is\_set[i]) continue;

out << "Q " << names\_Q.data[i];

Queue<string>\* q = data\_Q.data[i];

for (int k=0; k<q->size; ++k) {

int at = (q->front + k) % q->capacity;

out << " " << q->data[at];

}

out << "\n";

}

// Стеки

for (int i=0; i<names\_S.max\_index; ++i) {

if (!names\_S.is\_set[i]) continue;

out << "S " << names\_S.data[i];

Stack<string>\* s = data\_S.data[i];

for (int k=0; k<s->top\_index; ++k) out << " " << s->data[k];

out << "\n";

}

// Деревья (FullBinaryTree)

for (int i = 0; i < names\_T.max\_index; ++i) {

if (!names\_T.is\_set[i]) continue;

out << "T " << names\_T.data[i];

FullBinaryTree<string>\* tree = data\_T.data[i];

if (!tree->root) { out << "\n"; continue; }

// BFS обход с пользовательской очередью

Queue<NodeFBT<string>\*> q = create\_queue<NodeFBT<string>\*>();

push\_queue(q, tree->root);

while (q.size > 0) {

NodeFBT<string>\* cur = pop\_queue(q);

out << " " << cur->data;

if (cur->left) push\_queue(q, cur->left);

if (cur->right) push\_queue(q, cur->right);

}

out << "\n";

}

out.close();

remove(path.c\_str());

rename(tmp.c\_str(), path.c\_str());

}

int parse\_index(const string& s) {

try {

long x = stol(s);

if (x < 1) return -1;

return (int)x;

} catch (...) {

return -1;

}

}

int find\_name\_index(Array<string>& names, const string& name) {

for (int i = 0; i < names.max\_index; ++i) {

if (names.is\_set[i] && names.data[i] == name) return i;

}

return -1;

}

### Код на Rust:

#![allow(unsafe\_code)]

use std::collections::HashMap;

use std::env;

use std::fs::File;

use std::io::{self, BufRead, Write};

use std::ptr;

// MyArray - manual dynamic array without Vec

struct MyArray {

data: \*mut String,

len: usize,

capacity: usize,

}

impl MyArray {

fn new() -> Self {

Self {

data: ptr::null\_mut(),

len: 0,

capacity: 0,

}

}

fn push(&mut self, val: String) {

if self.len == self.capacity {

let new\_capacity = if self.capacity == 0 { 4 } else { self.capacity \* 2 };

let layout = std::alloc::Layout::array::<String>(new\_capacity).unwrap();

let new\_data = unsafe { std::alloc::alloc(layout) as \*mut String };

if !self.data.is\_null() {

unsafe { ptr::copy\_nonoverlapping(self.data, new\_data, self.len) };

let old\_layout = std::alloc::Layout::array::<String>(self.capacity).unwrap();

unsafe { std::alloc::dealloc(self.data as \*mut u8, old\_layout) };

}

self.data = new\_data;

self.capacity = new\_capacity;

}

unsafe { ptr::write(self.data.add(self.len), val) };

self.len += 1;

}

fn insert(&mut self, index: usize, val: String) {

if index > self.len {

panic!("Index out of bounds");

}

self.push(String::new());

unsafe {

ptr::copy(self.data.add(index), self.data.add(index + 1), self.len - index - 1);

ptr::write(self.data.add(index), val);

}

}

fn get(&self, index: usize) -> &String {

if index >= self.len {

panic!("Index out of bounds");

}

unsafe { &\*self.data.add(index) }

}

fn remove(&mut self, index: usize) {

if index >= self.len {

panic!("Index out of bounds");

}

unsafe { ptr::drop\_in\_place(self.data.add(index)) };

unsafe { ptr::copy(self.data.add(index + 1), self.data.add(index), self.len - index - 1) };

self.len -= 1;

}

fn set(&mut self, index: usize, val: String) {

if index >= self.len {

panic!("Index out of bounds");

}

unsafe { ptr::write(self.data.add(index), val) };

}

fn len(&self) -> usize {

self.len

}

fn to\_serialized(&self) -> String {

(0..self.len).map(|i| self.get(i).clone()).collect::<Vec<\_>>().join("|")

}

fn from\_serialized(serialized: &str) -> Self {

let mut arr = Self::new();

for val in serialized.split('|') {

if !val.is\_empty() {

arr.push(val.to\_string());

}

}

arr

}

fn read(&self) -> String {

(0..self.len).map(|i| self.get(i).as\_str()).collect::<Vec<\_>>().join(" ")

}

}

impl Drop for MyArray {

fn drop(&mut self) {

if !self.data.is\_null() {

for i in 0..self.len {

unsafe { ptr::drop\_in\_place(self.data.add(i)) };

}

let layout = std::alloc::Layout::array::<String>(self.capacity).unwrap();

unsafe { std::alloc::dealloc(self.data as \*mut u8, layout) };

