Reminder

No Friday lectures from now on!

Instead, there will be lectures on Wednesdays, 8am, room HB2

Monday lecture continues as normal

2-3 trees, AA trees, B-trees

(Weiss chapter 4.6)

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*)

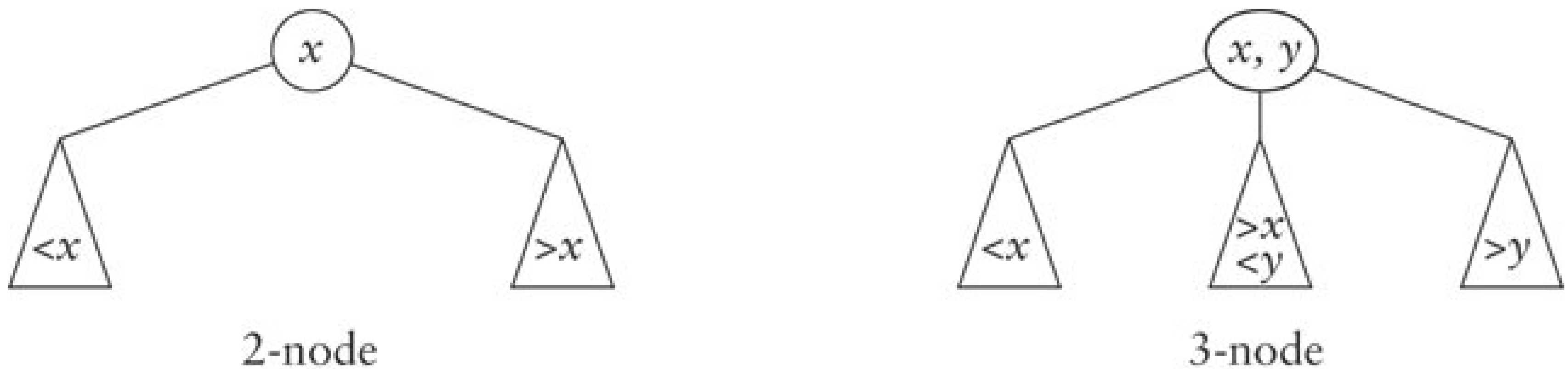
A 2-node is a normal BST node:

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

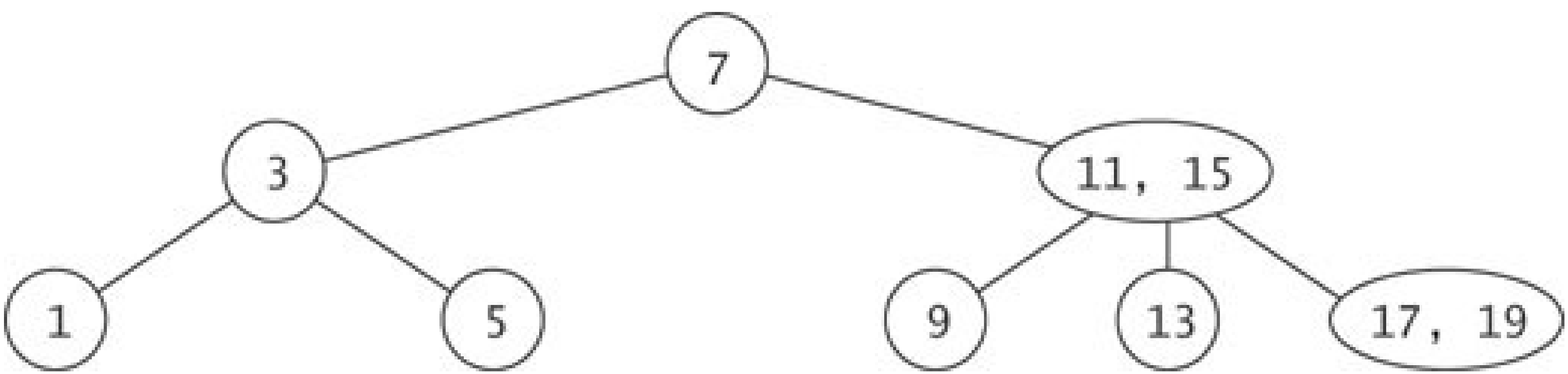
A 3-node is different:

- *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



An example of a 2-3 tree:



Why 2-3 trees?

With a 2-3 tree we can maintain the invariant:

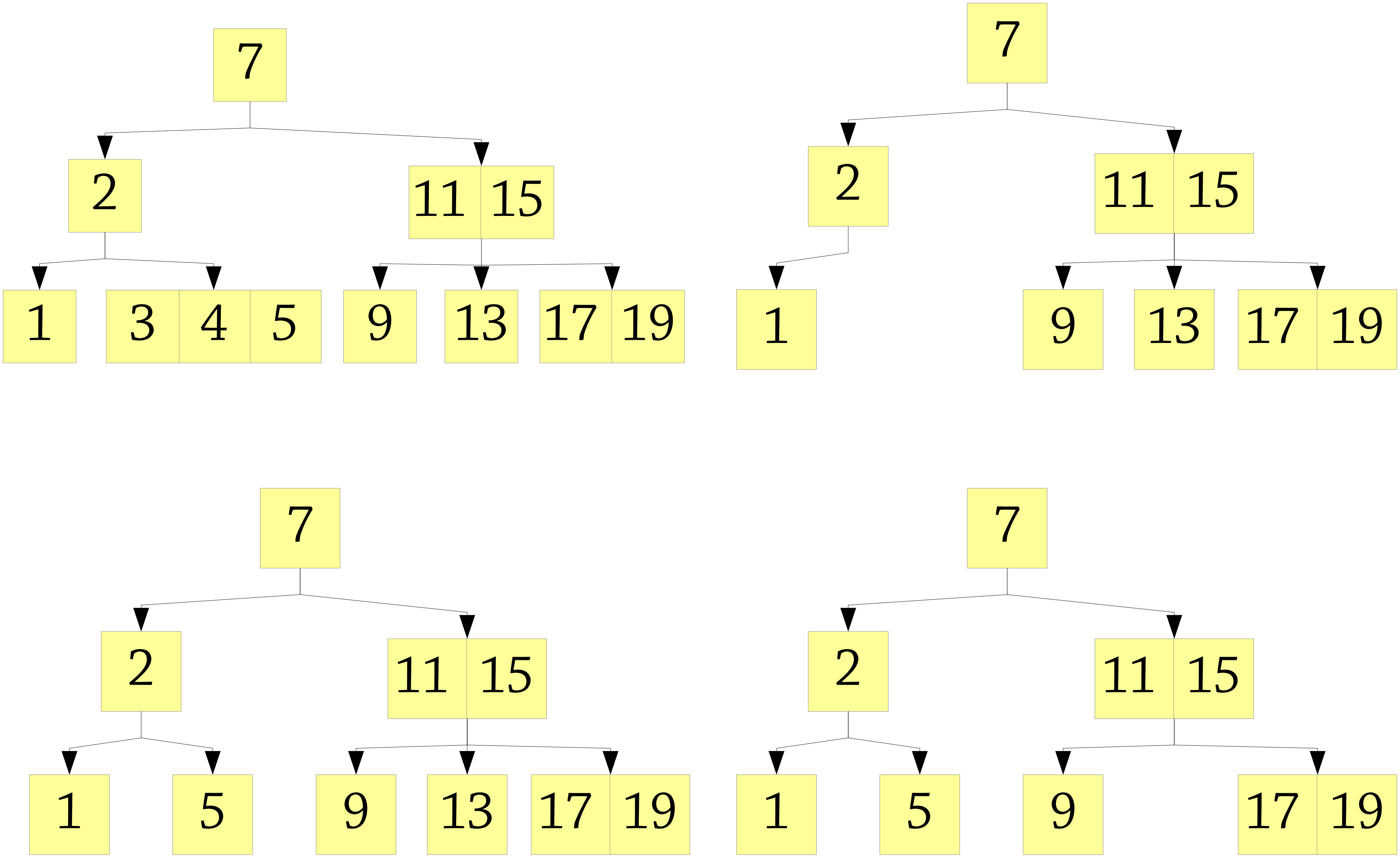
- *The tree is always* ***perfectly*** *balanced!*

