CS 240 – Data Structures and Data Management

Module 4: Dictionaries

O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2024

Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - deletion

Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - deletion

Dictionary ADT

- *Dictionary* ADT consists of a collection of items, each item contains
  - a *key*
  - a *value* (some data)
- Item is called a *key-value pair* (KVP)
- Keys can be compared and are (typically) unique
    - can extend to handle non-unique keys
- Operations
  - *search*(k)
    - also called *findElement*(k)
  - *insert*(k, v)
    - also called *insertItem*(k, v)
  - *delete*(k)
    - also called *removeElement*(k)
  - optional: *closestKeyBefore*, *join*, *isEmpty*, *size*, *etc.*

Dictionary ADT: Common Assumptions

- We will make the following assumptions
  - dictionary has n KVPs
  - each KVP uses constant space
    - if not, the "value" could be a pointer
  - keys can be compared in constant time

Elementary Implementations

- **Unordered array or linked list**

(7,'Ace')	(1,'Pot')	(3,'Top')	(2,'Dog')

  - *search* $\Theta(n)$
  - *insert* $\Theta(1)$
    - except if using array, the array occasionally needs to resize, so it is $\Theta(1)$ amortized time, but we do not discuss amortization details
  - *delete* $\Theta(n)$
    - need to search

- **Ordered array**

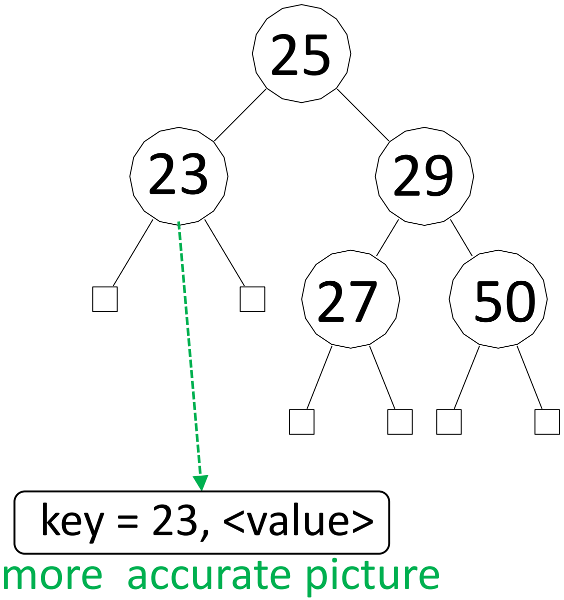
(1,'Pot')	(2,'Dog')	(3,'Top')	(7,'Ace')

  - *search* $\Theta(\log n)$
    - via binary search
  - *insert* $\Theta(n)$
  - *delete* $\Theta(n)$

Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - full code for insertion
    - deletion

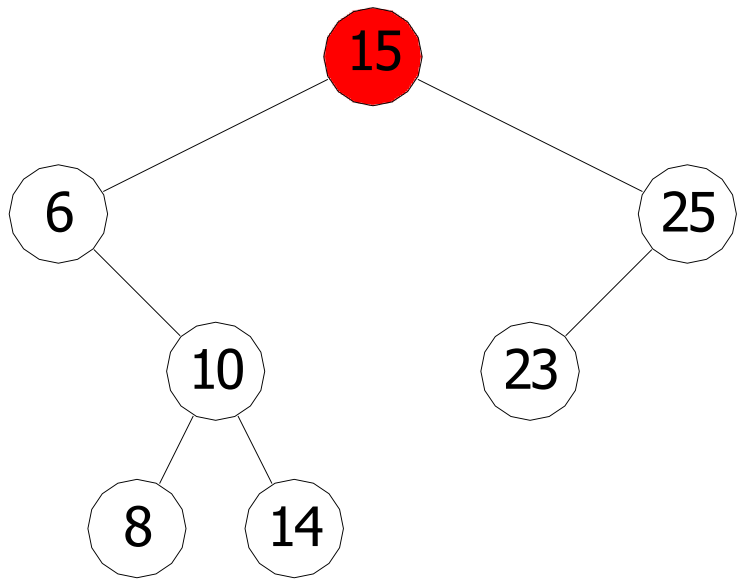
Binary Search Trees (review)



- **Structure**
  - binary tree is either empty or consists of nodes
  - all nodes have two (possibly empty) subtrees
    - *L* (left)
    - *R* (right)
  - every node stores a KVP
  - leaves store empty subtrees
  - empty subtrees usually not shown
- **Ordering**
  - every key k in the left subtree of node v is less than $v.key$
  - every key k the right subtree of node v greater than $v.key$

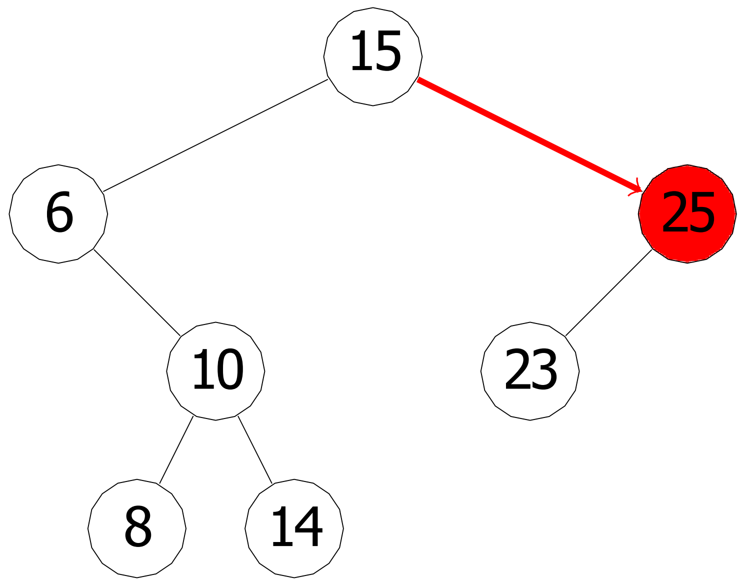
BST Search

- *BST::search*(k)
    - start at root, compare k to current node
    - stop if found or subtree is empty, else recurse at subtree
- Example: *BST::search*(24)



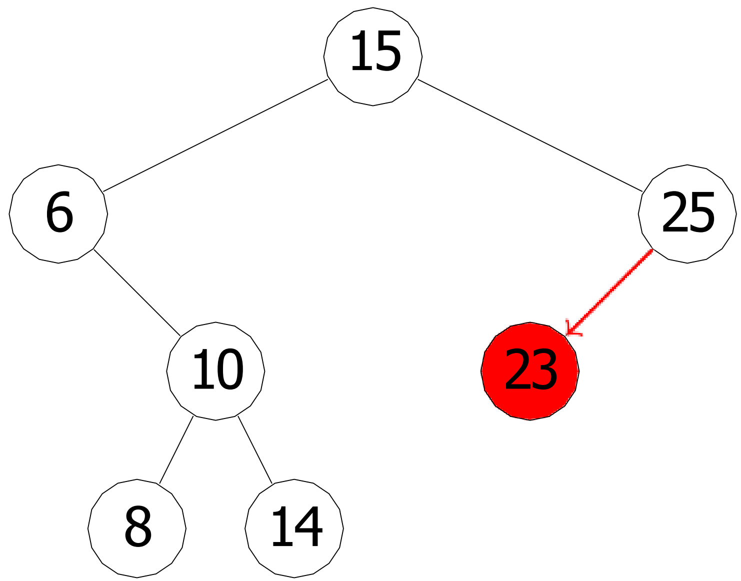
BST Search

- *BST::search*(k)
  - start at root, compare k to current node
  - stop if found or subtree is empty, else recurse at subtree
- Example: *BST::search*(24)



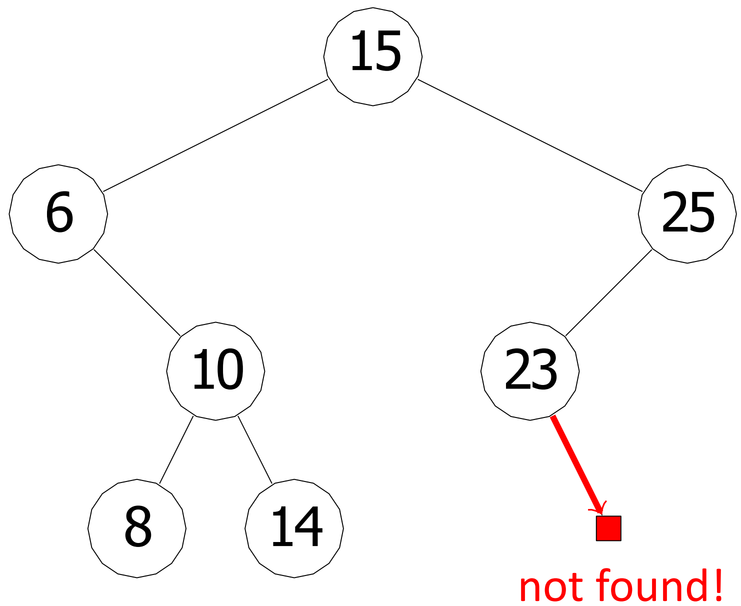
BST Search

- *BST::search*(k)
  - start at root, compare k to current node
  - stop if found or subtree is empty, else recurse at subtree
- Example: *BST::search*(24)



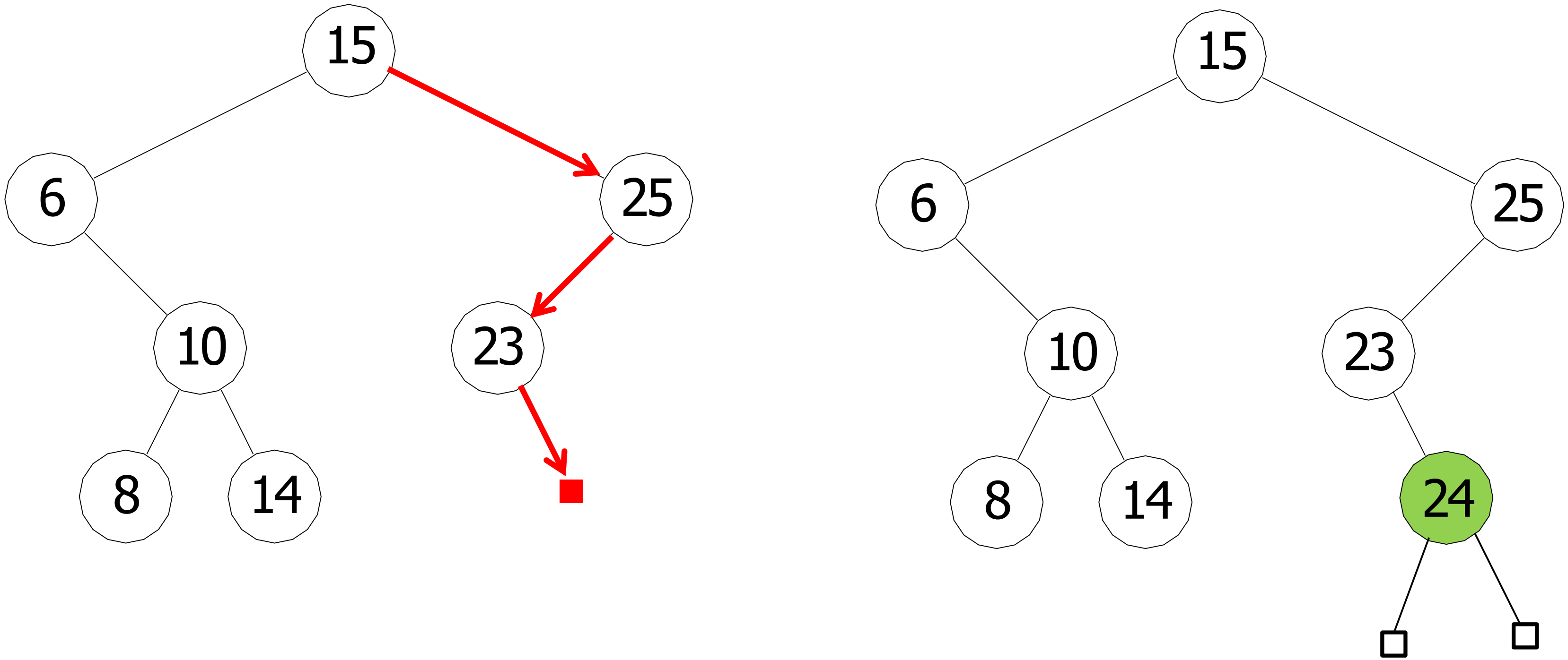
BST Search

- *BST::search*(k)
  - start at root, compare k to current node
  - stop if found or subtree is empty, else recurse at subtree
- Example: *BST::search*(24)



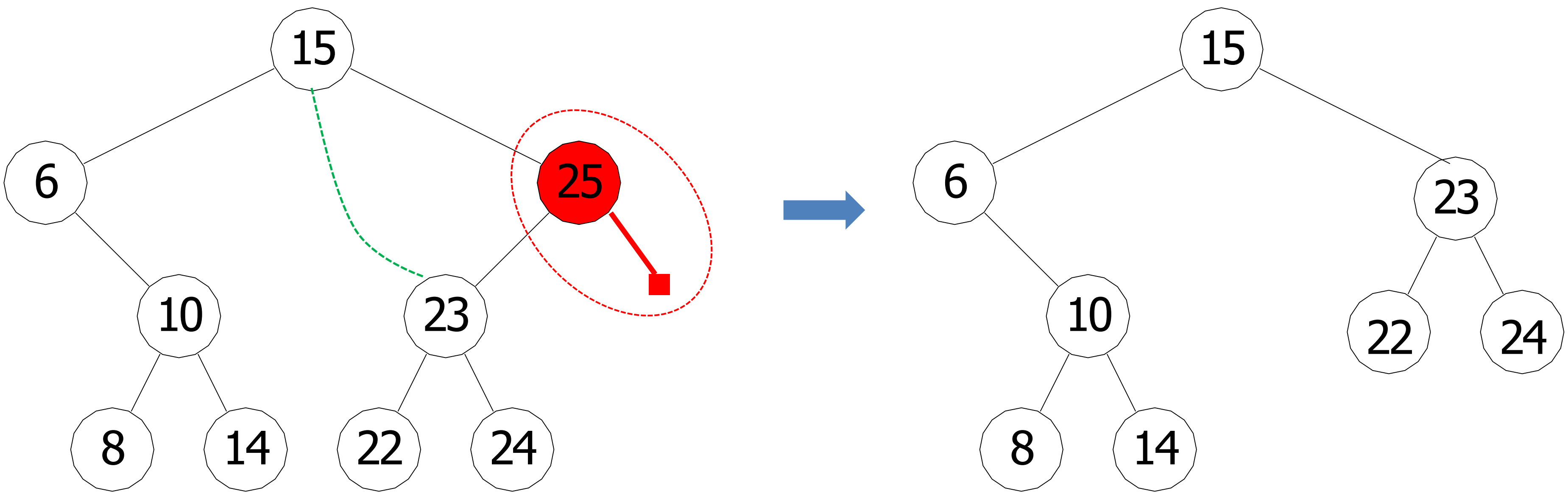
BST Insert

- *BST::insert*(k, v)
  - search for k, then insert (k, v) as a new node at the empty subtree where search stops
- Example: *BST::insert*$(24, v)$



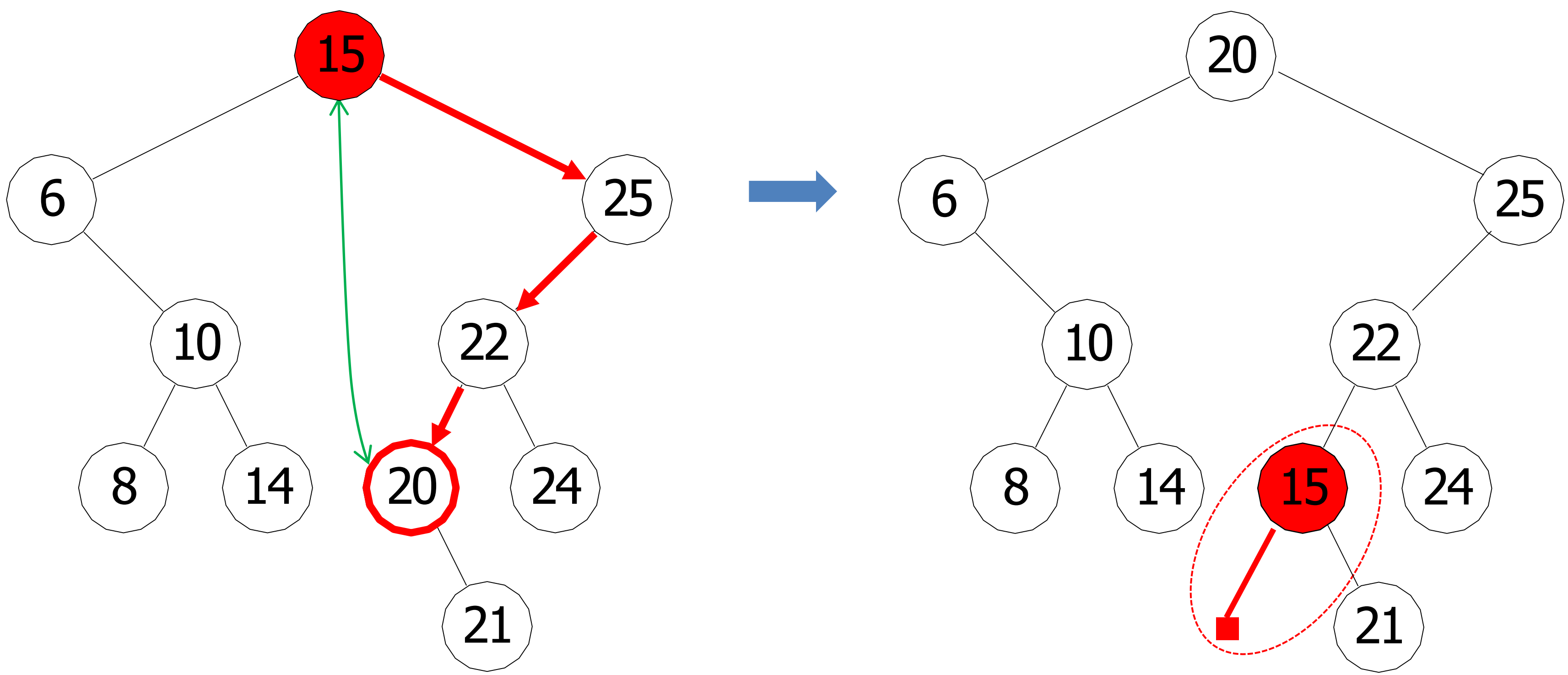
BST Delete: Case 1

- First search for node x containing the key
  1. If x has at an empty subtree
     - delete x with the empty subtree
     - If x has a parent, reconnect the other subtree of x to the parent of x
- Example: *BST::delete*(25)



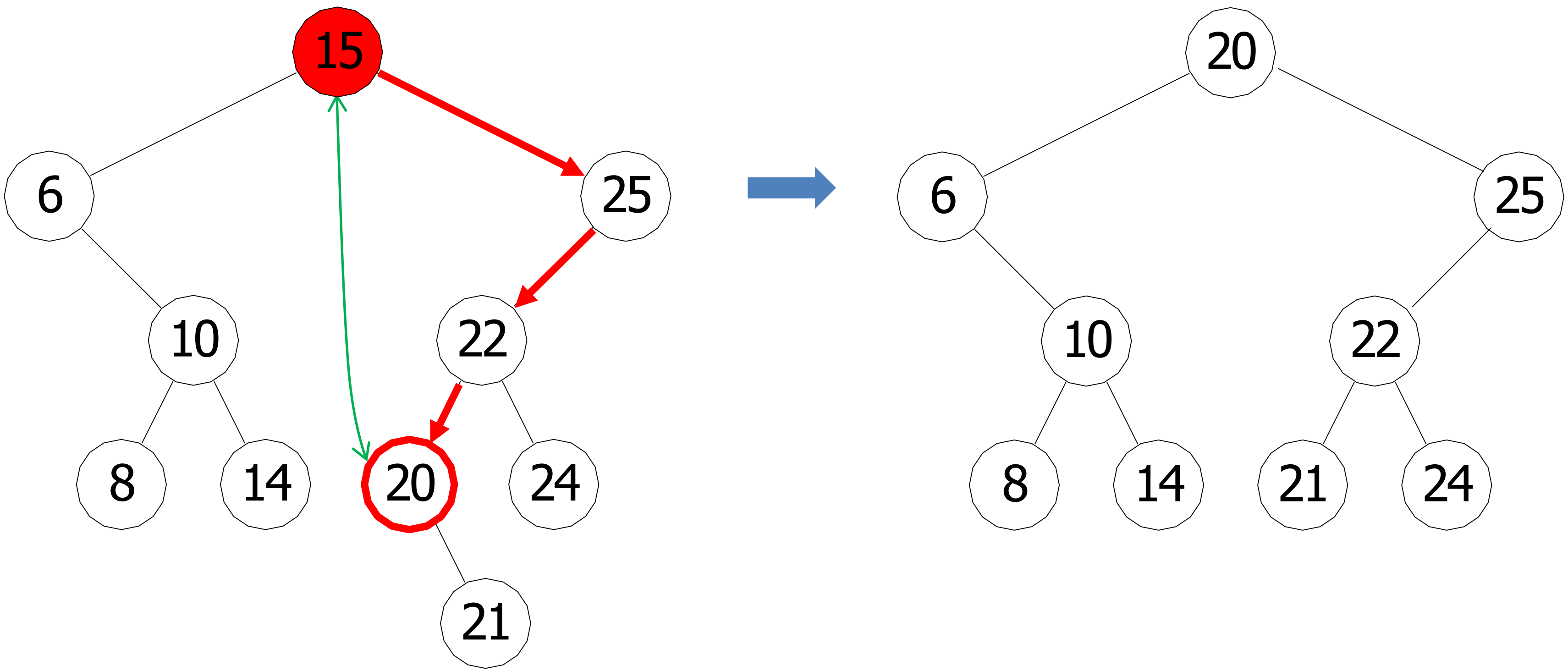
BST Delete: Case 2

- First search for node x containing the key
  2. If x has only non-empty subtrees
     - swap *KVP* at x with *KVP* at successor node (or predecessor node)
       - successor = smallest key node in the right subtree
     - delete successor node (or predecessor node)
       - now case 1 applies
- Example: *BST::delete*(15)

