FAST SEARCHING / BALANCED TREES

Martin Schobesberger, Markus Weninger,
Markus Jäger, Florian Beck, Achref Rihani

Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

Algorithms and Data Structures 2
Exercise – 2023W

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

RECAP :: AVL TREES :: INSERT

Insert is in general the same as for the binary search tree but may cause the AVL tree to become unbalanced → **restructuring required!**

Restructuring

1. Go up from the new node in the tree until the first node **x** is found, whose grandparent **z** is an unbalanced node
2. Define **y** as child of **z** (= the node we passed on the way to z);
   height(y) = height(sibling(y))+2
3. Define **x** as child of **y**
4. Rename **x**,**y**,**z** in **a**,**b**,**c** (according to Inorder traversal!)
5. Replace **z** (old subroot of unsorted part-tree) by **b** (new subroot of sorted part-tree)
6. Children of **b** are now **a** (left) and **c** (right)
7. Children of **a** and **c** are the subtrees T_0 … T_3, which have been children of **x**, **y** and **z** before → reassign and distinguish **4 cases…**

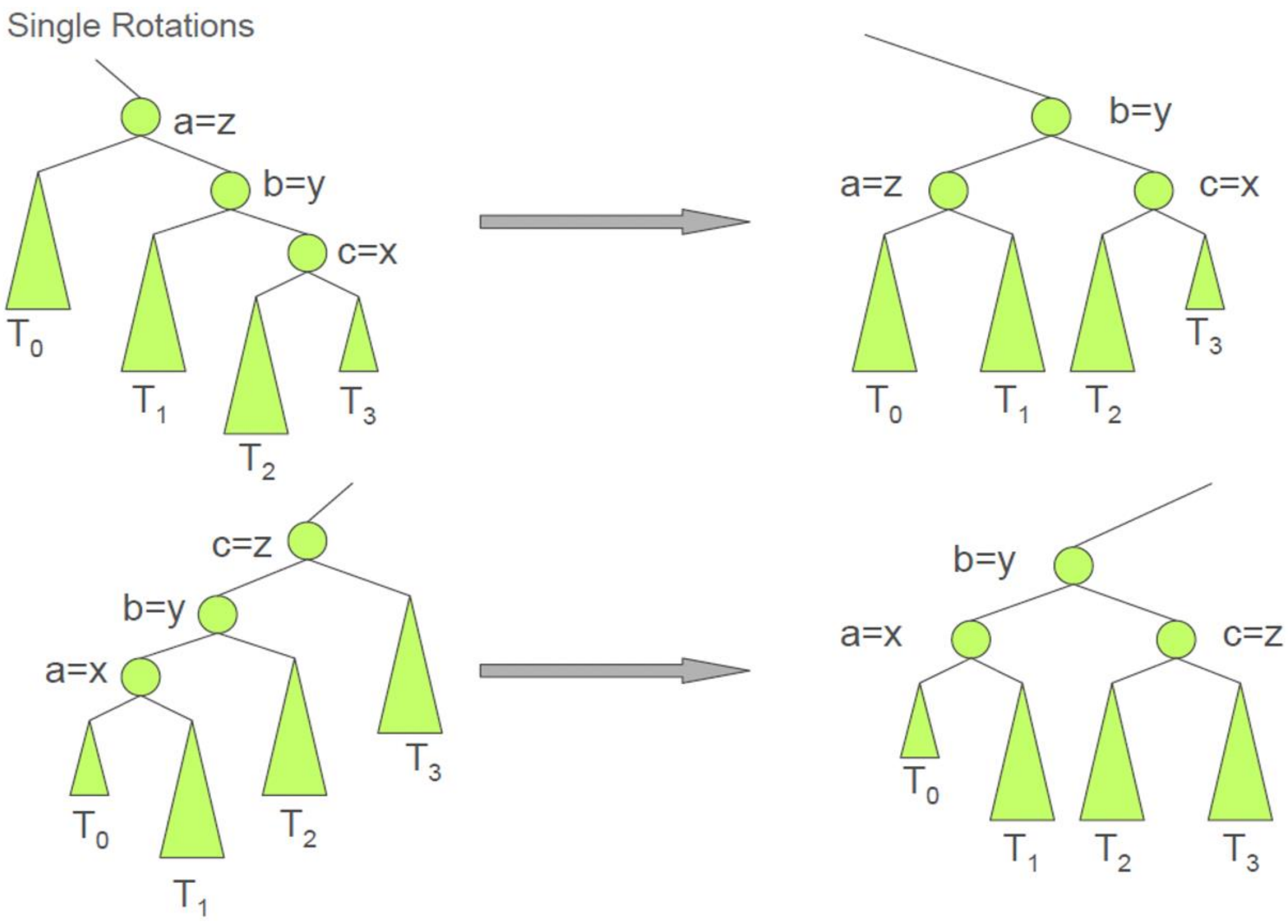
RECAP :: AVL TREES :: REMOVE

- Remove as in binary search tree
- Check the balance
    - starting from the parent node of the removed *Inorder* successor
    - and further parents up to the root if tree is still unbalanced
- **Restructure,** if necessary, until the tree is balanced

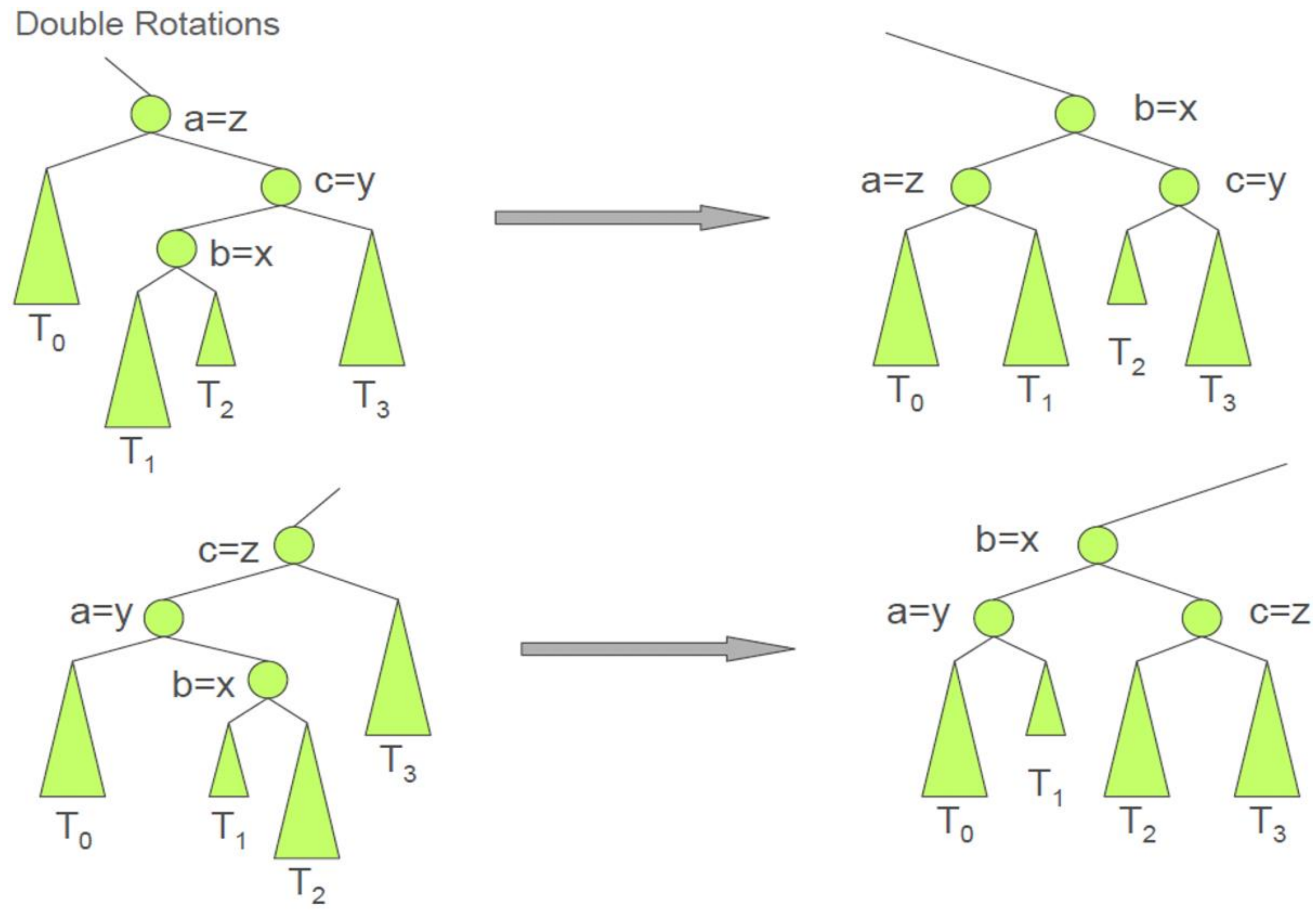
Procedure

1. Search for the first unbalanced node **z**
2. Put **y** on child of **z** with greatest height
3. Put **x** on child of **y** with greatest height

