|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ | | | | | | | | | | | | |
| Федеральное государственное автономное образовательное учреждение  высшего образования | | | | | | | | | | | | |
| **Дальневосточный федеральный университет** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **Департамент программной инженерии и искусственного интеллекта** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **О Т Ч Е Т** | | | | | | | | | | | | |
| по лабораторной работе №1.2  дисциплина «Фундаментальные структуры данных и алгоритмы» | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Студент гр. Б9121-09.03.04прогин | | | |
|  |  |  | | Шевелёв Р.В. | |
|  | | | | | | |  |  | (подпись) | |  | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Руководитель | | |  |
|  | | | | | | |  |  | ст. преподаватель | | | |
|  | | | |  |  | |  |  |  |  | О.А. Крестникова | |
|  | | | |  |  | |  |  | (подпись) |  | (И.О. Фамилия) | |
|  |  |  |  | | |  |  |  |  | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| г. Владивосток | | | | | | | | | | | | |
| 2022 | | | | | | | | | | | | |

**1 Неформальная постановка задачи**

Реализовать пакет подпрограмм для работы с бинарным деревом поиска (красно-чёрное дерево) с повторами, в котором элементы представлены в виде структуры (список из ИДЗ 1.1).  
Основные операции:  
1. Инициализация  
2. Освобождение памяти  
3. Добавление нового элемента (Ключ - номер рейса (1 поле - 2 буквы, 2 поле - 4 цифры))  
4. Удаление заданного элемента (замена на максимальный справа)  
5. Поиск заданного элемента  
6. Печать  
7. Сравнить два дерева (два дерева равны, если содержат одинаковые элементы)

**2 Описание типа + спецификация подпрограмм + тесты**

**struct** key – ключ узла

std::string num – первое значение ключа (строковый тип) – две буквы

int num – второе значение ключа (целочисленный тип) – четыре цифры

**struct** list\_elem – структура одного узла списка, находящегося в каждом узле дерева.

key data – поле значения элемента списка типа key

list\_elem\* next – значение следующего элемента списка (по адресу)

list\_elem\* prev – значение предыдущего элемента списка (по адресу)

int count – количество элементов (целочисленное значение)

**struct** tree\_elem – структура одного узла дерева.

list\_elem\* list – список, хранящий значение узла дерева

tree\_elem\* left – ссылка на левого потомка узла дерева

tree\_elem\* right – ссылка на правого потомка узла дерева

tree\_elem\* parent – ссылка на родителя узла дерева

bool color – значение цвета вершины логического типа (1 – черный, 0 - белый)

**list\_elem\*** list\_init() – инициализирует “голову” списка, возвращает голову со значением nullptr.

**void** list\_insert(list\_elem\*& head, key data) **–** вставка элемента в список. Если список пустой, создаёт голову, иначе увеличивает значение **count** списка.

**void** list\_erase(list\_elem\*& head, key data) – удаление элемента из списка. Если **count>1**, уменьшает его, иначе – удаляет голову и присваивает значение nullptr.

**void** print\_list(list\_elem\* root, bool color) – печать списка с заданным цветом (красный/белый).

**void** set\_color(char a) – установка необходимого цвета в консоли. ‘R’ – установка красного цвета, иначе – белого.

**tree\_elem\*** root\_init(tree\_elem\* nullnode) – инициализация корня дерева, возвращает корень со значением nullnode.

**tree\_elem**\* null\_init() – создаёт “нулевой лист” – nullnode для работы с деревом.

Выходные данные: nullnode типа tree\_elem.

**int** compare(key data1, key data2) – данная функция сравнивает два передаваемых в неё ключа по заданному правилу.

Входные данные: ключ1, ключ2

Выходные данные: {-1, 0 ,1}

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Первый ключ больше второго по первому значению | Ключ1: AZ1001  Ключ2: AA1001 | 1 |
| Первый ключ меньше второго по первому значению | Ключ1: AA1001  Ключ2: AZ1001 | -1 |
| Первый ключ больше второго по второму значению. | Ключ1: AA1002  Ключ2: AA1001 | 1 |
| Первый ключ меньше второго по второму значению. | Ключ1: AA1001  Ключ2: AA1002 | -1 |
| Ключи равны | Ключ1: AA1001  Ключ2: AA1001 | 0 |

bool search(tree\_elem\* root, tree\_elem\* nullnode, key data) – поиск заданного по ключу элемента в дереве.

