МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра систем автоматизированного проектирования

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Самобалансирующие двоичные деревья поиска

Студентка гр. 3353	____________	Родачина А. А.
Преподаватель	____________	Пестерев Д. О.

Санкт-Петербург
2024

Оглавление

Теоретическая часть

Определение АВЛ дерева

АВЛ-дерево — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Алгоритм вставки/удаления с последующей балансировкой АВЛ-дерева

Алгоритм вставки для АВЛ-дерева будет отличаться от алгоритма для бинарного дерева тем, что после вставки в АВЛ-дереве обязательна балансировка, иначе не будет выполняться главное условие АВЛ-дерева:

Вставка: чтобы вставить узел в дерево, нужно пройти от его начала вниз, на каждом шаге сравнивая значение нового узла с текущими. Алгоритм доходит до конца какого-либо поддерева и делает новый узел правым или левым его потомком в зависимости от значения. Так сохраняется главное правило двоичного дерева поиска — требование к расположению элементов по значению.

Балансировка: если разница в количестве уровней становится равна 2 или –2, запускается балансировка: связи между предками и потомками изменяются и перестраиваются так, чтобы сохранить правильную структуру. Обычно для этого какой-либо из узлов «поворачивается» влево или вправо, то есть меняет свое расположение. Поворот может быть простым, когда расположение изменяет только один узел, или большим: при нем два узла разворачиваются в разные стороны.

Алгоритм удаления для АВЛ-дерева так же будет основан на алгоритме для бинарного дерева, но с некоторыми правками.

Удаление: в дереве ищется узел, который нужно удалить. Если такого узла нет, ничего не делается. Если он находится, надо пройти по правому поддереву удаляемого узла и найти в нем узел с самым маленьким значением (min). После этого удаляемый узел нужно заменить на узел min, и структура дерева перестроится. Если правого поддерева у удаляемого узла нет, вместо min на место узла подставляется его левый потомок. Если левого потомка тоже нет, значит, удаляемый узел — лист, значит его можно просто удалить.

После удаления так же производится балансировка.

Верхняя оценка высоты АВЛ-дерева

Сначала распишем формулу для определения минимального количества узлов (N_h) для АВЛ-дерева с высотой h:

$$N_h = N_{h-1} + N_{h-2} + 1$$

Сразу можно посчитать количество узлов для АВЛ-дерева с высотой 0 и 1:

$N_0 = 1$ (дерево с высотой 0 имеет 1 узел — корень)

$N_1 = 2$ (дерево с высотой 1 имеет корень и один потомок)

Формулу для N_h можно выразить через число Фибоначчи:

$$N_h = F_{h+2} - 1 \ (F_0 = 0, F_1 = 1, F_k = F_{k-1} + F_{k-2})$$

Пусть n – общее количество узлов в дереве, тогда n ≥ N_h, подставим:

$$n \geq F_{h+2} - 1$$

Запишем число Фибоначчи через формулу Бине:

$$F_k = \frac{\varphi^k - (-\varphi)^{-k}}{\sqrt{5}}, \text{ где } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618 \text{ (золотое сечение)}$$

Для больших k число $(-\varphi)^{-k}$ будет очень маленьким, так что формулу можно переписать как $F_k \approx \frac{\varphi^k}{\sqrt{5}}$. Подставим в формулу с n:

$$n \geq \frac{\varphi^{h+2}}{\sqrt{5}} - 1 => \sqrt{5}(n+1) \geq \varphi^{h+2}$$

$\log_\varphi \sqrt{5}(n+1) \geq h + 2$, *при асимптотической оценке константы опускаются:*

$$h \leq log(n)$$

Итоговая верхняя оценка АВЛ-дерева: h=O(log(n))

Определение красно-черного дерева

Красно-черное дерево - сбалансированное двоичное дерево поиска, которое гарантирует логарифмический рост высоты дерева с помощью черных и красных узлов.

Алгоритм вставки/удаления с последующей балансировкой красно-черного дерева

Алгоритм вставки для красно-черного дерева состоит из двух этапов – добавление нового узла как в бинарном дереве и восстановление свойств красно-черного дерева. Добавляем новый узел (по умолчанию он всегда красный) и назначаем ему черные листья (nil). Теперь важно восстановить свойства красно-черного дерева (перекрашивание узлов или повороты).

1. Если у нового узла родитель черный, то балансировка не требуется.
2. Если родитель красный, нарушается свойство о недопустимости двух красных узлов подряд.
    a. Если дядя (брат родителя) тоже красный, выполняется перекрашивания родителя и дяди в черный, а деда – в красный. Повторяется проверка для деда.
    b. Если дядя – черный, требуются повороты для восстановления баланса:

        b.1 Если новый узел справа от родителя – делаем левый поворот, чтобы новый узел стал левым ребенком.

        b.2 Если новый узел слева от родителя – делаем правый поворот вокруг деда. Родитель становится черным, дед – красным.

