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

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%206

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 26.03.2025

Оглавление

Описание задачи.

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.

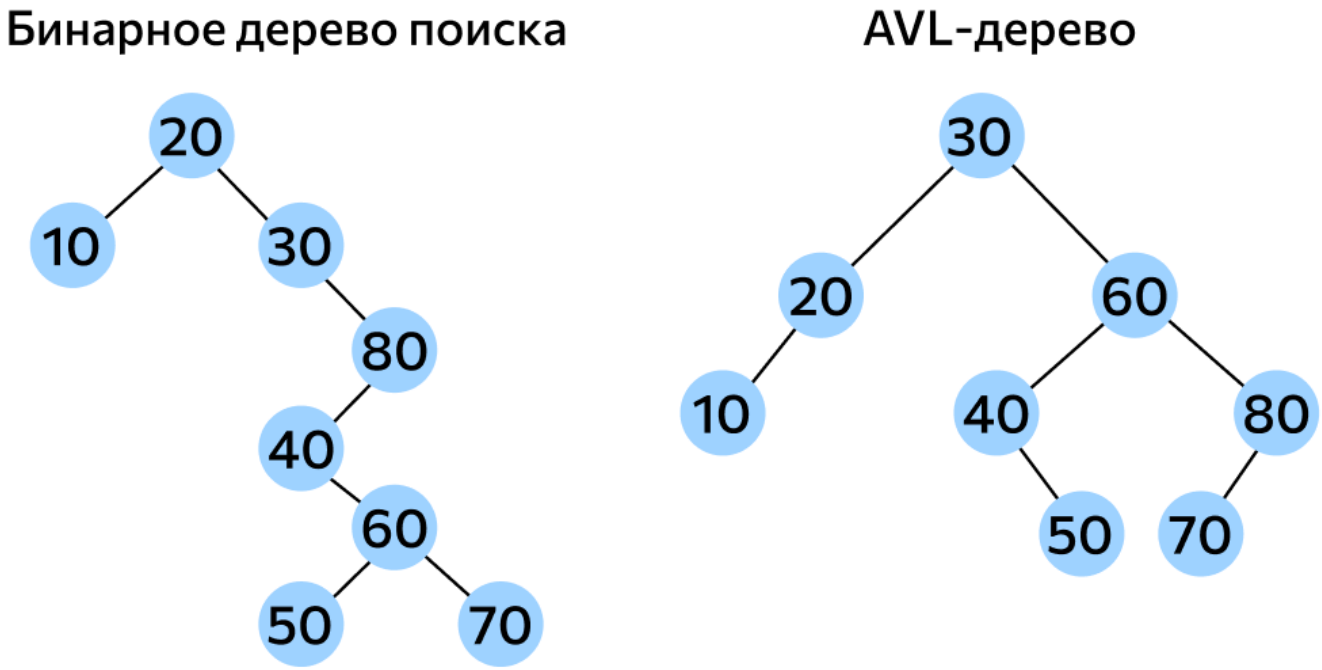
Описание бинарного дерева и AVL дерева.

Бинарное дерево — это структура данных, в которой каждый узел имеет не более двух дочерних узлов, обычно называемых левым и правым. Каждый узел состоит из:

- Данных (значение или информация, хранимая в узле),
- Указателя на левое поддерево (или null, если нет левого поддерева),
- Указателя на правое поддерево (или null, если нет правого поддерева).

Особенности бинарного дерева:

- Структура может быть сбалансированной или несбалансированной.
- Процесс поиска, вставки и удаления может быть неэффективным в несбалансированных деревьях (например, время работы может быть линейным в худшем случае).



AVL дерево (от англ. Adelson-Velsky and Landis — фамилии авторов) — это сбалансированное бинарное дерево поиска, где для каждого узла выполняется дополнительное условие: балансировка дерева. Балансировка заключается в том, чтобы разница высот левого и правого поддеревьев любого узла не превышала 1. Это условие гарантирует, что дерево будет сбалансированным и, следовательно, операции поиска, вставки и удаления будут выполняться за логарифмическое время (O(log n)).

Основные характеристики AVL-дерева:

1. Баланс-фактор каждого узла: разница между высотами левого и правого поддеревьев. Баланс-фактор узла может быть равен -1, 0 или 1.
2. После каждой операции вставки или удаления в AVL-дереве проводится перебалансировка (с помощью поворотов), если нарушено условие баланса.
3. Повороты: для восстановления баланса применяются следующие виды поворотов:

- Однообразный поворот (влево/вправо): когда несбалансированность возникает из-за одной стороны.
- Двойной поворот: когда несбалансированность возникает из-за комбинации левого/правого поддерева.