Входные данные: ключ.

Выходные данные: {false, true}.

|  |  |  |
| --- | --- | --- |
|  | **Входные данные** | **Выходные данные** |
| Поиск в пустом дереве | Пустое дерево  Элемент: AA1002 | 0 |
| Поиск элемента в дереве, где присутствует заданный элемент | Дерево: AA1001, AA1002, AA1030, AA2030  Элемент: AA2030 | 1 |
| Поиск элемента в дереве, где отсутствует заданный элемент. | Дерево: AA1001, AA1002, AA1030, AA2030  Элемент: AA2032 | 0 |

**void** rightRotate(tree\_elem\* element, tree\_elem\*& root, tree\_elem\* nullnode) – преобразует структуру дерева. Входные данные – узел дерева, корень дерева, нулевой лист.

**void** leftRotate(tree\_elem\* element, tree\_elem\*& root, tree\_elem\* nullnode) – преобразует структуру дерева. Входные данные – узел дерева, корень дерева, нулевой лист.

**void** replace\_node(tree\_elem\*& root, tree\_elem\* nullnode, tree\_elem\* n, tree\_elem\* child)– преобразует структуру дерева. Заменяет одно поддерево с корнем n типа **tree\_elem** (являющимся дочерним по отношению к своему родителю), другим поддеревом с корнем child типа **tree\_elem**. Входные данные – корень дерева, нулевой лист, поддерево1, поддерево2.

**void** print(tree\_elem\* root, tree\_elem\* nullnode, int h, int ln) – печать дерева, положенного на бок. Входные данные: корень дерева, нулевой лист, количество отступов (целочисленное), количество переносов после печати узла (целочисленное).

**tree\_elem\*** left\_max(tree\_elem\* element, tree\_elem\* nullnode) – поиск максимального слева элемента для замены. Входные данные: левое поддерево относительно корня, нулевой лист. Выходные данные: максимальный элемент левого поддерева корня.

**tree\_elem\*** insert(tree\_elem\*& root, tree\_elem\* nullnode, key data) – добавляет в дерево узел со значение ключа data. Если такой узел уже существует, то добавляет в список (list\_elem) узла дерева ещё один элемент.

Входные данные: дерево, нулевой лист, ключ.

Выходные данные: корень дерева.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Добавление в пустое дерево. | AA1000 |  |
| Добавление ключа в левое поддерево. | AA1001  AA1000 |  |
| Добавление ключа в правое поддерево. | AA1000  AA1001 |  |
| Добавление повторяющегося ключа в левое поддерево. | AA1007  AA1005  AA1009  AA1009 |  |
| Добавление повторяющегося ключа в правое поддерево. | AA1007  AA1005  AA1009  AA1005 |  |
| Добавление повторяющегося ключа в корень. | AA1007  AA1005  AA1009  AA1007 |  |
| Добавление некорректного ключа в дерево | AA999 | Ключ не добавится, так как второе поле должно быть 4-х значным |
| Добавление некорректного ключа в дерево | A1023 | Ключ не добавится, так как первое поле должно состоять из двух букв |

**tree\_elem\*** erase(tree\_elem\*& root, tree\_elem\* nullnode, key data) – удаляет из дерева узел со значение ключа data. Входные данные: дерево, нулевой лист, ключ.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Удаление элемента из пустого дерева | Пустое дерево  Элемент: AA1000 | Пустое дерево |
| Удаление элемента без потомков | Дерево:  Элемент: AA1003 |  |
| Удаление элемента с одним потомком | Дерево:  Элемент: AA1002 |  |
| Удаление элемента с двумя потомками | Дерево:  Элемент: AA1003 |  |
| Удаление повторяющегося элемента | Дерево:  Элемент: AA1003 |  |
| Удаление элемента, которого нет в дереве | Дерево:  Элемент: AA1341 |  |

