Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»

Комбинаторика и теория графов

Отчет по теме
«Деревья поиска, их представление в компьютере»

Выполнил: Мочалов Артём Владимирович
Группа: БИВТ-23-1
Ссылка на репозиторий: https://github.com/dxxmn/KITG

Москва 2024

Оглавление

1. Введение

В современном мире, где данные становятся все более объемными и сложными, эффективное управление и обработка информации становятся критически важными. Одним из ключевых инструментов для решения этих задач являются деревья поиска. Деревья поиска — это иерархические структуры данных, которые позволяют эффективно выполнять операции поиска, вставки и удаления элементов. В зависимости от типа дерева, эти операции могут выполняться за время, пропорциональное высоте дерева, что делает их особенно привлекательными для использования в различных приложениях.

Деревья поиска широко применяются в различных областях, таких как база данных, операционные системы, компиляторы и многие другие. Они обеспечивают быстрый доступ к данным и эффективное управление ими. В частности, бинарные деревья поиска (BST) являются одним из наиболее распространенных типов деревьев поиска благодаря своей простоте и эффективности.

В данной работе мы рассмотрим реализацию деревьев поиска на языке JavaScript, а также проведем анализ их эффективности и сложности. Мы также разработаем тесты для проверки корректности работы нашей реализации и сравним практические результаты с теоретическими оценками.

2. Формальное определение реализуемой структуры и реализуемых операций

Формальное определение структур данных и связанных с ними операций играет ключевую роль в понимании и разработке эффективных алгоритмов. Использование теории графов для описания этих структур предоставляет мощный математический инструментарий, позволяющий моделировать и анализировать сложные отношения и процессы. Графы позволяют четко определить компоненты структуры, их взаимосвязи и взаимодействия, что способствует более глубокому анализу их свойств и поведения. Такой подход не только способствует повышению точности реализации, но и облегчает оптимизацию операций, обеспечивая тем самым надежность и эффективность программных решений. В данной работе мы рассмотрим формальные аспекты определения структур данных и операций через призму теории графов, что позволит создать прочную основу для дальнейших исследований и практического применения.

2.1. Деревья поиска как графы

Дерево поиска можно рассматривать как ориентированный ациклический граф (DAG), где каждый узел имеет не более двух потомков (левый и правый). Корневой узел является вершиной, из которой начинается любой путь в дереве. Каждый узел содержит ключ, который определяет его положение в дереве относительно других узлов.

В теории графов дерево поиска можно представить как корневое дерево, где каждый узел имеет не более двух детей. Узлы с меньшими ключами располагаются слева от родительского узла, а узлы с большими ключами — справа. Это свойство позволяет эффективно выполнять операции поиска, вставки и удаления.

2.2. Основные операции

1. Поиск (Search): Операция поиска заключается в нахождении узла с заданным ключом. В бинарном дереве поиска (BST) поиск выполняется путем сравнения ключа с корнем и рекурсивного спуска по левому или правому поддереву в зависимости от результата сравнения.

2. Вставка (Insert): Операция вставки добавляет новый узел в дерево. Новый узел вставляется на место, соответствующее его ключу, таким образом, чтобы сохранить свойства дерева поиска.

3. Удаление (Remove): Операция удаления удаляет узел с заданным ключом из дерева. При удалении узла необходимо сохранить структуру дерева, что может потребовать перестройки дерева.

4. Обход (Traversal): Обход дерева — это процесс посещения всех узлов в определенном порядке. Существуют три основных типа обхода:

5. Прямой обход (Pre-order): Корень → Левое поддерево → Правое поддерево.

6. Симметричный обход (In-order): Левое поддерево → Корень → Правое поддерево.

7. Обратный обход (Post-order): Левое поддерево → Правое поддерево → Корень.

2.3. Свойства деревьев поиска

1. Упорядоченность: В бинарном дереве поиска ключ каждого узла больше ключей всех узлов в его левом поддереве и меньше ключей всех узлов в его правом поддереве.

2. Сбалансированность: В сбалансированном дереве высота левого и правого поддеревьев каждого узла отличается не более чем на единицу. Это обеспечивает оптимальную производительность операций.