Преимущества и недостатки:

- Преимущества: быстрые операции поиска, вставки и удаления благодаря балансировке.
- Недостатки: дополнительные операции по балансировке дерева могут немного увеличить время выполнения операций вставки и удаления, особенно если дерево сильно изменяется.

В целом, AVL-дерево эффективно используется там, где необходимо обеспечить быстрый доступ к данным при поддержке динамически изменяющихся структур данных.

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <algorithm>
#include <fstream>

using namespace std;
using namespace chrono;


struct Node {
    int key;
    Node* left;
    Node* right;
    int height;

    Node(int value) : key(value), left(nullptr), right(nullptr), height(1) {}
};
```

- Структура Node
    - Вход: узел в дереве с полями: (key: значение узла, left: указатель на левое поддерево, right: – правое, height: высота узла (используется в AVL-дереве для балансировки).
    - Работа: Структура управляет хранением информации о ключе и структурах поддеревьев.
    - Выход: Конструктор инициализирует узел значением ключа и устанавливает левое и правое поддерево в nullptr.

```
class BinarySearchTree {
public:
    Node* root;

    BinarySearchTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRec(root, key);
    }

    Node* insertRec(Node* node, int key) {
        if (node == nullptr) return new Node(key);

        if (key < node->key)
            node->left = insertRec(node->left, key);
        else if (key > node->key)
```

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

    return node;
}

bool search(int key) {
    return searchRec(root, key);
}

bool searchRec(Node* node, int key) {
    if (node == nullptr) return false;
    if (key == node->key) return true;
    if (key < node->key) return searchRec(node->left, key);
    return searchRec(node->right, key);
}

void remove(int key) {
    root = removeRec(root, key);
}

Node* removeRec(Node* node, int key) {
    if (node == nullptr) return node;

    if (key < node->key)
        node->left = removeRec(node->left, key);
    else if (key > node->key)
        node->right = removeRec(node->right, key);
    else {
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }

        Node* temp = minValueNode(node->right);
        node->key = temp->key;
        node->right = removeRec(node->right, temp->key);
    }
    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr) current = current->left;
    return current;
}

void preOrder(Node* root) {
```

```
        if (root != nullptr) {
            cout << root->key << " ";
            preOrder(root->left);
            preOrder(root->right);
        }
    }

    // Очистка памяти
    void clear() {
        root = clearRec(root);
    }

    Node* clearRec(Node* node) {
        if (node == nullptr) return nullptr;
        node->left = clearRec(node->left);
        node->right = clearRec(node->right);
        delete node;
        return nullptr;
    }
};
```

- Класс BinarySearchTree — реализует обычное двоичное дерево поиска.
    - Вход: root - указатель на корень дерева.
    - Работа:
        - insert(int key): Вставка нового ключа в дерево.
            - Вход: ключ для вставки.
            - Работа: Вставляет новый узел в правильное место, сохраняя свойства дерева поиска.
            - Выход: Корень дерева может быть изменен.
        - insertRec(Node* node, int key): Рекурсивная вставка.
            - Вход: текущий узел дерева и ключ для вставки.
            - Работа: Рекурсивно находит место для нового узла и вставляет его.
            - Выход: Новое поддерево (или сам узел).
        - search(int key): Поиск ключа в дереве.
            - Вход: ключ для поиска.
            - Работа: Рекурсивно ищет ключ в дереве.
            - Выход: Возвращает true, если ключ найден, иначе false.
        - remove(int key): Удаление ключа из дерева.
            - Вход: ключ для удаления.
            - Работа: Удаляет ключ, учитывая три случая (узел с двумя детьми, с одним или без детей).
            - Выход: Возвращает новый корень дерева.