Алгоритм удаления элемента для красно-черного дерева начинается с рассмотрения трех случаев, в зависимости от количества его детей. Если их нет, то изменяем указатель на элемент у родителя на nil. Если у него только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину. Теперь стоит проверить балансировку дерева (рассматривается только при удалении черной вершины):

1. Если брат ребёнка удаленного элемента красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас ребенок имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.
2. Если брат текущей вершины был чёрным, то получаем три случая:
    a. Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через брата, но добавит один к числу чёрных узлов на путях, проходящих через ребенка, восстанавливая тем самым влияние удаленного чёрного узла. Таким образом, после удаления

вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.

b. Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у ребенка есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни ребенок, ни его отец не влияют на эту трансформацию.

c. Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. У ребенка удаленной вершины теперь появился дополнительный чёрный предок: либо родитель стал чёрным, или он и был чёрным и брат был добавлен в качестве чёрного дедушки. Таким образом, проходящие через ребенка пути проходят через один дополнительный чёрный узел. Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

Верхняя оценка высоты красно-черного дерева

Одним из главных свойств красно-черного дерева - каждый путь от корня к листу содержит одинаковое количество чёрных узлов. Это количество называется черной высотой h_b. На любом пути от корня к листу не может быть двух подряд красных узлов. Максимальная высота дерева (общее количество узлов на пути от корня до листа) ограничена вдвое большей чёрной высотой:

$$h \leq 2 * h_b$$

Чёрная высота связана с количеством узлов следующим образом: в дереве с n узлами минимальное количество узлов достигается, если все узлы чёрные:

$$n \geq 2^{h_b} - 1$$

Отсюда, подставляя это значение в ограничение на общую высоту:

$$h_b \leq 2 * \log_2(n + 1)$$

Так как константы при асимптотической оценке опускаются, итоговая верхняя оценка будет равна:

$$O(\log(n))$$

Практическая часть

1. Структура, балансировка, операции поиска/вставки/удаления

*Смотреть раздел с кодом

```
Бинарное дерево:
└── 2
    ├── 6
    │   ├── 8
    │   │   ├── 9
    │   │   │   ├── 10
    │   │   │   │   ├── 14
    │   │   └── 7
    │   └── 3
    │       ├── 5
    └── 1
Найти 5: Найден
Упорядоченный массив после удаления 9:
1 2 3 5 6 7 8 10 14

Бинарное дерево:
└── 2
    ├── 6
    │   ├── 8
    │   │   ├── 10
    │   │   │   ├── 14
    │   │   └── 7
    │   └── 3
    │       ├── 5
    └── 1
```

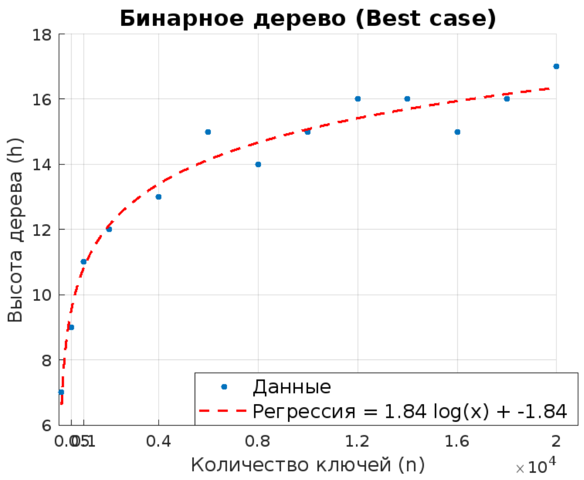
```
АВЛ-дерево:
└── 6
    ├── 9
    │   ├── 10
    │   │   ├── 14
    │   └── 8
    │       └── 7
    └── 3
        ├── 5
        └── 2
            └── 1

Найти 10: true
Найти 100: false

Дерево после удаления 8:
└── 6
    ├── 9
    │   ├── 10
    │   │   ├── 14
    │   └── 7
    └── 3
        ├── 5
        └── 2
            └── 1
```

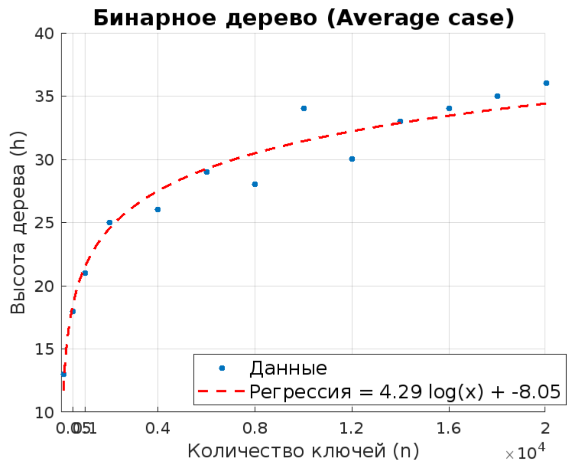
```
Красно-черное дерево:
└── [B]6
    ├── [R]9
    │   ├── [B]10
    │   │   ├── [R]14
    │   └── [B]8
    │       └── [R]7
    └── [R]3
        ├── [B]5
        └── [B]2
            └── [R]1

Найти 1: true
Найти 100: false

Дерево после удаления 6:
└── [B]7
    ├── [R]9
    │   ├── [B]10
    │   │   ├── [R]14
    │   └── [B]8
    └── [R]3
        ├── [B]5
        └── [B]2
            └── [R]1
```

2. Зависимость высоты дерева поиска от количества ключей (значение ключа – случайная величина)

