МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации

**

**ЛАБОРАТОРНАЯ РАБОТА №1**

«Абстрактные структуры данных»

**по дисциплине: «*Программирование*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБc-322», «АВТФ»  *Михайлов Егор Сергеевич*  «24» октября 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент кафедры ЗИ*  *Исаев Глеб Андреевич*  «\_\_\_» \_\_\_\_\_\_ 2024г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2024

**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 (С, Go или Rust).

3. Протестировать работу программы на различных исходных данных.

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

**Ссылка на GitHub:** [**https://github.com/Discorre/firLab**](https://github.com/Discorre/firLab)

**Задание 1 (С++):**

myHashMap.hpp:

#ifndef MYHASHMAP\_HPP

#define MYHASHMAP\_HPP

template <typename K, typename V>

class MyHashMap {

private:

    MyList<K, V>\* table; // Массив списков

    int capacity; // Вместимость хеш-таблицы

    // Простая хеш-функция

    int hash(const K& key) const {

        int hashValue = 0;

        for (char c : key) { // Простой способ для строк

            hashValue = (hashValue \* 31 + c) % capacity;

        }

        return hashValue;

    }

public:

    // Конструктор

    MyHashMap(int cap = 10) : capacity(cap) {

        table = new MyList<K, V>[capacity]; // Создаем массив списков

    }

    // Деструктор

    ~MyHashMap() {

        delete[] table; // Освобождаем память

    }

    // Метод для добавления элемента (HSET)

    void HSET(const K& key, const V& value) {

        table[hash(key)].push\_back(key, value);

    }

    // Метод для получения элемента по ключу (HGET)

    V HGET(const K& key) {

        V value{};

        if (table[hash(key)].find(key, value)) {

            return value;

        }

        else {

            throw std::out\_of\_range("Key not found");

        }

    }

    // Метод для удаления элемента по ключу (HDEL)

    void HDEL(const K& key) {

        if (!table[hash(key)].remove(key)) {

            throw std::out\_of\_range("Key not found");

        }

    }

    // Метод для печати хеш-таблицы

    void print() const {

        for (int i = 0; i < capacity; i++) {

            std::cout << "Bucket " << i << ": ";

            table[i].print();

        }

        std::cout << std::endl;

    }

    // Сохранение хеш-таблицы в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        for (int i = 0; i < capacity; i++) {

            // Получаем список из текущей корзины

            MyList<K, V> bucket = table[i];

            // Сохраняем все пары ключ-значение из корзины в файл

            for (int j = 0; j < bucket.size(); j++) {

                K key;

                V value;

                if (bucket.findAt(j, key, value)) {

                    file << key << " " << value << "\n";

                }

            }

        }

        file.close();

    }

    // Загрузка хеш-таблицы из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущую хеш-таблицу перед загрузкой

        K key;

        V value;

        while (file >> key >> value) {

            HSET(key, value); // Добавляем пары ключ-значение в хеш-таблицу

        }

        file.close();

    }

    // Очистка хеш-таблицы

    void clear() {

        for (int i = 0; i < capacity; i++) {

            table[i].clear(); // Очищаем каждую корзину

        }

    }

};

#endif

myArray.hpp:

#ifndef MYARRAY\_HPP

#define MYARRAY\_HPP

template <typename T>

class MyArray {

private:

    T\* data;        // Указатель на массив элементов

    int capacity;   // Текущая емкость массива

    int length;     // Текущая длина массива

public:

    // Конструктор по умолчанию

    MyArray(int size = 1) : capacity(10), length(0) {

        data = new T[capacity];  // Инициализация массива с начальной емкостью

    }

    // Деструктор

    ~MyArray() {

        delete[] data; // Освобождение выделенной памяти

    }

    // Метод для добавления элемента в конец массива

    void MPUSH(const T& element) {

        if (length == capacity) {

            resize(); // Увеличение емкости, если массив полон

        }

        data[length++] = element; // Добавление элемента и увеличение длины

    }

    // Метод для добавления элемента по индексу

    void MPUSH(int index, const T& element) {

        if (index < 0 || index > length) {

            throw std::out\_of\_range("Index out of range");

        }

        if (length == capacity) {

            resize(); // Увеличение емкости, если массив полон

        }

        // Сдвиг элементов вправо, чтобы освободить место

        for (int i = length; i > index; --i) {

            data[i] = data[i - 1];

        }

        data[index] = element; // Вставка элемента

        length++; // Увеличение длины

    }

    // Метод для удаления элемента по индексу

    void MDEL(int index) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        // Сдвиг элементов влево, чтобы заполнить пробел

        for (int i = index; i < length - 1; ++i) {

            data[i] = data[i + 1];

        }

        length--; // Уменьшение длины

    }

    // Метод для получения элемента по индексу (неконстантная версия)

    T& MGET(int index) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        return data[index]; // Возвращаем элемент

    }

    // Константная версия метода MGET

    const T& MGET(int index) const {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        return data[index]; // Возвращаем элемент

    }

    // Метод для замены элемента по индексу

    void MRESET(int index, const T& element) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        data[index] = element; // Замена элемента

    }

    // Увеличение емкости массива

    void resize() {

        capacity \*= 2; // Увеличиваем емкость в два раза

        T\* newData = new T[capacity]; // Создаем новый массив с новой емкостью

        for (int i = 0; i < length; i++) {

            newData[i] = data[i]; // Копируем старые данные в новый массив

        }

        delete[] data; // Освобождаем старый массив

        data = newData; // Перенаправляем указатель на новый массив

    }

    // Метод для получения текущей длины массива

    int lengtharr() const {

        return length; // Возвращаем длину

    }

    // Сохранение массива в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        for (size\_t i = 0; i < length; ++i) {

            file << data[i] << "\n";

