CS 240 – Data Structures and Data Management

Module 4: Dictionaries

Olga Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025-01-30 12:47

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restructuring a BST: Rotations
   - AVL insertion revisited
   - Deletion in AVL Trees

Outline

4. Dictionaries and Balanced Search Trees
   - ADT Dictionary
   - Binary Search Trees
   - AVL Trees
   - Insertion in AVL Trees
   - Restructuring a BST: Rotations
   - AVL insertion revisited
   - Deletion in AVL Trees

ADT Dictionary (review)

Dictionary: 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 *lookup*(k))
- *insert*(k, v)
- *delete*(k) (also called *remove*(k))
- optional: *successor*, *merge*, *is-empty*, *size*, *etc.*

Examples: symbol table, license plate database

Elementary Realizations (review)

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

Elementary Realizations (review)

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

We commonly make one more assumption (to keep pseudo-code simple):

- Dictionary is non-empty both before and after operation.

  (In a real-life implementation you would have to treat these special cases.)

Elementary Realizations (review)

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

We commonly make one more assumption (to keep pseudo-code simple):

- Dictionary is non-empty both before and after operation.

  (In a real-life implementation you would have to treat these special cases.)

Easy realizations:

	search	*insert*	*delete*
unsorted list/array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
binary search tree	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$	$\Theta(\mathit{height})$

Binary Search (review)

Only applies to a *sorted array*:

0	1	2	3	4	5	6
30	40	70	90	100	120	140

binary-search(A, n, k)
A: Sorted array of size n, k: key
1. $\ell \leftarrow 0$, $r \leftarrow n-1$
2. **while** $(\ell \leq r)$
3. $\quad m \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$
4. $\quad$ **if** $(A[m]$ equals $k)$ **then return** "found at $A[m]$"
5. $\quad$ **else if** $(A[m] < k)$ **then** $\ell \leftarrow m+1$
6. $\quad$ **else** $r \leftarrow m-1$
7. **return** "not found, but would be between $A[\ell-1]$ and $A[\ell]$"

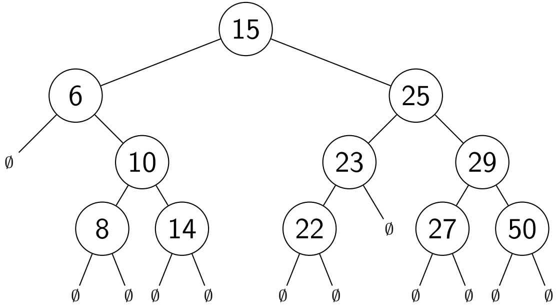
We will return to binary search (and sometimes improve it!) later.

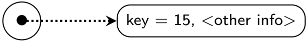
Outline

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.



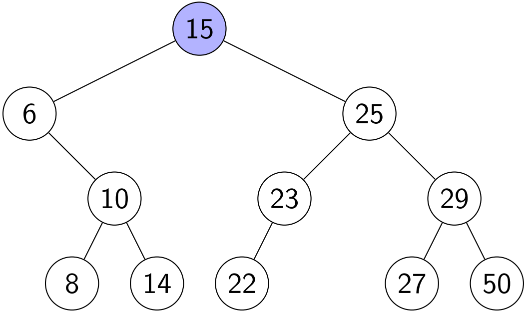
(In our examples we only show the keys, and we show them directly in the node. A more accurate picture would be 

)

BST as realization of ADT Dictionary (review)

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

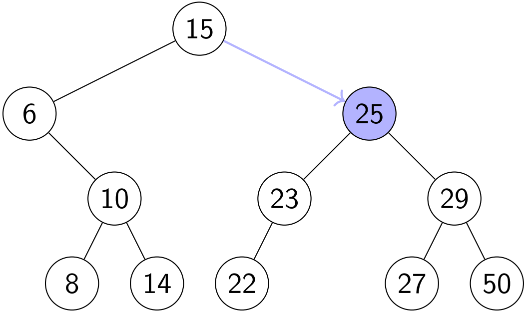
Example: *BST::search*(24)



BST as realization of ADT Dictionary (review)

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

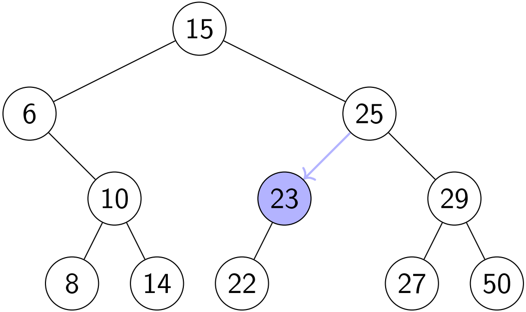
Example: *BST::search*(24)



BST as realization of ADT Dictionary (review)

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

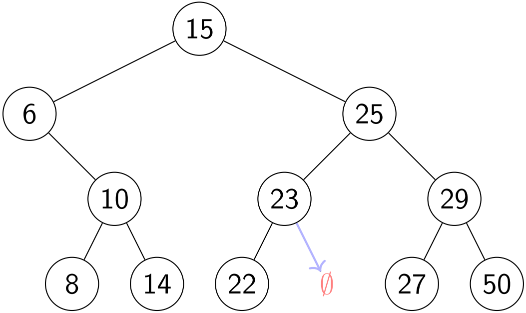
Example: *BST::search*(24)



BST as realization of ADT Dictionary (review)

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)

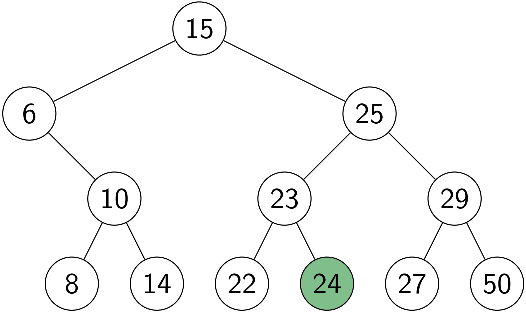


BST as realization of ADT Dictionary (review)

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

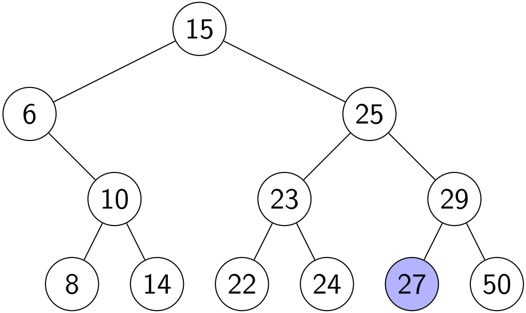
BST::insert(k, v) Search for k, then insert (k, v) as new node

