Durham University

Algorithms & Data Structures

Part – 3: Topic 3

Anish Jindal
anish.jindal@durham.ac.uk

Durham University

Balanced BSTs and AVL Trees

Balanced BST

OK if BST is **balanced**, then h = O(log n)

Bad if BST is **degenerated**, then h = Ω(n)

(easy to come up with inputs)

How about being somewhat clever, i.e., taking care that BST does **not** degenerate when inserting or deleting?

Preferably without excessive overhead work?

Many (many many. . .) approaches, we're going to see **AVL trees** and (very briefly) **red-black trees**

Basic re-balancing operations: **rotations**

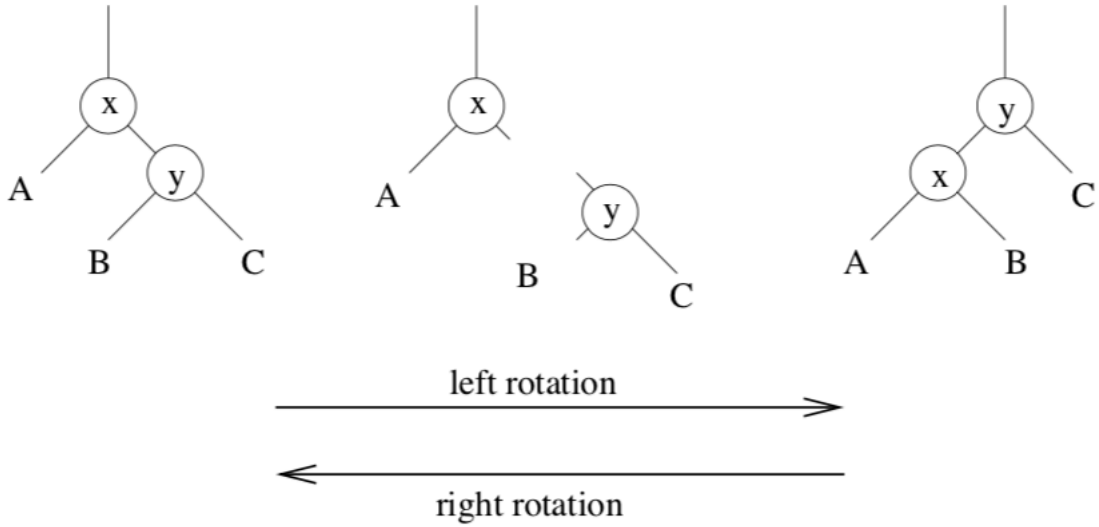
Local operation, preserves BST property

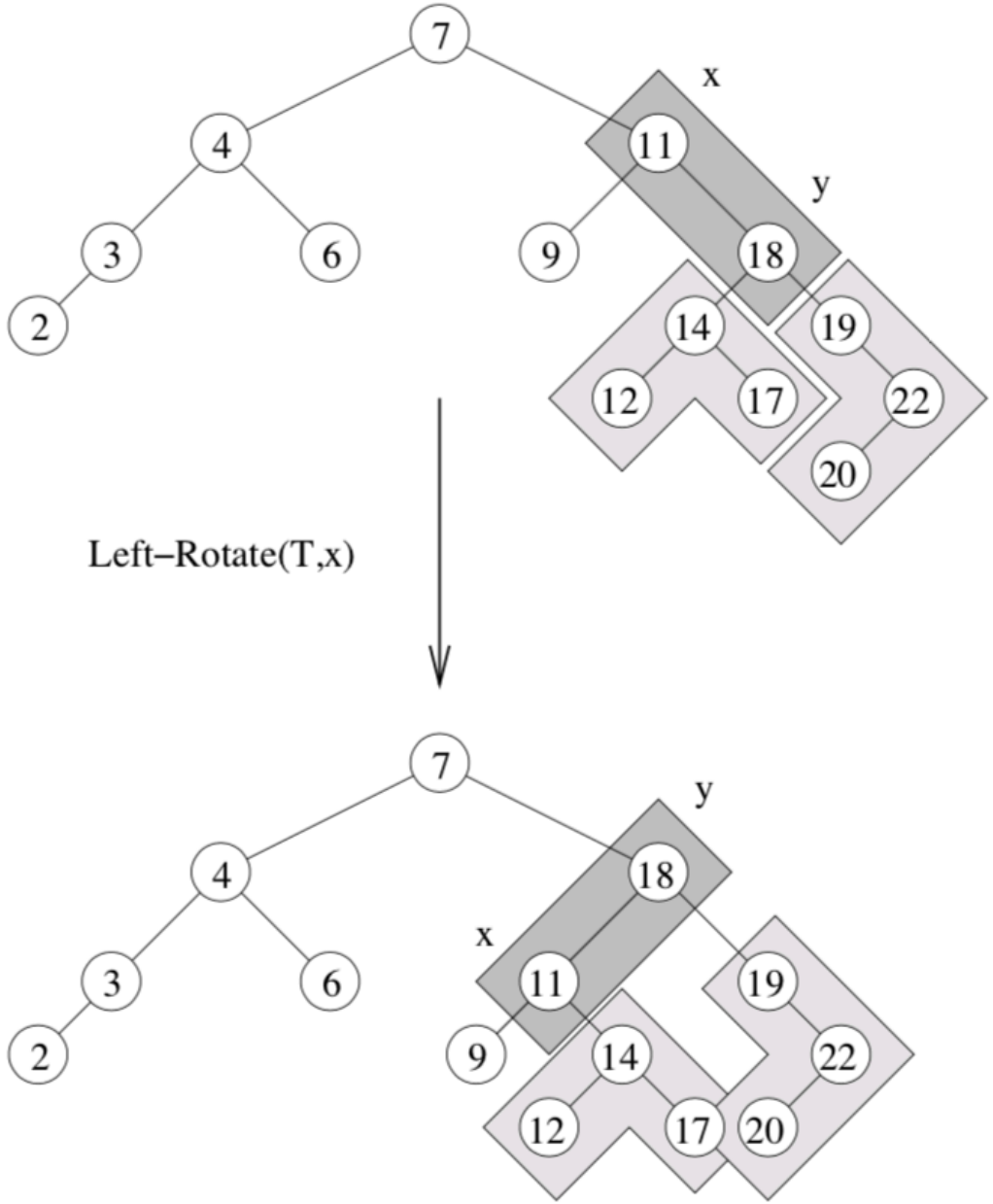
left and **right** rotations

Assumptions:

left-rotation on x: right child not NULL

right-rotation on y: left child not NULL




7
4
11
x
y
3
6
9
18
2
14
19
12
17
22
20
Left–Rotate(T,x)
7
y
4
18
x
3
6
11
19
2
9
14
22
12
17
20

The operation is local: it modifies a constant number of parent-child links.

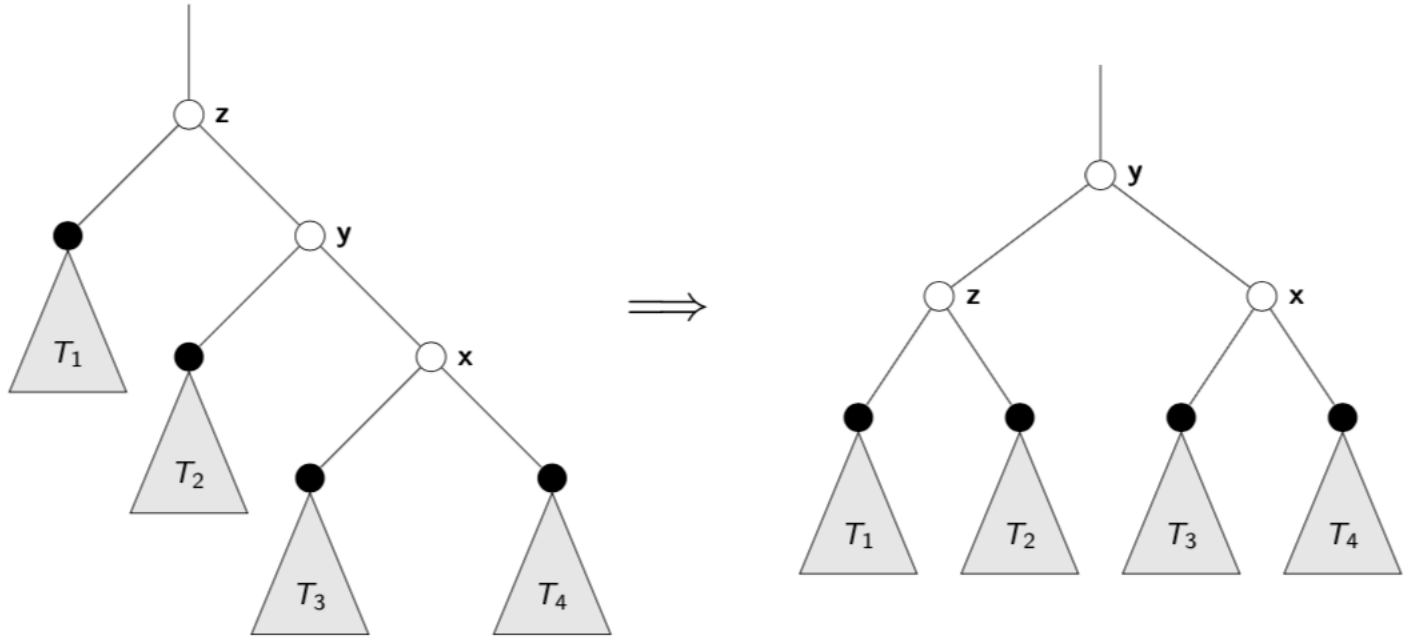
Hence each simple rotation takes O(1) time.

Simple rotations can be combined to provide re-balancing.

We call such a compound operation applied to a triple parent-child-grandchild **trinode restructuring**

The operation depends on whether child and grandchild are on the same side (i.e. both left or both right) or not.

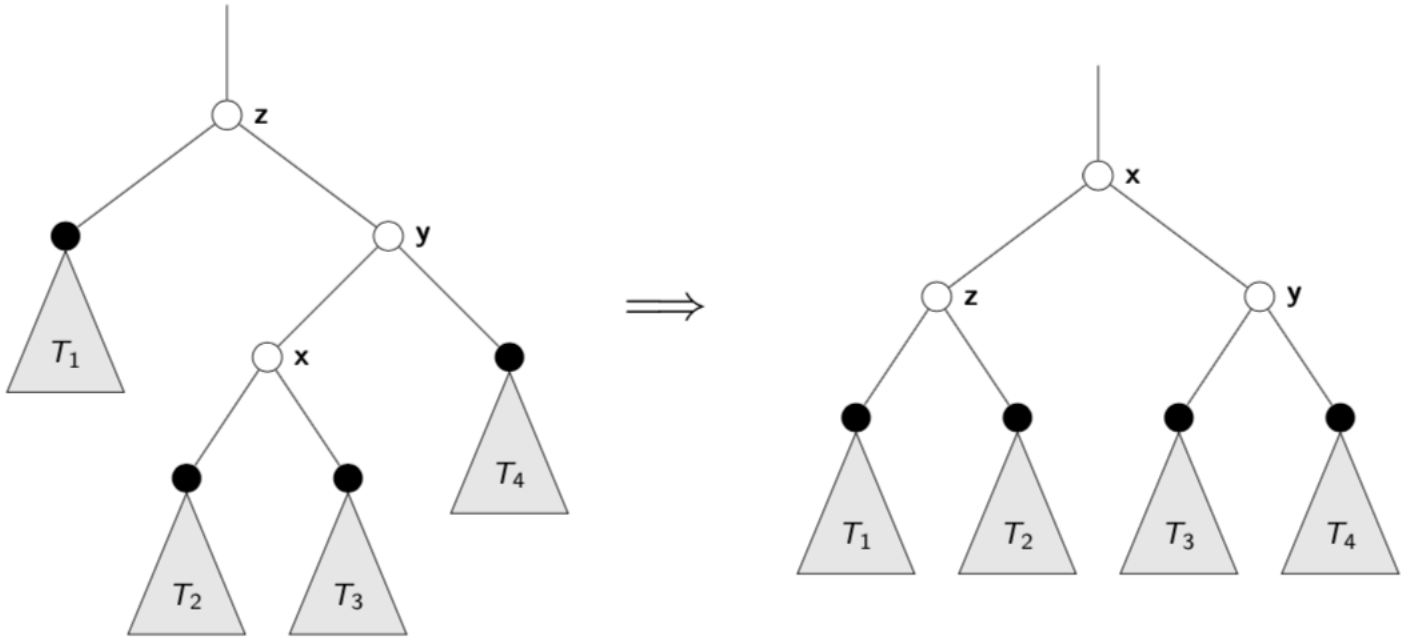
Trinode restructuring: single rotation



Single rotation: child and grandchild are on the same side

Here, both are right - do Left-Rotate(T,z)

Trinode restructuring: double rotation



Double rotation: child and grandchild are not on the same side Here, right/left
- do Right-Rotate(T,y), then Left-Rotate(T,z)

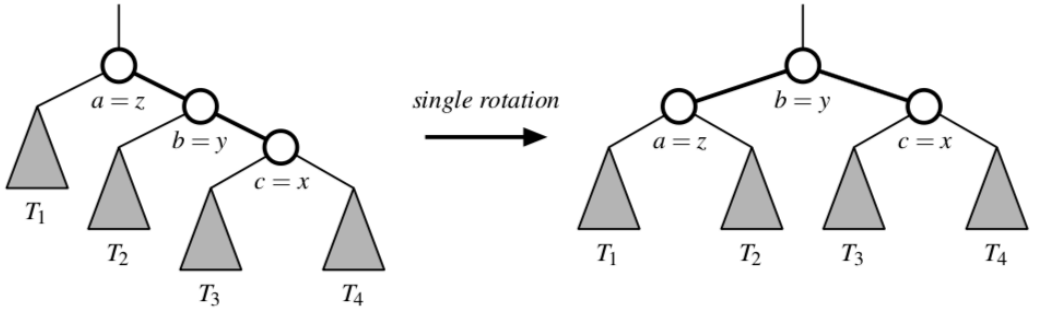
The median of x,y,z always ends up as the root of this subtree

The operation is still local - modifies a constant number of links

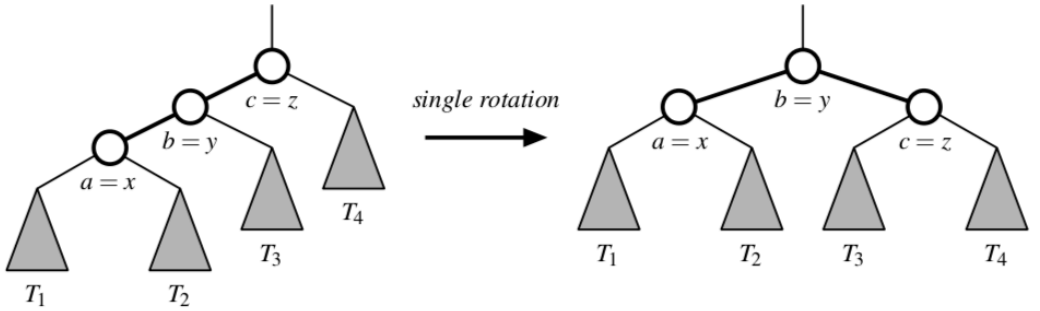
Hence, it still takes O(1) time.

