МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ образовательное учреждениевысшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации



**ОТЧЁТ по Лабораторной работе №1**

**по дисциплине «ПРОГРАММИРОВАНИЕ»**

**«Абстрактные структуры данных»**

|  |  |
| --- | --- |
| Факультет: АВТФ  Группа: АБс-222  Студент: Гатауллин Д.Р. | Преподаватель: Архипова А.Б. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

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

**Задание к работе**:

1. Реализовать структуры данных с базовым набором операций:

a. Массив

b. Список (односвязный, двусвязный).

c. Очередь

d. Стек

e. Хеш таблицы

f. Деревья (2 Full Binary Tree)

2. Обосновать сложность выполнения каждой операции с позиции BigO нотации.

3. В отчете в контрольном примере представить визуальные формы,

подтверждающие соответствие результатам работы программы.

**Текст программы, результаты выполнения**

LB1.c

Основная функция программы, которая определяет интерфейс

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <Windows.h>

#include <locale.h>

#include "table.h"

#include "stack.h"

#include "queue.h"

#include "massive.h"

#include "onelist.h"

#include "twolist.h"

#include "bin.h"

int main() {

setlocale(LC\_ALL, "Rus");

SetConsoleCP(1251); // устанавливаем кодировку Windows-1251 для ввода

SetConsoleOutputCP(1251);

char command[256];

char key[256];

char element[256];

HashTable\* ht = initHashTable();

Stack\* st = initStack();

Queue\* qu = initQueue();

StringArray\* ar = initStringArray();

List\* list1 = initList();

ListT\* list2 = initListT();

NodeTr\* tree = NULL;

printf("Список команд:\nhelp - Вывод Список Команд\n0 - Выход\n1 - Добавить элемент с ключом в Хеш-таблицу\n2 - Удалить элемент по ключу из Хеш-таблицы\n3 - Проверить наличия элемента по Хеш - Таблице\n4 - Вывести Хеш-таблицу\n5 - Очистить Хеш-таблицу\n6 - Добавить элемент в Стек\n7 - Удалить элемент из Стека\n8 - Вывести Стек\n9 - Очистить Стек\n10 - Добавить элемент в Очередь\n11 - Удалить элемент из Очереди\n12 - Вывести Очередь\n13 - Очистить Очередь\n14 - Добавление элемента в Массив\n14.1 - Замена элемента по индексу\n14.2 - Добавление элемента в конец массива\n14.3 - Проверка наличия элемента\n15 - Удаление элемента из Массива\n16 - Просмотр элемента в Массиве\n17 - Сравнение элементов в Массиве\n18 - Вывод Массива\n19 - Очистка Массива\n20 - Добавить элемент в начало Односвязного списка\n21 - Удалить элемент в начале Односвязного списка\n22 - Проверка наличие элемента в Односвязном списке\n23 - Удаление элемента по значению в Односвязном списке\n24 - Вывод Односвязного списка\n25 - Очистка Односвязного списка\n26 - Добавление в конец Двусвязного списка\n27 - Добавление в начало Двусвязного списка\n28 - Удаление в начале Двусвязного списка\n29 - Удаление в конце Двусвязного списка\n30 - Проверка наличие по значению элемента в Двусвязном списке\n31 - Удаление по значению в Двусвязном списке\n32 - Вывод Двусвязного списка\n33 - Очистка Двусвязного списка\n34 - Добавление элемента в Бинарное дерево по направлению\n35 - Проверка наличия элемента в Бинарном дереве\n36 - Удаление узлов после элементов\n37 - Вывод Бинарного дерева\n38 - Очистка Бинарного дерева\n39 - Проверка на полное бинарное дерево\n");

while (1) {

printf("Введите команду: ");

scanf("%255s", command);

if (strcmp(command, "0") == 0) {

break;

}

if (strcmp(command, "help") == 0) {

printf("Список команд:\nhelp - Вывод Список Команд\n0 - Выход\n1 - Добавить элемент с ключом в Хеш-таблицу\n2 - Удалить элемент по ключу из Хеш-таблицы\n3 - Проверить наличия элемента по Хеш - Таблице\n4 - Вывести Хеш-таблицу\n5 - Очистить Хеш-таблицу\n6 - Добавить элемент в Стек\n7 - Удалить элемент из Стека\n8 - Вывести Стек\n9 - Очистить Стек\n10 - Добавить элемент в Очередь\n11 - Удалить элемент из Очереди\n12 - Вывести Очередь\n13 - Очистить Очередь\n14 - Добавление элемента в Массив\n14.1 - Замена элемента по индексу\n14.2 - Добавление элемента в конец массива\n14.3 - Проверка наличия элемента\n15 - Удаление элемента из Массива\n16 - Просмотр элемента в Массиве\n17 - Сравнение элементов в Массиве\n18 - Вывод Массива\n19 - Очистка Массива\n20 - Добавить элемент в начало Односвязного списка\n21 - Удалить элемент в начале Односвязного списка\n22 - Проверка наличие элемента в Односвязном списке\n23 - Удаление элемента по значению в Односвязном списке\n24 - Вывод Односвязного списка\n25 - Очистка Односвязного списка\n26 - Добавление в конец Двусвязного списка\n27 - Добавление в начало Двусвязного списка\n28 - Удаление в начале Двусвязного списка\n29 - Удаление в конце Двусвязного списка\n30 - Проверка наличие по значению элемента в Двусвязном списке\n31 - Удаление по значению в Двусвязном списке\n32 - Вывод Двусвязного списка\n33 - Очистка Двусвязного списка\n34 - Добавление элемента в Бинарное дерево по направлению\n35 - Проверка наличия элемента в Бинарном дереве\n36 - Удаление узлов после элементов\n37 - Вывод Бинарного дерева\n38 - Очистка Бинарного дерева\n39 - Проверка на полное бинарное дерево\n");

}

if (strcmp(command, "1") == 0) {

printf("Введите ключ: ");

scanf("%255s", key);

printf("Введите элемент: ");

scanf("%255s", element);

HSET(ht, key, element);

printf("Вы добавили элемент %s с ключом %s\n", element, key);

}

if (strcmp(command, "2") == 0) {

printf("Введите ключ: ");

scanf("%255s", key);

