Московский авиационный институт

(Национальный исследовательский университет)

Факультет №3:

*Системы управления, информатика и электроэнергетика*.

Кафедра 304.

«Алгоритмы и обработка данных»

Отчёт по лабораторной работе №2

Тема

**«Бинарные деревья поиска»**

Группа: М3О-225Б-23

Выполнили:

Евстифеев А.С.

Никулин Г.С.

Приняла:

*Дмитриева Е.А.*

Москва 2023 г.

Оглавление

[Задание 2](#__RefHeading___Toc2072_2887589161)

[Код программы 3](#__RefHeading___Toc2083_2887589161)

[Результат работы программы 8](#__RefHeading___Toc2085_2887589161)

[Вывод 10](#__RefHeading___Toc2087_2887589161)

# Задание

1. Случайным образом сгенерировать массив размерностью 20-25 элементов, повторные значения не допустимы.
2. Реализовать функции вставки, поиска, удаления узла, обхода дерева, вывода дерева на экран, нахождения высоты дерева и количества узлов.
3. Реализовать дополнительно функцию в соответствии с вариантом: *T* – тип ключей, *D* – диапазон изменения значений ключей.
4. Для набора значений из пункта 1 построить рандомизированное дерево, сравнить высоты бинарного и рандомизированного дерева.

1 вариант

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | **int** | [100; 200] | Подсчет суммы длин путей от корня до каждого из узлов, содержащих четные числа |

# Код программы

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <algorithm>

const int ARRAY\_SIZE = 25; // Number of elements in the array

const int MIN\_VALUE = 100; // Minimum value in the range

const int MAX\_VALUE = 200; // Maximum value in the range

// Structure for binary tree node

struct TreeNode {

int value; // Value

int priority; // For randomized tree

TreeNode\* leftChild; // Pointer to the left child

TreeNode\* rightChild;// Pointer to the right child

TreeNode(int val) : value(val), priority(rand() % 100), leftChild(nullptr), rightChild(nullptr) {}

};

bool isValueInArray(int array[], int size, int value);

void generateUniqueArray(int array[], int size);

TreeNode\* addNode(TreeNode\* root, int val);

TreeNode\* addRandomizedNode(TreeNode\* root, int val);

TreeNode\* rotateRight(TreeNode\* root);

TreeNode\* rotateLeft(TreeNode\* root);

TreeNode\* findNode(TreeNode\* root, int val);

TreeNode\* deleteNode(TreeNode\* root, int val);

int calculateHeight(TreeNode\* root);

int countTotalNodes(TreeNode\* root);

int calculateEvenPathsSum(TreeNode\* root, int pathLength = 0);

void displayTree(TreeNode\* root, int space = 0, int indentation = 4);

void preOrderTraversal(TreeNode\* root);

void inOrderTraversal(TreeNode\* root);

void postOrderTraversal(TreeNode\* root);

int main() {

srand(time(0));

int randomArray[ARRAY\_SIZE];

generateUniqueArray(randomArray, ARRAY\_SIZE);

TreeNode\* bstRoot = nullptr;

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

bstRoot = addNode(bstRoot, randomArray[i]);

std::cout << "Binary Search Tree (BST):\n";

displayTree(bstRoot);

std::cout << "\n\nHeight of BST: " << calculateHeight(bstRoot) << "\n";

std::cout << "Total Amount of Nodes: " << countTotalNodes(bstRoot)<< "\n\n\n";

if(deleteNode(bstRoot, randomArray[5]))

printf("Node %d deleted!", randomArray[5]);

std::cout << "\nBinary Search Tree after deliting:\n";

displayTree(bstRoot);

std::cout<<"\nSearching element "<< randomArray[7]<<"...";

if(findNode(bstRoot, randomArray[7]))

printf("\nSuccess!");

else

printf("\nNo node with same value found!");

std::cout<<"\nSearching element "<< randomArray[5]<<"...";

if(findNode(bstRoot, randomArray[5]))

printf("\nSuccess!");

else

printf("\nNo node with same value found!");

std::cout << "\n\n\nSum of even path lengths in BST: "

