CS 240 – Data Structures and Data Management

Module 4: Dictionaries

Armin Jamshidpey

Based on lecture notes by many previous cs240 instructors

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

Winter 2023

version 2024–05–08 08:29

Outline

4 Dictionaries and Balanced Search Trees

- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations

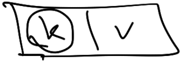
key | value

Outline

Dictionary ADT

Dictionary: An ADT consisting of a collection of items, each of which contains

- a *key*
- some *data* (the “value”)



and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

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.*

Examples: symbol table, license plate database

Elementary Implementations

Common 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

Unordered array or linked list

search $\Theta(n)$

insert $\Theta(1)$ (except array occasionally needs to resize)

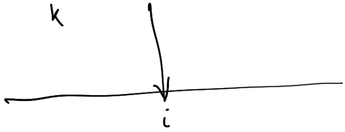
delete $\Theta(n)$ (need to search)

Ordered array

search $\Theta(\log n)$ (via binary search)

insert $\Theta(n)$

delete $\Theta(n)$



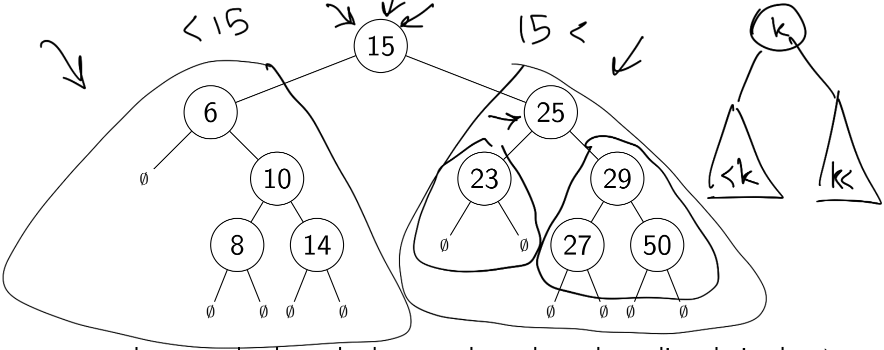
Outline

4 Dictionaries and Balanced Search Trees
- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations

Binary Search Trees (review)

Structure Binary tree: all nodes have two (possibly empty) subtrees
Every node stores a KVP
Empty subtrees usually not shown

Ordering Every key k in $T.left$ is less than the root key.
Every key k in $T.right$ is greater than the root key.

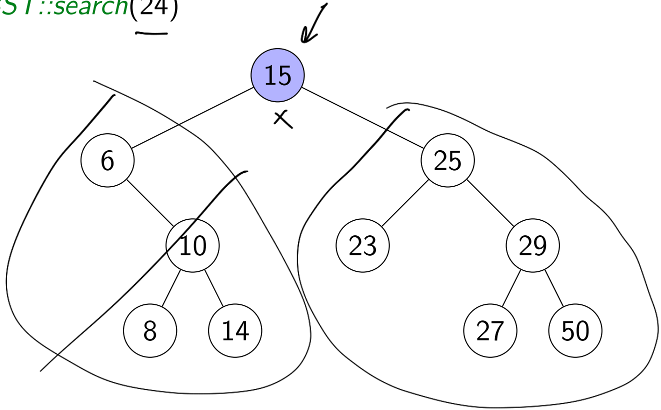


(In our examples we only show the keys, and we show them directly in the node. A more accurate picture would be ◯ ⋯⋯→ key = 15, <other info>)

BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key. Stop if found or subtree is empty, else recurse at subtree.

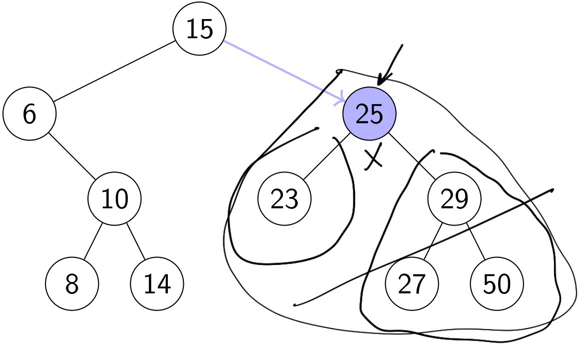
Example: *BST::search*(24)



BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

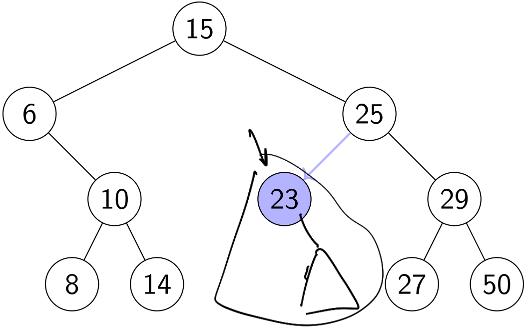
Example: *BST::search*(24)



BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

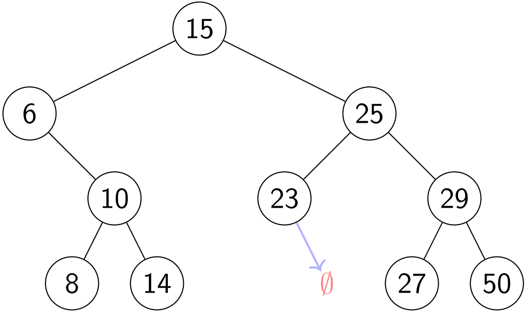
Example: *BST::search*(24)



BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

Example: *BST::search*(24)

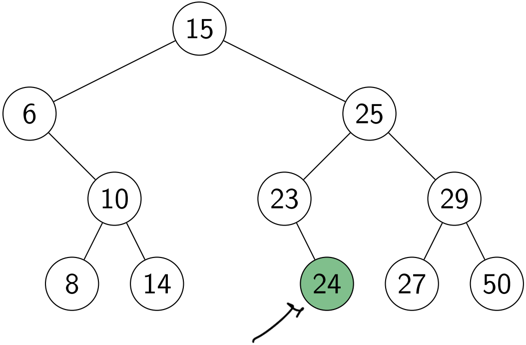


BST as realization of ADT Dictionary

BST::search(k) Start at root, compare k to current node's key.
Stop if found or subtree is empty, else recurse at subtree.

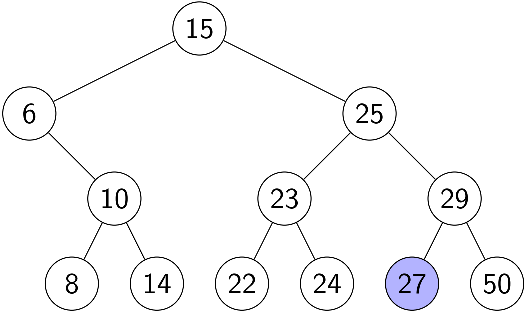
BST::insert(k, v) Search for k, then insert (k, v) as new node

Example: *BST::insert*(24, v)



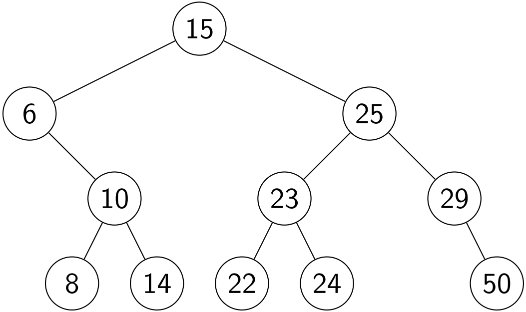
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.



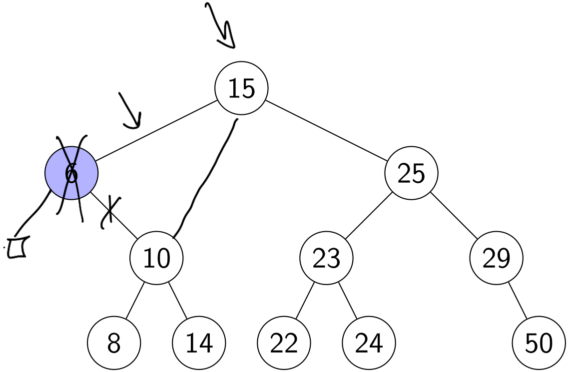
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.