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



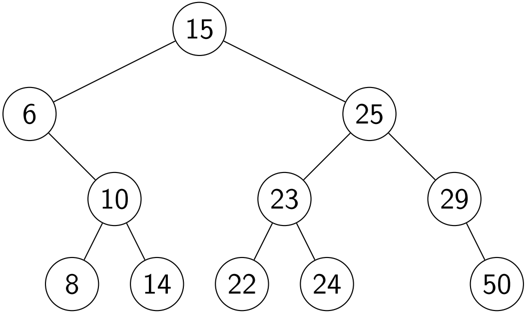
Deletion in a BST

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



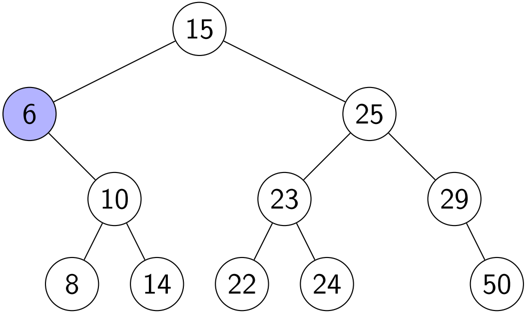
Deletion in a BST

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



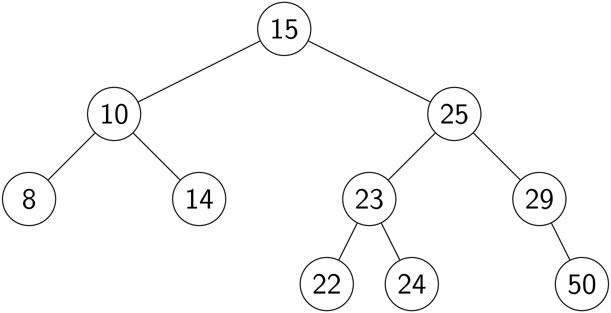
Deletion in a BST

- 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

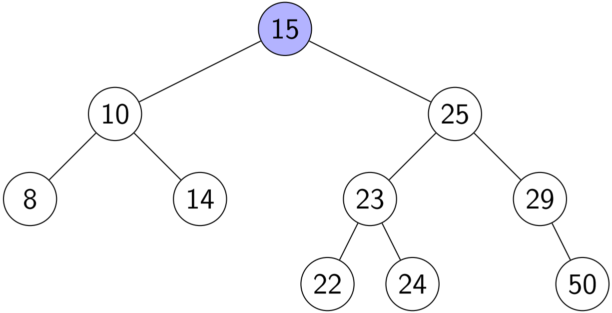
- First search for the node x that contains the key.
- If x is a **leaf** (both subtrees are empty), delete it.
- If x has one non-empty subtree, move child up



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** node and then delete that node.

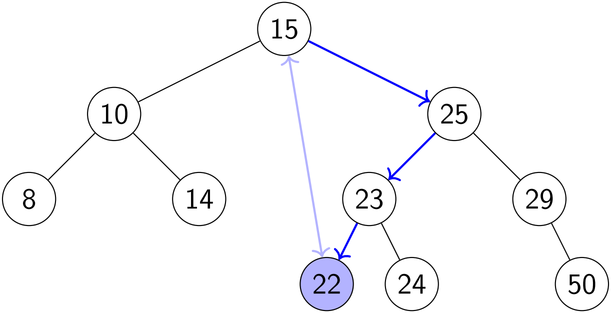
(Successor: next-smallest among all keys in the dictionary.)



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** node and then delete that node.

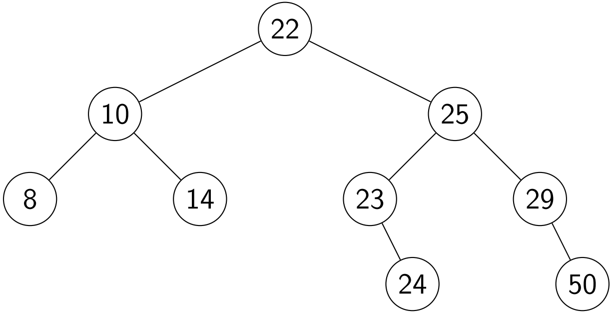
(Successor: next-smallest among all keys in the dictionary.)



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** node and then delete that node.

(Successor: next-smallest among all keys in the dictionary.)



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 $h \geq \log(n+1) - 1$
  (Layer i has at most 2^i nodes. So $n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$).

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 $h \geq \log(n+1) - 1$
  (Layer i has at most 2^i nodes. So $n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$).

Goal: Create subclasses of BSTs where the height is *always* good.

- Impose a structural property.
- Argue that the property implies logarithmic height.
- Discuss how to maintain the property during operatons.

Outline

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.

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.

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

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

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

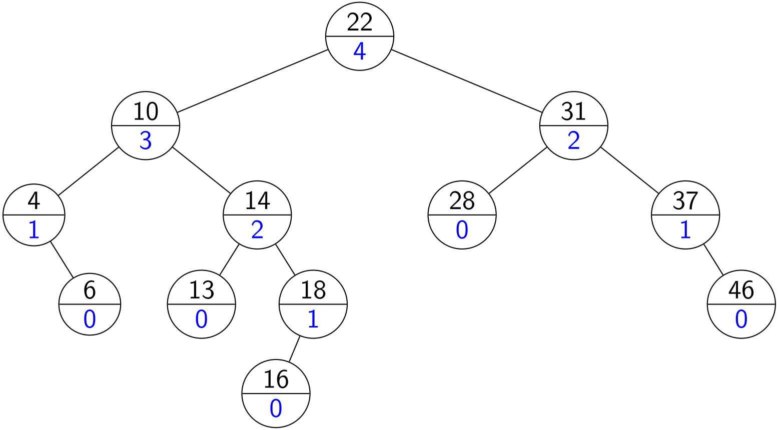
$\textit{balance}(v) = +1$ means v is *right-heavy*

- Need to store at each node v the height of the subtree rooted at it

(There are ways to implement AVL-trees where we only store *balance*(v), so fewer bits. But the code gets more complicated (no details).)