        }

        file.close();

    }

    // Очистка массива

    void clear() {

        length = 0; // Просто сбрасываем размер

    }

    // Загрузка массива из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущий массив перед загрузкой

        T value;

        while (file >> value) {

            MPUSH(value); // Добавляем элементы в массив

        }

        file.close();

    }

    // Метод для чтения всех элементов массива

    void print() const {

        for (int i = 0; i < length; i++) {

            std::cout << data[i] << " "; // Выводим каждый элемент

        }

        std::cout << std::endl; // Переход на новую строку

    }

};

#endif

myDoublyLinkedlist.hpp:

#ifndef MYDOUBLYLINKEDLIST\_HPP

#define MYDOUBLYLINKEDLIST\_HPP

template <typename T>

class MyDoublyLinkedList {

private:

    struct Node {

        T data;          // Данные узла

        Node\* next;      // Указатель на следующий узел

        Node\* prev;      // Указатель на предыдущий узел

        Node(const T& value) : data(value), next(nullptr), prev(nullptr) {}

    };

    Node\* head;         // Указатель на голову списка

    Node\* tail;         // Указатель на хвост списка

    size\_t size;        // Текущий размер списка

public:

    MyDoublyLinkedList() : head(nullptr), tail(nullptr), size(0) {}

    ~MyDoublyLinkedList() {

        clear();

    }

    // Добавление элемента в голову

    void LPUSHHEAD(const T& value) {

        Node\* newNode = new Node(value);

        if (!head) {

            head = tail = newNode;

        }

        else {

            newNode->next = head;

            head->prev = newNode;

            head = newNode;

        }

        ++size;

    }

    // Добавление элемента в хвост

    void LPUSHTAIL(const T& value) {

        Node\* newNode = new Node(value);

        if (!tail) {

            head = tail = newNode;

        }

        else {

            tail->next = newNode;

            newNode->prev = tail;

            tail = newNode;

        }

        ++size;

    }

    // Удаление элемента с головы

    void LDELHEAD() {

        if (!head) {

            throw std::runtime\_error("List is empty");

        }

        Node\* temp = head;

        head = head->next;

        if (head) {

            head->prev = nullptr;

        }

        else {

            tail = nullptr; // Если список стал пустым

        }

        delete temp;

        --size;

    }

    // Удаление элемента с хвоста

    void LDELTAIL() {

        if (!tail) {

            throw std::runtime\_error("List is empty");

        }

        Node\* temp = tail;

        tail = tail->prev;

        if (tail) {

            tail->next = nullptr;

        }

        else {

            head = nullptr; // Если список стал пустым

        }

        delete temp;

        --size;

    }

    // Удаление элемента по индексу

    void LDEL(int index) {

        if (index < 0 || index >= size) {

            throw std::out\_of\_range("Index out of range");

        }

        if (index == 0) {

            LDELHEAD();

            return;

        }

        if (index == size - 1) {

            LDELTAIL();

            return;

        }

        Node\* current = head;

        for (int i = 0; i < index; ++i) {

            current = current->next;

        }

        current->prev->next = current->next;

        current->next->prev = current->prev;

        delete current;

        --size;

    }

    // Поиск элемента

    bool contains(const T& value) const {

        Node\* current = head;

        while (current) {

            if (current->data == value) {

                return true;

            }

            current = current->next;

        }

        return false;

    }

    // Чтение элемента по индексу

    T LGET(int index) const {

        if (index < 0 || index >= size) {

            throw std::out\_of\_range("Index out of range");

        }

        Node\* current = head;

        for (int i = 0; i < index; ++i) {

            current = current->next;

        }

        return current->data;

    }

    // Печать элементов списка

    void print() const {

        Node\* current = head;

        while (current) {

            std::cout << current->data << " ";

            current = current->next;

        }

        std::cout << std::endl;

    }

    // Получение текущего размера списка

    size\_t getSize() const {

        return size;

    }

    // Сохранение списка в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        Node\* current = head;

        while (current) {

            file << current->data << "\n";

            current = current->next;

        }

        file.close();

    }

    // Загрузка списка из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущий список перед загрузкой

        T value;

        while (file >> value) {

            LPUSHTAIL(value); // Добавляем элементы в хвост списка

        }

        file.close();

    }

    // Очистка списка

    void clear() {

        while (head) {

            LDELHEAD();

        }

    }

};

#endif

myFullBinaryTree.hpp  
#ifndef MYFULLBINARYTREE\_HPP

#define MYFULLBINARYTREE\_HPP

template <typename T>

struct Node {

    T data;

    Node\* left;

    Node\* right;

    Node(T value) : data(value), left(nullptr), right(nullptr) {}

};

template <typename T>

class FullBinaryTree {

private:

    Node<T>\* root;  // Корень дерева

    // Вспомогательная функция для вставки элемента

    Node<T>\* insert(Node<T>\* node, T value) {

        if (node == nullptr) {

            return new Node<T>(value); // Указываем, что Node - это Node<T>

        }

        // Вставляем элемент в подходящее место (сначала левый, затем правый)

        if (node->left == nullptr) {

            node->left = insert(node->left, value);

        }

        else if (node->right == nullptr) {

            node->right = insert(node->right, value);

        }

        else {

            node->left = insert(node->left, value);  // Если оба узла заняты, идем в левое поддерево

        }

        return node;

    }

    // Вспомогательная функция для поиска элемента

    bool search(Node<T>\* node, T value) const {

        if (node == nullptr) {

            return false;

        }

        if (node->data == value) {

            return true;

        }

        // Поиск в левом и правом поддереве

        return search(node->left, value) || search(node->right, value);

