Institute of Pervasive Computing

Trees (Weight Balanced)

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

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

Balanced Trees

Two different approaches

- **Height-balanced trees**:

  The maximum allowed **height difference** of the two subtrees is limited

- **Weight-balanced trees**:

  The **ratio** of the **node weights (number of nodes)** of both **subtrees** meets certain conditions.

Balanced Trees

Balanced trees are introduced as a compromise between balanced and natural search trees, whereby **logarithmic search complexity** is required in the **worst case**.

For the **height** h_b of an AVL tree with N nodes we have:

$$\lfloor \log_2 N \rfloor \leq h_b \leq 1{,}44 * \log_2 (N+2)$$

- The upper limit can be derived from Fibonacci trees, a subclass of the AVL trees.
- Let N(h) be the **minimum number** of **nodes** of a **height-balanced** tree with **height h**. We have:
  - N(0)=**1**, N(1)=**2**, N(2)=**4**, N(3)=**7**, N(4)=**12**, N(5)=**20**, …
  - N(h) = **1 + N(h-1) + N(h-2)** = Fib(h+3) – 1
  - $\mathrm{Fib}(h) = 1/\sqrt{5} * (((1 + \sqrt{5})/2)^h - ((1 - \sqrt{5})/2)^h)$
  - for all h we have: $\mathrm{Fib}(h) \geq 1/\sqrt{5} * ((1 + \sqrt{5})/2)^h - 1$
  - if N(h)=Fib(h+3)-1 we have: $\log_2 (N(h)+2) \geq \log_2 (1/\sqrt{5}) + (h+3) \log_2 ((1 + \sqrt{5})/2))$
- From this the estimation follows: $h \leq 1{,}44 \log_2 (N(h)+2)$ → h = **O (log N(h))**

0,1,1,2,3,5,8,13,21, …

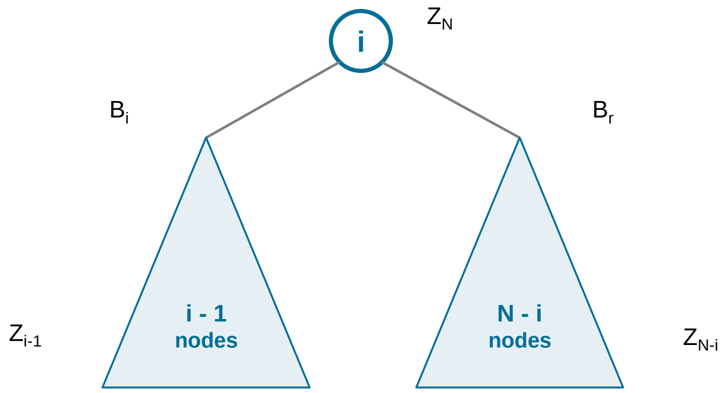
Minimum number of nodes **grows exponential** with **height**

→ so vice versa: **height grows logarithmically** with **node number**

Weight-Balanced Search Trees

Weight-balanced or BB trees (bounded balance):

Tolerated deviation of the structure from balanced binary tree is defined as the **difference** between the **number of nodes** in the **right** and **left subtree**.



Definition:

Let B be a binary search tree with left subtree B_l and l be the number of nodes in B_l

(let N be the corresponding number of nodes in B)

- ρ (**B**) = (l + 1) / (N + 1) is the **root balance** of B.
- A tree B is weight-balanced (BB(α)) or of **limited balance** α, if **for each subtree** B' of B we have: $\alpha \leq \rho\ (\mathbf{B'}) \leq 1 - \alpha$

Weight-Balanced Search Trees

Parameter α as degree of freedom in the tree

- **α = 1/2** : Balancing criterion only accepts complete binary trees
- **α < 1/2** : Structural restriction is increasingly relaxed

What effects does the relaxation of the balancing criterion have on costs?

Rebalancing

- Use of the same rotation types as for the AVL tree
- is guaranteed by the choice of $\alpha \leq 1 - \sqrt{2} / 2$

Search and update costs: **$O(\log_2 N)$**

Multipath Search Trees

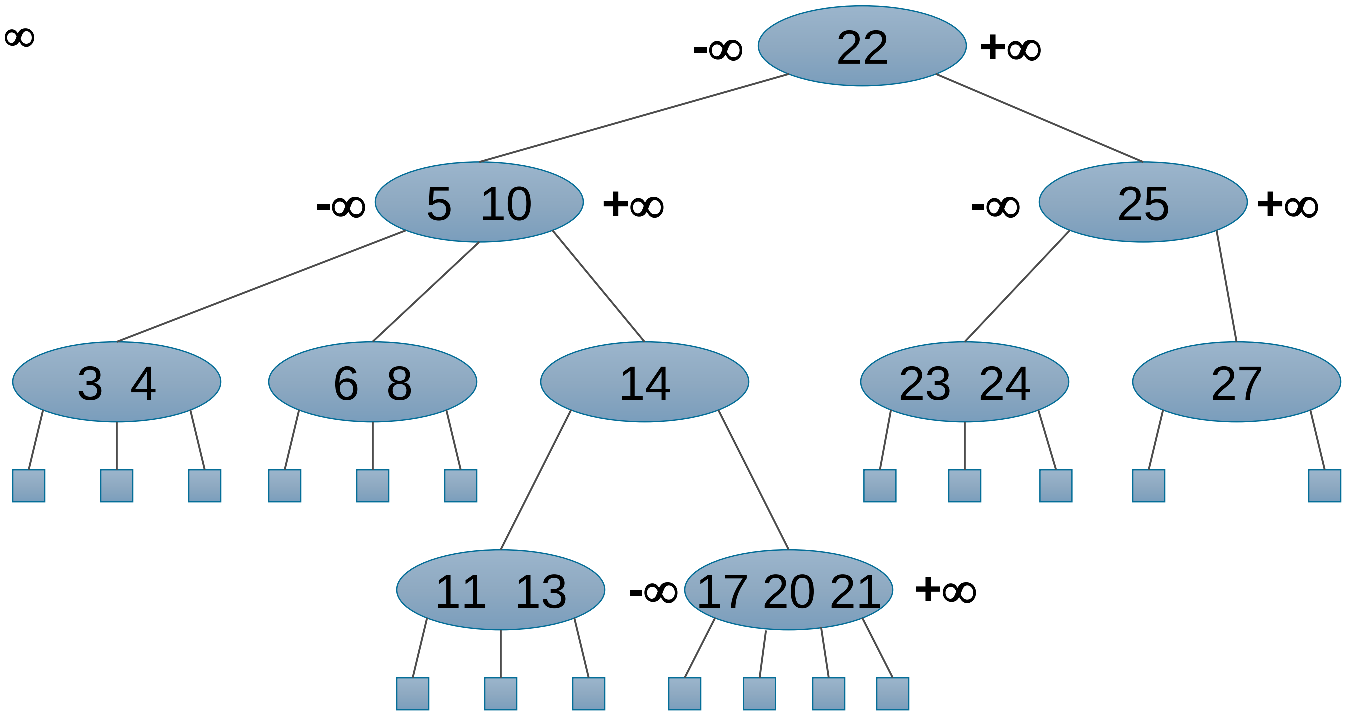
Each internal node of a **multipath search tree**

- has **d children**, where d ≥ 2
- Stores a **set of records(k, x)** where k is the key and x the element
- **Number of records: d-1**
- Additionally: **2 pseudo records: k_0 = -∞ and k_d = ∞**

For all children of an internal node we have:

- **Keys** of the **children** lie **between** the **keys** of the **respective records**

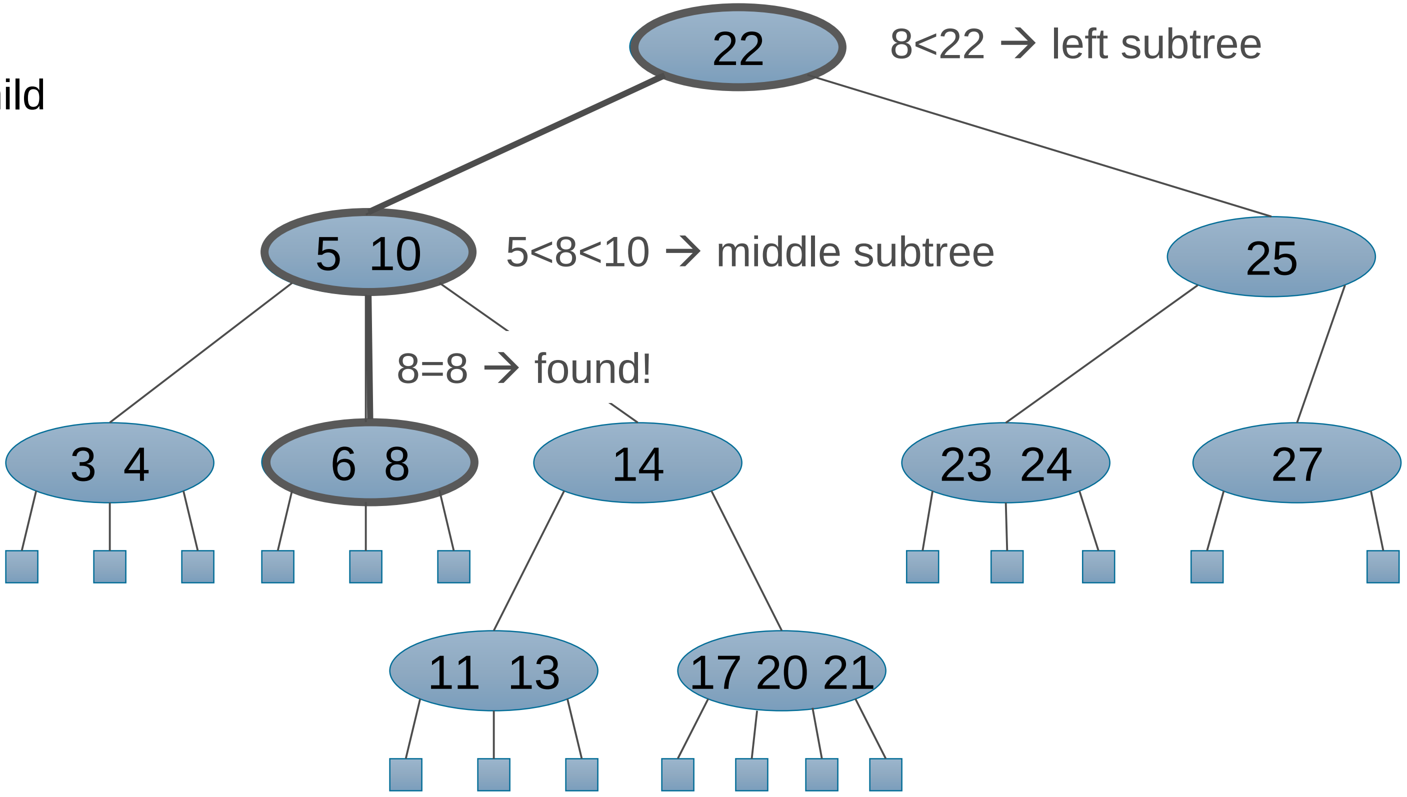
External nodes are placeholders



Multipath Search

Similar to binary search

- If $k < k_1$ **search in the leftmost** child
- If $k > k_{d-1}$ **search in the rightmost** child
- If d>2:
  find key k_{i-1} and k_i for which:
  $k_{i-1} < k < k_i$ and continue
  search in child v_i



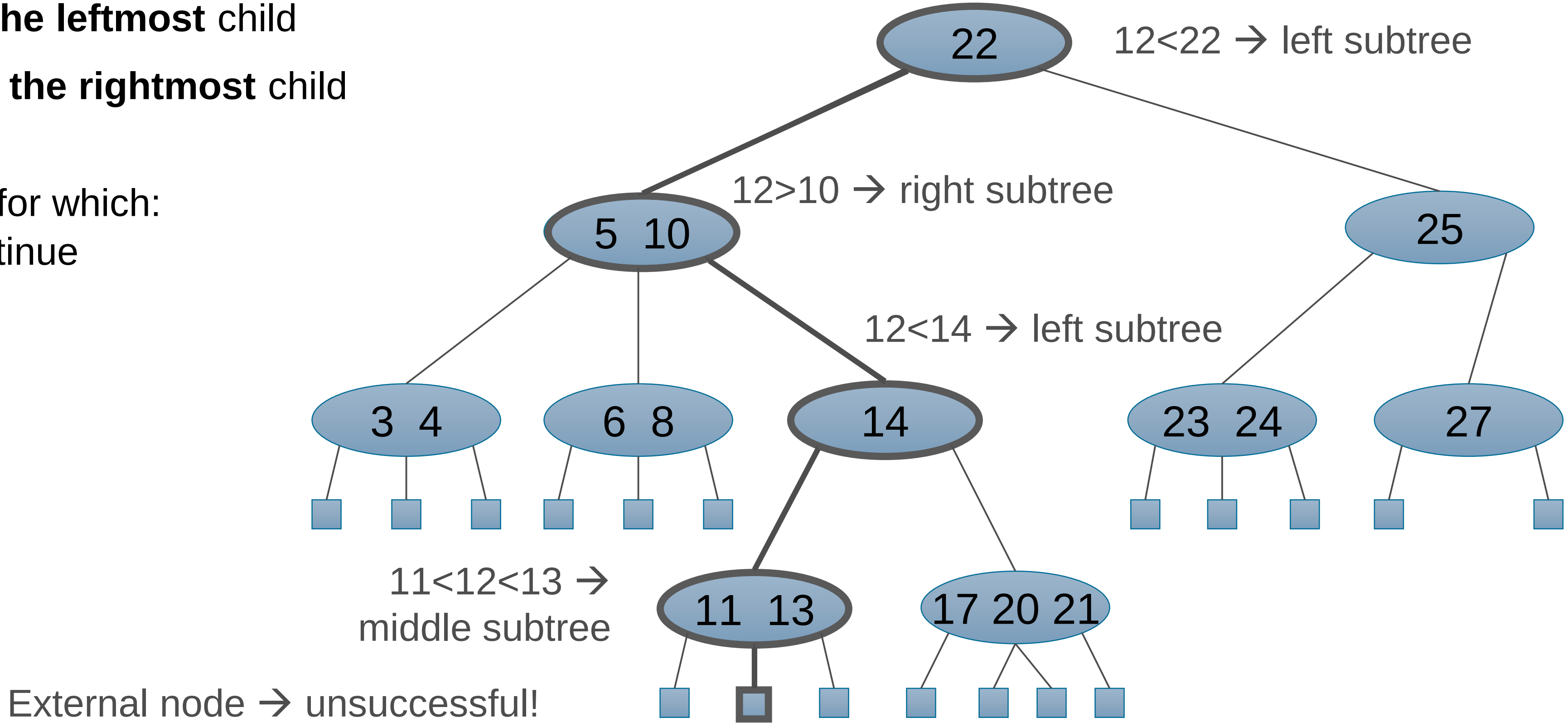
Example: Search 8

Multipath Search

Similar to binary search

- If $k < k_1$ **search in the leftmost** child
- If $k > k_{d-1}$ **search in the rightmost** child
- If d>2:
  find key k_{i-1} and k_i for which:
  $k_{i-1} < k < k_i$ and continue
  search in child v_i