HDEL(ht, key);

printf("Вы удалили элемент с ключом %s\n", key);

}

if (strcmp(command, "3") == 0) {

printf("Введите ключ: ");

scanf("%255s", key);

if (HGET(ht, key) != NULL) printf("True\n");

else printf("False\n");

}

if (strcmp(command, "4") == 0) {

printHashTable(ht);

}

if (strcmp(command, "5") == 0) {

freeHashTable(ht);

ht = initHashTable();

printf("Таблица очищена.\n");

}

if (strcmp(command, "6") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

SPUSH(st, element);

printf("Вы добавили элемент %s\n", element);

}

if (strcmp(command, "7") == 0) {

printf("Вы удалили элемент %s\n", SPOP(st));

}

if (strcmp(command, "8") == 0) {

SPrint(st);

}

if (strcmp(command, "9") == 0) {

SClear(st);

Stack\* st = initStack();

printf("Стек очищен.\n");

}

if (strcmp(command, "10") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

QPUSH(qu, element);

printf("Вы добавили элемент %s\n", element);

}

if (strcmp(command, "11") == 0) {

printf("Вы удалили элемент %s\n", QPOP(qu));

}

if (strcmp(command, "12") == 0) {

printQueue(qu);

}

if (strcmp(command, "13") == 0) {

clearQueue(qu);

Queue\* qu = initQueue();

printf("Очередь очищена.\n");

}

if (strcmp(command, "14") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

printf("Введите индекс: ");

scanf("%255s", key);

int num = atoi(key);

addString(ar, num, element);

}

if (strcmp(command, "14.1") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

printf("Введите индекс: ");

scanf("%255s", key);

int num = atoi(key);

replaceString(ar, num, element);

}

if (strcmp(command, "14.2") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

appendString(ar, element);

}

if (strcmp(command, "14.3") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

if (containsString(ar, element) == 1) printf("True\n");

else printf("False\n");

}

if (strcmp(command, "15") == 0) {

printf("Введите индекс: ");

scanf("%255s", key);

int num = atoi(key);

removeString(ar, num);

}

if (strcmp(command, "16") == 0) {

printf("Введите индекс: ");

scanf("%255s", key);

int num = atoi(key);

printString(ar, num);

}

if (strcmp(command, "17") == 0) {

printf("Введите индекс 1: ");

scanf("%255s", key);

int num1 = atoi(key);

printf("Введите индекс 2: ");

scanf("%255s", key);

int num2 = atoi(key);

compareStrings(ar, num1, num2);

}

if (strcmp(command, "18") == 0) {

printStringArray(ar);

}

if (strcmp(command, "19") == 0) {

freeStringArray(ar);

StringArray\* ar = initStringArray();

printf("Массив очищен.\n");

}

if (strcmp(command, "20") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

prepend(&list1, element);

printf("Вы успешно добавили элемент в начало.\n");

}

if (strcmp(command, "21") == 0) {

removeFirst(&list1);

printf("Вы успешно удалили элемент в начале.\n");

}

if (strcmp(command, "22") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

if (searchByValue(&list1, element) != NULL) printf("True\n");

else printf("False\n");

}

if (strcmp(command, "23") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

removeByValue(&list1, element);

printf("Вы успешно удалили элемент\n");

}

if (strcmp(command, "24") == 0) {

printList(&list1);

}

if (strcmp(command, "25") == 0) {

clearList(&list1);

List\* list1 = initList();

printf("Односвязный список очищен.\n");

}

if (strcmp(command, "26") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

appendT(&list2, element);

printf("Вы успешно добавили элемент в конец.\n");

}

if (strcmp(command, "27") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

prependT(&list2, element);

printf("Вы успешно добавили элемент в начало.\n");

}

if (strcmp(command, "28") == 0) {

removeFirstT(&list2);

printf("Вы успешно удалили элемент в начале.\n");

}

if (strcmp(command, "29") == 0) {

removeLastT(&list2);

printf("Вы успешно удалили элемент в конце.\n");

}

if (strcmp(command, "30") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

if (searchByValueT(&list2, element) != NULL) printf("True\n");

else printf("False\n");

}

if (strcmp(command, "31") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

removeByValueT(&list2, element);

printf("Вы успешно удалили элемент\n");

}

if (strcmp(command, "32") == 0) {

printListT(&list2);

}

if (strcmp(command, "33") == 0) {

clearListT(&list2);

ListT\* list2 = initList();

printf("Двусвязный список очищен.\n");

}

if (strcmp(command, "34") == 0) {

printf("Введите 1 элемент: ");

scanf("%255s", element);

printf("Введите 2 элемент: ");

scanf("%255s", command);

printf("Введите положение (LCR): ");

scanf("%255s", key);

tree = insertNode(tree, element, command, key);

printf("Вы успешно добавили элемент.\n");

}

if (strcmp(command, "35") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

if (findNode(tree, element) != NULL) printf("True\n");

else printf("False\n");

}

if (strcmp(command, "36") == 0) {

printf("Введите элемент: ");

scanf("%255s", element);

tree = deleteNode(tree, element);

printf("Вы успешно удалили узлы после элемента.\n");

}

if (strcmp(command, "37") == 0) {

printTree(tree);

}

if (strcmp(command, "38") == 0) {

clearTree(&tree);

NodeTr\* tree = NULL;

printf("Бинарное дерево очищено.\n");

}

if (strcmp(command, "39") == 0) {

if (isFullBinaryTree(tree) != 0) printf("True\n");

else printf("False\n");

}

}

freeHashTable(ht);

SClear(st);

clearQueue(st);

freeStringArray(ar);

clearList(&list1);

clearListT(&list2);

clearTree(&tree);

return 0;

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, Шрифт, меню

Автоматически созданное описание}

stack.c/stack.h

Операции над стеком. Стек: все операции выполняются за O(1), так как мы просто добавляем элемент в следующий указатель, также и удаляем с возвратом предыдущего указателя.

