Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

23 October 2023

Algorithms & Data Structures — Exercise sheet 5 — HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 30 October 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Sorting.

Exercise 5.1 *Sorting algorithms.*

Below you see four sequences of snapshots, each obtained in consecutive steps of the execution of one of the following algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

3	6	5	1	2	4	8	7
3	6	5	1	2	4	8	7
3	5	6	1	2	4	8	7

3	6	5	1	2	4	8	7
3	5	1	2	4	6	7	8
3	1	2	4	5	6	7	8

3	6	5	1	2	4	8	7
3	6	1	5	2	4	7	8
1	3	5	6	2	4	7	8

3	6	5	1	2	4	8	7
3	6	5	1	2	4	7	8
3	6	5	1	2	4	7	8

Exercise 5.2 *Guessing an interval* **(1 point)**.

Alice and Bob play the following game:

- Alice selects two integers $1 \leq a < b \leq 200$, which she keeps secret.
- Then, Alice and Bob repeat the following:
  - Bob chooses two integers $0 \leq a' < b' \leq 201$.
  - If $a = a'$ and $b = b'$, Bob wins.
  - If $a' < a$ and $b < b'$, Alice tells Bob 'my numbers are strictly between your numbers!'. A previous version had the mistake that Alice gave information to Bob when $a < a'$ and $b' < b$, which has now been corrected to $a' < a$ and $b < b'$.
  - Otherwise, Alice does not give any clue to Bob.

(a) Bob claims that he has a strategy to win this game in 12 attempts at most. Prove that such a strategy cannot exist.

Hint: *Represent Bob's strategy as a decision tree. Each edge of the decision tree corresponds to one of Alice's answers, while each leaf corresponds to a win for Bob.*

Hint: *After defining the decision tree, you can consider the sequence $k_0 = 1$ and $k_n = 2k_{n-1} + 2$ for $n \geq 1$, and prove that $k_n = 3 \cdot 2^n - 2$ for any $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The number of vertices in the decision tree should be related to k_n.*

(b)* Can Bob have a strategy to win the game in 13 or 14 attempts?

Hint: *Follow the same strategy as for (a). After defining the decision tree, try to analyse the number of leaves in the decision tree corresponding to Bob's strategy. The sequence $\ell_1 = 1$ and $\ell_n = 2\ell_{n-1} + 1$* for $n > 1$, for which you can prove $\ell_n = 2^n - 1$ for any $n \in \mathbb{N}$, might be helpful.

Exercise 5.3 *Building a Heap* **(1 point)**.

Recall that a binary tree is called *complete* if all of its layers are fully filled, except possibly the last layer, which should be filled from left to right. A *(max-)heap* is a complete binary tree with the extra property that for any node C with parent P,

$$\text{key}(P) \geq \text{key}(C). \qquad \text{(heap-condition)}$$

In this exercise, we formally prove the correctness of the following algorithm from the lecture, which adds a new node with key k to an existing, non-empty heap H. We will show that it performs at most $O(\log n)$ comparisons between keys, where n is the number of nodes in the heap H, and that it maintains the heap structure.

Algorithm 1 Heap insertion

```
function INSERT(H, k)
    Add a new node N with key k to the bottom layer of H, in the left-most free position. If the
  bottom layer is full, instead create a new layer and add the node in the left-most position.
    P ← the parent of N
    while key(P) < key(N) do                              ▷ N violates the heap-condition
        swap the keys of node N and P.
        N ← P
        if N is the root node then
            stop
        else
            P ← the parent of N
```

Let H be a heap consisting of $n \geq 1$ nodes, and let $k \in \mathbb{N}$. Let H' be the data structure that results from executing $\text{Insert}(H, k)$.

(a) Prove that at most $O(\log n)$ comparisons between keys are performed in the execution of $\text{Insert}(H, k)$.

Hint: *After each iteration of the while-loop, what can you say about the* depth *of the node N?*

(b) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth less than or equal to $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. (A node N satisfies the heap-condition if it is the root node, or otherwise if $\text{key}(N) \leq \text{key}(\text{parent(N)})$.)

Hint: *Use the fact that H was a heap before we inserted the new node. Consider separately the two different reasons for the algorithm to terminate.*

(c) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth strictly greater than $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. Using (b), conclude that H' is a heap.

Hint: *Let T be the depth of H'. Use induction to show that after t iterations of the while-loop, the heap-condition is satisfied by all nodes with depth strictly greater than $T - t$.*

Hint: *After swapping the keys of nodes N and P in an iteration of the while-loop, which nodes might potentially no longer satisfy the heap-condition?*

Data structures.

Exercise 5.4 *Implementing abstract data types.*

In the lecture, you saw how we can implement the abstract data type list with operations insert, get, delete and insertAfter. In this exercise, the goal is to see how we can implement two other abstract data types, namely the stack (german “Stapel”) and the queue (german “Schlange” or “Warteschlange”). The abstract data type stack is, as the name suggests, a stack of elements. For a stack S, we want to implement the two following operations; see also Figure 1.

- $\text{push}(x, S)$: Add x on top of the stack S.
- $\text{pop}(S)$: Remove (and return) the top element of the stack S.

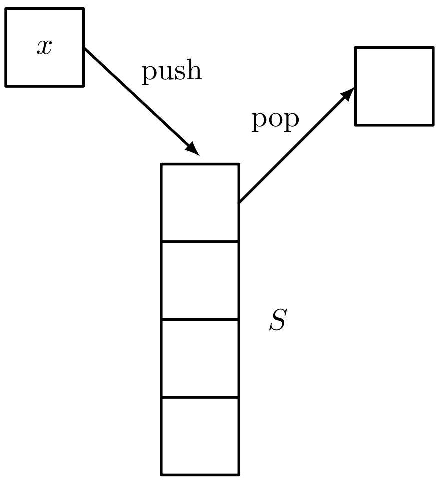


Figure 1: Abstract data type stack

The abstract data type queue is a queue of elements. For a queue Q, we want to implement the following two operations; see also Figure 2.

