Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

№ 6 ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33 Костяева К.С.

Москва
2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ

Бинарное дерево поиска

Бинарное дерево поиска (BST, Binary Search Tree) — это структура данных, состоящая из узлов, где каждый узел имеет два поддерева (левое и правое), и удовлетворяет следующим правилам:

1. **Ключи в левом поддереве** любого узла меньше, чем ключ в самом узле.
2. **Ключи в правом поддереве** любого узла больше, чем ключ в самом узле.
3. Каждый поддерево также является бинарным деревом поиска.

Структура узла дерева

Каждый узел в бинарном дереве поиска содержит:

• **Ключ (или данные)** — это как "информация", которую хранит узел. Например, это может быть число, слово или другой элемент. Это значение используется для того, чтобы упорядочить элементы в дереве.

• **Левый указатель** — это ссылка на другой узел, который содержит **меньшее** значение, чем текущий узел. То есть все элементы в левом поддереве будут **меньше** текущего узла.

Пример: если в текущем узле хранится число **50**, то все узлы в левом поддереве будут содержать числа **меньше 50**.

• **Правый указатель** — это ссылка на узел, который содержит **большее** значение, чем текущий узел. То есть все элементы в правом поддереве будут **больше** текущего узла.

Пример: если в текущем узле хранится число **50**, то все узлы в правом поддереве будут содержать числа **больше 50**.

Основные операции в бинарном дереве поиска

1. **Вставка** (Insert) Для вставки элемента в BST:
   - Сравниваем вставляемое значение с текущим узлом.
   - Если вставляемое значение меньше текущего узла, переходим в левое поддерево.
   - Если вставляемое значение больше текущего узла, переходим в правое поддерево.
   - Повторяем процесс, пока не найдем пустую позицию для нового узла.

   Время выполнения вставки: в худшем случае (если дерево вырождено в список) — **O(n)**, в лучшем случае (если дерево сбалансировано) — **O(log n)**.

2. **Поиск** (Search) Для поиска элемента:
   - Сравниваем искомое значение с текущим узлом.
   - Если искомое значение меньше текущего узла, переходим в левое поддерево.
   - Если искомое значение больше текущего узла, переходим в правое поддерево.
   - Если искомое значение равно текущему узлу, поиск завершен.

   Время выполнения поиска: также **O(n)** в худшем случае и **O(log n)** в лучшем случае.

3. **Удаление** (Delete) Удаление узла из дерева несколько сложнее:
   - Если узел не имеет потомков (листье), просто удаляем его.

- Если узел имеет одного потомка, заменяем его на этого потомка.
- Если узел имеет двух потомков, на его место ставим **наименьший узел в правом поддереве** или **наибольший узел в левом поддереве**, а затем удаляем этот узел.

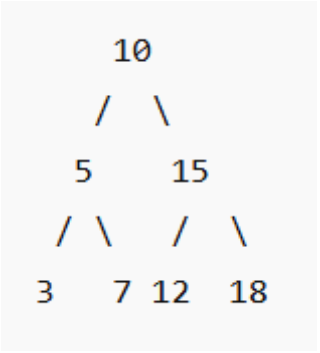
Время выполнения удаления: в худшем случае **O(n)**, в лучшем случае **O(log n)**.

Логарифм берется из того, что при каждом шаге обхода дерева количество элементов для поиска (или вставки) уменьшается в два раза.

Пример бинарного дерева поиска

Допустим, у нас есть числа: **10, 5, 15, 3, 7, 12, 18**.
Чтобы найти, например, число **7**, мы сравниваем:

- 7 < 10 → идём влево
- 7 > 5 → идём вправо
- Нашли 7



АВЛ-дерево

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Операции с AVL-деревом:

1. **Вставка**:
   - Как и в обычном бинарном дереве поиска, при вставке нового элемента мы начинаем с корня и двигаемся по дереву в зависимости от значения элемента (меньше или больше текущего узла).
   - После вставки нового узла мы проверяем балансировку каждого узла, начиная с места вставки и двигаясь вверх к корню. Если обнаруживается нарушение балансировки, выполняются необходимые повороты.
2. **Поиск**:
   - Поиск элемента в AVL-дереве работает так же, как и в обычном бинарном дереве поиска: мы начинаем с корня и на каждом шаге идём в левое или правое поддерево, сравнивая ключи.
   - Операция поиска работает за **O(log n)** времени, поскольку высота дерева всегда ограничена логарифмом от количества элементов.
3. **Удаление**:
   - При удалении элемента мы находим его, как в обычном бинарном дереве поиска.
   - После удаления элемента мы снова проверяем балансировку дерева, начиная с узла, где произошла удаление, и двигаясь вверх к корню. Если нарушение баланса обнаружено, выполняются повороты для восстановления сбалансированности дерева.