}

}

}

// SinglyNode

struct SinglyNode {

val: String,

next: Option<Box<SinglyNode>>,

}

// SinglyLinkedList

struct SinglyLinkedList {

head: Option<Box<SinglyNode>>,

}

impl SinglyLinkedList {

fn new() -> Self {

Self { head: None }

}

fn push\_head(&mut self, val: String) {

let new = Box::new(SinglyNode { val, next: self.head.take() });

self.head = Some(new);

}

fn push\_tail(&mut self, val: String) {

let mut current = &mut self.head;

while let Some(node) = current {

current = &mut node.next;

}

\*current = Some(Box::new(SinglyNode { val, next: None }));

}

fn push\_after(&mut self, after\_val: &str, val: String) {

let mut current = &mut self.head;

while let Some(node) = current {

if node.val == after\_val {

let new = Box::new(SinglyNode { val, next: node.next.take() });

node.next = Some(new);

return;

}

current = &mut node.next;

}

}

fn push\_before(&mut self, before\_val: &str, val: String) {

if let Some(ref mut head\_node) = self.head {

if head\_node.val == before\_val {

self.push\_head(val);

return;

}

}

let mut current = &mut self.head;

while let Some(node) = current {

if let Some(ref mut next\_node) = node.next {

if next\_node.val == before\_val {

let new\_node = Box::new(SinglyNode {

val,

next: node.next.take(),

});

node.next = Some(new\_node);

return;

}

}

current = &mut node.next;

}

}

fn delete(&mut self, val: &str) {

if let Some(ref mut head\_node) = self.head {

if head\_node.val == val {

self.head = head\_node.next.take();

return;

}

}

let mut current = &mut self.head;

while let Some(node) = current {

if let Some(ref mut next\_node) = node.next {

if next\_node.val == val {

node.next = next\_node.next.take();

return;

}

}

current = &mut node.next;

}

}

fn search(&self, val: &str) -> bool {

let mut current = &self.head;

while let Some(node) = current {

if node.val == val {

return true;

}

current = &node.next;

}

false

}

fn to\_serialized(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.clone());

current = &node.next;

}

vals.join("|")

}

fn from\_serialized(serialized: &str) -> Self {

let mut list = Self::new();

for val in serialized.split('|').rev() {

if !val.is\_empty() {

list.push\_head(val.to\_string());

}

}

list

}

fn read(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.as\_str());

current = &node.next;

}

vals.join(" ")

}

}

// DoublyNode

struct DoublyNode {

val: String,

prev: \*mut DoublyNode,

next: Option<Box<DoublyNode>>,

}

// DoublyLinkedList

struct DoublyLinkedList {

head: Option<Box<DoublyNode>>,

tail: \*mut DoublyNode,

}

impl DoublyLinkedList {

fn new() -> Self {

Self {

head: None,

tail: ptr::null\_mut(),

}

}

fn push\_head(&mut self, val: String) {

let mut new\_node = Box::new(DoublyNode { val, prev: ptr::null\_mut(), next: self.head.take() });

let new\_ptr: \*mut DoublyNode = &mut \*new\_node;

if let Some(ref mut old\_head) = new\_node.next {

old\_head.prev = new\_ptr;

} else {

self.tail = new\_ptr;

}

self.head = Some(new\_node);

}

fn push\_tail(&mut self, val: String) {

if self.head.is\_none() {

self.push\_head(val);

return;

}

let mut new\_node = Box::new(DoublyNode {

val: val.clone(),

prev: self.tail,

next: None

});

let new\_ptr: \*mut DoublyNode = &mut \*new\_node;

unsafe {

(\*self.tail).next = Some(new\_node);

}

self.tail = new\_ptr;

}

fn push\_after(&mut self, after\_val: &str, val: String) {

let mut current = &mut self.head;

while let Some(node) = current {

if node.val == after\_val {

let old\_next = node.next.take();

let mut new\_node = Box::new(DoublyNode {

val,

prev: &mut \*\*node as \*mut DoublyNode,

next: old\_next,

});

let new\_ptr: \*mut DoublyNode = &mut \*new\_node;

if let Some(ref mut next\_node) = new\_node.next {

next\_node.prev = new\_ptr;

} else {

self.tail = new\_ptr;

}

node.next = Some(new\_node);

return;

}

current = &mut node.next;

}

}

fn push\_before(&mut self, before\_val: &str, val: String) {

// Handle case where we insert before head

if let Some(ref mut head\_node) = self.head {

if head\_node.val == before\_val {

self.push\_head(val);

return;

}

}

let mut current = &mut self.head;

while let Some(node) = current {

if let Some(ref mut next\_node) = node.next {

if next\_node.val == before\_val {

let mut new\_node = Box::new(DoublyNode {

val,

prev: &mut \*\*node as \*mut DoublyNode,

next: node.next.take(),

});

let new\_ptr: \*mut DoublyNode = &mut \*new\_node;

if let Some(ref mut new\_next) = new\_node.next {

new\_next.prev = new\_ptr;

}

node.next = Some(new\_node);

return;

}

}

current = &mut node.next;

}

}

fn delete(&mut self, val: &str) {

// Handle head deletion

if let Some(ref mut head\_node) = self.head {

if head\_node.val == val {

let mut next = head\_node.next.take();

if let Some(ref mut next\_node) = next {

next\_node.prev = ptr::null\_mut();