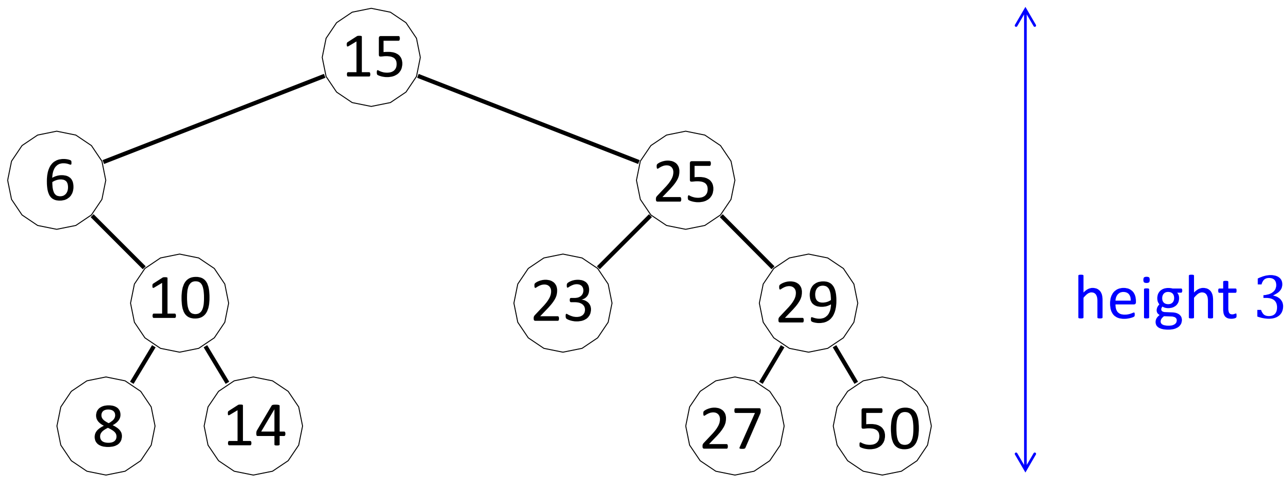


BST Delete: Case 2

- First search for node x containing the key
  2. If x has only non-empty subtrees
     - swap *KVP* at x with *KVP* at successor node (or predecessor node)
       - successor = smallest key node in the right subtree
     - delete successor node (or predecessor node)
       - now case 1 applies
- Example: *BST::delete*(15)



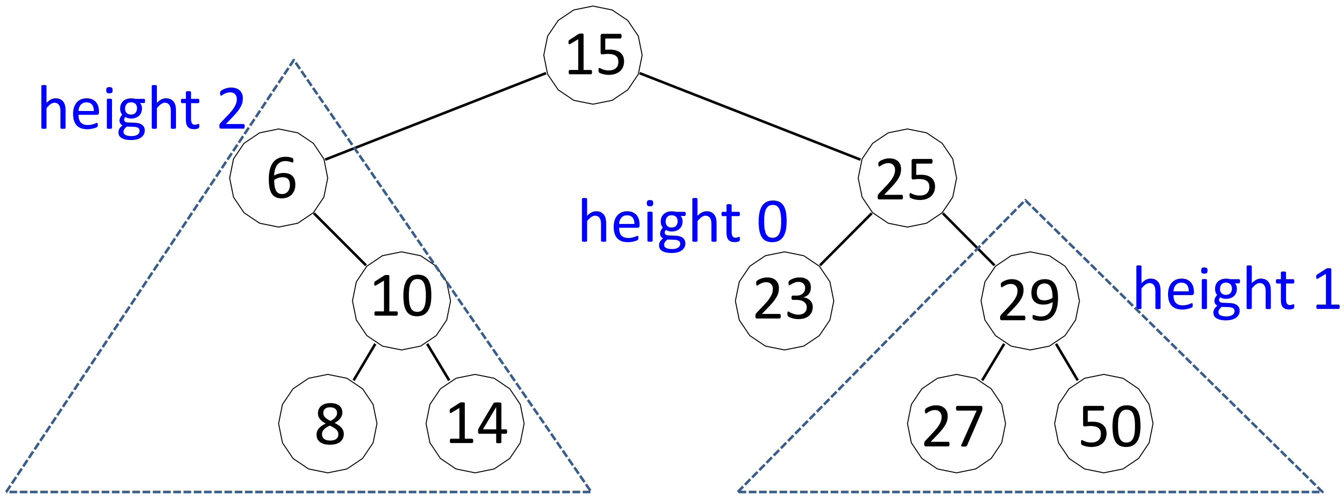
Height of a BST



- *BST::search*, *BST::insert*, *BST::delete* all have cost $\Theta(h)$
    - h = height of the tree = maximum length path from root to a leaf node
    - height of an empty tree is defined to be -1
- If n items are *BST::insert*ed one-at-a-time, how big is h?
    - worst-case is $n - 1 = \Theta(n)$
    - best case is $\Theta(\log n)$
        - binary tree with n nodes has height $\geq \log(n + 1) - 1$
    - can show if insert items in random order then height is $\Theta(\log n)$

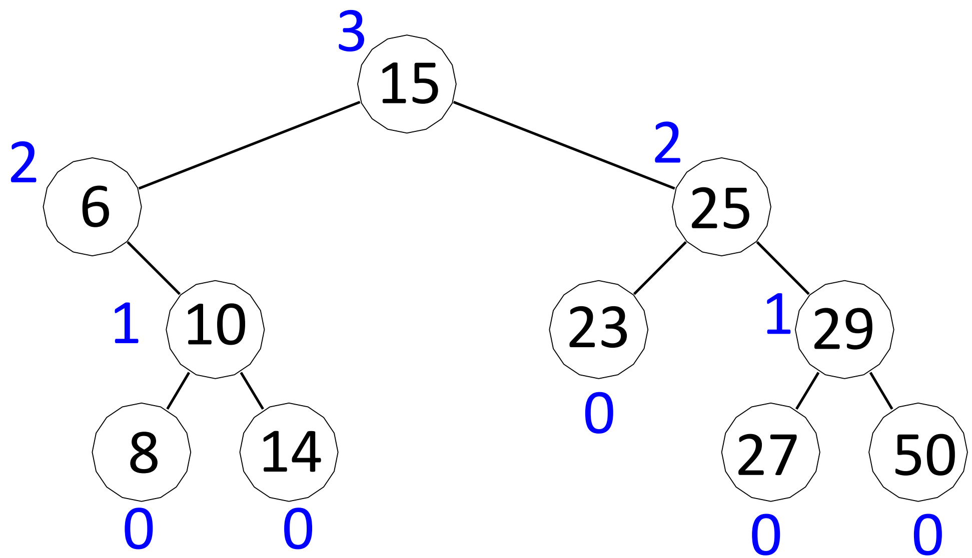
Height of a node

- Height of node v is the height of the tree rooted at node v



Height of a node

- Height of node v is the height of the tree rooted at node v



- Can compute heights of all nodes in post order traversal
  - leaf height is 0
  - height of any other node v is

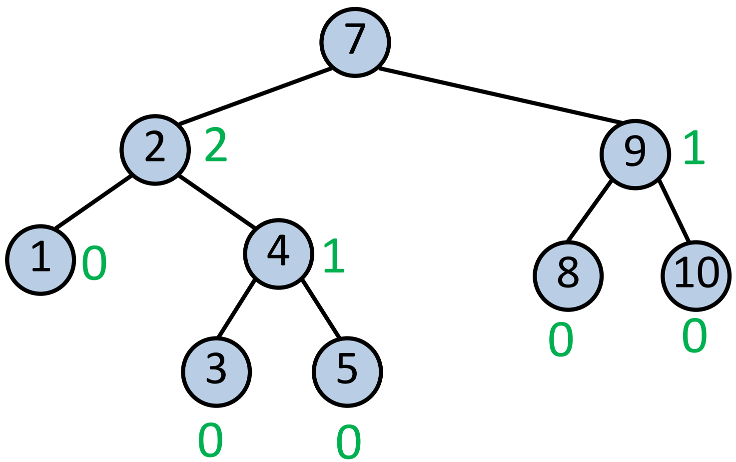
$$1 + \max\{\text{height}(v.\text{left}), \text{height}(v.\text{right})\}$$

Outline

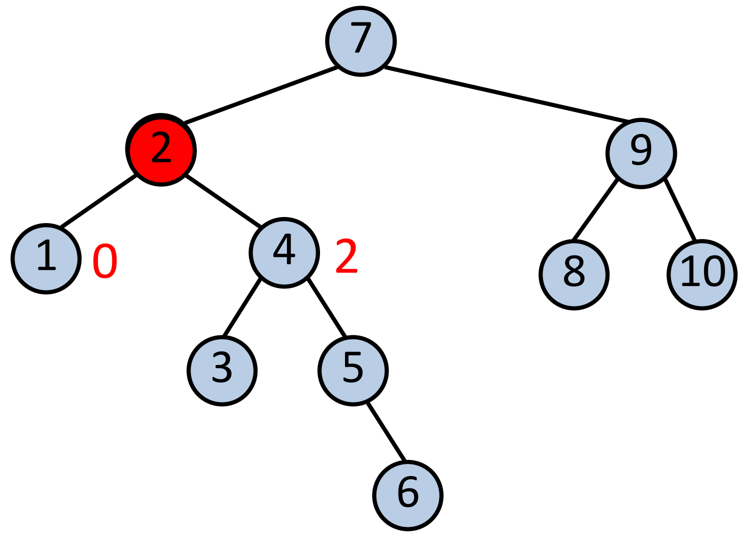
- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - full code for insertion
    - deletion

AVL Trees

- **A**delson-**V**elski and **L**andis, 1962
- *AVL Tree* is a BST with **height-balance** property
  - for any node v, heights of its left and right subtrees differ by at most 1

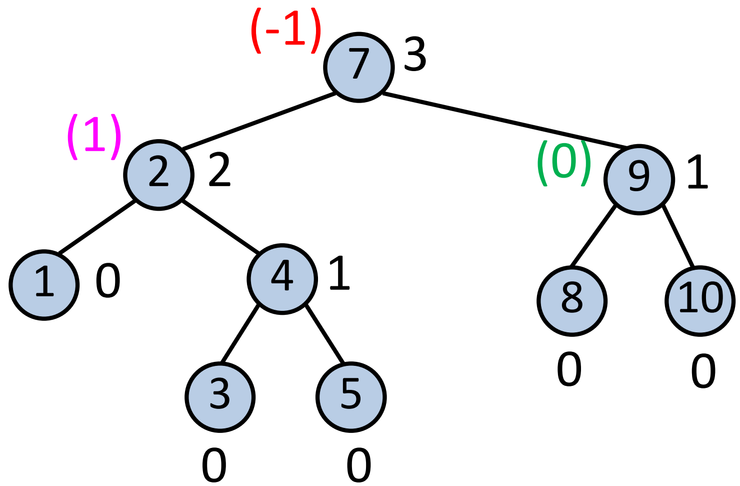


AVL Tree



not AVL Tree

AVL Trees



- AVL Tree is a BST with **height-balance** property
  - for any node v, heights of its left and right subtrees differ by at most 1
  - in other words, $height(v.right) - height(v.left) \in \{-1, 0, 1\}$
    - -1 means v is *left-heavy*
    - 0 means v is *balanced*
    - $+1$ means v is *right-heavy*
- Need to store at each node v its height
  - enough to store **balance factor** $= height(v.right) - height(v.left)$
    - fewer bits
    - but code more complicated, especially for deleting
    - no details

Height of an AVL tree

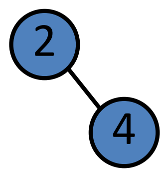
Theorem: AVL tree on n nodes has $\Theta(\log n)$ height

Proof:

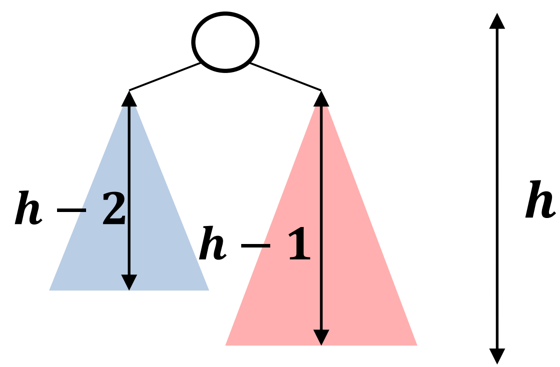
- Only need upper bound, as height is $\Omega(\log n)$
- Let $N(h)$ be the *smallest* number of nodes an AVL tree of height h can have
  - *any* AVL tree of height h has number of nodes $n \geq N(h)$

$N(0)$

$N(1)$



$N(h)$



- For $h \geq 2$

$$N(h) = N(h-1) + N(h-2) + 1 \geq N(h-2) + N(h-2) = 2N(h-2)$$

- Thus $N(h) \geq 2N(h-2)$
  - number of nodes doubles every two levels ⇒ exponential growth

Height of an AVL tree

Proof: (continued)

- $N(h)$ is the *least* number of nodes in height-h AVL tree
  - any AVL tree of height h has number of nodes $n \geq N(h)$
- $N(0) = 1, N(1) = 2$ and $N(h) \geq 2N(h-2)$ for $h \geq 2$ and
- Keep expanding until the base case

$$N(h) \geq 2N(h-2) \geq 2^2 N(h-2 \cdot 2) \geq 2^3 N(h-2 \cdot 3) \geq \cdots \geq 2^i N(h-2 \cdot i)$$

case 1: odd h

- expand until $h - 2 \cdot i = 1$
- rewriting, $i = (h-1)/2$

$$N(h) \geq 2^{(h-1)/2} N(1) = 2^{\frac{h-1}{2}} \cdot 2$$

- take log

$$\log N(h) \geq \frac{h-1}{2} + 1$$

- rearrange

$$h \leq 2\log N(h) - 2 \leq 2\log n - 2$$

case2: even h

- expand until $h - 2 \cdot i = 0$
- rewriting, $i = h/2$

$$N(h) \geq 2^{h/2} N(0) = 2^{\frac{h}{2}} \cdot 1$$

- take log

$$\log N(h) \geq \frac{h}{2}$$

- rearrange

$$h \leq 2\log N(h) \leq 2\log n$$

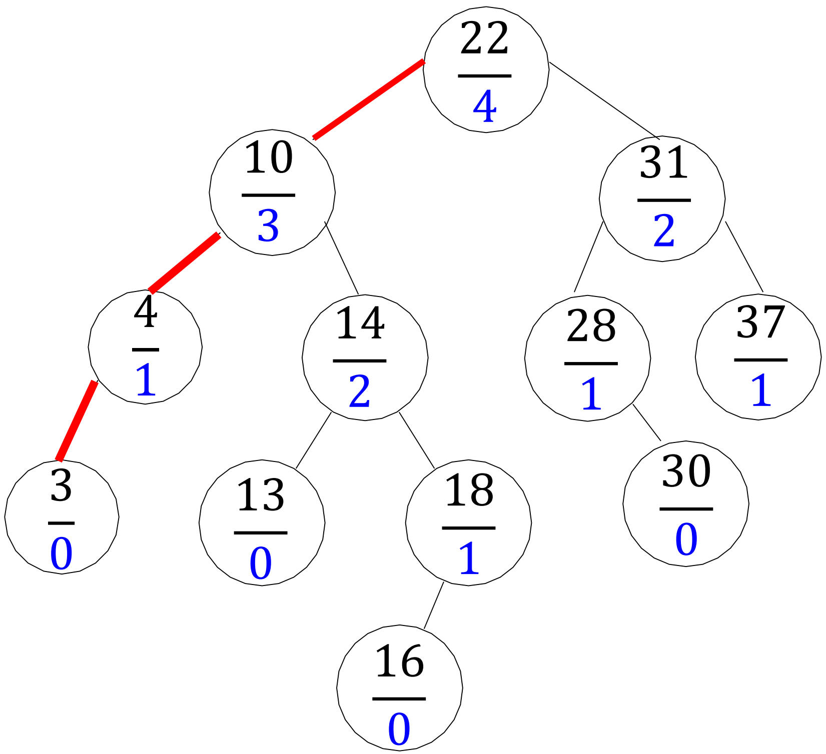
- In both cases, h is $O(\log n)$

Outline

- Dictionaries and Balanced Search Trees
    - Dictionary ADT
    - Review: Binary Search Trees
    - AVL Trees
        - insertion
        - restoring the AVL Property: Rotations
        - full code for insertion
        - deletion

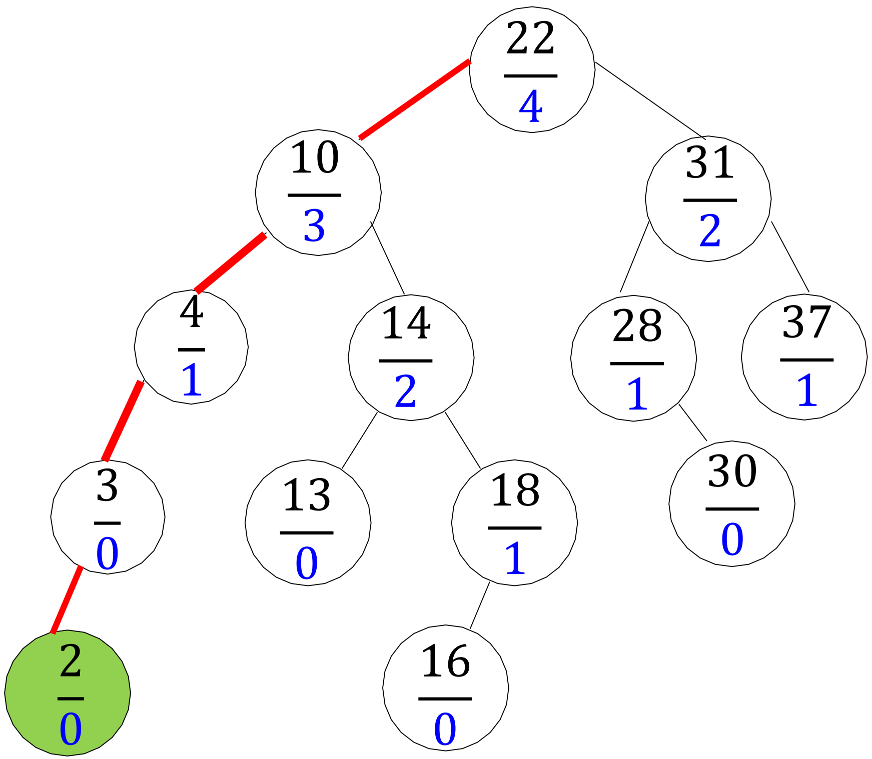
AVL Insertion Example

Example: *AVL::insert*(2)



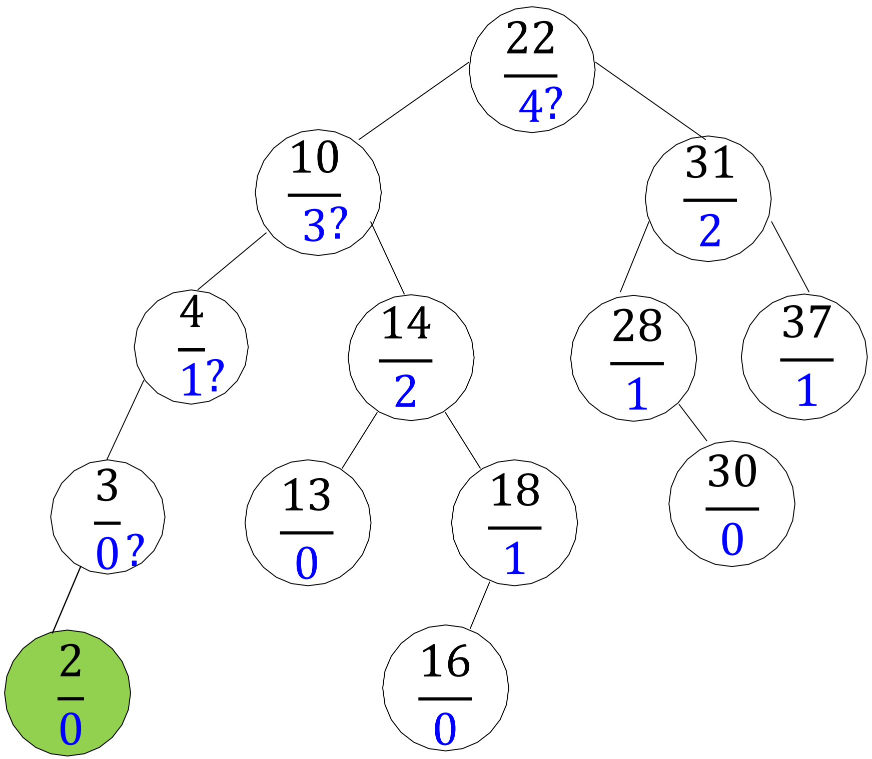
AVL Insertion Example

Example: *AVL::insert*(2)



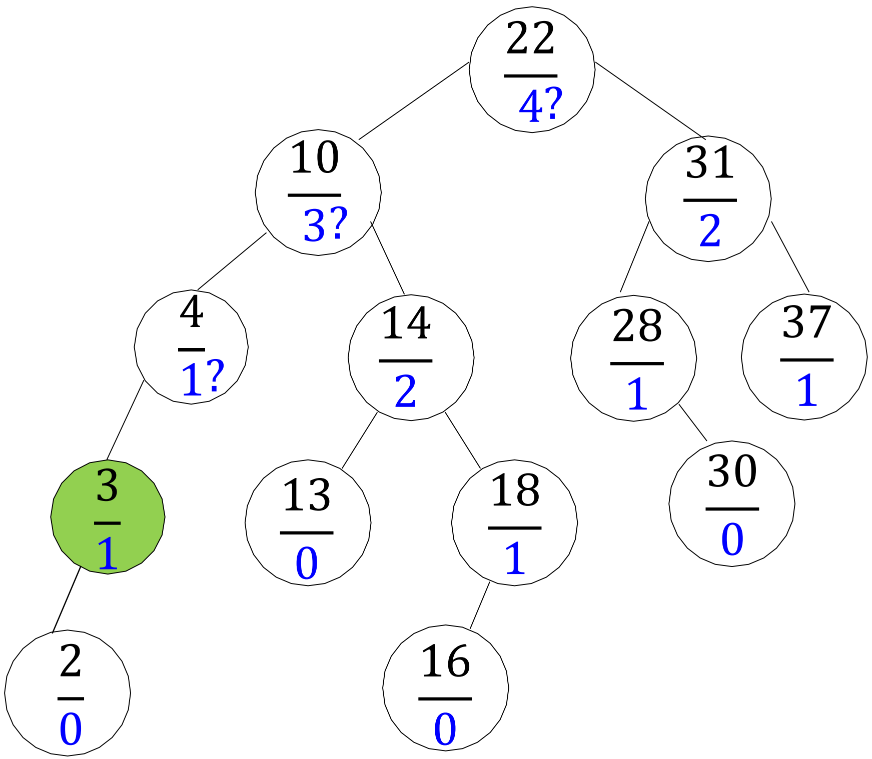
AVL Insertion Example

Example: *AVL::insert*(2)