RECAP :: AVL TREE :: ROTATIONS



RECAP :: AVL TREE :: ROTATIONS



AVL TREE :: RESTRUCTURING APPROACH

Procedure

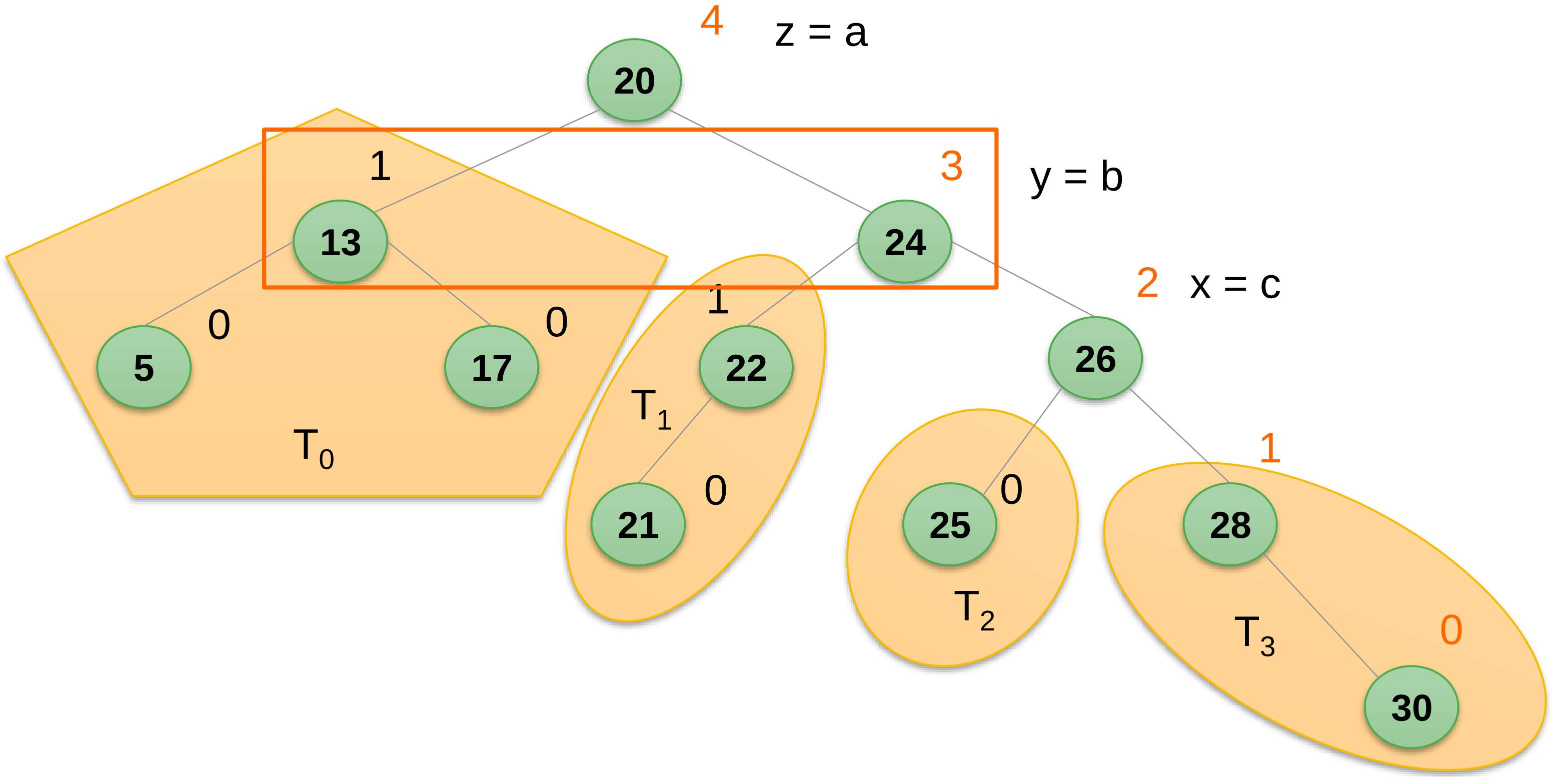
1. Find x, y, z

2. Create an auxiliary object to store a, b, c, T0, T1, T2, T3

3. Fill the auxiliary object according to in-order traversal. They are always encountered in the order: T0, a, T1, b, T2, c, T3.