Invariant: all children of each node *always* have the same height

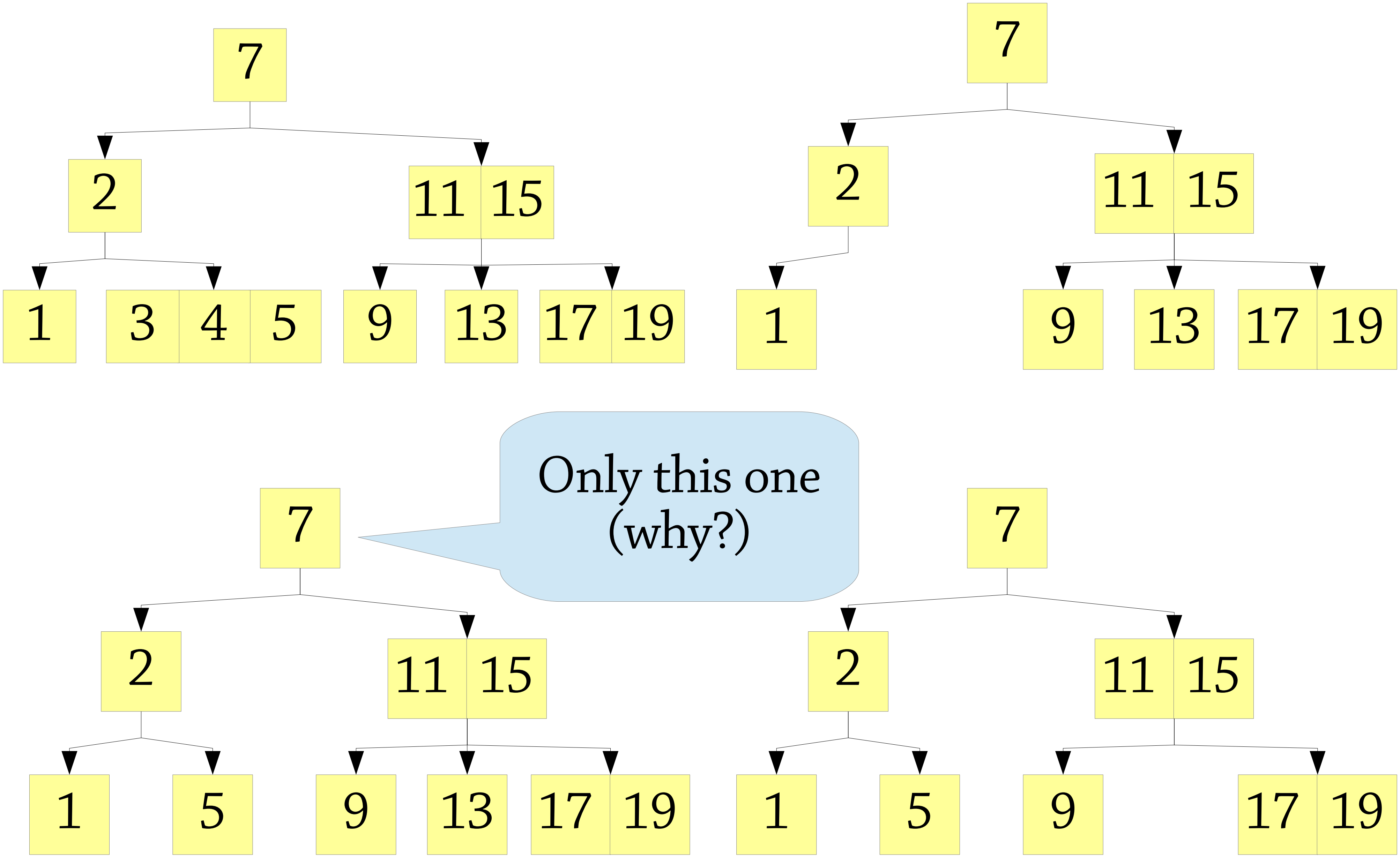
- Note: the empty tree (null) has height 0
- In particular, any non-leaf 2-node has 2 children
- Any non-leaf 3-node has 3 children

This wasn't possible with binary search trees

Which of these are 2-3 trees?

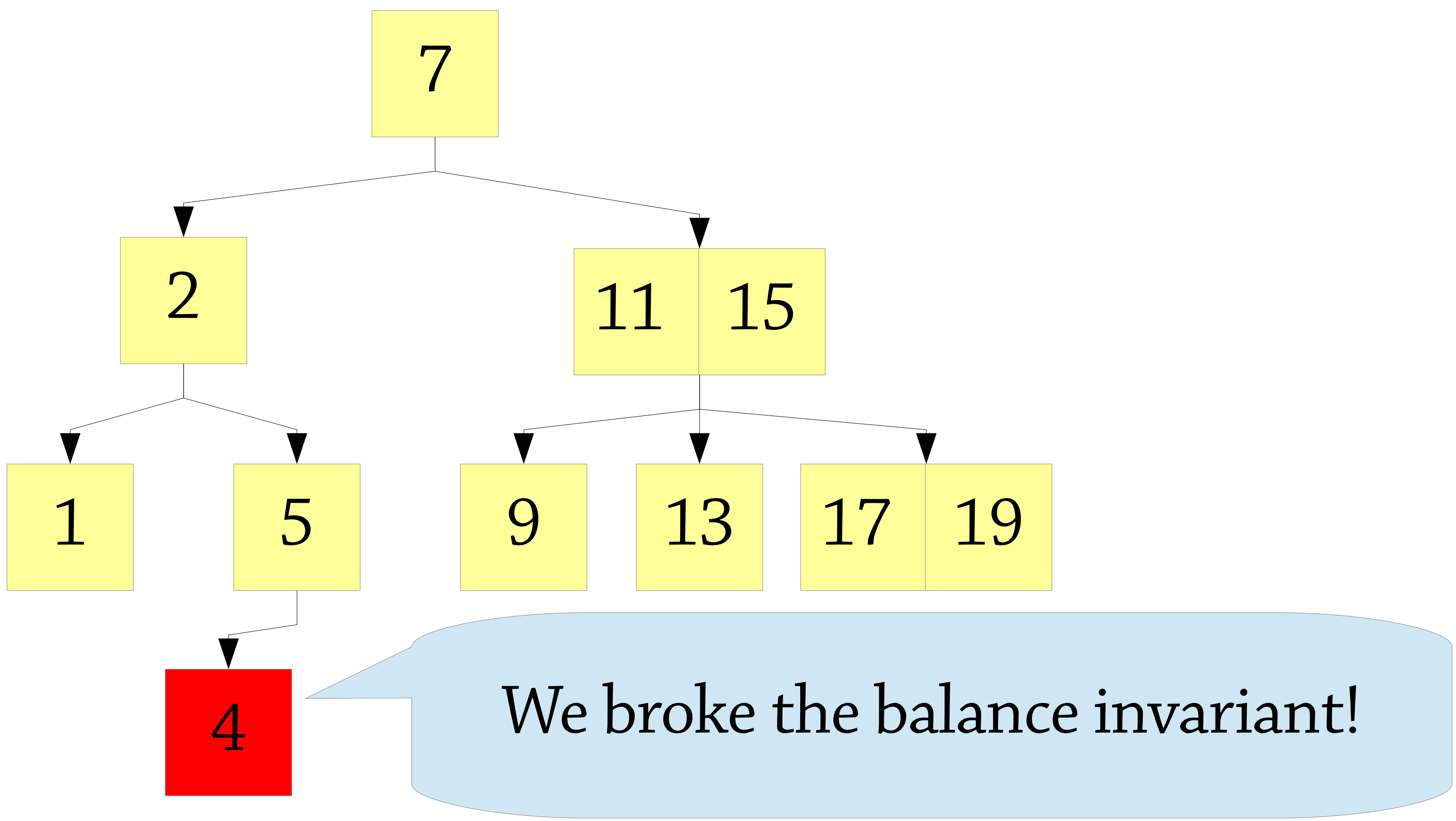


Which of these are 2-3 trees?



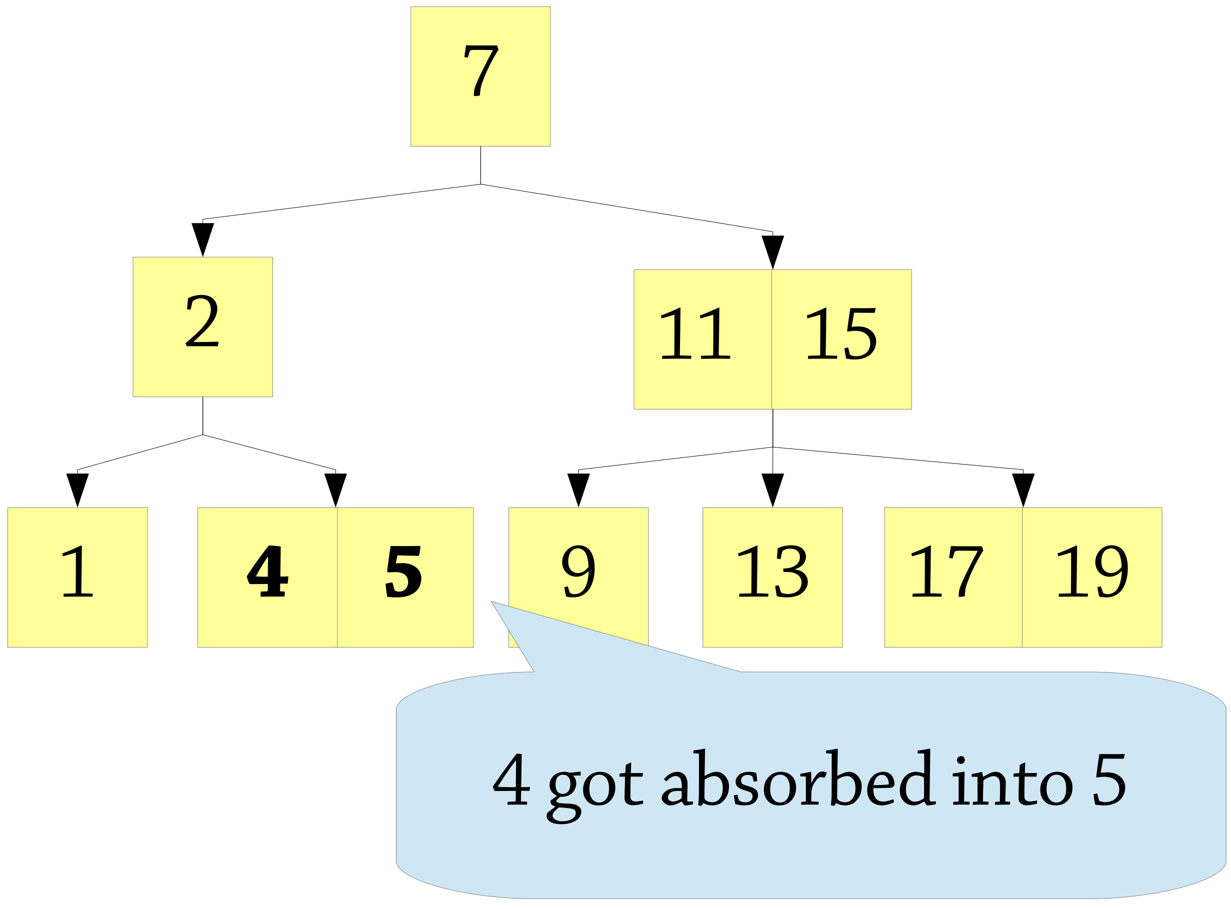
Insertion into a 2-3 tree

To insert a value (e.g. 4) into a 2-3 tree, we start by doing a normal insertion...