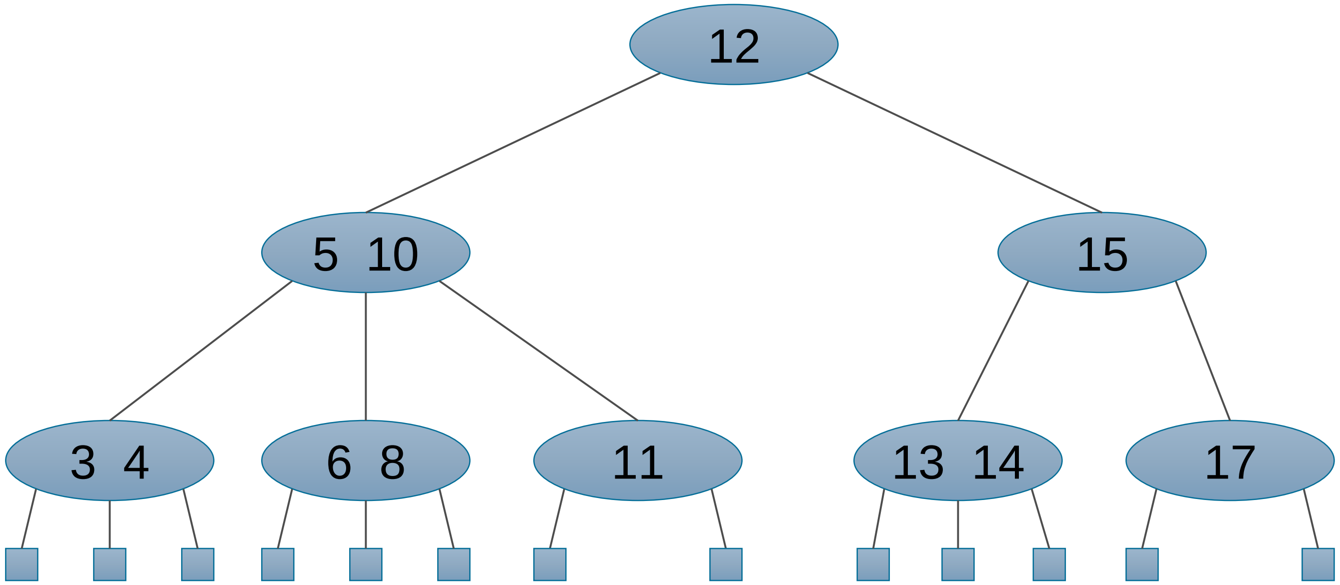
Example: Search 12



(2,4) Trees

Special case of a multipath tree:

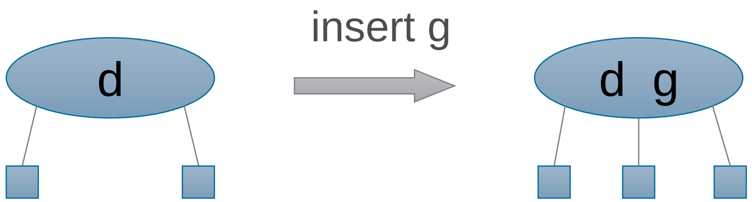
- Each node has a **minimum** of **2,** and a **maximum** of **4** children
- All **external** nodes have the **same depth**
- **Height** of the tree is **O(log N)**



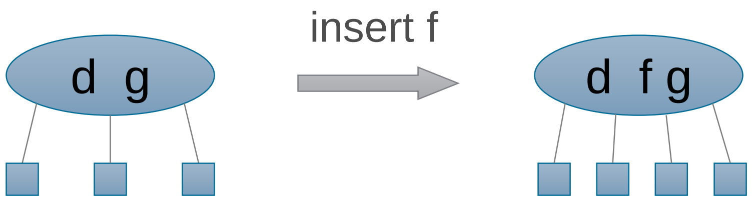
Insert in (2,4) Trees

Insert key in lowest internal node, that has been reached during search.

Case 1:
Node has 1 record

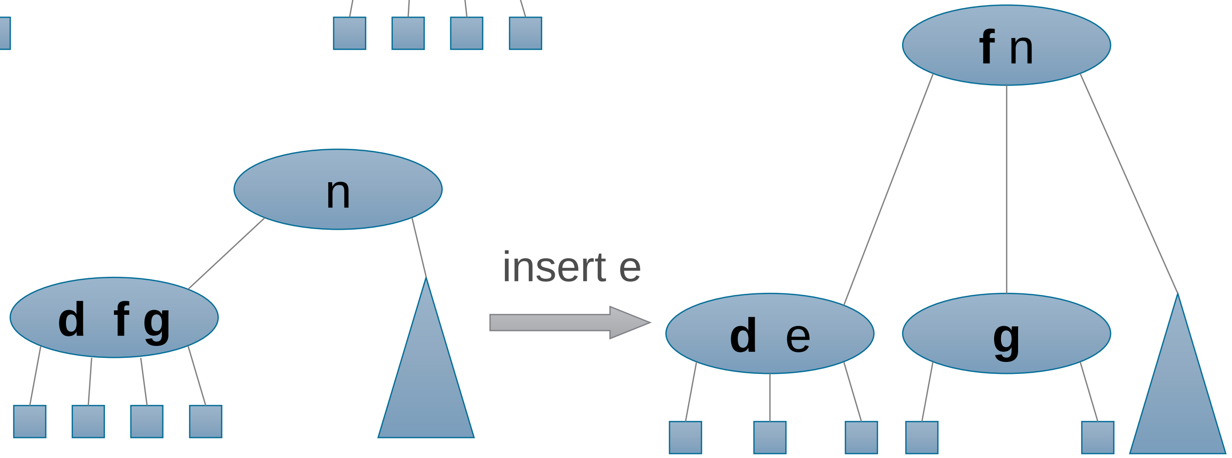


Case 2:
Node has 2 records



Case 3:
Node has already 3 records

- Node splitting: Split node into two 1-element nodes and move middle element into parent node



Insert in (2,4) Trees

Top-Down Insertion

- Starting with the root, **node-splitting** is done for **each node with three** elements, that is visited on the way when searching for the insertion position
- This ensures that inserting can be done according to case 1 or 2.

split concerns
constant number of nodes → O(1)

Bottom-Up Insertion

- Searcjh for insertion position
- If the node at the insertion position has already 3 elements, **node-splitting** is done
- If this results in an "**overflow**" in the **parent** node (by moving the middle element upwards), **node-splitting** is done again.

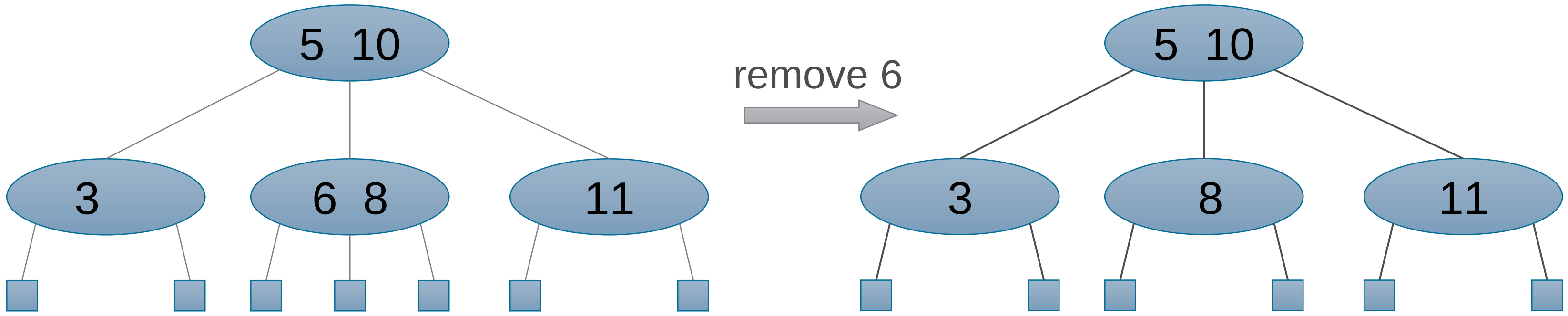
the maximum number of levels affected is
log N → O(log N)

Removal in (2,4) Trees

Principle

- **Find** the record to be deleted via key
- **Remove** the entry and **merge** (inverse operation to split), **if** node has **too few entries**

Example



The following special cases must be considered in detail:

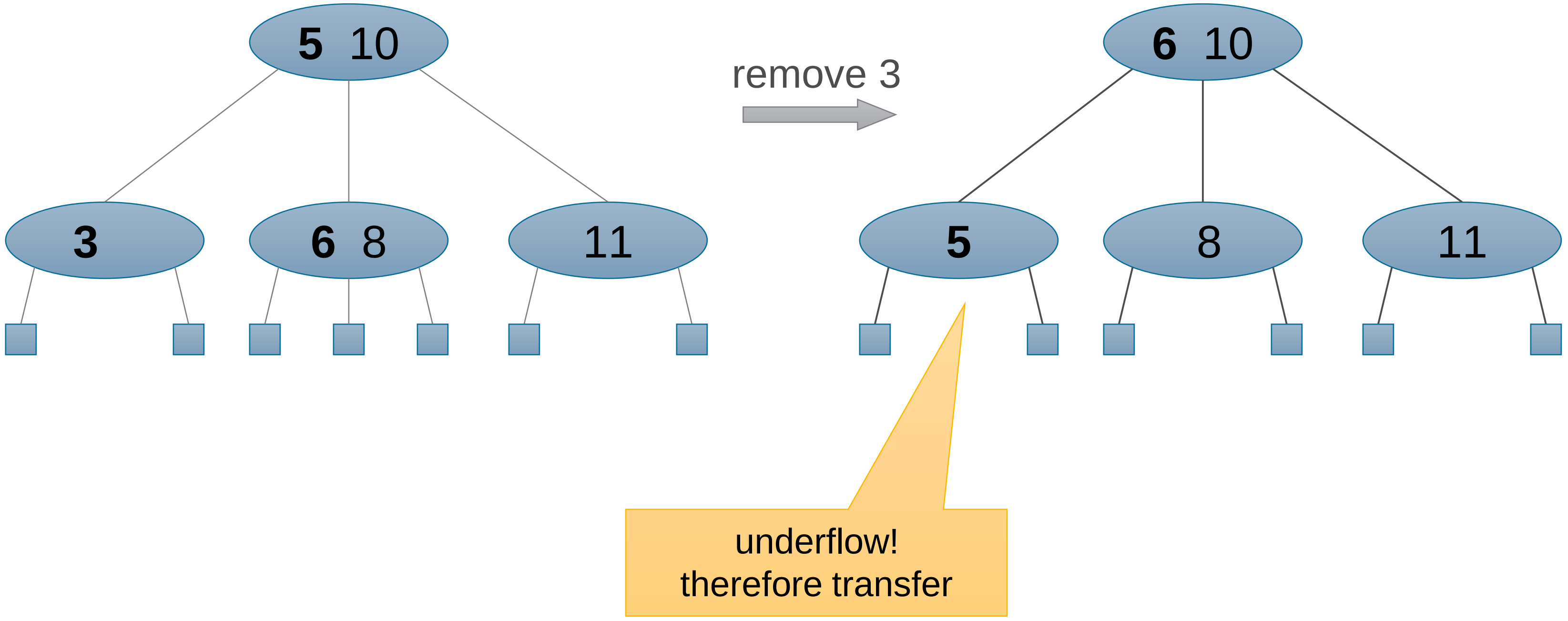
- If node has internal nodes among its children: **reduce** this case to the case where node has **only external nodes** as successors:
  - Search for previous entry according to **in-order traversing**
  - **Swap** entry with this predecessor
  - is **repeated until** the entry is at the **lowest level** of the tree.