    }

    // Вспомогательная функция для удаления элемента (удаление узлов с учетом 2 потомков)

    Node<T>\* deleteNode(Node<T>\* node, T value) {

        if (node == nullptr) {

            return nullptr;

        }

        // Поиск удаляемого узла

        if (node->data == value) {

            // Если у узла нет потомков

            if (node->left == nullptr && node->right == nullptr) {

                delete node;

                return nullptr;

            }

            // Если только один потомок

            if (node->left == nullptr) {

                Node<T>\* temp = node->right;

                delete node;

                return temp;

            }

            else if (node->right == nullptr) {

                Node<T>\* temp = node->left;

                delete node;

                return temp;

            }

            // Если 2 потомка, находим минимальное значение в правом поддереве

            Node<T>\* minNode = findMin(node->right);

            node->data = minNode->data;

            node->right = deleteNode(node->right, minNode->data);

        }

        else {

            node->left = deleteNode(node->left, value);

            node->right = deleteNode(node->right, value);

        }

        return node;

    }

    // Вспомогательная функция для поиска минимального значения

    Node<T>\* findMin(Node<T>\* node) {

        while (node->left != nullptr) {

            node = node->left;

        }

        return node;

    }

    // In-order обход для чтения

    void inOrder(Node<T>\* node) const {

        if (node != nullptr) {

            inOrder(node->left);

            std::cout << node->data << " ";

            inOrder(node->right);

        }

    }

    // Pre-order обход для сохранения в файл

    void savePreOrder(Node<T>\* node, std::ofstream& file) const {

        if (node == nullptr) {

            file << "# ";  // Специальный символ для обозначения null

            return;

        }

        file << node->data << " ";  // Сохраняем данные узла

        savePreOrder(node->left, file);   // Сохраняем левое поддерево

        savePreOrder(node->right, file);  // Сохраняем правое поддерево

    }

    // Pre-order обход для загрузки из файла

    Node<T>\* loadPreOrder(std::ifstream& file) {

        std::string val;

        if (!(file >> val) || val == "#") {

            return nullptr;  // Если считали '#', возвращаем null

        }

        Node<T>\* node = new Node<T>(std::stoi(val));  // Преобразуем строку в число

        node->left = loadPreOrder(file);   // Загружаем левое поддерево

        node->right = loadPreOrder(file);  // Загружаем правое поддерево

        return node;

    }

    // Вспомогательная функция для проверки, является ли дерево полным

    bool isFull(Node<T>\* node) const {

        if (node == nullptr) {

            return true;  // Пустое дерево - полное дерево

        }

        if ((node->left == nullptr && node->right != nullptr) || (node->left != nullptr && node->right == nullptr)) {

            return false;  // Если у узла один потомок, дерево не является полным

        }

        // Рекурсивно проверяем левое и правое поддерево

        return isFull(node->left) && isFull(node->right);

    }

public:

    // Конструктор

    FullBinaryTree() : root(nullptr) {}

    // Деструктор (рекурсивное удаление всех узлов)

    ~FullBinaryTree() {

        clear(root);

    }

    // Очистка дерева

    void clear(Node<T>\* node) {

        if (node != nullptr) {

            clear(node->left);

            clear(node->right);

            delete node;

        }

    }

    // Операция TINSERT — добавление элемента

    void TINSERT(T value) {

        root = insert(root, value);

    }

    // Операция TDEL — удаление элемента

    void TDEL(T value) {

        root = deleteNode(root, value);

    }

    // Операция TGET — поиск элемента

    bool TGET(T value) const {

        return search(root, value);

    }

    // Чтение дерева (in-order traversal)

    void print() const {

        inOrder(root);

        std::cout << std::endl;

    }

    // Проверка на полноту дерева

    bool isFull() const {

        return isFull(root);

    }

    // Сохранение дерева в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        savePreOrder(root, file);  // Сохраняем дерево в pre-order обходе

        file.close();

    }

    // Загрузка дерева из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(root);  // Очищаем текущее дерево перед загрузкой

        root = loadPreOrder(file);  // Загружаем дерево из файла

        file.close();

    }

};

#endif

myListKV.hpp:

#ifndef MYLISTKV\_HPP

#define MYLISTKV\_HPP

template <typename K, typename V>

class MyList {

private:

    struct Node {

        K key;

        V value;

        Node\* next;

        Node(K key, V value) : key(key), value(value), next(nullptr) {}

    };

    Node\* head;

    Node\* tail;

    int length;

public:

    // Конструктор

    MyList() : head(nullptr), tail(nullptr), length(0) {}

    // Деструктор

    ~MyList() {

        clear();

    }

    // Добавление пары ключ-значение в конец списка

    void push\_back(const K& key, const V& value) {

        Node\* newNode = new Node(key, value);

        if (tail == nullptr) {

            head = tail = newNode;

        }

        else {

            tail->next = newNode;

            tail = newNode;

        }

        length++;

    }

    // Поиск элемента по ключу

    bool find(const K& key, V& value) const {

        Node\* current = head;

        while (current != nullptr) {

            if (current->key == key) {

                value = current->value;

                return true;

            }

            current = current->next;

        }

        return false;

    }

    // Получение элемента по индексу

    bool findAt(int index, K& key, V& value) const {

        if (index < 0 || index >= length) {

            return false; // Индекс вне диапазона

        }

        Node\* current = head;

        for (int i = 0; i < index; i++) {

            current = current->next;

        }

        key = current->key;

        value = current->value;

        return true;

    }

    // Удаление элемента по ключу

    bool remove(const K& key) {

        Node\* current = head;

        Node\* prev = nullptr;

        while (current != nullptr) {

            if (current->key == key) {

                if (prev != nullptr) {

                    prev->next = current->next;

