

# CS 240 – Data Structures and Data Management

## Module 4: Dictionaries

T. Biedl   É. Schost   O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

# Outline

## 1 Dictionaries and Balanced Search Trees

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

# Outline

## 1 Dictionaries and Balanced Search Trees

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

# 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

## 1 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)

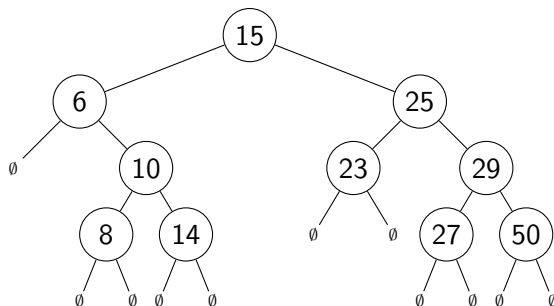
**Structure** Binary tree: all nodes have two (possibly empty) subtrees

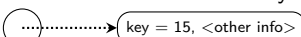
Every node stores a KVP

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.

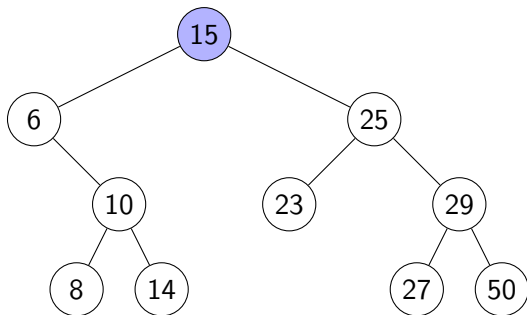


( In our examples we only show the keys, and we show them directly in the node. A more accurate picture would be  )

## 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)

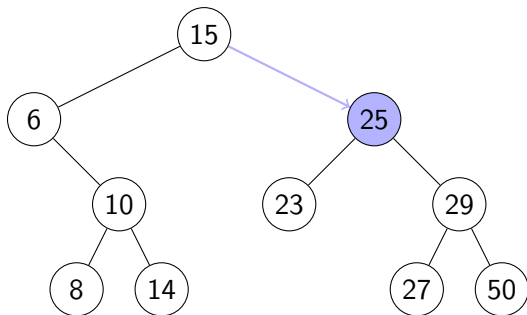




## BST as realization of ADT Dictionary

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

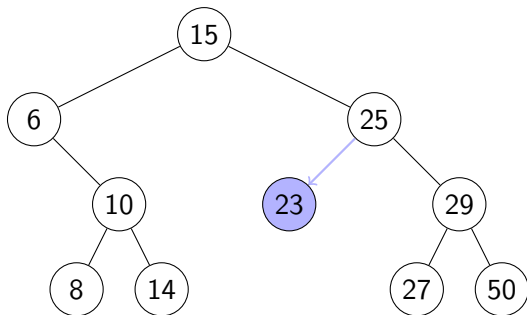
Example: *BST::search*(24)



## 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.

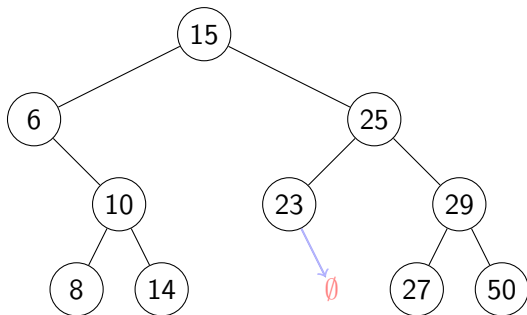
Example: *BST::search*(24)



## 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)

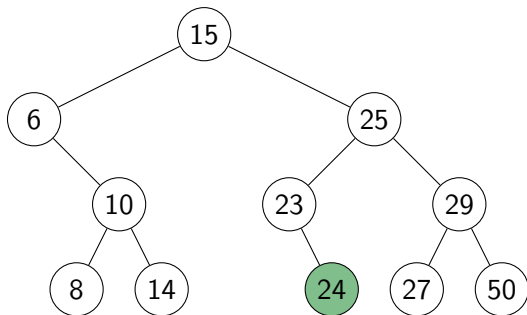


## BST as realization of ADT Dictionary

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

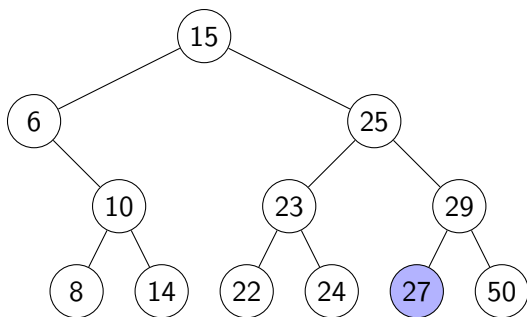
*BST::insert*( $k, v$ ) Search for  $k$ , then insert ( $k, v$ ) as new node