Insertion into a 2-3 tree

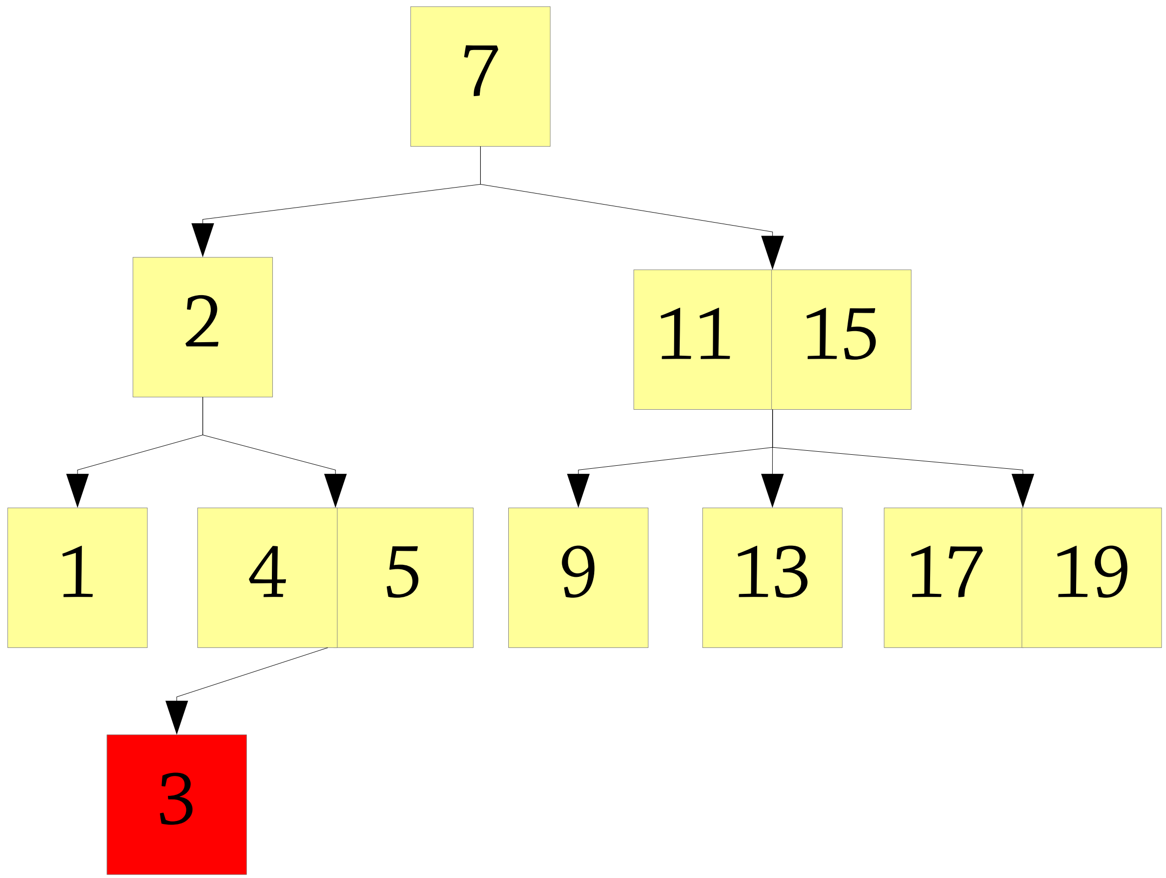
To fix the balance invariant, we *absorb* the bad node into its parent!



Insertion into a 2-3 tree

Now suppose we want to insert 3.

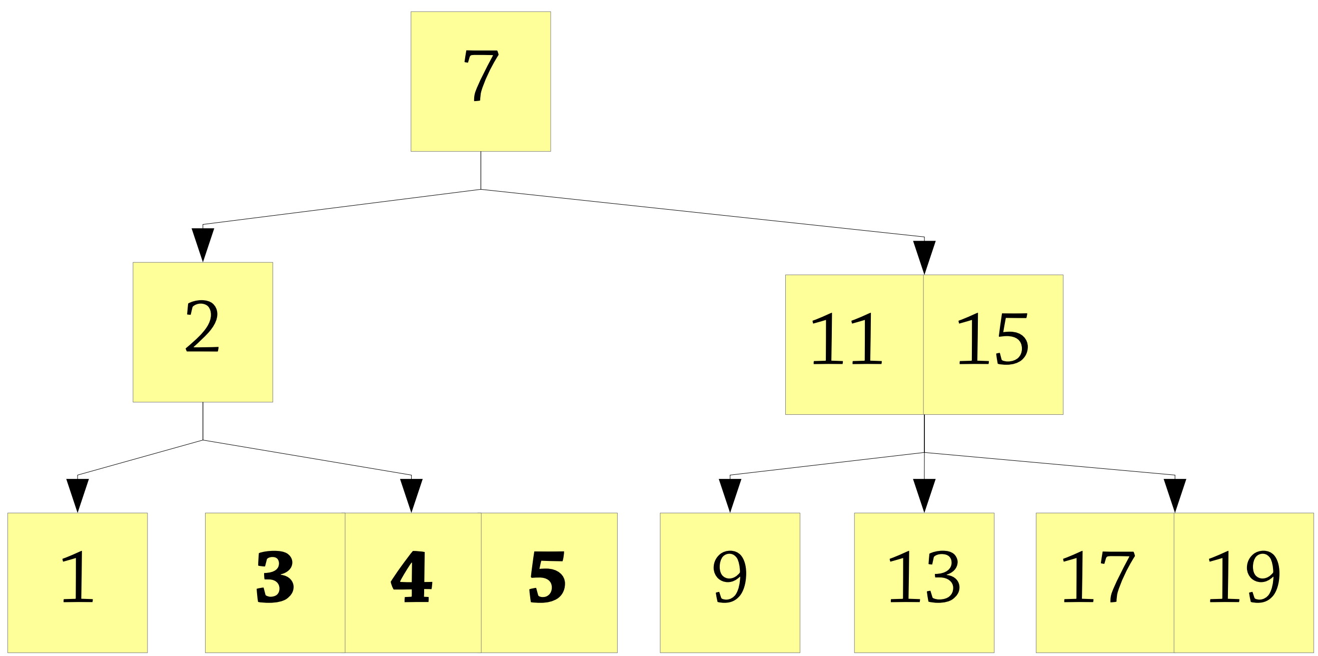
We'll absorb it into its parent as before...



Insertion into a 2-3 tree

We get a 4-node, which is not allowed!

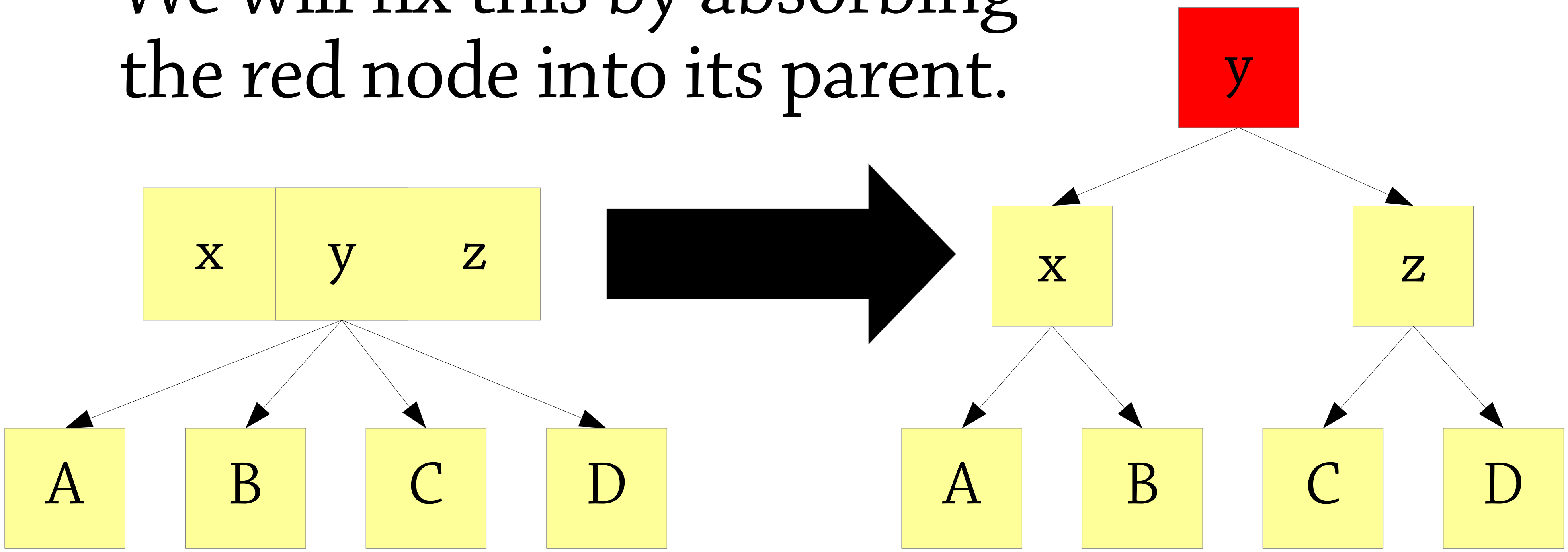
We fix this by a method called *splitting*.