The implementation goes through 4 cases:

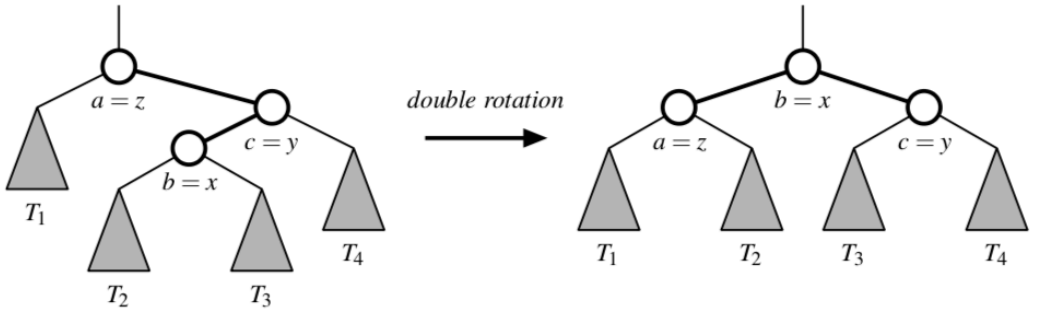
(a) left/left, (b) right/right, (c) left/right, (d) right/left



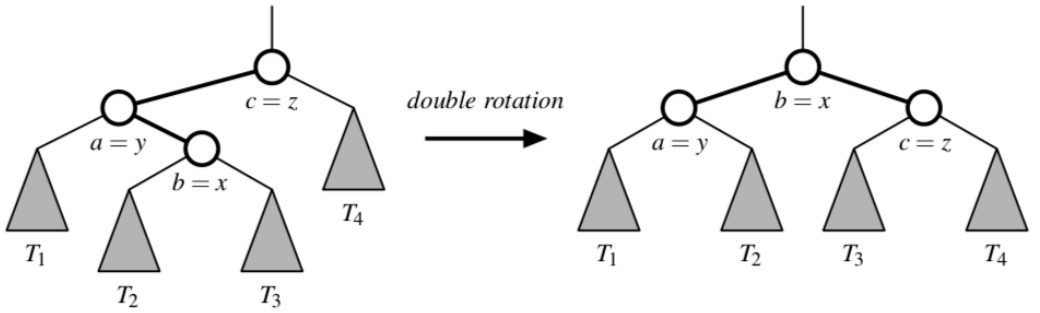
(a)



(b)



(c)



(d)

Durham University

AVL Trees

AVL Trees

Named after the inventors: Georgy **A**delson-**V**elsky and Evgenii **L**andis

An **AVL tree** is a self-balancing BST with the following additional property:

Height-balance property

For each node v, the heights of v's children differ by at most 1.

Will use the definition of height of a node:

- height of Null is 0 and height of a proper leaf is 1
- height of a parent = max height of a child +1.

Note: height a non-empty tree = height of its root -1.

Test your knowledge!!

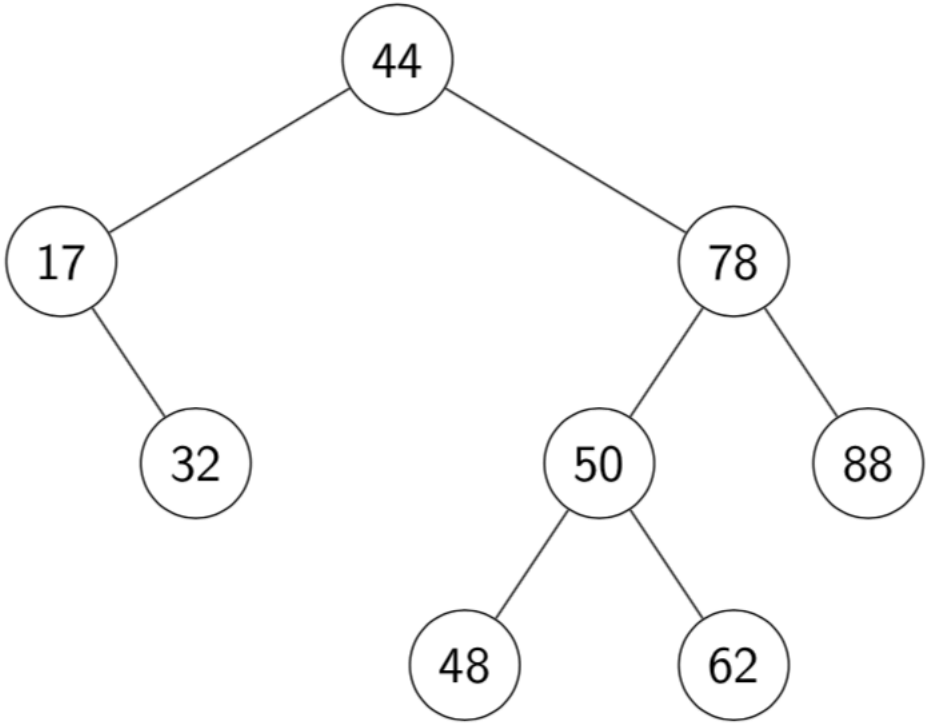
The pre-order traversal of a binary search tree gives the following output:

44, 17, 32, 78, 50, 48, 62, 88

Construct this tree.

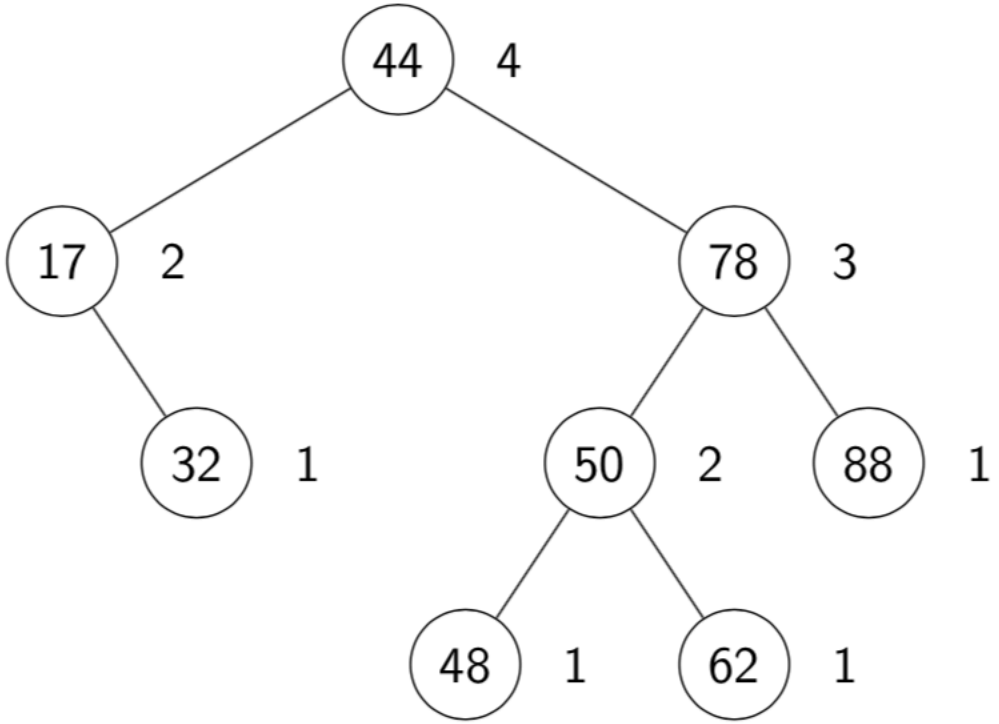
Example

Is this BST an AVL tree?



Example

Yes, the height-balance property is satisfied.



Key property

Fact: The height of an AVL tree storing n keys is O(log n).

Trick – Find lower bound on minimum number of nodes of height h.

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

We easily see that n(1) = 1 and n(2) = 2

For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.

That is, n(h) = 1 + n(h-1) + n(h-2)

Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

$n(h) > 2^i n(h-2i)$

Taking i=(h/2-1) so that (h – 2i is 1 or 2), on solving, we get: $n(h) > 2^{h/2-1}$

Taking logarithms: h < 2log n(h) +2 (i.e., at most 2log n(h) +2)

Since log(n) ≥ log (n(h), Thus the height of an AVL tree is O(log n)

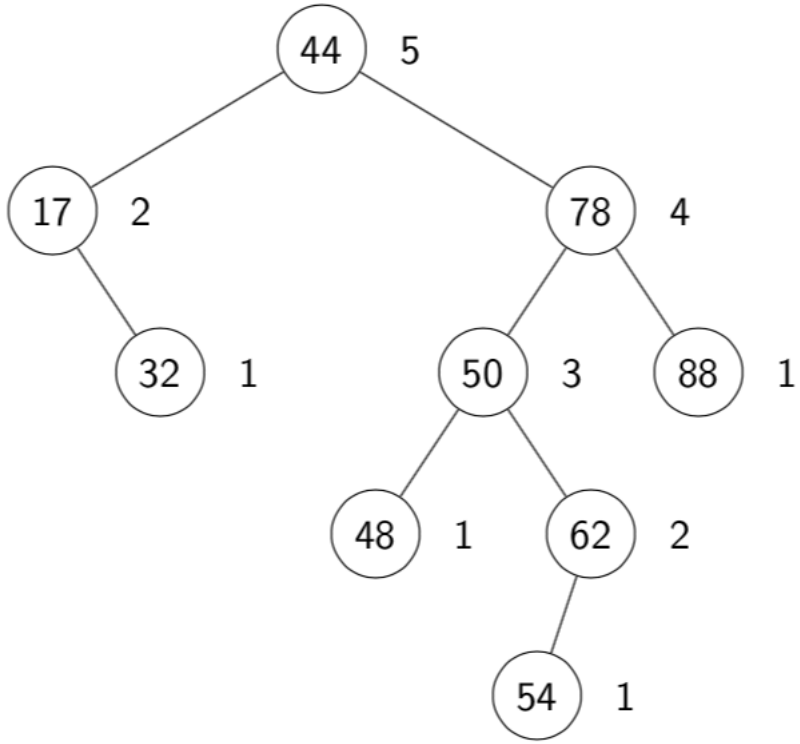
The standard BST operations in AVL trees take O(logn) time.

Are we done?

- No, insertion and deletion can potentially violate the height-balance property.
    - What happens if we insert 1, 2, 3 into an empty AVL tree?
- So need a way to restore it, preferably in O(logn) time.
- Trinode restructuring might be useful here.

Insertion

Take the AVL tree from an earlier example and insert 54:



Height-balance is now violated – how to restore it?

AVL rotations

There are 4 cases:

Let the node that needs rebalancing be α.

(require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

(require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

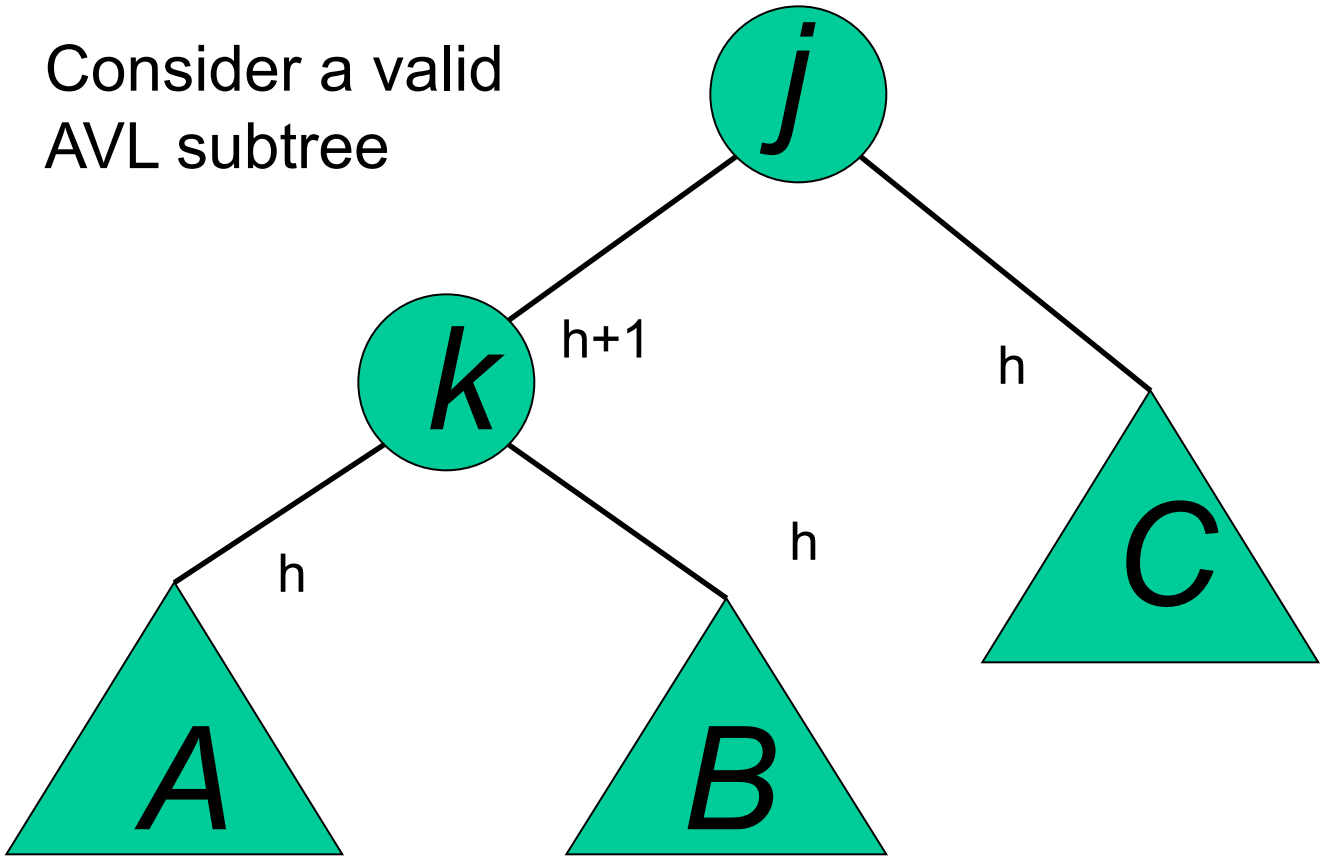
Rotations:

- Right
- Left

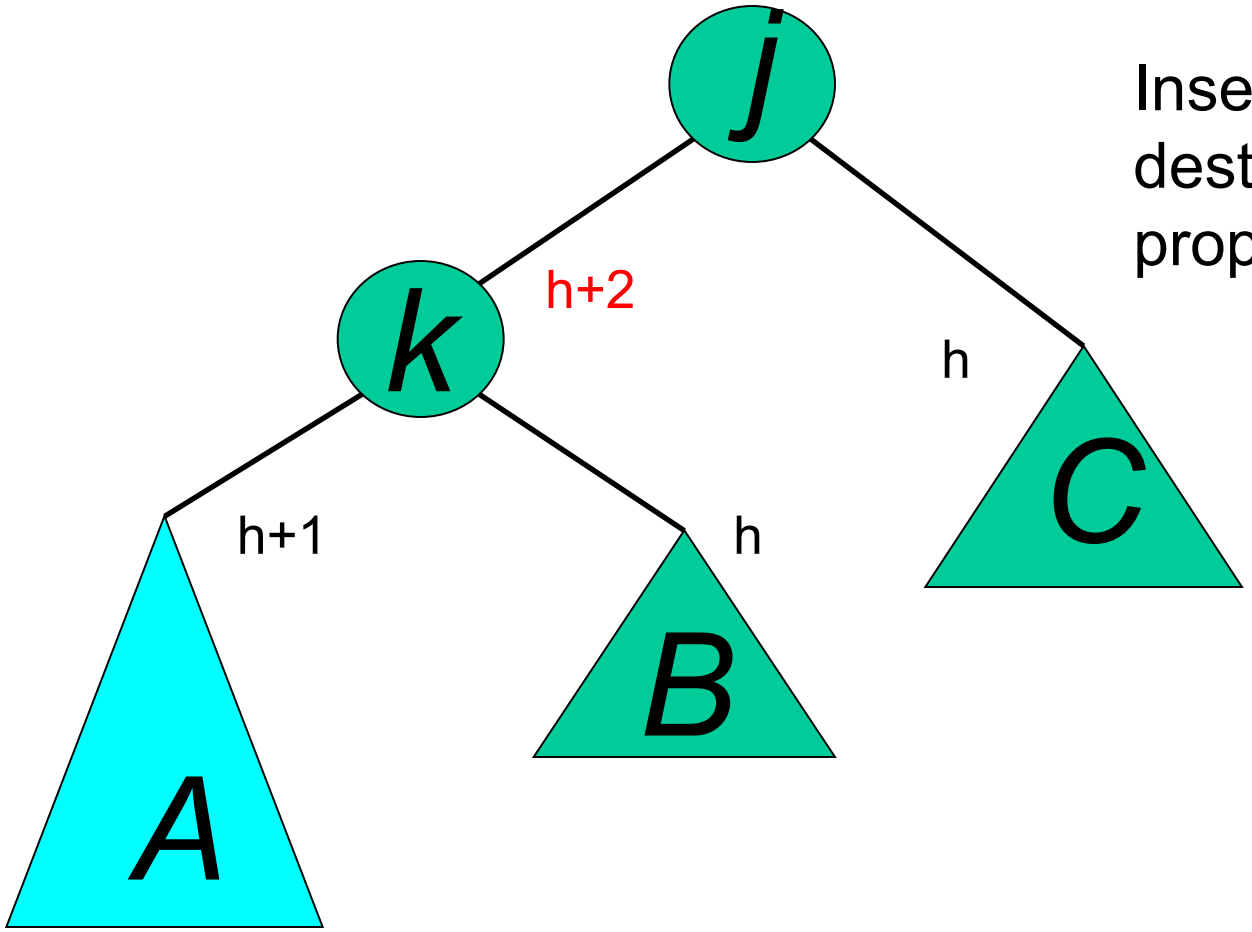
- Left-Right
- Right-Left

Case: 1

Consider a valid AVL subtree

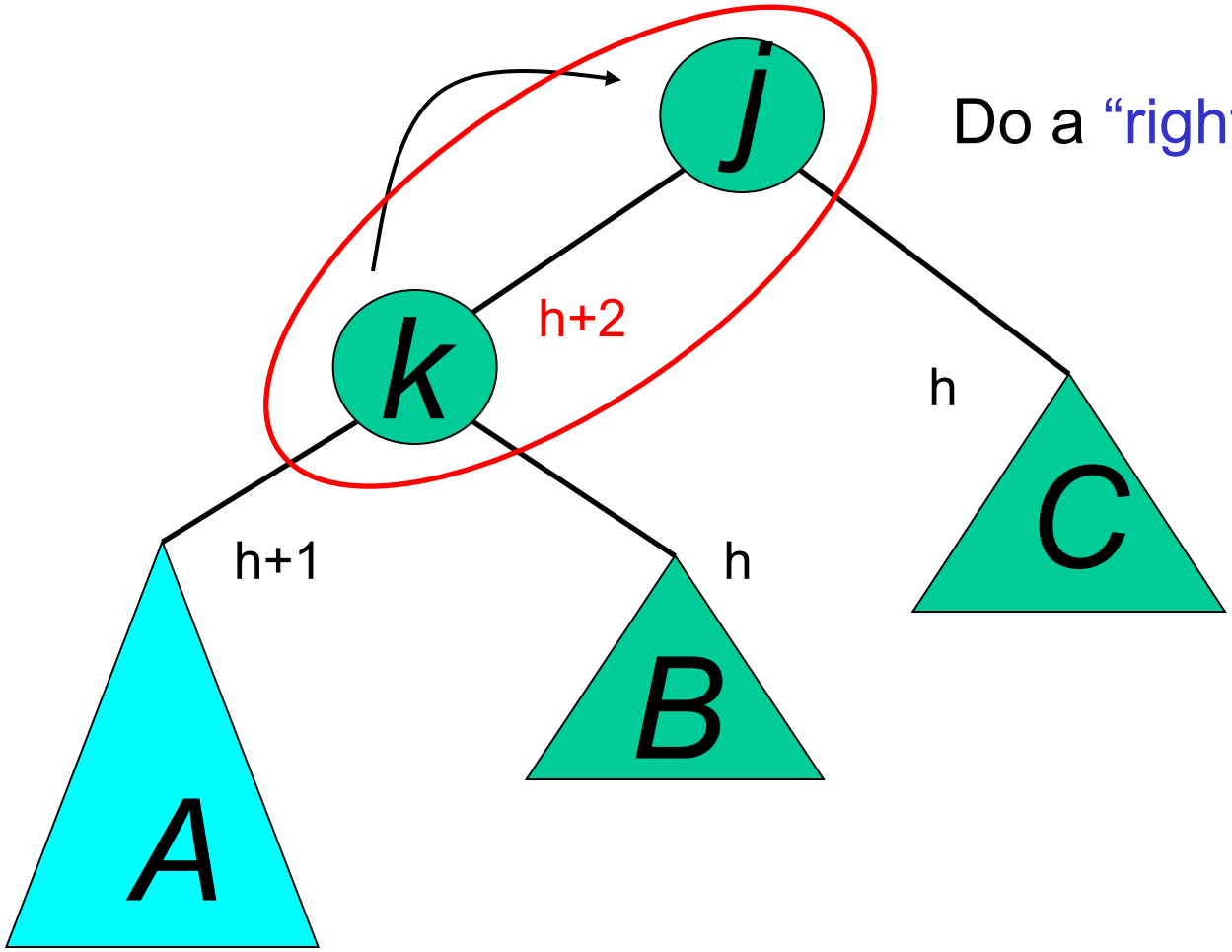


Case: 1



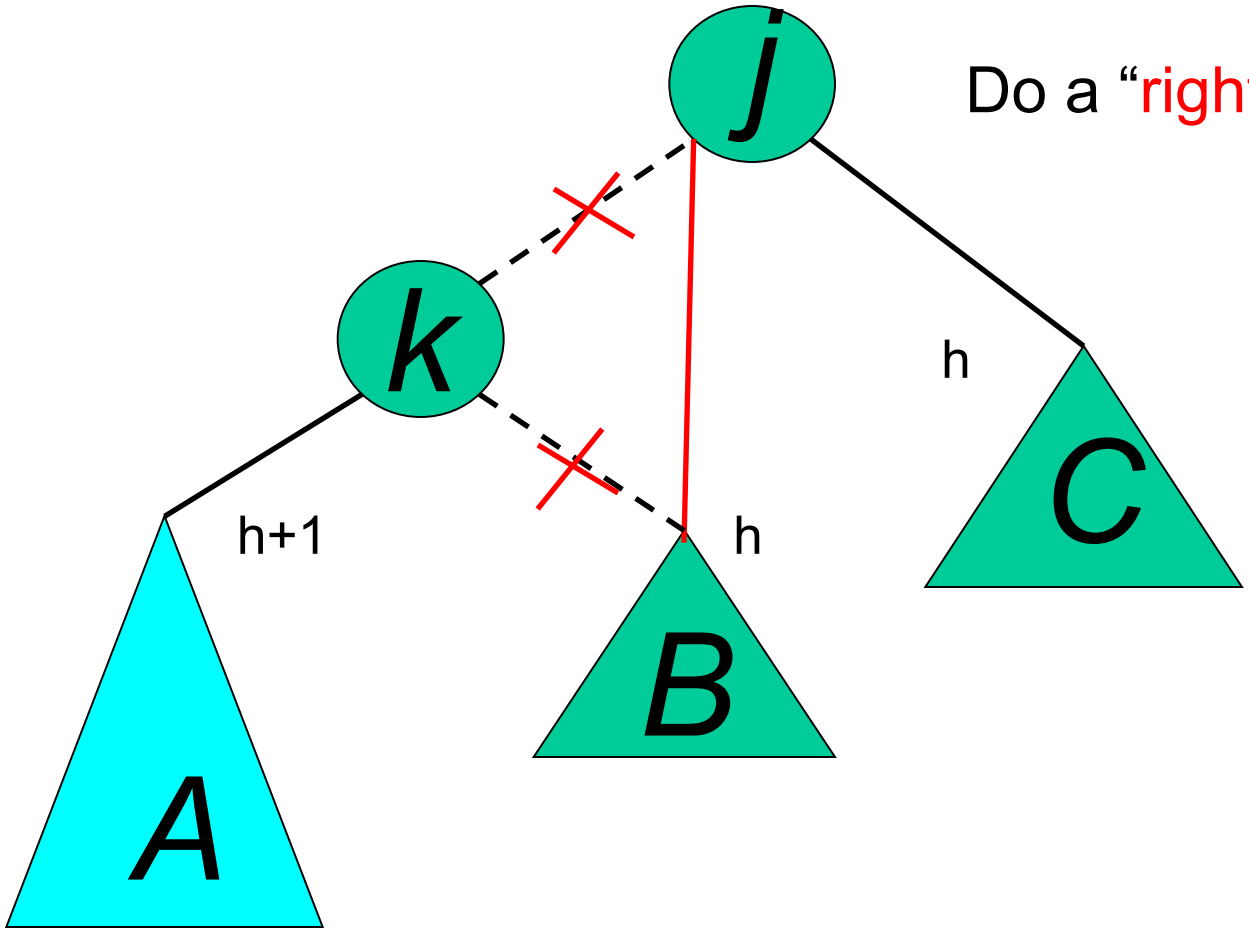
Inserting into X destroys the AVL property at node j

