Міністерство освіти і науки України

Національний технічний університет

«Харківський політехнічний інститут»

Навчально-науковий інститут комп’ютерних наук та інформаційних технологій

ЗВІТ

про виконання лабораторної роботи №3

Варіант 7

за дисципліною

«Алгоритми і структури даних»

Виконав:

студент групи КН-320Б

Миргород В.І.

Перевірила:

старший викладач

Мошко Є.О.

Харків 2022

**Лабораторна робота №3**

**Тема лабораторної роботи:** Бінарні дерева та червоно-чорні дерева

**Мета:** Набуття практичних вмінь та навичок опрацювання динамічних  
структур даних, представлених у вигляді бінарних та червоно-чорних дерев.

**Завдання за варіантом:**

1. Назва наукового журналу  
2. Рік випуску  
3. Кількість номерів

**Завдання:**

1. Написати програму, в якій дані варіанту (таблиця 1) структури записуються в  
бінарне дерево (використати три поля для вузла – текстове дане та два числові.  
Наприклад, вузол дерева містить таку корисну інформацію: прізвище студента, рік  
народження, оцінка по іспиту). Ввести з клавіатури декілька "студентів" у двійкове  
дерево, організоване за порядком текстового поля. Роздрукувати отримане бінарне  
дерево.  
2. Знайти середнє значення одного з числових полів, зчитуючи дані з дерева.  
3. Надрукувати значення бінарного дерева:  
а) при прямому обході дерева;  
б) при зворотному обході дерева;  
в) при симетричному обході дерева.  
4. Дописати функцію видалення з пам’яті всього бінарного дерева.  
5. Створити рекурсивну функцію, яка:  
− видаляє ліве піддерево, і ліву гілку занулює;  
− видаляє праве піддерево, і праву гілку занулює;  
− видаляє сам вузол, потім зануливши вказівник на нього.  
Наприкінці програми видалити з пам’яті дерево.  
6. "Пересипати" дані з першого дерева у друге дерево того ж типу, тільки  
організованого за першим числовим ключем (напр., рік народження) та  
роздрукувати його (а перше дерево стерти).  
7. Розфарбувати вершини заданого бінарного дерева в червоний і чорний  
кольори так, щоб воно стало червоно-чорним деревом.  
8. Провести перевірку властивості червоно-чорного дерева, а саме:

− кожен вузол промаркований червоним або чорним кольором;  
− корінь дерева – чорний;  
− кінцеві вузли дерева – чорні;  
− біля червоного вузла батьківський вузол — чорний;  
− усі прості шляхи з будь-якого вузла до листя містять однакову кількість  
чорних вузлів.  
9. У червоно-чорному дереві виконати додавання і видаленні вузлів та провести  
перевірку властивості червоно-чорного дерева.

Аналіз завдання:

Бінарне дерево – динамічна структура даних, яка складається з вузлів, кожен з яких має покажчик на 2 наступні вузли – лівий та правий. Сама структура зберігає покажчик на перший елемент дерева, тобто корінь (root). Нові елементи у бінарному дереві додаються на пусті місця, після вже існуючих вузлів.

Бінарне дерево пошуку – різновид бінарного дерева. Його особливість полягає в тому, що кожний лівий лист будь-якого вузла має значення, менше за сам вузел, а правий лист має значення більше значення вузла. При додаванні нових листів враховується це правило.

Структура вхідних та вихідних даних:

Дані вводяться користувачем з консолі або зчитуються з файлу. Результат виводиться до консолі у вигляді дерева. Тип даних – string, int, int.

**Алгоритм розв’язання задачі:**

При створенні нового дерева, покажчик на його корінь має значення NULL.

