Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6

Выполнил студент группы.....КС-38..... Прилепский Артем Сергеевич
Ссылка на репозиторий: .....github.com/news1d/Algorithms_and_structures

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

Дата сдачи: 31.03.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

- В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
- Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
- Массив должен быть помещен в оба вариант двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
- После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
- Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
- После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
- После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

Описание метода/модели.

Бинарное дерево - это структура данных, состоящая из узлов, каждый из которых имеет не более двух дочерних узлов (левый и правый), а также корневого узла, который является вершиной самого верхнего уровня дерева.

В бинарном дереве поиска каждый узел содержит значение, которое больше или равно значению в его левом поддереве и меньше или равно значению в его правом поддереве. Это свойство делает бинарное дерево поиска эффективной структурой данных для хранения и поиска элементов.

AVL-дерево - это сбалансированное бинарное дерево поиска, в котором для каждого узла высота его двух поддеревьев отличается не более чем на 1.

При вставке или удалении элементов из AVL-дерева, если его структура нарушается и какой-либо узел становится несбалансированным (то есть его высота левого и правого поддеревьев отличаются более чем на 1), то выполняется процедура балансировки. Балансировка в AVL-дереве производится путем поворотов поддеревьев вокруг узлов, чтобы восстановить сбалансированность.

Благодаря своей структуре, AVL-деревья обеспечивают быстрый доступ к данным и эффективное выполнение операций поиска, вставки и удаления элементов.

Выполнение задачи.

Для реализации программы был выбран язык программирования Python.

1) Класс узла для бинарного дерева:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

2) Класс бинарного дерева имеет следующий вид:

```
class Tree:
    def __init__(self):
        self.root = None

    # Ищем родительскую вершину, к которой будем добавлять новую вершину
    def __find(self, node, parent, value):
        # Если вершина не существует то
        if node is None:
            return None, parent, False
        # Если вершина с таким значением существует
        if value == node.data:
            return node, parent, True

        # Рекурсивно проходим по левой ветви
        if value < node.data:
            if node.left:
                return self.__find(node.left, node, value)

        # Рекурсивно проходим по правой ветви
        if value > node.data:
            if node.right:
                return self.__find(node.right, node, value)

        return node, parent, False

    # Добавление вершин
    def append(self, obj):
        # Если в дерево пустое, то добавляем вершину в качестве корня
        if self.root is None:
            self.root = obj
            return obj

        # Ищем родительскую вершину, к которой будем добавлять новую вершину
        s, p, fl_find = self.__find(self.root, None, obj.data)

        # Если вершины с таким значением нет и родительская вершина существует
        if not fl_find and s:
            # Если родительская вершина больше добавляемого значения
```

```
                if obj.data < s.data:
                    s.left = obj
                else:
                    s.right = obj

            return obj

        # Удаляем одного потомка
        def __del_one_child(self, s, p):
            # Если удаляемый потомок является левым
            if p.left == s:
                if s.left is None:
                    p.left = s.right
                elif s.right is None:
                    p.left = s.left
            # Если удаляемый потомок является правым
            elif p.right == s:
                if s.left is None:
                    p.right = s.right
                elif s.right is None:
                    p.right = s.left

        # Ищем минимальную вершину
        def __find_min(self, node, parent):
            # Если левый потомок существует, то продолжаем рекурсию
            if node.left:
                return self.__find_min(node.left, node)
            return node, parent

        # Удаляем вершину
        def del_node(self, key):
            # Ищем вершину с нужным значением
            s, p, fl_find = self.__find(self.root, None, key)

            # Вершина не найдена
            if not fl_find:
                return None

            # Если вершина не имеет потомков
            if s.left is None and s.right is None:
                if p.left == s:
                    p.left = None
                elif p.right == s:
                    p.right = None

            # Если вершина имеет только одного потомка
            elif s.left is None or s.right is None:
                self.__del_one_child(s, p)

            # Если вершина имеет обоих потомков
            else:
                # Ищем минимальную вершину в правом потомке
                sr, pr = self.__find_min(s.right, s)
                s.data = sr.data
                self.__del_one_child(sr, pr)

        # Функция для поиска ершины с заданным значением
        def _search_helper(self, node, value):
            # Если дерево пустое или значение найдено
            if node is None or node.data == value:
                return node
            # Если значение меньше значения вершины, рекурсивно ищем в левом потомке
            if value < node.data:
                return self._search_helper(node.left, value)
            else:
                # Если значение больше значения вершины, рекурсивно ищем в правом
потомке
```

```
            return self._search_helper(node.right, value)

    # Поиск вершины
    def search(self, value):
        return self._search_helper(self.root, value)
```

