Eidgenössische
Technische Hochschule
Zürich

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

Departement of Computer Science
Markus Püschel, David Steurer
Johannes Lengler, Gleb Novikov, Chris Wendler, Ulysse Schaller

9. November 2020

Algorithms & Data Structures — Exercise sheet 8 — HS 20

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

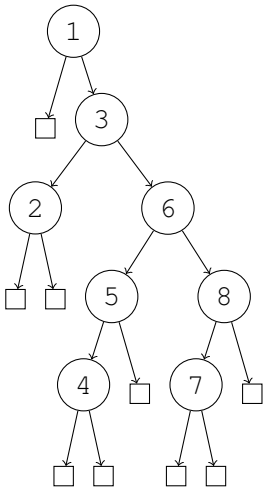
Points: ____

Submission: On Monday, 16 November 2020, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 8.1 *Search Trees* **(1 point)**.

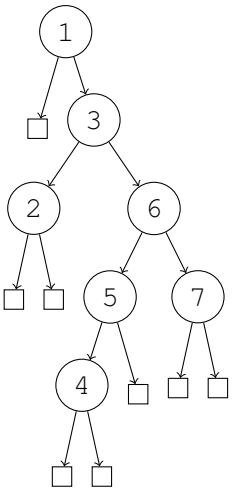
a) Draw the resulting tree when the keys 1, 3, 6, 5, 4, 8, 7, 2 in this order are inserted into an initially empty binary (natural) search tree.

Solution:

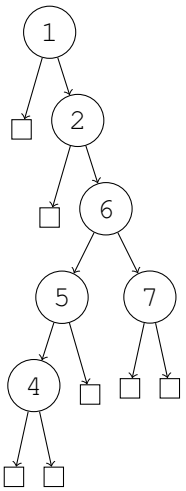


b) Delete key 8 in the above tree, and afterwards key 3 in the resulting tree.

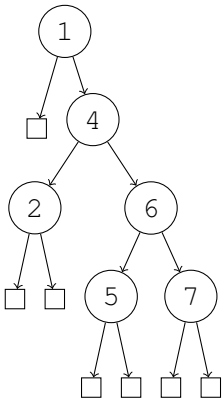
Solution: Key 8 has one child 7, so it can just be replaced by 7:



Key 3 must either be replaced by its predecessor key, 2, or its successor key, 4. If key 3 is replaced by its predecessor:



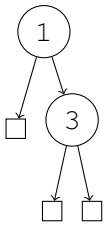
If key 3 is instead replaced by its successor:



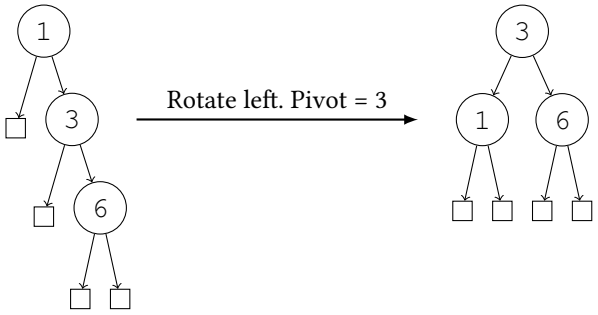
c) Draw the resulting tree when the keys are inserted into an initially empty AVL tree. Give also the intermediate states before and after each rotation that is performed during the process.

Solution:

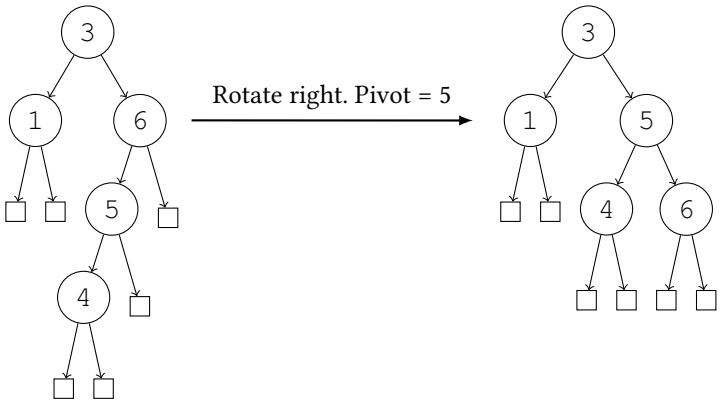
Insert 1 and then 3:



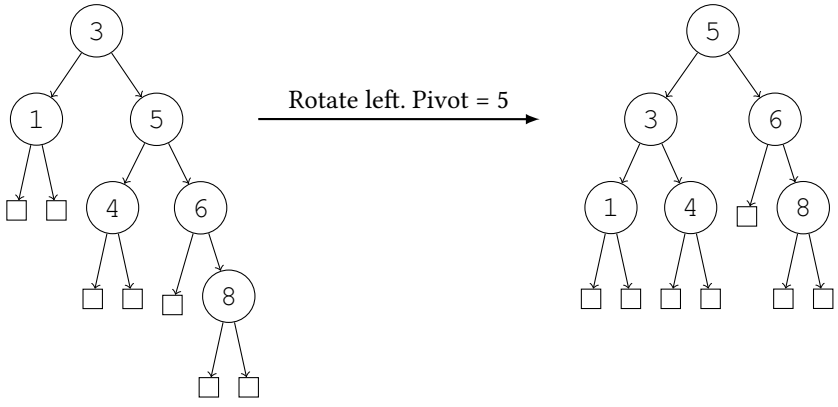
Insert 6:



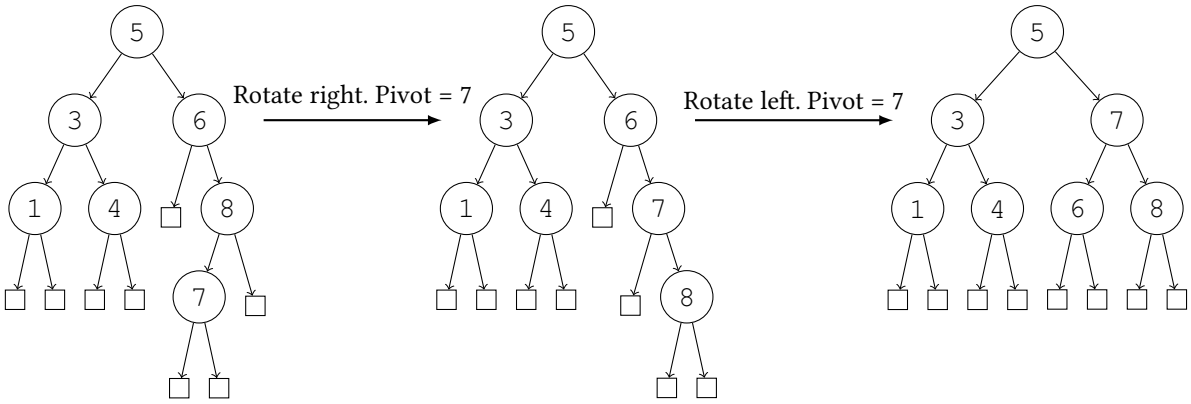
Insert 5 and then 4:



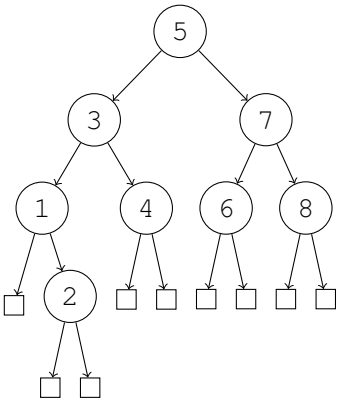
Insert 8:



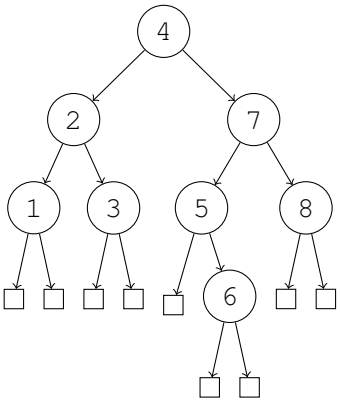
Insert 7:



Insert 2:



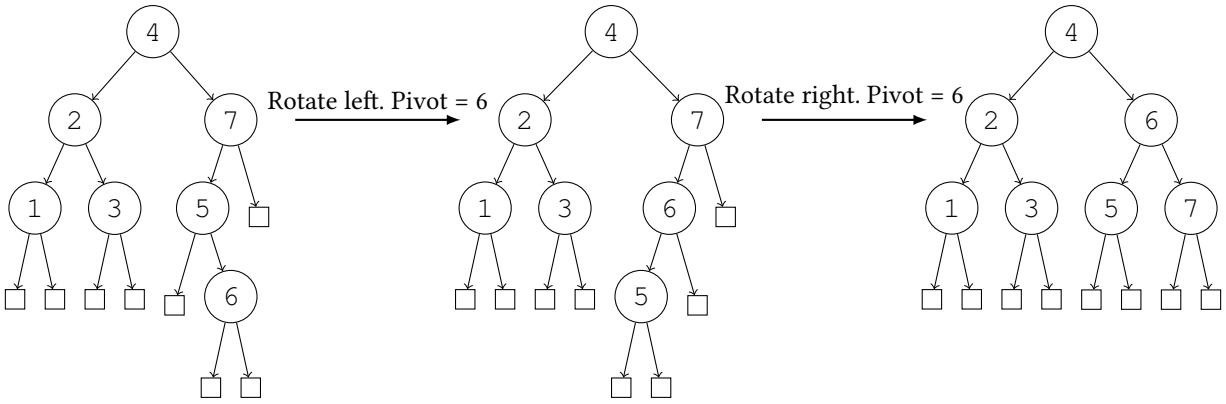
d) Consider the following AVL tree:



Delete key 8 in this tree, and afterwards key 2 in the resulting tree. Give also the intermediate states before and after each rotation is performed during the process.

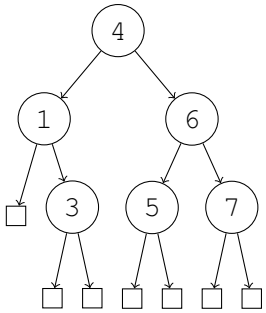
Solution:

Delete 8:

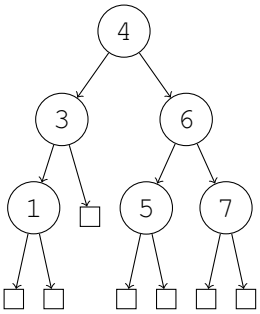


Delete 2:

Key 2 can either be replaced by its predecessor key, 1, or its successor key, 3. If key 2 is replaced by its predecessor:



If key 2 is instead replaced by its successor:



Exercise 8.2 *Finding an invariant (exercise 6.5. revisited).*

A	1	3	2	1
3	2	1	1	B

Figure 1: Runner problem for a cost array of size 2×5.

In this exercise we revisit exercise 5 from sheet 6, where we had a runner who wants to run from A to B in Fig. 1. There are two lanes available. One is represented by the first row and the other by the second row. On some sections, the first lane is faster than the second lane, and vice versa. The runner

can change lanes at any time, but this costs 1 minute every time. In this exercise, you are supposed to provide a dynamic programming algorithm that computes the optimal track.

Formally, the problem is defined in terms of a cost array $c \in \mathbb{N}^{2\times n}$. In Fig. 1 $n = 5$. Now, the runner starts at position $(1, 1)$ and wants to run to $(2, n)$. Running along a lane from field (i, j) to the field $(i, j + 1)$ requires $c_{i,j+1}$ minutes. Changing lanes from field $(1, j)$ to $(2, j)$ requires $1 + c_{2,j}$ minutes, and from field $(2, j)$ to $(1, j)$ requires $1 + c_{1,j}$ minutes.

Consider the following implementation of our DP solution for computing the minimal time required to get from A to B (note that in this implementation `previous` and `DP` are both one dimensional arrays with 2 elements):

Algorithm 1 `optimal_time`(c)

```
previous ← [0, 1 + c_{2,1}]
DP ← [0, 0]
for j = 2, ..., n do
    DP[1] = min(previous[1] + c_{1,j}, previous[2] + c_{2,j} + c_{1,j} + 1)
    DP[2] = min(previous[2] + c_{2,j}, previous[1] + c_{1,j} + c_{2,j} + 1)
    previous ← DP
    // Your invariant from a) must hold here.
return previous[2]
```

Note that this implementation is more memory-efficient than the solution of 6.5. It only requires $O(1)$ additional memory. (It also has disadvantages: it only computes the optimal time,not the corresponding track.) Your task is to prove that this algorithm is correct.

a) Formulate an invariant $INV(j)$ that holds after the iteration with the running index j of the for loop.

Solution: We are going to use the following invariant $INV(j)$: "After iteration j, `previous`$[1]$ contains the minimal time required to reach $(1, j)$ and `previous`$[2]$ the minimal time required to reach $(2, j)$."

Moreover, we will show that the same statement holds for $j = 1$ after initialization `previous` (before the first for loop).

b) Prove the correctness of the algorithm `optimal_time` by induction using your invariant.

