Institute of
Pervasive Computing

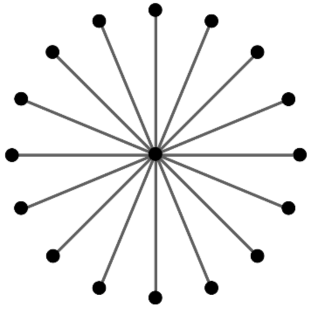
Trees (Height Balanced)

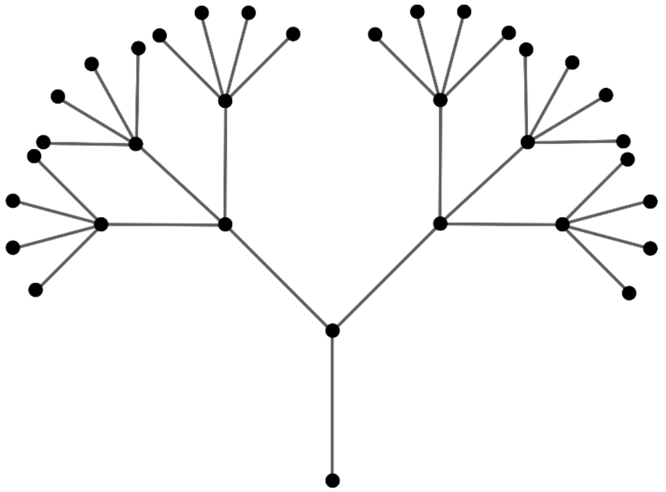
Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

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

Definition & Terminology

A **tree** is
- an acyclic, simple, coherent graph
- i.e. does not contain loops and cycles:
  between each pair of nodes there is at maximum one edge

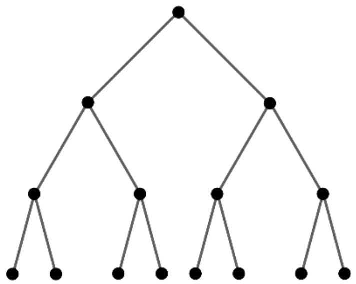




Generalization of lists:
- Element (node) may have **multiple successors**.
- Exactly 1 node has no predecessor: **root**
- Nodes without successors: **leaves**

Frequently used data structure: decision trees, syntax trees, derivation trees, search trees, ...

Frequently used representation form: quantity, bracket, recursive indentation, graph, ...



Here: Use of trees to **store keys** and realization of dictionary operations (**search**, **insert**, **remove**) in e.g. binary trees

Definition & Terminology

Tree *B* is **ordered**, if successors of each node are ordered (1., 2., 3. etc.; left, right).

In an ordered tree, the **subtrees B_i** of each node form an **ordered set** (e.g.: arithmetic expression)

Order of B: maximum number of successors of a node

Path of length k: Follow $p_0, \ldots, p_k$ of nodes, such that p_i is successor of p_{i-1}

- **Height** of a **tree**:
  maximum distance of a leaf to the root.
- **Depth** of a **node**:
  distance to the root (the **number of edges** on a path from this node to the root)
  The nodes at level i are all nodes with depth i.

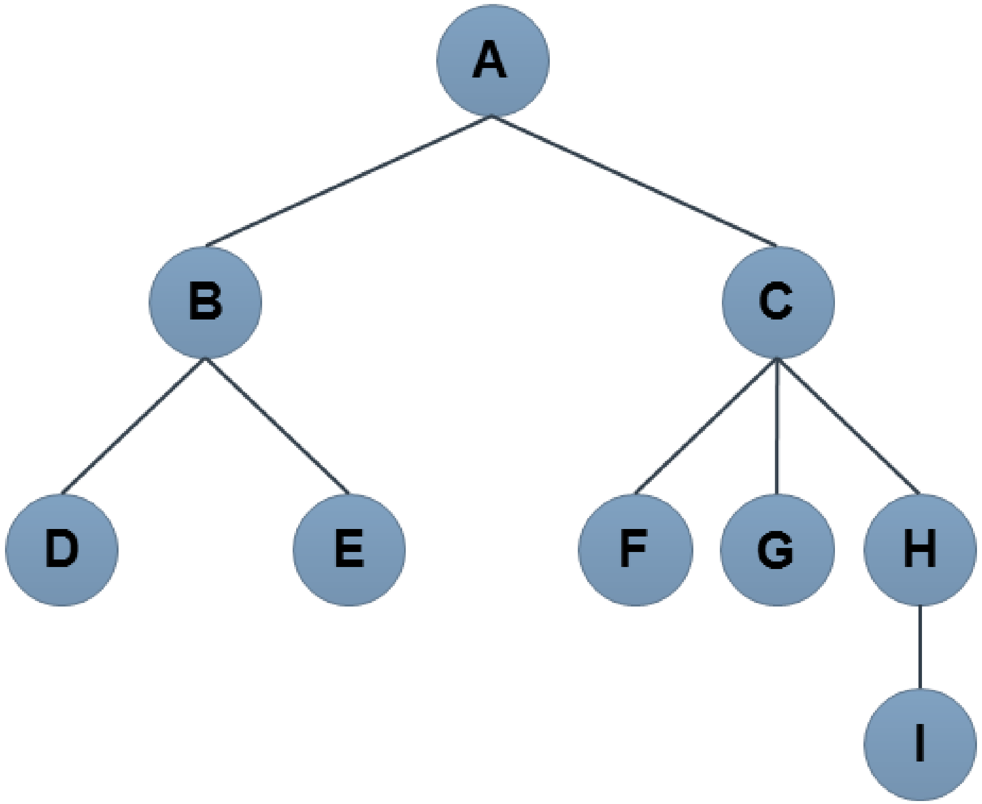
A tree of order n is called **complete** if all leaves have the **same depth**
and the maximum number of nodes is present at each level.

Definition & Terminology

A	**root**
B	is **parent** of D and E
C	is **sibling** of B
D and E	are **children** of B
D, E, F, G, I	are **external nodes** or **leaves**
A, B, C, H	are **internal nodes**

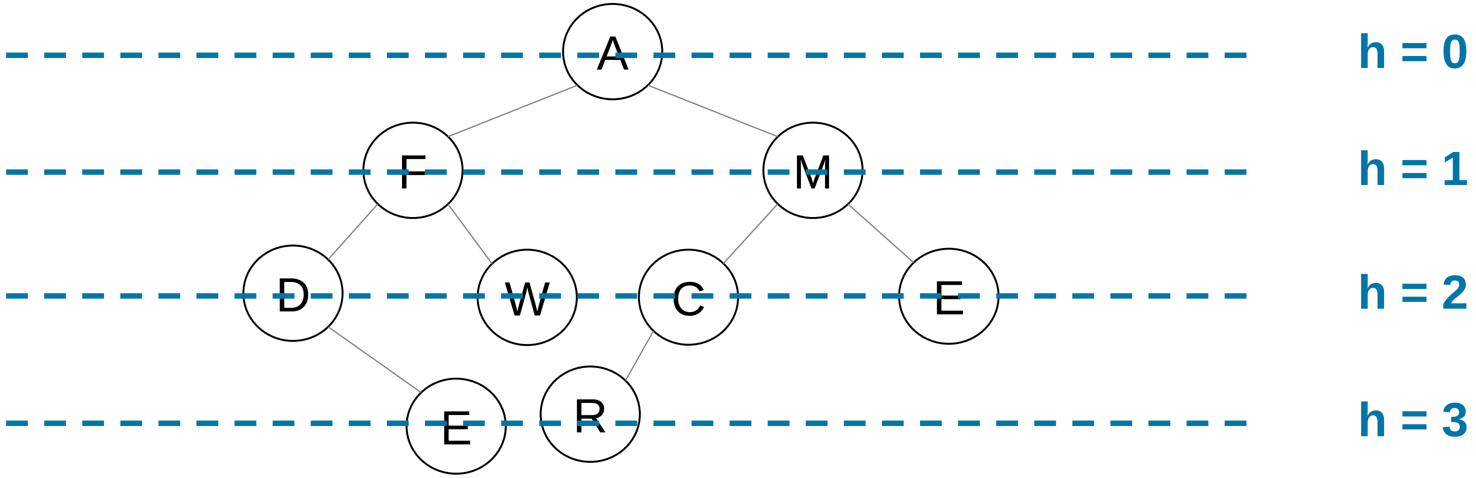
The **depth** of E is 2.
The **height** of the tree is 3.
The **order** of B is 2, the order of C is 3.

We always have: number of edges = number of nodes - 1



Binary Trees

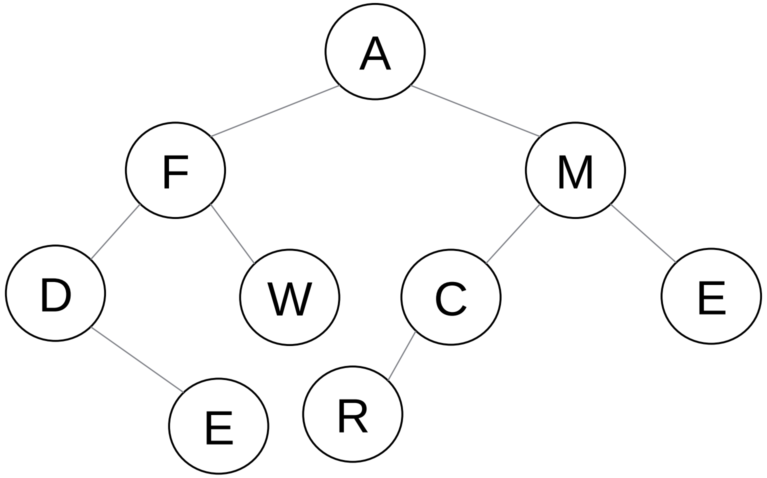
If ***T*** is a **binary tree** **(order of all nodes <= 2)** with **n nodes** and a **height *h***, then *T* has the following properties



- Number of **external nodes** is at least ***h+1*** and at maximum $\boldsymbol{2^h}$

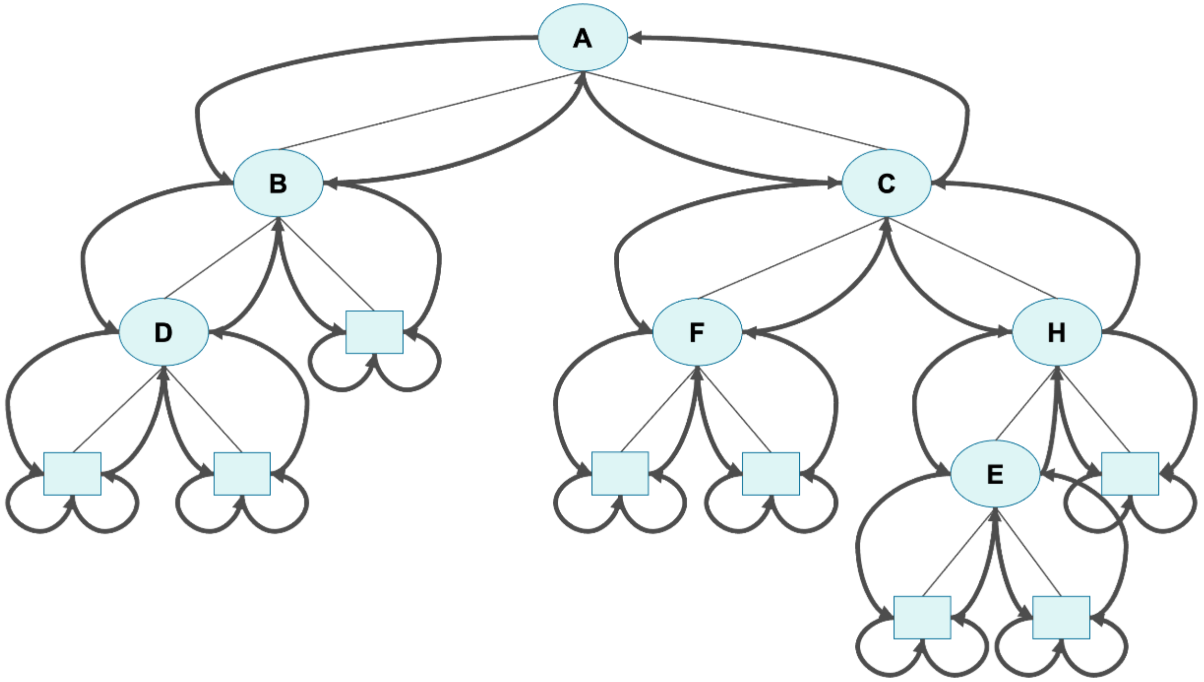
Binary Trees

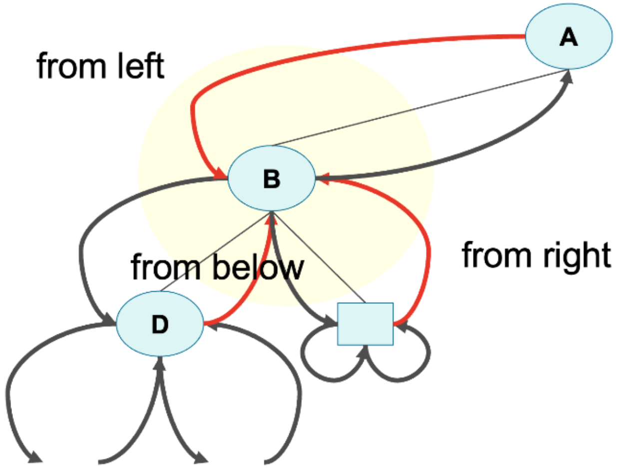
If T is a **binary tree** **(order of all nodes <= 2)** with **n nodes** and a **height** h, then T has the following properties