#ifndef STACK\_H

#define STACK\_H

typedef struct NodeStack {

char\* element;

struct NodeStack\* next;

} NodeStack;

typedef struct Stack {

NodeStack\* top;

} Stack;

Stack\* initStack();

void SClear(Stack\* stack);

void SPrint(Stack\* stack);

void SPUSH(Stack\* stack, char\* element);

char\* SPOP(Stack\* stack);

#endif

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "stack.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Функция для инициализации стека

Stack\* initStack() {

Stack\* stack = (Stack\*)malloc(sizeof(Stack));

stack->top = NULL;

return stack;

}

// Функция для добавления элемента в стек

void SPUSH(Stack\* stack, char\* element) {

NodeStack\* newNode = (NodeStack\*)malloc(sizeof(NodeStack)); // Выделение памяти под новый узел

newNode->element = \_strdup(element); // Копирование элемента в узел

newNode->next = stack->top; // Установка следующего узла после нового узла

stack->top = newNode; // Установка нового узла в качестве вершины стека

}

// Функция для удаления элемента из стека

char\* SPOP(Stack\* stack) {

// Проверка пустоты стека

if (stack->top == NULL) {

printf("Стек пуст\n");

return NULL;

}

NodeStack\* poppedNode = stack->top; // Получение узла для удаления

stack->top = poppedNode->next; // Установка следующего узла в качестве вершины стека

char\* element = poppedNode->element; // Получение элемента из узла

free(poppedNode); // Освобождение памяти, занятой узлом

return element; // Возврат элемента

}

// Функция для очистки стека

void SClear(Stack\* stack) {

while (stack->top != NULL) {

NodeStack\* temp = stack->top;

stack->top = stack->top->next;

free(temp->element); // Освобождение памяти, занятой элементом

free(temp); // Освобождение памяти, занятой узлом

}

}

// Функция для вывода содержимого стека

void SPrint(Stack\* stack) {

NodeStack\* current = stack->top;

printf("Содержимое стека: ");

while (current != NULL) {

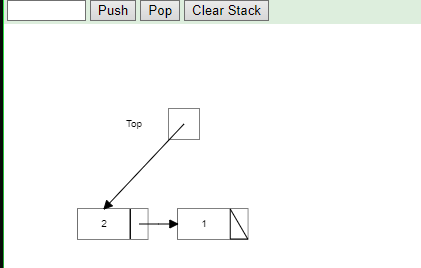
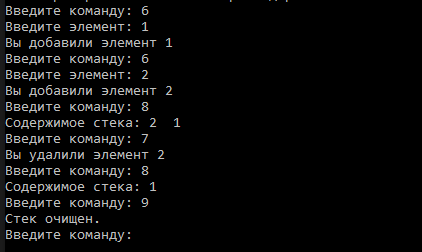
printf("%s ", current->element);

current = current->next;

}

printf("\n");

}

Изображение выглядит как снимок экрана, текст, диаграмма, часы

Автоматически созданное описание

queue.c/queue.h

Операции над очередью. То же самое, что и в стеке, т.е. все операции за O(1).

#ifndef QUEUE\_H

#define QUEUE\_H

typedef struct NodeQueue {

char\* element;

struct NodeQueue\* next;

} NodeQueue;

typedef struct Queue {

NodeQueue\* front;

NodeQueue\* rear;

} Queue;

Queue\* initQueue();

void clearQueue(Queue\* queue);

void printQueue(Queue\* queue);

void QPUSH(Queue\* queue, char\* element);

char\* QPOP(Queue\* queue);

#endif

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "queue.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Функция инициализации очереди

Queue\* initQueue() {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

queue->front = NULL; // Установка указателя на начало очереди в NULL

queue->rear = NULL; // Установка указателя на конец очереди в NULL

return queue; // Возврат указателя на очередь

}

// Функция добавления элемента в очередь

void QPUSH(Queue\* queue, char\* element) {

// Новый узел

NodeQueue\* newNode = (NodeQueue\*)malloc(sizeof(NodeQueue));

newNode->element = \_strdup(element); // Копирование элемента в узел

newNode->next = NULL; // Установка указателя на следующий узел в NULL

if (queue->front == NULL) { // Если очередь пуста

queue->front = newNode; // Установка указателя на начало очереди на новый узел

queue->rear = newNode; // Установка указателя на конец очереди на новый узел

}

else { // Если очередь не пуста

queue->rear->next = newNode; // Добавление нового узла в конец очереди

queue->rear = newNode; // Установка указателя на конец очереди на новый узел

}

}

// Функция удаления элемента из очереди

char\* QPOP(Queue\* queue) {

// Проверка пустоты очереди

if (queue->front == NULL) {

printf("Очередь пуста\n");

return NULL;

}

NodeQueue\* poppedNode = queue->front; // Указатель на удаляемый узел

char\* element = poppedNode->element; // Указатель на удаляемый элемент

queue->front = poppedNode->next; // Сдвиг указателя на начало очереди на следующий узел

if (queue->front == NULL) { // Если очередь стала пустой

queue->rear = NULL; // Установка указателя на конец очереди в NULL

}

free(poppedNode); // Освобождение памяти от удаляемого узла

return element; // Возврат указателя на удаляемый элемент

}

// Функция очистки очереди

void clearQueue(Queue\* queue) {

while (queue->front != NULL) { // Пока очередь не пуста

NodeQueue\* temp = queue->front; // Временный указатель на начало очереди

queue->front = queue->front->next; // Сдвиг начала очереди

free(temp->element); // Освобождение памяти от элемента узла

free(temp); // Освобождение памяти от узла

}

queue->rear = NULL; // Установка указателя на конец очереди в NULL после очистки

}

// Функция вывода содержимого очереди

void printQueue(Queue\* queue) {

NodeQueue\* current = queue->front; // Указатель на начало очереди

printf("Содержимое очереди: ");

while (current != NULL) { // Пока не достигнут конец очереди

printf("%s ", current->element); // Вывод элемента

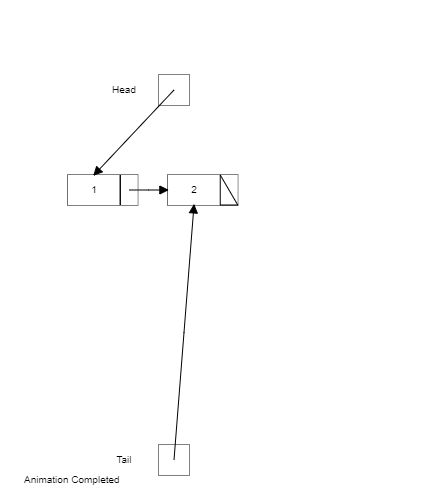
current = current->next; // Переход к следующему узлу

}

printf("\n");

}

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Автоматически созданное описаниеИзображение выглядит как диаграмма, линия, текст

Автоматически созданное описание

massive.c/massive.h

1. Операции над массивом. Массив: добавление элемента по индексу выполняется за O(N), так как мы сдвигаем все элементы, также и с удалением, получение значение, проверка по индексу, замена по индексу, добавление элемента в конец массива и сравнение двух значений выполняется за O(1), поскольку выполнятся эти функции за 1 операцию.