Solution:

Base case $j = 1$ The initialization of `previous` before the loop sets it to the minimal times required to reach $(1, 1)$ and $(2, 1)$ respectively. We can think of this as the first loop iteration $j = 1$. Thus, $INV(1)$ holds.

Induction hypothesis For some $j \geq 1$ our invariant $INV(j)$ holds.

Induction step $j \to j + 1$ In the iteration $j+1$, `DP`$[1] = \min($`previous`$[1]+c_{1,j+1},$ `previous`$[2]+ c_{2,j+1}+c_{1,j+1}+1)$ is computed. Because of our induction hypothesis `previous` contains the minimal times to reach $(1, j)$ and $(2, j)$ respectively. We claim that `DP`$[1]$ is set to the minimal time required to reach $(1, j + 1)$. Indeed, every way to get there must either leave the j-th column from $(1, j)$ or $(2, j)$. In the first case, the additional time to reach $(1, j + 1)$ is exactly $c_{1,j+1}$, in the second case it is $c_{2,j+1} + c_{1,j+1} + 1$, since the only way to reach $(1, j + 1)$ is to switch lanes from $(2, j+1)$. The minimum selects the better one of the two, so we have indeed

shown that DP[1] is set to the minimal time required to reach $(1, j+1)$. The exact same argumentation applies to the computation $\texttt{DP}[2] = \min(\texttt{previous}[2] + c_{2,j+1}, \texttt{previous}[1] + c_{1,j+1} + c_{2,j+1} + 1)$. Accordingly, DP[2] contains thanks to our induction hypothesis the minimal time required to reach $(2, j+1)$. Now, previous is set to DP which concludes our step.

The algorithm optimal_time returns previous[2] which contains the minimal time required to reach $(2, n)$ (or in other words B). Therefore, by the principle of mathematical induction optimal_time computes the minimal time required to reach B from A.

Exercise 8.3 *Exponential bounds for a sequence defined inductively.*

Consider the sequence $(a_n)_{n\in\mathbb{N}}$ defined by

$$\begin{aligned} a_0 &= 1, \\ a_1 &= 1, \\ a_2 &= 2, \\ a_i &= a_{i-1} + 2a_{i-2} + a_{i-3} \quad \forall i \geq 3. \end{aligned}$$

The goal of this exercise is to find exponential lower and upper bounds for a_n.

a) Find a constant $C > 1$ such that $a_n \leq \mathcal{O}(C^n)$ and prove your statement.

Solution:

Intuitively, the sequence $(a_n)_{n\in\mathbb{N}}$ seems to be increasing. Assuming so, we would have

$$a_i = a_{i-1} + 2a_{i-2} + a_{i-3} \leq a_{i-1} + 2a_{i-1} + a_{i-1} = 4a_{i-1},$$

which yields

$$a_n \leq 4a_{n-1} \leq \ldots \leq 4^n a_0 = 4^n.$$

This only comes from an intuition, but it is a good way to guess what the upper bound could be. Now let us actually prove (by induction) that $a_n \leq 4^n$ for all $n \in \mathbb{N}$.

Induction Hypothesis. We assume that for $k \geq 2$ we have

$$a_k \leq 4^k, \qquad a_{k-1} \leq 4^{k-1}, \qquad a_{k-2} \leq 4^{k-2}. \tag{1}$$

Base case $k = 2$**.** Indeed we have $a_0 = 1 \leq 4^0$, $a_1 = 1 \leq 4^1$ and $a_2 = 2 \leq 4^2$.

Inductive step ($k \to k+1$**).** Let $k \geq 2$ and assume that the induction hypothesis (1) holds. To show that it also holds for $k+1$, we need to check that $a_{k+1} \leq 4^{k+1}$, $a_k \leq 4^k$ and $a_{k-1} \leq 4^{k-1}$. The two last inequalities clearly hold since they are part of the induction hypothesis, so we only need to check that $a_{k+1} \leq 4^{k+1}$. Indeed,

$$a_{k+1} = a_k + 2a_{k-1} + a_{k-2} \overset{(1)}{\leq} 4^k + 2 \cdot 4^{k-1} + 4^{k-2} \leq 4^k + 2 \cdot 4^k + 4^k = 4 \cdot 4^k = 4^{k+1}.$$

Thus, $a_n \leq 4^n$ for all $n \in \mathbb{N}$. In particular, we have shown that $a_n \leq \mathcal{O}(C^n)$ for $C = 4 > 1$.

b) Find a constant $c > 1$ such that $a_n \geq \Omega(c^n)$ and prove your statement.