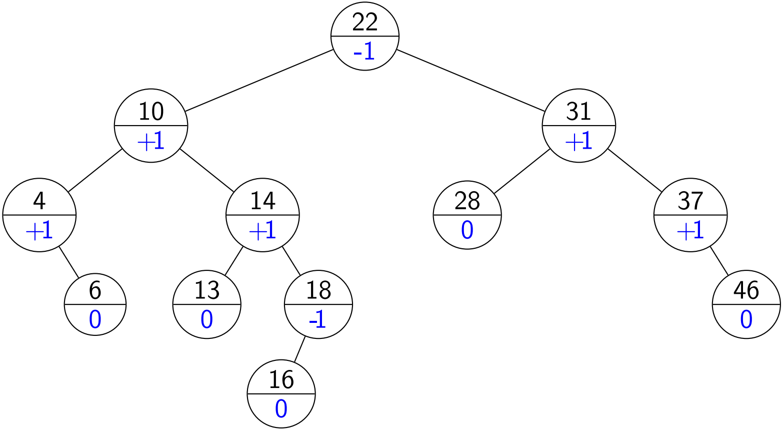
AVL tree example

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



AVL tree example

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



- Saves space (2 bits vs. 1 integer per node)
- Pseudo-code gets a lot more complicated $\rightsquigarrow$ not done here

Height of an AVL tree

Theorem: An AVL tree on n nodes has $\Theta(\log n)$ height.
$\Rightarrow$ *search*, *BST::insert*, *BST::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

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.

  (We assume for this that we have parent-links. This can be avoided if *BST::insert* returns the full path to z.)

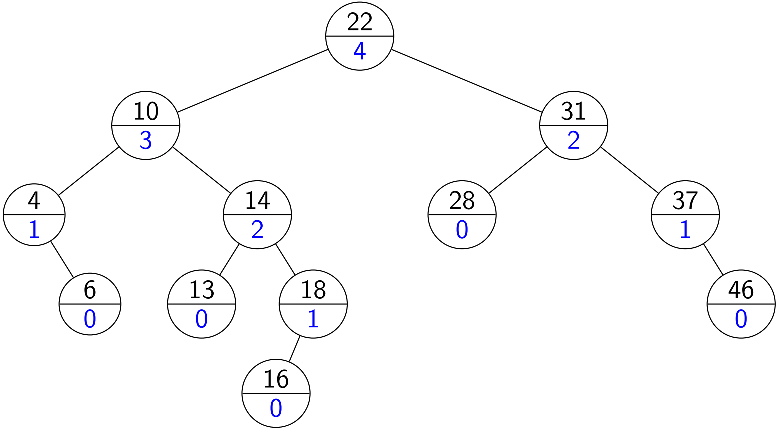
- Update height (easy to do in constant time):

  set-height-from-subtrees(u)
  1. $u.height \leftarrow 1 + \max\{u.left.height, u.right.height\}$

- If the height difference becomes ± 2 at node z, then z is **unbalanced**. Must re-structure the tree to rebalance.

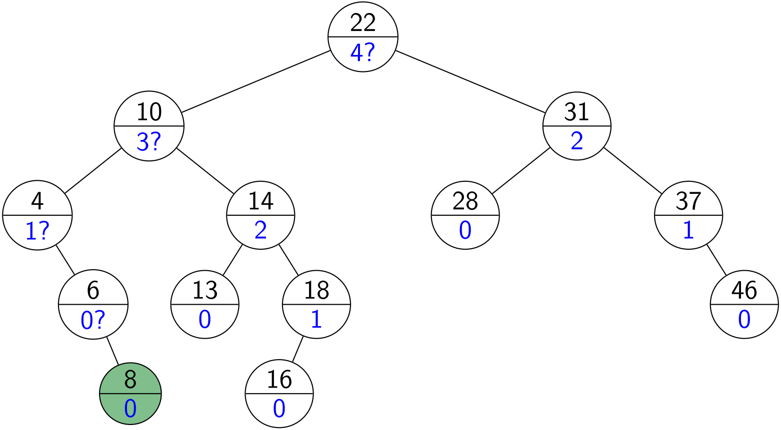
AVL Insertion Example

Example: *AVL::insert*(8)



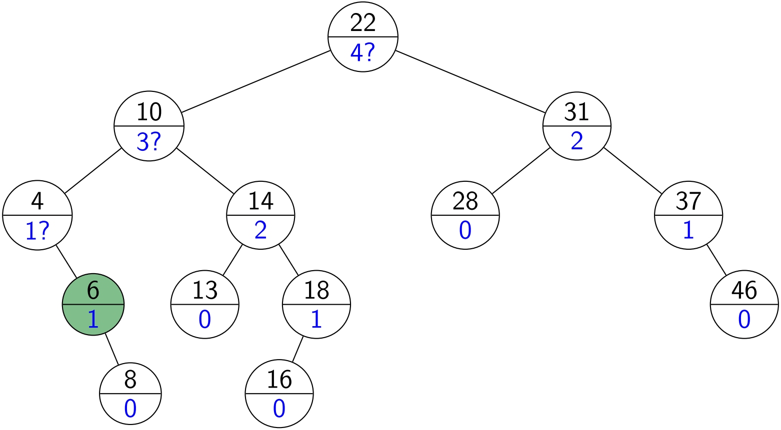
AVL Insertion Example

Example: *AVL::insert*(8)



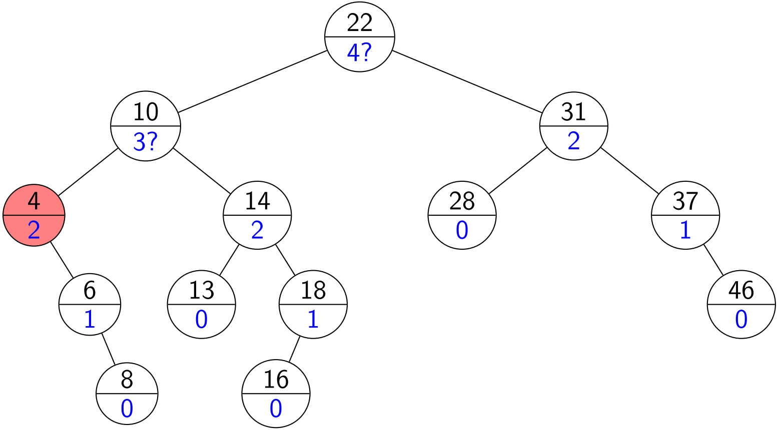
AVL Insertion Example

Example: *AVL::insert*(8)



AVL Insertion Example

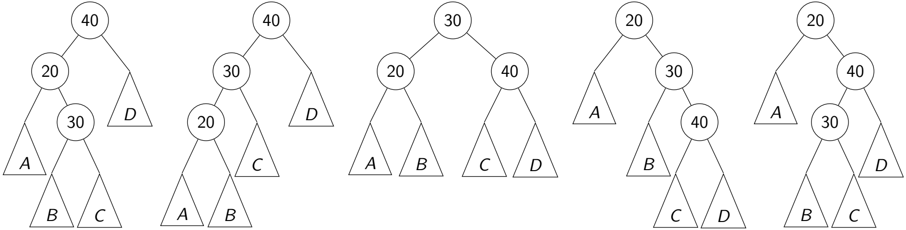
Example: *AVL::insert*(8)



