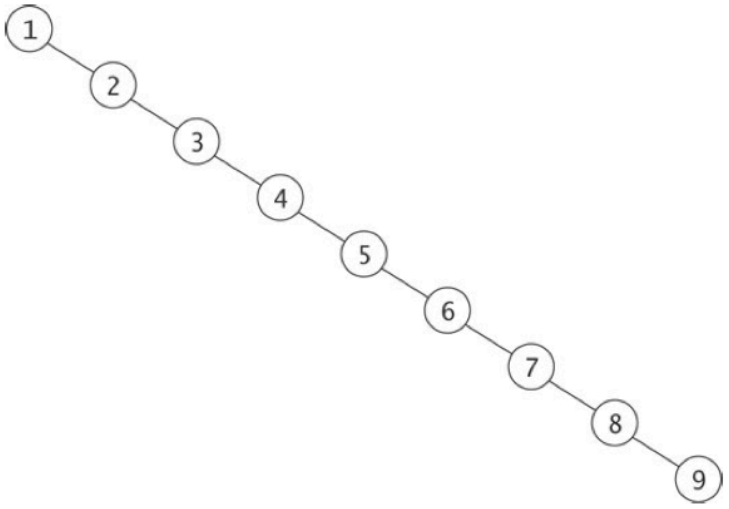
Lecture 10
Data Structures (DAT037)

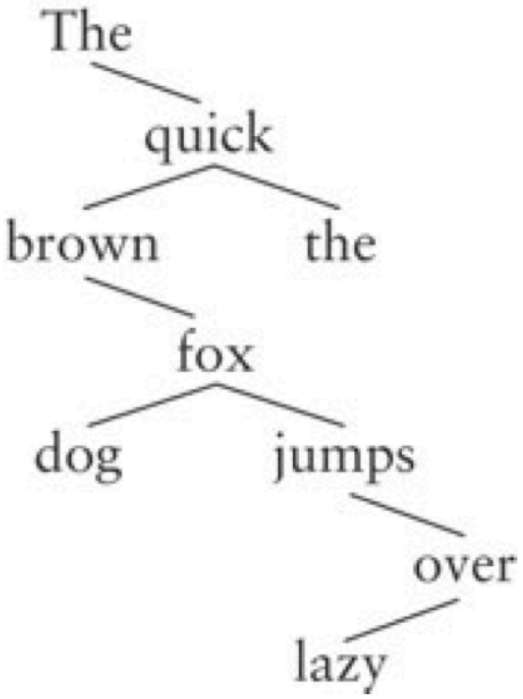
Ramona Enache

(with slides from Nick Smallbone and Nils Anders Danielsson)

Balanced BSTs: Problem

The BST operations take O(height of tree), so for unbalanced trees can take O(n) time





Balanced BSTs: Solution

Take BSTs and add an extra invariant that makes sure that the tree is balanced

- Height of tree must be O(log n)
  ➔ all operations will take O(log n) time

One possible idea for an invariant:
Height of left child = height of right child (for all nodes in the tree)

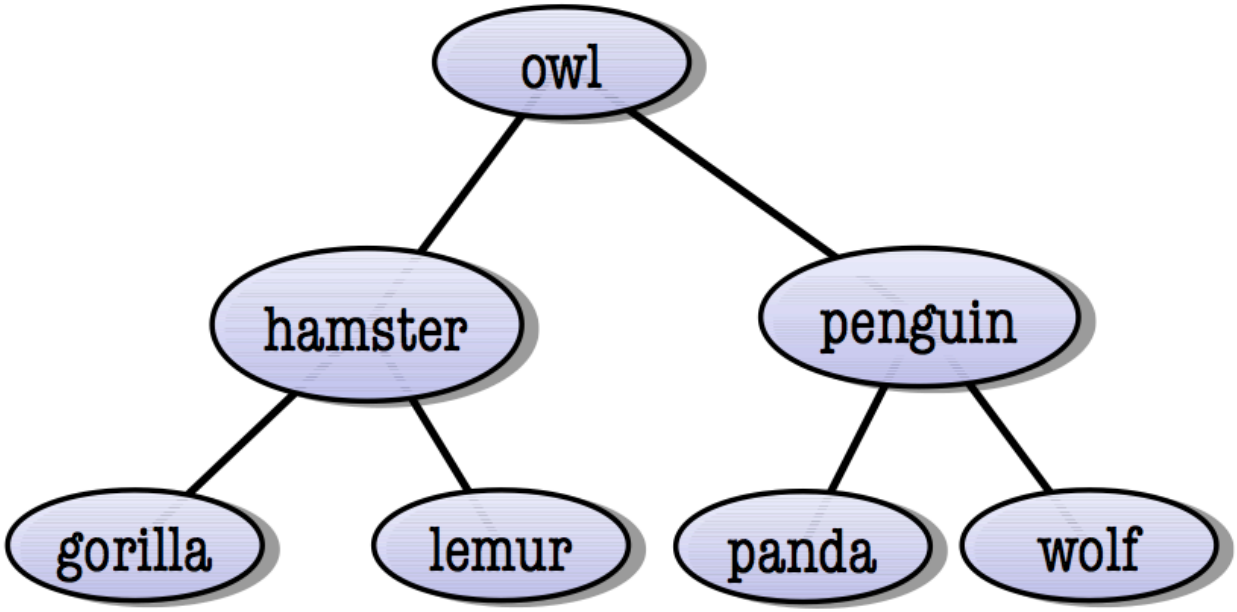
Tree would be sort of “perfectly balanced”

What's wrong with this idea?

Balanced BSTs: Solution

Perfect balance is too restrictive!

Number of nodes can only be 1, 3, 7, 15, 31, ...



Balanced BSTs: AVL

The **AVL tree** is the first balanced BST discovered (from 1962) – it's named after Adelson-Velsky and Landis

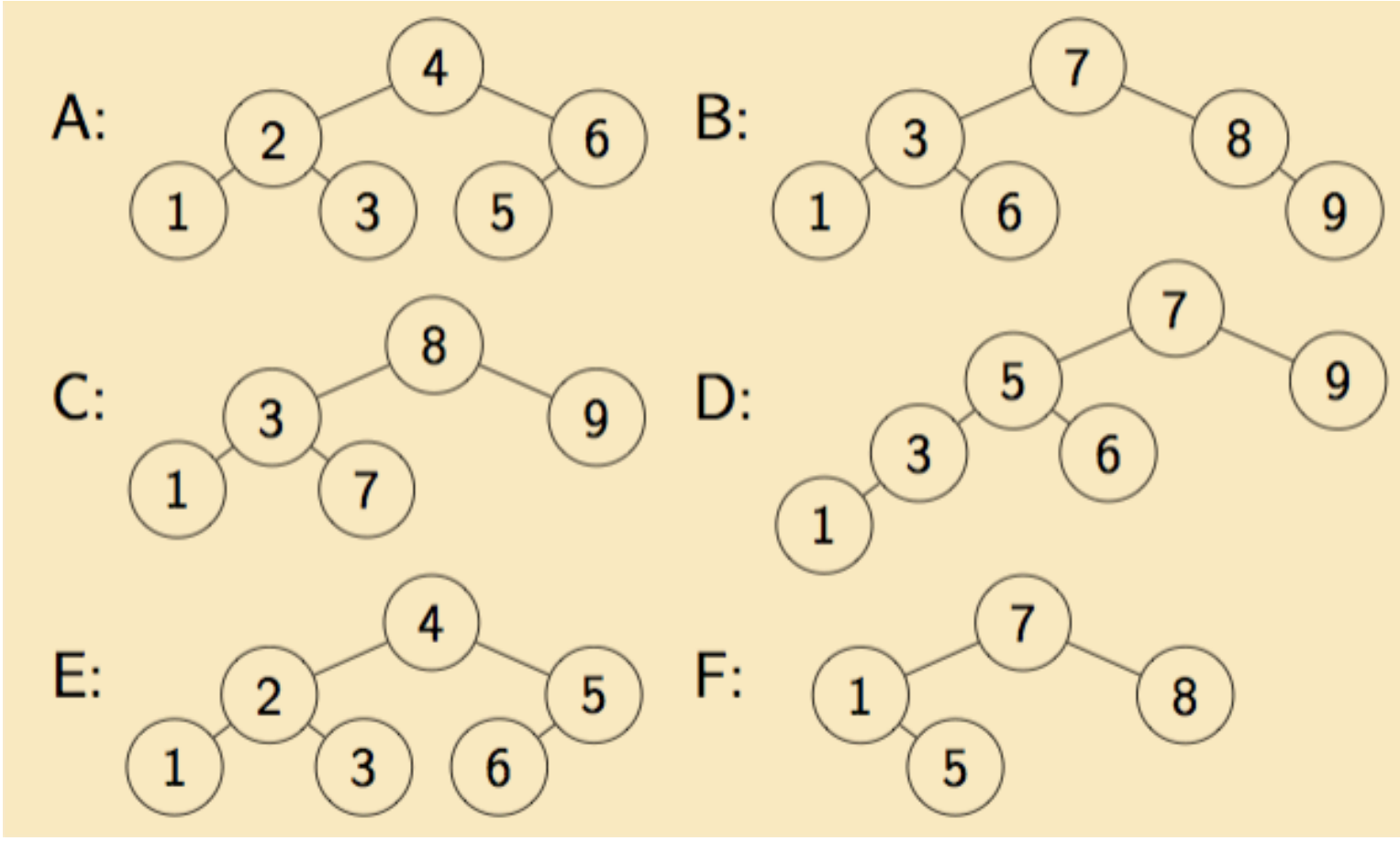
It's a BST with the following invariant:

The difference in heights between the left and right children of any node is at most 1

This makes the tree's height O(log n), so it's balanced

Balanced BSTs: AVL

Which of these are AVL trees?



Balanced BSTs: AVL

We call the quantity right height – left height of a node its balance

Thus the AVL invariant is:

the balance of every node is -1, 0, or 1

Whenever a node gets out of balance, we fix it with so-called tree rotations (next)

(Implementation: store the balance of each node as a field in the node, and remember to update it when updating the tree)

Question

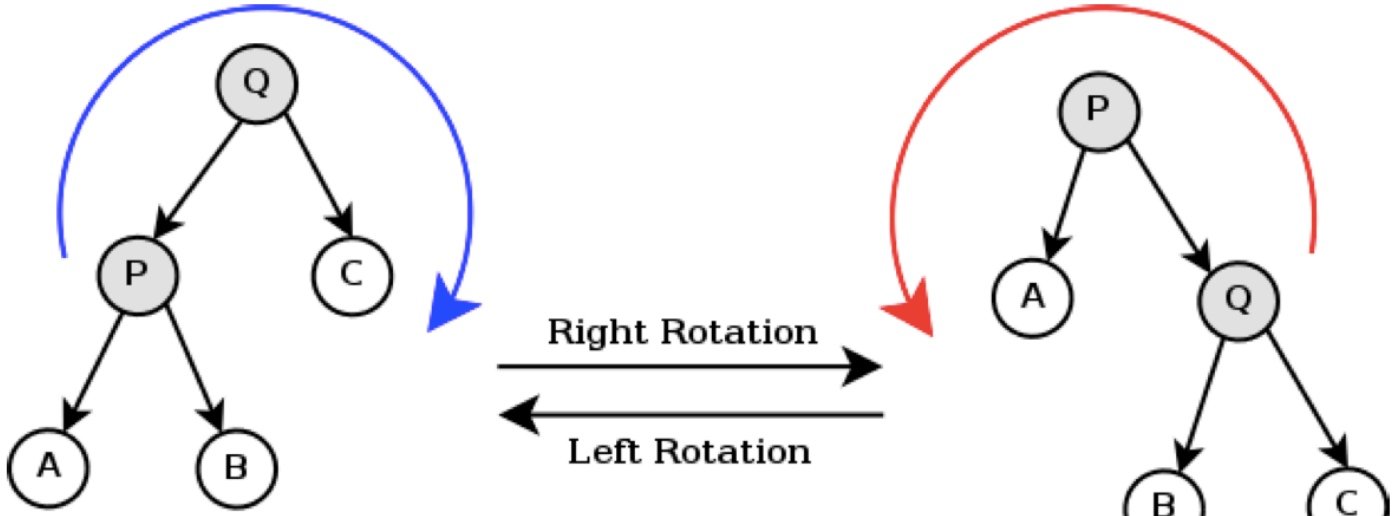
What is the number of AVL trees which contain only {1..5} ?

