



Algorithms and Data Structures – Conditional Course

Winter Term 2026/27

Exercise Sheet 7

Due: Wednesday, 09 December 2026, 12:00

Exercise 1: The AVL Property and Rotations (3 Points)

An AVL tree is a binary search tree in which, at every node, the heights of the left and right subtrees differ by at most one.

- a) Insert the keys 10, 20, 30, 40, 50, 25, in this order, into an initially empty AVL tree. Draw the tree after each insertion that involves a rotation. (2 Points)
- b) For each rotation in part (a), state whether it is a left, right, left-right, or right-left rotation. (1 Point)

Exercise 2: Programming: An AVL Tree in Python (10 Points)

Implement four central operations on AVL trees:

- `_rebalance(node)`,
- `_rotate_right(node)`,
- `insert(key, value=None)`,
- `successor(node)`.

Helpers for heights, balance factors, and left rotations are already supplied. Complete only the four functions marked `TODO` in `avl_tree.py`.

Requirements

- i) `_rebalance` restores the AVL property if the balance factor was invalid.
- ii) After each operation, structural information is correct, including the nodes' `height` values.
- iii) After a right rotation, the tree is still a binary search tree.
- iv) Every call to `insert` preserves the binary-search-tree property.
- v) Every call to `insert` preserves the AVL property.
- vi) Parent pointers are set correctly.
- vii) Inserting an existing key updates its value without creating a new node.
- viii) `successor(node)` returns the node with the next larger key, or `None` if no such node exists.
- ix) `insert` runs in $O(\log n)$ time.
- x) `successor` runs in $O(\log n)$ time.
- xi) `_rotate_right` runs in $O(1)$ time.

Optional: Checking Your Implementation

For your own confidence, you may try inserting into an empty tree, insertion sequences that trigger each type of rotation, repeated keys, and successor queries for the smallest and largest keys. These checks are optional and are not graded; no test file needs to be submitted.

Exercise 3: Finding Upcoming Events

(4 Points)

A calendar stores events by their start times. Each event has a unique integer key, such as its start time as a Unix timestamp. Events are stored in an AVL tree.

For a given event v , output the next k later events in chronological order without traversing the entire tree.

- a) Describe an algorithm `next_k_events(v, k)` that uses `successor` to output the next k events after v . *(1 Point)*
- b) Give pseudocode for your algorithm. *(1 Point)*
- c) Analyze its running time in terms of n and k , using the fact that the tree is an AVL tree. *(1 Point)*
- d) Explain why a full inorder traversal would be unnecessarily inefficient when $k \ll n$. *(1 Point)*

Submission

Submit a PDF with your theoretical solutions and your implementation in `avl_tree.py`.