Outline

Changing structure without changing order

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

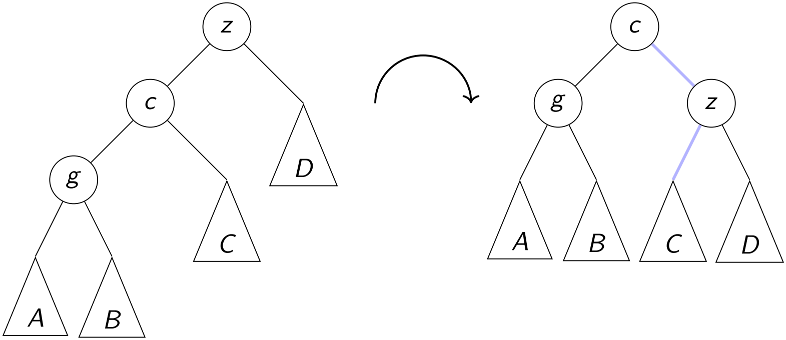


Goal: Change the *structure* locally nodes without changing the *order*.

Longterm goal: Restructure such the subtree becomes balanced.

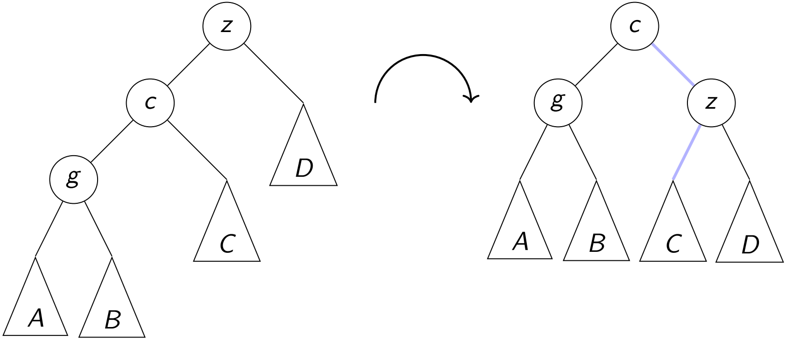
Right Rotation

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



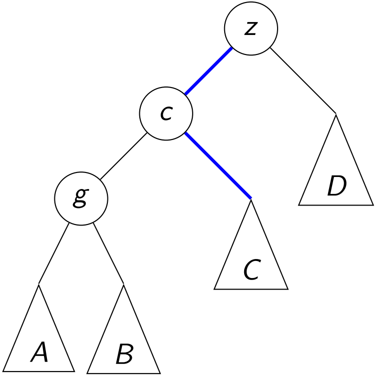
Right Rotation

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

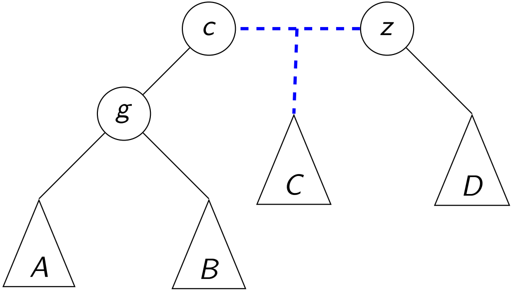


Note: Only $O(1)$ links are changed. Useful to fix left-left imbalance.

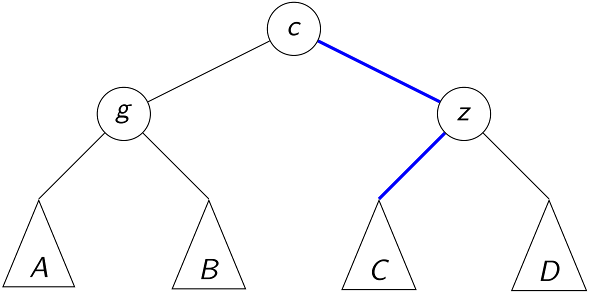
Why do we call this a rotation?



Why do we call this a rotation?



Why do we call this a rotation?



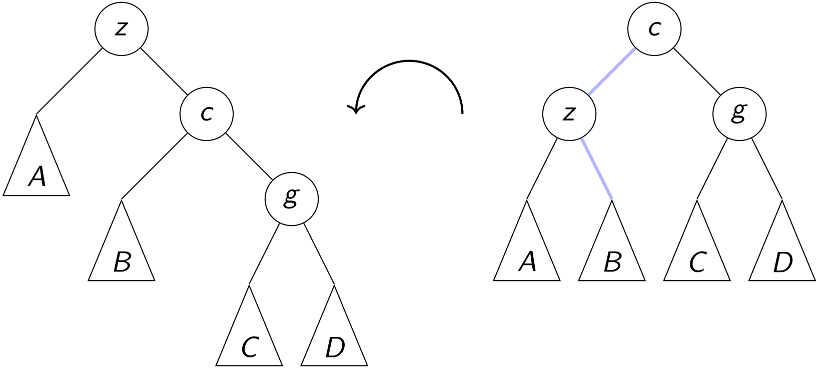
Right Rotation Pseudocode

```
rotate-right(z)
    c ← z.left

    // fix links connecting to above
    c.parent ← (p ← z.parent)
    if p = NULL then root ← c else
        if p.left = z then p.left ← c else p.right ← c

    // actual rotation
    z.left ← c.right, c.right.parent ← z
    c.right ← z, z.parent ← c

    set-height-from-subtrees(z), set-height-from-subtrees(c)
    return c        // returns new root of subtree
```

Run-time: $O(1)$

Left Rotation

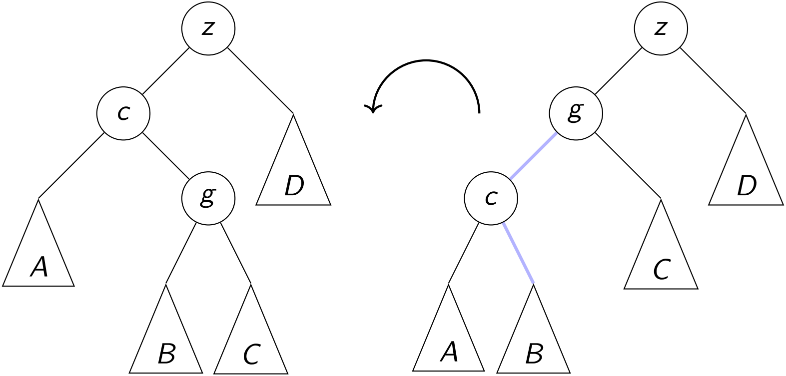
Symmetrically, this is a **left rotation** on node z:



Again, only $O(1)$ links need to be changed. Useful to fix right-right imbalance.