} else {

self.tail = ptr::null\_mut();

}

self.head = next;

return;

}

}

let mut current = &mut self.head;

while let Some(node) = current {

if let Some(ref mut next\_node) = node.next {

if next\_node.val == val {

let mut next\_next = next\_node.next.take();

if let Some(ref mut next\_next\_node) = next\_next {

next\_next\_node.prev = &mut \*\*node as \*mut DoublyNode;

} else {

self.tail = &mut \*\*node as \*mut DoublyNode;

}

node.next = next\_next;

return;

}

}

current = &mut node.next;

}

}

fn search(&self, val: &str) -> bool {

let mut current = &self.head;

while let Some(node) = current {

if node.val == val {

return true;

}

current = &node.next;

}

false

}

fn to\_serialized(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.clone());

current = &node.next;

}

vals.join("|")

}

fn from\_serialized(serialized: &str) -> Self {

let mut list = Self::new();

for val in serialized.split('|') {

if !val.is\_empty() {

list.push\_tail(val.to\_string());

}

}

list

}

fn read(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.as\_str());

current = &node.next;

}

vals.join(" ")

}

}

impl Drop for DoublyLinkedList {

fn drop(&mut self) {

let mut current = self.head.take();

while let Some(mut node) = current {

current = node.next.take();

}

}

}

// MyStack

struct MyStack {

head: Option<Box<SinglyNode>>,

}

impl MyStack {

fn new() -> Self {

Self { head: None }

}

fn push(&mut self, val: String) {

let new = Box::new(SinglyNode { val, next: self.head.take() });

self.head = Some(new);

}

fn pop(&mut self) -> Option<String> {

self.head.take().map(|node| {

self.head = node.next;

node.val

})

}

fn get(&self) -> Option<&String> {

self.head.as\_ref().map(|node| &node.val)

}

fn to\_serialized(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.clone());

current = &node.next;

}

vals.join("|")

}

fn from\_serialized(serialized: &str) -> Self {

let mut stack = Self::new();

for val in serialized.split('|').rev() {

if !val.is\_empty() {

stack.push(val.to\_string());

}

}

stack

}

fn read(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.as\_str());

current = &node.next;

}

vals.join(" ")

}

}

// MyQueue

struct MyQueue {

head: Option<Box<SinglyNode>>,

tail: \*mut SinglyNode,

}

impl MyQueue {

fn new() -> Self {

Self { head: None, tail: ptr::null\_mut() }

}

fn push(&mut self, val: String) {

let mut new\_node = Box::new(SinglyNode { val, next: None });

let new\_ptr: \*mut SinglyNode = &mut \*new\_node;

if !self.tail.is\_null() {

unsafe {

(\*self.tail).next = Some(new\_node);

}

} else {

self.head = Some(new\_node);

}

self.tail = new\_ptr;

}

fn pop(&mut self) -> Option<String> {

self.head.take().map(|node| {

self.head = node.next;

if self.head.is\_none() {

self.tail = ptr::null\_mut();

}

node.val

})

}

fn get(&self) -> Option<&String> {

self.head.as\_ref().map(|node| &node.val)

}

fn to\_serialized(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.clone());

current = &node.next;

}

vals.join("|")

}

fn from\_serialized(serialized: &str) -> Self {

let mut queue = Self::new();

for val in serialized.split('|') {

if !val.is\_empty() {

queue.push(val.to\_string());

}

}

queue

}

fn read(&self) -> String {

let mut vals = Vec::new();

let mut current = &self.head;

while let Some(node) = current {

vals.push(node.val.as\_str());

current = &node.next;

}

vals.join(" ")

}

}

impl Drop for MyQueue {

fn drop(&mut self) {

let mut current = self.head.take();

while let Some(node) = current {

current = node.next;

}

}

}

// TreeNode

struct TreeNode {

val: String,

left: Option<Box<TreeNode>>,

right: Option<Box<TreeNode>>,

}

// FullBinaryTree

struct FullBinaryTree {

root: Option<Box<TreeNode>>,

}

impl FullBinaryTree {

fn new() -> Self {

Self { root: None }

}

fn insert(&mut self, val: String) {

if self.root.is\_none() {

self.root = Some(Box::new(TreeNode { val, left: None, right: None }));

return;

}

// Level order to find first node with <2 children

let mut queue = std::collections::VecDeque::new();

queue.push\_back(self.root.as\_mut().unwrap());

while let Some(node) = queue.pop\_front() {

if node.left.is\_none() {

node.left = Some(Box::new(TreeNode { val, left: None, right: None }));

return;

} else {

queue.push\_back(node.left.as\_mut().unwrap());

}

if node.right.is\_none() {

node.right = Some(Box::new(TreeNode { val, left: None, right: None }));

return;

} else {

queue.push\_back(node.right.as\_mut().unwrap());

}

}

}

fn search(&self, val: &str) -> bool {

if self.root.is\_none() {

return false;

}

let mut queue = std::collections::VecDeque::new();

queue.push\_back(self.root.as\_ref().unwrap());

while let Some(node) = queue.pop\_front() {

if node.val == val {

return true;

