МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ образовательное учреждениевысшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации



**ОТЧЁТ по Лабораторной работе №2**

**по дисциплине «ПРОГРАММИРОВАНИЕ»**

**«Алгоритмы и абстрактные структуры данных»**

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

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

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

**Задание к работе**:

1. Обратная польская запись.

Постфиксная (или обратная польская) запись арифметического выражения — это способ записи выражений, в котором знак операции записывается после операндов. Так, операторы больше не являются неоднозначными по отношению к своим операндам. Например, запишем выражение 3+1\*4 в постфиксную форму и произведем вычисление. В постфиксной форме получим выражение 314\*+. Мы умножаем числа 4 и 1, и результат складываем с 3. Необходимо реализовать алгоритм, который вычисляет выражения в обратной польской записи с использованием стека.

1. Схожие подмножества.

Необходимо реализовать алгоритм, который должен разбить множество натуральных чисел на непересекающиеся подмножества, разница между суммами которых была бы минимальна. Вывести получившиеся подмножества и разницу их сумм.

Пример:

Множество S = {5, 8, 1, 14, 7}. Получим два подмножества {5, 14} и {8, 1, 7}. Разница между их суммами - 3.

1. Сопоставление с паттерном.

Необходимо реализовать алгоритм, который сравнивает последовательность символов с шаблоном и выводит сообщение о соответствии. Учтите, что символ "?" может заменять один символ, а "\*" - любую последовательность, в том числе нулевую.

Пример:

Последовательность "meow@stud.nstu.ru" соответствует шаблону "\*@stud.nstu.ru". Последовательность "hello" не соответствует шаблону "h?lo".

1. Удаление узла.

Дано бинарное дерево поиска, в котором хранятся уникальные целые числа. Найдите вершину с заданным значением и удалите её из дерева так, чтобы дерево осталось корректным бинарным деревом поиска.

1. Поиск правого соседа BST.

Задано двоичное дерево. Необходимо для каждого узла вывести узел, который находится "справа" от него, используя очередь. Если соседа нет, вывести null.

Пример:

22 – null, 16 – 51, 51 – null, 7 – 19, 19 – 43, 43 – 57, 57 – null

1. Метод цепочек

Реализовать хеш-таблицу, в которой коллизии будут обрабатываться при помощи метода цепочек.

**Текст программы**

LB2.1.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAXOP 100 // Макс. размер операнда или оператора

#define NUMBER '0' // Сигнал, что обнаружено число

int getop(char[]);

void push(double);

double pop(void);

// Обратная польская запись

int main() {

int type;

double op2;

char s[MAXOP];

while ((type = getop(s)) != EOF) {

switch (type) {

case NUMBER:

push(atof(s));

break;

case '+':

push(pop() + pop());

break;

case '\*':

push(pop() \* pop());

break;

case '-':

op2 = pop();

push(pop() - op2);

break;

case '/':

op2 = pop();

if (op2 != 0.0)

push(pop() / op2);

else

printf("error: zero divisor\n");

break;

case '\n':

printf("%.8g\n", pop());

if (pop() != 0.0 || pop() != 0.0) printf("Problems\n");

else printf("Its OK\n");

break;

default:

printf("error: unknown command %s\n", s);

break;

}

}

return 0;

}

#define MAXVAL 100 // Максимальная глубина стека

int sp = 0; // Следующая свободная позиция в стеке

double val[MAXVAL]; // Стек

// push: положить значение f в стек

void push(double f) {

if (sp < MAXVAL)

val[sp++] = f;

else

printf("error: stack full, can't push %g\n", f);

}

// pop: взять значение с вершины стека

double pop(void) {

if (sp > 0)

return val[--sp];

else {

printf("error: stack empty\n");

return 0.0;

}

}

// getop: получить следующий оператор или операнд

int getop(char s[]) {

int i, c;

while ((s[0] = c = getchar()) == ' ' || c == '\t')

;

s[1] = '\0';

if (!isdigit(c) && c != '.')

return c; // Не число

i = 0;

if (isdigit(c)) // Накапливаем целую часть

while (isdigit(s[++i] = c = getchar()))

;

if (c == '.') // Накапливаем дробную часть

while (isdigit(s[++i] = c = getchar()))