1. 5
2. 6
3. 7
4. 3

govote.at
Code 615748

Balanced BSTs: AVL

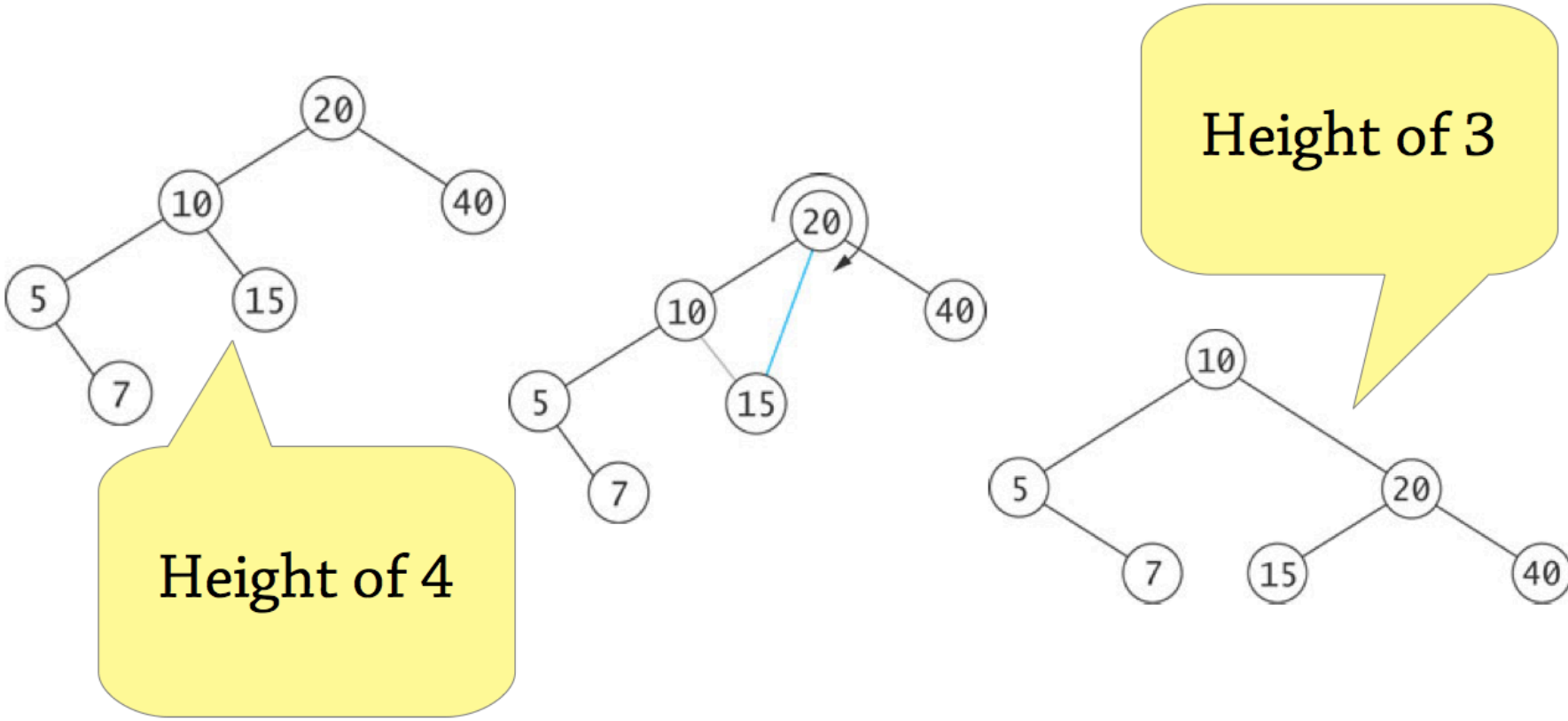
Rotation rearranges a BST by moving a different node to the root, without changing the BST's contents



(pic from Wikipedia)

Balanced BSTs: AVL

We can use rotations to adjust the relative height of the left and right branches of a tree

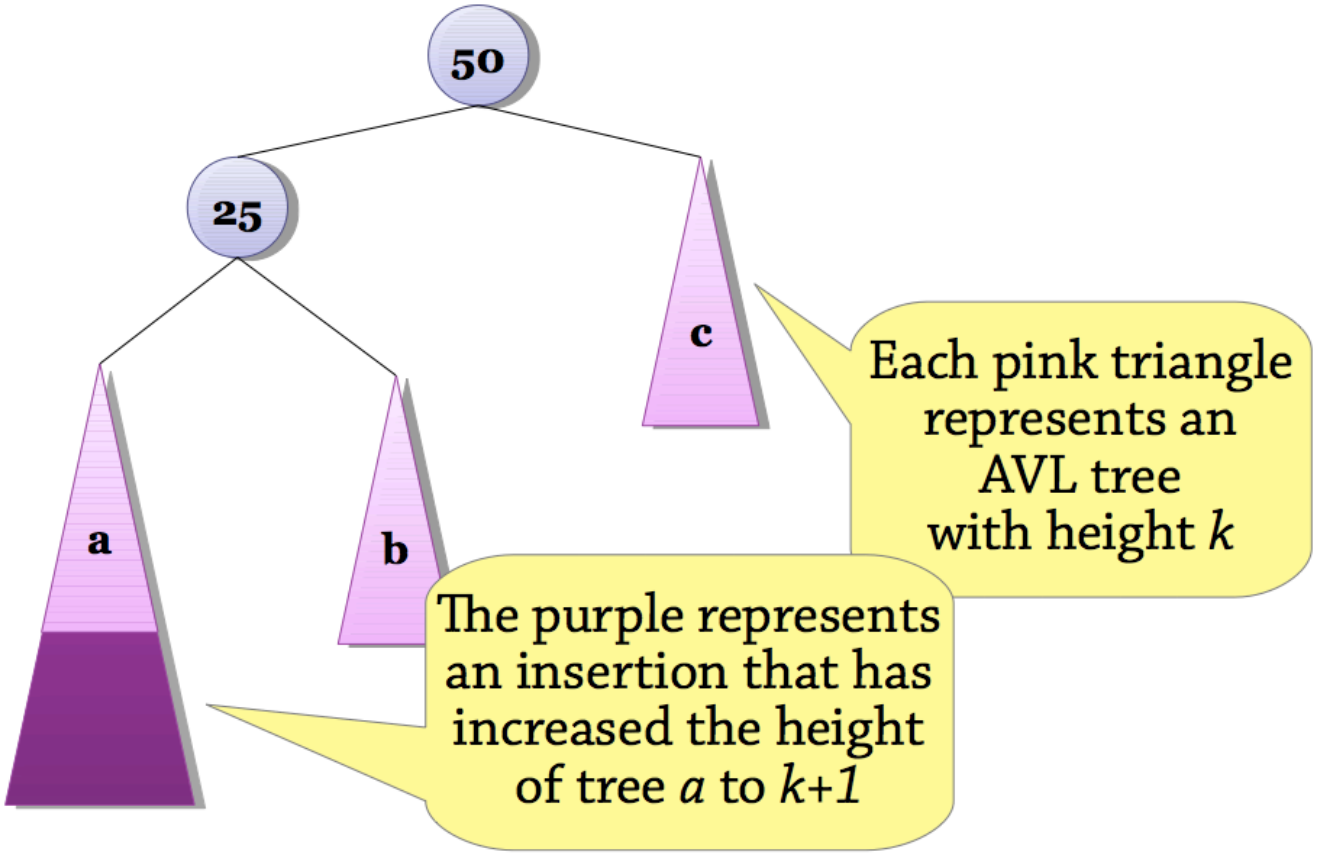


Balanced BSTs: AVL

Start by doing a BST insertion
This might break the AVL (balance) invariant
Then go upwards from the newly-inserted node, looking for nodes that break the invariant (unbalanced nodes)
Whenever you find one, rotate it
Then continue upwards in the tree
There are **4** cases depending on how the node is unbalanced