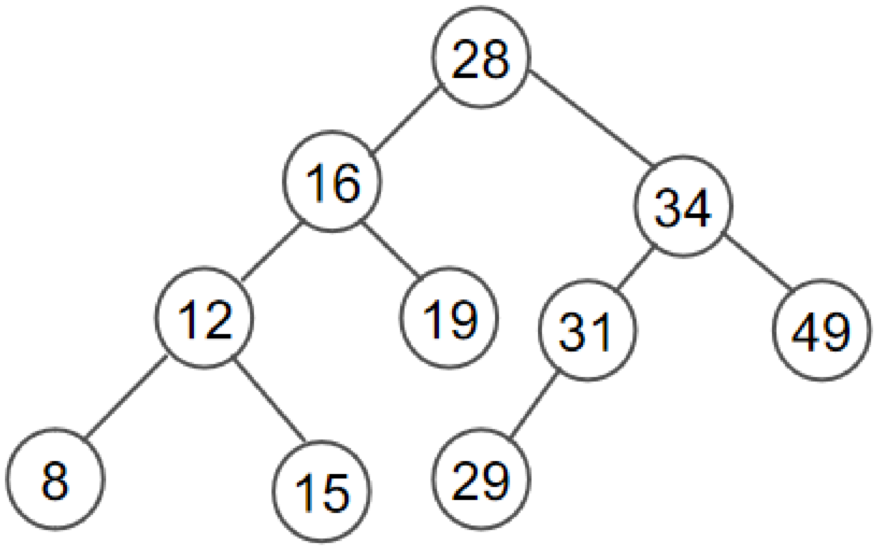
- Number of **external nodes** is at least $h+1$ and at maximum 2^h
- Number of **internal nodes** is at least h and at maximum $2^h - 1$
- Number of **nodes** is at least $2h+1$ and at maximum $2^{h+1} - 1$
- The **height** of T is at least $\log(n+1) - 1$ and at maximum $(n-1)/2$

Traversing Binary Trees





Traversing Binary Trees



- **Preorder**:
  28, 16, 12, 8, 15, 19, 34, 31, 29, 49
- **Postorder**:
  8, 15, 12, 19, 16, 29, 31, 49, 34, **28**
- **Inorder**:
  8, 12, 15, 16, 19, **28**, 29, 31, 34, 49

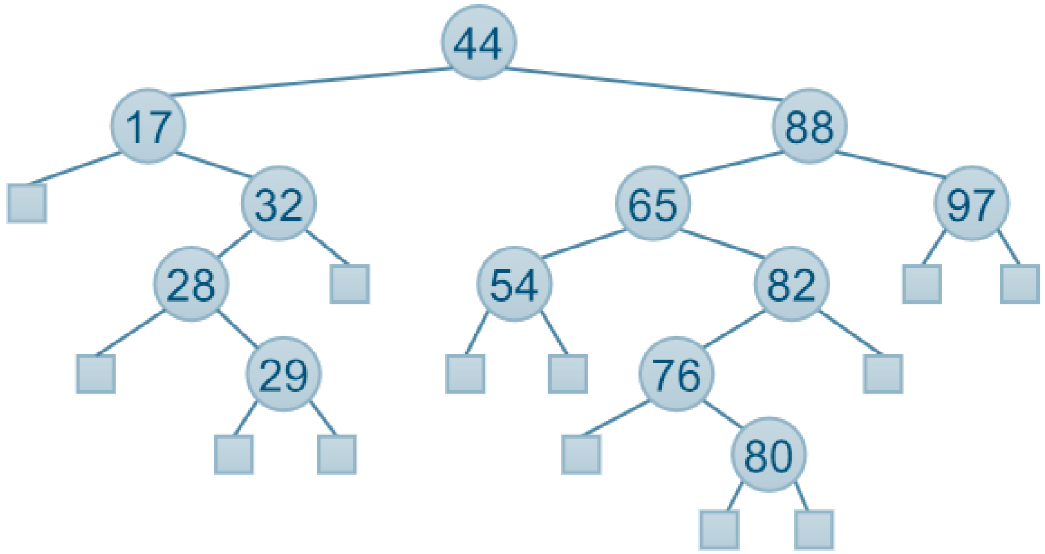
Hints: If you traverse the tree starting from the root, you have

- **Preorder** by visiting all nodes on the left side of which you pass by.
- **Postorder** by visiting all nodes on the right side of which you pass by.
- **Inorder** by visiting all nodes on the bottom side of which you pass by.

Binary Search Trees

A **binary search tree** is a **binary tree** ***T*** where:

- Each internal node stores a **key-value pair** of a dictionary
- Keys which are stored in nodes of the **left** subtree of a node *v*, are **less than** or **equal** to the key stored in *v*
- Keys which are stored in nodes of the **right** subtree of a node *v*, are **greater** than the key stored in *v*
- **External nodes** serve only as **placeholders** and do not store elements



Searching in Binary Search Trees

Assumption: For each node **n** with children $\mathbf{c_l}$, $\mathbf{c_r}$ and key **k** we have (search tree condition):

- All keys stored in subtree of $\mathbf{c_l}$ are **smaller** than **k**
- All keys stored in subtree of $\mathbf{c_r}$ are **larger** than **k**
- All keys stored in subtree of $\mathbf{c_l}$ are **smaller than all** keys stored in subtree of $\mathbf{c_r}$

Search for key x:

1. Compare key **k** of inspected node **n** with **x**.
2. If **n == null**, **x** is **not in** the binary tree.
3. If **k < x**, set $\mathbf{n = c_r(n)}$ and jump to 1.
4. If **k > x**, set $\mathbf{n = c_l(n)}$ and jump to 1.
5. If **k == x**, found.

Maximum number of inspected nodes: height of tree

Search within the nodes, e.g. by linear or binary search. As $l \leq m$, the complexity is constant.

Searching in Binary Search Trees

Binary search tree is a **decision tree**:
in each **internal node v** the key to be searched for is compared with the **key stored in v**

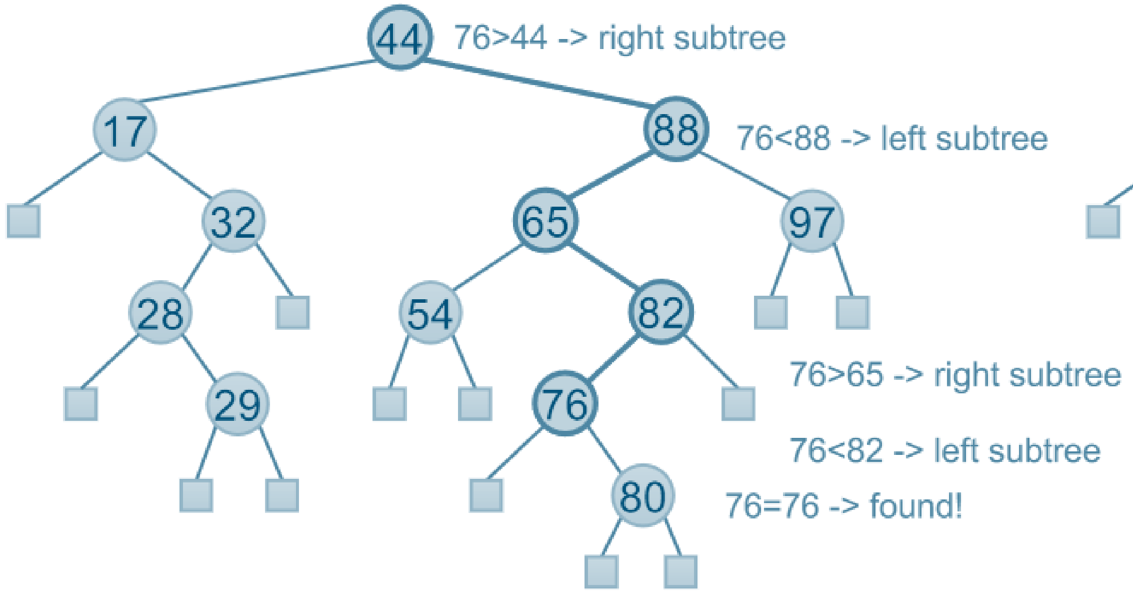
Pseudocode

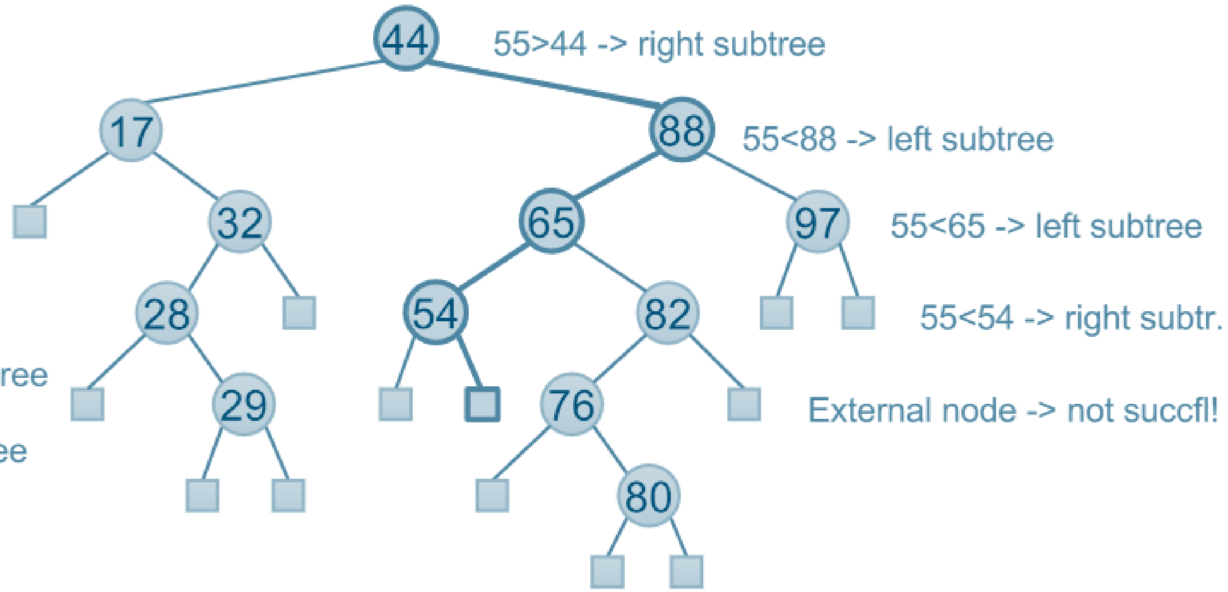
```
Algorithm TreeSearch(k, v)
  Input: key k to be searched, node v of a binary search tree
  Output: node w in subtree of v,
           for successful search internal node with key k,
           for unsuccessful search
    external node after all internal nodes smaller than k and before all internal nodes greater than k
  if v is an external node then
    return v
  if k = key(v) then
    return v
  else if k < key(v) then
    return TreeSearch(k, T.leftChild(v))
  else
    return Treesearch(k, T.rightChild(v))
```

Examples for Search in Binary Search Trees

TreeSearch(76, root)
(**sucessful** search)

TreeSearch(55, root)
(**unsucessful** search)



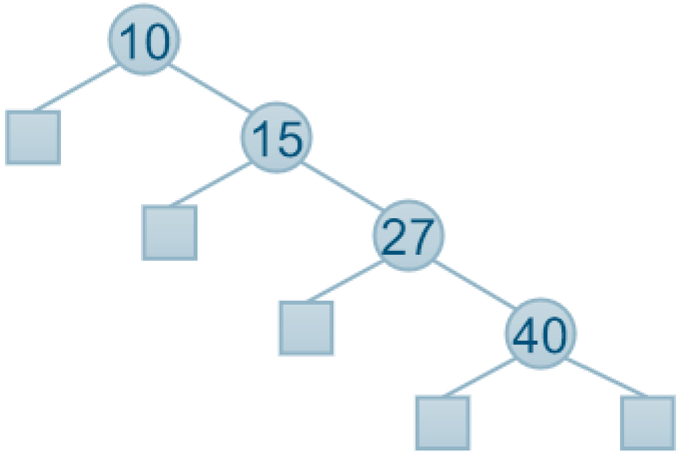


Complexity of Binary Search Trees

For **searching**, **inserting**, **removal**, nodes along a **root-leaf-path** (plus possibly siblings of such nodes) needs to be traversed.

- Complexity per node: O(1)
- Therefore the runtime is O(h), where h is the **height of the tree**

In worst case the height of the binary search tree is h = N
(tree „degenerates“ to sorted sequence)



- Worst case complexity **O(N)**
- Therefore, the tree must be kept **balanced** to allow operations in **O(log N)**

Trees

Problem: trees can degenerate (e.g. become lists)

Removal of a node

- **Very easy:** Removal of a leaf
- **Easy:** Removal of a node **p** with 1 successor:
  replace node with successor
- **More difficult:** Removal of a node **p** with 2 successors:
  Search for the **leftmost node q** in **right subtree** (symmetric successor).

  Replace **p** with **q**, remove **q** from its original position.

Binary Search Tree :: Access Cost

Cost measure: Number of nodes visited or number of search steps or key comparisons required.

Average access cost z of a tree **B** ...

