Trees for Sets

Christian Osendorfer

Motivation

- somehow store a set of data items in the computer
- each data item i is indexed by a unique value key(i) from a totally ordered set (universe), so data items are comparable
- the following operations should be provided:
- **find(k)**: given a key k, return the corresponding data item (if it is in the set)
- **insert(u)**: insert the data item u into the set
- **delete(k)**: delete the data item with key k from the set
- *dictionary operations*
- data item and its key are often used interchangeably

Terminology

- rooted tree
- binary tree: each node has at most two children
- extended binary tree: difference between internal nodes and external nodes(leaves); each internal node has exactly two children; a binary tree with n internal nodes has $n + 1$ leaves
- for an extended binary tree T, $|T|$ denotes the number of its *internal* nodes
- longest path from the root to a leaf is called the *height* of the tree

- binary search tree
- nice theoretical property: average height of a randomly *build (by insertions only)* binary search tree on n distinct nodes is $O(\lg n)$.
- tree might degenerate into a linear list
- considering all (structural) possible trees with n internal nodes, the average distance of a node from the root is $O(\sqrt{n})$.

AVL trees

- *Adelson-Velsky* and *Landis*: A binary search tree is an *AVL*-tree, if the height of the left subtree of every internal node never differs more than ± 1 from the height of the its right subtree
- only internal nodes carry information (i.e. keys ...)

Theorem A *The height of an AVL tree with n internal nodes always lies between* $\lg(n+1)$ *and* $1.44\ldots \cdot \lg(n+2)$

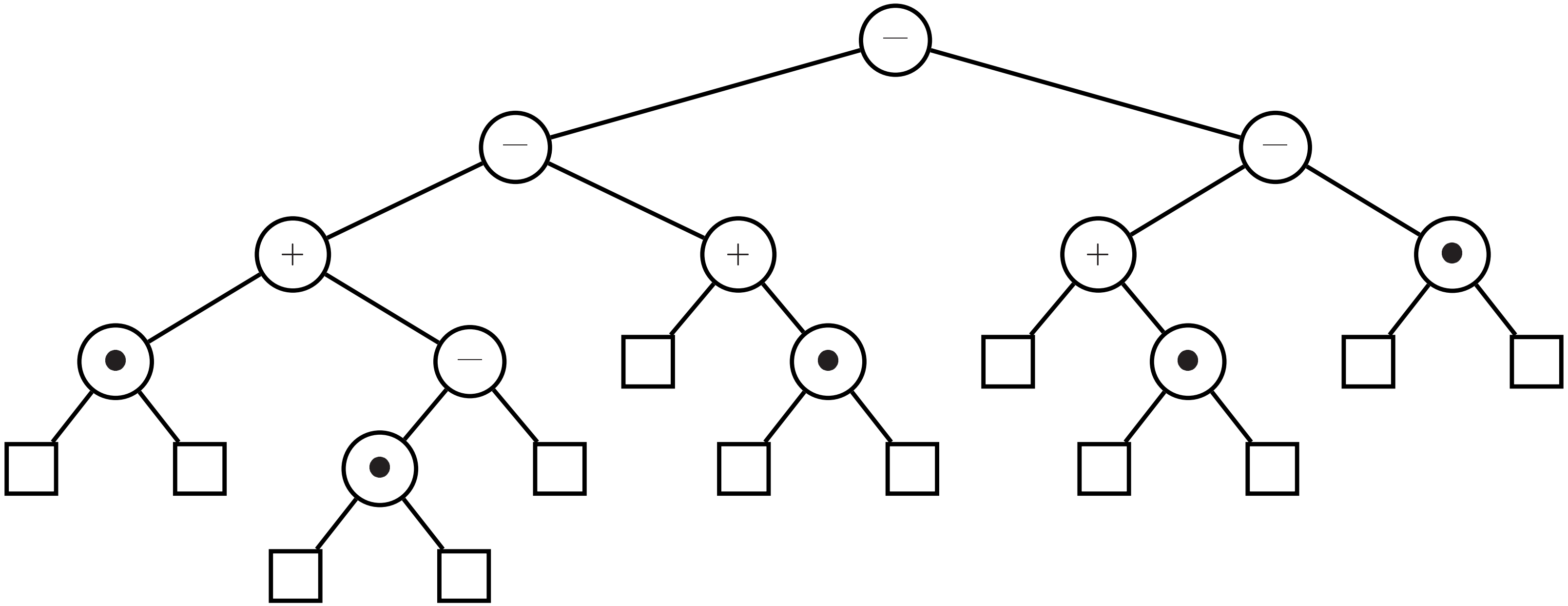
- a binary tree with height h can not have more than 2^h external nodes, so $n+1 \leq 2^h$, or: $\lg(n+1) \leq h$.

- consider an AVL-tree of height h with minimum number of internal nodes (denoted by T_h)
- by the structure of an AVL-tree we get:

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1$$

(with $T_0 = 0$ and $T_1 = 1$)

- $g_h := T_h + 1$ results in: $g_h = g_{h-1} + g_{h-2}$, (and $g_0 = 1$, $g_1 = 2$)
- $n \geq T_h \geq \phi^{h+2}/\sqrt{5} - 2$ where $\phi = (1+\sqrt{5})/2$
- $\log_\phi (n+2) \geq h + \underbrace{2 - (\log_\phi 5)/2}_{\geq 0}$
- $h \leq \log_\phi (n+2) = \underbrace{\log_\phi 2}_{=1.440\ldots} \lg (n+2)$



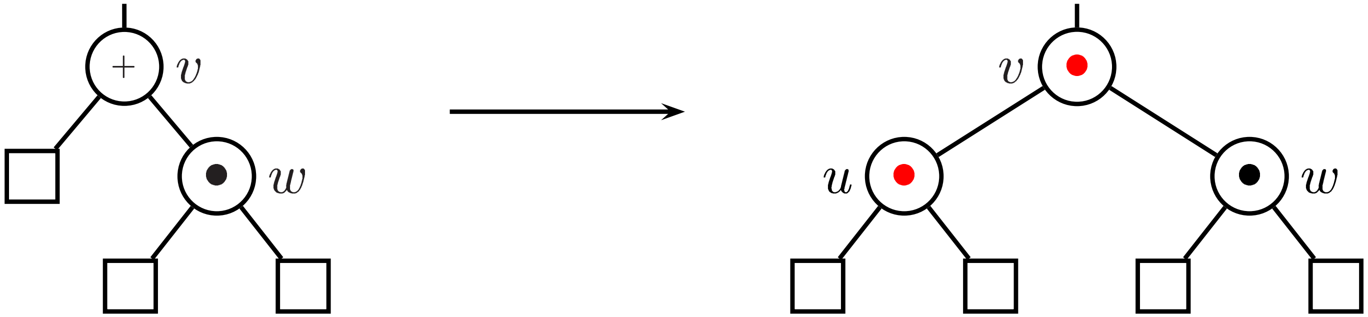
- with each node, we store its *balancefactor*, that is -1, 0 or 1, depending on the difference in heights of its right and left subtree
- the balancefactor will help us to maintain the AVL-property of the tree during insertion and deletion of elements

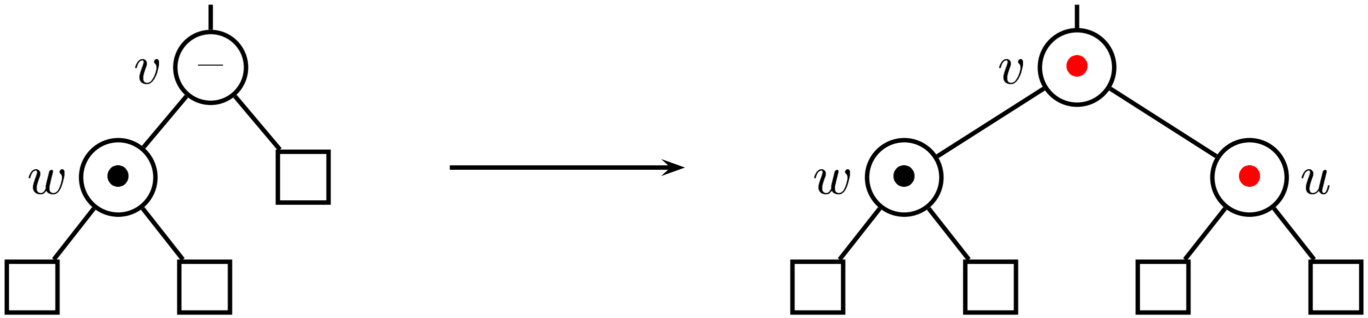
Analysis of dictionary operations for AVL trees