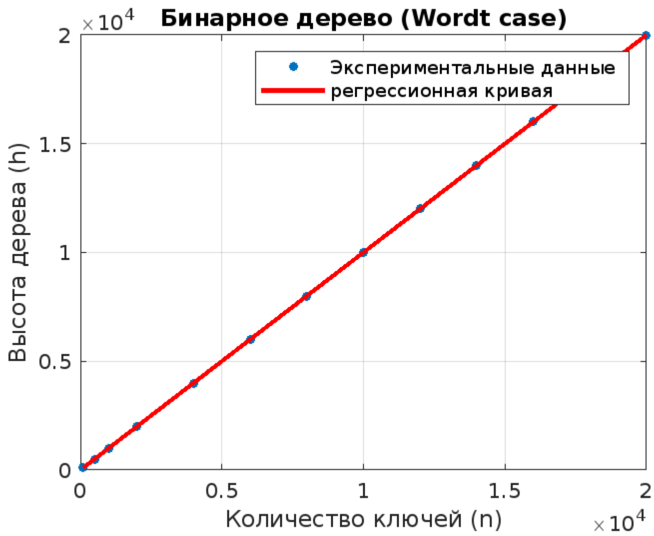
Количество ключей	Высота дерева (Best case)	Высота дерева (Average case)	Высота дерева (Worst case)
100	7	13	100
500	9	18	500
1000	11	21	1000
2000	12	25	2000
4000	13	26	4000
6000	15	29	6000
8000	14	28	8000
10000	15	34	10000
12000	16	30	12000
14000	16	33	14000
16000	15	34	16000
18000	16	35	18000
20000	17	36	20000

Возьмем диапазон количества ключей – от 100 до 20000. Графики построены для трех случаев, по взятым экспериментальным данным из таблицы.





Для лучшего и среднего случая регрессионная кривая, построенная по экспериментальным данным, будет описываться через логарифмические функции: $1.84 \log n - 1.84$ и $4.29 \log n - 8.05$. Это соответствует теоретической оценке высоты дерева – O(log(n)).



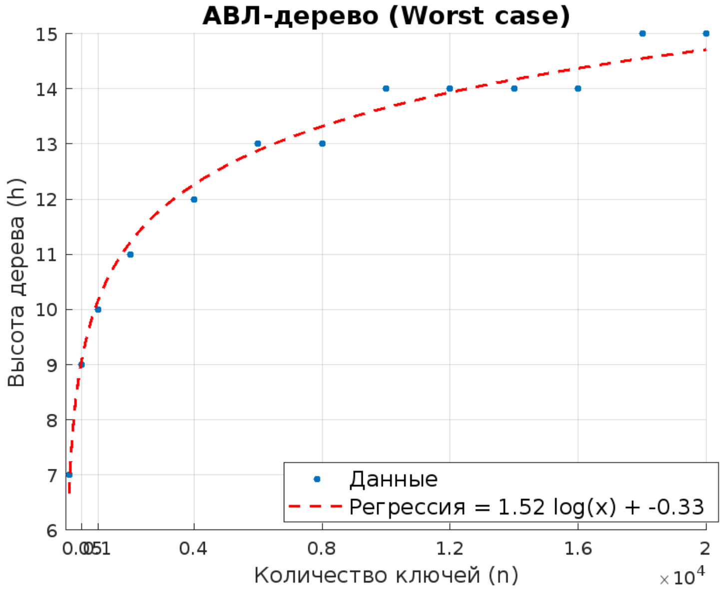
В худшем случае дерево приобретает линей вид, следовательно высота дерева будет равна количеству ключей: O(n). Это видно и на графике с экспериментальными данными.

3.1- 4. Зависимость АВЛ-дерева от количества ключей (значения ключей монотонно возрастают)

Количество ключей	Высота дерева (Best case)
100	7
500	9
1000	10
2000	11
4000	12
6000	13

8000	13
10000	14
12000	14
14000	14
16000	14
18000	15
20000	15

Проверим свойство балансировки у АВЛ-дерева. Возьмем значения ключей, которые монотонно возрастают:



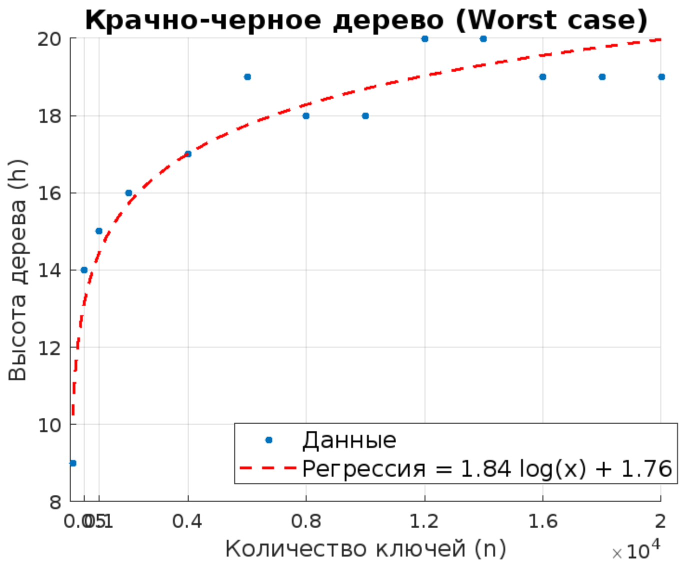
Видно, что регрессионная кривая, описывающая точки с экспериментальными данными, - логарифмическая функция: $1.52 \log n - 0.33$. Она соответствует верхней оценке высоты АВЛ-дерева – O(log(n)).