Double Right Rotation

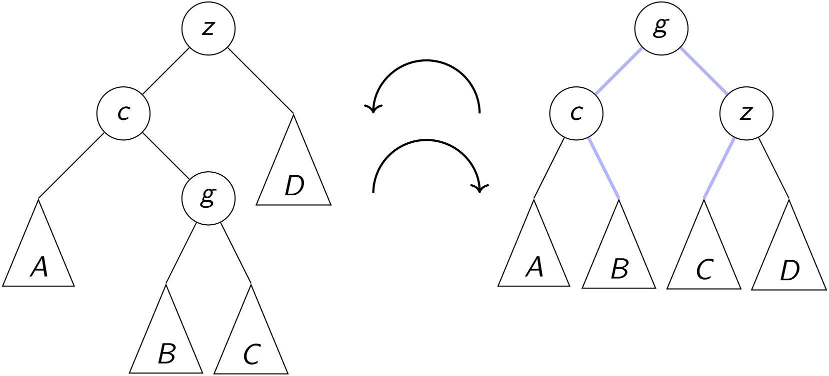
This is a **double right rotation** on node z:



First, a left rotation at c.

Double Right Rotation

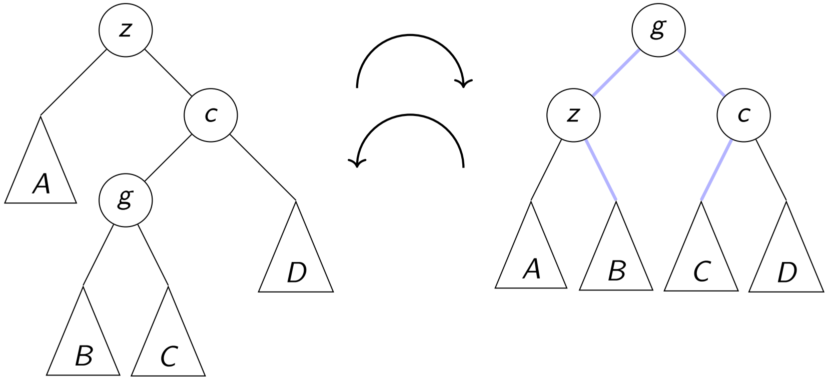
This is a **double right rotation** on node z:



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

Double Left Rotation

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



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

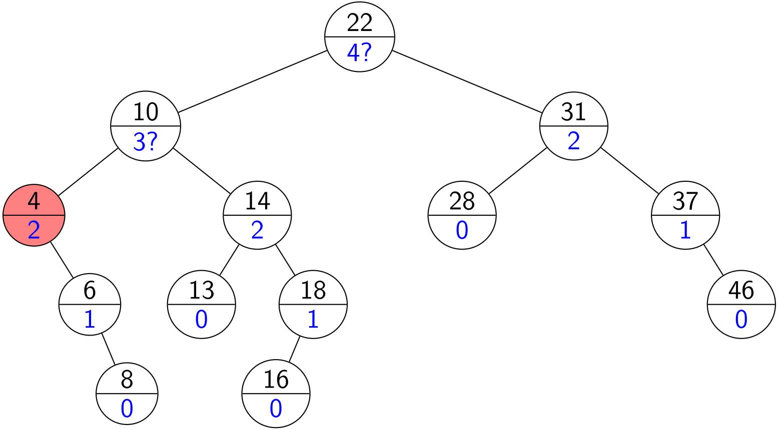
Outline

4 Dictionaries and Balanced Search Trees

- ADT Dictionary
- Binary Search Trees
- AVL Trees
- Insertion in AVL Trees
- Restructuring a BST: Rotations
- AVL insertion revisited
- Deletion in AVL Trees

AVL Insertion Example revisited

Example: *AVL::insert*(8)



AVL insertion revisited

- Imbalance at z: do (single or double) rotation
- Choose c as child where subtree has bigger height.

```
AVL::insert(k, v)
    z ← BST::insert(k, v)   // leaf where k is now stored
    while (z is not NULL)
        if (|z.left.height − z.right.height| > 1) then
            Let c be taller child of z
            Let g be taller child of c (so grandchild of z)
            z ← restructure(g, c, z) // see later
            break        // can argue that we are done
        set-height-from-subtrees(z)
        z ← z.parent
```

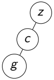
Can argue: For insertion *one* rotation restores all heights of subtrees.

$\Rightarrow$ No further imbalances, can stop checking.

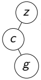
Fixing a slightly-unbalanced AVL tree

restructure(g, c, z)
node g is child of c which is child of z

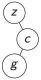
1. **case**



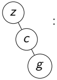
: `// Right rotation`
$u \leftarrow$ *rotate-right*(z)



: `// Double-right rotation`
rotate-left(c)
$u \leftarrow$ *rotate-right*(z)



: `// Double-left rotation`
rotate-right(c)
$u \leftarrow$ *rotate-left*(z)



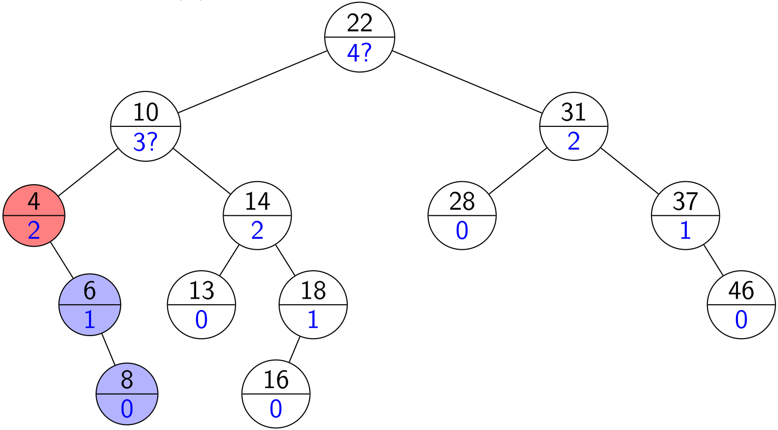
: `// Left rotation`
$u \leftarrow$ *rotate-left*(z)

2. **return** u

Rule: The middle key of g, c, z becomes the new root.