- **find(u)**: by Theorem A, it takes $O(\lg n)$ time to find an element in an AVL-tree with n nodes

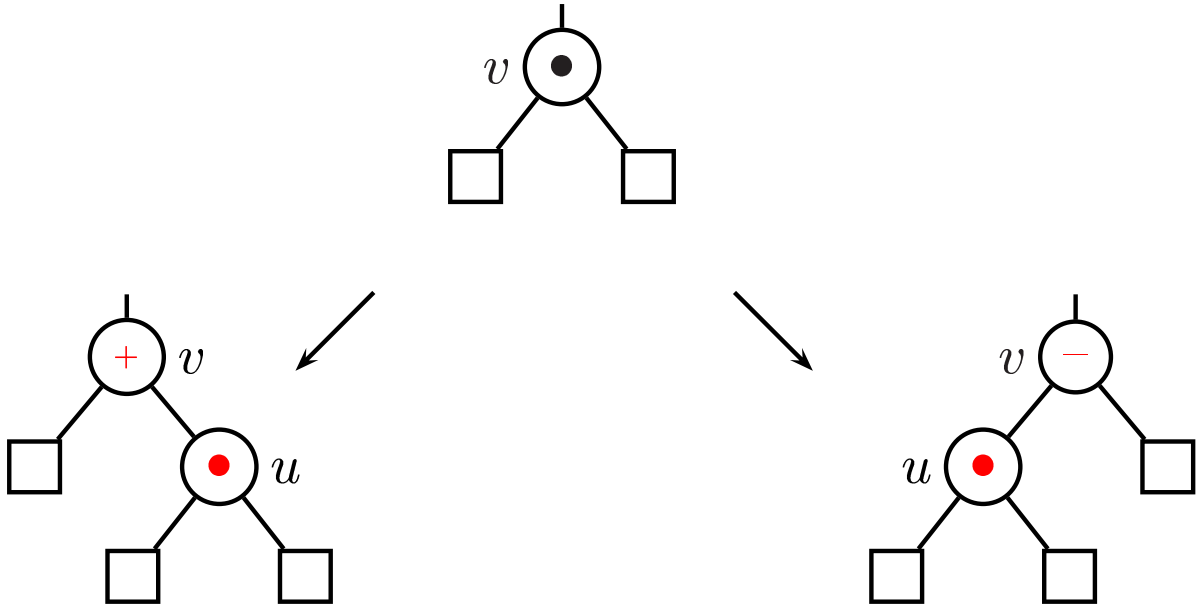
- **insert(u)**: obviously, if we insert an element u into an AVL-tree, the initial tree might get out of balance, so additional rebalancing might be necessary

- with find(u), we get the leaf where the new element u is to be inserted (unless it is already in the tree)

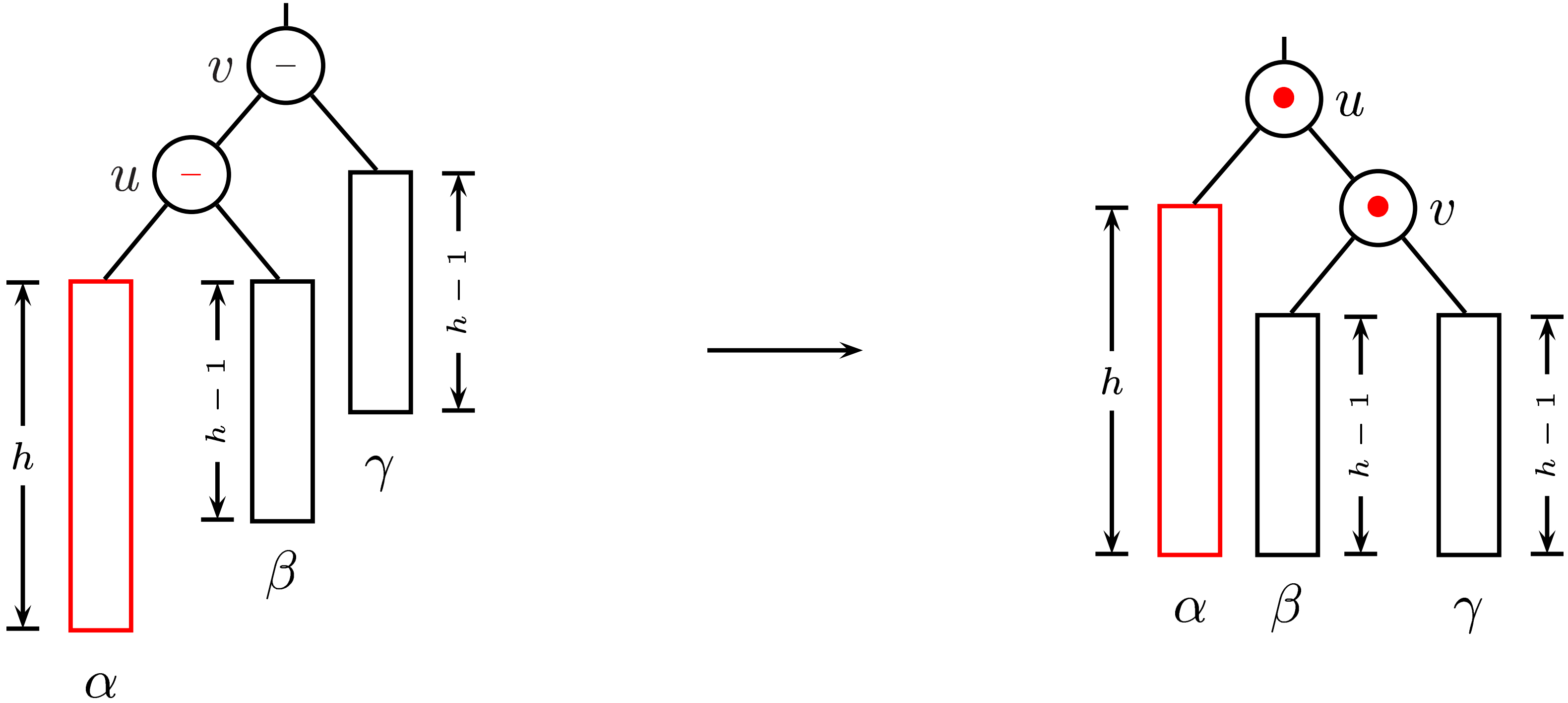




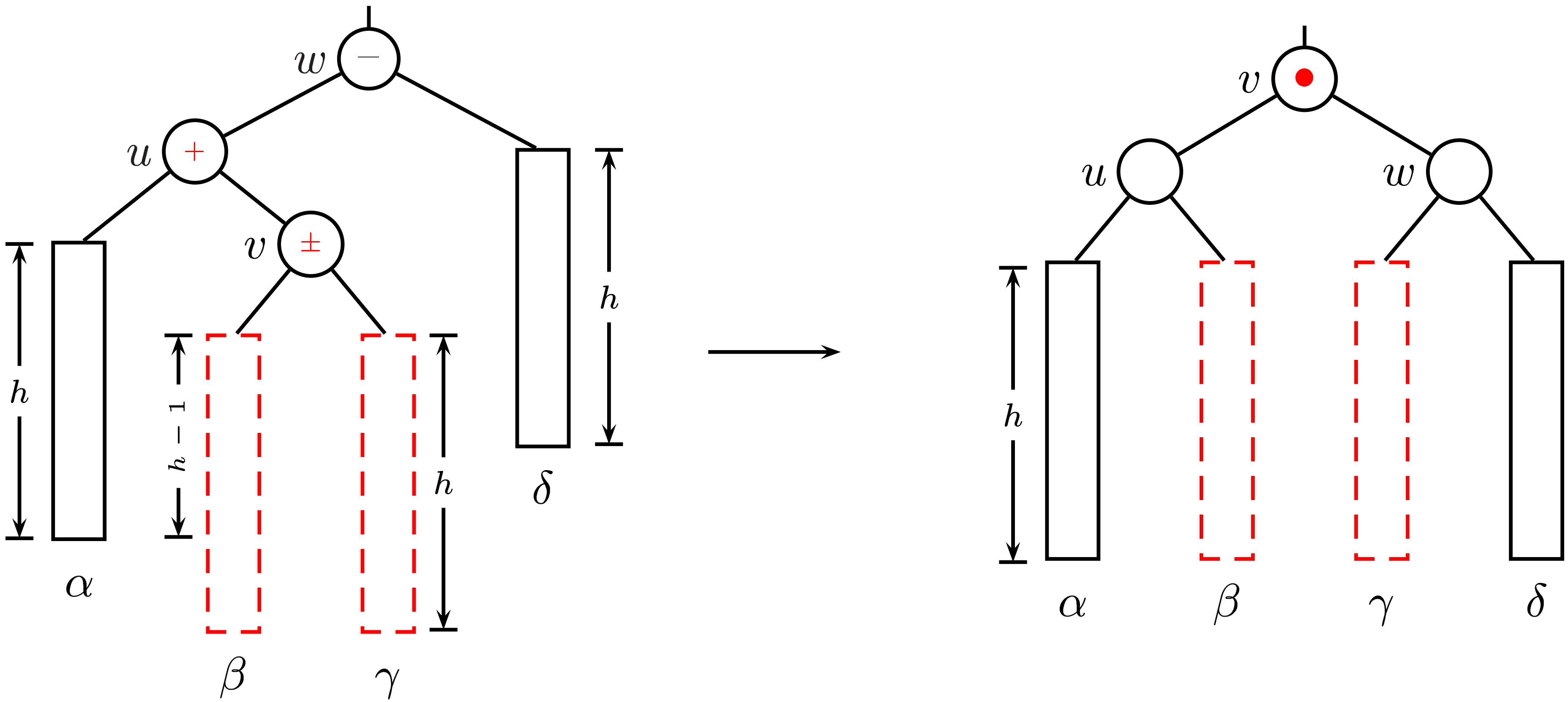
- after adding the new element u, we only have to adjust the balancefactor of u's father