#ifndef MASSIVE\_H

#define MASSIVE\_H

typedef struct {

char\*\* array;

size\_t size;

} StringArray;

StringArray\* initStringArray();

void addString(StringArray\* sa, int index, const char\* value);

void appendString(StringArray\* sa, const char\* value);

int containsString(StringArray\* sa, const char\* value);

void replaceString(StringArray\* sa, int index, const char\* value);

void removeString(StringArray\* sa, int index);

void printString(StringArray\* sa, int index);

void compareStrings(StringArray\* sa, int index1, int index2);

void printStringArray(StringArray\* sa);

void freeStringArray(StringArray\* sa);

#endif

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100000

#define MAX\_STRING\_LENGTH 256

// Определение структуры для массива строк

typedef struct {

char\*\* array;

size\_t size;

} StringArray;

// Функция инициализации массива строк

StringArray\* initStringArray() {

StringArray\* sa = (StringArray\*)malloc(sizeof(StringArray));

sa->array = (char\*\*)malloc(MAX\_SIZE \* sizeof(char\*));

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

sa->array[i] = NULL;

}

sa->size = 0;

return sa;

}

// Функция добавления строки в массив

void addString(StringArray\* sa, int index, const char\* value) {

if (index < 0 || index > sa->size) {

printf("Индекс вне диапазона.\n");

return;

}

// Сдвигаем элементы, чтобы освободить место для новой строки

for (size\_t i = sa->size; i > index; i--) {

sa->array[i] = sa->array[i - 1];

}

// Выделяем память и копируем строку

sa->size++;

sa->array[index] = (char\*)malloc(MAX\_STRING\_LENGTH \* sizeof(char));

strncpy(sa->array[index], value, MAX\_STRING\_LENGTH);

printf("Вы добавили элемент %s на позицию %d\n", value, index);

}

// Функция удаления строки по индексу

void removeString(StringArray\* sa, int index) {

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

printf("Индекс вне диапазона.\n");

return;

}

printf("Вы удалили элемент %s на позицию %d\n", sa->array[index], index);

free(sa->array[index]);

for (size\_t i = index; i < sa->size - 1; i++) {

sa->array[i] = sa->array[i + 1];

}

sa->size--;

}

// Функция вывода строки по индексу

void printString(StringArray\* sa, int index) {

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

printf("Индекс вне диапазона.\n");

return;

}

printf("Элемент на индексе %d: %s\n", index, sa->array[index]);

}

// Функция сравнения строк по индексам

void compareStrings(StringArray\* sa, int index1, int index2) {

if (index1 < 0 || index1 >= sa->size || index2 < 0 || index2 >= sa->size) {

printf("Индекс вне диапазона.\n");

return;

}

if (strcmp(sa->array[index1], sa->array[index2]) == 0) {

printf("Элементы равны.\n");

}

else {

printf("Элементы не равны.\n");

}

}

// Функция добавления строки в конец массива

void appendString(StringArray\* sa, const char\* value) {

if (sa->size >= MAX\_SIZE) {

printf("Массив полон.\n");

return;

}

sa->array[sa->size] = (char\*)malloc(MAX\_STRING\_LENGTH \* sizeof(char));

strncpy(sa->array[sa->size], value, MAX\_STRING\_LENGTH);

printf("Вы добавили элемент %s в конец массива\n", value);

sa->size++;

}

// Функция проверки наличия строки в массиве

int containsString(StringArray\* sa, const char\* value) {

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

if (strcmp(sa->array[i], value) == 0) {

return 1;

}

}

return 0;

}

// Функция замены строки по индексу

void replaceString(StringArray\* sa, int index, const char\* value) {

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

printf("Индекс вне диапазона.\n");

return;

}

strncpy(sa->array[index], value, MAX\_STRING\_LENGTH);

printf("Вы заменили элемент на позиции %d на %s\n", index, value);

}

// Функция вывода массива строк

void printStringArray(StringArray\* sa) {

printf("Ваш массив: ");

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

printf("%s ", sa->array[i]);

}

printf("\n");

}

// Функция освобождения памяти массива строк

void freeStringArray(StringArray\* sa) {

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

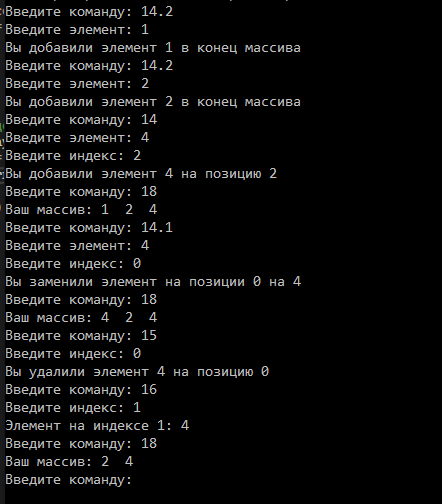
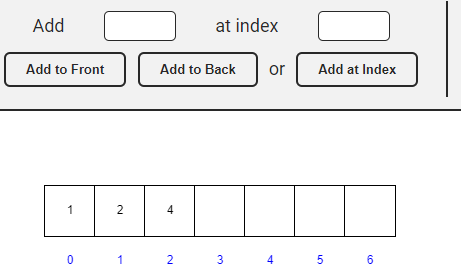
free(sa->array[i]);

}

free(sa->array);

free(sa);