- is obtained by calculating its **total path length PL** as the **sum of the lengths of all paths** from the root to each node $\mathbf{K_i}$.

$$PL(B) = \sum_{i=1}^{N} Level\ (K_i)$$

- The mean path length is calculated by $l = \frac{PL}{N}$

Maximum access cost We have the longest search path and thus the maximum access cost, when the binary search tree degenerates into a linear list.

$$height\ h = l_{max} + 1 = N$$

Binary Search Tree :: Access Cost

Maximum average access cost:

$$z_{max} = \frac{1}{N} * \sum_{i=0}^{N-1} (i+1) * 1 = N * \frac{(N+1)}{2N} = \frac{(N+1)}{2} = O(N)$$

Minimum average access cost:

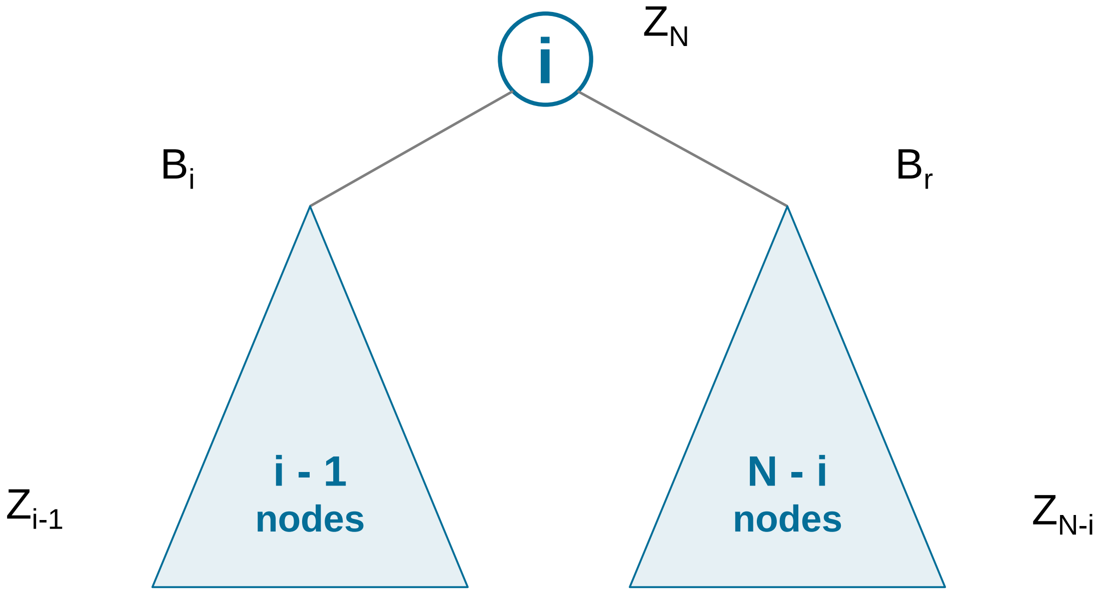
- They can be expected in an **almost complete** or **balanced** tree structure.
- Total number of nodes: $2^{h-1}-1 < N \leq 2^{h}-1$
- height $h = [\log_2 N] + 1$
- **Minimum average access cost**: $z_{min} \approx \log_2 N - 1$

Binary Search Tree :: Access Cost

Average access cost

- Extreme cases of average access costs have only little significance.
- The **difference** between the **average** and the **minimum** access cost is a measure of the **urgency of balancing**.

Determination of the average access cost:



Binary Search Tree :: Access Cost

N different keys with values 1, 2, ..., N are given in random order.

The probability that the first key has the value **i** is $\frac{1}{N}$
(Assumption: same access probability to all nodes)

For the tree with **i** as root we get:

$$Z_N(i) = \frac{1}{N} * \left((Z_{i-1} + 1) * (i - 1) + 1 + (Z_{N-i} + 1) * (N - i)\right)$$

The recursive equation can be represented in non-recursive, closed form by means of the harmonic function:

$$H_N = \sum_{i=1}^{N} \frac{1}{i}$$

Binary Search Tree :: Access Cost

We get:

$$z_N = 2 * \frac{(N+1)}{N} * H_N - 3 = 2ln(N) - c$$

Relative additional cost:

$$\frac{z_N}{z_{min}} = \frac{2ln(N)-c}{\log_2(N)-1} \sim \frac{2\ln(N)-c}{\log_2(N)} = 2\ln(2) = 1{,}386\ldots$$

The **balanced binary search tree** causes the **least cost** for all basic operations.

However, **perfect balancing** at any time is very **expensive**.

Balanced Trees

Efficiency of dictionary operations (**insert**, **search**, **remove**) on trees depends directly on the **tree height**.

Tree with N nodes:

- Minimum height: $\lfloor \mathbf{log_2\ N} \rfloor$, maximum height: ***N-1***
- Access on average $\boldsymbol{O(log_2\ N)}$**,** but in worst case linear.

Aim:

- Fast access with $\boldsymbol{z_{max} \sim O\ (log_2\ N)}$
- Insert and remove operations in logarithmic complexity.

Heuristics:

For each node in the tree, the number of nodes in each of its two subtrees should be kept as constant as possible.

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.

Height-Balanced Search Trees

- **B-Trees**
  always **balanced** due to **balance sustaining search**, **insert**, **remove**
  number of children per inner node between **t** (=m/2) and **2t** (=m)

  access in **O(log(n))**

- **AVL Trees**
  The heights of **each internal node's children** only **differ** by a **maximum of 1**

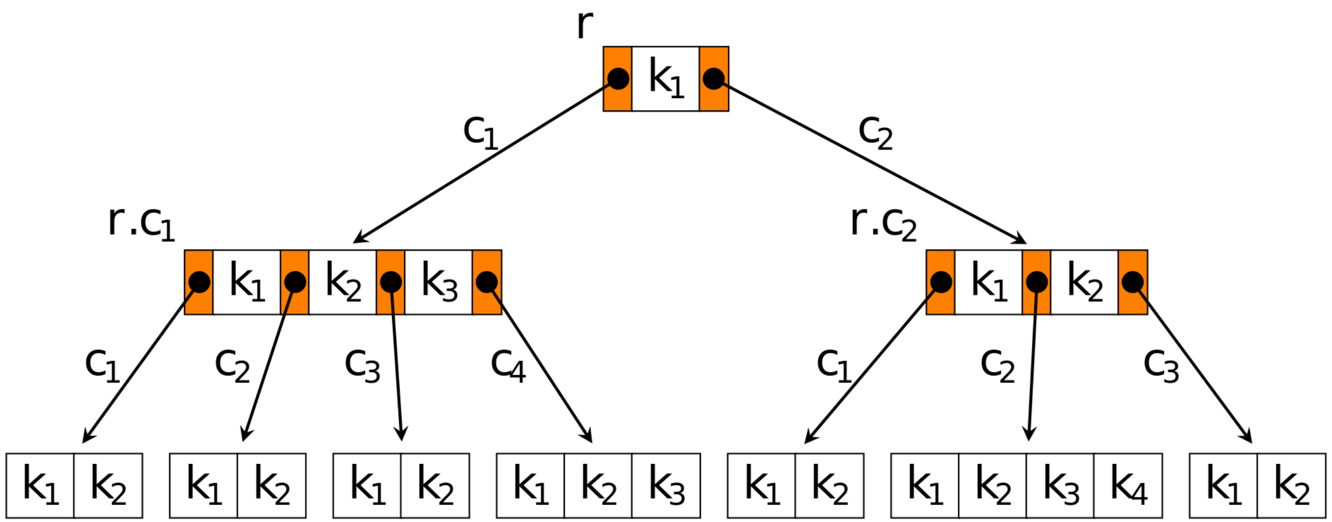
  access in **O(log(n))**

Example from https://de.wikipedia.org/wiki/B-Baum

B-Trees

Tree structure (m-Tree) that is **always balanced**

- **self-balancing** mechanism integrated in access operations (**search**, **insert**, **remove**)
- maintains **sorted** data (**keys**)
- **search**(), **insert**(), **remove**() in **logarithmic time**
- supports "**locality**"-**principle** in memory organization



Graphics: Flying sheep, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19226668

1. Every node has at most ***m*** children (***m*** = 2t).
2. Every non-leaf node (except root) has at least ⌈t = ***m***/2⌉ child nodes.
3. The root has at least 2 children if it is not a leaf node.
4. A non-leaf node with ***k*** children contains ***k***−1 keys.
5. All leaves appear in the same level (height h)

B-Trees

m = 2t ... **maximum** number of children per node
n ... number of keys stored,

note:
in algorithm examples below
(t-1) – (2t-1)

then the **height h** is: $h \leq \log_t \left(\frac{n+1}{2}\right) + 1$ (therefore access in **O(log(n))**)

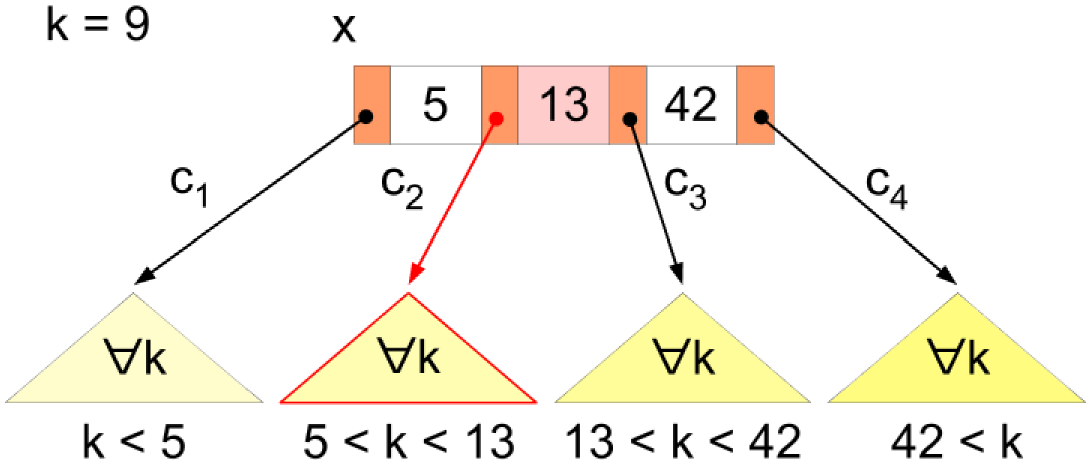
t ... **minimum** number of children per node
n ... **t^h-1**

let **t** = 1024, **h** = 4 then

$1024^4 - 1 = (2^{10})^4 - 1 = 2^{40} - 1 = 1.099.511.627.775$ keys (i.e. access time 10 times lower)

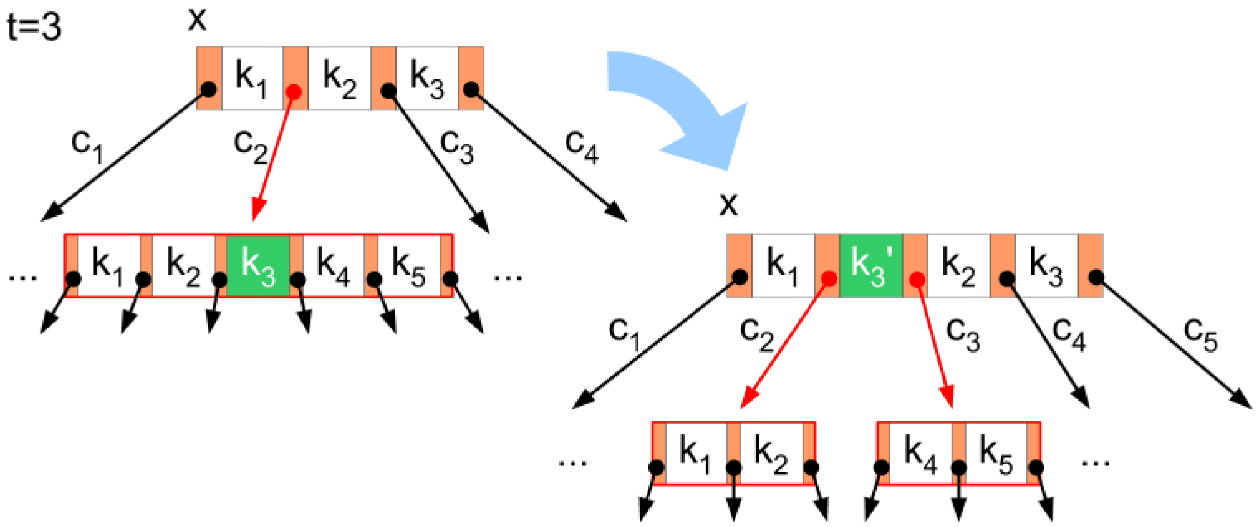
B-Trees

Search (key = 9)



Insert

Search for insert position
(inner node)
If inner node **full**
(i.e more than **2t-1** keys)
then preventively **split**
before stepping down



B-Trees

Remove

Removing (**inner node**) could destroy balance

before stepping down

check whether there are **enough** keys in the **subtree**

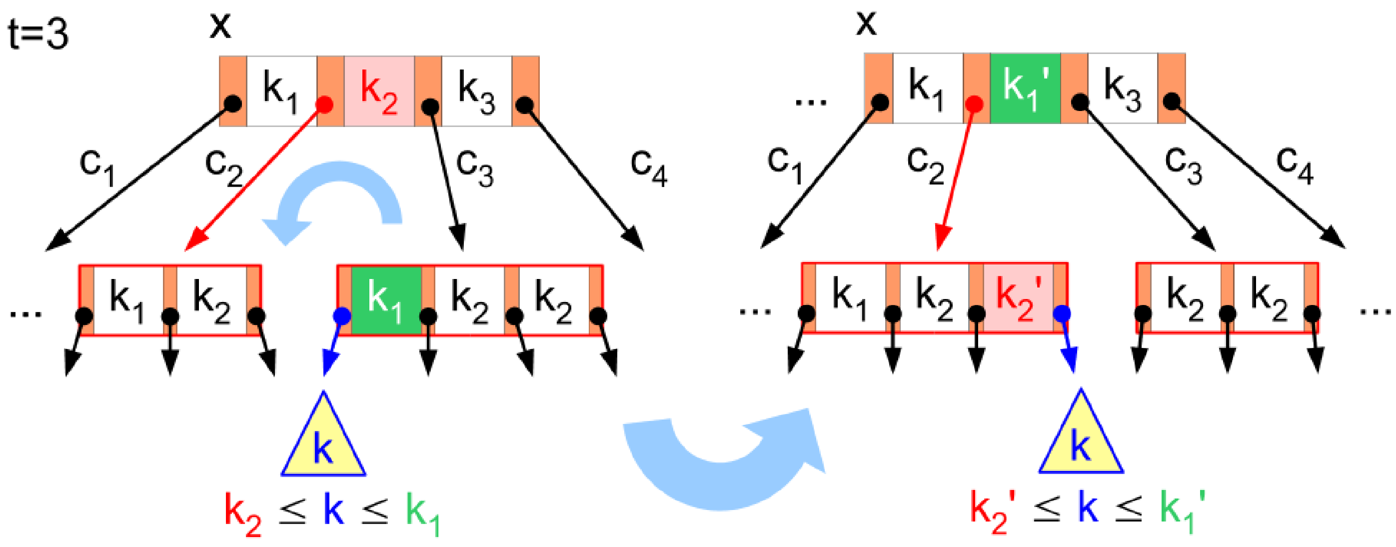
If considered inner node in subtree has just minimum number of keys (**t-1** keys), then