Splitting a 4-node

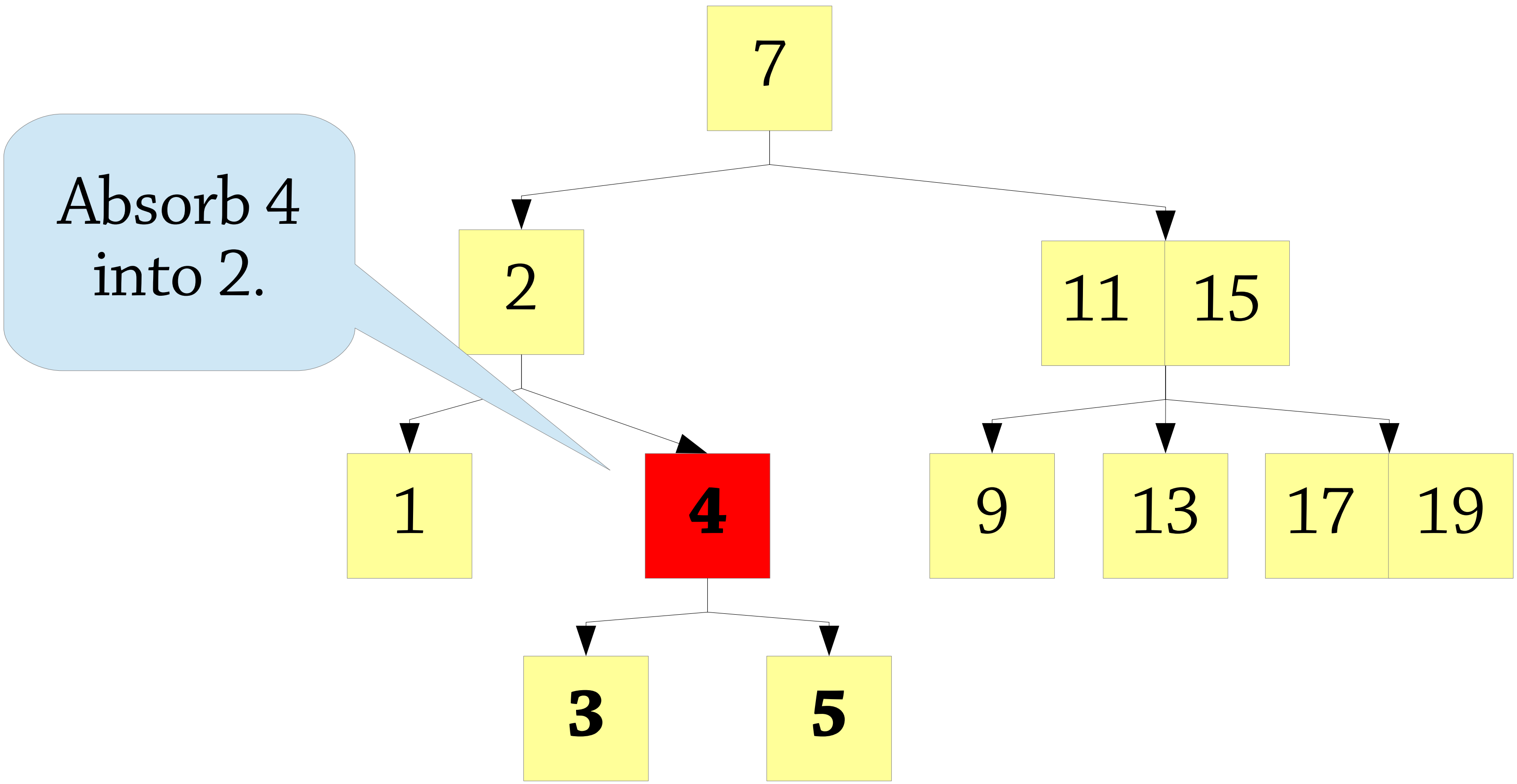
To get rid of a 4-node, we *split* it into several 2-nodes!

This creates an extra level in the tree. We will fix this by absorbing the red node into its parent.



Insertion into a 2-3 tree

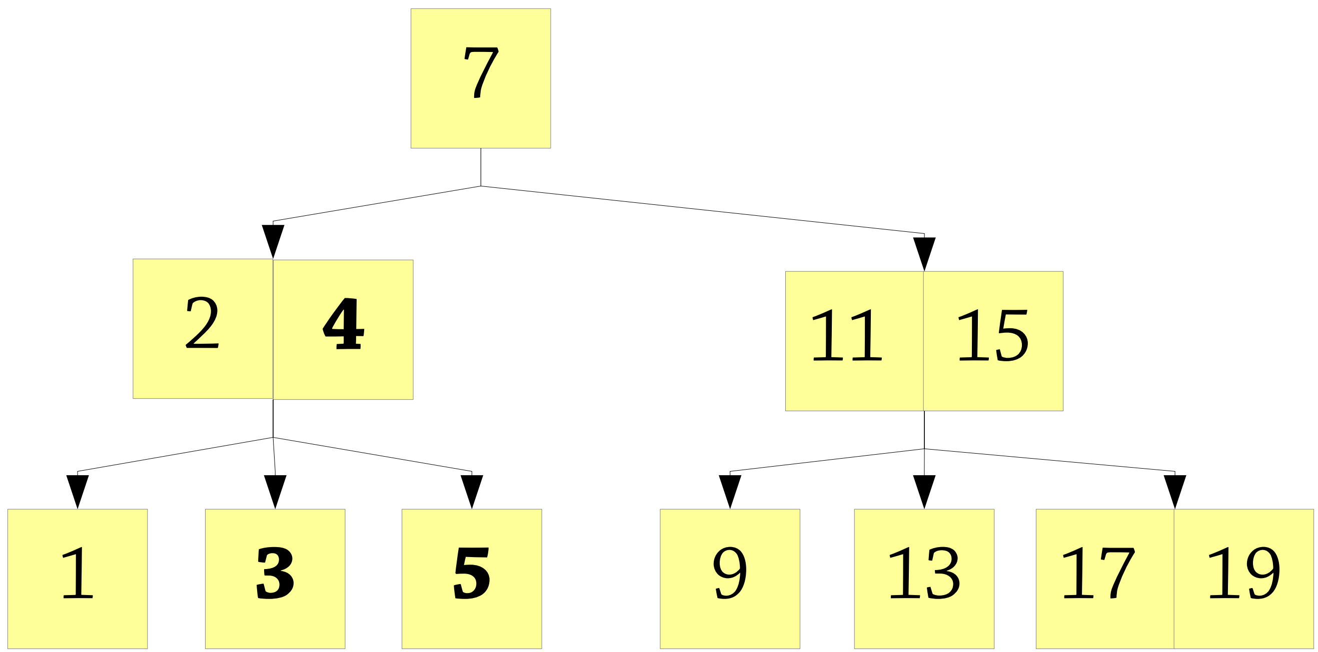
After splitting, we absorb the “extra level” node into its parent.



Insertion into a 2-3 tree

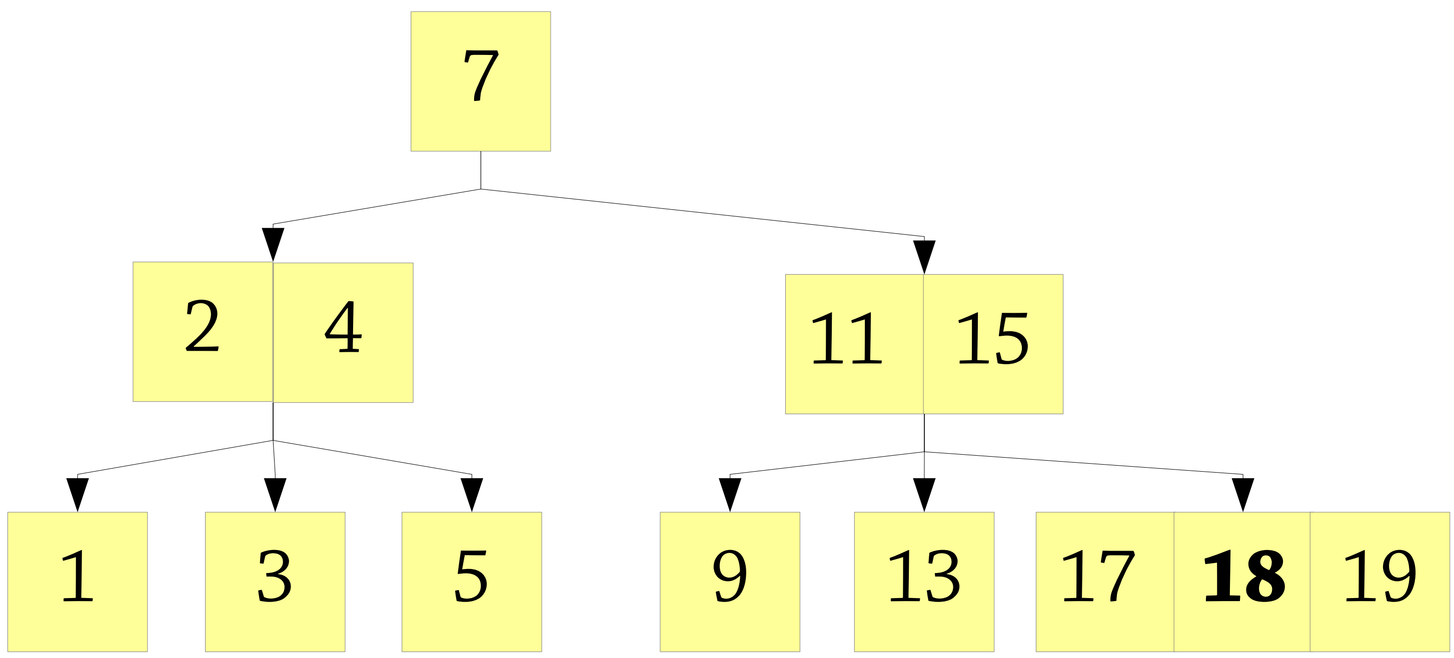
We restored the invariant!

Let's try inserting 18.