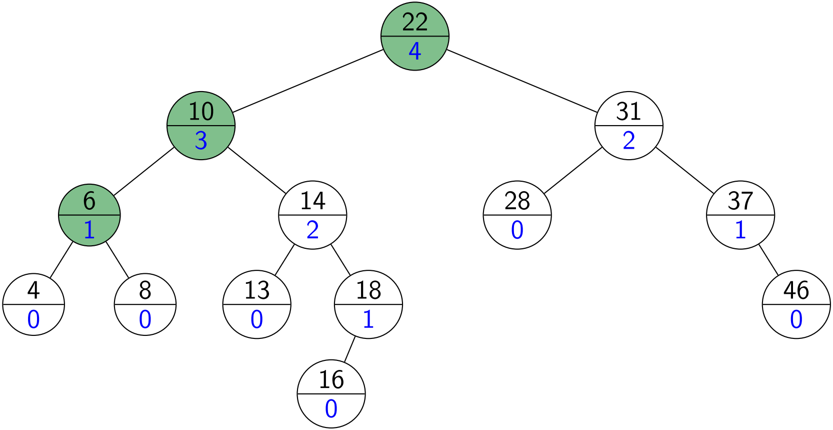
AVL Insertion Example revisited

Example: *AVL::insert*(8)



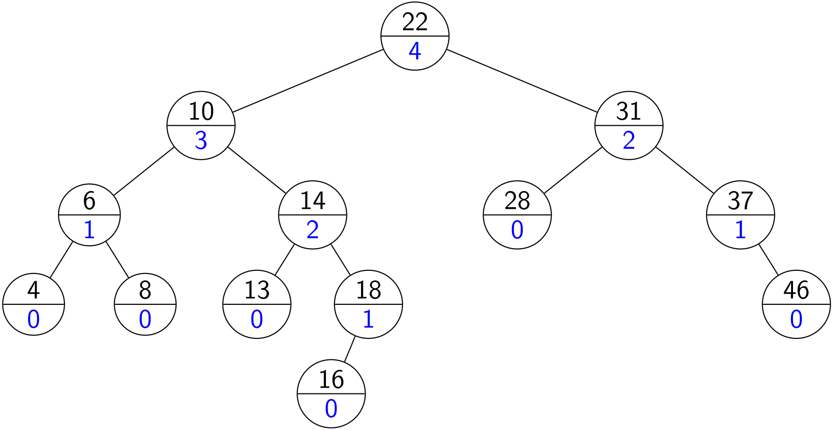
AVL Insertion Example revisited

Example: *AVL::insert*(8)



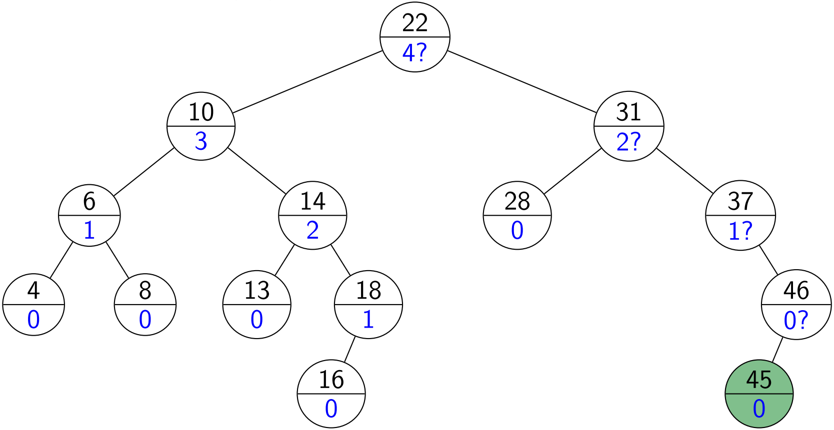
AVL Insertion: Second example

Example: *AVL::insert*(45)



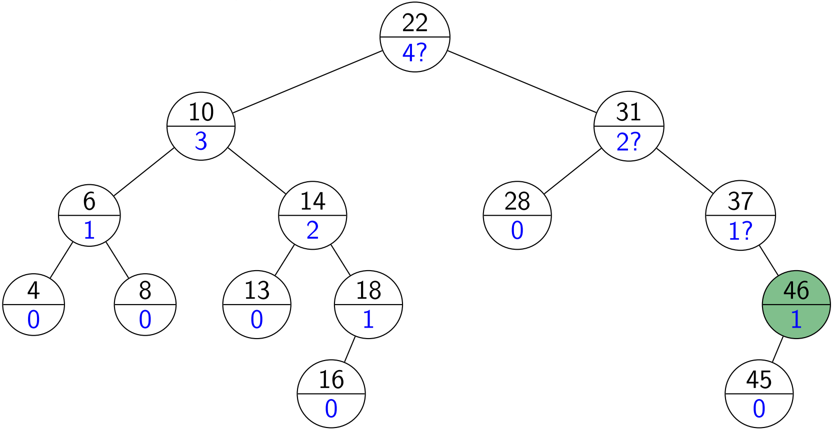
AVL Insertion: Second example

Example: *AVL::insert*(45)



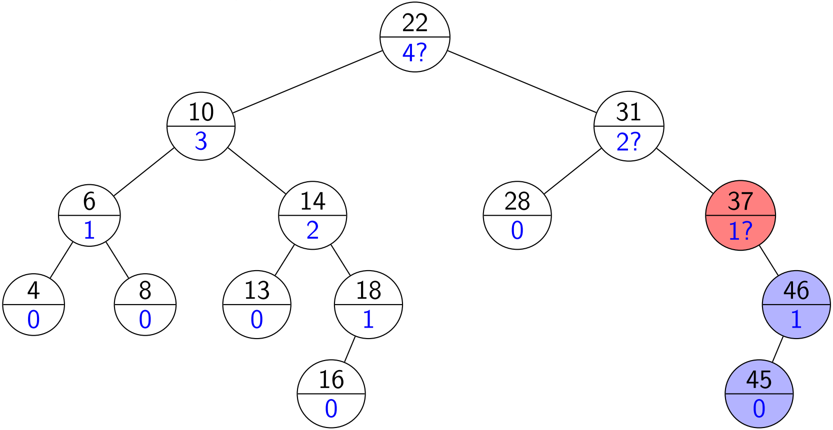
AVL Insertion: Second example

Example: *AVL::insert*(45)



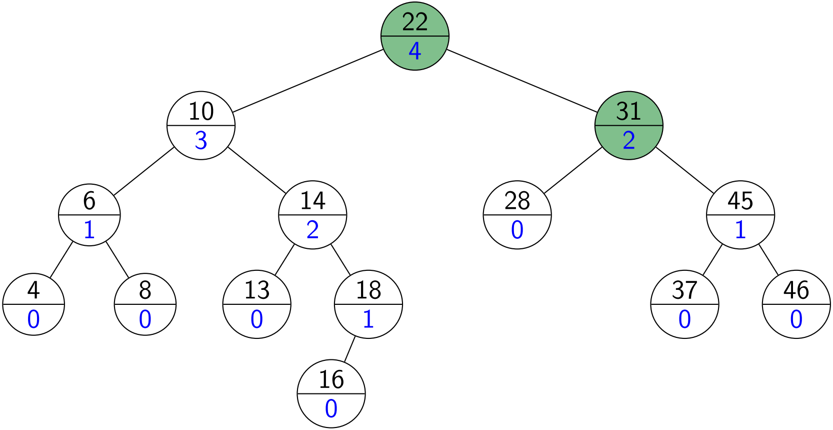
AVL Insertion: Second example

Example: *AVL::insert*(45)