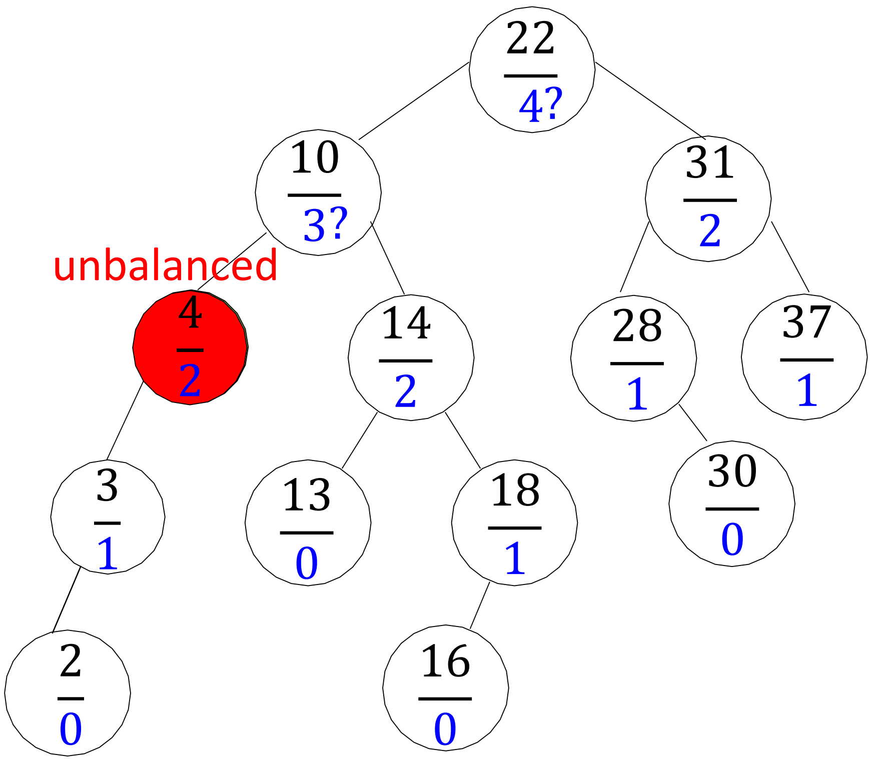
AVL Insertion Example

Example: *AVL::insert*(2)



AVL Insertion Example

Example: *AVL::insert*(2)



AVL insertion

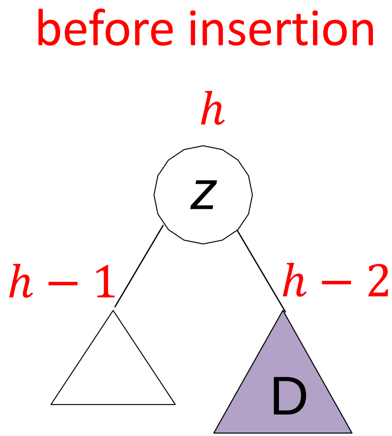
- *AVL::insert*(T, k, v)
  1. insert (k, v) into T with the usual BST insertion
     - assume insert returns new *leaf* where the key was inserted
     - heights of nodes on path from this *leaf* to root may have increased
       - by at most 1
  2. move up from the new *leaf* to the root, updating heights
     - either use parent-links, or BST::insert could return the path
  3. if the height difference becomes ± 2 for some node on this path, the node is *unbalanced*
     - must re-structure the tree to restore height-balance property

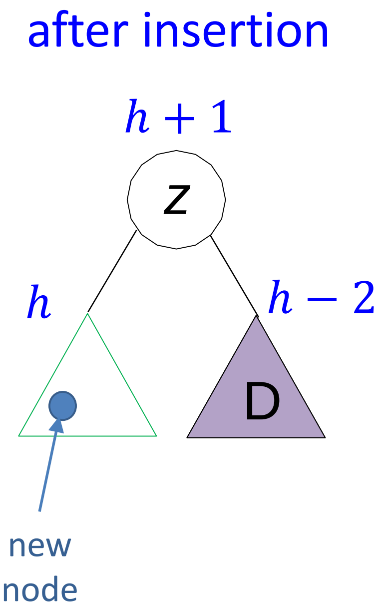
Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - full code for insertion
    - deletion

Restoring Height After Insertion

- Let *z* be *the first* unbalanced node on path from inserted node to root

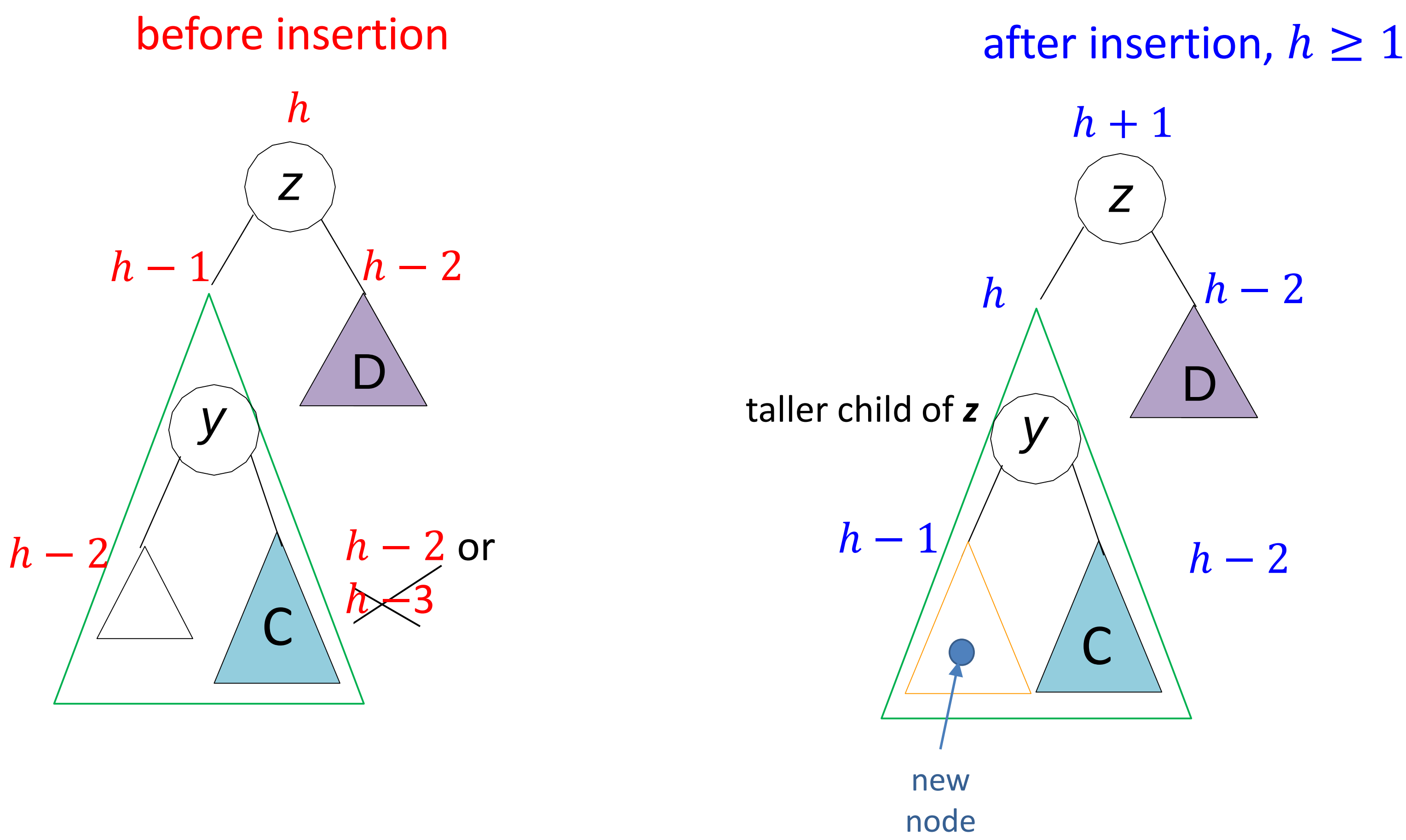




$$h - 2 \geq -1 \;\; \Rightarrow h \geq 1$$

Restoring Height After Insertion

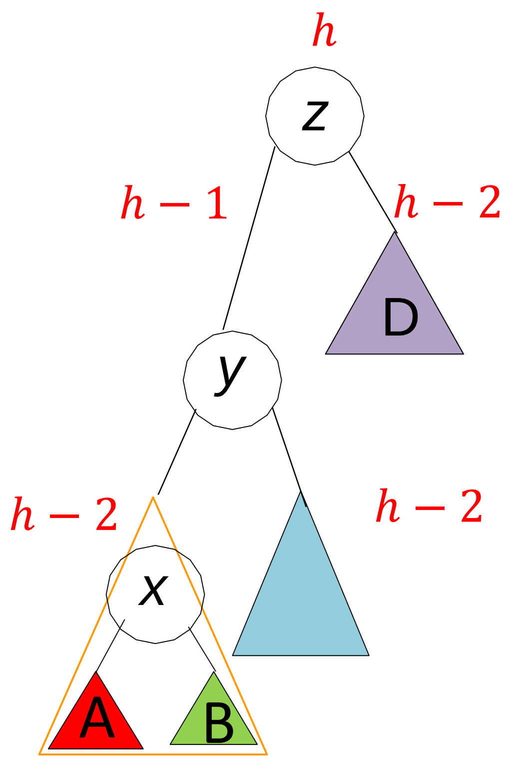
- Let z be *the first* unbalanced node on path from inserted node to root



Restoring Height After Insertion

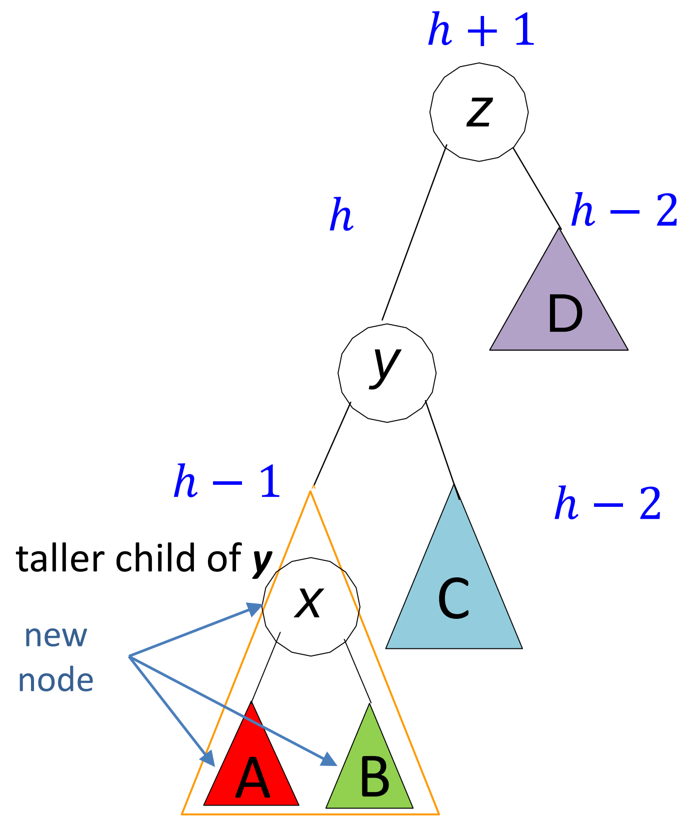
- Let z be *the first* unbalanced node on path from inserted node to root

before insertion



case $h > 1$:
both A,B have height $h-3$

after insertion, $h \geq 1$

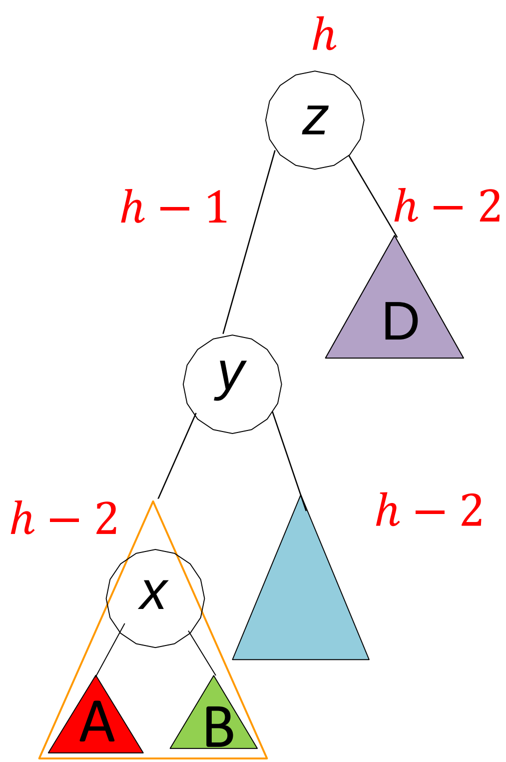


case $h = 1$: x = new node; A, B have height $= -1 = h-2$

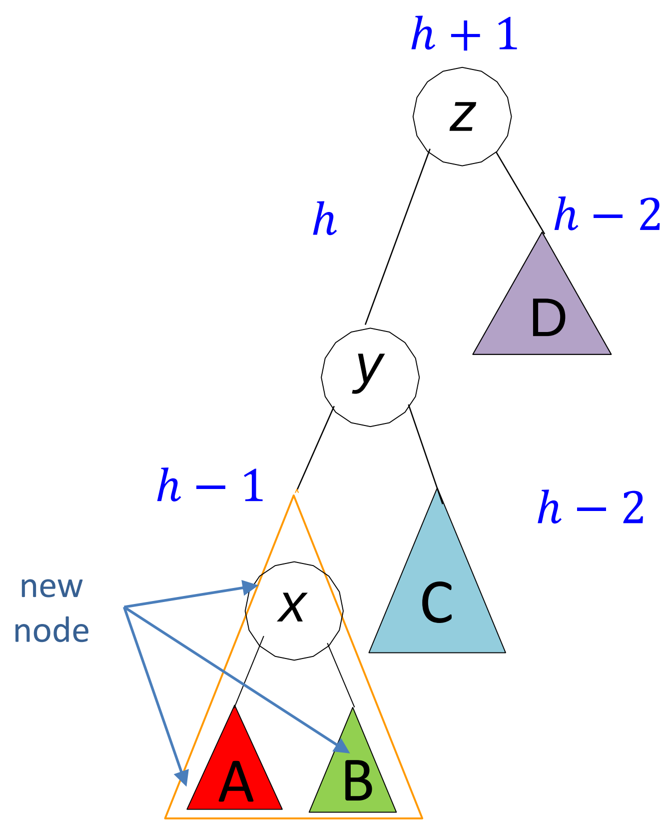
case $h > 1$: $x \neq$ new node; one of A,B has height $h-2$, another $h-3$

Restoring Height After Insertion

- Let *z* be *the first* unbalanced node on path from inserted node to root



case $h > 1$:
both A,B have height $h - 3$

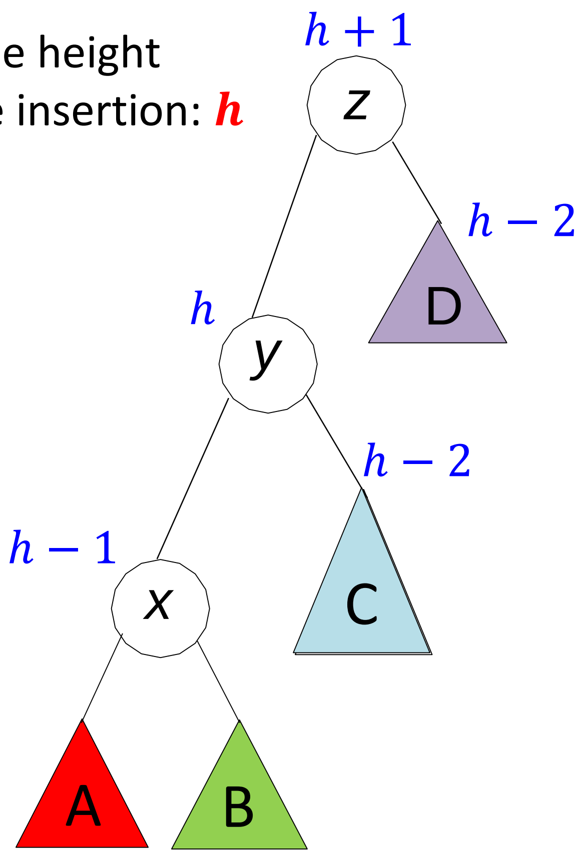


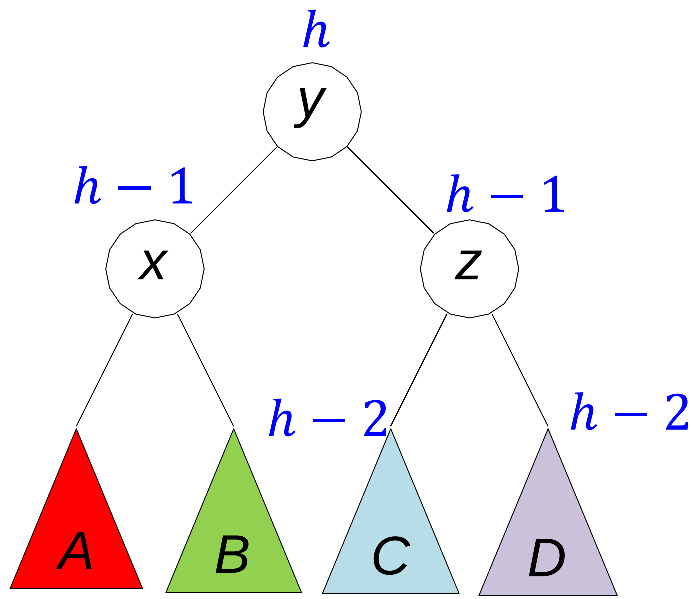
$h - 2$ or $h - 3$

Restoring Height: Right Rotation

- Let z be the first unbalanced node on path from inserted node to the root
- Right rotation is used for **left-left** imbalance (taller **left** child and **left** grandchild)

subtree height
before insertion: h

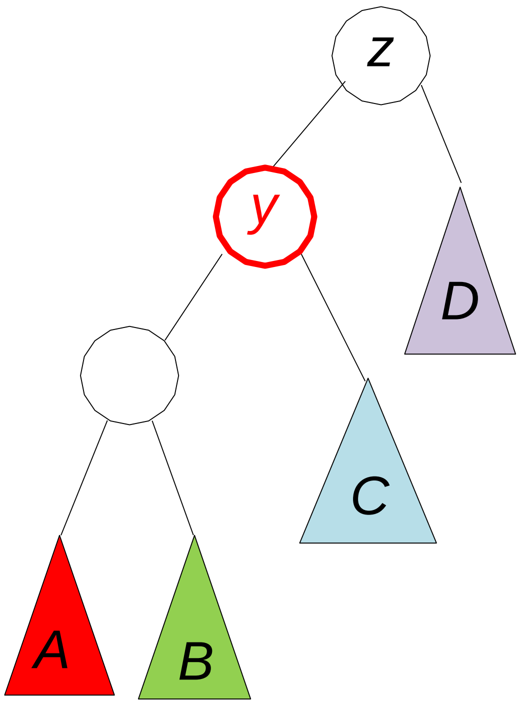




- BST order is preserved
- Balanced
- Same subtree height h as before insertion

Right Rotation Pseudocode

- Right rotation on node z



rotate-right(z)

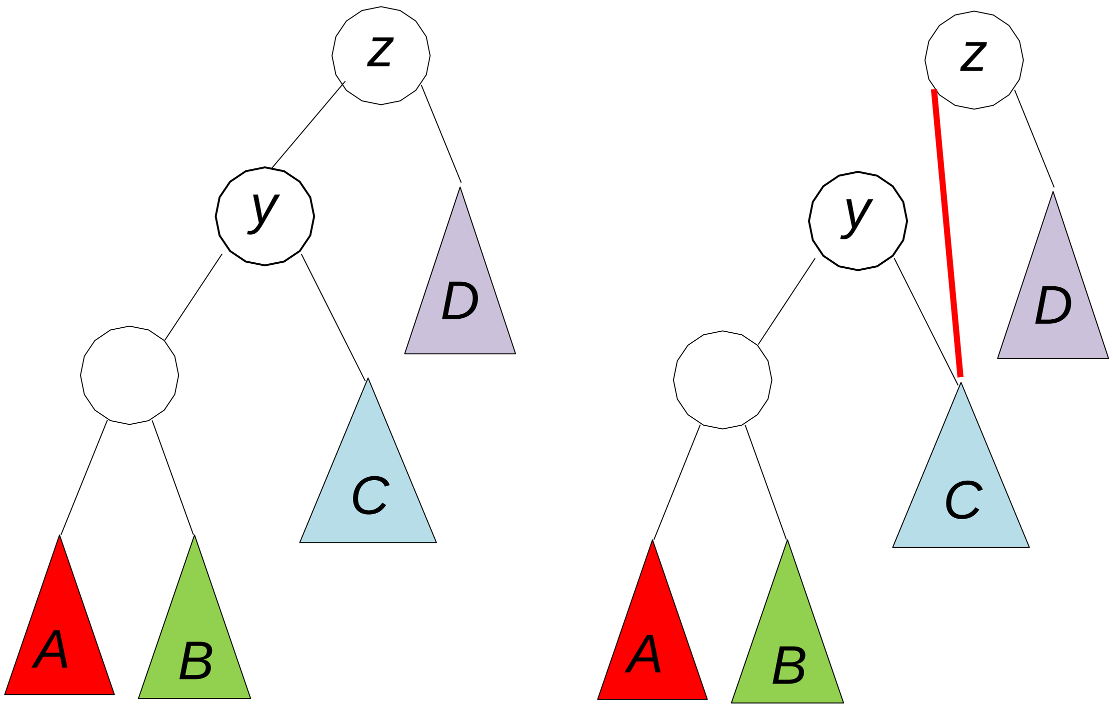
$y \leftarrow z.left$, $z.left \leftarrow y.right$, $y.right \leftarrow z$

setHeightFromChildren(z), *setHeightFromChildren*(y)

return y // returns new root of subtree

Right Rotation Pseudocode

- Right rotation on node z



rotate-right(z)