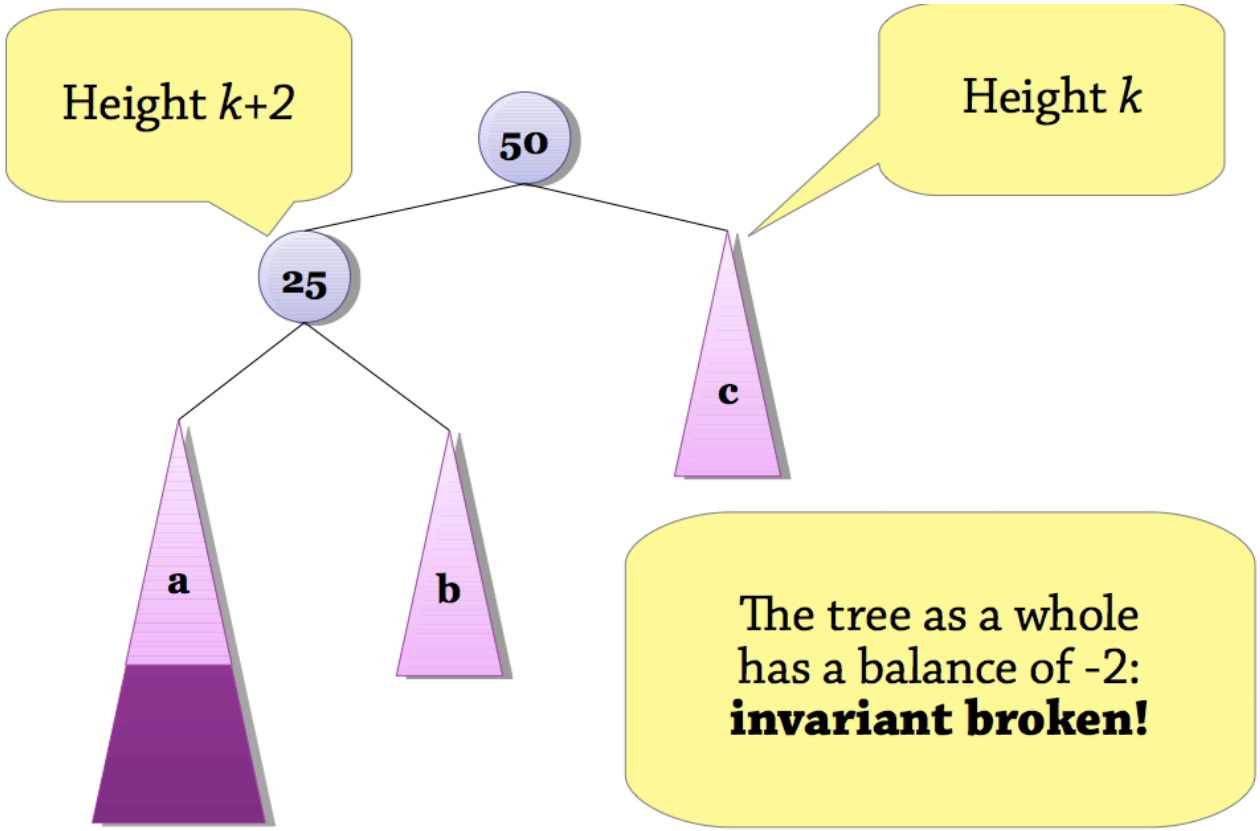
Balanced BSTs: AVL

Case 1: left-left tree



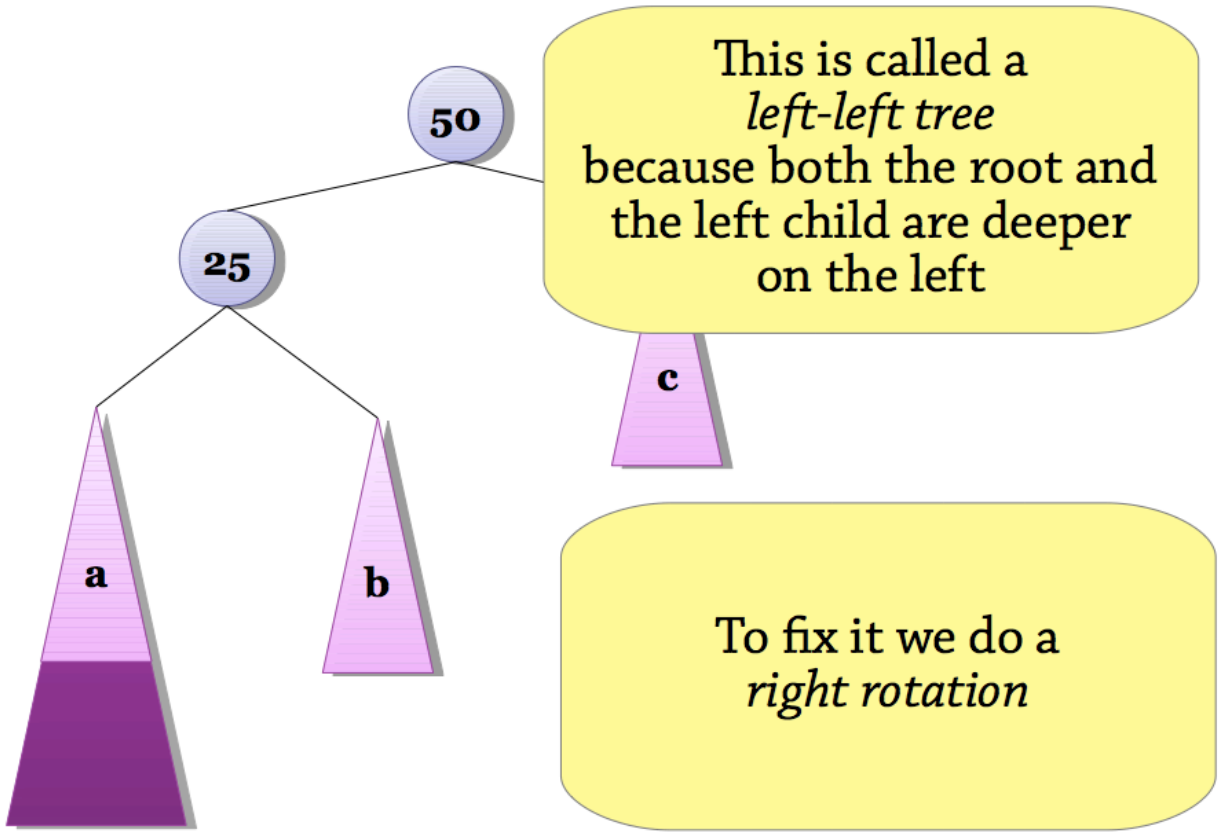
Balanced BSTs: AVL

Case 1: left-left tree



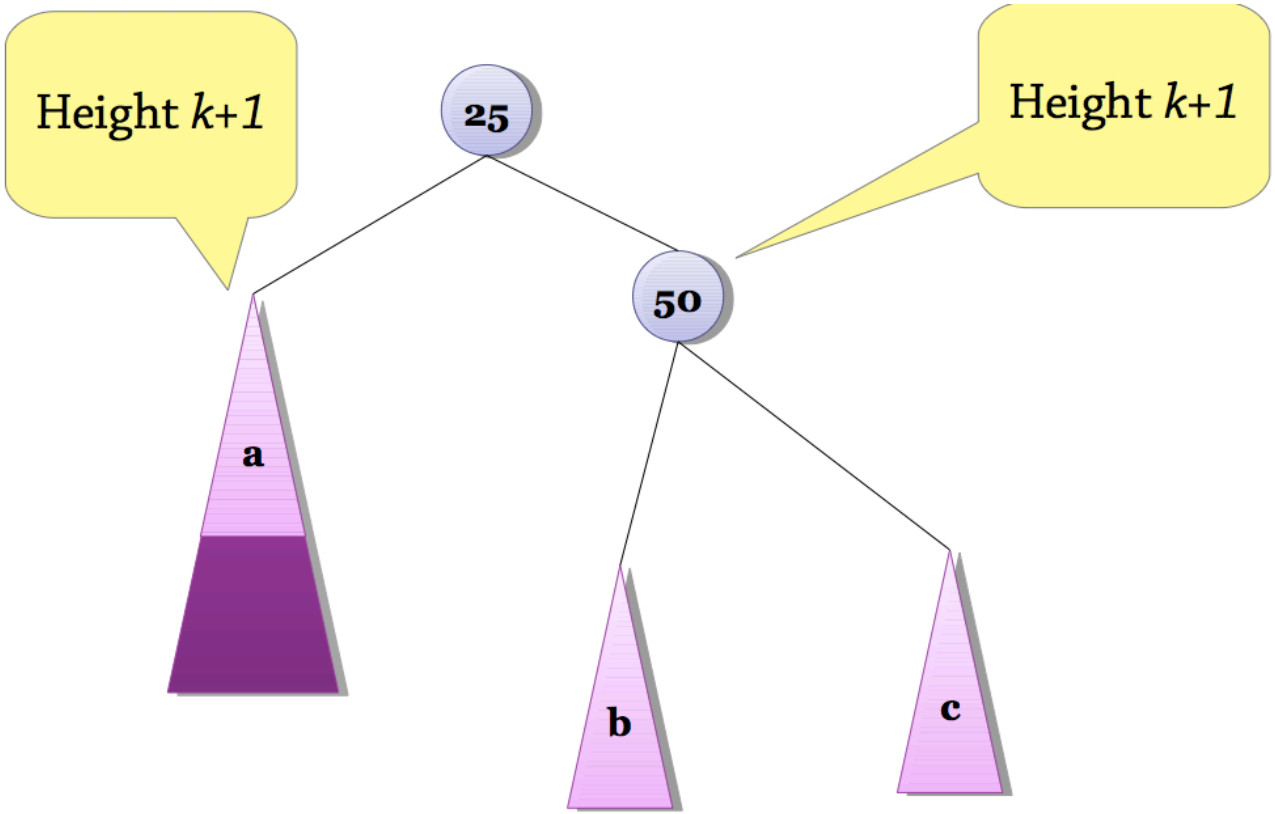
Balanced BSTs: AVL

Case 1: left-left tree



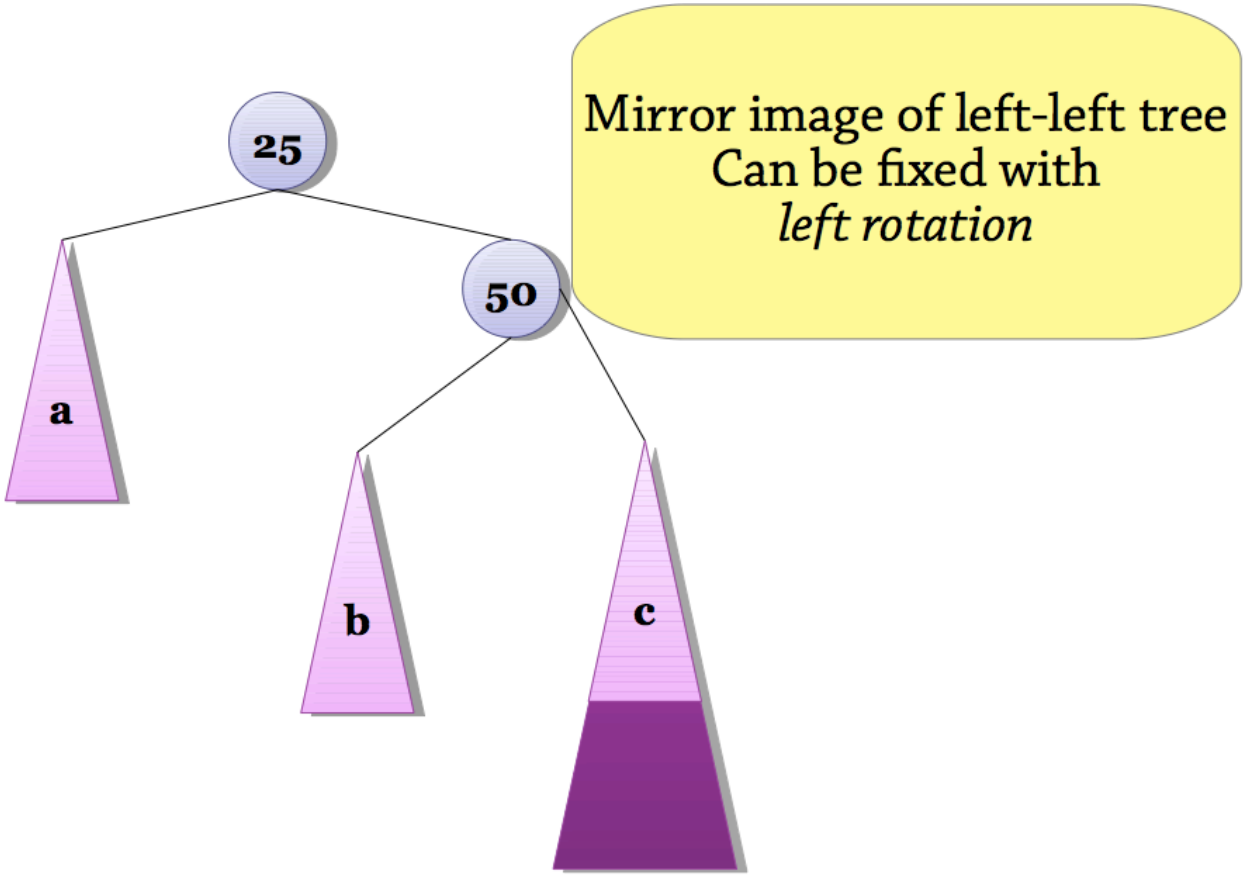
Balanced BSTs: AVL

Case 1: left-left tree



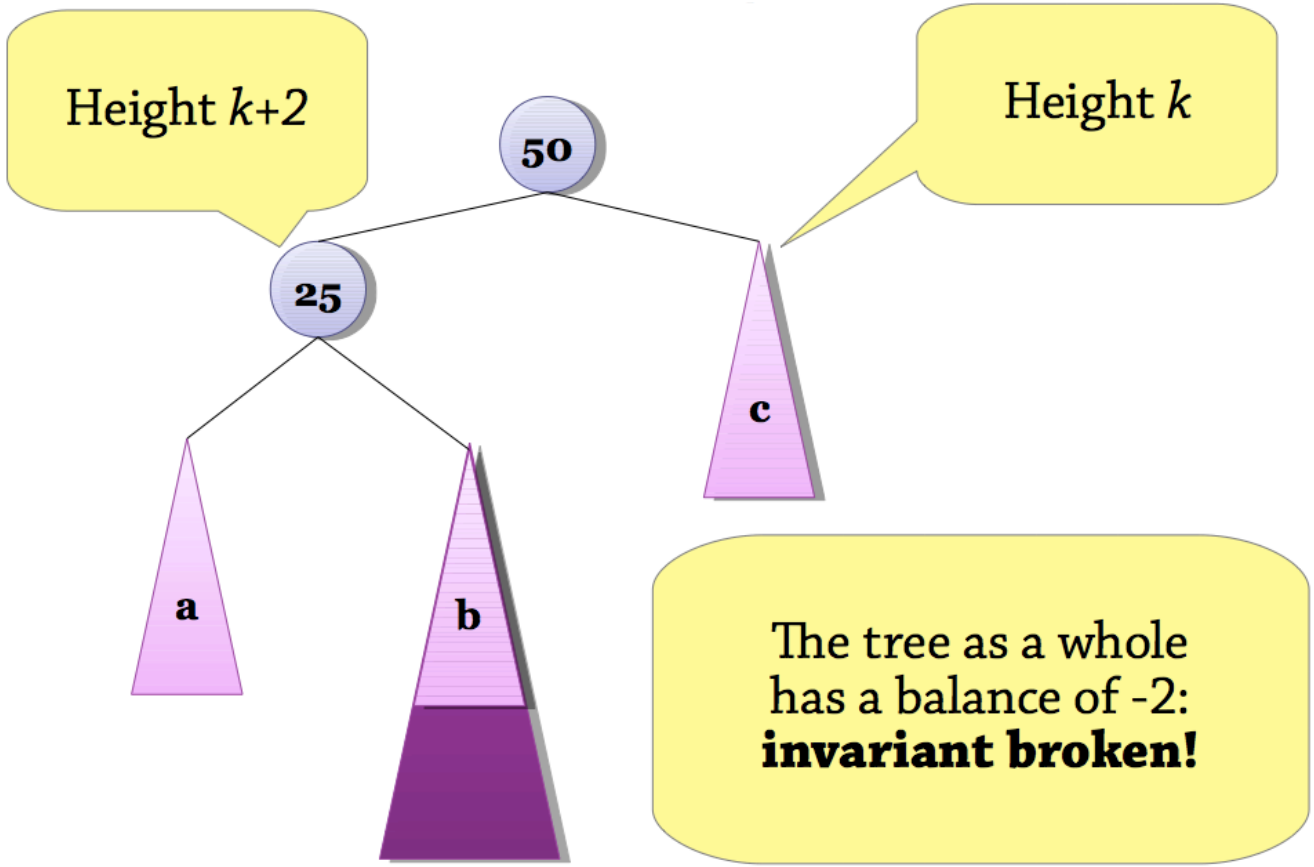
Balanced BSTs: AVL

Case 2: right-right tree



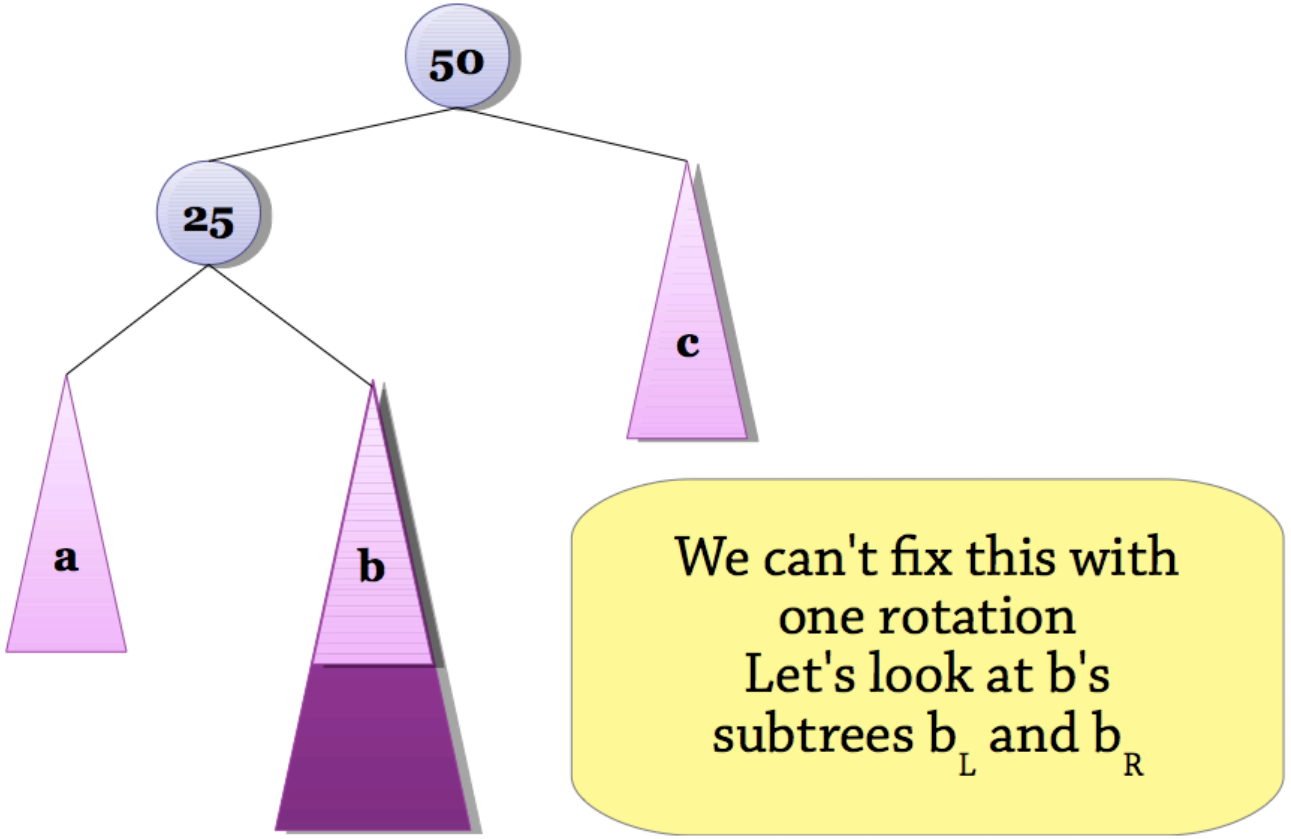
Balanced BSTs: AVL

Case 3: left-right tree



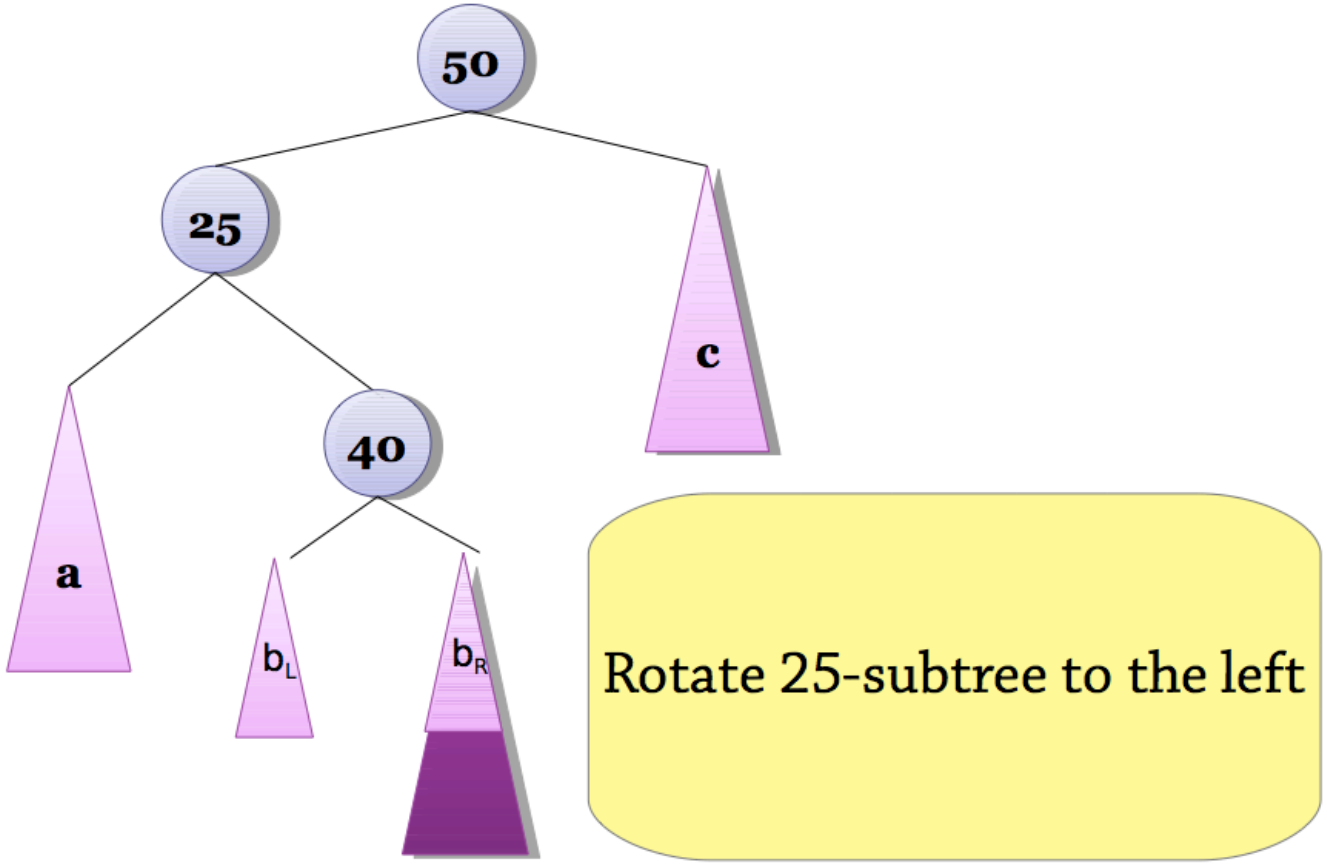
Balanced BSTs: AVL

Case 1: left-right tree



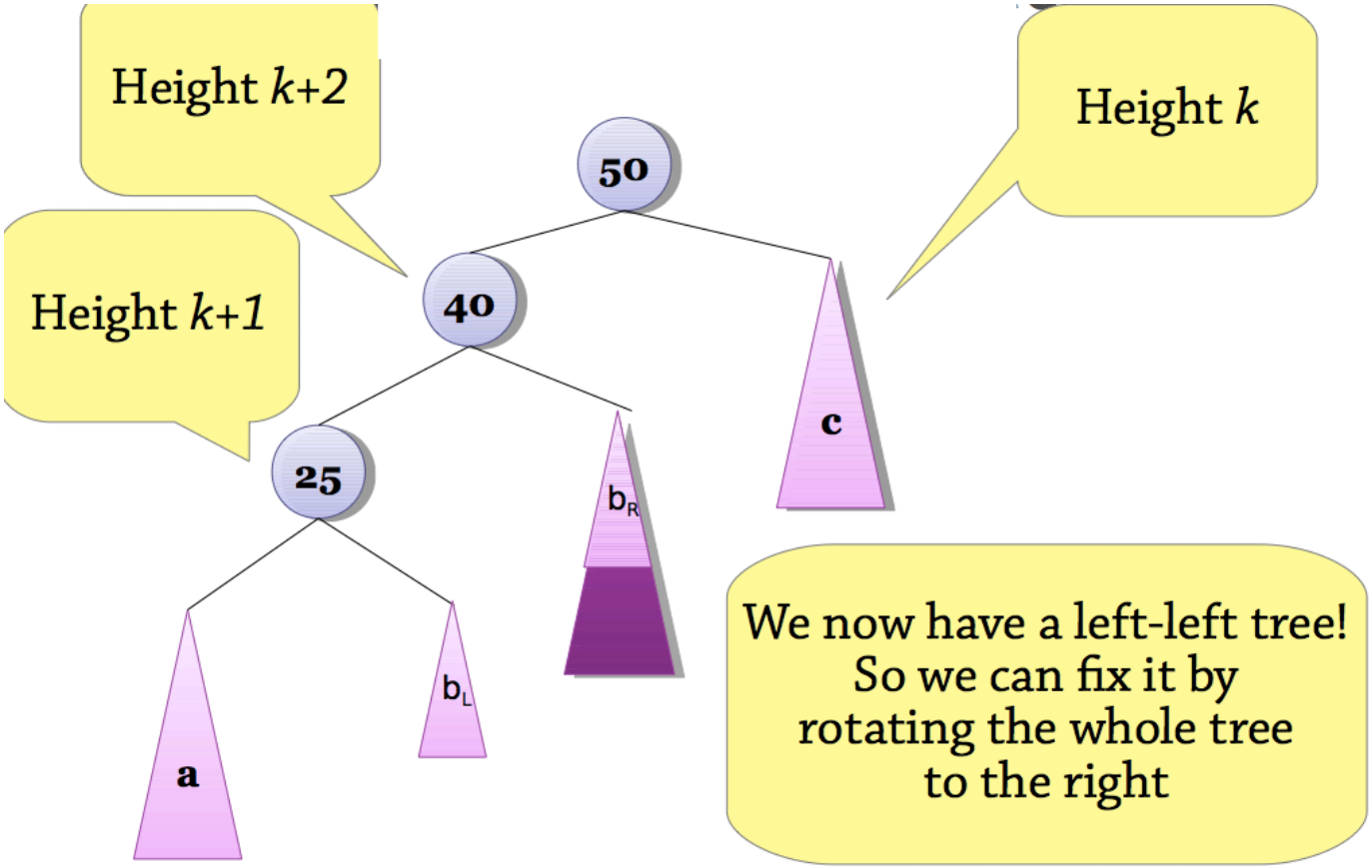
Balanced BSTs: AVL

Case 1: left-right tree



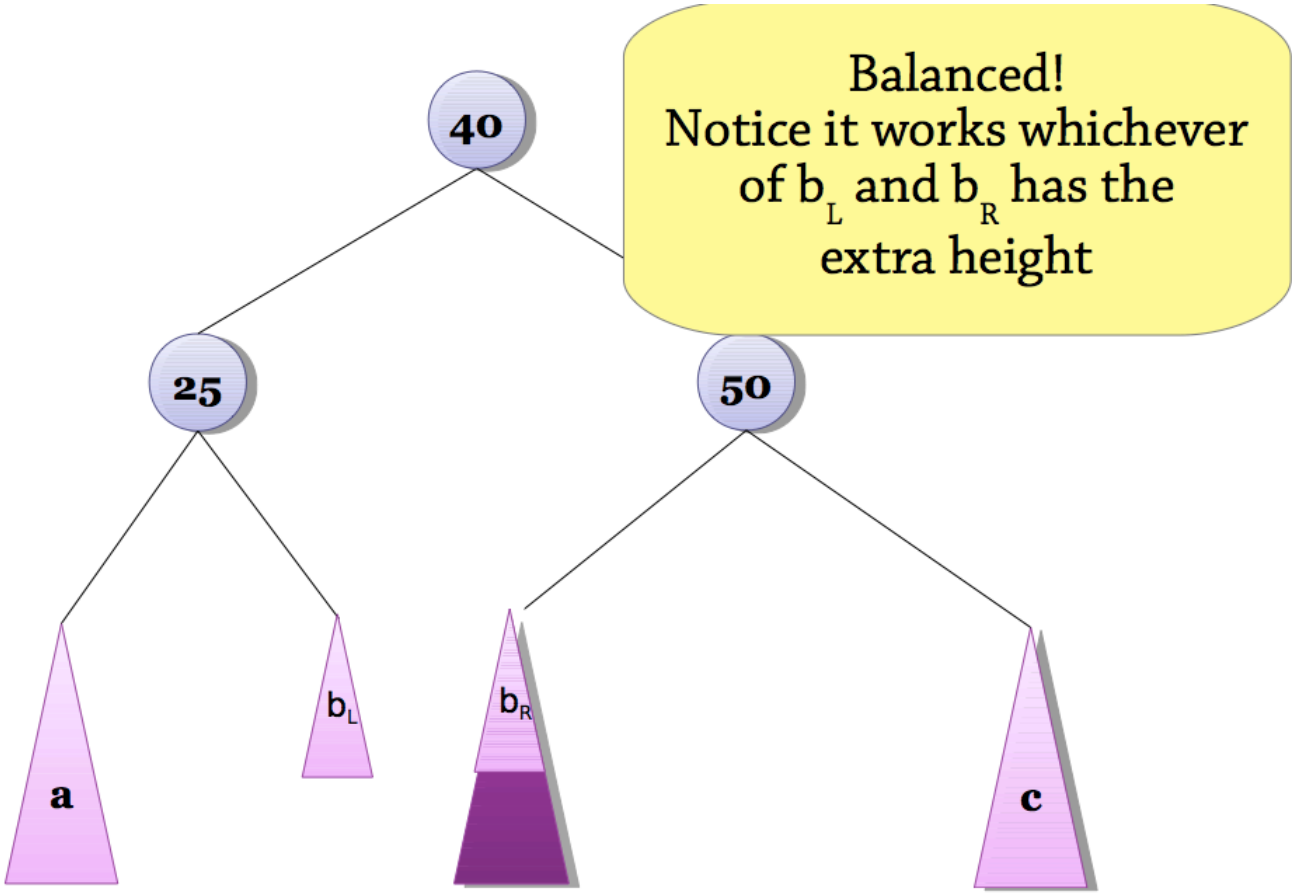
Balanced BSTs: AVL

Case 1: left-right tree



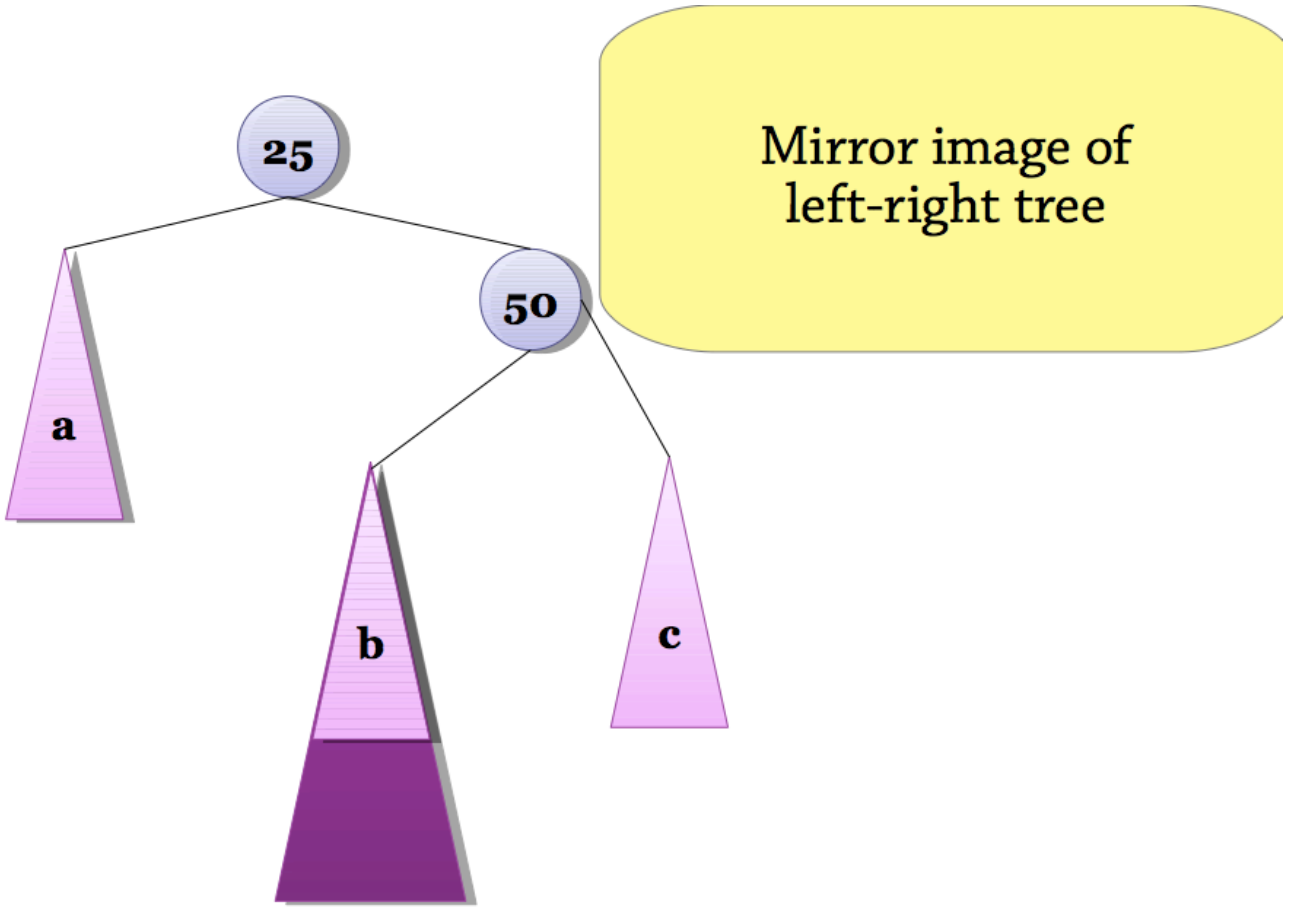
Balanced BSTs: AVL

Case 1: left-right tree



Balanced BSTs: AVL

