1. Give a small example of adding a number to an AVL-tree causing an unbalanced tree of type right-right. Rebalance your tree.

Solution:

By adding a node with key 3 we get a binary search tree that is not an AVL-tree. We rebalance using a single rotation which yields (in one step) an AVL-tree:

```
1
  2 3
```

2. Give a small example of adding a number to an AVL-tree causing an unbalanced tree of type right-left. Rebalance your tree.

Solution:

By adding a node with key 2 we get a binary search tree that is not an AVL-tree. We rebalance using a double rotation which yields (in one step) an AVL-tree:

```
1 2
  3
```

3. Give a small example of removing a number from an AVL-tree causing an unbalanced tree of type right-right. Rebalance your tree.

Solution:

By removing a node with key 1 we get a binary search tree that is not an AVL-tree. We rebalance using a single rotation which yields (in one step) an AVL-tree:
4. Remove from the following AVL-tree the node labelled 8, then add a node labelled 9.

Solution:

What is the result of first adding 9 and then remove 8?
The first step is to remove the node with key 8. On that position we put the node with key 9.

Next we need to rebalance the tree.
Now we have an AVL-tree. We add a node with key 9.

We do not need to rebalance.

First adding 9 and next removing 8 yields the following AVL-tree:

5. Remove the node labelled 9 from the following AVL-tree:
Solution:
We remove the node with key 9.

We need to rebalance.

And we need a next rebalance step.
6. Give an example showing that the difference in depth between two leaves in an AVL-tree can be larger than 2.

Solution:

In the next picture we do not give the complete tree. A circle is an internal node; the number indicates the height of the sub-tree starting there. The difference in depth between the leaf left (not explicitly in the picture) and the leaf right (not explicitly in the picture) is 3.

7. Design an algorithm to determine the smallest number in an AVL tree. What is the worst-case time complexity of your algorithm?

8. True or false: The smallest and the largest number in an AVL tree can always be found on either the last or the one-before-last level?

Not true; see for example the shape of the AVL tree in exercise 9.

9. Suppose that we add to the set of stack operations not only the operation \texttt{multipop}(S, k) but also the operation \texttt{multipush}(k, e) which pushes \(k\) times
the element $e$ on the stack. Is the amortized cost of stack operations still in $O(1)$?

Solution:

We have seen that for a stack with the usual operations for push and pop with in addition the new operation for multipop the amortized complexity of an operation is in $O(1)$, because we can show that a worst-case sequence of $n$ operations is in $O(n)$, and then we take as amortized cost for one operation the average cost, which is in $O(1)$.

If we moreover add the binary operation multipush the amortized complexity per operation is no longer elementary, because a sequence of $n$ operations is no longer in $O(n)$. Consider a sequence of $n$ operations of multipush$(k, e)$ followed by multipop$(S, k)$. The time complexity of such a sequence is in $O(n \cdot k)$ where $k$ is not necessarily small compared to $n$. In particular, given a fixed $c$ we cannot show that for all $k$ we have $n \cdot k \leq c \cdot n$. So the worst-case time complexity of such a sequence is not in $O(n)$, and we can no longer derive that the amortized cost of an operation (taken as the average cost for an operation in a worst-case sequence) is in $O(1)$.

Alternatively we can consider a sequence of $n$ operations multipush; we indeed assume there is no bound at all on the stack.

10. We consider the usual stack with operations push and pop. We assume that the size of the stack never exceeds $k$ (for some fixed natural number $k$). We add an operation that after $k$ operations makes a copy of the stack. Show that the cost of $n$ stack operations is in $O(n)$; use the accounting method.

(In addition, we can consider the stack with operations for push, pop, and multipop where the operation for copy is added, assuming again that the size of the stack never exceeds $k$.)

Solution:

The actual costs for the operations are as follows: 1 for push and also 1 for pop, and the size of the stack for copy. Since the size of the stack is bounded by $k$ the actual cost for copy is less than or equal to $k$.