Case: 1



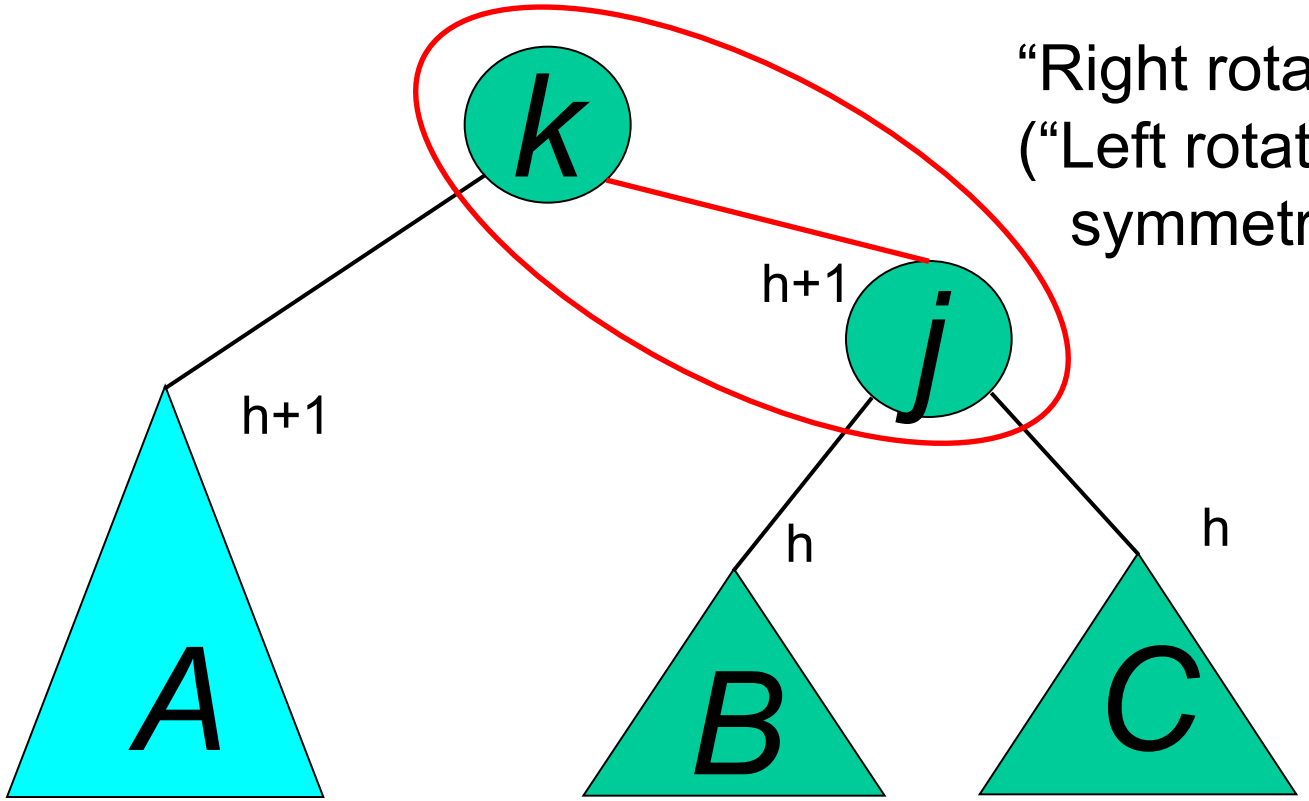
Do a “right rotation”

Single right rotation



Do a “right rotation”

Right rotation completion

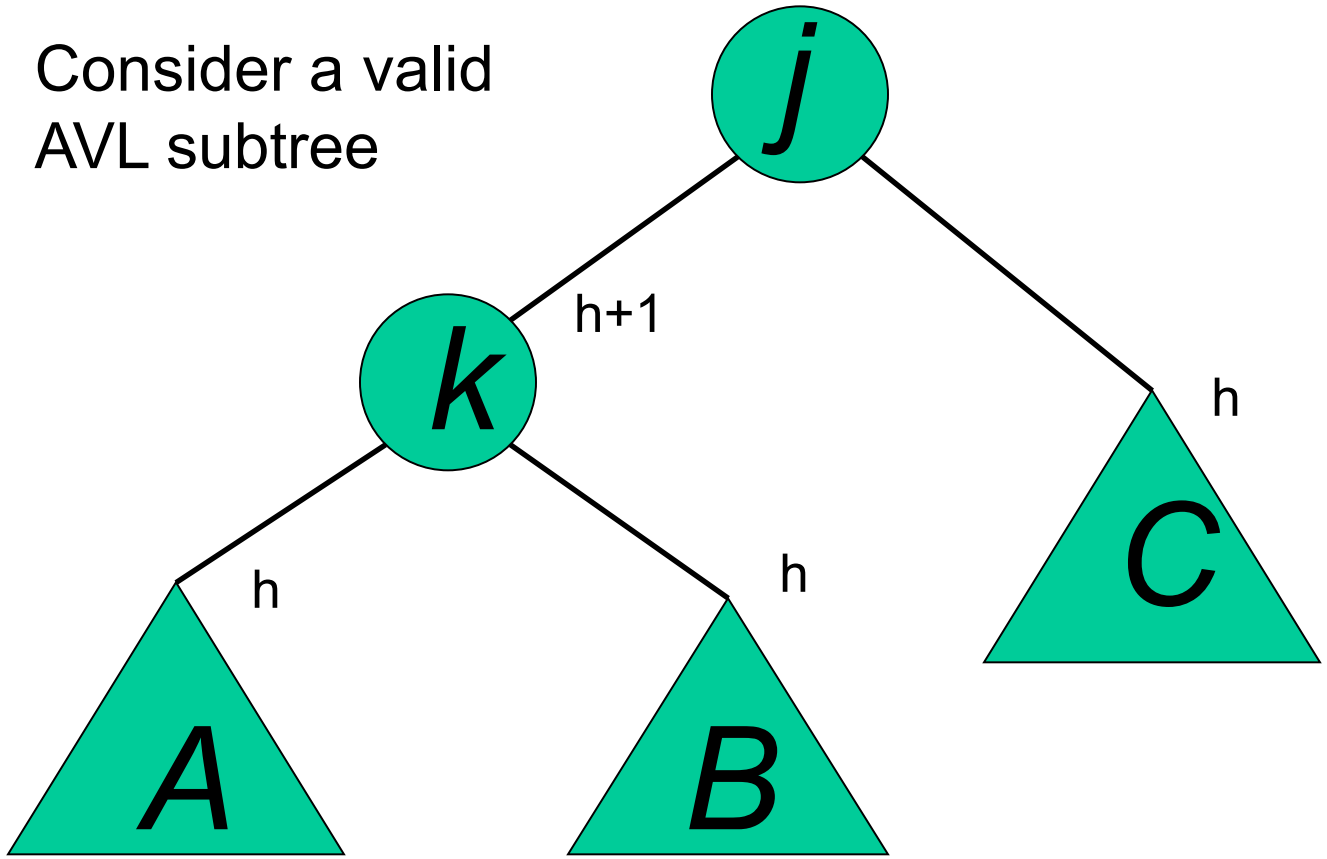


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

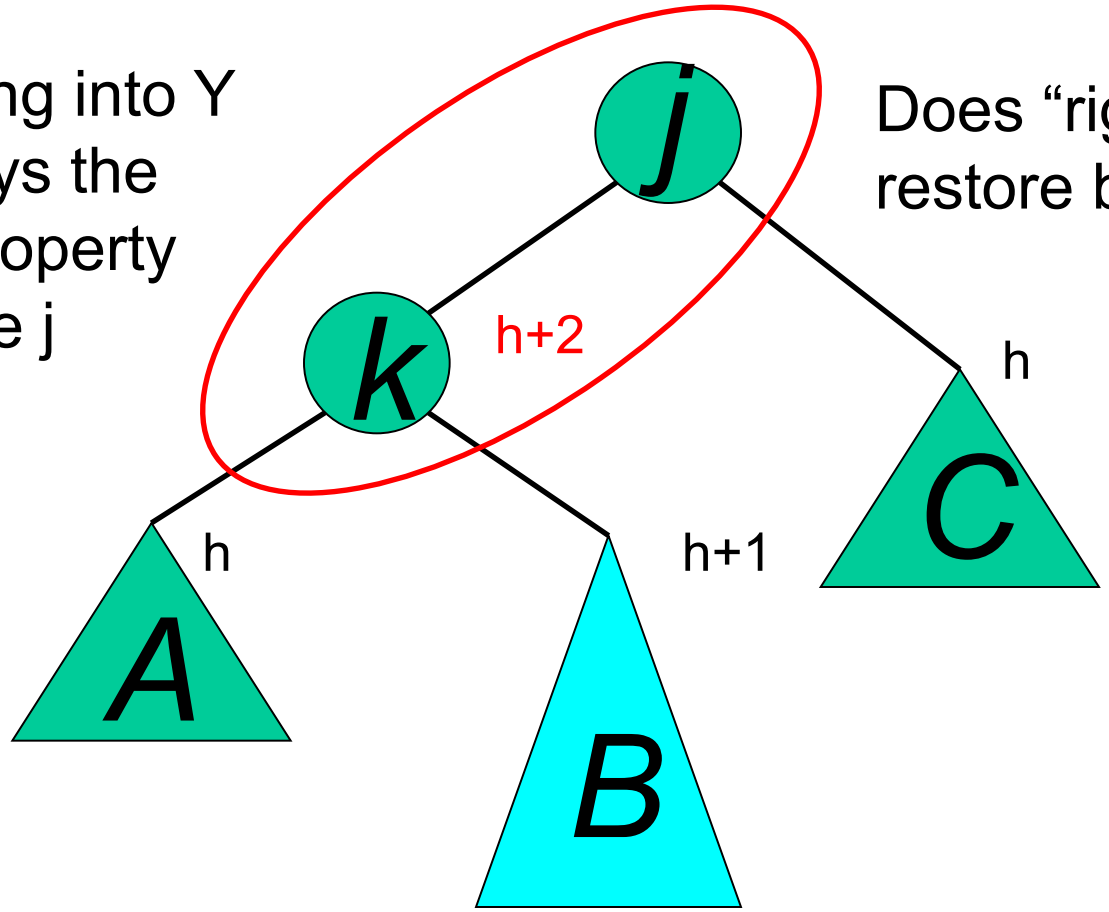
Case: 3

Consider a valid AVL subtree



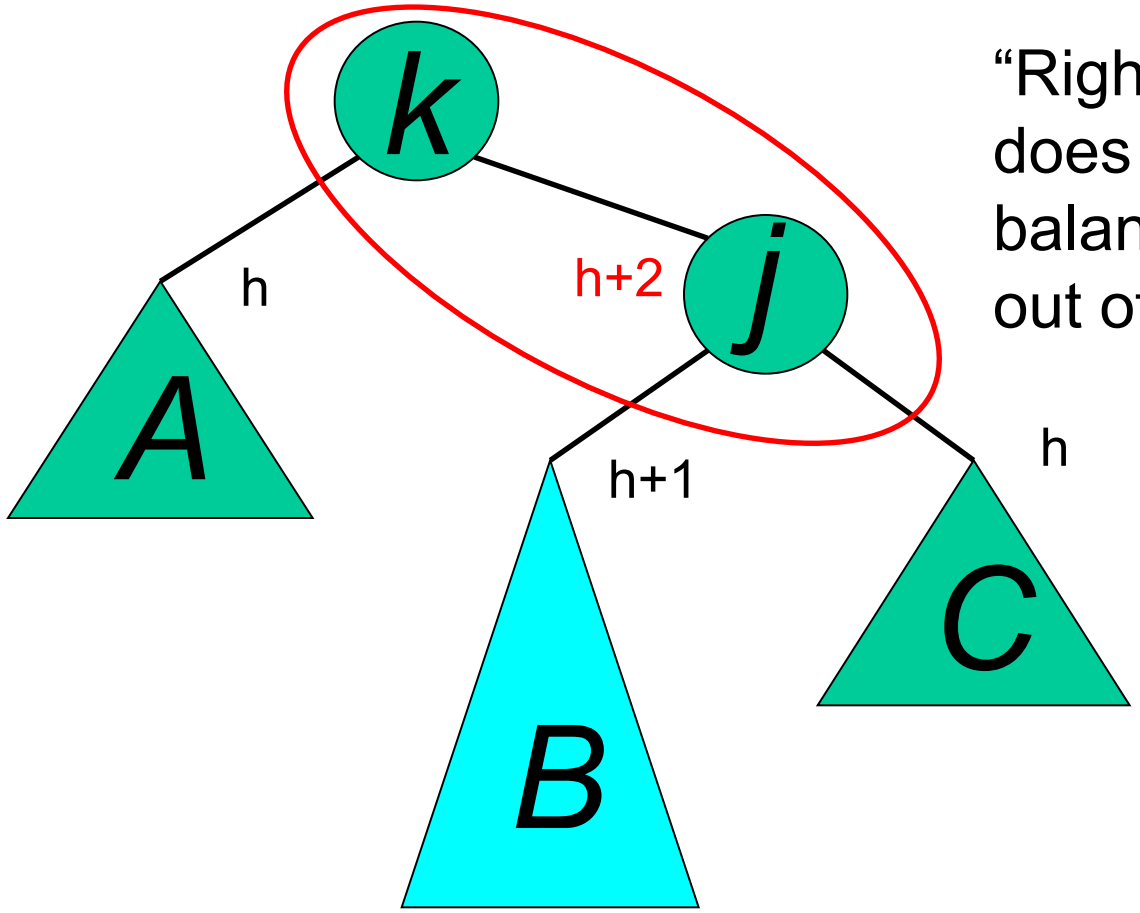
Case: 3

Inserting into Y destroys the AVL property at node j



Does “right rotation” restore balance?