;

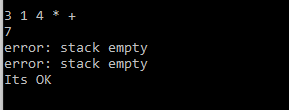
s[i] = '\0';

if (c != EOF)

ungetc(c, stdin);

return NUMBER;

}



LB2.2.c

#include <stdio.h>

#include <locale.h>

#include <limits.h>

// Функция для нахождения разницы между двумя числами

int findDifference(int a, int b) {

return a > b ? a - b : b - a;

}

// Глобальные переменные для хранения подмножеств с минимальной разницей

int minDifference = INT\_MAX;

int minSubset1[100], minSubset2[100];

int minSubsetSize1 = 0, minSubsetSize2 = 0;

// Функция для вывода подмножеств и их разницы

void printSubsets(int set[], int subsets[], int n, int subsetSize) {

int i, j;

int sumSubset1 = 0, sumSubset2 = 0;

// Вычисляем сумму каждого подмножества и находим минимальную разницу

for (i = 0; i < n; i++) {

if (subsets[i] == 1) {

sumSubset1 += set[i];

}

else {

sumSubset2 += set[i];

}

}

int difference = findDifference(sumSubset1, sumSubset2);

// Если разница меньше минимальной, сохраняем подмножества

if (difference < minDifference) {

minDifference = difference;

minSubsetSize1 = 0;

minSubsetSize2 = 0;

for (i = 0; i < n; i++) {

if (subsets[i] == 1) {

minSubset1[minSubsetSize1++] = set[i];

}

else {

minSubset2[minSubsetSize2++] = set[i];

}

}

}

}

// Функция для разбиения множества на непересекающиеся подмножества

void divideIntoSubsets(int set[], int subsets[], int subsetSize, int n, int current, int count) {

// Если достигли конца подмножества

if (current == subsetSize) {

printSubsets(set, subsets, n, subsetSize);

return;

}

// Если все элементы рассмотрены

if (count == n) {

return;

}

// Включаем текущий элемент в подмножество

subsets[count] = 1;

divideIntoSubsets(set, subsets, subsetSize, n, current + 1, count + 1);

// Исключаем текущий элемент из подмножества

subsets[count] = 0;

divideIntoSubsets(set, subsets, subsetSize, n, current, count + 1);

}

int main() {

setlocale(LC\_ALL, "Rus");

int set[] = { 5, 8, 1, 14, 7 };

const int n = sizeof(set) / sizeof(set[0]);

int subsets[n];

// Инициализируем все элементы подмножеств нулями

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

subsets[i] = 0;

}

// Размер каждого подмножества

int subsetSize = n / 2;

divideIntoSubsets(set, subsets, subsetSize, n, 0, 0);

// Выводим подмножества с минимальной разницей

printf("Первое подмножество: ");

for (int i = 0; i < minSubsetSize1; i++) {

printf("%d ", minSubset1[i]);

}

printf("\n");

printf("Второе подмножество: ");

for (int i = 0; i < minSubsetSize2; i++) {

printf("%d ", minSubset2[i]);

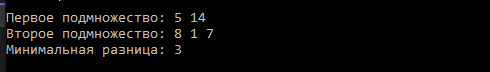
}

printf("\n");

printf("Минимальная разница: %d\n", minDifference);

return 0;

}



LB2.3.с

#include <stdio.h>

#include <stdbool.h>

#include <locale.h>

// Функция match проверяет, соответствует ли строка sequence шаблону pattern

bool match(char\* sequence, char\* pattern) {

// Если шаблон пуст, возвращаем true, если строка тоже пуста

if (\*pattern == '\0') return \*sequence == '\0';

// Если в шаблоне встречается '\*', проверяем оставшуюся часть строки

if (\*pattern == '\*') {

pattern++; // Пропускаем символ '\*'

if (\*pattern == '\0') return true; // Если после '\*' ничего нет, возвращаем true

// Продолжаем проверку до конца строки или пока не найдем соответствие

while (\*sequence != '\0' && !match(sequence, pattern)) {

sequence++;

}

return \*sequence != '\0';

}

// Если в шаблоне '?', или символы совпадают, проверяем следующие символы

else if (\*pattern == '?' || \*pattern == \*sequence) {

return match(sequence + 1, pattern + 1);

