**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЁТ**

**По лабораторной работе №1**

**По дисциплине «Алгоритмы и структуры данных»**

Студент гр. 3351 \_\_\_\_\_\_\_\_\_\_\_\_ Морозов А.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ Пестерев Д.О.

Санкт-Петербург

2024

цель работы

**Практическая часть:**

1. Реализовать бинарное дерево поиска, красно-черное дерево и АВЛ дерево (структура, балансировка, операции вставки/удаления/поиска).

2. Получить зависимость высоты дерева поиска от количества ключей, при условии, что значение ключа - случайная величина, распределенная равномерно. Какая асимпотика функции h(n) наблюдается у двоичного дерева поиска?

3. Получить зависимость АВЛ и красно-черного дерева поиска от количества ключей, при условии, что значения ключей монотонно возрастают.

4. Вывести полученные результаты на графики.

5. Сравнить с теоретической оценкой высоты.

6. Реализовать обходы в глубину и обход в ширину двоичного дерева с выводом результата.

**Теоретическая часть:**

1. Определение АВЛ дерева. Кратко описать алгоритм вставки/удаления с последующей балансировкой для АВЛ-дерева. Получить верхнюю оценку высоты АВЛ-дерева.

2. Определение красно-черного дерева. Кратко описать алгоритм вставки/удаления с последующей балансировкой для красно-черного дерева. Получить верхнюю оценку высоты красно-черного дерева.

**Текст программы**

**Бинарное дерево**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <vector>

#include <queue> // Для обхода в ширину

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

Node(int value) : key(value), left(nullptr), right(nullptr) {}

};

class BST {

public:

Node\* root;

BST() : root(nullptr) {}

void insert(int key) {

root = insertRec(root, key);

}

int height() {

return heightRec(root);

}

void preorder() {

cout << "Preorder traversal: ";

preorderRec(root);

cout << endl;

}

void inorder() {

cout << "Inorder traversal: ";

inorderRec(root);

cout << endl;

}

void postorder() {

cout << "Postorder traversal: ";

postorderRec(root);

cout << endl;

}

void levelOrder() {

cout << "Level order traversal: ";

if (root == nullptr) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

cout << current->key << " ";

if (current->left != nullptr) {

q.push(current->left);

}

if (current->right != nullptr) {

q.push(current->right);

}

}

cout << endl;

}

private:

Node\* insertRec(Node\* node, int key) {

if (node == nullptr) {

return new Node(key);

}

if (key < node->key) {

node->left = insertRec(node->left, key);

}

else {

node->right = insertRec(node->right, key);

}

return node;

}

int heightRec(Node\* node) {

if (node == nullptr) {

return 0;

}

int leftHeight = heightRec(node->left);

int rightHeight = heightRec(node->right);

return max(leftHeight, rightHeight) + 1;

}

void preorderRec(Node\* node) {

if (node == nullptr) return;

cout << node->key << " ";

preorderRec(node->left);

preorderRec(node->right);

}

void inorderRec(Node\* node) {

if (node == nullptr) return;

inorderRec(node->left);

cout << node->key << " ";

inorderRec(node->right);

}

void postorderRec(Node\* node) {

if (node == nullptr) return;

postorderRec(node->left);

postorderRec(node->right);

cout << node->key << " ";

}

};

int main() {

srand(static\_cast<unsigned int>(time(0))); // Инициализация генератора случайных чисел

vector<int> sizes = { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101 }; // Размеры

vector<int> heights;

const int numTrials = 10; // Количество попыток для усреднения

for (int size : sizes) {

int totalHeight = 0;

for (int trial = 0; trial < numTrials; ++trial) {

BST tree;

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

int key = rand() % 10000; // Генерация случайного ключа

tree.insert(key);

}

totalHeight += tree.height();

}

int averageHeight = totalHeight / numTrials;

heights.push\_back(averageHeight);

cout << "BST;" << size << ";" << averageHeight << endl;

}

// Обходы

BST tree;

for (int i = 2; i <= 101; ++i) {

tree.insert(i);

}

tree.inorder();

tree.preorder();

tree.postorder();

tree.levelOrder();

return 0;

}

**AVL дерево**

#include <iostream>

#include <vector>

#include <queue> // Для обхода в ширину

using namespace std;

struct AVLNode {

int key;

AVLNode\* left;

AVLNode\* right;

int height;

AVLNode(int value) : key(value), left(nullptr), right(nullptr), height(1) {}

};

class AVLTree {

public:

AVLNode\* root;

AVLTree() : root(nullptr) {}

int height(AVLNode\* node) {

return node ? node->height : 0;

}

int getBalance(AVLNode\* node) {

return node ? height(node->left) - height(node->right) : 0;

}

AVLNode\* rightRotate(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

AVLNode\* leftRotate(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

AVLNode\* insert(AVLNode\* node, int key) {

if (node == nullptr) {

return new AVLNode(key);