**tree\_elem\*** delete\_tree(tree\_elem\*& root, tree\_elem\* nullnode) – очистка дерева, освобождение памяти. Входные данные: корень дерева типа tree\_elem, нулевой лист. Выходные данные: нулевой лист.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Удаление пустого дерева | Пустое дерево | Нулевой лист (пустое дерево) |
| Удаление непустого дерева |  | Нулевой лист (пустое дерево) |
| Удаление непустого дерева |  | Нулевой лист (пустое дерево) |

**void** insert\_balance(tree\_elem\*& element, tree\_elem\*& root, tree\_elem\* nullnode) - данная функция балансирует дерево (восстанавливает свойства красно-черного дерева) с корнем равным root после добавления элемента element и листьями, ссылающимися на nullnode.

Входные данные: узел дерева, корень дерева, нулевой лист.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Добавление узла в левое поддерево.  Дядя красный. | Дерево с текущими узлами:  Добавление:  AA1000 |  |
| Добавление узла в правое поддерево. Дядя чёрный, менялось направление вставки. | Дерево с текущими узлами:  Добавление:  AA1001 |  |
| Добавление узла в правое поддерево. Дядя чёрный, не менялось направление вставки. | Дерево с текущими узлами:  После добавления:  Добавление:  AA1000 |  |
| Добавление узла в правое поддерево.  Дядя красный. | Дерево с текущими узлами:  Добавление:  AA1004 |  |
| Добавление узла в левое поддерево. Дядя чёрный, менялось направление вставки. | Дерево с текущими узлами:  Добавление:  AA1008 |  |
| Добавление узла в левое поддерево. Дядя чёрный, не менялось направление вставки. | Дерево с текущими узлами:  Добавление:  AA1009 |  |

**void** erase\_balance(tree\_elem\*& root, tree\_elem\* nullnode, tree\_elem\* element)-данная функция балансирует дерево (восстанавливает свойства красно-черного дерева) с корнем равным root после удаления элемента element и листьями, ссылающимися на nullnode.

Входные данные: корень дерева, нулевой лист, узел дерева.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Брат красный  (удаление слева) (1) | Дерево с текущими узлами:  Удаление: AA1004 |  |
| Брат красный (удаление справа) (1) | Дерево с текущими узлами:  Удаление: AA1013 |  |
| Брат черный. Оба потомка брата чёрные. (2) (Удаление слева) | Дерево с текущими узлами:    Удаление: АА1008 |  |
| Брат черный. Оба потомка брата чёрные. (2) (Удаление справа) | Дерево с текущими узлами:    Удаление: АА1010 |  |
| Брат чёрный. Внешний потомок брата чёрный. (3) (Удаление слева) | Дерево с текущими узлами:    Удаление: АА1005 |  |
| Брат чёрный. Внешний потомок брата чёрный. (3) (Удаление справа) | Дерево с текущими узлами:  Удаление: АА1006 |  |
| Брат чёрный. Внешний потомок брата – красный (4) (Удаление слева) | Дерево с текущими узлами:    Удаление: AA1012 |  |
| Брат чёрный. Внешний потомок брата – красный (4) (Удаление справа) | Дерево с текущими узлами:    Удаление: AA1004 |  |

**bool** equal\_tree(tree\_elem\* root, tree\_elem\* root1, tree\_elem\* nullnode) – функция, сравнивающая два дерева поэлементно на признак равенства. Два дерева равны, если содержат одинаковые элементы. Входные данные: дерево1, дерево2, нулевой лист. Выходные данные: {false, true}.