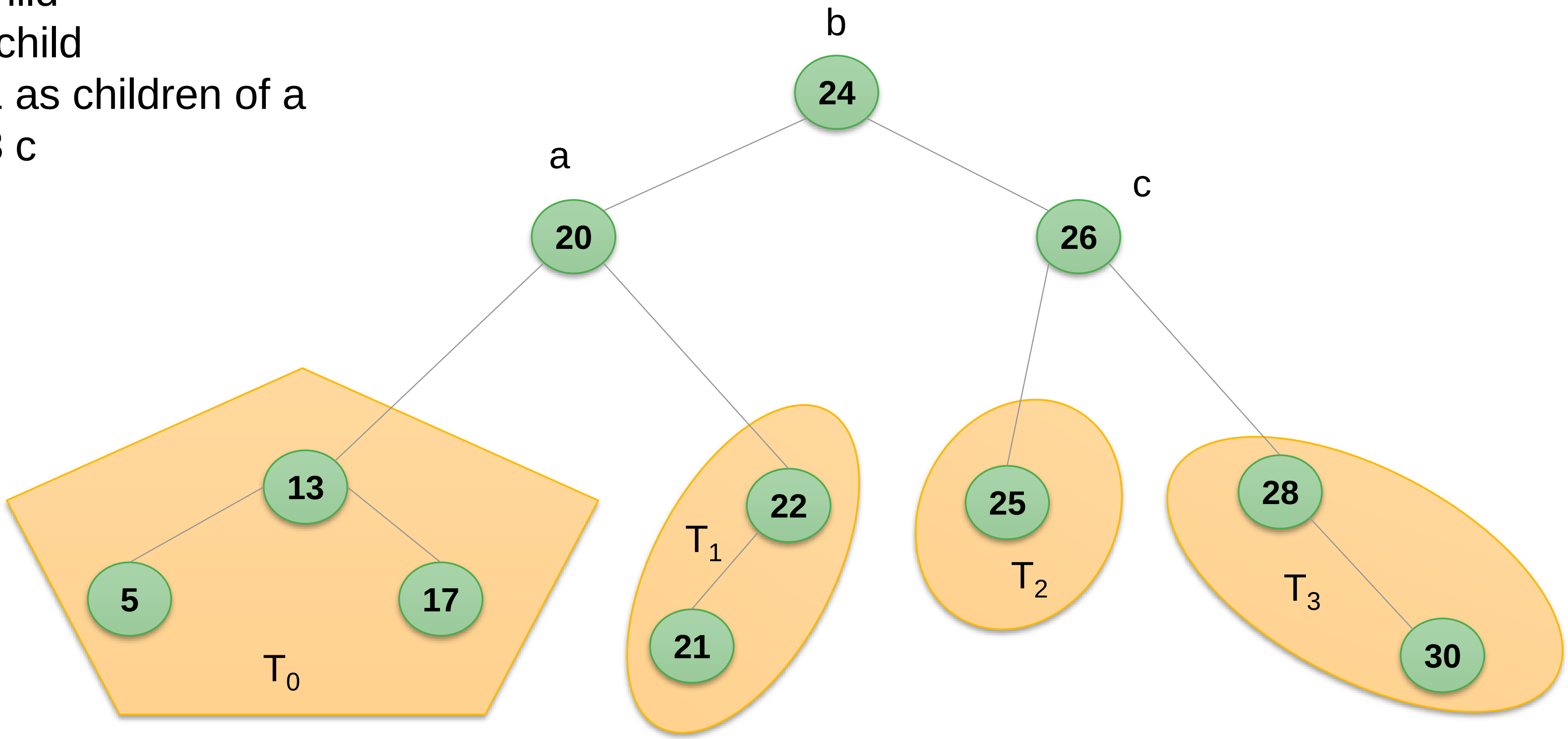
4. Restructure: Set the element b as root, a and c as left and right child of b, and finally T0 and T1 as children of 1 a, and T2 and T3 as children of c.