- $\text{enqueue}(x, Q)$: Add x to the end of Q.
- $\text{dequeue}(Q)$: Remove (and return) the first element of Q.

(a) Which data structure from the lecture can be used to implement the abstract data type stack efficiently? Describe for the operations push and pop how they would be implemented with this data structure and what the run time would be.

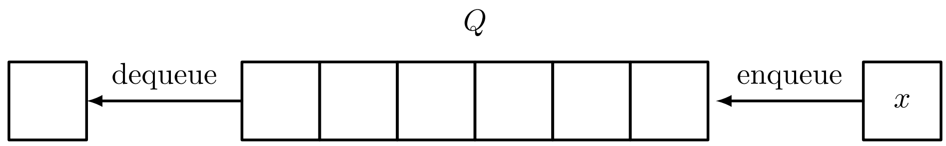


Figure 2: Abstract data type queue

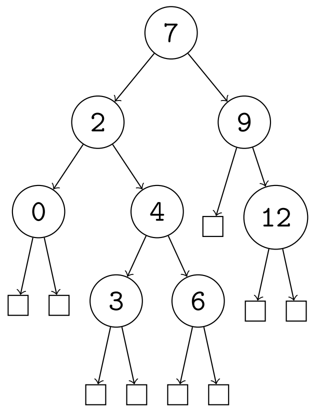
(b) Which data structure from the lecture can be used to implement the abstract data type queue efficiently? Describe for the operations enqueue and dequeue how they would be implemented with this data structure and what the run time would be.

Remark: The following exercise 5.5 is related to the content of the lecture on Tuesday, October 24.

Exercise 5.5 *AVL trees* **(1 point)**.

(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.

(b) Consider the following AVL tree.



Draw the tree obtained by deleting 6, 12, 7 and 4 in this order from this tree. Give also all the intermediate states after every deletion and before and after each rotation that is performed during the process.

Exercise 5.2 *Guessing an interval* **(1 point).**

Alice and Bob play the following game:

- Alice selects two integers $1 \leq a < b \leq 200$, which she keeps secret.
- Then, Alice and Bob repeat the following:
  - Bob chooses two integers $0 \leq a' < b' \leq 201$.
  - If $a = a'$ and $b = b'$, Bob wins.
  - If $a' < a$ and $b < b'$, Alice tells Bob 'my numbers are strictly between your numbers!'. A previous version had the mistake that Alice gave information to Bob when $a < a'$ and $b' < b$, which has now been corrected to $a' < a$ and $b < b'$.
  - Otherwise, Alice does not give any clue to Bob.

(a) Bob claims that he has a strategy to win this game in 12 attempts at most. Prove that such a strategy cannot exist.

Hint: *Represent Bob's strategy as a decision tree. Each edge of the decision tree corresponds to one of Alice's answers, while each leaf corresponds to a win for Bob.*

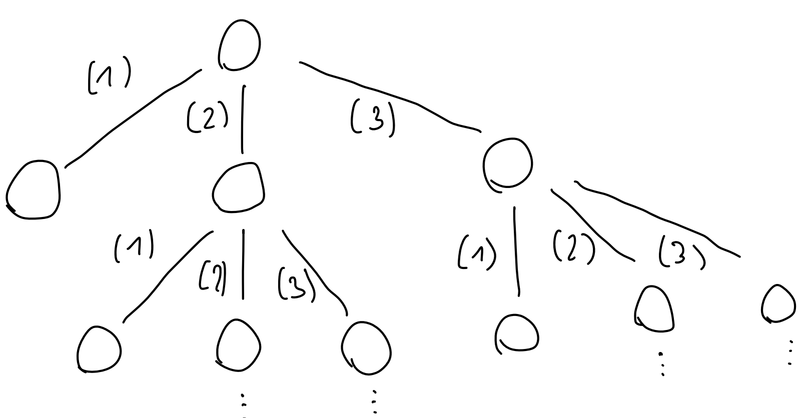
Hint: *After defining the decision tree, you can consider the sequence $k_0 = 1$ and $k_n = 2k_{n-1} + 2$ for $n \geq 1$, and prove that $k_n = 3 \cdot 2^n - 2$ for any $n \in \mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The number of vertices in the decision tree should be related to k_n.*

What was wrong with the first version of sheet 5? Why did have to be corrected?

(1) $a' = a$ and $b' = b$

(2) $a' < a$ and $b < b'$

(3) all other cases



We observe the (three) following (facts):

Each vertex has at most 3 children. One of the children (case (1)) doesn't have children, the other two can have the same structure as their parent. (1)

Assuming Bob plays optimally, not every vertex (that is not case (1)) needs to have 3 children. For example, if Bob has figured out Alice's numbers after a sequence of steps, there is only one option left (case (1)) (2)

The depth of the tree is the number of guesses Bob has to make in the worst case. (3)

Let $k_n :=$ maximum # of vertices, then

$$k_n = \begin{cases} 1 & n = 0 \\ 2k_{n-1} + 2 & n \geq 1 \end{cases}$$

Note that at root level $(n = 0)$ we have just a single vertex (thus $k_0 = 1$) and from observation (2) we know that we have 1 min node and possibly (maximally) two nodes with the same structure as their parent (thus the recursion). Additionally we have 1 root for a total of $2k_{n-1} + 1 + 1$ vertices for k_n if $n \geq 1$.

Hint $\Rightarrow$ we show $k_n = 3 \cdot 2^n - 2$.

- **Base Case.**
  For $n = 0$, we have $k_0 = 1 = 3 \cdot 2^0 - 2$, so the base case holds.
- **Induction Hypothesis.**
  Assume that the statement holds for $j \in \mathbb{N}$, i.e., $k_j = 3 \cdot 2^j - 2$.
- **Inductive Step.**
  We compute

$$k_{j+1} = 2k_j + 2 = 2 \cdot (3 \cdot 2^j - 2) + 2 = 3 \cdot 2^{j+1} - 4 + 2 = 3 \cdot 2^{j+1} - 2.$$

Thus, the statement also holds for $j + 1$. By the principle of mathematical induction, we have $k_n = 3 \cdot 2^n - 2$ for any $n \in \mathbb{N}_0$.