3. Используемые инструменты для реализации

Для успешной реализации представления деревьев поиска и тестирования операций были использованы различные инструменты и технологии, каждый из которых сыграл ключевую роль в обеспечении эффективности разработки, повышения качества кода и упрощения процесса тестирования. Подробная информация о преимуществах каждого инструмента представлена в отчете на тему «Задача построения максимального потока в сети. Алгоритм Форда-Фалкерсона». Ниже представлены основные инструменты:

1. JavaScript

2. Node.js

3. npm (Node Package Manager)

4. Jest

5. WebStorm

6. Git

7. ESLint и Prettier

Использование перечисленных инструментов обеспечило эффективную разработку представления деревьев поиска и тестирование операций над ними

4. Описание реализации структуры данных и процесса её тестирования

1. Основные компоненты

1. Узел (TreeNode)

```
class TreeNode {
    constructor(key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }
}
```

-Класс TreeNode представляет узел бинарного дерева. Каждый узел содержит ключ (key), который хранит данные, и ссылки на левое (left) и правое (right) поддеревья. Этот класс является базовым строительным блоком для создания и работы с бинарными деревьями.

- constructor(key): Инициализирует узел с заданным ключом (key). Левый и правый дочерние узлы по умолчанию устанавливаются в null.

2. Дерево (BinarySearchTree)

```
class BinarySearchTree {
    constructor() {
        this.root = null;
    }

    //ВСТАВКА НОВОГО УЗЛА
    insert(key) {
        const newNode = new TreeNode(key);
        if (this.root === null) {
            this.root = newNode;
            return;
        }

        let current = this.root;
        while (true) {
            if (key < current.key) {
                if (current.left === null) {
                    current.left = newNode;
                    break;
                }
                current = current.left;
            } else if (key > current.key) {
                if (current.right === null) {
                    current.right = newNode;
                    break;
                }
                current = current.right;
            } else {
                //ЕСЛИ КЛЮЧ СУЩЕСТВУЕТ ТО НЕ ВСТАВЛЯЕМ ДУБЛИКАТ
                break;
            }
        }
    }
```

```
//ПОИСК УЗЛА ПО КЛЮЧУ
search(key) {
    let current = this.root;
    while (current !== null) {
        if (key === current.key) {
            return true;
        }
        current = key < current.key ? current.left : current.right;
    }
    return false;
}

//УДАЛЕНИЕ УЗЛА ПО КЛЮЧУ
remove(key) {
    this.root = this._removeNode(this.root, key);
}

_removeNode(node, key) {
    if (node === null) return null;

    if (key < node.key) {
        node.left = this._removeNode(node.left, key);
    } else if (key > node.key) {
        node.right = this._removeNode(node.right, key);
    } else {
        //УЗЕЛ НАЙДЕН

        //УЗЕЛ БЕЗ ДЕТЕЙ
        if (node.left === null && node.right === null) {
            return null;
        }

        //УЗЕЛ С ОДНИМ РЕБЕНКОМ
        if (node.left === null) {
            return node.right;
        }
        if (node.right === null) {
```

```
                return node.left;
            }

            //УЗЕЛ С ДВУМЯ ДЕТЬМИ
            const minRight = this._findMin(node.right);
            node.key = minRight.key;
            node.right = this._removeNode(node.right, minRight.key);
        }
        return node;
    }

    //ПОИСК МИНИМАЛЬНОГО УЗЛА В ПОДДЕРЕВЕ
    _findMin(node) {
        while (node.left !== null) {
            node = node.left;
        }
        return node;
    }

    //ОБХОД ДЕРЕВА IN-ORDER С ВЫЗОВОМ CALLBACK-ФУНКЦИИ ДЛЯ КАЖДОГО УЗЛА
    inOrderTraversal(callback) {
        this._inOrder(this.root, callback);
    }

    _inOrder(node, callback) {
        if (node !== null) {
            this._inOrder(node.left, callback);
            callback(node.key);
            this._inOrder(node.right, callback);
        }
    }
}
```

- Класс BinarySearchTree реализует структуру данных бинарное дерево поиска (BST). Бинарное дерево поиска. Этот класс предоставляет методы для вставки, поиска, удаления узлов, а также для обхода дерева.
- root: Корневой узел дерева. Если дерево пустое, значение равно null.