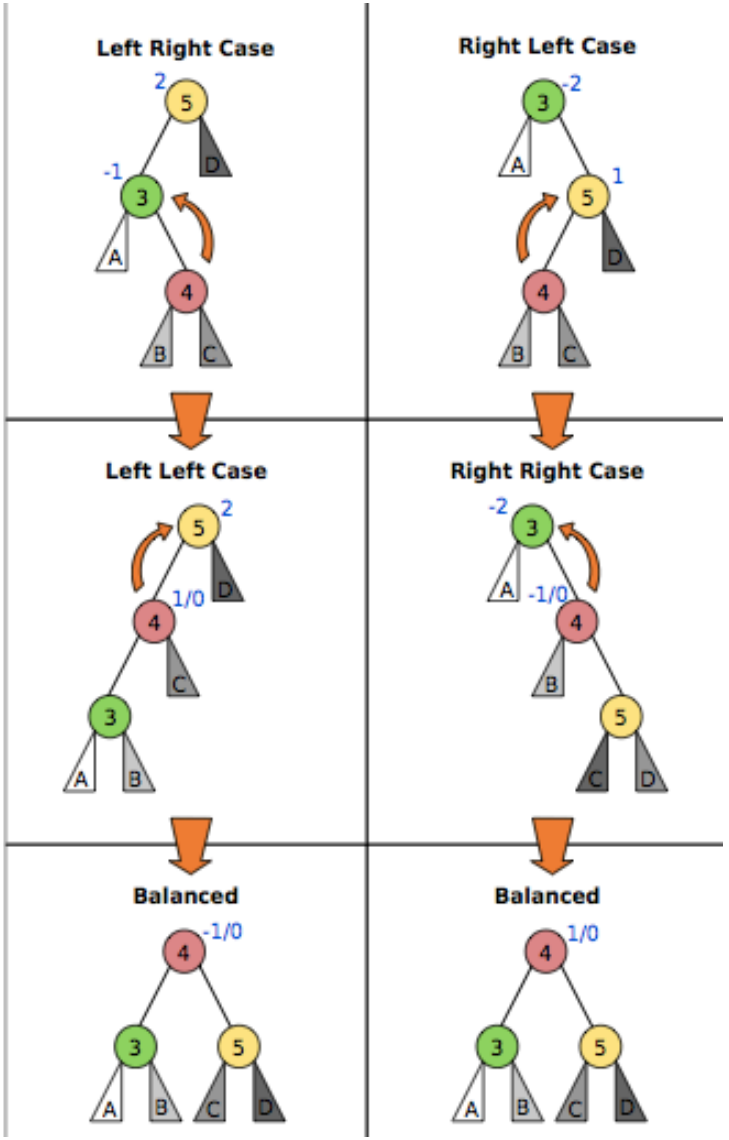
Case 1: right-left tree



Balanced BSTs: AVL

The four cases

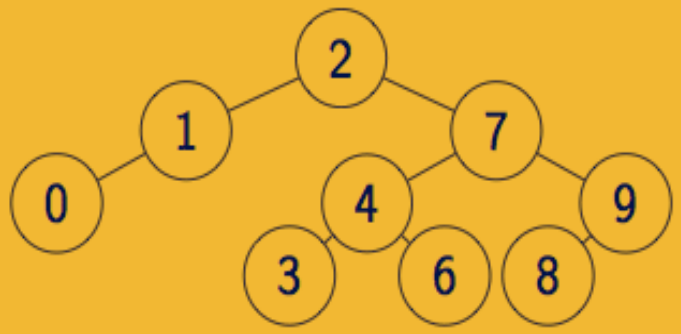
(picture from Wikipedia)



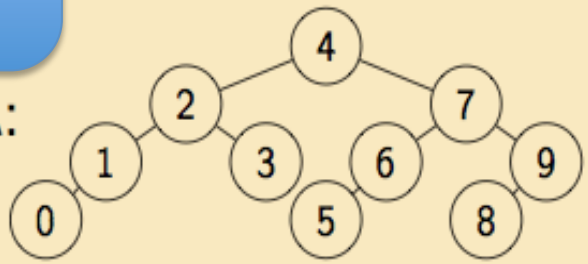
Question (from last year)

govote.at
Code 635112

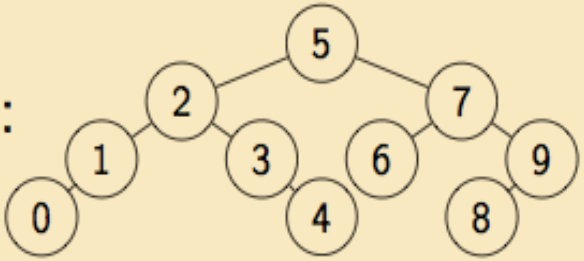
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



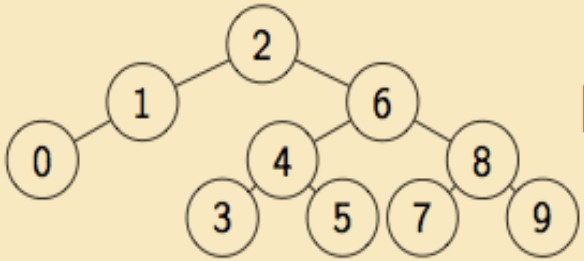
A:



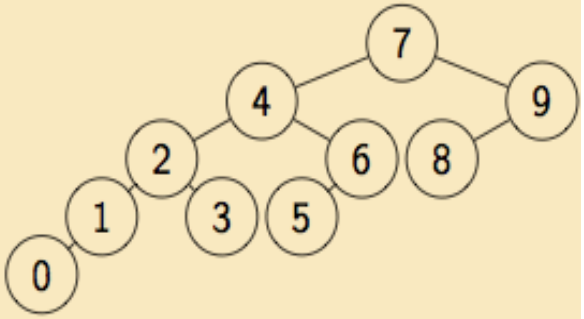
B:



C:



D:



Answer

Slightly tweaked (every node info +1)

- Go to

http://visualgo.net/bst.html

- Choose AVL
- Create Empty
- Insert 1,2,3,8,10,4,5,7,9 – equivalent tree
- Insert 6 ➔ Result

AVL trees

- A balanced BST that maintains balance by rotating the tree
- **Insertion**: insert as in a BST and move upwards from the inserted node, rotating unbalanced nodes
- **deletion** (in book if you're interested): delete as in a BST and move upwards from the node that disappeared, rotating unbalanced nodes