Далі, заповнюємо дерево значення, для цього послідовно викликаємо метод Add(Node\* node).

void BinaryTree<T1, T2, T3>::Add(Node<T1, T2, T3>\* node, Node<T1, T2, T3>\* leaf) {

if (leaf == NULL) {

root = node;

return;

}

if (node->key < leaf->key)

{

if (leaf->left != NULL)

Add(node, leaf->left);

else

leaf->left = node;

}

else if (node->key >= leaf->key)

{

if (leaf->right != NULL)

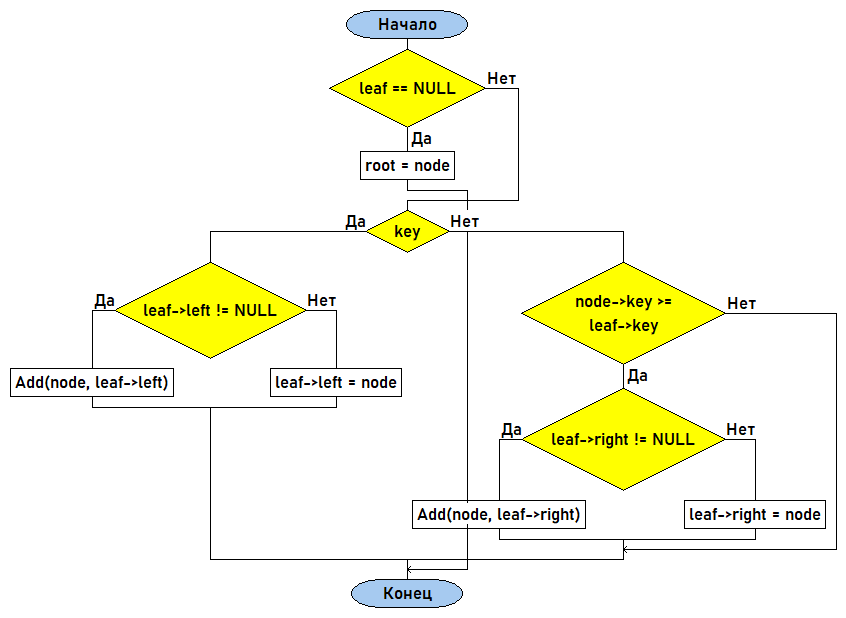
Add(node, leaf->right);

else

leaf->right = node;

}

}



Щоб видалити вузол з дерева, використовується метод DeleteNode(Node\*). Він отримує адресу вузла, що потрібно видалити, і далі перевіряє наявність листів у даного вузла. В залежності від того, чи є в цього вузла один лист, два, чи жодного, обирається спосіб видалення. Як правило, вузол замінюється одним зі своїх листів, або листів дерева.

void BinaryTree<T1, T2, T3>::DeleteNode(Node<T1, T2, T3>\* node, Node<T1, T2, T3>\* parent) {

if (node == NULL)

return;

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

if (parent == NULL) {

delete node;

this->root = NULL;

}

else {

if (parent->left->key == node->key)

parent->left = NULL;

else

parent->right = NULL;

delete node;

}

return;

}

if (node->left == NULL) {

if (parent == NULL) {

this->root = node->right;

delete node;

return;

}

if (parent->left != NULL && parent->left->key == node->key)

parent->left = node->right;

else

parent->right = node->right;

delete node;

return;

}

if (node->right == NULL) {

if (parent == NULL) {

this->root = node->left;

delete node;

return;

}

if (parent->left != NULL && parent->left->key == node->key)

parent->left = node->left;

else

parent->right = node->left;

delete node;

return;

}

Node<T1, T2, T3>\* temp1 = node->right;

if (temp1->left == NULL) {

temp1->left = node->left;

if (parent == NULL)

this->root = temp1;

else {

if (parent->left != NULL && parent->left->key == node->key)

parent->left = temp1;

else

parent->right = temp1;

}

delete node;

return;

}

while (temp1->left->left != NULL)

temp1 = temp1->left;

temp1->left->left = node->left;

temp1->left->right = node->right;

if (parent == NULL)

this->root = temp1->left;

else {

if (parent->left != NULL && parent->left->key == node->key)

parent->left = temp1->left;

else

parent->right = temp1->left;

