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 № 7

Выполнил студент группы ....КС-30 .... Лихолат Полина Николаевна
Ссылка на репозиторий: .... https://github.com/MUCTR-IKT-CPP/Likholat_algorithms.git

Приняли: .... Пысин Максим Дмитриевич
.... Краснов Дмитрий Олегович
.... Лобанов Алексей Владимирович
.... Крашенинников Роман Сергеевич

Дата сдачи: 09.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить красно-черное дерево поиска.

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

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

В рамках одной серии понадобится сделать следующее:

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

Для анализа структуры потребуется построить следующие графики:

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

Описание метода/модели.

Красно-черные деревья относятся к сбалансированным бинарным деревьям поиска.

Как бинарное дерево, красно-черное обладает свойствами:

1) Оба поддерева являются бинарными деревьями поиска.

2) Для каждого узла с ключом k выполняется критерий упорядочения:

ключи всех левых потомков < k < ключи всех правых потомков.

Это неравенство должно быть истинным для всех потомков узла, а не только его дочерних узлов.

Свойства красно-черных деревьев:

1) Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).

2) Корень окрашен в черный цвет.

3) Листья (так называемые NULL-узлы) окрашены в черный цвет.

4) Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.

5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов(это черная высота).

Красно-черные деревья не гарантируют строгой сбалансированности (разница высот двух поддеревьев любого узла не должна превышать 1), как в АВЛ-деревьях. Но соблюдение свойств красно-черного дерева позволяет обеспечить выполнение операций вставки, удаления и выборки за время O(logN).

Если путь от корневого узла до листового содержит минимальное количество красных узлов (т.е. ноль), значит этот путь равен bh.

Если же путь содержит максимальное количество красных узлов (bh в соответствии со свойством 4), то этот путь будет равен 2bh.

То есть, пути из корня к листьям могут различаться не более, чем вдвое ($h <= 2\log(N+1)$, где h — высота поддерева), этого достаточно, чтобы время выполнения операций в таком дереве было O(logN).

Вставка в красно-черное дерево начинается со вставки элемента, как в обычном бинарном дереве поиска. Только здесь элементы вставляются в позиции NULL-листьев. Вставленный узел всегда окрашивается в красный цвет. Далее идет процедура проверки сохранения свойств красно-черного дерева 1-5.

Свойство 1 не нарушается, поскольку новому узлу сразу присваивается красный цвет.

Свойство 2 нарушается только в том случае, если у нас было пустое дерево и первый вставленный узел (он же корень) окрашен в красный цвет. Здесь достаточно просто перекрасить корень в черный цвет.

Свойство 3 также не нарушается, поскольку при добавлении узла он получает черные листовые NULL-узлы.

В основном встречаются 2 других нарушения:

1) Красный узел имеет красный дочерний узел (нарушено свойство 4).
2) Пути в дереве содержат разное количество черных узлов (нарушено свойство 5).

Выполнение задачи.

Алгоритм вставки элемента в красно-черное дерево можно описать следующим образом:

1. Вставляем новый узел как в обычное бинарное дерево поиска, закрашивая его в красный цвет.
2. Если новый узел является корневым, закрашиваем его в черный цвет.
3. Если родитель нового узла является черным, то свойства красно-черного дерева не нарушены, завершаем операцию вставки.
4. Если родитель нового узла является красным, то возможны два случая:
   - Родитель является левым ребенком своего родителя и узел вставляется также как левый ребенок. В этом случае проверяем цвет дяди нового узла:
     - Если дядя является красным, то перекрашиваем родителя и дядю в черный цвет, а дедушку в красный. Затем повторяем алгоритм для дедушки.
     - Если дядя является черным или отсутствует, то поворачиваем поддерево вокруг родителя нового узла, чтобы получить левый поворот. Затем применяем правый поворот для дедушки и перекрашиваем родителя и дедушку в противоположные цвета.
   - Родитель является правым ребенком своего родителя и узел вставляется также как правый ребенок. Этот случай аналогичен предыдущему, но нужно выполнить зеркальные отражения.

Алгоритм завершается, когда все свойства красно-черного дерева сохранены. Это гарантирует, что высота дерева будет логарифмической по отношению к числу узлов, что обеспечивает быстрый доступ к элементам дерева.

```cpp
/**
    * Inserts a new node with the given key into the Red-Black Tree.
    * @param key The key to be inserted.
    */
void insert(int key) {
        Node* current = new Node(key);
        Node* y = nullptr;
        Node* x = root;
        while (x != nullptr) {
            y = x;
            if (current->key < x->key)
                x = x->left;
            else
                x = x->right;
        }
        current->parent = y;
        if (y == nullptr)
            root = current;
        else if (current->key < y->key)
            y->left = current;
        else
            y->right = current;
        fixInsert(current);
    }
```