                }

                else {

                    head = current->next;

                }

                if (current == tail) {

                    tail = prev;

                }

                delete current;

                length--;

                return true;

            }

            prev = current;

            current = current->next;

        }

        return false;

    }

    // Метод для печати элементов списка

    void print() const {

        Node\* current = head;

        while (current != nullptr) {

            std::cout << "{" << current->key << ": " << current->value << "} ";

            current = current->next;

        }

        std::cout << std::endl;

    }

    // Получение длины списка

    int size() const {

        return length;

    }

    // Очистка списка

    void clear() {

        while (head != nullptr) {

            Node\* temp = head;

            head = head->next;

            delete temp;

        }

        tail = nullptr;

        length = 0;

    }

};

#endif

myQueue.hpp:

#ifndef MYQUEUE\_HPP

#define MYQUEUE\_HPP

template <typename T>

class Queue {

private:

    T\* data;        // Указатель на динамический массив

    size\_t front;   // Индекс первого элемента

    size\_t back;    // Индекс следующего элемента для добавления

    size\_t capacity; // Вместимость массива

    size\_t size;     // Текущий размер очереди

    // Увеличение размера массива

    void resize() {

        capacity = capacity == 0 ? 1 : capacity \* 2; // Увеличиваем вдвое

        T\* newData = new T[capacity]; // Новый массив

        for (size\_t i = 0; i < size; ++i) {

            newData[i] = data[(front + i) % capacity]; // Копируем старые данные

        }

        delete[] data; // Удаляем старый массив

        data = newData; // Указатель на новый массив

        front = 0; // Обнуляем индекс первого элемента

        back = size; // Обновляем индекс добавления

    }

public:

    Queue() : data(nullptr), front(0), back(0), capacity(0), size(0) {}

    ~Queue() {

        delete[] data; // Освобождаем память

    }

    void Q\_PUSH(const T& value) {

        if (size == capacity) {

            resize(); // Увеличиваем размер при необходимости

        }

        data[back] = value; // Добавляем новый элемент

        back = (back + 1) % capacity; // Обновляем индекс добавления

        ++size; // Увеличиваем размер очереди

    }

    void Q\_DEL() {

        if (size == 0) {

            throw std::out\_of\_range("Queue is empty");

        }

        front = (front + 1) % capacity; // Обновляем индекс первого элемента

        --size; // Уменьшаем размер очереди

    }

    T Q\_POP() {

        if (size == 0) {

            throw std::out\_of\_range("Queue is empty");

        }

        T value = data[front]; // Считываем значение

        Q\_DEL(); // Удаляем элемент

        return value; // Возвращаем значение

    }

    size\_t getSize() const {

        return size; // Возвращаем текущий размер очереди

    }

    bool isEmpty() const {

        return size == 0; // Проверка на пустоту

    }

    // Метод для печати элементов очереди

    void print() const {

        if (size == 0) {

            std::cout << "Queue is empty." << std::endl;

            return;

        }

        std::cout << "Queue elements: ";

        for (size\_t i = 0; i < size; ++i) {

            std::cout << data[(front + i) % capacity] << " "; // Печатаем каждый элемент

        }

        std::cout << std::endl;

    }

    // Сохранение очереди в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        for (size\_t i = 0; i < size; ++i) {

            file << data[(front + i) % capacity] << "\n"; // Сохраняем каждый элемент

        }

        file.close();

    }

    // Загрузка очереди из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущую очередь перед загрузкой

        T value;

        while (file >> value) {

            Q\_PUSH(value); // Добавляем каждый считанный элемент в очередь

        }

        file.close();

    }

    // Очистка очереди

    void clear() {

        delete[] data; // Освобождаем память

        data = nullptr; // Обнуляем указатель

        front = back = capacity = size = 0; // Сбрасываем все параметры

    }

};

#endif

mySinglyLinkedList.hpp:

#ifndef MYSINGLYLINKEDLIST\_HPP

#define MYSINGLYLINKEDLIST\_HPP

template <typename T>

class MySinglyLinkedList {

private:

    struct Node {

        T data;        // Данные узла

        Node\* next;    // Указатель на следующий узел

        Node(const T& value) : data(value), next(nullptr) {}

    };

    Node\* head;       // Указатель на голову списка

    Node\* tail;       // Указатель на хвост списка

    size\_t size;      // Текущий размер списка

public:

    MySinglyLinkedList() : head(nullptr), tail(nullptr), size(0) {}

    ~MySinglyLinkedList() {

        clear();

    }

    // Добавление элемента в голову

    void LPUSHHEAD(const T& value) {

        Node\* newNode = new Node(value);

        if (!head) {

            head = tail = newNode;

        }

        else {

            newNode->next = head;

            head = newNode;

        }

        ++size;

    }

    // Добавление элемента в хвост

    void LPUSHTAIL(const T& value) {

        Node\* newNode = new Node(value);

        if (!tail) {

            head = tail = newNode;

        }

        else {

            tail->next = newNode;

            tail = newNode;

        }

        ++size;

    }

    // Удаление элемента с головы

    void LDELHEAD() {

        if (!head) {

            throw std::runtime\_error("List is empty");

        }

        Node\* temp = head;

        head = head->next;

        delete temp;

        --size;

        if (!head) {

            tail = nullptr; // Если список стал пустым

        }

    }

    // Удаление элемента с хвоста

    void LDELTAIL() {

        if (!tail) {

            throw std::runtime\_error("List is empty");

        }

        Node\* temp = tail;

        if (head == tail) {

            head = tail = nullptr; // Если список стал пустым

        }

        else {

            Node\* current = head;

            while (current->next != tail) {

                current = current->next;

            }

            current->next = nullptr;

            tail = current;

        }

        delete temp;