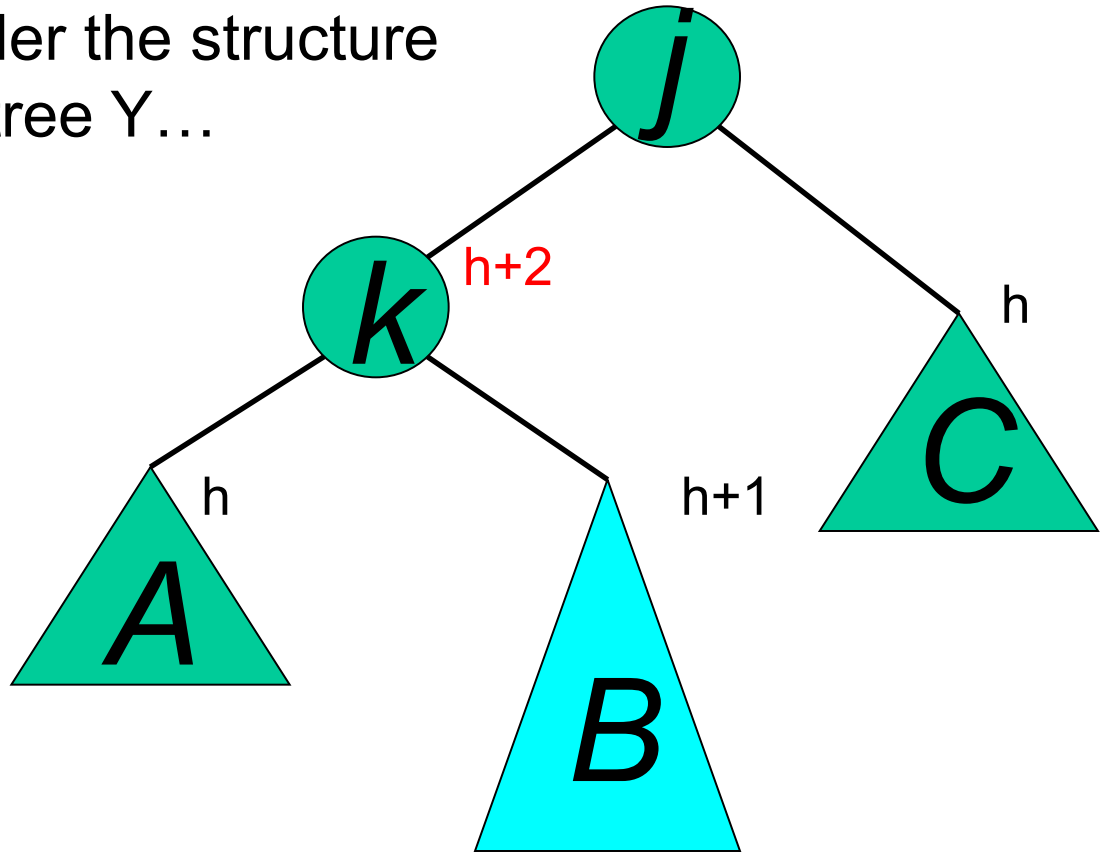
Case: 3



"Right rotation" does not restore balance… now k is out of balance

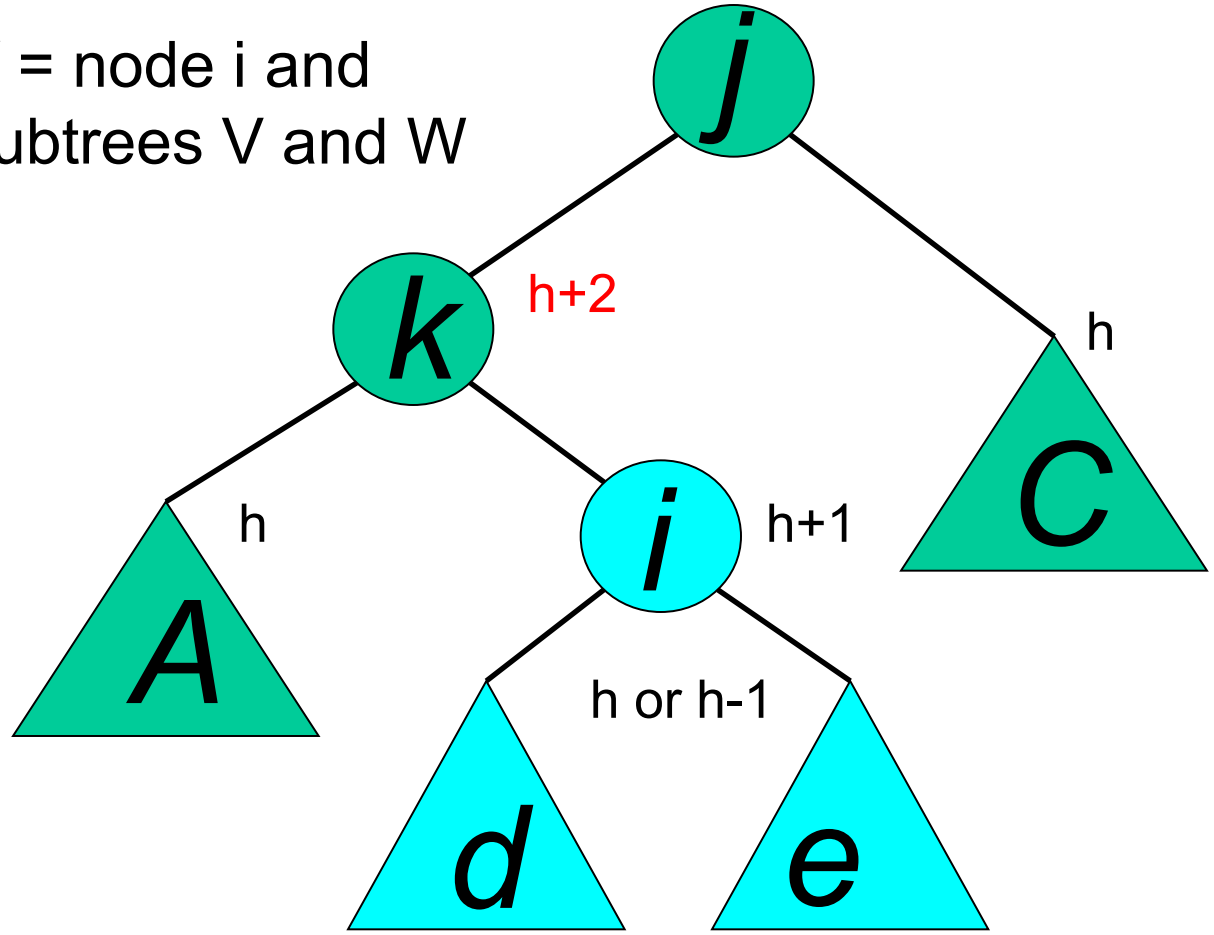
Case: 3

Consider the structure of subtree Y…

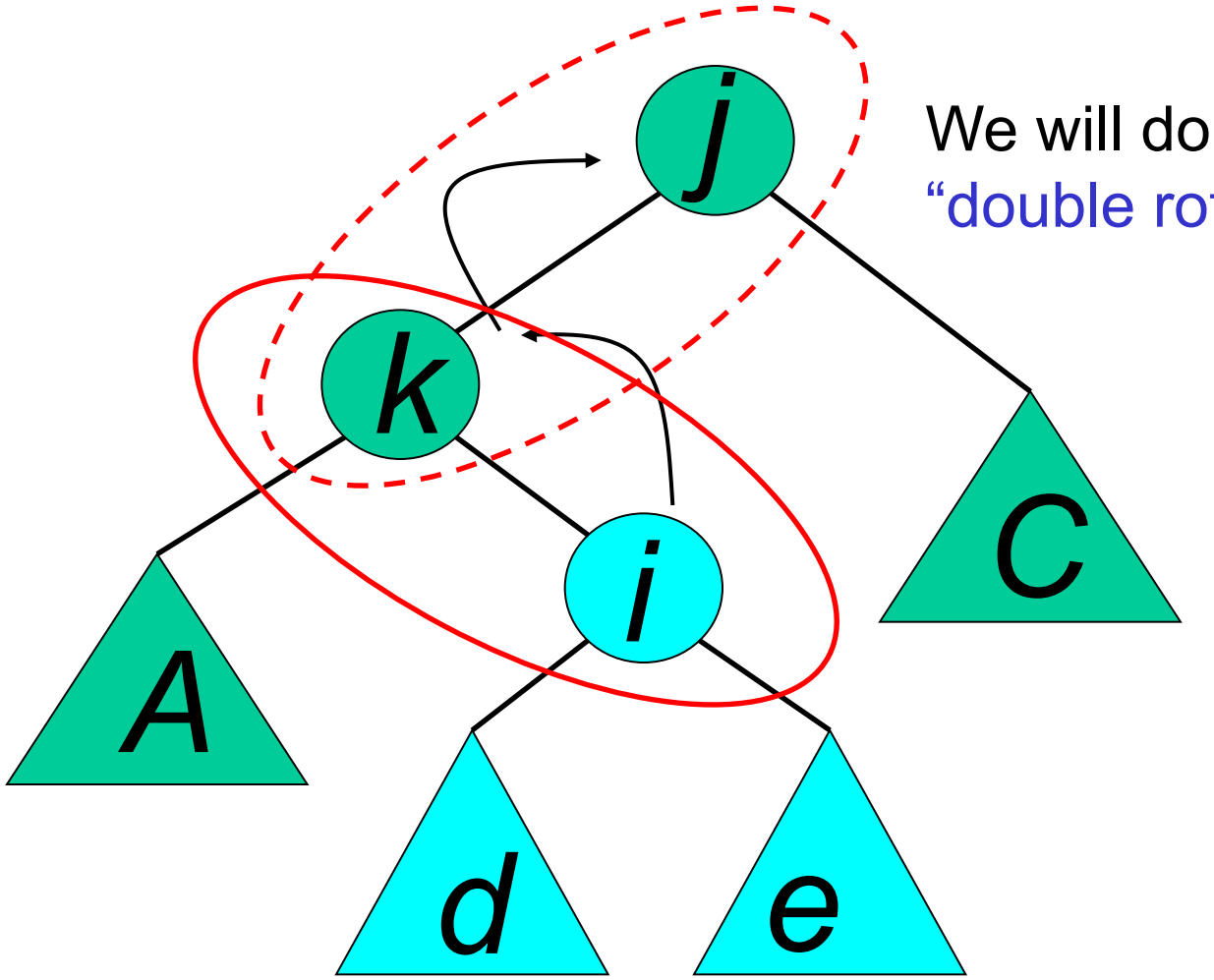


Case: 3

Y = node i and
subtrees V and W

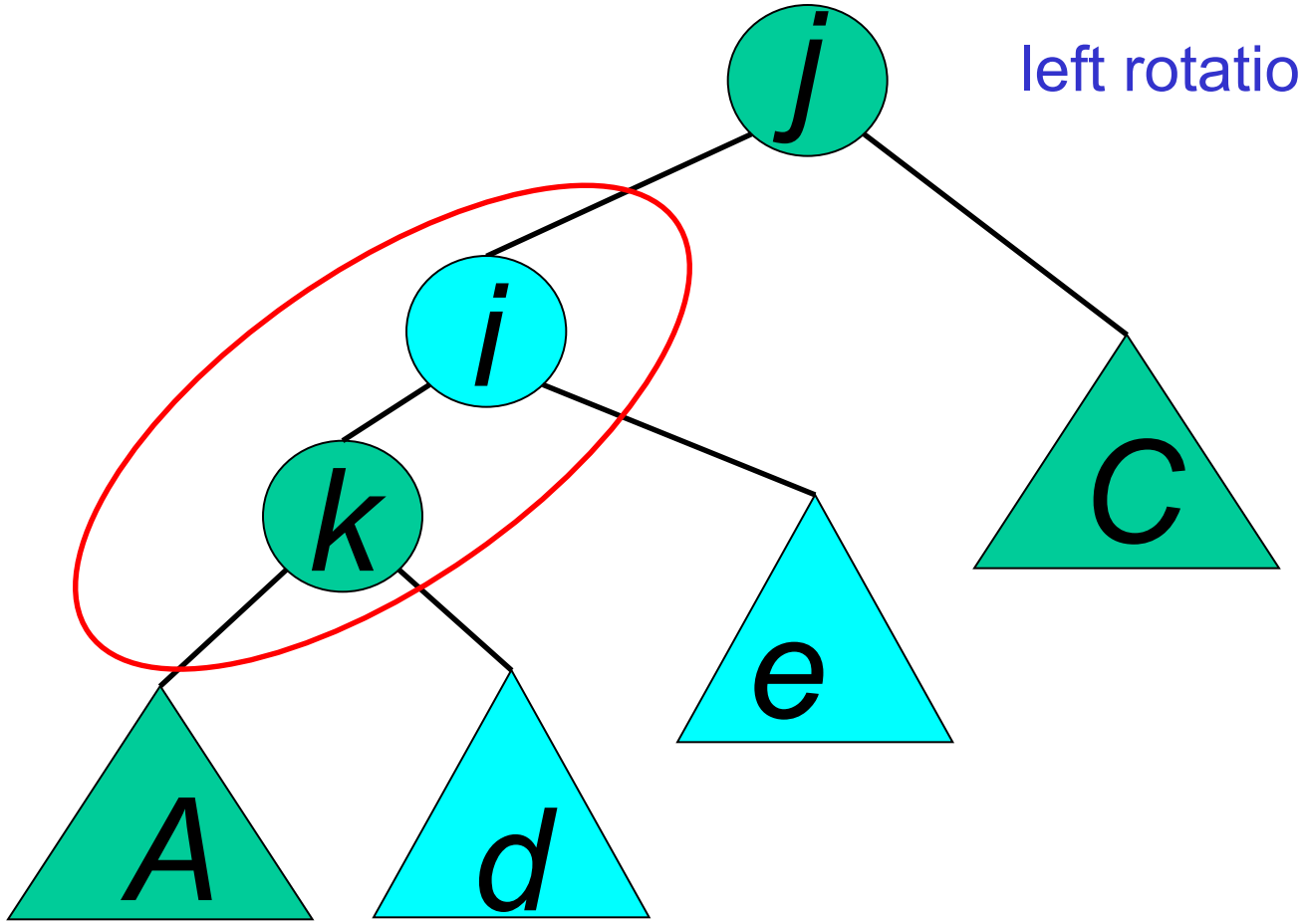


Case: 3



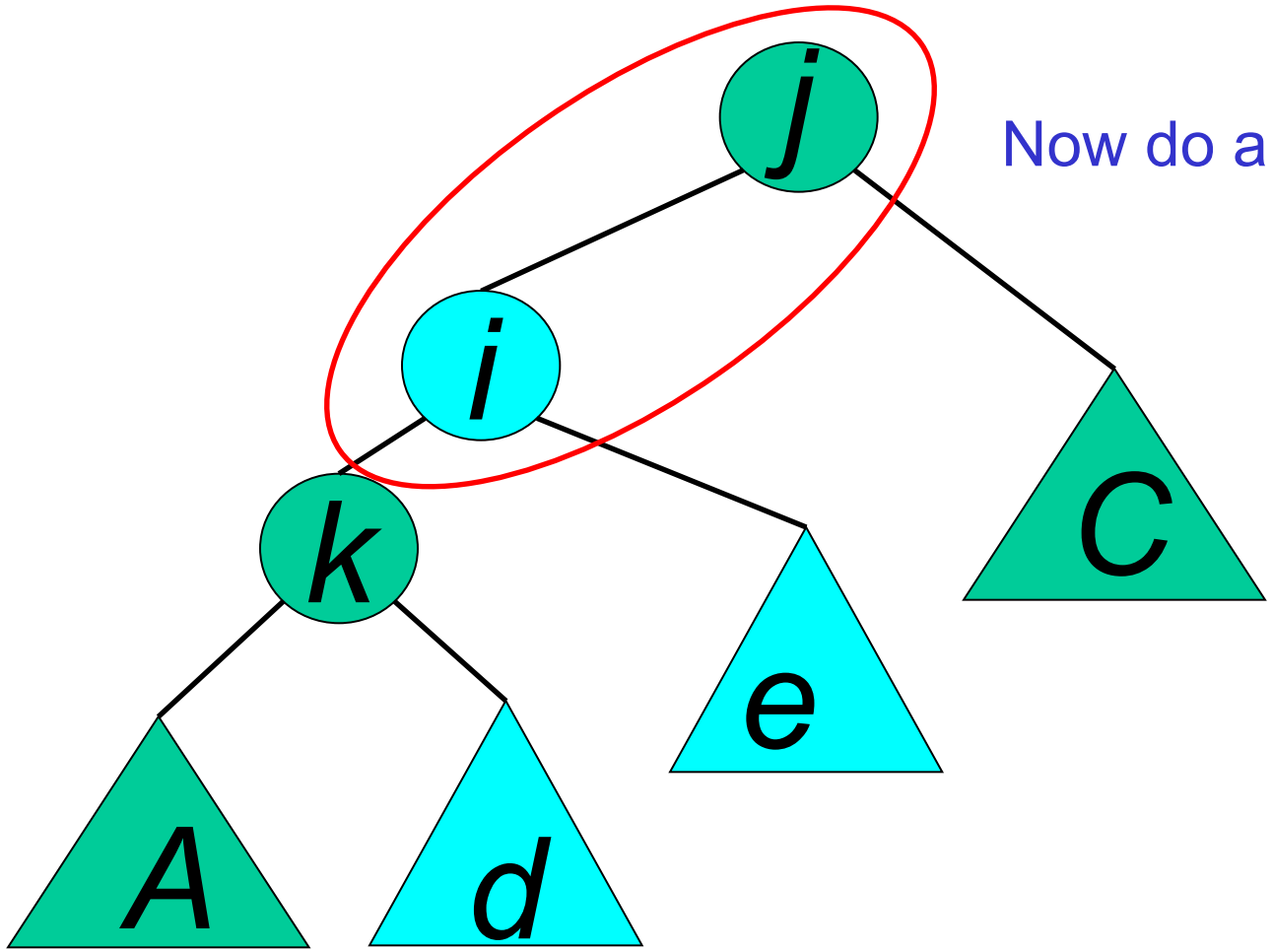
We will do a left-right "double rotation" . . .

Case 3: First Rotation



left rotation complete

Case 3: Second Rotation

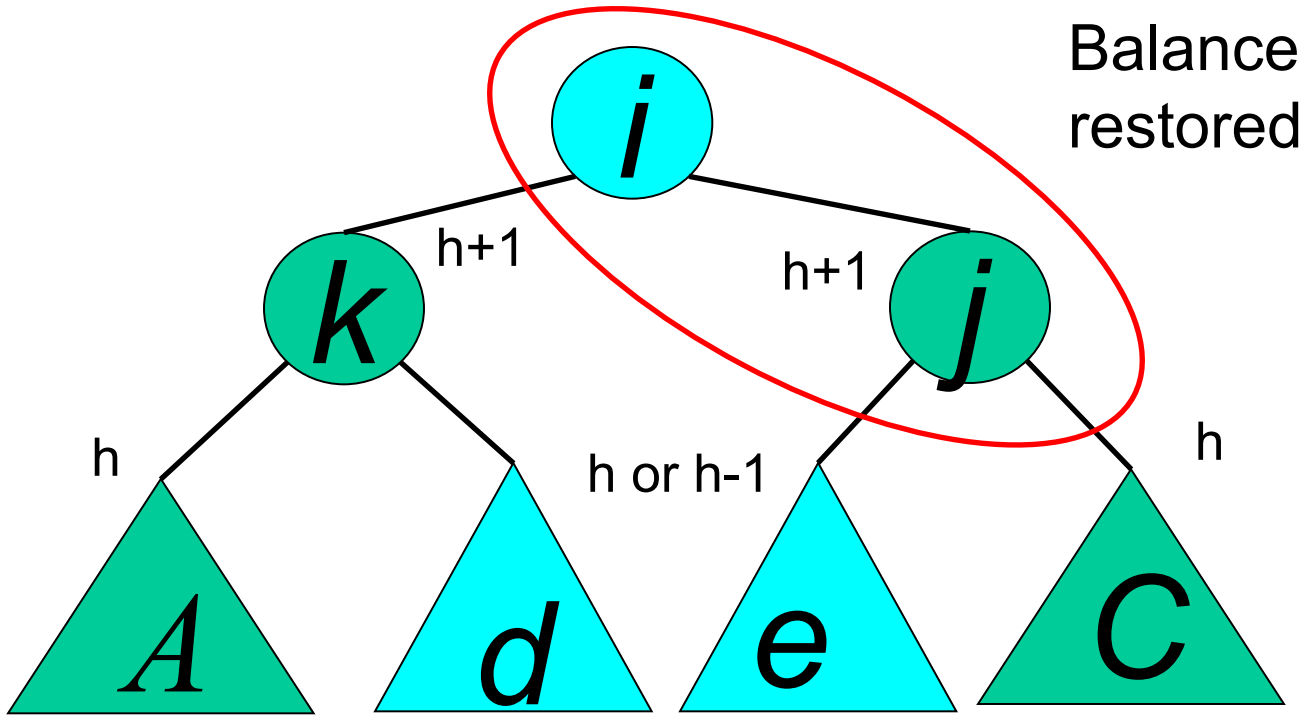


Now do a right rotation

Case 3: Rotations Complete

right rotation complete

Balance has been restored



Post-insertion fix-up

Let ***p*** be the inserted node - where are the nodes whose height changed? They can be only on the path from ***p*** up to the root.

Simple “search-and-repair” recipe:

•From the newly inserted node ***p***, go up towards the root, adjusting the heights (of nodes on the path)

•Look for the first (from ***p***) node ***z*** which is unbalanced

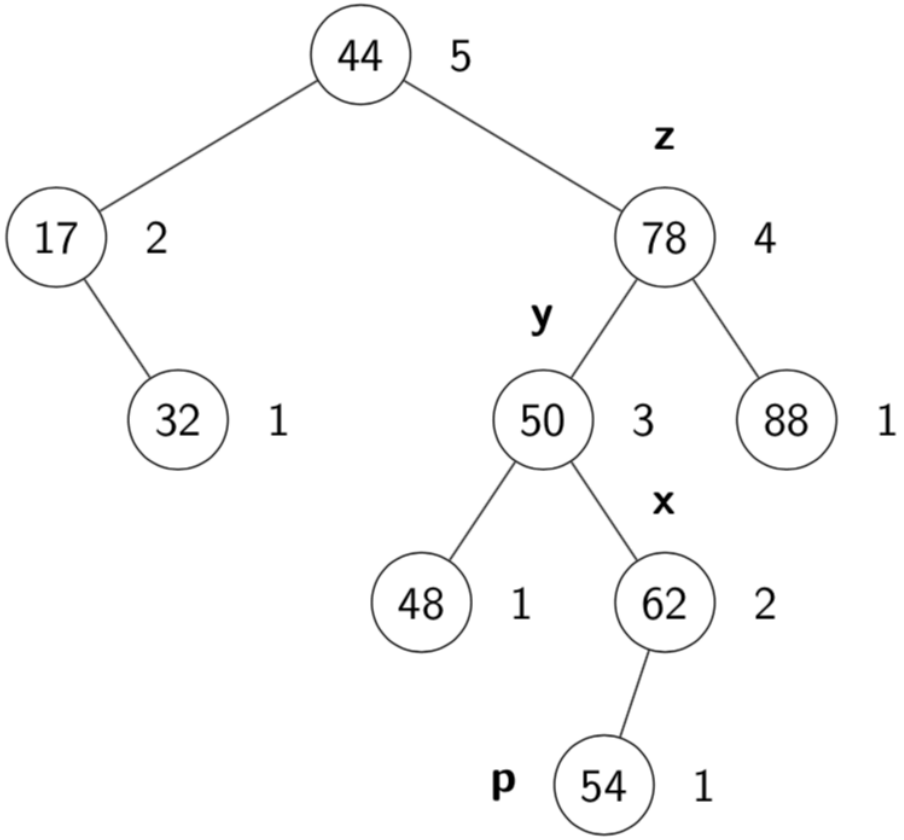
- i.e. heights of its children differ by more than 1

•If no such ***z*** is found, the height balance is not violated

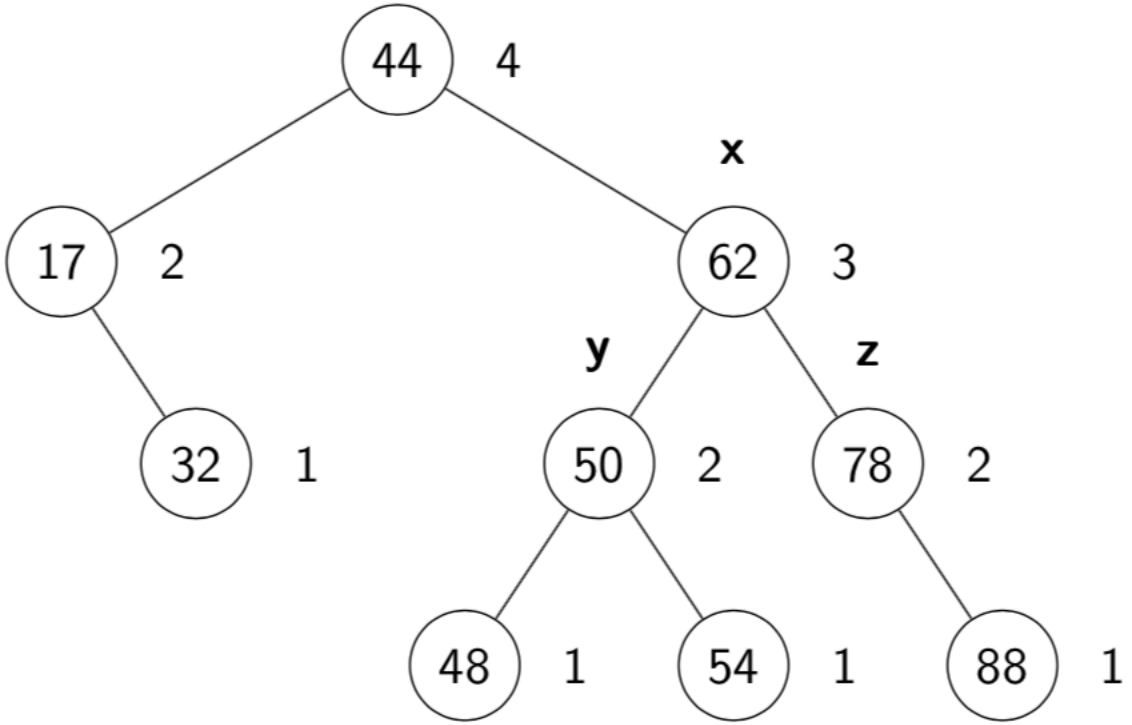
•If it is found, let ***y*** and ***x*** be the child and grandchild (respectively) of ***z*** on the path to ***p***.
Do trinode restructuring on ***z - y - x***

•This restores the height-balance property (will see why).

Previous example



After trinode restructuring



Note: the height-balance property is restored.

Another example

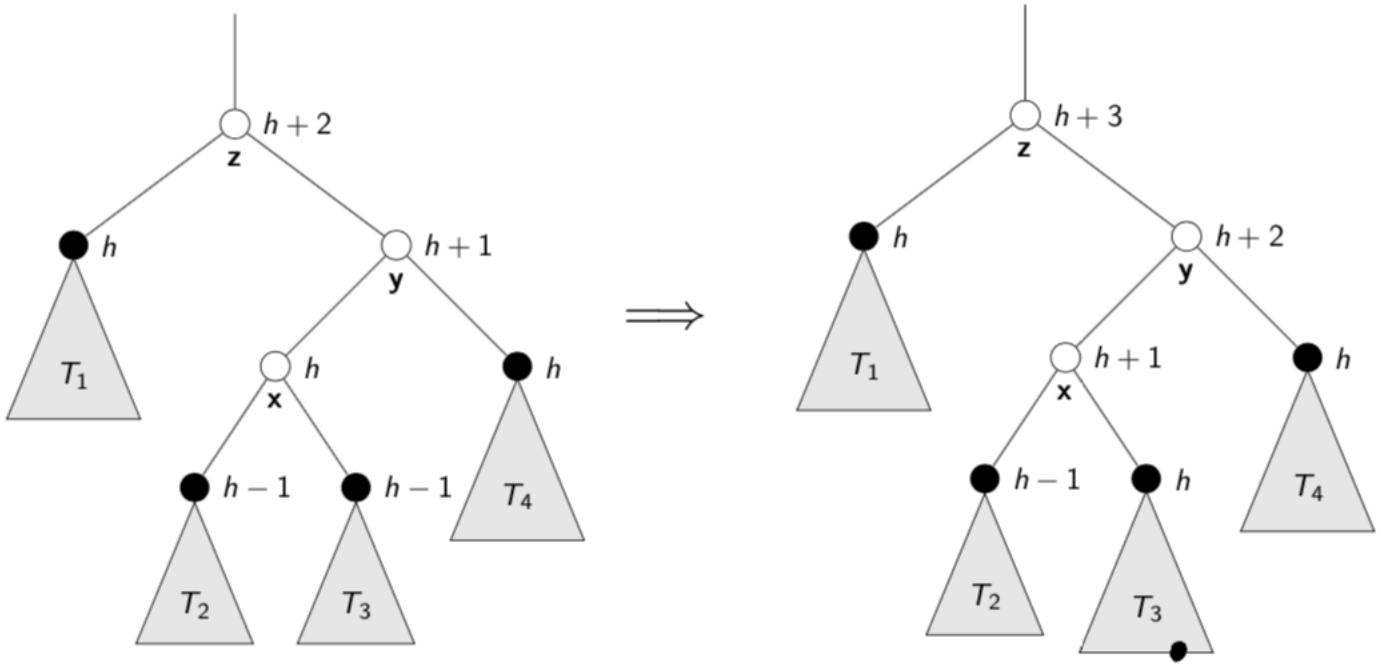
Insert the following numbers into an AVL Tree:

10, 20, 15, 25, 30, 16, 18, 19

Proof of re-balancing

Assume z is the first unbalanced node on the path from p up.

