University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Schneider

Algorithms and Data Structures
Winter Term 2019/2020
Sample Solution Exercise Sheet 7

Remark: For this exercise, the material of the eleventh video lecture is relevant.

Exercise 1: Binary Search Trees - Finding the Successor

(a) Give an algorithm that takes as input a node of a binary search tree and outputs its successor in the tree, i.e., the node with the next larger key in $\mathcal{O}(d)$, where d is the depth of the tree.

(b) Explain the correctness and the running time of your algorithm.

Sample Solution

(a)

Algorithm 1 `successor`(v)

```
if v.right ≠ None then                    ▷ if there is a right subtree, successor must be there
    crt ← v.right
    while crt.left ≠ None do
        crt ← crt.left
    return crt
else                                      ▷ otherwise some parent might be successor
    crt ← u
    prt ← crt.parent
    while prt ≠ None and prt.left ≠ crt do
        crt ← prt
        prt ← crt.parent
    return prt                            ▷ prt is None if no fitting successor is found
```

(b) **Runtime:** In the worst case, we traverse the tree from the root to a leaf or the other way around in one of the while loops, which takes at most $\mathcal{O}(d)$.

Correctness: If v has a right subtree, the a successor must be there, because all nodes outside of the right subtree of v are either smaller than the key of v or bigger than all keys in the right subtree of v. The smallest key in the right subtree is the "leftmost node". We obtain it by traversing to the right child of v and then to the left child as long as possible. We do this in the first half of the algorithm.

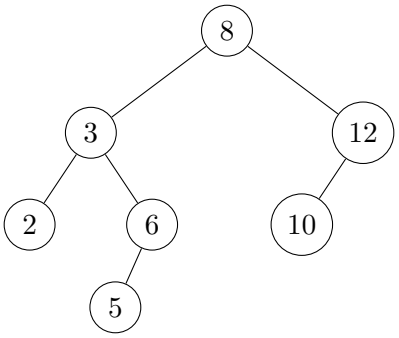
If v has no right subtree then another node might be its successor. A successor can not be in the left subtree of v since it contains only smaller keys. The successor can also not be in another subtree of an ancestor of v (a subtree which v is not member of). This is because if v is in the right subtree of an ancestor then all keys in the left subtree of that ancestor are smaller and can thus not be successors. If v is in the left subtree of an ancestor than the ancestor has a bigger key than v, and all keys in the right subtree of that ancestor have an even bigger key than said ancestor (which rules them out as successor).

So we only have to focus on ancestors of v. No ancestor of v where v is in the right subtree can be its successor, since it has a smaller key. Let w be the first ancestor of v, where v is in the left subtree of w. Then w has a bigger key than v. Any other ancestor of w (and thereby of v) has a key which is either smaller than that of v or larger than that of w, and both rules them out as successors. So w is the only candidate we care about (in case v has no right subtrees).

To obtain said ancestor w we traverse to parent nodes starting from v as long as the current node is a right child of its parent. We stop as soon as we are at the root, or we find a node which is left child of its parent in which case we return the parent as successor (this happens in the second half of the algorithm).

Exercise 2: AVL Trees

Consider the following AVL tree



(a) In the above tree, perform the operations `insert`(4), `insert`(7) and `insert`(1) and the necessary rotations to re-balance the AVL-tree. Draw the state of the tree after each operation.

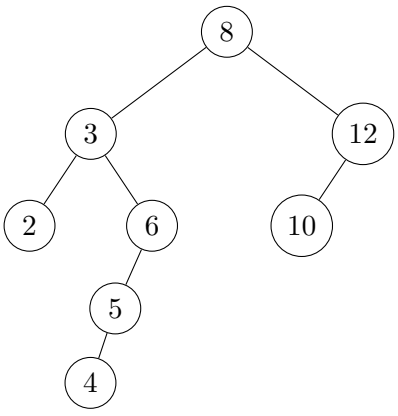
Remark: Inserting works the same as in binary search trees. Afterwards, for each ancestor of the inserted node (bottom up), repair the AVL condition (if violated) by performing an according rotation (left or right).

(b) In the resulting tree, perform the operations `delete`(5) and `delete`(7) and the necessary rotations to re-balance the AVL-tree. Draw the state of the tree after each operation.

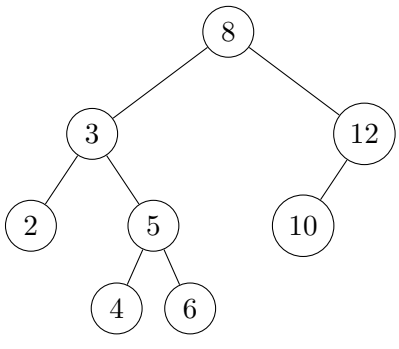
Remark: Deleting works the same as in binary search trees. Afterwards, starting at the position of the node that was used to replace the deleted key, for each ancestor (bottom up) repair the AVL condition (if violated) by performing an according rotation (left or right or double rotations).