}

temp1->left = NULL;

delete node;

}

Для пошуку потрібного вузла викликається метод Find(). Він в тому числі потрібен під час видалення елемента, де передається вже адреса вузла. Метод повертає адресу потрібного вузла, а також адресу його батьківського вузла.

void BinaryTree<T1, T2, T3>::FindNode(Node<T1, T2, T3>\*\* node, Node<T1, T2, T3>\*\* parent, T1 val) {

if (this->root == NULL) {

cout << "Бинарное дерево пустое.\n";

return;

}

if ((\*node)->key > val) {

if ((\*node)->left != NULL) {

(\*parent) = (\*node);

(\*node) = (\*node)->left;

FindNode(node, parent, val);

}

else {

cout << "Не найдено.\n";

(\*node) = NULL;

(\*parent) = NULL;

}

}

else if ((\*node)->key < val) {

if ((\*node)->right != NULL) {

(\*parent) = (\*node);

(\*node) = (\*node)->right;

FindNode(node, parent, val);

}

else {

cout << "Не найдено.\n";

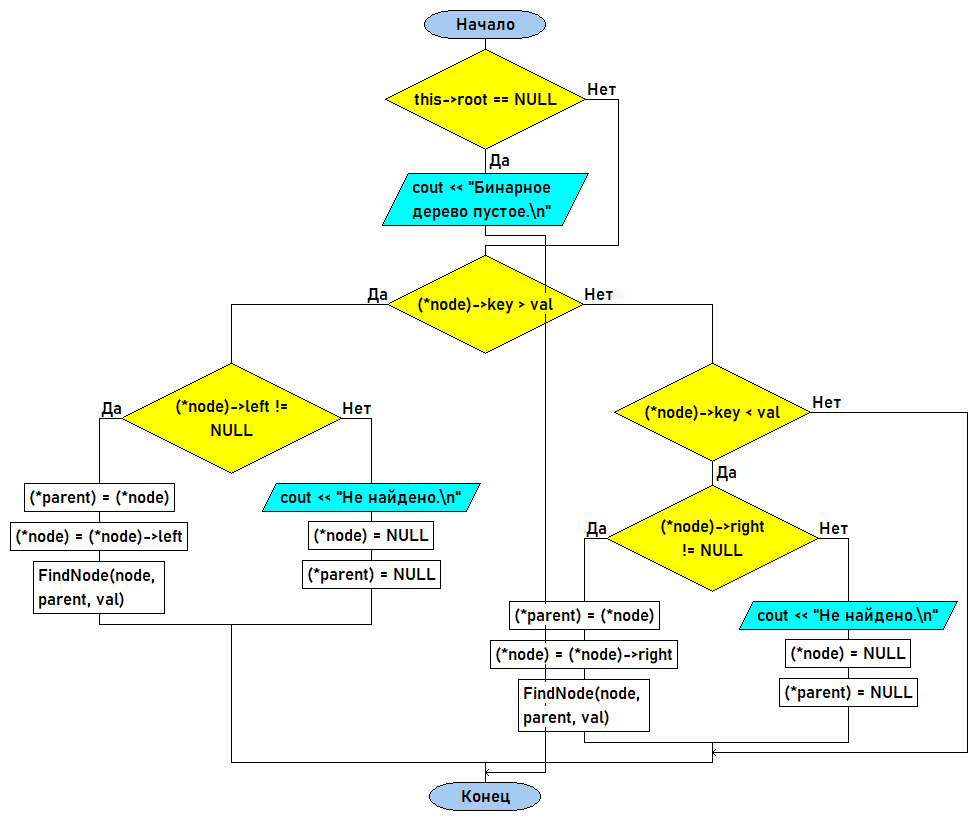
(\*node) = NULL;

(\*parent) = NULL;