Метод insert добавляет новый узел в красно-черное дерево и поддерживает его свойства с помощью вызова функции fixInsert.

```cpp
void fixInsert(Node* n) {
        while (n->parent != nullptr && n->parent->color == RED) {
            if (n->parent == n->parent->parent->left) {
                Node* u = n->parent->parent->right;
                if (u != nullptr && u->color == RED) {
                    n->parent->color = BLACK;
                    u->color = BLACK;
                    n->parent->parent->color = RED;
                    n = n->parent->parent;
                }
                else {
                    if (n == n->parent->right) {
                        n = n->parent;
                        leftRotate(n);
                    }
                    n->parent->color = BLACK;
                    n->parent->parent->color = RED;
                    rightRotate(n->parent->parent);
```

```
                }
            }
            else {
                Node* u = n->parent->parent->left;
                if (u != nullptr && u->color == RED) {
                    n->parent->color = BLACK;
                    u->color = BLACK;
                    n->parent->parent->color = RED;
                    n = n->parent->parent;
                }
                else {
                    if (n == n->parent->left) {
                        n = n->parent;
                        rightRotate(n);
                    }
                    n->parent->color = BLACK;
                    n->parent->parent->color = RED;
                    leftRotate(n->parent->parent);
                }
            }
        }
        root->color = BLACK;
    }
```

Алгоритм удаления элемента из красно-черного дерева включает в себя следующие шаги:

1. Найдите удаляемый узел и его родителя в красно-черном дереве, как в обычном бинарном дереве поиска.
2. Если удаляемый узел не имеет потомков, просто удалите его и завершите операцию.
3. Если удаляемый узел имеет только одного потомка, замените его потомком и завершите операцию.
4. Если удаляемый узел имеет двух потомков, замените его на следующий по значению узел, который находится в правом поддереве, или на следующий по значению узел в левом поддереве, если правое поддерево пусто. Также скопируйте цвет заменяемого узла в новый узел.
5. Удалите узел, который заменил удаленный узел, используя шаги 1-4.
6. Если удаленный узел был черным, проверьте, нарушены ли свойства красно-черного дерева:
   - Если удаляемый узел был корневым и был черным, то завершите операцию.
   - Если родитель удаленного узла или его потомок (заменитель) красный, закрасьте его в черный цвет, чтобы сохранить свойство 4. Завершите операцию.

- Иначе узел удален, его заменитель - новый черный узел, а его брат - красный. В этом случае выполняйте следующие действия:
  - Если брат узла черный, проверьте его потомков. Если один или оба потомка красные, выполните повороты и перекрашивания, чтобы преобразовать ситуацию в одну из следующих:
    - Брат - черный, левый потомок брата - черный, правый потомок брата - красный.
    - Брат - черный, правый потомок брата - черный, левый потомок брата - красный.
  - Затем перекрашивайте брата в черный цвет, а его левого и правого потомков - в красный. Завершите операцию.
  - Если брат узла красный, выполните повороты, чтобы преобразовать ситуацию в одну из следующих:
    - Брат - черный, левый потомок брата - красный, правый потомок брата - черный.
    - Брат - черный, правый потомок брата - красный, левый потомок брата - черный.
  - Затем перекрасьте брата в красный цвет и повторите операцию удаления для родителя удаленного узла.

```
/**
    *Removes a node with the given key from the Red-Black tree.
    *
    *@param key The key of the node to be removed.
    */
    void remove(int key) {
        Node* z = find(key); // Find the node with the given key
        if (z == nullptr)
            return; // If the node does not exist, return
        Node* x, * y;
        if (z->left == nullptr || z->right == nullptr)
            y = z; // If the node has no or only one child, set y to the node itself
        else
            y = minimum(z->right); // If the node has two children, set y to the minimum
node in its right subtree
        if (y->left != nullptr)
            x = y->left;
        else
            x = y->right;
        if (x == nullptr) // If x is null, return
            return;
```

```
        x->parent = y->parent;

        if (y->parent == nullptr)
            root = x;
        else if (y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;
        if (y != z)
            z->key = y->key; // Copy y's key to z, if necessary
        if (y->color == BLACK)
            fixDelete(x); // Fix the Red-Black tree properties, if necessary
        delete y; // Delete y
    }
```