}

  Изображение выглядит как снимок экрана, линия, диаграмма, текст

Автоматически созданное описание

table.c/table.h

Операции над хеш-таблицей. Хеш-таблица: все операции выполняются за O(1), так как мы работаем фактически с индексом (хеширование)

#ifndef TABLE\_H

#define TABLE\_H

typedef struct NodeHashTable {

char\* key;

char\* element;

struct NodeHashTable\* next;

struct NodeHashTable\* prev;

} NodeHashTable;

typedef struct HashTable {

NodeHashTable\*\* nodes;

int count;

} HashTable;

HashTable\* initHashTable();

void printHashTable(HashTable\* ht);

void freeHashTable(HashTable\* ht);

void HSET(HashTable\* hashtable, char\* key, char\* value);

void HDEL(HashTable\* hashtable, char\* key);

char\* HGET(HashTable\* hashtable, char\* key);

#endif

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "table.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100000

// Функция для инициализации хеш-таблицы

HashTable\* initHashTable() {

HashTable\* ht = (HashTable\*)malloc(sizeof(HashTable));

ht->nodes = (NodeHashTable\*\*)malloc(MAX\_SIZE \* sizeof(NodeHashTable\*));

ht->count = 0;

for (int i = 0; i < MAX\_SIZE; i++) {

ht->nodes[i] = NULL;

}

return ht;

}

// Функция для вычисления хеша

int calculateHashT(const char\* element) {

int hash = 0;

for (int i = 0; element[i] != '\0'; i++) {

hash = 31 \* hash + element[i];

}

return abs(hash) % MAX\_SIZE;

}

// Функция для добавления элемента в хеш-таблицу

void HSET(HashTable\* ht, char\* key, char\* value) {

int hash = calculateHashT(key);

// Создаем новый узел для хранения ключа и значения

NodeHashTable\* newNode = (NodeHashTable\*)malloc(sizeof(NodeHashTable));

newNode->key = \_strdup(key);

newNode->element = \_strdup(value);

newNode->next = NULL; // Устанавливаем указатель на следующий узел как NULL

newNode->prev = NULL; // Устанавливаем указатель на предыдущий узел как NULL

// Обработка коллизий и проверка на дубликаты ключей

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

while (current != NULL) {

if (strcmp(current->key, key) == 0) { // Если ключ уже существует

// Освобождаем память нового узла

free(newNode->key);

free(newNode->element);

free(newNode);

printf("Ключ уже существует.\n");

return;

}

if (current->next == NULL) { // Если достигли конца цепочки

break;

}

current = current->next; // Переходим к следующему узлу

}

// Добавление нового узла

if (current == NULL) { // Если цепочка пуста

ht->nodes[hash] = newNode; // Устанавливаем новый узел как начало цепочки

}

else {

current->next = newNode; // Добавляем новый узел в конец цепочки

newNode->prev = current; // Устанавливаем предыдущий узел для нового узла

}

ht->count++;

}

// Функция для получения элемента из хеш-таблицы

char\* HGET(HashTable\* ht, const char\* key) {

int hash = calculateHashT(key);

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

while (current != NULL) { // Перебираем узлы в цепочке

if (strcmp(current->key, key) == 0) { // Если ключ совпадает

return current->element;

}

current = current->next; // Переходим к следующему узлу

}

return NULL;

}

// Функция для удаления элемента из хеш-таблицы

void HDEL(HashTable\* ht, const char\* key) {

int hash = calculateHashT(key);

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

NodeHashTable\* nodeToRemove = NULL;

while (current != NULL) { // Перебираем узлы в цепочке

if (strcmp(current->key, key) == 0) { // Если ключ совпадает

nodeToRemove = current; // Устанавливаем узел для удаления

break;

}

current = current->next; // Переходим к следующему узлу

}

if (nodeToRemove != NULL) {

if (nodeToRemove->prev != NULL) { // Если у узла есть предыдущий узел

nodeToRemove->prev->next = nodeToRemove->next; // Удаляем узел из цепочки

}

else {

ht->nodes[hash] = nodeToRemove->next; // Устанавливаем следующий узел как начало цепочки

}

if (nodeToRemove->next != NULL) { // Если у узла есть следующий узел

nodeToRemove->next->prev = nodeToRemove->prev; // Устанавливаем предыдущий узел для следующего узла

}

// Освобождаем память удаляемого узла

free(nodeToRemove->key);

free(nodeToRemove->element);

free(nodeToRemove);

ht->count--;

}

else {

printf("Ключ не найден.\n");

}

}

// Функция для освобождения памяти хеш-таблицы

void freeHashTable(HashTable\* ht) {

for (int i = 0; i < MAX\_SIZE; i++) {

NodeHashTable\* current = ht->nodes[i];

while (current != NULL) {

NodeHashTable\* temp = current;

current = current->next;

free(temp->key);

free(temp->element);

free(temp);

}

}

free(ht->nodes);

free(ht);

}

// Функция для вывода хеш-таблицы

void printHashTable(HashTable\* ht) {

printf("Хеш-таблица:\n");

for (int i = 0; i < MAX\_SIZE; i++) {

NodeHashTable\* current = ht->nodes[i];

if (current != NULL) {

printf("Индекс %d: ", i);

while (current != NULL) {

printf("(%s, %s) ", current->key, current->element);

current = current->next;

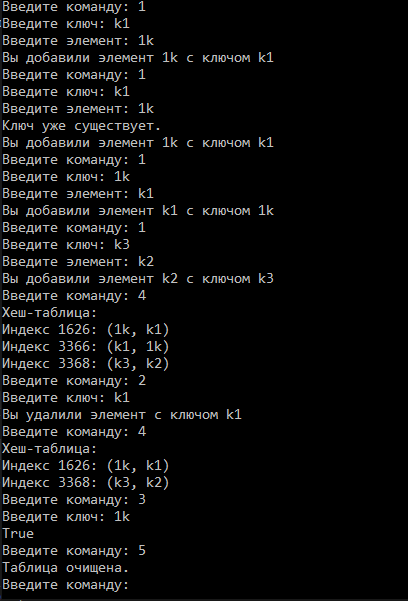
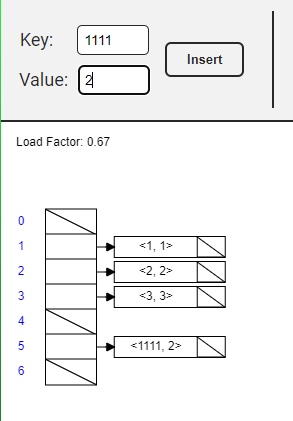
}

printf("\n");

}

}

}

  Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

onelist.c/onelist.h

Операции над односвязным списком. Односвязный список: операции добавления и удаление в начале списка выполняются за O(1), так как мы сохраняем указатель на голову списка и просто с ней работаем, а вот удаление и поиск по значению выполняются за O(N), поскольку мы проходимся по всем элементам до нужного.