- removeRec(Node* node, int key): Рекурсивное удаление.
  - Вход: текущий узел дерева и ключ для удаления.
  - Работа: Рекурсивно находит узел для удаления и обновляет связи.
  - Выход: Новое поддерево (или измененный узел).
- clear(): Очистка памяти дерева.
  - Вход: нет.
  - Работа: Удаляет все узлы дерева.
  - Выход: дерево очищается.

```
class AVLTree {
public:
    Node* root;

    AVLTree() : root(nullptr) {}

    void insert(int key) {
        root = insertRec(root, key);
    }

    Node* insertRec(Node* node, int key) {
        if (node == nullptr) return new Node(key);

        if (key < node->key)
            node->left = insertRec(node->left, key);
        else if (key > node->key)
            node->right = insertRec(node->right, key);
        else
            return node;

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

        int balance = getBalance(node);

        if (balance > 1 && key < node->left->key)
            return rightRotate(node);

        if (balance < -1 && key > node->right->key)
            return leftRotate(node);

        if (balance > 1 && key > node->left->key) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        if (balance < -1 && key < node->right->key) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
```

```
    return node;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* 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;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* 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;
}

int height(Node* node) {
    if (node == nullptr) return 0;
    return node->height;
}

int getBalance(Node* node) {
    if (node == nullptr) return 0;
    return height(node->left) - height(node->right);
}

bool search(int key) {
    return searchRec(root, key);
}

bool searchRec(Node* node, int key) {
    if (node == nullptr) return false;
    if (key == node->key) return true;
    if (key < node->key) return searchRec(node->left, key);
    return searchRec(node->right, key);
}

void remove(int key) {
    root = removeRec(root, key);
}

Node* removeRec(Node* node, int key) {
    if (node == nullptr) return node;
```

```
        if (key < node->key)
            node->left = removeRec(node->left, key);
        else if (key > node->key)
            node->right = removeRec(node->right, key);
        else {
            if (node->left == nullptr) {
                Node* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == nullptr) {
                Node* temp = node->left;
                delete node;
                return temp;
            }

            Node* temp = minValueNode(node->right);
            node->key = temp->key;
            node->right = removeRec(node->right, temp->key);
        }
        return node;
    }

    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) current = current->left;
        return current;
    }

    void preOrder(Node* root) {
        if (root != nullptr) {
            cout << root->key << " ";
            preOrder(root->left);
            preOrder(root->right);
        }
    }

    // Очистка памяти
    void clear() {
        root = clearRec(root);
    }

    Node* clearRec(Node* node) {
        if (node == nullptr) return nullptr;
        node->left = clearRec(node->left);
        node->right = clearRec(node->right);
        delete node;
        return nullptr;
    }
};
```

- Класс AVLTree— реализует AVL-дерево — сбалансированное двоичное дерево поиска.
    - Вход: root: указатель на корень дерева.

- Работа
  - insert(int key): Вставка с балансировкой дерева.
    - Вход: ключ для вставки.
    - Работа: Вставляет новый узел и выполняет балансировку дерева (повороты).
    - Выход: Корень дерева может быть изменен.
  - insertRec(Node* node, int key): Рекурсивная вставка с балансировкой.
    - Вход: текущий узел и ключ для вставки.
    - Работа: Рекурсивно вставляет ключ и корректирует высоту и баланс дерева.
    - Выход: Новый поддерево (или сам узел).
  - rightRotate(Node* y): Правый поворот.
    - Вход: узел для поворота.
    - Работа: Выполняет правый поворот для сбалансирования дерева.
    - Выход: Новый корень поддерева после поворота.
  - leftRotate(Node* x): Левый поворот.
    - Вход: узел для поворота.
    - Работа: Выполняет левый поворот для сбалансирования дерева.
    - Выход: Новый корень поддерева после поворота.
  - height(Node* node): Получение высоты узла.
    - Вход: узел.
    - Работа: Возвращает высоту узла.
    - Выход: Высота узла.
  - getBalance(Node* node): Получение баланса узла.
    - Вход: узел.
    - Работа: Возвращает разницу высот левого и правого поддеревьев.
    - Выход: Баланс узла (положительное значение — левое поддерево выше, отрицательное — правое).
  - search(int key): Поиск ключа.
    - Вход: ключ для поиска.
    - Работа: Рекурсивно ищет ключ в дереве.
    - Выход: Возвращает true, если ключ найден, иначе false.
  - remove(int key): Удаление ключа с балансировкой.
    - Вход: ключ для удаления.
    - Работа: Удаляет ключ и восстанавливает баланс дерева.

        - Выход: Новый корень дерева.
    - removeRec(Node* node, int key): Рекурсивное удаление с балансировкой.
        - Вход: текущий узел и ключ для удаления.
        - Работа: Рекурсивно находит узел для удаления и выполняет балансировку.
        - Выход: Новый поддерево (или измененный узел).
    - clear(): Очистка памяти.
        - Вход: нет.
        - Работа: Удаляет все узлы дерева.
        - Выход: дерево очищается.

```
void generateRandomArray(vector<int>& arr, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand() % 10000; // Генерация случайных чисел
    }
}
```

- generateRandomArray(vector<int>& arr, int size):
    - Вход: массив и его размер.
    - Работа: Заполняет массив случайными числами.
    - Выход: Массив с случайными числами.

```
void generateSortedArray(vector<int>& arr, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = i;
    }
}
```

- generateSortedArray(vector<int>& arr, int size):
    - Вход: массив и его размер.
    - Работа: Заполняет массив отсортированными числами.
    - Выход: Массив с отсортированными числами.

```
bool linearSearch(const vector<int>& arr, int key) {
    for (int num : arr) {
        if (num == key) {
            return true;
        }
    }
    return false;
}
```

- linearSearch(const vector<int>& arr, int key):
    - Вход: массив и ключ для поиска.