Sample Solution

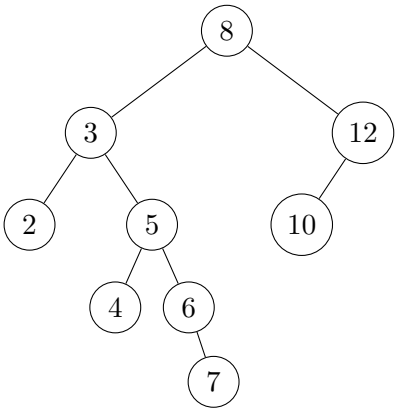
(a) `insert`(4), before balance:



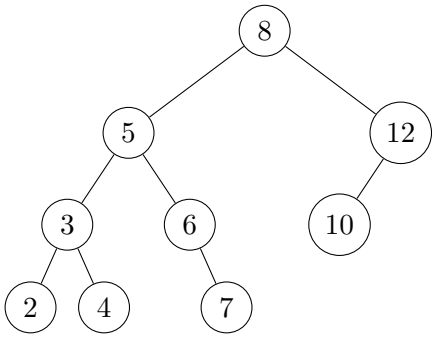
`insert`(4), after balance:



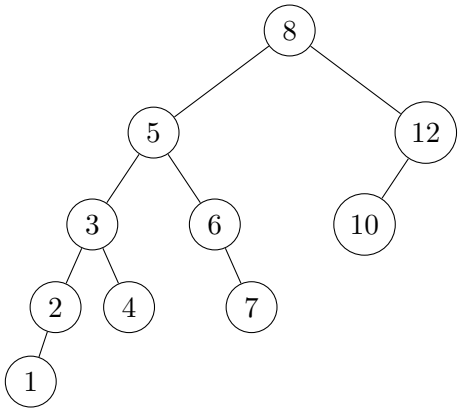
`insert`(7), before balance:



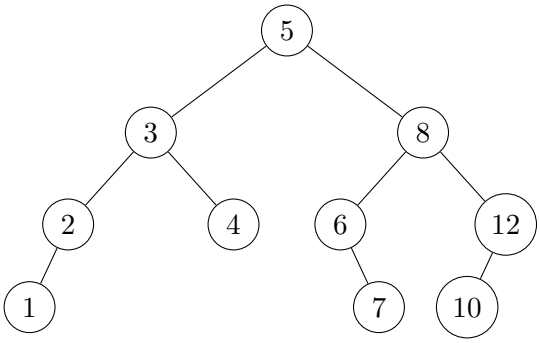
`insert`(7), after balance:



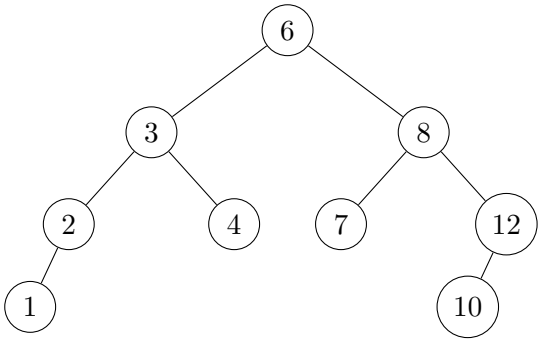
`insert`(1), before balance:



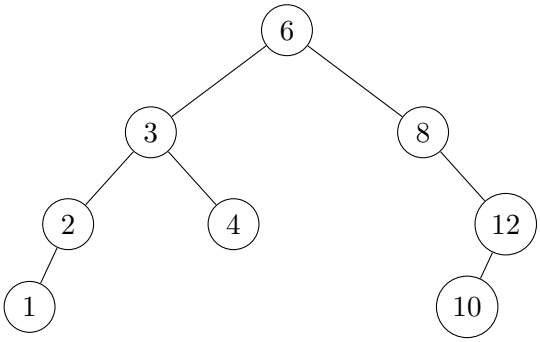
`insert`(1), after balance:



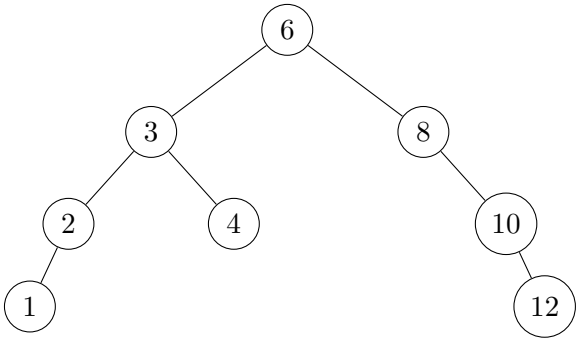
(b) `delete`(5), (no balance required):