}

if let Some(ref left) = node.left {

queue.push\_back(left);

}

if let Some(ref right) = node.right {

queue.push\_back(right);

}

}

false

}

fn is\_full(&self) -> bool {

if self.root.is\_none() {

return true;

}

let mut queue = std::collections::VecDeque::new();

queue.push\_back(self.root.as\_ref().unwrap());

while let Some(node) = queue.pop\_front() {

let has\_left = node.left.is\_some();

let has\_right = node.right.is\_some();

if has\_left != has\_right {

return false;

}

if has\_left {

queue.push\_back(node.left.as\_ref().unwrap());

}

if has\_right {

queue.push\_back(node.right.as\_ref().unwrap());

}

}

true

}

fn read(&self) -> String {

if self.root.is\_none() {

return "empty".to\_string();

}

let mut res = String::new();

let mut queue = std::collections::VecDeque::new();

queue.push\_back(self.root.as\_ref().unwrap());

while let Some(node) = queue.pop\_front() {

res.push\_str(&node.val);

res.push(' ');

if let Some(ref left) = node.left {

queue.push\_back(left);

}

if let Some(ref right) = node.right {

queue.push\_back(right);

}

}

res.trim().to\_string()

}

fn serialize(&self) -> String {

Self::serialize\_helper(&self.root)

}

fn serialize\_helper(node: &Option<Box<TreeNode>>) -> String {

if node.is\_none() {

return "# ".to\_string();

}

let n = node.as\_ref().unwrap();

format!("{} {} {}", n.val, Self::serialize\_helper(&n.left), Self::serialize\_helper(&n.right))

}

fn deserialize(serialized: &str) -> Self {

let mut iter = serialized.split\_whitespace();

let root = Self::deserialize\_helper(&mut iter);

Self { root }

}

fn deserialize\_helper<'a>(iter: &mut std::str::SplitWhitespace<'a>) -> Option<Box<TreeNode>> {

if let Some(token) = iter.next() {

if token == "#" {

return None;

}

let val = token.to\_string();

let left = Self::deserialize\_helper(iter);

let right = Self::deserialize\_helper(iter);

Some(Box::new(TreeNode { val, left, right }))

} else {

None

}

}

}

// DataStructure

enum DataStructure {

M(MyArray),

F(SinglyLinkedList),

L(DoublyLinkedList),

S(MyStack),

Q(MyQueue),

T(FullBinaryTree),

}

// DB

struct DB {

structures: HashMap<String, DataStructure>,

}

impl DB {

fn new() -> Self {

Self {

structures: HashMap::new(),

}

}

fn get\_mut\_or\_create\_m(&mut self, name: &str) -> &mut MyArray {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::M(MyArray::new()));

if let DataStructure::M(ref mut a) = \*entry {

a

} else {

panic!("Wrong type for {}", name);

}

}

fn get\_mut\_or\_create\_f(&mut self, name: &str) -> &mut SinglyLinkedList {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::F(SinglyLinkedList::new()));

if let DataStructure::F(ref mut f) = \*entry {

f

} else {

panic!("Wrong type for {}", name);

}

}

fn get\_mut\_or\_create\_l(&mut self, name: &str) -> &mut DoublyLinkedList {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::L(DoublyLinkedList::new()));

if let DataStructure::L(ref mut l) = \*entry {

l

} else {

panic!("Wrong type for {}", name);

}

}

fn get\_mut\_or\_create\_s(&mut self, name: &str) -> &mut MyStack {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::S(MyStack::new()));

if let DataStructure::S(ref mut s) = \*entry {

s

} else {

panic!("Wrong type for {}", name);

}

}

fn get\_mut\_or\_create\_q(&mut self, name: &str) -> &mut MyQueue {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::Q(MyQueue::new()));

if let DataStructure::Q(ref mut q) = \*entry {

q

} else {

panic!("Wrong type for {}", name);

}

}

fn get\_mut\_or\_create\_t(&mut self, name: &str) -> &mut FullBinaryTree {

let entry = self.structures.entry(name.to\_string()).or\_insert\_with(|| DataStructure::T(FullBinaryTree::new()));

if let DataStructure::T(ref mut t) = \*entry {

t

} else {

panic!("Wrong type for {}", name);

}

}

fn get(&self, name: &str) -> Option<&DataStructure> {

self.structures.get(name)

}

}

fn load\_db(file\_path: &str) -> DB {

let mut db = DB::new();

if let Ok(file) = File::open(file\_path) {

let reader = io::BufReader::new(file);

for line in reader.lines() {

if let Ok(line) = line {

let parts: Vec<&str> = line.splitn(3, ' ').collect();

if parts.len() < 3 {

continue;

}

let typ = parts[0];

let name = parts[1].to\_string();

let serialized = parts[2];

match typ {

"M" => db.structures.insert(name, DataStructure::M(MyArray::from\_serialized(serialized))),

"F" => db.structures.insert(name, DataStructure::F(SinglyLinkedList::from\_serialized(serialized))),