- Работа: Выполняет линейный поиск ключа в массиве.
- Выход: true, если ключ найден, иначе false.

```
   int main() {
       srand(time(0));  // Инициализация генератора случайных чисел

       ofstream outfile("output.txt");

       for (int i = 0; i < 20; ++i) {  // 20 циклов для каждой серии
           int n = 1 << (10 + (i % 10));  // Размер массива 2^(10 + i)

           vector<int> arr(n);
           bool isRandom = i < 10;  // Первая половина серий с случайными данными

           if (isRandom) {
               generateRandomArray(arr, n);  // Заполняем случайными числами
           } else {
               generateSortedArray(arr,  n);    //  Заполняем  отсортированными
числами
           }

           auto start1 = high_resolution_clock::now();
           for (int j = 0; j < 1000; ++j) {
               linearSearch(arr, searchKey);
           }
           auto end1 = high_resolution_clock::now();
           auto  durationLinearSearch1  =  duration_cast<microseconds>(end1  -
start1);
           cout  <<  "Series  "  <<  i+1  <<  "  (Linear  Search)  -  Time:  "  <<
durationLinearSearch1.count()  <<  "  microseconds,  per  operation:  "  <<
durationLinearSearch1.count() / 1000.0 << " microseconds" << endl;
           outfile  <<  "Series  "  <<  i+1  <<  "  (Linear  Search)  -  Time:  "  <<
durationLinearSearch1.count()  <<  "  microseconds,  per  operation:  "  <<
durationLinearSearch1.count() / 1000.0 << " microseconds" << endl;


           // 1. Замер времени для Binary Search Tree

           BinarySearchTree bst;
           auto start = high_resolution_clock::now();
           for (int num : arr) {
               bst.insert(num);
           }
           auto end = high_resolution_clock::now();
           auto durationBST = duration_cast<microseconds>(end - start);
           cout  <<  "Series  "  <<  i+1  <<  "  (BST  Insert)  -  Time:  "  <<
durationBST.count() << " microseconds" << endl;
           outfile  <<  "Series  "  <<  i+1  <<  "  (BST  Insert)  -  Time:  "  <<
durationBST.count() << " microseconds" << endl;
           cout << "2." << endl;
```

```
            // 2. Замер времени для AVL Tree

            AVLTree avl;
            start = high_resolution_clock::now();
            for (int num : arr) {
                avl.insert(num);
            }
            end = high_resolution_clock::now();
            auto durationAVL = duration_cast<microseconds>(end - start);
            cout << "Series " << i+1 << " (AVL Insert) - Time: " <<
durationAVL.count() << " microseconds" << endl;
            outfile << "Series " << i+1 << " (AVL Insert) - Time: " <<
durationAVL.count() << " microseconds" << endl;

            // 3. Замер времени поиска в Binary Search Tree

            int searchKey = rand() % 10000;  // Случайный ключ для поиска
            start = high_resolution_clock::now();
            for (int j = 0; j < 1000; ++j) {
                bst.search(searchKey);
            }
            end = high_resolution_clock::now();
            auto durationBSTSearch = duration_cast<microseconds>(end - start);
            cout << "Series " << i+1 << " (BST Search) - Time: " <<
durationBSTSearch.count() << " microseconds, per operation: " <<
durationBSTSearch.count() / 1000.0 << " microseconds" << endl;
            outfile << "Series " << i+1 << " (BST Search) - Time: " <<
durationBSTSearch.count() << " microseconds, per operation: " <<
durationBSTSearch.count() / 1000.0 << " microseconds" << endl;

            // 4. Замер времени поиска в AVL Tree

            start = high_resolution_clock::now();
            for (int j = 0; j < 1000; ++j) {
                avl.search(searchKey);
            }
            end = high_resolution_clock::now();
            auto durationAVLSearch = duration_cast<microseconds>(end - start);
            cout << "Series " << i+1 << " (AVL Search) - Time: " <<
durationAVLSearch.count() << " microseconds, per operation: " <<
durationAVLSearch.count() / 1000.0 << " microseconds" << endl;
            outfile << "Series " << i+1 << " (AVL Search) - Time: " <<
durationAVLSearch.count() << " microseconds, per operation: " <<
durationAVLSearch.count() / 1000.0 << " microseconds" << endl;

            // 5. Замер времени удаления из Binary Search Tree

            start = high_resolution_clock::now();
            for (int j = 0; j < 1000; ++j) {
                bst.remove(searchKey);
            }
            end = high_resolution_clock::now();
```

```
            auto durationBSTRemove = duration_cast<microseconds>(end - start);
            cout << "Series " << i+1 << " (BST Remove) - Time: " <<
durationBSTRemove.count() << " microseconds, per operation: " <<
durationBSTRemove.count() / 1000.0 << " microseconds" << endl;
            outfile << "Series " << i+1 << " (BST Remove) - Time: " <<
durationBSTRemove.count() << " microseconds, per operation: " <<
durationBSTRemove.count() / 1000.0 << " microseconds" << endl;

            // 6. Замер времени удаления из AVL Tree

            start = high_resolution_clock::now();
            for (int j = 0; j < 1000; ++j) {
                avl.remove(searchKey);
            }
            end = high_resolution_clock::now();
            auto durationAVLRemove = duration_cast<microseconds>(end - start);
            cout << "Series " << i+1 << " (AVL Remove) - Time: " <<
durationAVLRemove.count() << " microseconds, per operation: " <<
durationAVLRemove.count() / 1000.0 << " microseconds" << endl;
            outfile << "Series " << i+1 << " (AVL Remove) - Time: " <<
durationAVLRemove.count() << " microseconds, per operation: " <<
durationAVLRemove.count() / 1000.0 << " microseconds" << endl;

            // Очищаем память

            bst.clear();
            avl.clear();
        }
        outfile.close();

        return 0;
    }
```