|  |  |  |
| --- | --- | --- |
| **Описание тестовой ситуации** | **Входные данные** | **Выходные данные** |
| Оба дерева пустые | Дерево1: пустое дерево  Дерево2: пустое дерево | 1 |
| Одно из деревьев пустое | (Дерево1 пустое И Дерево2 непустое) ИЛИ (Дерево1 непустое И Дерево2 пустое) | 0 |
| Деревья с одинаковыми элементами и одинаковым их количеством | Дерево1: 20 ключей по возрастанию  Дерево2: 20 ключей по убыванию | 1 |
| Деревья с разными элементами | Дерево1: AA1002, AA1003  Дерево2: AA1002, AA1004 | 0 |
| Деревья с одинаковыми элементами, но с разным их количеством | Дерево1: AA1002(1), AA1003(2)  Дерево2: AA1002(1), AA1003(4) | 0 |

**3 Текст программы**

#include <iostream>

#include <Windows.h>

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

struct key {

std::string sym;

int num;

};

struct list\_elem {

key data;

list\_elem\* next, \* prev;

int count;

};

list\_elem\* list\_init()

{

list\_elem\* root = nullptr;

return root;

}

void list\_insert(list\_elem\*& head, key data)

{

list\_elem\* insert = new(list\_elem);

insert->count = 1;

insert->data = data;

insert->next = insert;

insert->prev = insert;

if (head == nullptr) head = insert;

else head->count++;

}

void list\_erase(list\_elem\*& head, key data) {

if (head != nullptr) {

if (head->count == 1) {

head->count--;

delete head;

head = nullptr;

}

else head->count--;

}

}

void set\_color(char a) {

if (a == 'R') SetConsoleTextAttribute(hConsole, 12);

else SetConsoleTextAttribute(hConsole, 15);

}

void print\_list(list\_elem\* root, bool color)

{

if (root != nullptr) {

if (color) set\_color('B');

else set\_color('R');

list\_elem\* temp = root;

while (temp->next != root) {

std::cout << temp->data.sym << temp->data.num << " (" << temp->count << ")";

temp = temp->next;

}

std::cout << temp->data.sym << temp->data.num << " (" << temp->count << ")";

}

SetConsoleTextAttribute(hConsole, 15);

}

struct tree\_elem {

list\_elem\* list;

tree\_elem\* left, \* right, \* parent;

bool color; // True - Black

};

tree\_elem\* root\_init(tree\_elem\* nullnode) {

tree\_elem\* root = nullnode;

return root;

}

tree\_elem\* null\_init() {

tree\_elem\* nullnode = new tree\_elem;

nullnode->left = nullptr;

nullnode->right = nullptr;

nullnode->list = nullptr;

nullnode->color = true;

return nullnode;

}

tree\_elem\* nullnode = null\_init();

int compare(key data1, key data2) {

if (data1.sym > data2.sym) return 1;

if (data1.sym < data2.sym) return -1;

if (data1.num > data2.num) return 1;

if (data1.num < data2.num) return -1;

return 0;

}

bool search(tree\_elem\* root, tree\_elem\* nullnode, key data) {

tree\_elem\* temp = root;

while (temp != nullnode) {

if (compare(temp->list->data, data) == 0) return true;

if (compare(data, temp->list->data) == 1) temp = temp->right;

else temp = temp->left;

}

return false;

}

void rightRotate(tree\_elem\* element, tree\_elem\*& root, tree\_elem\* nullnode) {

tree\_elem\* y = element->left;

element->left = y->right;

if (y->right != nullnode) y->right->parent = element;

y->parent = element->parent;

if (element->parent == nullptr) root = y;

else if (element == element->parent->right) element->parent->right = y;

else element->parent->left = y;

y->right = element;

element->parent = y;

}

void leftRotate(tree\_elem\* element, tree\_elem\*& root, tree\_elem\* nullnode) {

tree\_elem\* y = element->right;

element->right = y->left;

if (y->left != nullnode) y->left->parent = element;

y->parent = element->parent;

if (element->parent == nullptr) root = y;

else if (element == element->parent->left) element->parent->left = y;

else element->parent->right = y;

y->left = element;

element->parent = y;