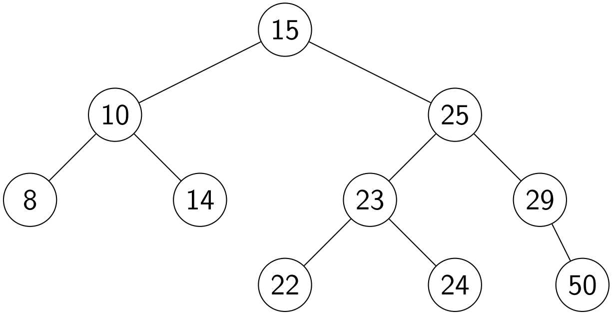
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up



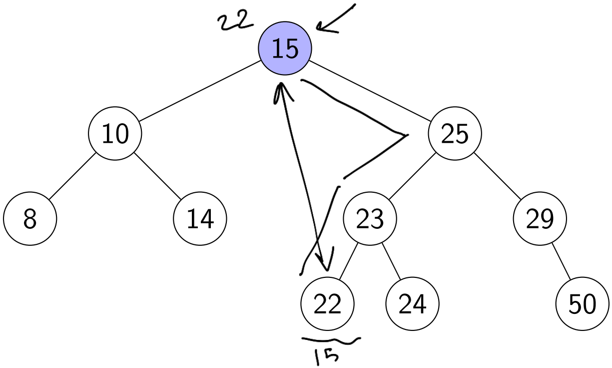
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up



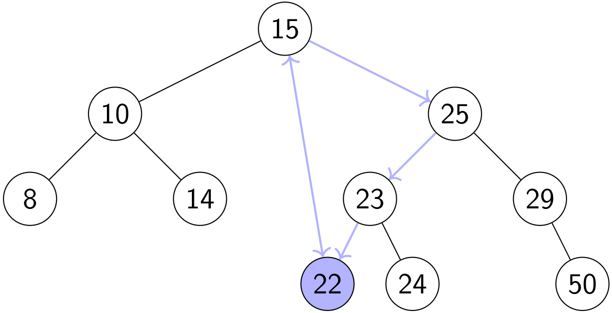
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** or **predecessor** node and then delete that node



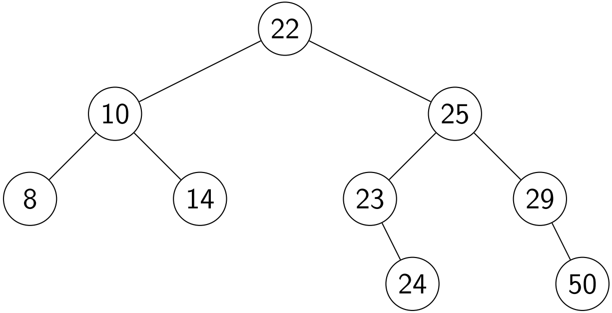
Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** or **predecessor** node and then delete that node



Deletion in a BST

- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up
- Else, swap key at x with key at **successor** or **predecessor** node and then delete that node

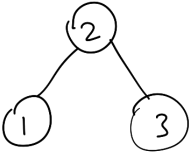


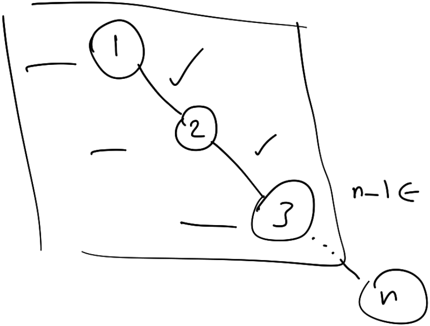
Height of a BST

BST::search, *BST::insert*, *BST::delete* all have cost $\Theta(h)$, where
h = height of the tree = max. path length from root to leaf

If n items are inserted one-at-a-time, how big is h?

- Worst-case:





$n-1 \in \Theta(n)$

Height of a BST

BST::search, *BST::insert*, *BST::delete* all have cost $\Theta(h)$, where
$h =$ height of the tree $=$ max. path length from root to leaf

If n items are inserted one-at-a-time, how big is h?

- Worst-case: $n - 1 = \Theta(n)$
- Best-case:

Height of a BST

BST::search, *BST::insert*, *BST::delete* all have cost $\Theta(h)$, where
$h =$ height of the tree $=$ max. path length from root to leaf

If n items are inserted one-at-a-time, how big is h?

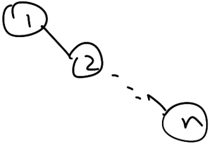
- Worst-case: $n - 1 = \Theta(n)$
- Best-case: $\Theta(\log n)$.
  Any binary tree with n nodes has height $\geq \boxed{\log(n+1) - 1}$
- Average-case:

Height of a BST

BST::search, *BST::insert*, *BST::delete* all have cost $\Theta(h)$, where
h = height of the tree = max. path length from root to leaf

If n items are inserted one-at-a-time, how big is h?

- Worst-case: $\underline{n - 1 = \Theta(n)}$
- Best-case: $\Theta(\log n)$.
  Any binary tree with n nodes has height $\geq \log(n+1) - 1$
- Average-case: Can show $\underline{\Theta(\log n)}$



Outline

4 Dictionaries and Balanced Search Trees
- ADT Dictionary
- Review: Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restoring the AVL Property: Rotations

AVL Trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an **AVL Tree** is a BST with an additional **height-balance property** at every node:

The heights of the left and right subtree differ by at most 1.

(The height of an empty tree is defined to be -1.)

Rephrase: If node v has left subtree L and right subtree R, then

$$\textbf{balance}(v) := \mathit{height}(R) - \mathit{height}(L) \text{ must be in } \{-1, 0, 1\}$$

$\mathit{balance}(v) = -1$ means v is *left-heavy*

$\mathit{balance}(v) = +1$ means v is *right-heavy*

AVL Trees

Introduced by Adel'son-Vel'skiĭ and Landis in 1962, an **AVL Tree** is a BST with an additional **height-balance property** at every node:

The heights of the left and right subtree differ by at most 1.

(The height of an empty tree is defined to be -1.)

Rephrase: If node v has left subtree L and right subtree R, then

$$\textbf{balance}(v) := height(R) - height(L) \text{ must be in } \{-1, 0, 1\}$$

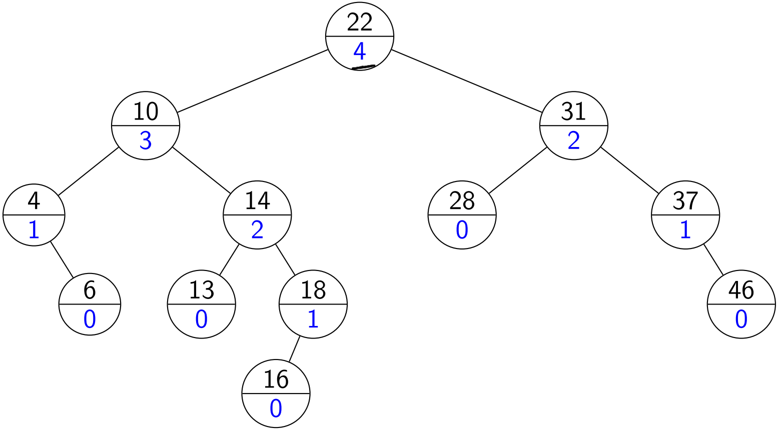
$balance(v) = -1$ means v is *left-heavy*

$balance(v) = +1$ means v is *right-heavy*

- Need to store at each node v the height of the subtree rooted at it
- Can show: It suffices to store $balance(v)$ instead
  - uses fewer bits, but code gets more complicated

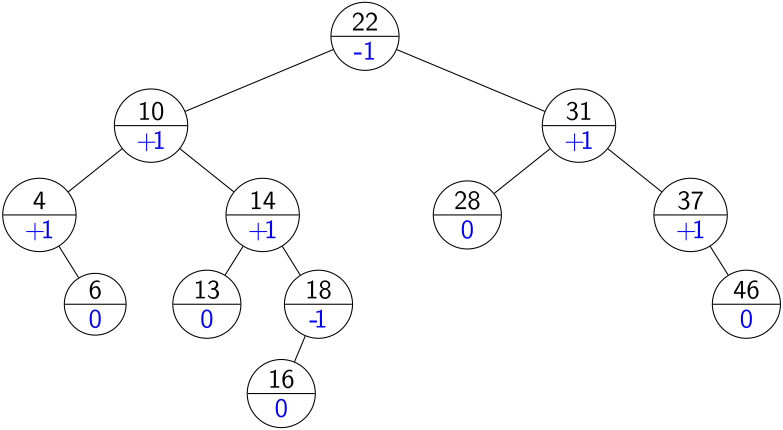
AVL tree example

(The lower numbers indicate the height of the subtree.)