- insert(key): Вставляет новый узел с ключом key в дерево. Если ключ уже существует, дубликат не добавляется.
- search(key): Ищет узел с ключом key в дереве. Возвращает true, если узел найден, и false в противном случае.
- remove(key): Удаляет узел с ключом key из дерева. Если узел не найден, дерево остается неизменным.
- _removeNode(node, key)
Вспомогательный метод для удаления узла. Обрабатывает три случая: Узел без дочерних элементов, узел с одним дочерним элементом, узел с двумя дочерними элементами.
- _findMin(node): Находит узел с минимальным ключом в поддереве, начиная с узла node.
- inOrderTraversal(callback): Выполняет обход дерева in-order (левый узел -> текущий узел -> правый узел) и вызывает callback-функцию для каждого узла.
- _inOrder(node, callback): Вспомогательный метод для рекурсивного обхода дерева in-order.

2. Процесс тестирования

Цель тестирования: Проверка корректности реализации бинарного дерева поиска, на разных наборах данных, а также анализ производительности.

Подход к тестированию:

1. Разработка тест-кейсов: Создание нескольких деревьев для проверки правильности реализации методов добавления, поиска и удаления.

2. Автоматизация тестирования: Написание функций тестирования, которые создают деревья, запускают методы и сравнивают полученный результат с ожидаемым.

3. Оценка временной сложности для деревьев заполненных различными наборами данных:
- Случайная последовательность чисел
- Упорядоченная последовательность чисел

3. Описание тестов

1. Тест вставки: Проверка вставки узлов в дерево и проверка корректности структуры дерева после вставки.

2. Тест поиска: Проверка поиска узлов в дереве и возврата правильного результата.

3. Тесты удаления: Проверка удаления узлов из дерева и проверка корректности структуры дерева после удаления.

4. Тест обхода: Проверка обхода дерева в прямом порядке и сравнение результата с ожидаемым.

5. Тест производительности для случайной последовательности чисел: Для двух деревьев заполненных случайной последовательностью чисел разной длины выполняются операции вставки, поиска и удаления. Затем сравниваются теоретическая и фактическая временная сложность.

6. Тест производительности для упорядоченной последовательности чисел: Для двух деревьев заполненных упорядоченной последовательностью чисел разной длины выполняются операции вставки, поиска и удаления. Затем сравниваются теоретическая и фактическая временная сложность