}

void insert\_balance(tree\_elem\*& element, tree\_elem\*& root, tree\_elem\* nullnode) {

tree\_elem\* uncle;

while (element->parent->color == false) {

if (element->parent == element->parent->parent->left) {

uncle = element->parent->parent->right;

if (uncle->color == false) {

uncle->color = true;

element->parent->color = true;

element->parent->parent->color = false;

element = element->parent->parent;

}

else {

if (element == element->parent->right) {

element = element->parent;

leftRotate(element, root, nullnode);

}

element->parent->color = true;

element->parent->parent->color = false;

rightRotate(element->parent->parent, root, nullnode);

}

}

else {

uncle = element->parent->parent->left;

if (uncle->color == false) {

uncle->color = true;

element->parent->color = true;

element->parent->parent->color = false;

element = element->parent->parent;

}

else {

if (element == element->parent->left) {

element = element->parent;

rightRotate(element, root, nullnode);

}

element->parent->color = true;

element->parent->parent->color = false;

leftRotate(element->parent->parent, root, nullnode);

}

}

if (element == root) { break; }

}

root->color = true;

}

tree\_elem\* insert(tree\_elem\*& root, tree\_elem\* nullnode, key data) {

key key = data;

if ((((data.num < -999) && (data.num > -10000)) || ((data.num < 10000) && (data.num > 999))) && (size(data.sym) == 2)) {

if (!search(root, nullnode, key)) {

tree\_elem\* element = new tree\_elem;

element->parent = nullptr;

element->left = nullnode;

element->right = nullnode;

element->color = false;

element->list = list\_init();

list\_insert(element->list, key);

tree\_elem\* parent\_buffer = nullptr;

tree\_elem\* runner = root;

while (runner != nullnode) {

parent\_buffer = runner;

if (compare(element->list->data, runner->list->data) == -1) runner = runner->left;

else runner = runner->right;

}

element->parent = parent\_buffer;

if (parent\_buffer == nullptr) root = element;

else if (compare(element->list->data, parent\_buffer->list->data) == -1) parent\_buffer->left = element;

else { parent\_buffer->right = element; }

if (element->parent == nullptr) {

element->color = true;

return root;

}

if (element->parent->parent == nullptr) return root;

insert\_balance(element, root, nullnode);

}

else {

tree\_elem\* temp = root;

while (temp != nullnode) {

if (compare(temp->list->data, key) == 0) { list\_insert(temp->list, key); break; }

if (compare(key, temp->list->data) == 1) { temp = temp->right; }

else { temp = temp->left; }

}

}

return root;

}

return nullnode;

}

void replace\_node(tree\_elem\*& root, tree\_elem\* nullnode, tree\_elem\* n, tree\_elem\* child) {

if ((n->parent == nullnode) || (n->parent == nullptr)) root = child;

else if (n == n->parent->left) n->parent->left = child;

else n->parent->right = child;

child->parent = n->parent;

}

void erase\_balance(tree\_elem\*& root, tree\_elem\* nullnode, tree\_elem\* element) {

tree\_elem\* brother;

while ((element != root) && (element->color == 1)) {

if (element == element->parent->left) {

brother = element->parent->right;

if (brother->color == false) {

brother->color = true;

element->parent->color = false;

leftRotate(element->parent, root, nullnode);

brother = element->parent->right;

}

if ((brother->left->color == true) && (brother->right->color == true)) {

brother->color = false;

element = element->parent;

}

else {

if (brother->right->color == true) {

brother->left->color = true;

brother->color = false;

rightRotate(brother, root, nullnode);

brother = element->parent->right;

}

brother->color = element->parent->color;

element->parent->color = true;

brother->right->color = true;

leftRotate(element->parent, root, nullnode);

element = root;

}

}

else {

brother = element->parent->left;

if (brother->color == false) {

brother->color = true;

element->parent->color = false;

rightRotate(element->parent, root, nullnode);

brother = element->parent->left;

}

if ((brother->left->color == true) && (brother->right->color == true)) {

brother->color = false;

element = element->parent;

}

else {

if (brother->left->color == true) {

brother->right->color = true;

brother->color = false;

leftRotate(brother, root, nullnode);

brother = element->parent->left;