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



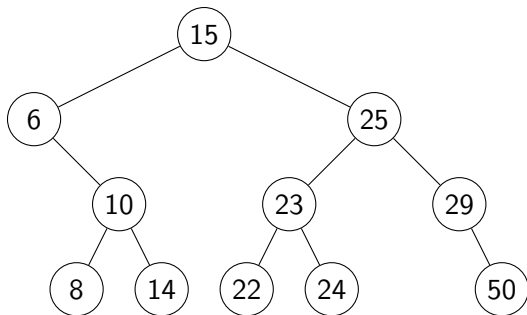
## Deletion in a BST

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



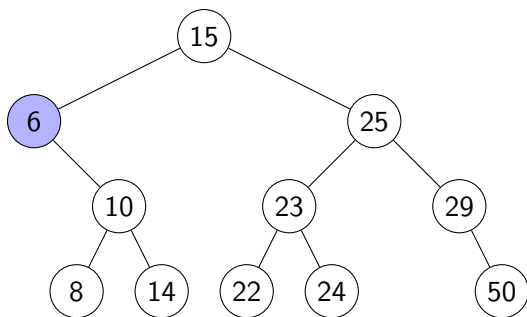
## Deletion in a BST

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



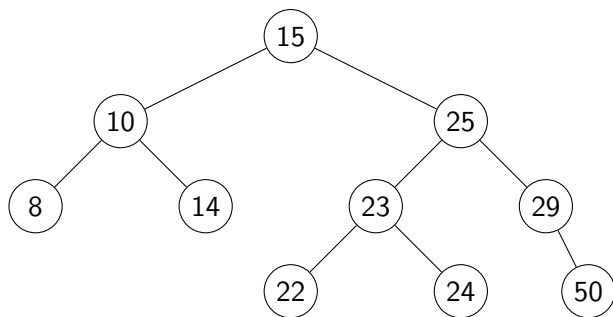
## Deletion in a BST

- 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

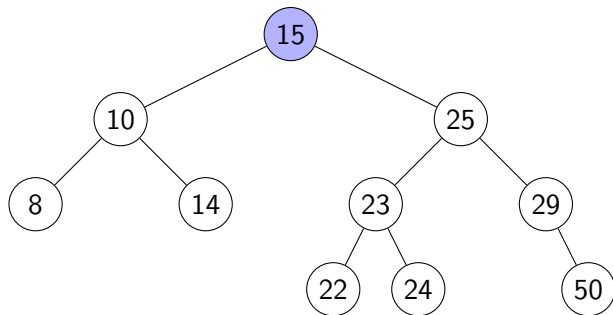
- First search for the node  $x$  that contains the key.
- If  $x$  is a **leaf** (both subtrees are empty), delete it.
- If  $x$  has one non-empty subtree, move child up





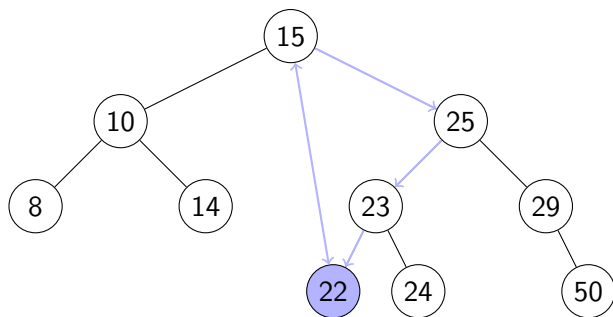
## Deletion in a BST

- 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



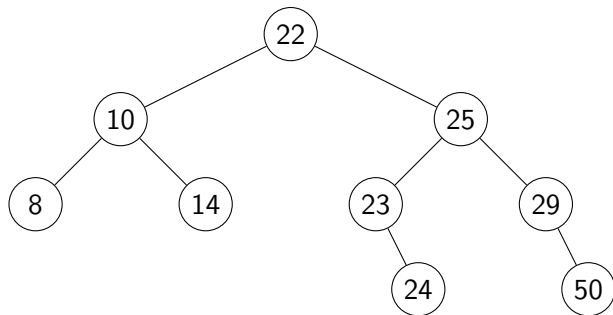
## Deletion in a BST

- 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:

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

# Outline

## 1 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

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

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

$\text{balance}(v) = +1$  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

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

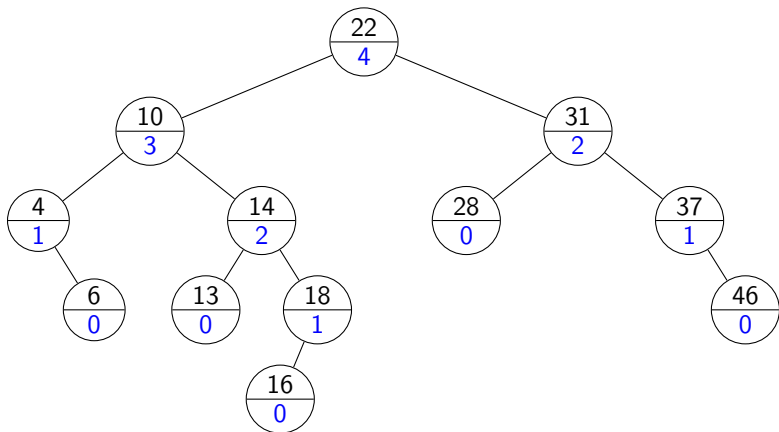
$balance(v) = -1$  means  $v$  is *left-heavy*

$balance(v) = +1$  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

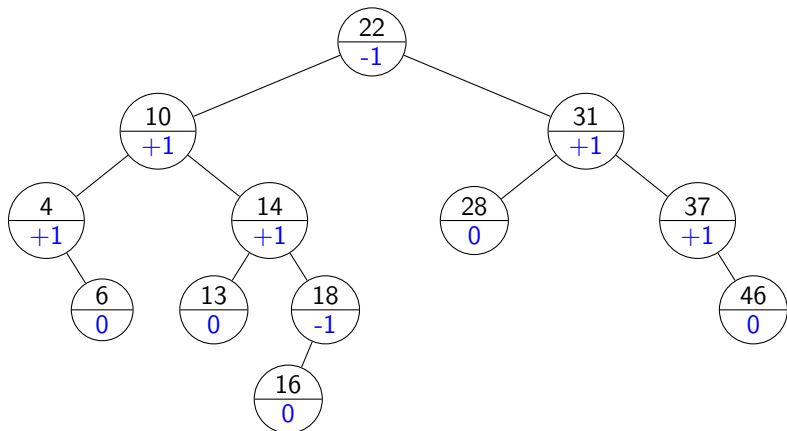
## AVL tree example

(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?