AVL tree example

Alternative: store balance (instead of height) at each node.



Height of an AVL tree

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *insert*, *delete* all cost $\Theta(\log n)$ in the *worst case!*

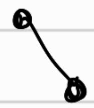
Proof:

- Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.
- What is a recurrence relation for $N(h)$?
- What does this recurrence relation resolve to?

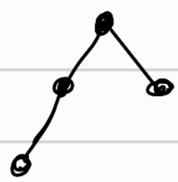
h	-1	0	1	2	3	4	5	
$N(h)$	0	1	2	4	7	$\boxed{12}$	20	$\dots$
$N(h) \geq$		2^0	2^1	2^2	$\cancel{2^3}$	2^4		
$N(h) \geq$		2^0-1	2^1-1	2^2-1	2^3-1	$\cancel{2^4-1}$		
$\boxed{N(h) \geq \sqrt{2}^h - 1}$		$\sqrt{2}^0-1$	$\sqrt{2}^1-1$	$\sqrt{2}^2-1$	$\dots$	$\dots$	$\dots$	$\dots$

$N(-1) = 0$ $\qquad N(0) = 1$

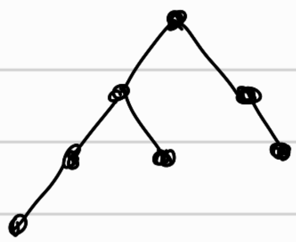
$N(1) = 2$

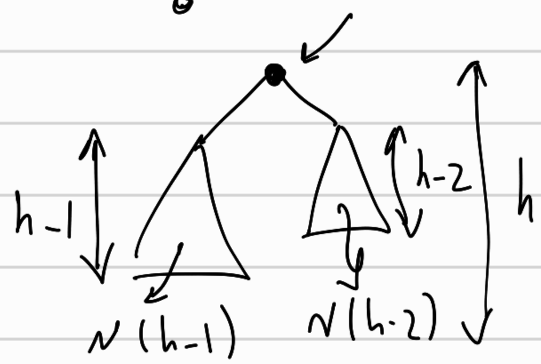


$N(2) = 4$



$N(3) = 7$





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

Claim: $N(h) \geq \sqrt{2}^h - 1$, $\quad h \geq -1$

$h = -1$, $\quad 0 = N(-1) \geq \sqrt{2}^{-1} - 1$ ✓

$h = 0$, $\quad 1 = N(0) \geq \sqrt{2}^{1} - 1$ ✓

Assume the statement is true for $m < h$, $h \geq 1$

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

$$\geq 1 + 2\left(\sqrt{2}^{(h-2)} - 1\right)$$

$$= 1 + \sqrt{2}^h - 2 = \sqrt{2}^h - 1 \quad ✓$$

Let n be the number of nodes in an AVL tree of height h.

$$n \geq N(h) \geq \sqrt{2}^{h} - 1 \Rightarrow \underline{\sqrt{2}^{h} \leq n - 1}$$

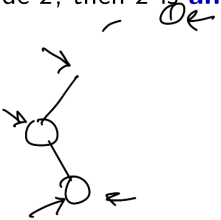
$$\Rightarrow \underline{\underline{h \in O(\log n)}}$$

Outline

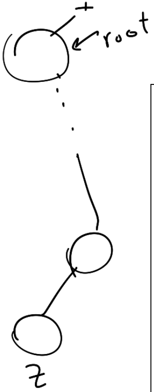
AVL insertion

To perform *AVL::insert*(k, v):

- First, insert (k, v) with the usual BST insertion.
- We assume that this returns the new leaf z where the key was stored.
- Then, move up the tree from z, updating heights.
  - We assume for this that we have parent-links. This can be avoided if *BST::Insert* returns the full path to z.
- If the height difference becomes ± 2 at node z, then z is **unbalanced**. Must re-structure the tree to rebalance.

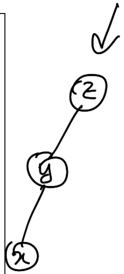


AVL insertion



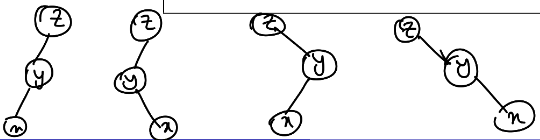
AVL::insert(k, v)

```
z ← BST::insert(k, v)    // leaf where k is now stored
while (z is not NIL)
    if (|z.left.height − z.right.height| > 1) then
        Let y be taller child of z
        Let x be taller child of y
        z ← restructure(x, y, z)  // see later
        break                     // can argue that we are done
    setHeightFromSubtrees(z)
    z ← z.parent
```



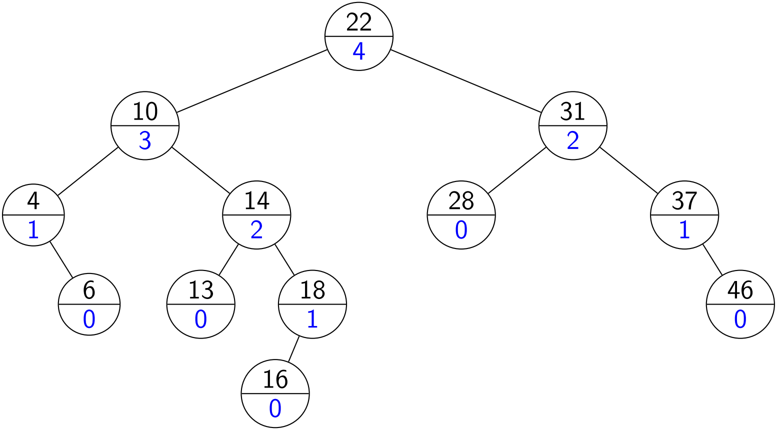
setHeightFromSubtrees(u)

```
u.height ← 1 + max{u.left.height, u.right.height}
```



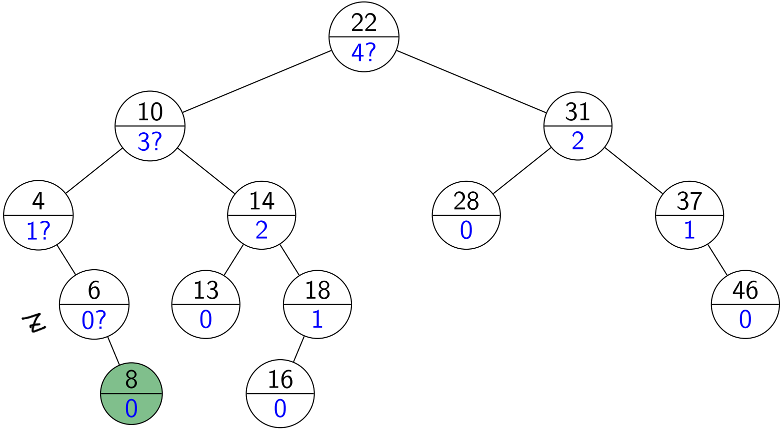
AVL Insertion Example

Example: *AVL::insert*(8)