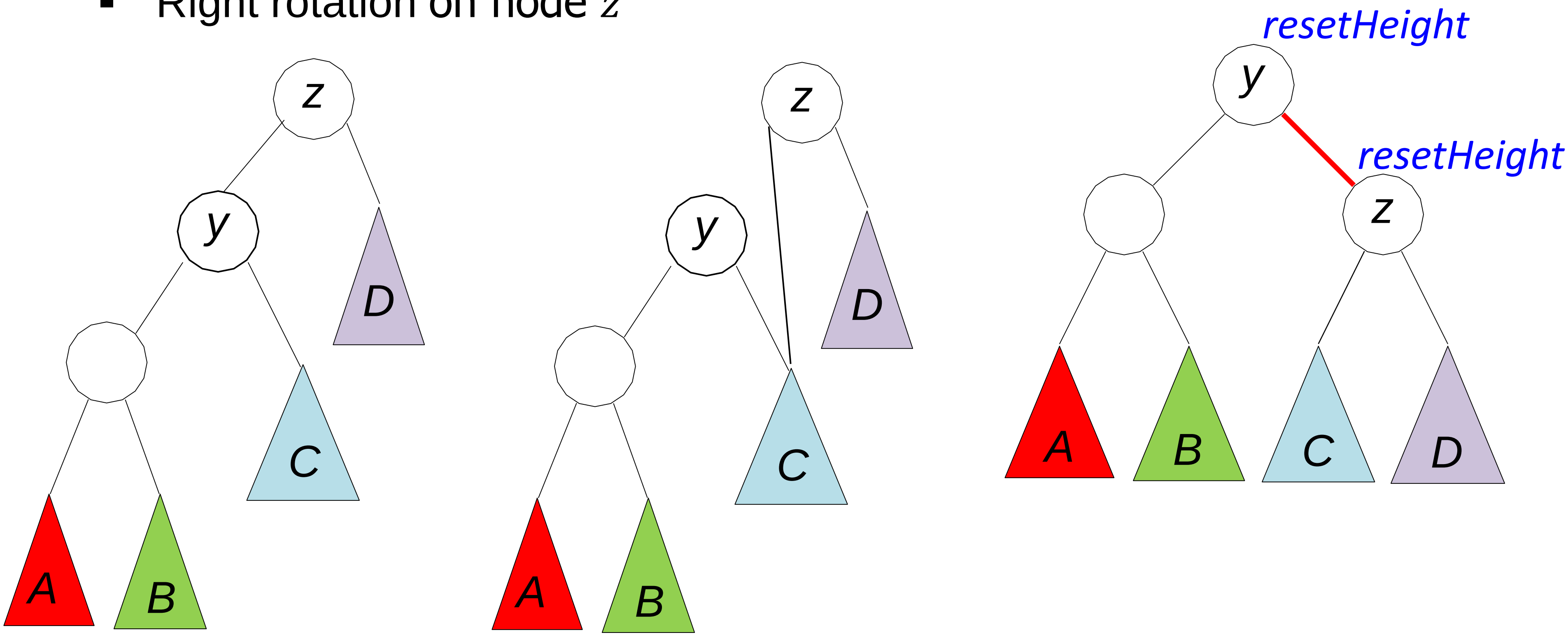
$y \leftarrow$ *z.left*, ***z.left*** $\leftarrow$ ***y.right***, *y.right* $\leftarrow z$

setHeightFromChildren(z), *setHeightFromChildren*(y)

return y // returns new root of subtree

Right Rotation Pseudocode

- Right rotation on node z



rotate-right(z)

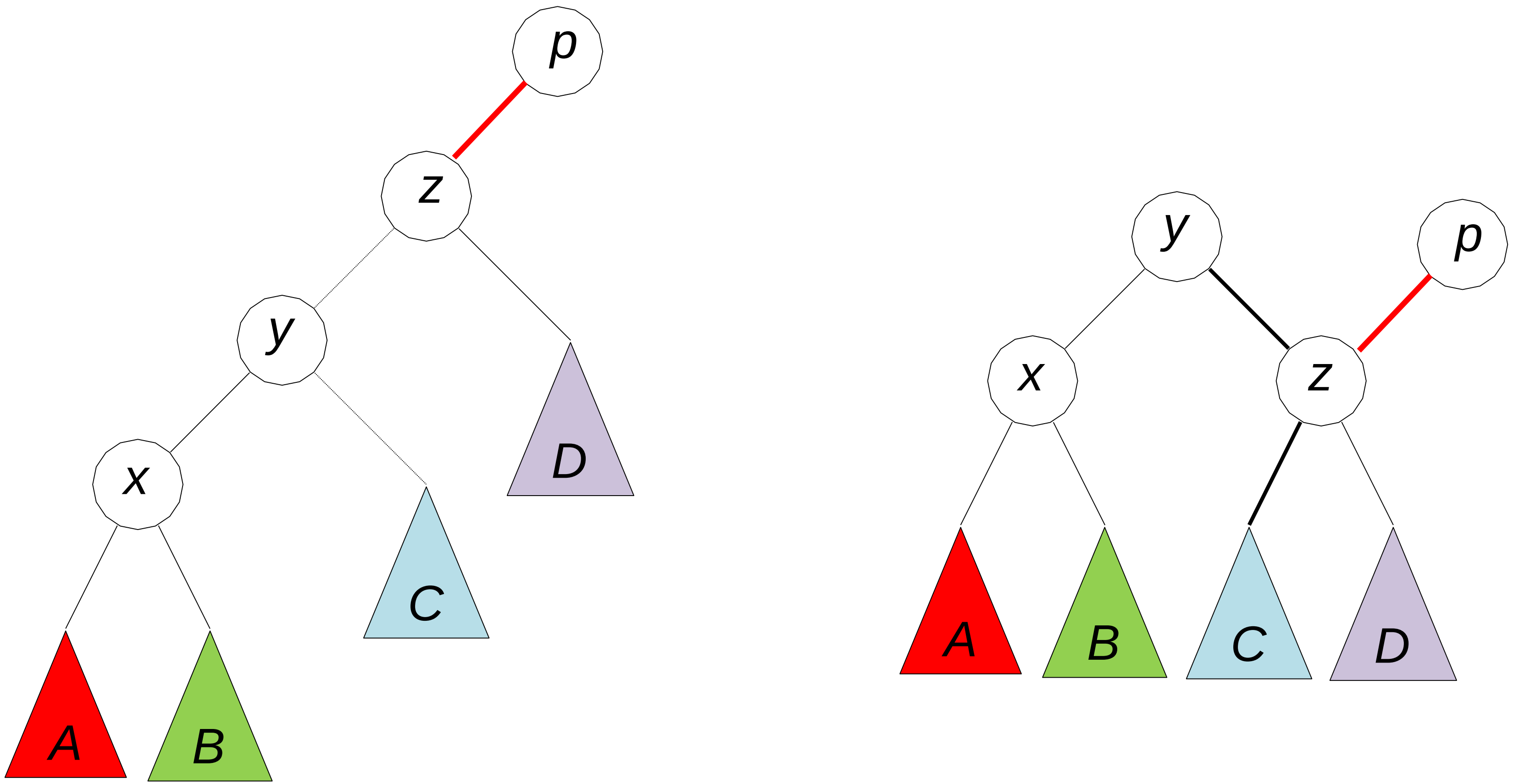
$y \leftarrow z.left$, $z.left \leftarrow y.right$, **$y.right \leftarrow z$**

setHeightFromChildren(z), *setHeightFromChildren*(y)

return y // returns new root of subtree

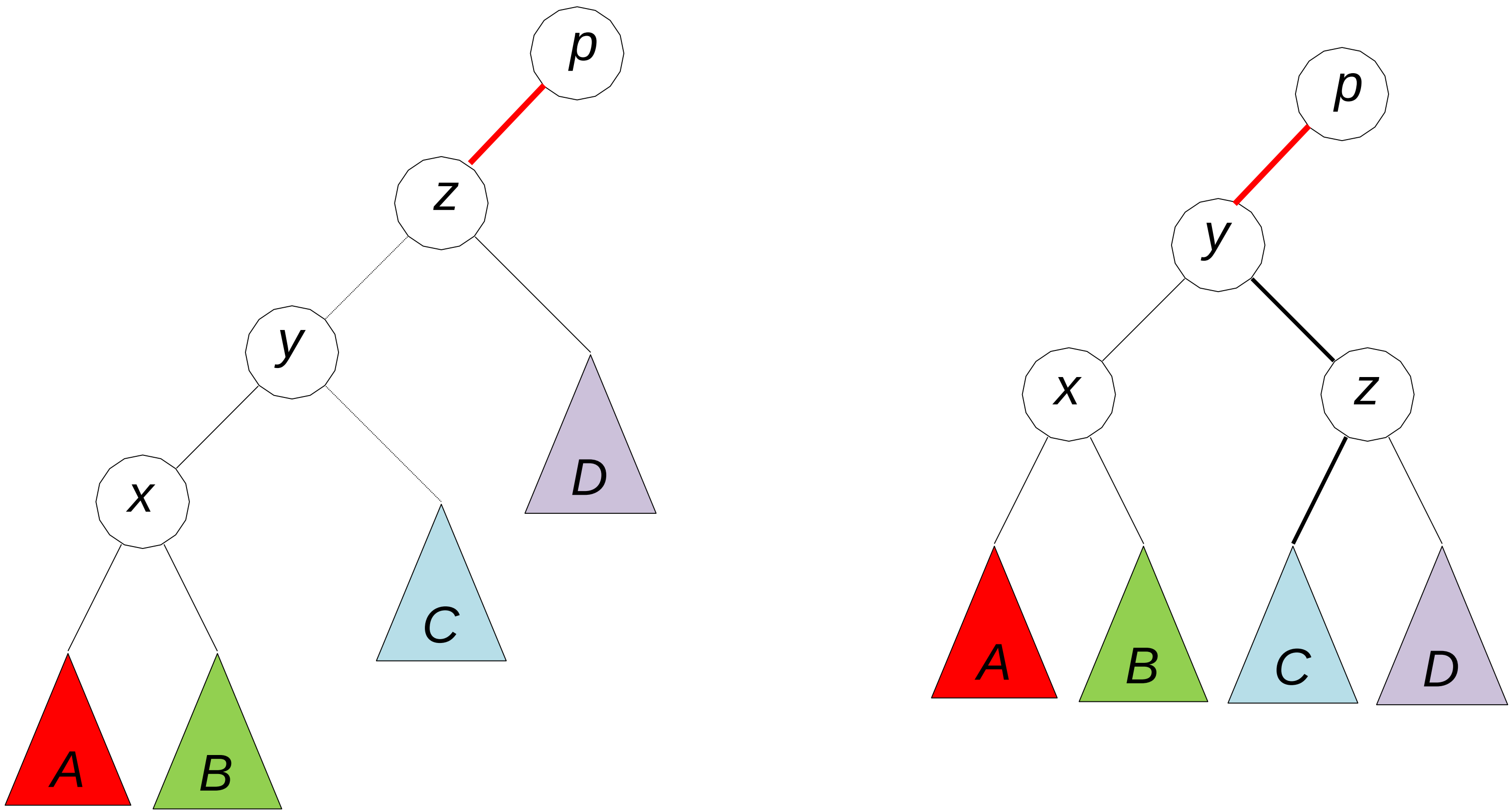
After Rotation:

- If z had a parent p, need to set y as the new child of p



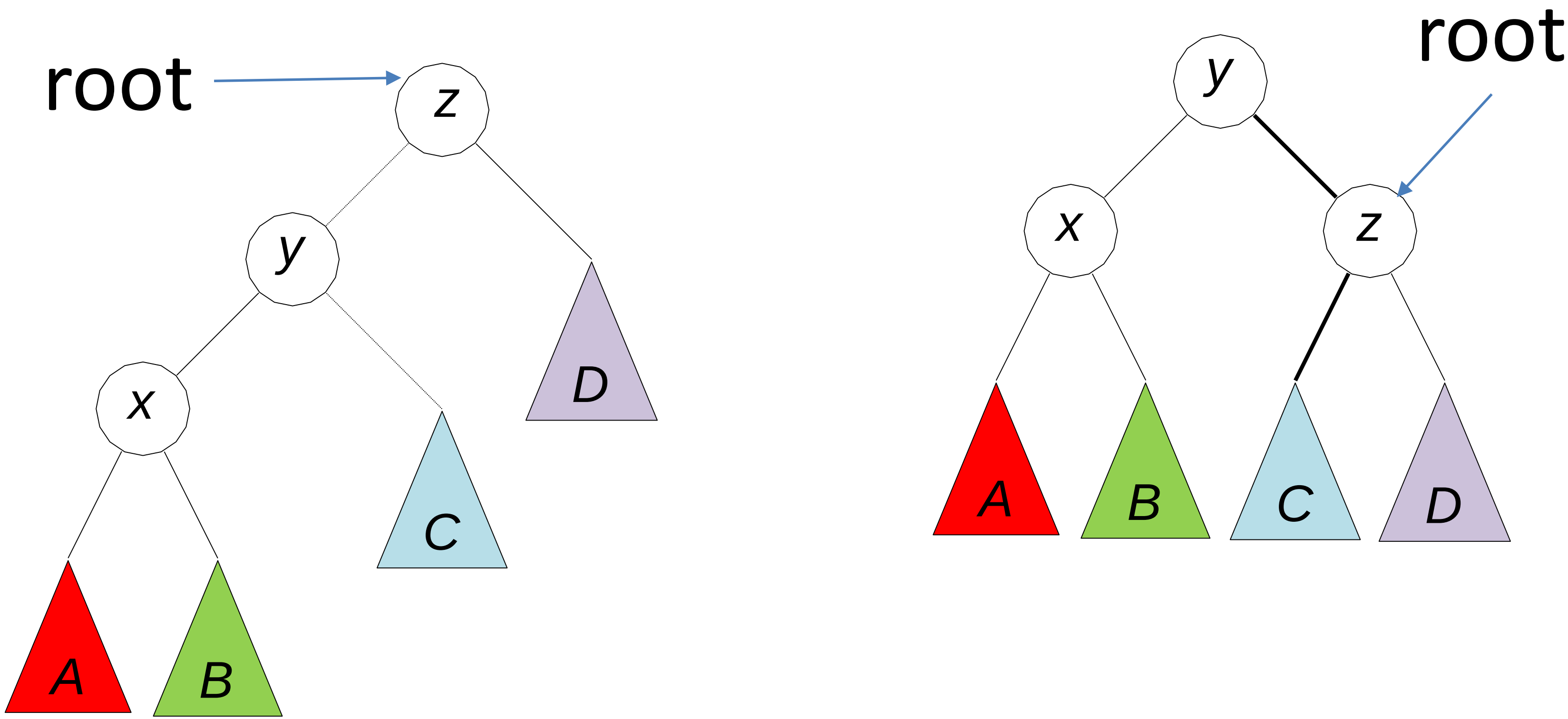
After Rotation:

- If z had a parent p, need to set y as the new child of p



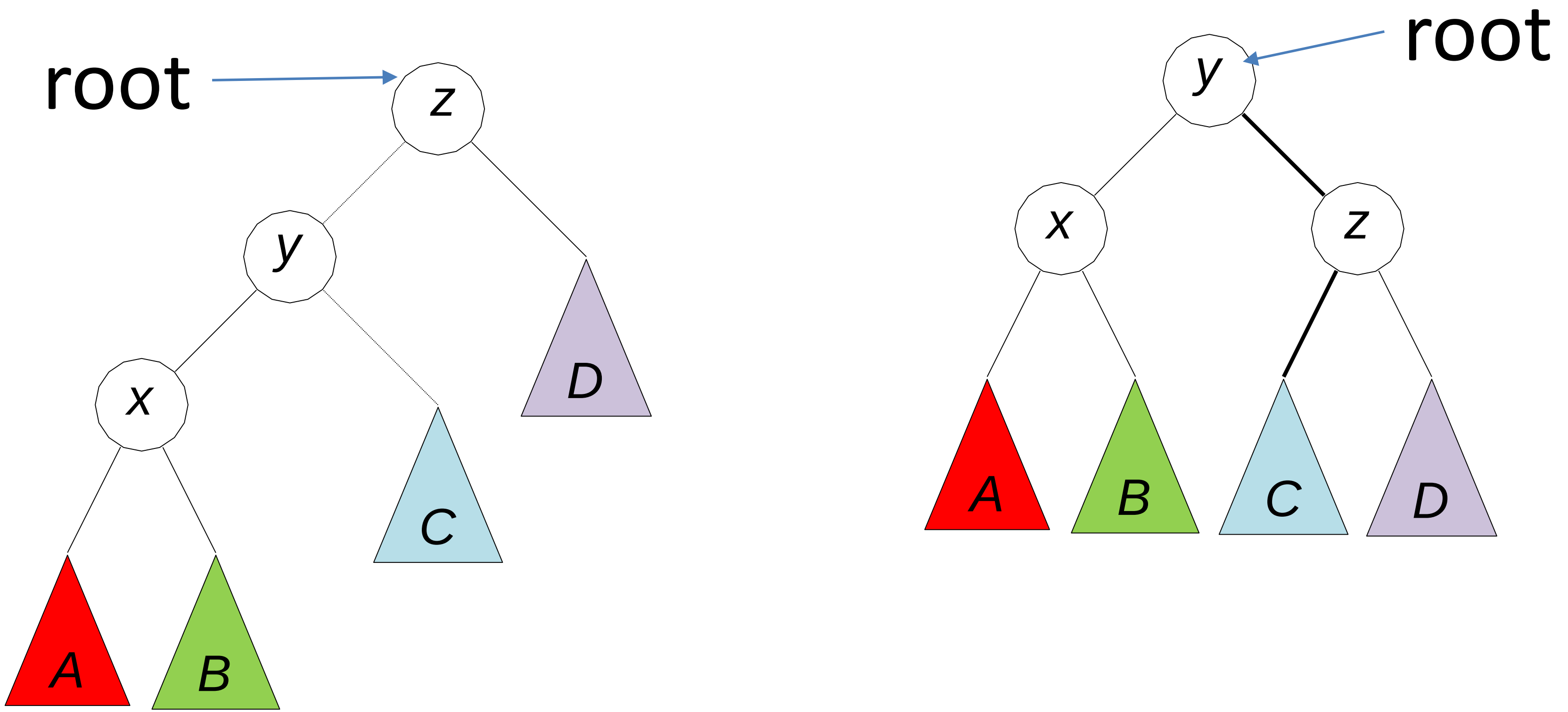
After Rotation:

- If node z was the tree root, then y becomes new tree root

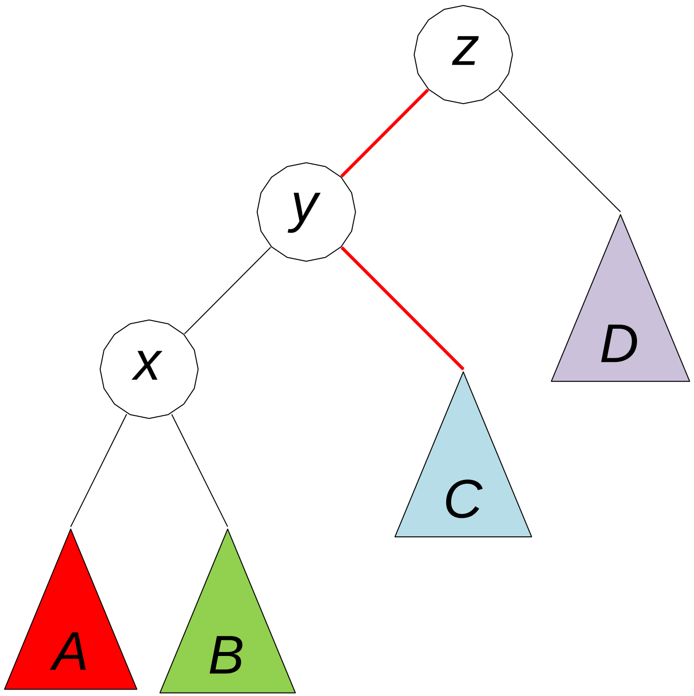


After Rotation:

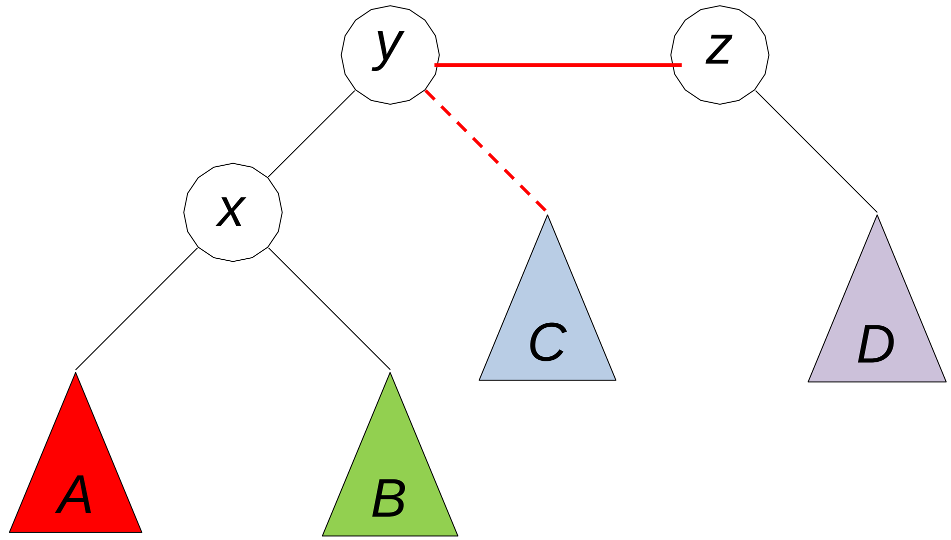
- If node z was the tree root, then y becomes new tree root



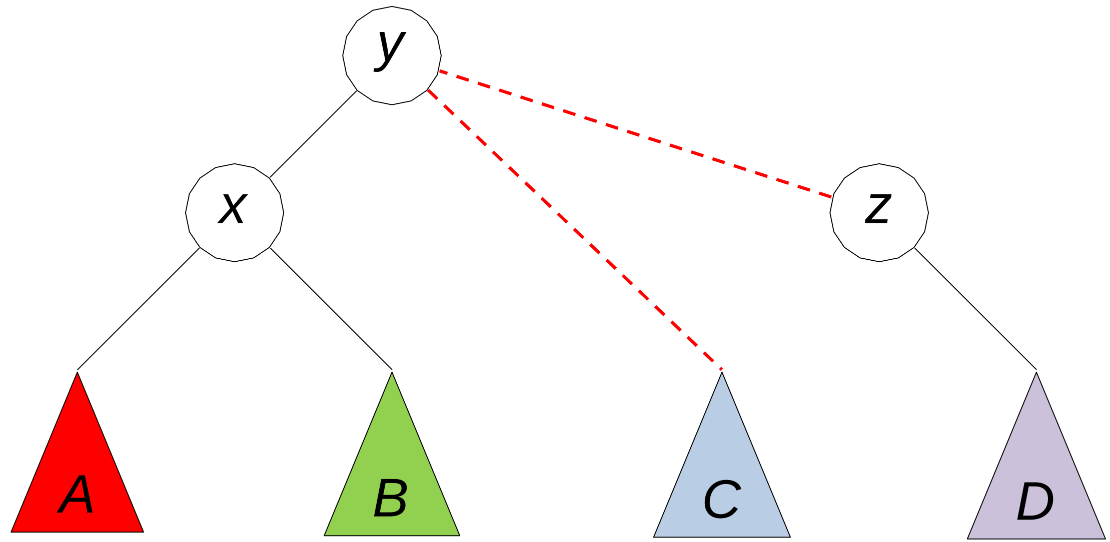
Why do we call this a rotation?



