Red-black trees *(19.5)*,
B-trees *(19.8)*,
2-3-4 trees

Red-black trees

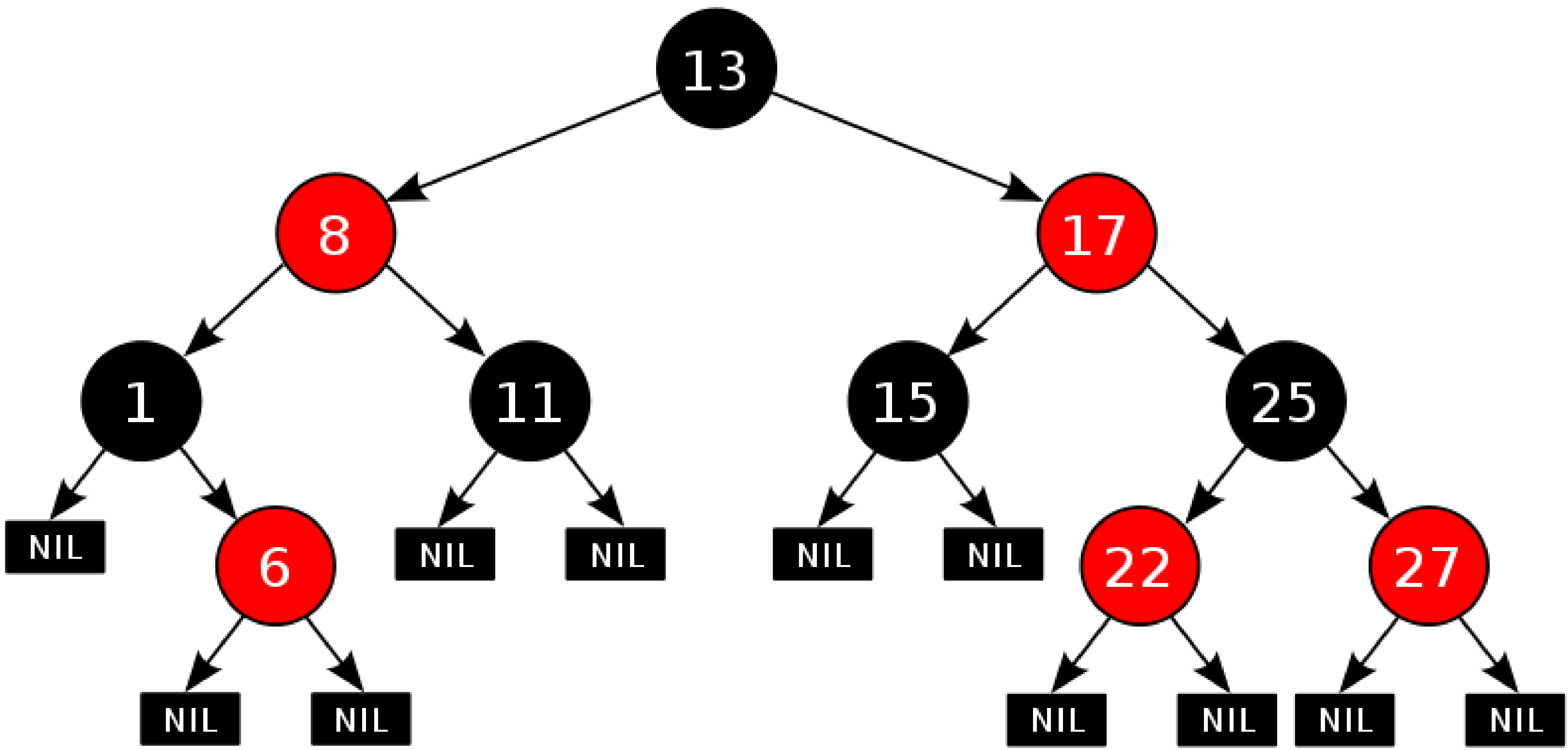
A red-black tree is a balanced BST

It has a more complicated invariant than an AVL tree:

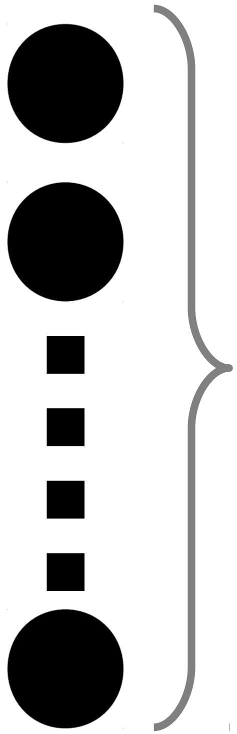
- Each node is coloured red or black
- A red node cannot have a red child
- In any path from the root to a null, the number of black nodes is the same
- The root node is black

Implicitly, a *null* is coloured black

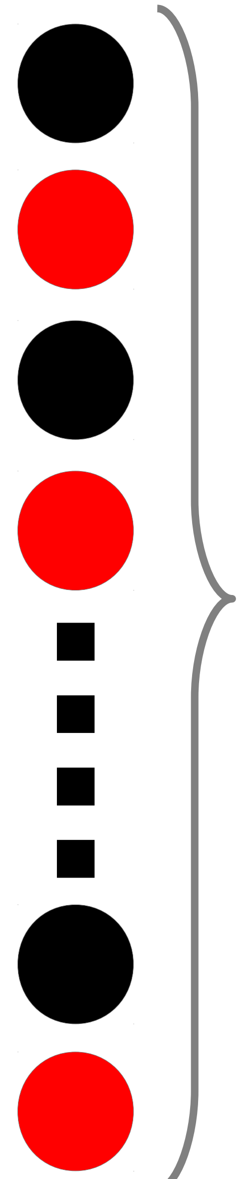
A red-black tree



Red-black trees – invariant



If the shortest path has k nodes (all black)…



"A red node cannot have a red child"

"In each path from the root to a leaf, the number of black nodes is the same"

…then the longest path can only have 2k nodes

Maximum height
2 log n,
where n is number of nodes

Maintaining the red-black invariant

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

AVL versus red-black trees

To insert a value into an AVL tree:

- go *down* the tree to find the parent of the new node
- insert a new node as a child
- go *up* the three, rebalancing

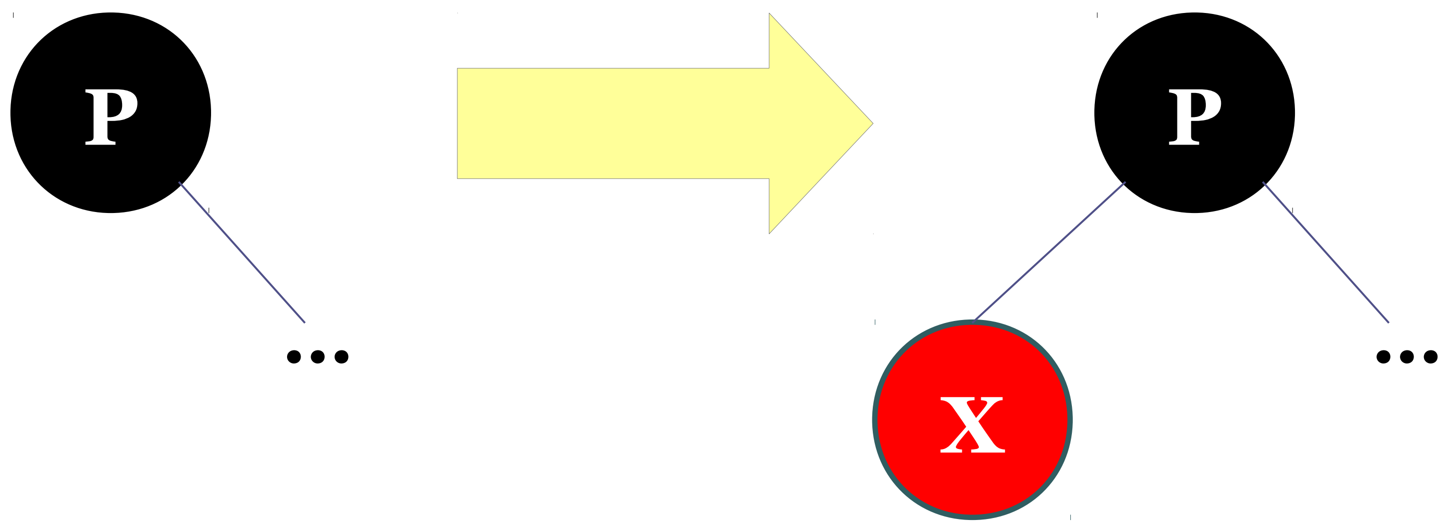
...so two passes of the tree (down and up) required in the worst case

In a red-black tree:

- go *down* the tree to find the parent of the new node...
- ...but rebalance and recolour the tree as you go down
- after inserting, no need to go up the tree again

Red-black insertion

First, add the new node as in a BST, making it *red*



If the new node's parent is black, everything's fine

Red-black insertion

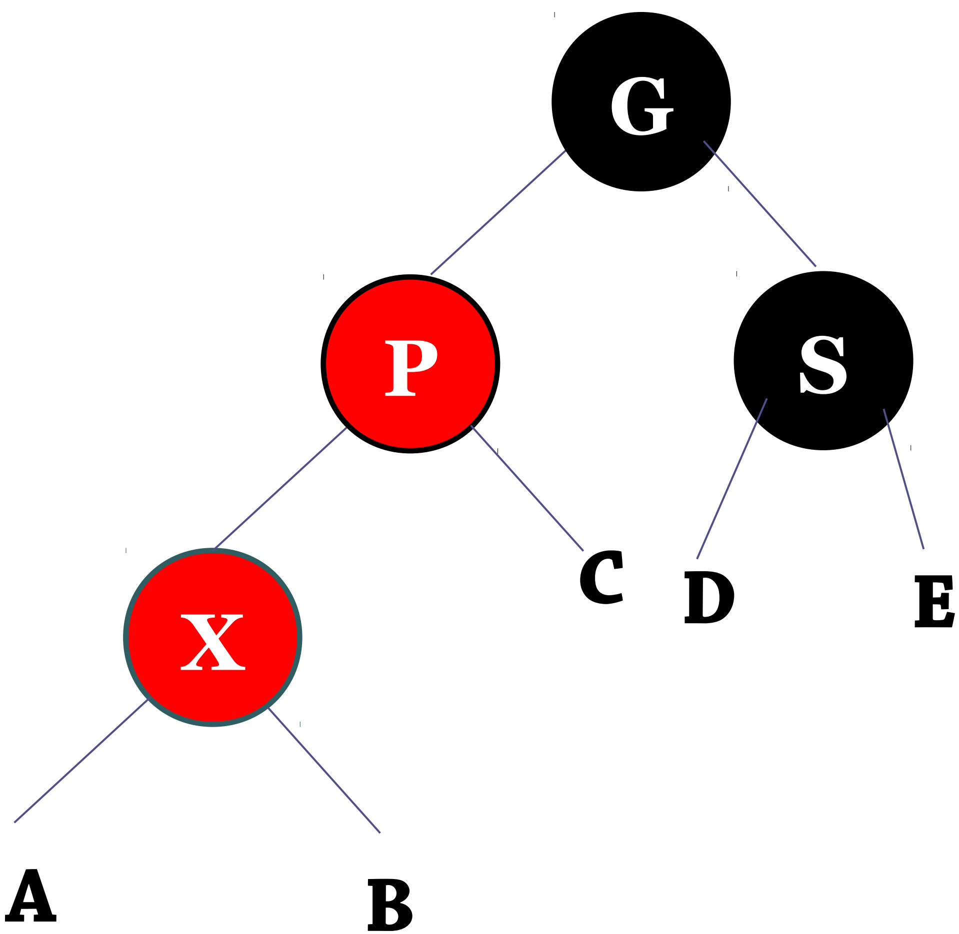
If the parent of the new node is red, we have broken the invariant. (How?) We need to repair it.

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.

Left-left tree (“outside grandchild”)



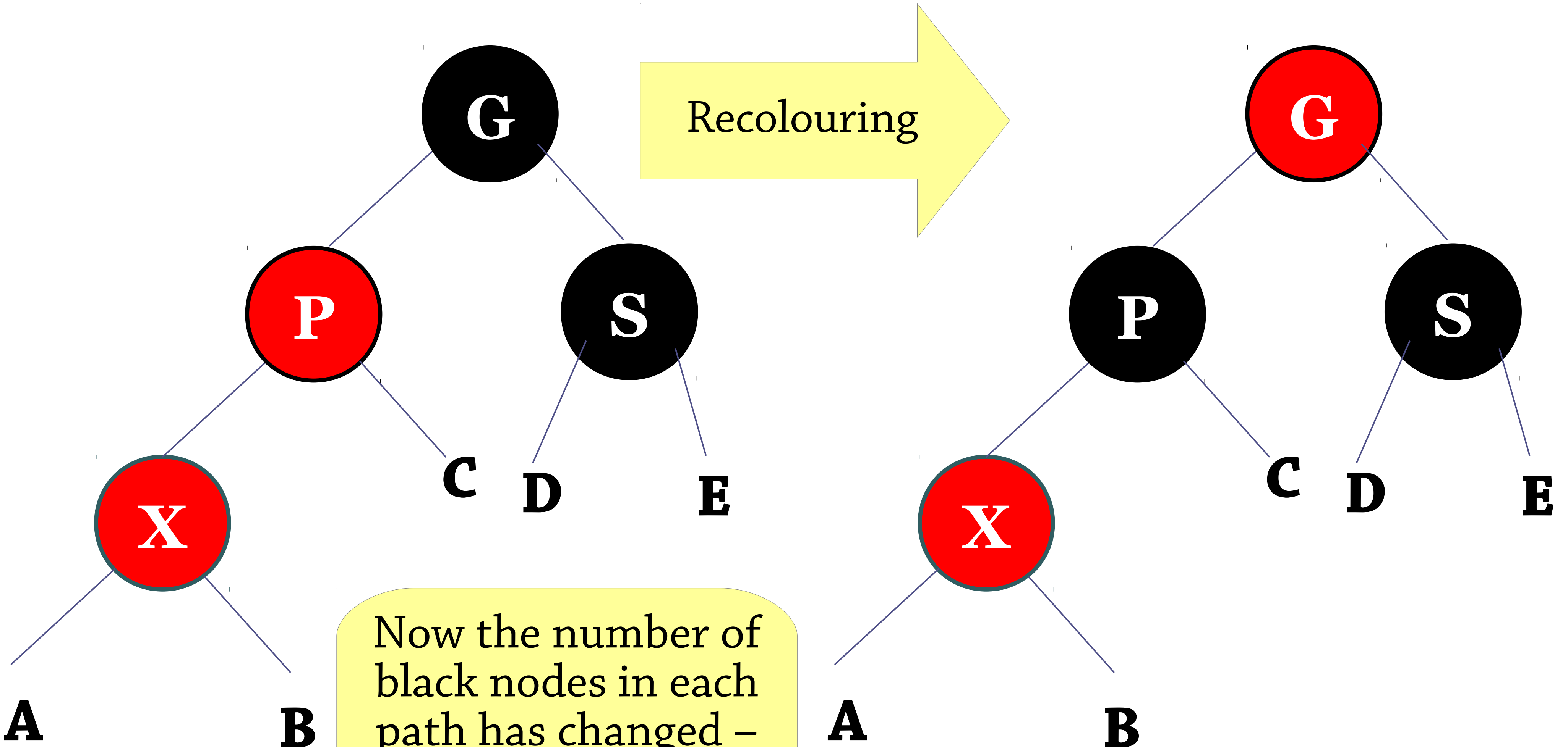
X: Newly-inserted node, breaks invariant