Solution:

If we again assume that the sequence is increasing, we would get

$$a_i = a_{i-1} + 2a_{i-2} + a_{i-3} \geq a_{i-3} + 2a_{i-3} + a_{i-3} = 4a_{i-3},$$

which yields

$$a_n \geq 4a_{n-3} \geq \ldots \geq 4^{\lfloor n/3 \rfloor} a_0 = 4^{\lfloor n/3 \rfloor}.$$

So we will aim to prove a lower bound of the form $a_n \geq \varepsilon \cdot 4^{n/3}$ for some constant $\varepsilon > 0$. We see that taking $\varepsilon := \min\{1, 4^{-1/3}, 2 \cdot 4^{-2/3}\} = 4^{-1/3}$ will make the inequality satisfied for the base case, so let's prove by induction that $a_n \geq 4^{-1/3} 4^{n/3}$ for all $n \in \mathbb{N}$.

Induction Hypothesis. We assume that for $k \geq 2$ we have

$$a_k \geq 4^{-1/3} 4^{k/3}, \qquad a_{k-1} \geq 4^{-1/3} 4^{(k-1)/3}, \qquad a_{k-2} \geq 4^{-1/3} 4^{(k-2)/3}. \tag{2}$$

Base case $k = 2$**.** Indeed we have $a_0 = 1 \geq 4^{-1/3} \cdot 4^0$, $a_1 = 1 \geq 4^{-1/3} 4^{1/3}$ and $a_2 = 2 \geq 4^{1/3} = 4^{-1/3} 4^{2/3}$.

Inductive step ($k \to k+1$**).** Let $k \geq 2$ and assume that the induction hypothesis (2) holds. To show that it also holds for $k+1$, we need to check that $a_{k+1} \geq 4^{-1/3} 4^{(k+1)/3}$, $a_k \geq 4^{-1/3} 4^{k/3}$ and $a_{k-1} \geq 4^{-1/3} 4^{(k-1)/3}$. The two last inequalities clearly hold since they are part of the induction hypothesis, so we only need to check that $a_{k+1} \geq 4^{-1/3} 4^{(k+1)/3}$. Indeed,

$$\begin{aligned} a_{k+1} = a_k + 2a_{k-1} + a_{k-2} &\overset{(2)}{\geq} 4^{-1/3} \left(4^{k/3} + 2 \cdot 4^{(k-1)/3} + 4^{(k-2)/3}\right) \\ &\geq 4^{-1/3} \left(4^{(k-2)/3} + 2 \cdot 4^{(k-2)/3} + 4^{(k-2)/3}\right) = 4^{-1/3} \cdot 4 \cdot 4^{(k-2)/3} = 4^{-1/3} 4^{(k+1)/3}. \end{aligned}$$

Thus, $a_n \geq 4^{-1/3} 4^{n/3}$ for all $n \in \mathbb{N}$. In particular, we have shown that $a_n \geq \Omega(c^n)$ for $c = 4^{1/3} > 1$.

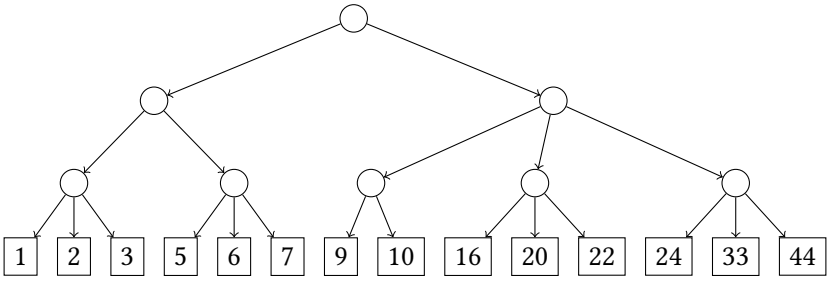
Remark. One can actually show that $a_n = \Theta(\phi^n)$, where $\phi \approx 2.148$ is the unique positive solution of the equation $x^3 = x^2 + 2x + 1$.

Exercise 8.4 *The (2, 3)-tree datastructure* **(2 points)**.