# Outline

## 1 Dictionaries and Balanced Search Trees

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

# 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  $\pm 2$  at node  $z$ , then  $z$  is **unbalanced**. Must re-structure the tree to rebalance.

# AVL insertion

*AVL::insert*( $k, v$ )

1.  $z \leftarrow \text{BST::insert}(k, v)$  // leaf where  $k$  is now stored
2. **while** ( $z$  is not NIL)
3.     **if** ( $|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1$ ) **then**
4.         Let  $y$  be taller child of  $z$
5.         Let  $x$  be taller child of  $y$
6.          $z \leftarrow \text{restructure}(x, y, z)$  // see later
7.         **break** // can argue that we are done
8.     *setHeightFromSubtrees*( $z$ )
9.      $z \leftarrow z.\text{parent}$

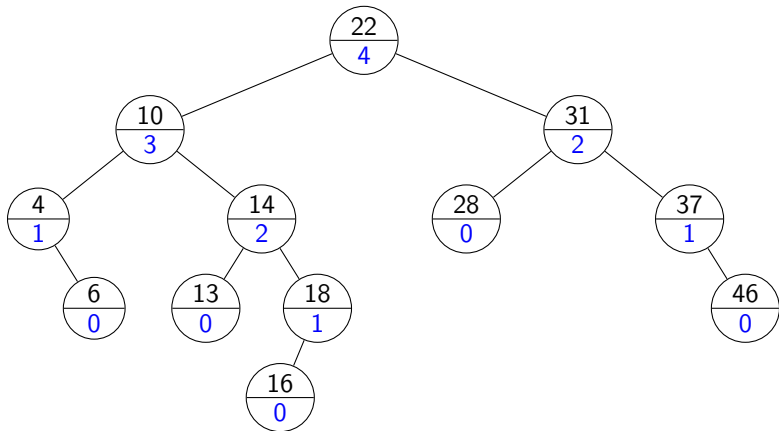
*setHeightFromSubtrees*( $u$ )

1.  $u.\text{height} \leftarrow 1 + \max\{u.\text{left}.\text{height}, u.\text{right}.\text{height}\}$



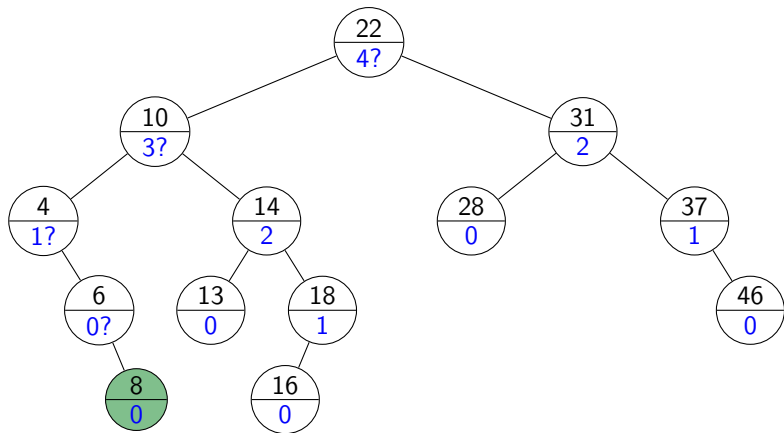
# AVL Insertion Example

Example: *AVL::insert*(8)



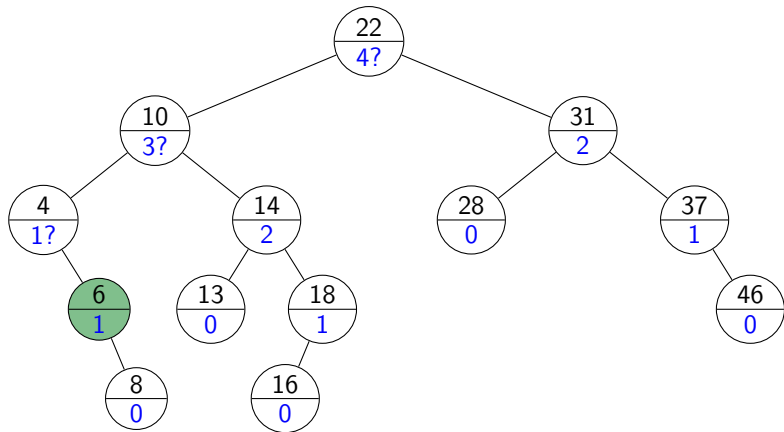
# AVL Insertion Example

Example: *AVL::insert*(8)



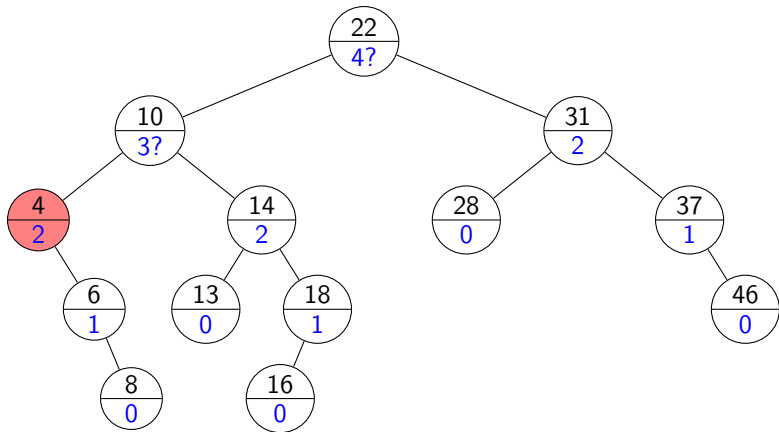
# AVL Insertion Example

Example: *AVL::insert*(8)