}

if (key < node->key) {

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

}

else if (key > node->key) {

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

}

else {

return node; // Duplicate keys are not allowed

}

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

// Left Left Case

if (balance > 1 and key < node->left->key) {

return rightRotate(node);

}

// Right Right Case

if (balance < -1 and key > node->right->key) {

return leftRotate(node);

}

// Left Right Case

if (balance > 1 and key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 and key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void insertKey(int key) {

root = insert(root, key);

}

int getHeight() {

return height(root);

}

// Обход в глубину (слева)

void preorder() {

cout << "Preorder traversal: ";

preorderRec(root);

cout << endl;

}

// Обход в глубину (симметричный)

void inorder() {

cout << "Inorder traversal: ";

inorderRec(root);

cout << endl;

}

// Обход в глубину (справа)

void postorder() {

cout << "Postorder traversal: ";

postorderRec(root);

cout << endl;

}

// Обход в ширину

void levelOrder() {

cout << "Level order traversal: ";

if (root == nullptr) return;

queue<AVLNode\*> q;

q.push(root);

while (!q.empty()) {

AVLNode\* current = q.front();

q.pop();

cout << current->key << " ";

if (current->left != nullptr) {

q.push(current->left);

}

if (current->right != nullptr) {

q.push(current->right);

}

}

cout << endl;

}

private:

void preorderRec(AVLNode\* node) {

if (node == nullptr) return;

cout << node->key << " ";

preorderRec(node->left);

preorderRec(node->right);

}

void inorderRec(AVLNode\* node) {

if (node == nullptr)

return;

inorderRec(node->left);

cout << node->key << " ";

inorderRec(node->right);

}

void postorderRec(AVLNode\* node) {

if (node == nullptr)

return;

postorderRec(node->left);

postorderRec(node->right);

cout << node->key << " ";

}

};

int main() {

AVLTree avlTree;

for (int n = 2; n <= 101; ++n) {

avlTree.insertKey(n);

cout << "AVL;" << n << ";" << avlTree.getHeight() << endl;

}

avlTree.preorder();

avlTree.inorder();

avlTree.postorder();

avlTree.levelOrder();

return 0;

}

**Красно-чёрное дерево**

#include <iostream>

#include <queue> // Для обхода в ширину

using namespace std;

enum Color { RED, BLACK };

struct Node {

int data;

Color color;

Node\* left, \* right, \* parent;

Node(int data) : data(data), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}

};

class RedBlackTree {

private:

Node\* root;

void rotateLeft(Node\*& node) {

Node\* node\_right = node->right;

node->right = node\_right->left;

if (node->right != nullptr) {

node->right->parent = node;

}

node\_right->parent = node->parent;

if (node->parent == nullptr) {

root = node\_right;

}

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

node->parent->left = node\_right;

}

else {

node->parent->right = node\_right;

}

node\_right->left = node;

node->parent = node\_right;

}

void rotateRight(Node\*& node) {

Node\* node\_left = node->left;

node->left = node\_left->right;

if (node->left != nullptr) {

node->left->parent = node;

}

node\_left->parent = node->parent;

if (node->parent == nullptr) {

root = node\_left;

}

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

node->parent->left = node\_left;

}

else {

node->parent->right = node\_left;

}

node\_left->right = node;

node->parent = node\_left;

}

void fixViolation(Node\*& node) {

Node\* parent = nullptr;

Node\* grandparent = nullptr;

while ((node != root) and (node->color == RED) and (node->parent->color == RED)) {

parent = node->parent;

grandparent = parent->parent;

if (parent == grandparent->left) {

Node\* uncle = grandparent->right;

if (uncle != nullptr and uncle->color == RED) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

node = grandparent;

}

else {

if (node == parent->right) {

rotateLeft(parent);

node = parent;

parent = node->parent;

}

rotateRight(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

}

else {

Node\* uncle = grandparent->left;

if ((uncle != nullptr) and (uncle->color == RED)) {

grandparent->color = RED;

parent->color = BLACK;

uncle->color = BLACK;

node = grandparent;

}

else {

if (node == parent->left) {

rotateRight(parent);

node = parent;

parent = node->parent;

}

rotateLeft(grandparent);

swap(parent->color, grandparent->color);

node = parent;

}

}

}

root->color = BLACK;

}

public:

RedBlackTree() : root(nullptr) {}

void insert(const int& data) {

Node\* newNode = new Node(data);

root = BSTInsert(root, newNode);

fixViolation(newNode);

}

Node\* BSTInsert(Node\* root, Node\* newNode) {

if (root == nullptr) {

return newNode;

}

if (newNode->data < root->data) {

root->left = BSTInsert(root->left, newNode);

root->left->parent = root;

}

else if (newNode->data > root->data) {

root->right = BSTInsert(root->right, newNode);

root->right->parent = root;

}

return root;

}

void inorder() {

inorderHelper(root);