A (2,3)-tree is a tree in which each inner node has between 2 and 3 children. Additionally, all leaves are at the same depth. There are two different types of inner nodes:

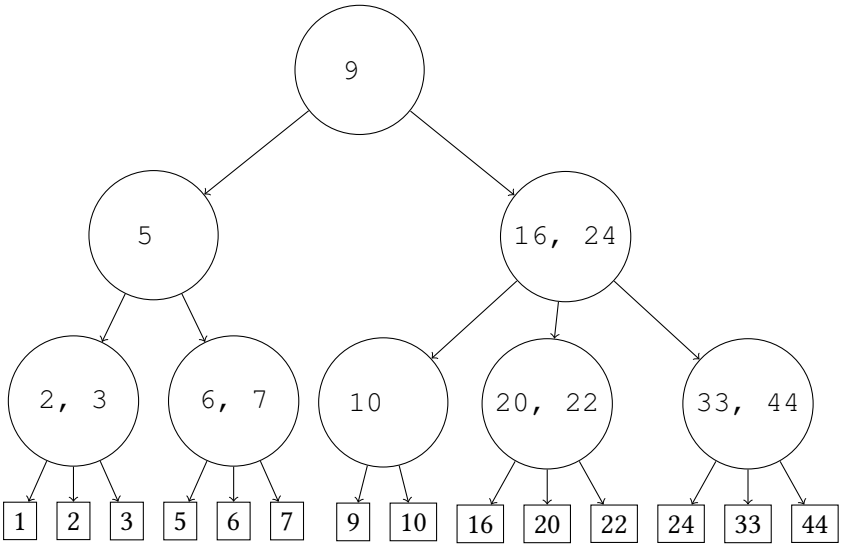
1. **2-nodes**: A 2-node is a node with two children l and r of the same height.
2. **3-nodes**: A 3-node is a node with three children l, m and r of the same height.

The keys are stored in the leafs and ordered from left to right.



a) Come up with a way to augment the inner nodes with information that enables efficient search for keys. Provide pseudocode of your search method.

Solution: We augment each 2-node by one integer a that is larger than all the keys in its subtree l and smaller or equal to the smallest key in r. Similarly, we augment each 3-node by two integers a and b such that all keys in l are smaller than a, all keys in m are greater or equal a, all keys in m are smaller than b and all keys in r are greater or equal b. One way to augment the example above is:



Using our augmentation, searching for a key in a (2, 3)-tree becomes similar to searching for a key in a binary tree. Let T be the root of our $(2, 3)$-tree and k the key we are looking for.

Algorithm 2 `search` (T, k)

```
if T is a leaf then
    if T.key = k then
        return We found the key.
    else
        return The (2,3)-tree does not contain the key.
else if T is a 2-node then
    if k < T.a then
        return search(T.l, k)
    else
        return search(T.r, k)
else if T is a 3-node then
    if k < T.a then
        return search(T.l, k)
    else if k < T.b then
        return search(T.m, k)
    else
        return search(T.r, k)
```

b) Find a way to insert a new key at the correct position. Your method should run in time $\mathcal{O}(h)$, where h is the height of the (2,3)-tree. Provide pseudocode of your insertion method. You may ignore the information in the inner nodes for this task, i.e., you don't need to describe how to update this

information. Also, you don't need to discuss how exactly you would store the nodes. You can assume that from a node, you can access parent, siblings and children in constant time.

Hint: *When the parent of the new leaf has more than 3 leafs after the insertion, split it into two. Be cautious, what happens to the 'grandparent'?*

Solution:

Algorithm 3 `insert(`T, k`)`

if k already is in T **then**
    We are done.
else
    Find the parent of the key k (i.e., traverse the tree as in `search`).
    We end up with the largest leaf L with a smaller key.
    We add a new leaf with the key k to the parent P of L, on the right side of L.
    (Or as leftmost child of P if k is smaller than all keys in T.)
    `repair(`P`)` // P might have too many children.