Move / **Merge**

to prevent from imbalance after **Remove**

Move

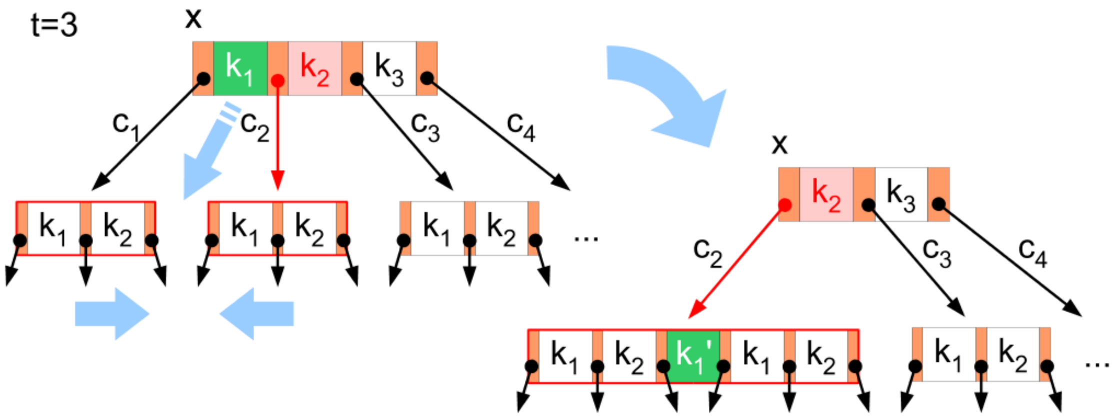
transfer a sibling
to **expand (minimum) inner node**



B-Trees

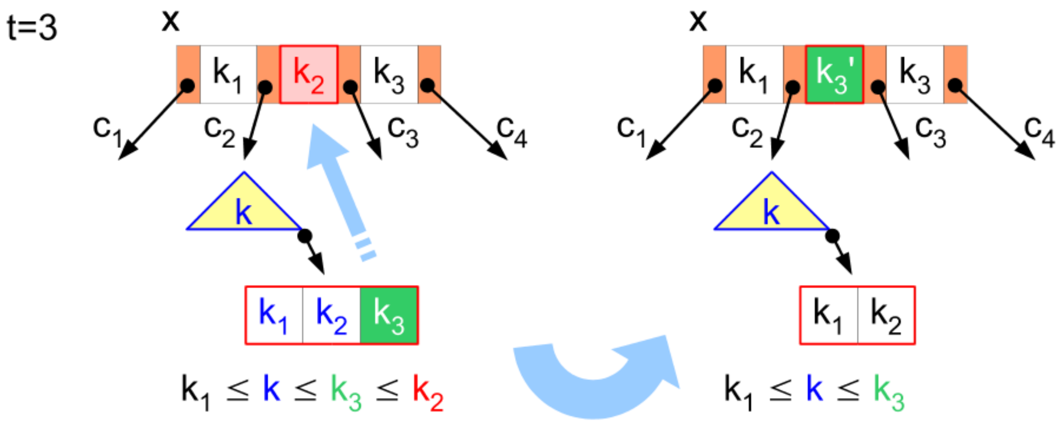
Merge

before stepping down
merge two a siblings
if both (left and right child)
are at **minimum number of inner nodes**



Remove

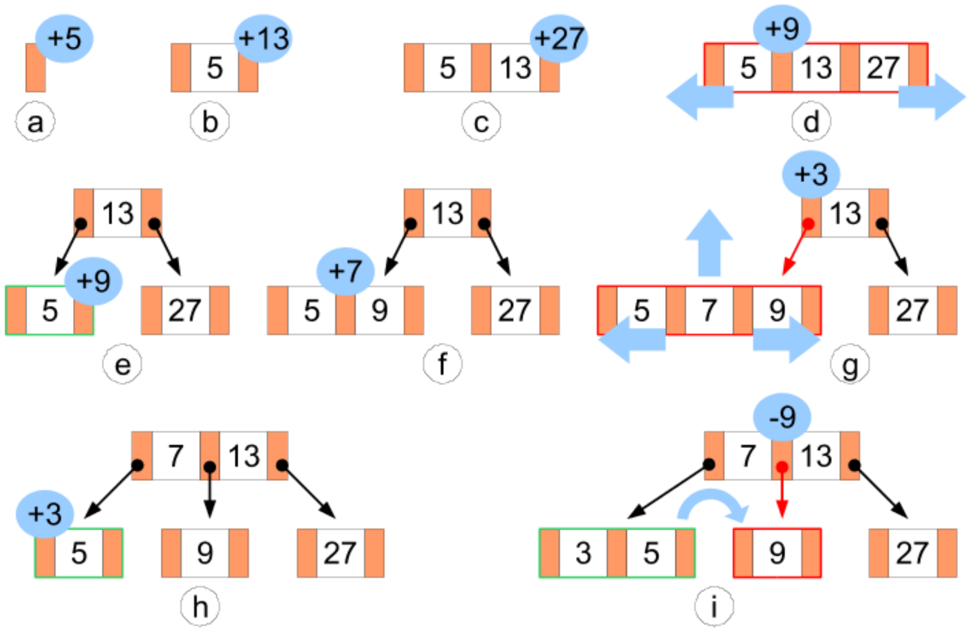
before deleting key from inner node
separate/adjust the **key ranges**
of both left and right child
(here: inorder transfer,
i.e. rightmost of left child)

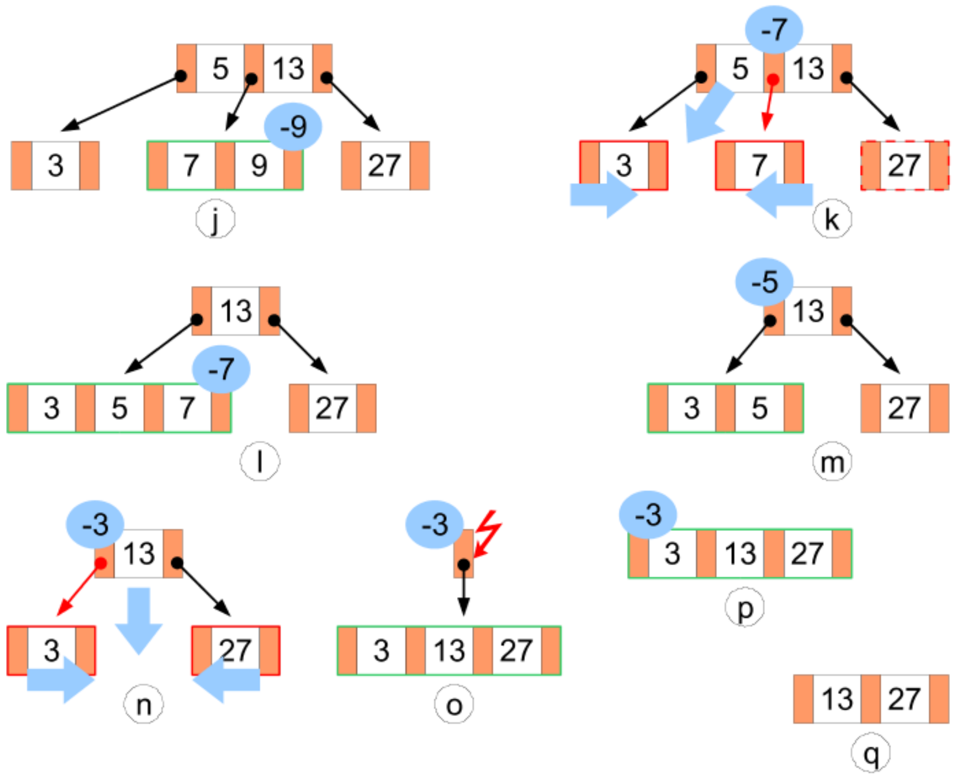


B-Trees

Example (**t=2**) i.e. min 1, max 3 keys per inner node

a–c)	**insert** 5, 13, 27
d–e)	**insert** 9 – induces **splitting** of root
f)	**insert** 7 leaf node
g–h)	**insert** 3 needs **splitting**
i–j)	**remove** 9, needs induces **move** of sibling
k–l)	**remove** 7 induces **merge**
m)	**remove** 5 (**leaf**)
n–q)	**remove** 3 induces **merge** of two children of root empty root replaced by only child





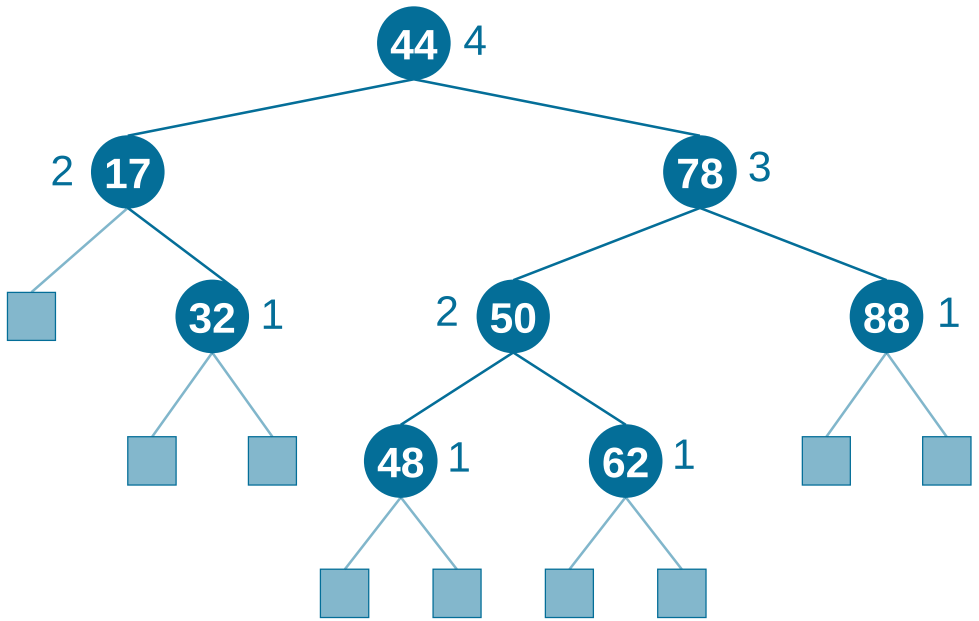
Example from https://de.wikipedia.org/wiki/B-Baum; Graphics: Haui, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=250188

AVL Trees (Adelson-Velskii, Landis 1962)

AVL trees are **balanced binary search trees**

An AVL tree is a binary search tree, in which the heights of **each internal node's children** only **differ** by a **maximum of 1**.

Example (numbers next to the nodes indicate their height):



Height of an AVL tree

Assertion: The **height** of an AVL tree T that stores **n keys** is **O(log n)**

Sketch of a proof:
Find n(h), the smallest possible number of internal nodes in an AVL tree of height h.

- Trivial: n(1) = 1, n(2) = 2

Height of an AVL tree

Assertion: The height of an AVL tree T that stores n keys is O(log n)

Sketch of a proof:

Find n(h), the smallest possible number of internal nodes in an AVL tree of height h.

- Trivial: $n(1) = 1$, $n(2) = 2$
- $n \geq 3$: The AVL tree of height h with minimum n(h) consists of a root node, an AVL subtree of height h-1 and an AVL subtree of height h-2.

i.e. $n(h) = 1 + n(h-1) + n(h-2)$

Fibonacci progression (exponential!)

as $n(h-1) > n(h-2)$ it follows:

$n(h) > 2\, n(h-2)$

$n(h) > 4\, n(h-4)$

$n(h) > 8\, n(h-6)$

...

$n(h) > 2^i\, n(h-2i)$ then $n(h) \geq 2^{h/2-1}$

choose $i = \lceil h/2 \rceil - 1$

- findElement: O(logn)
- findAllElements O(log n+s) (s=number of findings)

Using the logarithm we get: $h < 2 \log n(h) + 2$

Therefore we have: The height of an AVL tree is O(log n)

$\log n(h) > h/2 - 1$

Insert in AVL trees

By **inserting** a node into an AVL tree, the **height** of some nodes in this tree changes