P: **P**arent of new node

G: **G**randparent of new node

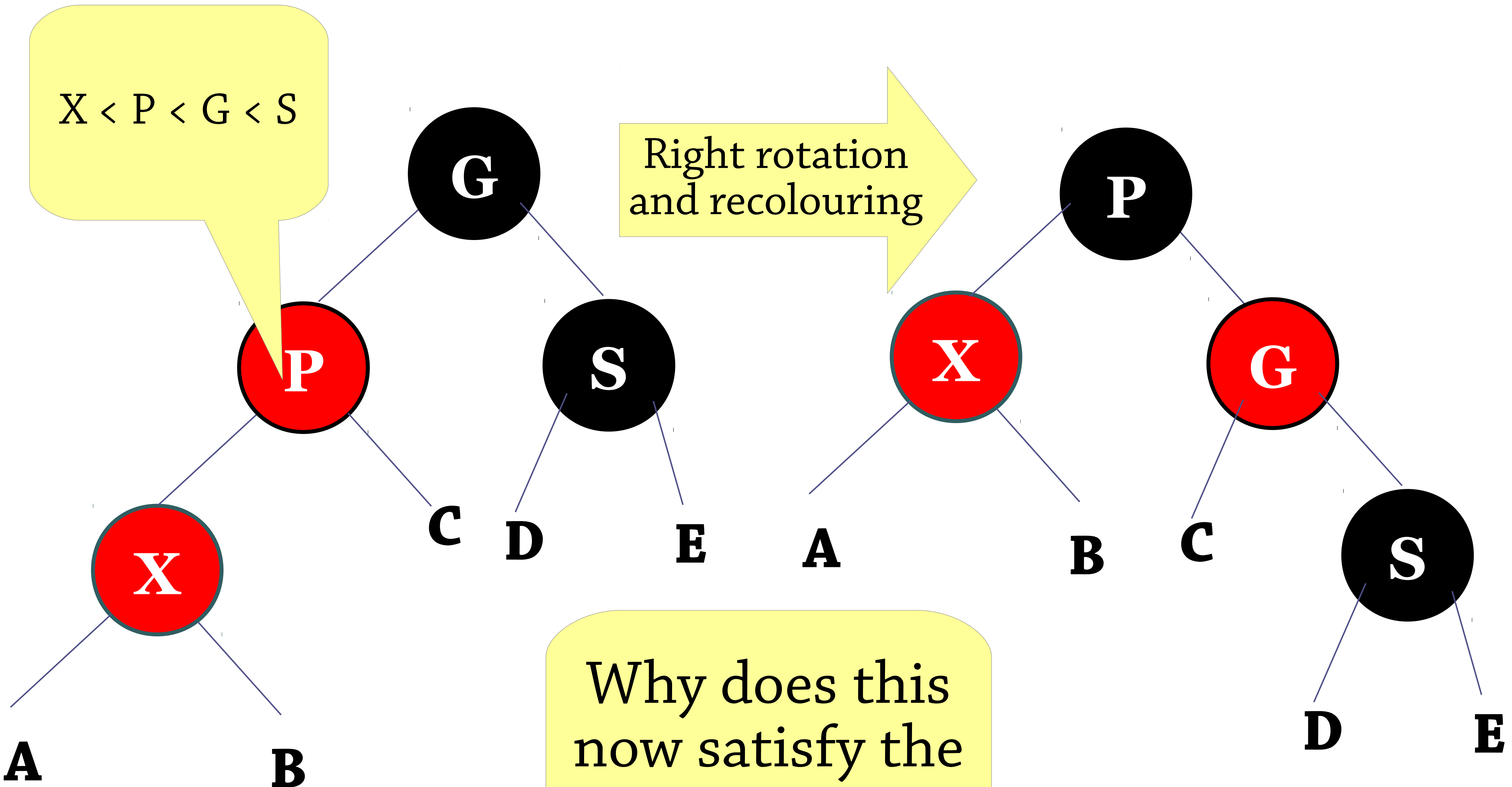
S: **S**ibling of parent

Left-left tree ("outside grandchild")



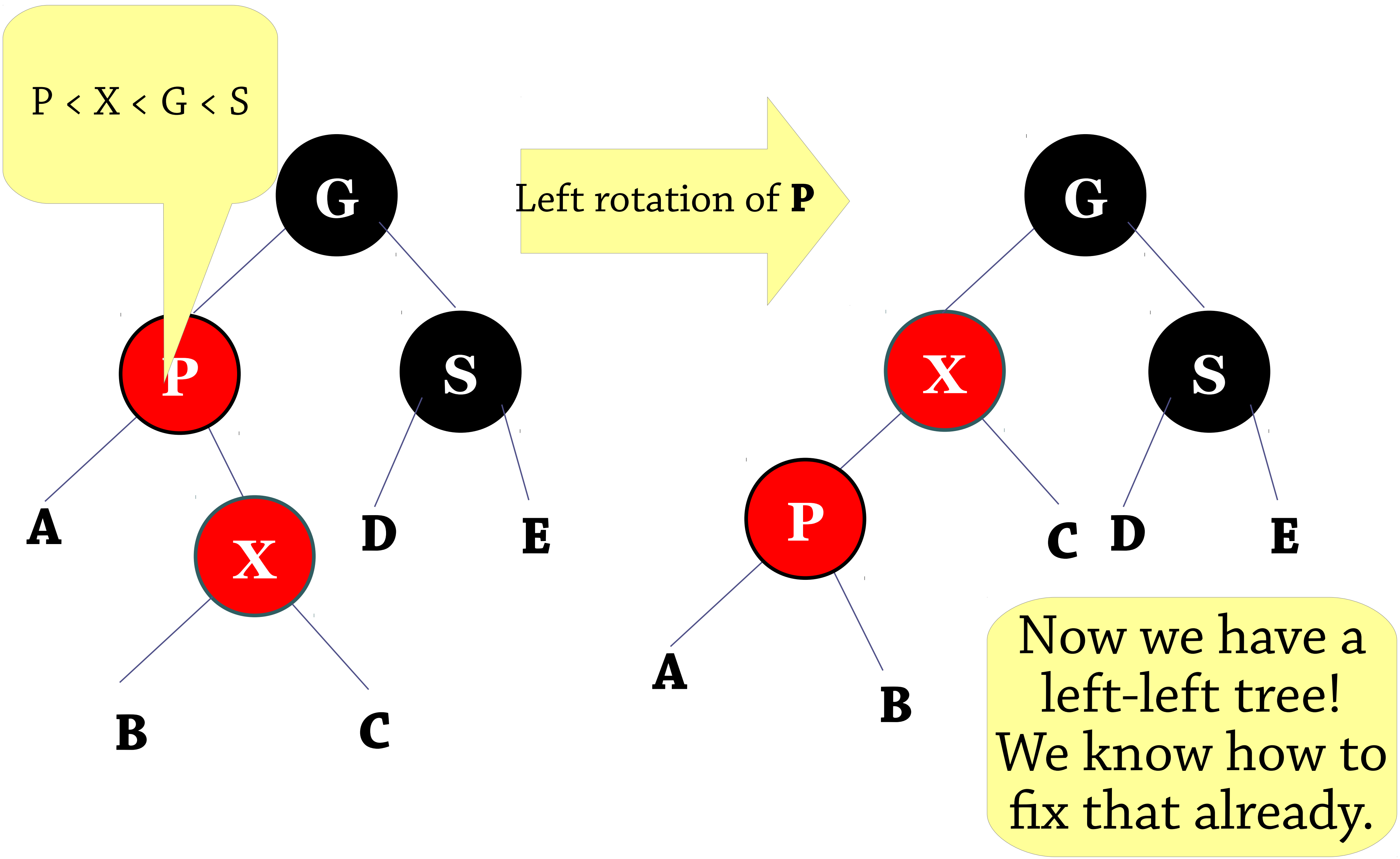
Now the number of black nodes in each path has changed – but *right rotation* will fix it

Left-left tree ("outside grandchild")

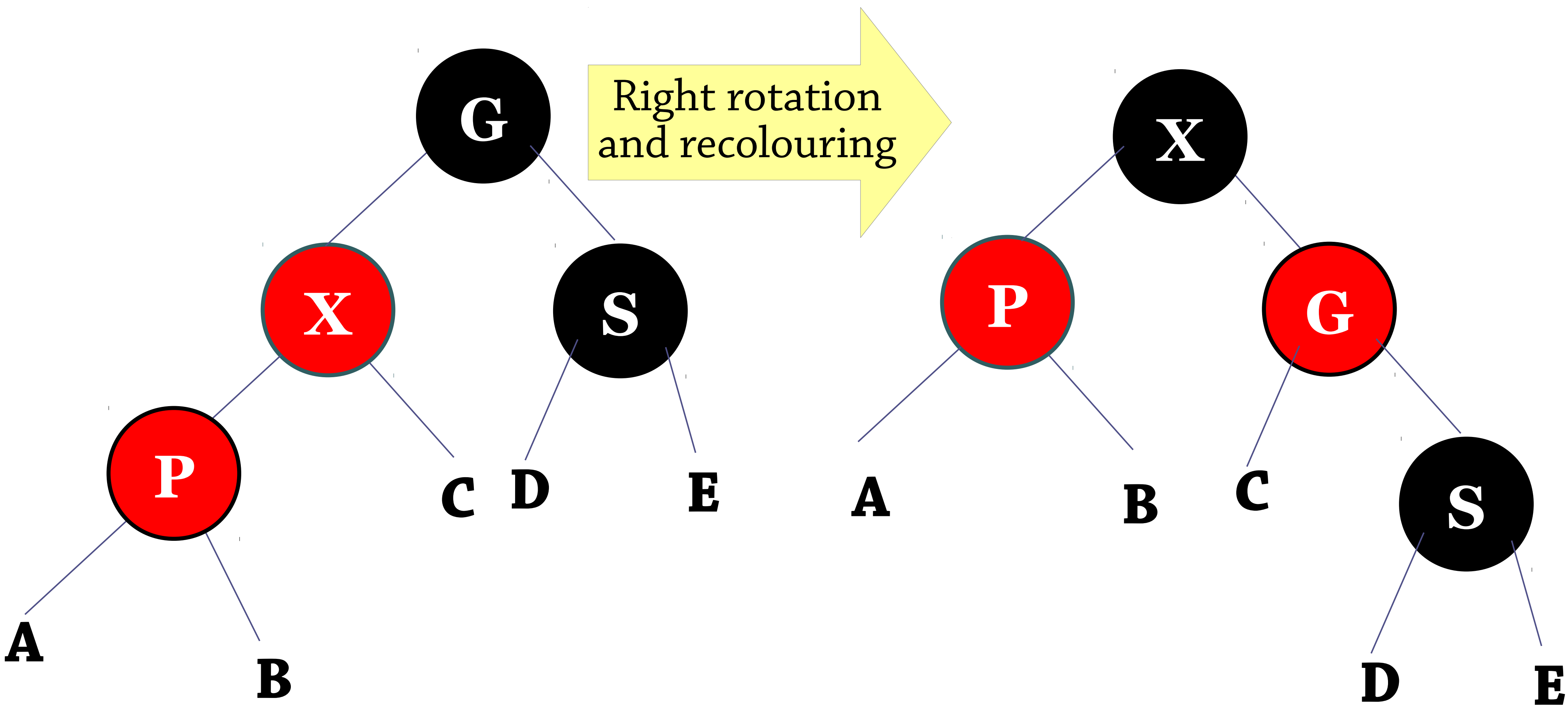


Why does this now satisfy the invariant?

Left-right tree ("inside grandchild")



Left-right tree ("inside grandchild")



Summary so far

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)

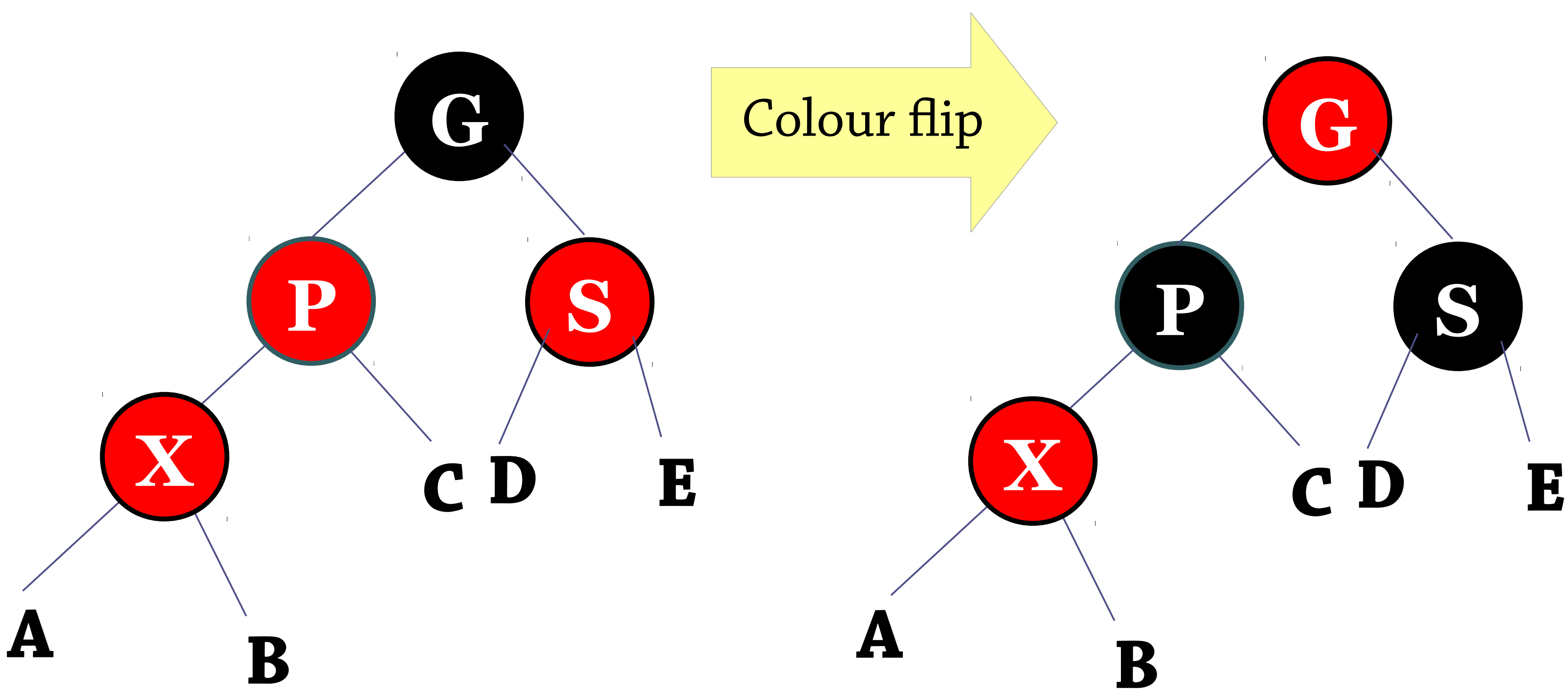
To fix a red node with a red child:

- 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)

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



Bottom-up insertion

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**.