- Основная функция main
    - Вход: Инициализация генератора случайных чисел и потоков вывода.
    - Работа: В цикле выполняются операции с массивами данных для различных структур данных (линейный поиск, вставка, удаление и поиск в бинарном дереве и AVL-дереве).
    - Выход: Результаты замеров времени для каждого типа операции записываются в файл output.txt.

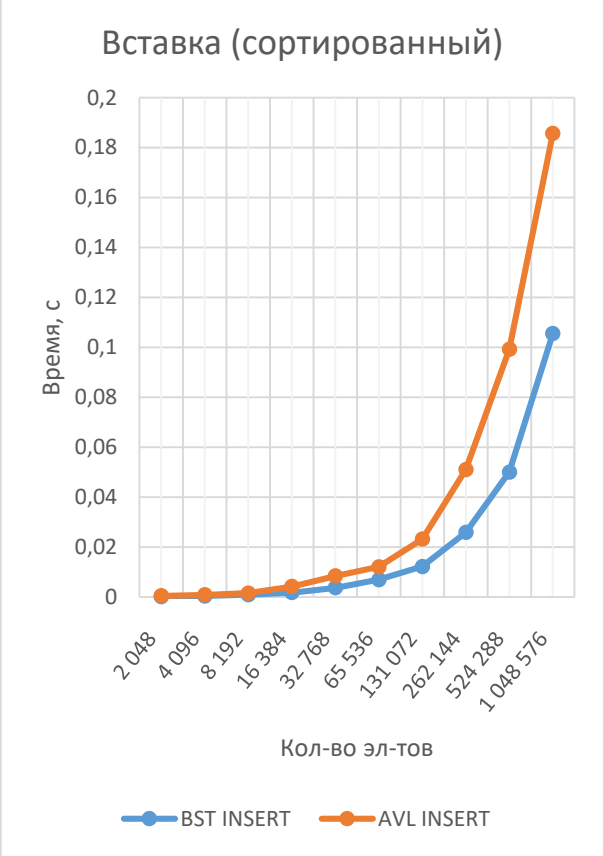
Заключение.

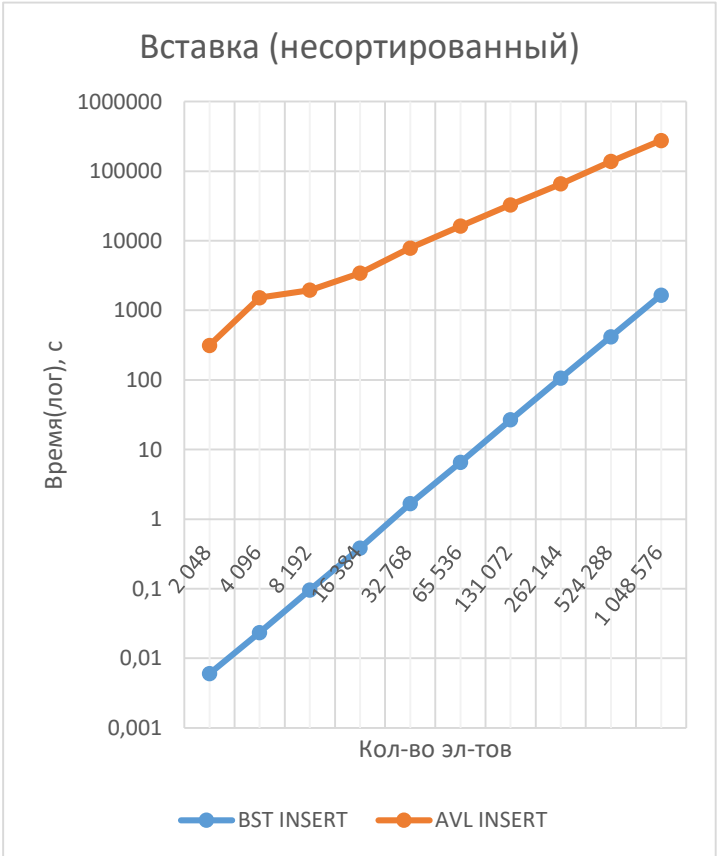
Результаты работы были представлены в виде таблицы значений.