# AVL Insertion Example

Example: *AVL::insert*(8)



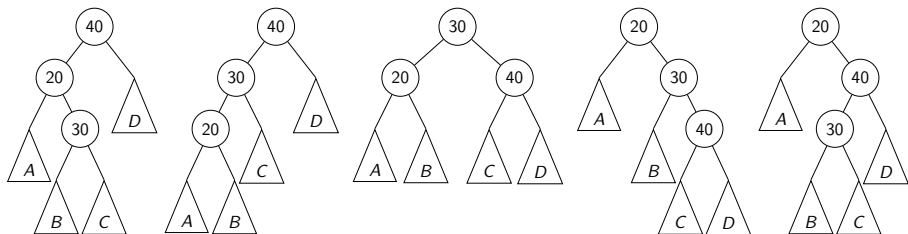
# Outline

## 1 Dictionaries and Balanced Search Trees

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

# How to “fix” an unbalanced AVL tree

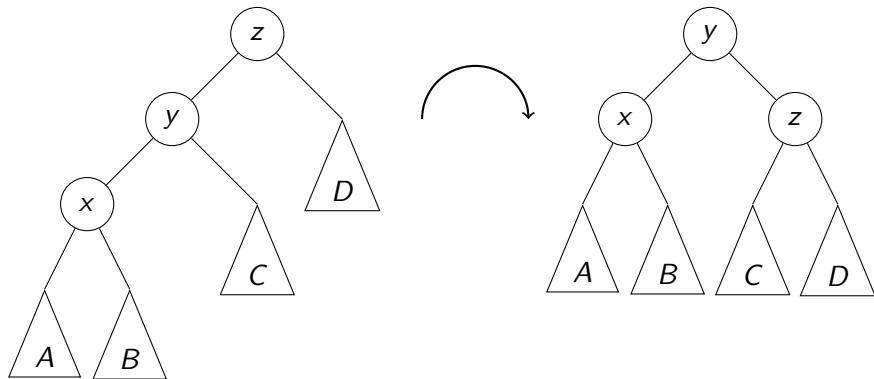
**Note:** there are many different BSTs with the same keys.



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

## Right Rotation

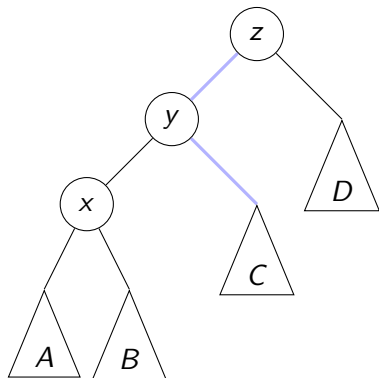
This is a **right rotation** on node  $z$ :



*rotate-right*( $z$ )

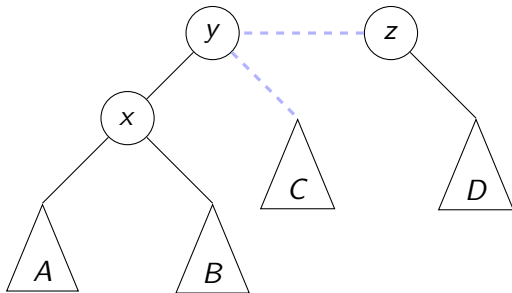
1.  $y \leftarrow z.\text{left}$ ,  $z.\text{left} \leftarrow y.\text{right}$ ,  $y.\text{right} \leftarrow z$
2. *setHeightFromSubtrees*( $z$ ), *setHeightFromSubtrees*( $y$ )
3. **return**  $y$  // returns new root of subtree

## Why do we call this a rotation?

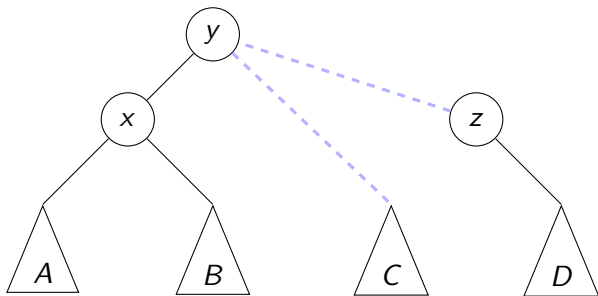




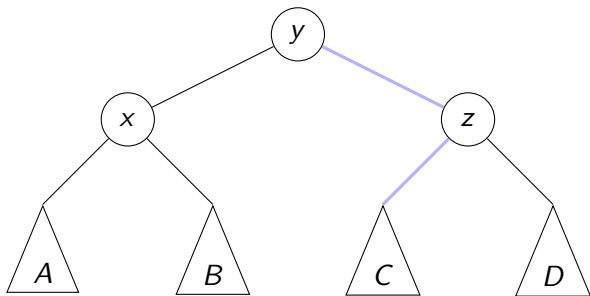
## Why do we call this a rotation?