Removal in (2,4) Trees

The entry to be deleted is the **last entry** in the node:

- **Get** entry **from parent** in this node
- Replace "**gap**" in **parent** node with entry from **sibling** (**transfer**)

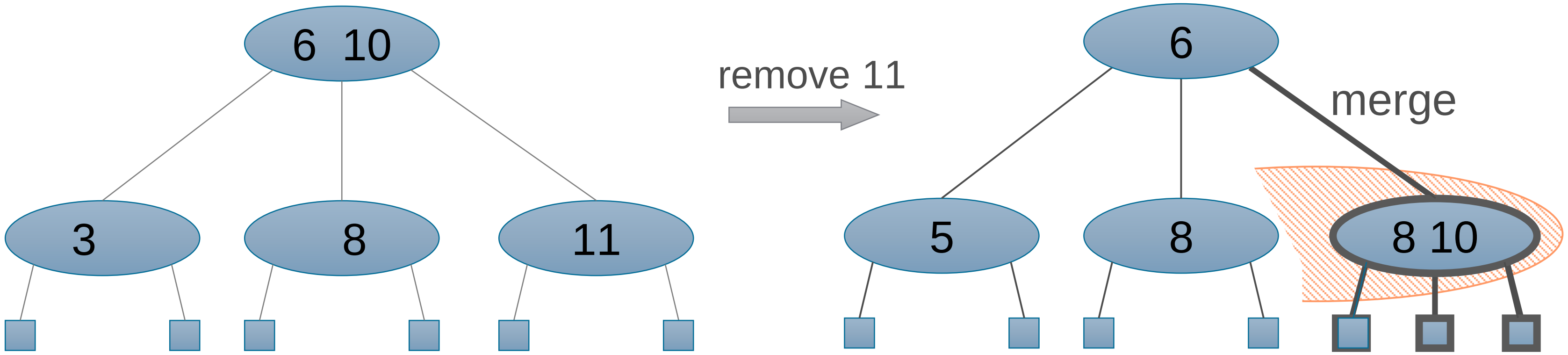
Example: Remove 3



Removal in (2,4) Trees

All siblings have **only one records**:

- Get entry from parent node
- "**Merge**" 2 siblings

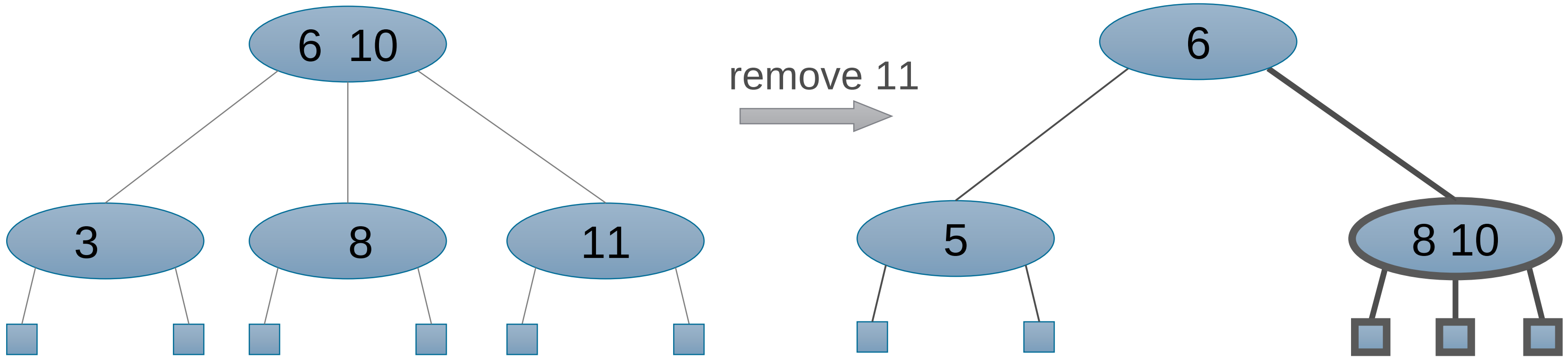


Example: Remove 11

Removal in (2,4) Trees

All siblings have **only one records**:

- Get entry from parent node
- “**Merge**“ 2 siblings



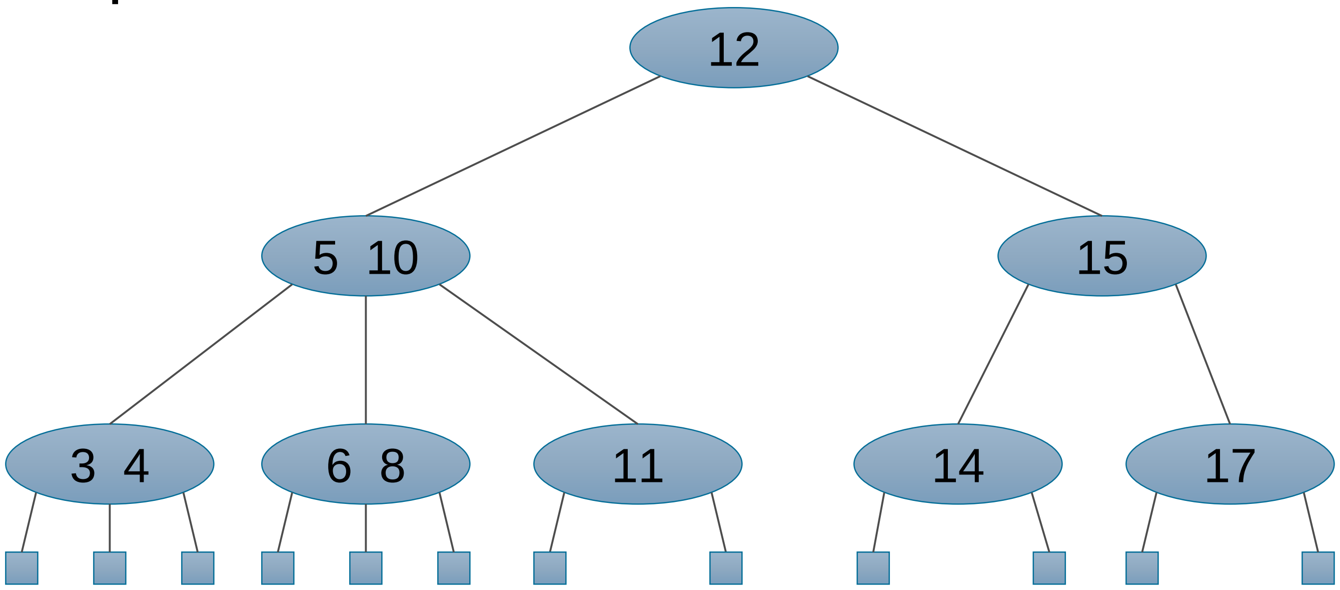
Example: Remove 11

Removal in (2,4) Trees

Parent has **only one record**:

- **Merging propagates upwards**

Example: Remove 14

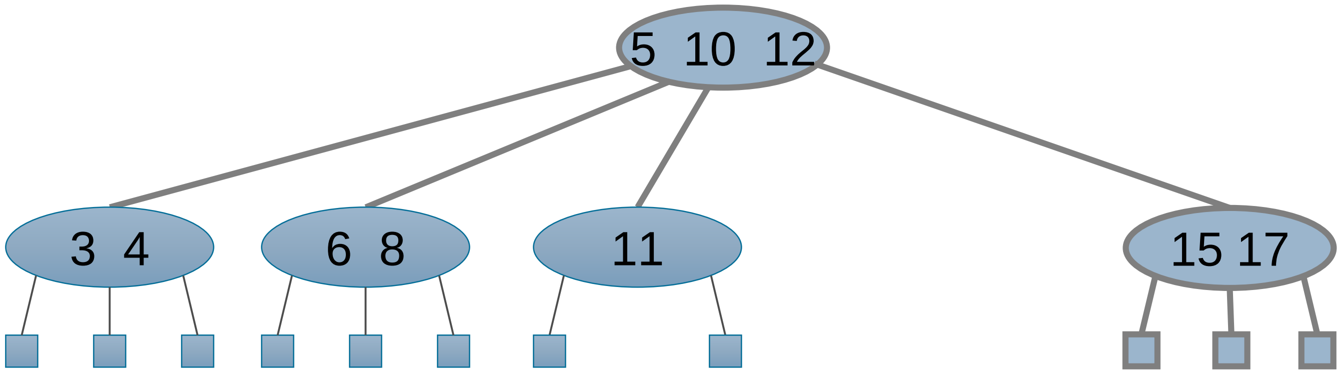


Removal in (2,4) Trees

Parent has **only one record**:

- **Merging propagates upwards**

Example: Rem



(2,4) Trees :: Complexity

Splitting, Transfer, Merge: **O(1)**

Search

- **Runtime** corresponds to **height** of the **tree**, therefore **O(log N)**

Insert

- During search O(log N) nodes are visited
- **Inserting requires (at maximum) O(log N) node-splitting** operations
- A node-splitting operation can be done in constant time, O(1)
- Therefore the overall complexity for insert is: **O(log N)**

Remove

- During search O(log N) nodes are visited
- **Remove requires at maximum O(log N) update** operations (Transfer, Merge)
- Therefore the overall complexity for remove is: **O(log N)**

From (2,4) Trees to Red-Black Trees

Properties of (2,4) trees:

- balanced
- **Search, Insert, Remove: O(log N)**
- but: structure!!

From (2,4) Trees to Red-Black Trees

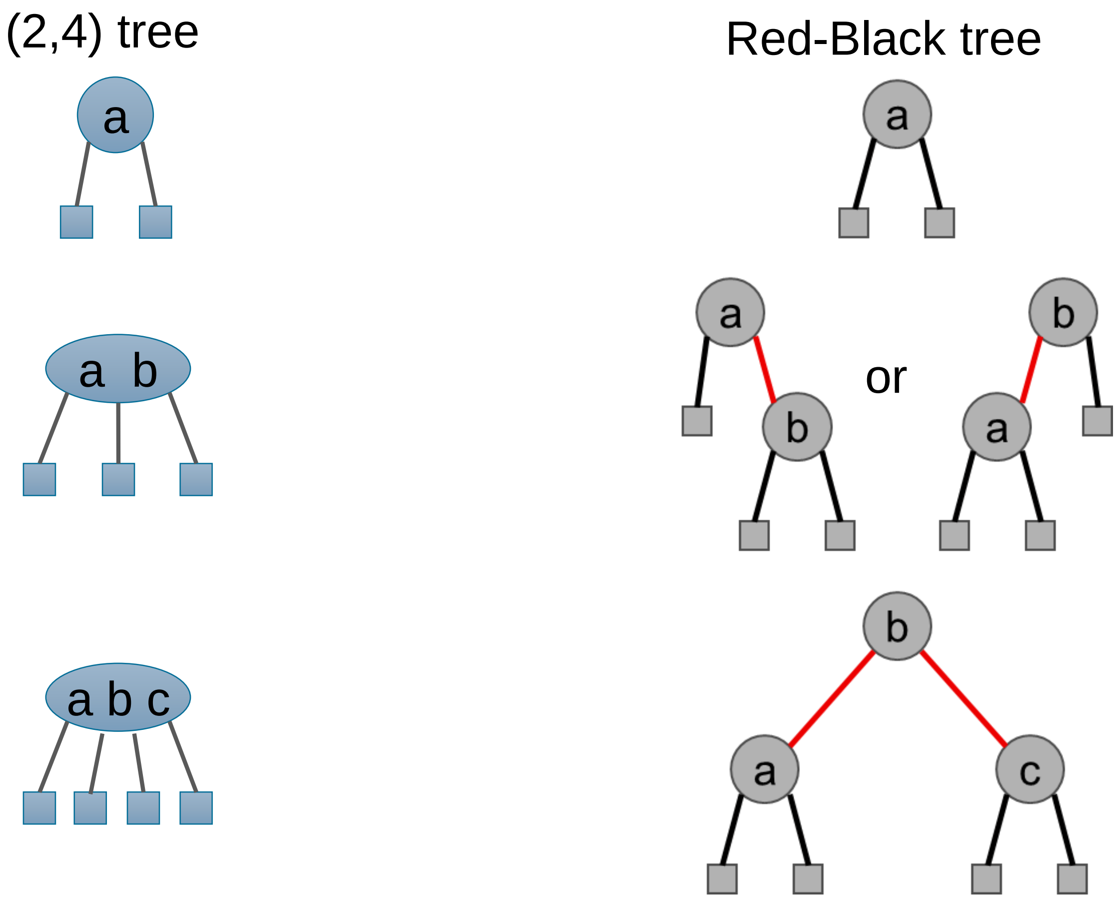
Properties of (2,4) trees:

- balanced
- **Search, Insert, Remove: O(log N)**
- but: structure!!

Red-Black trees

- balanced
- **Search, Insert, Remove: O(log N)**
- Binary tree structure!

Red-black Trees

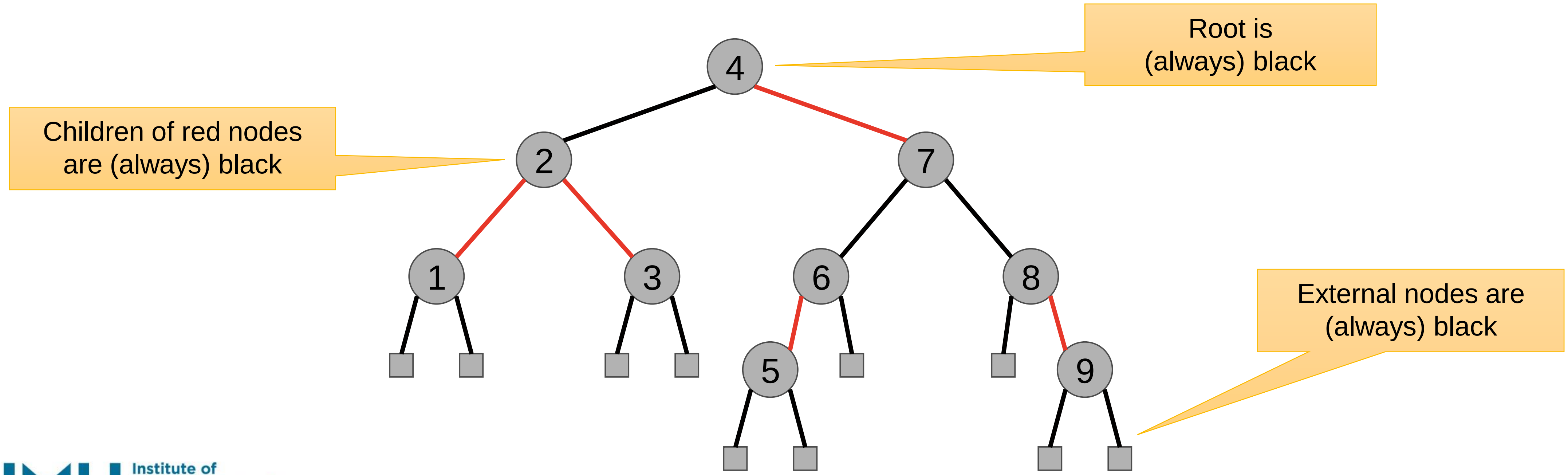


Red-black trees are another way of representing (2,4) trees.

Red-black Trees

A red-black tree is a **binary search tree** with the following properties

- Edges are colored **red** or **black**
- In **no** path from root to leaf there are **two consecutive red** edges
- The number of **black edges** is the same for each path from the root to a leaf ("**black height**")
- **Edges** that lead to **leaves** are **always black**.