Now we want to check, how many possibilities there are and how it would affect any of Bob's winning strategies. A guaranteed winning strategy must cover all possibilities.

For example, let's say someone throws a fair die a couple times and each time we add the number of the die to a sum that is zero at the start of the game. If the objective was, that the sum ≥ 6, and I say "it will only take 1 throw to complete the objective" then you can immediately see, that that only happens if the die shows 6. But what about 1+1+1+1+1+1? Or 2+2+2? This is what we mean by considering all cases.

Next, we want to count the number of pairs Alice can choose. Once she has chosen b, she has $b-1$ possibilities for a (the numbers in the set $\{1, 2, \ldots, b-1\}$). Thus, the total number of pairs Alice can choose is

$$\sum_{b=1}^{200}(b-1) = \left(\sum_{b=1}^{200} b\right) - 200 = \frac{200 \cdot 201}{2} - 200 = 19900,$$

where the second equality uses $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for any $n \in \mathbb{N}$, which was proven in exercise 0.1. In order for Bob's strategy to allow him to win for any pair of integers chosen by Alice, the tree representing his strategy must have at least 19900 leaves (one for each choice of Alice). If Bob's statement is true (i.e. he wins after at most 12 turns), this tree has depth at most 12 and therefore at most k_{12} vertices. Since $k_{12} = 12286 < 19900$, the decision tree corresponding to Bob's strategy cannot have 19900 leaves, hence Bob cannot certainly win in at most 12 attemps.

for (b)* refer to official solution.

Exercise 5.3 *Building a Heap* **(1 point).**

Recall that a binary tree is called *complete* if all of its layers are fully filled, except possibly the last layer, which should be filled from left to right. A *(max-)heap* is a complete binary tree with the extra property that for any node C with parent P,

$$\text{key}(P) \geq \text{key}(C). \qquad \text{(heap-condition)}$$

In this exercise, we formally prove the correctness of the following algorithm from the lecture, which adds a new node with key k to an existing, non-empty heap H. We will show that it performs at most $O(\log n)$ comparisons between keys, where n is the number of nodes in the heap H, and that it maintains the heap structure.

Algorithm 1 Heap insertion

```
function INSERT(H, k)
    Add a new node N with key k to the bottom layer of H, in the left-most free position. If the
    bottom layer is full, instead create a new layer and add the node in the left-most position.
    P ← the parent of N
    while key(P) < key(N) do                              ▷ N violates the heap-condition
        swap the keys of node N and P.
        N ← P
        if N is the root node then
            stop
        else
            P ← the parent of N
```

Let H be a heap consisting of $n \geq 1$ nodes, and let $k \in \mathbb{N}$. Let H' be the data structure that results from executing Insert(H, k).

If you are comfortable with heaps, chances are, that you quickly "see what's going on" but don't really know how to write it. This is quite common with AnD and nothing to be worried about! It takes time and practice.

(a) Prove that at most $O(\log n)$ comparisons between keys are performed in the execution of $\text{Insert}(H, k)$.

Hint: *After each iteration of the while-loop, what can you say about the* depth *of the node N?*

official solutions.

(b) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth less than or equal to $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. (A node N satisfies the heap-condition if it is the root node, or otherwise if $\text{key}(N) \leq \text{key}(\text{parent(N)})$.)

Hint: *Use the fact that H was a heap before we inserted the new node. Consider separately the two different reasons for the algorithm to terminate.*

<u>Case N_{stop} = root:</u>

Since N_{stop} (the root) is at depth zero, there is nothing to check.

<u>Case $N_{stop} \neq$ root:</u>

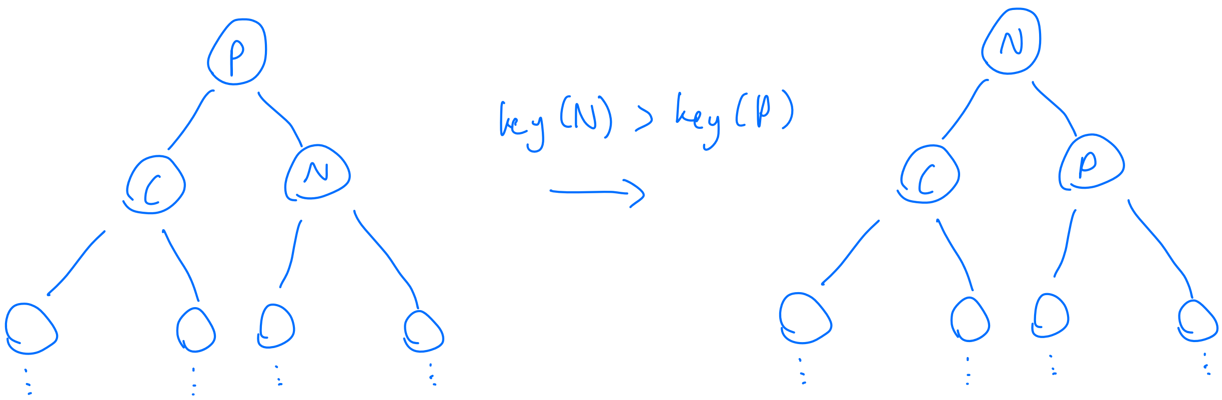
Since $N_{stop} \neq$ root, we must have key$(P) \geq$ key(N_{stop}). Thus the heap condition is fulfilled for N_{stop}. For all nodes with depth less than N_{stop}, there was no change, and since (Hint) H was a heap, they all satisfy the heap condition.

(c) Let N_{stop} be the final node considered by the algorithm. Prove that all nodes in H' with depth strictly greater than $\text{depth}(N_{\text{stop}})$ satisfy the heap-condition. Using (b), conclude that H' is a heap.

