Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6

Выполнил студент группы КС-30 Джотян Давид Араикович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/DzhotyanDA\_30/tree/main/BinarySearchTree

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 14.04.2025

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# Описание задачи.

В рамках лабораторной работы необходимо изучить рандомизированное дерево

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом. Для анализа работы алгоритма понадобиться провести серии тестов:

* В одной серии тестов проводится 50 повторений
* Требуется провести серии тестов для N = 2^i элементов, при этом i от 10 до 18 включительно.

В рамках одной серии понадобится сделать следующее:

* Генерируем N случайных значений.
* Заполнить два дерева N количеством элементов в одинаковом порядке.
* Для каждого из серий тестов замерить максимальную глубину полученного деревьев.
* Для каждого дерева после заполнения провести 1000 операций вставки и замерить время.
* Для каждого дерева после заполнения провести 1000 операций удаления и замерить время.
* Для каждого дерева после заполнения провести 1000 операций поиска.
* Для каждого дерева замерить глубины всех веток дерева.

Для анализа структуры потребуется построить следующие графики:

* График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График максимальной высоты полученного дерева в зависимости от N.
* Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.
* Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов.

Задания со звездочкой = + 5 дополнительных первичных баллов:

* Аналогичная серия тестов и сравнение ее для отсортированного заранее набора данных
* Реализовать красно черное дерево и провести все те же проверки с ним.

# Описание метода/модели.

Бинарное дерево - это структура данных, в которой каждый узел имеет не более двух дочерних узлов: **левый** и **правый**.

Каждый **узел** содержит:

* значение (ключ)
* ссылку на левое поддерево (меньшие значения)
* ссылку на правое поддерево (большие значения)

АВЛ дерево - это разновидность **самобалансирующегося бинарного дерева поиска.**

**Высота дерева автоматически поддерживается сбалансированной**, чтобы операции поиска, вставки и удаления выполнялись за логарифмическое время: **O(log n)**.

Баланс поддерживается поворотами:

* Правый поворот - если левое поддерево слишком высокое (**LL**-случай).
* Левый поворот - если правое поддерево слишком высокое (**RR**-случай).
* Лево-правый поворот - если сначала выросло влево, потом вправо (**LR**-случай).
* Право-левый поворот – если сначала выросло вправо, потом влево (**RL**-случай).

# Выполнение задачи.

Файл Node.cs

namespace BinarySearchTree

{

internal class Node

{

public int? value;

public Node? left;

public Node? right;

public int height;

public Node(int value)

{

this.value = value;

left = null;

right = null;

height = 1;

}

}

}

Файл BinaryTree.cs

namespace BinarySearchTree

{

internal class BinaryTree

{

public void InorderTraversal(Node? node)

{

if (node == null)

return;

InorderTraversal(node.left);

Console.WriteLine(node.value);

InorderTraversal(node.right);

}

public Node Insert(Node? root, int value)

{

if (root == null)

{

root = new Node(value);

}

else if (value < root.value)

{

root.left = Insert(root.left, value);

}

else if (value > root.value)

{

root.right = Insert(root.right, value);

}

return root;

}

public Node Search(Node? node, int value)

{

if (node == null || node.value == value)

return node;

if (value < node.value)

return Search(node.left, value);

else

return Search(node.right, value);

}

public Node? Delete(Node? root, int value)

{

if (root == null)

return root;

if (value < root.value)

{

root.left = Delete(root.left, value);

}

else if (value > root.value)

{

root.right = Delete(root.right, value);

}

else if (root.left != null && root.right != null)

{

Node minNode = FindMinimum(root.right);

root.value = minNode.value;

root.right = Delete(root.right, (int)minNode.value);

}

else

{

if (root.left != null)

root = root.left;

else if (root.right != null)

root = root.right;

else

root = null;

}

return root;

}

public void PrintTree(Node? node, string indent = "", bool isLeft = true)

{

if (node == null)

return;

if (node.right != null)

{

PrintTree(node.right, indent + (isLeft ? "│ " : " "), false);

}

Console.Write(indent);

Console.Write(isLeft ? "└── " : "┌── ");

Console.WriteLine(node.value);

if (node.left != null)

{

PrintTree(node.left, indent + (isLeft ? " " : "│ "), true);