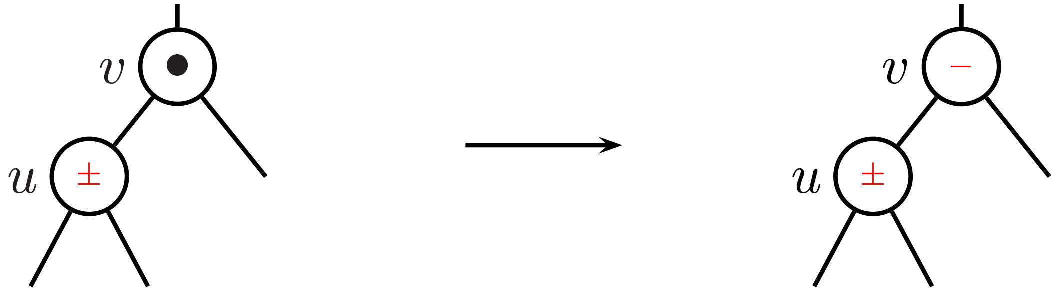
- we must adjust the balancefactor of v's father
- if the subtree rooted at a given node u has grown in height (i.e. the *new* balancefactor of u is either -1 or $+1$), we must adjust the balancefactor of u's father v



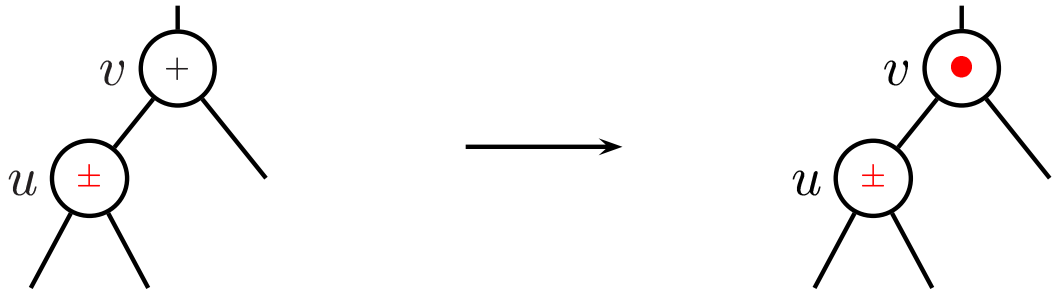
- right rotation
- in this case we are done! (why?)



- node v either has a balancefactor of $+1$ or -1 (on the left)
- after a double rotation, we are done



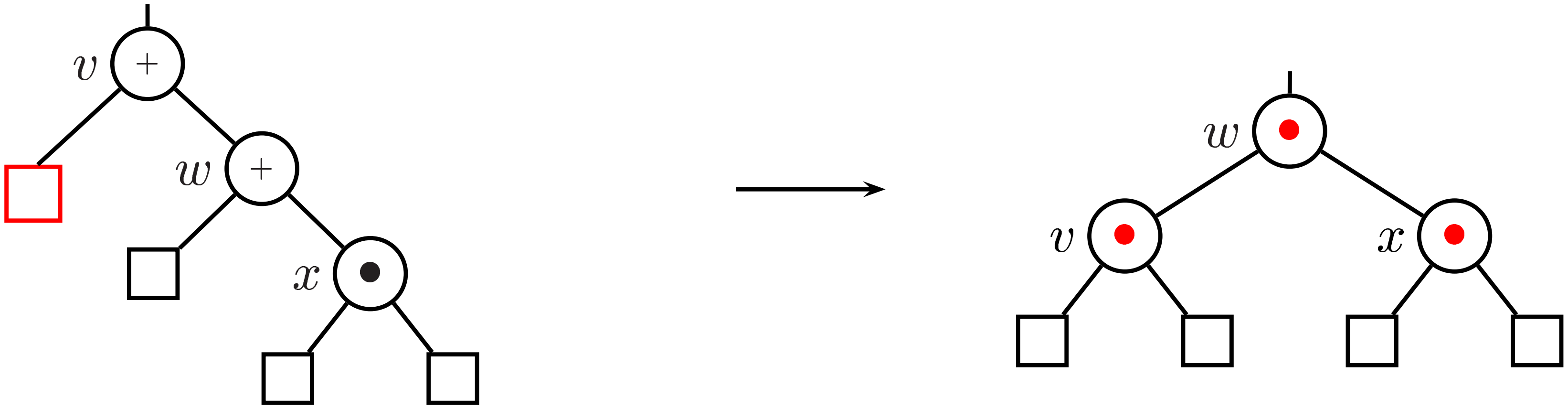
- the tree rooted at v has grown in height, therefore we must adjust the balancefactor of v's father

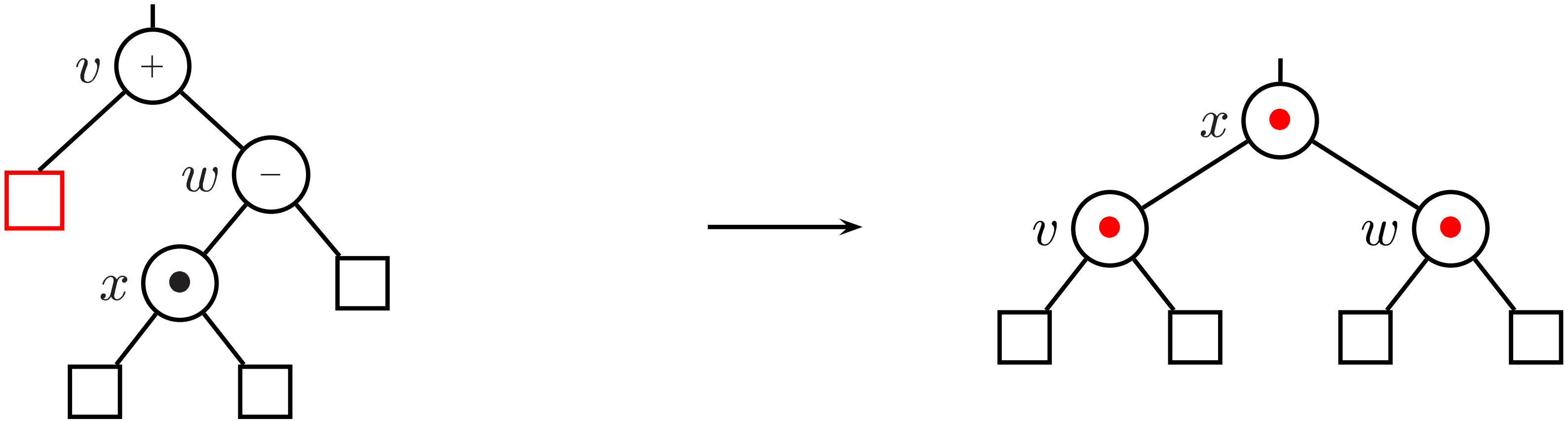


- the tree rooted at v did *not* grow in height; so after adjusting v's balancefactor, we are done

- if u is a right child of its father, we get symmetric cases
- at most one single/double rotation per insertion
- insertion of a new key in an AVL-tree with n internal nodes takes $O(\lg n)$ time