"L" => db.structures.insert(name, DataStructure::L(DoublyLinkedList::from\_serialized(serialized))),

"S" => db.structures.insert(name, DataStructure::S(MyStack::from\_serialized(serialized))),

"Q" => db.structures.insert(name, DataStructure::Q(MyQueue::from\_serialized(serialized))),

"T" => db.structures.insert(name, DataStructure::T(FullBinaryTree::deserialize(serialized))),

\_ => None,

};

}

}

}

db

}

fn save\_db(db: &DB, file\_path: &str) {

let mut file = File::create(file\_path).unwrap();

for (name, structure) in &db.structures {

let (typ, serialized) = match structure {

DataStructure::M(a) => ("M", a.to\_serialized()),

DataStructure::F(f) => ("F", f.to\_serialized()),

DataStructure::L(l) => ("L", l.to\_serialized()),

DataStructure::S(s) => ("S", s.to\_serialized()),

DataStructure::Q(q) => ("Q", q.to\_serialized()),

DataStructure::T(t) => ("T", t.serialize()),

};

writeln!(file, "{} {} {}", typ, name, serialized).unwrap();

}

}

fn main() {

let args: Vec<String> = env::args().collect();

let mut file\_path = String::new();

let mut query = String::new();

let mut i = 1;

while i < args.len() {

if args[i] == "--file" {

i += 1;

file\_path = args[i].clone();

} else if args[i] == "--query" {

i += 1;

query = args[i].clone();

}

i += 1;

}

if file\_path.is\_empty() || query.is\_empty() {

println!("Usage: ./dbms --file file.data --query 'COMMAND'");

return;

}

let mut db = load\_db(&file\_path);

// Improved parsing that handles spaces in values

let parts: Vec<&str> = query.splitn(3, ' ').collect();

if parts.is\_empty() {

return;

}

let command = parts[0];

match command {

"MPUSH" | "FPUSHHEAD" | "FPUSHTAIL" | "LPUSHHEAD" | "LPUSHTAIL" | "SPUSH" | "QPUSH" | "TINSERT" if parts.len() >= 2 => {

// For commands with structure name and value (possibly with spaces)

if parts.len() == 2 {

println!("Missing value for {}", command);

return;

}

let name = parts[1];

let value = parts[2];

match command {

"MPUSH" => {

db.get\_mut\_or\_create\_m(name).push(value.to\_string());

println!("{}", value);

}

"FPUSHHEAD" => {

db.get\_mut\_or\_create\_f(name).push\_head(value.to\_string());

println!("{}", value);

}

"FPUSHTAIL" => {

db.get\_mut\_or\_create\_f(name).push\_tail(value.to\_string());

println!("{}", value);

}

"LPUSHHEAD" => {

db.get\_mut\_or\_create\_l(name).push\_head(value.to\_string());

println!("{}", value);

}

"LPUSHTAIL" => {

db.get\_mut\_or\_create\_l(name).push\_tail(value.to\_string());

println!("{}", value);

}

"SPUSH" => {

db.get\_mut\_or\_create\_s(name).push(value.to\_string());

println!("{}", value);

}

"QPUSH" => {

db.get\_mut\_or\_create\_q(name).push(value.to\_string());

println!("{}", value);

}

"TINSERT" => {

db.get\_mut\_or\_create\_t(name).insert(value.to\_string());

println!("{}", value);

}

\_ => unreachable!(),

}

save\_db(&db, &file\_path);

}

"MINSERT" | "MSET" if parts.len() >= 4 => {

let name = parts[1];

let index\_result = parts[2].parse::<usize>();

let value = parts[3];

let index = match index\_result {

Ok(idx) => idx,

Err(\_) => {

println!("Invalid index");

return;

}

};

match command {

"MINSERT" => {

db.get\_mut\_or\_create\_m(name).insert(index, value.to\_string());

println!("{}", value);

}

"MSET" => {

db.get\_mut\_or\_create\_m(name).set(index, value.to\_string());

println!("set");

}

\_ => unreachable!(),

}

save\_db(&db, &file\_path);

}

"FPUSHAFTER" | "FPUSHBEFORE" | "LPUSHAFTER" | "LPUSHBEFORE" if parts.len() >= 4 => {

let name = parts[1];

let target = parts[2];

let value = parts[3];

match command {

"FPUSHAFTER" => {

db.get\_mut\_or\_create\_f(name).push\_after(target, value.to\_string());

println!("{}", value);

}

"FPUSHBEFORE" => {

db.get\_mut\_or\_create\_f(name).push\_before(target, value.to\_string());

println!("{}", value);

}

"LPUSHAFTER" => {

db.get\_mut\_or\_create\_l(name).push\_after(target, value.to\_string());

println!("{}", value);

}

"LPUSHBEFORE" => {

db.get\_mut\_or\_create\_l(name).push\_before(target, value.to\_string());

println!("{}", value);

}

\_ => unreachable!(),

}

save\_db(&db, &file\_path);