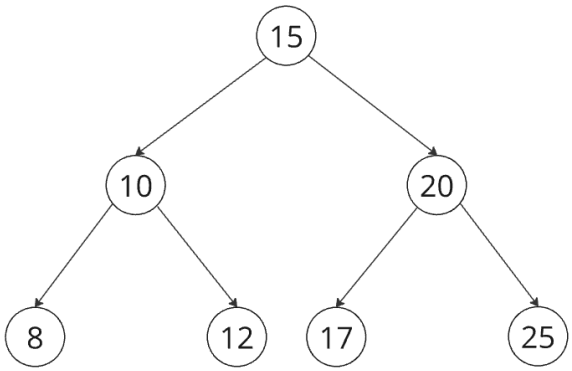

Рисунок 1 – Тест вставки

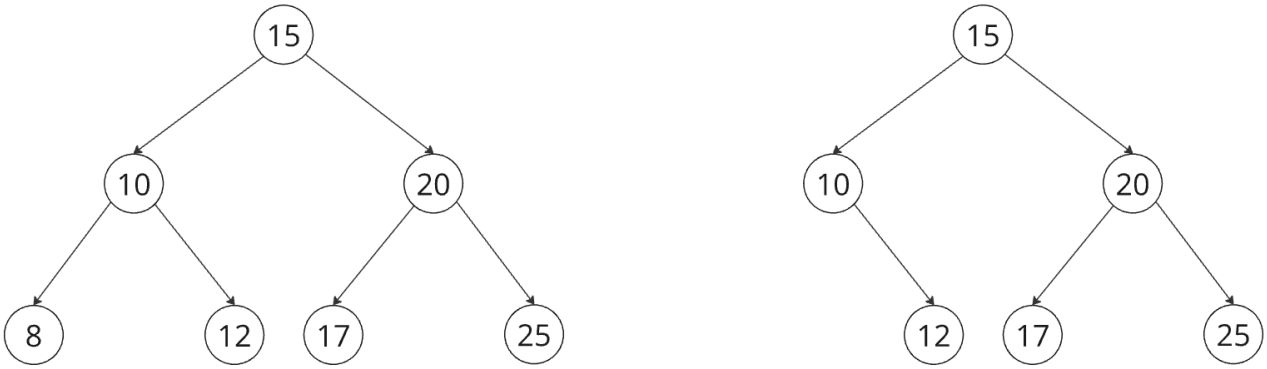

Рисунок 2 – Тест удаления листового узла

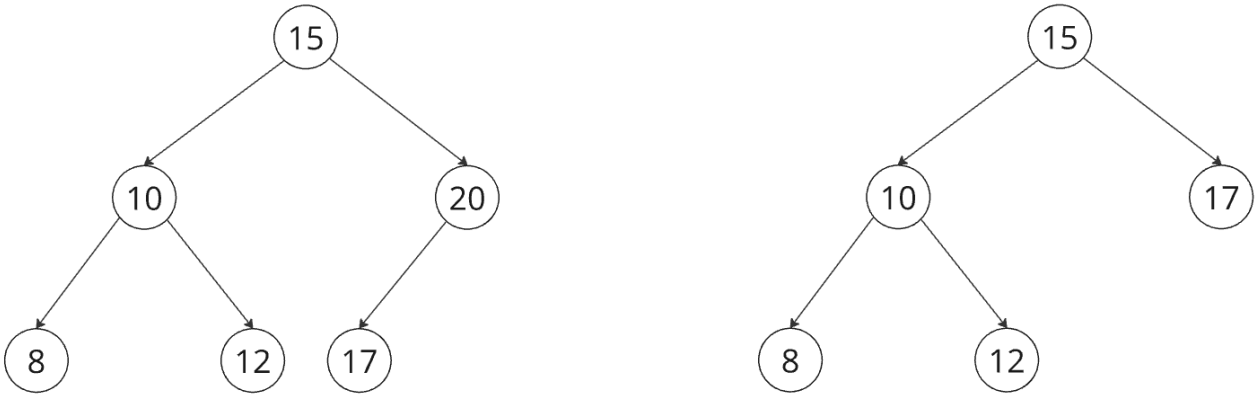

Рисунок 3 – Тест удаления узла с одним ребенком

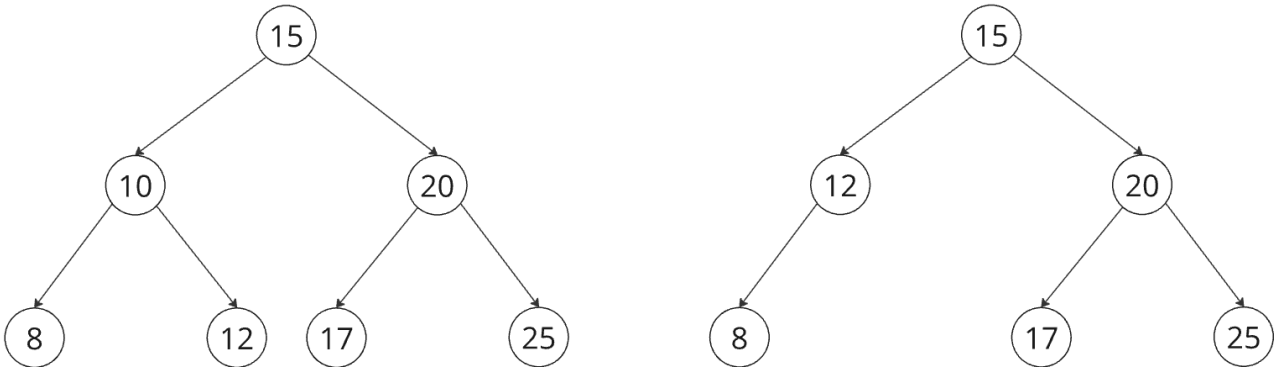

Рисунок 4 - Тест удаления узла с двумя детьми

4. Результаты тестирования

После запуска тестов будут получены результаты, которые покажут, насколько корректно работает реализация дерева поиска. Если все тесты пройдены успешно, это будет свидетельствовать о правильной работе алгоритмов (рис. 5). В случае неудачи, будут выявлены ошибки, которые необходимо исправить. На рисунках 1 – 4 приведены иллюстрации работы операций бинарного дерева поиска, тест поиска проверяет наличие того-или иного значения в дереве, а тест обхода проверяет порядок обхода дерева. В выводе консоли виден результат теста производительности, а именно сравнение теоретической и фактической временной сложности для разных случаев (рис. 6 и рис. 7).