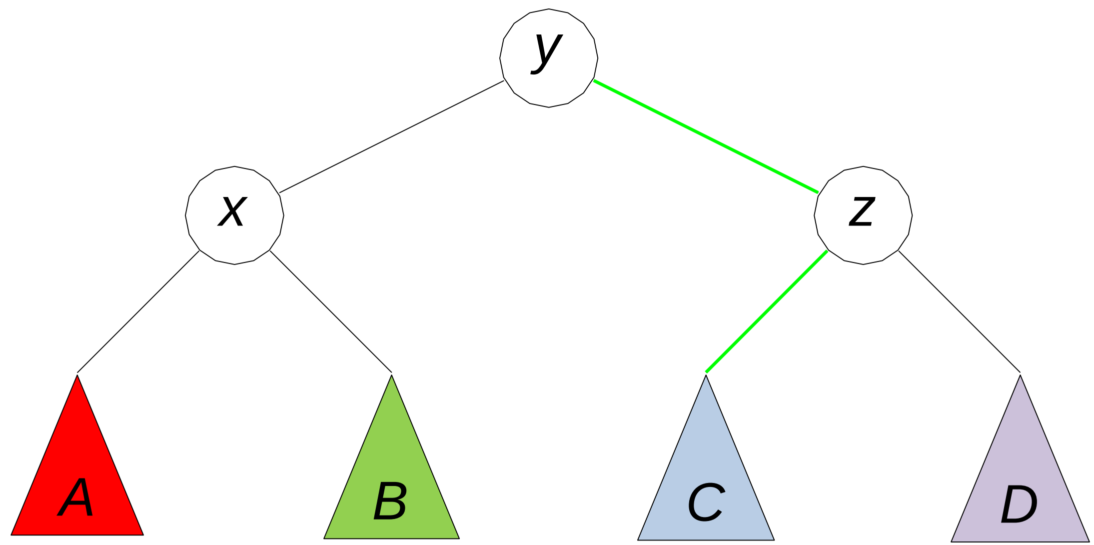
Why do we call this a rotation?



Why do we call this a rotation?

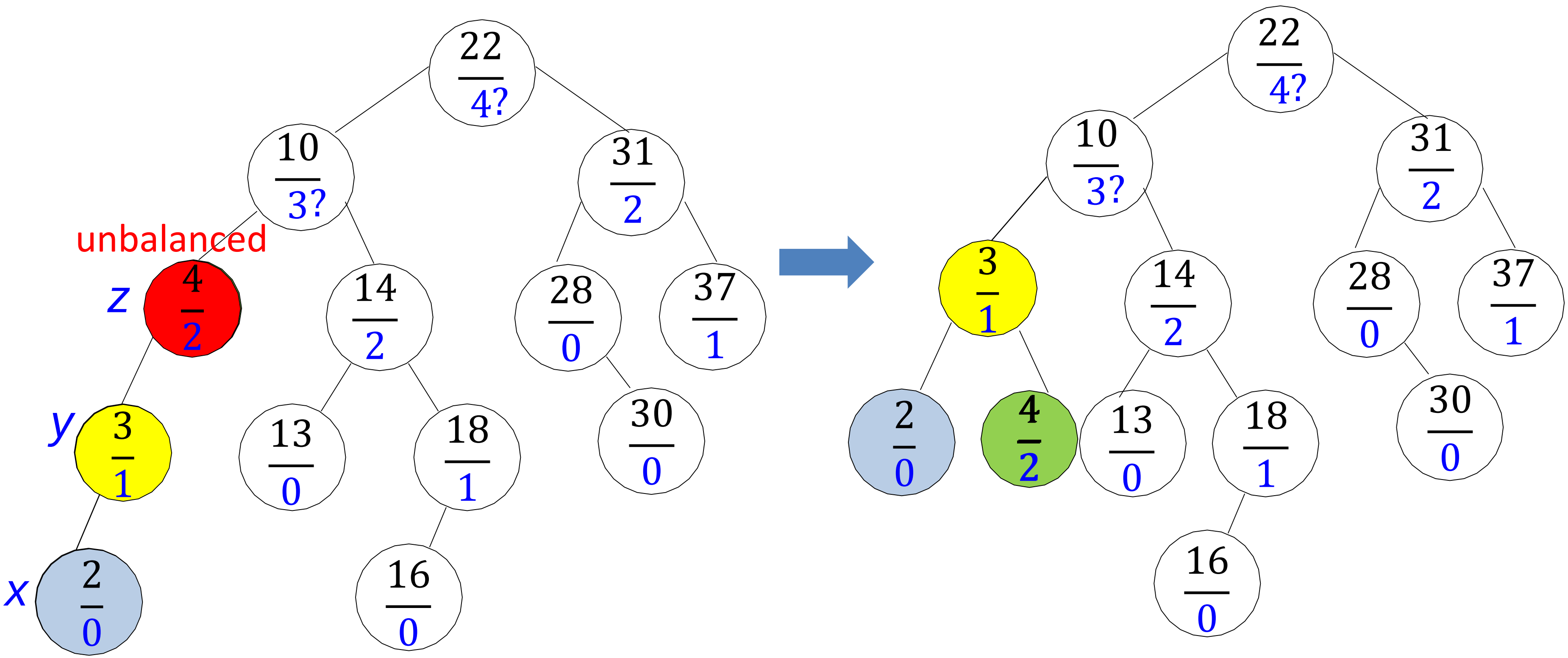


Why do we call this a rotation?



AVL Insertion Example

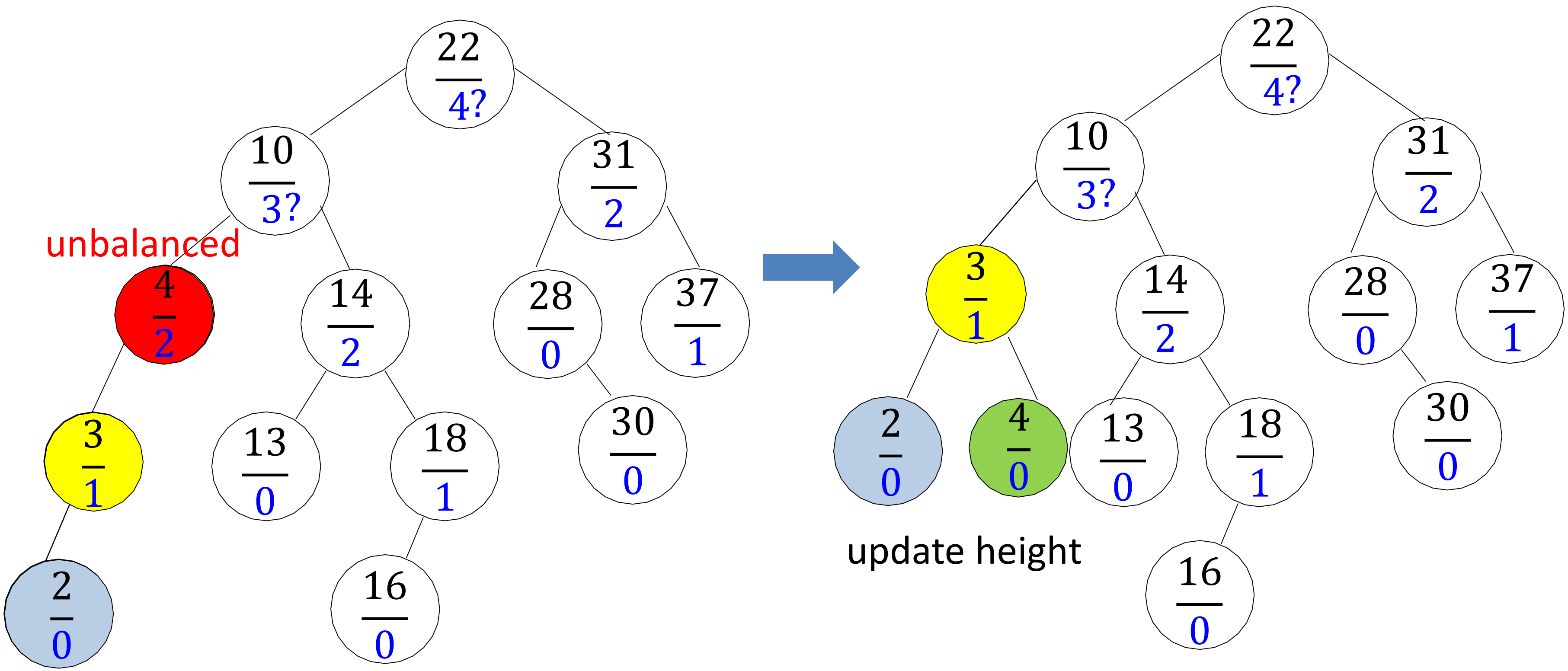
Example: *AVL::insert*(2)



- Fix with right rotation on node ***z***

AVL Insertion Example

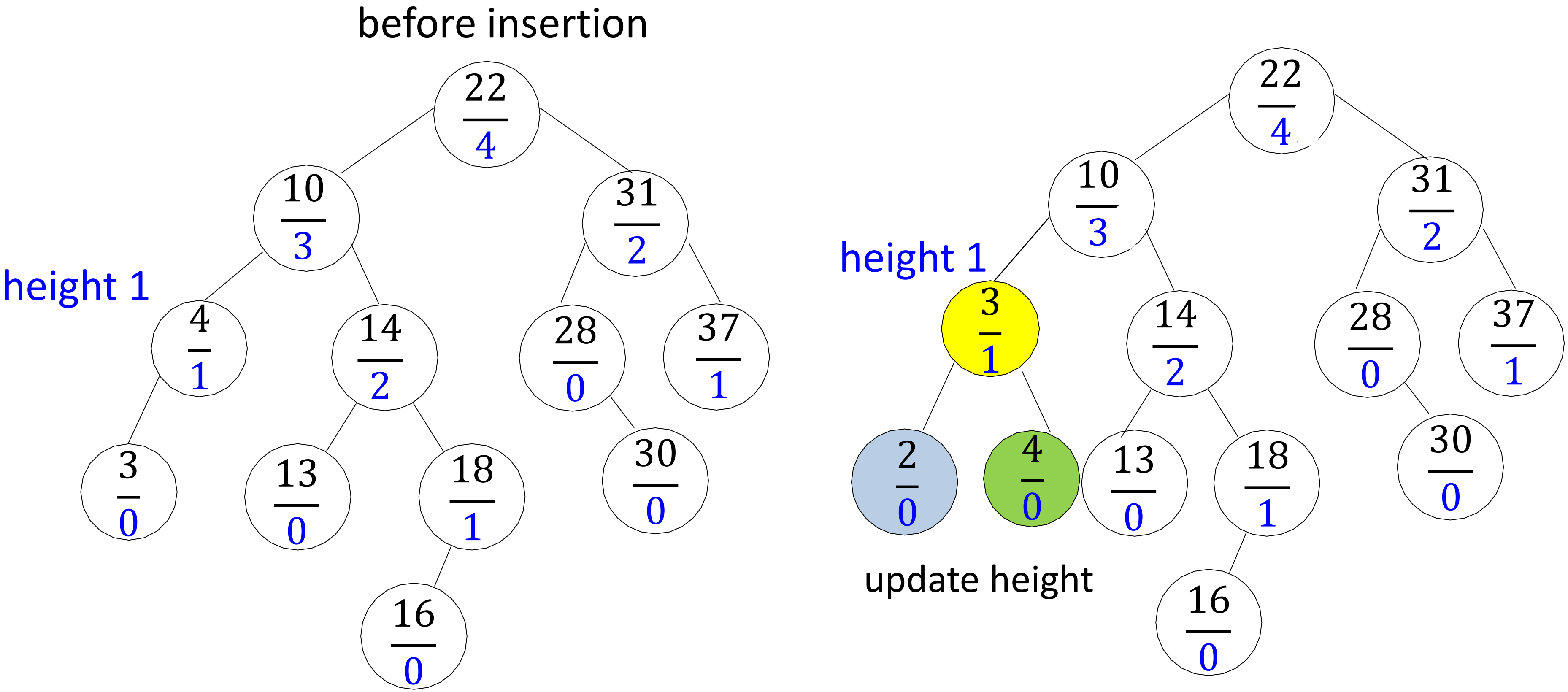
Example: *AVL::insert*(2)



- Fix with right rotation on node ***z***

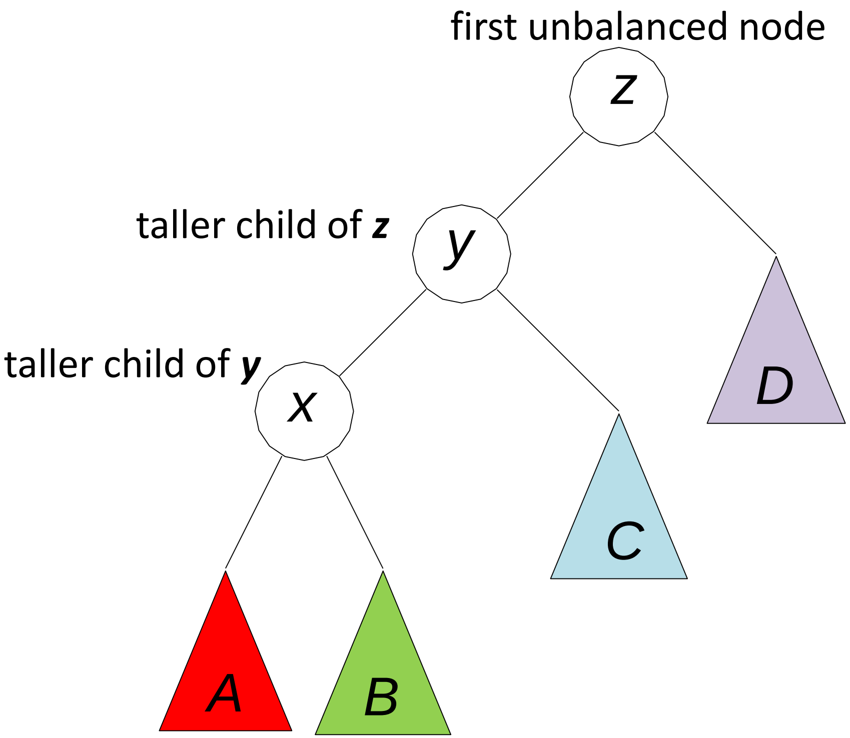
AVL Insertion Example

Example: *AVL::insert*(2)



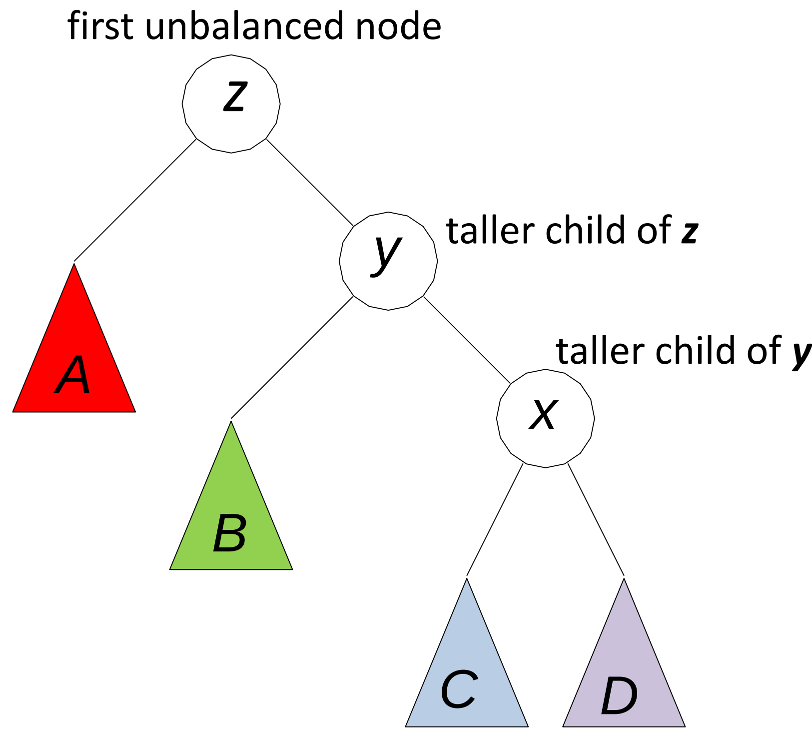
- After rotation all node heights are correct
  - can stop traversing up

Restoring Height Balance, Case 2



Case 1: Fixed with right rotation

left-left imbalance

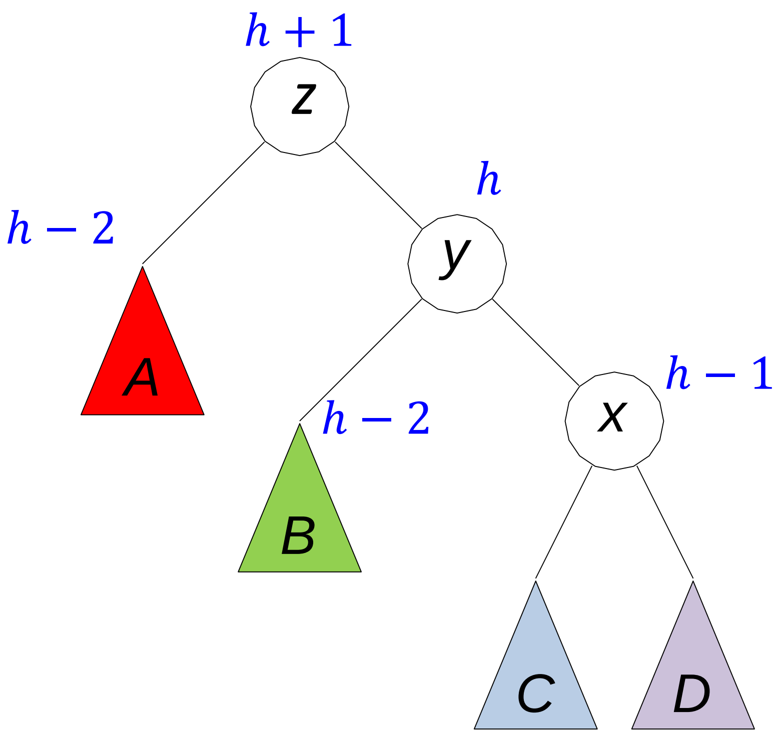


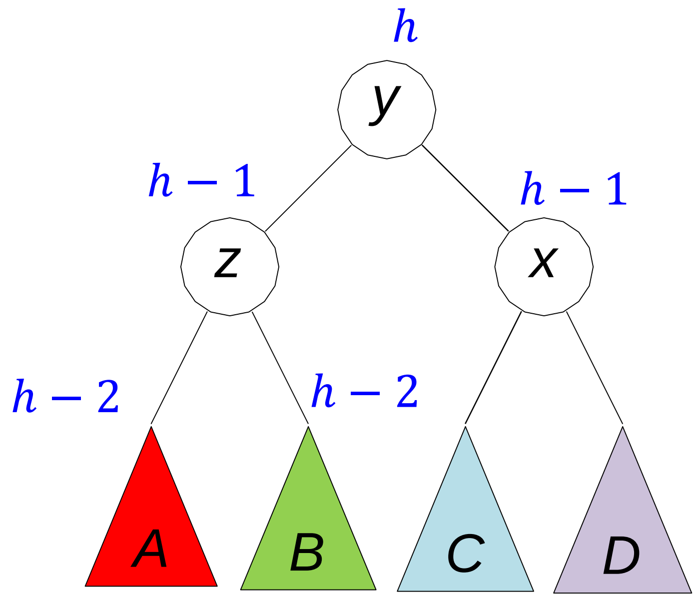
Case 2: Fixed with left rotation

right-right imbalance

Case 2: Left Rotation

- *Left rotation* on node *z* is symmetric to right rotation
- Used to fix right-right imbalance