}

void inorderHelper(Node\* root) {

if (root == nullptr) return;

inorderHelper(root->left);

cout << root->data << " ";

inorderHelper(root->right);

}

void preorder() {

cout << "Preorder traversal: ";

preorderHelper(root);

cout << std::endl;

}

void preorderHelper(Node\* node) {

if (node == nullptr) return;

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

preorderHelper(node->left);

preorderHelper(node->right);

}

void postorder() {

cout << "Postorder traversal: ";

postorderHelper(root);

cout << std::endl;

}

void postorderHelper(Node\* node) {

if (node == nullptr)

return;

postorderHelper(node->left);

postorderHelper(node->right);

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

}

void levelOrder() {

cout << "Level order traversal: ";

if (root == nullptr) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

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

if (current->left != nullptr) {

q.push(current->left);

}

if (current->right != nullptr) {

q.push(current->right);

}

}

cout << endl;

}

int getHeight() {

return getHeightHelper(root);

}

int getHeightHelper(Node\* node) {

if (node == nullptr) return 0;

return 1 + max(getHeightHelper(node->left), getHeightHelper(node->right));

}

};

int main() {

RedBlackTree rbt;

cout << "Red-Black Tree" << endl;

for (int i = 2; i <= 101; ++i) {

rbt.insert(i);

cout << "RBT;" << i << ";" << rbt.getHeight() << endl;

}

cout << "Inorder traversal of Red-Black Tree: ";

rbt.inorder();

cout << endl;

rbt.preorder();

rbt.postorder();

rbt.levelOrder();

return 0;

}

**Описание программы**

Программное обеспечение – Microsoft Visual Studio 2022. Язык программирования – C++.

**Теоретическая часть**

**Определение АВЛ-дерева**

АВЛ-дерево (названное в честь его изобретателей Адельсона-Вельского и Ландиса) — это самобалансирующееся двоичное дерево поиска, в котором для каждого узла разница высот его левого и правого поддеревьев (называемая фактором баланса) может быть только -1, 0 или +1. Это означает, что дерево всегда остается сбалансированным, что обеспечивает логарифмическую сложность операций поиска, вставки и удаления.

**Алгоритм вставки с последующей балансировкой для АВЛ-дерева**

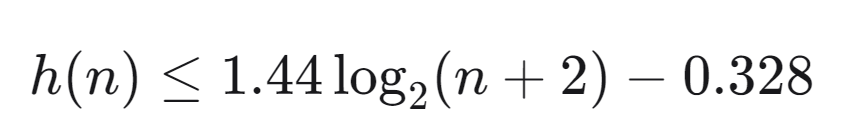
1. Вставка узла:
   * Вставка узла в АВЛ-дерево происходит аналогично вставке в обычное двоичное дерево поиска.
   * Найдите подходящее место для нового узла и вставьте его.
2. Обновление высот:
   * После вставки узла необходимо обновить высоты всех узлов, начиная с родителя вставленного узла и до корня дерева.
3. Проверка баланса:
   * Для каждого узла, начиная с родителя вставленного узла и до корня, проверяйте фактор баланса.
   * Если фактор баланса становится ±2, необходимо выполнить балансировку.
4. Балансировка:
   * В зависимости от значения фактора баланса и положения вставленного узла, выполняются одно из четырех возможных вращений:
     + Правое вращение (Right Rotate): Если фактор баланса узла равен +2 и фактор баланса его левого поддерева равен +1.
     + Левое вращение (Left Rotate): Если фактор баланса узла равен -2 и фактор баланса его правого поддерева равен -1.
     + Лево-правое вращение (Left-Right Rotate): Если фактор баланса узла равен +2 и фактор баланса его левого поддерева равен -1.
     + Право-левое вращение (Right-Left Rotate): Если фактор баланса узла равен -2 и фактор баланса его правого поддерева равен +1.

**Алгоритм удаления с последующей балансировкой для АВЛ-дерева**

1. Удаление узла:
   * Удаление узла в АВЛ-дереве происходит аналогично удалению в обычном двоичном дереве поиска.
   * Найдите узел для удаления и удалите его, заменяя его подходящим узлом (например, минимальным узлом в правом поддереве).
2. Обновление высот:
   * После удаления узла необходимо обновить высоты всех узлов, начиная с родителя удаленного узла и до корня дерева.
3. Проверка баланса:
   * Для каждого узла, начиная с родителя удаленного узла и до корня, проверяйте фактор баланса.
   * Если фактор баланса становится ±2, необходимо выполнить балансировку.
4. Балансировка:
   * В зависимости от значения фактора баланса и положения удаленного узла, выполняются одно из четырех возможных вращений (аналогично алгоритму вставки).

**Получение верхней оценки высоты АВЛ-дерева**

Верхняя оценка высоты АВЛ-дерева может быть получена с помощью рекуррентного соотношения. Пусть h(n) — это высота АВЛ-дерева с n узлами. Тогда:



Это соотношение показывает, что высота АВЛ-дерева растет логарифмически с увеличением числа узлов, что обеспечивает эффективные операции поиска, вставки и удаления.