- **Worst-case** (it turns out) 1.44log n, typical log n comparisons for any operation – very balanced. This means lookups are quick.
- Insertion and deletion can be slower than in a naïve BST, because you have to do a bit of work to repair the invariant

Balanced BSTs: Red-Black Trees

- A **red-black tree** is a balanced BST
- It has a more complicated invariant than an AVL tree:

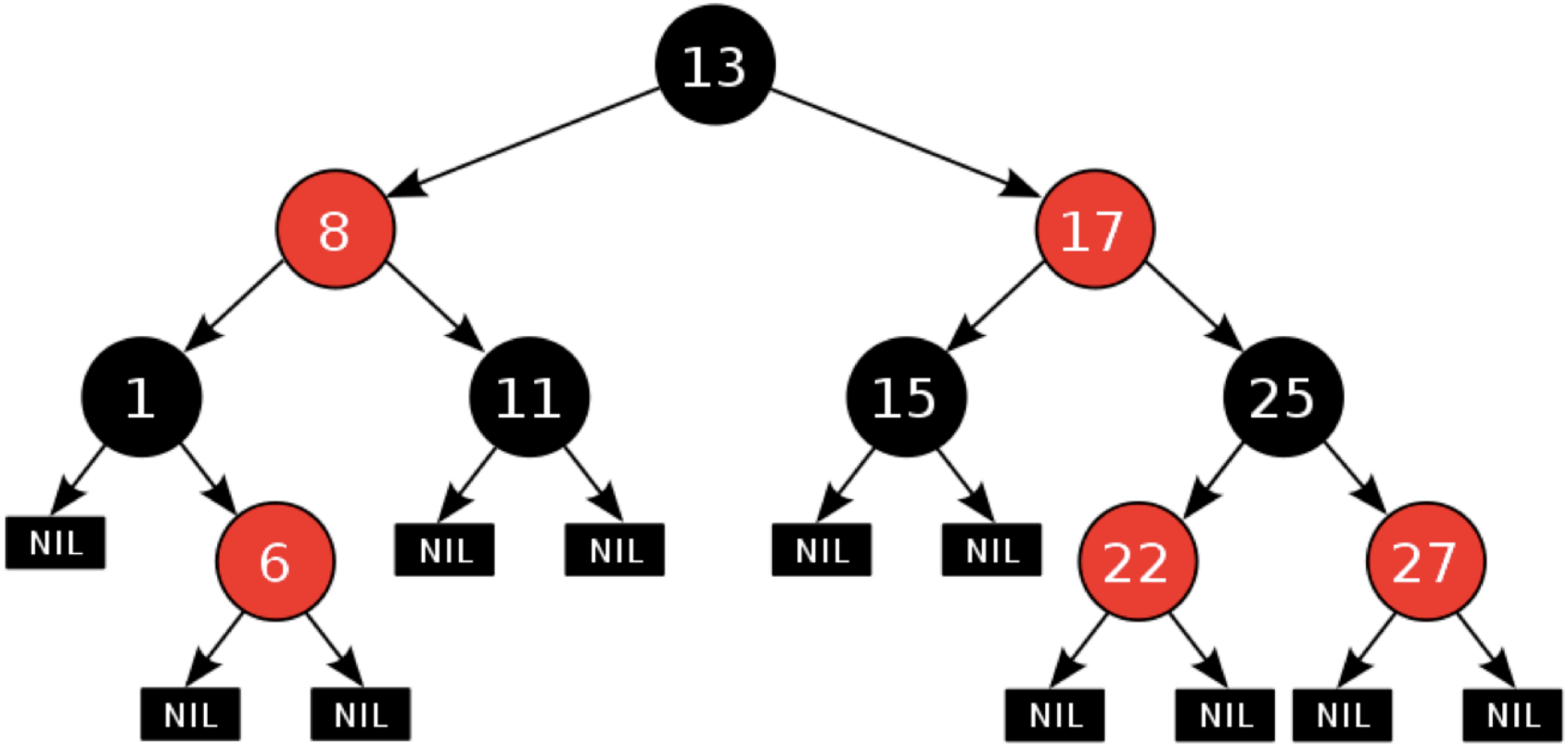
  Each node is coloured red or **black**

  A red node cannot have a red child

  In any path from the root to a null, the number of **black** nodes is the same

  (The root node is **black**)
  Implicitly, a **null** is coloured **black**

Balanced BSTs: Red-Black Trees



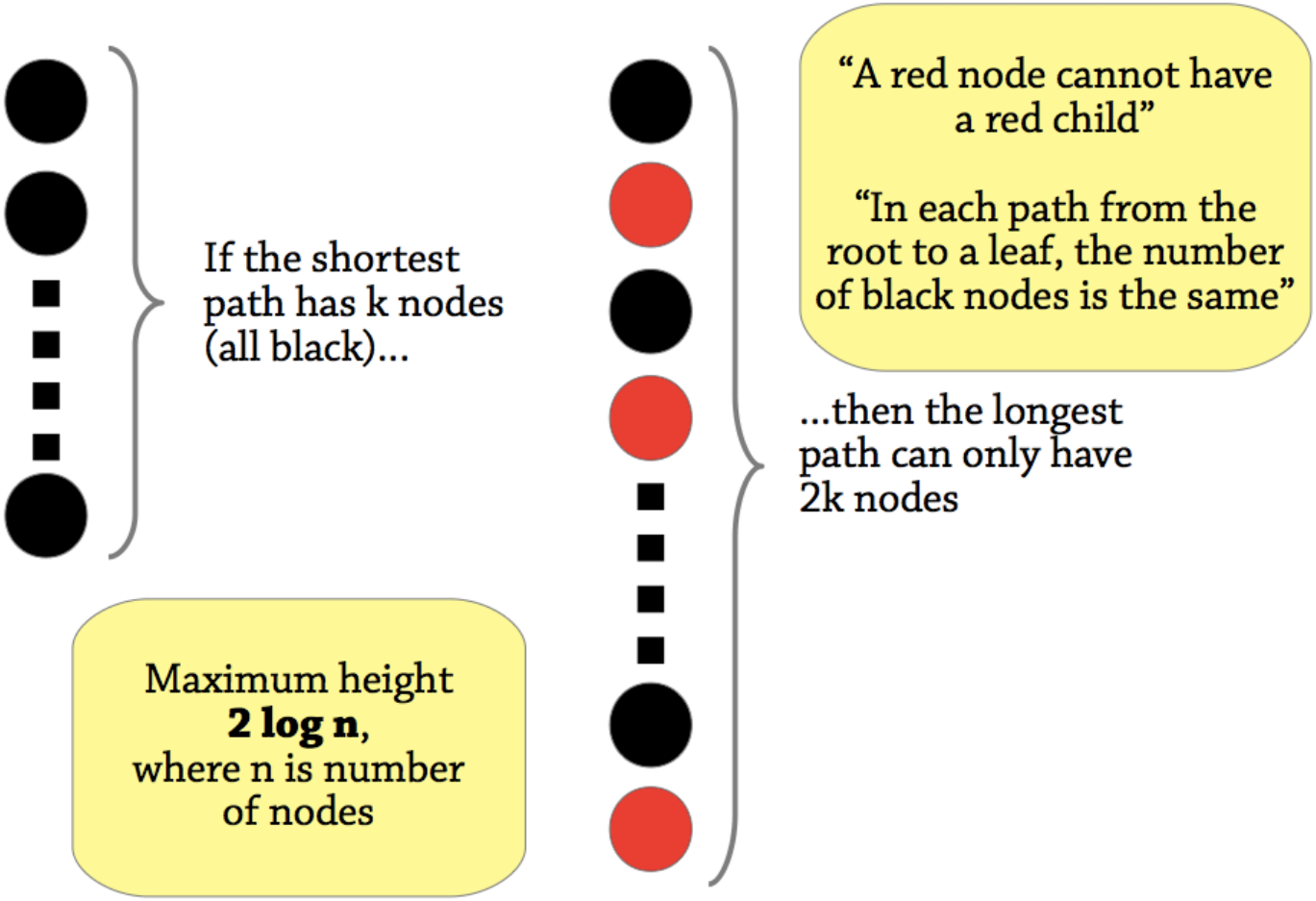
Question

What is the maximum amount of red nodes that a Red-Black tree with 7 non-null nodes and a black root could have ?