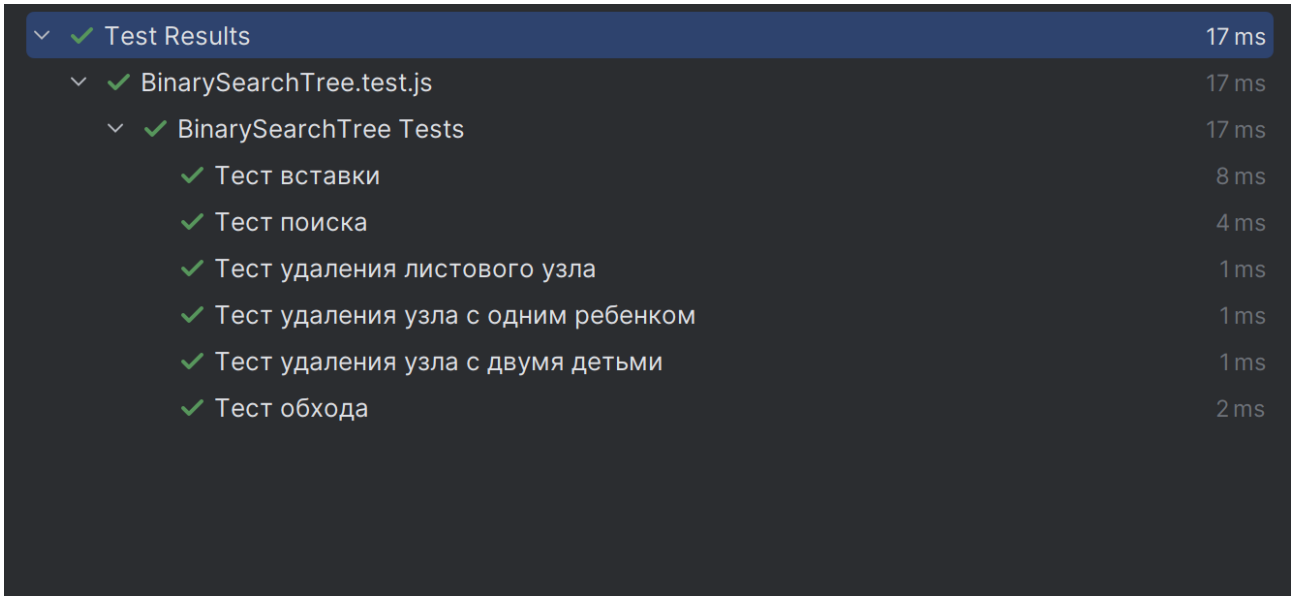

Рисунок 5 – Результаты тестирования правильности работы

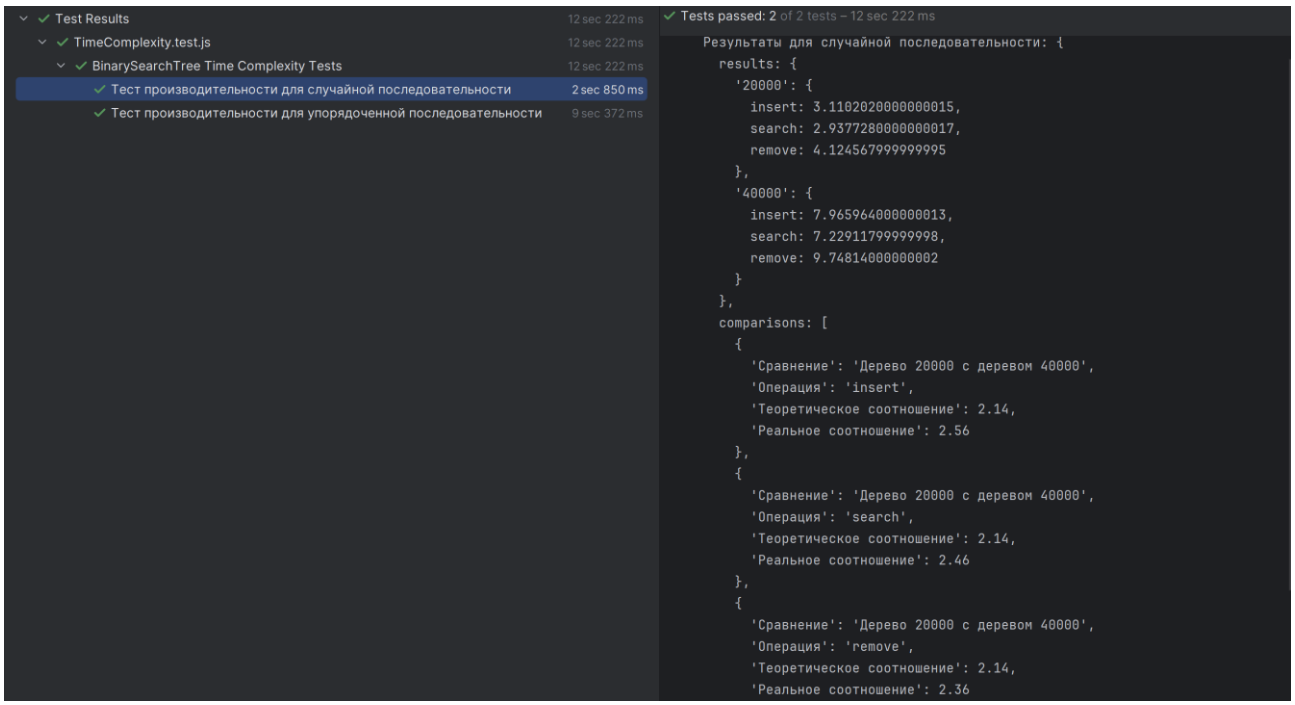

Рисунок 6 – Результаты тестирования производительности