The subtree rooted at z before and after insertion (into T3):



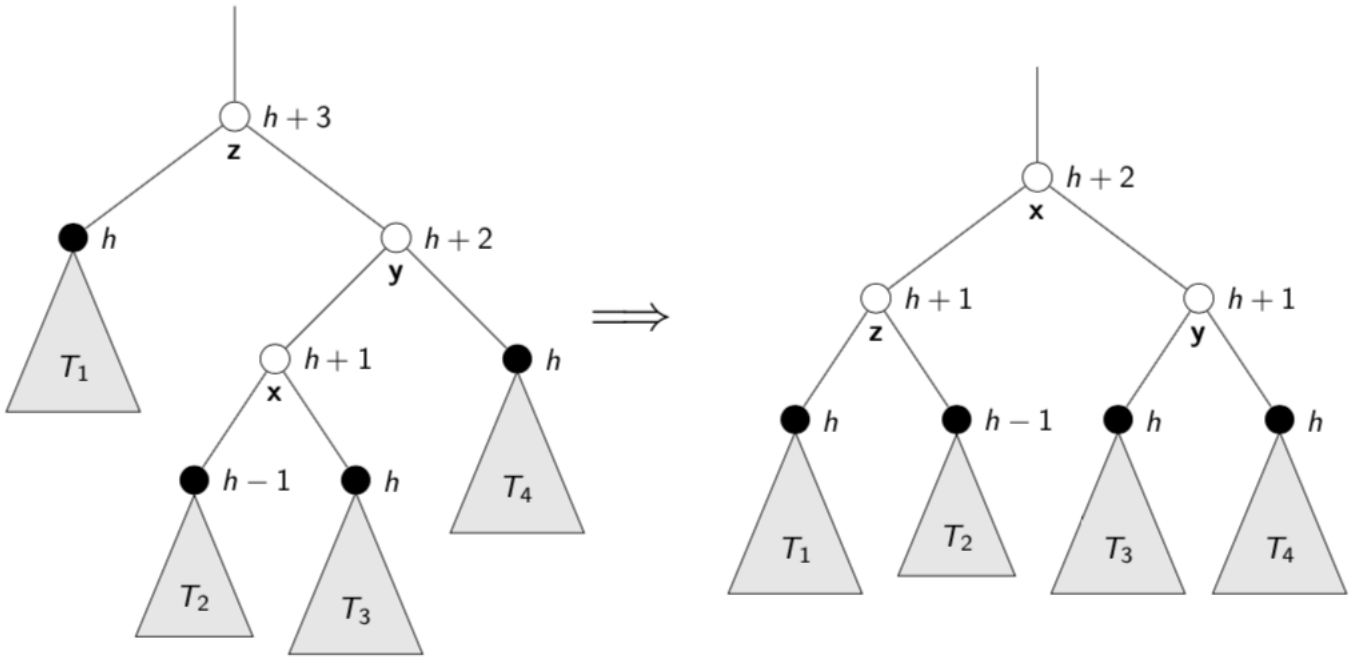
Observations

Since z becomes unbalanced, but y doesn’t, we must have:

- The height of z goes up by 1
- If y’ is the sibling of y then the height of y goes up by 1 and is now the height of y’ + 2
- The children of y had equal height before the insertion, and the height of x goes up by 1
- Since the height of x goes up by 1, have x = p (and its height goes from 0 to 1) or the children of x had equal height

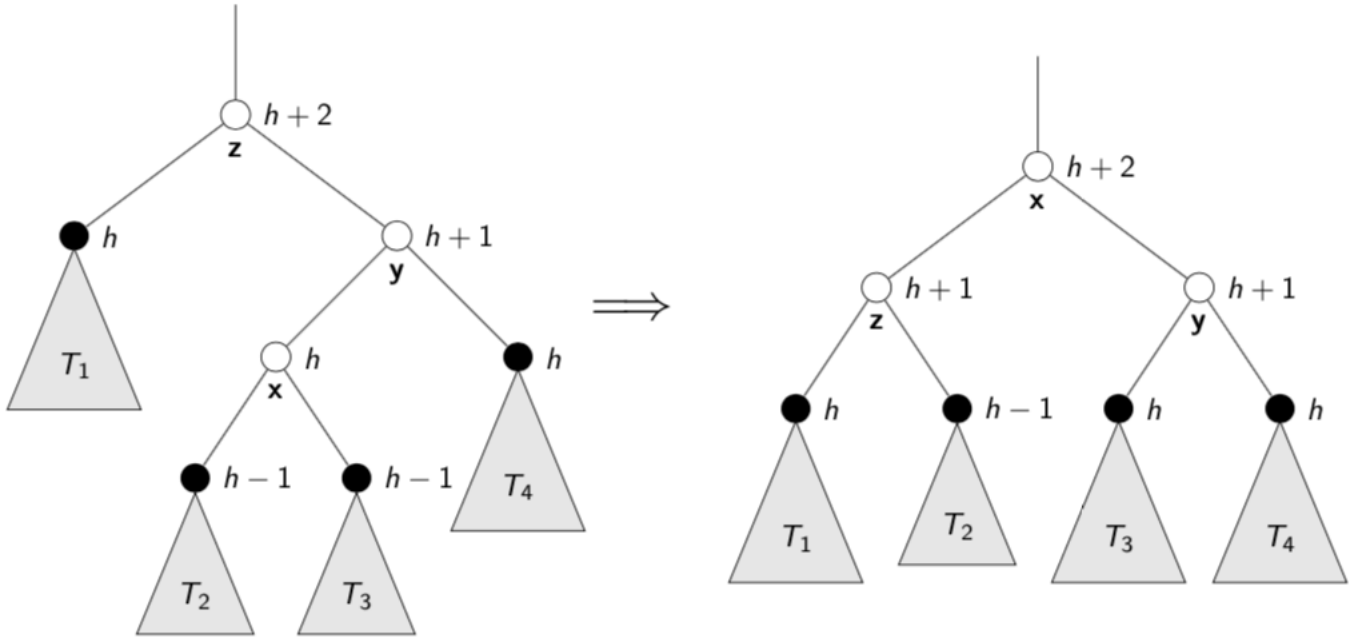
Proof of re-balancing cont'd

The subtree rooted at z before and after trinode restructuring:



Proof of re-balancing finished

Insertion and fix-up (trinode restructuring) combined:



This subtree is now balanced and has the same height as before insertion. So, the height-balance property is restored globally. (We checked the double rotation case, the other case is similar)

Complexity of insertion

The standard BST insertion in AVL trees is O(logn).

Fix-up procedure:

- travel up the tree from the inserted node (looking for an unbalanced node)

— O(log n) time.

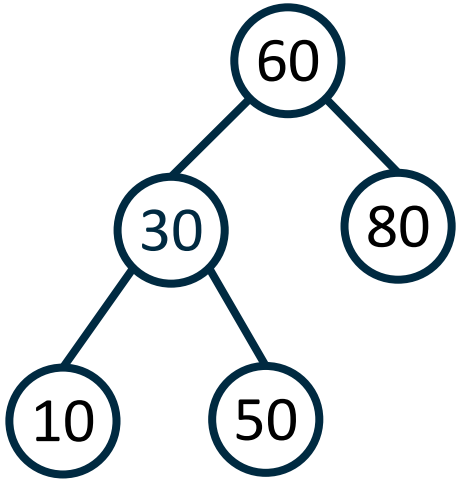
- do trinode restructuring at most once

— O(1) time.

All in all, the worst-case running time is O(logn).

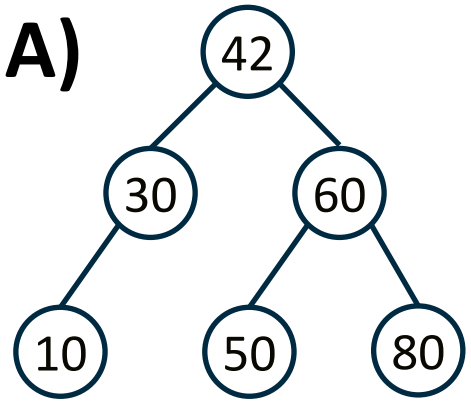
Test your knowledge!

Starting with this AVL tree:

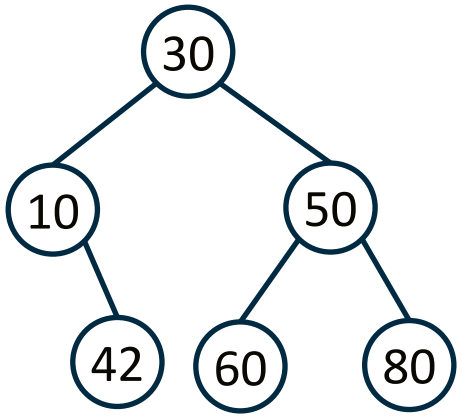


Which of the following is the updated AVL tree after inserting 42?

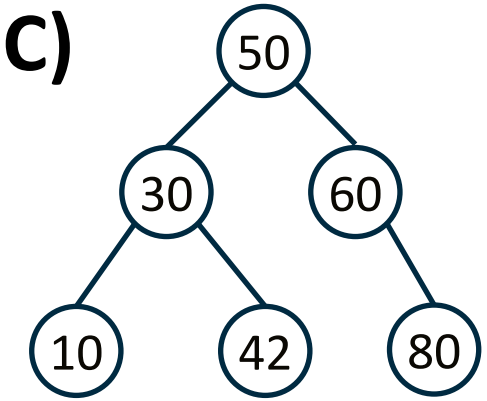
A)



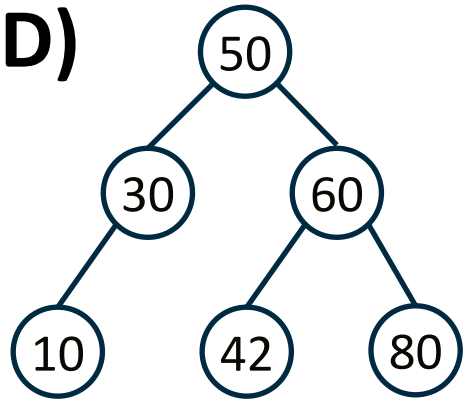
B)



C)

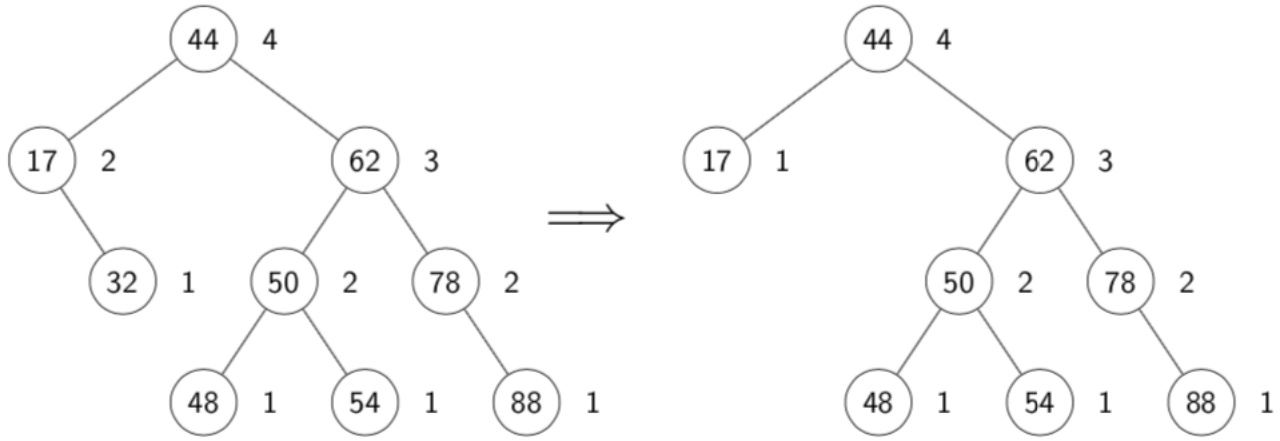


D)



Deletion

Standard BST deletion can violate the height-balance property.



So again we might need to fix it up after deletion.

Post-deletion fix-up

If p is the parent of the deleted node (which is not always the node that contained deleted data!), there may be unbalanced nodes on the path from p to the root.

Can use trinode restructuring to re-balance:

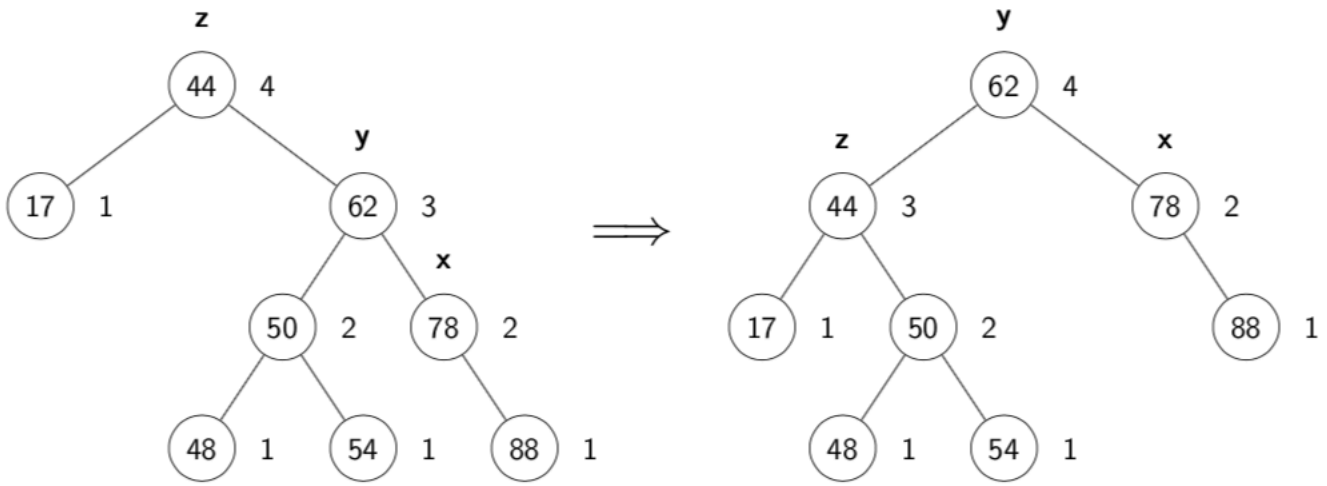
- Let z be the first unbalanced node on the path
- Let y be the taller child of z (i.e. the one not on the path)
- Let x be the taller child of y (or, if children of y have the same height, on the same left/right side as y)
- Use trinode restructuring on z − y − x.