1. 2
2. 3
3. 4
4. 5

govote.at
Code 128448

Balanced BSTs: Red-Black Trees



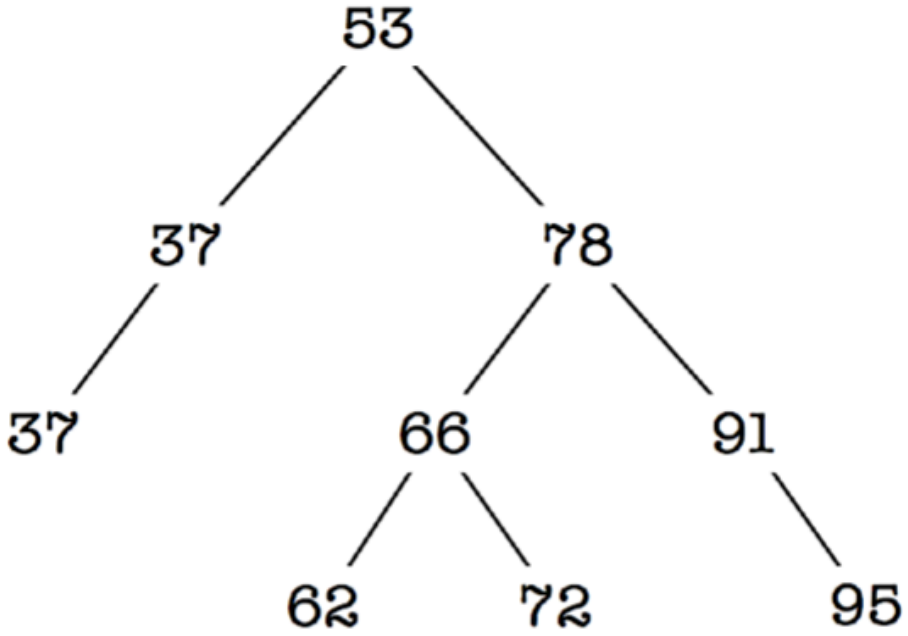
Question (from re-exam August)

After colouring the following tree as a Red-Black tree with black root, how many red nodes will we have ?

1. 5 2. 6

3. 4 4. 3

govote.at
Code 989807



Balanced BSTs: Red-Black Trees

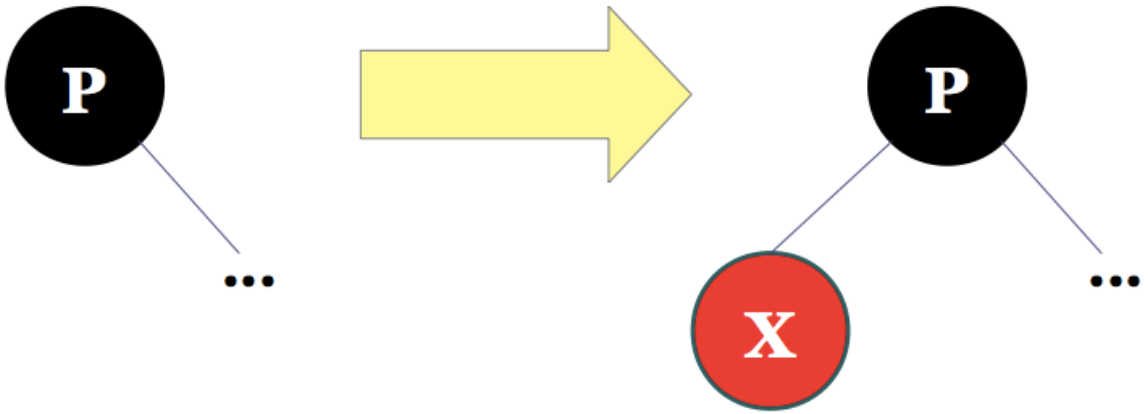
- In AVL trees, we maintained the invariant by *rotating* parts of the tree
- In red-black trees, we use two operations:

  rotations

  recolouring: changing a red node to black or vice versa
- Recolouring is an "administrative" operation that doesn't change the structure or contents of the tree

Balanced BSTs: Red-Black Trees

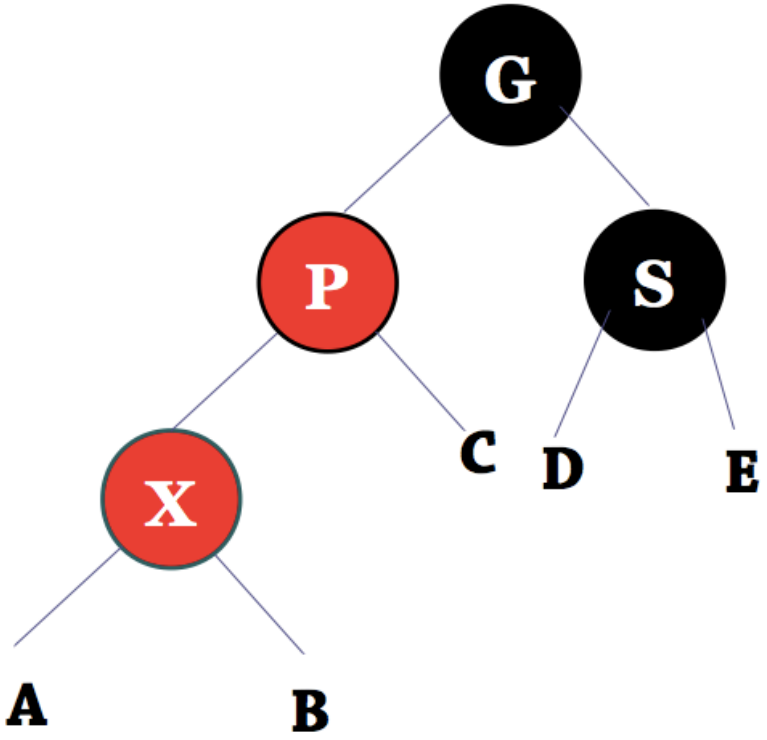
- First, add the new node as in a BST, making it red
- If the new node's parent is black, everything's fine



Balanced BSTs: Red-Black Trees

- If the parent of the new node is red, we have broken the invariant. (How?) We need to repair it.
- We need to consider several cases.
- In all cases, since the parent node is red, the grandparent is black. (Why?)
- Let's take the case where the parent's sibling is black.

Balanced BSTs: Red-Black Trees



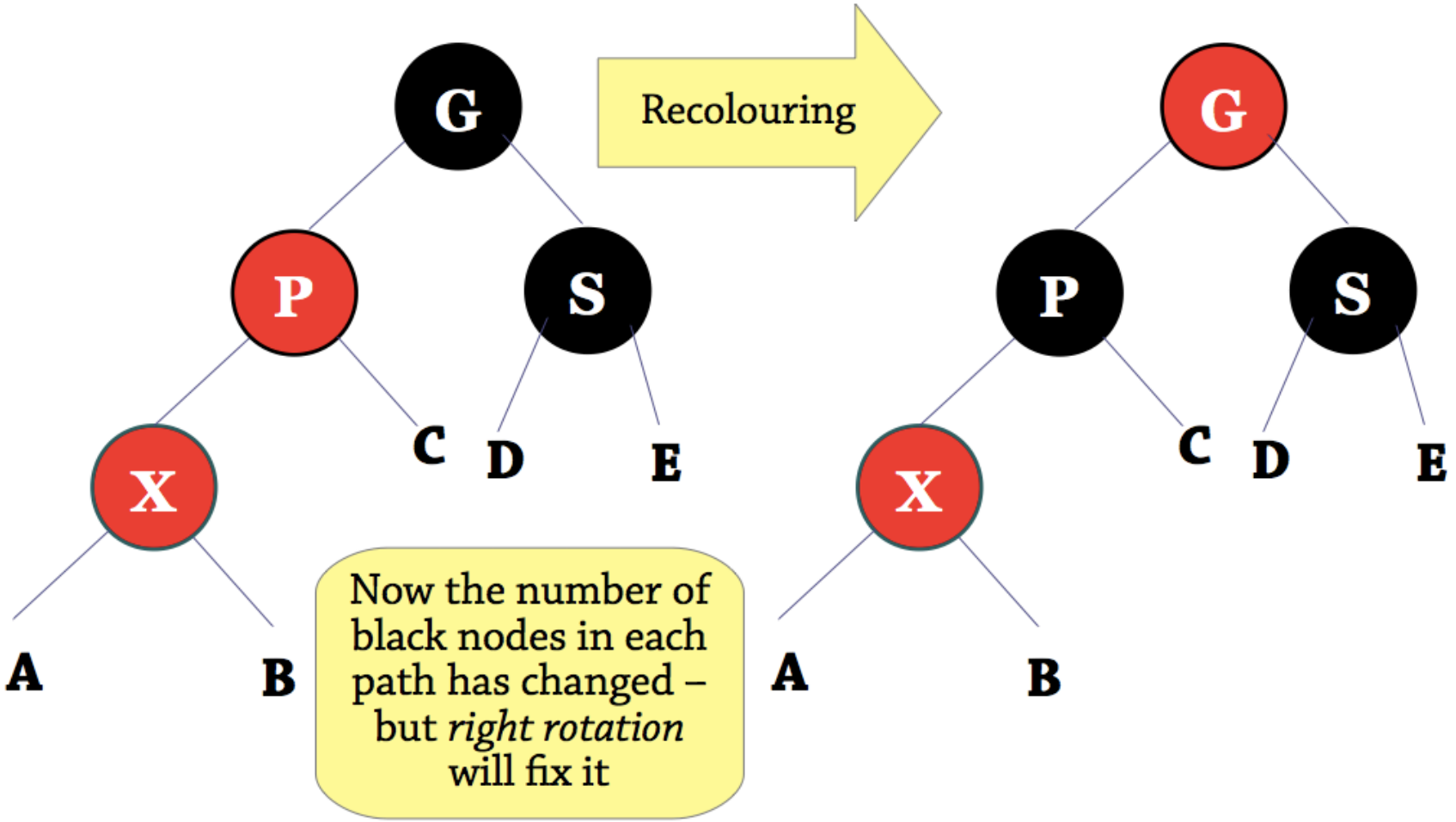
X: Newly-inserted node, breaks invariant