3.2- 4. Зависимость красно-черного дерева от количества ключей (значения ключей монотонно возрастают)

Количество ключей	Высота дерева (Best case)
100	9
500	14
1000	15
2000	16
4000	17
6000	19
8000	18
10000	18
12000	20

14000	20
16000	19
18000	19
20000	19

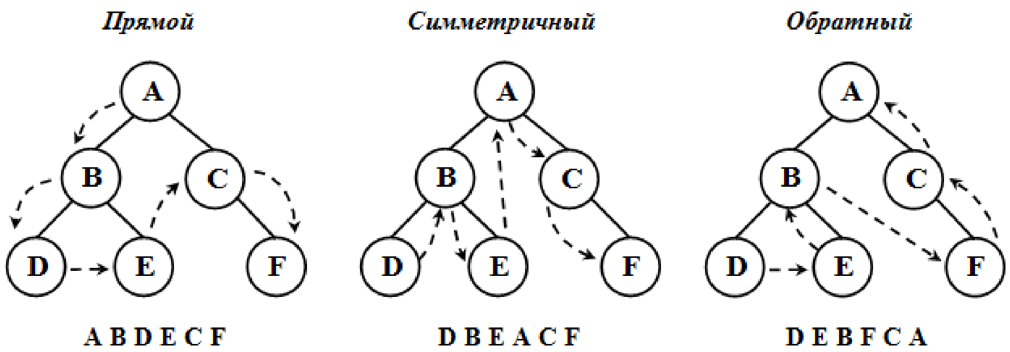
Проверим свойство балансировки у красно-черного дерева. Возьмем значения ключей, которые монотонно возрастают:



Видно, что регрессионная кривая, описывающая точки с экспериментальными данными, - логарифмическая функция: $1.84 \log n + 1.76$. Она соответствует верхней оценке высоты красно-черного дерева – O(log(n)).

5. Обходы в глубину и обход в ширину двоичного дерева

Обходы в глубину:



1. Прямой:

1. Посетить текущий узел.
2. Рекурсивно обойти левое поддерево.
3. Рекурсивно обойти правое поддерево.

2. Симметричный:

1. Рекурсивно обойти левое поддерево.

2. Посетить текущий узел.

3. Рекурсивно обойти правое поддерево.

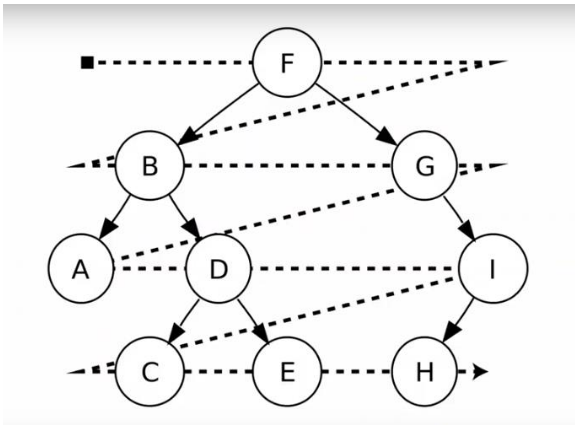
3. Обратный:

1. Рекурсивно обойти левое поддерево.

2. Рекурсивно обойти правое поддерево.

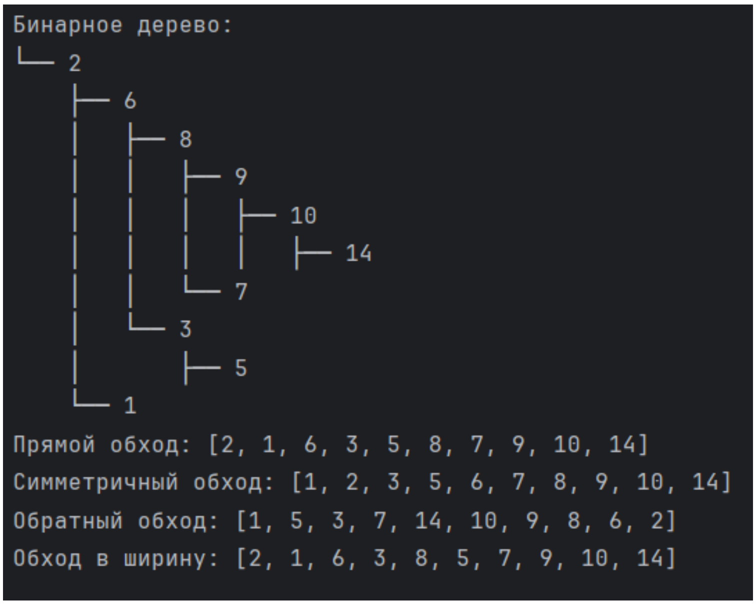
3. Посетить текущий узел.

Обход в ширину:



Обход в ширину посещает узлы уровня за уровнем, начиная с корня. Узлы текущего уровня посещаются до перехода к следующему уровню.

Массив для примера: [2, 6, 3, 8, 5, 9, 10, 1, 14, 7]



Обход для бинарного дерева

```
АВЛ-дерево:
└── 6
    ├── 9
    │   ├── 10
    │   │   ├── 14
    │   └── 8
    │       └── 7
    └── 3
        ├── 5
        └── 2
            └── 1
Прямой обход: [6, 3, 2, 1, 5, 9, 8, 7, 10, 14]
Симметричный обход: [1, 2, 3, 5, 6, 7, 8, 9, 10, 14]
Обратный обход: [1, 2, 5, 3, 7, 8, 14, 10, 9, 6]
Обход в ширину: [6, 3, 9, 2, 5, 8, 10, 1, 7, 14]
```

Обход для АВЛ-дерева

```
Красно-черное дерево:
└── [B]6
    ├── [R]9
    │   ├── [B]10
    │   │   ├── [R]14
    │   └── [B]8
    │       └── [R]7
    └── [R]3
        ├── [B]5
        └── [B]2
            └── [R]1
Прямой обход: [6, 3, 2, 1, 5, 9, 8, 7, 10, 14]
Симметричный обход: [1, 2, 3, 5, 6, 7, 8, 9, 10, 14]
Обратный обход: [1, 2, 5, 3, 7, 8, 14, 10, 9, 6]
Обход в ширину: [6, 3, 9, 2, 5, 8, 10, 1, 7, 14]
```

Обход для красно-черного дерева

Вывод

Посмотрев наглядно, как высота бинарного дерева зависит от количества ключей, можно сделать вывод, что теоретическая логарифмическая оценка высоты сходится с регрессионной кривой, описывающей экспериментальные значения. Также мы выяснили, что АВЛ-дерево и красно-черное дерево с помощью самобалансировки лучше работают, так как даже для худшего случая (монотонно возрастающих ключей) оценка высоты будет логарифмической.