- **delete(u)**: if we delete an element u from an AVL-tree, we again have to update balancefactors and probably restructure the new tree, so that it is still an AVL-tree
- for the node u at disposition, there might be three cases
- *case 1*: u has two leaves
- substitute u by a leaf and change the balancefactor of its father v
  - if v has a balancefactor of 0, change it to $+1$ and we are done (because the height of the tree rooted at v did not change)
  - if v's balancefactor is -1, change it to 0; now the tree rooted at v has fallen in height, so we must adjust the balancefactor of v's father
  - the balancefactor of v is $+1$; we must rebalance the subtree rooted at v

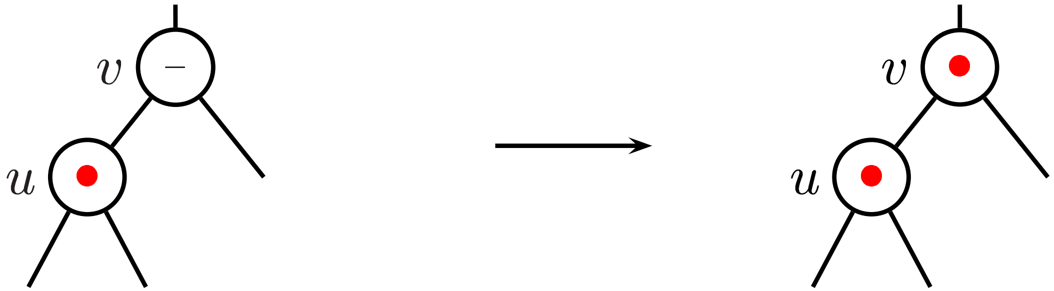




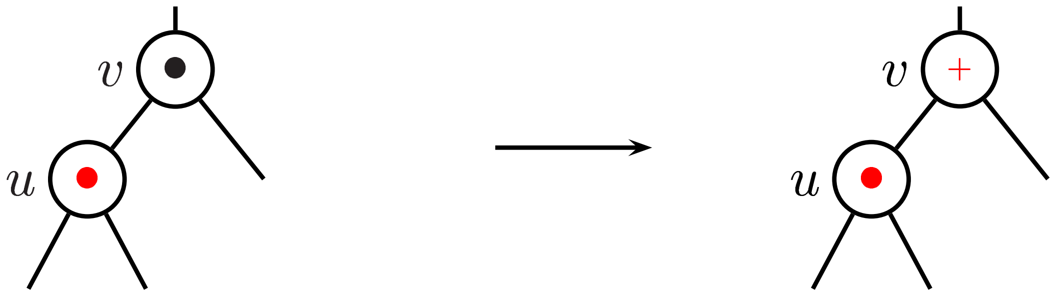
- the new tree rooted at w/x has fallen in height (compared to the old one), so we must adjust the balancefactor of the father of the new root

- *case 2*: u has only one leaf
  - substitute u by its non-leaf child w
  - the new tree rooted at w has fallen in height (from 2 to 1), hence we must adjust the balancefactor of w's (new) father
- *case 3*: u has no leaf nodes
  - substitute u by its symmetric successor v
  - delete v from the right subtree of u, i.e. this case falls back to either case 1 or case 2 (why?)
- after deleting u from the tree (and doing some local updating) we (probably) still have to update balancefactors of nodes
- proceeding is *somewhat* similar to the one shown for inserting an element

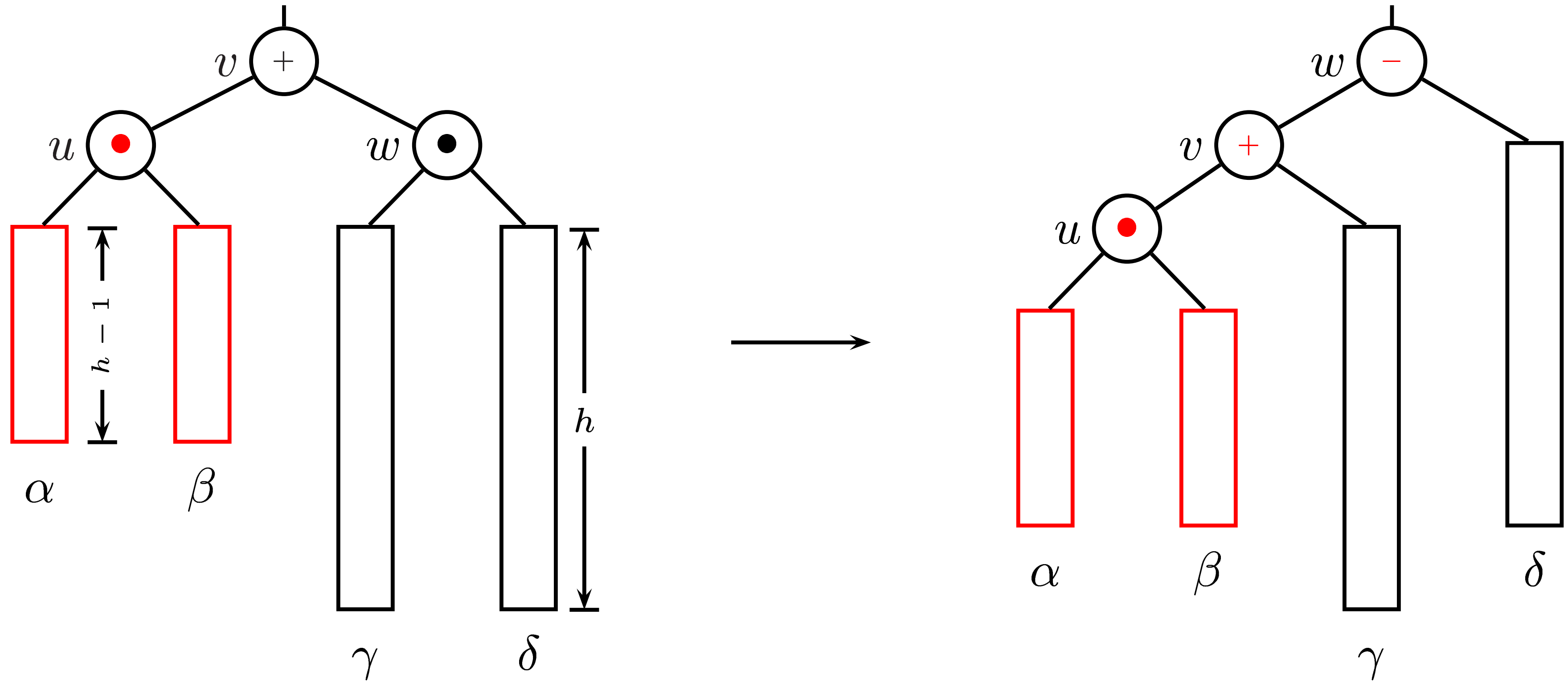
- whenever the *new* balance factor of a node u is 0, the height of the subtree rooted at u has decreased by one
- hence we must adjust the balancefactor of u's father