        --size;

    }

    // Удаление элемента по индексу

    void LDEL(int index) {

        if (index < 0 || index >= size) {

            throw std::out\_of\_range("Index out of range");

        }

        if (index == 0) {

            LDELHEAD();

            return;

        }

        Node\* current = head;

        for (int i = 0; i < index - 1; ++i) {

            current = current->next;

        }

        Node\* temp = current->next;

        current->next = temp->next;

        if (temp == tail) {

            tail = current; // Если удален хвост

        }

        delete temp;

        --size;

    }

    // Поиск элемента

    bool contains(const T& value) const {

        Node\* current = head;

        while (current) {

            if (current->data == value) {

                return true;

            }

            current = current->next;

        }

        return false;

    }

    // Чтение элемента по индексу

    T LGET(int index) const {

        if (index < 0 || index >= size) {

            throw std::out\_of\_range("Index out of range");

        }

        Node\* current = head;

        for (int i = 0; i < index; ++i) {

            current = current->next;

        }

        return current->data;

    }

    // Печать элементов списка

    void print() const {

        Node\* current = head;

        while (current) {

            std::cout << current->data << " ";

            current = current->next;

        }

        std::cout << std::endl;

    }

    // Получение текущего размера списка

    size\_t getSize() const {

        return size;

    }

    // Сохранение списка в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        Node\* current = head;

        while (current) {

            file << current->data << "\n";

            current = current->next;

        }

        file.close();

    }

    // Загрузка списка из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущий список перед загрузкой

        T value;

        while (file >> value) {

            LPUSHTAIL(value); // Добавляем элементы в хвост списка

        }

        file.close();

    }

    // Очистка списка

    void clear() {

        while (head) {

            LDELHEAD();

        }

    }

};

#endif

myStack.hpp:

#ifndef MYSTACK\_HPP

#define MYSTACK\_HPP

template <typename T>

class MyStack {

private:

    T\* data;        // Указатель на массив элементов

    int capacity;   // Текущая емкость массива

    int length;     // Текущая длина стека

public:

    // Конструктор по умолчанию

    MyStack() : capacity(10), length(0) {

        data = new T[capacity];  // Инициализация массива с начальной емкостью

    }

    // Деструктор

    ~MyStack() {

        delete[] data; // Освобождение выделенной памяти

    }

    // Метод для добавления элемента в стек

    void SPUSH(const T& element) {

        if (length == capacity) {

            resize(); // Увеличение емкости, если стек полон

        }

        data[length++] = element; // Добавление элемента и увеличение длины

    }

    // Метод для удаления элемента из стека

    void SDEL() {

        if (length == 0) {

            throw std::out\_of\_range("Stack is empty");

        }

        length--; // Уменьшение длины (удаление последнего элемента)

    }

    // Метод для чтения (получения) верхнего элемента стека

    T& SPOP() {

        if (length == 0) {

            throw std::out\_of\_range("Stack is empty");

        }

        return data[length - 1]; // Возвращаем верхний элемент

    }

    // Метод для получения текущего размера стека

    int size() const {

        return length; // Возвращаем длину стека

    }

    // Метод для увеличения емкости стека

    void resize() {

        capacity \*= 2; // Увеличиваем емкость в два раза

        T\* newData = new T[capacity]; // Создаем новый массив с новой емкостью

        for (int i = 0; i < length; i++) {

            newData[i] = data[i]; // Копируем старые данные в новый массив

        }

        delete[] data; // Освобождаем старый массив

        data = newData; // Перенаправляем указатель на новый массив

    }

    // Метод для печати элементов стека

    void print() const {

        if (length == 0) {

            std::cout << "Stack is empty." << std::endl;

            return;

        }

        std::cout << "Stack elements: ";

        for (int i = 0; i < length; i++) {

            std::cout << data[i] << " "; // Выводим каждый элемент

        }

        std::cout << std::endl;

    }

    // Сохранение стека в файл

    void saveToFile(const std::string& filename) const {

        std::ofstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for writing");

        }

        for (int i = 0; i < length; i++) {

            file << data[i] << "\n"; // Сохраняем каждый элемент

        }

        file.close();

    }

    // Загрузка стека из файла

    void loadFromFile(const std::string& filename) {

        std::ifstream file(filename);

        if (!file) {

            throw std::runtime\_error("Unable to open file for reading");

        }

        clear(); // Очищаем текущий стек перед загрузкой

        T value;

        while (file >> value) {

            SPUSH(value); // Добавляем каждый считанный элемент в стек

        }

        file.close();

    }

    // Очистка стека

    void clear() {

        delete[] data; // Освобождаем память

        data = new T[capacity]; // Создаем новый массив с той же емкостью

        length = 0; // Сбрасываем длину

    }

};

#endif

myVector.hpp

#ifndef MYVECTOR\_HPP

#define MYVECTOR\_HPP

template <typename T>

class MyVector {

private:

    T\* data;        // Указатель на массив элементов

    int capacity;   // Текущая емкость массива

    int length;     // Текущая длина массива

public:

    // Конструктор по умолчанию

    MyVector(int size = 10) : capacity(10), length(0) {

        data = new T[capacity];  // Инициализация массива с начальной емкостью

    }

    // Деструктор

    ~MyVector() {

        delete[] data; // Освобождение выделенной памяти

    }

    // Метод для добавления элемента в конец массива

    void push\_back(const T& element) {

        if (length == capacity) {

            resize(); // Увеличение емкости, если массив полон

        }

        data[length++] = element; // Добавление элемента и увеличение длины

    }

    // Метод для добавления элемента по индексу

    void insert(int index, const T& element) {

        if (index < 0 || index > length) {

            throw std::out\_of\_range("Index out of range");