Код

Вставка/удаление/поиск для бинарного дерева

```
class Node {
    int key;
    Node left, right;

    public Node(int key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }
}

class BinarySearchTree {
    public Node root;

    public BinarySearchTree() {
        root = null;
    }

    // Вставка узла
    public void insert(int key) {
        root = insertRec(root, key);
    }

    private Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        return root;
    }

    // Поиск узла
    public boolean search(int key) {
        return searchRec(root, key) != null;
    }

    private Node searchRec(Node root, int key) {
        if (root == null || root.key == key) {
            return root;
```

```
        }
        if (key < root.key) {
            return searchRec(root.left, key);
        }
        return searchRec(root.right, key);
    }

    // Удаление узла
    public void delete(int key) {
        root = deleteRec(root, key);
    }

    private Node deleteRec(Node root, int key) {
        if (root == null) {
            return root;
        }
        if (key < root.key) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.key) {
            root.right = deleteRec(root.right, key);
        } else {
            // Узел с одним или без детей
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }
            // Узел с двумя детьми: получить наименьший узел из правого
поддерева
            root.key = minValue(root.right);
            root.right = deleteRec(root.right, root.key);
        }
        return root;

    }

    private int minValue(Node root) {
        int minValue = root.key;
        while (root.left != null) {
            root = root.left;
            minValue = root.key;
        }
        return minValue;
    }

    public void inorder() {
        inorderRec(root);
        System.out.println();
    }

    private void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.key + " ");
            inorderRec(root.right);
        }
    }

    public void printTree() {
        printTree(root, "", true);
    }

    private void printTree(Node node, String indent, boolean last) {
```

```
        if (node != null) {
            System.out.println(indent + (last ? "└── " : "├── ") + node.key);

            indent += last ? "    " : "|    ";

            printTree(node.right, indent, false);
            printTree(node.left, indent, true);
        }
    }

    public static void picTree_bst(BinarySearchTree tree) {
        System.out.println("\nБинарное дерево:");
        tree.printTree();
    }

    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};

        for (int k : keys) {
            bst.insert(k);
        }

        picTree_bst(bst);

        // Поиск узла
        int key = 5;
        System.out.println("Search for " + key + ": " + (bst.search(key) ?
"Found" : "Not Found"));

        // Удаление узла
        int delkey = 9;
        bst.delete(delkey);
        System.out.println("Inorder traversal after deleting "+delkey+":");
        bst.inorder();
        picTree_bst(bst);
    }
}
```

Вставка/удаление/поиск для АВЛ-дерева

```
import java.util.*;

public class AVL_tree {

    // Узел АВЛ-дерева
    static class AVLNode {
        int value, height;
        AVLNode left, right;

        AVLNode(int value) {
            this.value = value;
            this.height = 1; // Высота нового узла равна 1
        }
    }

    // Реализация АВЛ-дерева
    static class AVLTree {
        AVLNode root;

        int height(AVLNode node) {
            return node == null ? 0 : node.height;
```

```
        }

        int getBalance(AVLNode node) {
            return node == null ? 0 : height(node.left) - height(node.right);
        }

        AVLNode rotateRight(AVLNode y) {
            AVLNode x = y.left;
            AVLNode T2 = x.right;

            x.right = y;
            y.left = T2;

            y.height = Math.max(height(y.left), height(y.right)) + 1;
            x.height = Math.max(height(x.left), height(x.right)) + 1;

            return x;
        }

        AVLNode rotateLeft(AVLNode x) {
            AVLNode y = x.right;
            AVLNode T2 = y.left;

            y.left = x;
            x.right = T2;

            x.height = Math.max(height(x.left), height(x.right)) + 1;
            y.height = Math.max(height(y.left), height(y.right)) + 1;

            return y;
        }

        AVLNode insert(AVLNode node, int value) {
            if (node == null) return new AVLNode(value);

            if (value < node.value) {
                node.left = insert(node.left, value);
            } else if (value > node.value) {
                node.right = insert(node.right, value);
            } else {
                return node; // Дубликаты игнорируются
            }

            node.height = Math.max(height(node.left), height(node.right)) +
1;

            int balance = getBalance(node);

            // Балансировка
            if (balance > 1 && value < node.left.value) {
                return rotateRight(node);
            }
            if (balance < -1 && value > node.right.value) {
                return rotateLeft(node);
            }
            if (balance > 1 && value > node.left.value) {
                node.left = rotateLeft(node.left);
                return rotateRight(node);
            }
            if (balance < -1 && value < node.right.value) {
                node.right = rotateRight(node.right);
                return rotateLeft(node);
            }
```

```
            return node;
        }

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

        int height() {
            return height(root);
        }

        AVLNode delete(AVLNode root, int value) {
            if (root == null) return root;

            // Удаление
            if (value < root.value) {
                root.left = delete(root.left, value);
            } else if (value > root.value) {
                root.right = delete(root.right, value);
            } else {
                if ((root.left == null) || (root.right == null)) {
                    AVLNode temp = root.left != null ? root.left :
root.right;
                    root = temp;
                } else {
                    AVLNode temp = minValueNode(root.right);
                    root.value = temp.value;
                    root.right = delete(root.right, temp.value);
                }
            }

            if (root == null) return root;

            // Обновляем высоту
            root.height = Math.max(height(root.left), height(root.right)) +
1;

            // Балансировка
            int balance = getBalance(root);

            if (balance > 1 && getBalance(root.left) >= 0) {
                return rotateRight(root);
            }
            if (balance > 1 && getBalance(root.left) < 0) {
                root.left = rotateLeft(root.left);
                return rotateRight(root);
            }
            if (balance < -1 && getBalance(root.right) <= 0) {
                return rotateLeft(root);
            }
            if (balance < -1 && getBalance(root.right) > 0) {
                root.right = rotateRight(root.right);
                return rotateLeft(root);
            }

            return root;
        }

        void delete(int value) {
            root = delete(root, value);
        }
```

```
        AVLNode minValueNode(AVLNode node) {
            AVLNode current = node;
            while (current.left != null) current = current.left;
            return current;
        }

        boolean search(AVLNode node, int value) {
            if (node == null) return false;

            if (value < node.value) {
                return search(node.left, value);
            } else if (value > node.value) {
                return search(node.right, value);
            } else {
                return true;
            }
        }

        boolean search(int value) {
            return search(root, value);
        }

        public void printTree() {
            printTree(root, "", true);
        }

        private void printTree(AVLNode node, String indent, boolean last) {
            if (node != null) {
                System.out.println(indent + (last ? "└── " : "├── ") +
node.value);

                indent += last ? "    " : "│   ";

                printTree(node.right, indent, false);
                printTree(node.left, indent, true);
            }
        }

    }
    public static void picTree_AVL(AVLTree tree) {
        System.out.println("\nАВЛ-дерево:");
        tree.printTree();
    }

    public static void main(String[] args) {
        AVLTree tree = new AVLTree();
        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};

        for (int k : keys) {
            tree.insert(k);
        }

        picTree_AVL(tree);

        System.out.println("\nПоиск элемента 10: " + tree.search(10));
        System.out.println("Поиск элемента 100: " + tree.search(100));

        tree.delete(8);
        System.out.println("\nДерево после удаления 8:");
        tree.printTree();
    }
}
```