3) Класс узла для AVL дерева имеет схожий вид, за исключением дополнительного параметра height, который отвечает за высоту:

```
class AVLNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1
```

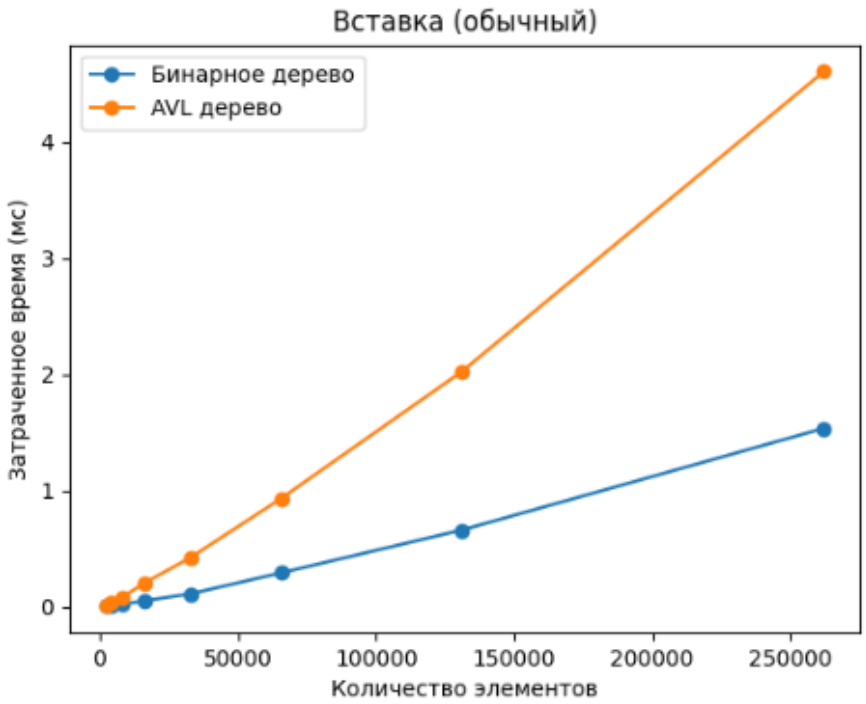
4) Реализованный класс для AVL дерева:

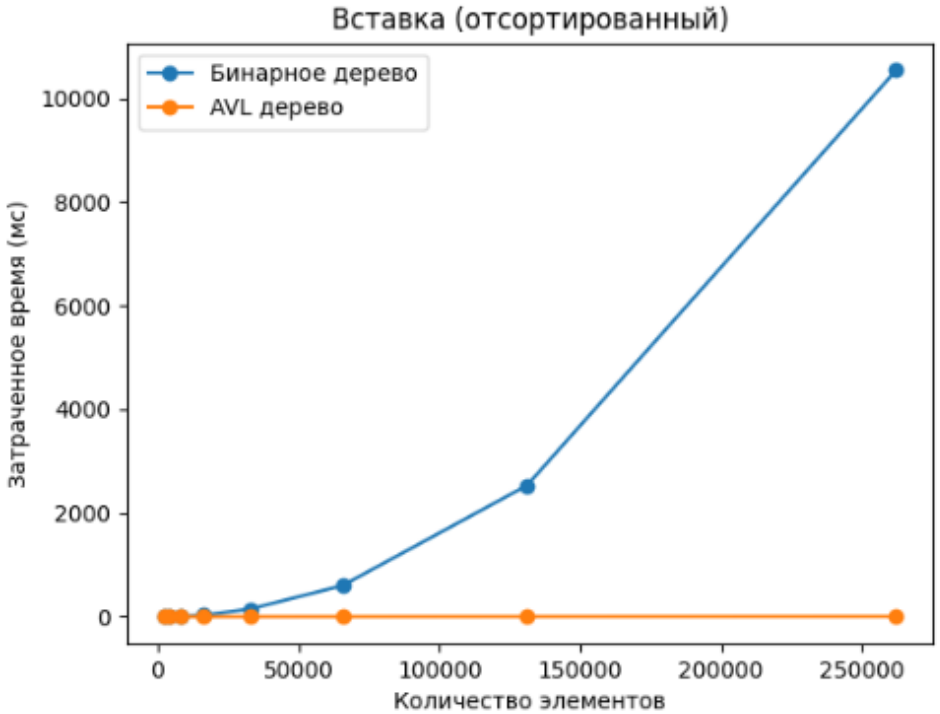
```
class AVLTree:
    def __init__(self):
        self.root = None

    # Вспомогательная функция для получения высоты узла
    def _get_height(self, node):
        if node is None:
            return 0
        return node.height

    # Вспомогательная функция для получения баланс-фактора узла
    def _get_balance_factor(self, node):
        if node is None:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    # Получение узла с минимальным значением из дерева
    def _get_min_value_node(self, node):
        if node is None or node.left is None:
            return node
        return self._get_min_value_node(node.left)

    # Функция для правого поворота
    def _right_rotate(self, node):
        # Новой вершине присваивается значение левого потомка родительской
вершины
        new_root = node.left
        # Левому потомку родительской вершины присваивается значение правого
потомка новой вершины
        node.left = new_root.right
        # Правому потомку новой вершины присваивается значение родительской
вершиной
        new_root.right = node

        # Пересчитываем высоты узлов
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        new_root.height = 1 + max(self._get_height(new_root.left),
self._get_height(new_root.right))

        return new_root

    # Функция для левого поворота
    def _left_rotate(self, node):
        # Новой вершине присваивается значение левого потомка родительской
вершины
        new_root = node.right
```