Insert operation may cause the AVL tree to be **unbalanced**

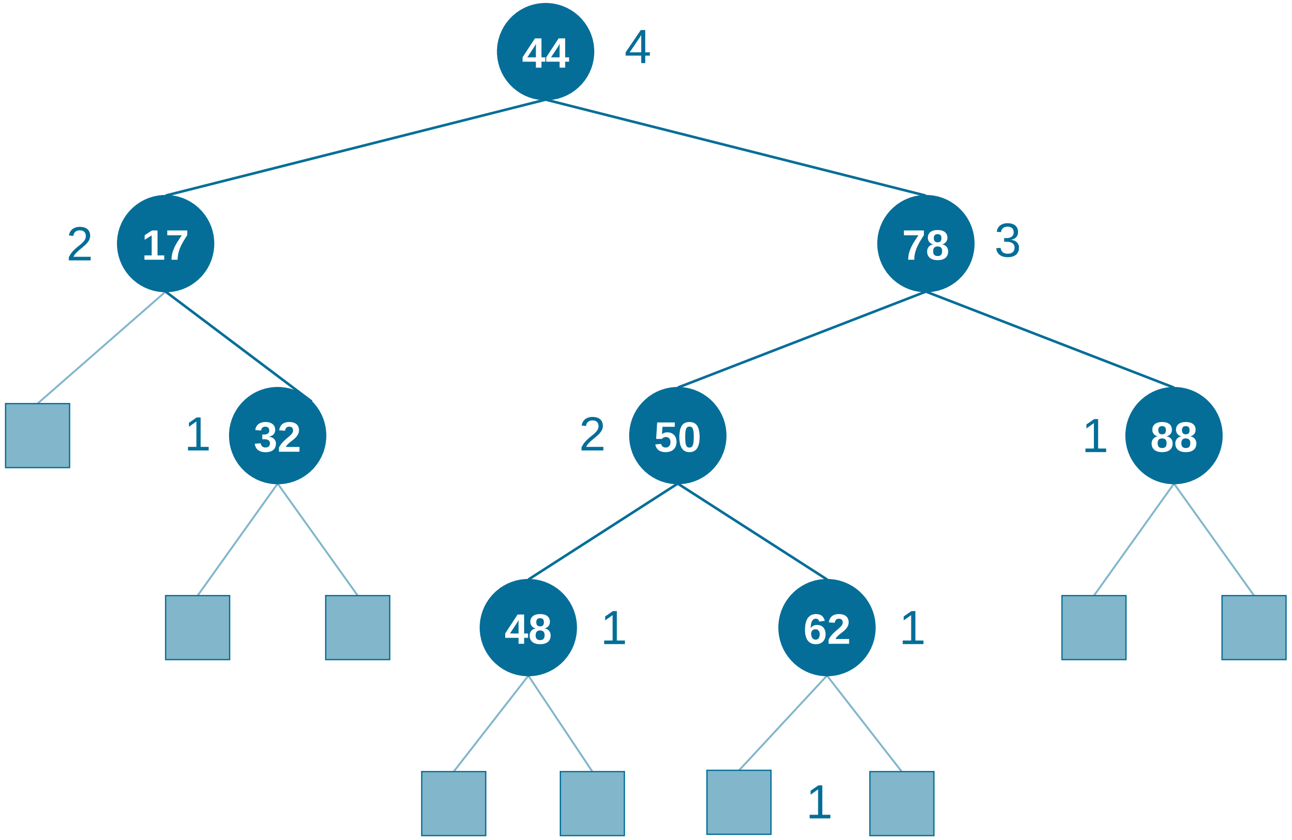
- Go up **from the new node** in the tree until the first node **x** is found, whose "grandfather" **z** is an unbalanced node.
- Since **z** was unbalanced by inserting a node that lies in a sub-tree with root **y** (where y is a child of **z**), we have height(y) = height(sibling(y)) + 2

Re-balancing of the sub-tree with root z requires **restructuring**:

- **x**, **y** and **z** are renamed to **a**, **b**, **c** (according to **in-order traversing**)
- **z** is replaced by **b**, whose children are now **a** and **c**, whose children are again **four subtrees**, which were formerly the children of x, y and z

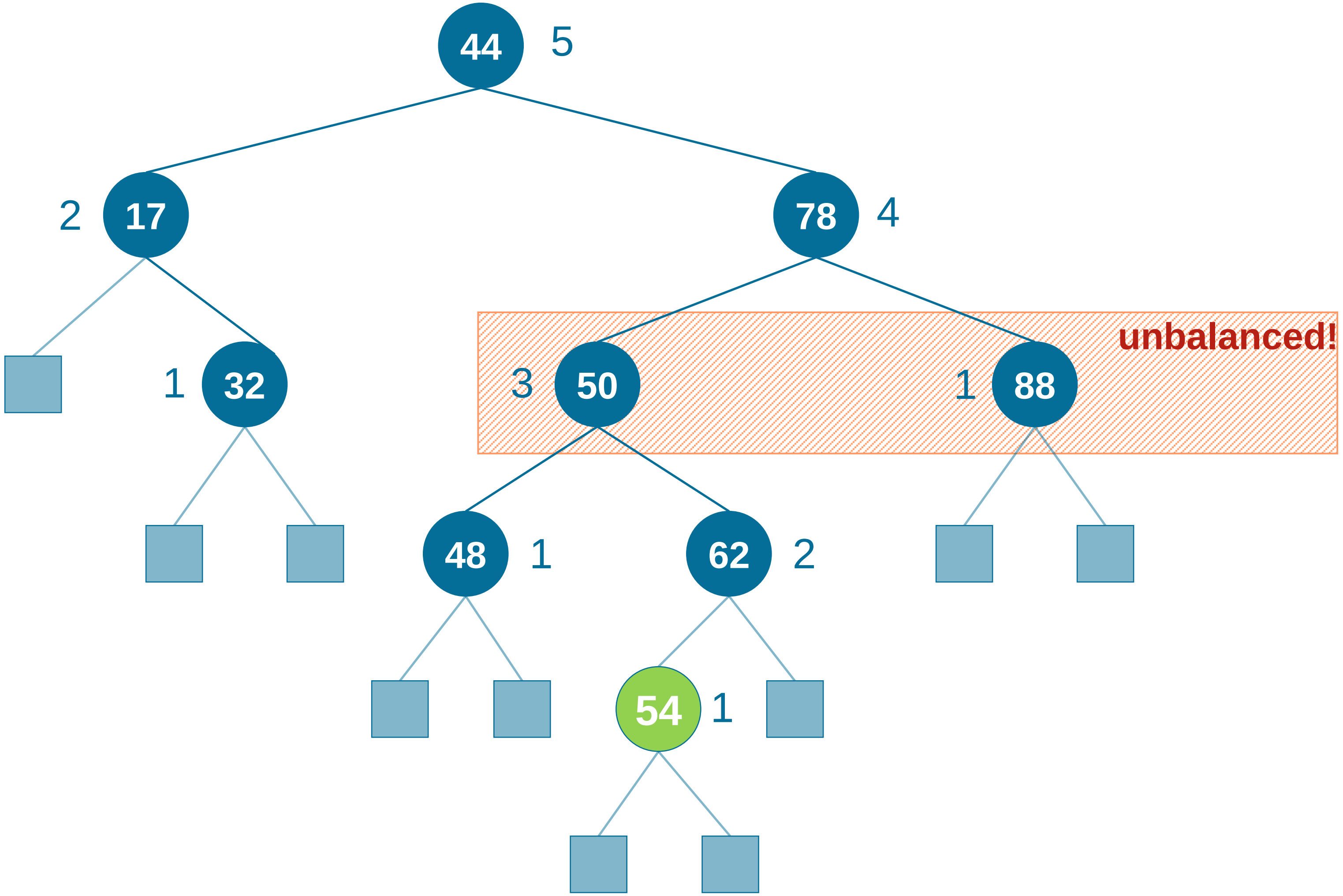
Insert in AVL trees

Insert 54



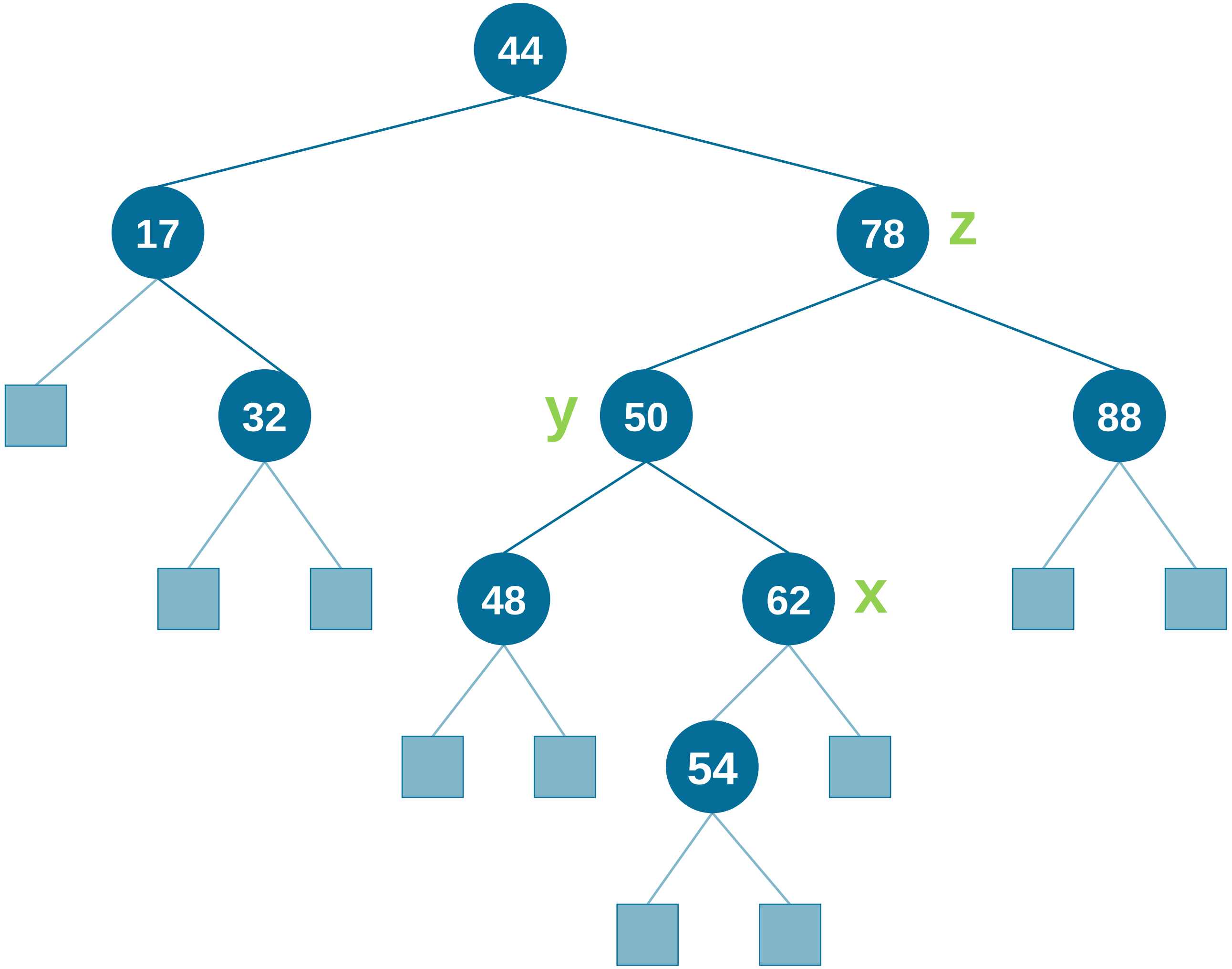
Insert in AVL trees

After inserting 54:



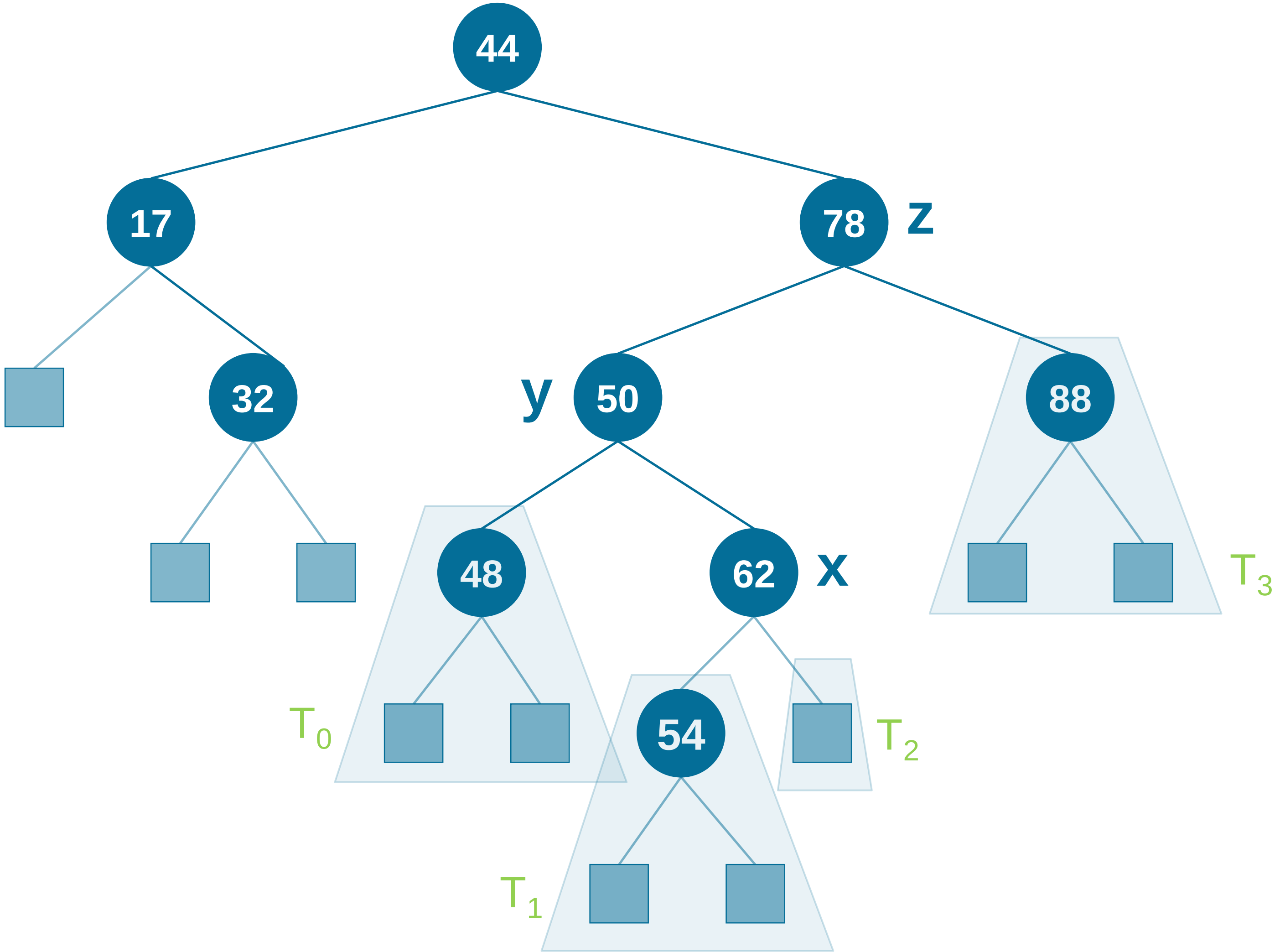
Insert in AVL trees

After inserting 54:



Insert in AVL trees

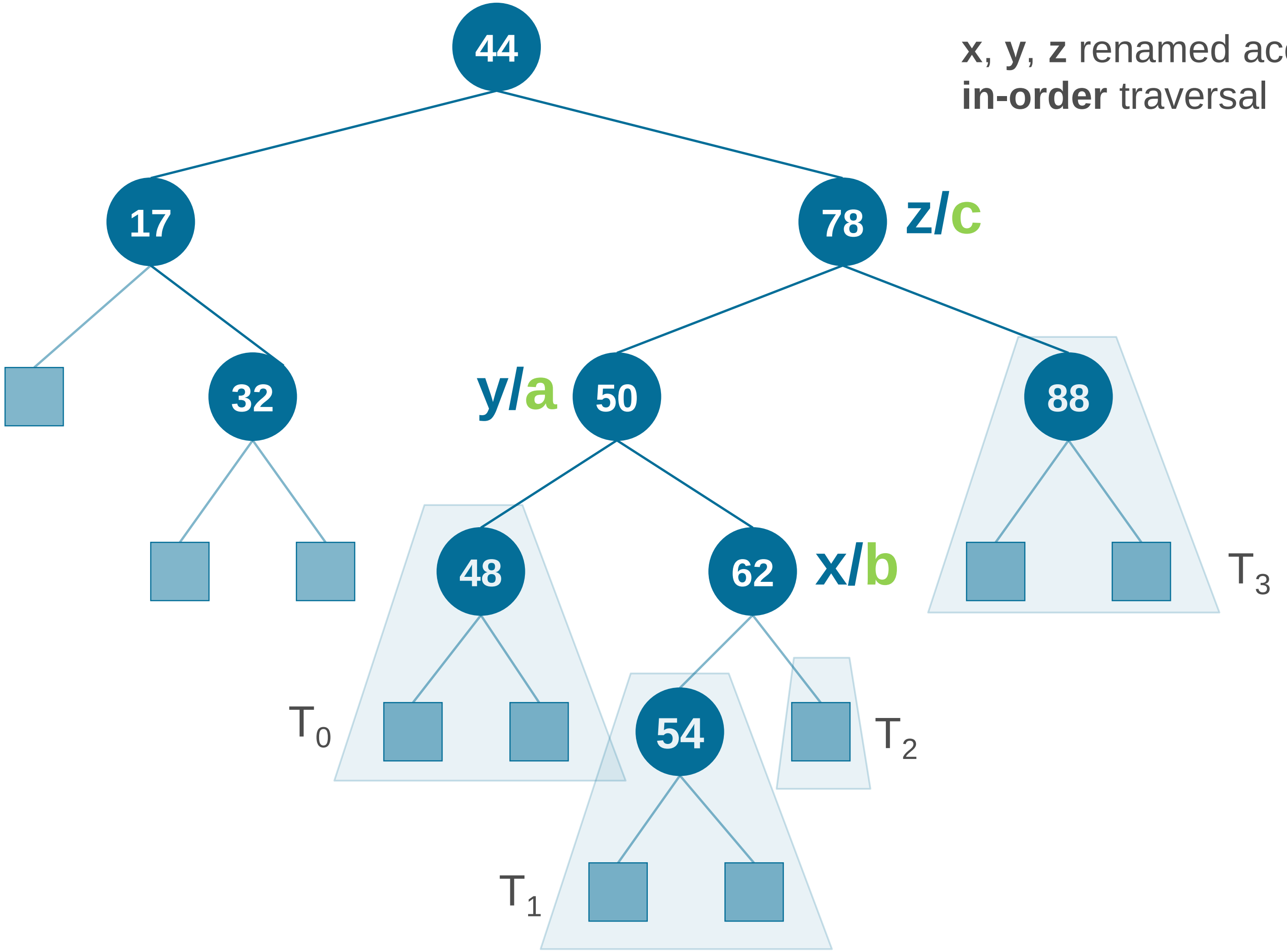
After inserting 54:



Insert in AVL trees

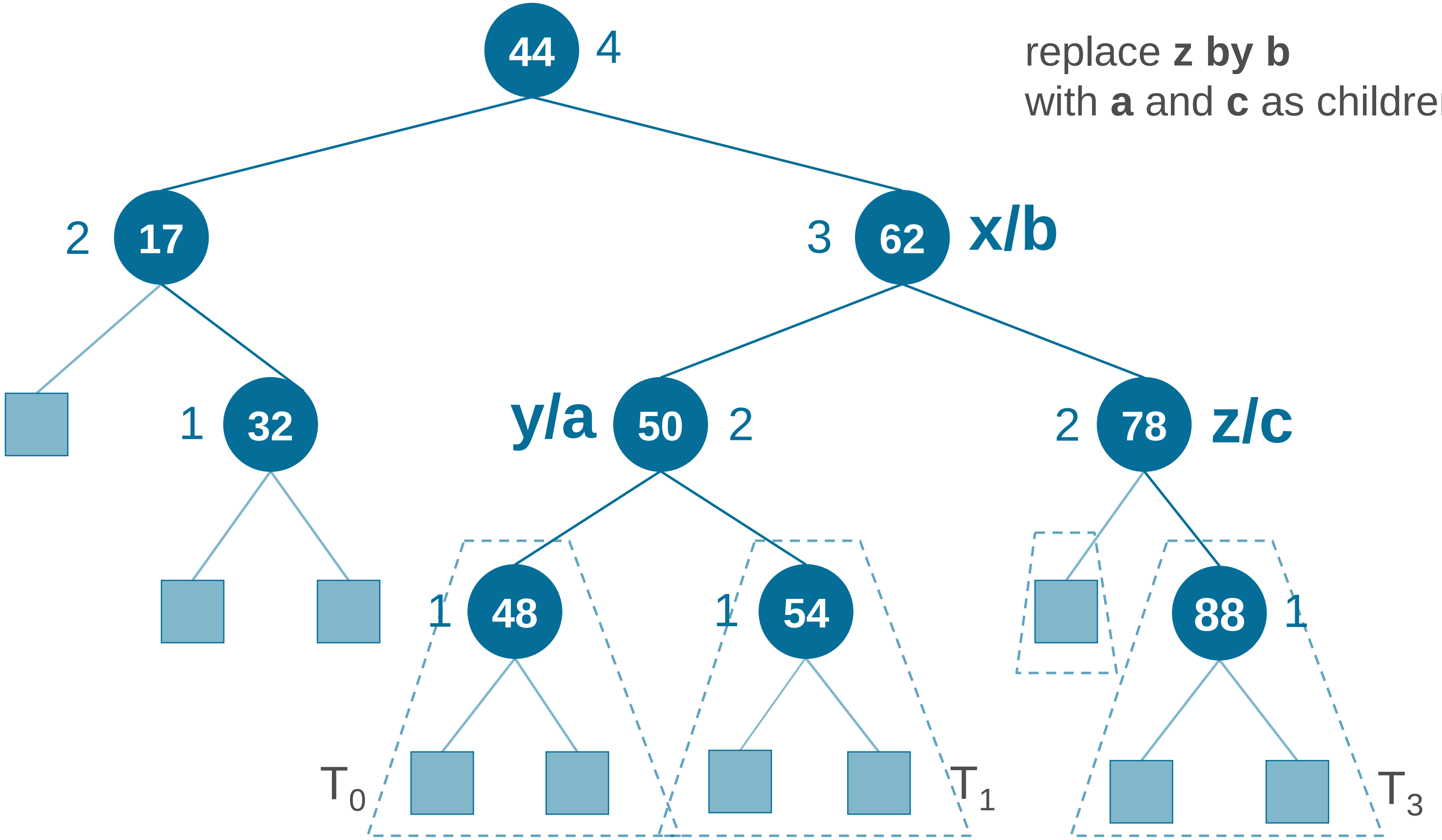
After inserting 54:

x, **y**, **z** renamed according to **in-order** traversal



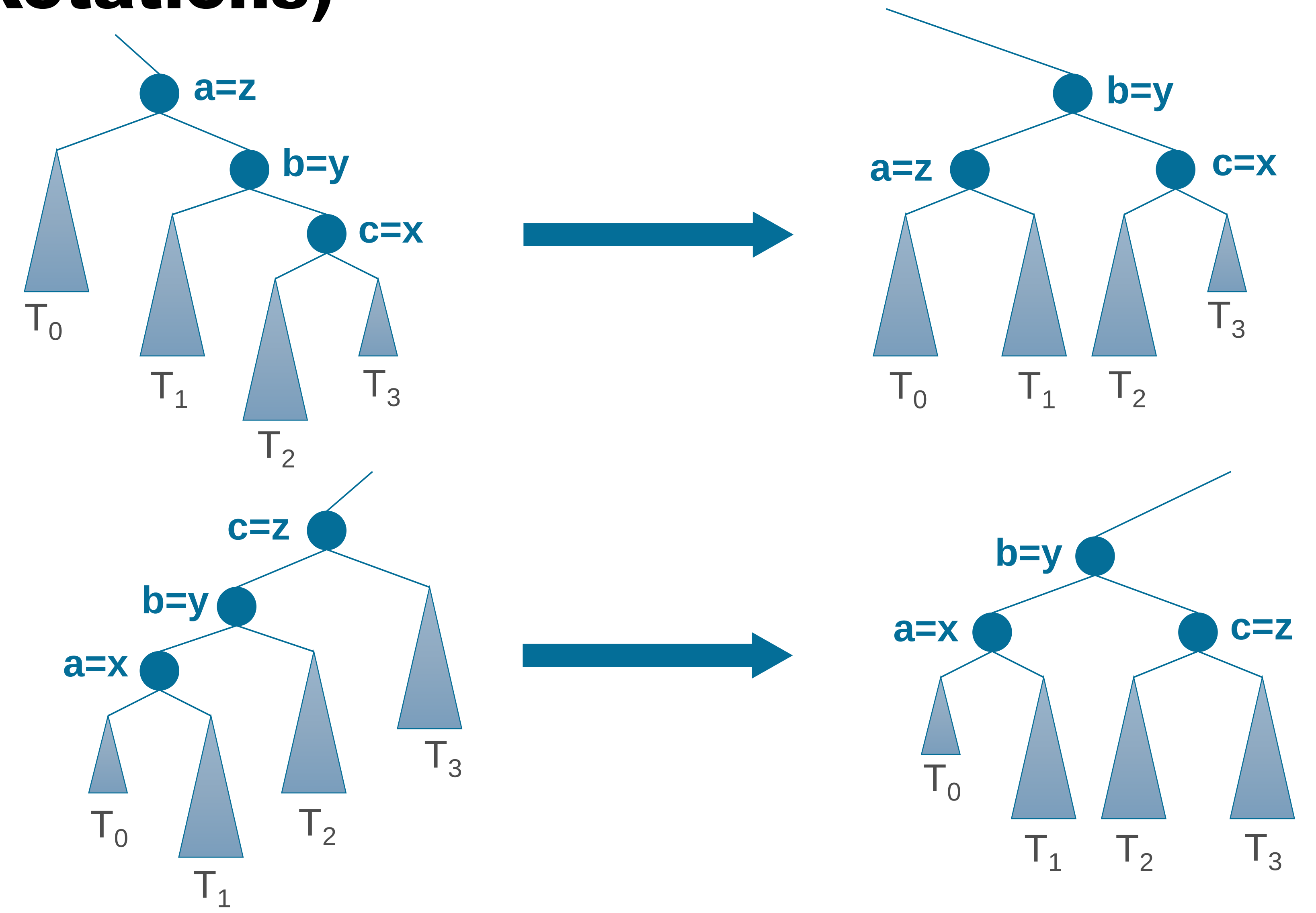
Insert in AVL trees

After inserting 54:

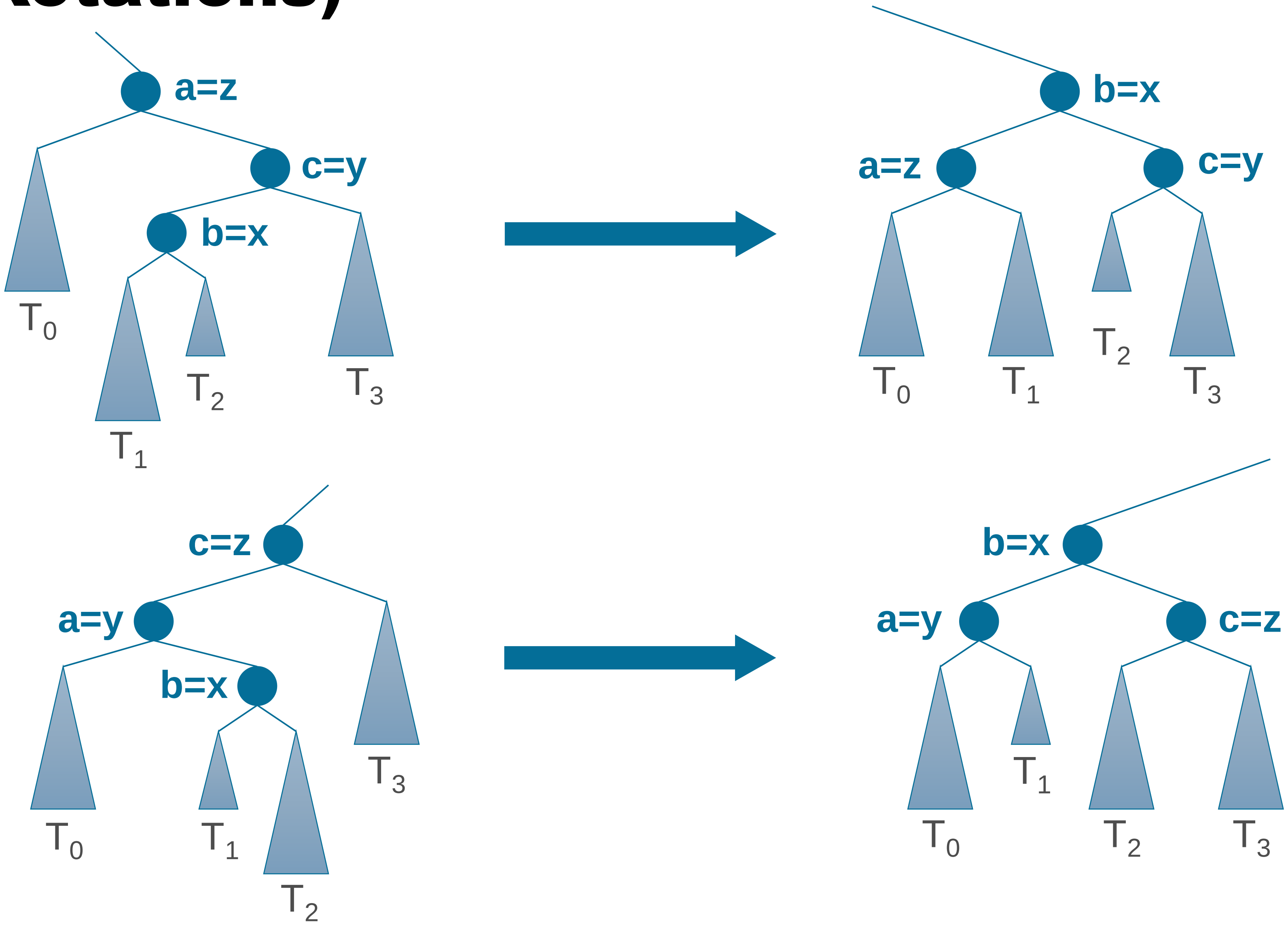


replace **z by b**
with **a** and **c** as children

Restructuring in AVL trees (Single Rotations)