Properties of red-black Trees

Let

N be the number of **internal nodes**

L be the number of **leaves** (L = N+1)

H be the **height**

B be the **Black Height** (height according to black edges)

Property 1:

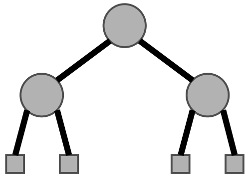
$$2^B \leq N+1 \leq 4^B$$

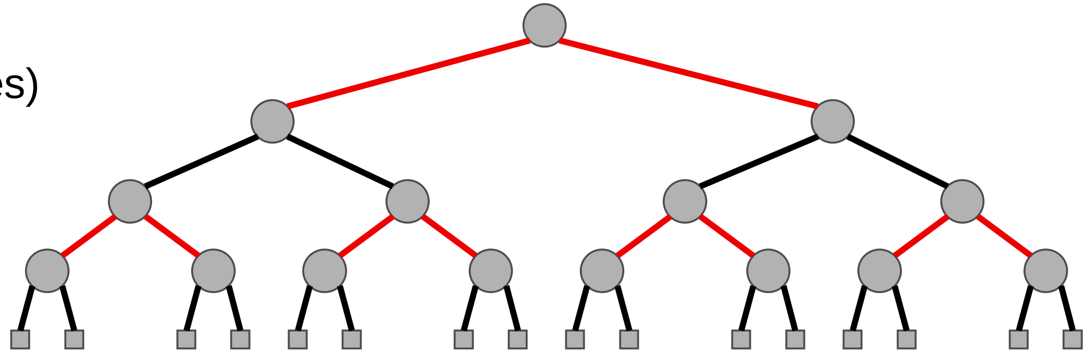
Property 2:

$$1/2 \log_2 (N+1) \leq B \leq \log_2 (N+1)$$

Property 3:

$$\log_2 (N+1) \leq H \leq 2 \log_2 (N+1)$$



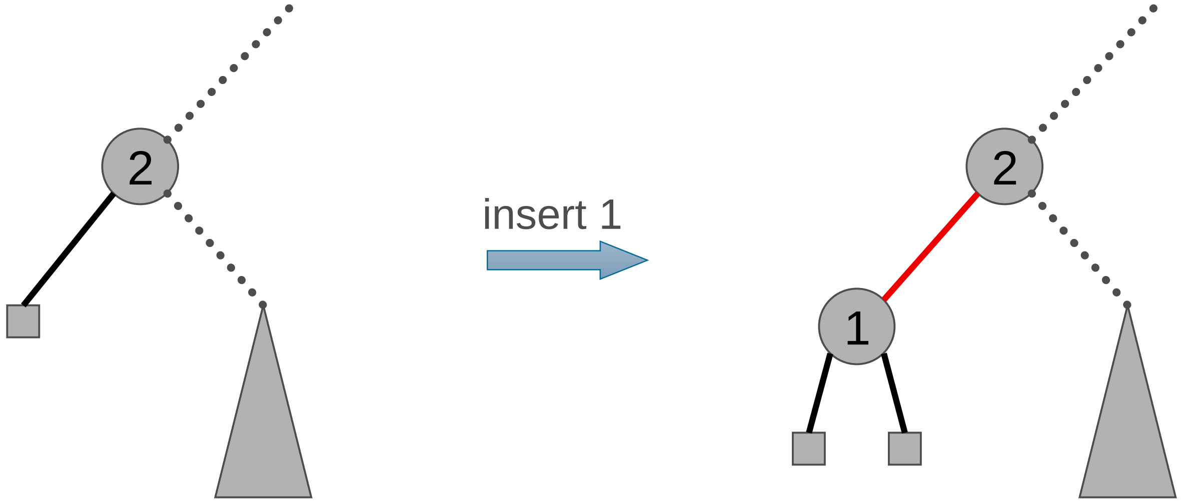


Search algorithm in Red-Black trees is identical to search in binary trees

Implies a search complexity of O(log N)
Height is twice as high as that of the corresponding (2,4) tree

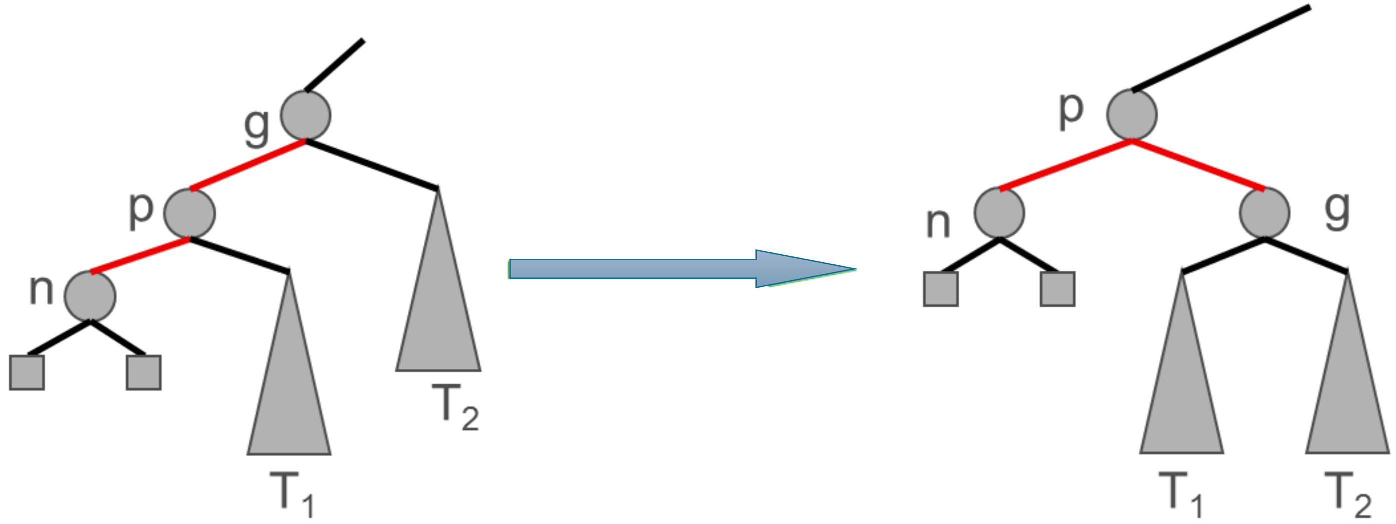
Insert in red-black Trees

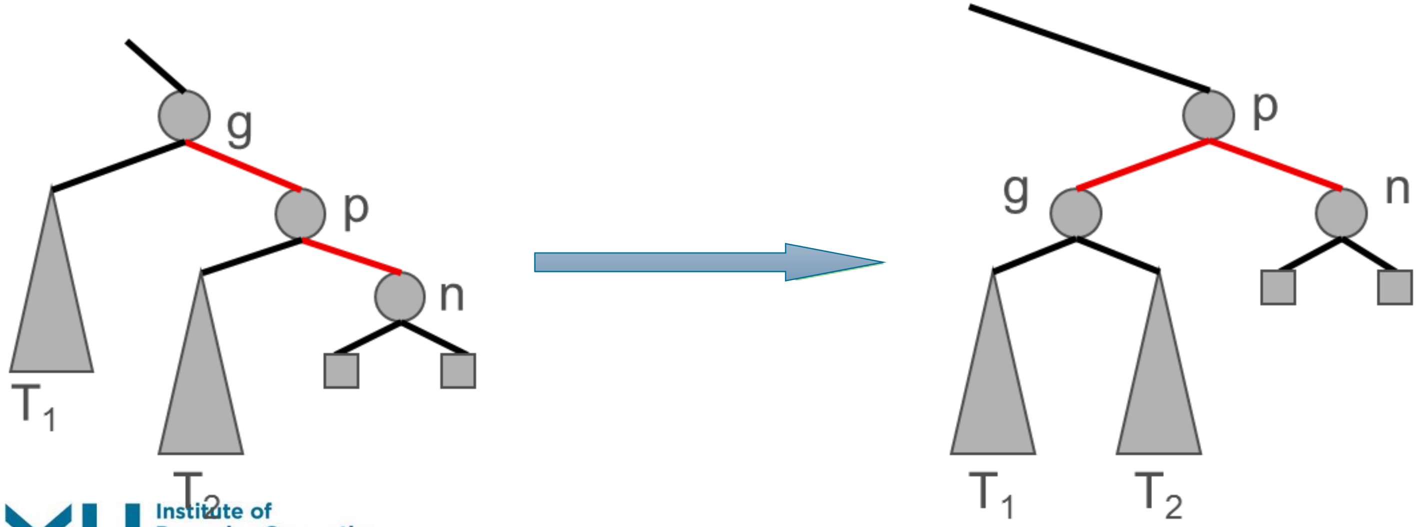
- **Search** for **leaf position** at which the new key is to be **inserted**.
- **Replace leaf** with **internal node** containing **new key**
- **Color** the edge leading into the new node **red**.
- **Attach** two **new leaves** to the new node via **black edges**



- If the parent of the new node has already an incoming red edge, two red edges would follow each other. Therefore restructuring by **rotation** or **promotion** is required.

Insert in red-black Trees

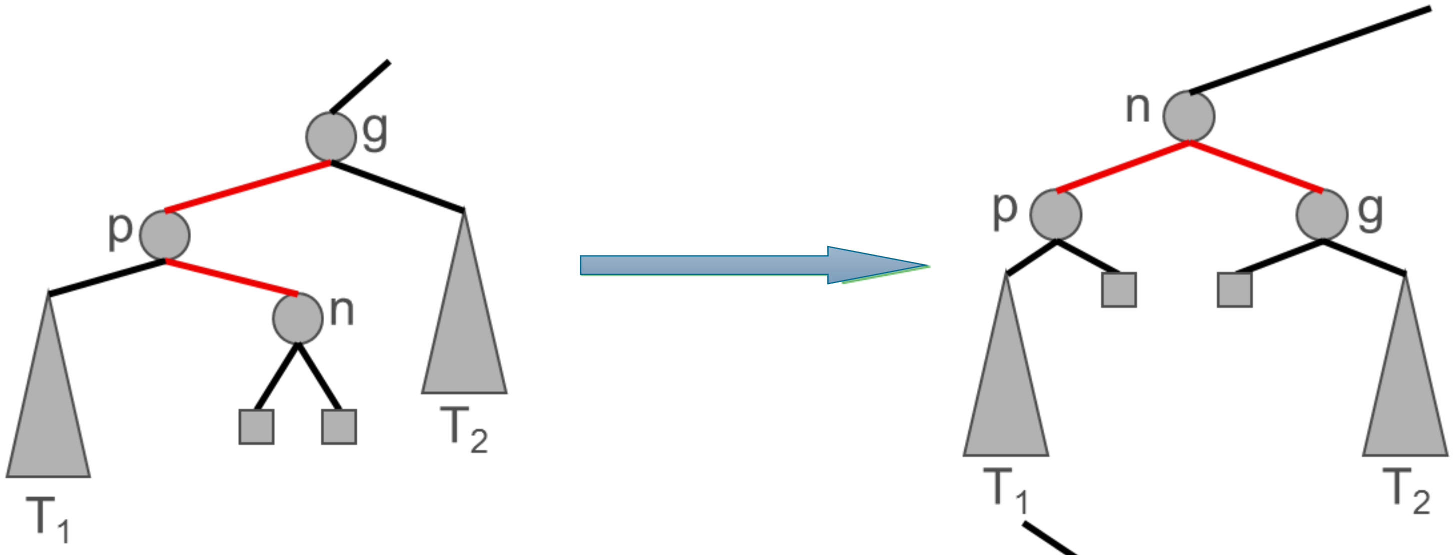


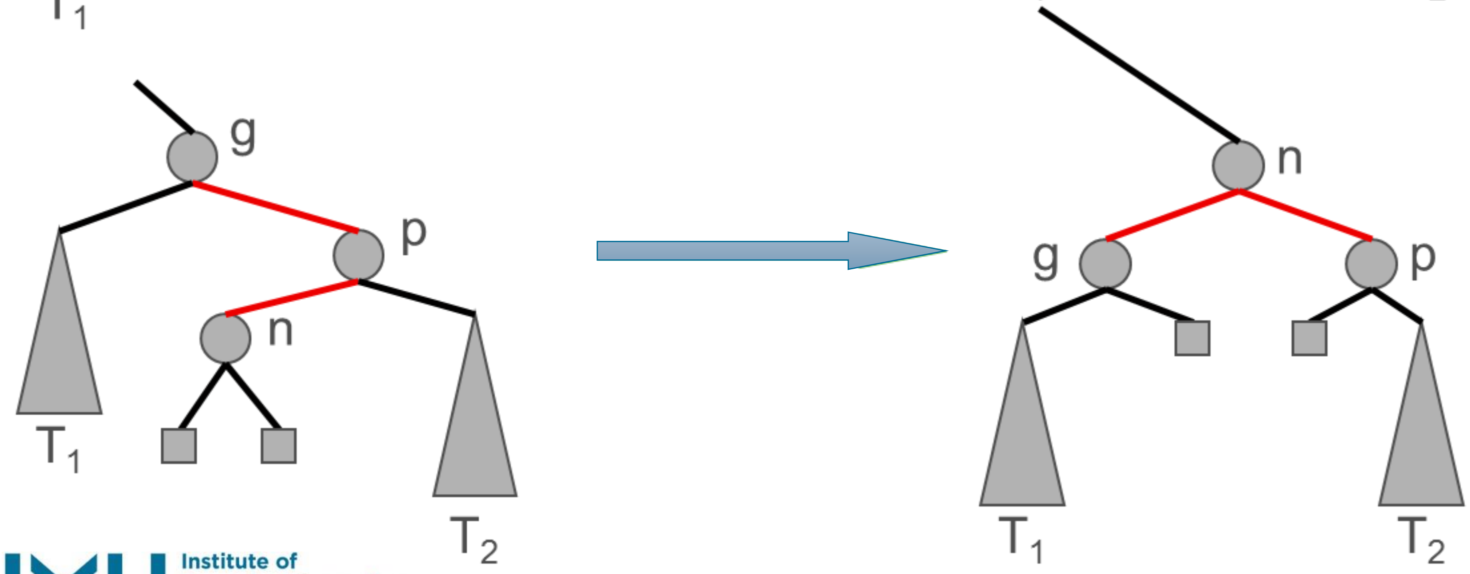


Restructuring by **rotation** (**single**)

n ... new node
p ... parent
g ... grandparent

Insert in red-black Trees





Restructuring by **rotation** (**double**)

n ... new node
p ... parent
g ... grandparent

Insert in red-black Trees

Restructuring by **promotion** (if sibling of p is also red)

n ... new node
p ... parent
g ... grandparent

Can propagate upwards (if parent of g has a red incoming edge)