<< calculateEvenPathsSum(bstRoot) << "\n";

std::cout << "\nTree Traversals:\n";

std::cout << "Pre-order: ";

preOrderTraversal(bstRoot);

std::cout << "\nIn-order: ";

inOrderTraversal(bstRoot);

std::cout << "\nPost-order: ";

postOrderTraversal(bstRoot);

std::cout << "\n";

TreeNode\* randomRoot = nullptr;

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

randomRoot = addRandomizedNode(randomRoot, randomArray[i]);

}

std::cout << "\nRandomized Binary Search Tree:\n";

displayTree(randomRoot);

std::cout << "\n\nHeight of Randomized BST: " << calculateHeight(randomRoot) << "\n";

return 0;

}

// Checks if a value is already in the array

bool isValueInArray(int array[], int size, int value) {

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

if (array[i] == value) {

return true; // Value found in the array

}

}

return false; // Value not found

}

// Generates a unique array of random integers

void generateUniqueArray(int array[], int size) {

srand(time(0)); // Seed for random number generation

int count = 0; // Number of unique values added

while (count < size) {

int randomNum = MIN\_VALUE + rand() % (MAX\_VALUE - MIN\_VALUE + 1);

if (!isValueInArray(array, count, randomNum)) { // Ensure the value is unique

array[count++] = randomNum; // Add the unique value to the array

}

}

}

// Inserts a value into a binary search tree (BST)

TreeNode\* addNode(TreeNode\* root, int val) {

if (!root) return new TreeNode(val); // Create a new node if tree is empty

if (val < root->value) {

root->leftChild = addNode(root->leftChild, val); // Insert in the left subtree

} else if (val > root->value) {

root->rightChild = addNode(root->rightChild, val); // Insert in the right subtree

}

return root; // Return the updated tree

}

// Adds a node to the tree using a randomized insertion method

TreeNode\* addRandomizedNode(TreeNode\* root, int value) {

if (!root) return new TreeNode(value);

if (value < root->value) {

root->leftChild = addRandomizedNode(root->leftChild, value);

if (root->leftChild->priority > root->priority)

root = rotateRight(root);

} else {

root->rightChild = addRandomizedNode(root->rightChild, value);

if (root->rightChild->priority > root->priority)

root = rotateLeft(root);

}

return root;

}

TreeNode\* rotateRight(TreeNode\* root) {

if (!root || !root->leftChild) return root;

TreeNode\* newRoot = root->leftChild;

root->leftChild = newRoot->rightChild;

newRoot->rightChild = root;

return newRoot;

}

TreeNode\* rotateLeft(TreeNode\* root) {

if (!root || !root->rightChild) return root;

TreeNode\* newRoot = root->rightChild;

root->rightChild = newRoot->leftChild;

newRoot->leftChild = root;

return newRoot;

}

TreeNode\* findNode(TreeNode\* root, int val) {

if (!root || root->value == val) return root; // Found the node or reached a leaf

if (val < root->value) {

return findNode(root->leftChild, val); // Search in the left subtree

}

return findNode(root->rightChild, val); // Search in the right subtree

}

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

if (!root) return nullptr; // Value not found

if (val < root->value) {

root->leftChild = deleteNode(root->leftChild, val); // Delete in the left subtree

} else if (val > root->value) {

root->rightChild = deleteNode(root->rightChild, val); // Delete in the right subtree

} else {

// Node with only one child or no child

if (!root->leftChild) {

TreeNode\* temp = root->rightChild; // Replace with right child

delete root;

return temp;

} else if (!root->rightChild) {

TreeNode\* temp = root->leftChild; // Replace with left child

delete root;

return temp;

}

TreeNode\* temp = root->rightChild;

while (temp->leftChild) temp = temp->leftChild; // Get the smallest value

root->value = temp->value; // Replace value

root->rightChild = deleteNode(root->rightChild, temp->value); // Delete successor