**Определение красно-черного дерева**

Красно-черное дерево — это самобалансирующееся двоичное дерево поиска, которое использует дополнительный бит (цвет) для каждого узла, чтобы обеспечить сбалансированность дерева. Каждый узел в красно-черном дереве имеет один из двух цветов: красный или черный. Дерево должно удовлетворять следующим свойствам:

1. Каждый узел либо красный, либо черный.
2. Корень всегда черный.
3. Все листья (NIL-узлы) черные.
4. Если узел красный, то оба его дочерних узла черные (красные узлы не могут быть соседними).
5. Любой путь от данного узла до его листьев содержит одинаковое количество черных узлов.

**Алгоритм вставки с последующей балансировкой для красно-черного дерева**

1. Вставка узла:
   * Вставка узла в красно-черное дерево происходит аналогично вставке в обычное двоичное дерево поиска.
   * Новый узел всегда вставляется как красный.
2. Восстановление свойств:
   * После вставки узла необходимо восстановить свойства красно-черного дерева. Это делается с помощью следующих шагов:
     + Перекрашивание: Если родитель нового узла черный, то дерево остается сбалансированным. Если родитель красный, то необходимо выполнить перекрашивание и/или вращения.
     + Вращения: В зависимости от конфигурации узлов, могут потребоваться левые или правые вращения для восстановления свойств.
3. Пример вращений:
   * Левое вращение (Left Rotate): Выполняется, если правый ребенок узла красный, а левый ребенок черный.
   * Правое вращение (Right Rotate): Выполняется, если левый ребенок узла красный, а правый ребенок черный.

**Алгоритм удаления с последующей балансировкой для красно-черного дерева**

1. Удаление узла:
   * Удаление узла в красно-черном дереве происходит аналогично удалению в обычном двоичном дереве поиска.
   * Найдите узел для удаления и удалите его, заменяя его подходящим узлом (например, минимальным узлом в правом поддереве).
2. Восстановление свойств:
   * После удаления узла необходимо восстановить свойства красно-черного дерева. Это делается с помощью следующих шагов:
     + Перекрашивание: Если удаленный узел был черным, то необходимо выполнить перекрашивание и/или вращения для восстановления свойств.
     + Вращения: В зависимости от конфигурации узлов, могут потребоваться левые или правые вращения для восстановления свойств.
3. Пример вращений:
   * Левое вращение (Left Rotate): Выполняется, если правый ребенок узла черный, а левый ребенок красный.
   * Правое вращение (Right Rotate): Выполняется, если левый ребенок узла черный, а правый ребенок красный.

**Получение верхней оценки высоты красно-черного дерева**

Верхняя оценка высоты красно-черного дерева может быть получена с помощью анализа свойств дерева. Пусть h(n) — это высота красно-черного дерева с n узлами. Тогда:



Это соотношение показывает, что высота красно-черного дерева растет логарифмически с увеличением числа узлов, что обеспечивает эффективные операции поиска, вставки и удаления.

**Практическая часть**

Как работает код бинарного дерева?

Структура Node представляет узел дерева. Каждый узел содержит ключ (значение), а также указатели на левого и правого потомков. Класс BST представляет бинарное дерево поиска. Он содержит методы для вставки элементов, вычисления высоты дерева и различных обходов дерева (прямой, симметричный, обратный и по уровням).

В функции main выполняются следующие действия:

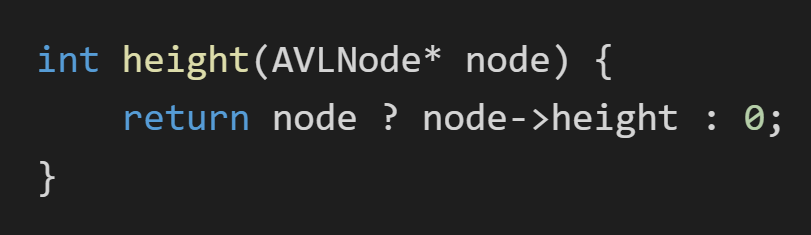
1. Инициализация генератора случайных чисел.
2. Создание вектора sizes, содержащего размеры деревьев, которые будут протестированы.
3. Для каждого размера из sizes выполняется 10 попыток (numTrials) создания дерева с случайными ключами и вычисления его высоты. Средняя высота для каждого размера выводится на экран.
4. Создается дерево tree и в него вставляются числа от 2 до 101.
5. Выполняются различные обходы дерева (симметричный, прямой, обратный и по уровням) и результаты выводятся на экран.

Как работает код AVL дерева?

Структура AVLNode представляет узел AVL-дерева. Каждый узел содержит ключ, указатели на левого и правого потомков, а также высоту поддерева с этим узлом в качестве корня. Класс AVLTree представляет AVL-дерево и содержит методы для вставки узлов, вычисления высоты дерева, а также различных обходов дерева (прямой, симметричный, обратный и по уровням).