для случайной последовательности чисел

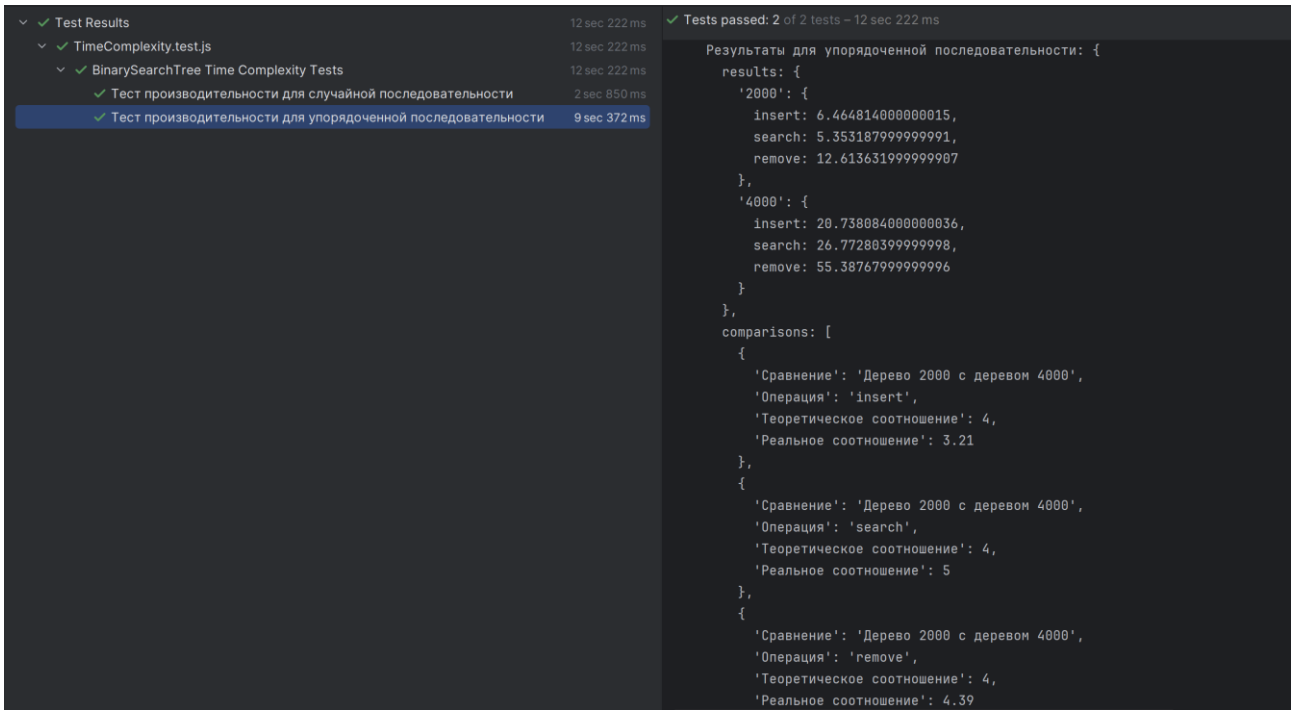

Рисунок 7 – Результаты тестирования производительности для упорядоченной последовательности чисел