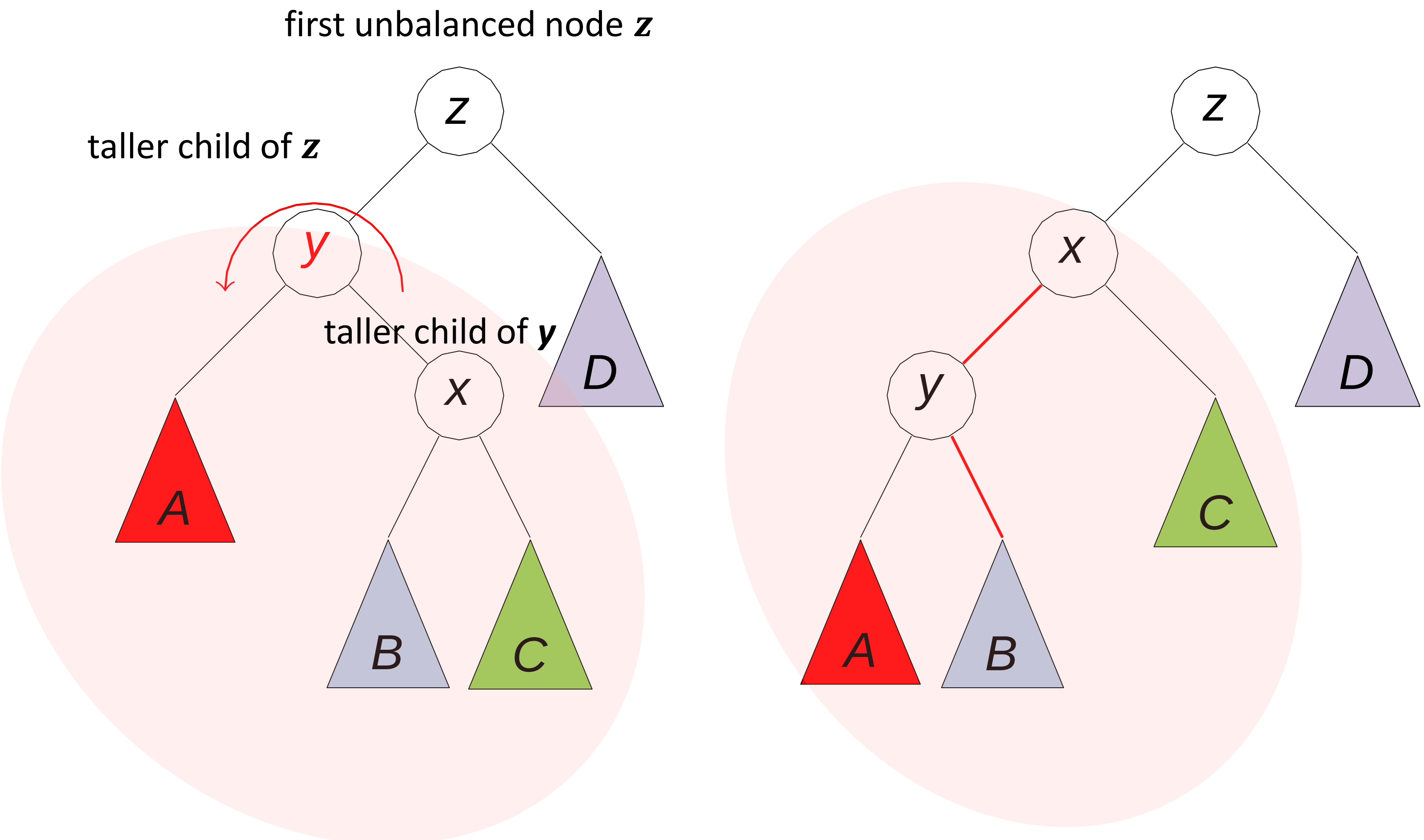




heights for case 2 are deduced exactly as for case 1

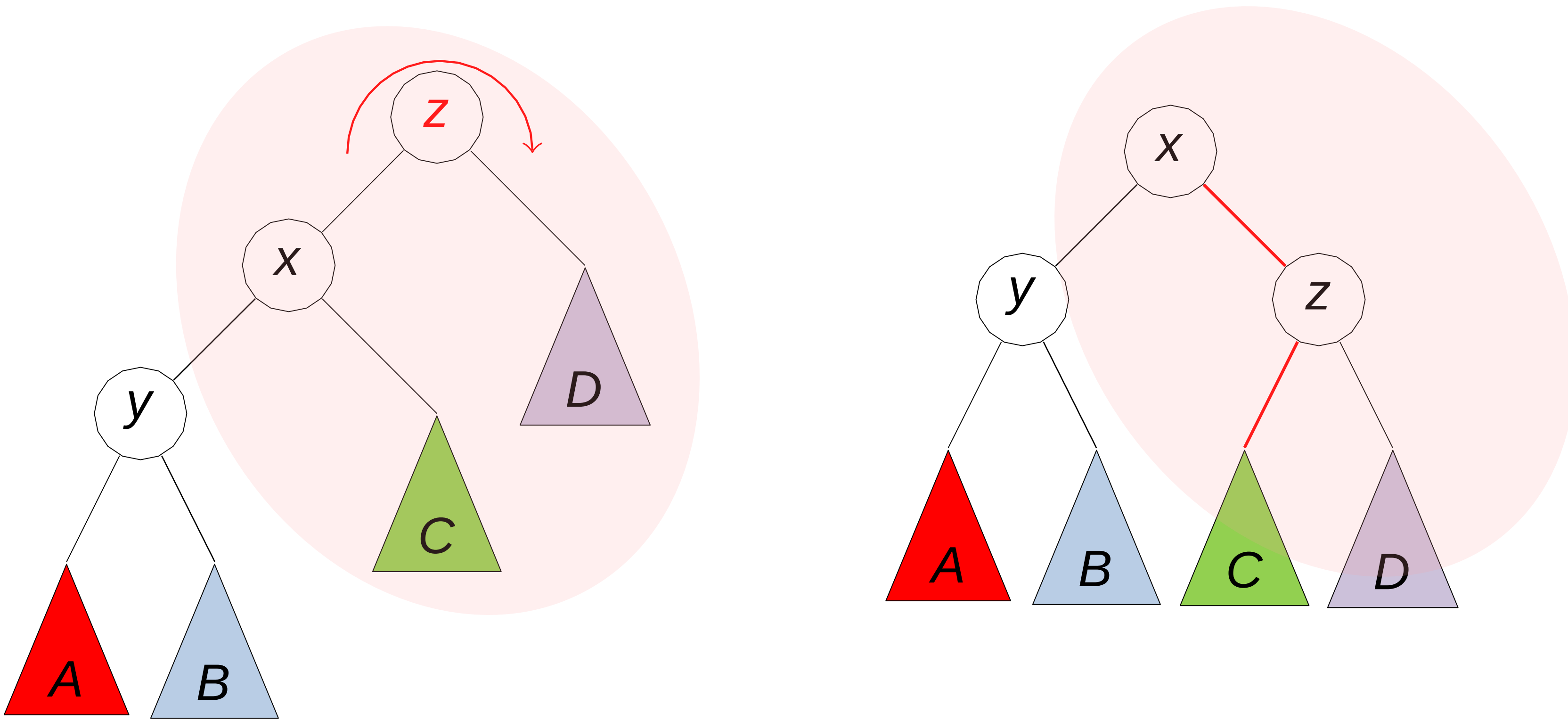
- BST order is preserved
- Balanced
- Same height as before insertion

Case 3: Left-Right imbalance



- Fix with double right rotation on node z
  - **first, left rotation at y**

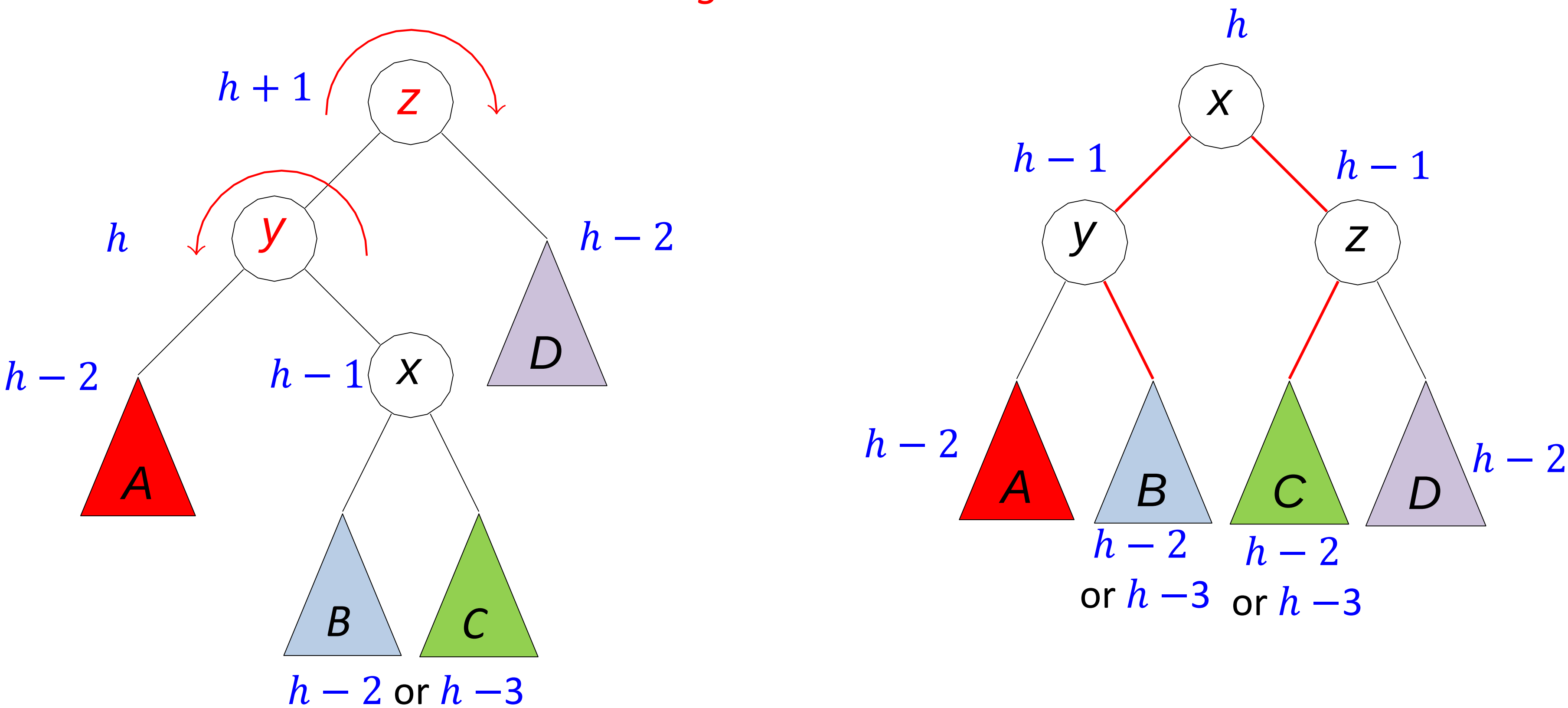
Case 3: Left-Right imbalance



- Fix with double rotation on node z
  - first, left rotation at y
  - **second, right rotation at z**

Case 3: Left-Right imbalance

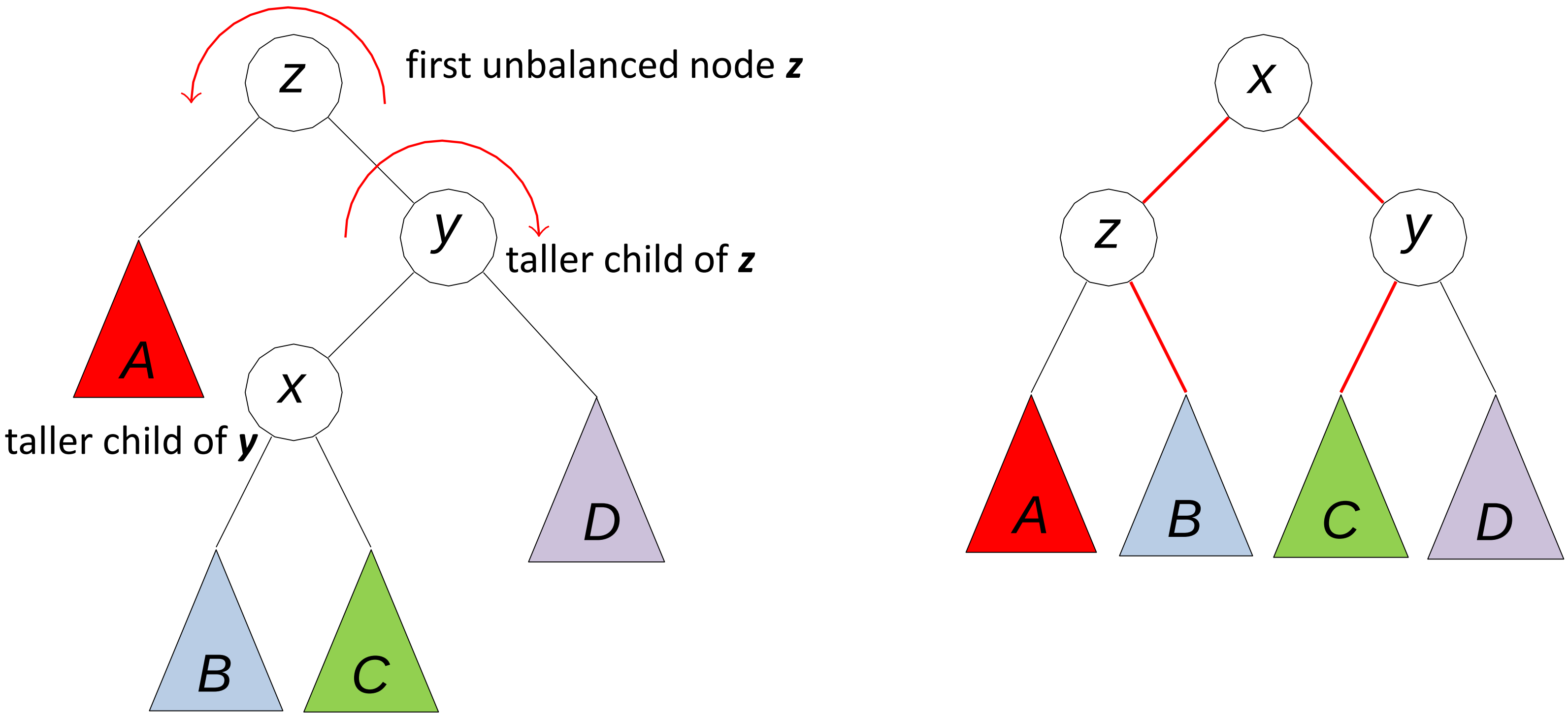
- Cumulative result of *double right rotation* on node z



- Left rotation at y, right rotation at z
- BST order is preserved
- Useful for left-right imbalance
    - can argue BST ordering is preserved, as before
    - can argue height balance property restored, as before

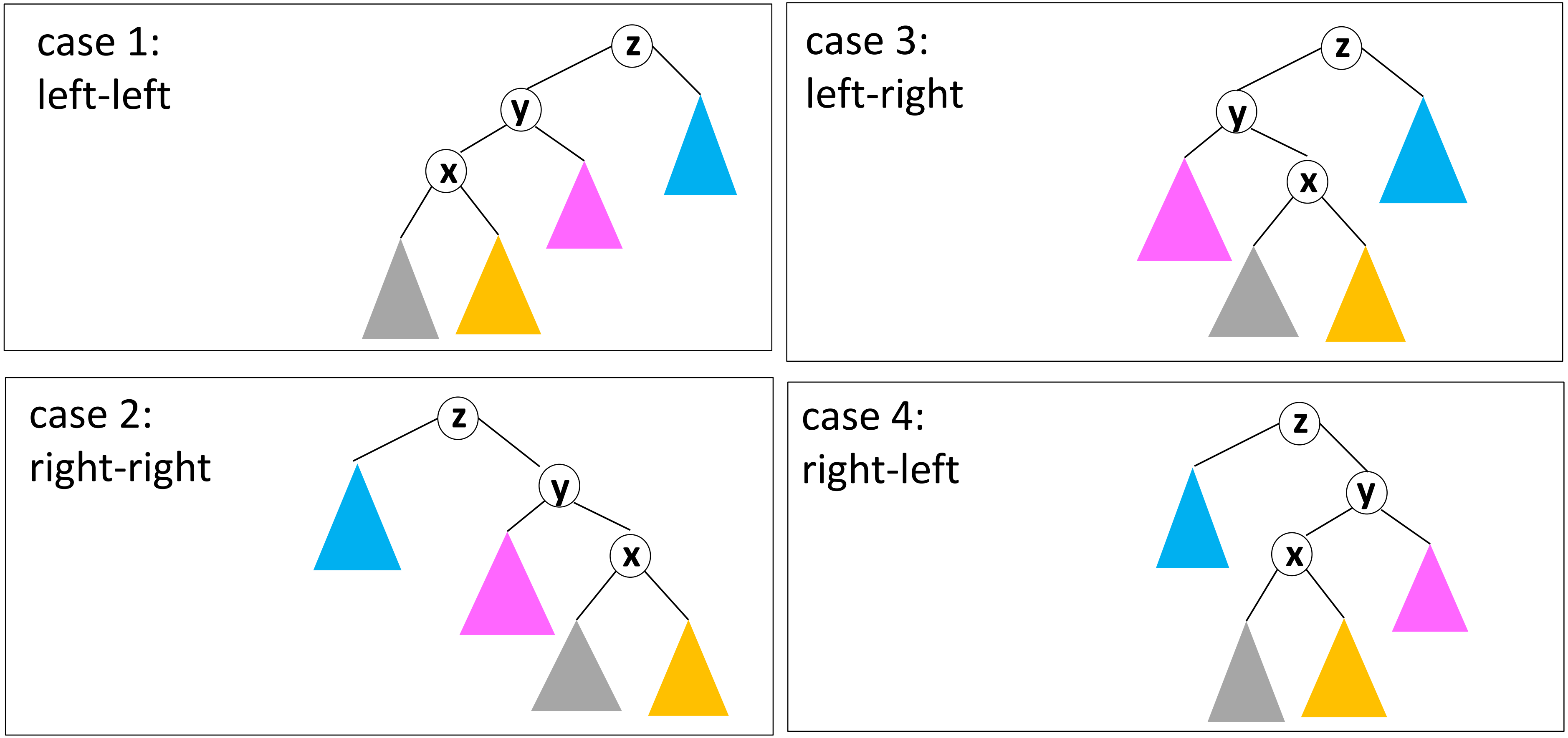
Case 4: Right-Left Imbalance

- Symmetrically, there is a *double left rotation* on node z



- First, a right rotation at y, second, a left rotation at z
- BST order is preserved
- Used for right-left imbalance
  - can argue BST ordering is preserved, as before
  - can argue height balance property restored, as before

Unbalanced Node z: all 4 cases



- z is the first unbalanced node on the path from inserted node to the root
- y is the taller child of z
  - z is guaranteed to have one child taller than the other
- x is the taller child of y
  - y is guaranteed to have one child taller than the other

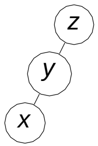
Fixing Unbalanced AVL tree

restructure(x, y, z)

x : node of BST that has an unbalanced grandparent,

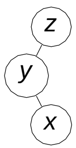
y and z: the parent and grandparent of x

case

case 1


: // Right rotation

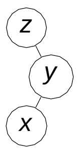
return *rotate-right*(z)

case 3


: // Double-right rotation

z.left ← *rotate-left*(y)

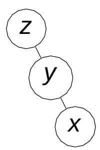
return *rotate-right*(z)

case 4


: // Double-left rotation

z.right ← *rotate-right*(y)

return *rotate-left*(z)

case 2


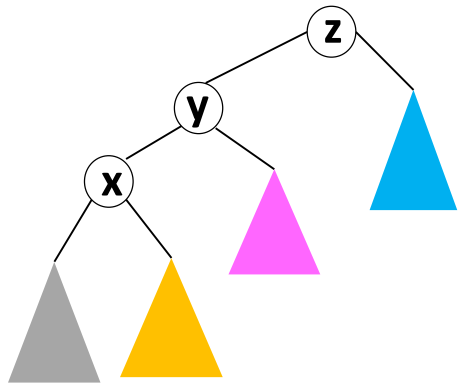
: // Left rotation

return *rotate-left*(z)

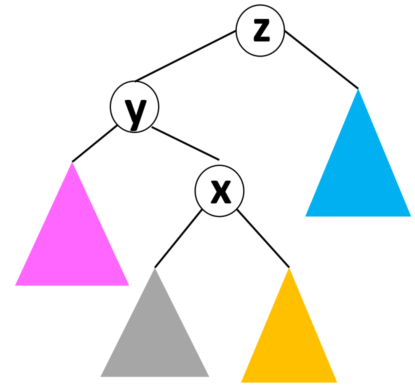
- In each case, the middle key of x, y, z becomes the new root of the subtree
- Running time is $\Theta(1)$

Tri-Node Restructuring

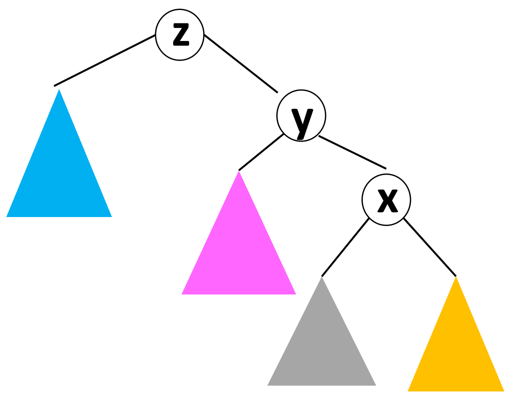
case 1:
left-left



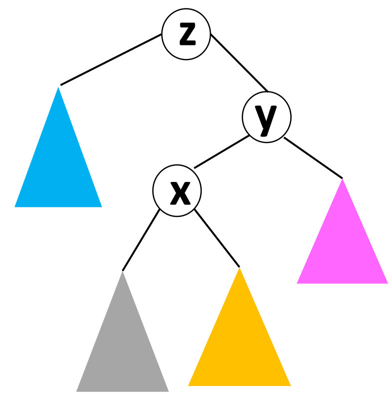
case 3:
left-right



case 2:
right-right

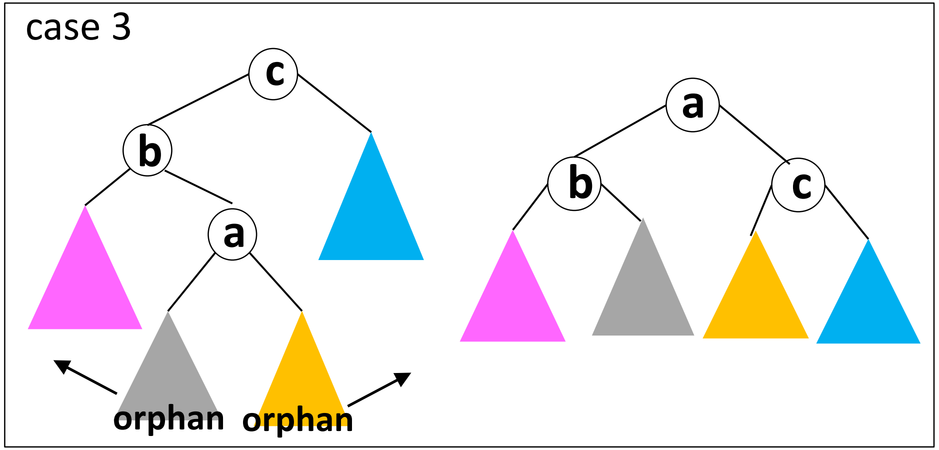


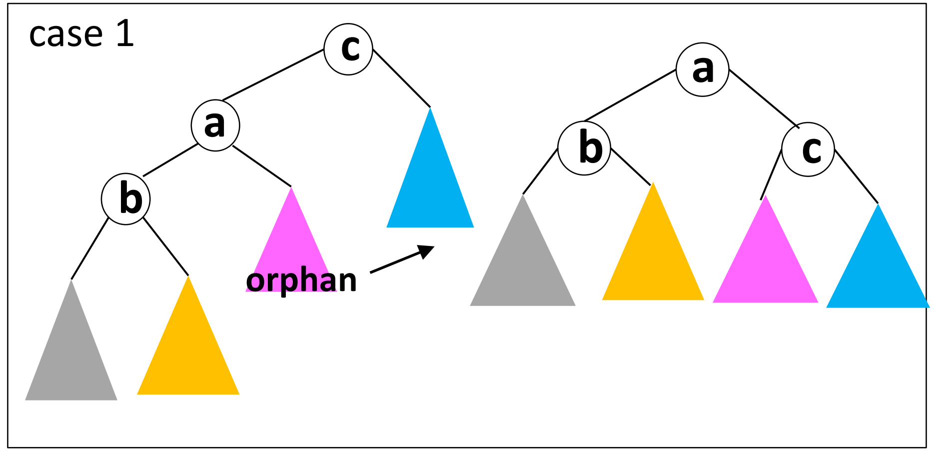
case 4:
right-left



- All four cases can be handled with one method, *Tri-Node restructuring*

Tri-Node Restructuring for Case 1 and Case 3





- Rename
  - ***a*** = node with middle key
  - ***b*** = node with smallest key
  - ***c*** = node with largest key
- Restructure
  - **a** becomes new subtree parent
  - **b** becomes left child of **a**
  - **c** becomes right child of **a**
  - subtrees of **b**, **c** with root not equal to **a** stay attached to where they were
  - one or two subtrees of **a** get “orphaned”
    - left subtree, if orphan, becomes right child of **b**
    - right subtree, if orphan, becomes left child of **c**

Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - full code for insertion
    - deletion

AVL insertion

```
AVL::insert(k, v)
    z ← BST::insert(k, v)
    while (z is not NIL)
        if (|z.left.height − z.right.height| > 1) then
            let y be tallest child of z
            let x be tallest child of y
            z ← restructure(x, y, z)
            break                    // done after one restructure
        setHeightFromSubtrees(z)
        z ← parent of z
```