Hint: *Let T be the depth of H'. Use induction to show that after t iterations of the while-loop, the heap-condition is satisfied by all nodes with depth strictly greater than $T - t$.*

Hint: *After swapping the keys of nodes N and P in an iteration of the while-loop, which nodes might potentially no longer satisfy the heap-condition?*

The second hint is most important. Intuition:



By transitivity of the $\geq$ relation (i.e. $\text{key}(X) \geq \text{key}(Y)$ and $\text{key}(Y) \geq \text{key}(Z) \Rightarrow \text{key}(X) \geq \text{key}(Z)$) the heap condition of the children of P is fulfilled even after swapping the nodes since $\text{key}(N) > \text{key}(P) \Longrightarrow \text{key}(N) \geq \text{key}(X)$ for all X at depth higher than P.

For the set of nodes at depth $> \ell$ for some $\ell \in \mathbb{N}$ we write $H_{>\ell}$.

<u>IB:</u> $t = 0$

$H_{>T-t} = H_{>T}$, thus there is nothing to show.

<u>IH:</u> after t iterations of the while loop, the heap condition is satisfied by all nodes in $H_{>T-t}$ for some t.

<u>IS:</u> $t \rightsquigarrow t+1$

By IH. we know that all nodes in $H_{>T-t}$ satisfy the heap condition.

We swap keys of N and P in iteration $t+1$.

Since we swapped these keys we must have $key(N) > key(P)$. By transitivity of the $\geq$ relation we now know

$$key(N) \geq key(X) \text{ for all } X \in H'_{>T-t}$$

where H' is the heap after we swap.

We conclude all nodes in $H'_{>T-t}$ fulfill the heap condition.

At depth exactly $T-t$ in H', the only affected nodes are N and a potential second child C of P.

Before the swap we had $key(P) \geq key(C)$ and $key(N) > key(P)$.

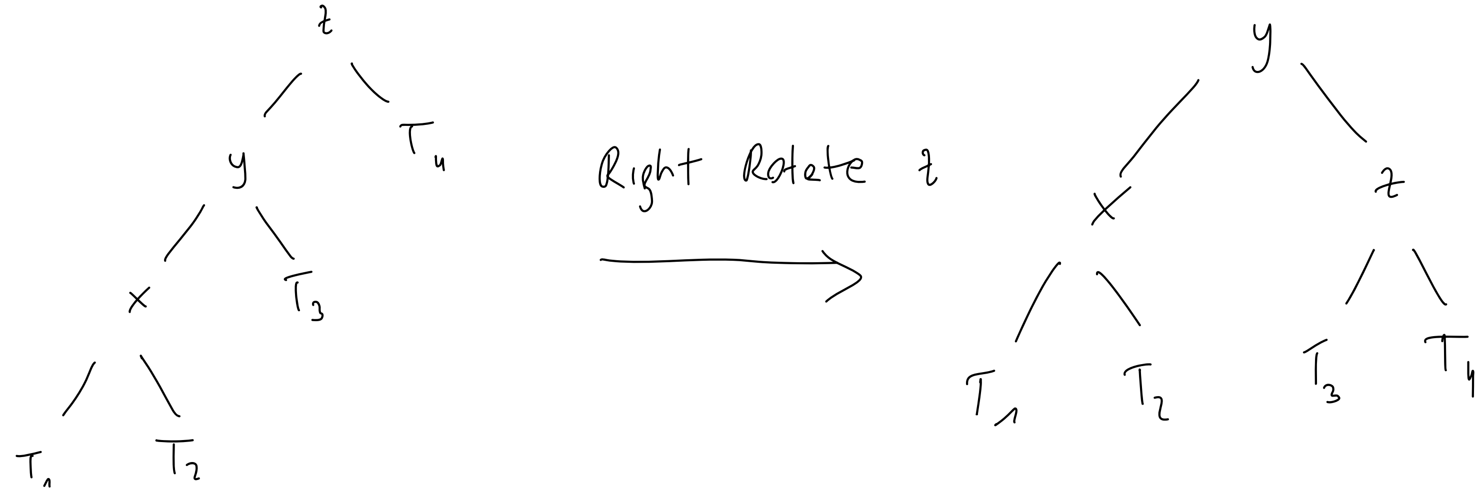
After the swap we have $key(P) \geq key(N)$ and $key(P) \geq key(C)$. Thus N and C satisfy the heap condition.

To conclude, it remains to note that $\text{depth}(N_{\text{stop}}) = T - t_{\text{stop}}$, where t_{stop} is the total number of iterations of the while-loop in the execution of $\text{Insert}(H, k)$.