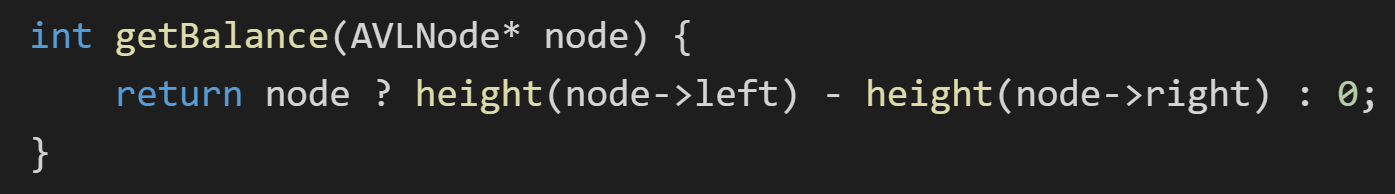
**Основные методы класса AVLTree**

Высота узла:



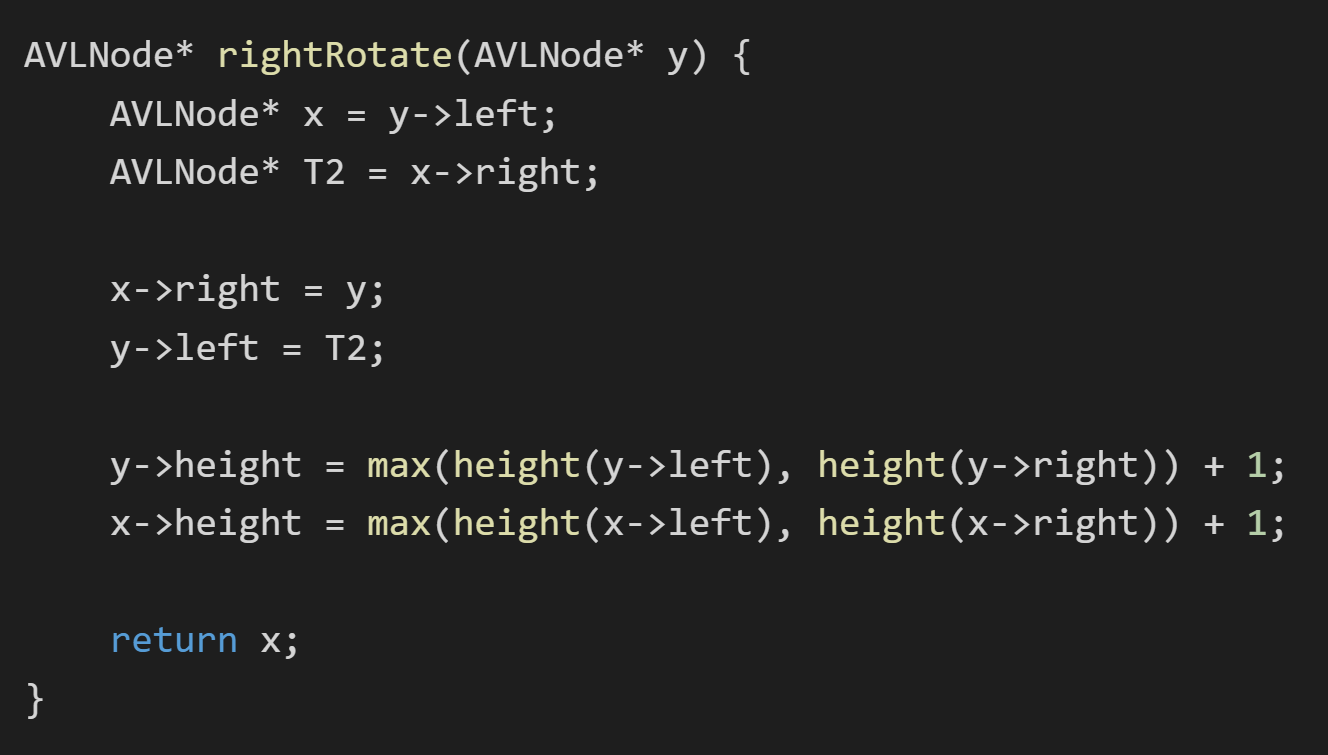
Возвращает высоту узла. Если узел **nullptr**, возвращает 0.