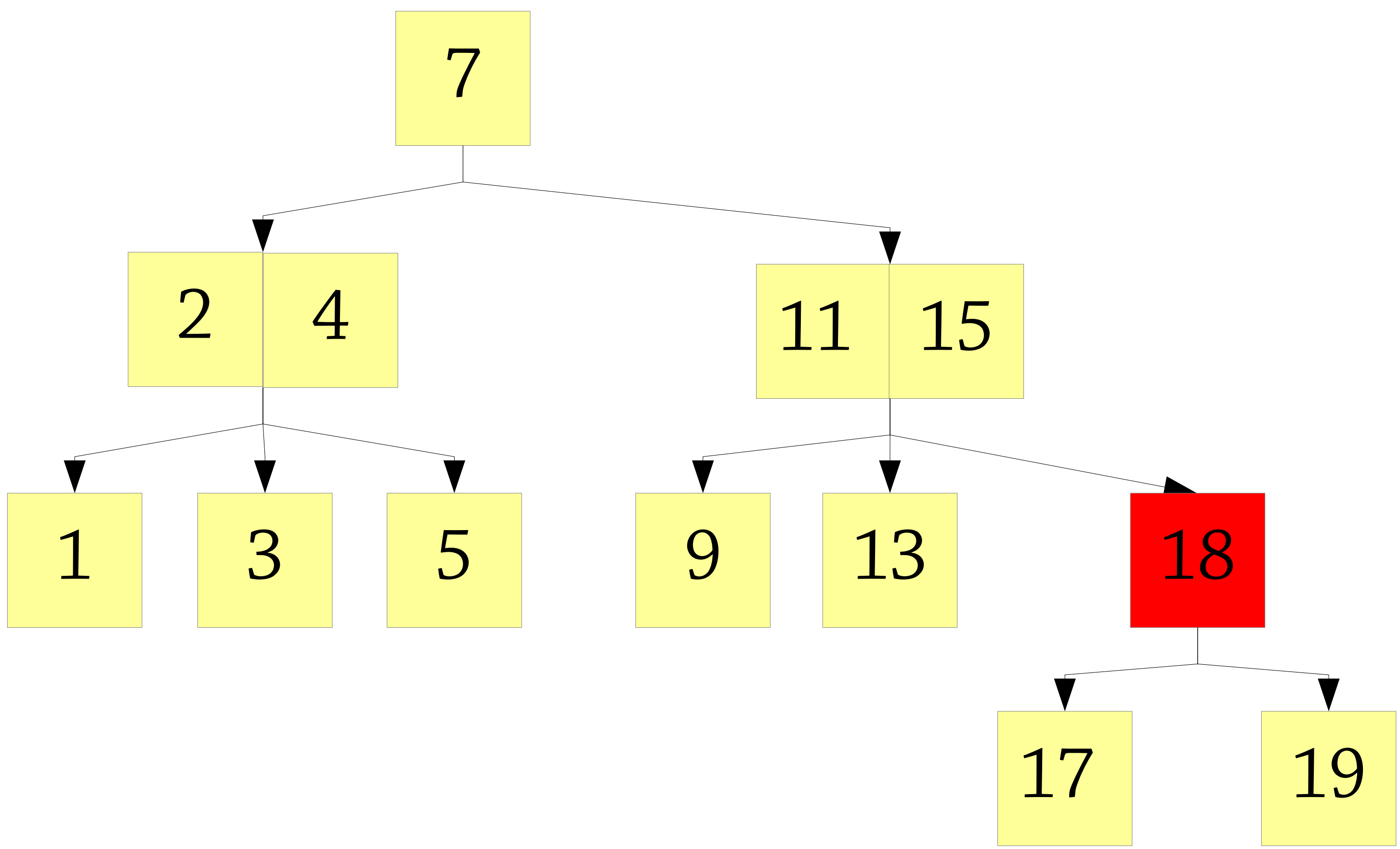
Insertion into a 2-3 tree

First add and absorb.
We got a 4-node, split it.



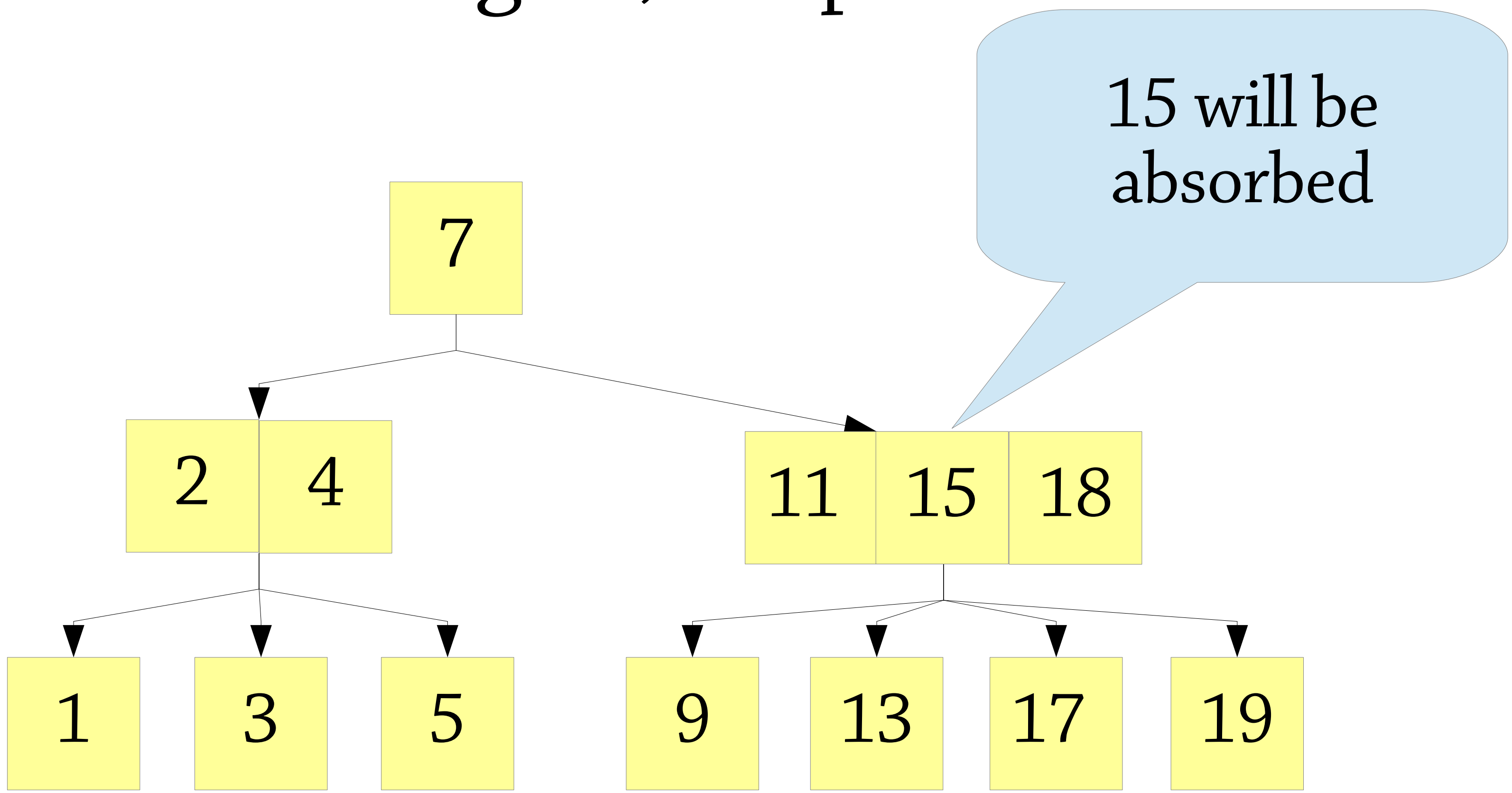
Insertion into a 2-3 tree

Absorb the extra node into the parent.



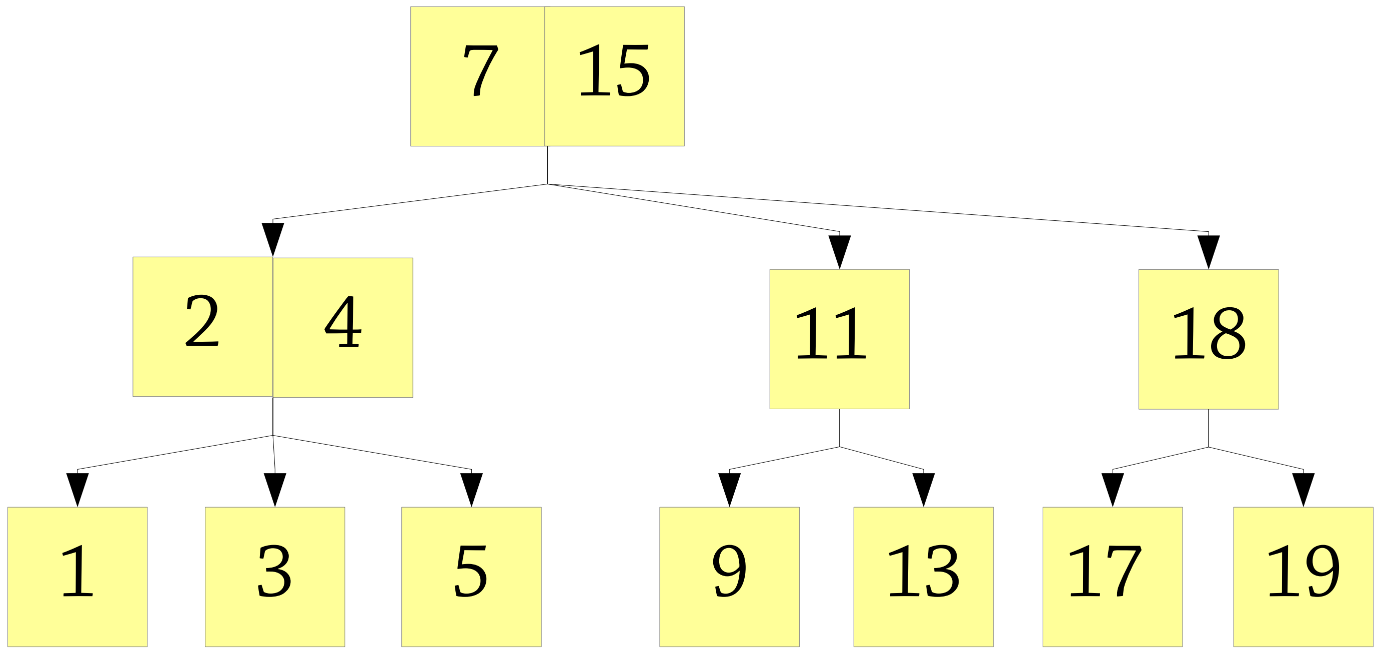
Insertion into a 2-3 tree

We got a 4-node again, so split and absorb.