Вставка/удаление/поиск для красно-черного дерева

```
import java.util.*;

public class RandB_tree {

    // Реализация Красно-черного дерева
    static class RedBlackTree {
        static class Node {
            int value;
            boolean isRed;
            Node left, right, parent;

            Node(int value) {
                this.value = value;
                this.isRed = true; // Новый узел всегда красный
            }
        }

        Node root;

        void insert(int value) {
            Node newNode = new Node(value);
            root = bstInsert(root, newNode);
            fixViolations(newNode);
        }

        private Node bstInsert(Node root, Node newNode) {
            if (root == null) return newNode;

            if (newNode.value < root.value) {
                root.left = bstInsert(root.left, newNode);
                root.left.parent = root;
            } else if (newNode.value > root.value) {
                root.right = bstInsert(root.right, newNode);
                root.right.parent = root;
            }

            return root;
        }

        private void fixViolations(Node newNode) {
            while (newNode != root && newNode.parent.isRed) {
                Node parent = newNode.parent;
                Node grandparent = parent.parent;

                if (parent == grandparent.left) {
                    Node uncle = grandparent.right;
                    if (uncle != null && uncle.isRed) {
                        grandparent.isRed = true;
                        parent.isRed = false;
                        uncle.isRed = false;
                        newNode = grandparent;
                    } else {
                        if (newNode == parent.right) {
                            rotateLeft(parent);
                            newNode = parent;
                            parent = newNode.parent;
                        }
                        rotateRight(grandparent);
                        boolean temp = parent.isRed;
                        parent.isRed = grandparent.isRed;
                        grandparent.isRed = temp;
```

```
                    newNode = parent;
                }
            } else {
                Node uncle = grandparent.left;
                if (uncle != null && uncle.isRed) {
                    grandparent.isRed = true;
                    parent.isRed = false;
                    uncle.isRed = false;
                    newNode = grandparent;
                } else {
                    if (newNode == parent.left) {
                        rotateRight(parent);
                        newNode = parent;
                        parent = newNode.parent;
                    }
                    rotateLeft(grandparent);
                    boolean temp = parent.isRed;
                    parent.isRed = grandparent.isRed;
                    grandparent.isRed = temp;
                    newNode = parent;
                }
            }
        }
        root.isRed = false;
    }

    private void rotateLeft(Node node) {
        Node temp = node.right;
        node.right = temp.left;
        if (temp.left != null) temp.left.parent = node;
        temp.parent = node.parent;
        if (node.parent == null) root = temp;
        else if (node == node.parent.left) node.parent.left = temp;
        else node.parent.right = temp;
        temp.left = node;
        node.parent = temp;
    }

    private void rotateRight(Node node) {
        Node temp = node.left;
        node.left = temp.right;
        if (temp.right != null) temp.right.parent = node;
        temp.parent = node.parent;
        if (node.parent == null) root = temp;
        else if (node == node.parent.right) node.parent.right = temp;
        else node.parent.left = temp;
        temp.right = node;
        node.parent = temp;
    }

    int height(Node node) {
        if (node == null) return 0;
        return 1 + Math.max(height(node.left), height(node.right));
    }

    int height() {
        return height(root);
    }

    public void printTree() {
        printTree(root, "", true);
    }
```

```
        private void printTree(Node node, String indent, boolean last) {
            if (node != null) {
                String nodeInfo = node.isRed ? "[R]" + node.value : "[B]" +
node.value;
                System.out.println(indent + (last ? "└── " : "├── ") +
nodeInfo);

                indent += last ? "    " : "│   ";

                printTree(node.right, indent, false);
                printTree(node.left, indent, true);
            }
        }

        // Функция для поиска элемента
        public boolean search(Node node, int value) {
            if (node == null) return false;

            if (value < node.value) return search(node.left, value);
            else if (value > node.value) return search(node.right, value);
            else return true;
        }

        public boolean search(int value) {
            return search(root, value);
        }

        // Функция для удаления узла
        public void delete(int value) {
            Node nodeToDelete = findNode(root, value);
            if (nodeToDelete != null) {
                deleteNode(nodeToDelete);
            }
        }

        private Node findNode(Node root, int value) {
            if (root == null || root.value == value) return root;
            if (value < root.value) return findNode(root.left, value);
            return findNode(root.right, value);
        }

        private void deleteNode(Node node) {
            Node child, parent;
            boolean nodeColor;

            if (node.left == null || node.right == null) {
                parent = node.parent;
                child = (node.left != null) ? node.left : node.right;
                nodeColor = node.isRed;

                if (child != null) child.parent = parent;

                if (parent == null) root = child;
                else if (node == parent.left) parent.left = child;
                else parent.right = child;

                if (!nodeColor) {
                    fixDeletion(child, parent);
                }
            } else {
                Node successor = minValueNode(node.right);
                node.value = successor.value;
                deleteNode(successor);
```

```
            }
        }

        private Node minValueNode(Node node) {
            while (node.left != null) node = node.left;
            return node;
        }

        private void fixDeletion(Node node, Node parent) {
            while (node != root && (node == null || !node.isRed)) {
                if (node == parent.left) {
                    Node sibling = parent.right;
                    if (sibling != null && sibling.isRed) {
                        sibling.isRed = false;
                        parent.isRed = true;
                        rotateLeft(parent);
                        sibling = parent.right;
                    }
                    if ((sibling.left == null || !sibling.left.isRed) &&
                            (sibling.right == null || !sibling.right.isRed))
{
                        sibling.isRed = true;
                        node = parent;
                        parent = node.parent;
                    } else {
                        if (sibling.right == null || !sibling.right.isRed) {
                            if (sibling.left != null) sibling.left.isRed =
false;
                            sibling.isRed = true;
                            rotateRight(sibling);
                            sibling = parent.right;
                        }
                        sibling.isRed = parent.isRed;
                        parent.isRed = false;
                        if (sibling.right != null) sibling.right.isRed =
false;
                        rotateLeft(parent);
                        node = root;
                    }
                } else {
                    Node sibling = parent.left;
                    if (sibling != null && sibling.isRed) {
                        sibling.isRed = false;
                        parent.isRed = true;
                        rotateRight(parent);
                        sibling = parent.left;
                    }
                    if ((sibling.left == null || !sibling.left.isRed) &&
                            (sibling.right == null || !sibling.right.isRed))
{
                        sibling.isRed = true;
                        node = parent;
                        parent = node.parent;
                    } else {
                        if (sibling.left == null || !sibling.left.isRed) {
                            if (sibling.right != null) sibling.right.isRed =
false;
                            sibling.isRed = true;
                            rotateLeft(sibling);
                            sibling = parent.left;
                        }
                        sibling.isRed = parent.isRed;
                        parent.isRed = false;
```

```
                        if (sibling.left != null) sibling.left.isRed = false;
                        rotateRight(parent);
                        node = root;
                    }
                }
            }
            if (node != null) node.isRed = false;
        }
    }

    public static void grafik_RandB() {
        int[] sizes = {100, 500, 1000, 2000, 4000, 6000, 8000, 10000, 12000,
14000, 16000, 18000, 20000};
        System.out.printf("%-10s %-10s\n", "Keys", "AVL");
        for (int n : sizes) {
            RandB_tree.RedBlackTree rbTree = new RandB_tree.RedBlackTree();

            for (int i = 1; i <= n; i++) {
                rbTree.insert(i);
            }
            System.out.printf("%-10d %-10d\n", n, rbTree.height());
        }
    }

    public static void picTree_RB(RandB_tree.RedBlackTree tree) {
        System.out.println("\nКрасно-черное дерево:");
        tree.printTree();
    }

    public static void main(String[] args) {

        grafik_RandB();

        RandB_tree.RedBlackTree tree = new RandB_tree.RedBlackTree();

        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};

        for (int k : keys) {
            tree.insert(k);
        }

        picTree_RB(tree);

        System.out.println("Дерево после вставки:");
        tree.printTree();

        System.out.println("\nПоиск элемента 1: " + tree.search(1));
        System.out.println("Поиск элемента 100: " + tree.search(100));

        tree.delete(6);
        System.out.println("\nДерево после удаления 6:");
        tree.printTree();
    }
}
```