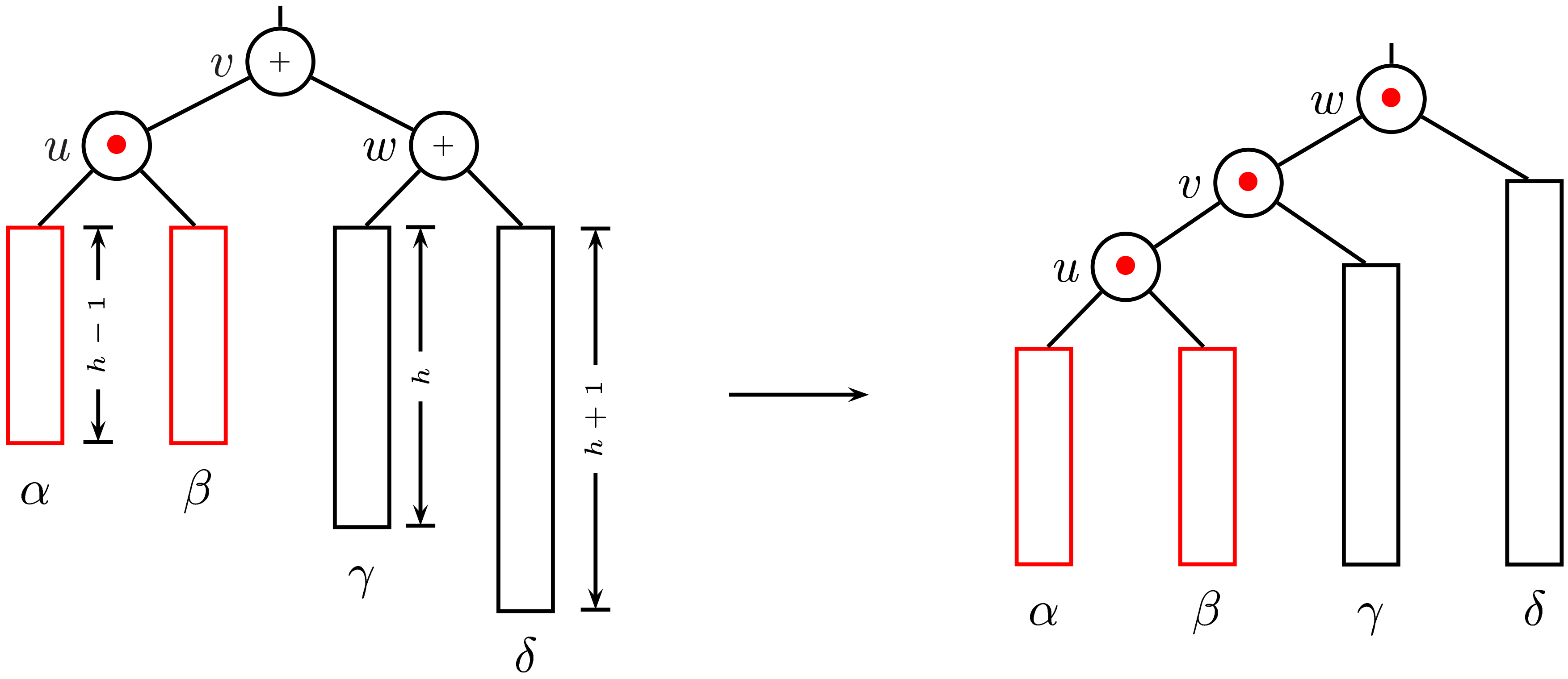
- the overall height of the subtree rooted at v has decreased, because the height of the subtree rooted at u decreased



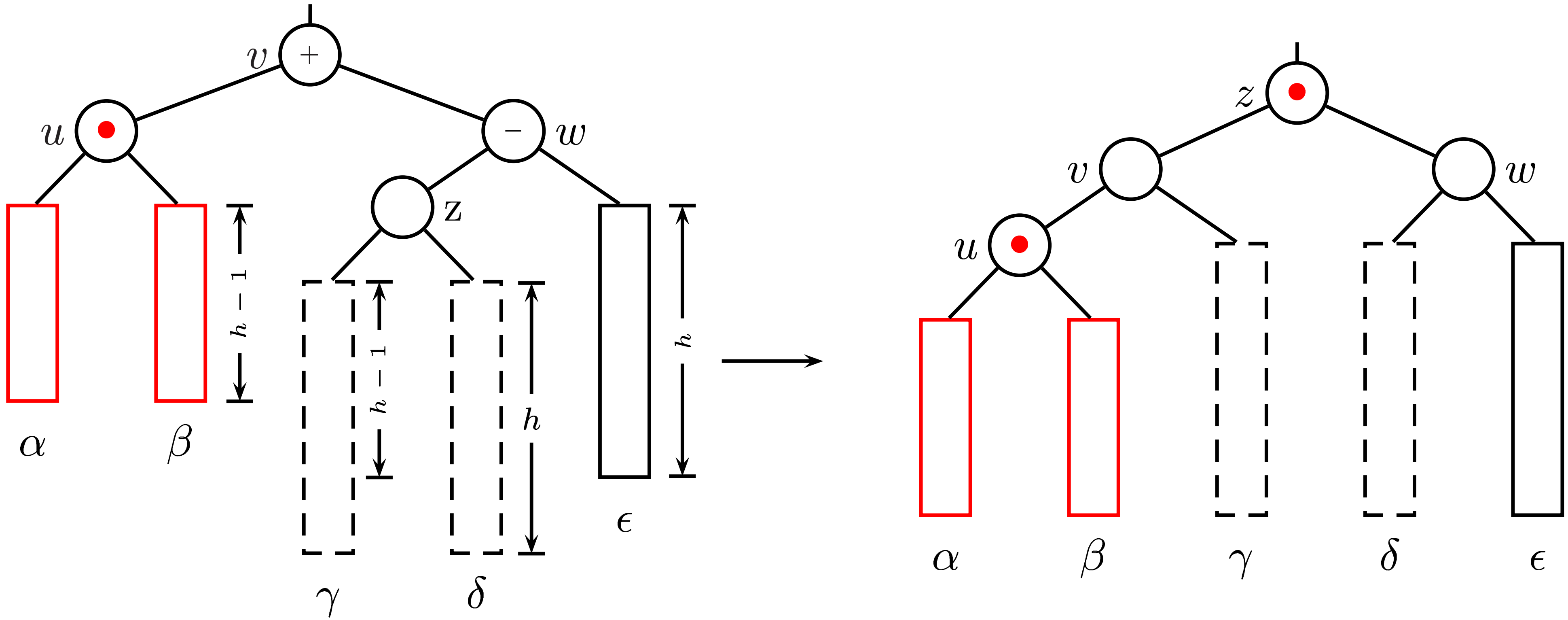
- the height of the subtree rooted at u has fallen, but the tree rooted at v still has the same height as before the deletion operation



- without (left) rotation, node v would no longer fulfill the property of an AVL-tree; after the rotation, the new subtree rooted at w has the same height as the subtree rooted at v before the deletion operation, i.e. no further updates are necessary



- rotate left
- the new tree's height is smaller than the height of the original tree



- double rotation (right-left)
- decrease in the tree's height after a double rotation

- symmetric cases, if u is a right child
- (unlike insertion) there might be up to $O(\lg n)$ single/ double rotations necessary
- deleting an element from an AVL-tree with n nodes takes $o(\lg n)$ time

Remarks

- (straight forward) recursive implementation
- no theoretical determination of the average behavior yet
- empirical results:
  - each second insertion a rotation
  - each fifth deletion a rotation

$(2,3)$ **trees**

- (*Hopcroft*): A tree is called a $(2,3)$-tree if:
  - each node has at least two children
  - each node has at most three children
  - all leaves are at the same height
- all data is saved in the *leaves*
- internal nodes save key-values only (directing the search for data items)
- internal node with two children has one key K such that all key values in its left subtree are at most equal to K, and all values in its right subtree are larger than K

- internal node with three children contains two keys K and K' (with $K < K'$) and all keys in the left subtree are at most equal to K; all keys in its middle tree have values greater than K and at most equal to K' and all values in the right subtree are greater than K'

Theorem B *The height of a* $(2,3)$*-tree with* n *leaves is between* $\lg n/\lg 3$ *and* $\lg n$