AVL Insertion: Second example

Example: *AVL::insert*(45)



Outline

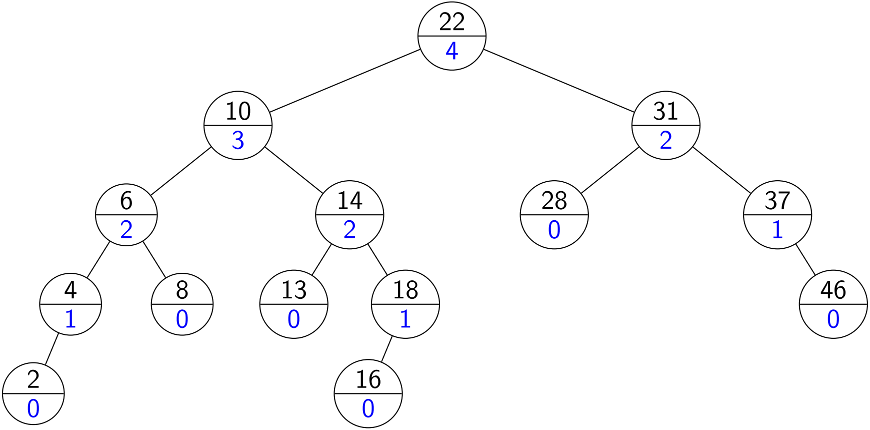
AVL Deletion

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)
$z \leftarrow$ *BST::delete*(k)
`// Assume z is the parent of the BST node that was removed`
while (z is not NULL)
  if $(|z.left.height - z.right.height| > 1)$ **then**
    Let c be taller child of z
    Let g be taller child of c (break ties to avoid double rotation)
    $z \leftarrow$ *restructure*(g, c, z)
  `// Always continue up the path`
  set-height-from-subtrees(z)
  $z \leftarrow z.parent$

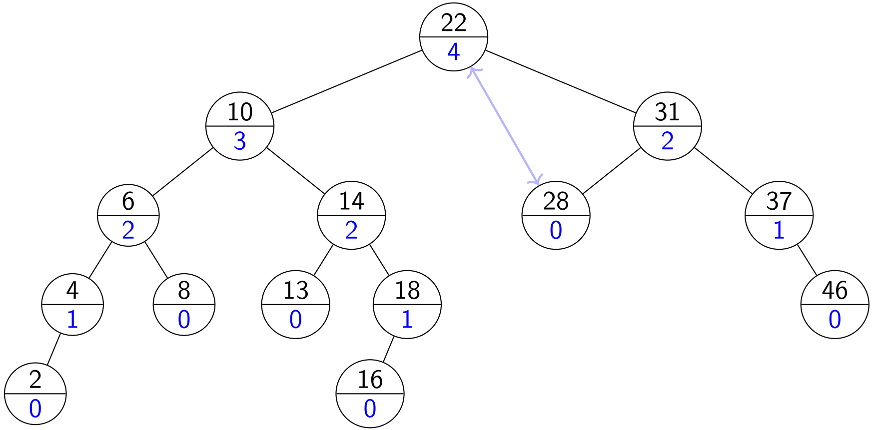
AVL Deletion Example

Example: *AVL::delete*(22)



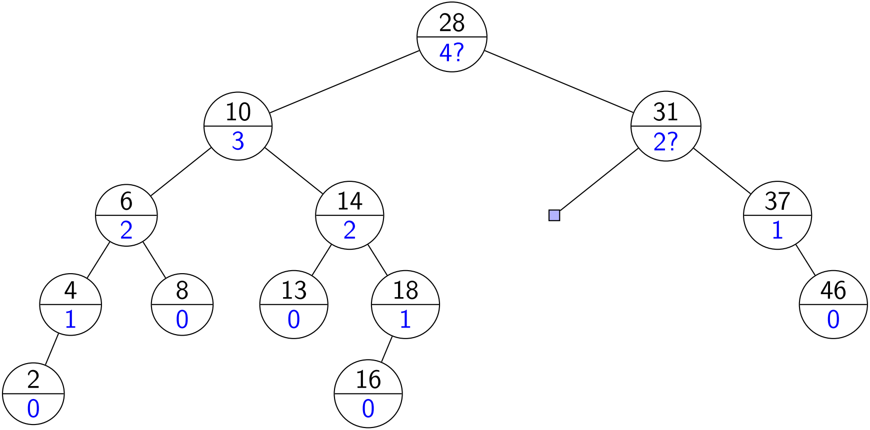
AVL Deletion Example

Example: *AVL::delete*(22)



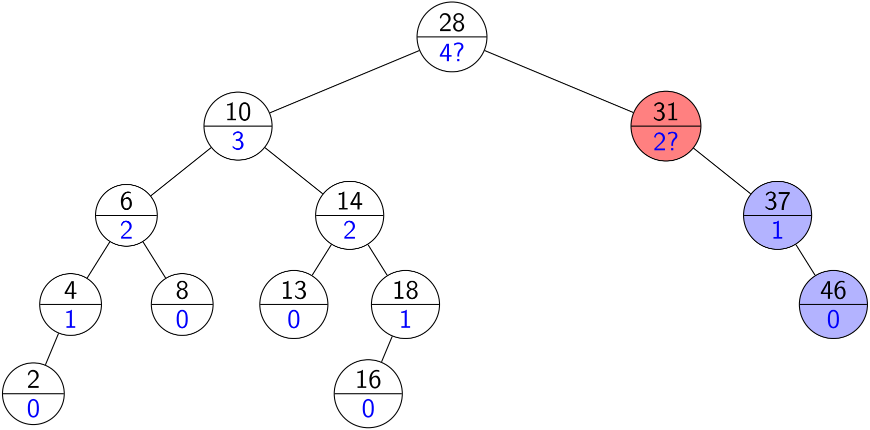
AVL Deletion Example

Example: *AVL::delete*(22)