Bottom-up insertion

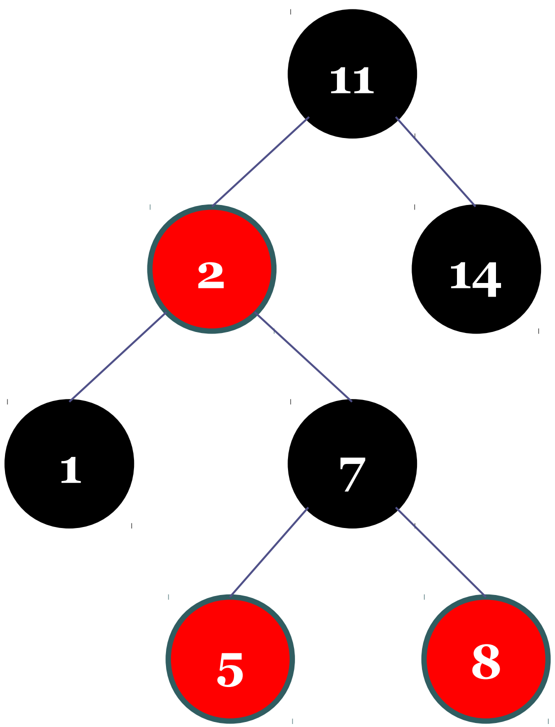
Insert the new node as in a BST, make it red

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

Insättning, ett enkelt exempel

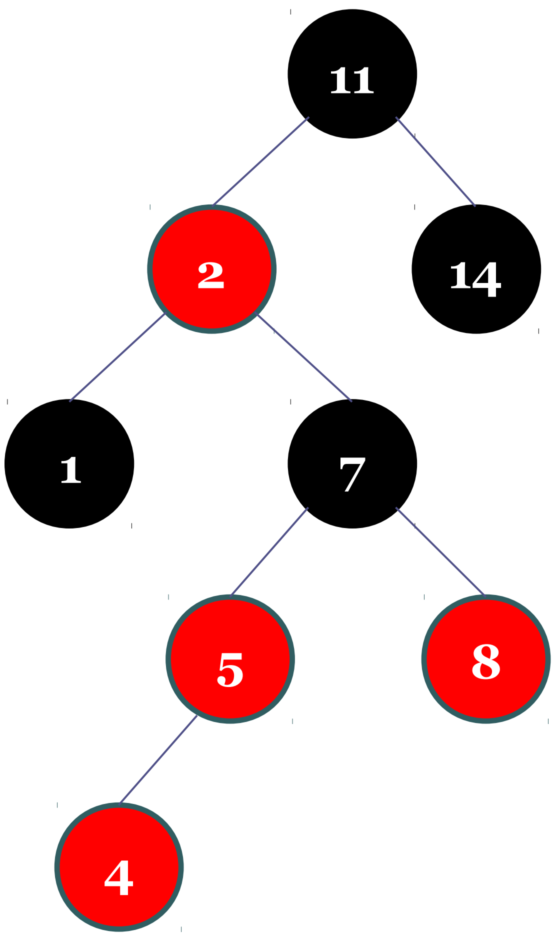


Invariants:

- A node is either red or black

1. The root is always black
2. A red node always has black children (a null reference is considered to refer to a black node)
3. The number of black nodes in any path from the root to a leaf is the same

Insättning, ett enkelt exempel

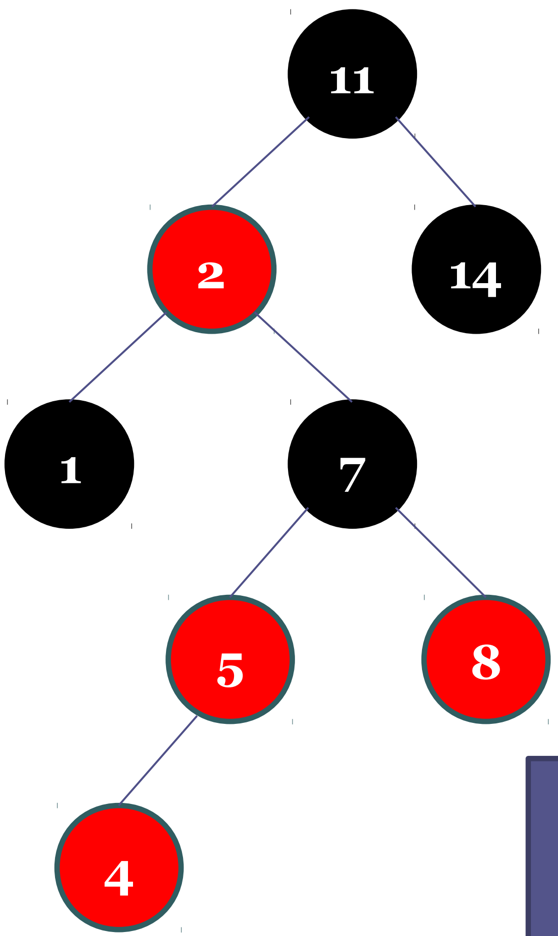


Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Insättning, ett enkelt exempel



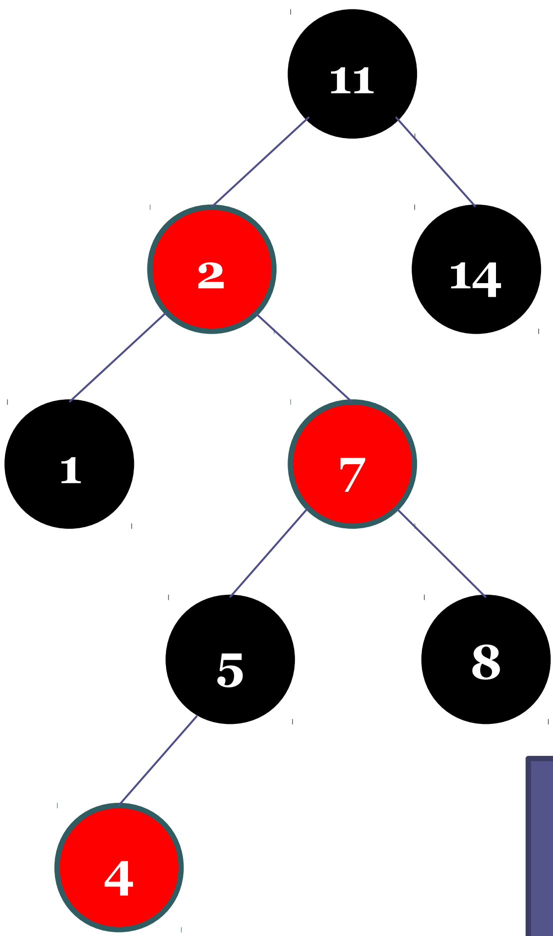
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Colour flip!

Insättning, ett enkelt exempel



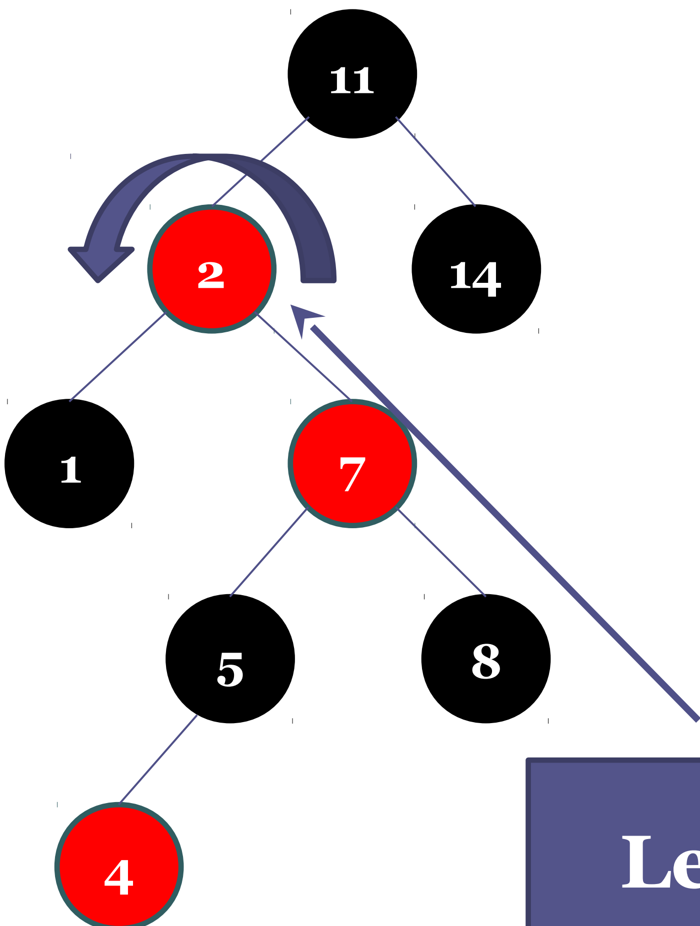
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

The problem has now shifted up the tree

Insättning, ett enkelt exempel



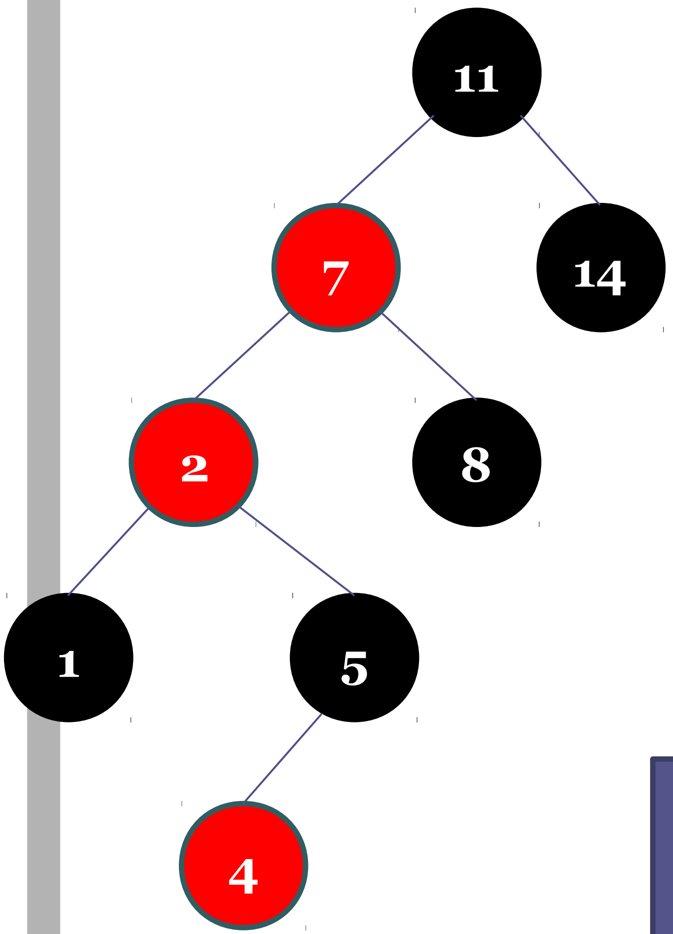
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a `null` reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-right tree: Rotate left about 2

Insättning, ett enkelt exempel



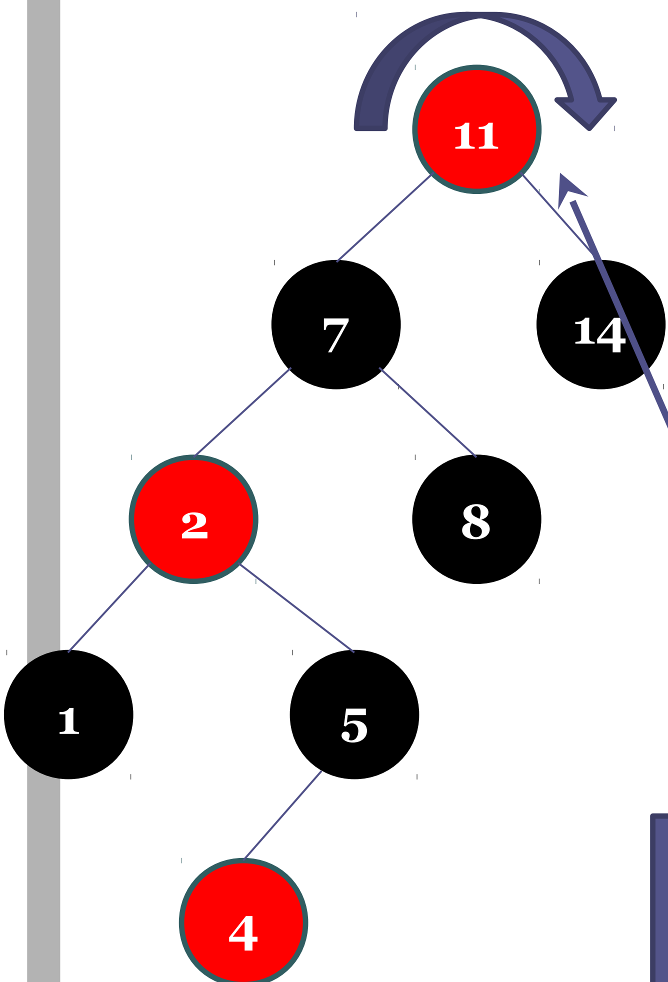
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-left tree: swap colours of 7 and 11