## Why do we call this a rotation?

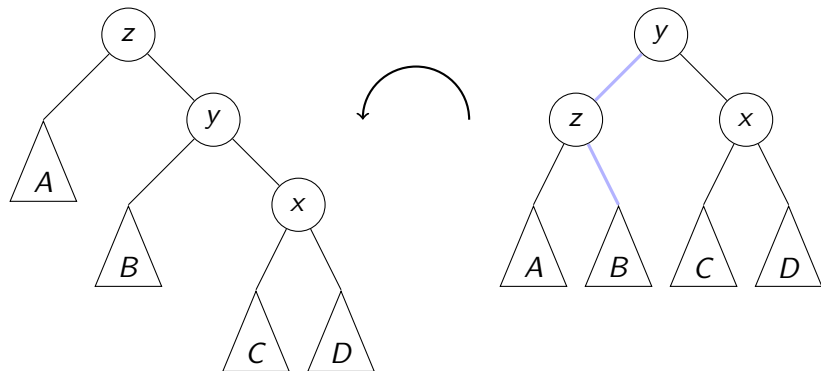


## Why do we call this a rotation?



## Left Rotation

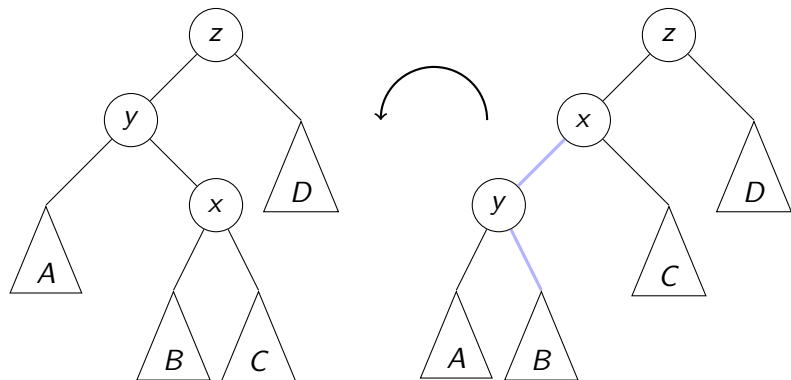
Symmetrically, this is a **left rotation** on node  $z$ :



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

## Double Right Rotation

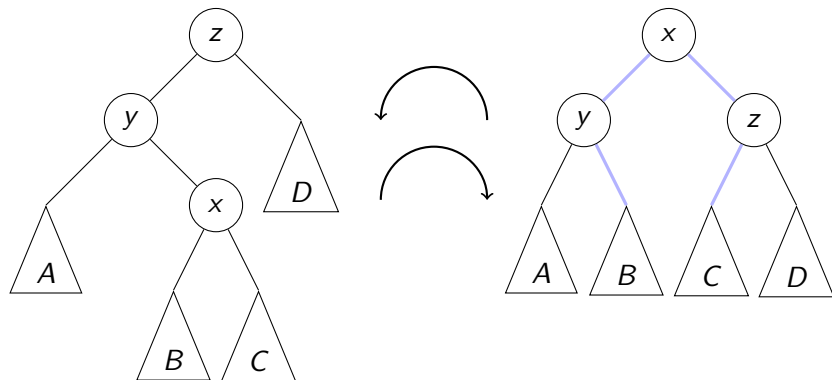
This is a **double right rotation** on node  $z$ :



First, a left rotation at  $y$ .

## Double Right Rotation

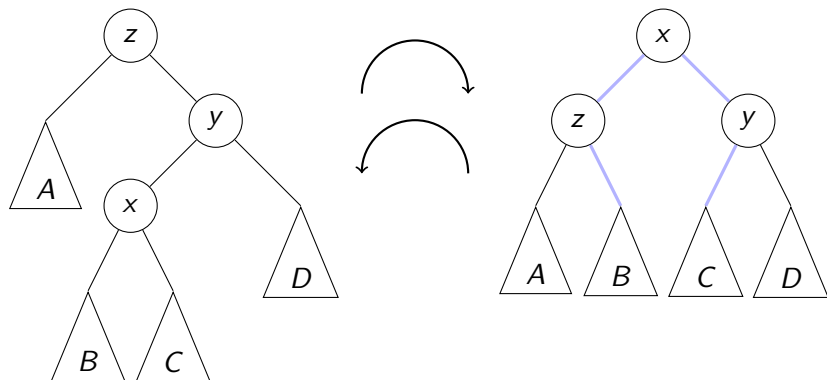
This is a **double right rotation** on node  $z$ :



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  $z$ .


## 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  
 $z.\text{left} \leftarrow$  *rotate-left*( $y$ )  
**return** *rotate-right*( $z$ )

 : // Double-left rotation  
 $z.\text{right} \leftarrow$  *rotate-right*( $y$ )  
**return** *rotate-left*( $z$ )

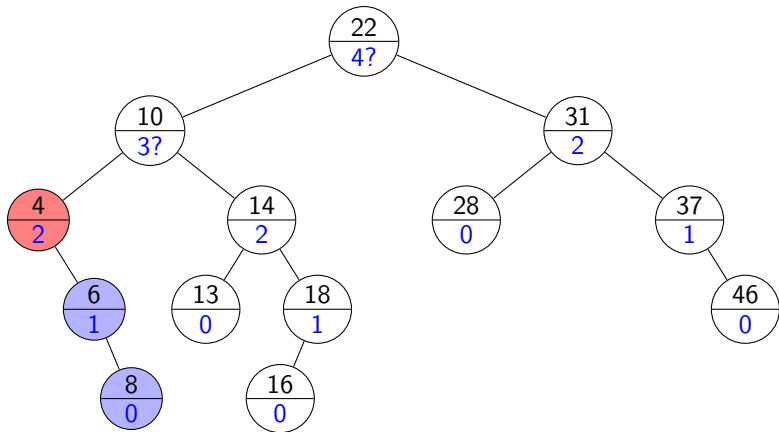
 : // Left rotation  
**return** *rotate-left*( $z$ )