1. Правый поворот (Right Rotation)

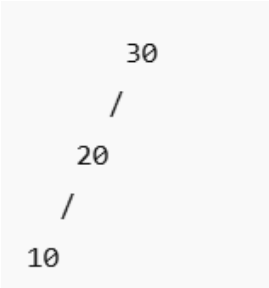
Когда применяется:
Правый поворот выполняется, когда левое поддерево узла слишком высокое (баланс-фактор узла

больше 1), то есть нарушен баланс в левом поддереве.

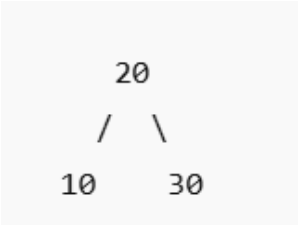
Как работает:
Правый поворот перемещает левое поддерево узла вниз, а сам узел поднимается вверх, становясь правым дочерним узлом.

Пример:

До поворота:



После правого поворота:



Шаги выполнения правого поворота:

- Узел `30` становится правым дочерним узлом узла `20`.
- Узел `10` остается левым дочерним узлом узла `20`.
- Узел `20` становится новым корнем.

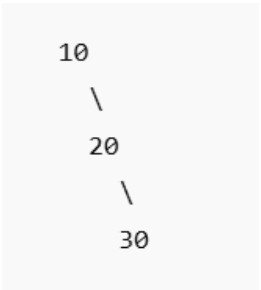
2. Левый поворот (Left Rotation)

Когда применяется:
Левый поворот выполняется, когда правое поддерево узла слишком высокое (баланс-фактор узла меньше -1), то есть нарушен баланс в правом поддереве.

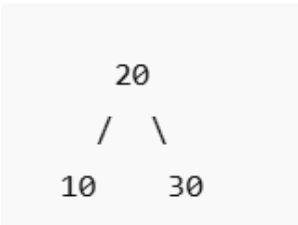
Как работает:
Левый поворот перемещает правое поддерево узла вниз, а сам узел поднимается вверх, становясь левым дочерним узлом.

Пример:

До поворота:



После левого поворота:



Шаги выполнения левого поворота:

- Узел `10` становится левым дочерним узлом узла `20`.
- Узел `30` остается правым дочерним узлом узла `20`.
- Узел `20` становится новым корнем.

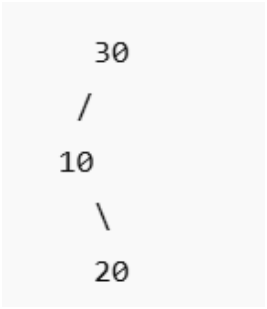
3. Левый-правый поворот (Left-Right Rotation)

Когда применяется:

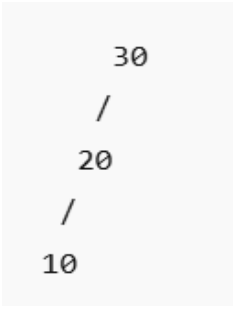
Этот поворот нужен, когда у левого поддерева есть правый перекос.

Пример:

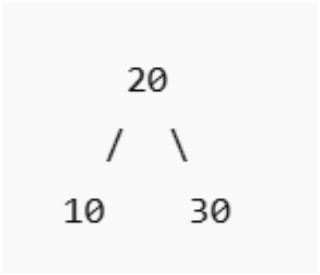
До поворота:



Шаг 1: Выполняем **левый поворот** на узле 10:



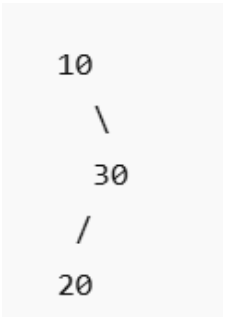
Шаг 2: **Выполняем** правый поворот **на узле 30:**



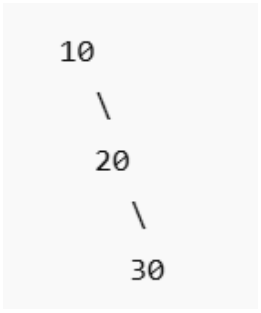
4. **Правый-левый поворот (Right-Left Rotation)**

Этот поворот нужен, когда у правого поддерева есть левый перекос.