Insättning, ett enkelt exempel



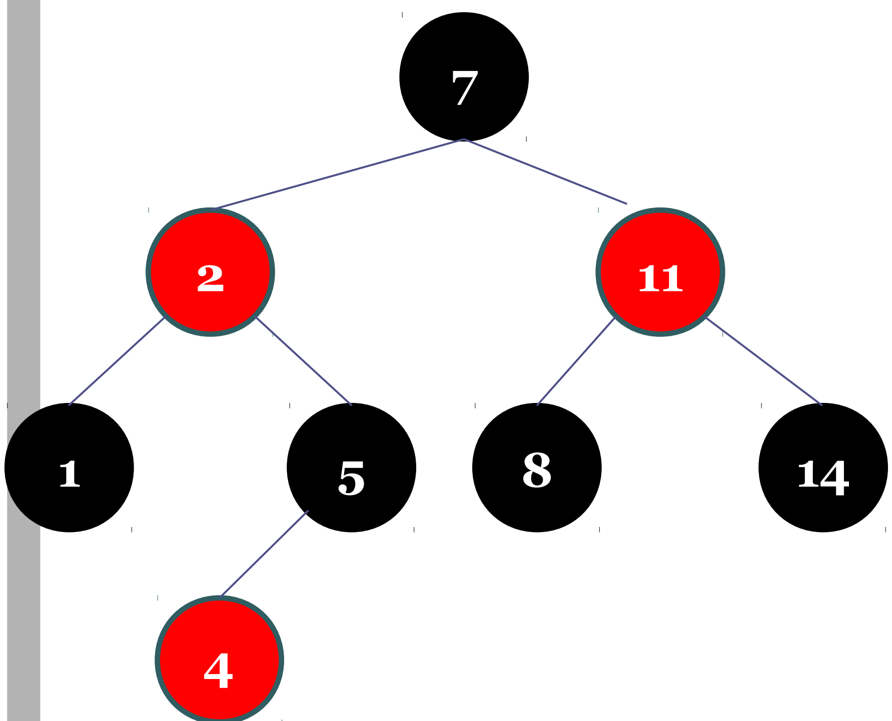
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a `null` reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-left tree: Rotate right around 11 to restore the balance

Insättning, ett enkelt exempel



Invariants:
- A node is either red or black

1. The root is always black
2. A red node always has black children (a null reference is considered to refer to a black node)
3. The number of black nodes in any path from the root to a leaf is the same

Top-down insertion

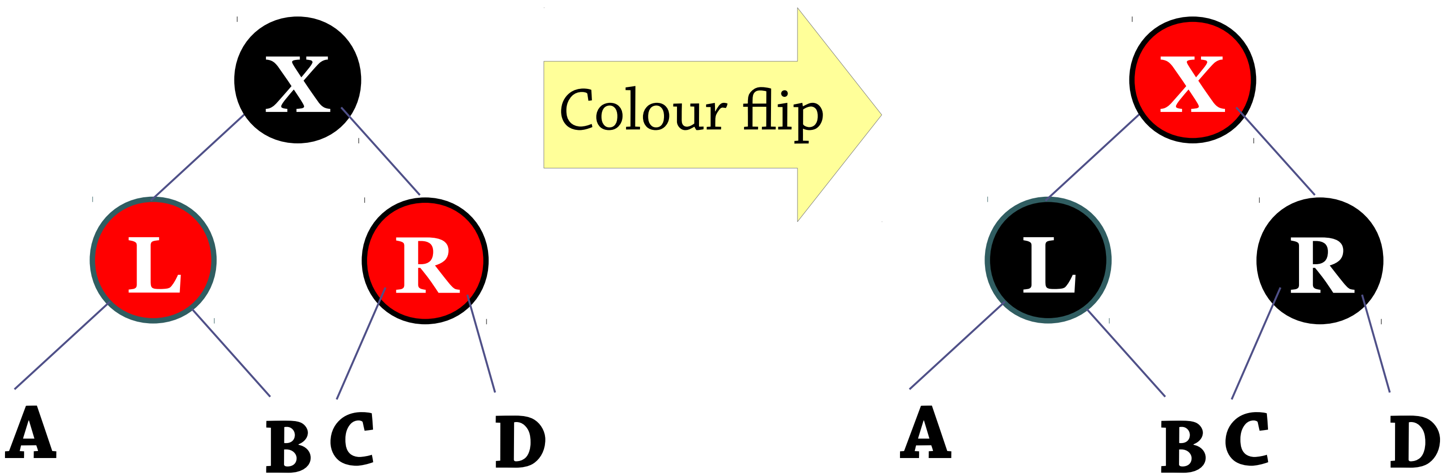
In bottom-up insertion, we sometimes need to move up the tree rebalancing and recolouring it after we insert an element

But this only happens if **P** and **S** are both red

Idea: as we go *down* the tree looking for the insertion point, rebalance and recolour the tree so that either **P** or **S** is black – that way we never need to move up the tree again after insertion!

Top-down insertion

If on the way down we come across a node **X** with two red children, colour-flip it immediately!



But what if **X**'s parent is also red? We break the invariant!

Observation: **X**'s parent's sibling must be black (or we would've colour-flipped them on the way down), so a single rotation + recolouring will fix the invariant!

Top-down insertion

Go down the tree, looking for the parent node

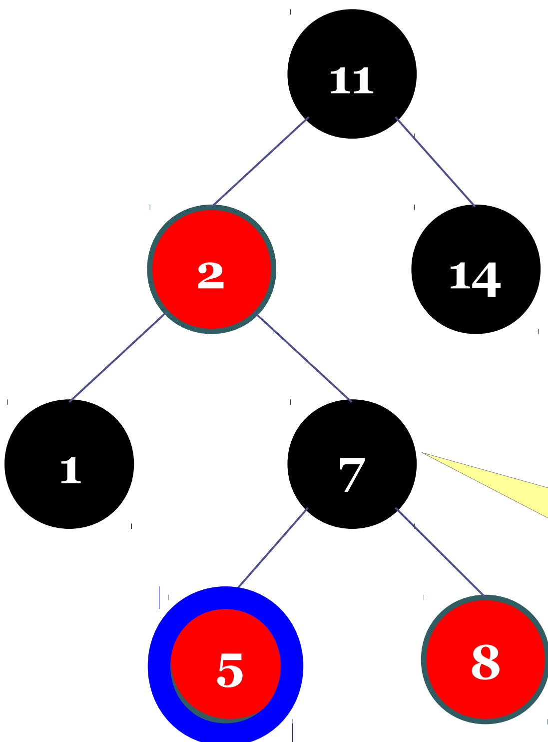
Whenever a node **X** has two red children, colour-flip; if **X**'s parent **P** is red, use rotations and recolourings as before to fix it

- This is easy because **P**'s sibling must be black

When you get to the right node **P**, add a red child; if **P** is also red, use rotations and recolourings to fix it

- Again, **P**'s sibling is black so we avoid the difficult case

Insättning, ett enkelt exempel



Invariants:

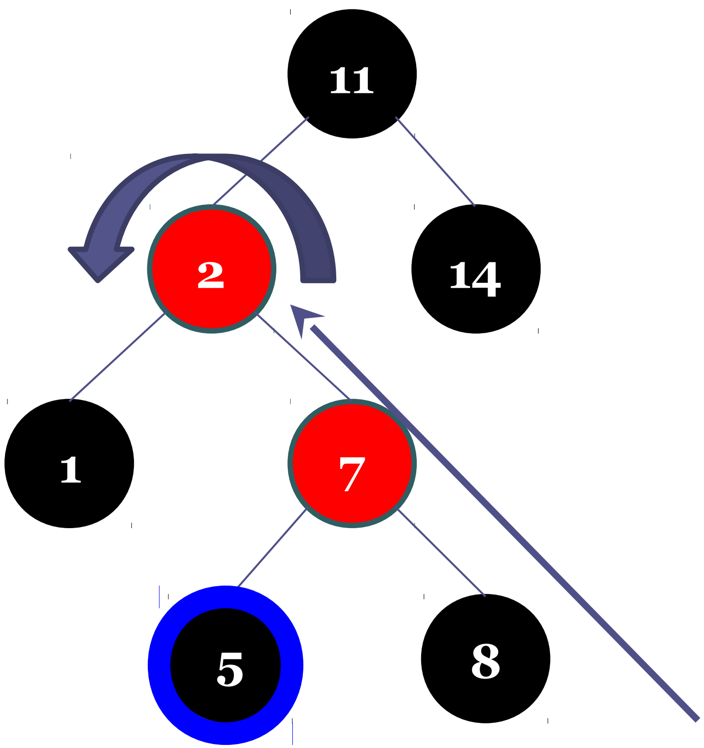
- A node is either red or black

1. The root is always black
2. A red node always has black children (a null reference is considered to refer to a black node)
3. T… odes … to a le…

Inserting 4,
we get to a node with
two red children:
colour flip!

We would've visited
5 next: remember it!

Insättning, ett enkelt exempel

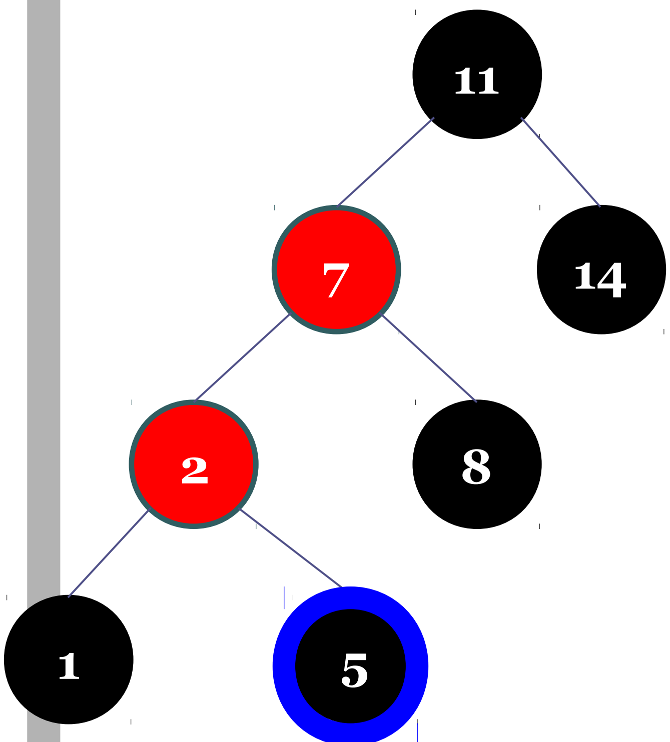


Invariants:
- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-right tree: Rotate left about 2

Insättning, ett enkelt exempel



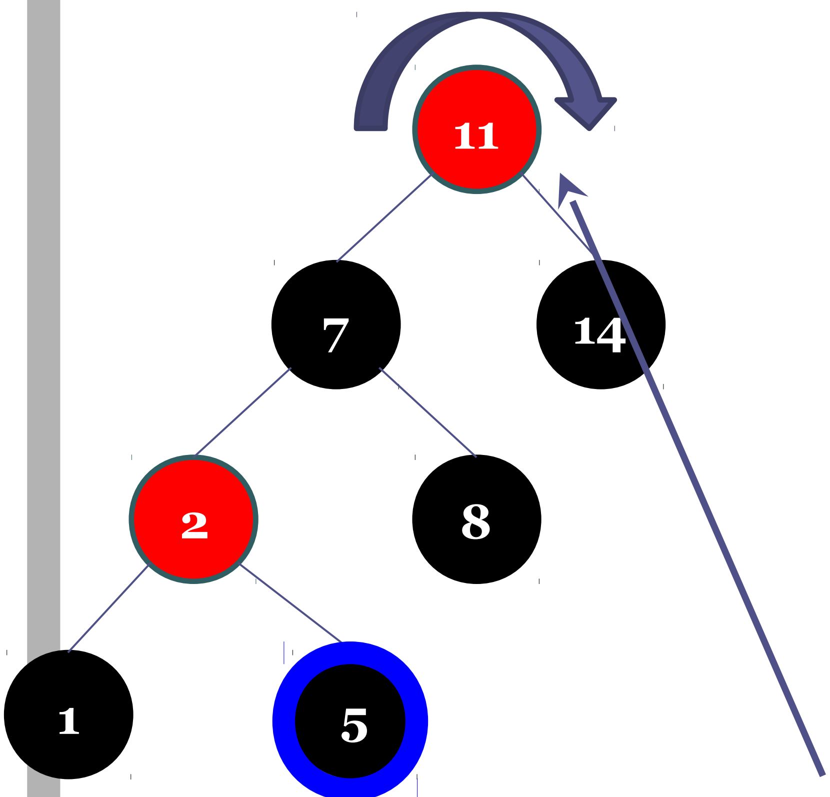
Invariants:

- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-left tree: swap colours of 7 and 11

Insättning, ett enkelt exempel

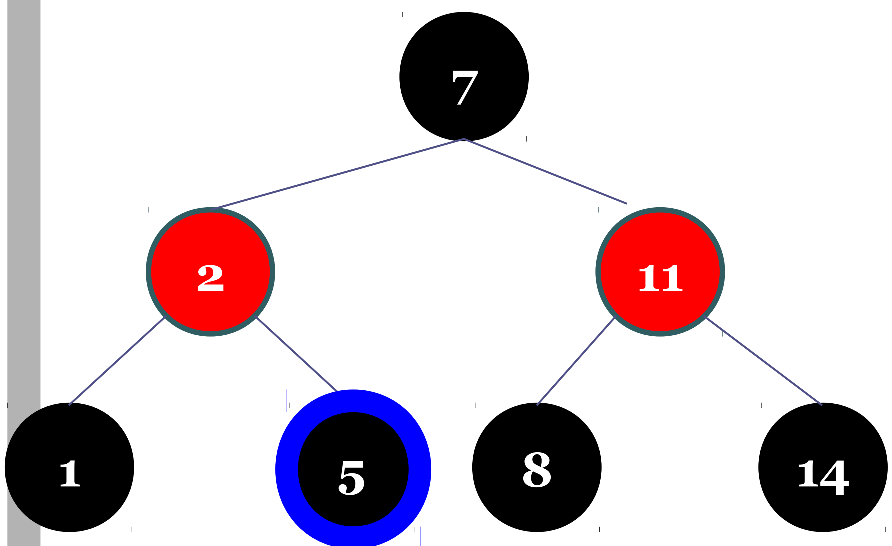


Invariants:
- A node is either red or black
- The root is always black
- A red node always has black children (a null reference is considered to refer to a black node)