}

return root; // Return updated tree

}

int calculateHeight(TreeNode\* root) {

if (!root) return 0; // Base case: empty tree has height 0

return 1 + std::max(calculateHeight(root->leftChild), calculateHeight(root->rightChild));

}

// Counts the total number of nodes in the tree

int countTotalNodes(TreeNode\* root) {

if (!root) return 0; // no nodes

return 1 + countTotalNodes(root->leftChild) + countTotalNodes(root->rightChild);

}

// Calculates the sum of path lengths for even-valued nodes

int calculateEvenPathsSum(TreeNode\* root, int pathLength) {

if (!root) return 0; // Base case: null node

int sum = 0;

if (root->value % 2 == 0) {

sum += pathLength; // Add path length if node value is even

}

sum += calculateEvenPathsSum(root->leftChild, pathLength + 1); // Left subtree

sum += calculateEvenPathsSum(root->rightChild, pathLength + 1); // Right subtree

return sum; // Total sum

}

// Displays the tree structure with indentation

void displayTree(TreeNode\* root, int space, int indentation) {

if (!root) return; // Base case: empty tree

space += indentation; // Increase the indentation level

displayTree(root->rightChild, space, indentation); // Print the right subtree

std::cout << std::endl; // Newline before printing the current node

for (int i = indentation; i < space; i++) std::cout << " "; // Add spaces

std::cout << root->value; // Print the node value

displayTree(root->leftChild, space, indentation); // Print the left subtree

}

// Traverses the tree in pre-order (root, left, right)

void preOrderTraversal(TreeNode\* root) {

if (!root) return;

std::cout << root->value << " ";

preOrderTraversal(root->leftChild);

preOrderTraversal(root->rightChild);

}

// Traverses the tree in in-order (left, root, right)

void inOrderTraversal(TreeNode\* root) {

if (!root) return;

inOrderTraversal(root->leftChild);

std::cout << root->value << " ";

inOrderTraversal(root->rightChild);

}

// Traverses the tree in post-order (left, right, root)

void postOrderTraversal(TreeNode\* root) {

if (!root) return;

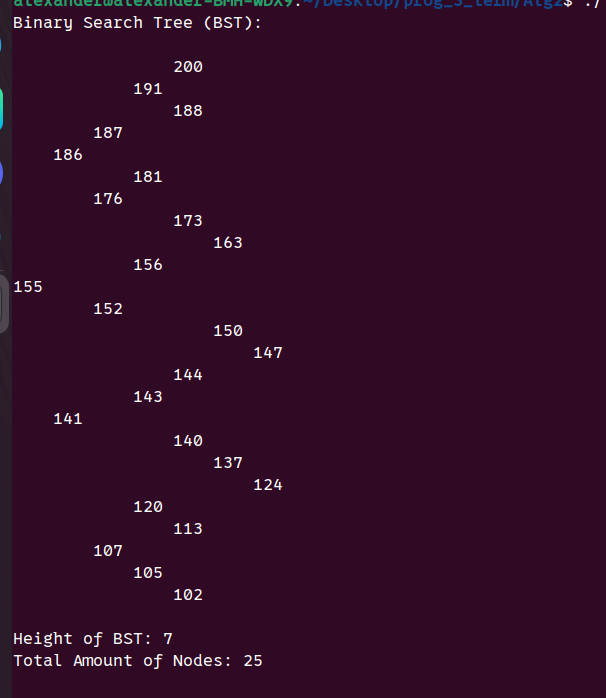
postOrderTraversal(root->leftChild);