        }

        if (length == capacity) {

            resize(); // Увеличение емкости, если массив полон

        }

        // Сдвиг элементов вправо, чтобы освободить место

        for (int i = length; i > index; --i) {

            data[i] = data[i - 1];

        }

        data[index] = element; // Вставка элемента

        length++; // Увеличение длины

    }

    // Метод для удаления элемента по индексу

    void erase(int index) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        // Сдвиг элементов влево, чтобы заполнить пробел

        for (int i = index; i < length - 1; ++i) {

            data[i] = data[i + 1];

        }

        length--; // Уменьшение длины

    }

    // Метод для получения элемента по индексу (неконстантная версия)

    T& get(int index) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        return data[index]; // Возвращаем элемент

    }

    // Константная версия метода get

    const T& get(int index) const {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        return data[index]; // Возвращаем элемент

    }

    // Метод для замены элемента по индексу

    void set(int index, const T& element) {

        if (index < 0 || index >= length) {

            throw std::out\_of\_range("Index out of range");

        }

        data[index] = element; // Замена элемента

    }

    // Увеличение емкости массива

    void resize() {

        capacity \*= 2; // Увеличиваем емкость в два раза

        T\* newData = new T[capacity]; // Создаем новый массив с новой емкостью

        for (int i = 0; i < length; i++) {

            newData[i] = data[i]; // Копируем старые данные в новый массив

        }

        delete[] data; // Освобождаем старый массив

        data = newData; // Перенаправляем указатель на новый массив

    }

    // Метод для получения текущей длины массива

    int size() const {

        return length; // Возвращаем длину

    }

    // Метод для чтения всех элементов массива

    void print() const {

        for (int i = 0; i < length; i++) {

            std::cout << data[i] << " "; // Выводим каждый элемент

        }

        std::cout << std::endl; // Переход на новую строку

    }

    // Метод begin() для работы с диапазоном

    T\* begin() {

        return data;

    }

    // Метод end() для работы с диапазоном

    T\* end() {

        return data + length;

    }

    // Константные версии методов begin() и end()

    const T\* begin() const {

        return data;

    }

    const T\* end() const {

        return data + length;

    }

};

#endif

mainMenu.hpp:

#include <iostream>

#include <stdexcept>

#include <string.h>

#include <string>

#include <fstream>

#include "myListKV.hpp"

#include "myVector.hpp"

#include "myArray.hpp"

#include "mySinglyLinkedList.hpp"

#include "myDoublyLinkedList.hpp"

#include "myQueue.hpp"

#include "myStack.hpp"

#include "myHashMap.hpp"

#include "myFullBinaryTree.hpp"

// Меню для работы с массивом

static void arrayMenu() {

    char choice;

    std::string value;

    int key;

    MyArray<std::string> array;

    do {

        std::cout << "[M] Массив\n1. MPUSH\n2. MPUSH(по индексу)\n3. MDEL\n4. MGET\n5. MRESET\n6. LOADFROMFILE\n7. SAVETOFILE\n8. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            array.MPUSH(value);

            break;

        case '2':

            std::cout << "Введите индекс значение для добавления: ";

            std::cin >> key >> value;

            array.MPUSH(key, value);

            break;

        case '3':

            std::cout << "Введите индекс для удаления: ";

            std::cin >> key;

            array.MDEL(key);

            break;

        case '4':

            std::cout << "Введите индекс для вывода значения: ";

            std::cin >> key;

            std::cout << array.MGET(key) << std::endl;

            break;

        case '5':

            std::cout << "Введите индекс и значение для добавления: ";

            std::cin >> key >> value;

            array.MRESET(key, value);

            break;

        case '6':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            array.loadFromFile(value);

            break;

        case '7':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            array.saveToFile(value);

            break;

        case '8':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '8');

}

// Меню для работы со списком

static void SinglelistMenu() {

    char choice;

    std::string value;

    int key;

    MySinglyLinkedList<std::string> singleList;

    do {

        std::cout << "[L] Список\n1. LPUSHHEAD\n2. LPUSHTEIL\n3. LDELHEAD\n4. LDELTAIL\n5. LDEL\n6. LGET\n\n7. LOADFROMFILE\n8. SAVE2FILE\n9. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            singleList.LPUSHHEAD(value);

            break;

        case '2':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            singleList.LPUSHTAIL(value);

            break;

        case '3':

            singleList.LDELHEAD();

            break;

        case '4':

            singleList.LDELTAIL();

            break;

        case '5':

            std::cout << "Введите индекс для удаления: ";

            std::cin >> key;

            singleList.LDEL(key);

            break;

        case '6':

            std::cout << "Введите ключ для вывода: ";

            std::cin >> key;

            std::cout << singleList.LGET(key) << std::endl;

            break;

        case '7':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            singleList.loadFromFile(value);

            break;

        case '8':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            singleList.saveToFile(value);

            break;

        case '9':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '9');

}

// Меню для работы со списком

static void DoublylistMenu() {

    char choice;

    std::string value;

    int key;

    MyDoublyLinkedList<std::string> doublyList;

    do {

        std::cout << "[L] Список\n1. LPUSHHEAD\n2. LPUSHTAIL\n3. LDELHEAD\n4. LDELTAIL\n5. LDEL\n6. LGET\n7. LOADFROMFILE\n8. SAVETOFILE\n9. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            doublyList.LPUSHHEAD(value);

            break;

        case '2':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            doublyList.LPUSHTAIL(value);

            break;

        case '3':

            doublyList.LDELHEAD();

            break;

        case '4':

            doublyList.LDELTAIL();

            break;

        case '5':

            std::cout << "Введите индекс для удаления: ";

            std::cin >> key;

            doublyList.LDEL(key);

            break;

        case '6':

            std::cout << "Введите индекс для вывода: ";

            std::cin >> key;

            std::cout << doublyList.LGET(key) << std::endl;

            break;

        case '7':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            doublyList.loadFromFile(value);

            break;

        case '8':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            doublyList.saveToFile(value);

            break;

        case '9':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '9');