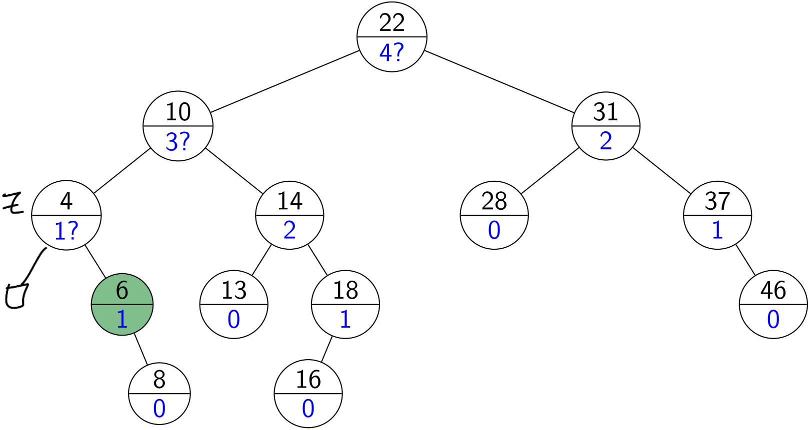
AVL Insertion Example

Example: *AVL::insert*(8)



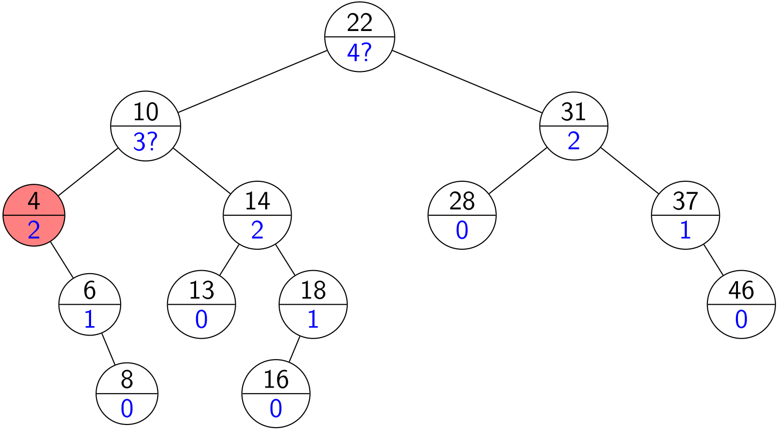
AVL Insertion Example

Example: *AVL::insert*(8)



AVL Insertion Example

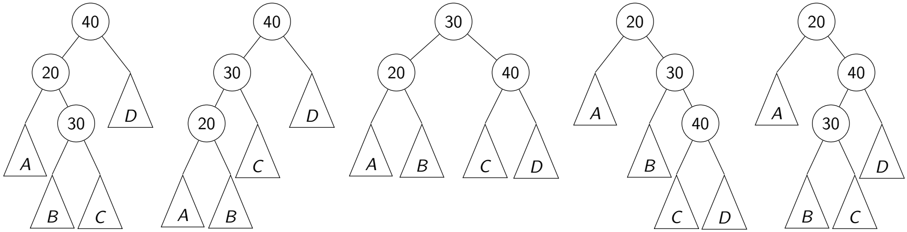
Example: *AVL::insert*(8)



Outline

How to “fix” an unbalanced AVL tree

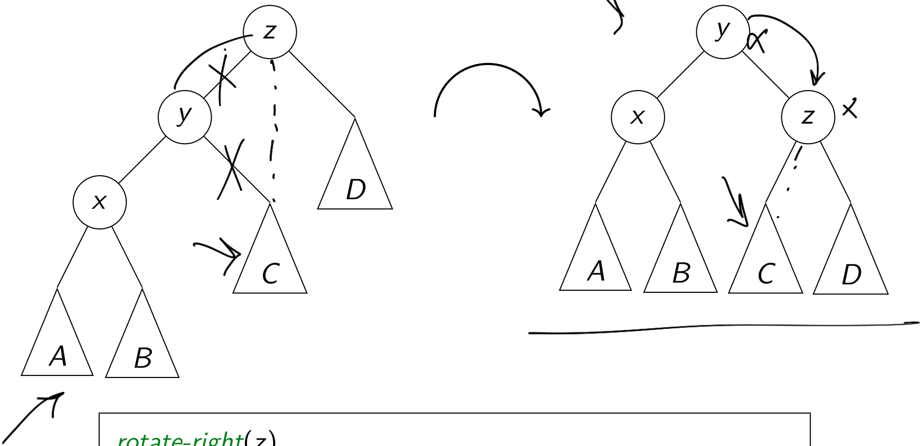
Note: there are many different BSTs with the same keys.



Goal: change the *structure* among three nodes without changing the *order* and such that the subtree becomes balanced.

Right Rotation

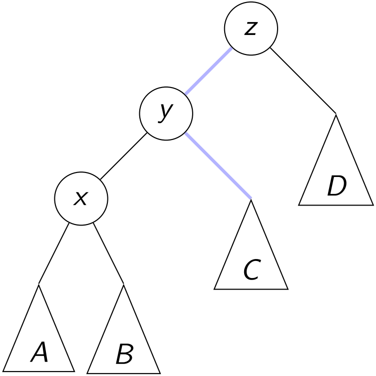
This is a **right rotation** on node z:



rotate-right(z)

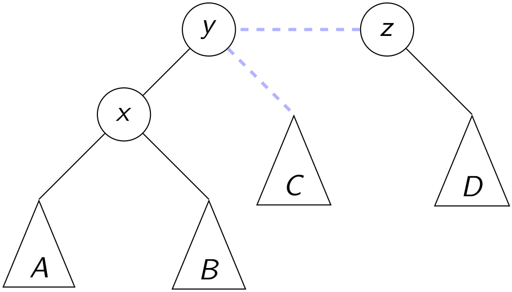
1. $y \leftarrow z.left$, $z.left \leftarrow y.right$, $y.right \leftarrow z$
2. *setHeightFromSubtrees*(z), *setHeightFromSubtrees*(y)
3. **return** y // returns new root of subtree

Why do we call this a rotation?

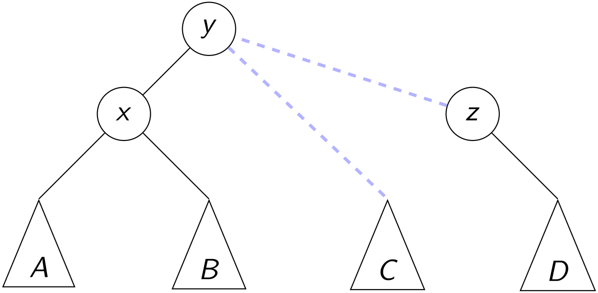


$x < \boxed{y} < z$

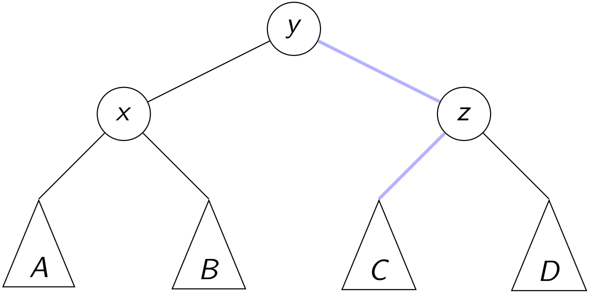
Why do we call this a rotation?



Why do we call this a rotation?

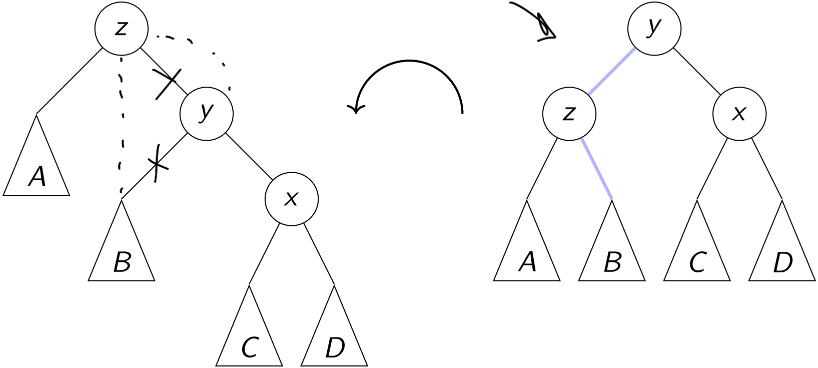


Why do we call this a rotation?