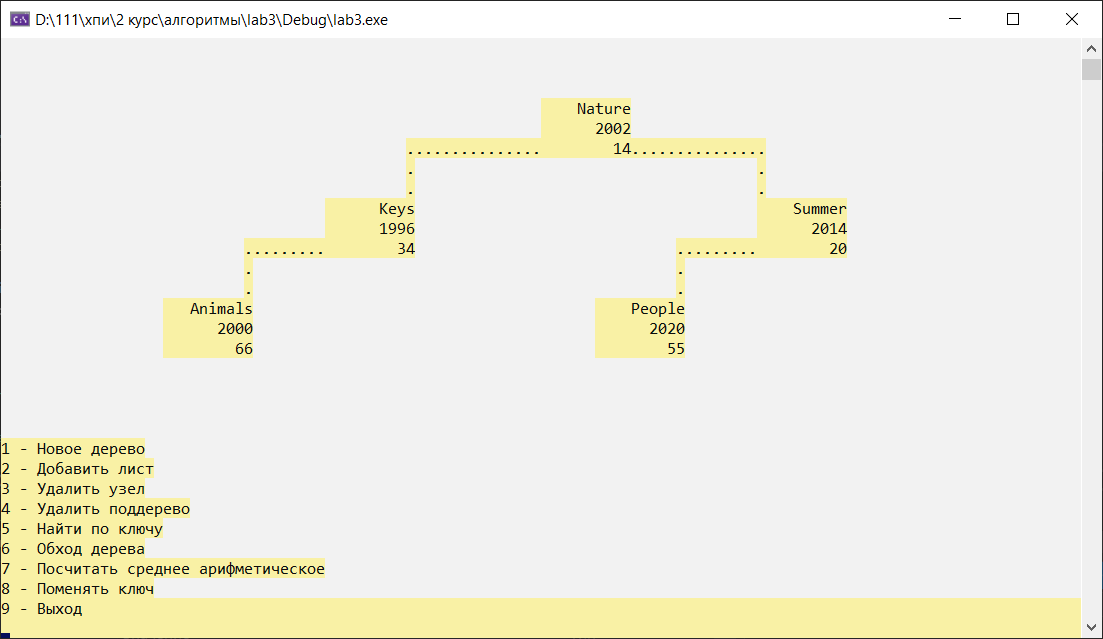
}

}

}



Програма під час роботи:



Червоно-чорне дерево – один з видів бінарного дерева пошуку. Кожний елемент такого дерева зберігає додатково колір вузла. Колір може бути червоним або чорним. Також це дерево є самозбалансованим, тобто при додаванні або видаленні вузла, дерево автоматично змінює порядок або колір вузлів, щоб стати збалансованим. Збалансоване дерево – в якого будь-який шлях від листа до кореня має однакову чорну висоту. Також червоно-чорне дерево має відповідати кільком правилам:

* Корінь завжди чорний
* Кожний лист дерева (NULL-елемент) – чорний
* Червоний вузол має 2 наступні елементи чорного кольору

Для додавання елементів у дерево використовуються майже ті ж самі методи, що й для звичайного бінарного дерева. Але під час додавання в червоно-чорне дерево, потрібно змінювати колір деяких елементів, так щоб виконувались властивості червоно-чорного дерева.

void RBTree<T1, T2, T3>::Add(RBNode<T1, T2, T3>\* node, RBNode<T1, T2, T3>\* leaf) {

if (leaf == NULL) {

root = node;

Case1(node);

return;

}

if (node->key < leaf->key)

{

if (leaf->left != NULL)

Add(node, leaf->left);

else {

leaf->left = node;

node->parent = leaf;

}

}

else if (node->key >= leaf->key)

{

if (leaf->right != NULL)

Add(node, leaf->right);

else {

leaf->right = node;

node->parent = leaf;

}

}

Case1(node);

}  
Після додавання елемента, в деяких випадках потрібна перевірка, що відповідає за збереження основних властивостей дерева.