We assign amortized costs as follows: 2 for push, 2 for pop, and 0 for copy. The intuition is that with push we pay for the operation itself and we save 1 as credit for a future copy operation. Similarly, with pop we pay for the operation itself and we save 1 as credit for a future copy operation. After $k$ operations, the credit saved for copying is $k$ and we can use this for copying the stack. Then the cost of $n$ stack operations is in $O(n)$.

If in addition we also have multipop then we proceed in a similar way. We assign amortized costs as follows: 3 for push, 1 for pop, and 1 for multipop. The intuition is that the payment of 3 for push is used as follows: 1 to pay for the operation itself, 1 to store as credit for a future pop (either a normal one or one inside multipop), and 1 to store as credit for a future
copy. The payment of 1 for pop is used to store as credit for a future copy; the operation pop is payed from the credit given by push. The payment of 1 for multipop is used to store as credit for a future copy; the operation multipop is payed from the credit given by push. With these amortized costs per operation a sequence of \( n \) operations is in \( \mathcal{O}(n) \).

11. Use the accounting method to analyze the amortized complexity of increment in the binary counter. (Cf the lecture.)

Solution:

The actual cost of an increment operation is the number of bits that flips. We define amortized costs as follows:

2 for flipping a bit from 0 to 1, and 0 for flipping a bit from 1 to 0. From the amortized cost of 2 for the 01 flip, we use 1 to ‘pay’ for the actual cost, and the remaining 1 is stored as credit. Intuitively, every bit 1 has its own piggy bank containing 1 to pay for a possibly later occurring 10 flip.

<table>
<thead>
<tr>
<th>value</th>
<th>actual cost</th>
<th>amortized cost</th>
<th>total credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

After every step the total credit remains at least 0. The reason is that the number of 1 in the counter does not become negative. The operation increment performs at most one 01 flip, with amortized cost 2. The amortized cost for \( n \) operations in \( \mathcal{O}(n) \) bounds the actual cost for \( n \) operations.

12. Reconsider the exercise from exercise class 5 concerning implementing a queue using two stacks. Give the amortized complexity of the operations enqueue and dequeue from the implementation of the queue using two stacks. Use aggregate analysis and the accounting method.

Solution:

We consider the implementation of a queue using two stacks.

A naive analysis of the worst-case time complexity: The worst-case time complexity of enqueue is in \( \mathcal{O}(1) \) because it is implemented as push. The worst-case time complexity of dequeue is in \( \mathcal{O}(n) \), with \( n \) the number of elements on the stack, because in the worst case we need to remove all \( n \) elements from the one stack and to put them on the other stack, and then remove one element (\( 2n + 1 \) operations).

We show that using amortized analysis we can give a more tight bound. In a sequence of \( k \) operations, the total cost of operations enqueue is at most \( k \). An element can only be dequeued if it has been enqueued first. One dequeue costs 3 steps (remove from the first stack, add to the second
stack, remove from the second stack). So in a sequence of \( k \) operations, the total cost of operations dequeue is bounded by 3 times the number of enqueue operations, so at most \( 3k \). Hence, the total cost of a worst-case sequence of \( k \) operations is at most \( 4k \). So a worst-case sequence of length \( k \) is in \( \mathcal{O}(k) \) and hence the amortized cost of one operation, takes as the average cost, is in \( \mathcal{O}(1) \).

Using the accounting method we proceed as follows. The actual cost of enqueue is 1. The actual cost of dequeue depends on the situation. We take the following amortized costs: 3 for enqueue and 1 for dequeue. For enqueue we use 1 to pay for the actual cost of the operation, and the additional 2 are stored to pay for removal from the first stack and addition to the second stack later on. For dequeue, we use that 2 as just explained, and we use the 1 of the amortized cost of dequeue to pay for removal from the second stack. In this case we have at any moment a positive credit. Hence for any sequence of operations, the amortized cost is an upper bound on the actual cost. Since the amortized complexity for a sequence of \( n \) operations is in \( \mathcal{O}(n) \), so is the actual cost.