Баланс узла:



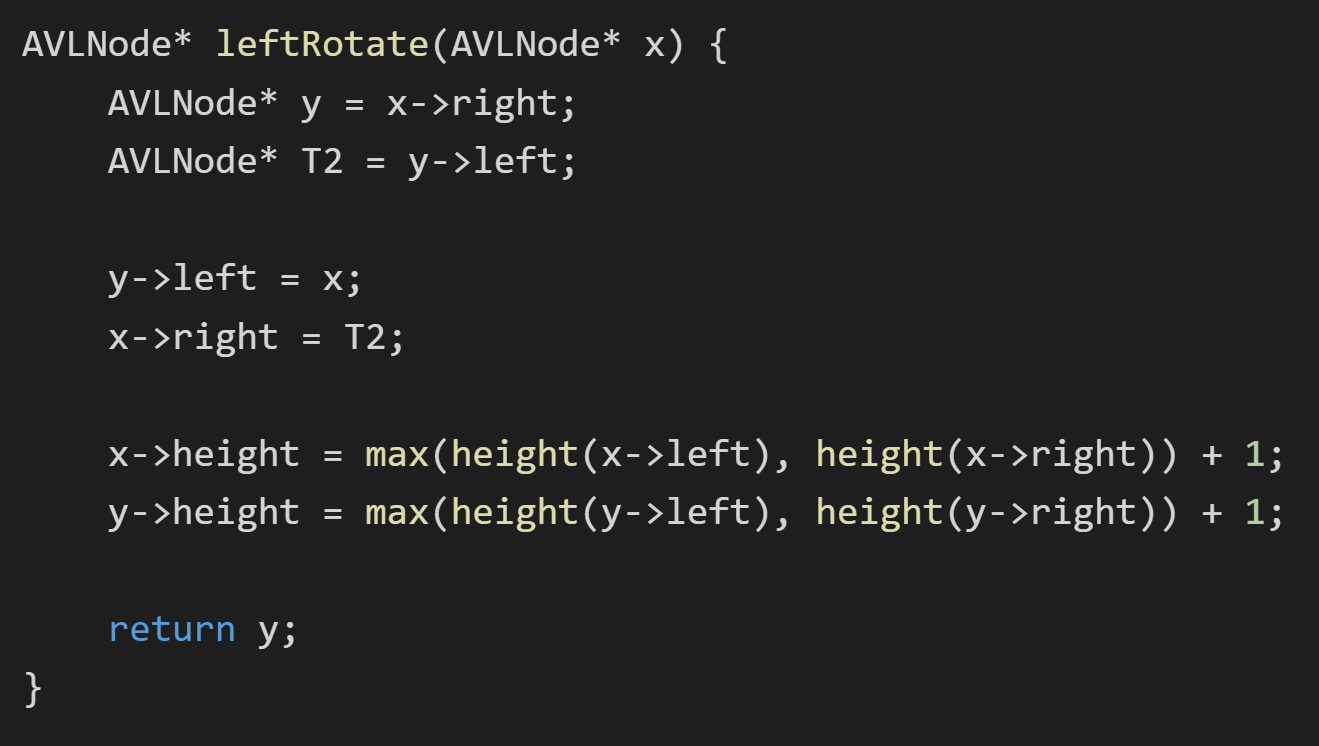
Возвращает баланс узла, который определяется как разность высот левого и правого поддеревьев.

Правый поворот:



Выполняет правый поворот вокруг узла **y**.

Левый поворот:



Выполняет левый поворот вокруг узла **x**.

Как работает код красно-чёрного дерева?

Перечисление Color определяет два возможных цвета узлов в красно-черном дереве: красный и черный. Структура Node представляет узел красно-черного дерева. Каждый узел содержит данные, цвет, указатели на левого и правого потомков, а также указатель на родительский узел.

В функции main выполняются следующие действия:

1. Создается красно-черное дерево rbt.
2. В дерево вставляются числа от 2 до 101.
3. Для каждого вставленного числа выводится высота дерева.
4. Выполняются различные обходы дерева (симметричный, прямой, обратный и по уровням) и результаты выводятся на экран.