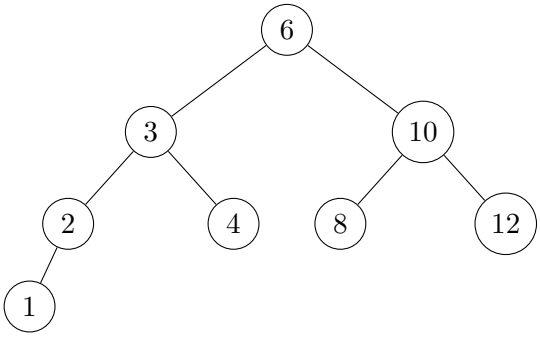
`delete`(7), before balance:



`delete`(7), double rotation part 1:

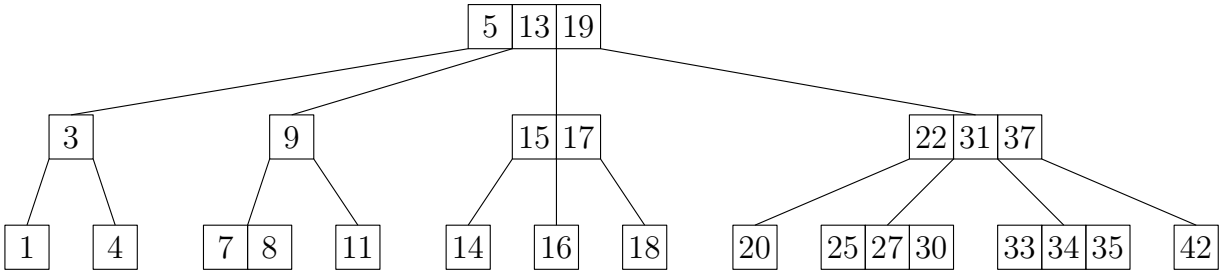


`delete`(7), double rotation part 2:



Exercise 3: (a, b)-Trees

Consider the following $(2, 4)$-tree



(a) In the given tree, perform the operations `insert`(2), `insert`(26) and `insert`(36). Draw the state of the $(2, 4)$-tree after each operation.

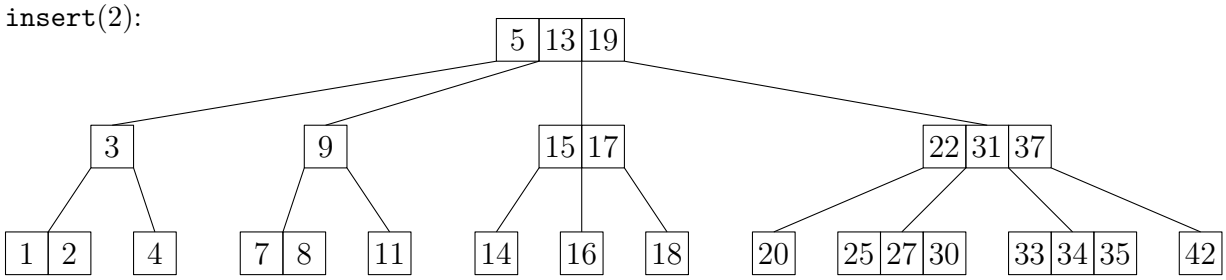
(b) In the resulting tree, perform the operations `delete`(11), `delete`(3). Draw the state of the $(2, 4)$-tree after each operation.

(c) *For exercise lesson:* In the resulting tree from part (b), perform `delete`(13) and draw the state of the $(2, 4)$-tree.

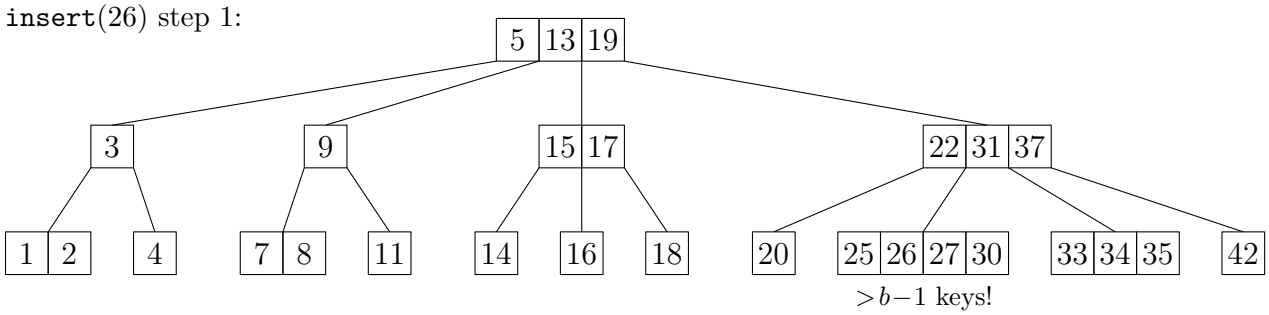
Remark: For comprehensive details on all cases of the `delete` *operation consider, e.g., "Introduction to Algorithms" by Cormen, Leierson, Rivest and Stein.*