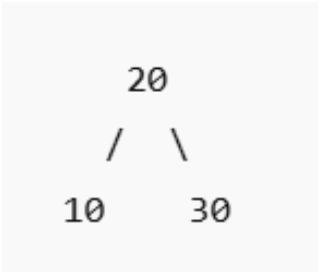
До поворота:



Шаг 1: Выполняем **правый поворот** на узле 30:



Шаг 2: Выполняем **левый поворот** на узле 10:



Баланс-фактор:

- Баланс-фактор (balance factor) каждого узла определяется как разница между высотами его левого и правого поддеревьев. Баланс-фактор для каждого узла может быть:
  - **+1**: левое поддерево выше правого (левый перекос).
  - **0**: поддеревья одинаковой высоты.
  - **-1**: правое поддерево выше левого (правый перекос).

Если баланс-фактор узла становится меньше -1 или больше +1, необходимо выполнить поворот.

ЗАДАНИЕ

Задание на лабораторную #6.

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.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

язык программирования C++

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
#include <fstream>
#include <cmath>
#include <stack>

using namespace std;
using namespace chrono;

// Узел бинарного дерева поиска (BST)
struct Node {
    int key; // Значение (ключ), по которому дерево упорядочено
    Node* left; // Указатель на левое поддерево (элементы, меньшие key)
    Node* right; // Указатель на правое поддерево (элементы, большие key)
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Итеративное добавление элемента в BST
Node* insertBST(Node* root, int key) {
    Node** curr = &root;
    while (*curr) { //вниз по дереву
        // Если ключ меньше текущего узла, переходим в левое поддерево, иначе в правое.
        curr = (key < (*curr)->key) ? &(*curr)->left : &(*curr)->right;
    }
    *curr = new Node(key); // Создаем новый узел в найденной пустой позиции
    return root;
}

// Итеративный поиск элемента в BST
bool searchBST(Node* root, int key) {
    Node* curr = root;
    while (curr) {
        if (curr->key == key) return true; //текущий узел имеет ключ равный искомому,
функция возвращает true
        curr = (key < curr->key) ? curr->left : curr->right; // Иначе идем в
соответствующее поддерево
    }
    return false; // Если дошли до конца элемент не найден(nullptr), то фолс
}

// Итеративное удаление узла из BST
Node* deleteBST(Node* root, int key) {
    Node* parent = nullptr;
    Node* curr = root;

    // Поиск удаляемого узла
    while (curr && curr->key != key) {
        parent = curr;
        curr = (key < curr->key) ? curr->left : curr->right;
    }
    if (!curr) return root; // Если узел не найден, возвращаем дерево без изменений

    // Если у узла меньше одного потомка (один или ни одного)
    if (!curr->left || !curr->right) {
        Node* child = curr->left ? curr->left : curr->right;//находим чаилда
        if (!parent) { // Если удаляем корень
            delete curr;
            return child;
        }
```

```
            // Обновляем указатель родительского узла
            if (parent->left == curr) parent->left = child;
            else parent->right = child;
            delete curr;
        }
        else { // Если у узла два потомка
            // Находим наименьший узел в правом поддереве (преемника)
            Node* successor = curr->right;
            Node* successorParent = curr;
            while (successor->left) {
                successorParent = successor;
                successor = successor->left;
            }
            curr->key = successor->key; // Копируем ключ преемника в текущий узел
            // Удаляем преемника (он имеет не более одного потомка)
            if (successorParent->left == successor)
                successorParent->left = successor->right;
            else
                successorParent->right = successor->right;
            delete successor;
        }
        return root;
    }

    // Освобождение памяти BST (итеративный обход)
    void freeBST(Node* root) {
        stack<Node*> nodes; //итеративный обход дерева с помощью стека
        if (root) nodes.push(root);

        while (!nodes.empty()) { //в цикле из стека извлекаются узлы
            Node* current = nodes.top();
            nodes.pop();