				MKS										
SORTED	SEARCH	oper	SERIES	BST INSERT	AVL INSERT	BST SEARCH	oper	AVL SEARCH	oper	BST REMOVE	oper	AVL REMOVE	oper	volume
1	8220	8,22	1	158	438	80	0,08	77	0,077	61	0,061	57	0,057	2 048
1	15393	15,393	2	410	840	86	0,086	59	0,059	81	0,081	62	0,062	4 096
1	26314	26,314	3	898	1584	30	0,03	54	0,054	30	0,03	61	0,061	8 192
1	25137	25,137	4	1777	4150	98	0,098	66	0,066	105	0,105	81	0,081	16 384
1	8213	8,213	5	3667	8424	69	0,069	72	0,072	68	0,068	81	0,081	32 768
1	86151	86,151	6	6928	12098	55	0,055	51	0,051	51	0,051	54	0,054	65 536
1	68798	68,798	7	12225	23151	67	0,067	59	0,059	67	0,067	74	0,074	131 072
1	13320	13,32	8	25846	50982	95	0,095	62	0,062	108	0,108	73	0,073	262 144
1	17645	17,645	9	50006	99169	95	0,095	61	0,061	93	0,093	84	0,084	524 288
1	70999	70,999	10	105481	185558	79	0,079	53	0,053	96	0,096	59	0,059	1 048 576
0	7292	7,292	11	6030	314	10931	10,931	56	0,056	11968	11,968	49	0,049	2 048
0	22769	22,769	12	23559	1950	12781	12,781	39	0,039	13360	13,36	53	0,053	4 096
0	29515	29,515	13	95909	1520	43868	43,868	59	0,059	49020	49,02	56	0,056	8 192
0	60456	60,456	14	383264	3423	67840	67,84	40	0,04	76707	76,707	65	0,065	16 384
0	19843	19,843	15	1668369	7863	15277	15,277	51	0,051	16564	16,564	75	0,075	32 768
0	52320	52,32	16	6536472	16247	40219	40,219	51	0,051	43617	43,617	65	0,065	65 536
0	27302	27,302	17	26834107	32948	18948	18,948	63	0,063	20788	20,788	61	0,061	131 072
0	34137	34,137	18	106797917	65896	37171	37,171	57	0,057	40935	40,935	72	0,072	262 144
0	11060	11,06	19	417317347	138213	33116	33,116	62	0,062	36309	36,309	76	0,076	524 288
0	5135	5,135	20	1659395067	276426	23788	23,788	62	0,062	25498	25,498	80	0,08	1 048 576

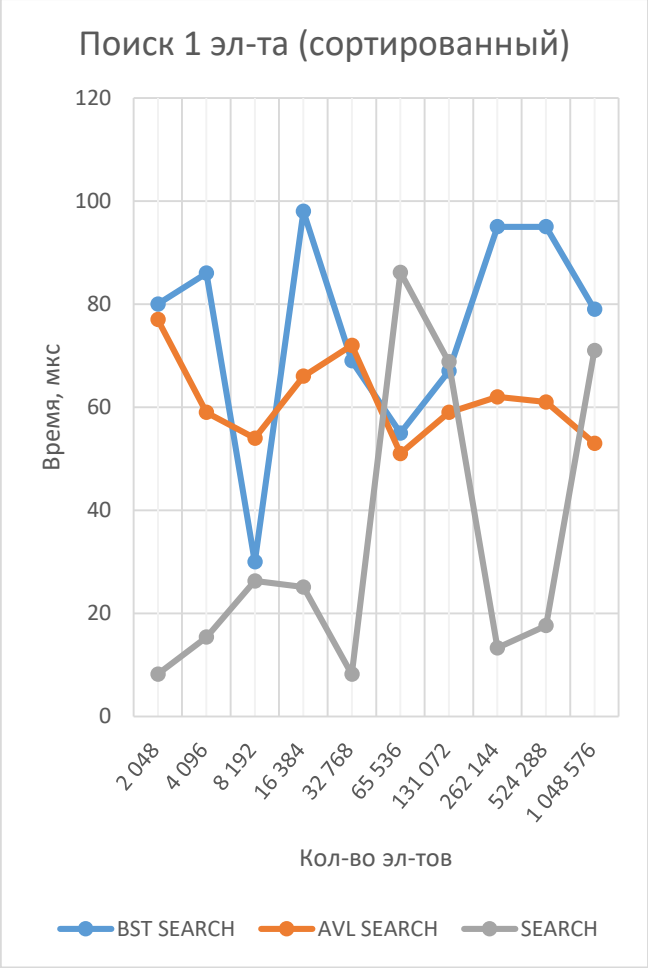
По построенной таблице можно сделать следующие выводы:

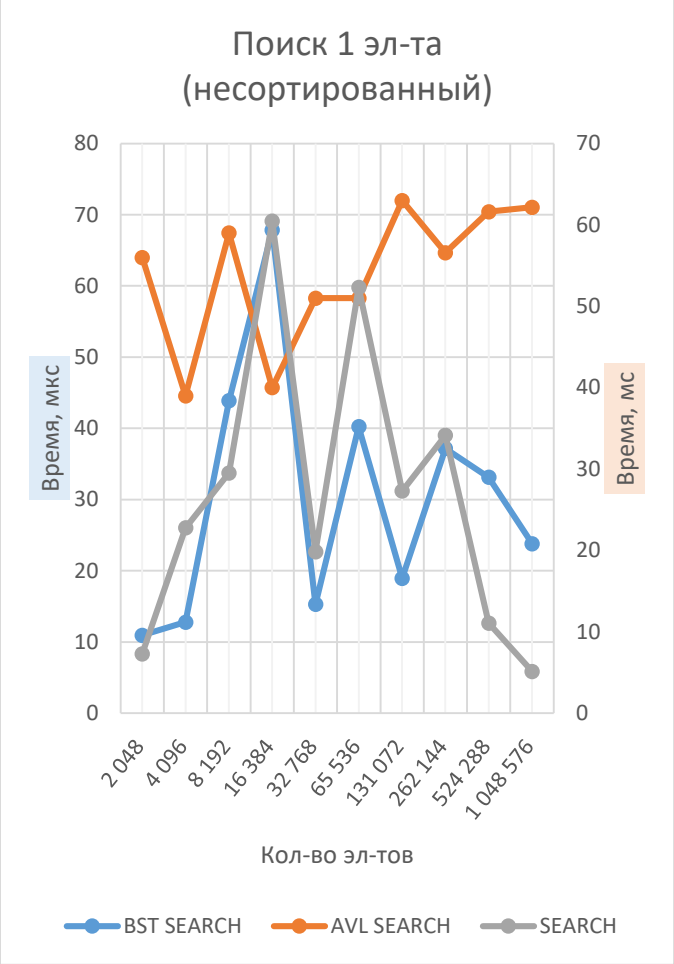
1. Сравнение производительности деревьев (BST и AVL)
    - Для каждой серии видно, что время вставки в AVL дерево всегда больше, чем в BST. Это объясняется тем, что сбалансированное дерево (AVL) требует дополнительной переработки структуры дерева (балансировки) после каждой операции вставки.