}

// Меню для работы с очередью

static void queueMenu() {

    char choice;

    std::string value;

    Queue<std::string> queue;

    do {

        std::cout << "[Q] Очередь\n1. QPUSH\n2. QPOP\n3. QGET\n4. LOADFROMFILE\n5. SAVETOFILE\n6. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            queue.Q\_PUSH(value);

            break;

        case '2':

            queue.Q\_DEL();

            break;

        case '3':

            std::cout << queue.Q\_POP() << std::endl;

            break;

        case '4':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            queue.loadFromFile(value);

            break;

        case '5':

            std::cout << "Введите название файла: ";

            std::cin >> value;

            queue.saveToFile(value);

            break;

        case '6':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '6');

}

// Меню для работы с деревом

static void treeMenu() {

    char choice;

    int value;

    std::string filename;

    FullBinaryTree<int> tree;

    do {

        std::cout << "[T] Дерево\n1. TINSERT\n2. TDEL\n3. TGET\n4. LOADFROMFILE\n5. SAVETOFILE\n6. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для вставки: ";

            std::cin >> value;

            tree.TINSERT(value);

            break;

        case '2':

            std::cout << "Введите значение для удаления: ";

            std::cin >> value;

            tree.TDEL(value);

            break;

        case '3':

            std::cout << "Введите значение для вывода: ";

            std::cin >> value;

            std::cout << tree.TGET(value) << std::endl;

            break;

        case '4':

            std::cout << "Введите название файла: ";

            std::cin >> filename;

            tree.loadFromFile(filename);

            break;

        case '5':

            std::cout << "Введите название файла: ";

            std::cin >> filename;

            tree.saveToFile(filename);

            break;

        case '6':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '6');

}

// Меню для работы с хеш-таблицей

static void hashTableMenu() {

    char choice;

    std::string key;

    std::string value;

    std::string filename;

    MyHashMap<std::string, std::string> hashTable;

    do {

        std::cout << "[H] Хеш-таблица\n1. HSET\n2. HDEL\n3. HGET\n4. LOADFROMFILE\n5. SAVETOFILE\n6. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите ключ: ";

            std::cin >> key;

            std::cout << "Введите значение: ";

            std::cin >> value;

            hashTable.HSET(key, value);

            break;

        case '2':

            std::cout << "Введите ключ для удаления: ";

            std::cin >> key;

            hashTable.HDEL(key);

            break;

        case '3':

            std::cout << "Введите ключ для получения значения: ";

            std::cin >> key;

            std::cout << hashTable.HGET(key) << std::endl;

            hashTable.print();

            break;

        case '4':

            std::cout << "Введите название для загрузки данных из файла: ";

            std::cin >> filename;

            hashTable.loadFromFile(filename);

            break;

        case '5':

            std::cout << "Введите название для выгрузки данных в файл: ";

            std::cin >> filename;

            hashTable.saveToFile(filename);

            break;

        case '6':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '6');

}

// Меню для работы со стеком

static void stackMenu() {

    char choice;

    std::string value;

    MyStack<std::string> stack;

    do {

        std::cout << "[S] Стек\n1. SPUSH\n2. SPOP\n3. SGET\n4. LOADFROMFILE\n5. SAVETOFILE\n6. Назад\nВаш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case '1':

            std::cout << "Введите значение для добавления: ";

            std::cin >> value;

            stack.SPUSH(value);

            break;

        case '2':

            stack.SDEL();

            break;

        case '3':

            stack.SPOP();

            break;

        case '4':

            std::cout << "Введите название для загрузки данных из файла: ";

            std::cin >> value;

            stack.loadFromFile(value);

            break;

        case '5':

            std::cout << "Введите название для выгрузки данных в файл: ";

            std::cin >> value;

            stack.saveToFile(value);

            break;

        case '6':

            return;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != '6');

}

// Главное меню программы

static void mainMenu() {

    char choice;

    do {

        std::cout << "=========================================\n";

        std::cout << "            МЕНЮ СТРУКТУР ДАННЫХ\n";

        std::cout << "=========================================\n";

        std::cout << "[M] Массив\n[X] ОдСписок\n[D] ДвСписок\n[Q] Очередь\n[S] Стек\n[H] Хеш-таблица\n[T] Дерево\n[E] Выход\n";

        std::cout << "Ваш выбор: ";

        std::cin >> choice;

        switch (choice) {

        case 'M':

            arrayMenu();

            break;

        case 'X':

            SinglelistMenu();

            break;

        case 'D':

            DoublylistMenu();

            break;

        case 'Q':

            queueMenu();

            break;

        case 'S':

            stackMenu();

            break;

        case 'H':

            hashTableMenu();

            break;

        case 'T':

            treeMenu();

            break;

        case 'E':

            std::cout << "Завершение работы...\n";

            break;

        default:

            std::cout << "Неверный выбор!\n";

        }

    } while (choice != 'E');

}

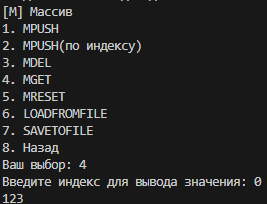
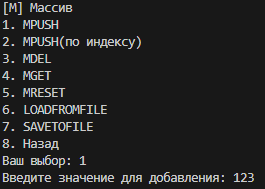
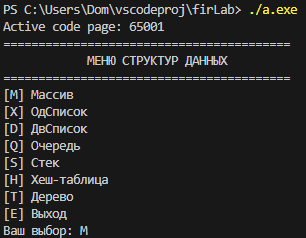
int main() {

    system("chcp 65001");

    mainMenu();