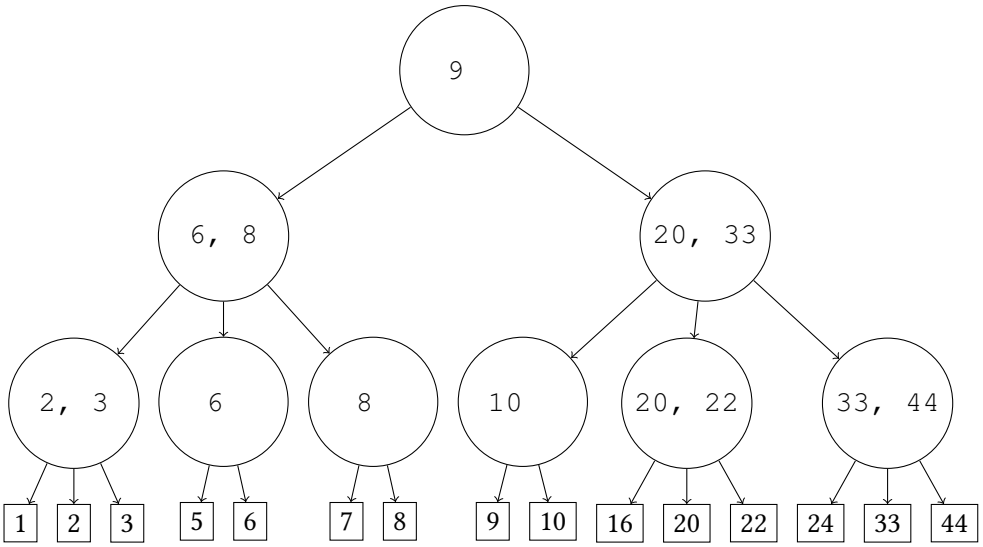
Algorithm 4 `repair(`P`)`

if P has at most three children **then**
    Nothing to do.
else // P has four children
    if P is not the root **then**
        We split P into two nodes P_1 and P_2, i.e., P_1 and P_2 are now two consecutive children of the parent S of P. We split the four children of P between P_1 and P_2: We put the two smaller children below P_1 and the larger ones below P_2.
        `repair(`S`)` // S might have too many children.
    else // P is the root
        We split P into P_1 and P_2 as before, but we create a new root R with children P_1 and P_2.

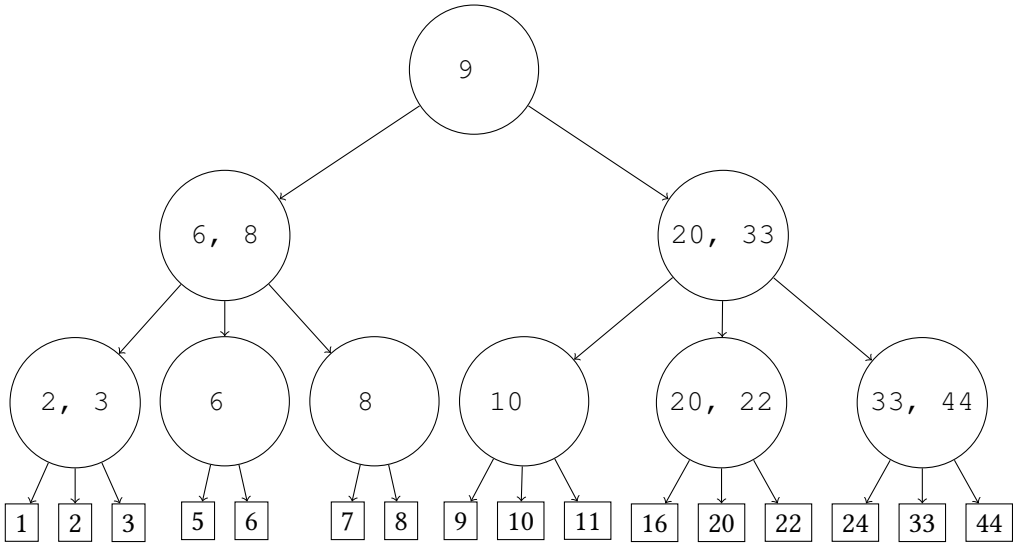
The insertion runs in time $\mathcal{O}(h)$, because finding the position to insert the new key is in $\mathcal{O}(h)$ as we only have to follow one path from the root to a leaf. Adding the new key to a 2-node requires only constant time. Each repair procedure requires constant time before the recursive call, and in each repair procedure we go up one level (or stop). So the number of repair procedure is at most the number of levels.

c) Insert the key 8 into the example above and draw the result. Now, insert the key 11 into the tree from the previous step and draw the result. Finally, insert the key 4 into the tree from the previous step and draw the result.