В конце вызывается метод fixDelete(x), который выполняет повороты и перекрашивания узлов, чтобы сохранить свойства красно-черного дерева.

```
/**
     * fixDelete function fixes the violation of the red-black tree properties caused by
deleting a node from the tree
     *
     * @param x - the node where the fix starts
     */
    void fixDelete(Node* x) {
        while (x != root && x->color == BLACK) {
            if (x == x->parent->left) {
                // x is a left child
                Node* w = x->parent->right;
                // case 1: x's sibling w is red
                if (w->color == RED) {
                    // case 1: x's sibling w is red
                    w->color = BLACK;
                    x->parent->color = RED;
                    leftRotate(x->parent);
                    w = x->parent->right;
                }
                if (w->left->color == BLACK && w->right->color == BLACK) {
                    // case 2: x's sibling w is black, and both of w's children are black
                    w->color = RED;
                    x = x->parent;
                }
                else {
                    if (w->right->color == BLACK) {
                        // case 3: x's sibling w is black, w's left child is red, and w's
right child is black
                        w->left->color = BLACK;
```

```
                    w->color = RED;
                    rightRotate(w);
                    w = x->parent->right;
                }
                // case 4: x's sibling w is black, and w's right child is red
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                leftRotate(x->parent);
                x = root;
            }
        }
        else {
            // x is a right child
            Node* w = x->parent->left;
            if (w->color == RED) {
                // case 1: x's sibling w is red
                w->color = BLACK;
                x->parent->color = RED;
                rightRotate(x->parent);
                w = x->parent->left;
            }
            if (w->right->color == BLACK && w->left->color == BLACK) {
                // case 2: x's sibling w is black, and both of w's children are black
                w->color = RED;
                x = x->parent;
            }
            else {
                if (w->left->color == BLACK) {
                    // case 3: x's sibling w is black, w's right child is red, and w's
left child is black
                    w->right->color = BLACK;
                    w->color = RED;
                    leftRotate(w);
                    w = x->parent->left;
                }
                // case 4: x's sibling w is black, and w's left child is red
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->left->color = BLACK;
                rightRotate(x->parent);
                x = root;
            }
        }
    }
    x->color = BLACK;
```

```
}
```

Алгоритм поиска элемента в черно-красном дереве включает в себя следующие шаги:

1. Начните поиск с корня дерева.
2. Сравните значение элемента, который вы ищете, с значением текущего узла.
   - Если значение равно значению текущего узла, то вы нашли элемент и можете вернуть его.
   - Если значение меньше значения текущего узла, перейдите к левому потомку текущего узла и повторите шаг 2 для этого узла.
   - Если значение больше значения текущего узла, перейдите к правому потомку текущего узла и повторите шаг 2 для этого узла.
3. Если вы дошли до конца дерева и не нашли элемент, то он отсутствует в дереве.

```
Node* find(int key) {
    Node* x = root;
    while (x != nullptr) {
        if (key < x->key)
            x = x->left;
        else if (key > x->key)
            x = x->right;
        else
            return x;
    }
    return nullptr;
}
```

- На рисунках 1.1-1.2. представлены графики максимальной высоты полученных деревьев в зависимости от количества элементов. По ним видно, что RBT дерево имеет большую высоту, чем AVL, что связано с особенностям балансировки деревьев.