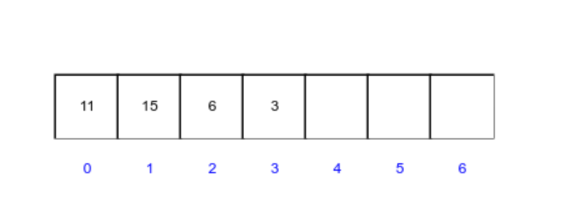
}

**Результат работы программы:**

****

**Визуализация**

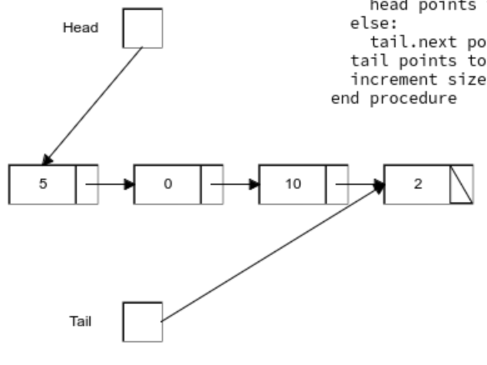
**Array:**



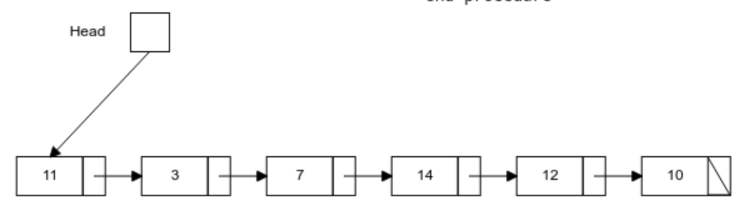
**Map:**



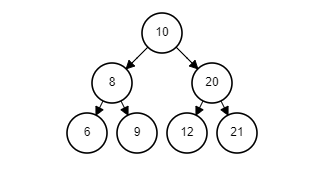
**Queue:**



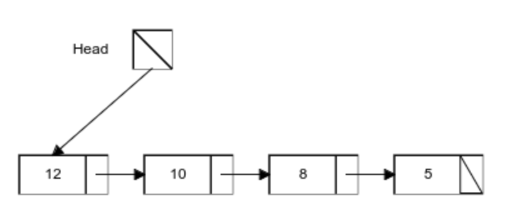
**Stack:**



**FullBinaryTree:**



**List:**



**Сложности операций по времени**

**Массив**

1. **MPUSH** (в конец): *O(1)* амортизировано, *O(n)* если массив полон (увеличение емкости).

2. **MPUSH** (по индексу): *O(n)*, *O(n)* если массив полон.

3. **MDEL**: *O(n)*.

4. **MGET**: *O(1)*.

5. **MRESET**: *O(1)*.

**Список Односвязный**

1. **LPUSHHEAD** (добавление в голову): *O(1)* — добавление нового узла в начало списка.

2. **LPUSHTAIL** (добавление в хвост): *O(1)* — добавление нового узла в конец списка, так как есть указатель на хвост.

3. **LDELHEAD** (удаление из головы): *O(1)* — удаление первого узла списка.

4. **LDELTAIL** (удаление из хвоста): *O(n)* — требуется пройти весь список для нахождения предпоследнего узла, чтобы обновить указатель на хвост.

5. **LDEL** (удаление по индексу): *O(n)* — требуется пройти до нужного индекса.

6. **LGET** (чтение по индексу): *O(n)* — нужно пройти список до нужного индекса.

7. **contains** (поиск элемента): *O(n)* — требуется пройти весь список в худшем случае для поиска значения.

**Список Двусвязный**

1. **LPUSHHEAD** (добавление в голову): *O(1)* — добавление нового узла в начало списка не требует сдвига элементов.

2. **LPUSHTAIL** (добавление в хвост): *O(1)* — добавление нового узла в конец списка также не требует сдвига.

3. **LDELHEAD** (удаление из головы): *O(1)* — удаление первого узла.

4. **LDELTAIL** (удаление из хвоста): *O(1)* — удаление последнего узла.

5. **LDEL** (удаление по индексу): *O(n)* — для удаления узла по индексу требуется пройти список до нужного элемента.

6. **LGET** (чтение по индексу): *O(n)* — нужно пройти список до нужного индекса для чтения данных.

7. **contains** (поиск элемента): *O(n)* — требуется пройти весь список в худшем случае для поиска значения.

**Очередь**

1. **Q\_PUSH**: В среднем *O(1)*, в худшем случае *O(n)* при увеличении массива.

2. **Q\_DEL**: *O(1)* — удаление первого элемента.

3. **Q\_POP**: *O(1)* — удаление и возврат первого элемента.

**Стек**

1. **SPUSH**: *O(1)* в среднем, *O(n)* при увеличении емкости.

2. **SDEL**: *O(1)* — удаление верхнего элемента.

3. **SPOP**: *O(1)* — чтение верхнего элемента.

**Хэш-таблица**

4. **HSET**: добавление ключ-значение в соответствующий список, используя хеш-функцию. *O(1)* в среднем для добавления.

5. **HGET**: поиск элемента по ключу в списке. *O(1)* в среднем, *O(n)* в худшем случае.

6. **HDEL**: удаление ключа. *O(1)* в среднем, *O(n)* в худшем случае.

**Full Binary дерево**

1. ***TINSERT*** (Вставка): *O(log n)*

2. ***TDEL*** (Удаление): *O(log n)*

3. ***TGET*** (Поиск): *O(n)*