P: **P**arent of new node

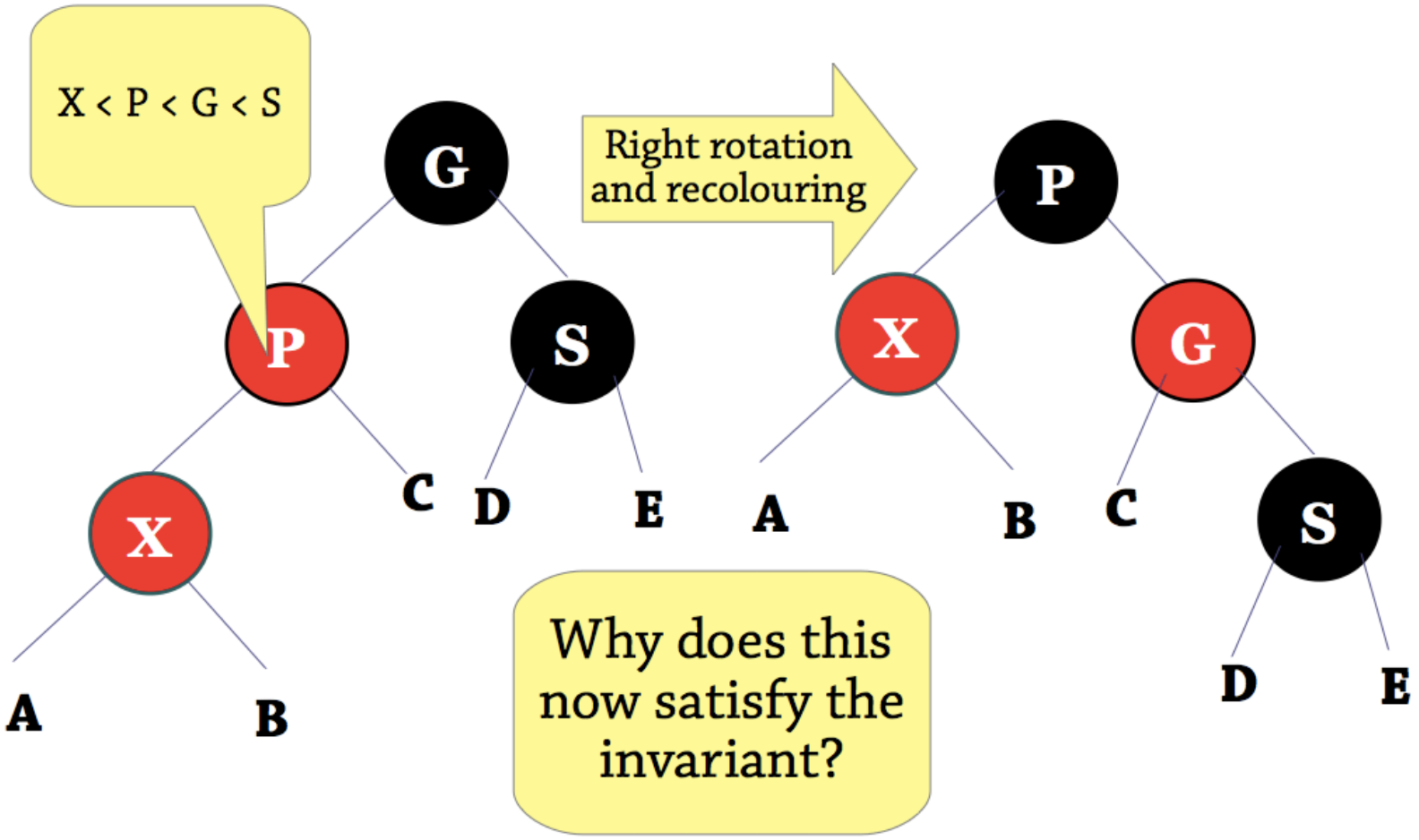
G: **G**randparent of new node

S: **S**ibling of parent

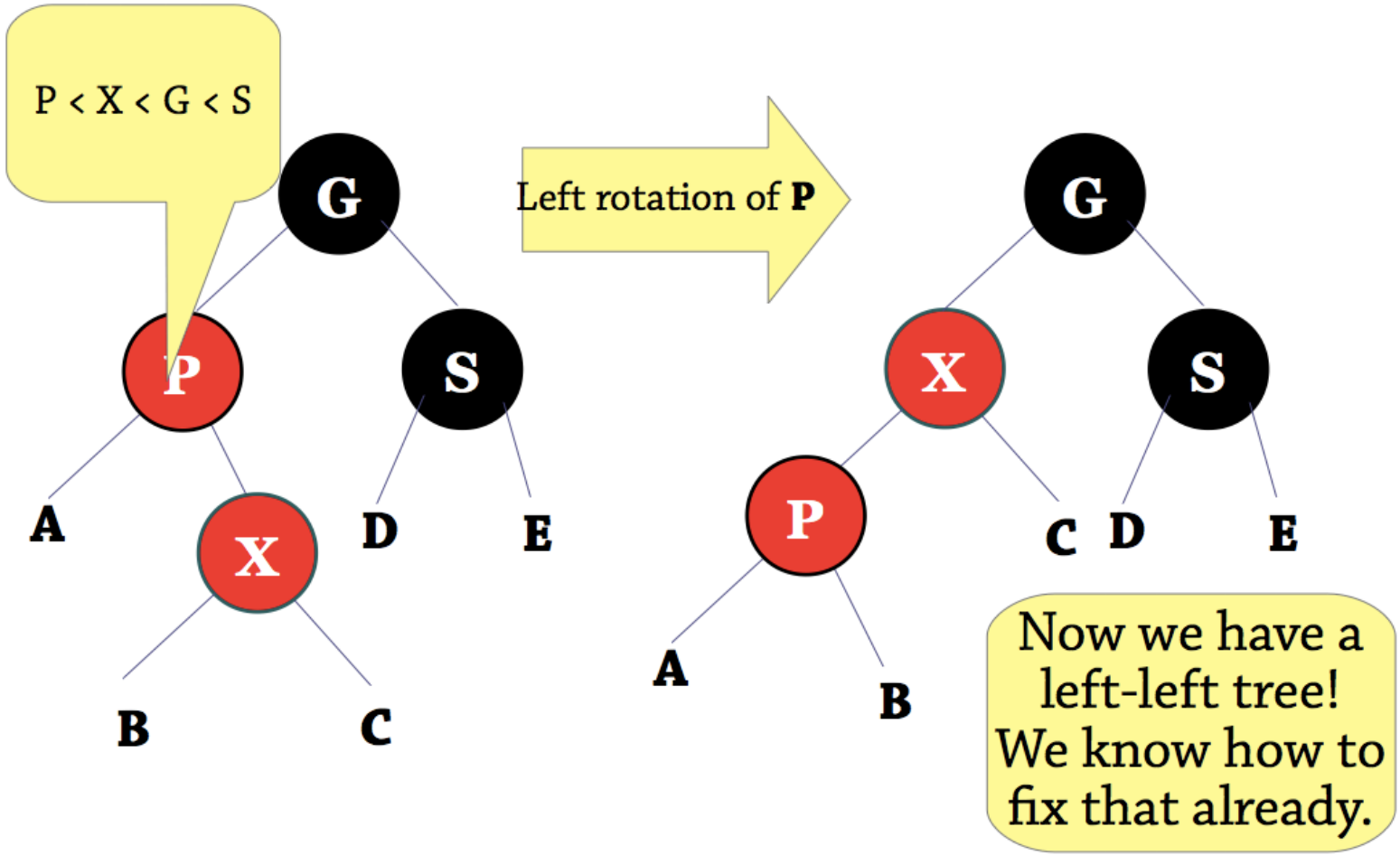
Balanced BSTs: Red-Black Trees



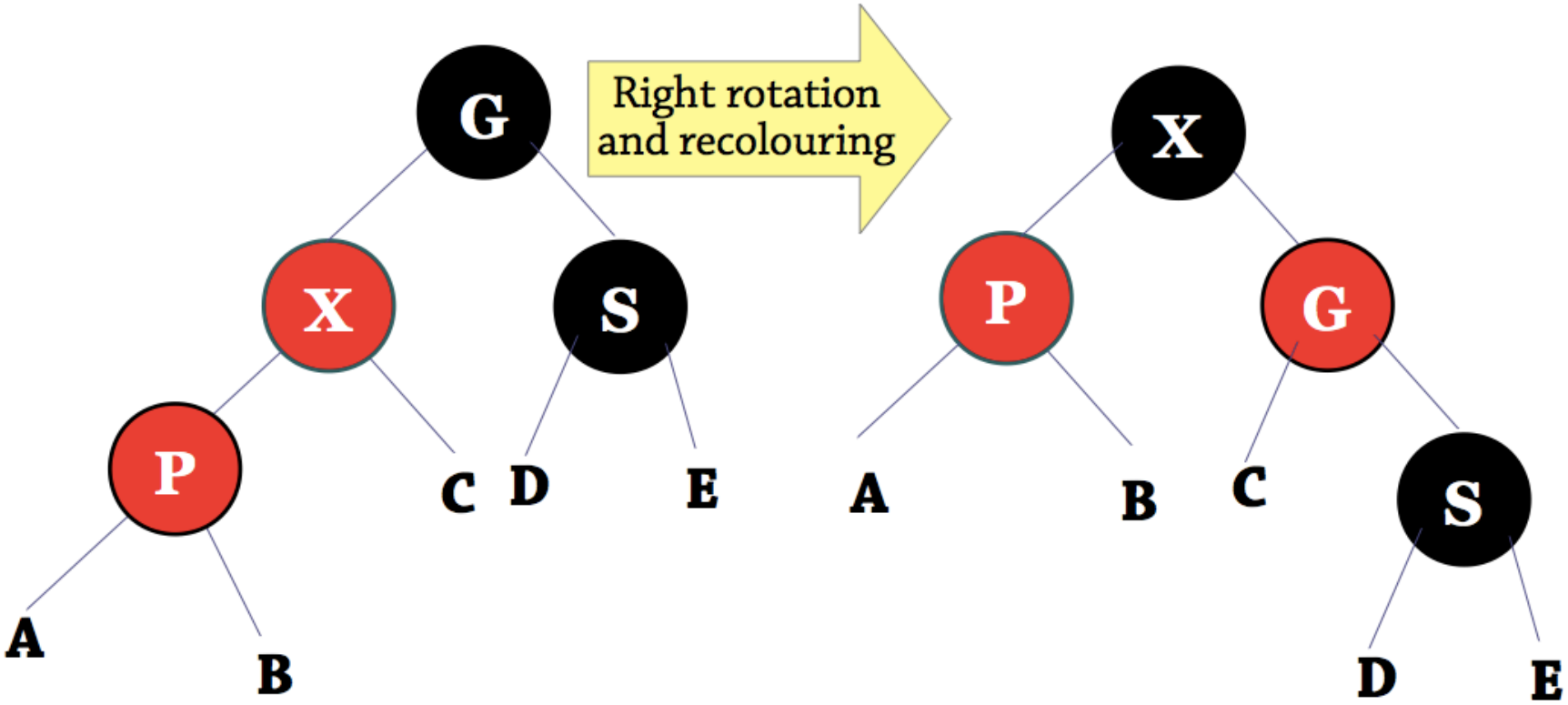
Balanced BSTs: Red-Black Trees



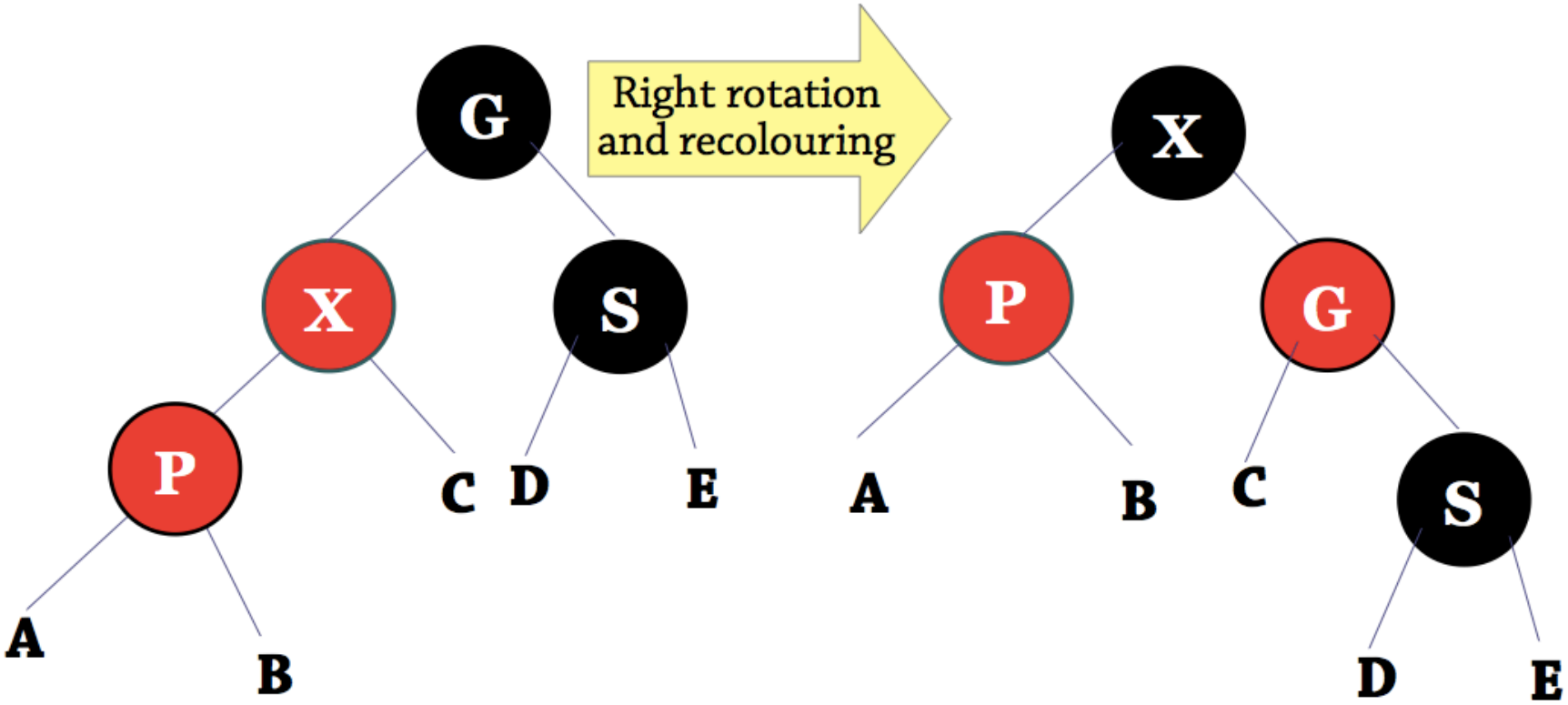
Balanced BSTs: Red-Black Trees



Balanced BSTs: Red-Black Trees



Balanced BSTs: Red-Black Trees



Balanced BSTs: Red-Black Trees

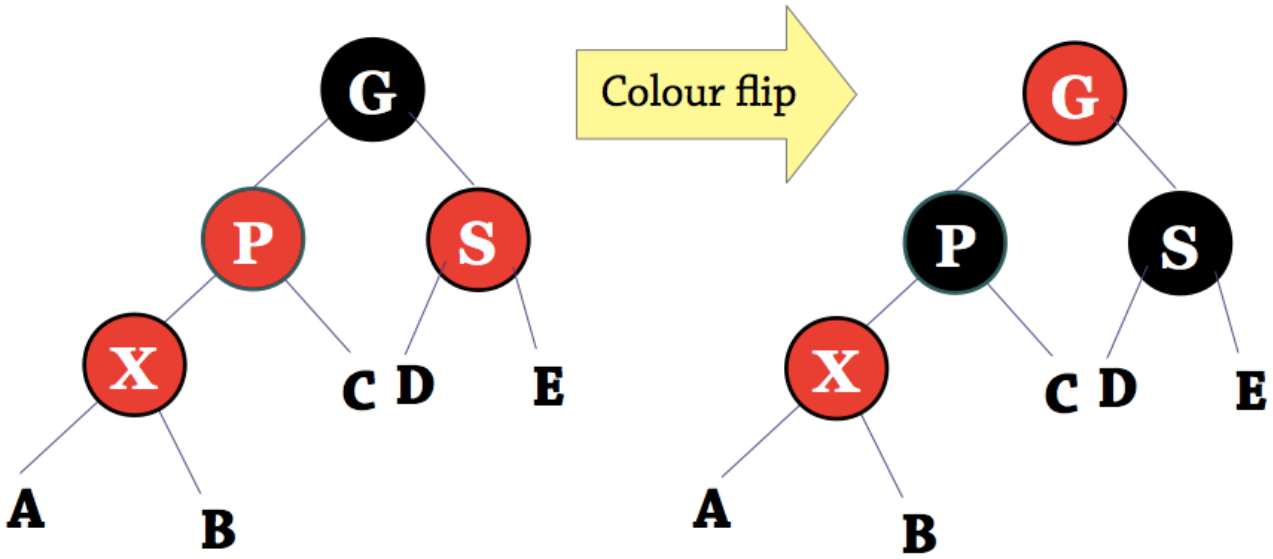
- Insert the new node as in a BST, make it red
- **Problem**: if the parent is red, the invariant is broken (red node with red child)
- To fix a red node with a red child:
- If the node has a **black** sibling, rotate and recolour
- If the node has a red sibling, ...? Two approaches, bottom-up (simpler) and top-down (more efficient)

Balanced BSTs: Red-Black Trees

- Bottom-Up Insertion

  If a new node, its parent and its parent's sibling are all red: do a colour flip

  Make the parent and its sibling black, and the grandparent red



Balanced BSTs: Red-Black Trees

- A *colour flip* almost restores the invariant...
- ...but if G has a red parent, we will have a red node with a red child
- So move up the tree to G and apply the same double-red repair process there as we did to X.