Restructuring in AVL trees (Double Rotations)



Algorithm for Restructuring

Algorithm restructure (x):

Input:
node x of a **binary tree T** with **parent y** and **grandparent z**
Output:
tree **T restructured** by (single or double) **rotation** of **nodes x, y, z**

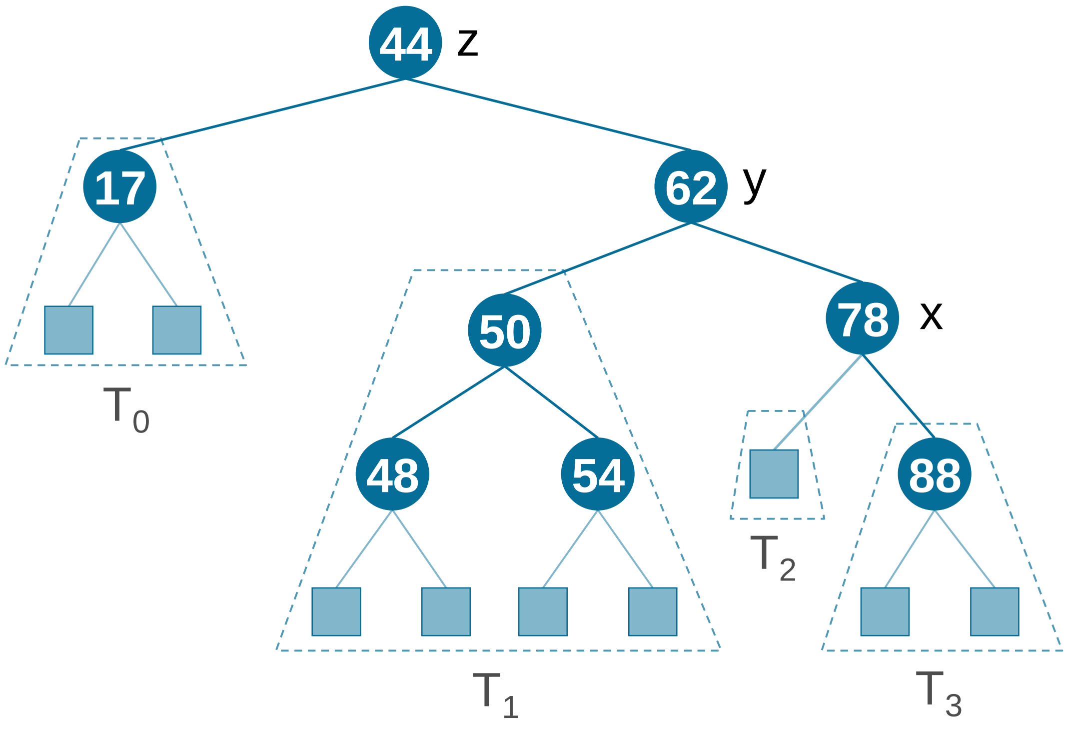
1. Let (**a**,**b**,**c**) be the **in-order-sequence** of the nodes **x, y, z** and let ($\mathbf{T_0}$, $\mathbf{T_1}$, $\mathbf{T_2}$, $\mathbf{T_3}$) be the in-order-sequence of the **subtrees** of x, y and z
2. **Replace** subtree with root z by subtree with root **b**
3. Assign **a as left child of b** and assign T_0 and T_1 as left/right **subtree** of **a**
4. Assign **c as right child of b** and T_2, T_3 as left/right **subtree** of **c**

Institute of Pervasive Computing

Cut/Link Restructuring Algorithm

Each tree to be balanced can be divided into **7 parts**:

- nodes **x, y, z** and
- **4 subtrees (T_0, T_1, T_2, T_3)**,
  with the children of x, y, z as root

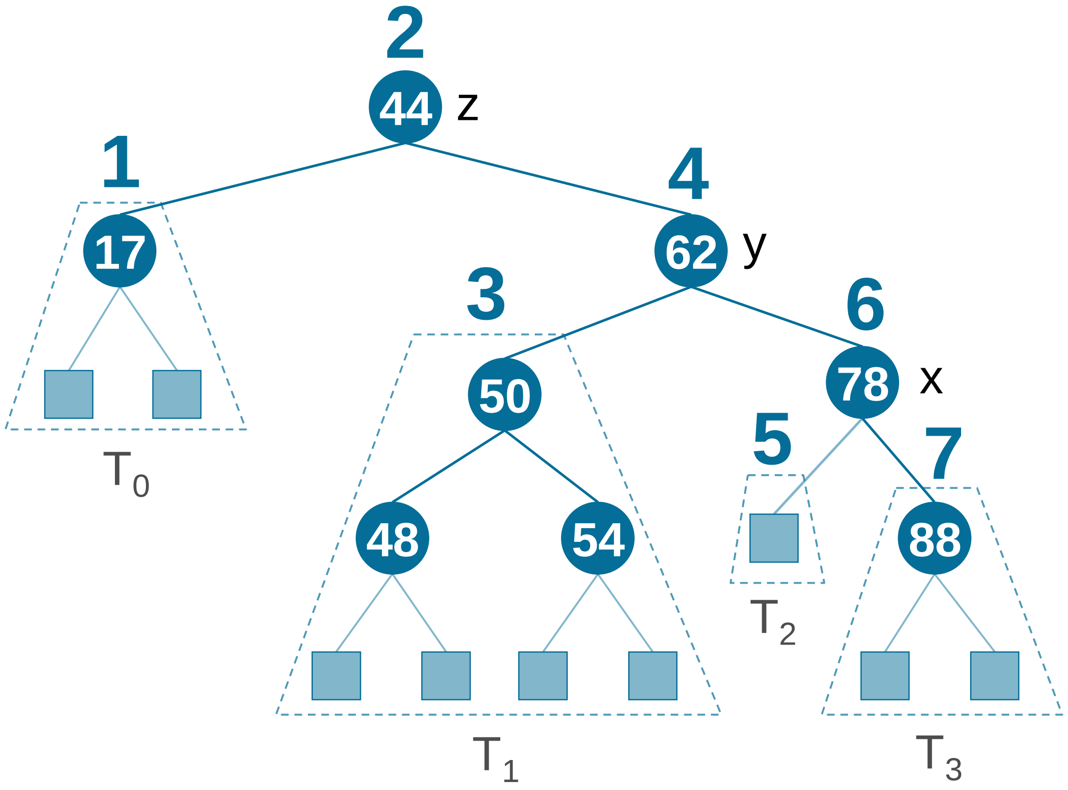


Create a **new tree** from the 7 parts, which is balanced
and in which the **in-order sequence** of the parts is retained.

Cut/Link Restructuring Algorithm

Example

number 7 parts according to in-order traversal



Cut/Link Restructuring Algorithm

Example

Create an array with indices 1..7

1	2	3	4	5	6	7

"Cut" the 4 subtrees and place them into the array according to their numbering.

T_0		T_1		T_2		T_3
1	2	3	4	5	6	7

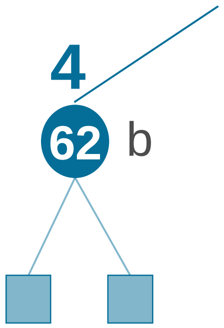
"Cut" x, y and z (in the order child, parent, grandparent) and put them in the array according to their numbering

T_0	z=a	T_1	y=b	T_2	x=c	T_3
1	2	3	4	5	6	7

Cut/Link Restructuring Algorithm

Example

- Reassemble the tree again
- Set element
  at position 4 (b) as root

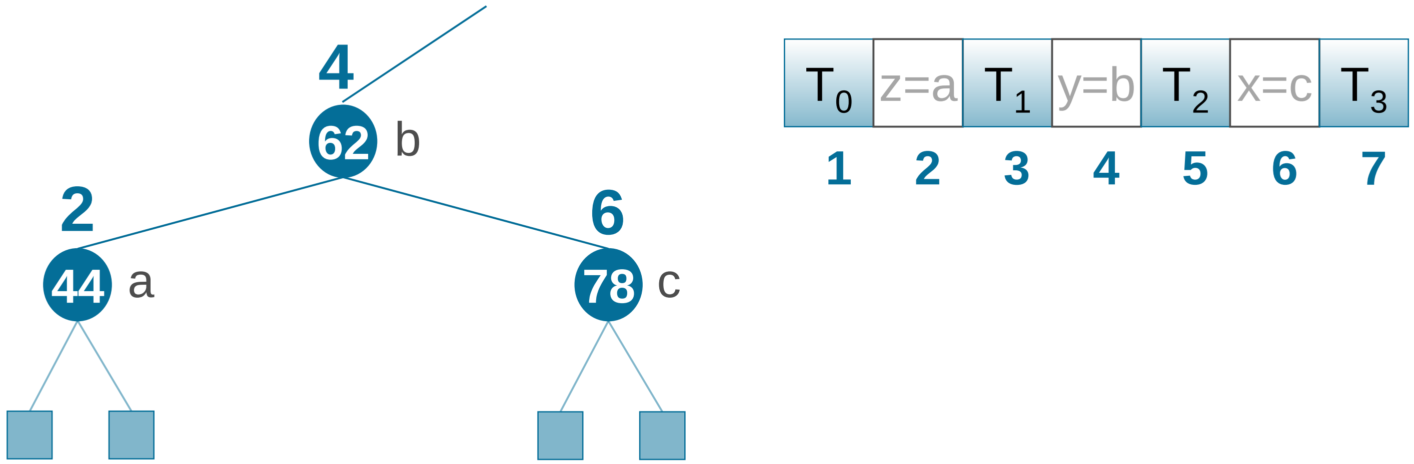


T_0	z=a	T_1	y=b	T_2	x=c	T_3
1	2	3	4	5	6	7

Cut/Link Restructuring Algorithm

Example

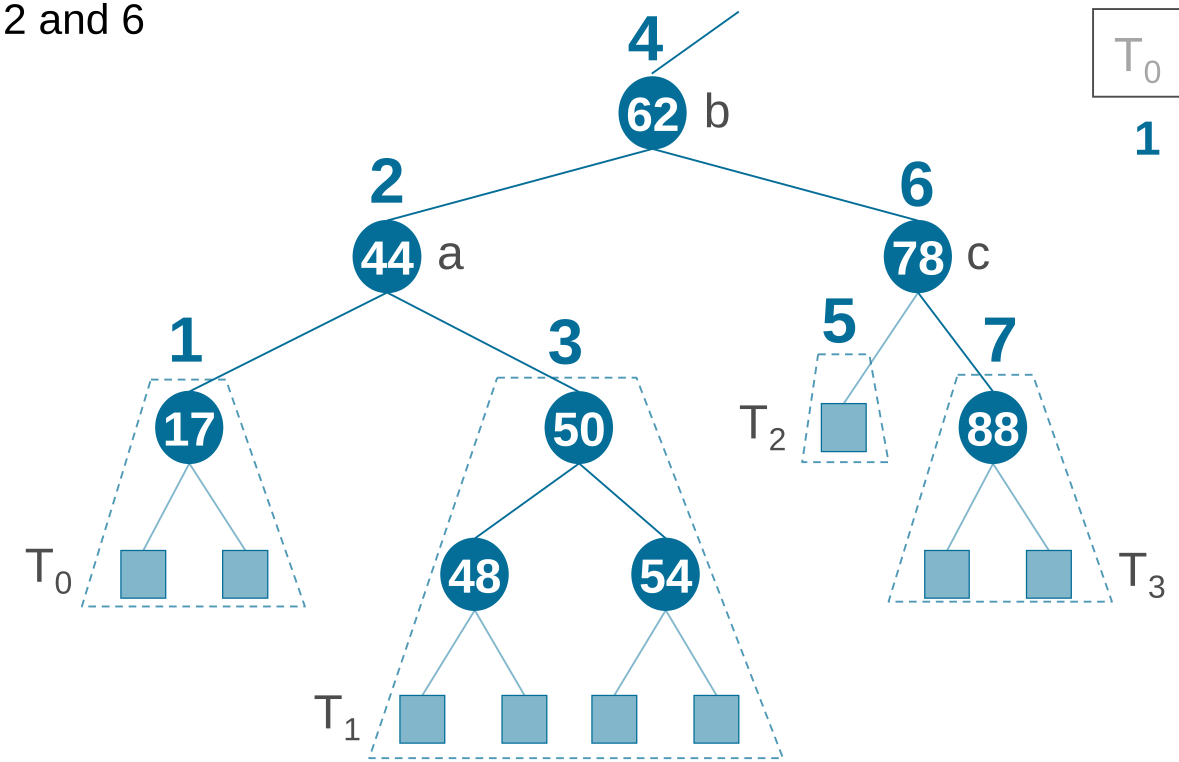
- Reassemble the tree again
- Set elements
  at position 2 and 6 as children



Cut/Link Restructuring Algorithm

Example

- Reassemble the tree again
- Set elements at position
  1, 3 or 5, 7 as child of 2 and 6



T_0	z=a	T_1	y=b	T_2	x=c	T_3
1	2	3	4	5	6	7

Cut/Link Restructuring Algorithm

Cut/Link restructuring algorithm has the **same effects** as the **four rotation cases** previously considered

Advantage:

- **No case distinction** necessary
- More „**elegant**“ solution

Disadvantage:

- May require more code

The two procedures **do not differ** in terms of **complexity** (**runtime**).