Left Rotation

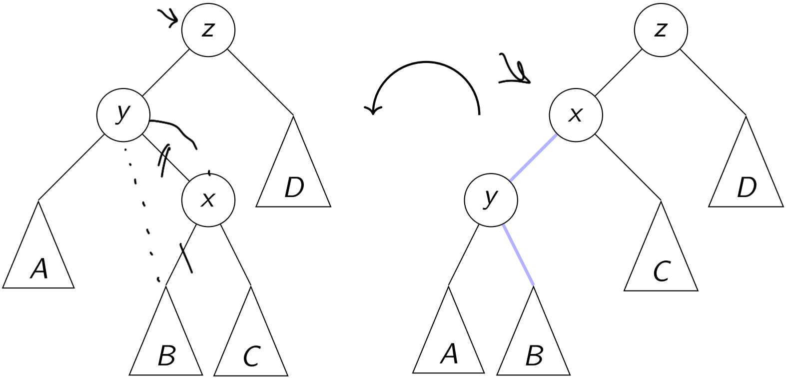
Symmetrically, this is a **left rotation** on node z:



Again, only two links need to be changed and two heights updated.
Useful to fix right-right imbalance.

Double Right Rotation

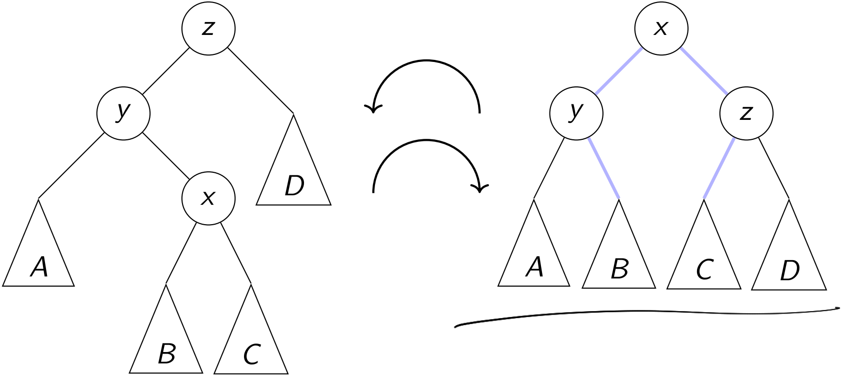
This is a **double right rotation** on node z:



First, a left rotation at y.

Double Right Rotation

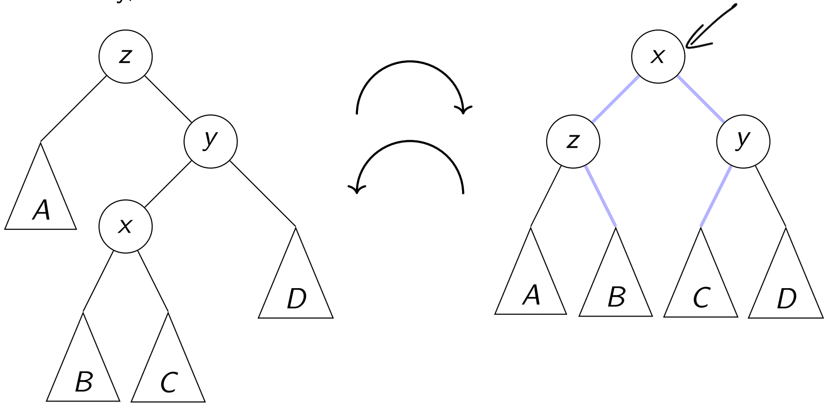
This is a **double right rotation** on node z:



First, a left rotation at y.
Second, a right rotation at z.

Double Left Rotation

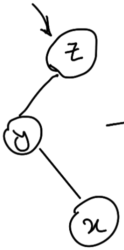
Symmetrically, there is a **double left rotation** on node z:



First, a right rotation at y.
Second, a left rotation at z.

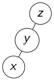
$z < \boxed{x} < y$

Fixing a slightly-unbalanced AVL tree

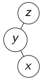


restructure(x, y, z)
node x has parent y and grandparent z

1. **case**

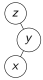


: // Right rotation
return *rotate-right*(z)

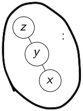


: // Double-right rotation
z.left ← *rotate-left*(y)
return *rotate-right*(z)

$y < x < z$



: // Double-left rotation
z.right ← *rotate-right*(y)
return *rotate-left*(z)

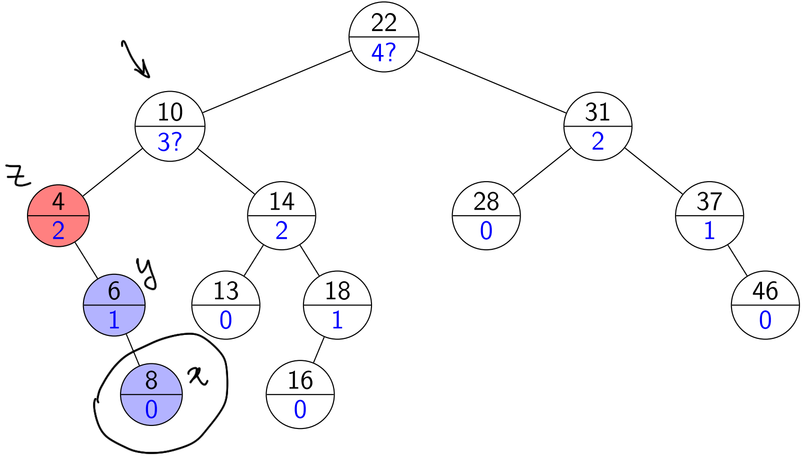


: // Left rotation
return *rotate-left*(z)

Rule: The middle key of x, y, z becomes the new root.

AVL Insertion Example revisited

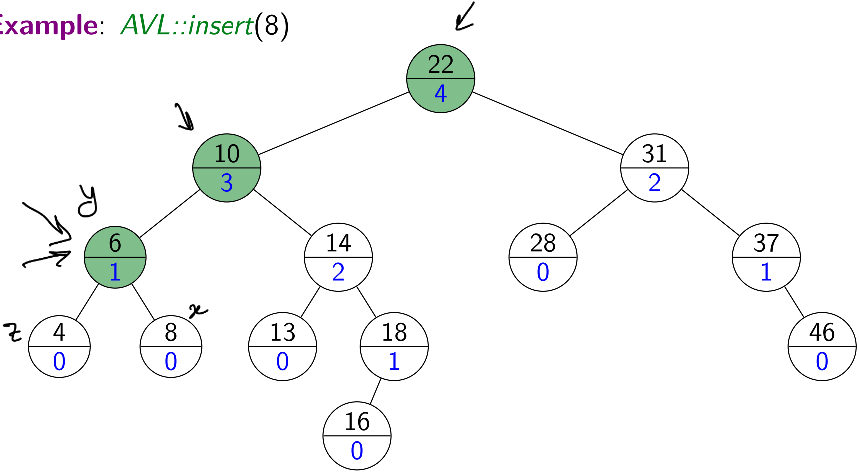
Example: *AVL::insert*(8)



$z < \underline{y} < x$

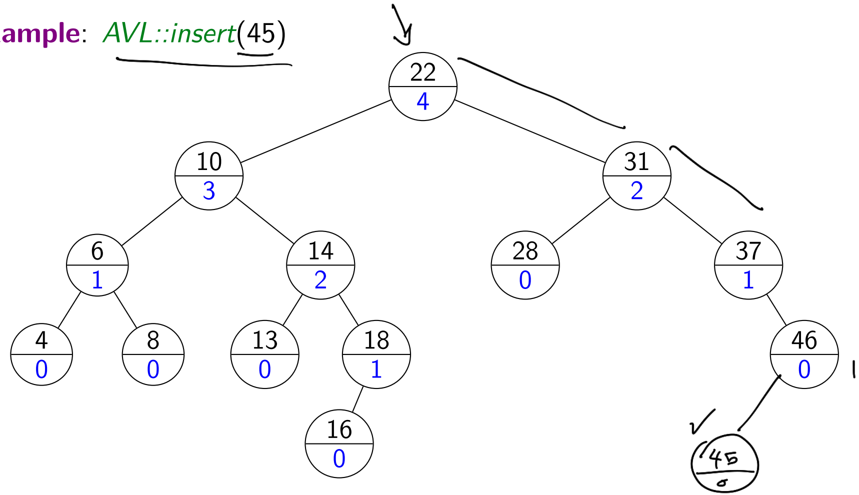
AVL Insertion Example revisited

Example: *AVL::insert*(8)