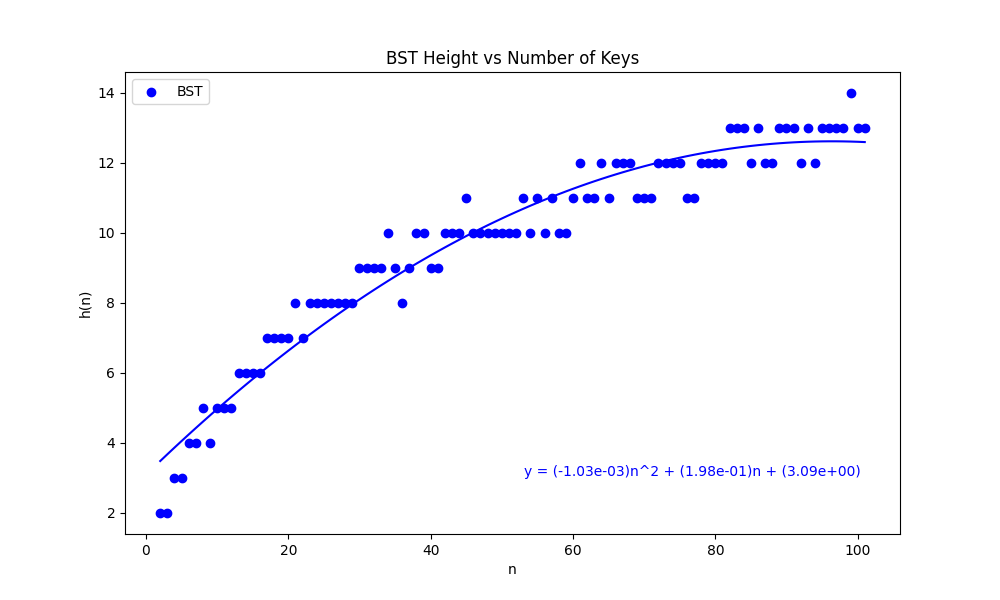


Рис. 1.1 – Бинарное дерево поиска (версия без теоретической оценки)

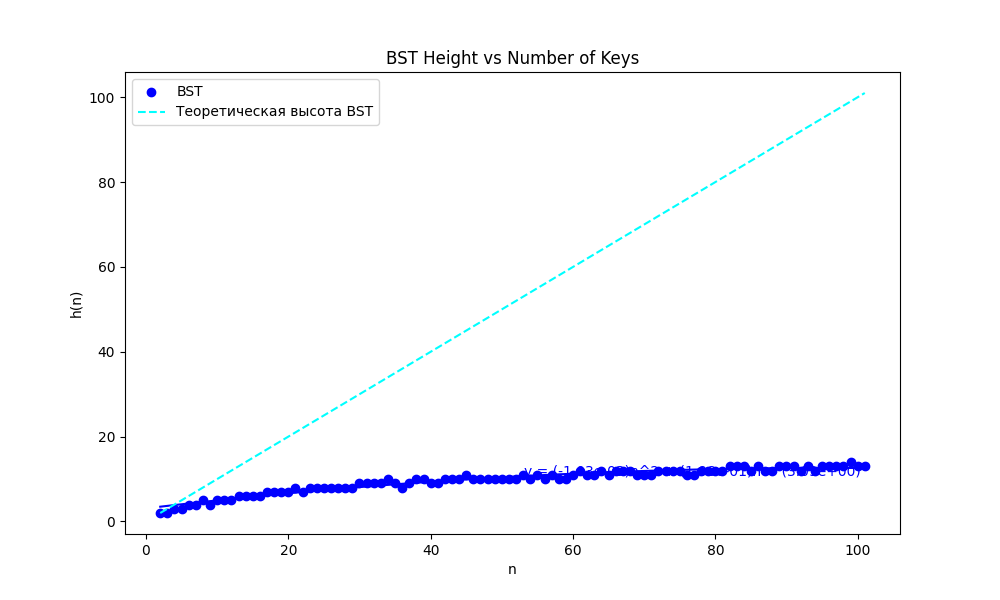


Рис. 1.2 – Бинарное дерево поиска (Худший случай O(n))