1. The number of black nodes in any path from the root to a leaf is the same

Left-left tree: Rotate right around 11 to restore the balance

Insättning, ett enkelt exempel



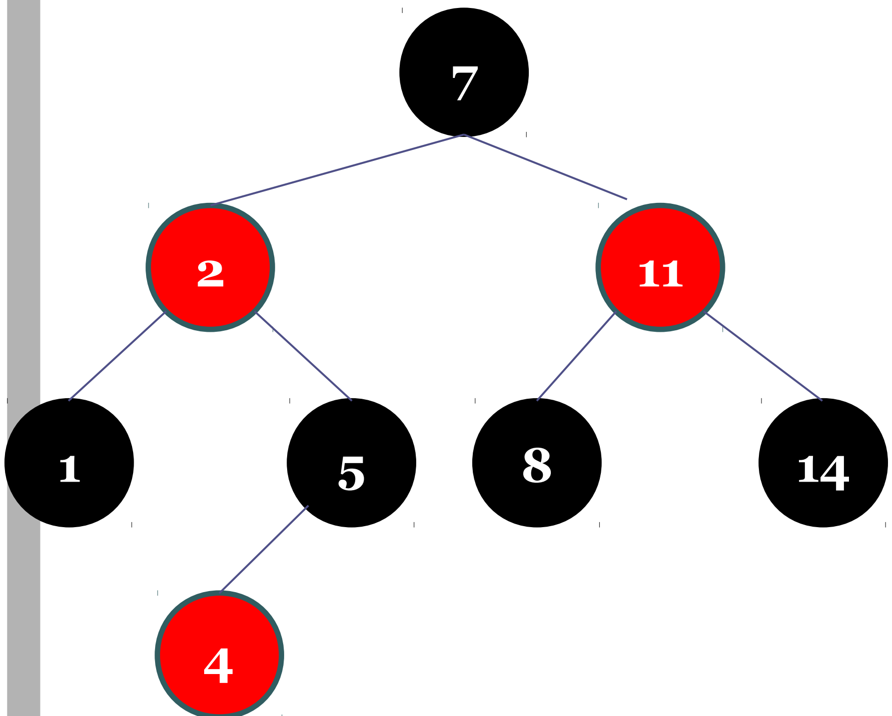
Invariants:

- A node is either red or black

1. The root is always black
2. A red node always has black children (a null reference is considered to refer to a black node)
3. The number of black nodes in any path from the root to a leaf is the same

**The colour flip is finished.
Now we continue down and insert 4!**

Insättning, ett enkelt exempel



Invariants:

- A node is either red or black

1. The root is always black
2. A red node always has black children (a null reference is considered to refer to a black node)
3. The number of black nodes in any path from the root to a leaf is the same

No need to go up the tree afterwards

Red-black deletion

Use the normal BST deletion algorithm, which will end up removing a leaf from the tree

If the leaf is *red*, everything's fine

If the leaf is *black*, the invariant is broken

Idea: go down the tree, making sure that the *current node* is always red

Lots of special cases! See book 19.5.4.

Red-black versus AVL trees

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

2-3 trees

In a binary tree, each node has two children

In a *2-3 tree*, each node has either 2 children (a *2-node*) or 3 (a *3-node*)

A 2-node is a normal BST node:

- One data value *x*, which is greater than all values in the left subtree and less than all values in the right subtree

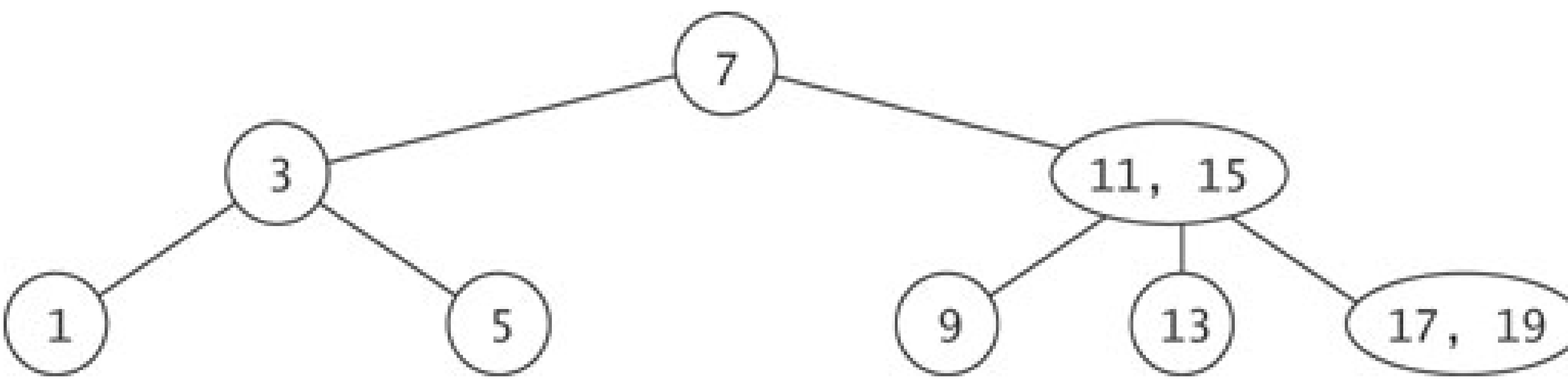
A 3-node is different:

- *Two* data values *x* and *y*
- All the values in the left subtree are less than *x*
- All the values in the middle subtree are between *x* and *y*
- All the values in the right subtree are greater than *y*

2-3 trees

2-node 3-node

An example of a 2-3 tree:



Why 2-3 trees?

To get a balanced BST we had to find funny invariants and define our operations in odd ways

With a 2-3 tree we have the invariant:

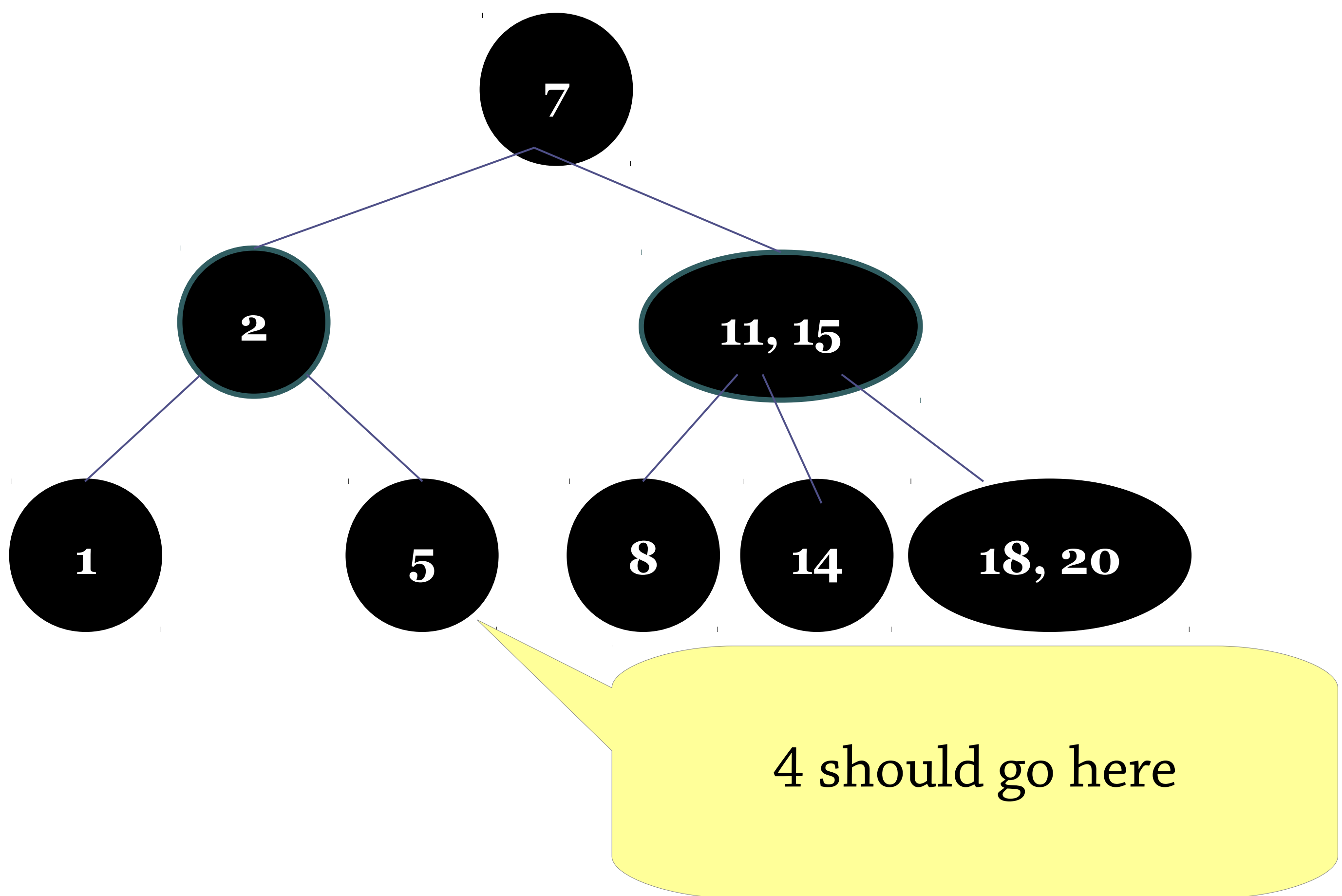
- *The tree is always **perfectly** balanced*

and we can maintain it!

Insertion into a 2-3 tree

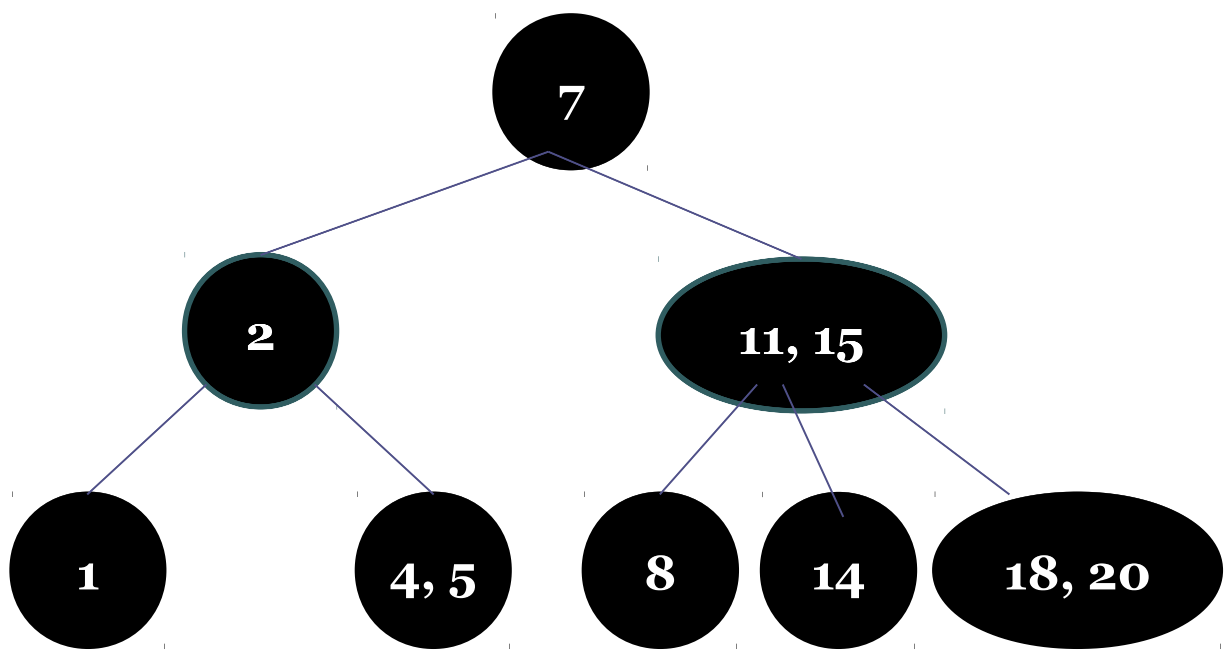
Suppose we want to insert 4

First, find the right leaf node



Insertion into a 2-3 tree

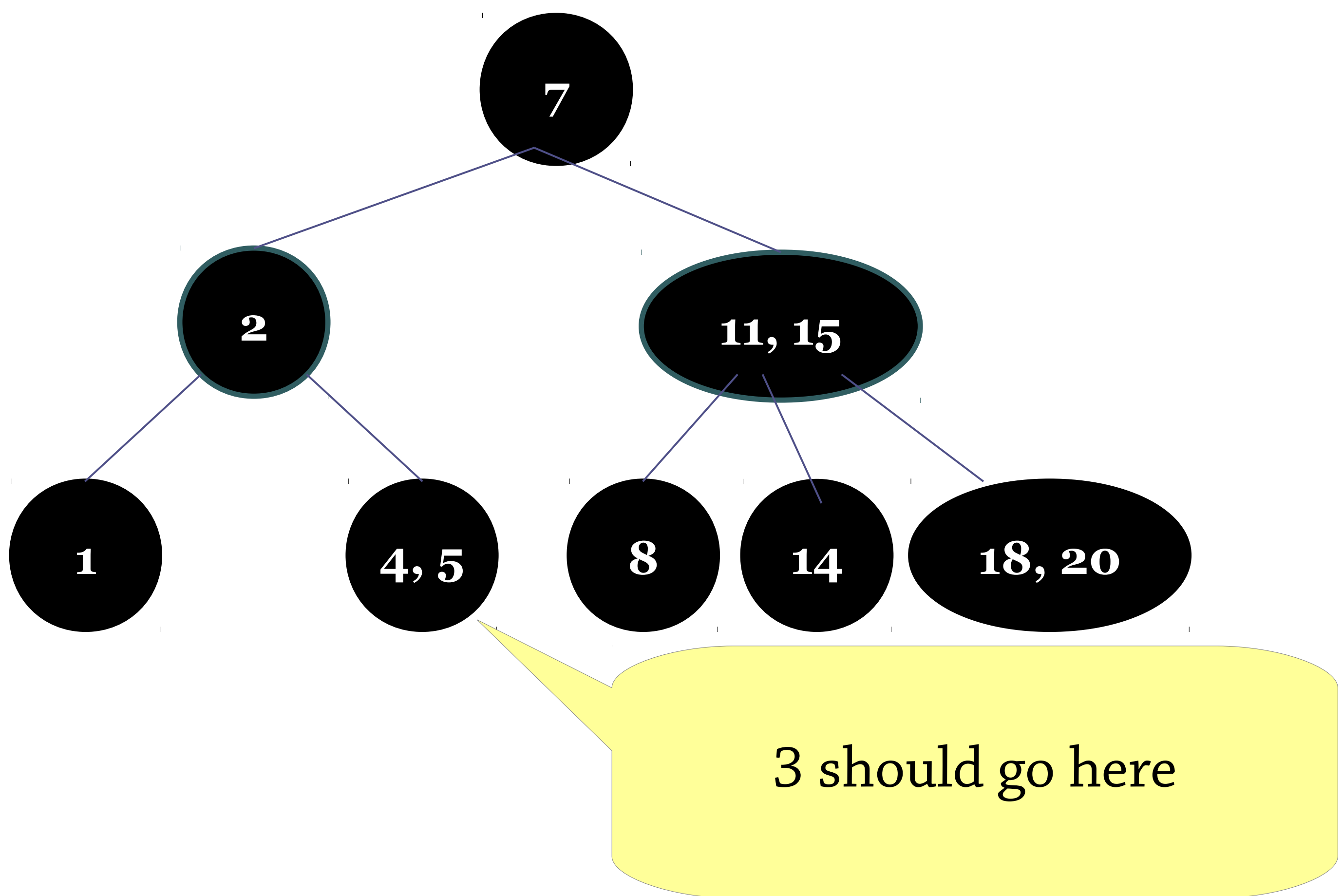
If it's a 2-node, turn it into a 3-node by adding the value!