Insertion into a 2-3 tree

Done! (If we insert even more, eventually the root will split, which adds a new level to the tree.)



2-3 insertion algorithm

Insert the new node into the tree

Then alternate 2 steps:

- *absorb* the node into its parent, move up to the parent
- if the node is a 4-node, *split* it into 2-nodes

Stop once you don't need to split

2-3 trees, implementation (?)

```
class Node<E> {
  boolean isTwoNode;
  E value, secondValue;
  Node<E> left, right, middle;

  boolean member(E key) {
    if (key.compareTo(value) == 0) return true;
    else if (key.compareTo(value) < 0)
      return left.member(value);
    else {
      if (!isTwoNode) {
        if (key.compareTo(secondValue) == 0)
          return true;
        else if (key.compareTo(secondValue) < 0)
          return middle.member(value);
      }
      return right.member(value);
    }
  }
}
```

Space wasted by storing two values even in 2-nodes, fixing this is annoying

Lots of cases compared to BSTs

Even worse: insertion temporarily creates 4-nodes!

2-3 trees, summary

2-3 trees do not use rotation, unlike balanced BSTs – instead, they keep the tree perfectly balanced

- Invariant maintained using *absorption* (to remove unwanted nodes) and *splitting* (to eliminate 4-nodes)

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

Conceptually much simpler than AVL trees!

But implementation is really annoying :(

- Fix this by using AA trees, next

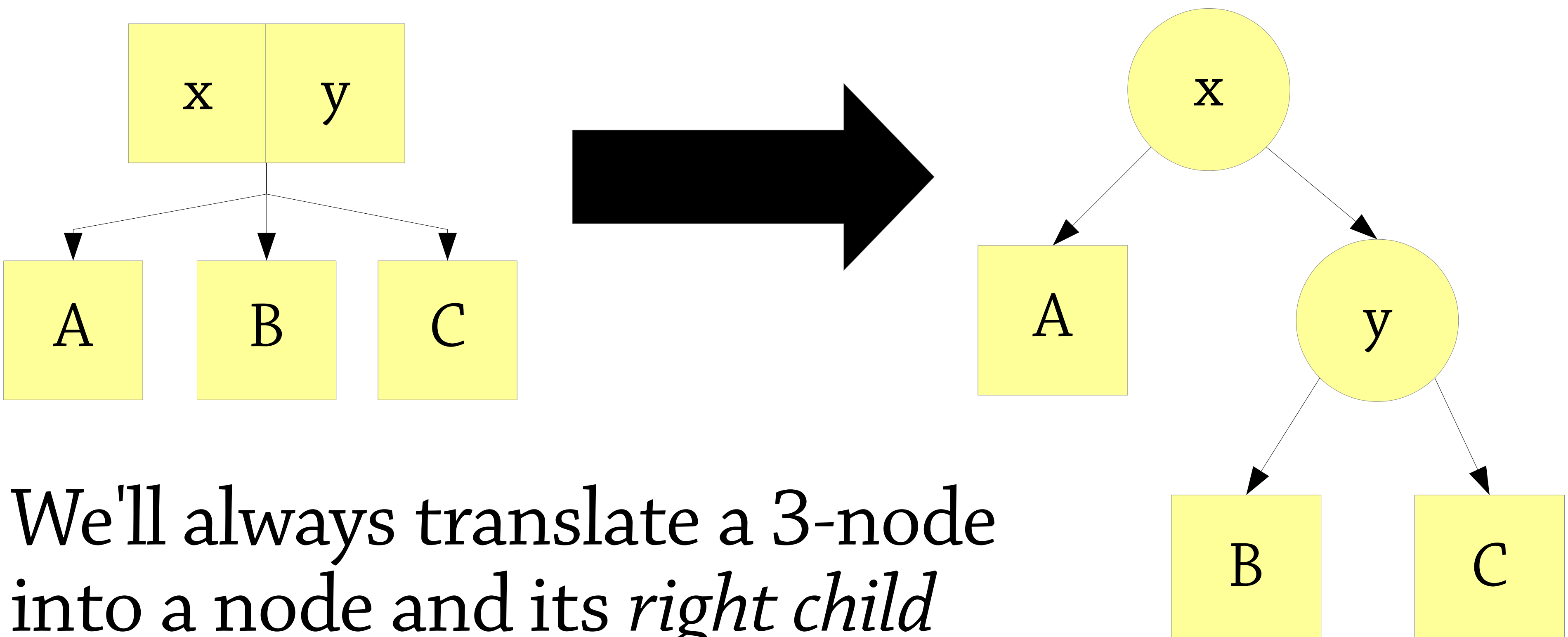
AA trees

AA trees

AA trees implement a 2-3 tree using a BST!

A 2-node becomes a BST node

A 3-node becomes *two* BST nodes:



We'll always translate a 3-node into a node and its *right child*

AA trees, the plan

An AA tree is really a 2-3 tree, but we store it in a binary search tree

- A bit like what we did for binary heaps

We'll need to add extra information to the nodes, and invariants, so that:

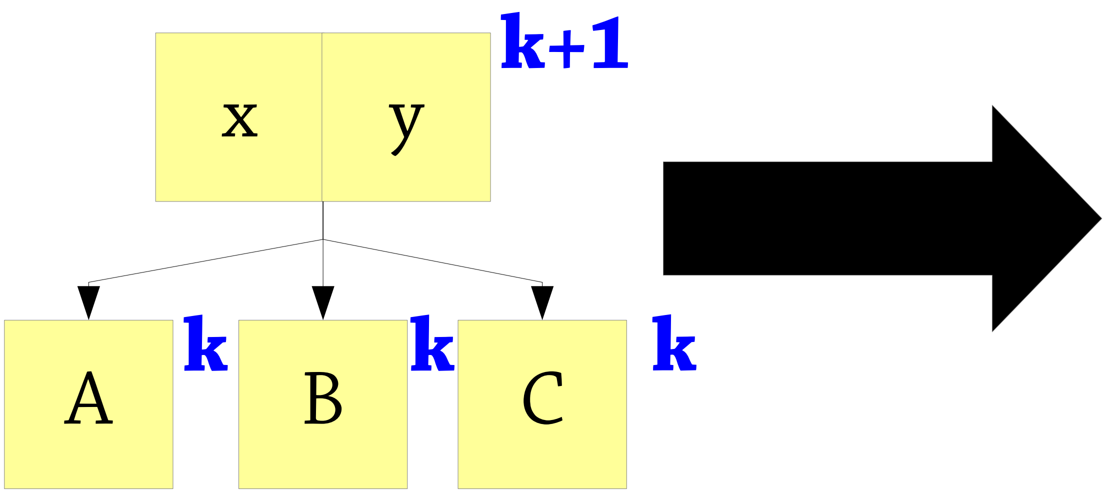
- Any AA tree must correspond to a 2-3 tree
- We can tell whether each node in the tree is a 2-node, or part of a 3-node

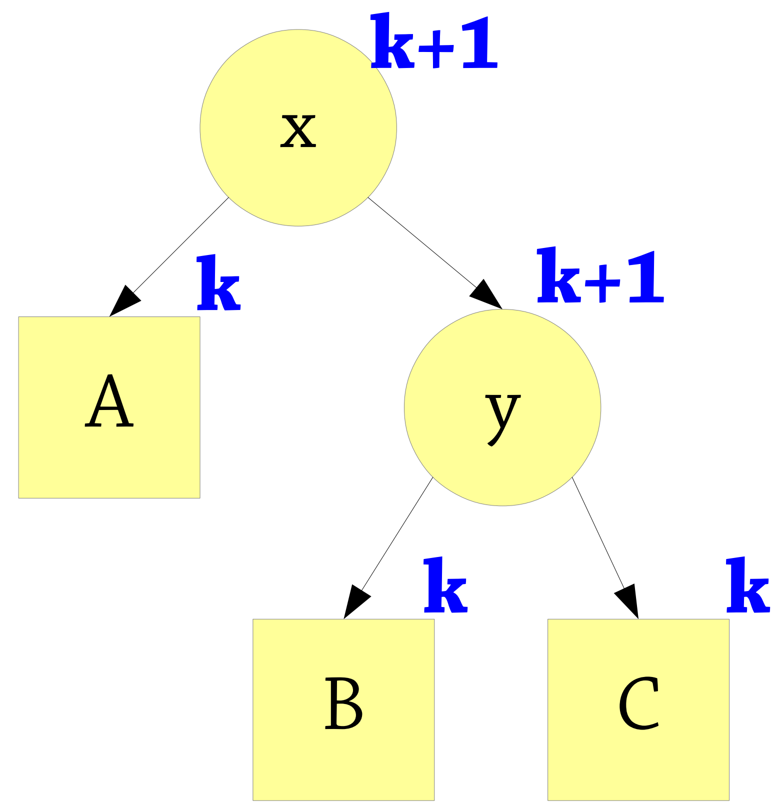
Then we can adapt 2-3 insertion to AA trees!