}

}

private Node FindMinimum(Node node)

{

if (node.left == null)

return node;

else

return FindMinimum(node.left);

}

}

}

Файл AVLTree.cs

namespace BinarySearchTree

{

internal class AVLTree

{

public Node? root;

public int Height(Node? node)

{

return node?.height ?? 0;

}

public int GetBalance(Node? node)

{

return node == null ? 0 : Height(node.left) - Height(node.right);

}

public Node? Search(Node? node, int key)

{

if (node == null || node.value == key)

return node;

if (key < node.value)

return Search(node.left, key);

else

return Search(node.right, key);

}

public Node Insert(Node? node, int key)

{

if (node == null)

return new Node(key);

if (key < node.value)

node.left = Insert(node.left, key);

else if (key > node.value)

node.right = Insert(node.right, key);

else

return node; // дубликаты не вставляем

// Обновляем высоту

node.height = 1 + Math.Max(Height(node.left), Height(node.right));

// Получаем баланс

int balance = GetBalance(node);

// Балансировка

// LL

if (balance > 1 && key < node.left!.value)

return RotateRight(node);

// RR

if (balance < -1 && key > node.right!.value)

return RotateLeft(node);

// LR

if (balance > 1 && key > node.left!.value)

{

node.left = RotateLeft(node.left);

return RotateRight(node);

}

// RL

if (balance < -1 && key < node.right!.value)

{

node.right = RotateRight(node.right);

return RotateLeft(node);

}

return node;

}

private Node RotateRight(Node y)

{

Node x = y.left!;

Node T2 = x.right!;

x.right = y;

y.left = T2;

y.height = 1 + Math.Max(Height(y.left), Height(y.right));

x.height = 1 + Math.Max(Height(x.left), Height(x.right));

return x;

}

private Node RotateLeft(Node x)

{

Node y = x.right!;

Node T2 = y.left!;

y.left = x;

x.right = T2;

x.height = 1 + Math.Max(Height(x.left), Height(x.right));

y.height = 1 + Math.Max(Height(y.left), Height(y.right));

return y;

}

public Node? Delete(Node? root, int key)

{

if (root == null)

return root;

// 1. Обычное удаление как в BST

if (key < root.value)

root.left = Delete(root.left, key);

else if (key > root.value)

root.right = Delete(root.right, key);

else

{

// Узел с одним или без детей

if (root.left == null || root.right == null)

{

Node? temp = root.left ?? root.right;

if (temp == null)

{

root = null;

}

else

{

root = temp;

}

}

else

{

// Узел с двумя детьми — находим inorder-последователя (минимум в правом поддереве)

Node minLargerNode = GetMinValueNode(root.right);

root.value = minLargerNode.value;

root.right = Delete(root.right, (int)minLargerNode.value);

}

}

// Если дерево стало пустым

if (root == null)

return null;

// 2. Обновляем высоту

root.height = 1 + Math.Max(Height(root.left), Height(root.right));

// 3. Балансировка

int balance = GetBalance(root);

// LL

if (balance > 1 && GetBalance(root.left) >= 0)

return RotateRight(root);

// LR

if (balance > 1 && GetBalance(root.left) < 0)

{

root.left = RotateLeft(root.left);

return RotateRight(root);

}

// RR

if (balance < -1 && GetBalance(root.right) <= 0)

return RotateLeft(root);

// RL

if (balance < -1 && GetBalance(root.right) > 0)

{

root.right = RotateRight(root.right);

return RotateLeft(root);

}

return root;

}

private Node GetMinValueNode(Node node)

{

Node current = node;

while (current.left != null)

current = current.left;

return current;

}

// 🖨 Печать дерева

public void PrintTree(Node? node, string indent = "", bool isLeft = true)

{

if (node == null)

return;

if (node.right != null)

PrintTree(node.right, indent + (isLeft ? "│ " : " "), false);

Console.Write(indent);

Console.Write(isLeft ? "└── " : "┌── ");

Console.WriteLine(node.value);

if (node.left != null)

PrintTree(node.left, indent + (isLeft ? " " : "│ "), true);

}

}

}

Файл Program.cs

using System.Diagnostics;

namespace BinarySearchTree

{

class Program

{

public static void Main()