- $2^h \leq n \leq 3^h$

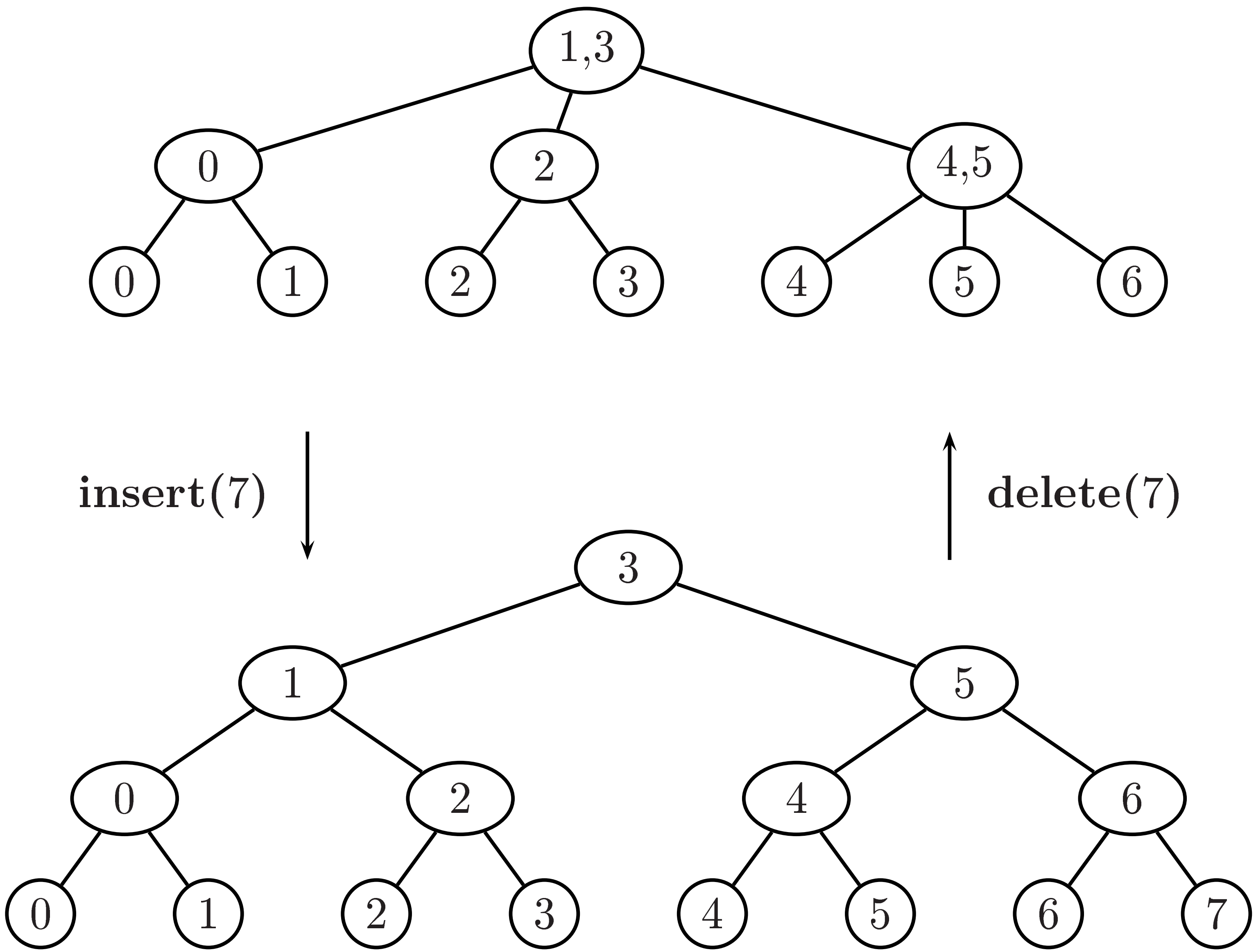
Analysis of dictionary operations for $(2, 3)$ trees

- **find(u)**: by Theorem B, it takes $O(\lg n)$ time to find an element in a $(2, 3)$-tree with n leaves
- **insert(u)**: find(u) points to a leaf $u' \neq u$
- add $\min(\text{key}(u), \text{key}(u'))$ as new key to the father v of u' and add u as a right or left children of this key to v
- if v now has three keys (i.e. *four* children), split p into two nodes (containing the first and third key respectively) and add the middle key as a new key and the newly created node as a new child to v's father

- if a node with no father is split (i.e. the root), the tree grows in height
- inserting an element into a $(2, 3)$-tree with n leaves takes $O(\lg n)$ time

- **delete(u)**: delete the node resulting in a call to find(u) *and* its key from u's father v
- v might became an empty node (with one remaining child u')
- possibility a: if v's left or right sibling has three children (i.e. two key values), add u' to the right or left of this sibling together with the appropriate key from v's father resulting in a node with three keys; hence split this node and move the middle key up to v's father (and delete v)

- possibility b: v has at least one sibling with a key, so make u' child of this sibling and add also the appropriate key from v's father to it (and delete v)
- possibility c: v has no siblings, i.e. it is the root; hence delete it (the tree is decreased in height)
- it takes $O(\lg n)$ to delete an element from a $(2, 3)$-tree with n leaves
- let's look at an example with worst-case behavior: inserting/deleting 7


1,3
0
2
4,5
0
1
2
3
4
5
6
insert(7)
delete(7)
3
1
5
0
2
4
6
0
1
2
3
4
5
6
7

Remarks

- special case of $B+$-tree
- special case of (a, b)-tree $(b \geq 2a - 1)$
- efficient algorithm (single pass?)

Amortized Analysis

- time required to perform a *sequence* of operations on a data structure
- total *amortized cost* must be a (tight) upper bound of the actual cost
- if we average over a sequence of operations, we can show that the average cost of an operation is small (though performed as a single operation, it might be expensive)
- different from average-case analysis (no probability is involved)
- different flavors: aggregate/accounting/potential method

- **aggregate method**: show that for all n, a sequence of n operations takes *worst case* time $T(n)$ in total
- amortized cost per operation is $T(n)/n$ (same for different operations)
- example: binary counter with k bits
- increment counter n times, starting at value 0 (all k bits are zero)

INCREMENT(A)

```
begin
    j := 0
    while ((j < length(A)) ∧ (A[j] = 1))
        A[j] := 0
        j := j + 1
    if j < length(A)
        then A[j] := 1
end
```

- worst case analysis leads to $O(k \cdot n)$

c	bits that are flipped	total number of flips
1	$A[0]$	1
2	$A[0], A[1]$	2
3	$A[0]$	1
4	$A[0], A[1], A[2]$	3
5	$A[0]$	1
6	$A[0], A[1]$	2
7	$A[0]$	1
8	$A[0], A[1], A[2], A[3]$	4
total number of flips for $X = 0$ to $X = 8$ is 15		

- $A[1]$ is flipped every second time (or $\lfloor n/2 \rfloor$ times), $A[2]$ is flipped every forth time (i.e. $\lfloor n/4 \rfloor$ times), ...
- this leads to
$$\sum_{i=1}^{\lg n} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=1}^{\infty} \frac{n}{2^i} = 2n$$
- hence the worst case time for a sequence of n increment operations (on an initially zero-valued counter) is $O(n)$, resulting in an amortized cost for one increment operation of $O(1)$
- **accounting method**: similar to potential method

- **potential method**: associate a *potential energy* with the given data structure
- each operation on the data structure changes the potential
- in a sequence of n operations, let D_i denote the data structure after the i^{th} operation, $1 \leq i \leq n$ (D_0 is the data structure at the beginning)
- c_i is the actual cost of the i^{th} operation
- Φ denotes the *potential function*, a mapping of each data structure D_i to its potential $\Phi(D_i)$ (a real number)
- $\hat{c}_i$ is the amortized cost of the i^{th} operation, defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- total amortized cost of n operations:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- total amortized cost should be an upper bound of the actual cost, hence define Φ in such a way, that $\Phi(D_n) \geq \Phi(D_0)$ (for *all* n)
- to get the desired time bounds, choose the right potential function
- example: binary counter with k bits

- define the potential to be the number of 1's in the counter
- suppose the i^{th} increment resets (sets to zero) t_i bits and before this operation, b_{i-1} bits were set
- the total actual cost is therefore at most $t_i + 1$ (at most one bit is set to 1) and the change in potential is

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

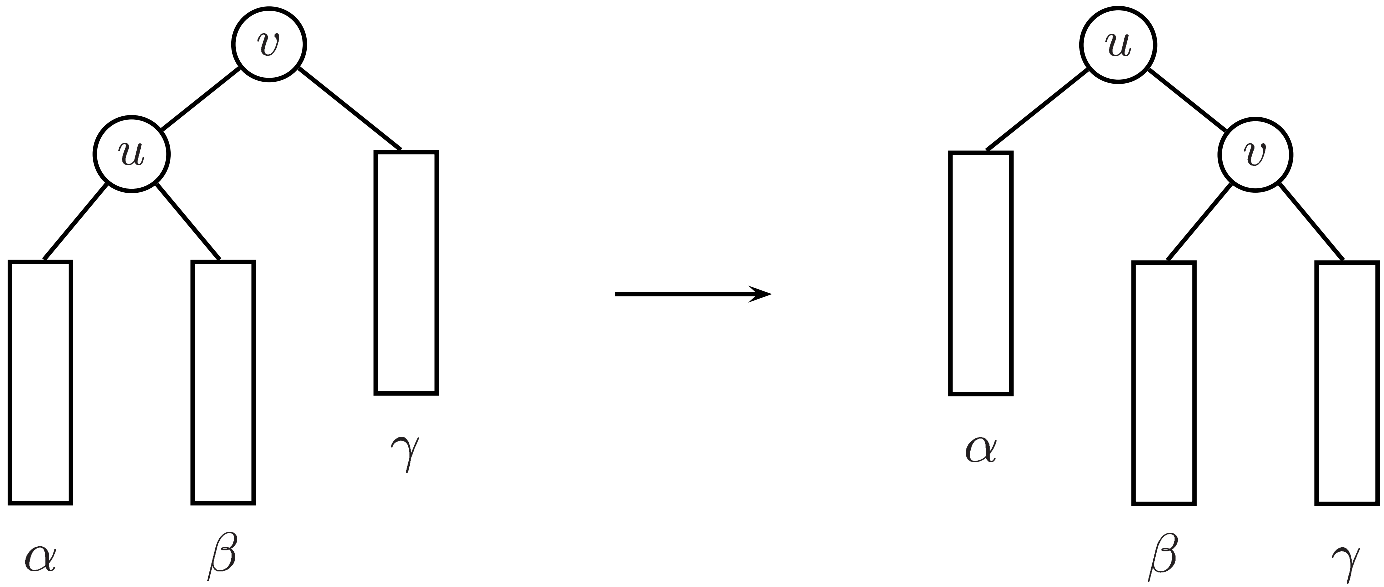
- the amortized cost is therefore:

$$\hat{c_i} \leq (t_i + 1) + (1 + t_i) = 2$$

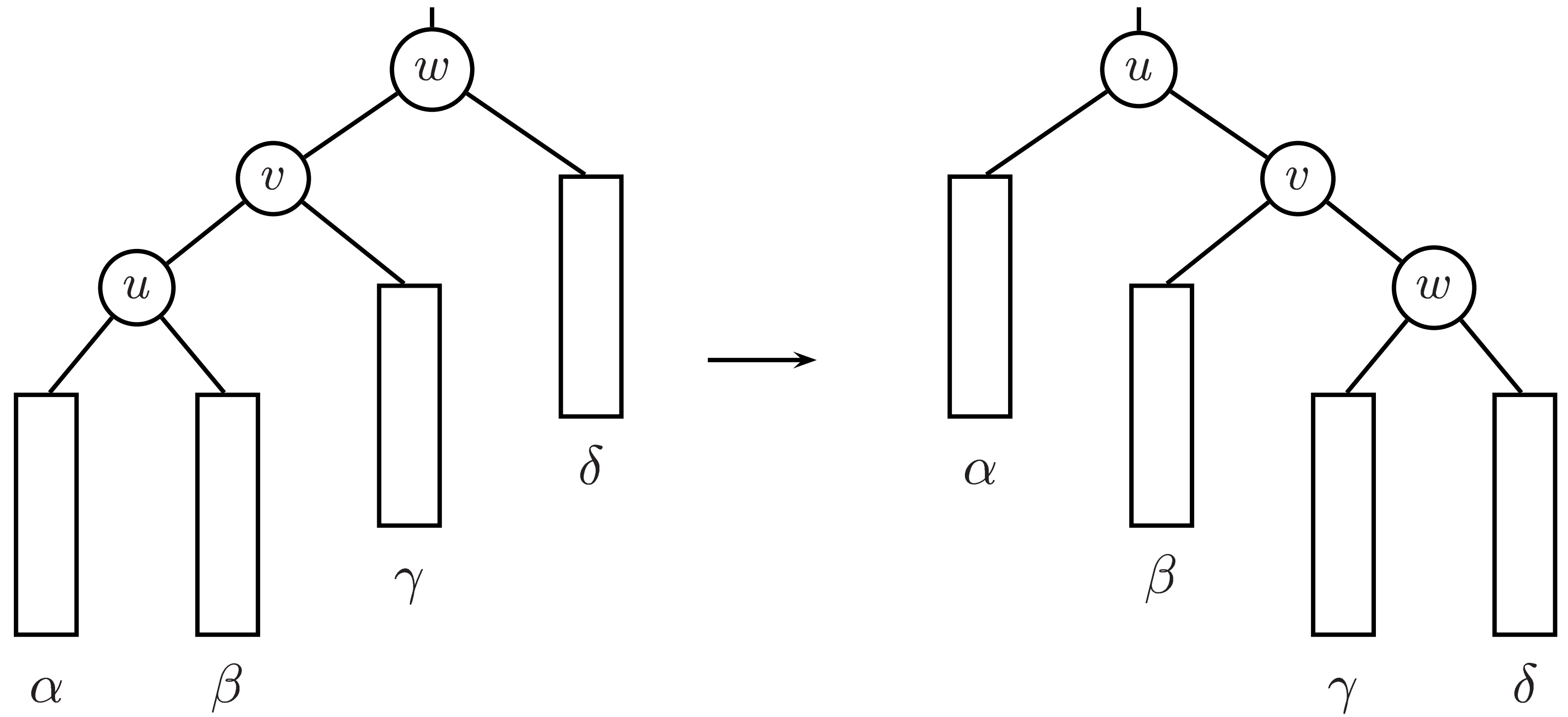
- $\Phi(D_i) \geq 0$ for all i, so the total amortized cost of a sequence of n increment operations is an upper bound on the total actual cost

Splay trees

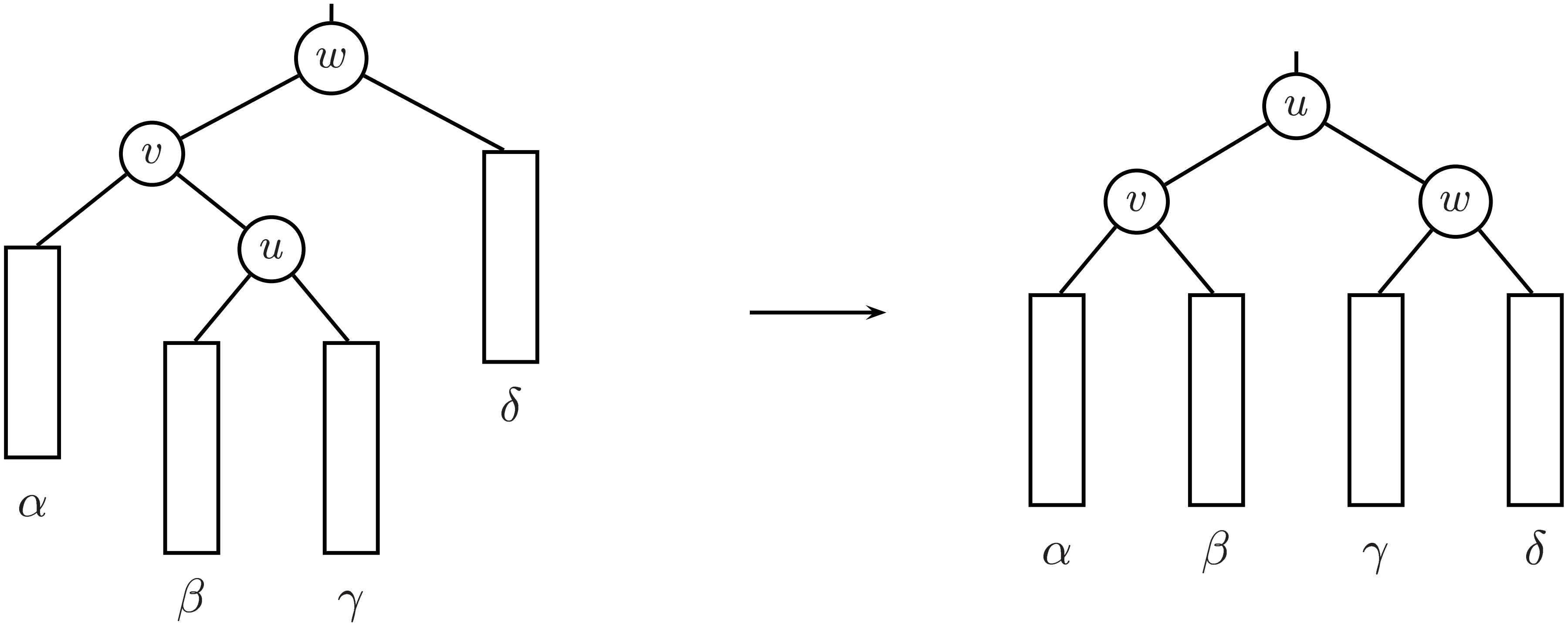
- (*Sleater* and *Tarjan*) A *Splaytree* is a binary tree, that uses *splaying* as its restructuring heuristic
- idea: whenever an item is accessed (find, insert, delete) move it (and its neighborhood) closer to root, because this item is likely to be accessed soon
- self adjusting data structures (restructuring takes even place when elements are accessed)
- e.g. move-to-front in lists
- splaying moves the accessed item all the way to the root
- to splay a tree at node u, we repeat the following splaying steps until u is the root of the tree