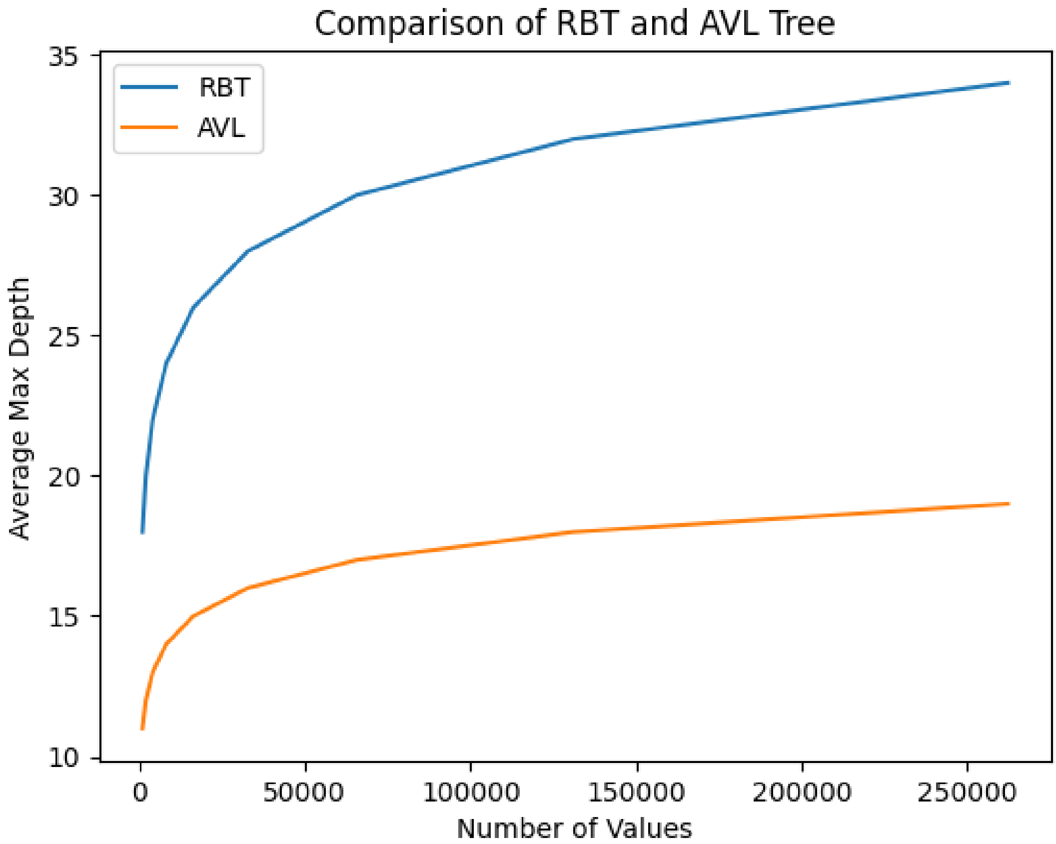

Рисунок 1.1Сравнение максимальной глубины для RBT и AVL деревьев

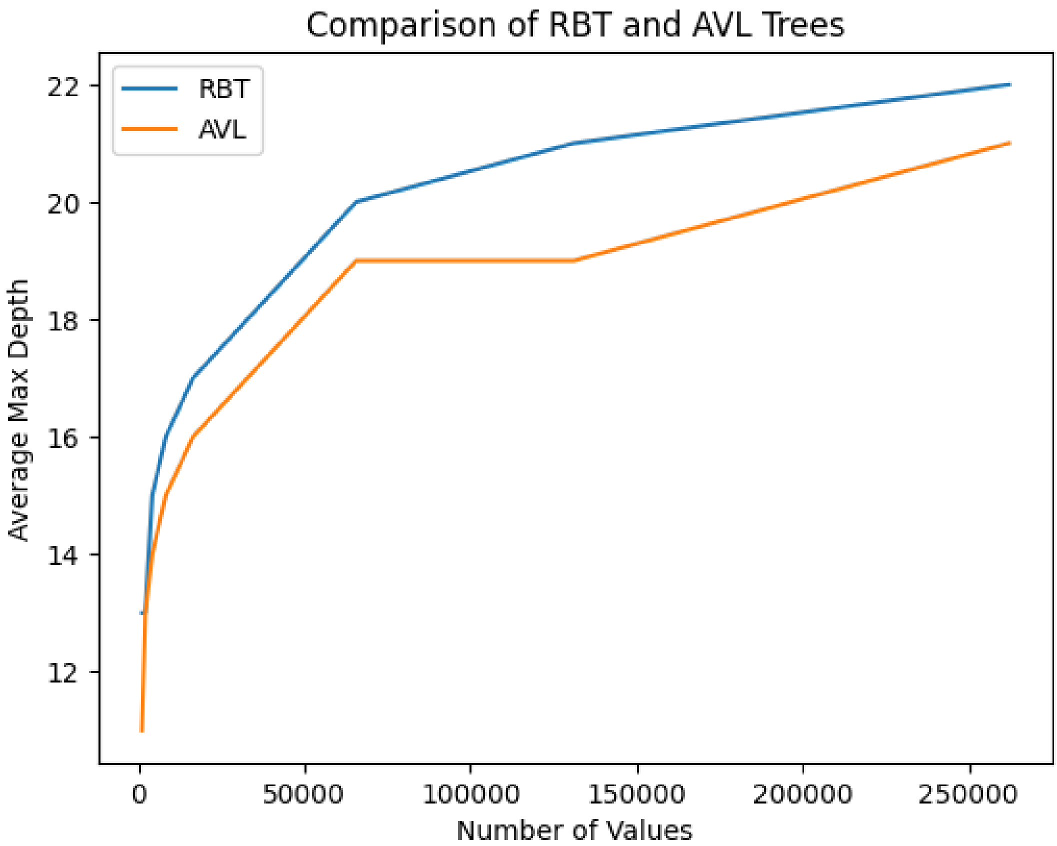

Рисунок 1.2. Сравнение максимальной глубины для RBT и AVL деревьев с сортированными данными

- На рисунках 1.3-1.4 представлены графики зависимости среднего времени удаления от количества элементов в изначальном дереве для RBT дерева и AVL дерева. По ним видно, что удаление элемента занимает больше времени в AVL дереве. Это связано с более сложной балансировкой. Однако в нашем примере разница крайне мала и сделать выводы по ней сложно.