}

brother->color = element->parent->color;

element->parent->color = true;

brother->left->color = true;

rightRotate(element->parent, root, nullnode);

element = root;

}

}

}

element->color = true;

}

tree\_elem\* left\_max(tree\_elem\* element, tree\_elem\* nullnode) {

while (element->right != nullnode) element = element->right;

return element;

}

tree\_elem\* erase(tree\_elem\*& root, tree\_elem\* nullnode, key data) {

if (root != nullptr) {

key key = data;

tree\_elem\* temp = root;

tree\_elem\* z = nullnode;

tree\_elem\* element;

tree\_elem\* y;

while (temp != nullnode) {

if (compare(temp->list->data, key) == 0) { z = temp; }

if ((compare(temp->list->data, key) == 0) || (compare(temp->list->data, key) == -1)) { temp = temp->right; }

else { temp = temp->left; }

}

if (z == nullnode) return root;

if (z->list->count > 1) list\_erase(z->list, key);

else {

y = z;

bool y\_original\_color = y->color;

if (z->left == nullnode) {

element = z->right;

replace\_node(root, nullnode, z, z->right);

}

else if (z->right == nullnode) {

element = z->left;

replace\_node(root, nullnode, z, z->left);

}

else {

y = left\_max(z->left, nullnode);

y\_original\_color = y->color;

element = y->left;

if (y->parent == z) {

element->parent = y;

}

else {

replace\_node(root, nullnode, y, y->left);

y->left = z->left;

y->left->parent = y;

}

replace\_node(root, nullnode, z, y);

y->right = z->right;

y->right->parent = y;

y->color = z->color;

}

list\_erase(z->list, key);

delete z;

if (y\_original\_color == true) erase\_balance(root, nullnode, element);

}

return root;

}

return nullptr;

}

void delete\_list(tree\_elem\*& root, tree\_elem\* nullnode, key key) {

tree\_elem\* temp = root;

if (temp != nullnode) {

while (temp->list->count > 1) list\_erase(temp->list, key);

list\_erase(temp->list, key);

temp->left = nullptr;

temp->right = nullptr;

temp->parent = nullptr;

temp->list = nullptr;

delete temp;

temp = nullptr;

}

}

void delete\_node(tree\_elem\*& root, tree\_elem\* nullnode) {

if (root->left != nullnode) delete\_node(root->left, nullnode);

if (root->right != nullnode) delete\_node(root->right, nullnode);

delete\_list(root, nullnode, root->list->data);

}

tree\_elem\* delete\_tree(tree\_elem\*& root, tree\_elem\* nullnode)

{

delete\_node(root, nullnode);

root = nullptr;

return nullnode;

}

void print(tree\_elem\* root, tree\_elem\* nullnode, int h, int ln) {

if (root != nullptr) {

if ((root != nullnode) && (root != nullptr)) {

print(root->right, nullnode, h + 4, ln);

for (int i = 1; i <= h; i++) std::cout << ' ';

print\_list(root->list, root->color);

for (int i = 0; i < ln; i++) std::cout << '\n';

print(root->left, nullnode, h + 4, ln);

}

}

}

bool equal(tree\_elem\* root, tree\_elem\* root1, tree\_elem\* nullnode, bool ok = false) {

if (ok) {

if (root1 == nullnode) return false;

else return ((compare(root->list->data, root1->list->data) == 0) && (root->list->count == root1->list->count)) || equal(root, root1->left, nullnode, true) || equal(root, root1->right, nullnode, true);

}

else {

if (root == nullnode) return true;

else return equal(root->left, root1, nullnode) && equal(root, root1, nullnode, true) && equal(root->right, root1, nullnode);

}

}

bool equal\_tree(tree\_elem\* root, tree\_elem\* root1, tree\_elem\* nullnode) {

if (root == nullnode && root1 == nullnode) return true;

else if (root == nullnode or root1 == nullnode) return false;

return equal(root, root1, nullnode);

}

int main() {

setlocale(LC\_ALL, "Russian");

tree\_elem\* root = root\_init(nullnode);

}