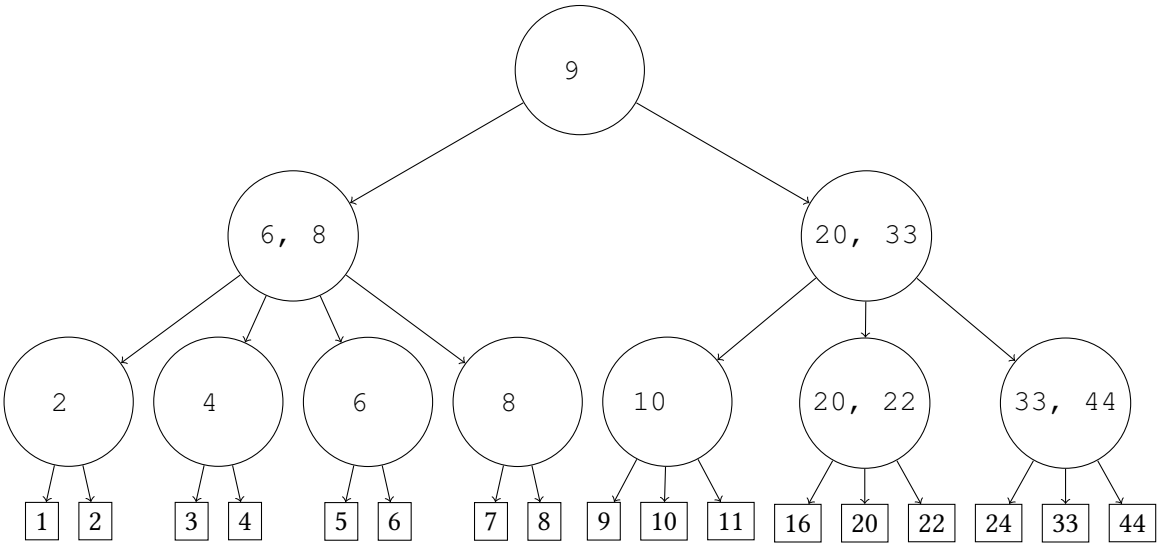
Solution: Inserting 8 yields:



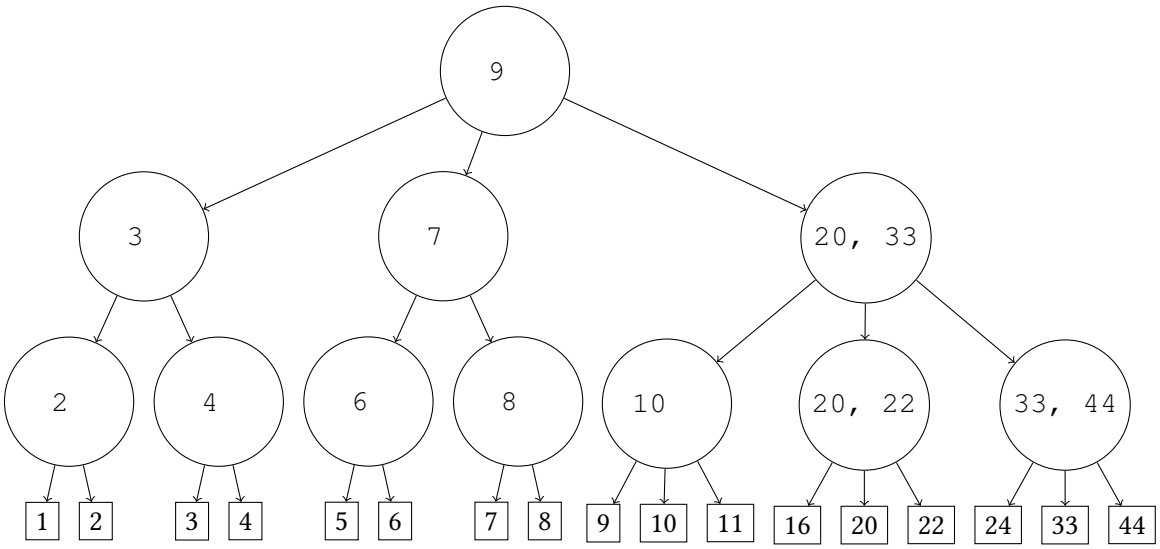
Now, inserting 11 yields:



When inserting 4 we need two repair steps. First repair step:



Now the '6, 8'-node has four children and we need to perform a second repair step:



d) Prove asymptotically matching upper and lower bounds for the height h of a (2,3)-tree with n leaves. "Asymptotically matching" means that they are of the form $h = \mathcal{O}(f(n))$ and $h = \Omega(f(n))$ for the same function $f(n)$.

Solution: On the zero-th level, there is exactly one vertex (the root). Each vertex has either 2 or 3 children. So if there are x_i vertices on level i, then th number of vertices on level $i+1$ is bounded by $2x_i \leq x_{i+1} \leq 3x_i$. By a trivial induction, the number of vertices in level i is therefore between 2^i and 3^i. In particular, since the leaves are the vertices in level h, the number n of leaves satisfies $2^h \leq n \leq 3^h$. Solving both inequalities for h yields $\log_3(n) \leq h \leq \log_2(n)$, and therefore $h = \Theta(\log n)$.