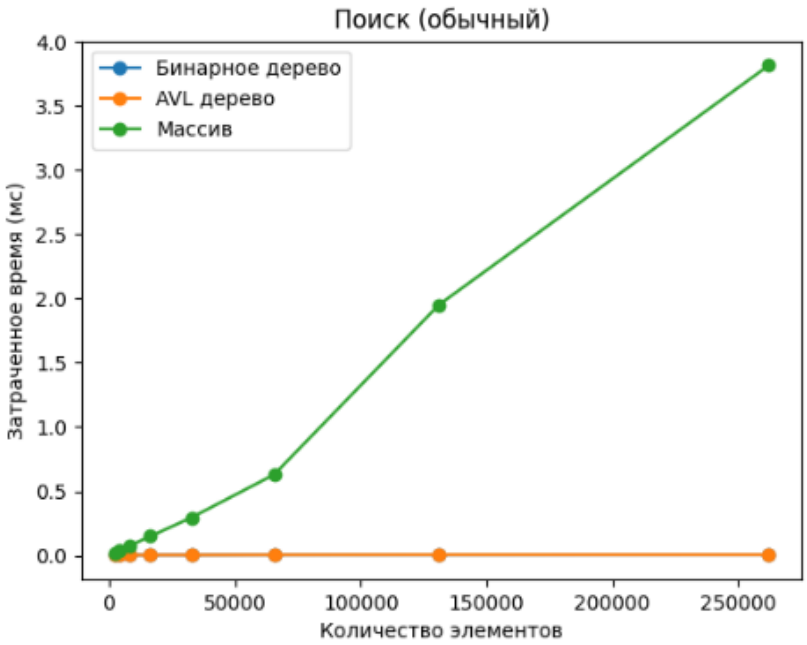
```
        # Правому потомку родительской вершины присваивается значение левого
потомка новой вершины
        node.right = new_root.left
        # Левому потомку новой вершины присваивается значение родительской
вершиной
        new_root.left = node

        # Пересчитываем высоты узлов
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        new_root.height = 1 + max(self._get_height(new_root.left),
self._get_height(new_root.right))

        return new_root

    # Функция для вставки вершины
    def _insert_node(self, node, value):
        # Если дерево пустое, то добавляем вершину в качестве корня
        if node is None:
            return AVLNode(value)
        # Если родительская вершина больше добавляемого значения
        elif value < node.value:
            node.left = self._insert_node(node.left, value)
        else:
            node.right = self._insert_node(node.right, value)

        # Обновляем высоту текущей вершины
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        # Получаем баланс-фактор текущей вершины
        balance_factor = self._get_balance_factor(node)

        # Если вершина несбалансирована, то выполняем соответствующие операции
для балансировки дерева
        if balance_factor > 1:
            # Если баланс левого потомка неотрицательный
            if self._get_balance_factor(node.left) >= 0:
                # Правый поворот родительской вершины
                return self._right_rotate(node)
            else:
                # Левый поворот левого потомка
                node.left = self._left_rotate(node.left)
                # Правый поворот родительской вершины
                return self._right_rotate(node)
        if balance_factor < -1:
            # Если баланс правого узла неположительный
            if self._get_balance_factor(node.right) <= 0:
                # Левый поворот родительской вершины
                return self._left_rotate(node)
            else:
                # Правый поворот правого потомка
                node.right = self._right_rotate(node.right)
                # Левый поворот родительской вершины
                return self._left_rotate(node)

        return node

    def insert(self, value):
        self.root = self._insert_node(self.root, value)

    # Функция для удаления вершины
    def _delete_node(self, node, value):
        if node is None:
            return node

        # Если значение вершины больше искомого значения
```