Insert in red-black Trees :: Summary

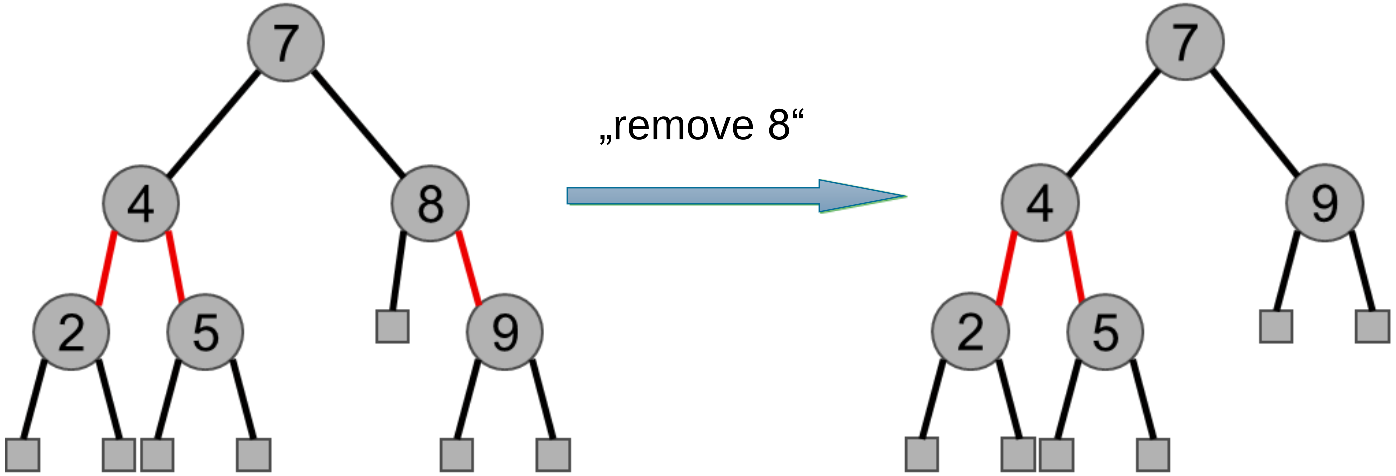
If two red edges follow each other after insertion, then

- **restructure** tree by **single** or **double rotation** - done!
- or **recolor** edges (if necessary propagate upwards)

Runtime

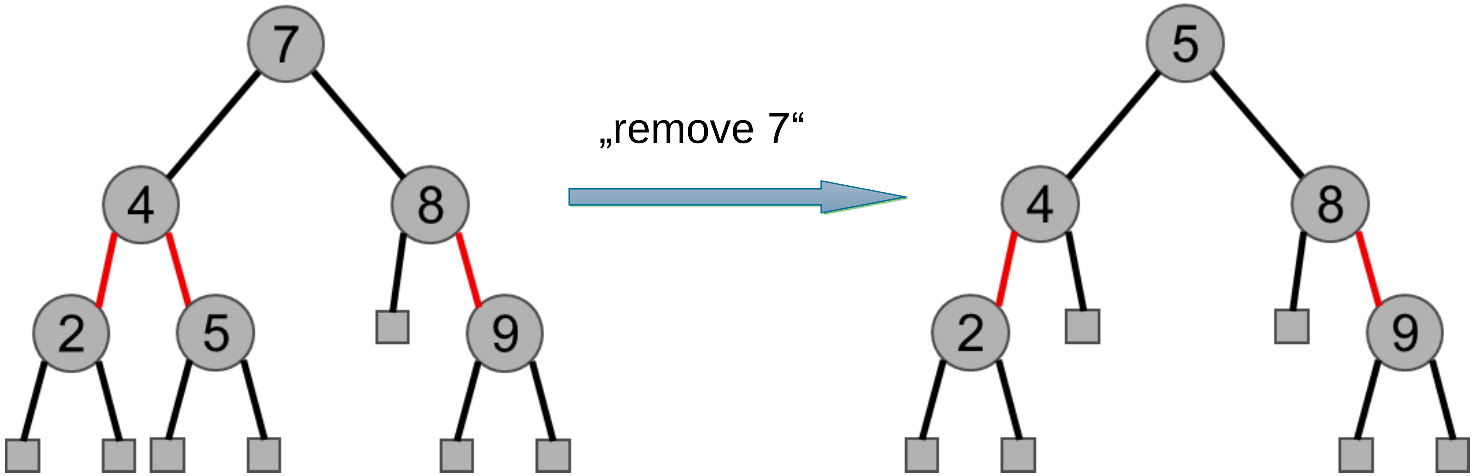
- **Restructuring**: O(1)
- **Promotion (Recoloring)**: O(log N)
  if it propagates until root is reached
- Therefore the **overall** complexity for insert is: O(log N)

Removal in red-black Trees



Case 1:

The node to be deleted has **at least one external** node as a child.



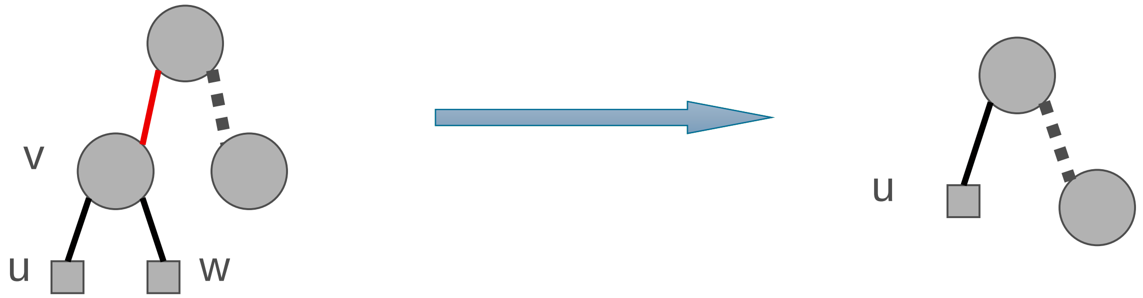
Case 2:

The node to be deleted has **no external node** as child, then replace node with in-order predecessor (or successor).

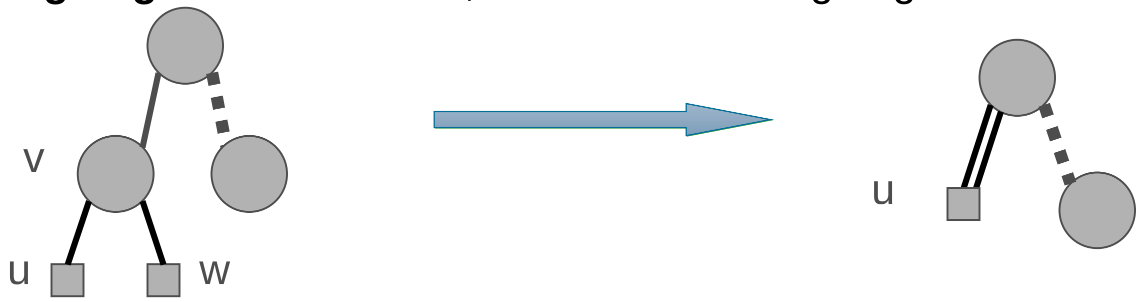
Removal in red-black Trees

Algorithm for Remove:

1. Remove **v** by “**removeAboveExternal**” operation on a leaf **w**, which is a child of **v**.
2. If **parent incoming edge** to **v** was red, color the incoming edge to **u** now black



3. If **parent incoming edge** to **v** was black, color the incoming edge to **u** double-black.



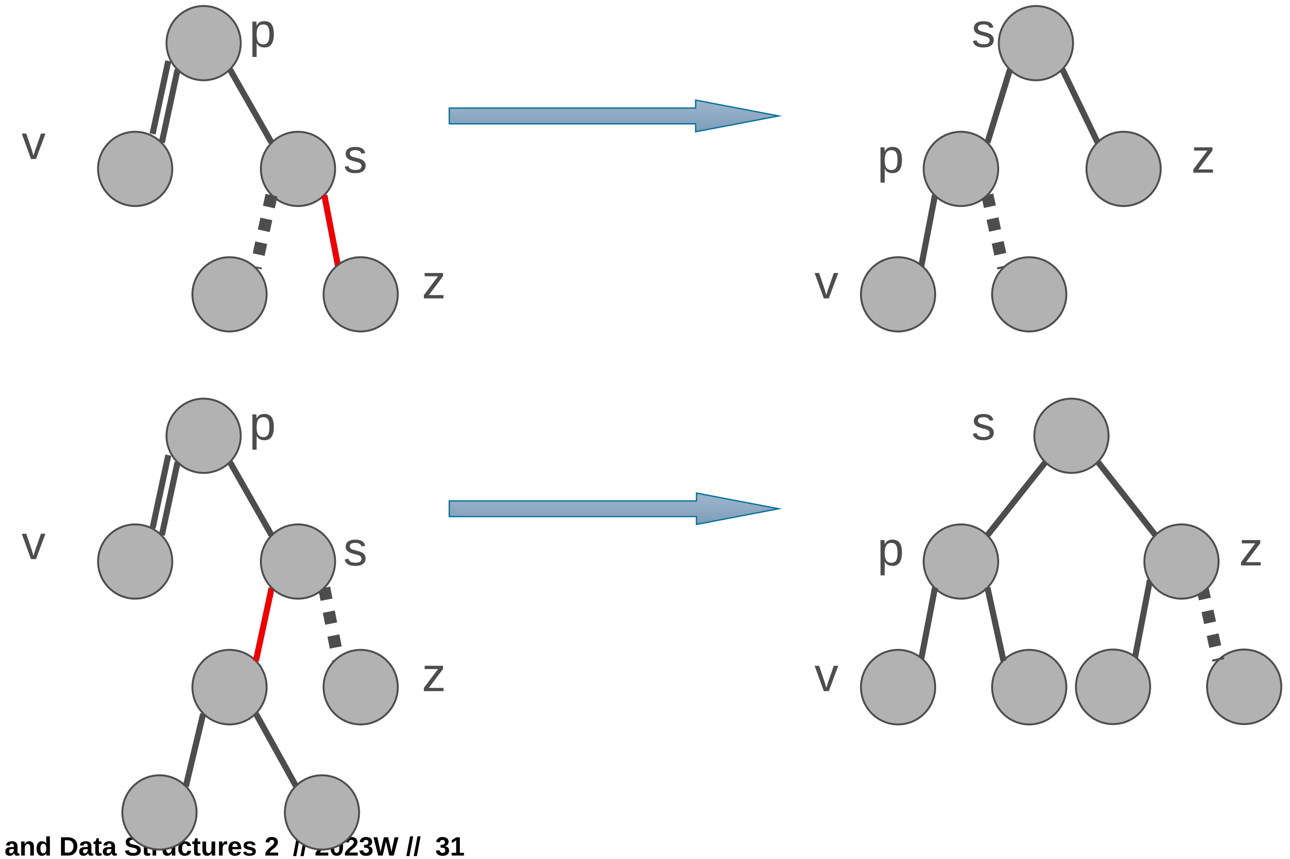
4. As long as there are double-black colored edges, “**color compensation**” by **restructuring** or **recoloring** is required (total number of black edges must be preserved).

Removal in red-black Trees

Elimination of double-black colored edges

- Search „**nearby**“ **red** edge and **change colors** from (red, double-black) in (black, black)

Case 1: **black sibling** with **red child**;
Restructuring

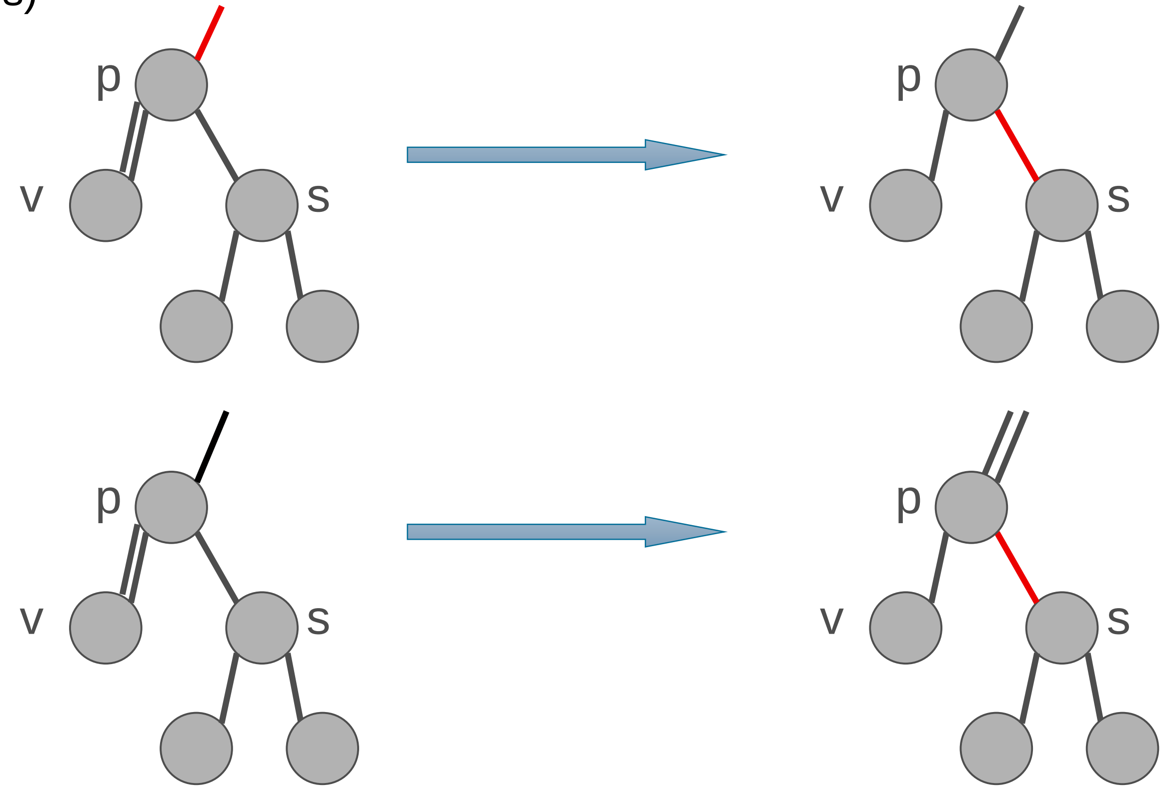


Removal in red-black Trees

Elimination of double-black colored edges

- Search „nearby“ red edge and change colors from (red, double-black) in (black, black)

Case 2: black sibling with **black child**;
Recoloring (with possible **propagation** upwards)

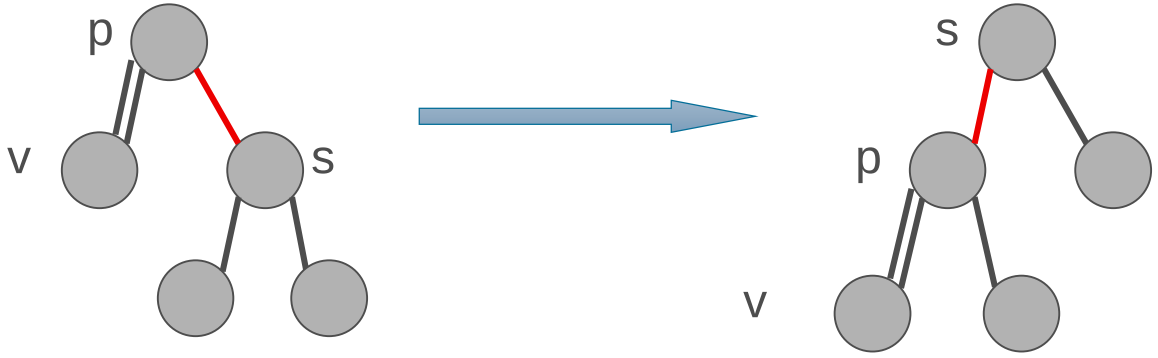


Removal in red-black Trees

Elimination of double-black colored edges

- Search „nearby“ red edge and change colors from (red, double-black) in (black, black)

Case 3: red sibling;
Adjustment



Then proceed according to case 1 or 2.