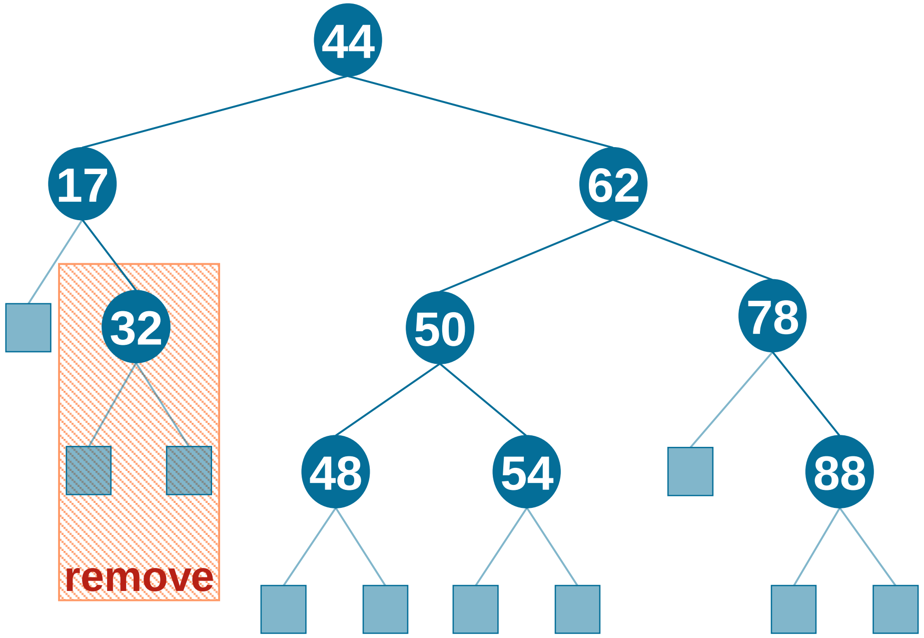
Removal of Nodes

Using **removeAboveExternal(w)** can unbalance a tree:

- Let **z be the first unbalanced node** which is visited while traversing **up** in the tree.
- Let **y** be the **child of z** with the **largest height** and let **x** be the **child of y** with **largest height**.
- The algorithm **restructure(x)** can be used to restructure and balance the subtree with root *z*.
- However, restructuring can **destroy the balance at higher levels**, so that the verification (and restructuring if necessary) must be **continued** until the root node is reached.

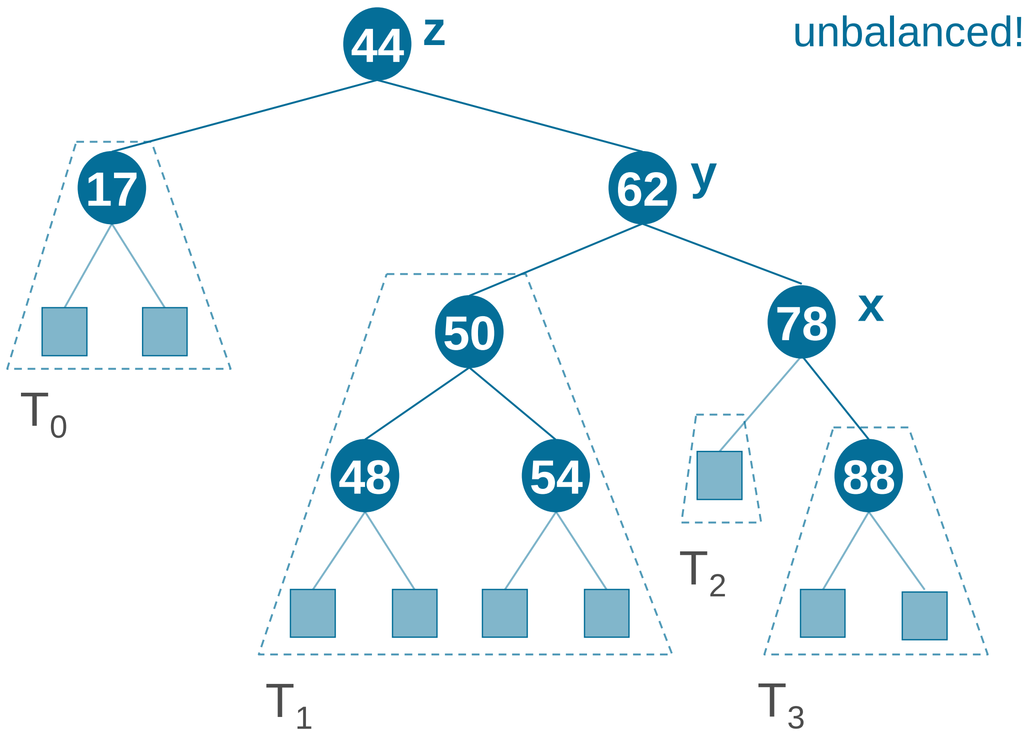
Removal of Nodes

Example



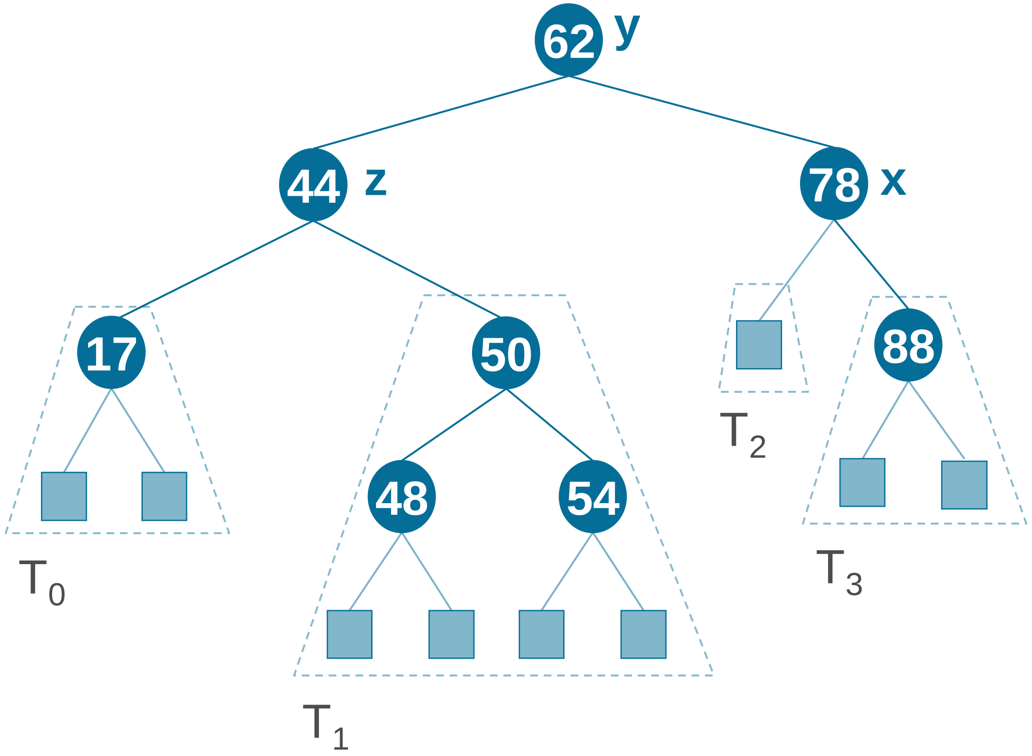
Removal of Nodes

Example



Removal of Nodes

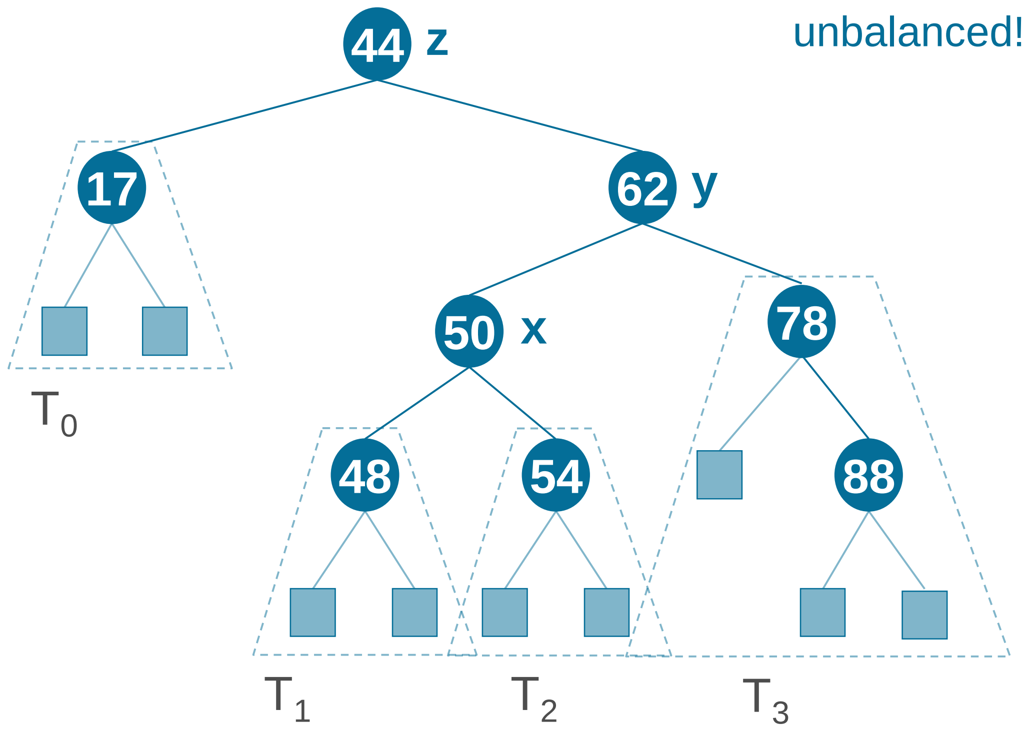
Example



Removal of Nodes

Example

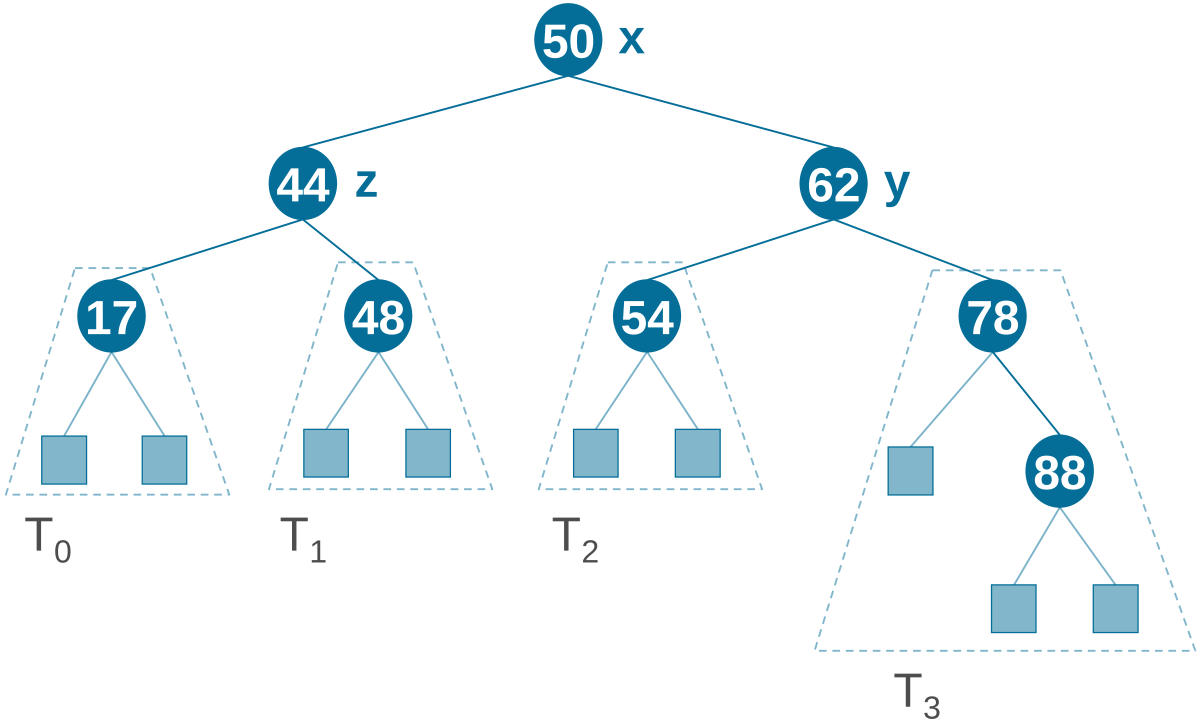
Alternative



Removal of Nodes

Example

Alternative



AVL Trees :: Complexity

Possible complexity: N = number of keys

`size(), isEmpty()`	$\Theta(1)$
`get(), insert(), remove()`	$O(\log N)$
`getAll(), removeAll()`	$O(\log N + s)$

(s = Number of elements in respective **`enumeration`**)

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
  - $Fib(h) = 1/\sqrt{5} * (((1 + \sqrt{5})/2)^h - ((1 - \sqrt{5})/2)^h)$
  - for all h we have: $Fib(h) \geq 1/\sqrt{5} * ((1 + \sqrt{5})/2)^h – 1$
  - 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

Trees (Height Balanced)

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

Institute of Pervasive Computing

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