This will fix up the subtree of z, but may reduce its height.

Must keep going up the tree and do restructuring whenever we meet an unbalanced node.

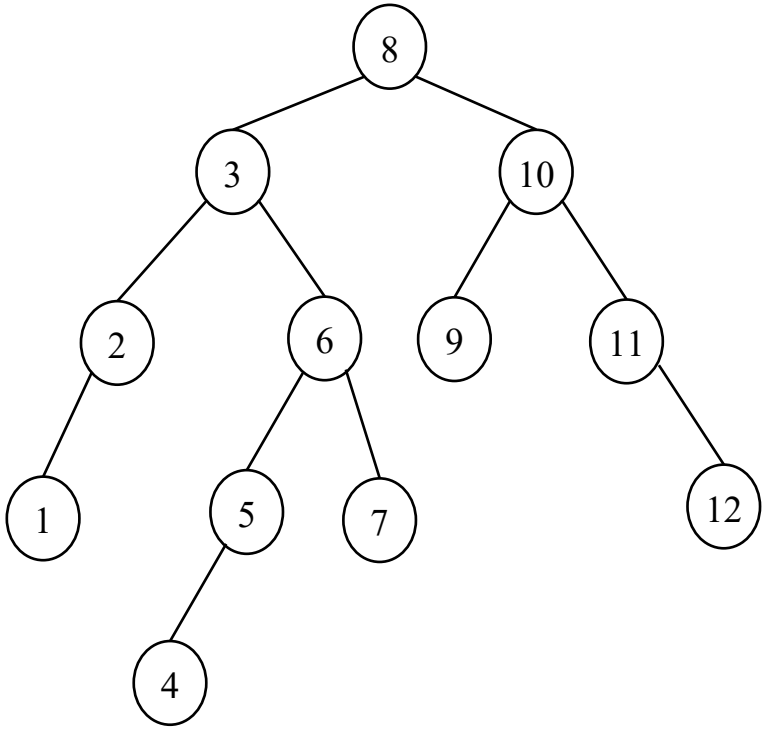
Example

Restoring height balance after deletion



Another example

delete(8)



Complexity of deletion

The standard BST deletion in AVL trees is O(logn).

The fix-up procedure uses O(1) time per level

Done at most once per level, and the height is O(logn).

All in all, the worst-case running time is O(logn).

The post-processing for insertion and deletion can be unified:

- Both trace an upward path from some node
- Both use trinode restructuring when an unbalanced node is found

“Early stop” condition: Can stop the post-processing when

- reach an ancestor whose height didn’t change, or
- trinode restructuring of a subtree results in the same height as before insertion/deletion

To detect this, can store the “old” height of each node

AVL Tree – Performance Summary

AVL tree storing n items

- The data structure uses O(n) space
- A single restructuring takes O(1) time
  - using a linked-structure binary tree
- Searching takes O(log n) time
  - height of tree is O(log n), no restructures needed
- Insertion takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)
- Removal takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)

Durham University

Red-black trees (briefly)

Red-black trees (optional – not examined)

A red-black tree is an ordinary BST with one extra bit of storage per node: its colour, red or black

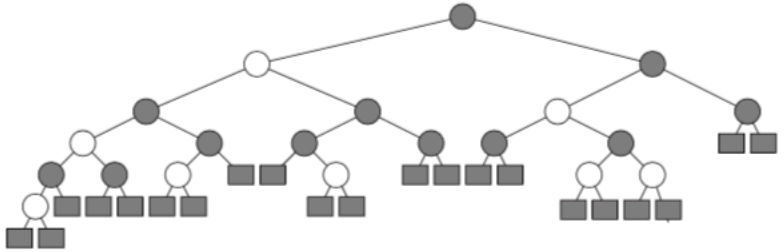
Constraining how nodes can be coloured results in (approximate) balancedness of tree

Assumption: if either parent, left, or right does not exist, pointer to NULL

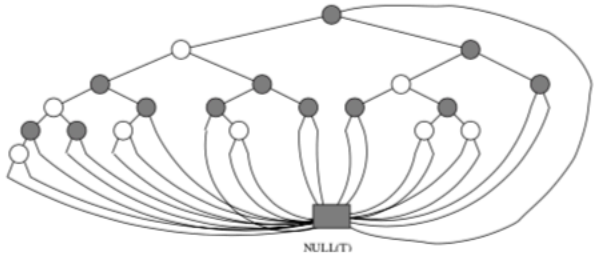
Regard NULLs as external nodes (leaves) of tree, thus normal nodes are internal

A BST is a **red-black tree** if

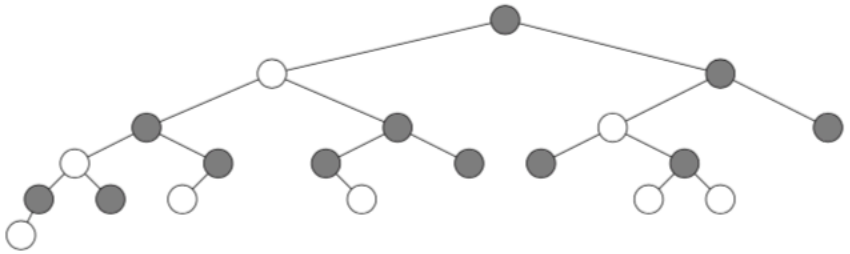
1. every node is either red or black
2. the root is black
3. every leaf (Null) is black
4. red nodes have black children
5. for each node, all paths from the node to descendant leaves contain same number of black nodes



red-black tree, according to definition



replace NULLs with single "sentinel" NULL(T)



normal drawing style (no NULLleaves or sentinel), but sentinel **is** there

Most important property:

Lemma

A red-black tree with n internal nodes has height at most $2\log(n+1)$.

Proof omitted (idea: have same number of black nodes on each path from root to any leaf, so it's black-balanced, plus can't have too many red nodes).

As with AVL trees,

- all operations are O(log n) time
- insertion and deletion require post-processing (fix-up)
  - can be done with recolouring and rotations

Insertion

Suppose we insert new node z

First, **ordinary BST insertion** of z according to key value (walk down from root, open new leaf)

Then, colour z **red**

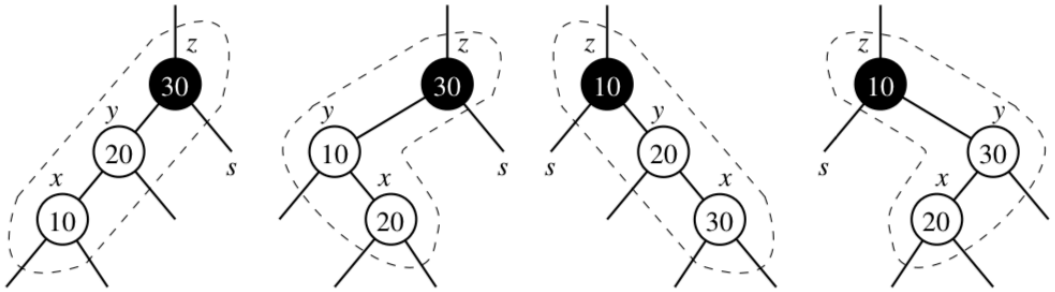
May have **violated red-black property**

(e.g., z's parent is red)

Call a fix-up procedure to restore red-black property

Several cases, with fix-up procedures that look like so:

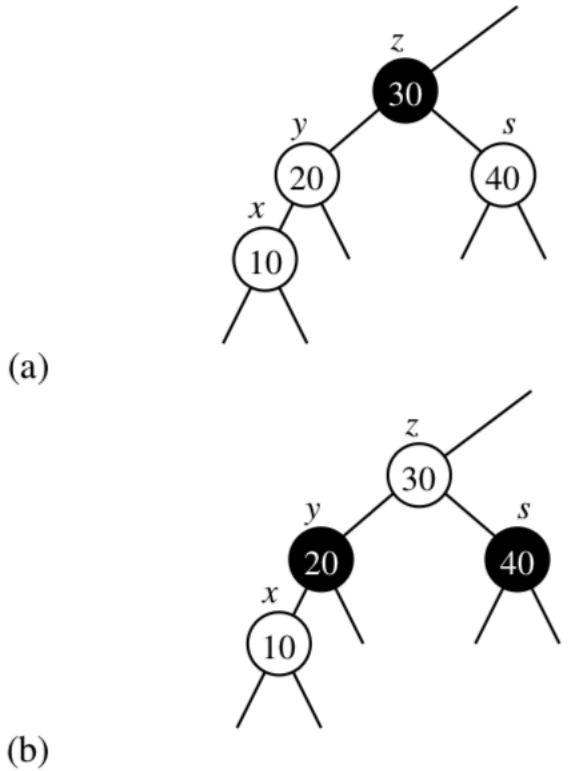
Case 1: ***The Sibling s of y Is Black (or None)***



Restructuring a red-black tree to remedy a double red: (a) the four configurations for x, y, and z before restructuring; (b) after restructuring.

Case 2: ***The Sibling s of y Is Red.***

Do a ***recoloring***: color y and s black and their parent z red (unless z is the root, in which case, it remains black).

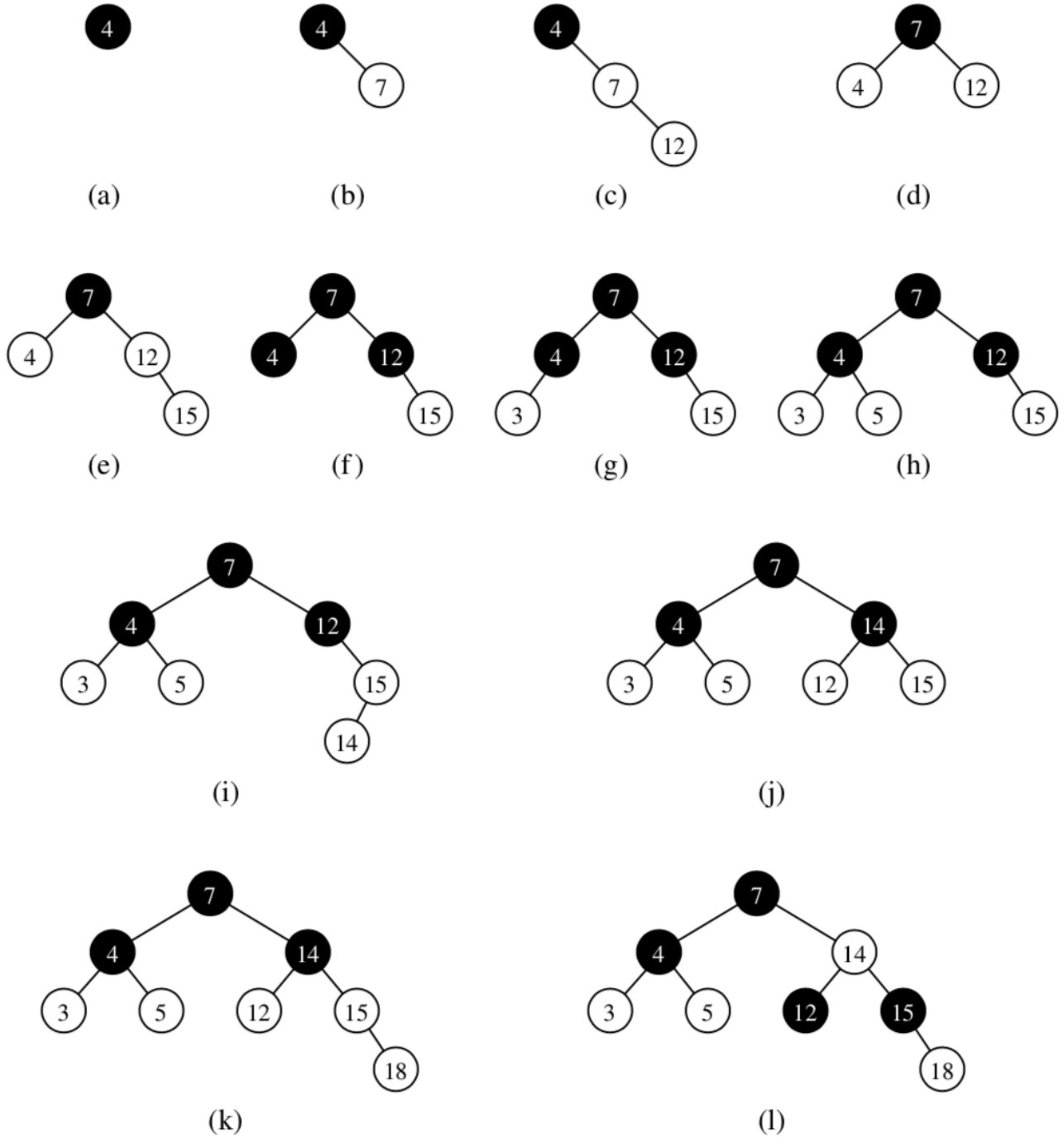


Durham University

Example

A sequence of insertions in a red-black tree:

(a) initial tree;
(b) insertion of 7;
(c) insertion of 12, which causes a double red;
(d) after restructuring;
(e) insertion of 15, which causes a double red;
(f) after recoloring (the root remains black);
(g) insertion of 3; (h) insertion of 5;
(i) insertion of 14, which causes a double red;
(j) after restructuring;
(k) insertion of 18, which causes a double red;
(l) after recoloring.



AVL vs. red-black

Some trade-offs between AVL trees and red-black trees:

To control heights, AVL either use more space than red-black (to store heights in nodes), but still O(n) space, or use extra time to compute heights (if not stored) - how much extra?

AVL trees are better for lookup-intensive tasks - because the balance condition is stricter

Red-black trees are more efficient for insertion/deletion - because the balance condition is more relaxed (easier to fix up).

Durham University

Thank you