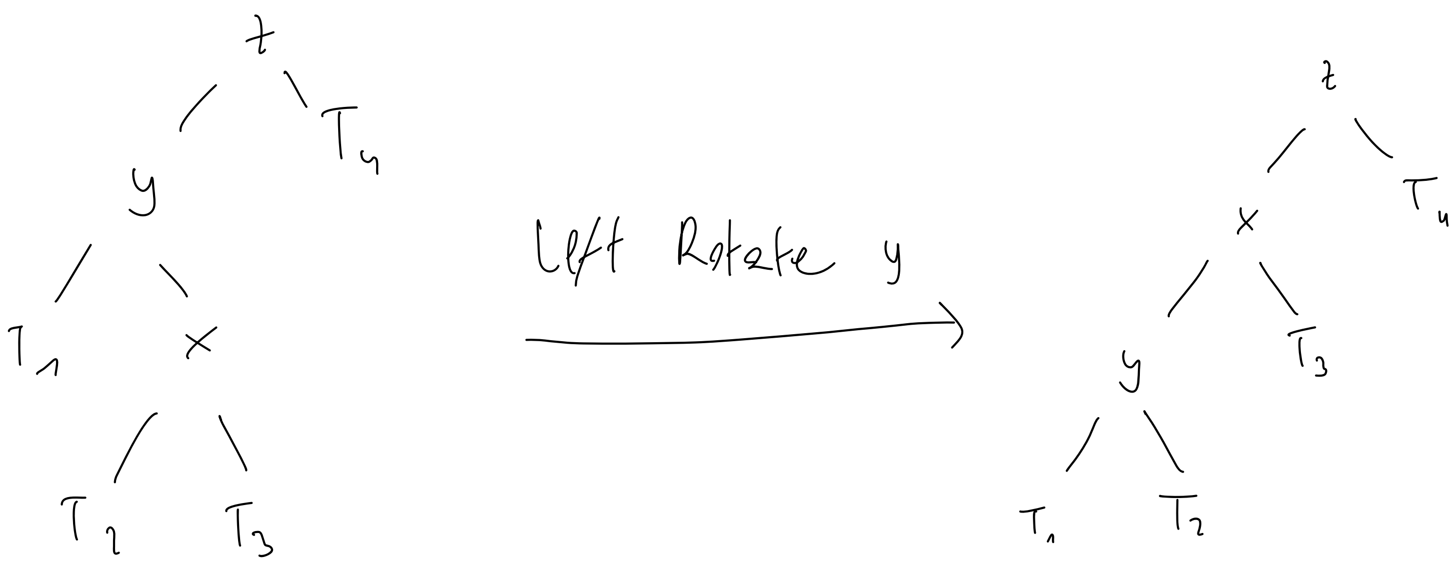
Let T_1, T_2, T_3, T_4 denote subtrees.

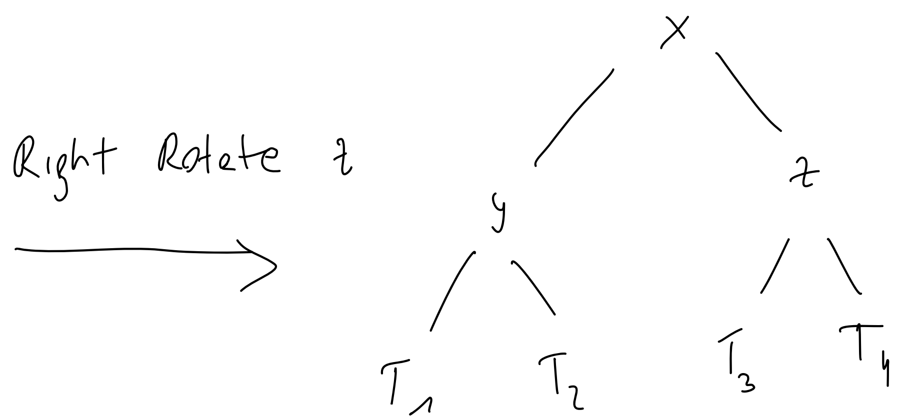
AVL-Tree insertion cases:

Left Left:

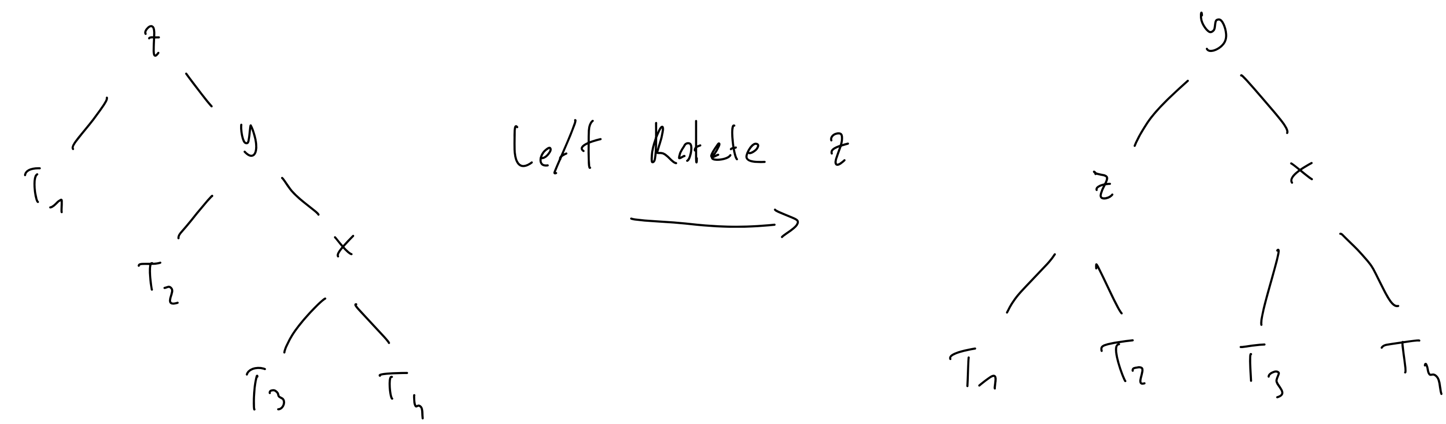


Left Right:

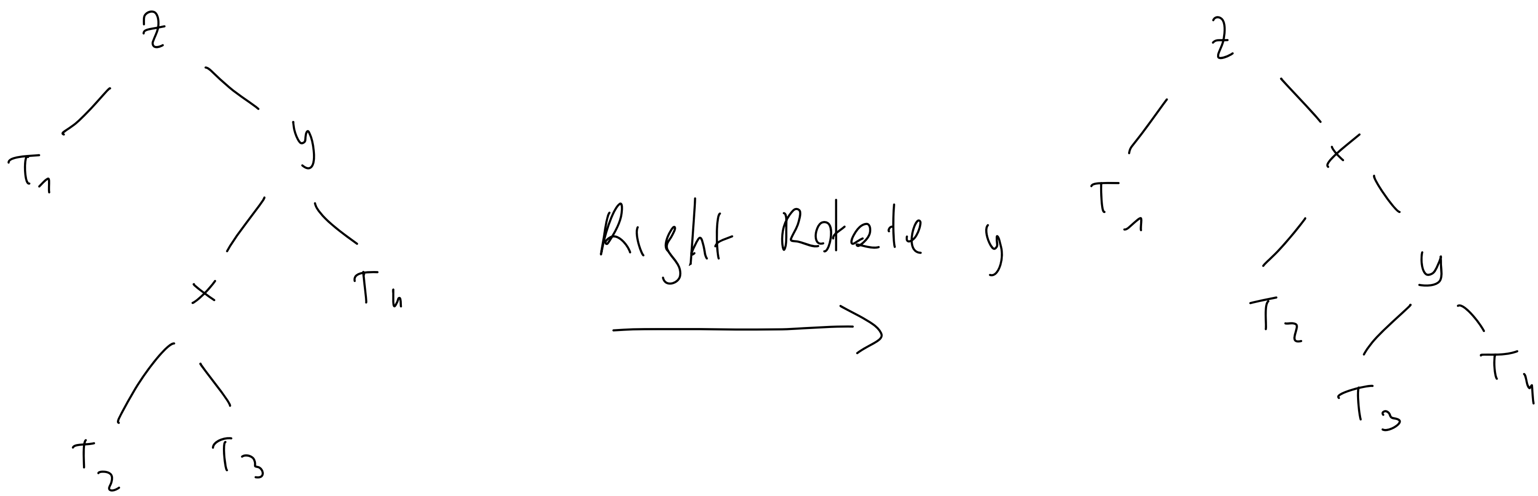




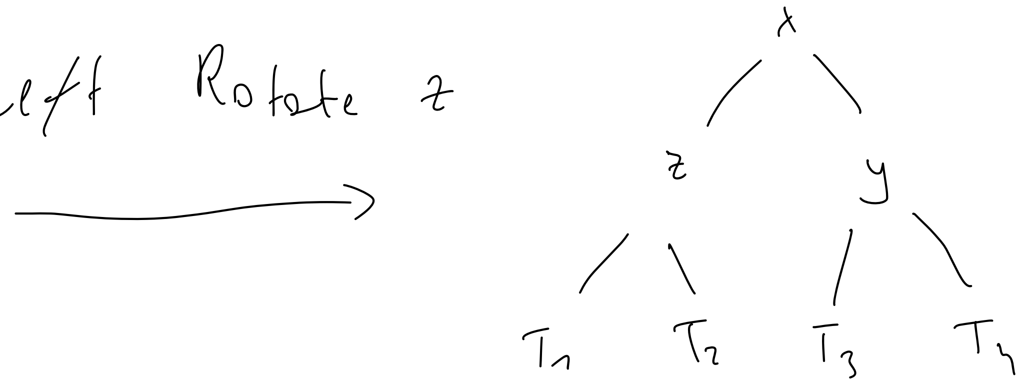
Right Right:



Right Left:



Left Rotate z



Exercise 5.5 *AVL trees* **(1 point)**.

(a) Draw the tree obtained by inserting the keys 3, 8, 6, 5, 2, 9, 1 and 0 in this order into an initially empty AVL tree. Give also all the intermediate states after every insertion and before and after each rotation that is performed during the process.

AVL tree condition:

STRUKTUR-BEDINGUNG

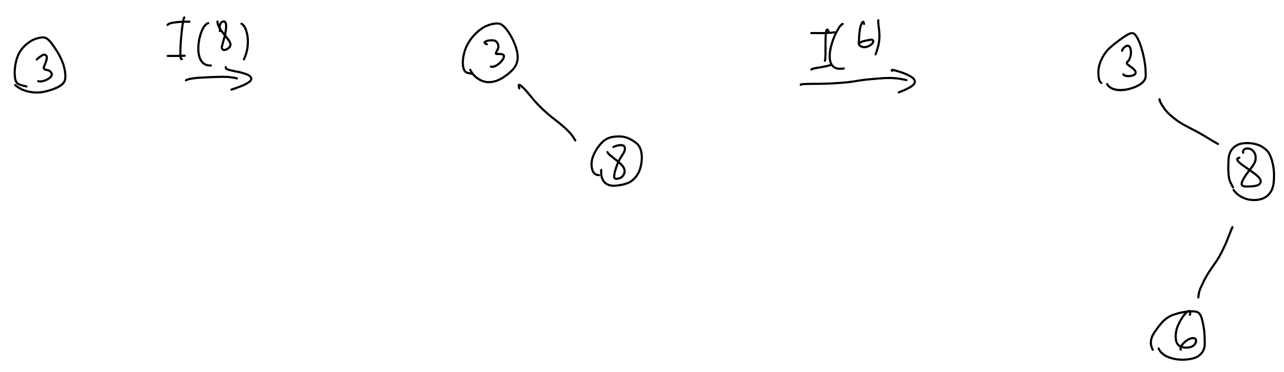
Sei nun T ein Baum mit der Wurzel v, der linke Teilbaum von v sei $T_l(v)$, und $T_r(v)$ der rechte (man beachte, dass sowohl $T_l(v)$ als auch $T_r(v)$ ein Blatt, d.h. ein Nullzeiger, sein können). Wir definieren die *Balance* des Knotens v als

$$\mathrm{bal}(v) := h(T_r(v)) - h(T_l(v)), \tag{104}$$

AVL-BEDINGUNG

wobei $h(T_l(v))$ bzw. $h(T_r(v))$ die Höhe von $T_l(v)$ bzw. $T_r(v)$ angeben. Die *AVL-Bedingung* besagt nun, dass für *alle* Knoten v des Baums $\mathrm{bal}(v) \in \{-1, 0, 1\}$ gilt.

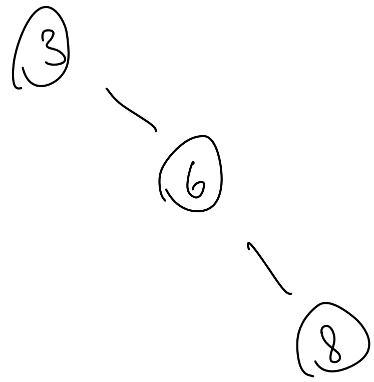
Put differently: for any vertex v, the height of the left and right subtree of v can differ by at most one.