            // если есть дочерние элементы, они добавляются в стек
            if (current->left) nodes.push(current->left);
            if (current->right) nodes.push(current->right);

            delete current; //После обхода каждый узел удаляется
        }
    }

    // Узел AVL-дерева
    struct AVLNode {
        int key, height; // key - значение узла, height - высота поддерева с этим узлом в
    корне
        AVLNode* left;
        AVLNode* right;
        AVLNode(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
    };

    // Получение высоты узла AVL-дерева
    int getHeight(AVLNode* node) {
        return node ? node->height : 0;
    }

    // Получение баланса узла AVL-дерева
    int getBalance(AVLNode* node) {
        return node ? getHeight(node->left) - getHeight(node->right) : 0;
    }

    // Правый поворот
    AVLNode* rotateRight(AVLNode* y) {
        AVLNode* x = y->left; // x - левый ребенок y
        AVLNode* T2 = x->right; // T2 - правое поддерево x, которое станет левым поддеревом y
        x->right = y; // x становится родительским, y перемещается вправо
        y->left = T2; // T2 присоединяется как левое поддерево y
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
```

```
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1; // сам узел считается
высотой 1
    return x;
}

// Левый поворот
AVLNode* rotateLeft(AVLNode* x) {
    AVLNode* y = x->right; // y - правый ребенок x
    AVLNode* T2 = y->left; // T2 - левое поддерево y, которое станет правым поддеревом x
    y->left = x; // y становится родительским, x перемещается влево
    x->right = T2; // T2 присоединяется как правое поддерево x
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    return y;
}

// Рекурсивное добавление элемента в AVL-дерево с балансировкой
AVLNode* insertAVL(AVLNode* node, int key) {
    if (!node) return new AVLNode(key); // Если узел пустой, создаем новый узел
    // Рекурсивно вставляем элемент в левое или правое поддерево(зависит от key)
    if (key < node->key)
        node->left = insertAVL(node->left, key);
    else if (key > node->key)
        node->right = insertAVL(node->right, key);
    else
        return node; // Если ключ уже существует, не вставляем повторно

    // Обновляем высоту текущего узла
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int balance = getBalance(node); // Рассчитываем баланс-фактор

    //Left Left: если ключ вставляется в левое поддерево левого ребенка
    if (balance > 1 && key < node->left->key)
        return rotateRight(node);
    //Right Right: если ключ вставляется в правое поддерево правого ребенка
    if (balance < -1 && key > node->right->key)
        return rotateLeft(node);
    //Left Right: если ключ вставляется в правое поддерево левого ребенка
    if (balance > 1 && key > node->left->key) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }
    //Right Left: если ключ вставляется в левое поддерево правого ребенка
    if (balance < -1 && key < node->right->key) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

// Освобождение памяти AVL-дерева
void freeAVL(AVLNode* root) {
    if (root) {
        freeAVL(root->left);
        freeAVL(root->right);
        delete root;
    }
}

// Функция тестирования работы BST и AVL
void runTests() {
    ofstream out("results.csv");
    out <<
"Size,InsertBST,InsertAVL,SearchBST,SearchAVL,SearchArray,DeleteBST,DeleteAVL,ArrayType\n
";

    random_device rd;
```

```
    mt19937 gen(rd());

    const int test_series = 10;
    const int cycles per series = 20;
    const int search_ops = 1000;
    const int delete_ops = 1000;

    for (int i = 0; i < test_series; ++i) {
        int size = 1 << (10 + i);
        vector<int> arr(size);
        iota(arr.begin(), arr.end(), 1);

        for (int cycle = 0; cycle < cycles_per_series; ++cycle) {
            bool is_sorted = (cycle >= cycles_per_series / 2);
            if (!is_sorted) shuffle(arr.begin(), arr.end(), gen);

