1. Specification

1.1. Binary search tree (BST)

Supports the **insert** operation.

A **size** and a **lookup** operation were also defined. The purpose of these are discussed in 1.2 and 3.2 respectively.

1.2. Weight-balanced binary search tree

Balancing determined by a parameter $\frac{1}{2} < c < 1$.

All operations of the regular BST are defined analogously, with the added step that for each node N in the search path during insertion, if either of these conditions apply:

$$\begin{aligned} \text{size}(\text{left}(N)) &> c \cdot \text{size}(N) \\ \text{size}(\text{right}(N)) &> c \cdot \text{size}(N), \end{aligned} \tag{*}$$

the tree rooted at n is rebalanced.

The consequence of this definition is that for values of c close to $\frac{1}{2}$, the tree is very strictly balanced, potentially for every insertion. For c close to 1, balancing is very lax and potentially never occurs, similar to the regular BST.

This definition also necessitates a way to determine a tree's size. The size of a tree can be easily determined given its root, though this is an $O(n)$ operation, where n is the number of nodes in the tree. Instead, the decision was made to internally keep track of the tree's size and update it on insertion.

2. Overview

2.1. Insertion

Given a key k to be inserted, do the following:

- Let N be the current node, initially the root, and k_N its key.
  - If N doesn't already exist, create it with the key k and finish.
- Compare k to k_n:
  - If $k < k_N$, recursively insert k into $\text{left}(N)$.
  - If $k > k_N$, recursively insert k into $\text{right}(N)$.
  - Otherwise, if $k = k_N$, do nothing.
- For the weight-balanced variant only, if either of (*) hold, rebalance the tree rooted at N.
- If the value was inserted, increase the tree's size.

2.2. Lookup

Given a key k to be looked up, do the following:

- Let N be the current node, initially the root, and k_N its key.
  - If N doesn't exist, the key did not exist in the tree.
- Compare k to k_n:
  - If $k < k_N$, recursively look for k in $\text{left}(N)$.

- If $k > k_N$, recursively look for k in $\text{right}(N)$.
- Otherwise, if $k = k_N$, the key exists in the tree.

2.3. Balance

- Let N be the current node. This node exists.
- Let I be sequence representing the inorder traversal of the tree.
- Construct a tree T from I.
- Replace the tree rooted at N with T.

Note: this is a fairly expensive way to balance larger trees, as a lot of memory is used for the traversal and intermediate tree. There are more efficient ways to balance binary trees, for example through tree rotations.

3. Experimentation

3.1. Conditions

The decision was made to perform a total of 1500 insertions and lookups for each tree. Larger sizes risk causing memory issues in the worst case[1].

Four trees were used: a regular BST, and three weight-balanced with $c = 0.9$, $c = 0.75$ and $c = 0.6$. For each tree, two sequences of values were inserted:

- An entirely random sequence, roughly uniformly distributed over 750 unique values. This was to ensure a miss of new and duplicate keys.
- A mostly-increasing random sequence, with a roughly $\frac{1}{5}$ chance to produce a value smaller than the previous, and the same chance to produce a value equal to the previous.

3.2. Hypotheses

Balancing a tree is a direct read-to-write-time payoff; a lookup in a balanced tree is expected to be *at least* as fast as the same lookup in any other tree containing the same keys. However, insertion also becomes more expensive when balancing occurs. As such, insertion is expected to become gradually more expensive for more strictly balanced trees, while lookup becomes cheaper.

The nature of the values affects both insertion and lookup time; a sequence of strictly increasing values creates a single long branch in an unbalanced tree, meaning $O(n)$ insertions and lookups as they potentially have to traverse the whole branch (the entire tree). For a perfectly balanced tree, however, lookup cost is mitigated, instead showing the typical $O(\log(n))$, but insertions become very expensive as balancings occur frequently.

3.3. Benchmarking results

Below are the results averaged over 1000 runs of tests with the same starting conditions, save for random state. 1500 insertions were performed here. The averaging should theoretically eliminate most noise and edge cases such as when the choice of root is very poor. Time is measured in milliseconds.

[1] Max recursion depth in Python is around 1000 on the platform used.

	Random values		Increasing values	
Balance	Insertion	Lookup	Insertion	Lookup
(Unbalanced)	6.2727	5.1928	237.22	173.61
$c = 0.98$	12.473	4.8901	36.022	7.4240
$c = 0.9$	13.519	4.5024	29.790	4.6474
$c = 0.75$	13.813	4.2845	29.293	4.4476
$c = 0.6$	16.510	4.1675	43.997	4.3608
$c = 0.52$	73.312	3.9029	188.23	3.7772

As expected, for random values, less balance means quicker insertion while more balance means quicker lookup. In practice, this can be applied by choosing c such that read-write cost is balanced to fit the application's needs.

For mostly increasing values, lookup is naturally still quicker for more balanced trees, but the case for insertion seems more complex. For no balance, insertion becomes very expensive as the tree grows larger and the main branch becomes longer, but for heavier balance the balancing cost outweighs the gain. More experimentation would be needed to determine the optimal c for the given data.

A larger tree with 1000000 insertions was also benchmarked, and the results are shown below.

Balance	Insertion	Lookup
$c = 0.9$	8113.3807	2834.9578
$c = 0.75$	7718.1964	2788.5721
$c = 0.6$	7390.9709	2733.1314