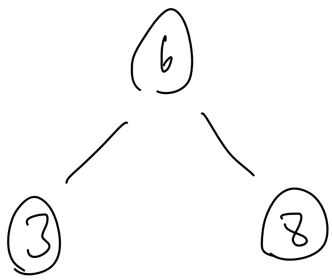
AVL condition at root not given.

Case right left:

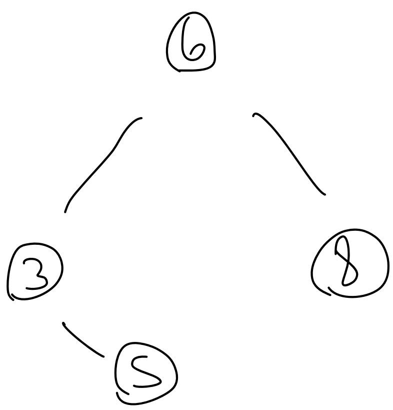
Right Rotate 8



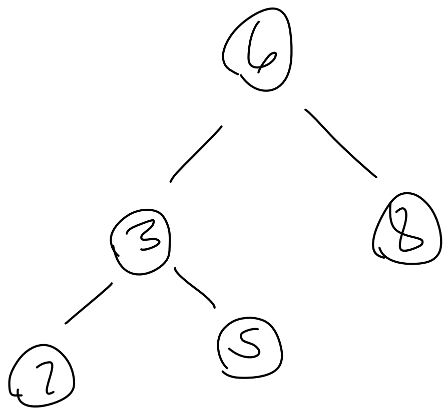
Left Rotate 8



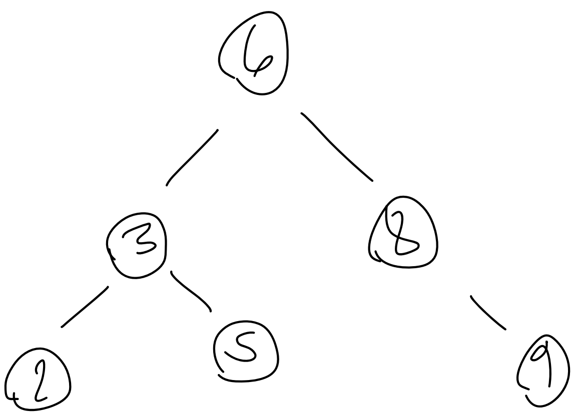
I(5)



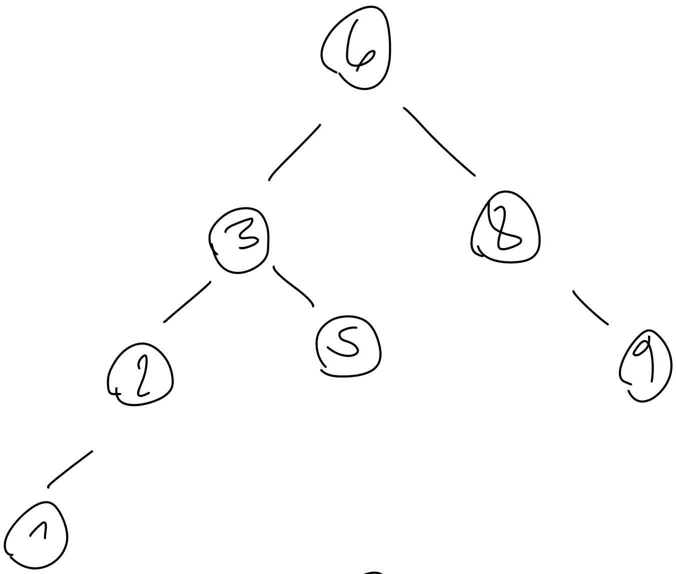
I(2)



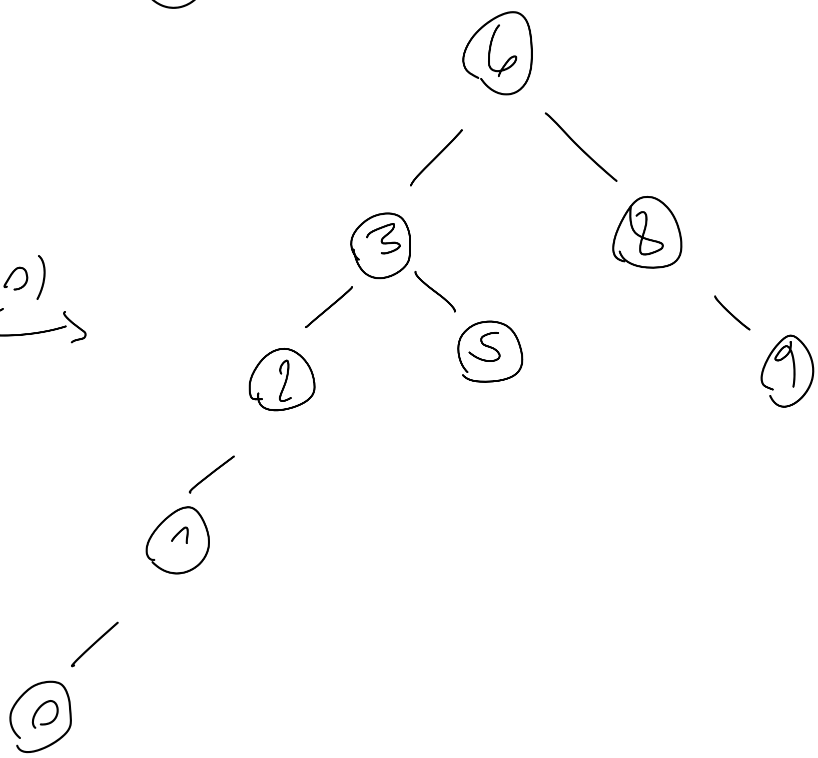
I(9)