Sample Solution

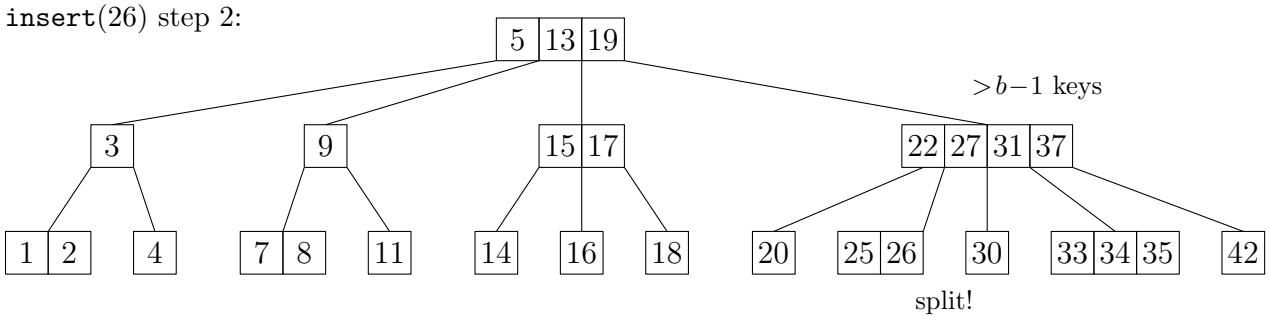
(a) `insert`(2):



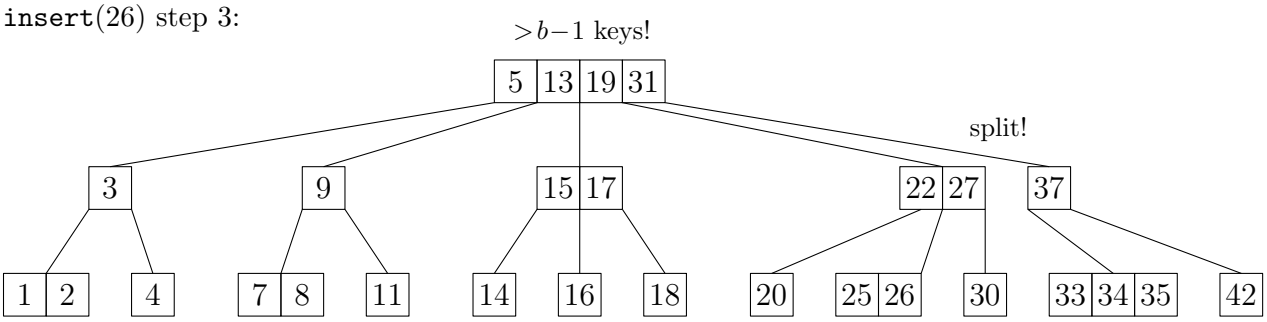
`insert`(26) step 1:



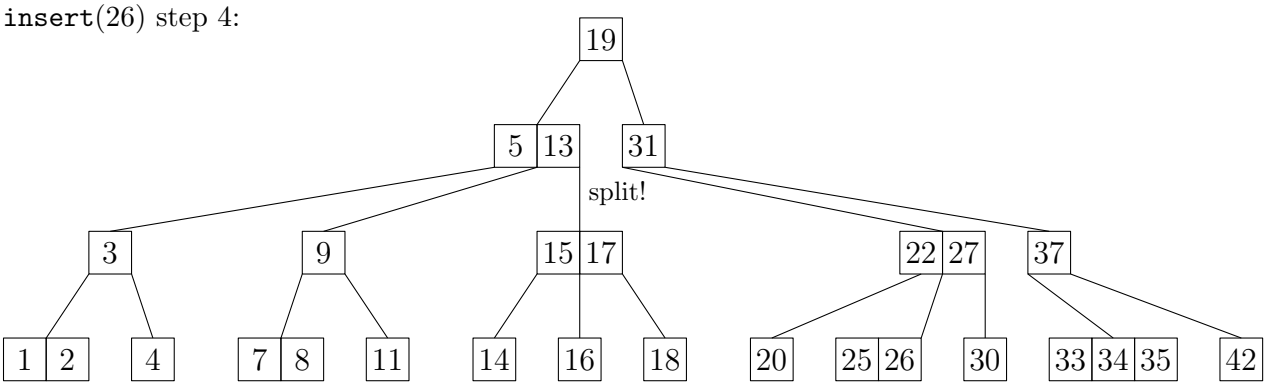
`insert`(26) step 2:



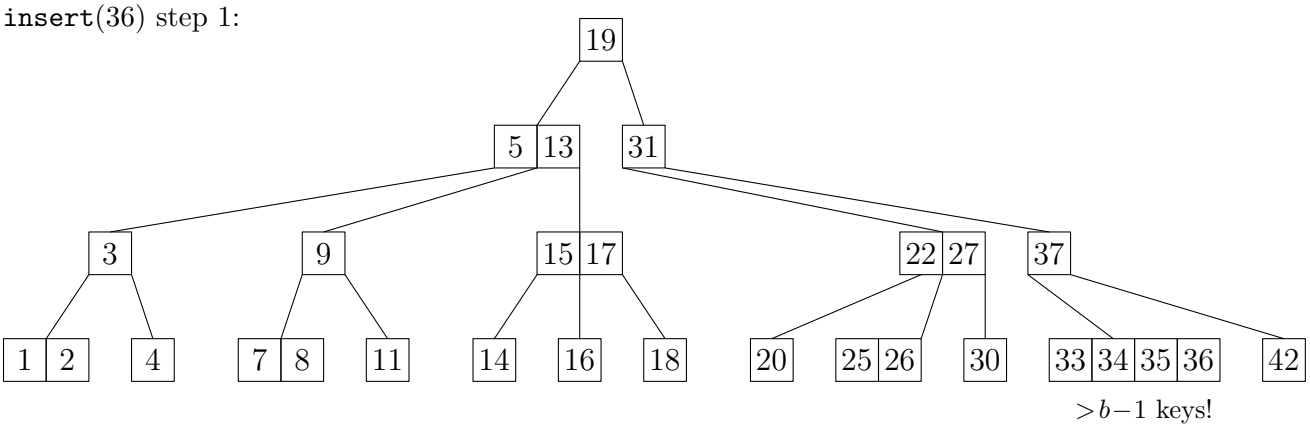
`insert`(26) step 3:



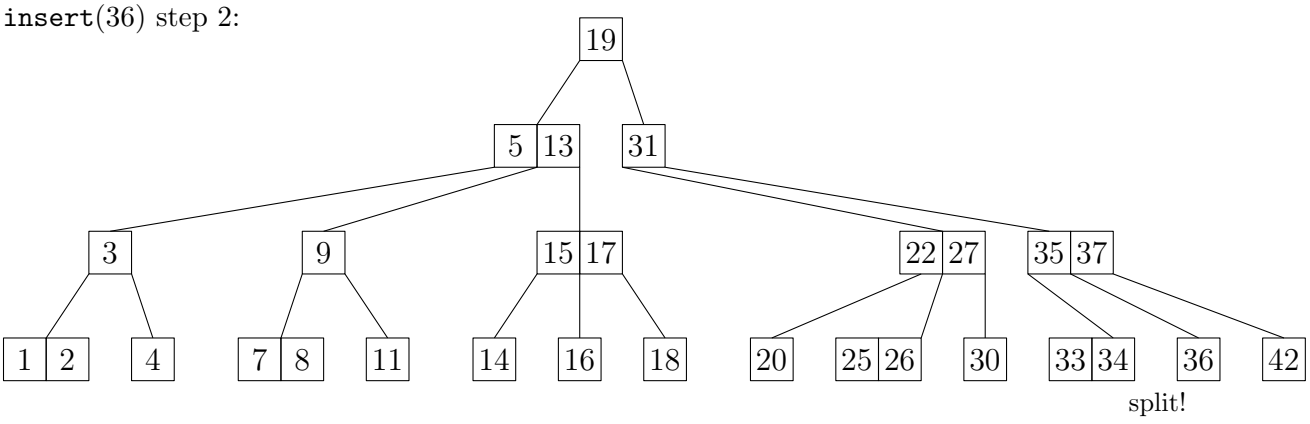
`insert`(26) step 4:



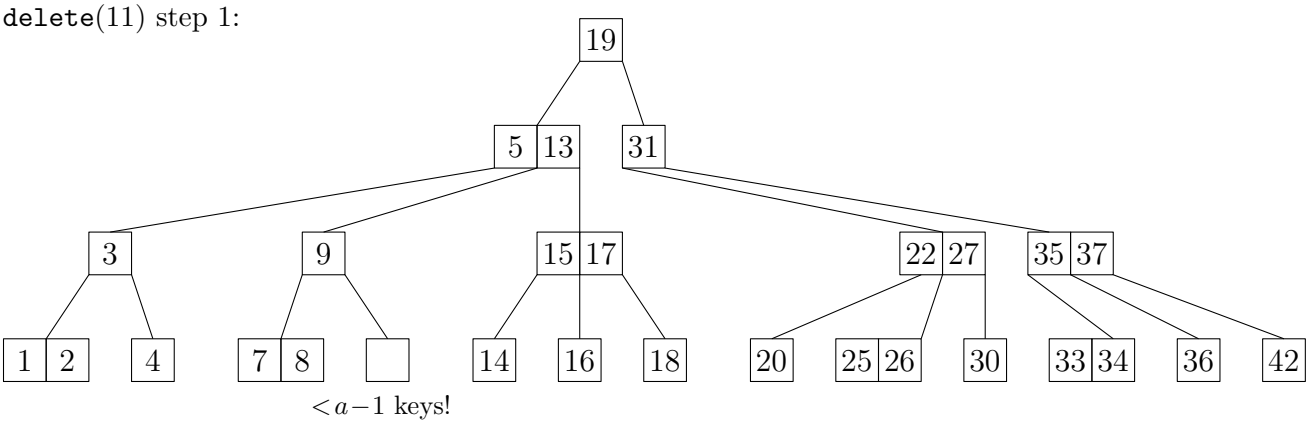
`insert`(36) step 1:



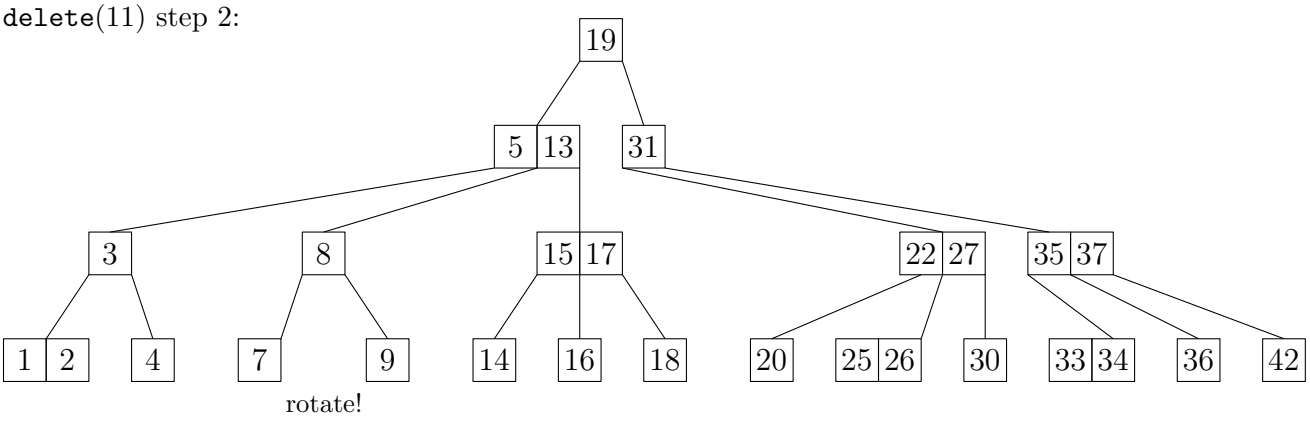
`insert`(36) step 2:



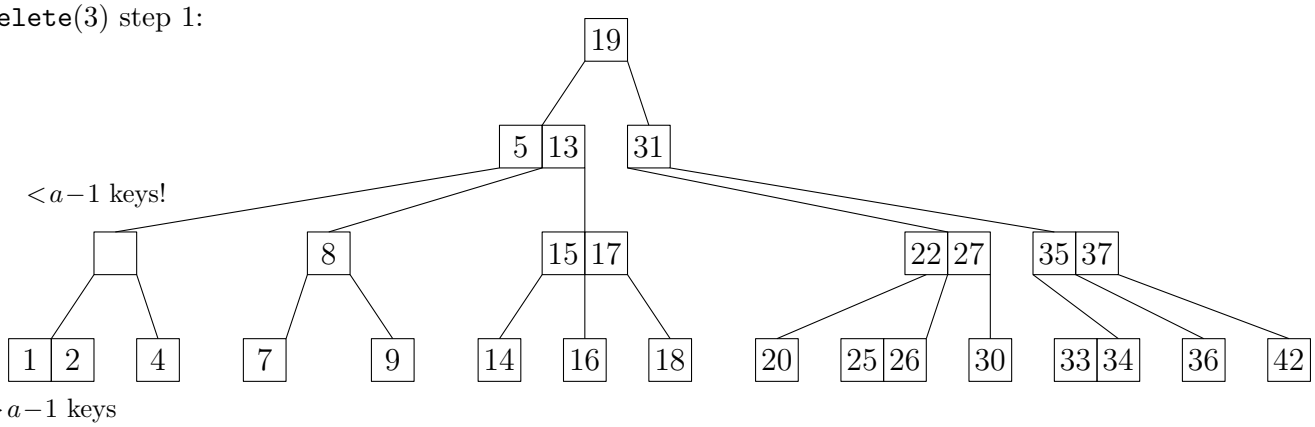
(b) `delete`(11) step 1:



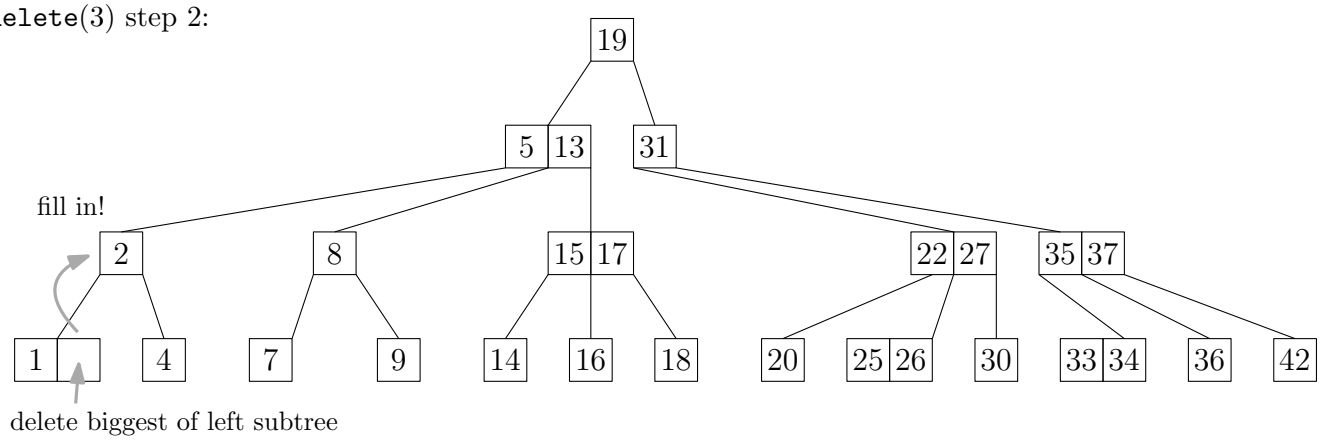
`delete`(11) step 2:



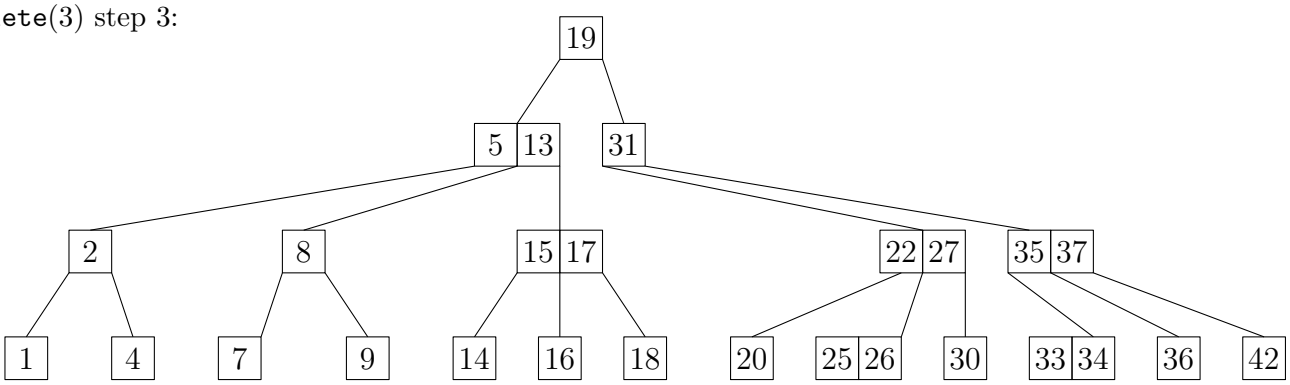
`delete`(3) step 1:



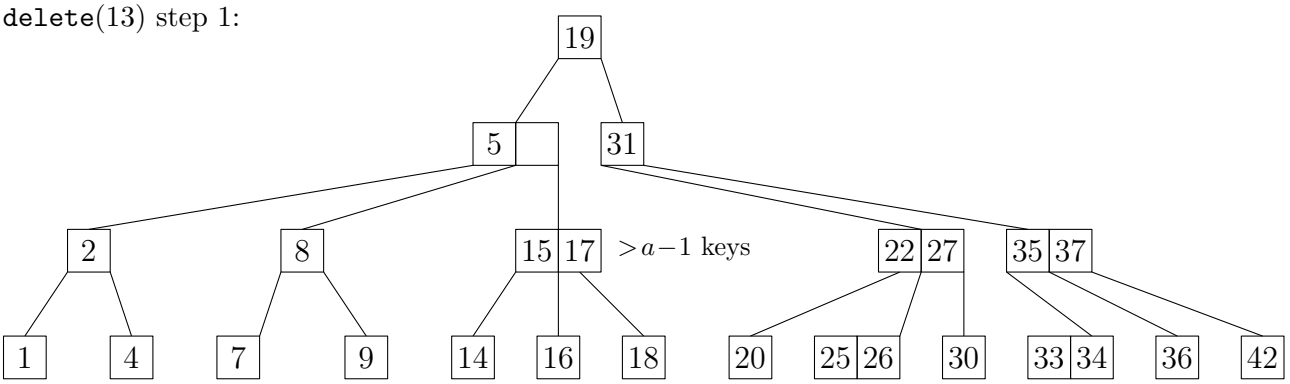
`delete`(3) step 2:



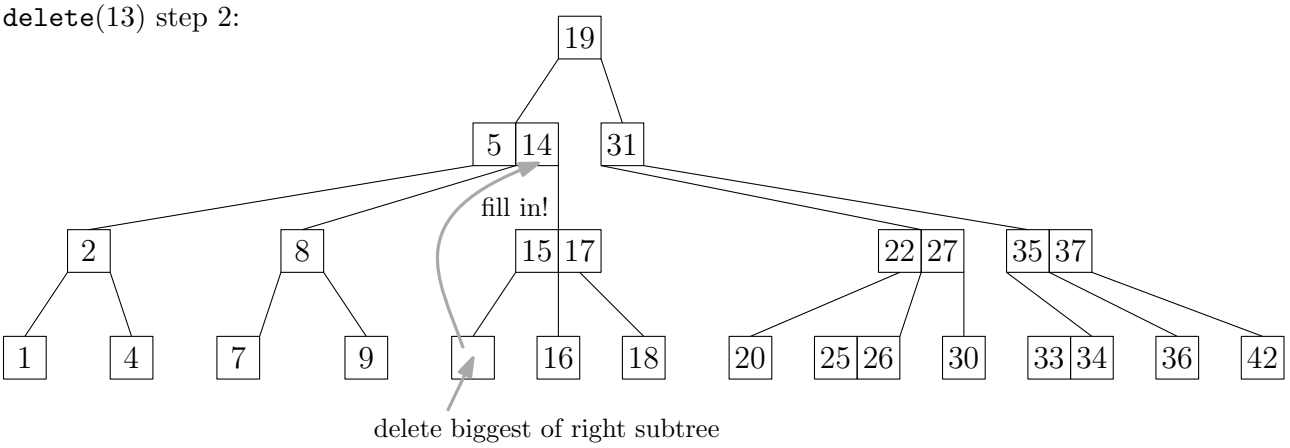
`delete`(3) step 3:



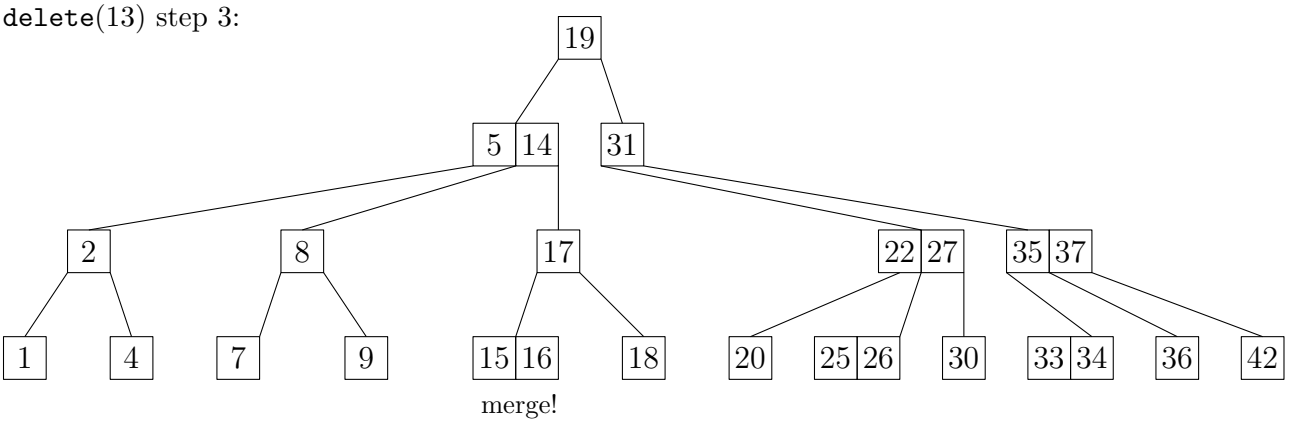
(c) `delete`(13) step 1:



`delete`(13) step 2:



`delete`(13) step 3:



Remark: For more details on all cases of the **`delete`** *operation consider e.g. "Introduction to Algorithms" by Cormen, Leierson, Rivest and Stein.*