Видалення елемента в червоно-чорному дереві більш складне. Саме видалення відбувається майже так, як і у звичайному дереві, але також залежить від кольору вузла, що видаляється, і кольорів його сусідніх вузлів. Під час видалення чорного вузла, що не має нащадків, потрібно викликати додатковий метод балансування. Завдяки цьому, видалення елементів не шкодить основному правилу червоно-чорного дерева – чорна висота всіх листів має однакове значення.

void RBTree<T1, T2, T3>::FindNode(RBNode<T1, T2, T3>\*\* node, RBNode<T1, T2, T3>\*\* parent, T1 val) {

//node и parent - указатели, при вызове создать новые указатели на root и null !!!

//в них же вернется результат - указатель на node и его root

if (this->root == NULL) {

cout << "Бинарное дерево пустое.\n";

return;

}

if ((\*node)->key > val) {

if ((\*node)->left != NULL) {

(\*parent) = (\*node);

(\*node) = (\*node)->left;

FindNode(node, parent, val);

}

else {

cout << "Не найдено.\n";

(\*node) = NULL;

(\*parent) = NULL;

}

}

else if ((\*node)->key < val) {

if ((\*node)->right != NULL) {

(\*parent) = (\*node);

(\*node) = (\*node)->right;

FindNode(node, parent, val);

}

else {

cout << "Не найдено.\n";

(\*node) = NULL;

(\*parent) = NULL;

}

}

}

template <class T1, class T2, class T3>

void RBTree<T1, T2, T3>::DeleteNode(RBNode<T1, T2, T3>\* node) {

if (node == NULL)

return;

if (node->color == "red" && node->left == NULL && node->right == NULL) { //красный узел без потомков

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

node->parent->left = NULL;

delete node;

}

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

node->parent->right = NULL;

delete node;

}

}

else if (node->color == "red" && node->left != NULL && node->right != NULL) { //красные узел, 2 потомка

RBNode<T1, T2, T3>\* temp = node->right;

while (temp->left != NULL) //находим похожий элемент, для замены

temp = temp->left;

T1 k = temp->key;

T2 d1 = temp->data1;

T3 d2 = temp->data2;

temp->key = node->key;

temp->data1 = node->data1;

temp->data2 = node->data2;

node->key = k;

node->data1 = d1;

node->data2 = d2;

DeleteNode(temp);

}

else if (node->color == "black" && node->left == NULL && node->right == NULL) { //черный узел без наследников

RBNode<string, int, int>\* p = node->parent;

if (p == NULL) {

Clear(node);

this->root = NULL;

return;

}

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

node->parent->left = NULL;

delete node;

Balance(p, "left");

}

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

node->parent->right = NULL;

delete node;

Balance(p, "right");

}

}

else if (node->color == "black" && (node->left == NULL || node->right == NULL)) {

if (node->left != NULL) {

node->key = node->left->key;

node->data1 = node->left->data1;

node->data2 = node->left->data2;

DeleteNode(node->left);

}

else if (node->right != NULL) {

node->key = node->right->key;

node->data1 = node->right->data1;

node->data2 = node->right->data2;

DeleteNode(node->right);

}

}

else if (node->color == "black" && node->left != NULL && node->right != NULL) {

RBNode<T1, T2, T3>\* temp = node->right;

while (temp->left != NULL) //находим похожий элемент, для замены

temp = temp->left;

T1 k = temp->key;

T2 d1 = temp->data1;

T3 d2 = temp->data2;

temp->key = node->key;

temp->data1 = node->data1;

temp->data2 = node->data2;

node->key = k;

node->data1 = d1;

node->data2 = d2;

DeleteNode(temp);

}

}

Після видалення чорного вузла без нащадків, викликається метод Balance, який збалансовує дерево.

void RBTree<T1, T2, T3>::Balance(RBNode<T1, T2, T3>\* p, string type) {

RBNode<T1, T2, T3>\* b;

if (type == "left") {

b = p->right;

if (b->color == "black") {

if (b->right != NULL && b->right->color == "red") {

b->color = p->color;

p->color = "black";

b->right->color = "black";

RotateLeft(b);

return;