Red-black Trees :: Summary

Insert or **Remove** can cause a **local interference** (successive red or double-black colored edges)

Resolving the interference

- **Locally** by **restructuring**
- **Globally** by **propagation** on higher levels (recoloring)

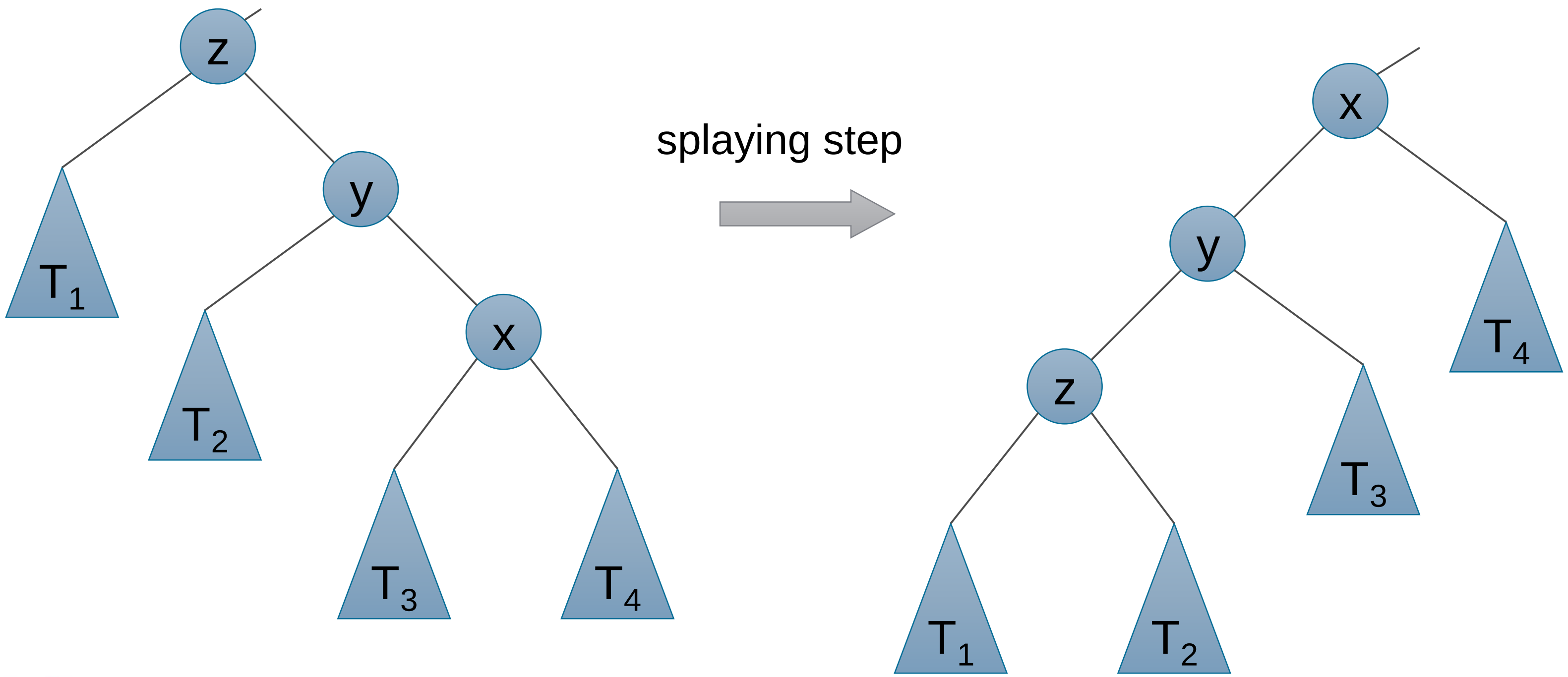
Complexity

- One restructure or recolor step: O(1)
- **Insert**: at maximum **1 restructure** step or O(log N) **recolor** steps
- **Remove**: at maximum **2 restructure** steps or O(log N) **recolor** steps
- **Overall** complexity: O(log N)

Splay Trees

Binary search tree, in which “splaying” is done **after each access operation** => adaption to search queries

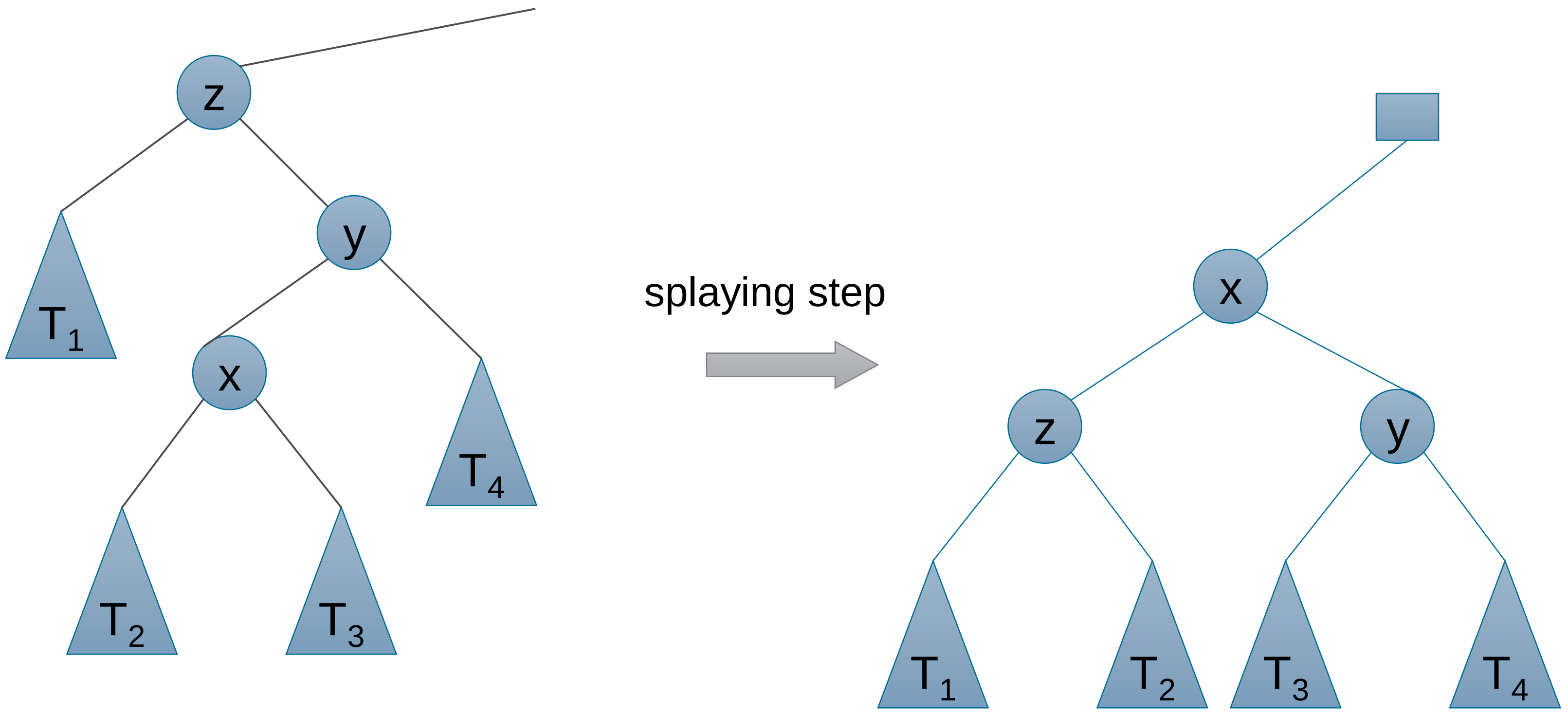
- **Splaying**: special **move-to-root operation** is applied to node **x**
- **Three cases** for one step in splaying:
  - zick-zick: **x** is right (left) child of **y**, **y** is right (left) child of **z**



Splay Trees

Three cases for one step in splaying:

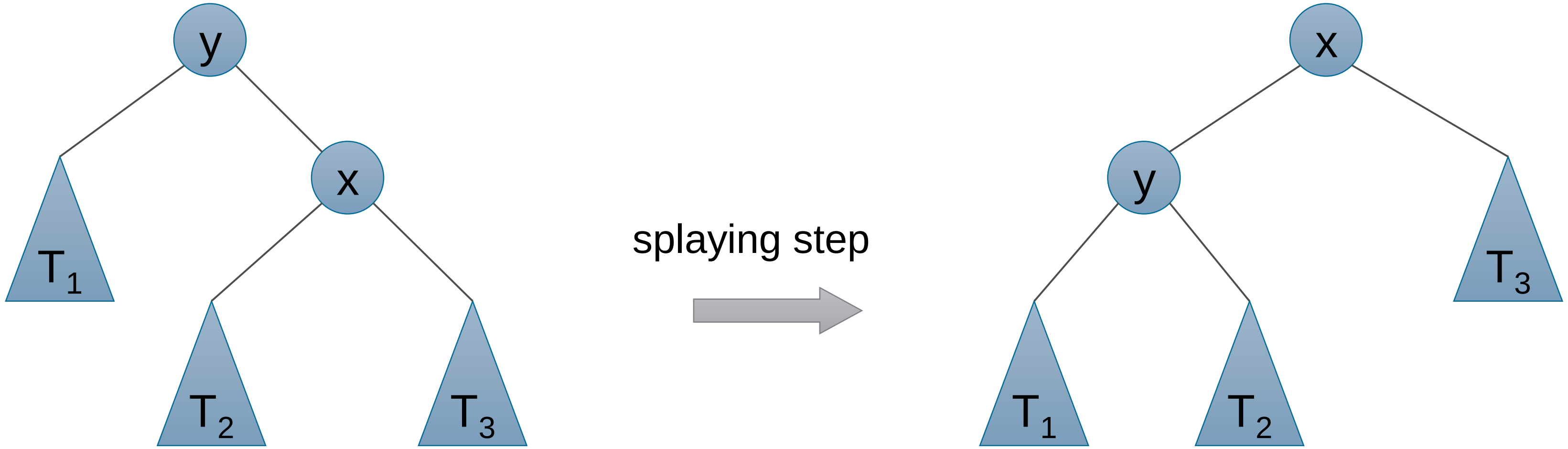
- **zick-zack:** **x** is left (right) child of **y**, **y** is right (left) child of **z**



Splay Trees

Three cases for one step in splaying:

- zick: **x** is left (right) child of **y**, **y** is root



Splay Trees

Splaying operation **starts** at the **lowest** node **x**, which is **visited** in an **access operation (insert, delete, find)**

Is executed **until** this node **x** is the **root**

In each zick-zick or zick-zack step the **height** of **x decreases** by **2**, in one zick step by 1

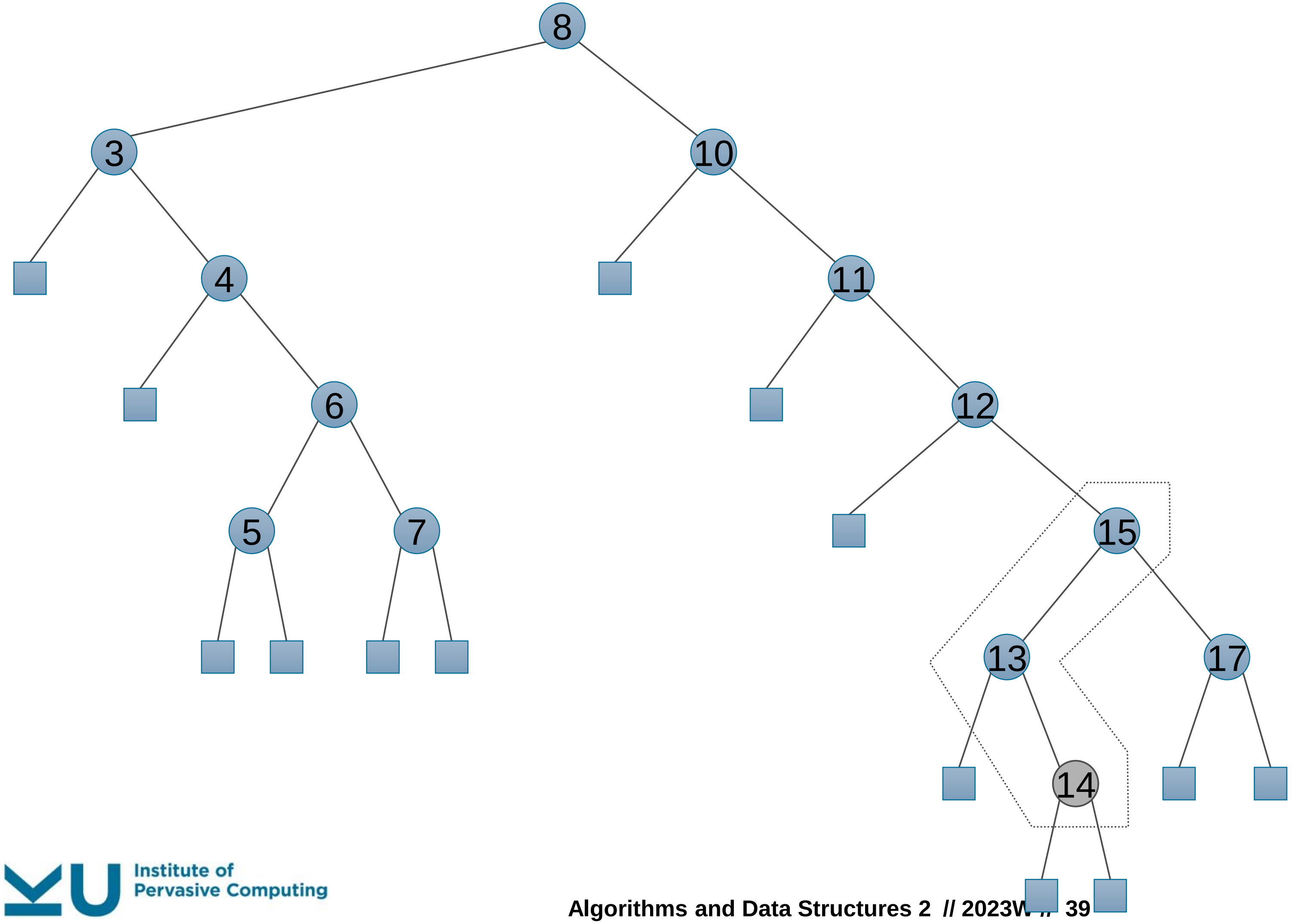
- Therefore, splaying the node **x** with **depth d** requires $\lfloor \mathbf{d/2} \rfloor$ zick-zacks or zick-zicks if **d** is even and an additional zick if **d** is odd