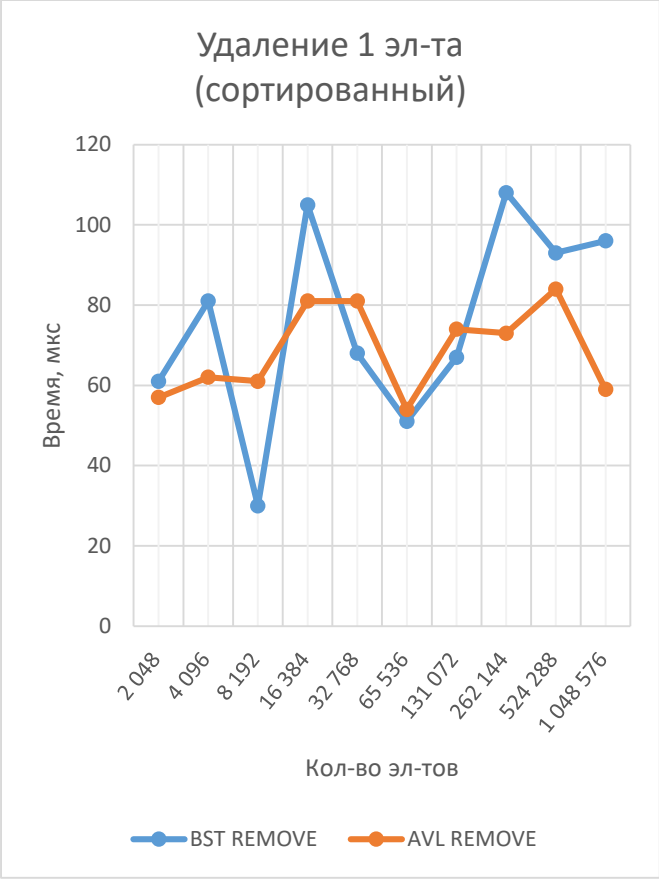


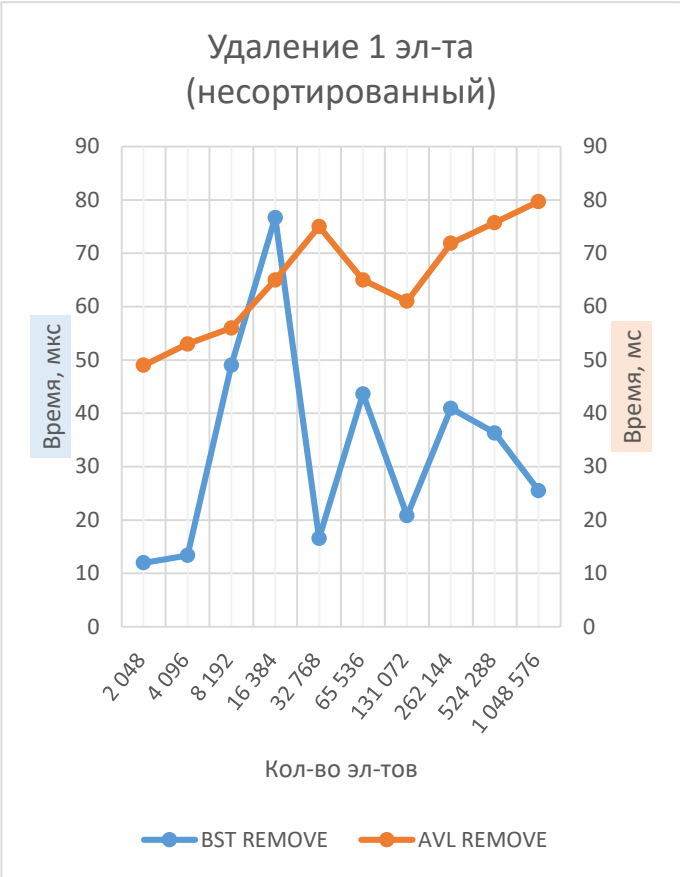
- Время поиска в AVL дереве, как правило, немного меньше, чем в BST. Это связано с тем, что балансировка деревьев в AVL гарантирует более равномерное распределение элементов, что снижает высоту дерева и уменьшает количество операций при поиске.





- Время удаления в AVL дереве также в целом больше, чем в BST, по той же причине, что балансировка требует дополнительных шагов.





2. Влияние сортировки данных
    - Когда данные отсортированы, для всех операций время в целом значительно увеличивается. Например, для поиска и удаления время в сортированных данных значительно выше. Это может быть связано с тем, что для несортированных данных дерево может быть более "сбалансированным" с точки зрения случайного распределения элементов, в то время как в сортированном случае элементы дерева могут быть более сгруппированы, что усложняет операции.
    - Для поиска: в случае отсортированных данных BST может деградировать до структуры, похожей на связанный список (в худшем случае), что увеличивает время поиска.
    - Для удаления: так как данные отсортированы, шанс того, что операция удаления затронет крайние элементы дерева, увеличивается.
3. Тренды по сериям
    - В более поздних сериях видно значительное увеличение времени операций. Это может быть связано с ростом количества элементов в деревьях (по мере увеличения числа операций вставки) или увеличением сложности балансировки в случае AVL деревьев.
4. Тенденции по времени операций
    - Время вставки AVL дерева растет гораздо быстрее, чем у BST, что объясняется дополнительными шагами балансировки.

- Время поиска AVL дерева стабилизируется быстрее, поскольку его высота остается ниже.
- Время удаления в обоих деревьях имеет схожие тенденции, но AVL дерево всегда чуть медленнее из-за необходимости поддержания баланса после удаления.

То есть BST более быстрые для операций вставки и удаления, но медленнее в поиске по сравнению с AVL деревьями, особенно при большом количестве данных. AVL деревья обеспечивают лучший поиск, но их операции вставки и удаления дороже по времени из-за необходимости поддержания сбалансированности. Сортировка данных делает операции более дорогими как для BST, так и для AVL деревьев, что подтверждает важность структуры данных для производительности алгоритмов.

Можно предположить, что если критична скорость вставки и удаления, возможно, лучше использовать BST деревья, особенно при работе с неотсортированными данными. Если приоритет – это быстрый поиск, и операции вставки и удаления не так критичны, то стоит использовать AVL деревья.