}

else if (b->left != NULL && b->left->color == "red") {

b->color = "red";

b->left->color = "black";

RotateRight(b->left);

Balance(p, "left");

}

else {

b->color = "red";

if (p->color == "red") {

p->color = "black";

return;

}

else {

if (p->parent != NULL && p->parent->left == p)

Balance(p->parent, "left");

else if (p->parent != NULL && p->parent->right == p)

Balance(p->parent, "right");

}

}

}

else if (b->color == "red") {

p->color = "red";

b->color = "black";

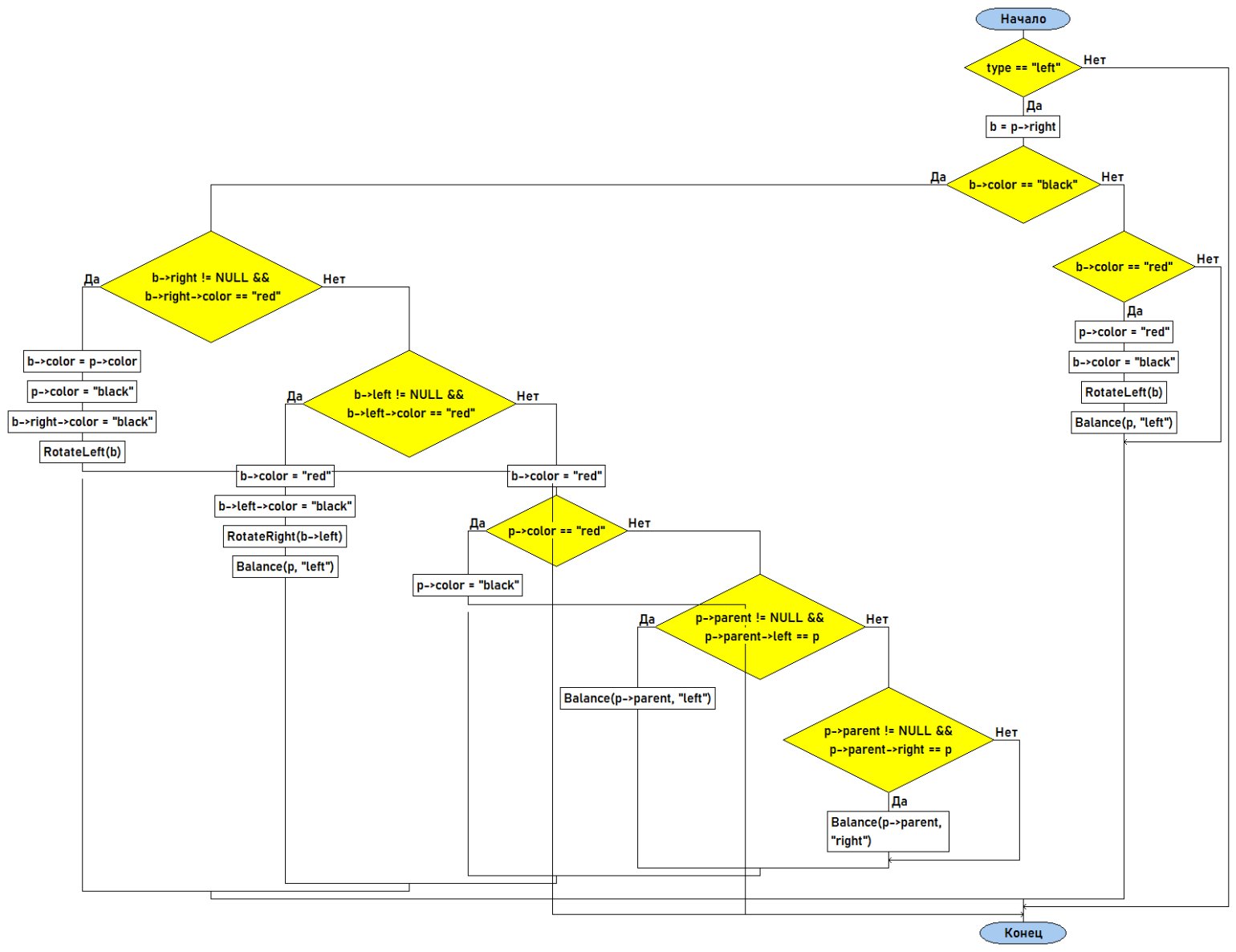
RotateLeft(b);