#ifndef ONELIST\_H

#define ONELIST\_H

typedef struct Node {

char\* data;

struct Node\* next;

} NodeL;

typedef struct List {

NodeL\* head;

} List;

List\* initList();

void prepend(List\*\* list, char\* data);

void removeFirst(List\*\* list);

void printList(List\*\* list);

void clearList(List\*\* list);

void removeByValue(List\*\* list, char\* data);

NodeL\* searchByValue(List\*\* list, char\* data);

#endif

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "onelist.h"

// Инициализация списка

List\* initList() {

List\* list = (List\*)malloc(sizeof(List)); // Выделение памяти под список

list->head = NULL; // Установка головы списка в NULL

return list; // Возвращение указателя на список

}

// Добавление элемента в начало списка

void prepend(List\*\* list, char\* data) {

NodeL\* node = (NodeL\*)malloc(sizeof(NodeL)); // Выделение памяти под узел

node->data = \_strdup(data); // Копирование данных

node->next = (\*list)->head; // Установка указателя на следующий элемент

(\*list)->head = node; // Устанавливаем голову списка как текущий узел

}

// Удаление первого элемента списка

void removeFirst(List\*\* list) {

if (!(\*list)->head) {

return; // Если голова списка не существует, выходим из функции

}

NodeL\* temp = (\*list)->head; // Сохраняем указатель на голову списка

(\*list)->head = (\*list)->head->next; // Устанавливаем голову списка на следующий элемент

free(temp->data); // Освобождаем память от данных

free(temp); // Освобождаем память от узла

}

// Вывод списка

void printList(List\*\* list) {

NodeL\* temp = (\*list)->head; // Начинаем с головы списка

while (temp) {

printf("%s ", temp->data); // Выводим данные

temp = temp->next; // Переходим к следующему элементу

}

printf("\n"); // Переходим на новую строку после вывода всех элементов

}

// Очистка списка

void clearList(List\*\* list) {

while ((\*list)->head) {

removeFirst(list); // Удаляем первый элемент, пока голова списка существует

}

}

// Удаление элемента списка по значению

void removeByValue(List\*\* list, char\* data) {

NodeL\* temp = (\*list)->head; // Начинаем с головы списка

NodeL\* prev = NULL; // Инициализируем указатель на предыдущий элемент

while (temp) {

if (strcmp(temp->data, data) == 0) { // Если данные совпадают

if (prev) {

prev->next = temp->next; // Устанавливаем следующий элемент для предыдущего элемента

}

else {

(\*list)->head = temp->next; // Иначе устанавливаем голову списка на следующий элемент

}

free(temp->data); // Освобождаем память от данных

free(temp); // Освобождаем память от узла

return; // Выходим из функции

}

prev = temp; // Сохраняем текущий узел как предыдущий

temp = temp->next; // Переходим к следующему элементу

}

}

// Поиск элемента списка по значению

NodeL\* searchByValue(List\*\* list, char\* data) {

NodeL\* temp = (\*list)->head; // Начинаем с головы списка

while (temp) {

if (strcmp(temp->data, data) == 0) { // Если данные совпадают

return temp; // Возвращаем указатель на узел

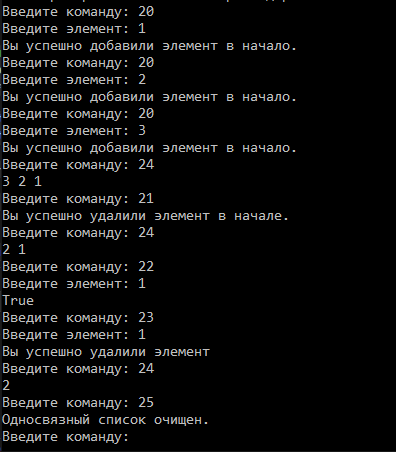
}

temp = temp->next; // Переходим к следующему элементу

}

return NULL; // Если элемент не найден, возвращаем NULL

}



twolist.c/twolist.h

Операции над двусвязным списком. Двусвязный список: добавление и удаление в начало и в конец списка выполняются за O(1), так как я запоминаю указатели на хвост и голову и непосредственно работаю с ними. А вот удаление и поиск по значению выполняются за O(N), поскольку мы проходимся по всем элементам до нужного.

#ifndef TWOLIST\_H

#define TWOLIST\_H

typedef struct NodeT {

char\* data;

struct NodeT\* prev;

struct NodeT\* next;

} NodeL2;

typedef struct ListT {

NodeL2\* head;

NodeL2\* tail;

} ListT;

ListT\* initListT();

void prependT(ListT\*\* list, char\* data);

void appendT(ListT\*\* list, char\* data);

void removeFirstT(ListT\*\* list);

void removeLastT(ListT\*\* list);

void printListT(ListT\*\* list);

void clearListT(ListT\*\* list);

void removeByValueT(ListT\*\* list, char\* data);

NodeL2\* searchByValueT(ListT\*\* list, char\* data);

#endif

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "twolist.h"

// Инициализация списка

ListT\* initListT() {

ListT\* list = (ListT\*)malloc(sizeof(ListT)); // Выделение памяти под список

list->head = NULL; // Установка головы списка в NULL

list->tail = NULL; // Установка хвоста списка в NULL

return list; // Возвращение указателя на список

}

// Добавление элемента в начало списка

void prependT(ListT\*\* list, char\* data) {

NodeL2\* node = (NodeL2\*)malloc(sizeof(NodeL2)); // Выделение памяти под узел

node->data = \_strdup(data); // Копирование данных

node->next = (\*list)->head; // Установка указателя на следующий элемент

node->prev = NULL; // Установка указателя на предыдущий элемент

if ((\*list)->head) {

(\*list)->head->prev = node; // Если голова списка существует, устанавливаем ее предыдущий элемент как текущий узел