Обходы для бинарного дерева

```
import java.util.ArrayList;
import java.util.LinkedList;
```

```java
import java.util.List;
import java.util.Queue;

public class orders_bin {
    // Прямой обход (Pre-order)
    static void preOrder(Node node, List<Integer> result) {
        if (node == null) return;
        result.add(node.key);
        preOrder(node.left, result);
        preOrder(node.right, result);
    }

    // Симметричный обход (In-order)
    static void inOrder(Node node, List<Integer> result) {
        if (node == null) return;
        inOrder(node.left, result);
        result.add(node.key);
        inOrder(node.right, result);
    }

    // Обратный обход (Post-order)
    static void postOrder(Node node, List<Integer> result) {
        if (node == null) return;
        postOrder(node.left, result);
        postOrder(node.right, result);
        result.add(node.key);
    }

    // Обход в ширину (BFS)
    static List<Integer> bfs(Node root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Queue<Node> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            result.add(current.key);

            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }

        return result;
    }

    public static void main(String[] args) {

        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};
        BinarySearchTree tree = new BinarySearchTree();
        for (int k : keys) {
            tree.insert(k);
        }
        BinarySearchTree.picTree_bst(tree);
        // Обходы
        List<Integer> preOrderResult = new ArrayList<>();
        preOrder(tree.root , preOrderResult);

        List<Integer> inOrderResult = new ArrayList<>();
        inOrder(tree.root, inOrderResult);

        List<Integer> postOrderResult = new ArrayList<>();
```

```
        postOrder(tree.root, postOrderResult);

        List<Integer> bfsResult = bfs(tree.root);

        // Вывод результатов
        System.out.println("Прямой обход: " + preOrderResult);
        System.out.println("Симметричный обход: " + inOrderResult);
        System.out.println("Обратный обход: " + postOrderResult);
        System.out.println("Обход в ширину: " + bfsResult);
    }
}
```