Each of these operations affect a **constant** number of nodes, so the complexity is O(1)

- Therefore we have a **complexity** for **splaying** of **O(d)**

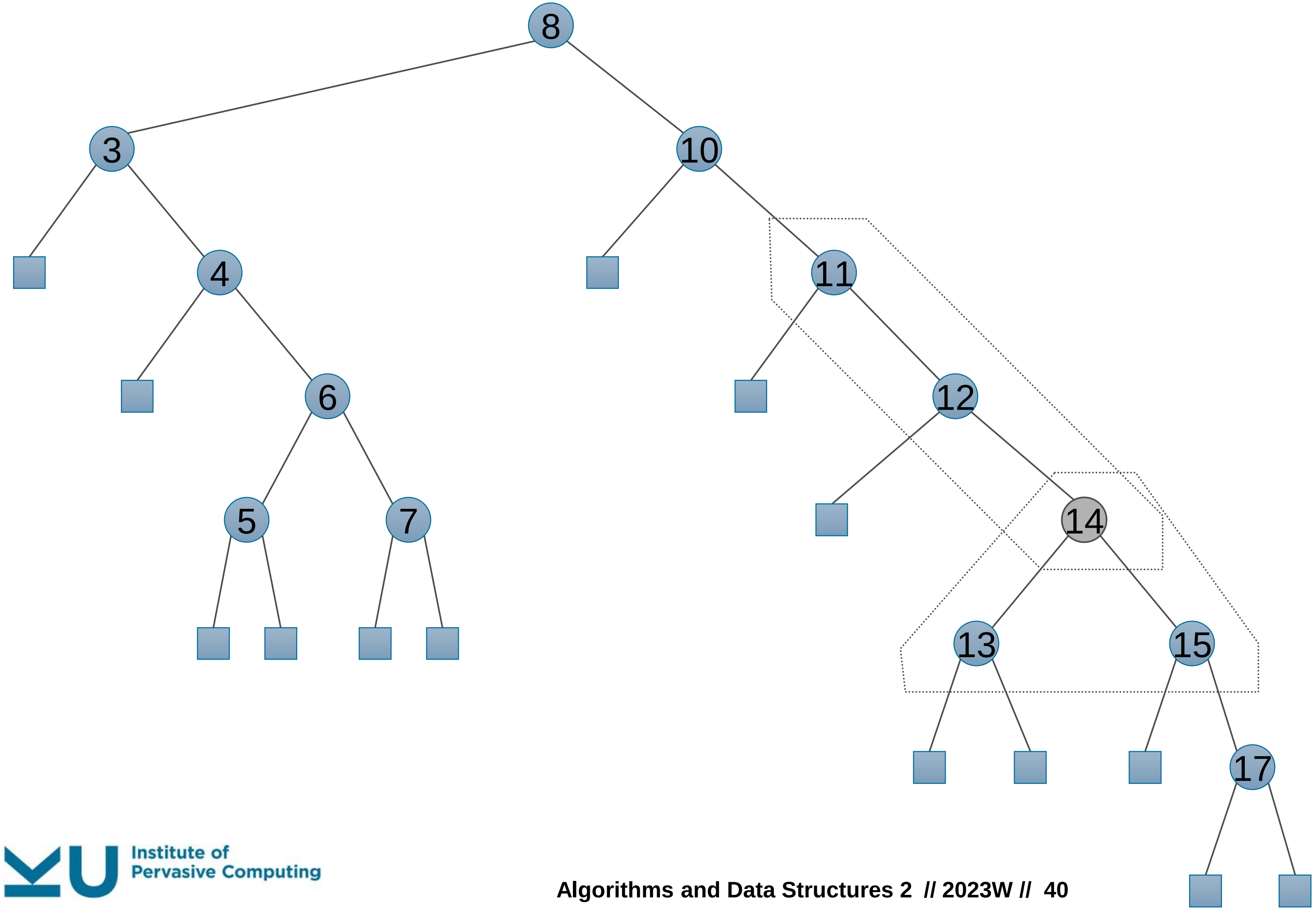
  d ... depth of the tree

Splay Trees :: Example



splaying starts at node 14: zick-zack

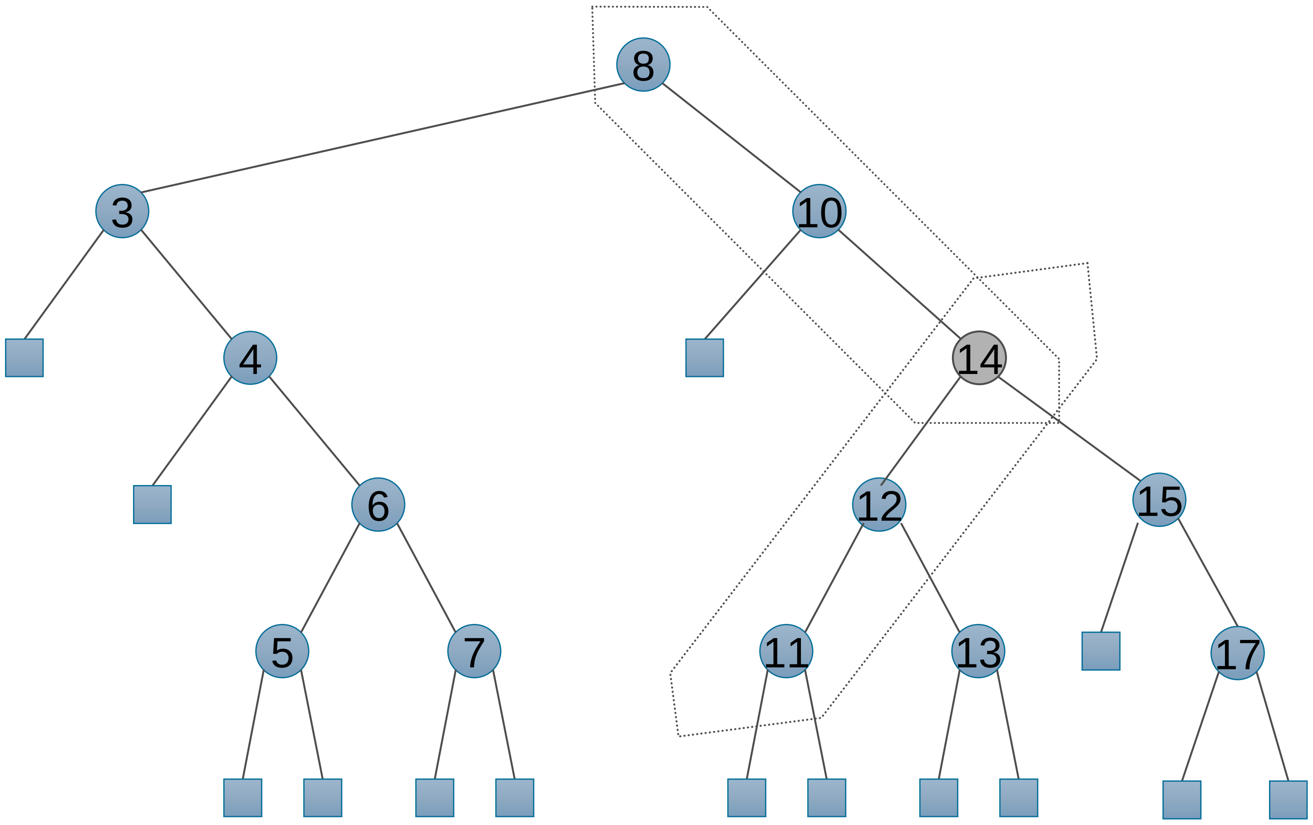
Splay Trees :: Example



after zick-zack

next step: zick-zick

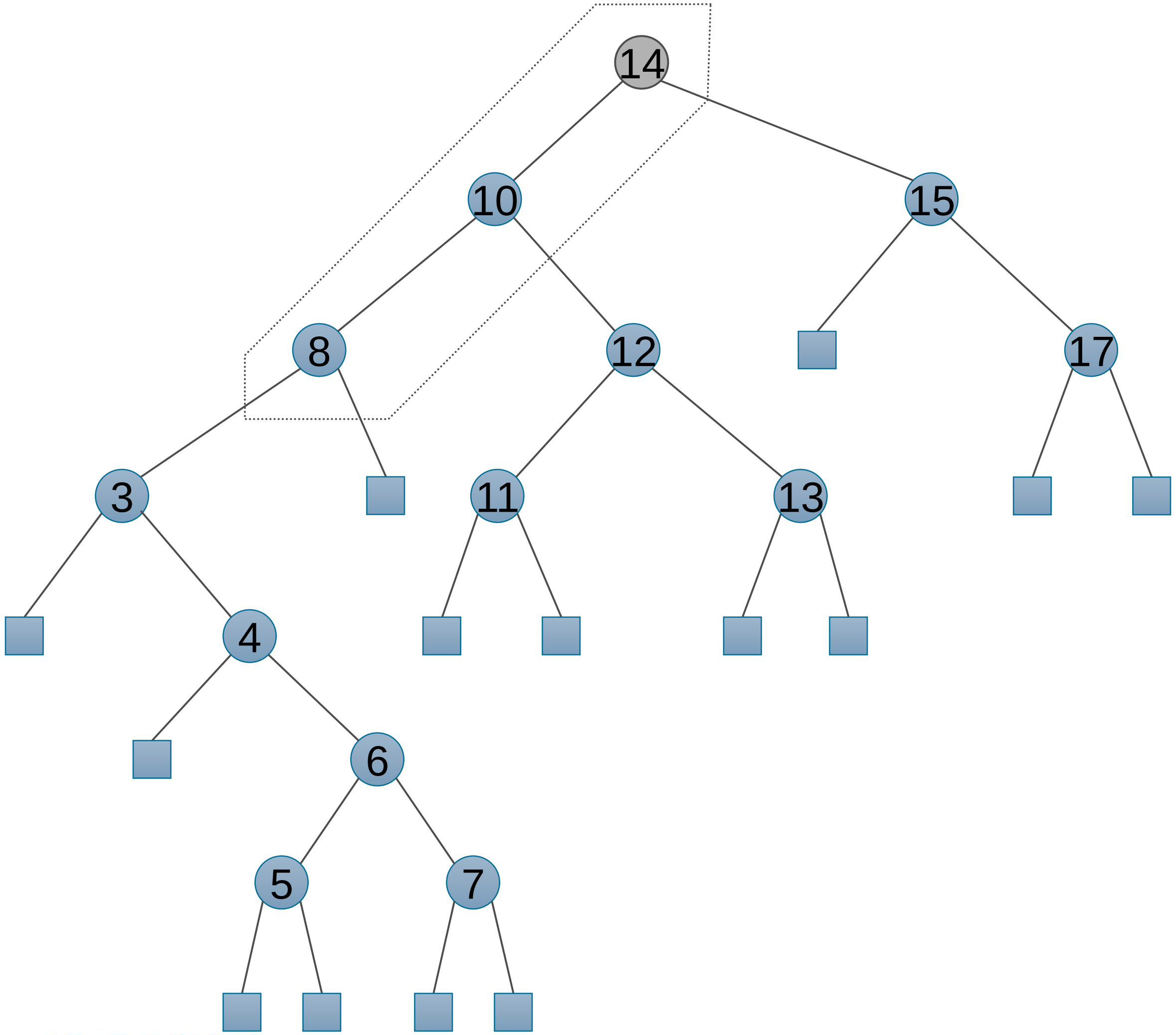
Splay Trees :: Example



after zick-zick

next step: zick-zick

Splay Trees :: Example



after zick-zick – done!

Splay Trees

When and at which node is splaying performed?