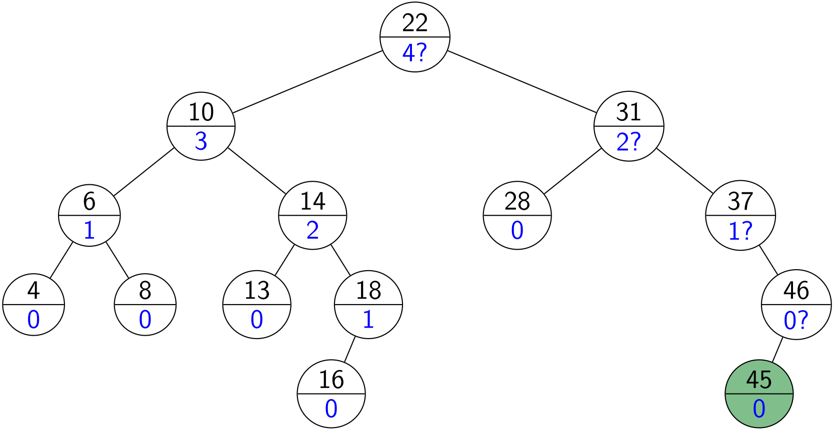
AVL Insertion: Second example

Example: *AVL::insert*(45)



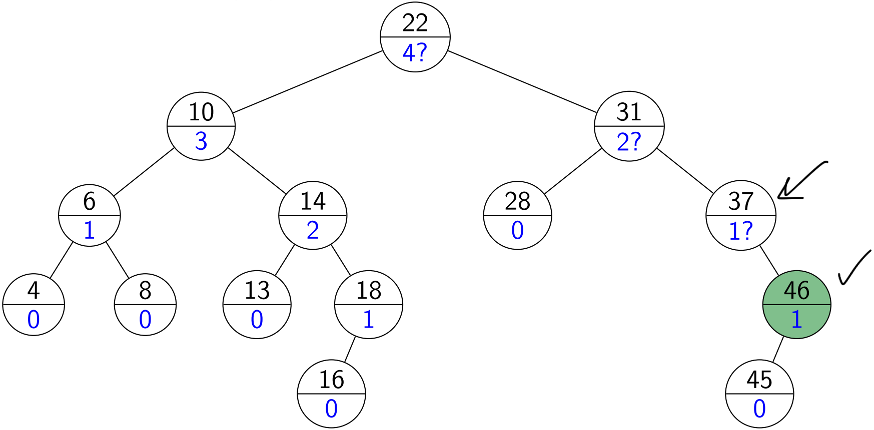
AVL Insertion: Second example

Example: *AVL::insert*(45)



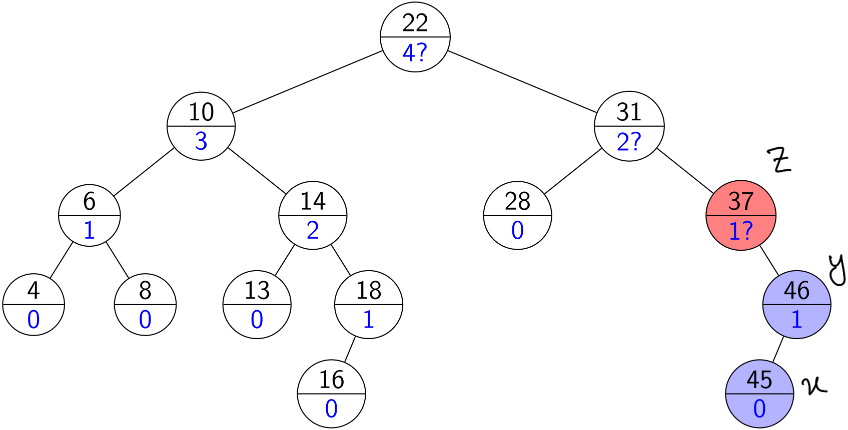
AVL Insertion: Second example

Example: *AVL::insert*(45)



AVL Insertion: Second example

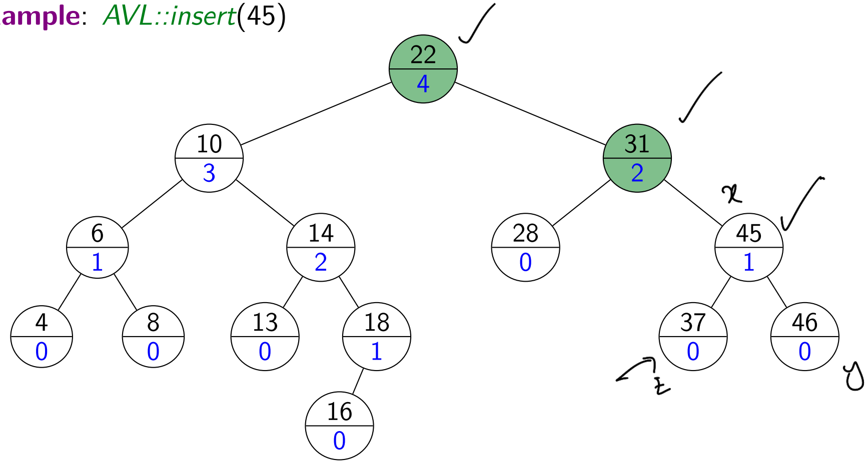
Example: *AVL::insert*(45)