}

else {

(\*list)->tail = node; // Иначе устанавливаем хвост списка как текущий узел

}

(\*list)->head = node; // Устанавливаем голову списка как текущий узел

}

// Добавление элемента в конец списка

void appendT(ListT\*\* list, char\* data) {

NodeL2\* node = (NodeL2\*)malloc(sizeof(NodeL2)); // Выделение памяти под узел

node->data = \_strdup(data); // Копирование данных

node->prev = (\*list)->tail; // Установка указателя на предыдущий элемент

node->next = NULL; // Установка указателя на следующий элемент

if ((\*list)->tail) {

(\*list)->tail->next = node; // Если хвост списка существует, устанавливаем его следующий элемент как текущий узел

}

else {

(\*list)->head = node; // Иначе устанавливаем голову списка как текущий узел

}

(\*list)->tail = node; // Устанавливаем хвост списка как текущий узел

}

// Удаление первого элемента списка

void removeFirstT(ListT\*\* list) {

if (!(\*list)->head) {

return; // Если голова списка не существует, выходим из функции

}

NodeL2\* temp = (\*list)->head; // Сохраняем указатель на голову списка

(\*list)->head = (\*list)->head->next; // Устанавливаем голову списка на следующий элемент

if ((\*list)->head) {

(\*list)->head->prev = NULL; // Если голова списка существует, устанавливаем ее предыдущий элемент в NULL

}

else {

(\*list)->tail = NULL; // Иначе устанавливаем хвост списка в NULL

}

free(temp->data); // Освобождаем память от данных

free(temp); // Освобождаем память от узла

}

// Удаление последнего элемента списка

void removeLastT(ListT\*\* list) {

if (!(\*list)->tail) {

return; // Если хвост списка не существует, выходим из функции

}

NodeL2\* temp = (\*list)->tail; // Сохраняем указатель на хвост списка

(\*list)->tail = (\*list)->tail->prev; // Устанавливаем хвост списка на предыдущий элемент

if ((\*list)->tail) {

(\*list)->tail->next = NULL; // Если хвост списка существует, устанавливаем его следующий элемент в NULL

}

else {

(\*list)->head = NULL; // Иначе устанавливаем голову списка в NULL

}

free(temp->data); // Освобождаем память от данных

free(temp); // Освобождаем память от узла

}

// Удаление элемента списка по значению

void removeByValueT(ListT\*\* list, char\* data) {

NodeL2\* temp = (\*list)->head; // Начинаем с головы списка

while (temp) {

if (strcmp(temp->data, data) == 0) { // Если данные совпадают

if (temp->prev) {

temp->prev->next = temp->next; // Устанавливаем следующий элемент для предыдущего элемента

}

else {

(\*list)->head = temp->next; // Иначе устанавливаем голову списка на следующий элемент

}

if (temp->next) {

temp->next->prev = temp->prev; // Устанавливаем предыдущий элемент для следующего элемента

}

else {

(\*list)->tail = temp->prev; // Иначе устанавливаем хвост списка на предыдущий элемент

}

free(temp->data); // Освобождаем память от данных

free(temp); // Освобождаем память от узла

return; // Выходим из функции

}

temp = temp->next; // Переходим к следующему элементу

}

}

// Поиск элемента списка по значению

NodeL2\* searchByValueT(ListT\*\* list, char\* data) {

NodeL2\* temp = (\*list)->head; // Начинаем с головы списка

while (temp) {

if (strcmp(temp->data, data) == 0) { // Если данные совпадают

return temp; // Возвращаем указатель на узел

}

temp = temp->next; // Переходим к следующему элементу

}

return NULL; // Если элемент не найден, возвращаем NULL

}

// Вывод списка

void printListT(ListT\*\* list) {

NodeL2\* temp = (\*list)->head; // Начинаем с головы списка

while (temp) {

printf("%s ", temp->data); // Выводим данные

temp = temp->next; // Переходим к следующему элементу

}

printf("\n"); // Переходим на новую строку после вывода всех элементов

}

// Очистка списка

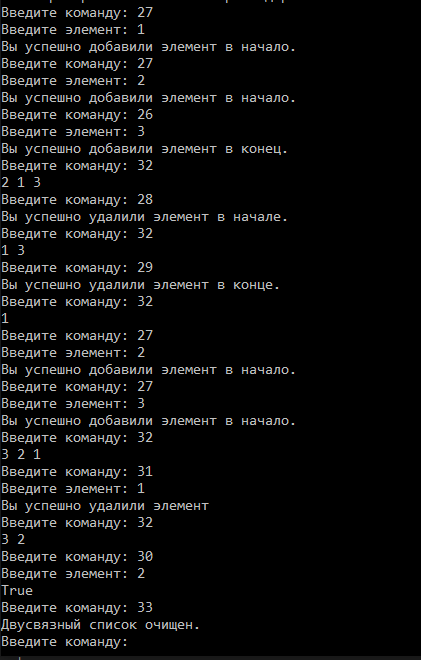
void clearListT(ListT\*\* list) {

while ((\*list)->head) {

removeFirstT(list); // Удаляем первый элемент, пока голова списка существует

}

}



bin.c/bin.h

Операции над Full Binary Tree. Операция вставки выполняется за O(log(N)), так как мы вставляем нужное нам место, обходя все ненужные элементы стороной. Операция проверки на FullBinaryTree выполняется в худшем случае за O(N), так как нам нужно проверить все узлы. Операция проверки наличия элемента в моём случае выполняется за O(N), поскольку я обхожу в худшем случае все элементы. Удаление узлов под элементами также выполняется за O(N).