Insertion into a 2-3 tree

Now suppose we want to insert 3.

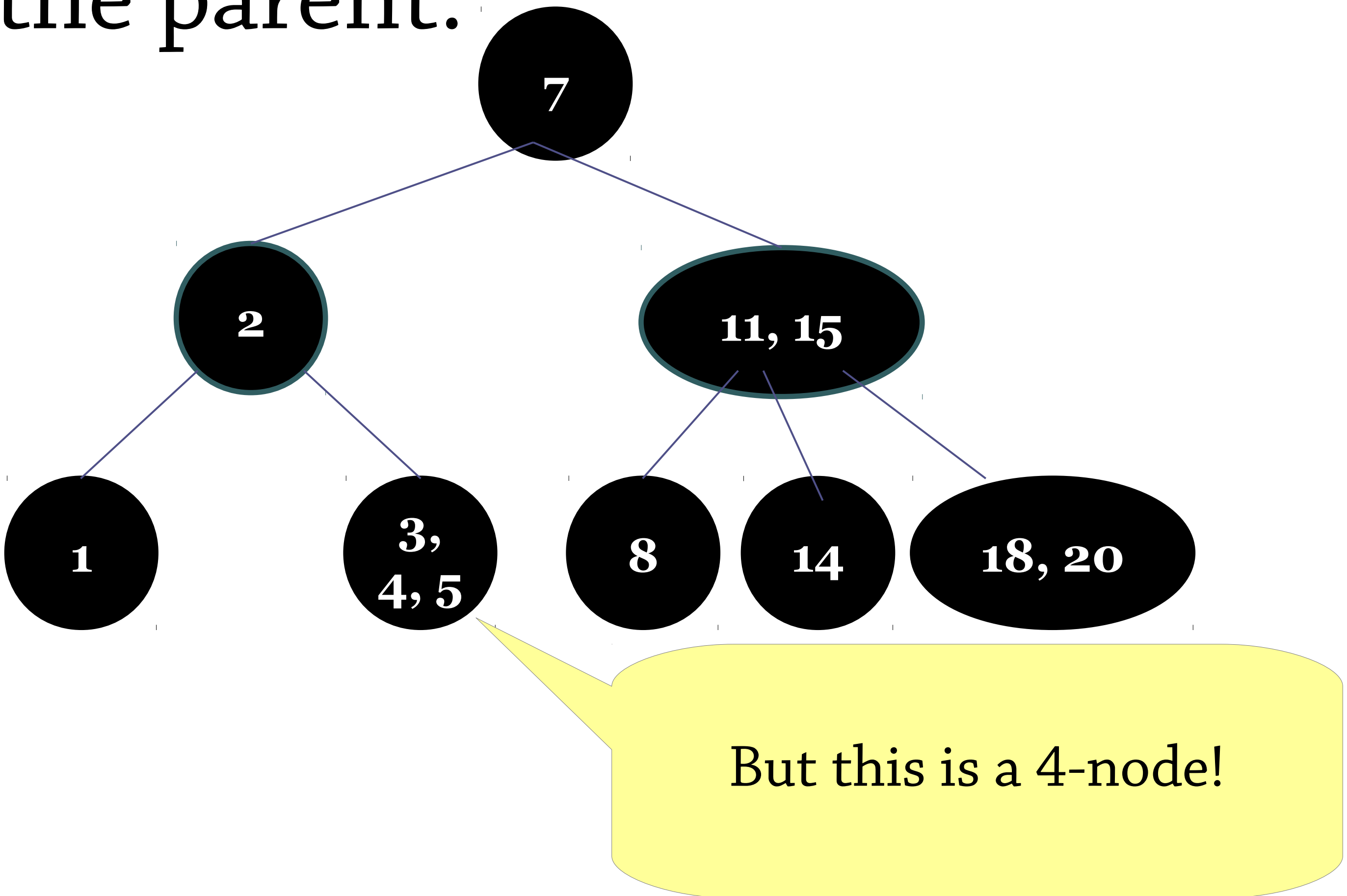
Find the right leaf node



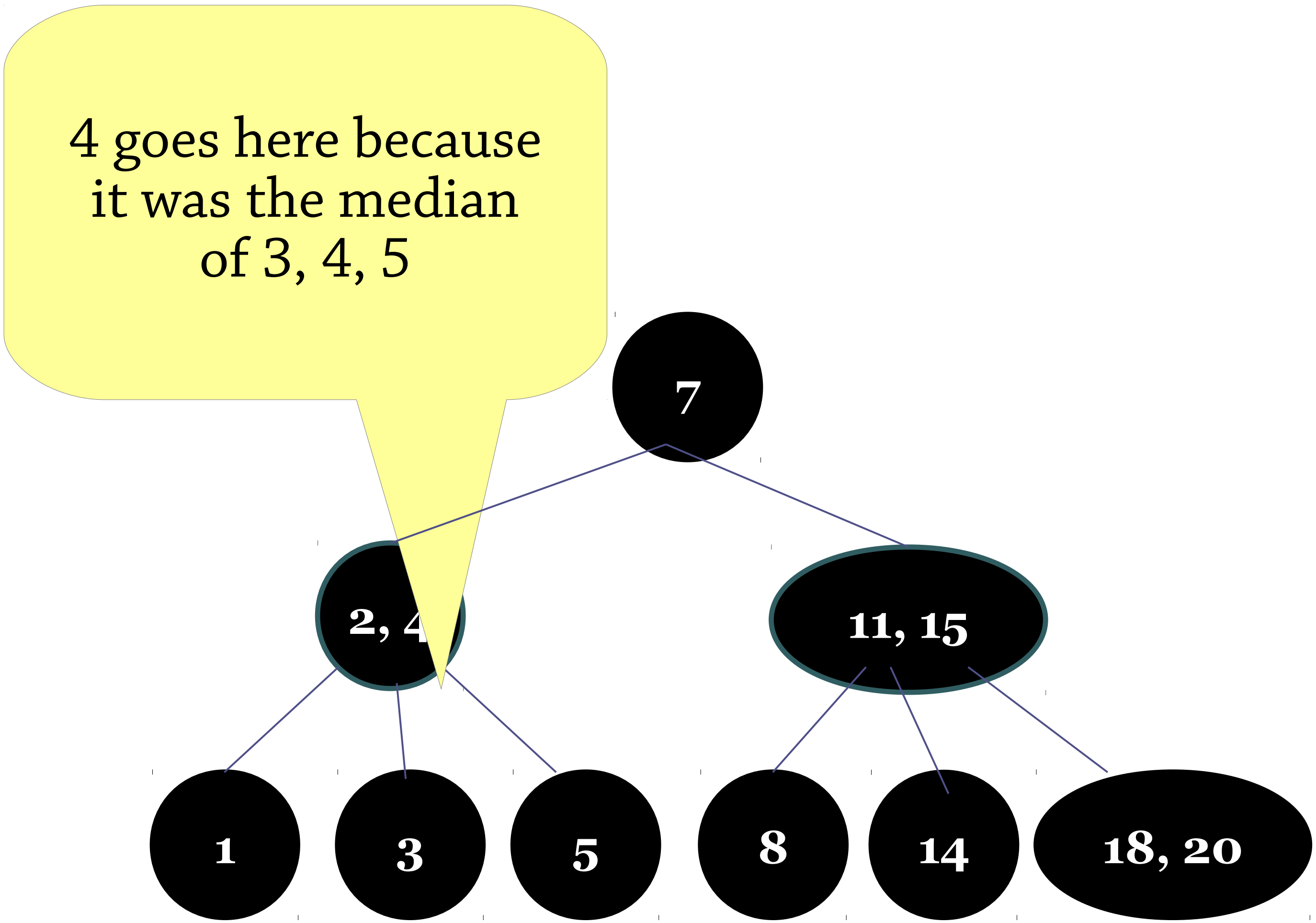
Insertion into a 2-3 tree

We now have a 4-node – not allowed!

Split it into two 2-nodes and attach them to the parent:



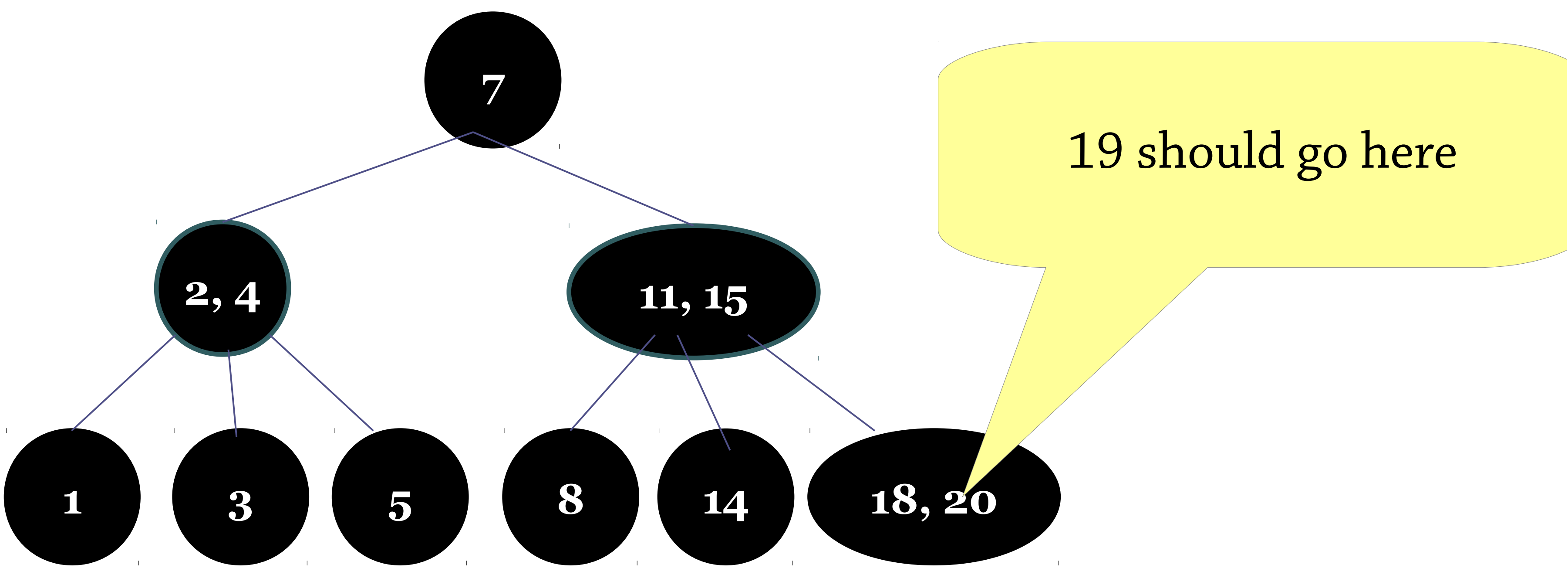
Insertion into a 2-3 tree



Insertion into a 2-3 tree

Now suppose we want to add 19.