{

var mainThread = new Thread(() => Test(), maxStackSize: 1024 \* 1024 \* 64);

mainThread.Start();

mainThread.Join();

}

public static void Test()

{

const int TEST\_COUNT = 10;

const int GENERATION\_COUNT = 20;

const int OPERATIONS\_COUNT = 1000;

const int LEFT\_BOUND\_RAND = 1;

const int RIGHT\_BOUND\_RAND = 10000;

Random random = new();

Stopwatch stopwatch = new();

List<double> insertBinaryRandom = new();

List<double> insertAVLRandom = new();

List<double> searchBinaryRandom = new();

List<double> searchAVLRandom = new();

List<double> searchArrayRandom = new();

List<double> deleteBinaryRandom = new();

List<double> deleteAVLRandom = new();

List<double> insertBinarySort = new();

List<double> insertAVLSort = new();

List<double> searchBinarySort = new();

List<double> searchAVLSort = new();

List<double> searchArraySort = new();

List<double> deleteBinarySort = new();

List<double> deleteAVLSort = new();

for (int i = 1; i <= TEST\_COUNT; i++)

{

int size = 1 << (10 + i);

for (int j = 0; j < GENERATION\_COUNT; j++)

{

int[] arr = new int[size];

// Заполнение массива случайными числами

if (j <= GENERATION\_COUNT / 2)

{

for (int k = 0; k < size; k++)

{

arr[k] = random.Next(LEFT\_BOUND\_RAND, RIGHT\_BOUND\_RAND);

}

}

// Заполнение массива последовательными числами

else

{

for (int k = 0; k < size; k++)

{

arr[k] = k;

}

}

// Вставка в BinaryTree

BinaryTree binaryTree = new();

Node? rootBinary = null;

stopwatch.Restart();

for (int k = 0; k < size; k++)

rootBinary = binaryTree.Insert(rootBinary, arr[k]);

stopwatch.Stop();

var insertBinaryTime = stopwatch.Elapsed.TotalMilliseconds;

// Вставка в AVLTree

AVLTree avlTree = new();

stopwatch.Restart();

for (int k = 0; k < size; k++)

avlTree.root = avlTree.Insert(avlTree.root, arr[k]);

stopwatch.Stop();

var insertAVLTime = stopwatch.Elapsed.TotalMilliseconds;

// 1000 операций поиска

int[] searchKeys = Enumerable.Range(0, OPERATIONS\_COUNT)

.Select(\_ => arr[random.Next(size)]).ToArray();

// BinaryTree поиск

stopwatch.Restart();

foreach (var key in searchKeys)

binaryTree.Search(rootBinary, key);

stopwatch.Stop();

var searchBinaryTime = stopwatch.Elapsed.TotalMilliseconds / OPERATIONS\_COUNT;

// AVLTree поиск

stopwatch.Restart();

foreach (var key in searchKeys)

avlTree.Search(avlTree.root, key);

stopwatch.Stop();

var searchAVLTime = stopwatch.Elapsed.TotalMilliseconds / OPERATIONS\_COUNT;

// Array поиск

stopwatch.Restart();

foreach (var key in searchKeys)

Array.IndexOf(arr, key);

stopwatch.Stop();

var searchArrayTime = stopwatch.Elapsed.TotalMilliseconds / OPERATIONS\_COUNT;

// Удаление

stopwatch.Restart();

foreach (var key in searchKeys)

rootBinary = binaryTree.Delete(rootBinary, key);

stopwatch.Stop();

var deleteBinaryTime = stopwatch.Elapsed.TotalMilliseconds / OPERATIONS\_COUNT;

stopwatch.Restart();

foreach (var key in searchKeys)

avlTree.root = avlTree.Delete(avlTree.root, key);

stopwatch.Stop();

var deleteAVLTime = stopwatch.Elapsed.TotalMilliseconds / OPERATIONS\_COUNT;

// Сохраняем в список (по выбору: случайные или отсортированные)

if (j < GENERATION\_COUNT / 2)

{

insertBinaryRandom.Add(insertBinaryTime);

insertAVLRandom.Add(insertAVLTime);

searchBinaryRandom.Add(searchBinaryTime);

searchAVLRandom.Add(searchAVLTime);

searchArrayRandom.Add(searchArrayTime);

deleteBinaryRandom.Add(deleteBinaryTime);

deleteAVLRandom.Add(deleteAVLTime);