Balanced BSTs: Red-Black Trees

- Insert the new node as in a BST, make it red If the new node has a red parent P:

  - If the parent's sibling S is black, use rotations and recolourings to fix it – the rotations are the same as in an AVL tree

  - If S is red, do a colour flip, which makes the grandparent G red – so you need to do the same double-red repair to G if its parent is red

  - Lastly: if you get to the root and the root is red, make it black

Balanced BSTs: Red-Black Trees

- Top-down approach

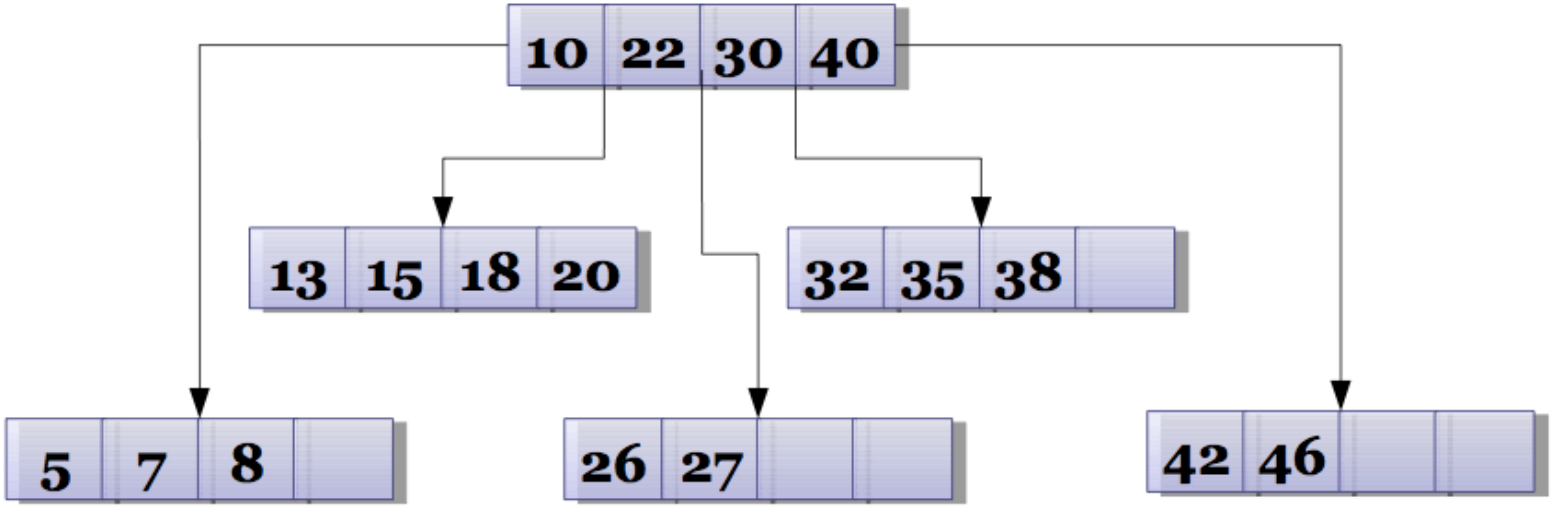
- Extra reading 12.2.2
- Not in exam!

Red-Black vs AVL

- Red-black trees have a weaker invariant than AVL trees (less balanced) – but still O(log n) running time
- **Advantage**: less work to maintain the invariant (top-down insertion – no need to go up tree afterwards), so insertion and deletion are cheaper
- **Disadvantage**: lookup will be slower if the tree is less balanced
- But in practice red-black trees are **faster** than AVL trees

B-trees

- In a **B-tree** of order k, a node can have k children
- Each non-root node must be at least half-full



B-trees

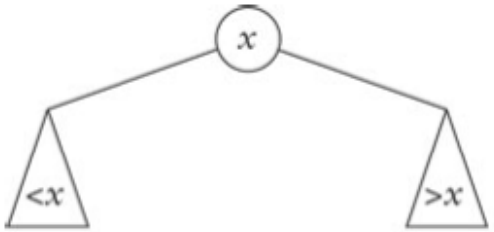
- B-trees are used for disk storage in databases:
- Hard drives read data in blocks of typically ~4KB
  For good performance, you want to minimise the number of blocks read
  This means you want: 1 tree node = 1 block
- B-trees with k about 1024 achieve this

B-trees

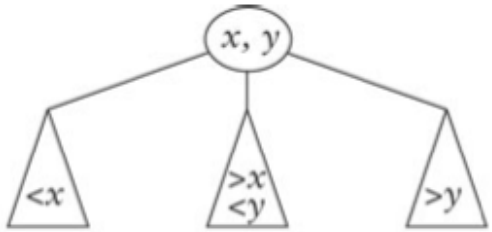
- B-trees are used for disk storage in databases:
- Hard drives read data in blocks of typically ~4KB
  For good performance, you want to minimise the number of blocks read
  This means you want: 1 tree node = 1 block
- B-trees with k about 1024 achieve this

B-trees

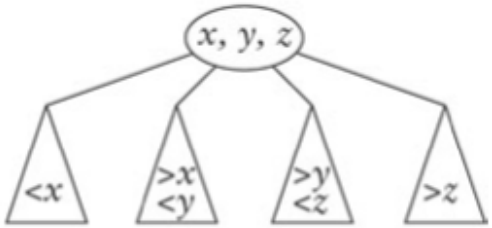
- Examples of nodes from a B-tree



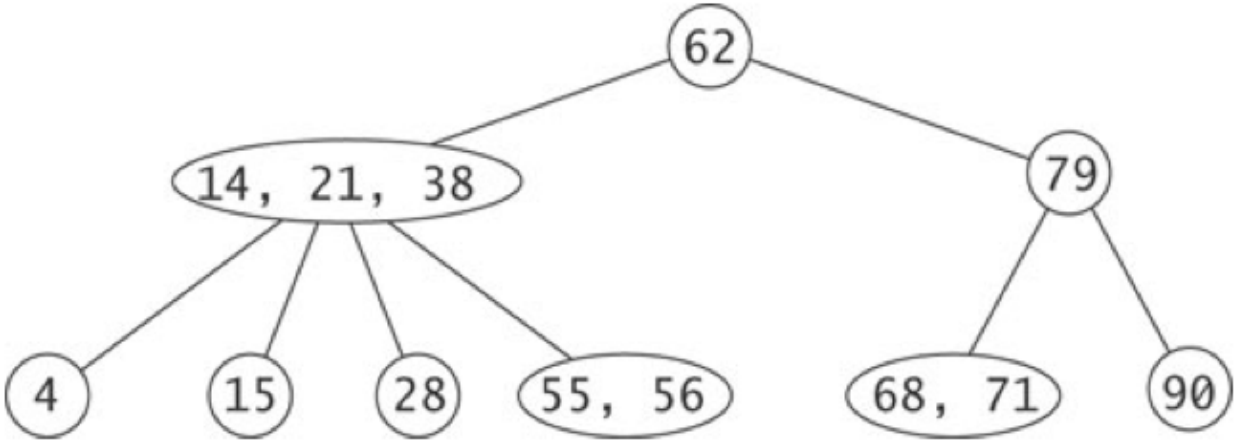
2-node



3-node

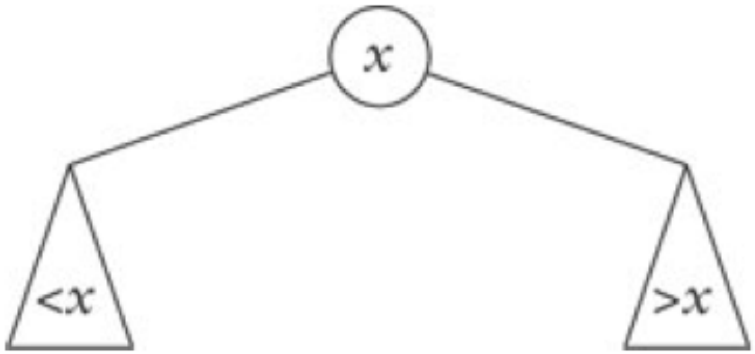


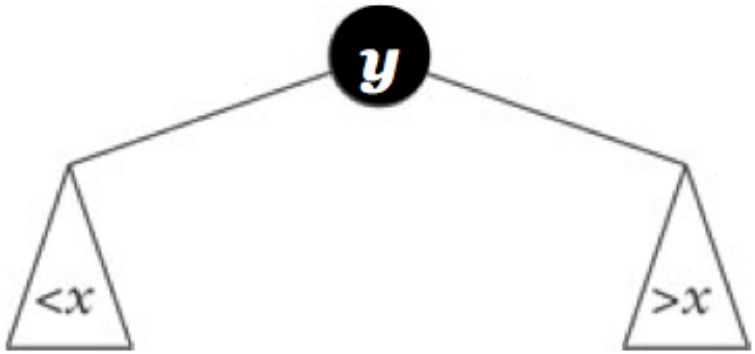
4-node



Red-Black trees are B-trees!

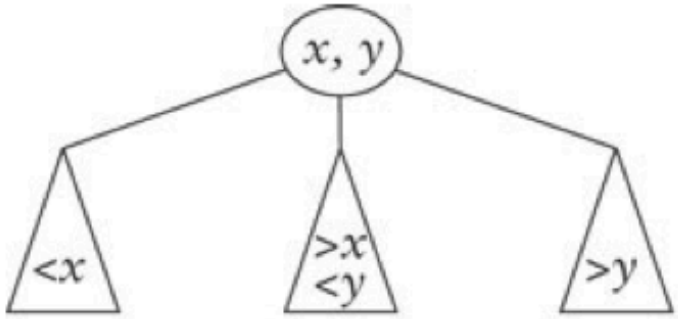
- A 2-node is a black node

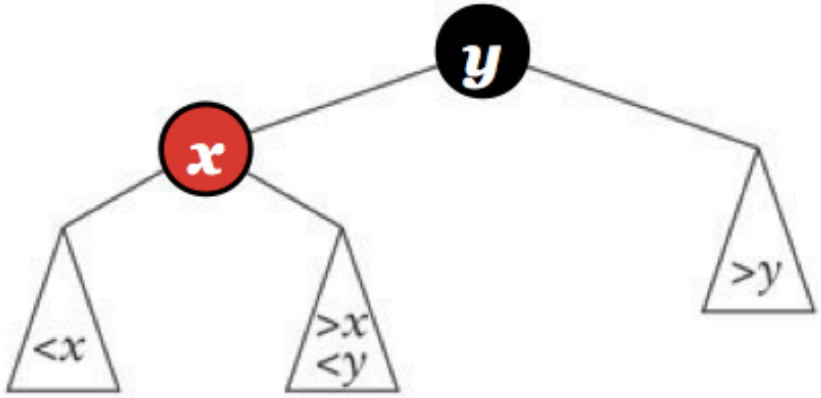


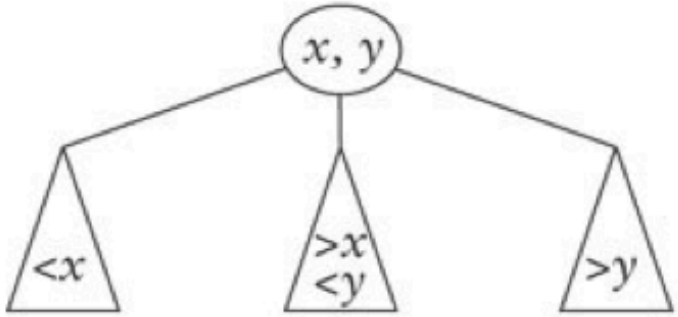


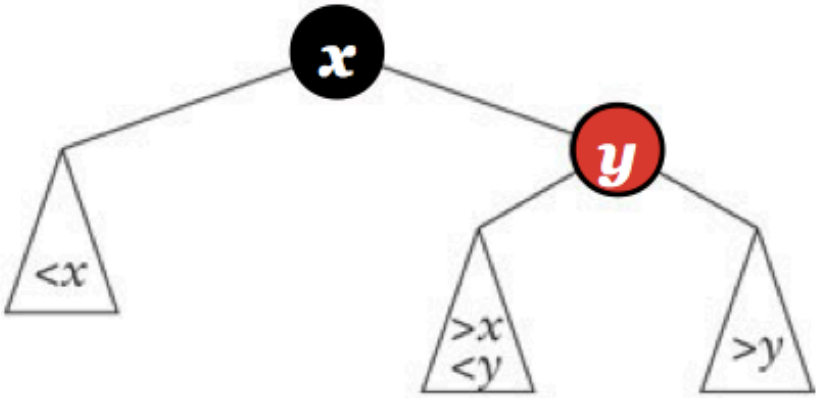
Red-Black trees are B-trees!

- A 3-node is a black node with one red child



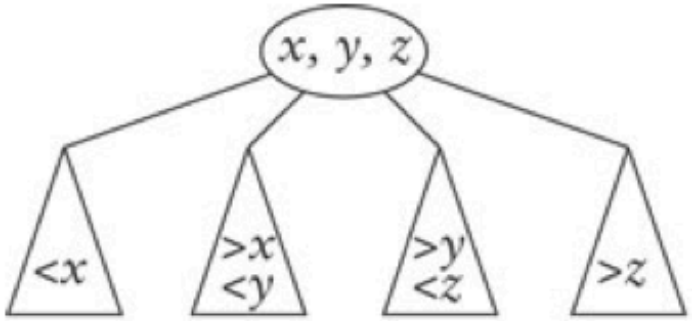


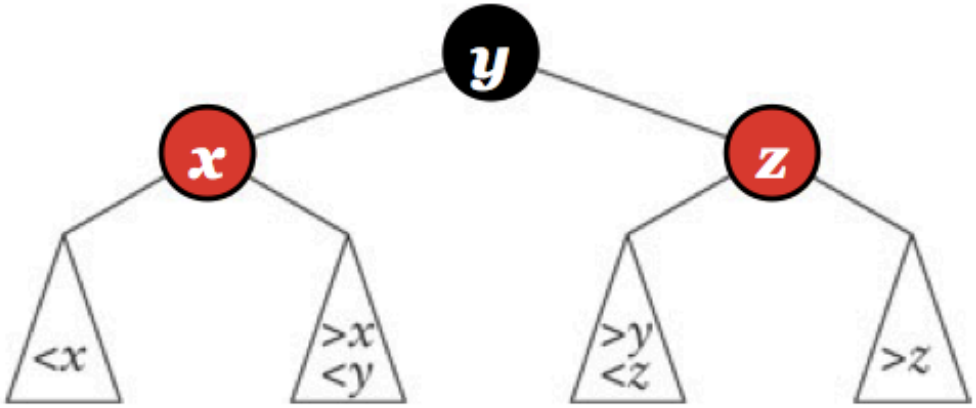




Red-Black trees are B-trees!

- A 4-node is a black node with two red children





To Do

Read from the book:

+ 4.4, 4.5, 4.7, 12.2
+ more details: Cormen and Wikipedia

Extra:

2,3-trees – the link between B-trees and AVL

Practice AVLs here:

visualalgo.net

Coming up:

+ more on sorting (lecture by Andreas)