Обходы для АВЛ-дерева

```
import java.util.*;

public class orders_AVL {
    // Прямой обход (Pre-order)
    static void preOrder(AVL_tree.AVLNode node, List<Integer> result) {
        if (node == null) return;
        result.add(node.value);
        preOrder(node.left, result);
        preOrder(node.right, result);
    }

    // Симметричный обход (In-order)
    static void inOrder(AVL_tree.AVLNode node, List<Integer> result) {
        if (node == null) return;
        inOrder(node.left, result);
        result.add(node.value);
        inOrder(node.right, result);
    }

    // Обратный обход (Post-order)
    static void postOrder(AVL_tree.AVLNode node, List<Integer> result) {
        if (node == null) return;
        postOrder(node.left, result);
        postOrder(node.right, result);
        result.add(node.value);
    }

    // Обход в ширину (BFS)
    static List<Integer> bfs(AVL_tree.AVLNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Queue<AVL_tree.AVLNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            AVL_tree.AVLNode current = queue.poll();
            result.add(current.value);

            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }

        return result;
    }

    public static void main(String[] args) {
```

```
        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};
        AVL_tree.AVLTree tree = new AVL_tree.AVLTree();
        for (int k : keys) {
            tree.insert(k);
        }
        AVL_tree.picTree_AVL(tree);
        // Обходы
        List<Integer> preOrderResult = new ArrayList<>();
        preOrder(tree.root, preOrderResult);

        List<Integer> inOrderResult = new ArrayList<>();
        inOrder(tree.root, inOrderResult);

        List<Integer> postOrderResult = new ArrayList<>();
        postOrder(tree.root, postOrderResult);

        List<Integer> bfsResult = bfs(tree.root);

        // Вывод результатов
        System.out.println("Прямой обход: " + preOrderResult);
        System.out.println("Симметричный обход: " + inOrderResult);
        System.out.println("Обратный обход: " + postOrderResult);
        System.out.println("Обход в ширину: " + bfsResult);
    }
}
```

Обходы для красно-черного дерева

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class orders_RB {
    // Прямой обход (Pre-order)
    static void preOrder(RandB_tree.RedBlackTree.Node node, List<Integer>
result) {
        if (node == null) return;
        result.add(node.value);
        preOrder(node.left, result);
        preOrder(node.right, result);
    }

    // Симметричный обход (In-order)
    static void inOrder(RandB_tree.RedBlackTree.Node node, List<Integer>
result) {
        if (node == null) return;
        inOrder(node.left, result);
        result.add(node.value);
        inOrder(node.right, result);
    }

    // Обратный обход (Post-order)
    static void postOrder(RandB_tree.RedBlackTree.Node node, List<Integer>
result) {
        if (node == null) return;
        postOrder(node.left, result);
        postOrder(node.right, result);
        result.add(node.value);
    }

    // Обход в ширину (BFS)
```

```
    static List<Integer> bfs(RandB_tree.RedBlackTree.Node root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Queue<RandB_tree.RedBlackTree.Node> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            RandB_tree.RedBlackTree.Node current = queue.poll();
            result.add(current.value);

            if (current.left != null) queue.add(current.left);
            if (current.right != null) queue.add(current.right);
        }

        return result;
    }

    public static void main(String[] args) {

        int[] keys = {2, 6, 3, 8, 5, 9, 10, 1, 14, 7};
        RandB_tree.RedBlackTree tree = new RandB_tree.RedBlackTree();
        for (int k : keys) {
            tree.insert(k);
        }
        RandB_tree.picTree_RB(tree);
        // Обходы
        List<Integer> preOrderResult = new ArrayList<>();
        preOrder(tree.root, preOrderResult);

        List<Integer> inOrderResult = new ArrayList<>();
        inOrder(tree.root, inOrderResult);

        List<Integer> postOrderResult = new ArrayList<>();
        postOrder(tree.root, postOrderResult);

        List<Integer> bfsResult = bfs(tree.root);

        // Вывод результатов
        System.out.println("Прямой обход: " + preOrderResult);
        System.out.println("Симметричный обход: " + inOrderResult);
        System.out.println("Обратный обход: " + postOrderResult);
        System.out.println("Обход в ширину: " + bfsResult);
    }
}
```

Ссылка на GitHub

https://github.com/vanda3000/aisd_laba2