I(1) →



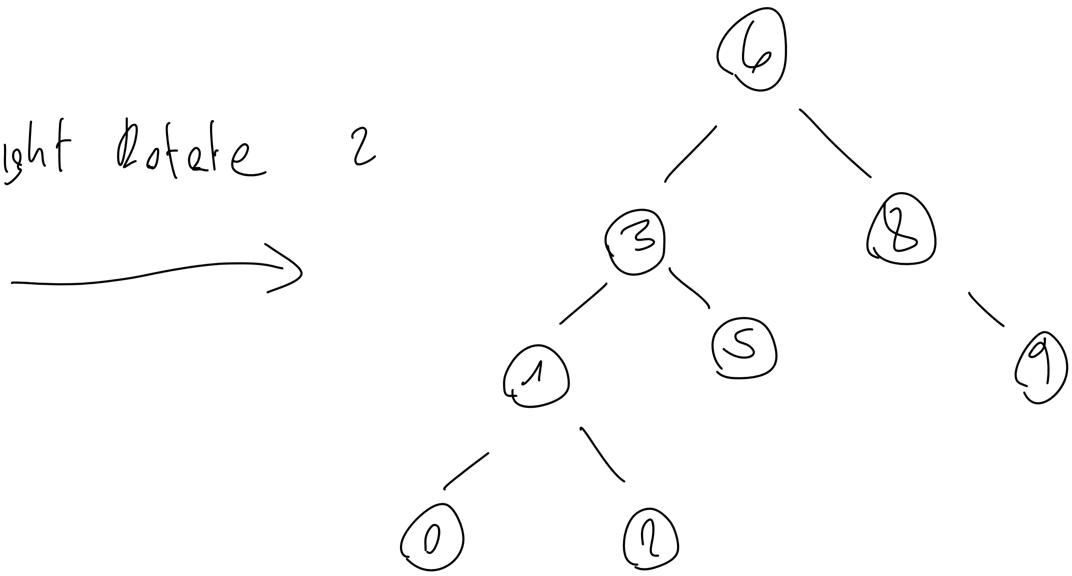
I(0) →



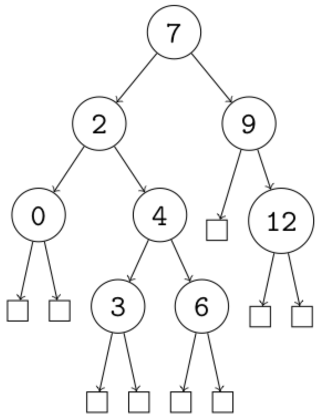
AVL condition at (2) not given.

Case left left:

Right Rotate 2

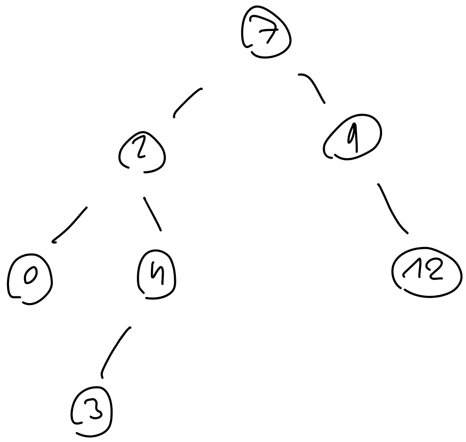


(b) Consider the following AVL tree.

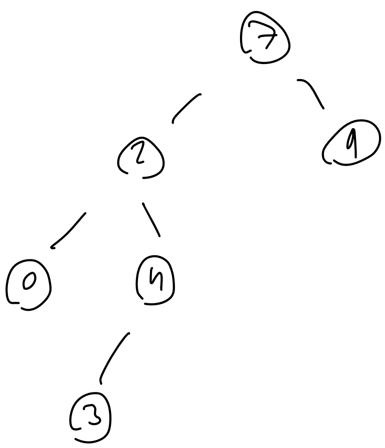


Draw the tree obtained by deleting 6, 12, 7 and 4 in this order from this tree. Give also all the intermediate states after every deletion and before and after each rotation that is performed during the process.

D(6) →



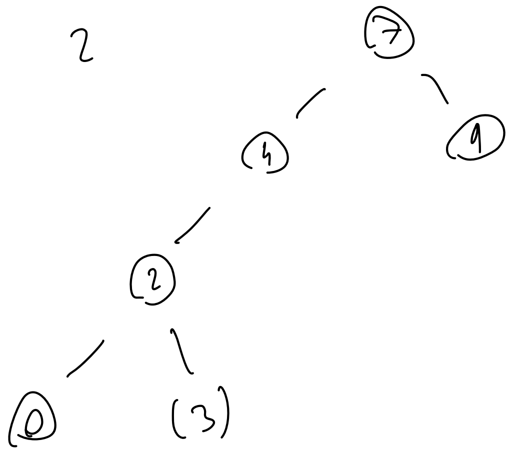
D(12) →



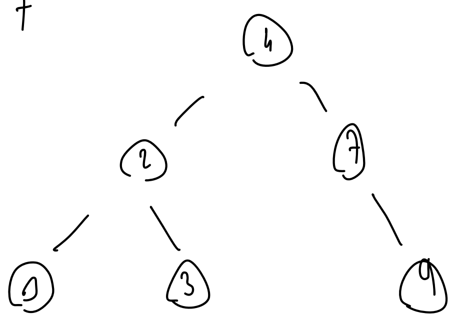
AVL condition at 7 not given.

Case left right.

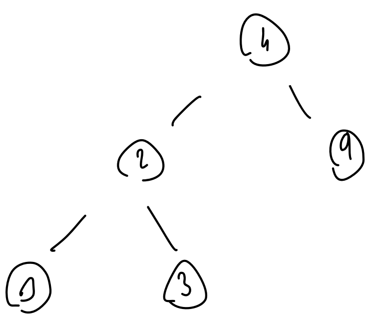
Lef Rotate 2



Right Rotate 7



D(7)



D(4)