Search for key k

- Case 1: node x **contains** key k, then splaying of x
  (previous example could be considered as splaying after “find 14”)
- Case 2: search unsuccessful, then splaying of **parent** of **last visited** leaf
  (previous example could be considered as splaying after “find 14,5”)

Insertion of key k

- Splaying is done **with new** node x containing k
  (previous example could be considered as splaying after “insert 14”)

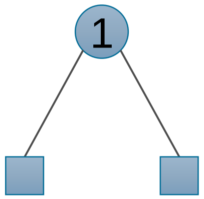
Removal of key k

- Splaying with **parent of removed** node

Splay Trees

Example for Insert

- original tree



Splay Trees

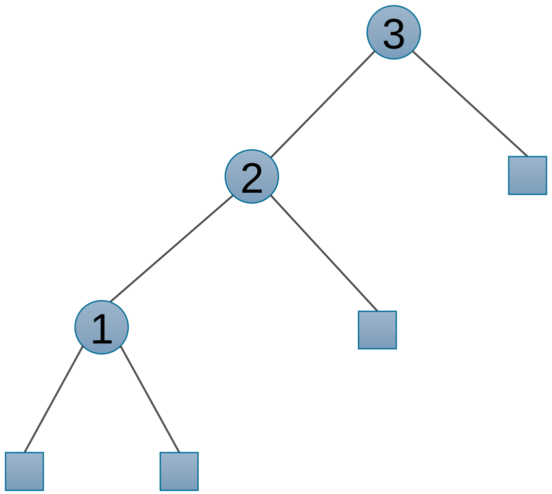
Example for Insert

- original tree
- after Insert of 2

  after splaying

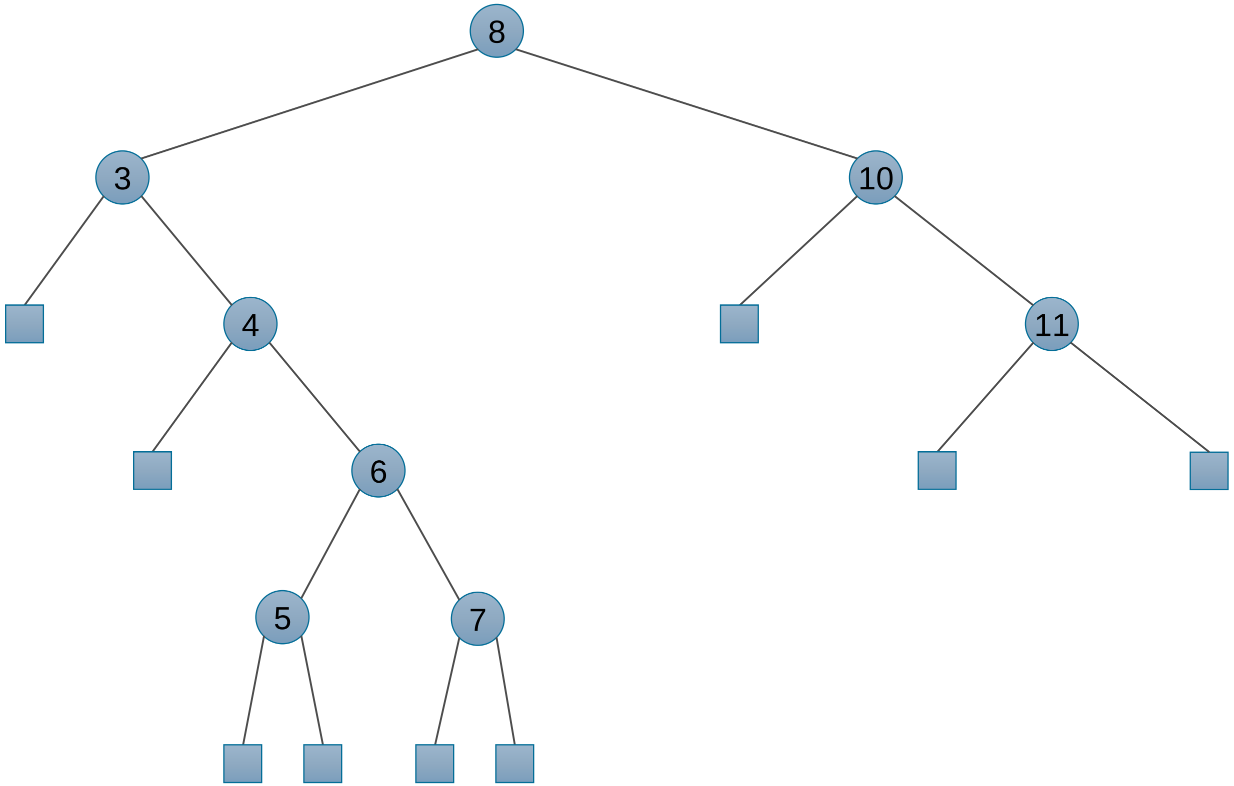
  after insertion of 3

  after splaying



Splay Trees

Example for Remove: remove 8

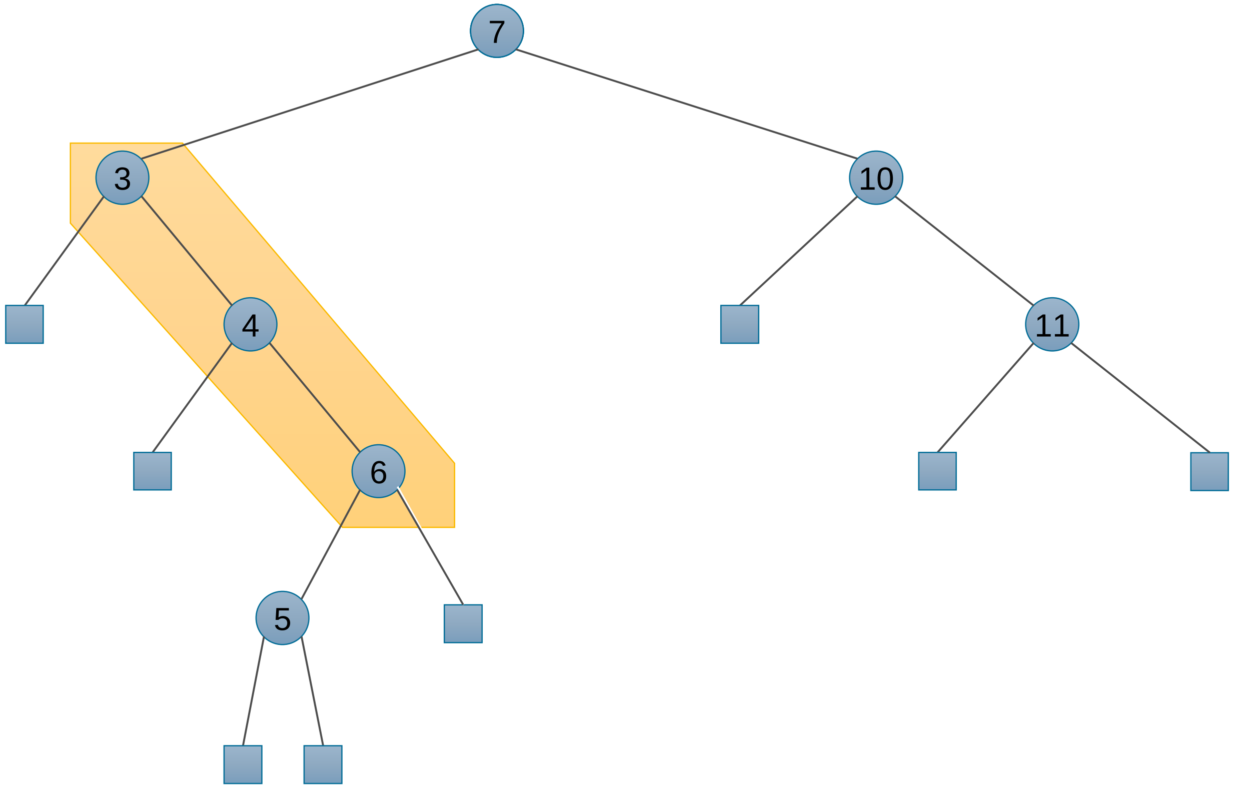


Splay Trees

Example for Remove: remove 8

Set the rightmost internal node of the left subtree of 8 at the position of 8

Splaying with paret of this node (node 6)

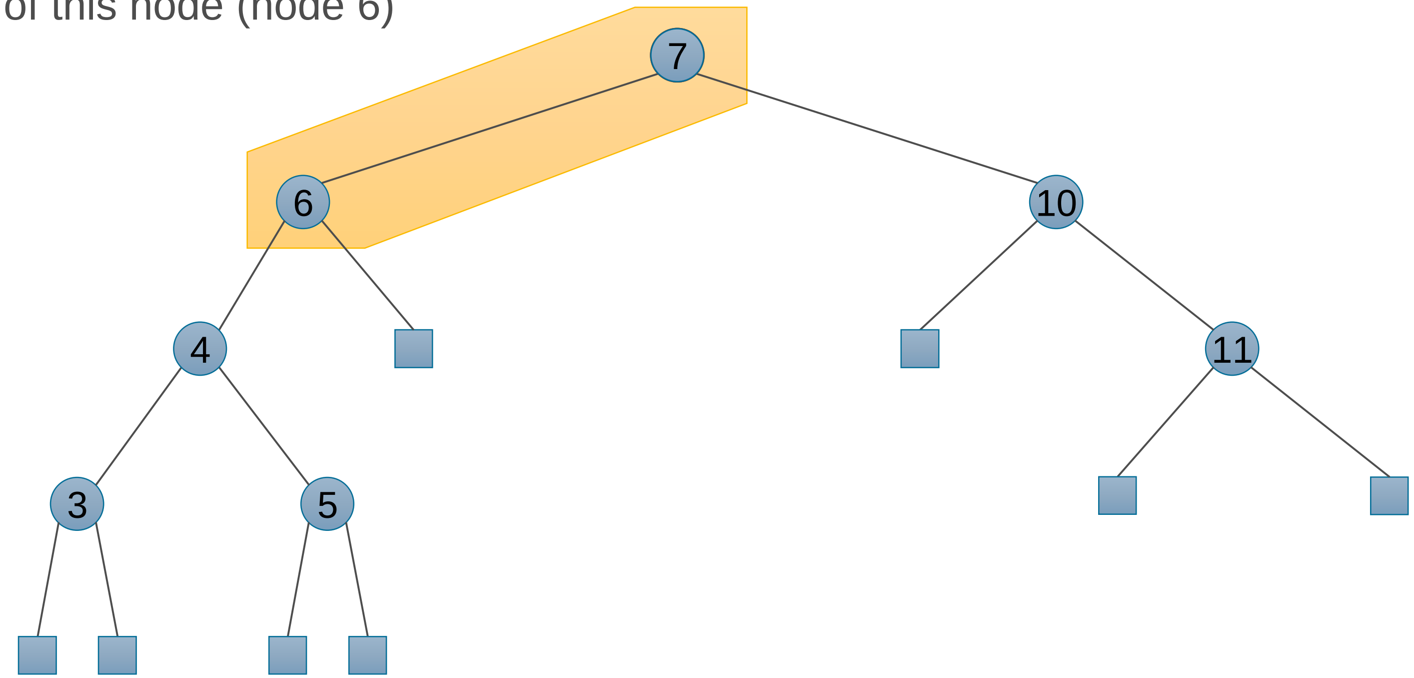


Splay Trees

Example for Remove: remove 8

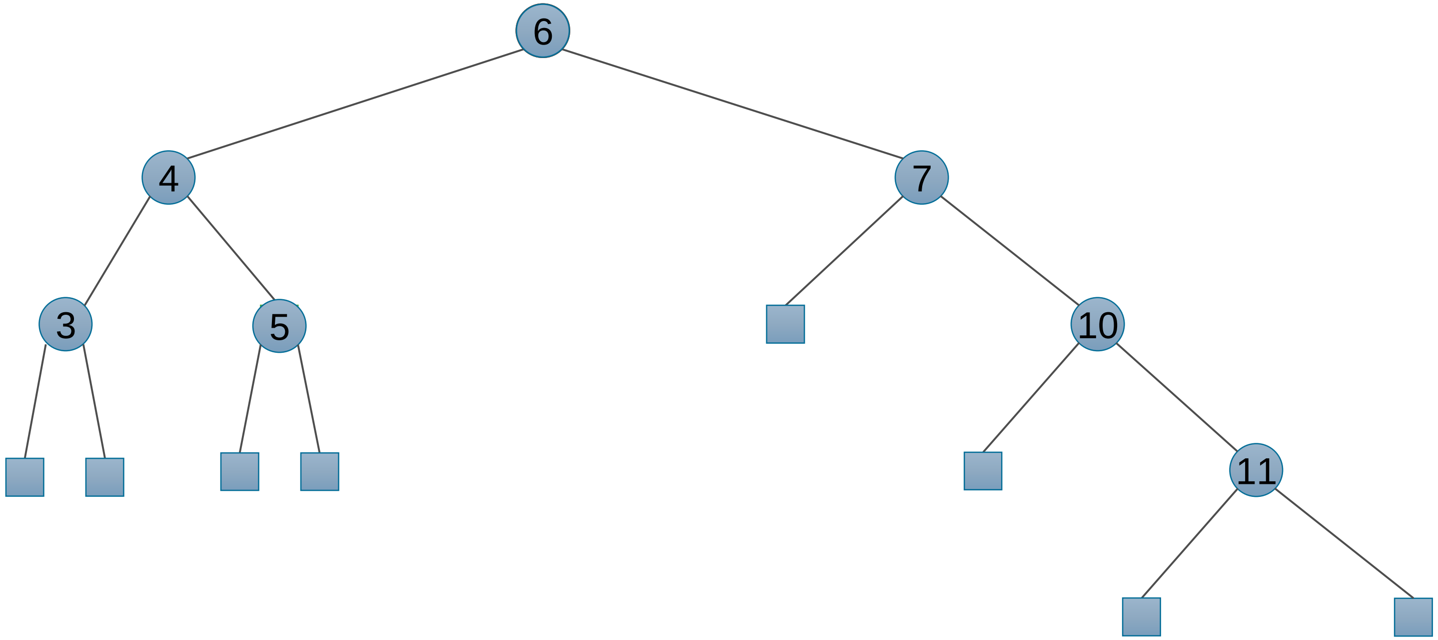
Set the rightmost internal node of the left subtree of 8 at the position of 8

Splaying with paret of this node (node 6)



Splay Trees

Example for Remove: remove 8



Splay Trees :: Analysis

Insert, Remove, Search: O(h)
(with *h* = depth of the tree)

in worst case: h = N therefore O(N)
(see example of insertion)

An amortized analysis (using the accounting method) can show that on average we have:
O(log N)

Splay Trees :: Analysis

Implementation	Search Time		Insertion Time		Deletion Time	
Skip lists	0.051 msec	(1.0)	0.065 msec	(1.0)	0.059 msec	(1.0)
Non-recursive AVL trees	0.046 msec	(0.91)	0.10 msec	(1.55)	0.085 msec	(1.46)
Recursive 2-3 trees	0.054 msec	(1.05)	0.21 msec	(3.2)	0.21 msec	(3.65)
Self-adjusting trees:						
Top-down splaying	0.15 msec	(3.0)	0.16 msec	(2.5)	0.18 msec	(3.1)
Bottom-up splaying	0.49 msec	(9.6)	0.51 msec	(7.8)	0.53 msec	(9.0)

Trees (Weight Balanced)

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

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