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-33..... (Вагенлейтнер Никита Сергеевич)

Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/VagenlejtnerNS_33_alg.git)

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

Дата сдачи: ..... (26.02.2025)

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 варианта двоичных деревьев. При этому замеряется время затраченное на всю операцию вставки всего массива.
- После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
- Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
- После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время затраченное на все операции, после чего вычислить время на 1 операцию.
- После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

Описание метода/модели.

Деревья

Бинарное дерево поиска (Binary Search Tree, BST) — это упорядоченное дерево, где:

Для каждой вершины:

- Значения в левом поддереве меньше ключа вершины.
- Значения в правом поддереве больше ключа вершины.

Операции (поиск, вставка, удаление) выполняются за O(h), где h — высота дерева.

- В сбалансированном BST $h \approx \log N$, а в дегенеративном (линейном) — $h \approx N$.

Проблема: Обычное BST может деградировать в список, если вставлять элементы в порядке возрастания/убывания.

AVL-дерево (Adelson-Velsky and Landis Tree) — это самобалансирующееся BST, где:

Для каждой вершины:

- Разница высот левого и правого поддерева (баланс-фактор) не превышает 1.
- При вставке и удалении выполняется автоматическая балансировка (повороты).
- Малый поворот (одно вращение)
- Большой поворот (двойное вращение)
- Операции выполняются за O(log N), так как дерево всегда сбалансировано.

Преимущество AVL-дерева:

- Гарантирует O(log N) для поиска, вставки и удаления.
- Хорош для задач, где частые операции поиска (например, базы данных).

Недостаток:

- Чуть сложнее в реализации из-за балансировки.
- Перестройки требуют дополнительных вычислений.

Выполнение задачи.

Для реализации и выполнения данного алгоритма был задействован язык Python. Были реализованы следующие программные блоки: алгоритмический блок, тестовый блок, аналитический блок полученных результатов работы алгоритма.

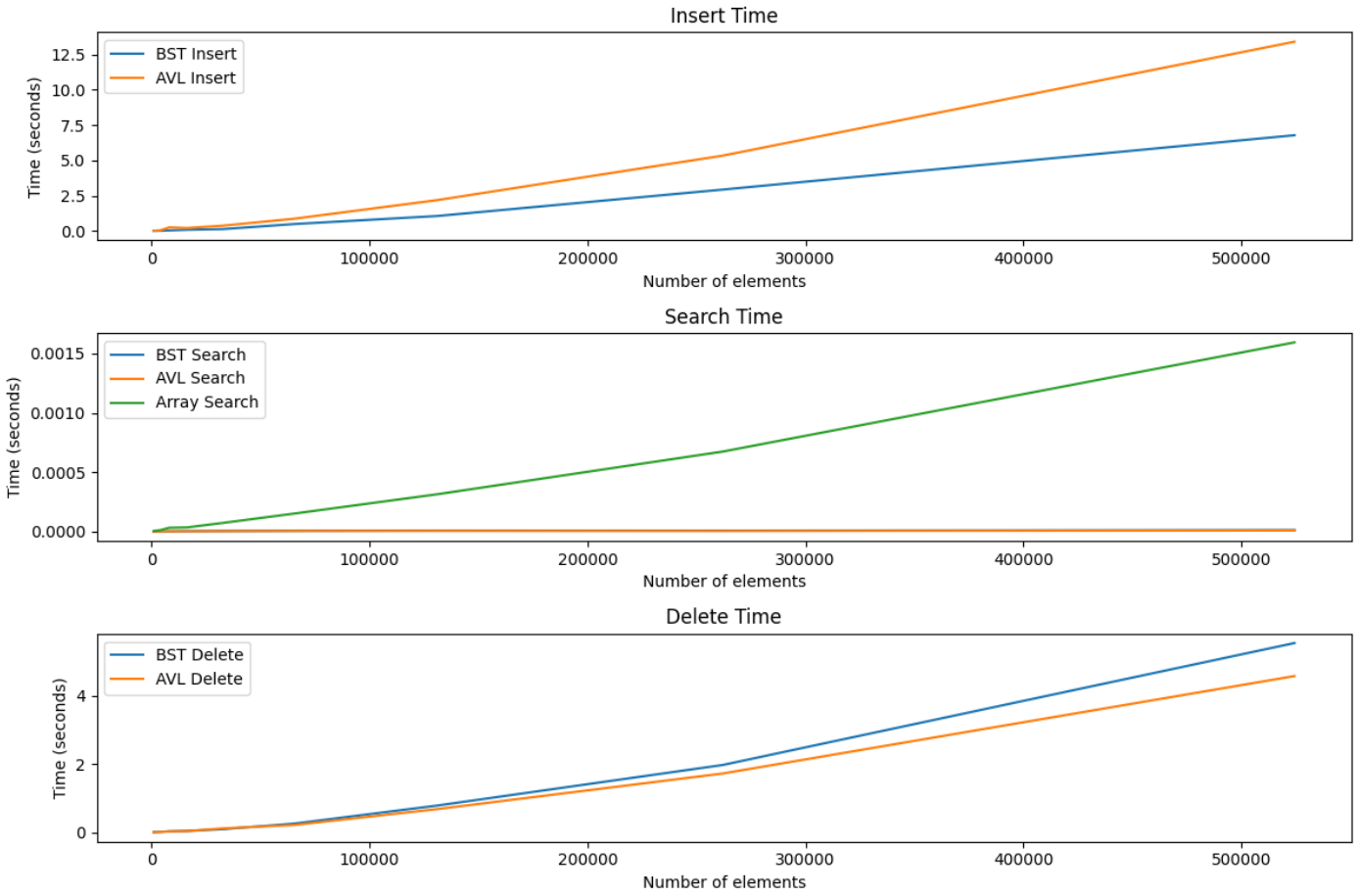
Блок кода отвечающий за реализацию деревьев:

```
    class Node:
        def __init__(self, key):
            self.key = key
            self.left = None
            self.right = None
            self.height = 1
    # Бинарное дерево поиска (BST)
    class BST:
        def __init__(self):
            self.root = None
        def insert(self, key):
            self.root = self._insert(self.root, key)
        def _insert(self, node, key):
            # Базовый случай: если узел пустой, создаём новый узел
            if node is None:
                return Node(key)
            # Проверка на дубликат: если ключ уже есть, ничего не делаем
            if key == node.key:
                return node  # Добавляем обработку дубликатов
            # Вставка элемента: идём влево или вправо в зависимости от значения
            if key < node.key:
                node.left = self._insert(node.left, key)
            else:
                node.right = self._insert(node.right, key)
            return node
        def search(self, key):
            return self._search(self.root, key)
        def _search(self, node, key):
            if not node or node.key == key:
                return node
            if key < node.key:
                return self._search(node.left, key)
            return self._search(node.right, key)
        def delete(self, key):
            self.root = self._delete(self.root, key)
        def _delete(self, node, key):
            if not node:
                return node
            if key < node.key:
```