AVL TREE :: FIND COMPONENTS

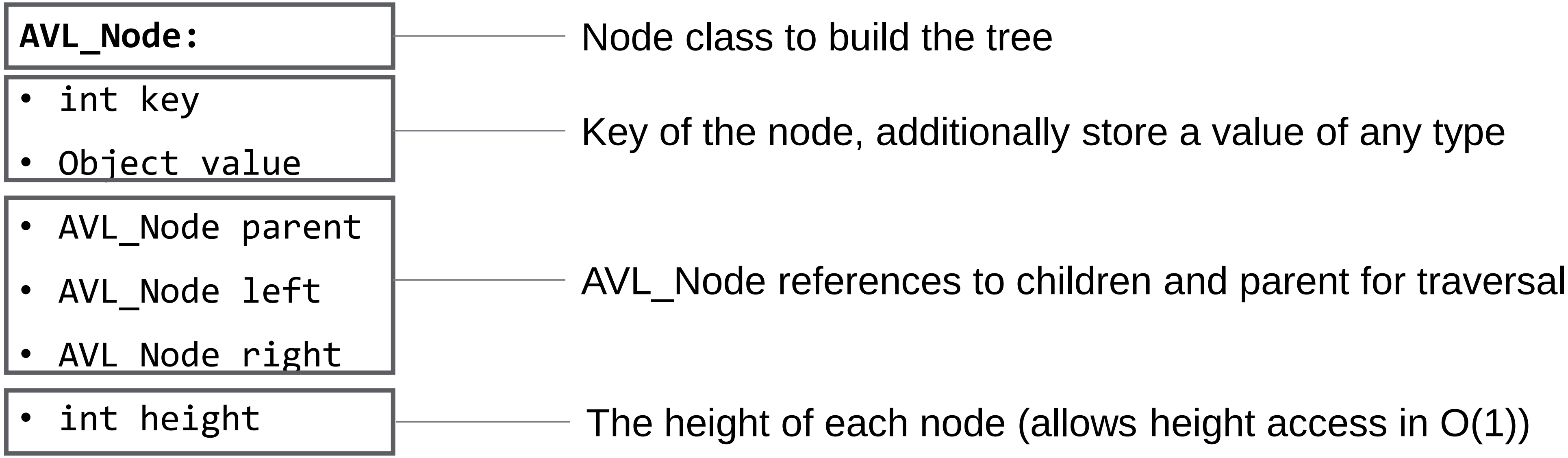


AVL TREE :: RESTRUCTURE

Element b as root,
a as left child
c as right child
T0 and T1 as children of a
T2 and T3 c



AVL TREE :: IMPLEMENTATION



AVL_Node:

Node class to build the tree

- int key
- Object value

Key of the node, additionally store a value of any type

- AVL_Node parent
- AVL_Node left
- AVL_Node right

AVL_Node references to children and parent for traversal

- int height

The height of each node (allows height access in O(1))

AVL TREE :: IMPLEMENTATION

```
insert(key, value):
    n = AVL_Node(key, value)
    BST_insert(n)
    update_heights(n)
    x = n
    if not is_balanced(x.grandparent):
        z = x.grandparent, y = x.parent
    restructure(x, y, z)
    else:
        move up in the tree and check again
```

Insert function, takes a key and value as parameters

Create a new AVL_Node to insert into the tree

Insert the new node in the AVL tree just like in a BST (binary search tree)

Update the heights in the tree starting with the newly inserted node

Check if new node corresponds to “first node x whose grandparent is unbalanced” → if yes start rotation

Otherwise move up in the tree and check again for x, y, z

AVL TREE :: IMPLEMENTATION

```
update_heights(node):
    current = node
    set height of current node based on height of its children
    move to parent of current node and repeat (recursion)
```

Function to update the height of a node

```
is_balanced(node):
    check height difference of node.left and node.right
    return true/false accordingly
```

Function to check the balance of a given node

AVL TREE :: IMPLEMENTATION

```
remove(key):
    BST_remove(key)
    update_heights(parent of inorder successor)
    z = find first unbalanced node, starting from parent of inorder successor
        if found:
            y = z.child with greatest height
            x = y.child with greatest height
            restructure(x, y, z)
            move up in tree towards root and repeat if needed
```

Function to remove a node based on a given key

AVL TREE :: IMPLEMENTATION

```
restructure(x, y, z):
    get_components(x, y, z)
    relink components accordingly
```

Rebalance the tree using rotations / cut-and-link

```
get_components(x, y, z):
    identify a, b, c
    identify T0...T3
```

Find the nodes a, b, c based on one of the four possible cases

Find subtrees T0...T3 based on one of the four possible cases

ASSIGNMENT 02

Coding session

JOHANNES KEPLER
UNIVERSITY LINZ

FAST SEARCHING / BALANCED TREES

Algorithms and Data Structures 2
Exercise – 2023W