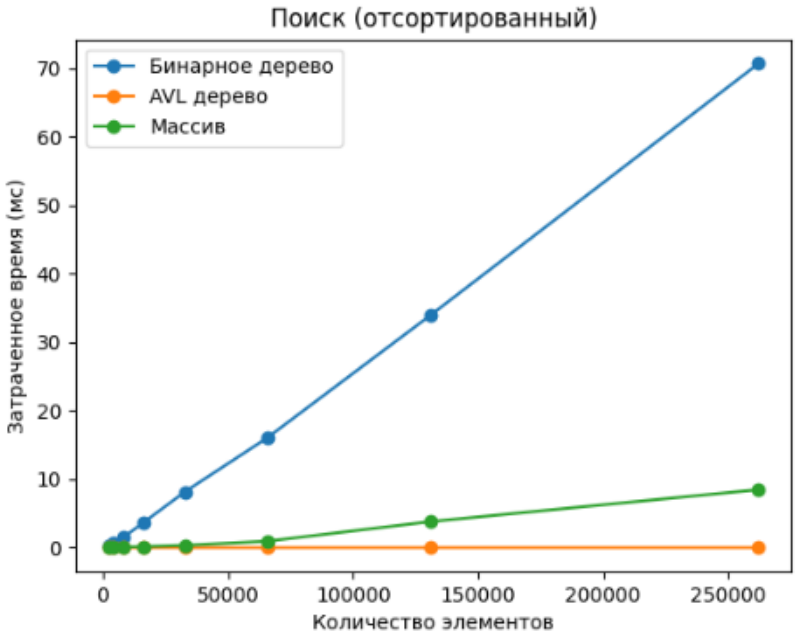
```
        elif value < node.value:
            # Вызываем функцию для левого потомка
            node.left = self._delete_node(node.left, value)
        # Если значение вершины меньше искомого значения
        elif value > node.value:
            # Вызываем функцию для правого потомка
            node.right = self._delete_node(node.right, value)
        # Если значение вершины равно искомому значению
        else:
            # Если вершина с одним или без потомков
            if node.left is None and node.right is None:
                node = None
                return node
            elif node.left is None:
                node = node.right
                return node
            elif node.right is None:
                node = node.left
                return node

            # Узел с двумя потомками
            temp = self._get_min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete_node(node.right, temp.value)

        # Если дерево имело только одну вершину, то возвращаем его
        if node is None:
            return node

        # Обновляем высоту текущего узла
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        # Получаем баланс-фактор текущего узла
        balance factor = self. get balance factor(node)

        # Если вершина несбалансирована, то выполняем соответствующие операции
для балансировки дерева
        if balance_factor > 1:
            # Если баланс левого потомка неотрицательный
            if self._get_balance_factor(node.left) >= 0:
                # Правый поворот родительской вершины
                return self._right_rotate(node)
            else:
                # Левый поворот левого потомка
                node.left = self._left_rotate(node.left)
                # Правый поворот родительской вершины
                return self._right_rotate(node)
        if balance_factor < -1:
            # Если баланс правого узла неположительный
            if self._get_balance_factor(node.right) <= 0:
                # Левый поворот родительской вершины
                return self._left_rotate(node)
            else:
                # Правый поворот правого потомка
                node.right = self._right_rotate(node.right)
                # Левый поворот родительской вершины
                return self._left_rotate(node)

        return node

    def delete(self, value):
        self.root = self._delete_node(self.root, value)

    # Функция для поиска ершины с заданным значением
    def _search_node(self, node, value):
        # Если дерево пустое или значение найдено
```

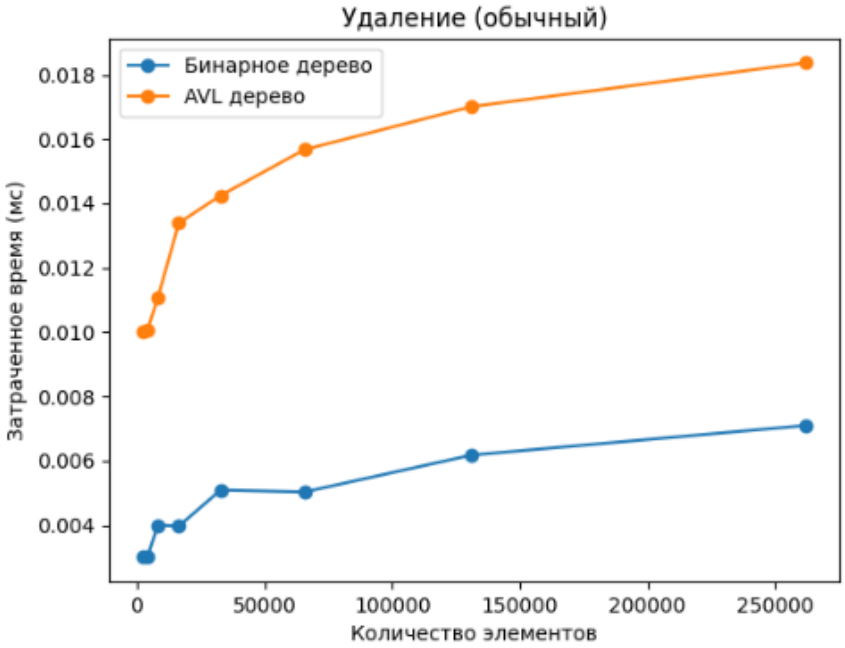
```
        if node is None or node.value == value:
            return node

        # Если значение меньше значения вершины, рекурсивно ищем в левом потомке
        if value < node.value:
            return self._search_node(node.left, value)
        else:
            # Если значение больше значения вершины, рекурсивно ищем в правом
потомке
            return self._search_node(node.right, value)

    def search(self, value):
        return self._search_node(self.root, value)
```