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



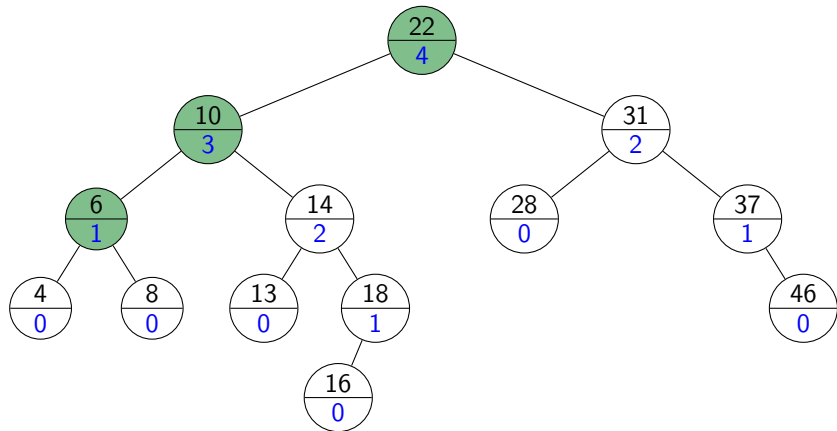
# AVL Insertion Example revisited

Example: *AVL::insert*(8)



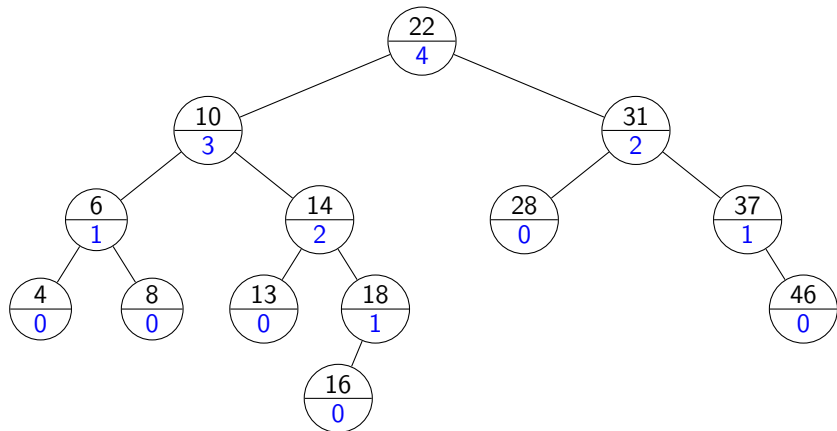
# AVL Insertion Example revisited

Example: *AVL::insert*(8)