Блоки кода отвечающие за тестирование:

```
    def run_tests():
        times_insert_bst = []
        times_insert_avl = []
        times_search_bst = []
        times_search_avl = []
        times_search_array = []
        times_delete_bst = []
        times_delete_avl = []
        for i in range(10):
            size = 2 ** (10 + i)
            random_array = [random.randint(0, size) for _ in range(size)]
            # sorted_array = list(range(size)) # No longer need sorted array
            # for array, name in [(random_array, "Random"), (sorted_array, "Sorted")]:
# change here
            for array, name in [(random_array, "Random")]:
                bst = BST()
                avl = AVLTree()
                # Вставка элементов в деревья
                start_time = time.time()
                for num in array:
                    bst.insert(num)
                times_insert_bst.append(time.time() - start_time)
                start_time = time.time()
                for num in array:
                    avl.insert(num)
                times_insert_avl.append(time.time() - start_time)
                # Поиск 1000 случайных элементов
                search_times_bst = []
                search_times_avl = []
                for _ in range(1000):
                    key = random.choice(array)
                    start_time = time.time()
                    bst.search(key)
                    search_times_bst.append(time.time() - start_time)
                    start_time = time.time()
                    avl.search(key)
                    search_times_avl.append(time.time() - start_time)
                times_search_bst.append(sum(search_times_bst) / 1000)
                times_search_avl.append(sum(search_times_avl) / 1000)
                # Поиск в обычном массиве
```

Результаты работы программы выводится в виде графика зависимости и приведены ниже.



Заключение.

Бинарное дерево поиска (BST) эффективно для хранения и поиска данных, но может деградировать в линейную структуру при неудачном порядке вставки, приводя к худшему времени работы O(N).

- AVL-дерево решает эту проблему автоматической балансировкой, гарантируя высоту O(log N) и обеспечивая стабильную производительность для операций поиска, вставки и удаления.
- BST лучше, если данные вставляются случайно и дерево остаётся сбалансированным естественным образом.

AVL предпочтительно для частого поиска (например, базы данных) из-за меньшей высоты, но требует дополнительных вычислений при балансировке. В общем случае, если важна скорость поиска и обновлений — AVL-дерево более надёжное решение.