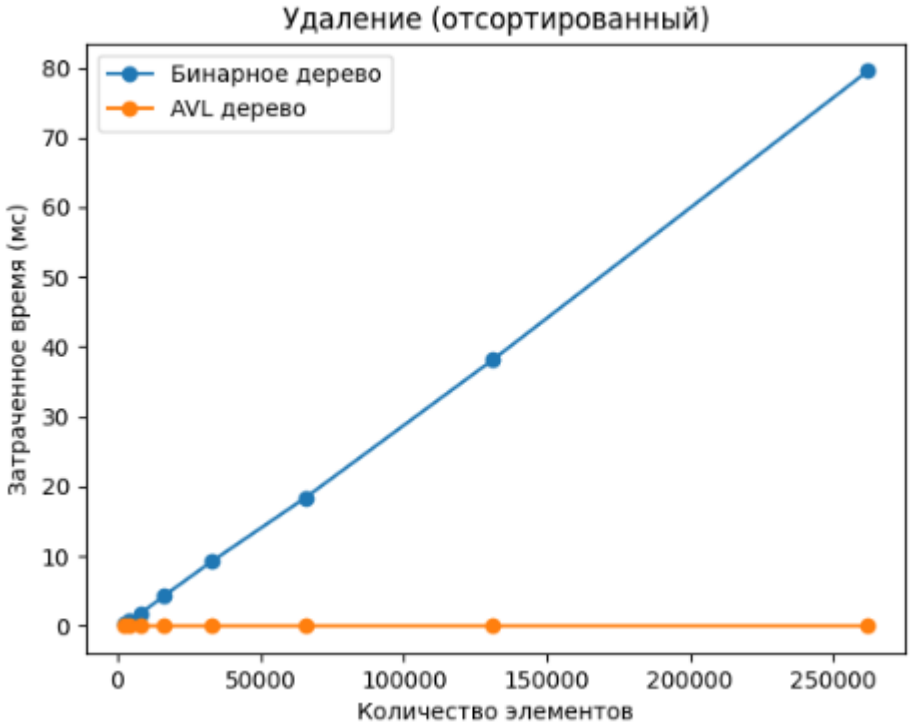





Поиск (обычный)
Бинарное дерево
AVL дерево
Массив
Затраченное время (мс)
Количество элементов
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
50000
100000
150000
200000
250000


Поиск (отсортированный)
Бинарное дерево
AVL дерево
Массив
Затраченное время (мс)
Количество элементов
0
10
20
30
40
50
60
70
0
50000
100000
150000
200000
250000


Удаление (обычный)
Бинарное дерево
AVL дерево
Затраченное время (мс)
Количество элементов
0.004
0.006
0.008
0.010
0.012
0.014
0.016
0.018
0
50000
100000
150000
200000
250000



Заключение.

Из результатов можно сделать вывод, что при работе со случайно сгенерированным массивом целесообразно использовать обычное бинарное дерево без балансировки для операций вставки, в то время как при работе с отсортированным массивом следует использовать AVL.

Обычное бинарное дереве целесообразно использовать при вставке основанной на случайном сгенерированном массиве в случайно сгенерированный массив, в то время как на отсортированном массиве следует использовать AVL, из-за необходимости балансировки на каждом шаге вставки.

Поиск в классическом бинарном дереве и AVL примерно одинаковый на случайно сгенерированном массиве, но на отсортированном AVL лучше.

Удаление на случайно сгенерированном массиве не имеет сильных различий между обычным бинарным деревом и AVL, но на отсортированном массиве AVL становится предпочтительнее.