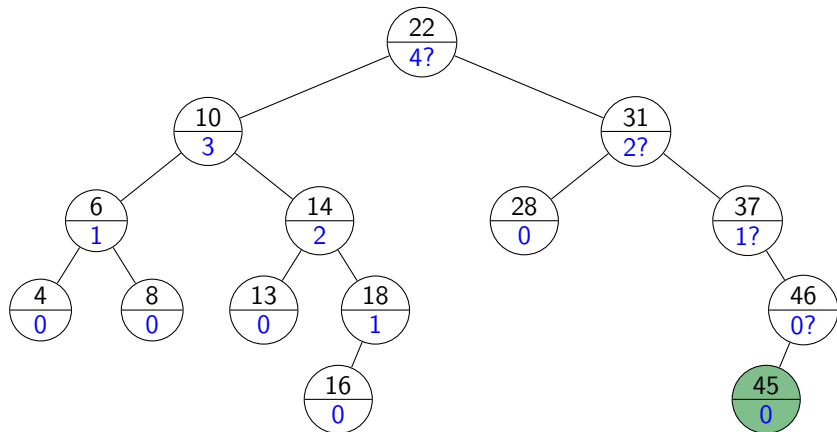
## AVL Insertion: Second example

**Example:** *AVL::insert*(45)



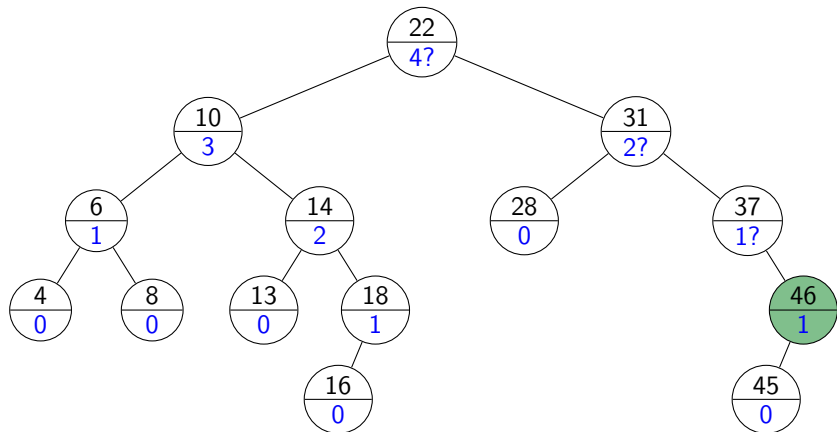
## AVL Insertion: Second example

**Example:** *AVL::insert*(45)



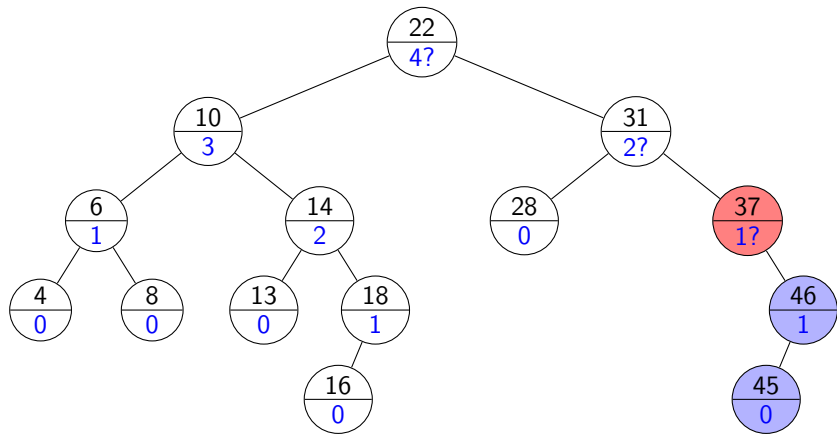
## AVL Insertion: Second example

**Example:** *AVL::insert*(45)



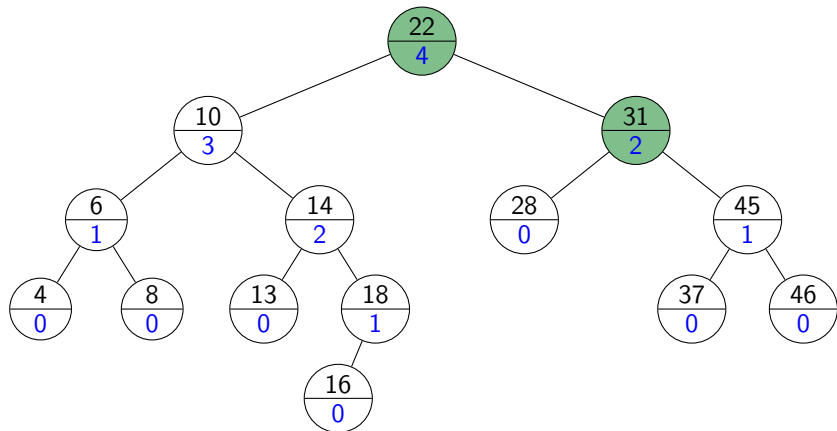
## AVL Insertion: Second example

Example: *AVL::insert*(45)



## AVL Insertion: Second example

**Example:** *AVL::insert*(45)



## 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.

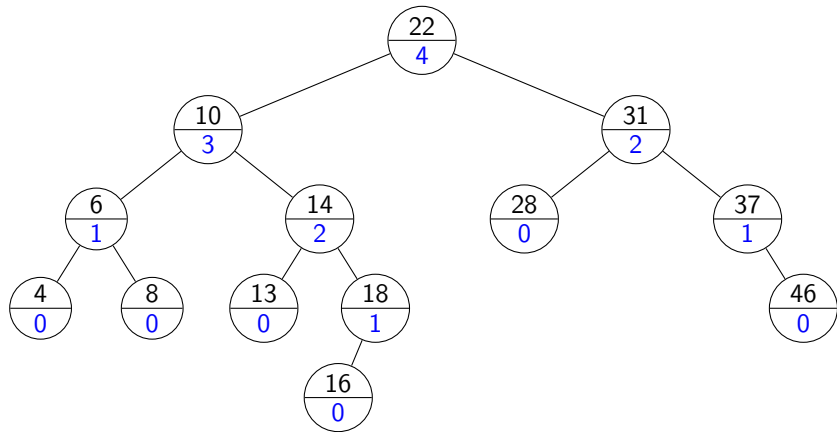
```
AVL::delete(k)
```

1.  $z \leftarrow \text{BST::delete}(k)$
2. // Assume  $z$  is the parent of the BST node that was removed
3. **while** ( $z$  is not NIL)
4.     **if** ( $|z.\text{left}.\text{height} - z.\text{right}.\text{height}| > 1$ ) **then**
5.         Let  $y$  be taller child of  $z$
6.         Let  $x$  be taller child of  $y$  (break ties to prefer single rotation)
7.          $z \leftarrow \text{restructure}(x, y, z)$
8.     // *Always* continue up the path and fix if needed.
9.     *setHeightFromSubtrees*( $z$ )
10.      $z \leftarrow z.\text{parent}$



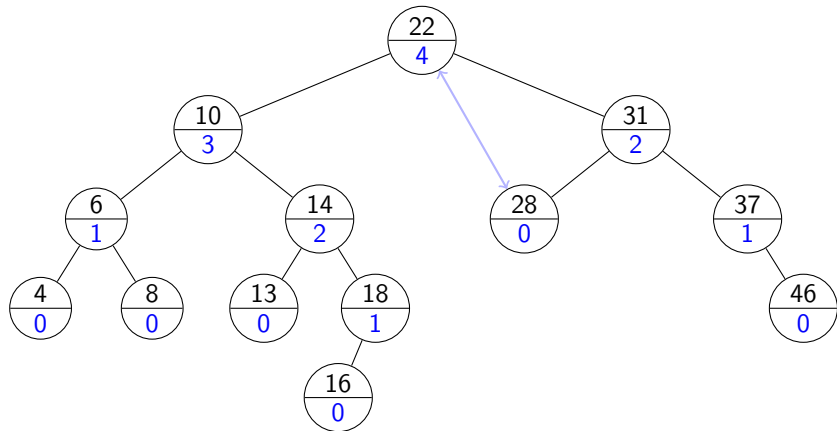
# AVL Deletion Example

Example: *AVL::delete*(22)