}

"MDEL" | "MGET" | "FDEL" | "FSEARCH" | "LDEL" | "LSEARCH" | "ISMEMBER" if parts.len() == 3 => {

let name = parts[1];

let param = parts[2];

match command {

"MDEL" => {

let index\_result = param.parse::<usize>();

let index = match index\_result {

Ok(idx) => idx,

Err(\_) => {

println!("Invalid index");

return;

}

};

db.get\_mut\_or\_create\_m(name).remove(index);

println!("deleted");

}

"MGET" => {

let index\_result = param.parse::<usize>();

let index = match index\_result {

Ok(idx) => idx,

Err(\_) => {

println!("Invalid index");

return;

}

};

let val = db.get\_mut\_or\_create\_m(name).get(index);

println!("{}", val);

}

"FDEL" => {

db.get\_mut\_or\_create\_f(name).delete(param);

println!("deleted");

}

"FSEARCH" => {

let found = db.get\_mut\_or\_create\_f(name).search(param);

println!("{}", if found { "TRUE" } else { "FALSE" });

}

"LDEL" => {

db.get\_mut\_or\_create\_l(name).delete(param);

println!("deleted");

}

"LSEARCH" => {

let found = db.get\_mut\_or\_create\_l(name).search(param);

println!("{}", if found { "TRUE" } else { "FALSE" });

}

"ISMEMBER" => {

let found = db.get\_mut\_or\_create\_t(name).search(param);

println!("{}", if found { "TRUE" } else { "FALSE" });

}

\_ => unreachable!(),

}

save\_db(&db, &file\_path);

}

"MLEN" | "FGET" | "LGET" | "SPOP" | "SGET" | "QPOP" | "QGET" | "TGET" | "TISFULL" | "PRINT" if parts.len() == 2 => {

let name = parts[1];

match command {

"MLEN" => {

let len = db.get\_mut\_or\_create\_m(name).len();

println!("{}", len);

}

"FGET" => {

let read = db.get\_mut\_or\_create\_f(name).read();

println!("{}", read);

}

"LGET" => {

let read = db.get\_mut\_or\_create\_l(name).read();

println!("{}", read);

}

"SPOP" => {

if let Some(val) = db.get\_mut\_or\_create\_s(name).pop() {

println!("{}", val);

} else {

println!("empty");

}

}

"SGET" => {

if let Some(val) = db.get\_mut\_or\_create\_s(name).get() {

println!("{}", val);

} else {

println!("empty");

}

}

"QPOP" => {

if let Some(val) = db.get\_mut\_or\_create\_q(name).pop() {

println!("{}", val);

} else {

println!("empty");

}

}

"QGET" => {

if let Some(val) = db.get\_mut\_or\_create\_q(name).get() {

println!("{}", val);

} else {

println!("empty");

}

}

"TGET" => {

let read = db.get\_mut\_or\_create\_t(name).read();

println!("{}", read);

}

"TISFULL" => {

let is\_full = db.get\_mut\_or\_create\_t(name).is\_full();

println!("{}", if is\_full { "TRUE" } else { "FALSE" });

}

"PRINT" => {

if let Some(structure) = db.get(name) {

let read = match structure {

DataStructure::M(a) => a.read(),

DataStructure::F(f) => f.read(),

DataStructure::L(l) => l.read(),

DataStructure::S(s) => s.read(),

DataStructure::Q(q) => q.read(),

DataStructure::T(t) => t.read(),

};

println!("{}", read);

} else {

println!("not found");

}

}

\_ => unreachable!(),

}

if command != "PRINT" {

save\_db(&db, &file\_path);

}

}

\_ => {

println!("Unknown command or wrong number of arguments: {}", command);