}

else

{

insertBinarySort.Add(insertBinaryTime);

insertAVLSort.Add(insertAVLTime);

searchBinarySort.Add(searchBinaryTime);

searchAVLSort.Add(searchAVLTime);

searchArraySort.Add(searchArrayTime);

deleteBinarySort.Add(deleteBinaryTime);

deleteAVLSort.Add(deleteAVLTime);

}

}

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine($"Test {i} completed. Size: {size}");

Console.ForegroundColor = ConsoleColor.White;

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Random part");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine($"BinaryTree Insert Random: {insertBinaryRandom.Average()} ms");

Console.WriteLine($"AVLTree Insert Random: {insertAVLRandom.Average()} ms");

Console.WriteLine($"BinaryTree Search Random: {searchBinaryRandom.Average()} ms");

Console.WriteLine($"AVLTree Search Random: {searchAVLRandom.Average()} ms");

Console.WriteLine($"Array Search Random: {searchArrayRandom.Average()} ms");

Console.WriteLine($"BinaryTree Delete Random: {deleteBinaryRandom.Average()} ms");

Console.WriteLine($"AVLTree Delete Random: {deleteAVLRandom.Average()} ms");

Console.ForegroundColor = ConsoleColor.Yellow;

Console.WriteLine("Sorted part");

Console.ForegroundColor = ConsoleColor.White;

Console.WriteLine($"BinaryTree Insert Sorted: {insertBinarySort.Average()} ms");

Console.WriteLine($"AVLTree Insert Sorted: {insertAVLSort.Average()} ms");

Console.WriteLine($"BinaryTree Search Sorted: {searchBinarySort.Average()} ms");

Console.WriteLine($"AVLTree Search Sorted: {searchAVLSort.Average()} ms");

Console.WriteLine($"Array Search Sorted: {searchArraySort.Average()} ms");

Console.WriteLine($"BinaryTree Delete Sorted: {deleteBinarySort.Average()} ms");

Console.WriteLine($"AVLTree Delete Sorted: {deleteAVLSort.Average()} ms");

Console.WriteLine("-------------------------------------------------");

WriteToFile(i, insertBinaryRandom, insertAVLRandom, searchBinaryRandom, searchAVLRandom, searchArrayRandom, deleteBinaryRandom, deleteAVLRandom, insertBinarySort, insertAVLSort, searchBinarySort, searchAVLSort, searchArraySort, deleteBinarySort, deleteAVLSort);

}

}

public static void WriteToFile(

int testNumber,

List<double> insertBinaryRandom,

List<double> insertAVLRandom,

List<double> searchBinaryRandom,

List<double> searchAVLRandom,

List<double> searchArrayRandom,

List<double> deleteBinaryRandom,

List<double> deleteAVLRandom,

List<double> insertBinarySort,

List<double> insertAVLSort,

List<double> searchBinarySort,

List<double> searchAVLSort,

List<double> searchArraySort,

List<double> deleteBinarySort,

List<double> deleteAVLSort)

{

var path = Path.Combine(Directory.GetCurrentDirectory(), "..", "..", "..", "Measurements.txt");

using (StreamWriter sw = new StreamWriter(path, true, System.Text.Encoding.Default))

{

sw.WriteLine($"Test {testNumber}");

sw.WriteLine($"Size: {1 << (10 + testNumber)}");

sw.WriteLine("Random part");

sw.WriteLine($"BinaryTree Insert Random: {insertBinaryRandom.Average():F6} ms");

sw.WriteLine($"AVLTree Insert Random: {insertAVLRandom.Average():F6} ms");

sw.WriteLine($"BinaryTree Search Random: {searchBinaryRandom.Average():F6} ms");

sw.WriteLine($"AVLTree Search Random: {searchAVLRandom.Average():F6} ms");

sw.WriteLine($"Array Search Random: {searchArrayRandom.Average():F6} ms");

sw.WriteLine($"BinaryTree Delete Random: {deleteBinaryRandom.Average():F6} ms");

sw.WriteLine($"AVLTree Delete Random: {deleteAVLRandom.Average():F6} ms");

sw.WriteLine("Sorted part");