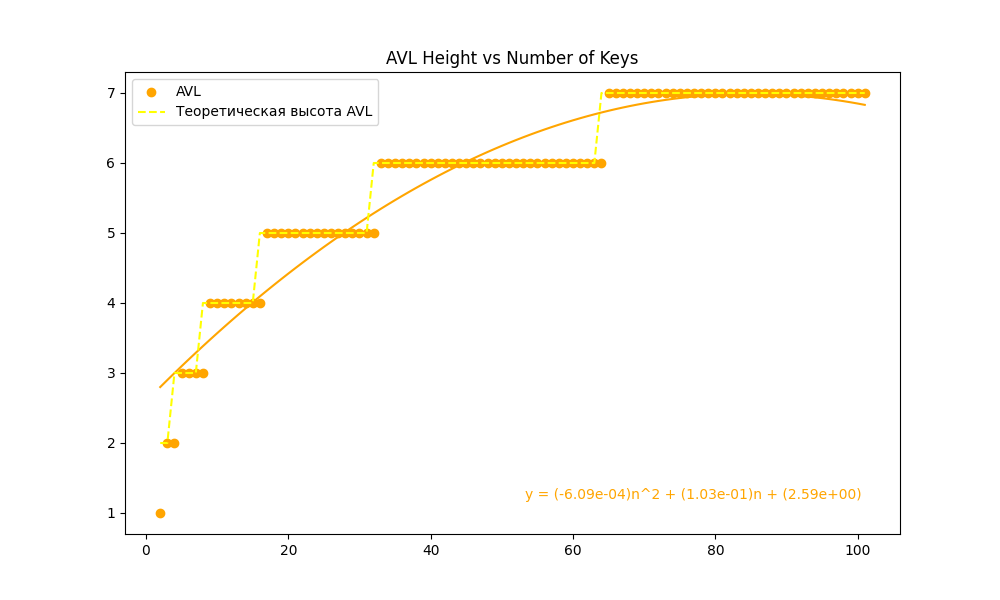


Рис. 2 – Самобалансирующееся AVL дерево

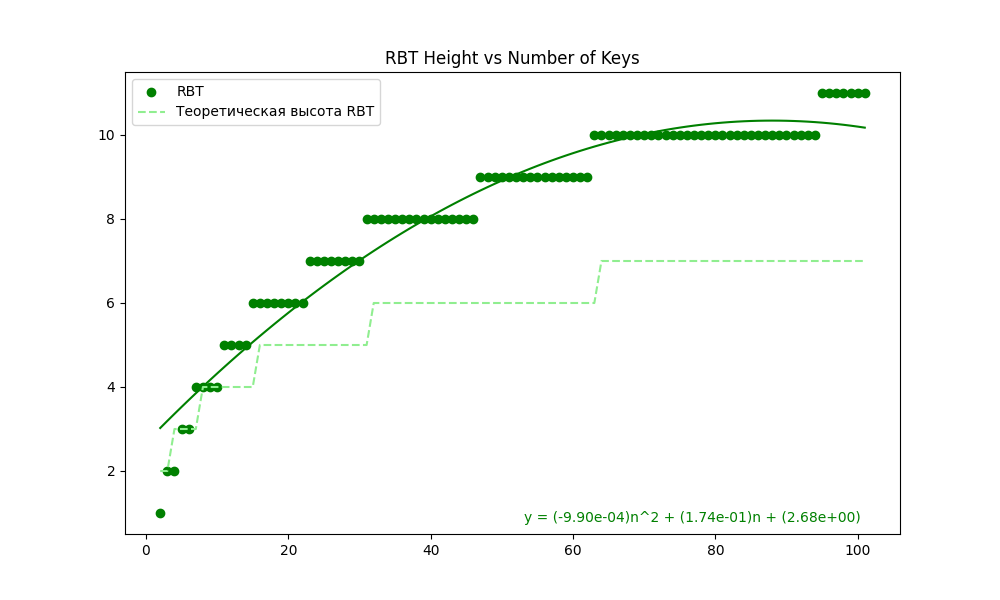


Рис. 3 – Красно-чёрное дерево

**AVL-дерево (AVL)**

Теоретическая оценка высоты: Для AVL-дерева высота h*h* ограничена следующим образом: h≤1.44log⁡2(n+2)−0.328*h*≤1.44log2​(*n*+2)−0.328 где n*n* — количество узлов в дереве.

Экспериментальные данные: На графике видно, что высота AVL-дерева увеличивается логарифмически с увеличением количества ключей. Экспериментальные данные (оранжевые точки) лежат близко к теоретической кривой (оранжевая пунктирная линия).

**Бинарное дерево поиска (BST)**

Теоретическая оценка высоты: Для бинарного дерева поиска высота h*h* в худшем случае (если дерево вырождается в линейную структуру) равна n−1*n*−1, где n*n* — количество узлов. В среднем случае высота BST логарифмическая: h≈log⁡2(n)*h*≈log2​(*n*)

Экспериментальные данные: На графике видно, что высота BST (синие точки) значительно ниже теоретической оценки в худшем случае (синяя пунктирная линия). Это указывает на то, что в данном эксперименте дерево не вырождалось в линейную структуру и его высота ближе к логарифмической.

**Красно-черное дерево (RBT)**

Теоретическая оценка высоты: Для красно-черного дерева высота h*h* ограничена следующим образом: h≤2log⁡2(n+1)*h*≤2log2​(*n*+1) где n*n* — количество узлов в дереве.

Экспериментальные данные: На графике видно, что высота RBT (зеленые точки) увеличивается логарифмически с увеличением количества ключей. Экспериментальные данные лежат близко к теоретической кривой (зеленая пунктирная линия).

**Вывод о проделанной лабораторной работе**

В ходе выполнения лабораторной работы были рассмотрены и проанализированы различные виды деревьев, такие как бинарное дерево поиска, AVL дерево и красно-чёрное дерево.

1. AVL-дерево: Экспериментальные данные соответствуют теоретической оценке. Высота дерева увеличивается логарифмически с увеличением количества ключей.
2. BST-дерево: Экспериментальные данные значительно ниже теоретической оценки в худшем случае, что указывает на то, что дерево не вырождалось в линейную структуру. Высота дерева ближе к логарифмической.
3. RBT-дерево: Экспериментальные данные соответствуют теоретической оценке. Высота дерева увеличивается логарифмически с увеличением количества ключей.

Таким образом, экспериментальные данные подтверждают теоретические оценки высоты для AVL-деревьев и красно-черных деревьев. Для бинарных деревьев поиска экспериментальные данные указывают на то, что дерево не вырождалось в линейную структуру и его высота ближе к логарифмической.

**Ссылка на GIT:**

[**https://github.com/millkun/AiSD.git**](https://vk.com/away.php?to=https%3A%2F%2Fgithub.com%2Fmillkun%2FAiSD.git&cc_key=)