            Node* bstRoot = nullptr;
            AVLNode* avlRoot = nullptr;
            for (int x : arr) {
                bstRoot = insertBST(bstRoot, x);
                avlRoot = insertAVL(avlRoot, x);
            }
            freeBST(bstRoot);
            freeAVL(avlRoot);
        }
    }
    out.close();
}

int main() {
    runTests();
    return 0;
}
```

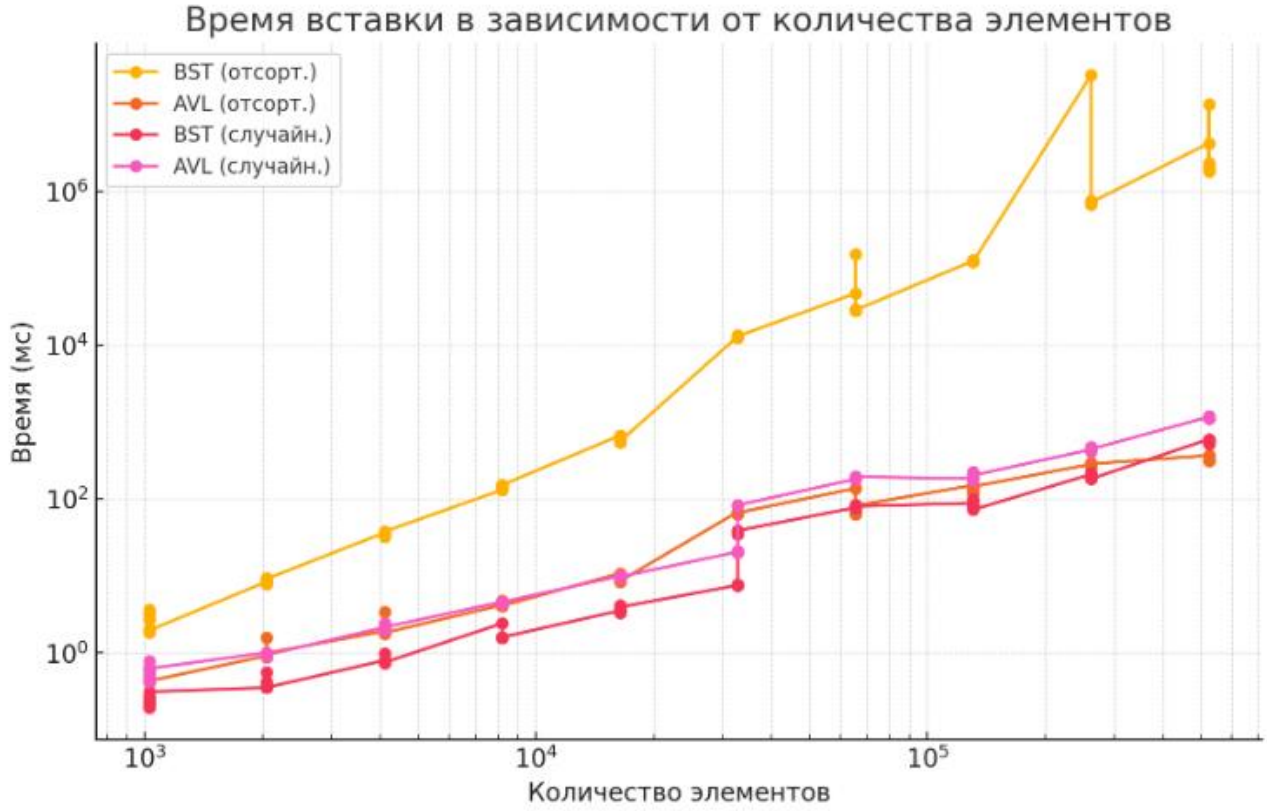
Результат работы программы:

вывод результатов в файл

	A	B	C	D	E	F	G	H	I	J	K
1	Size,InsertBST,InsertAVL,SearchBST,SearchAVL,SearchArray,DeleteBST,DeleteAVL,ArrayType										
2	1024,0.1982,0.4385,7.23e-05,7.44e-05,0.0008448,0.0003829,0.0005497,random										
3	1024,0.1927,0.4568,7.28e-05,7.54e-05,0.0006028,0.0001489,0.0004042,random										
4	1024,0.27,0.5389,9.2e-05,0.000102,0.0006862,0.0001922,0.0004574,random										
5	1024,0.2181,0.5878,9.52e-05,9.02e-05,0.0006342,0.0002354,0.0006424,random										
6	1024,0.2457,0.6557,0.0001224,0.0001099,0.0006785,0.0001473,0.0003881,random										
7	1024,0.2184,0.4201,8.94e-05,8e-05,0.0008701,0.0003638,0.0006913,random										
8	1024,0.4328,0.7824,0.0001005,0.0001053,0.000701,0.0001327,0.0003672,random										
9	1024,0.2236,0.5168,9.32e-05,0.0001034,0.0005856,0.0002261,0.0004218,random										
10	1024,0.2627,0.4771,9.72e-05,7.6e-05,0.0006817,0.0001447,0.000367,random										
11	1024,0.3113,0.6247,0.0001508,0.0001247,0.000827,0.0001761,0.0004249,random										
12	1024,3.6438,0.4445,0.0025431,0.0001083,0.0006366,0.0018353,0.0004181,sorted										
13	1024,1.9648,0.4185,0.0017608,7.46e-05,0.0006076,0.001125,0.0003803,sorted										
14	1024,3.0878,0.4998,0.002241,8.36e-05,0.0005993,0.0012297,0.0006229,sorted										
15	1024,3.206,0.5674,0.0017679,7.34e-05,0.0005915,0.0011278,0.0003411,sorted										