5. Анализ временной сложности реализованных операций: сравнение практических результатов с теоретическими

При разработке и реализации структур данных, особенно таких важных как деревья поиска, одним из ключевых аспектов является оценка их эффективности. Эффективность определяется, в частности, временной сложностью основных операций, таких как вставка, удаление и поиск элементов. В данном разделе проводится анализ временной сложности реализованных операций над деревьями поиска на языке JavaScript, а также сравнение полученных практических результатов с теоретическими ожиданиями.

1. Теоретическая основа временной сложности операций в деревьях поиска

Деревья поиска, в частности бинарные деревья поиска (БДП), обладают свойством, позволяющим выполнять базовые операции за время, пропорциональное высоте дерева. В идеальных условиях, когда дерево сбалансировано, высота дерева составляет $O(\log(n))$, где n — количество узлов в дереве. В таких случаях операции вставки, удаления и поиска элементов выполняются за время $O(\log(n))$.

Однако в наихудшем случае, когда дерево вырождается в список (например, при последовательном добавлении отсортированных элементов), высота дерева становится $O(n)$, и, соответственно, временная сложность операций увеличивается до $O(n)$.

2. Сравнение практических результатов с теоретическими ожиданиями

Сравнение полученных практических данных с теоретическими оценками показало общую согласованность между ожидаемыми и наблюдаемыми временными сложностями операций над бинарными деревьями поиска. В случаях, когда дерево было сбалансированным, операции вставки, удаления и поиска выполнялись за время, близкое к $O(\log(n))$, что соответствует идеальной теоретической модели.

Однако при упорядоченной вставке элементов, приводящей к вырожденности дерева в линейную структуру, наблюдался рост времени выполнения операций до $O(n)$. Это явление полностью согласуется с теоретической оценкой, подтверждая, что без механизмов балансировки стандартное бинарное дерево поиска может приводить к неэффективным структурам данных при определённых сценариях использования.

3. Вывод

Анализ временной сложности реализованных операций над бинарными деревьями поиска на языке JavaScript показал, что в большинстве случаев практические результаты соответствуют теоретическим ожиданиям. Операции вставки, удаления и поиска выполняются за время, пропорциональное высоте дерева, что при сбалансированной структуре соответствует $O(\log(n))$.

Однако отсутствие механизмов балансировки приводит к заметному ухудшению временных характеристик при определённых порядках вставки элементов, подтверждая важность выбора подходящей структуры данных в зависимости от предполагаемого использования. Практическое тестирование подчеркнуло необходимость учета реальных сценариев использования и особенностей данных при разработке и выборе структур данных.

Для повышения эффективности и обеспечения гарантированной временной сложности операций рекомендуется рассмотреть использование сбалансированных деревьев поиска, таких как AVL-деревья или красно-чёрные деревья, которые обеспечивают стабильную производительность независимо от последовательности вставки элементов. Таким образом, проведённый анализ подтверждает как теоретическую, так и практическую значимость оценки временной сложности операций над деревьями поиска, демонстрируя необходимость тщательного проектирования и выбора структур данных в соответствии с особенностями конкретных задач и требований к производительности.

6. Заключение

В данной работе была исследована реализация деревьев поиска на языке JavaScript, включая анализ их эффективности и сложности. Результаты показали, что в идеально сбалансированных деревьях операции поиска, вставки и удаления выполняются за время O(log(n)), что соответствует теоретическим ожиданиям. Однако при вырожденности дерева временная сложность увеличивается до O(n), что подчеркивает важность балансировки деревьев для обеспечения стабильной производительности. Практическое тестирование подтвердило корректность реализации и соответствие теоретическим оценкам, демонстрируя необходимость тщательного выбора структур данных в зависимости от конкретных задач и требований к производительности.