}

}

}

### Результат выполнения программы на C++:

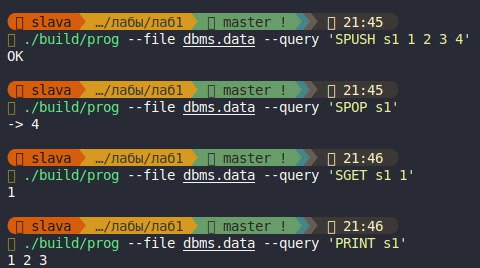


Рисунок 1 – реализация команд для стэка на C++

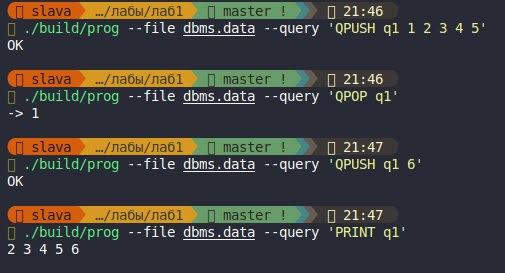


Рисунок 2 – реализация команд для очереди на C++

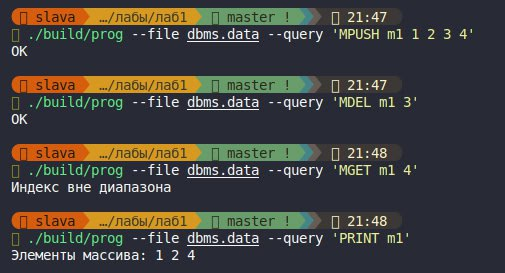


Рисунок 3 – реализация команд для массива на C++

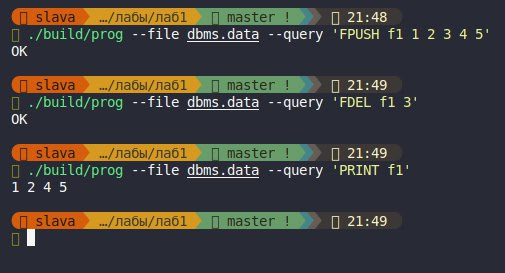


Рисунок 4 – реализация команд для односвязного списка на C++

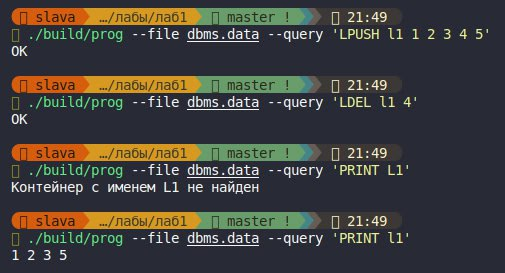


Рисунок 5 – реализация команд для двухсвязного списка на C++

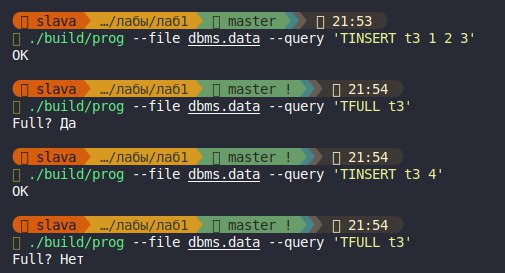


Рисунок 6 – реализация команд для Full Binary Tree на C++

### Реузльтат выполнение программы на Rust:

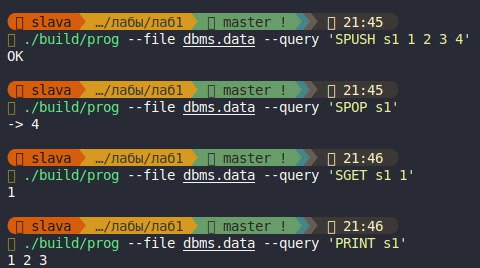


Рисунок 7 – реализация команд для стэка на Rust

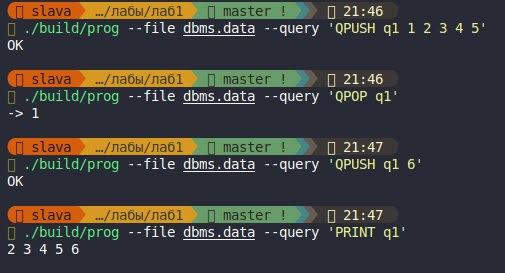


Рисунок 8 – реализация команд для очереди на Rust

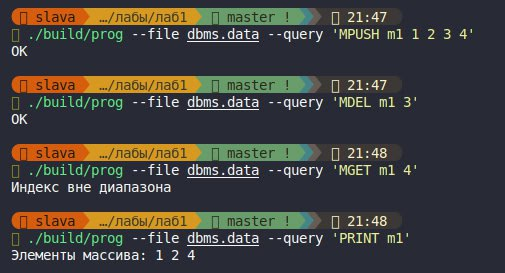


Рисунок 9 – реализация команд для массива на Rust

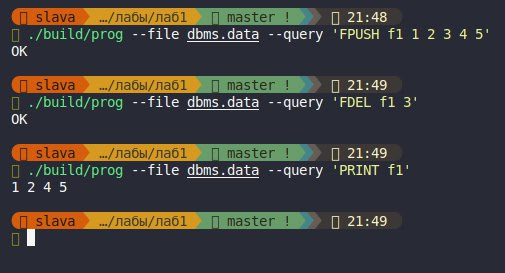


Рисунок 10 – реализация команд для односвязного списка на Rust

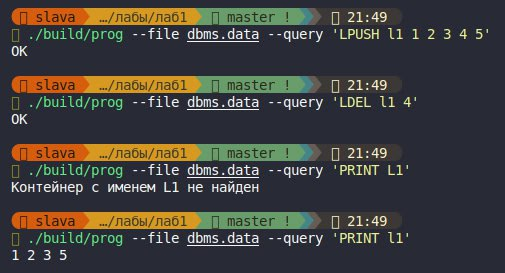


Рисунок 11 – реализация команд для двухсвязного списка на Rust

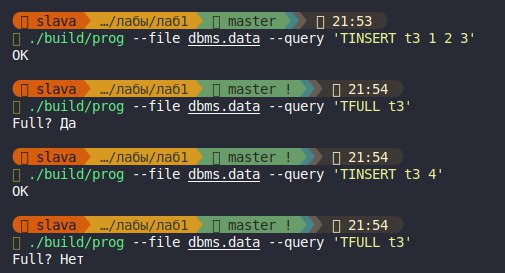


Рисунок 12 – реализация команд для Full Binary Tree на Rust

## Реализация задания 3

|  |  |  |  |
| --- | --- | --- | --- |
| Тип контейнера | Добавление | Удаление | Чтение |
| Массив | MPUSH: O(1) или O(n) при необходимости расширения | MDEL: O(n) | MGET: O(1) |
| Односвязный список | FPUSH: O(1) | FDEL: O(n) | FGET: O(n) |
| Двусвязный список | LPUSH: O(1) | LDEL: O(n) | LGET: O(n) |
| Очередь | QPUSH: O(1) | QPOP: O(1) | QGET: O(1) |
| Дерево | TINSERT: O(log n) | TDEL: O(log n) | TGET: O(log n) |
| Стэк | SPUSH: O(1) | SPOP: O(1) | SGET: O(n) |