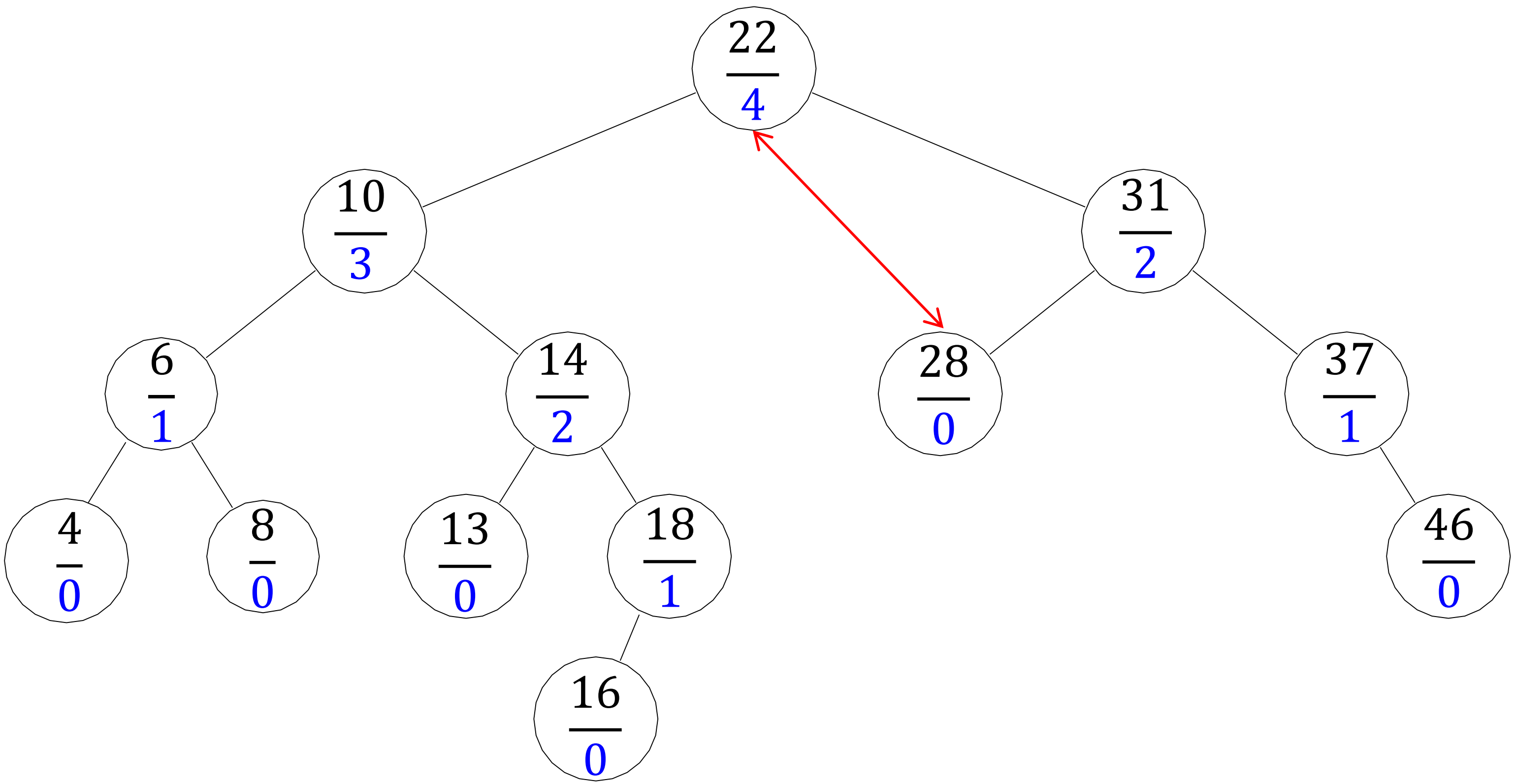
```
setHeightFromSubtrees(u)
  if u is not an empty subtree
     u.height ← 1 + max{u.left.height, u.right.height}
```

Outline

- Dictionaries and Balanced Search Trees
  - Dictionary ADT
  - Review: Binary Search Trees
  - AVL Trees
    - insertion
    - restoring the AVL Property: Rotations
    - full code for insertion
    - deletion

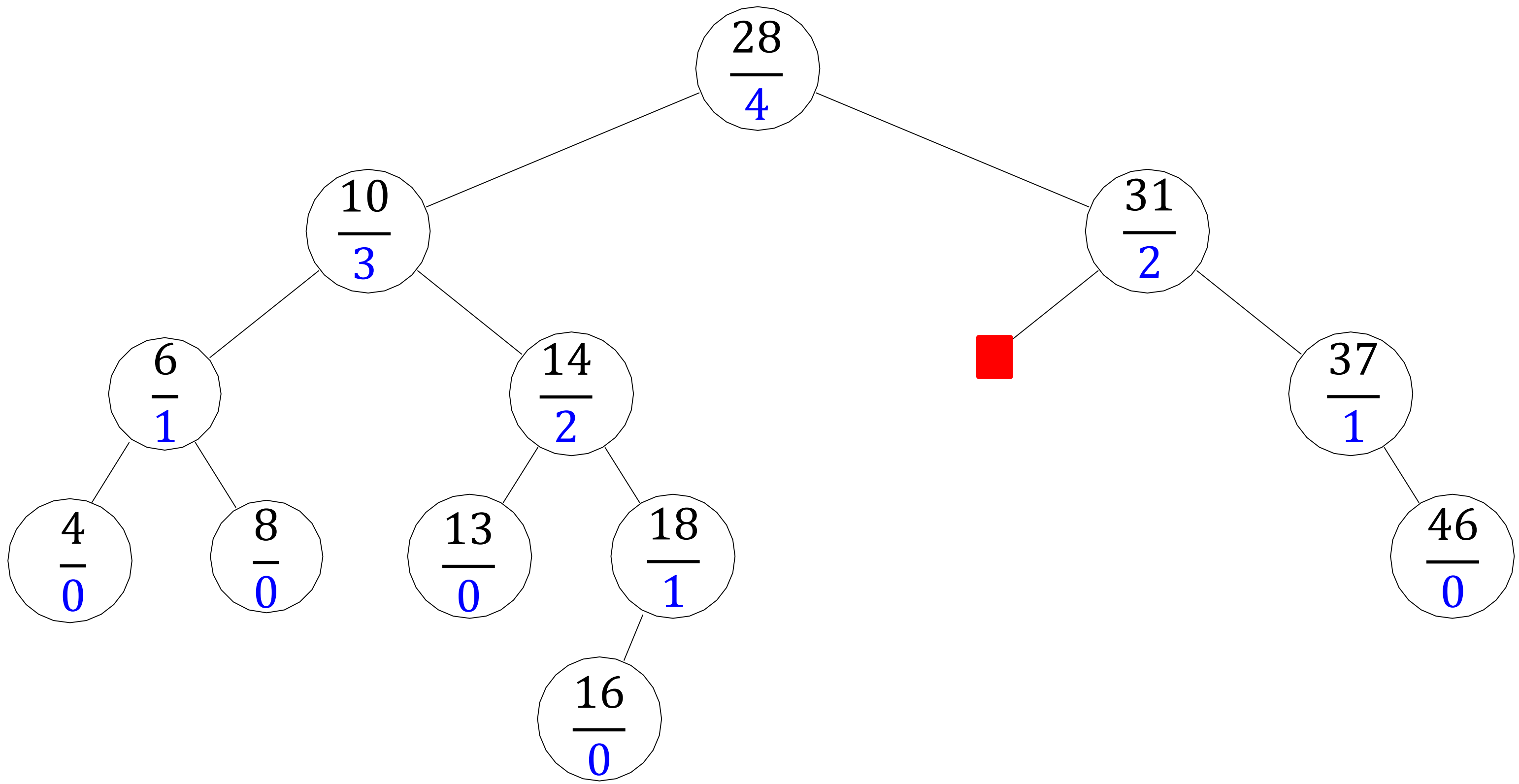
AVL Deletion Example

Example: *AVL::delete*(22)



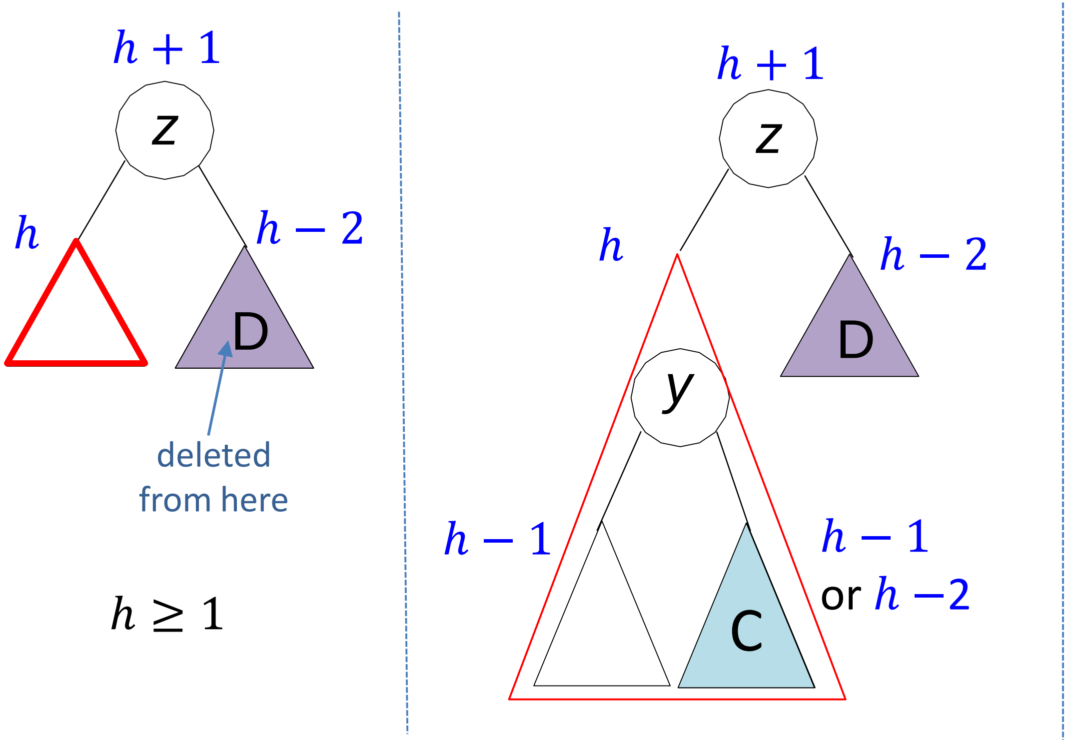
AVL Deletion Example

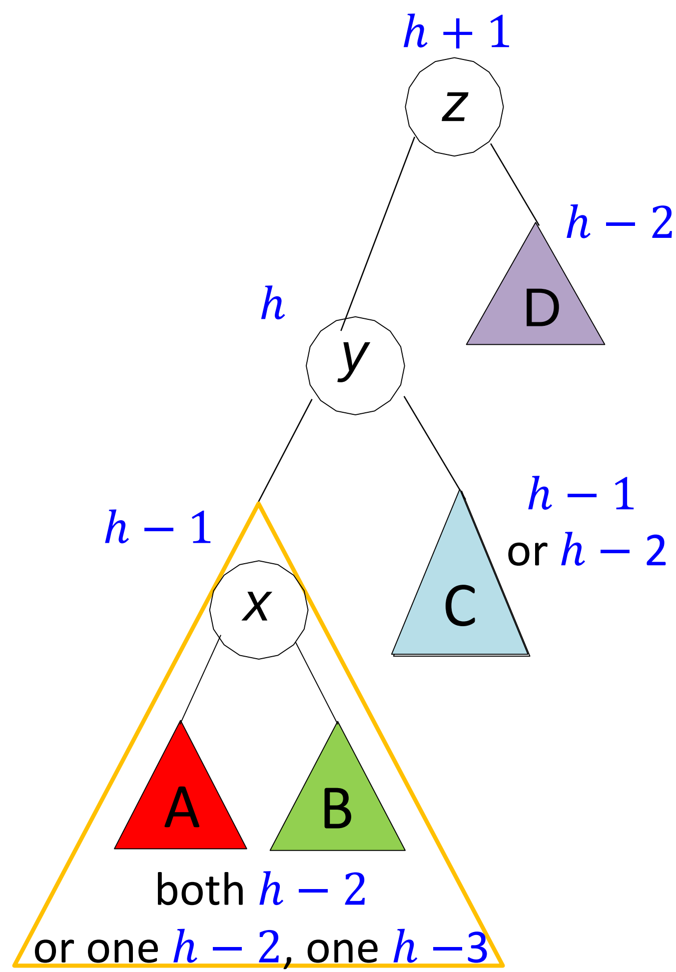
Example: *AVL::delete*(22)



Restoring Height After Deletion: Case 1

- Let z be the *first* unbalanced node on path from deleted node to the root

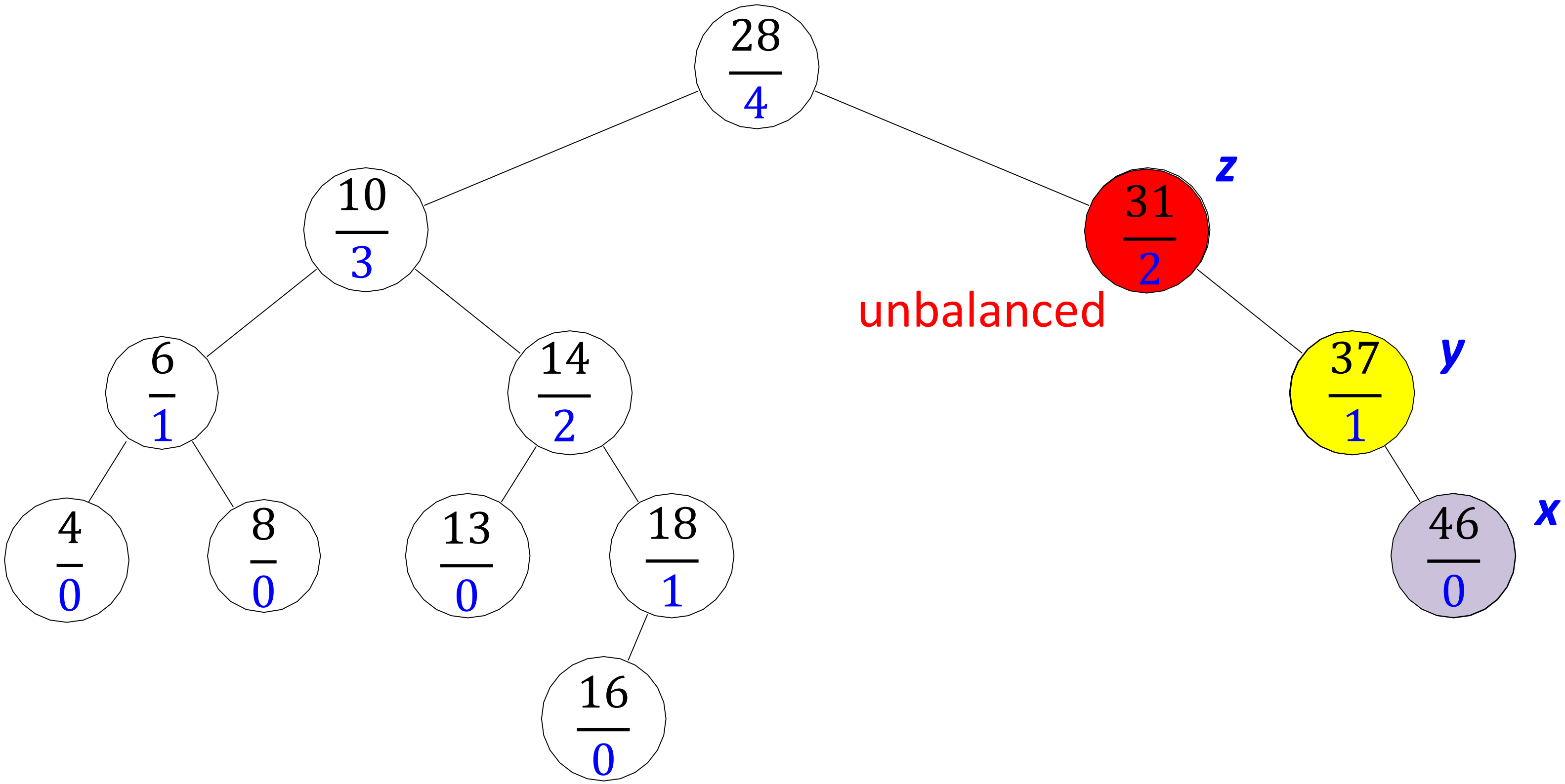




- Rebalancing is similar to that after insertion, **but**
  - while z is guaranteed to have one taller child
  - y may have both children of the same height
    - which child to take as x?

AVL Deletion Example

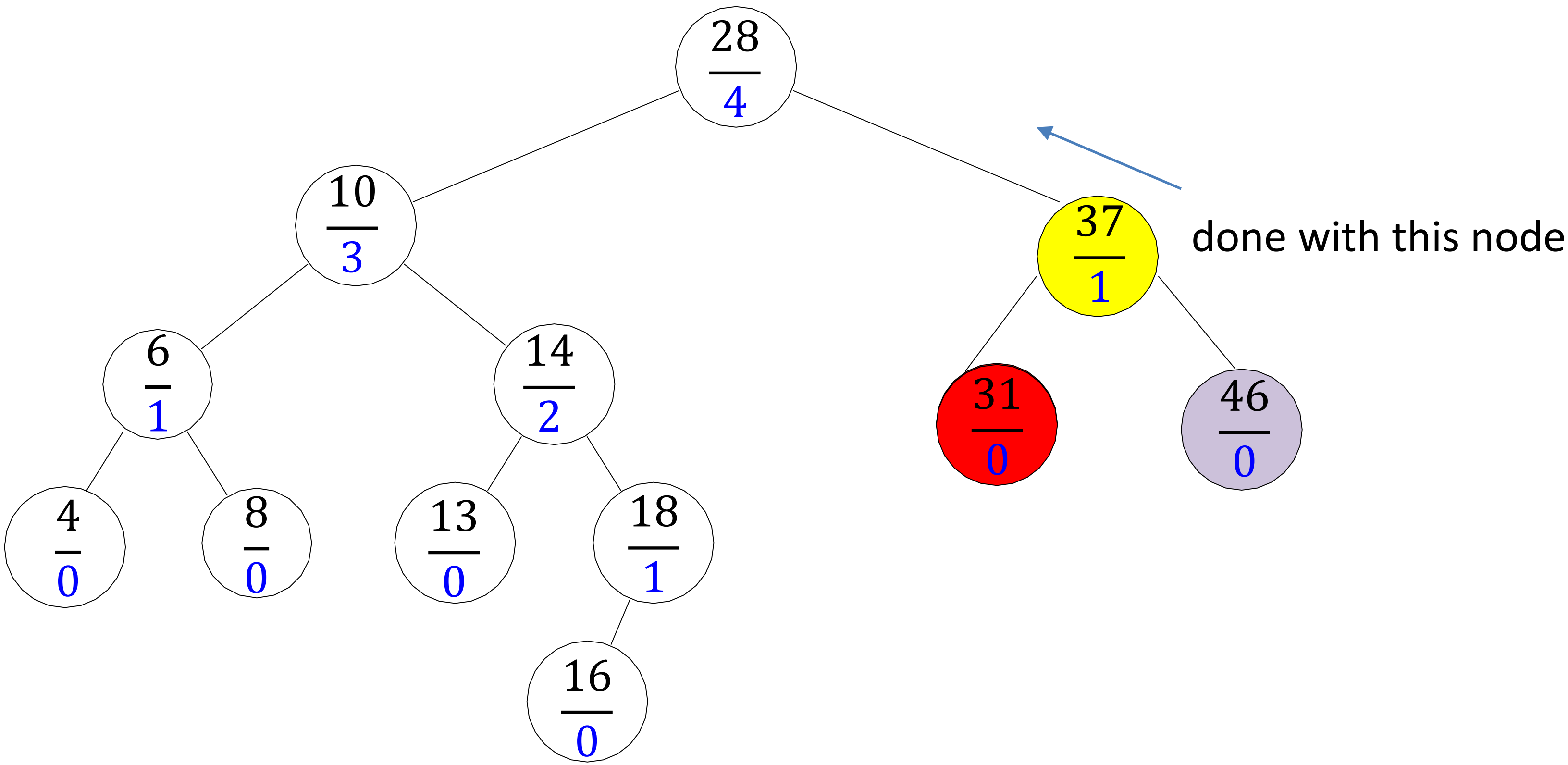
Example: *AVL::delete*(22)



- Fix with left rotation on node z
- Or trinode restructuring on node z

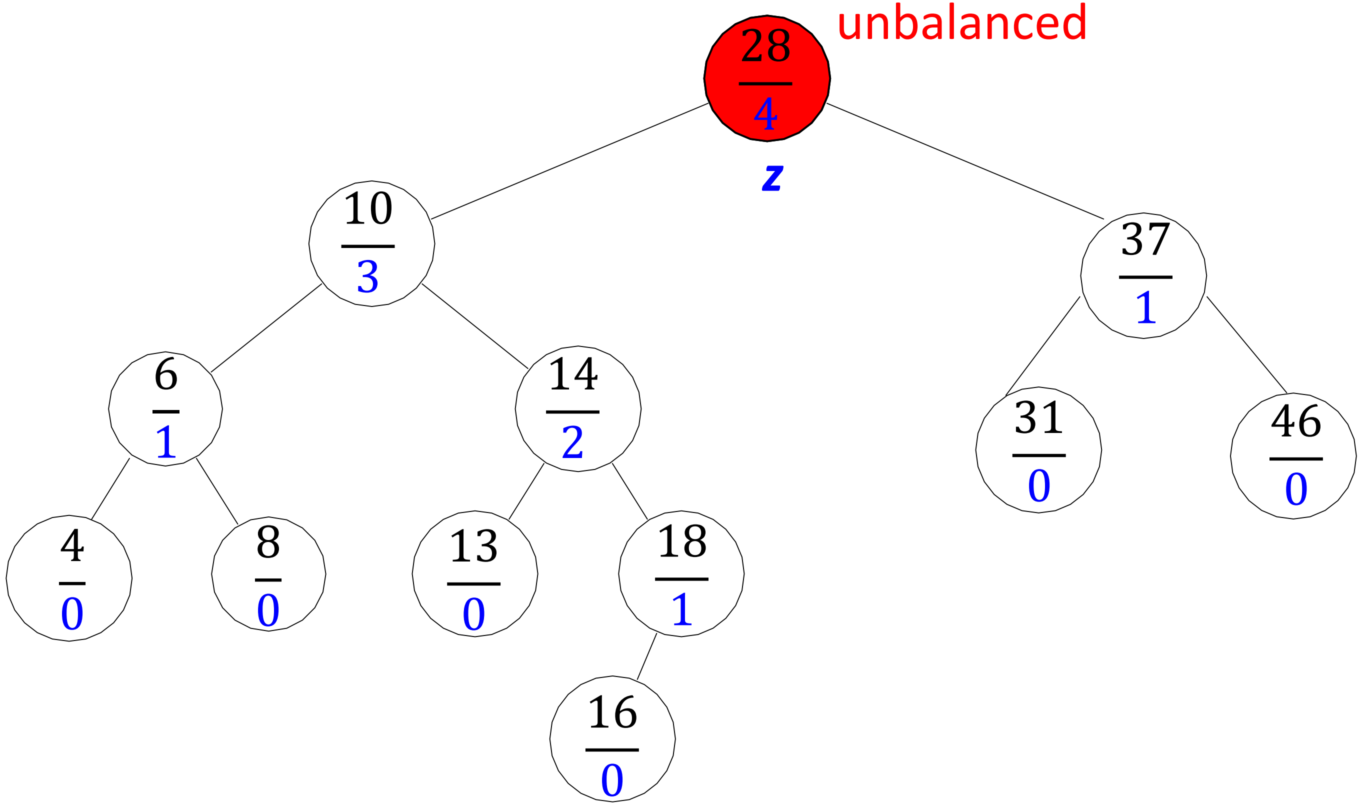
AVL Deletion Example

Example: *AVL::delete*(22)