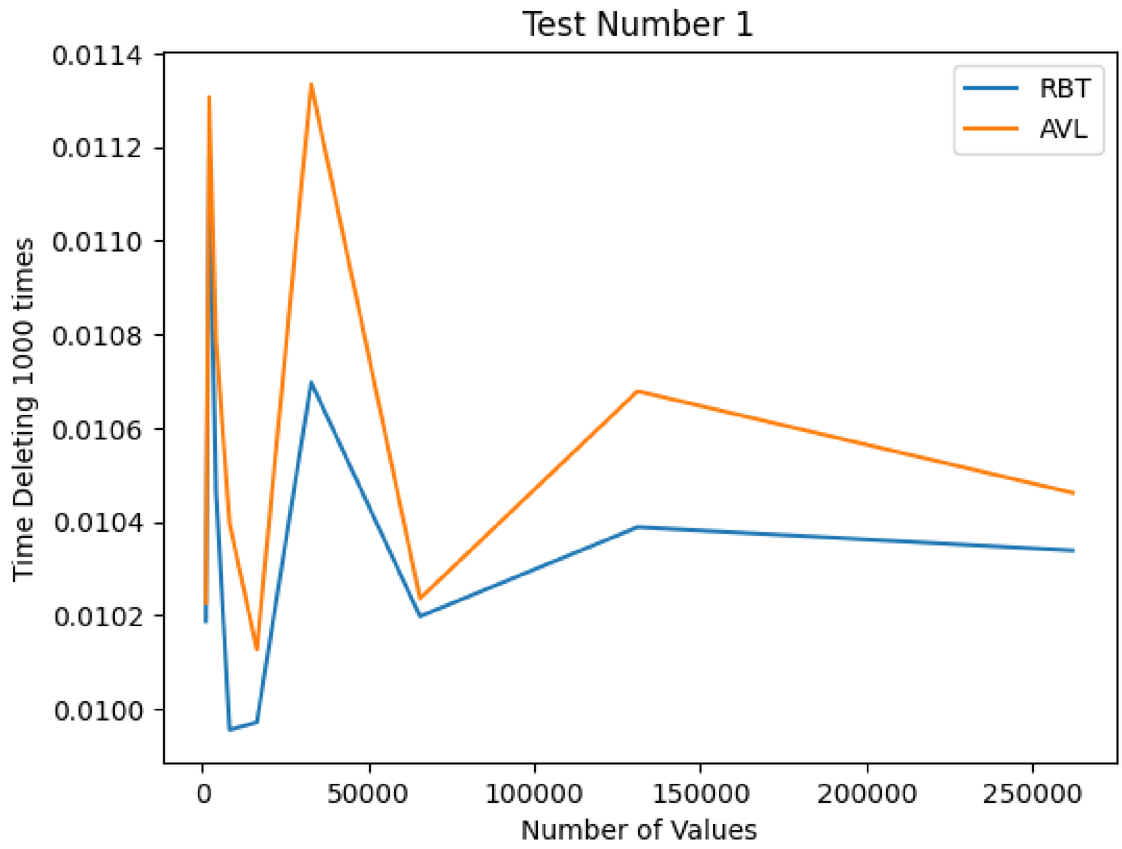

Рисунок 1.3. Сравнение времени удаления элементов из RBT и AVL деревьев

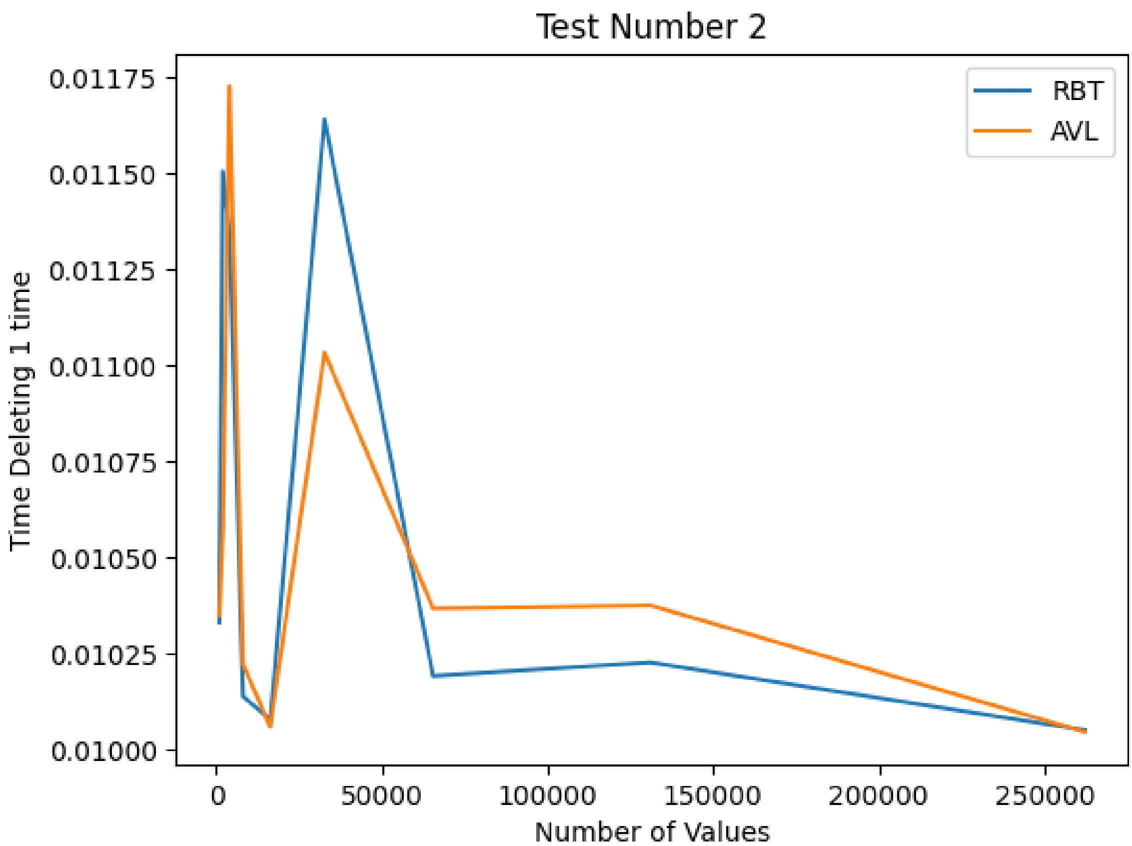

Рисунок 1.4. Сравнение времени удаления элементов из RBT и AVL деревьев с сортированными данными

- На рисунках 1.5-1.6 представлены графики зависимости среднего времени поиска от количества элементов в изначальном дереве для RBT дерева и AVL дерева. Время поиска будет примерно одинаково в обоих деревьях, так как оба они сбалансированы.