- For searching, we can just use BST search

AA trees

We store with each node a *level*, which is the height of the corresponding 2-3 tree node

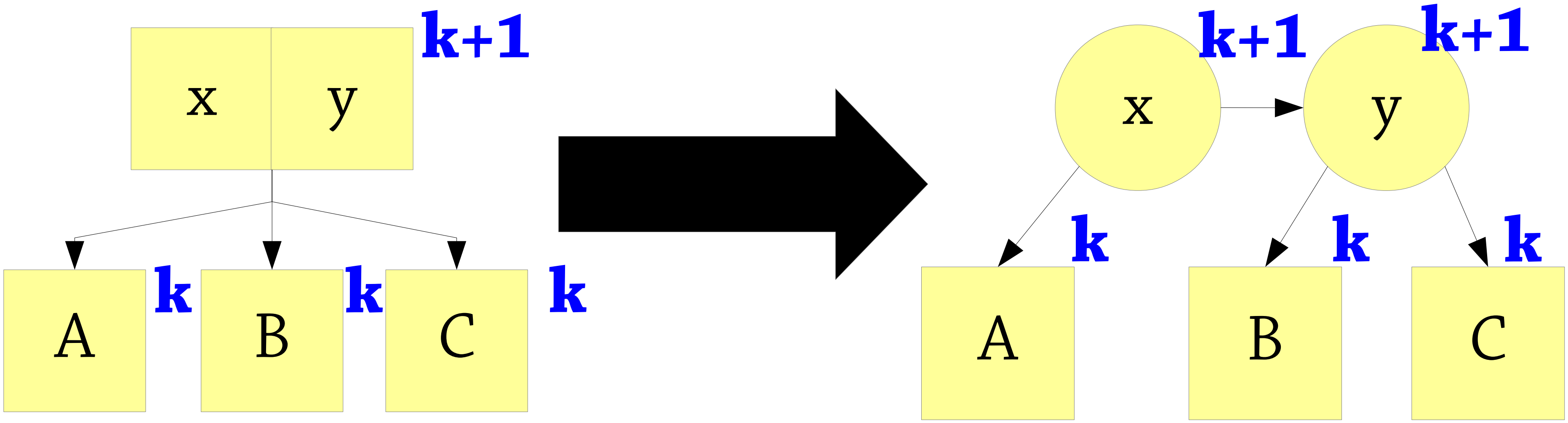




Notice that x and y have the same level. That's how we can tell they represent a 3-node. Our invariant will talk about levels.

AA trees

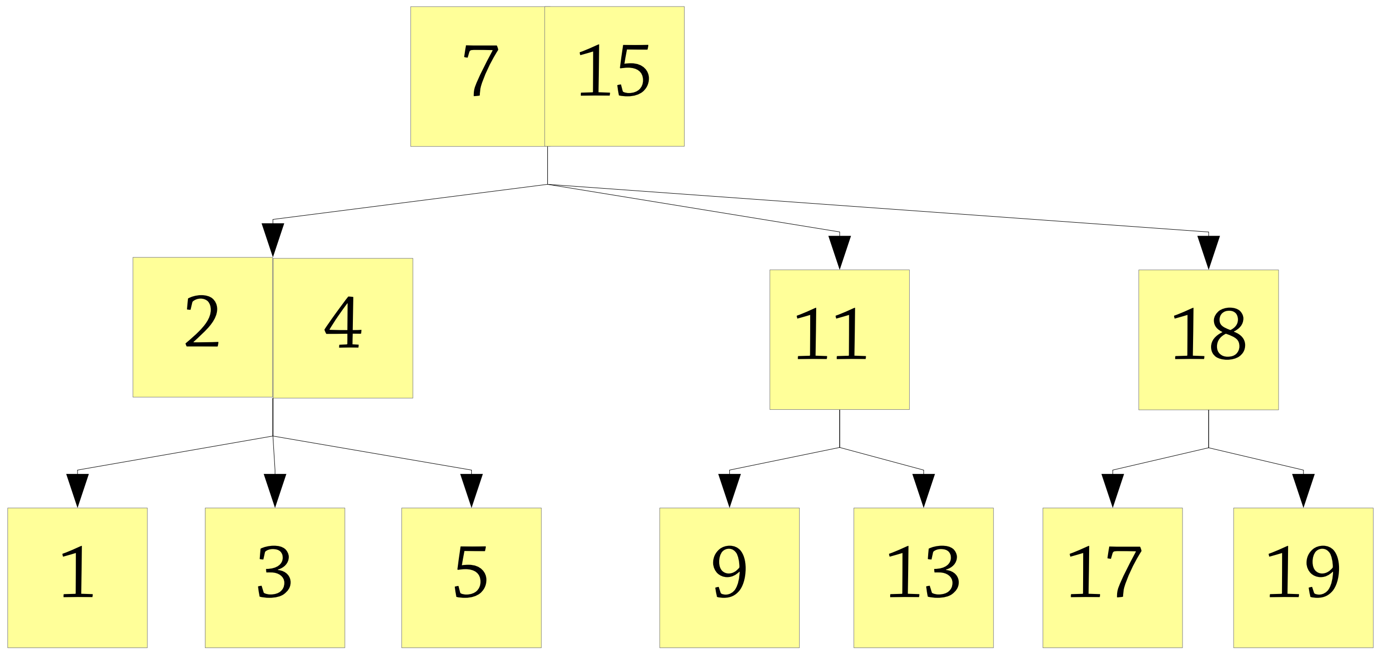
If a node has the same level as its parent, we'll draw them next to each other.



This emphasises the levels in the tree.

2-3 trees as AA trees

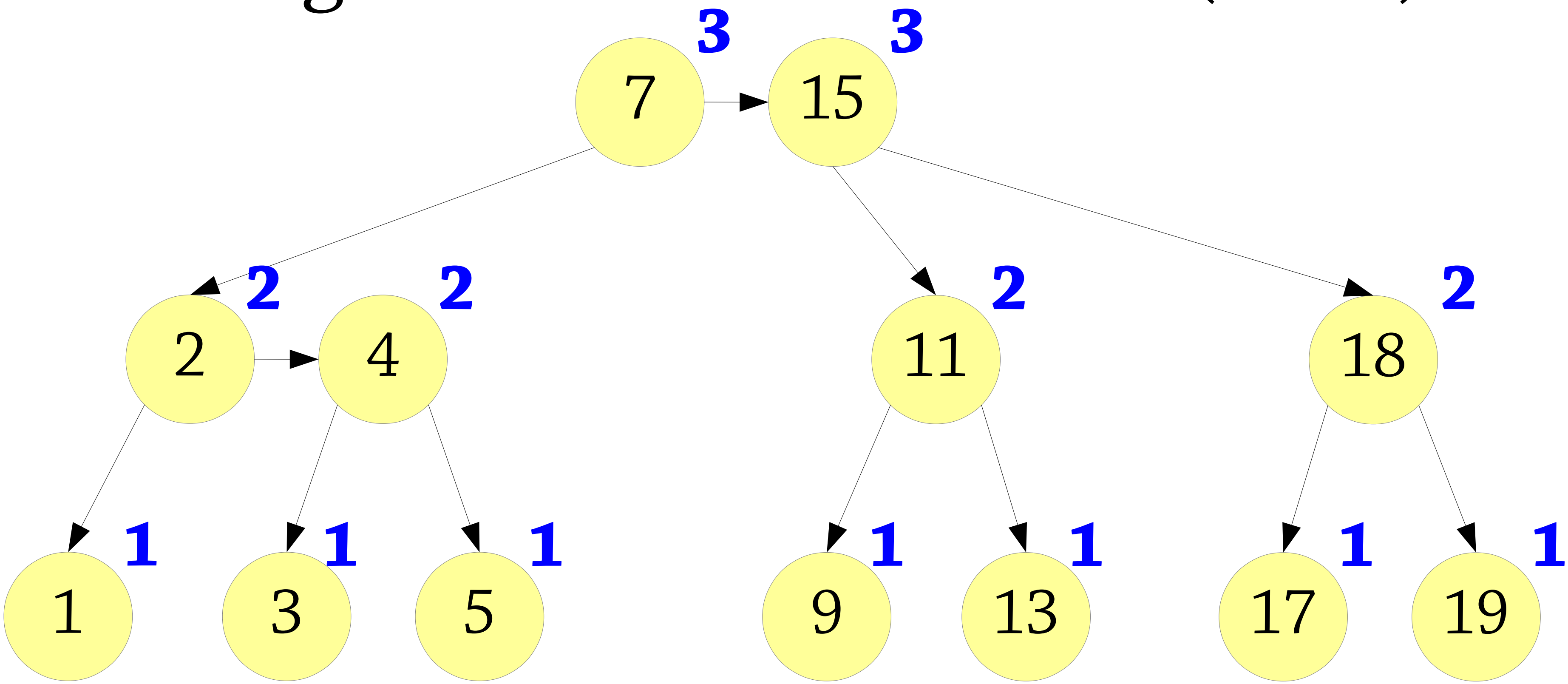
Here was the 2-3 tree from before...



2-3 trees as AA trees

...and here is the corresponding AA tree!

We can identify the 2- and 3-nodes by looking at the level of the nodes (how?)



AA trees

We can translate a 2-3 tree to an AA tree

And, by looking at the levels, we can go the other way

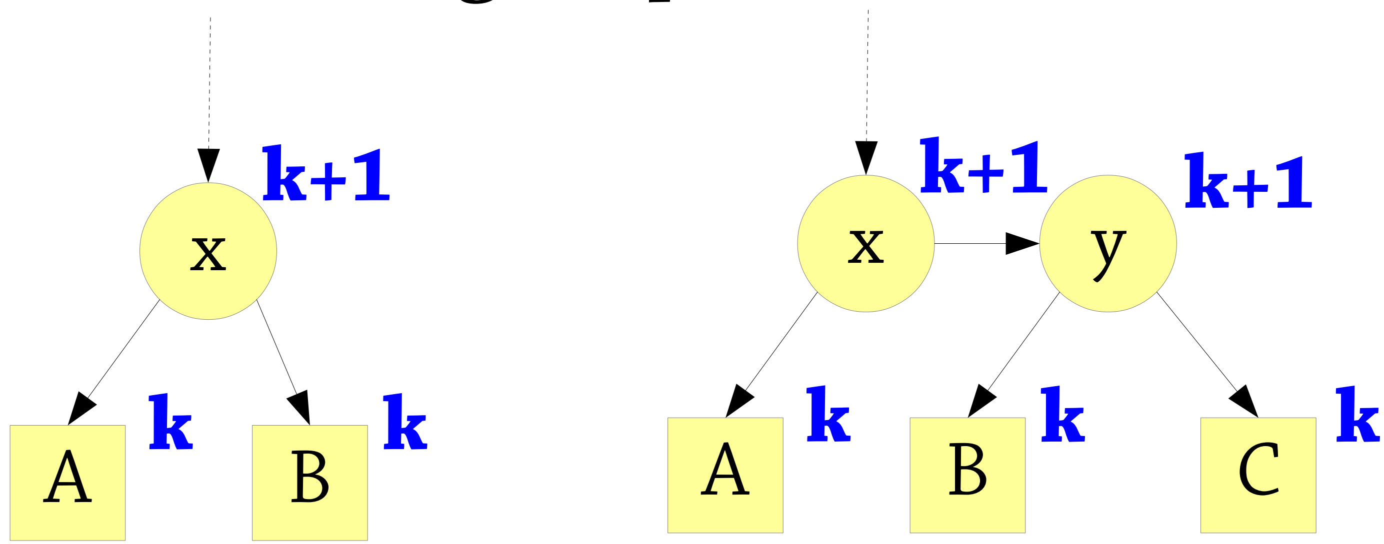
- If a node has the same level as its right child, the two nodes together make a 3-node
- Otherwise it's a 2-node