16	1024,1.8653,0.5079,0.0018139,0.0001069,0.0006318,0.0012942,0.0005212,sorted										
17	1024,2.7388,0.6516,0.0020943,8.49e-05,0.0006037,0.0011665,0.0003771,sorted										
18	1024,1.87,0.4084,0.001638,8.02e-05,0.0005852,0.0010114,0.0004058,sorted										
19	1024,2.0936,0.4987,0.0019467,9.26e-05,0.0006061,0.0013057,0.0004316,sorted										
20	1024,2.72,0.4293,0.0018815,9.14e-05,0.0006106,0.0010773,0.0003498,sorted										
21	1024,1.952,0.4304,0.0017504,8.01e-05,0.0006541,0.0010067,0.000343,sorted										
22	2048,0.3533,1.0088,0.0001068,8.29e-05,0.0008278,0.0001541,0.0004189,random										
23	2048,0.3591,0.9298,8.03e-05,8.58e-05,0.0008145,0.0001579,0.0004173,random										
24	2048,0.3551,0.9182,9.62e-05,8.55e-05,0.0008194,0.0001886,0.0004569,random										
25	2048,0.4217,0.9165,7.3e-05,8.13e-05,0.0008377,0.0001446,0.0004192,random										
26	2048,0.3628,0.9352,6.92e-05,8.55e-05,0.0008168,0.0001522,0.000424,random										

Графики:

язык программирования Python



- По оси **X** – количество элементов в структуре.
- По оси **Y** – время вставки (суммарное время, потраченное на вставку всех элементов).
- Две группы графиков: **случайные данные** и **отсортированные данные**.

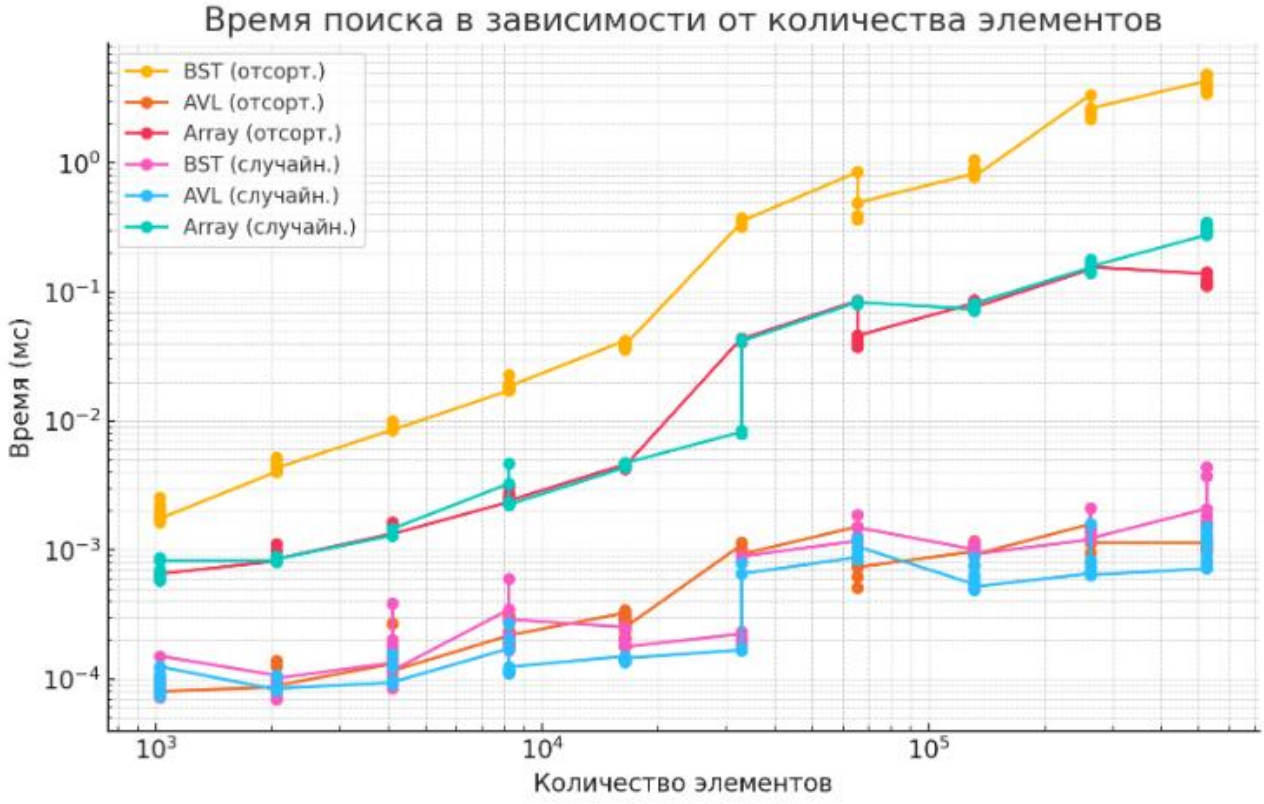
Что можно заметить?
BST (отсортированные данные) – время резко возрастает экспоненциально! Это подтверждает, что BST превращается в односвязный список, давая худший случай O(n).

BST (случайные данные) – работает лучше, но не идеально, так как баланс не поддерживается автоматически.

AVL (оба случая) – линии идут плавно и близко друг к другу, что подтверждает хорошую балансировку. AVL обеспечивает O(log(n)), поэтому рост времени происходит более медленно.

Вывод:

AVL – лучший выбор, если важна быстрая вставка без ухудшения структуры.
BST на отсортированных данных – катастрофа, лучше не использовать без балансировки.



- Ось **X** – количество элементов.
- Ось **Y** – среднее время поиска одного элемента.
- Линии:
  - **BST (отсортированные и случайные данные)**
  - **AVL (отсортированные и случайные данные)**
  - **Линейный поиск в массиве (Array)**

Что можно заметить?
BST (отсортированные данные) – работает очень плохо, почти линейный рост O(n).

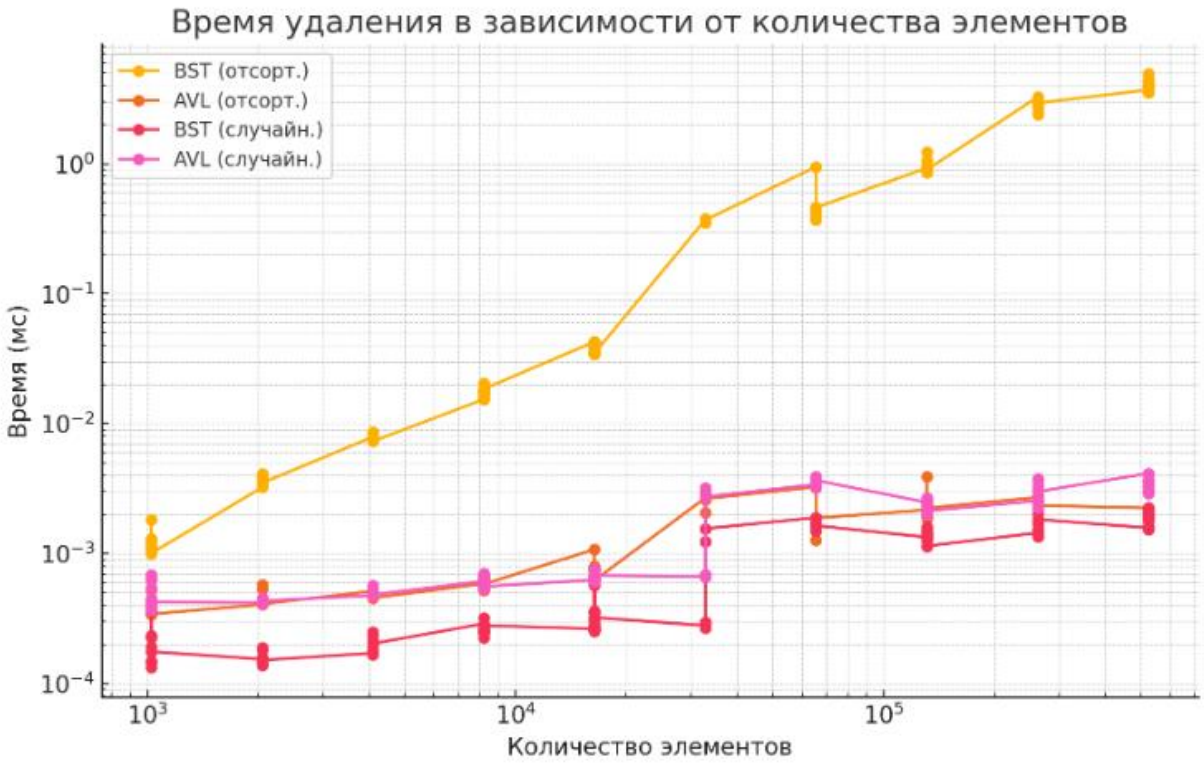
BST (случайные данные) – лучше, но все равно не стабильно.

AVL – почти не изменяясь даже при увеличении данных.

Линейный поиск (Array) – ожидаемо самый медленный, так как идет полное перебирание массива.

Вывод:

AVL снова лучший.
BST без балансировки – плохая идея, особенно если данные заранее отсортированы (при вставке их в обычное (не сбалансированное) BST получается структура, которая превращается в "цепочку").
Линейный поиск не подходит для больших объемов данных.



- Ось **X** – количество элементов.
- Ось **Y** – среднее время удаления одного элемента.

Что можно заметить?
BST (отсортированные данные) – удаление медленное, потому что дерево превратилось в список.

BST (случайные данные) – чуть лучше, но без балансировки возможны скачки времени.

AVL – опять показывает стабильное O(log(n)) время.

Вывод:

AVL – лучший по скорости удаления.
BST без балансировки на отсортированных данных – боль, лучше его не использовать.

ВЫВОД

1. **Различия в производительности BST и AVL-деревьев:**
   На основании полученных графиков и измерений времени операций мы убедились, что обычное бинарное дерево поиска (BST) показывает высокую эффективность при работе со случайными данными, однако при заранее отсортированных данных его производительность резко ухудшается. Это связано с тем, что при вставке отсортированных данных BST вырождается в линейную структуру (связанный список), где высота дерева становится равной количеству элементов.