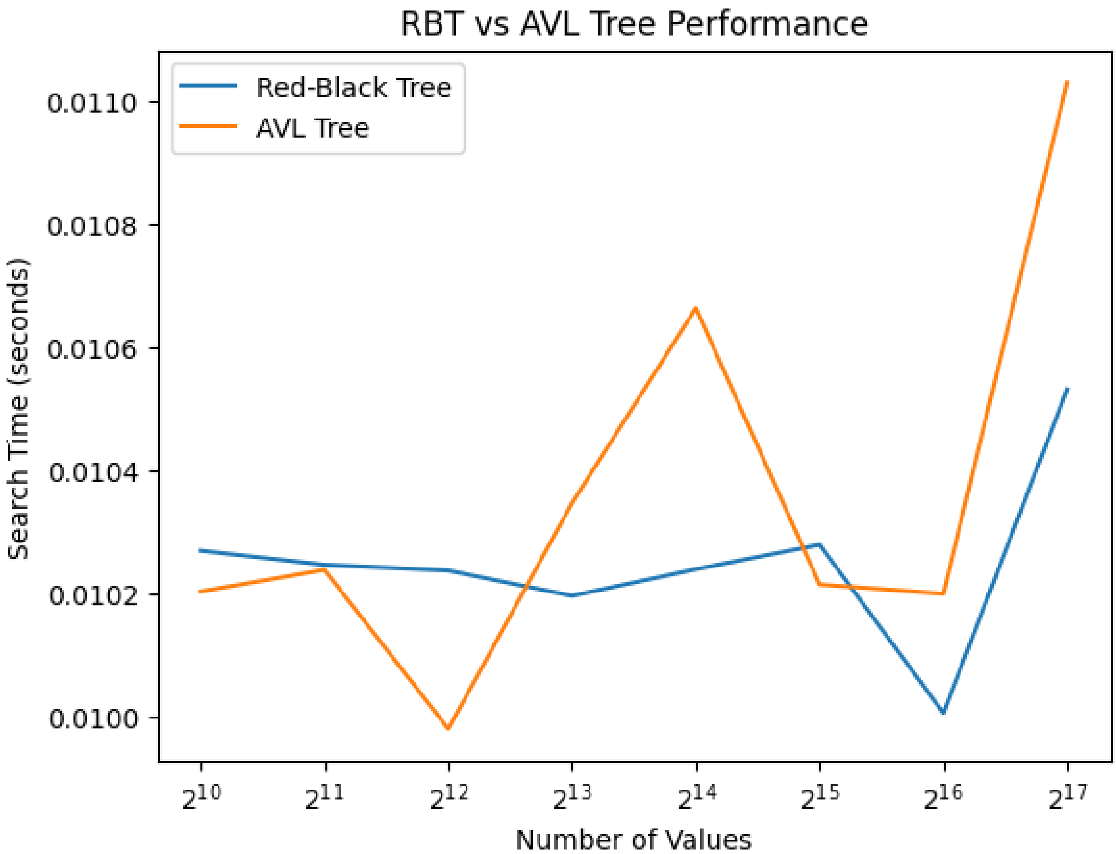

Рисунок 1.5. Зависимость времени поиска от кол-ва элементов в RBT и AVL деревьях

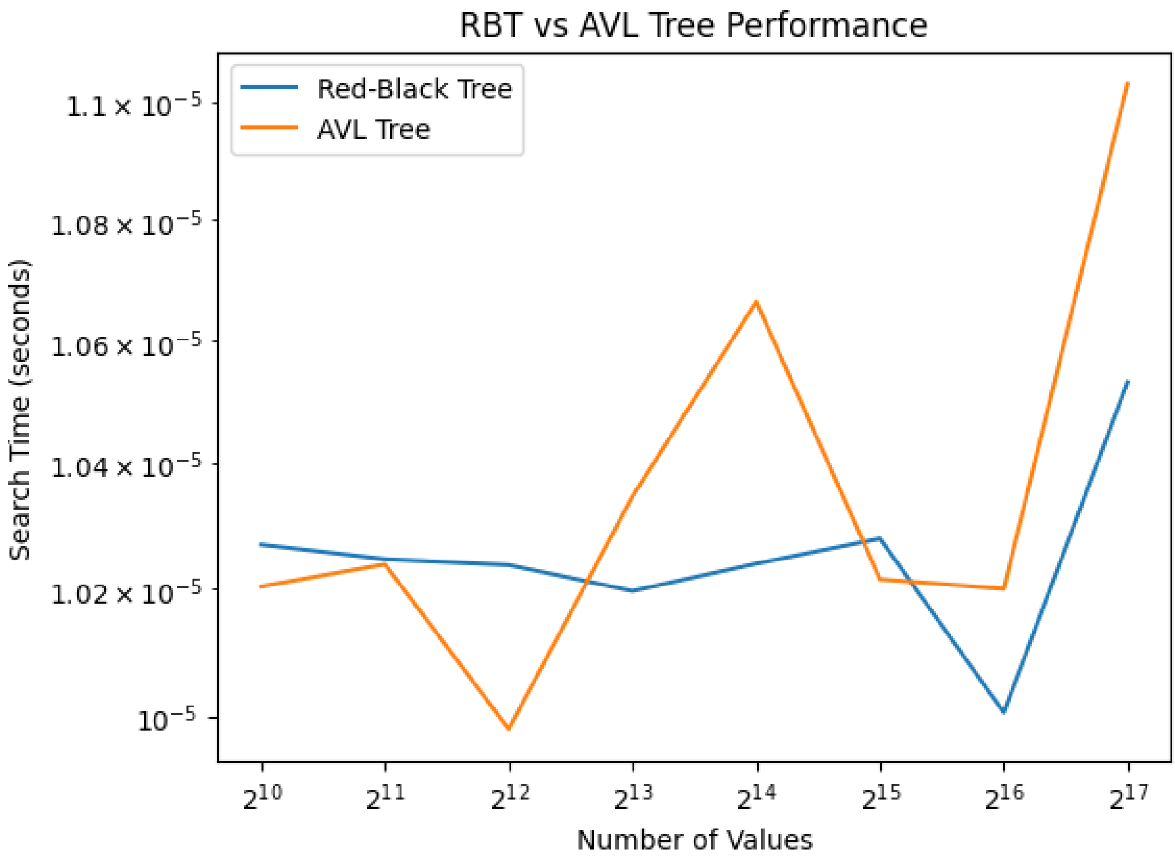

Рисунок 1.6. Зависимость времени поиска от кол-ва элементов в RBT и AVL деревьях с сортированными данными