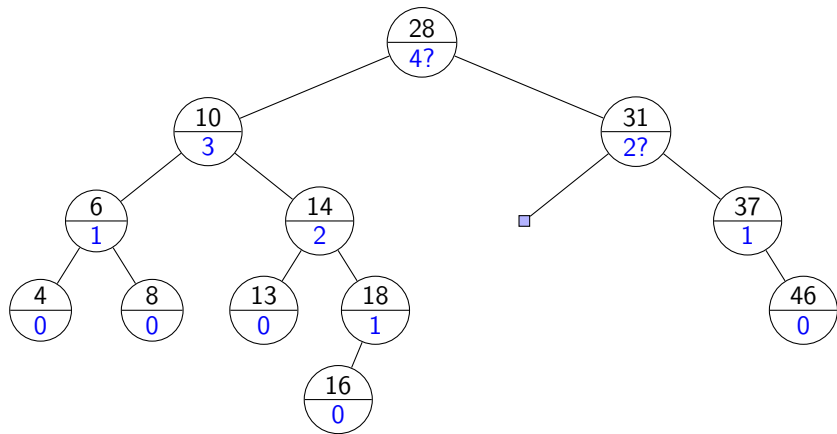
# AVL Deletion Example

Example: *AVL::delete*(22)



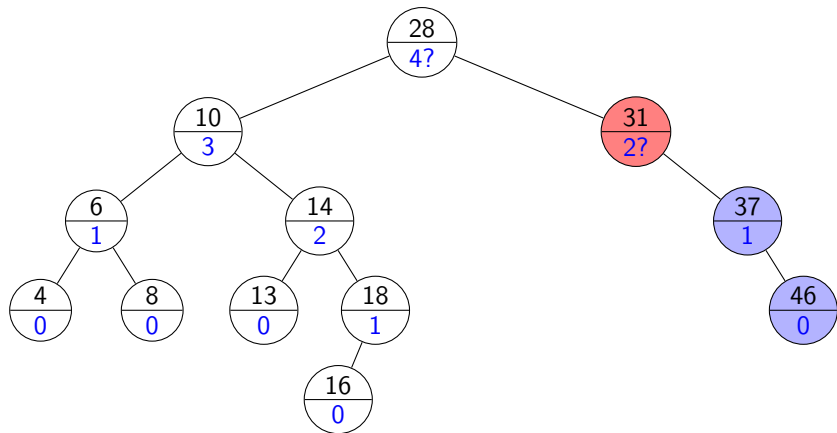
# AVL Deletion Example

Example: *AVL::delete*(22)



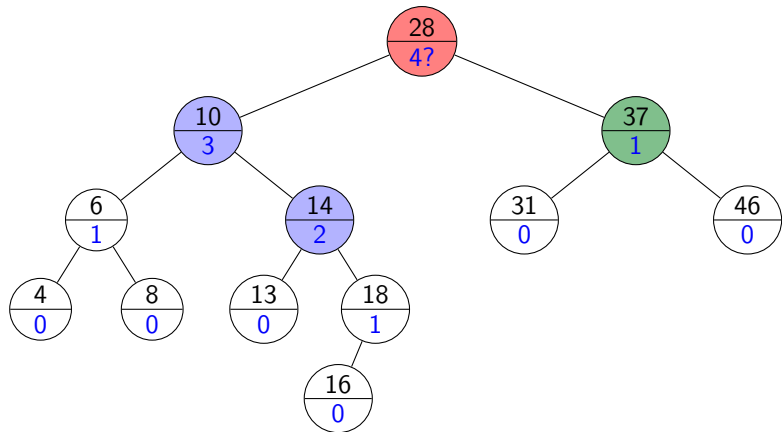
# AVL Deletion Example

Example: *AVL::delete*(22)



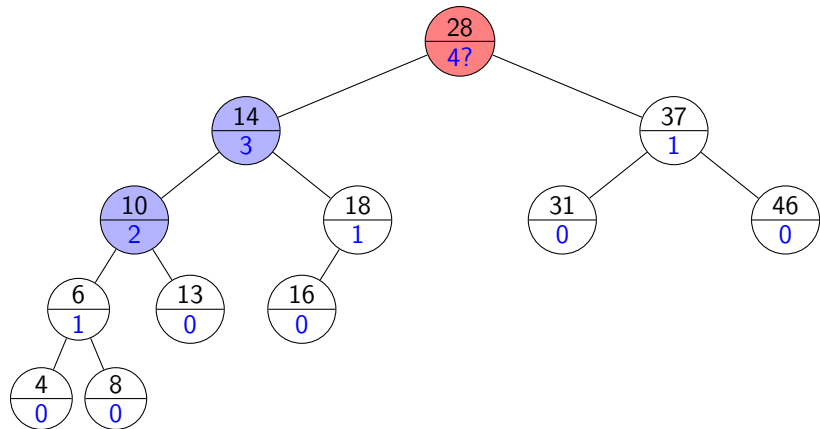
# AVL Deletion Example

Example: *AVL::delete*(22)



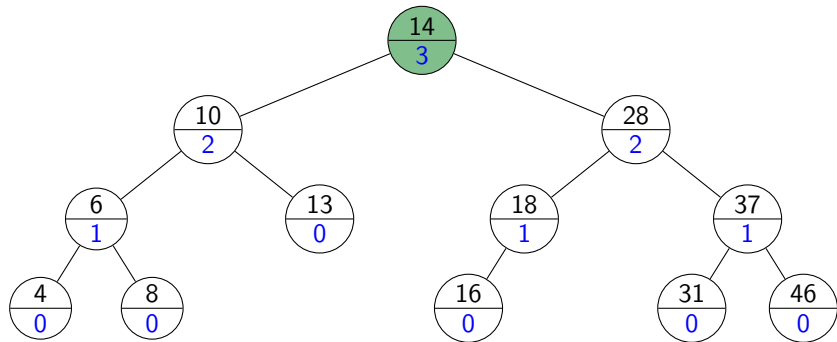
# AVL Deletion Example

Example: *AVL::delete*(22)



# AVL Deletion Example

Example: *AVL::delete*(22)



# AVL Tree Operations Runtime

**search:** Just like in BSTs, costs  $\Theta(\text{height})$

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

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

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

But in practice, the constant is quite large.