}

// Если ни одно условие не выполнено, возвращаем false

return false;

}

int main() {

setlocale(LC\_ALL, "Rus"); // Установка русской локализации

// Проверка соответствия строки шаблону с использованием функции match

char sequence[] = "meow@stud.nstu.ru";

char pattern[] = "\*@stud.nstu.ru";

if (match(sequence, pattern)) {

printf("Последовательность \"%s\" соответствует шаблону \"%s\".\n", sequence, pattern);

}

else {

printf("Последовательность \"%s\" не соответствует шаблону \"%s\".\n", sequence, pattern);

}

// Проверка другой строки и шаблона

char sequence2[] = "hello";

char pattern2[] = "h?lo";

if (match(sequence2, pattern2)) {

printf("Последовательность \"%s\" соответствует шаблону \"%s\".\n", sequence2, pattern2);

}

else {

printf("Последовательность \"%s\" не соответствует шаблону \"%s\".\n", sequence2, pattern2);

}

return 0;

}



LB2.4.c

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

// Определение структуры для узла дерева

typedef struct TreeNode {

int value; // Значение в узле

struct TreeNode\* left; // Указатель на левое поддерево

struct TreeNode\* right; // Указатель на правое поддерево

} TreeNode;

// Функция для поиска узла с минимальным значением в дереве

TreeNode\* findMin(TreeNode\* node) {

while (node->left != NULL) node = node->left; // Перемещение в самый левый узел

return node; // Возврат узла с минимальным значением

}

// Функция для удаления узла из дерева

TreeNode\* deleteNode(TreeNode\* root, int key) {

if (root == NULL) return root; // Если дерево пустое, возвращаем NULL

// Поиск узла для удаления

if (key < root->value)

root->left = deleteNode(root->left, key); // Переход к левому поддереву

else if (key > root->value)

root->right = deleteNode(root->right, key); // Переход к правому поддереву

else {

// Найден узел для удаления

if (root->left == NULL) {

TreeNode\* temp = root->right; // Узел имеет только правое поддерево

free(root); // Освобождение памяти удаляемого узла

return temp; // Возврат правого поддерева

}

else if (root->right == NULL) {

TreeNode\* temp = root->left; // Узел имеет только левое поддерево

free(root); // Освобождение памяти удаляемого узла

return temp; // Возврат левого поддерева

}

// Узел имеет оба поддерева

TreeNode\* temp = findMin(root->right); // Поиск минимального значения в правом поддереве

root->value = temp->value; // Замена значения удаляемого узла

root->right = deleteNode(root->right, temp->value); // Удаление узла с минимальным значением

}

return root; // Возврат корня дерева

}

// Функция для создания нового узла

TreeNode\* newNode(int item) {

TreeNode\* temp = (TreeNode\*)malloc(sizeof(TreeNode)); // Выделение памяти для нового узла

temp->value = item; // Присвоение значения узлу

temp->left = temp->right = NULL; // Инициализация указателей на поддеревья как NULL

return temp; // Возврат нового узла

}

void printTree(TreeNode\* root, int space) {

int i;

// Базовый случай: если дерево пустое

if (root == NULL)

return;

// Увеличение расстояния между уровнями

space += 10;

// Обработка правого поддерева сначала (вывод в обратном порядке)

printTree(root->right, space);

// Печать текущего узла после пробелов

printf("\n");

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

printf(" ");

printf("%d\n", root->value);

// Обработка левого поддерева

printTree(root->left, space);

}

// Функция для печати дерева

void print(TreeNode\* root) {

// Первоначальный пробел

int space = 0;

// Вызов функции printTree

printTree(root, space);

}

int main() {

setlocale(LC\_ALL, "Rus");