- График зависимости среднего времени вставки от количества элементов в изначальном дереве для RBT и AVL дерева. Вставка элемента занимает чуть больше времени в AVL дереве, что может быть связано со сложностью балансировки.

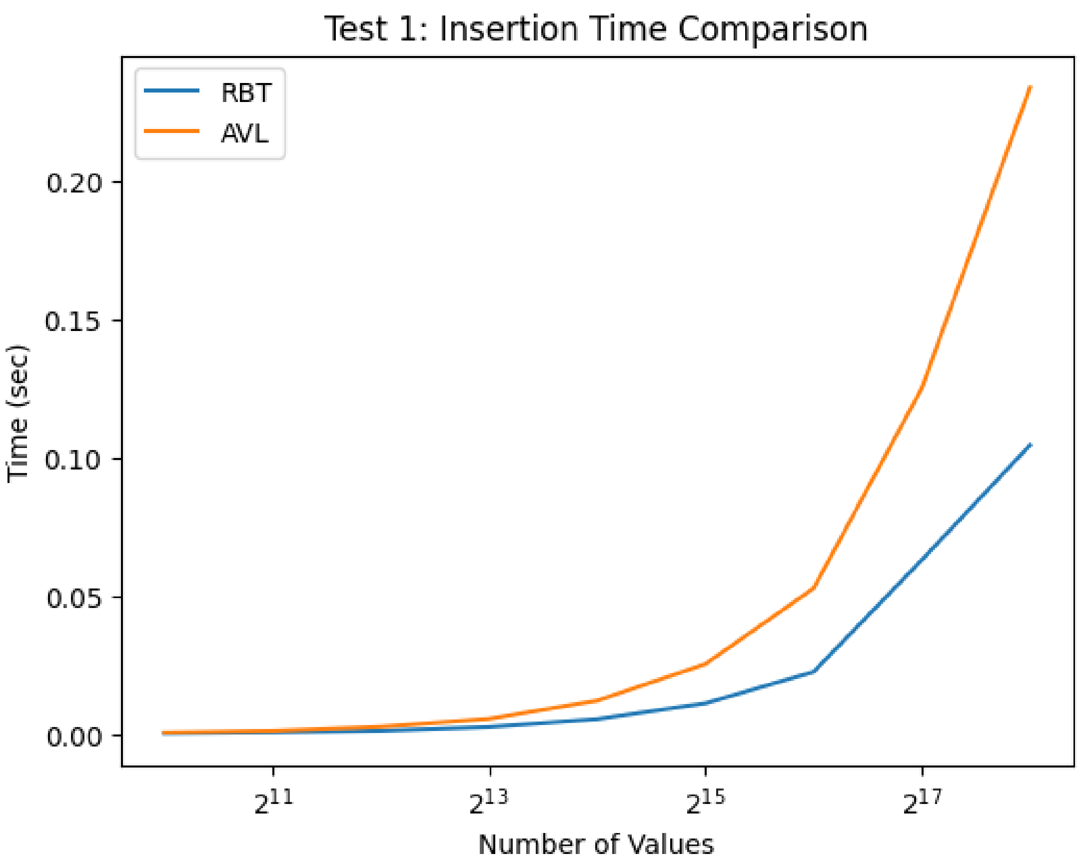

Рисунок 1.7. Зависимость времени вставки от кол-ва элементов в RBT и AVL деревьях