2. **Преимущества сбалансированных деревьев (AVL):**
   AVL-деревья, благодаря поддержанию строгой балансировки, демонстрируют стабильное логарифмическое время выполнения операций (вставка, поиск, удаление) независимо от порядка вставки данных. Это подтверждает, что балансировка дерева является критически важной для обеспечения эффективной работы.

3. **Сравнение с линейным поиском:**
   Анализ графиков для линейного поиска в массиве показал, что даже при относительно небольшом количестве элементов линейный перебор оказывается значительно медленнее, чем поисковые операции в сбалансированных деревьях.

4. **Наглядность результатов:**
   Построенные графики с логарифмической шкалой на осях позволили наглядно увидеть разницу между теоретическими оценками сложности (O(log(n)) против O(n)) и реальными временными затратами. Резкие скачки на графиках BST для отсортированных данных иллюстрируют, насколько критично может ухудшаться производительность при неблагоприятном порядке ввода данных.

Итог:

Работа позволила нам наглядно осознать, что выбор структуры данных имеет решающее значение для производительности алгоритмов. Использование сбалансированных деревьев, таких как AVL, является предпочтительным решением для динамических множеств, так как оно обеспечивает гарантированное логарифмическое время выполнения операций независимо от входных данных. В то время как обычное BST может хорошо работать на случайных данных, его эффективность значительно снижается при работе с отсортированными данными, что демонстрирует важность балансировки при реализации деревьев поиска.