// Создание дерева с корнем и поддеревьями

TreeNode\* root = newNode(50);

root->left = newNode(30);

root->right = newNode(70);

root->left->left = newNode(20);

root->left->right = newNode(40);

root->right->left = newNode(60);

root->right->right = newNode(80);

printf("Исходное дерево: ");

print(root); // Вывод исходного дерева

printf("\n");

root = deleteNode(root, 30); // Удаление узла с ключом 50

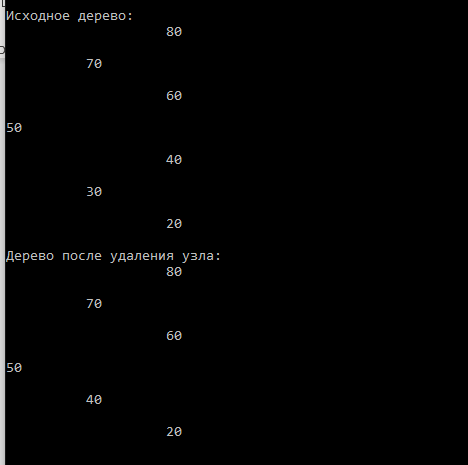
printf("Дерево после удаления узла: ");

print(root); // Вывод дерева после удаления узла

printf("\n");

return 0;

}



LB2.5.c

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

// Определение структуры узла дерева

typedef struct Node {

int data;

struct Node\* left, \* right;

} Node;

// Функция для создания нового узла

Node\* newNode(int data) {

Node\* node = (Node\*)malloc(sizeof(Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

// Функция для печати правого соседа каждого узла

void printRightNeighbor(Node\* root) {

if (root == NULL) return;

// Создаем очередь для уровневого обхода

Node\* queue[100];

int front = 0, rear = 0;

// Добавляем корень дерева в очередь

queue[rear++] = root;

queue[rear++] = NULL; // Маркер конца уровня

while (front < rear) {

Node\* current = queue[front++];

if (current == NULL) {

if (front < rear) {

queue[rear++] = NULL; // Добавляем маркер конца уровня

}

}

else {

// Печатаем данные текущего узла и его правого соседа

printf("%d – ", current->data);

if (queue[front] != NULL) {

printf("%d", queue[front]->data);

}

else {

printf("null");

}

printf(", ");

// Добавляем дочерние узлы в очередь

if (current->left != NULL) queue[rear++] = current->left;

if (current->right != NULL) queue[rear++] = current->right;

}

}

}

int main() {

setlocale(LC\_ALL, "Rus");

// Создаем узлы дерева согласно примеру

Node\* root = newNode(22);

root->left = newNode(16);

root->right = newNode(51);

root->left->left = newNode(7);

root->left->right = newNode(19);

root->right->left = newNode(43);

root->right->right = newNode(57);

// Печатаем правого соседа каждого узла

printRightNeighbor(root);

return 0;

}



LB2.6.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <locale.h>

#define MAX\_SIZE 100

typedef struct NodeHashTable {

char\* key;

char\* element;

struct NodeHashTable\* next;

struct NodeHashTable\* prev;

} NodeHashTable;

typedef struct HashTable {

NodeHashTable\*\* nodes;

int count;

} HashTable;

// 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, const char\* key, const 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 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");

}

}

}

// 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);

}

int main() {

setlocale(LC\_ALL, "Rus");

// Инициализация хеш-таблицы

HashTable\* ht = initHashTable();

// Добавление элементов в хеш-таблицу

HSET(ht, "AXkLn", "элемент1");

HSET(ht, "kqyzI", "элемент2");

HSET(ht, "ключ3", "элемент3");

// Вывод хеш-таблицы

printHashTable(ht);

// Получение элемента из хеш-таблицы

char\* element = HGET(ht, "kqyzI");

if (element != NULL) {

printf("Элемент с ключом 'ключ2': %s\n", element);

}

else {

printf("Элемент с ключом 'ключ2' не найден.\n");

}

// Удаление элемента из хеш-таблицы

HDEL(ht, "kqyzI");

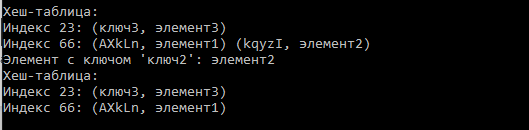
printHashTable(ht);

// Освобождение памяти хеш-таблицы

freeHashTable(ht);

return 0;

}



**Вывод**: Все программы корректно работают: в контрольном примере были взяты, если оно было, дано из самих заданий, и ответы моих программных кодов совпадали с заданиями.