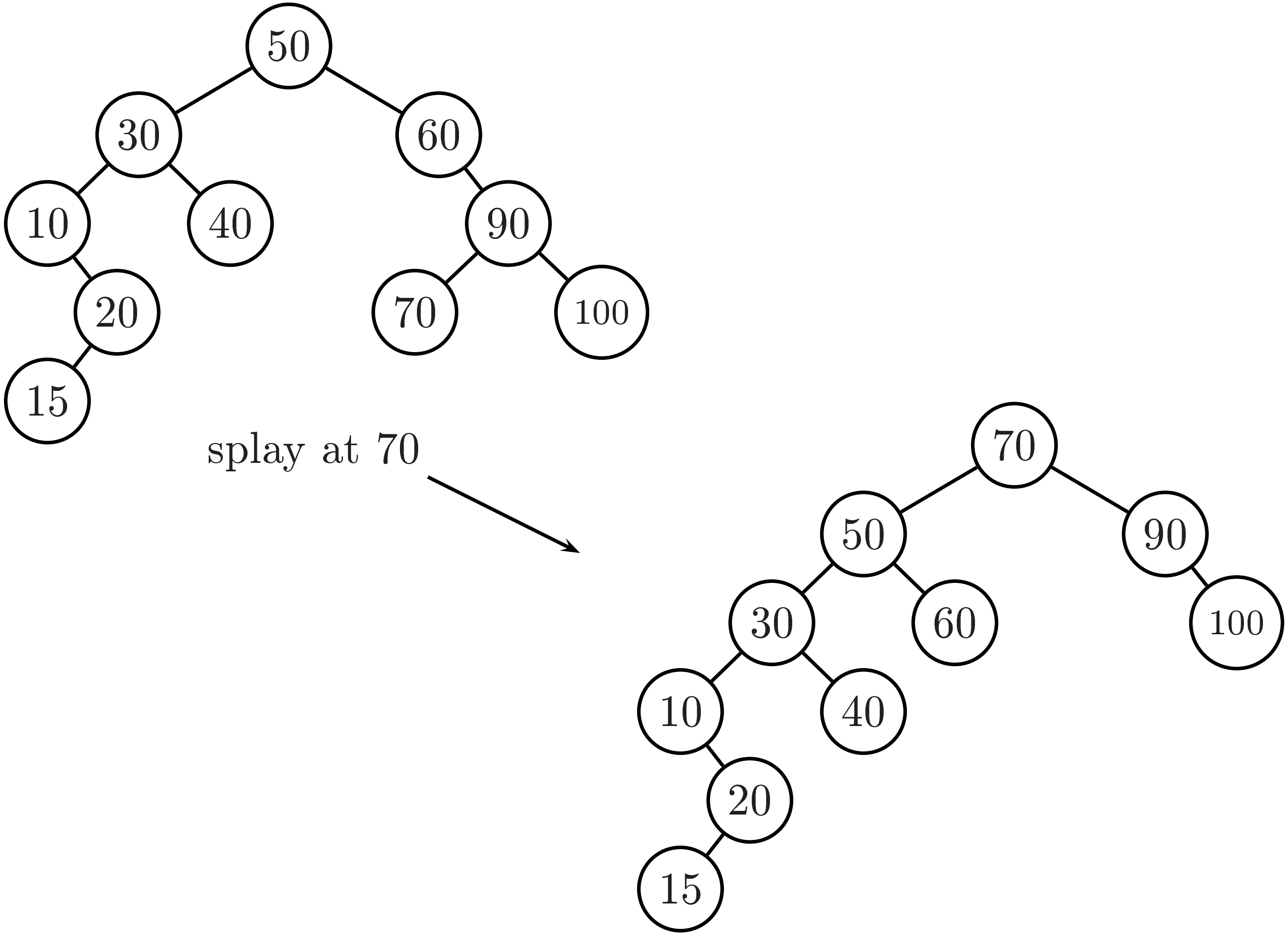
- case 1 (zig): if $p(u)$, the father of u is the tree root, rotate the edge joining u with $p(u)$ (terminal case, performed at most once)



- case 2 (zigzig): if $p(v)$ is not the root and v and $p(v)$ are both left or both right children, rotate the edge joining $p(v)$ with $g(v)$ (the grandfather or v), and then rotate the edge joining v with $p(v)$



- case 3 (zigzag): if $p(v)$ is not the root and v is a left child and $p(v)$ a right child, or vice versa, rotate the edge joining (v) with $p(v)$ and then rotate the edge joining v with the new $p(v)$


50
30
60
10
40
90
20
70
100
15
splay at 70
70
50
90
30
60
100
10
40
20
15

- amortized analysis of splaying: actual cost plus change in potential
- each node i has a (arbitrary but fixed) weight $w(i)$
- the size $s(x)$ of a node x in the tree is the sum of the individual weights of all items in the subtree rooted at x
- the rank $r(x)$ of node x is $\log s(x)$
- the potential of a given tree is the sum of the ranks of all its nodes
- let r and r' denote the rank functions just before and after zig/zigzig/zigzag step

- the amortized time for a zig step is $1 + 3(r'(x) - r(x))$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + [r'(u) + r'(v)] - r(u) - r(v) \\
&\leq 1 + r'(u) - r(u) \\
&\leq 1 + 3(r'(u) - r(u))
\end{aligned}$$

- only u and v can change rank
- $r(v) \leq r'(v)$
- $r'(u) \geq r(u)$
- similarly, one can show that the amortized time for both the zigzig and zigzag steps are each equal to $3(r'(u) - r(u))$

- splaying of a node u consists of a sequence of splaying steps, so we get :

Theorem C *The amortized cost to splay a tree with root t at a node u is at most* $3(r(t) - r(u)) + 1 = O(\log(s(t)/s(u)))$

Analysis of dictionary operations for splaytrees

- additional operations: **join**(t_1, t_2) and **split**(v, t)
- join(t_1, t_2: combine trees t_1 with t_2 into a single tree containing all items from both trees and return the resulting tree; this operation assumes, that all items in t_1 are less than all those in t_2
- split(i, t): construct and return two trees t_1 and t_2, where t_1 contains all items in t less than or equal to i and t_2 contains all items in t greater than i
- **access**(u): if the search finds u, splay u to the root and return a pointer to u; otherwise splay the last element of the search to the root and return a null pointer

- **join**(t_1, t_2): first, access the largest element in t_1; now the root of t_1 has an empty right child, hence add here t_2
- **split**(v, t): access node v and break either the left or right subtree of the newly restructured tree to get the two new trees
- **insert**(v): perform split(v, t) and make the returning two trees t_1 and t_2 left and right subtrees of v
- **delete**(v): access node v, bringing v to the root; now join the left and right subtree of v
- all operations are based on the access operation

- the amortized cost of **access(u)** is determined by the amortized cost of splaying a node, so we get:

$$\textsc{Access}(u) : \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + O(1), & \text{if } u \text{ is in the tree} \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + O(1), & \text{if } u \text{ is not in the tree} \end{cases}$$

- for any item u in a tree, let u^- and u^+, respectively, denote the item preceding u and the item following u in the tree. If u^- is undefined, $w(u^-) = \infty$ and if u^+ is undefined, let $w(v^+) = \infty$.

$$\textsc{Split}(u) : \begin{cases} 3\log\left(\frac{W}{w(u)}\right) + O(1), & \text{if } u \text{ is in the tree} \\ 3\log\left(\frac{W}{\min\{w(u^-),\ w(u^+)\}}\right) + O(1), & \text{if } u \text{ is not in the tree} \end{cases}$$

$$\textsc{Join}(t_1, t_2) : 3\log\left(\frac{W}{w(u)}\right) + O(1),\ u \text{ maximal in } t_1$$

$$\textsc{Insert}(u) : 3\log\left(\frac{W - w(u)}{min\{w(u^-), w(u^+)\}}\right) + \log\left(\frac{W}{w(u)}\right) + O(1)$$

$$\textsc{Delete}(u) : 3\log\left(\frac{W}{w(u)}\right) + 3\log\left(\frac{W - w(u)}{w(u^-)}\right) + O(1)$$

Remarks

- amortized analysis is independent of weights (adopt weights for the desired time bound)
- though presented bottom-up, in practice splaying is implemented top-down
- lots of restructuring might be a disadvantage
- used in many applications (SQUID, WinNT, GCC . . .)

Theorem D *In a sequence of m accesses on an n-node splay tree, the total access time is* $O(m + m \log n + n \log n)$ *(Balancetheorem)*

- $n \log n$ is the maximum possible difference in potential
- on a sufficiently long sequence of accesses a splay tree is as efficient as any form of uniformly balanced tree

- there is an interesting conjecture about splay trees called the *dynamic optimality conjecture*: splay trees are as asymptotically fast on *any* sequence of operations as *any* other type of search tree with rotations. i.e. the dynamic optimality conjecture guesses that if *any* search tree can exploit the structure of a sequence of accesses to achieve asymptotically faster running time, so can splay trees.