#ifndef BIN\_H

#define BIN\_H

typedef struct NodeTr {

char\* value;

struct NodeTr\* left;

struct NodeTr\* right;

} NodeTr;

NodeTr\* insertNode(NodeTr\* root, const char\* value1, const char\* value2, const char\* path);

NodeTr\* findNode(NodeTr\* root, const char\* value);

NodeTr\* deleteNode(NodeTr\* root, const char\* value);

int isFullBinaryTree(NodeTr\* root);

void printTree(NodeTr\* root);

void clearTree(NodeTr\*\* rootPtr);

#endif

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "bin.h"

// Функция для проверки, является ли дерево полным бинарным деревом

int isFullBinaryTree(NodeTr\* root) {

// Если дерево пустое, оно является полным бинарным деревом

if (root == NULL) return 1;

// Если у узла нет детей, это полное бинарное дерево

if (root->left == NULL && root->right == NULL) return 1;

// Если у узла есть оба ребенка, проверяем, являются ли поддеревья полными бинарными деревьями

if (root->left != NULL && root->right != NULL) {

return isFullBinaryTree(root->left) && isFullBinaryTree(root->right);

}

// В противном случае это не полное бинарное дерево

return 0;

}

// Создание нового узла дерева

NodeTr\* createNodeTr(const char\* value) {

NodeTr\* newNode = (NodeTr\*)malloc(sizeof(NodeTr));

newNode->value = \_strdup(value);

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Вспомогательная функция для печати дерева

void printTreeHelper(NodeTr\* root, int space) {

int i;

if (root == NULL) return;

space += 10; // Увеличиваем расстояние между уровнями

printTreeHelper(root->right, space); // Сначала печатаем правое поддерево

printf("\n");

for (i = 10; i < space; i++) {

printf(" "); // Печатаем пробелы перед узлом

}

printf("%s\n", root->value); // Печатаем текущий узел

printTreeHelper(root->left, space); // Затем печатаем левое поддерево

}

void printTree(NodeTr\* root) {

printTreeHelper(root, 0); // Начальный вызов с отступом 0

}

// Очистка дерева

void clearTree(NodeTr\*\* rootPtr) {

if (\*rootPtr == NULL) return;

clearTree(&((\*rootPtr)->left));

clearTree(&((\*rootPtr)->right));

free((\*rootPtr)->value);

(\*rootPtr)->value = NULL;

free(\*rootPtr);

\*rootPtr = NULL;

}

// Вставка узла в дерево

NodeTr\* insertNode(NodeTr\* root, const char\* value1, const char\* value2, char\* path) {

if (path[0] == 'C') {

root = createNodeTr(value1);

return root;

}

if (root == NULL) {

root = createNodeTr("null");

}

if (path[0] != '\0' && path[1] == 'L' && root->right == NULL) root->right = createNodeTr("null");

if (path[0] != '\0' && path[1] == 'R' && root->left == NULL) root->left = createNodeTr("null");

if (path[1] == '\0') {

if (path[0] == 'L') {

if (root->left == NULL) {

root->left = createNodeTr(value1);

}

else {

free(root->left->value);

root->left->value = \_strdup(value1);

}

if (root->right == NULL) {

root->right = createNodeTr(value2);

}

else {

free(root->right->value);

root->right->value = \_strdup(value2);

}

}

else if (path[0] == 'R') {

if (root->right == NULL) {

root->right = createNodeTr(value2);

}

else {

free(root->right->value);

root->right->value = \_strdup(value2);

}

if (root->left == NULL) {

root->left = createNodeTr(value1);

}

else {

free(root->left->value);

root->left->value = \_strdup(value1);

}

}

}

if (path[0] == 'L') {

root->left = insertNode(root->left, value1, value2, path + 1);

}

else if (path[0] == 'R') {

root->right = insertNode(root->right, value1, value2, path + 1);

}

return root;

}

// Поиск узла в дереве

NodeTr\* findNode(NodeTr\* root, const char\* value) {

if (root == NULL) return NULL;

if (strcmp(root->value, value) == 0) return root;

NodeTr\* foundNode = findNode(root->left, value);

if (foundNode == NULL) foundNode = findNode(root->right, value);

return foundNode;

}

// Освобождение памяти узла дерева

void freeTree(NodeTr\* node) {

if (node == NULL) return;

freeTree(node->left);

freeTree(node->right);

free(node->value);

free(node);

}

// Удаление детей узла из дерева

NodeTr\* deleteNode(NodeTr\* root, const char\* value) {

if (root == NULL) return NULL;

if (strcmp(root->value, value) == 0) {

freeTree(root->left);

freeTree(root->right);

root->left = NULL;

root->right = NULL;

return root;

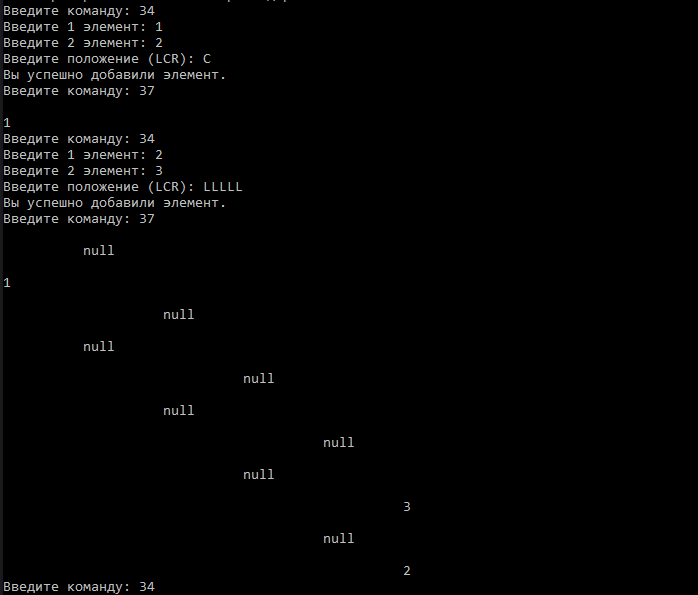
}

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

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

return root;

}

 Изображение выглядит как текст, снимок экрана

Автоматически созданное описание Изображение выглядит как текст, снимок экрана

Автоматически созданное описание Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

**Вывод**: Был создан интерфейс, где пользователь, по его желанию, вводить различные команды, связанные со структурами данных различного рода. Все временные сложности операций расписаны. Продемонстрированы работы каждой операции.