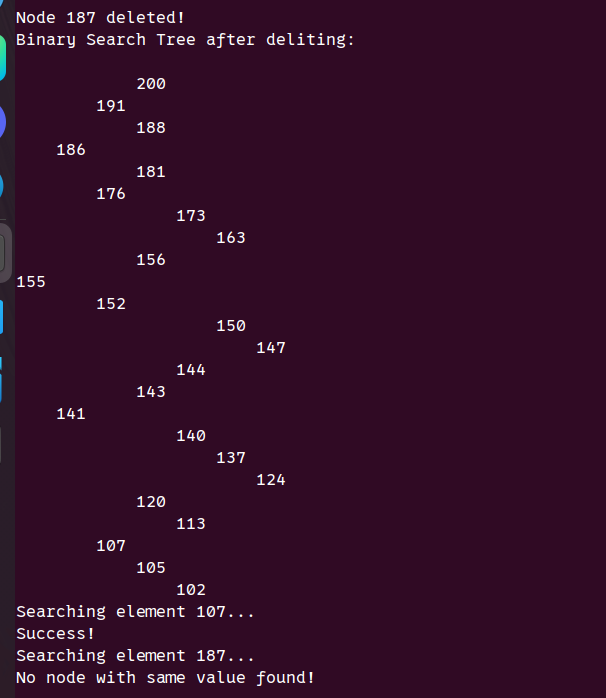
postOrderTraversal(root->rightChild);

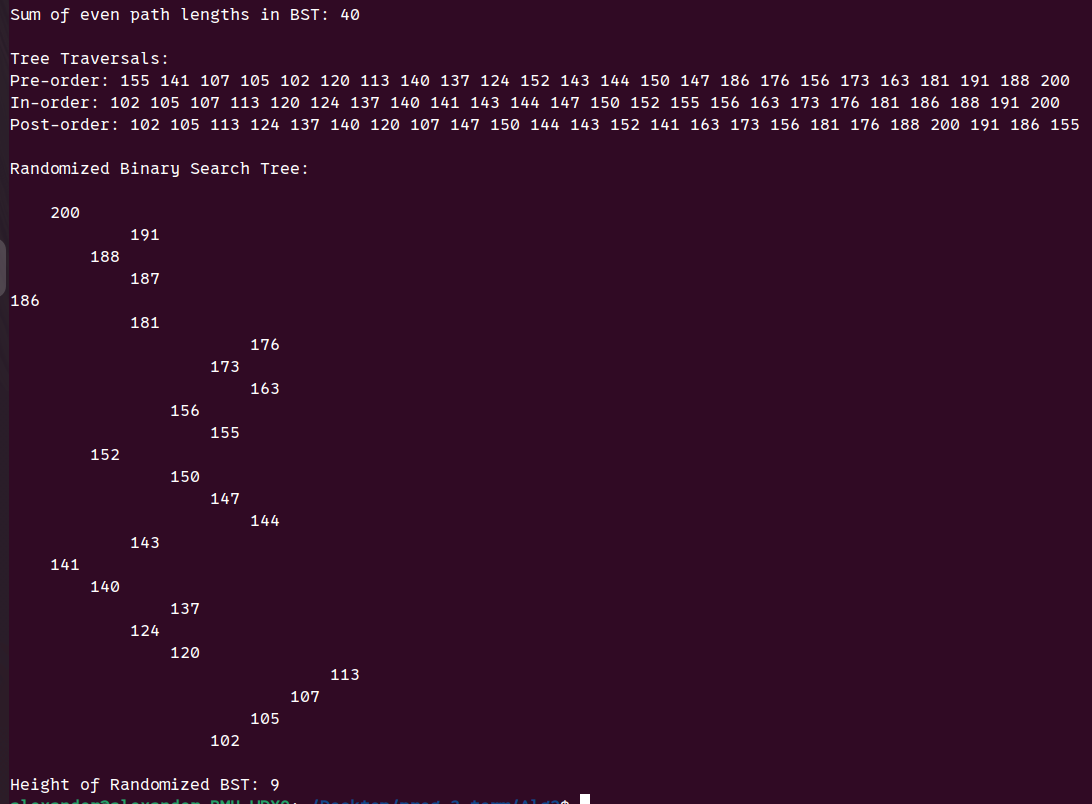
std::cout << root->value << " ";

}

# Результат работы программы

**1**

**2**

**3**

# Вывод

Реализована генерация бинарного дерева с сопутствующими функциями.