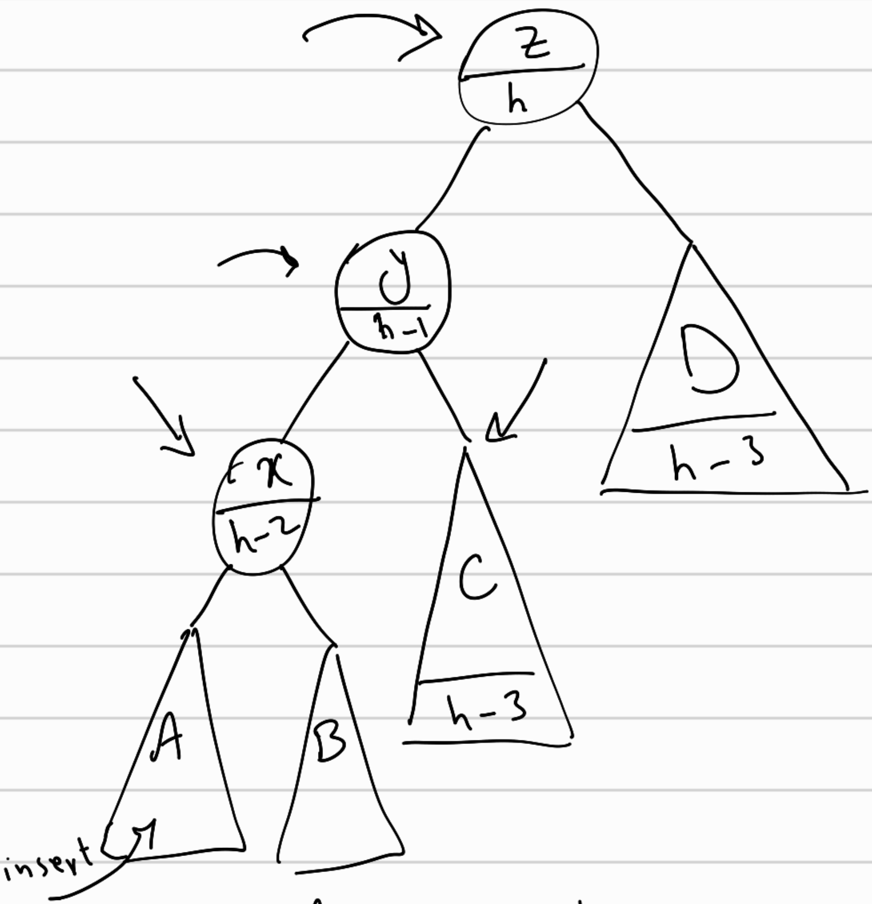
z < x < y

AVL Insertion: Second example

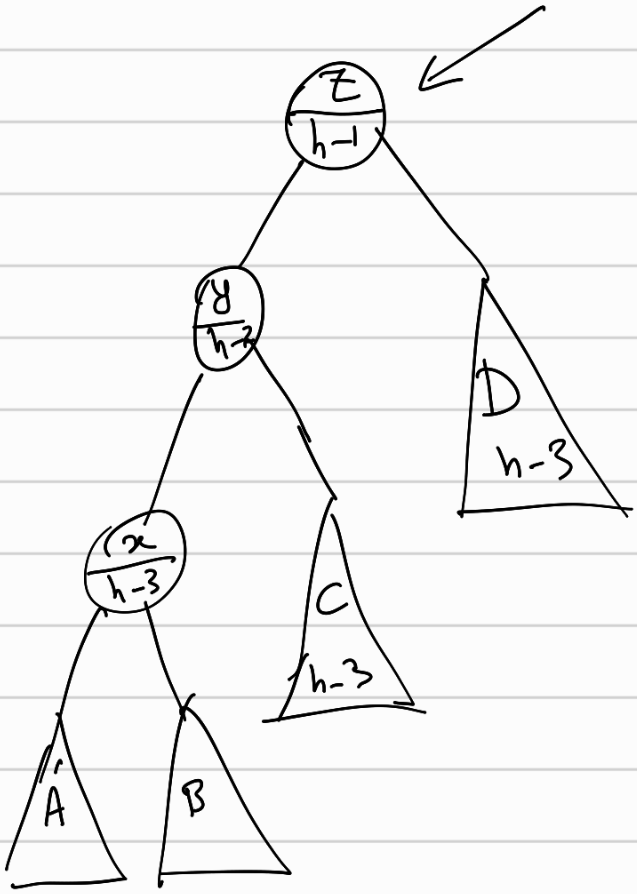
Example: *AVL::insert*(45)



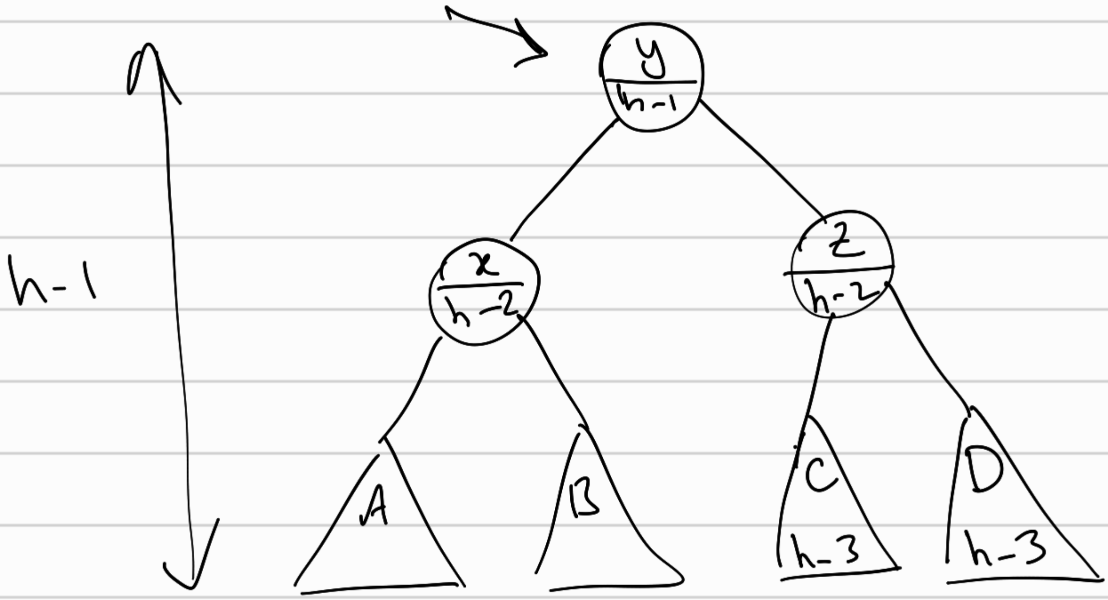
Right rotation:



After insert



Before insert

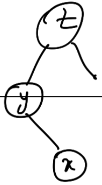


After restructure

1. all nodes are balanced
2. The height after restructure is the same as the height before insert

AVL Deletion

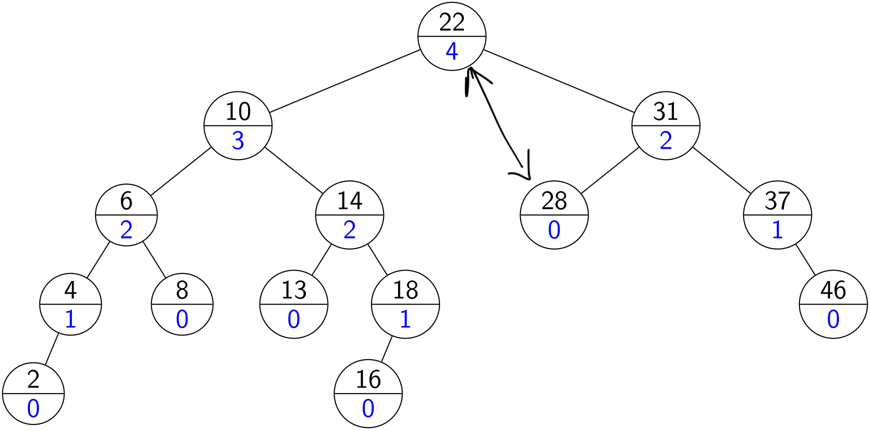
Remove the key k with *BST::delete*.
Find node where *structural* change happened.
(This is not necessarily near the node that had k.)
Go back up to root, update heights, and rotate if needed.




```
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 taller child of z
            Let x be taller child of y (break ties to prefer single rotation)
            z ← restructure(x, y, z)
        // Always continue up the path and fix if needed.
        setHeightFromSubtrees(z)
        z ← z.parent
```

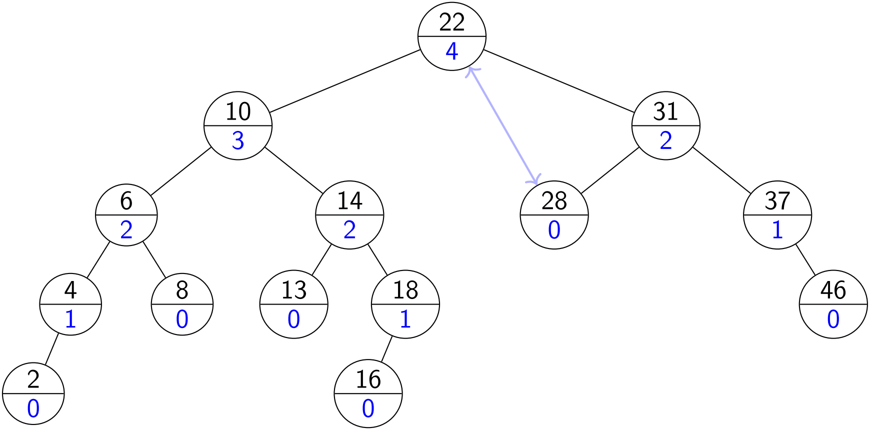
AVL Deletion Example

Example: *AVL::delete*(22)



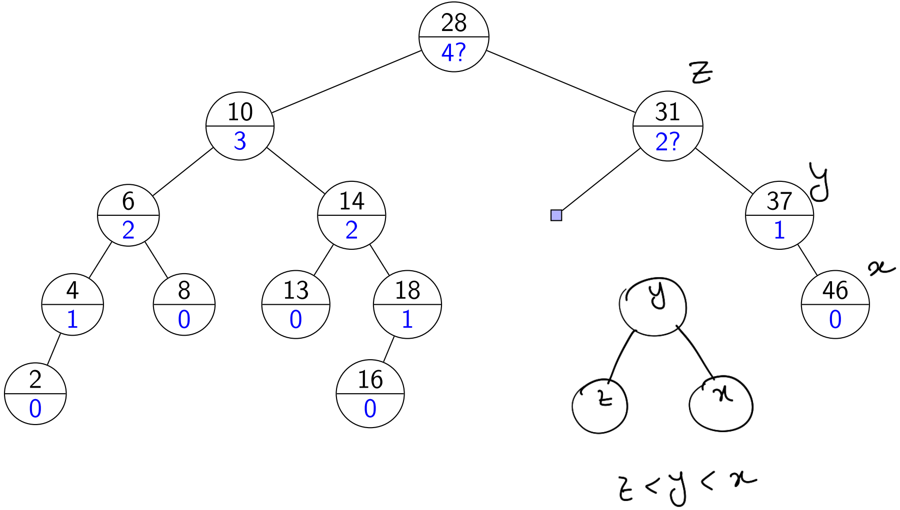
AVL Deletion Example

Example: *AVL::delete*(22)