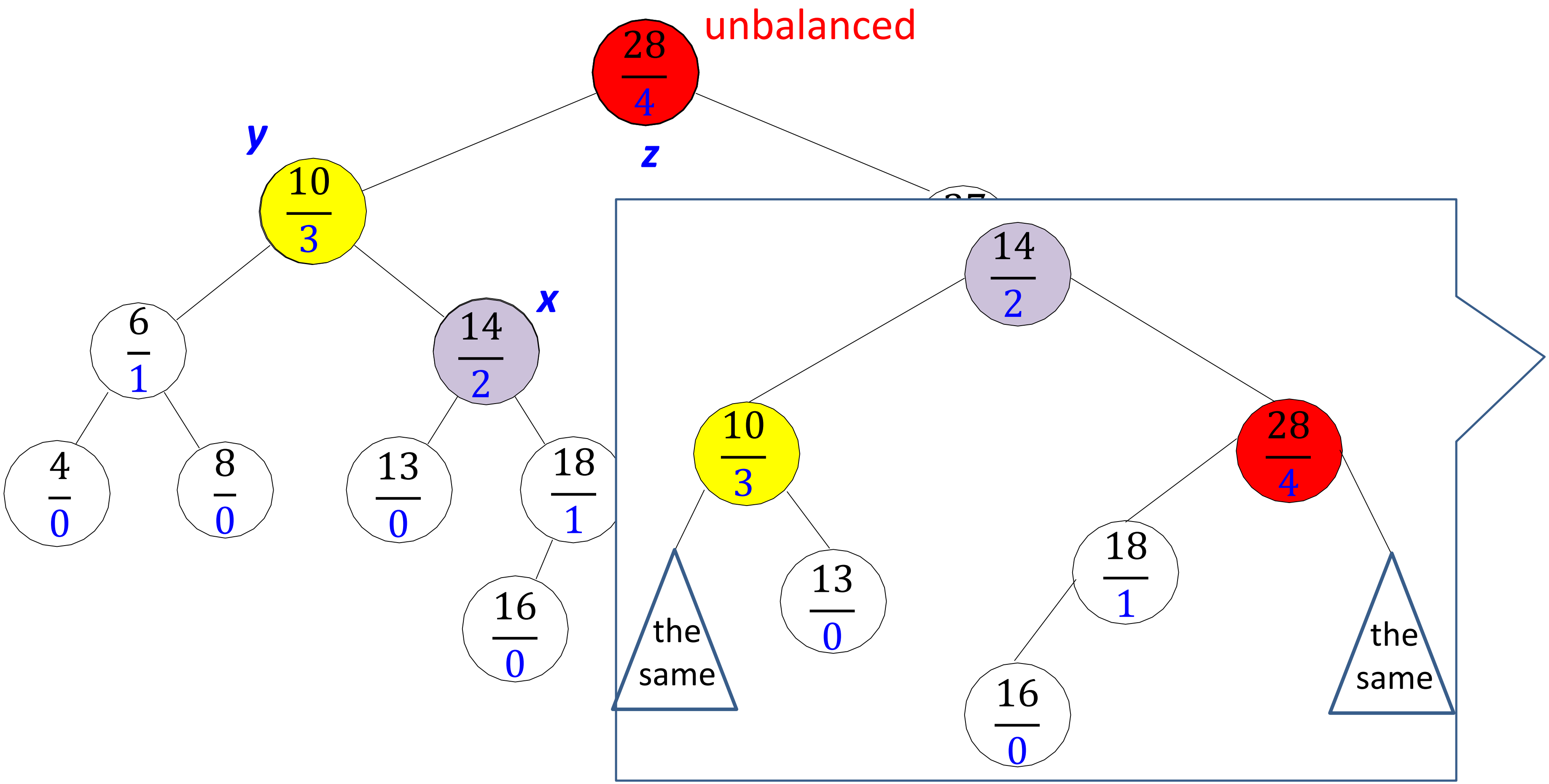
AVL Deletion Example

Example: *AVL::delete*(22)



AVL Deletion Example

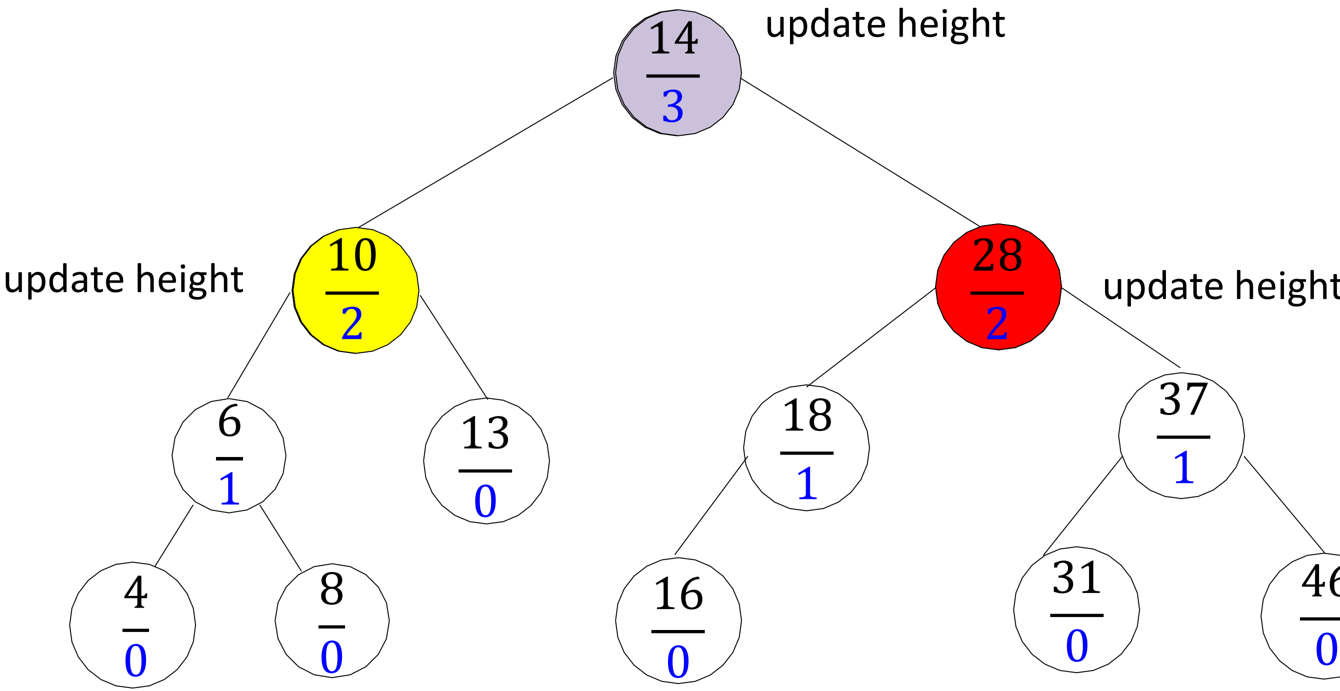
Example: *AVL::delete*(22)



- Fix with double right rotation (left rotate y, then rotate right z)
- Or trinode restructuring on node z

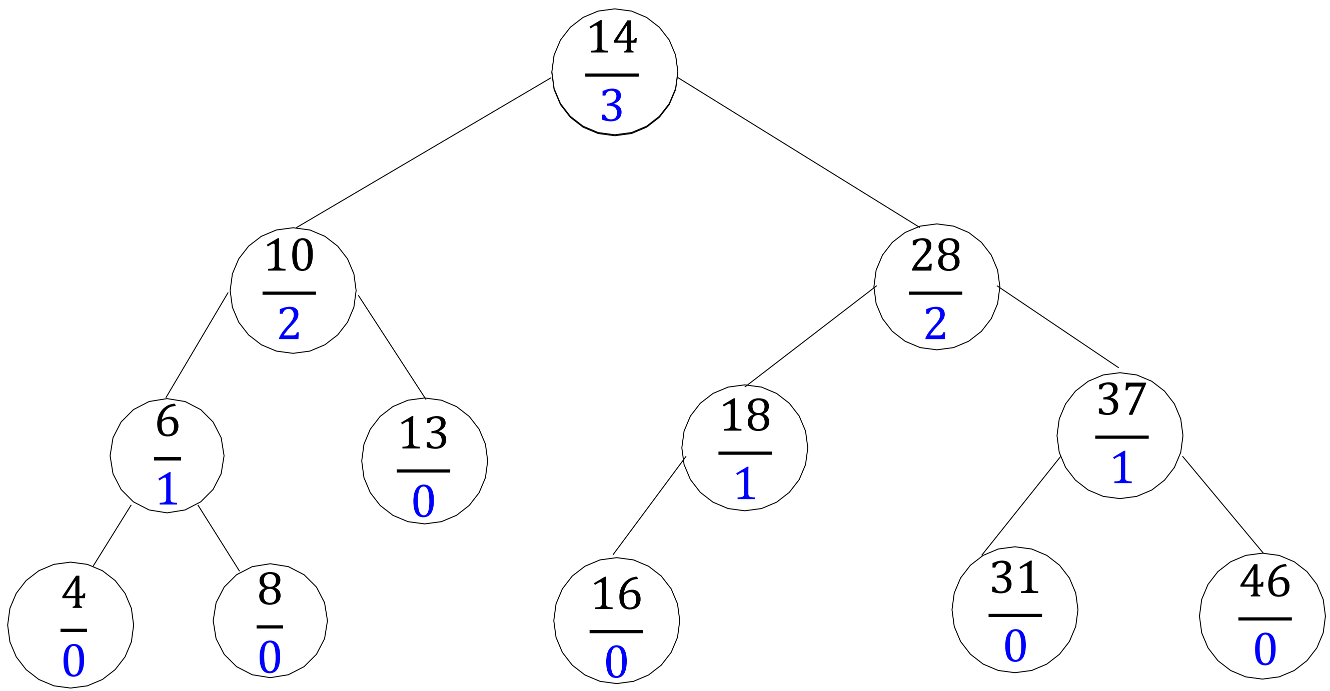
AVL Deletion Example

Example: *AVL::delete*(22)



AVL Deletion Example

Example: *AVL::delete*(22)

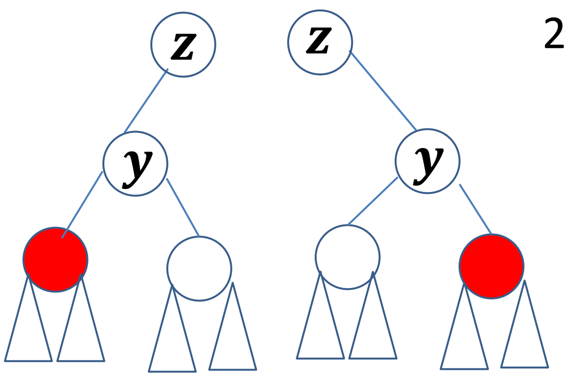


- Rebalanced

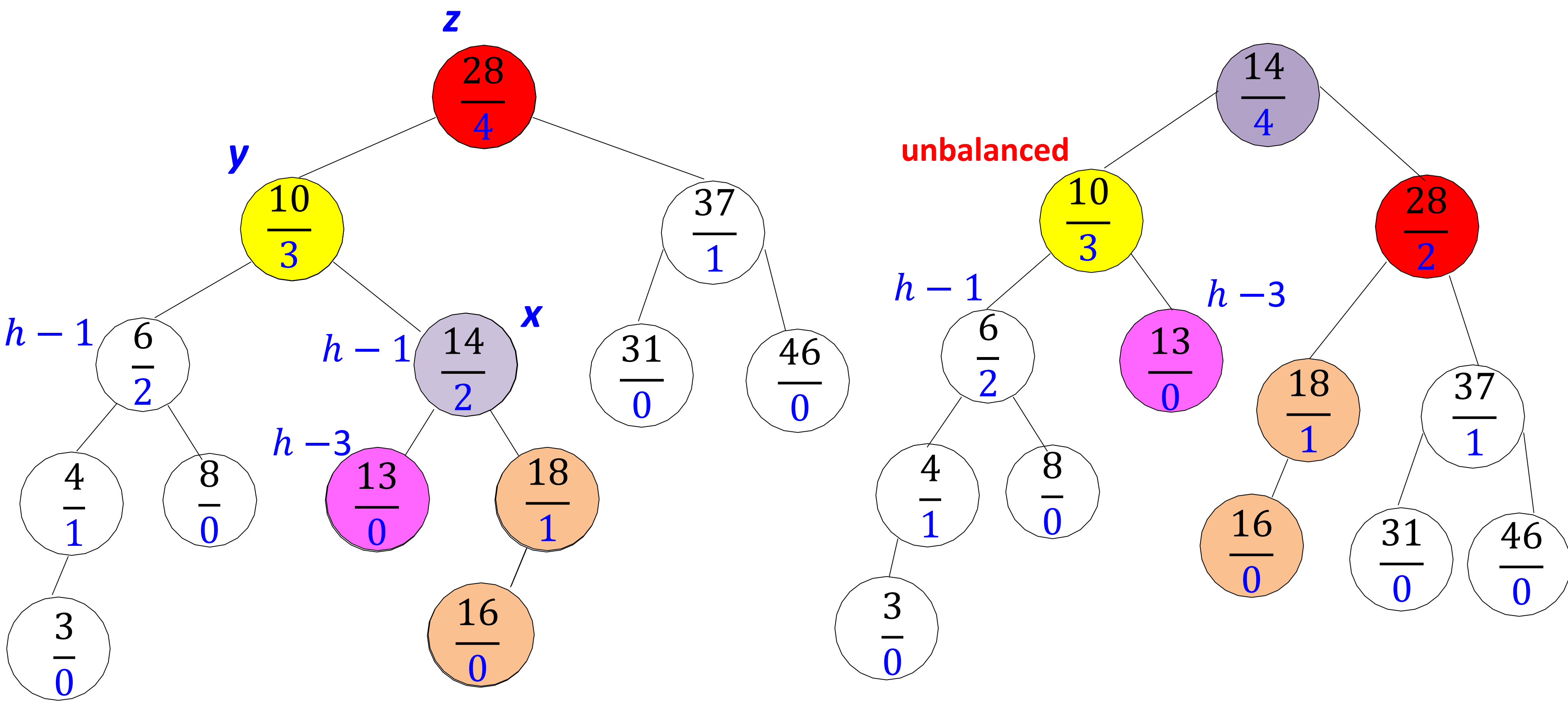
AVL Deletion

- *AVL::delete*(T, k)
  - first, delete k from T with BST deletion
    - delete returns parent z of the deleted node
    - heights of nodes on path from z to root may have decreased
  - next, move up the tree from z, updating heights
    - if height difference is ± 2 at node z , then z is *unbalanced*
      - re-structure tree to restore height-balance property
      - like rebalancing for insertion, with two differences
        1. restructuring after deletion does not guarantee to restore tree height to what it was before deletion
           - must continue path up the tree, fixing any imbalances
        2. tallerChild(y)
           - *if* left and right children of y have the same height **must** apply *same side rule*:
             - return left child of y if y is itself the left child
             - return right child of y if y is itself the right child

tallerChild

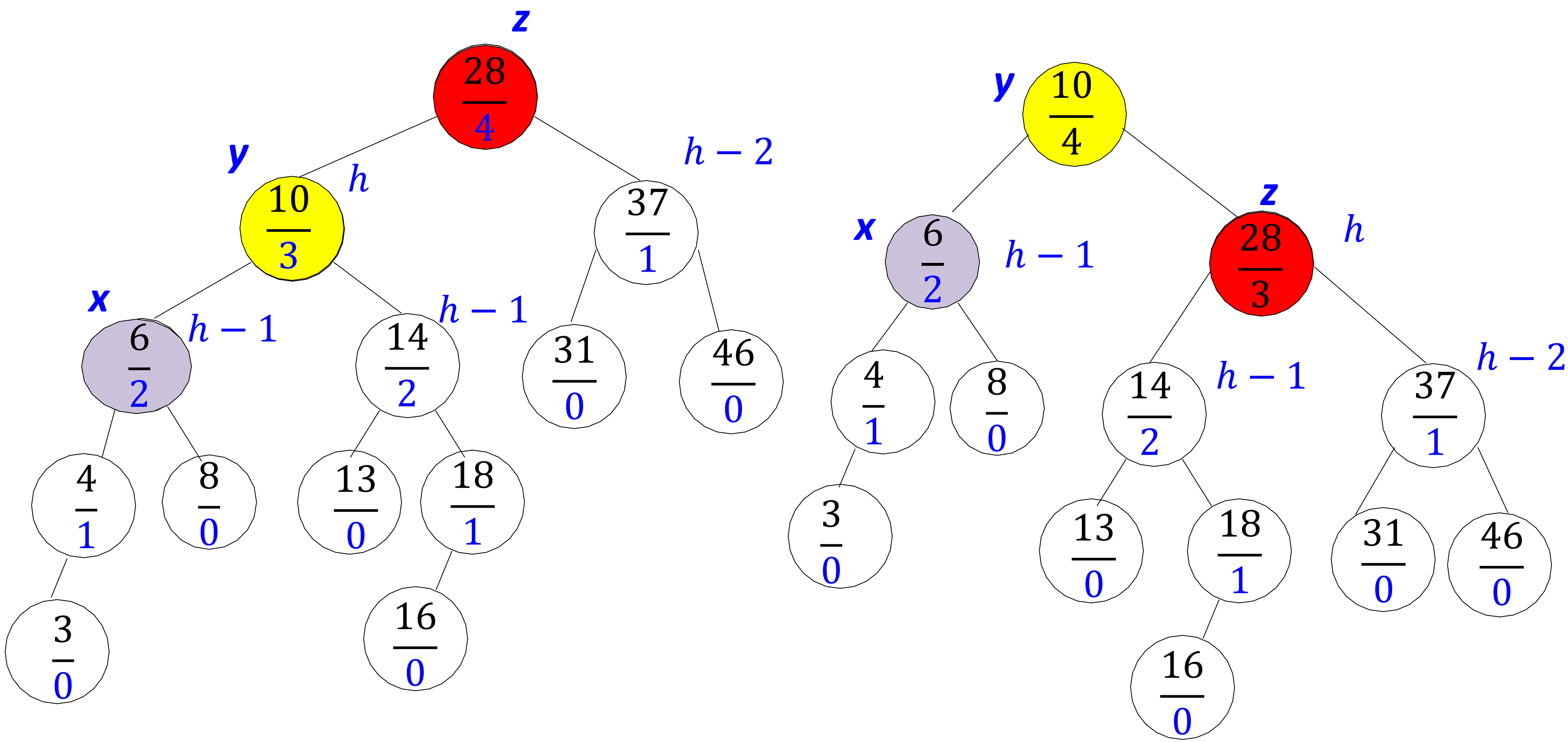


Incorrect Deletion Example **not** Following *Same Side* Rule



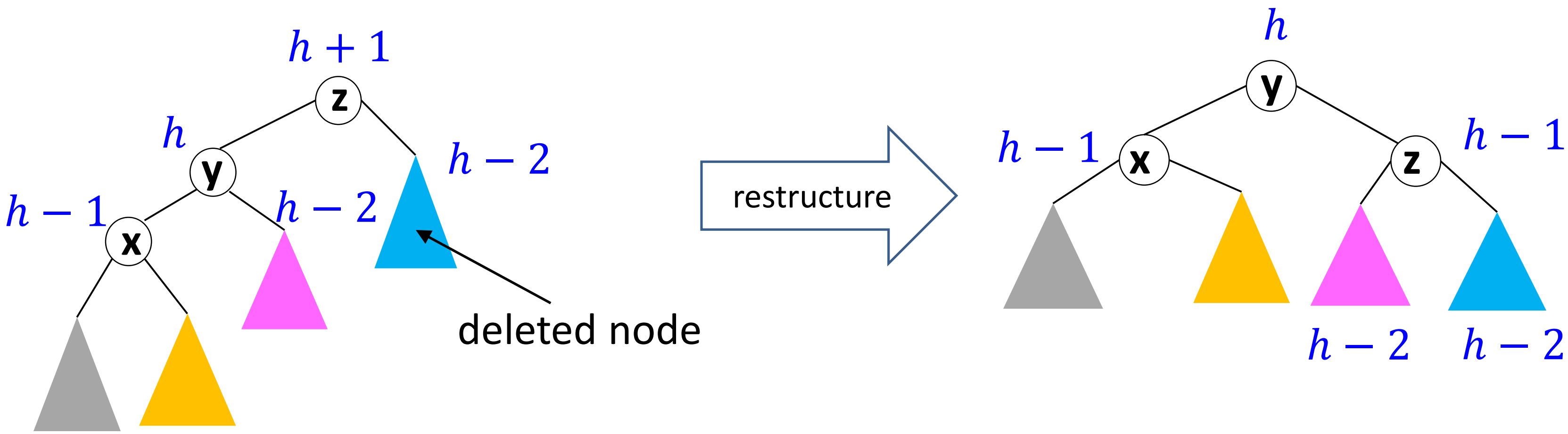
- The "other" child of y has height $h - 1$
  - children of x get separated
  - one of them has height $h - 3$ and becomes a sibling of the "other" child of y which has height $h - 1$

AVL Deletion Example Following *Same Side* Rule



- Rotate or trinode restructuring
- Rebalanced!
  - children of x do not separate

Reduced Height after Deletion



- If 'not the tallest' child of $\boldsymbol{y}$ has height $h-2$, height decreases after rebalancing
  - might cause imbalance higher up the tree

AVL Delete Pseudocode

```
AVL::delete(k)
    z ← BST::delete(k)
    // Assume z is the parent of the BST node that was removed
    while (z is not NIL)
        if (|z.left.height − z.right.height| > 1) then
            let y be tallest child of z
            let x be tallest child of y
            // break ties to prefer 'the same side'
            z ← restructure(x, y, z)
        setHeightFromSubtrees(z)
        // must continue checking the path upwards
        z ← parent of z
```

AVL Tree Operations Runtime

- **AVL::search**
  - implemented just like in BSTs, runtime is $\Theta(height)$
- **AVL::insert**
  - *BST::insert*
  - then check and update along path to new leaf
    - *restructure* restores the height of the tree to what it was
    - so *restructure* will be called *at most once*
  - total cost $\Theta(height)$
- **AVL::delete**
  - *BST::delete*, then check and update along path to deleted node
    - *restructure* may be called $\Theta(height)$ times
  - total cost $\Theta(height)$
- Total cost for all operations is $\Theta(height) = \Theta(\log n)$
  - but in practice, the constant is quite large
- There are other realizations of ADT dictionary that are better in practice