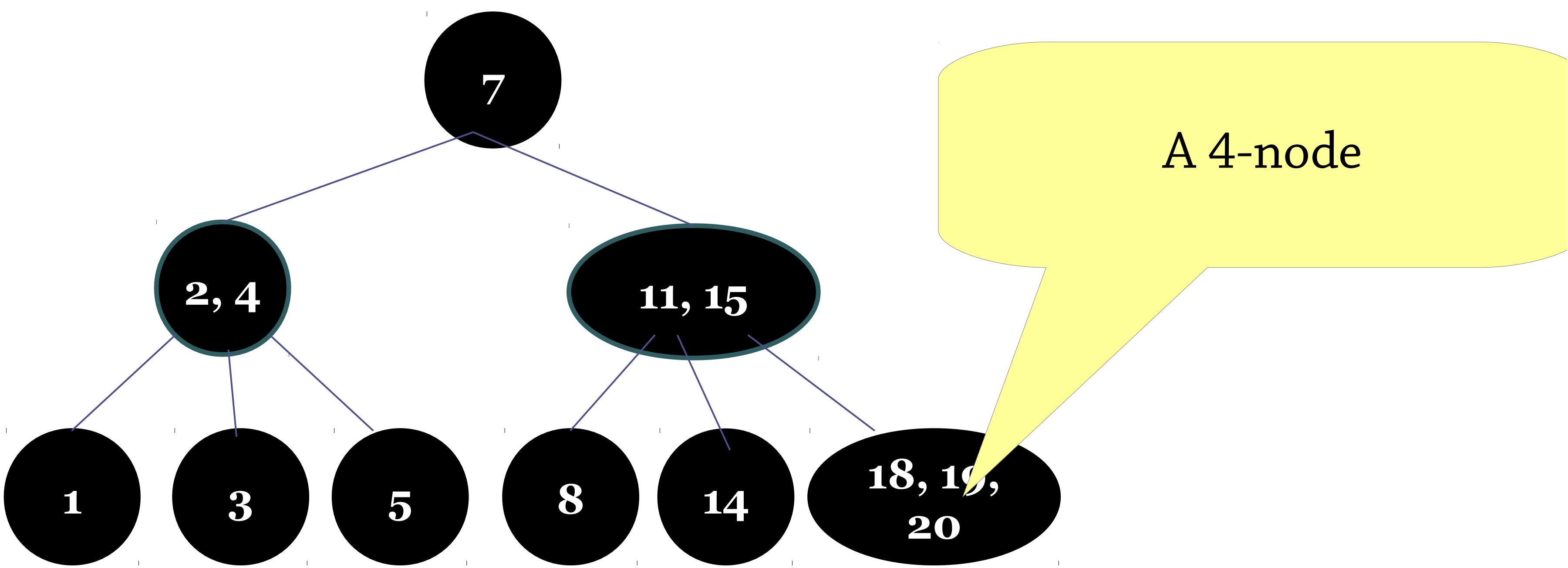
Find the right leaf node and add it



Insertion into a 2-3 tree

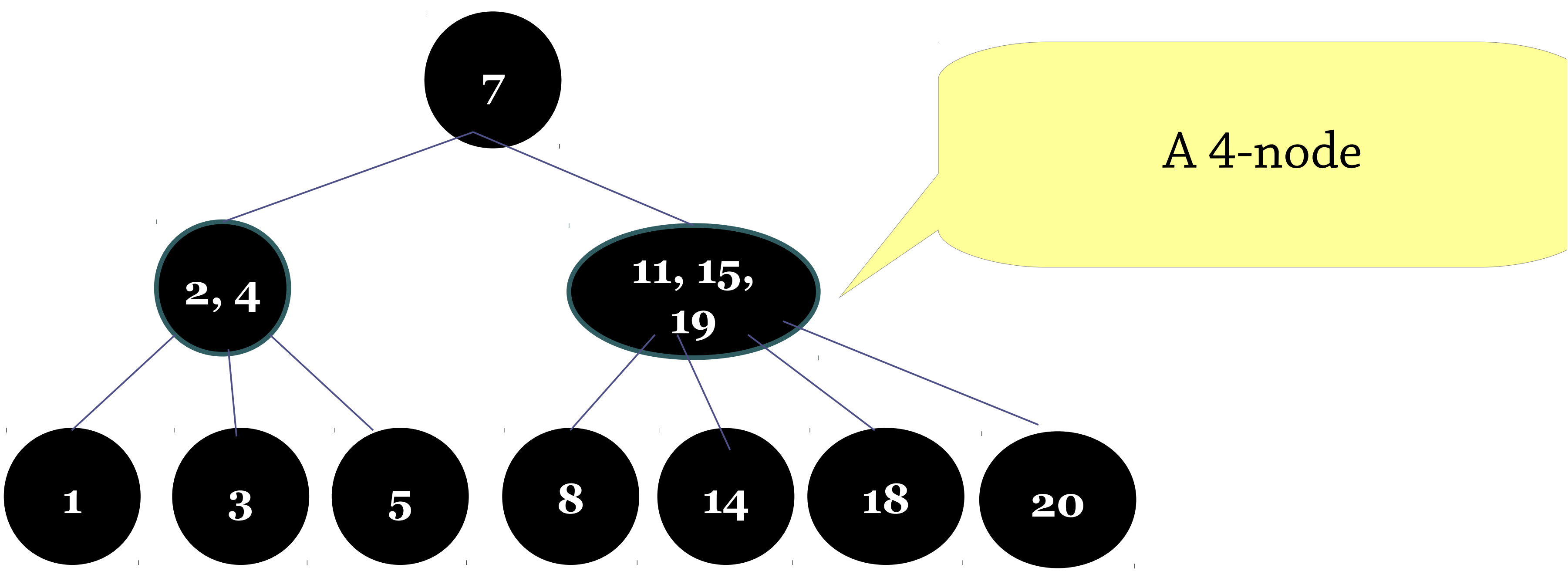
Now suppose we want to add 19.

Again, we have a 4-node – split it



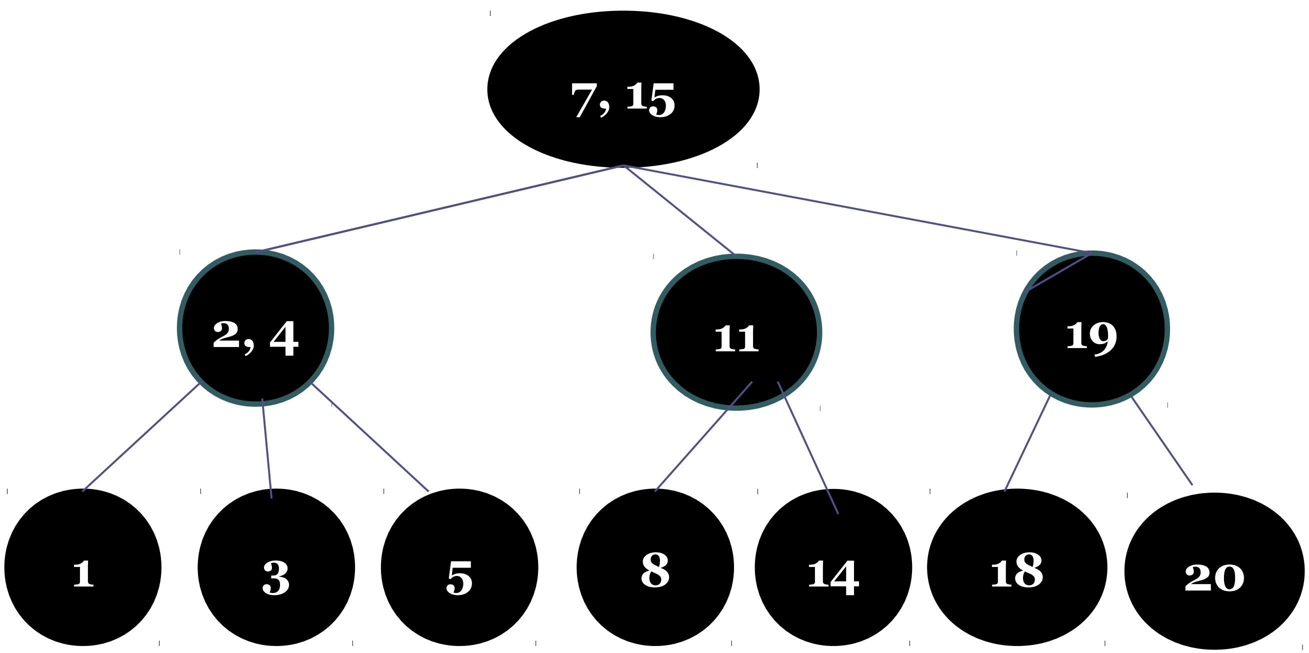
Insertion into a 2-3 tree

But now we have a 4-node one level above! Split that.



Insertion into a 2-3 tree

Finally we have a 2-3 tree again.



2-3 trees, summary

2-3 trees do not use rotation, unlike balanced BSTs

Instead, they keep the tree perfectly balanced and use *splits* when there is no room for a new node

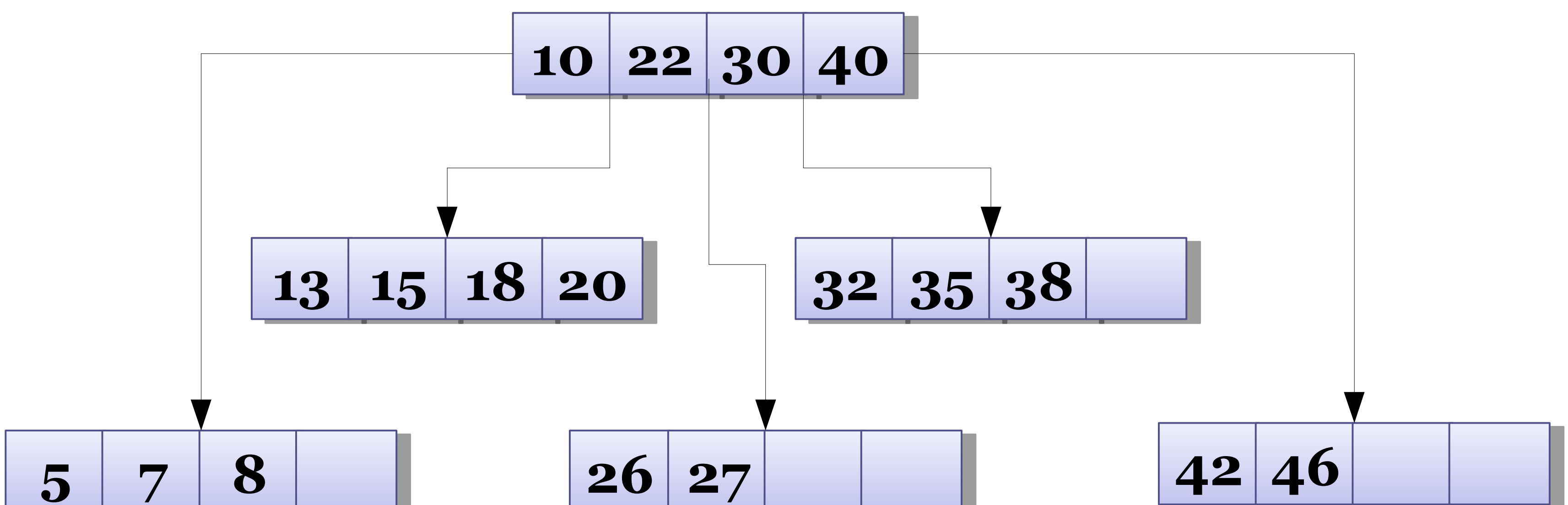
Complexity is O(log n), as tree is perfectly balanced

Much simpler than e.g. red-black trees!

B-trees

B-trees generalise 2-3 trees:

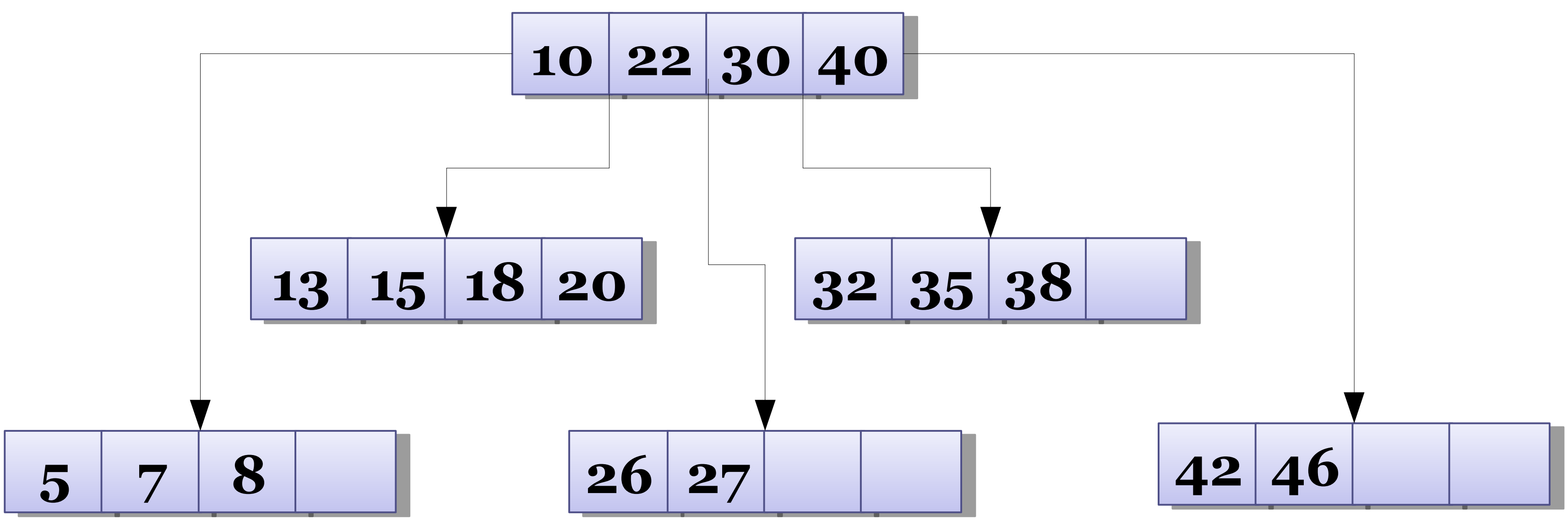
- In a B-tree of order *k*, a node can have *k* children
- Each non-root node must be at least half-full
- A 2-3 tree is a B-tree of order 3



Why B-trees

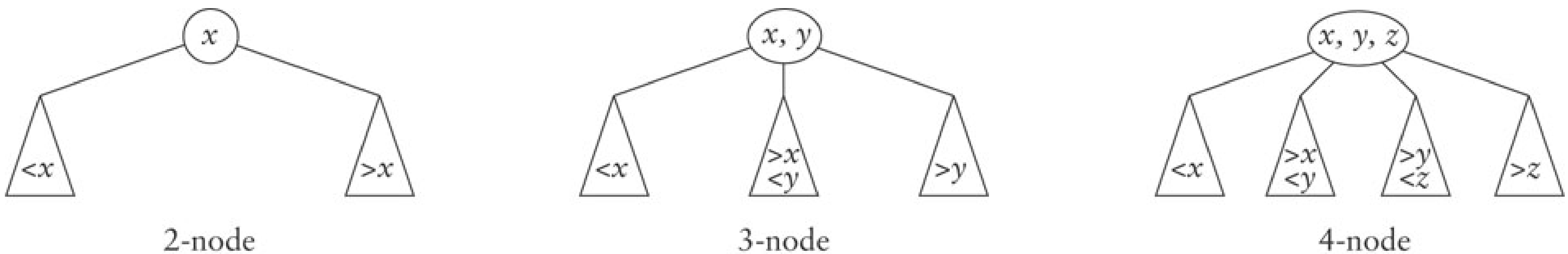
B-trees are used for disk storage in databases:

- 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

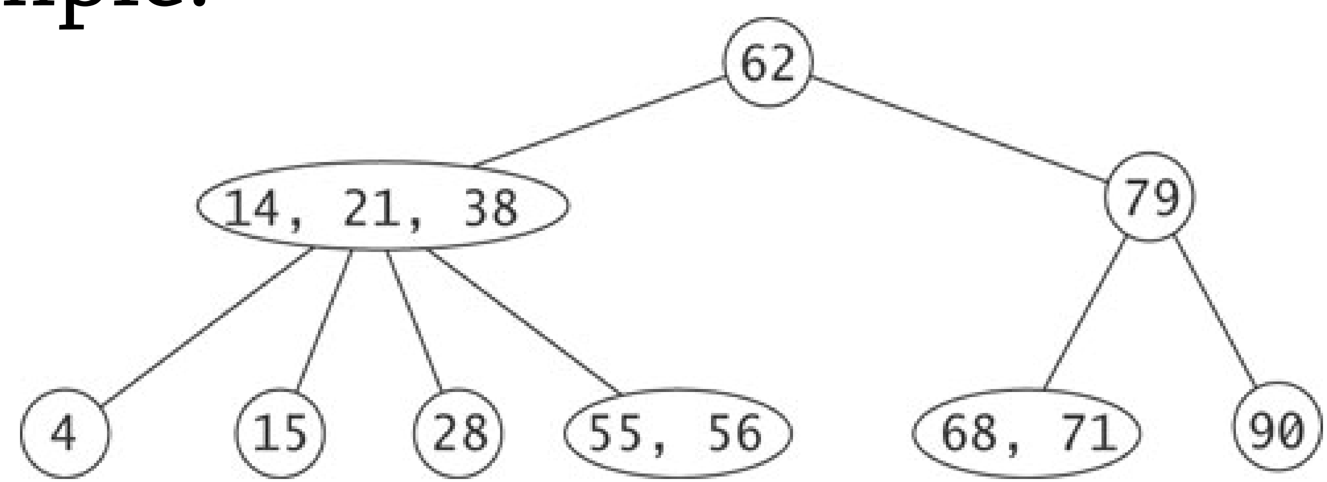


2-3-4 trees

A 2-3-4 tree is a B-tree of order 4



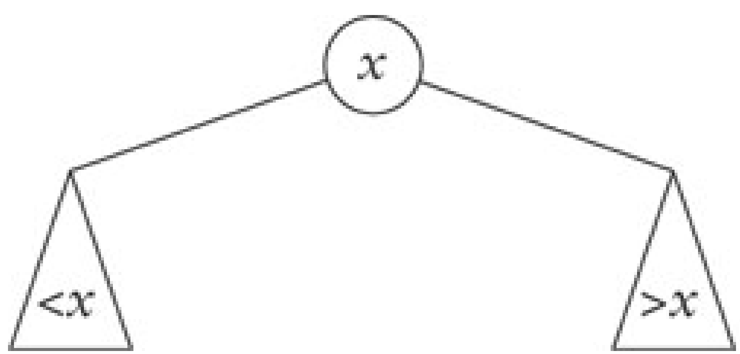
Example:

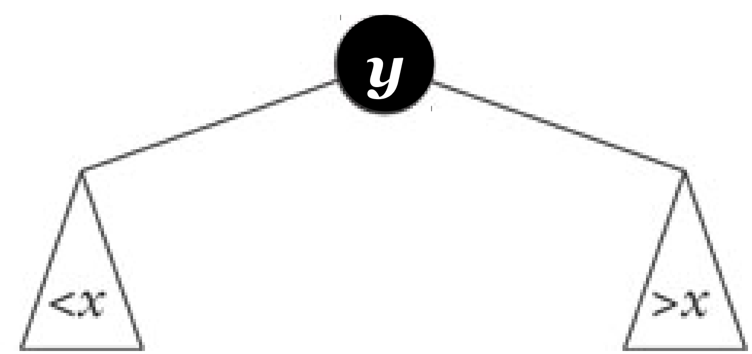


Red-black trees are 2-3-4 trees!

Any red-black tree is equivalent to a 2-3-4 tree!

- A 2-node is a black node

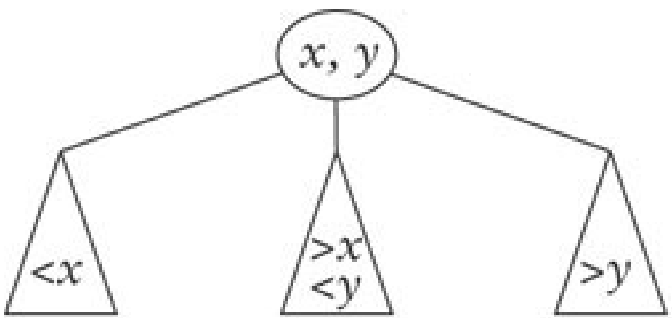


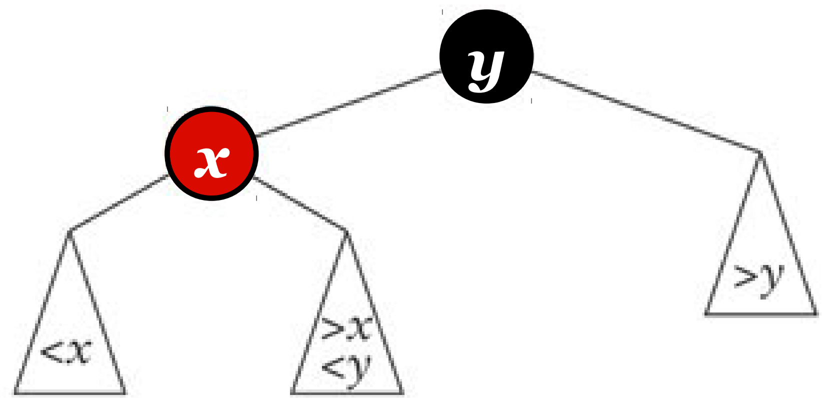


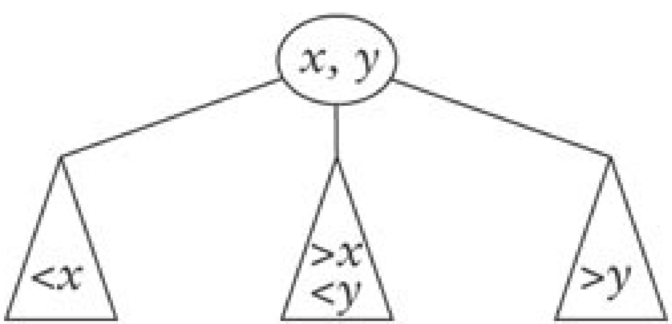
Red-black trees are 2-3-4 trees!

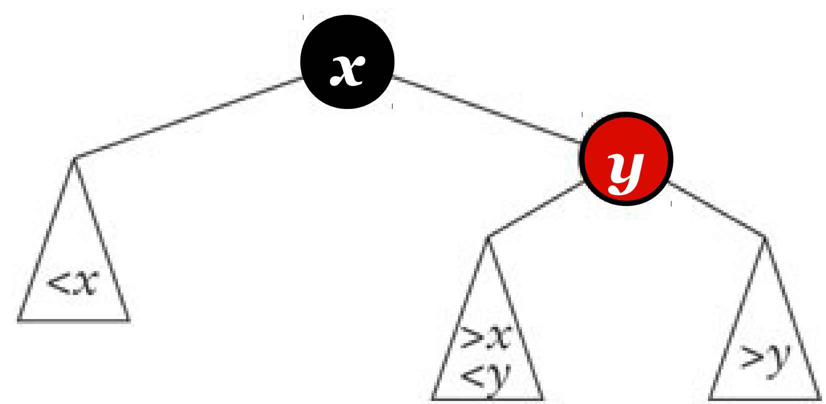
Any red-black tree is equivalent to a 2-3-4 tree!

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





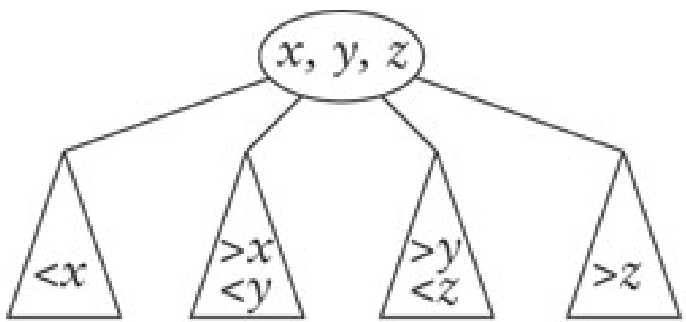


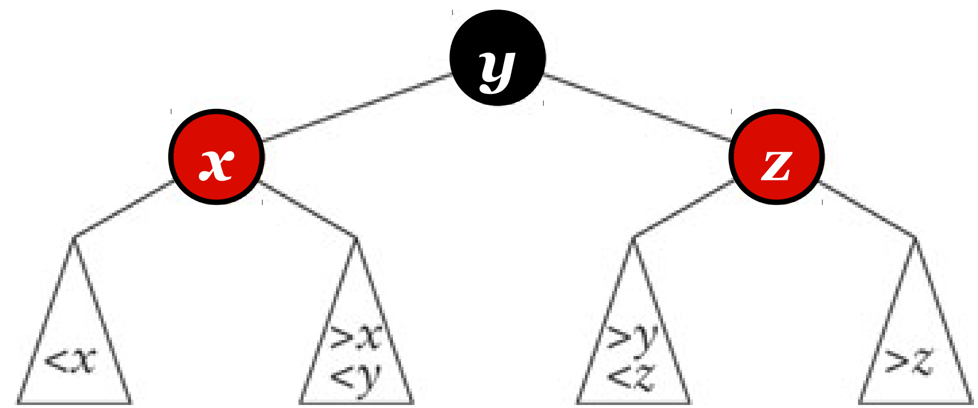


Red-black trees are 2-3-4 trees!

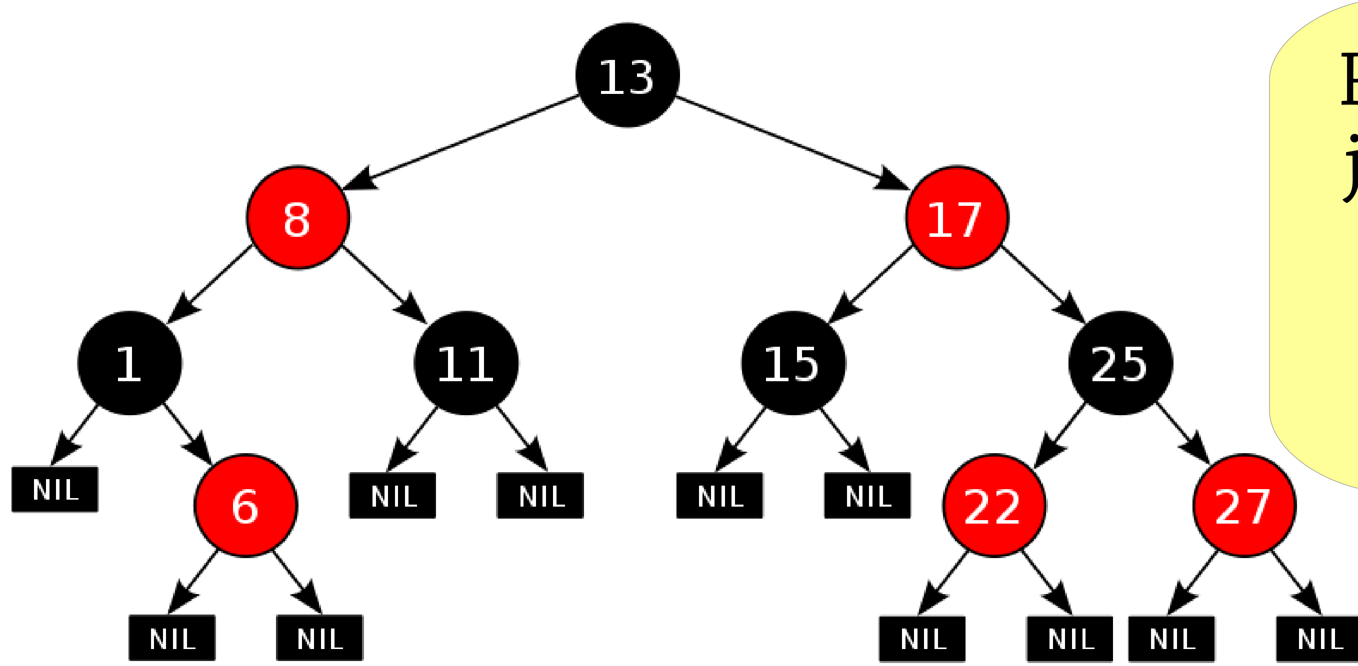
Any red-black tree is equivalent to a 2-3-4 tree!

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

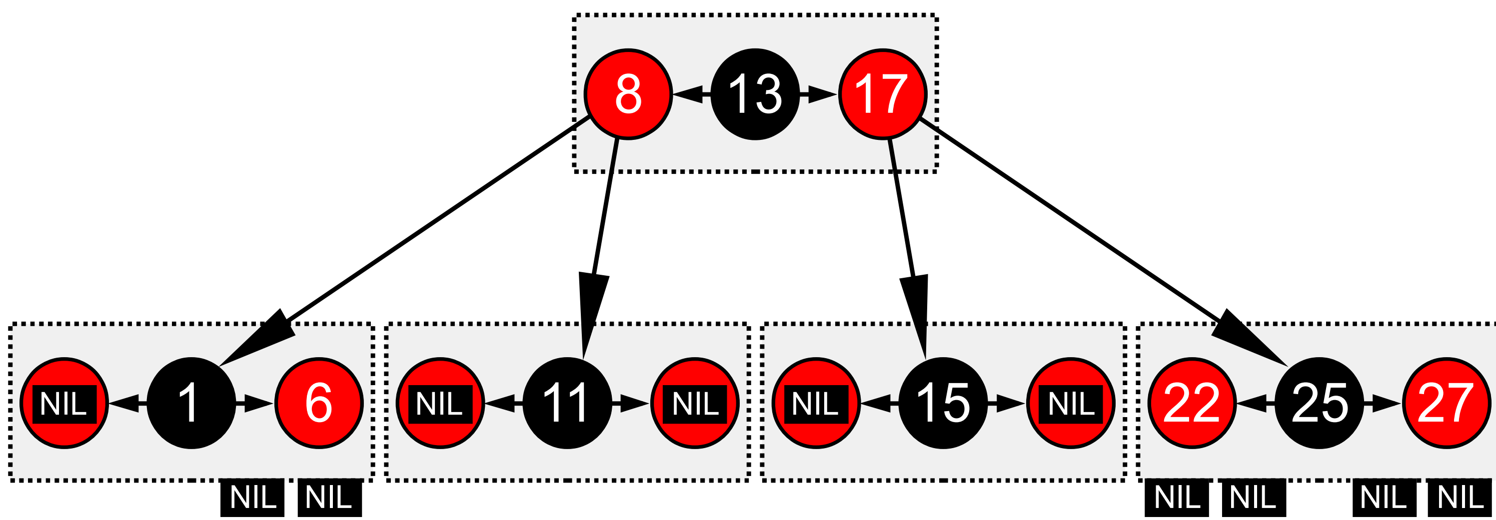




Surprise!



Red-black trees are just a fancy way of representing a 2-3-4 tree using a binary tree!



Red-black trees vs 2-3-4 trees

Red-black trees	**2-3-4 trees**
Black node with no red children	2-node
Black node with one red child	3-node
Black node with two red children	4-node
Add a red child to a black node	Change a 2-node to a 3-node
Add a red child to a red node with a black sibling and rotate	Change a 3-node to a 4-node
Colour change + rotate	Split a 4-node

Exercise: check for yourself how the red-black tree operations correspond to 2-3-4 tree operations!

Summary

Red-black trees – normally faster than AVL trees because there is no need to go *up* the tree after inserting or deleting

- On the other hand, trickier to implement

2-3 trees: allow 2 or 3 children per node

- Possible to keep perfectly balanced
- Slightly annoying to implement

B-trees: generalise 2-3 trees to *k* children

- If *k* is big, the height is very small – useful for on-disk trees e.g. databases

Red-black trees are 2-3-4 trees in disguise!