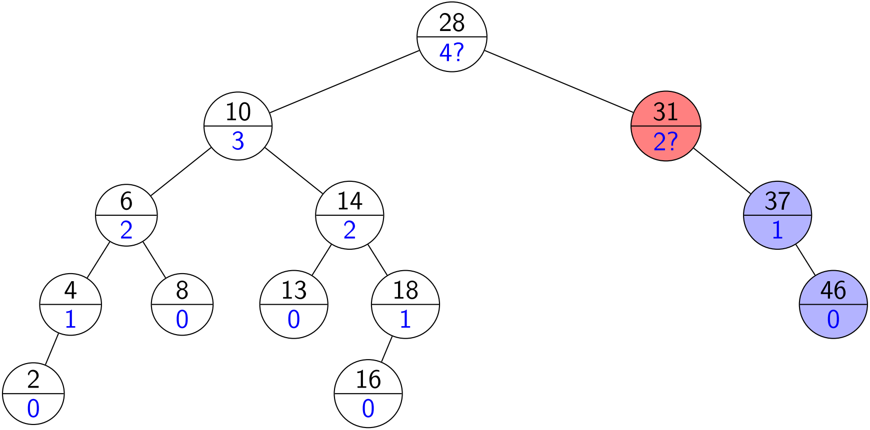
AVL Deletion Example

Example: *AVL::delete*(22)



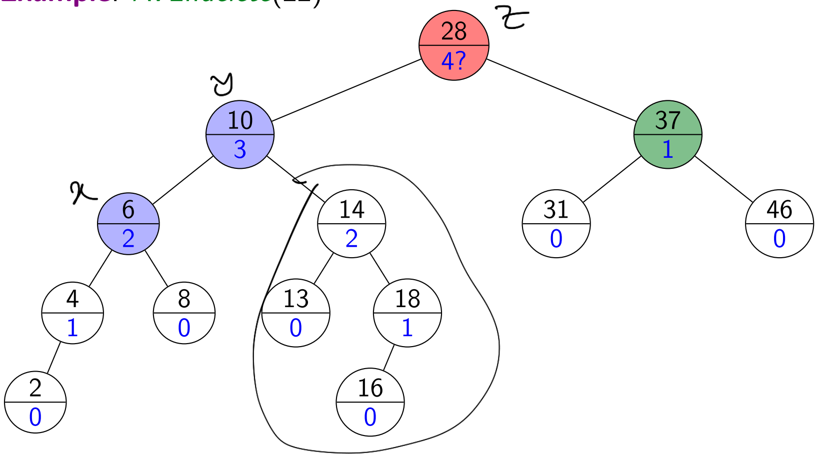
AVL Deletion Example

Example: *AVL::delete*(22)



AVL Deletion Example

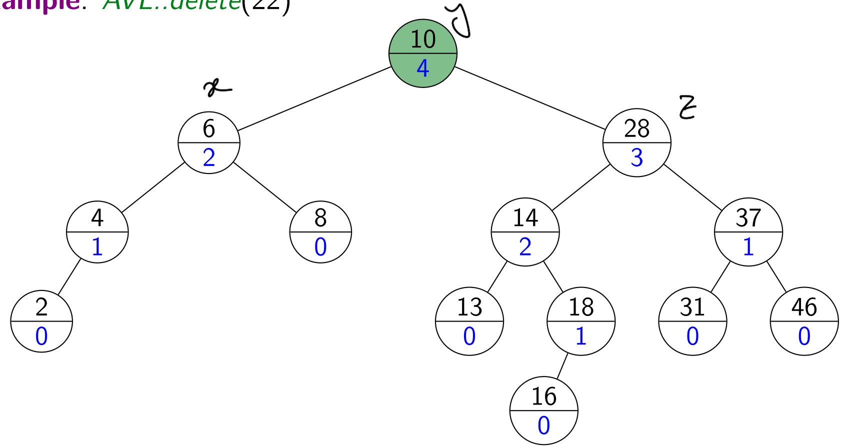
Example: *AVL::delete*(22)



$x < y < z$

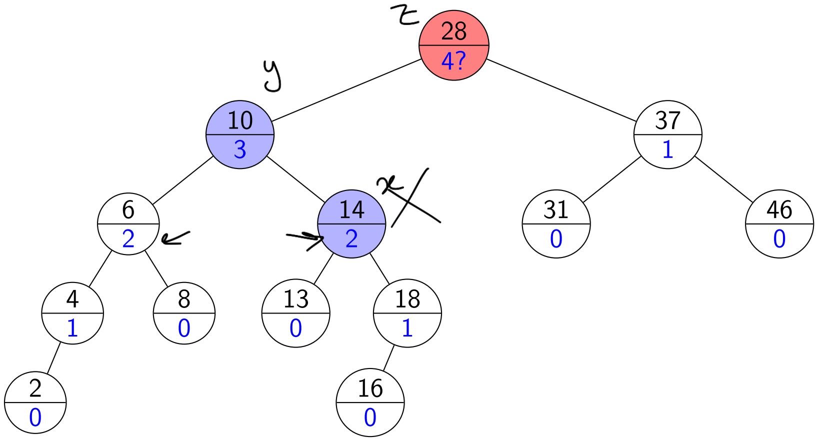
AVL Deletion Example

Example: *AVL::delete*(22)



AVL Deletion Example

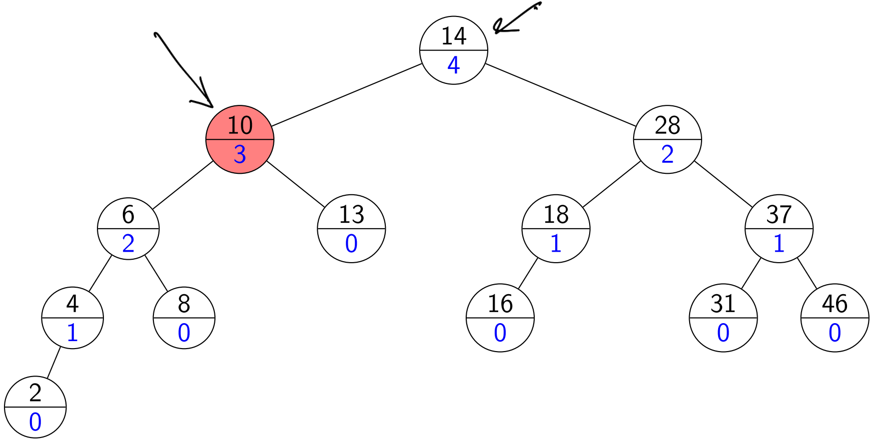
Important: Ties *must* be broken to prefer single rotation.
Consider again the above example. If we applied double-rotation:



AVL Deletion Example

Important: Ties *must* be broken to prefer single rotation.

Consider again the above example. If we applied double-rotation:



Resulting tree is *not* an AVL-tree.

AVL Tree Operations Runtime

search: Just like in BSTs, costs $\Theta(height)$

insert: *BST::insert*, then check & update along path to new leaf

- total cost $\Theta(height)$
- *restructure* restores the height of the subtree to what it was,
- so *restructure* will be called *at most once*.

delete: *BST::delete*, then check & update along path to deleted node

- total cost $\Theta(height)$
- *restructure* may be called $\Theta(height)$ times.

Worst-case cost for all operations is $\Theta(height) = \Theta(\log n)$.

But in practice, the constant is quite large.