## Реализация задания 4

Массив:

• Достоинства: Простота реализации, быстрый доступ по индексу (O(1)), эффективен для фиксированного размера.

• Недостатки: Сложность изменения размера (O(n)), ограниченная гибкость.

• Сфера применения: Хранение данных с известным размером, например, буферы или таблицы.

• Односвязный список:

• Достоинства: Гибкость при добавлении/удалении (O(1) для головы), динамическая структура.

• Недостатки: Медленный доступ (O(n)), сложность обратного обхода.

• Сфера применения: Реализация очередей, стека, где требуется частое добавление/удаление.

• Двусвязный список:

• Достоинства: Поддержка двустороннего обхода, гибкость при удалении/добавлении.

• Недостатки: Дополнительная память на указатели, медленный доступ (O(n)).

• Сфера применения: Реализация сложных очередей, редактируемые списки.

• Очередь:

• Достоинства: Простота реализации, O(1) для добавления и удаления.

• Недостатки: Ограниченная функциональность, сложность при приоритетах.

• Сфера применения: Управление задачами, обработка запросов.

• Стек:

• Достоинства: Простота, O(1) для операций, удобен для рекурсии.

• Недостатки: Ограниченная доступность (только верхний элемент).

• Сфера применения: Обратная польская нотация, вызов функций.

• Дерево:

• Достоинства: Эффективный поиск, вставка, удаление (O(log n) для сбалансированных), иерархическая структура.

• Недостатки: Сложность поддержания баланса, больше памяти.

• Сфера применения: Базы данных, поисковые структуры, графы.

## Реализация задания 5

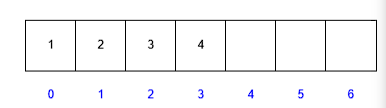


Рисунок 13 – визуализация добавления в стэк

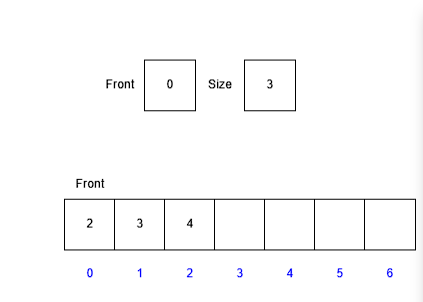


Рисунок 14 – визуализация добавления в очередь

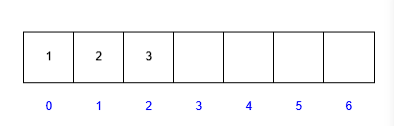


Рисунок 15 – визуализация добавления в массив

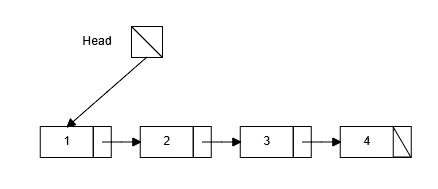


Рисунок 16 – визуализация добавления в односвязный список

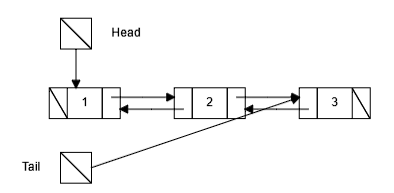


Рисунок 17 – визуализация добавления в двусвязный список

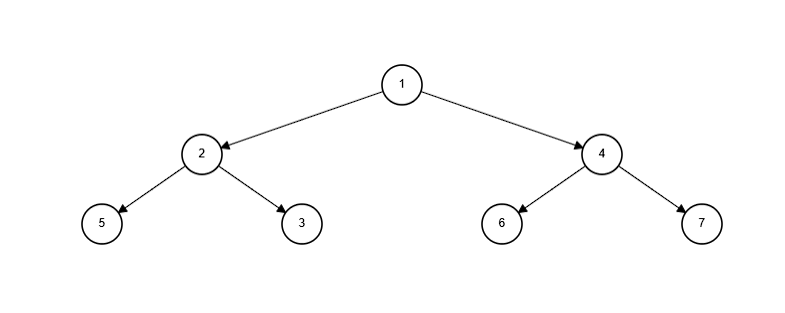


Рисунок 18 – визуализация добавления в дерево

# Вывод

В результате данной лабораторной работы по программированию были выполнены все поставленные цели и задачи, нацеленные на изучение алгоритмов формирования и работы с абстрактными структурами данных.