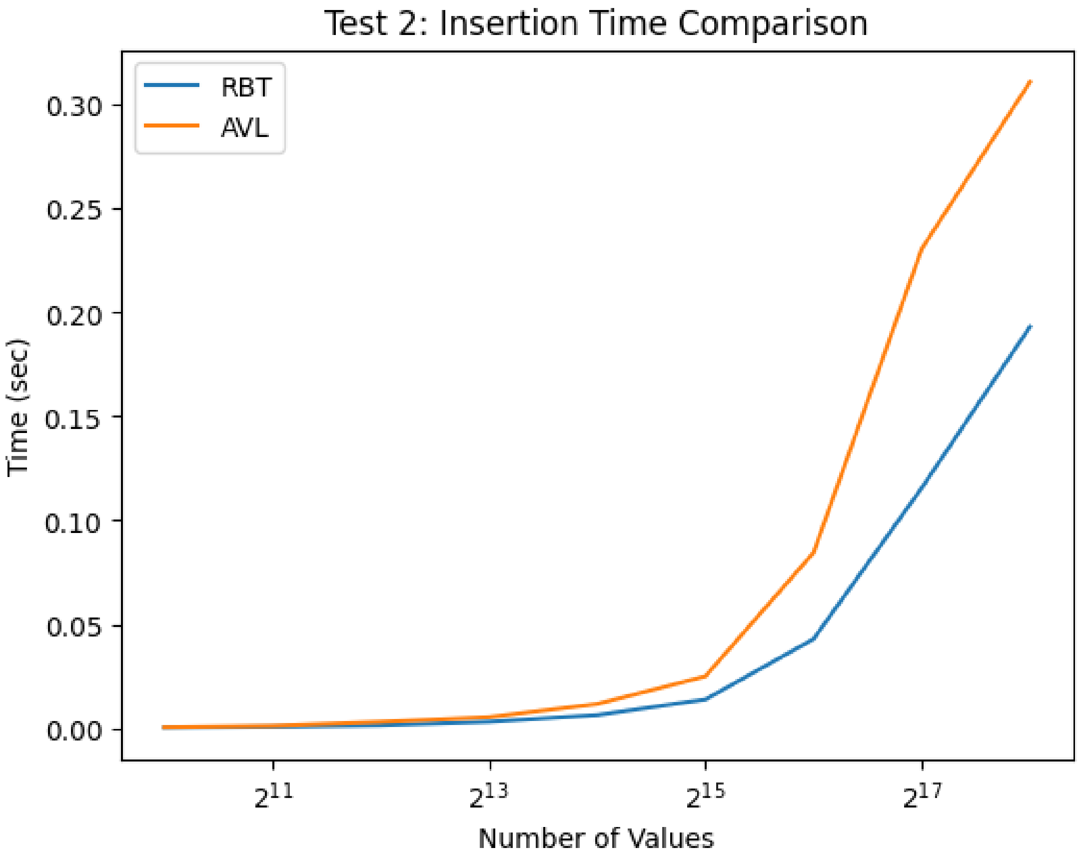

Рисунок 1.8. Зависимость времени вставки от кол-ва элементов в RBT и AVL деревьях с сортированными данными

Заключение.

В результате выполнения лабораторной работы было проведено сравнение AVL-дерева и красно-черного дерева. Обе структуры данных представляют собой бинарные деревья поиска, которые обеспечивают быстрый поиск, вставку и удаление элементов.

Поскольку красно-чёрное дерево, в худшем случае, выше, поиск в нём медленнее, чем в AVL-дереве.

Вставка требует до 2 поворотов в обоих видах деревьев. Однако из-за большей высоты красно-чёрного дерева вставка может занимать больше времени.

Удаление из красно-чёрного дерева требует до 3 поворотов, в АВЛ-дереве оно может потребовать числа поворотов до глубины дерева (до корня). Поэтому удаление из красно-чёрного дерева быстрее, чем из АВЛ-дерева. Тем не менее, тесты показывают, что АВЛ-деревья быстрее красно-чёрных во всех операциях.

Таким образом, при выборе структуры данных для конкретной задачи необходимо учитывать особенности работы алгоритмов и требования к производительности. Если требуется быстрое выполнение операций вставки и удаления в худшем случае, то AVL-дерево является более подходящим выбором. Если же необходима возможность эффективной балансировки дерева, то красно-черное дерево может быть более эффективным выбором.