1. Array → BST. Design an algorithm which takes on input a sorted array of numbers, and creates a perfectly balanced BST in linear time. (A BST T is *perfectly balanced* if for all $v \in T$ it holds that $\big||L(v)| - |R(v)|\big| \leq 1$, that is, the size of the left and right subtrees differs by at most 1.)

2. `BST_SPLIT`. Design a `BST_SPLIT` operation which takes on input a BST T and a value s and outputs two BSTs T_1, T_2 such that T_1 contains those values of T which are smaller than s, and T_2 contains those values of T which are $\geq s$.

3. AVL evolution. Draw the evolution of an AVL tree as we insert (in this order) the numbers 10, 20, 15, 25, 30, 16, 18, 19. What happens when we then delete 30?

4. BST improvements. Consider a general BST maintaing (key, value) pairs, sorted by the key. Implement the following operations while maintaining the $\mathcal{O}(\text{dept})$ complexity. You may need to store some additional information in the nodes.

(1) Min, Max of a given interval of keys. (E.g. $\max(r, s)$ should return the largest value of keys between r and s.)
(2) Average of a given interval of keys.
(3) Assume that the values are matrices $A_1, \ldots, A_n$ of size $m \times m$ and for any interval of keys r, s we want to be able to quickly compute what is the matrix product $A_r \cdot A_{r+1} \cdots A_s$. (Caution: this operation is not commutative. Unlike for max above, the order of matrix multiplications matters. Disregard the complexity of a single matrix multiplication when estimating the $O(\text{depth})$ complexity.)
(4) Adding δ to all values in a given interval.

5. AVL improvements. What adjustments are needed to make all of this work for an AVL tree? (So that the complexity of these operations is $\mathcal{O}(\log n)$.)

6. Depth. Choose a representation of an AVL tree (e.g., during the lecture we said that we will maintain a sign $-, 0, +$ at each vertex). You would like to now adjust the `INSERT/DELETE` operations so that you can answer in $\mathcal{O}(1)$ a query for the height of any subtree. Is it possible? You may need to change the representation.

7. Data Structure 1. Construct a (composite) data structure which can handle the following operations in the required time:

- `Init()` – initializes the data structure– $\mathcal{O}(1)$.
- `INSERT(`X`)` – inserts element X, if it is not yet in the structure – $\mathcal{O}(\log n)$.
- `DELETE(`X`)` – deletes X, if it is in the structure – $\mathcal{O}(\log n)$.
- `DELETE_IN_PLACE(`I`)` – deletes element which was the I-th added – $\mathcal{O}(\log n)$.
- `GET_PLACE(`X`)` – returns a number I such that X was the I-th added element – $\mathcal{O}(\log n)$.

8. Data Structure 2. An electrician wants to maintain a list of clients indexed by their IDs together with a record of whether they are male or female *(bonus task: handle more genders)*. Design a data structure which handles the following operations in the time $\mathcal{O}(\log n)$:

- `INSERT(`K`, `C`)` – inserts a new client C with `ID`$=K$, designates them female.
- `UPDATE(`K`)` – designates client with `ID`$=K$ as male.
- `FINDDIFF(`K`)` – finds the difference between the numbers of male and female clients among those with `ID` $\leq K$.

9. Window. Numbers are arriving on input. Whenever a new number arrives, report the median and average of the last k numbers. Try to attain $\mathcal{O}(\log k)$ complexity per report.

10. Subsequence. We are given a sequence of n numbers and we want to find the longest increasing subsequence (doesn't have to be contiguous) in time $\mathcal{O}(n \log n)$. (We have already seen this task in our first tutorial, and we could only solve it in time $\mathcal{O}(n^2)$ by finding the longest path in a DAG.)