Balance(p, "left");

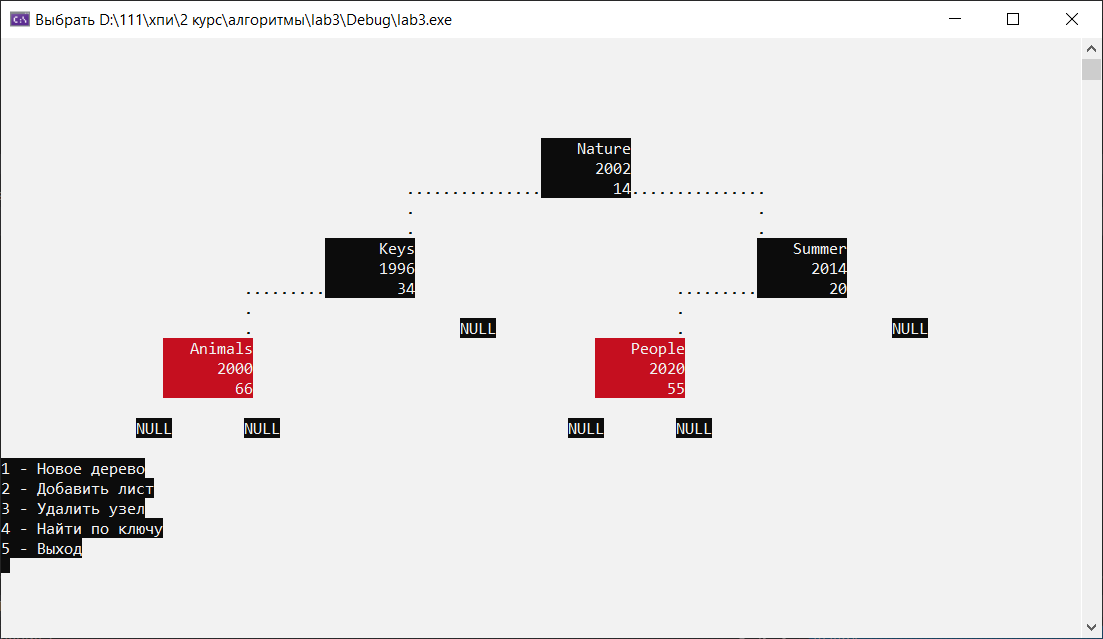
}

}

}



Програма під час роботи з червоно-чорним деревом:



**Висновки:**

Бінарне дерево є зручною структурою для зберігання даний. Дозволяє легко та швидко змінювати розмір структури, додаючи або видаляючи вузли. В залежності від типу дерева, додавання та видалення може проходити з деякими умовами.

У бінарному дереві пошуку всі елементи додаються так, щоб лівий елемент був менший за батьківський, а правий – більший. Таке розташування елементів дозволяє швидко здійснювати пошук елемента у дереві.

Червоно-чорне дерево має всі переваги бінарного дерева пошуку, а також є самозбалансованим. Всі висоти дерева відрізняються не більше ніж на h – 1, а під час додавання вузла потрібно виконати максимум 3 повороти.

Складність виконання дій над деревом:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Бінарне дерево пошуку | | Червоно-чорне дерево | |
| Пошук | O(logn) | O(n) | O(logn) | O(logn) |
| Вставка | O(logn) | O(n) | O(logn) | O(logn) |
| Видалення | O(logn) | O(n) | O(logn) | O(logn) |
|  | Середній в. | Найгірший в. | Середній в. | Найгірший в. |