sw.WriteLine($"BinaryTree Insert Sorted: {insertBinarySort.Average():F6} ms");

sw.WriteLine($"AVLTree Insert Sorted: {insertAVLSort.Average():F6} ms");

sw.WriteLine($"BinaryTree Search Sorted: {searchBinarySort.Average():F6} ms");

sw.WriteLine($"AVLTree Search Sorted: {searchAVLSort.Average():F6} ms");

sw.WriteLine($"Array Search Sorted: {searchArraySort.Average():F6} ms");

sw.WriteLine($"BinaryTree Delete Sorted: {deleteBinarySort.Average():F6} ms");

sw.WriteLine($"AVLTree Delete Sorted: {deleteAVLSort.Average():F6} ms");

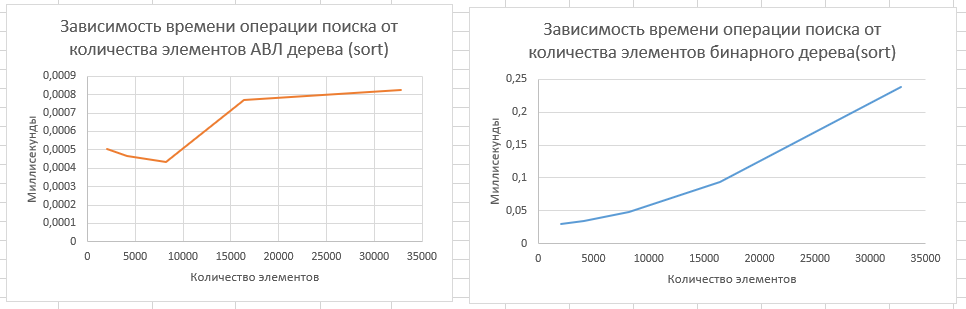
sw.WriteLine("-------------------------------------------------");

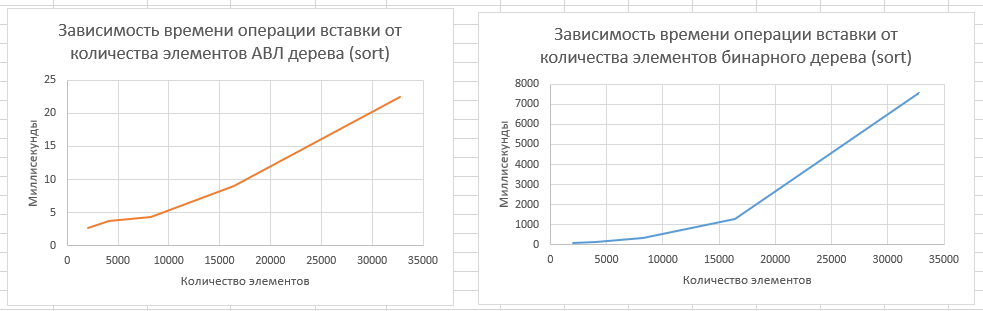
}

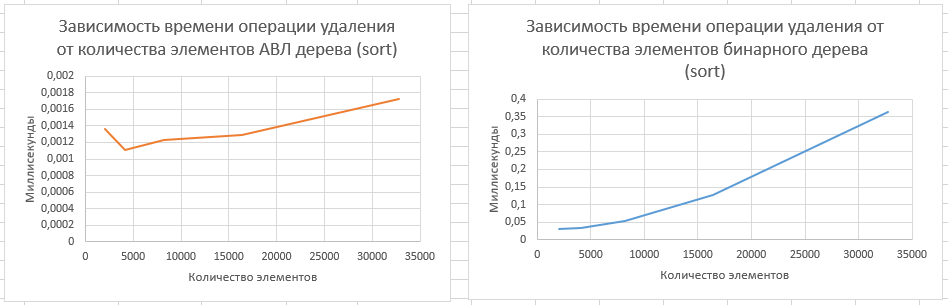
}

}

}







# Заключение.

В случайно сгенерированном массиве данных АВЛ-дерево в сравнении с бинарным показывает лучше результаты работы на операциях вставки и удаления, но не поиска.

Если же массив данных является отсортированным, то бинарное дерево просто теряет свой смысл и работать с ним очень долго. АВЛ-дерево же балансирует себя, тем самым скорость операций быстрее, чем у бинарного дерева.