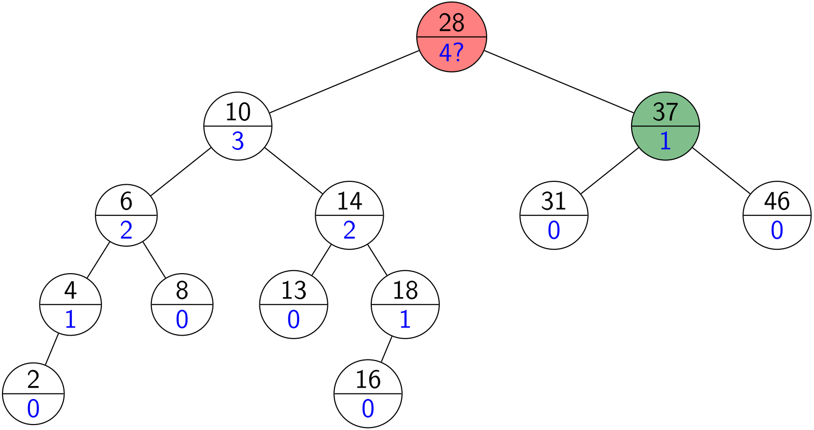
AVL Deletion Example

Example: *AVL::delete*(22)



AVL Deletion Example

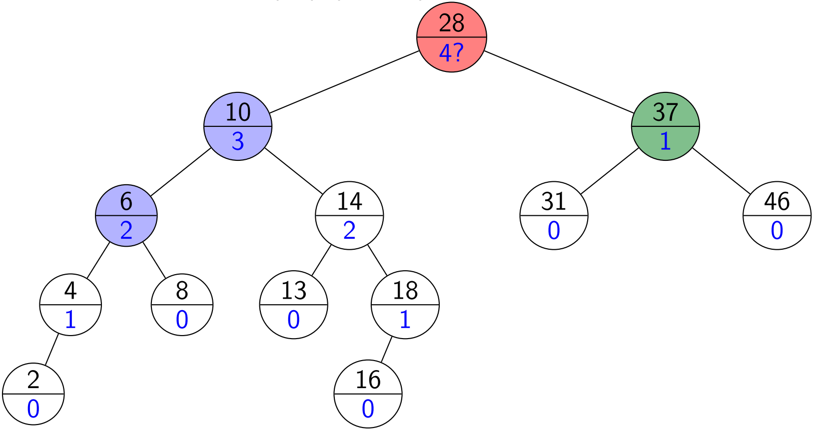
Example: *AVL::delete*(22)



A single *restructure* is not enough to restore all balances.

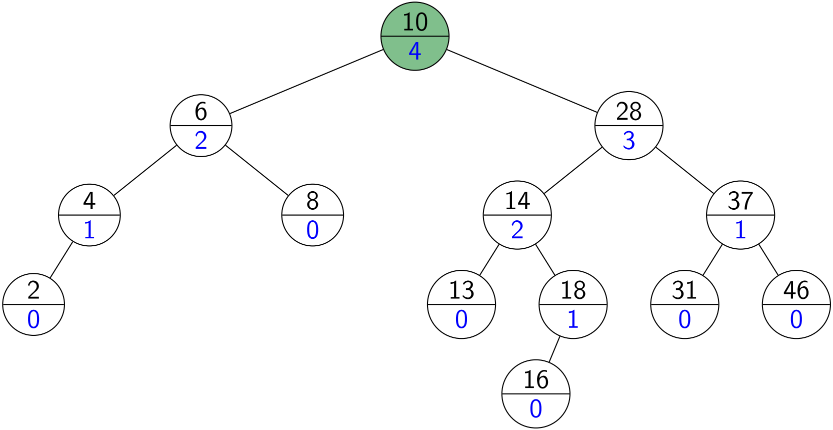
AVL Deletion Example

Example: *AVL::delete*(22) (cont'd)



AVL Deletion Example

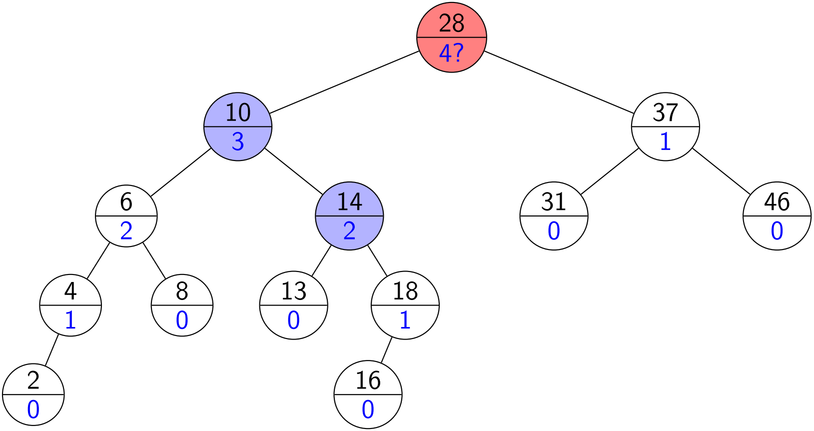
Example: *AVL::delete*(22) (cont'd)



AVL Deletion Example

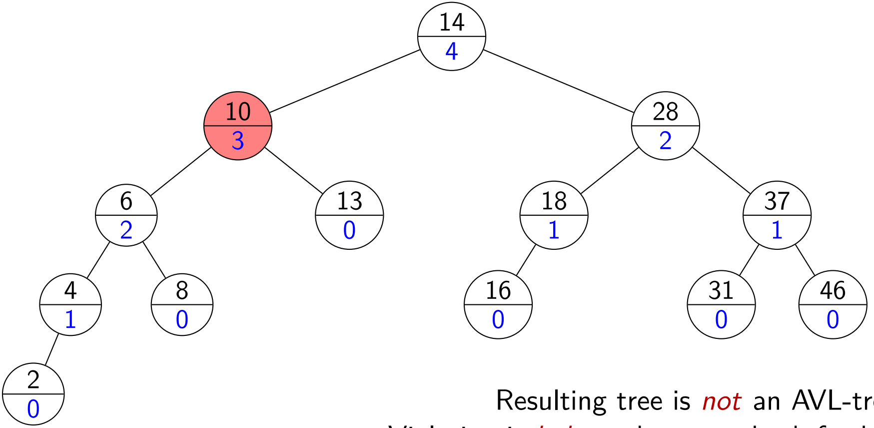
Important: Ties *must* be broken to avoid double rotation.

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



AVL Deletion Example

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



Resulting tree is *not* an AVL-tree.
Violation is *below* where we check further.

AVL Tree Summary

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

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

- total cost $\Theta(height)$
- *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 $\Theta(height) = \Theta(\log n)$.

- In practice, the constant is quite large.
- Other realizations of ADT Dictionary are better in practice ($\rightarrow$ later)