Now we need an invariant to check that:

- We only have 2-nodes and 3-nodes
- The levels match the heights in the 2-3 tree
- The 2-3 tree is perfectly balanced

AA tree invariant, a first attempt

An AA tree must be built up only from subtrees of the following shape:



Notice that the level of x/y must be exactly one more than the level of A/B/C

(we consider null to have a level of 0 – this means a leaf must have a level of 1)

AA tree invariant, part 1

It turns out to be better to break this invariant into pieces, so that it says something about each BST node

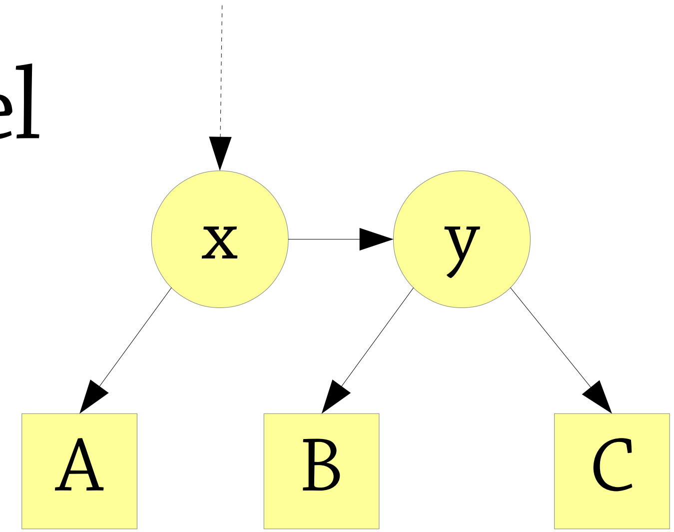
First, the level of a child node in the BST must be either:

- equal to the level of its parent, or
- one less than the level of its parent

(where the level of null is 0)

AA tree invariant, part 2

If a node has the same level as its child, it must be the root of a 3-node.

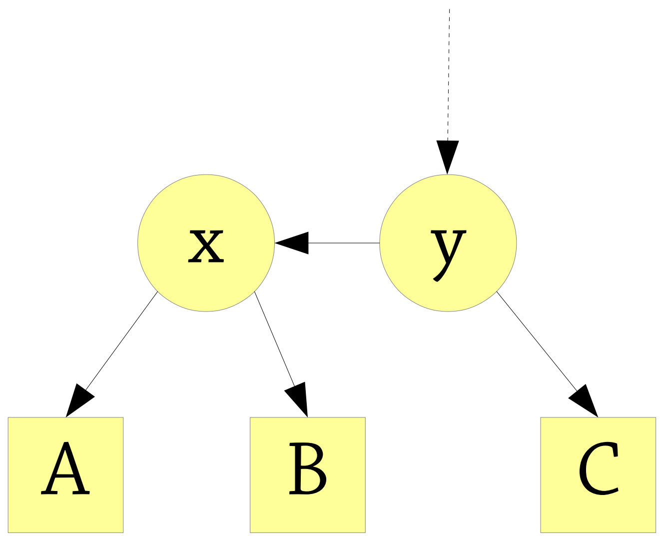


So we can say:

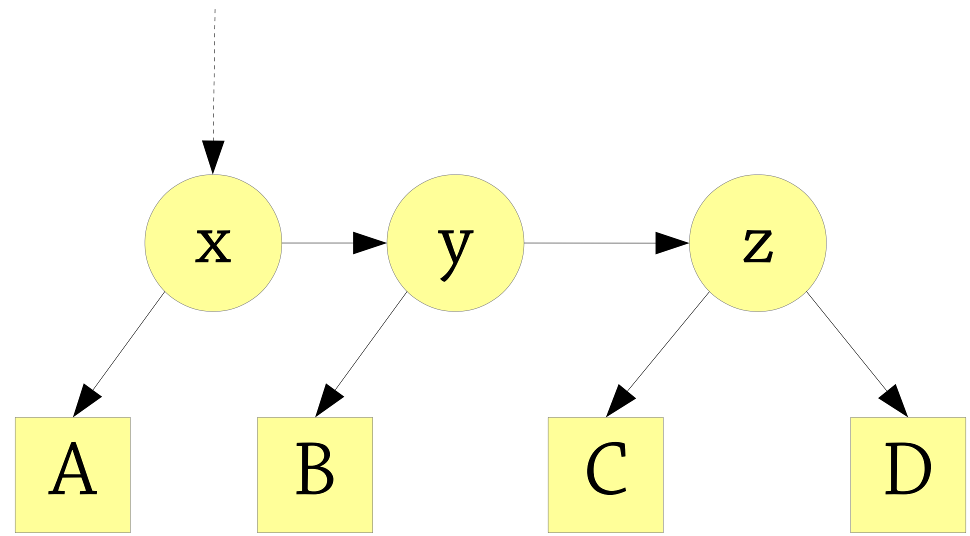
- A node's level must be greater than its left child: level(node) > level(node.left)
- And also greater than its right-right grandchild: level(node) > level(node.right.right)

AA tree – not allowed

Bad: malformed 3-node (left child at same height)



Bad: 4-node (right grandchild at same height)



We'll get these trees during insertion!

AA tree invariant, summary

We consider the level of null to be 0

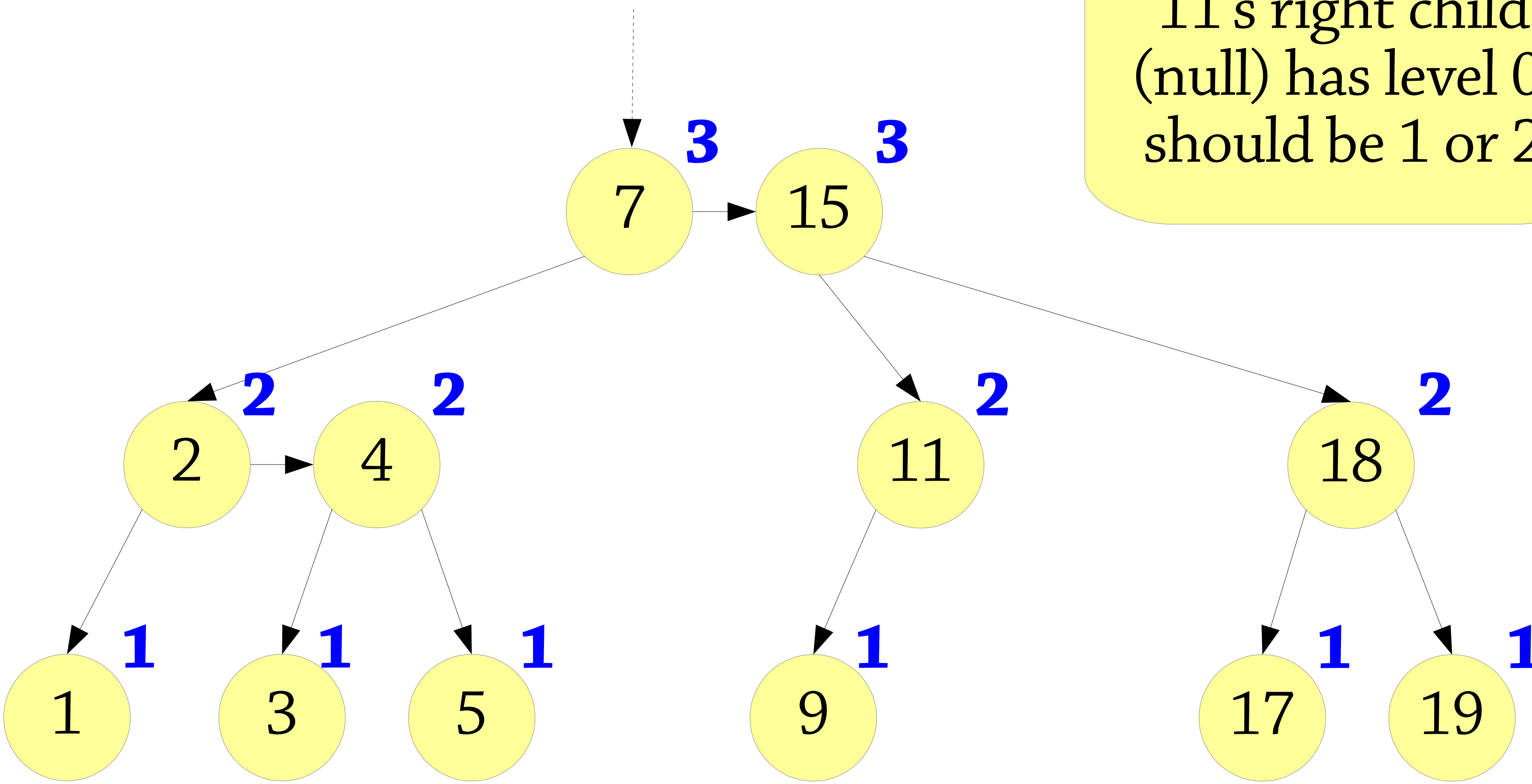
For each node in the tree, the following must hold:

- The node's children must have a level either equal to or one less than the node itself
- level(node) > level(node.left)
  (x ← y not allowed)
- level(node) > level(node.right.right)
  (x → y → z not allowed)

This implies that any leaf node has a level of 1

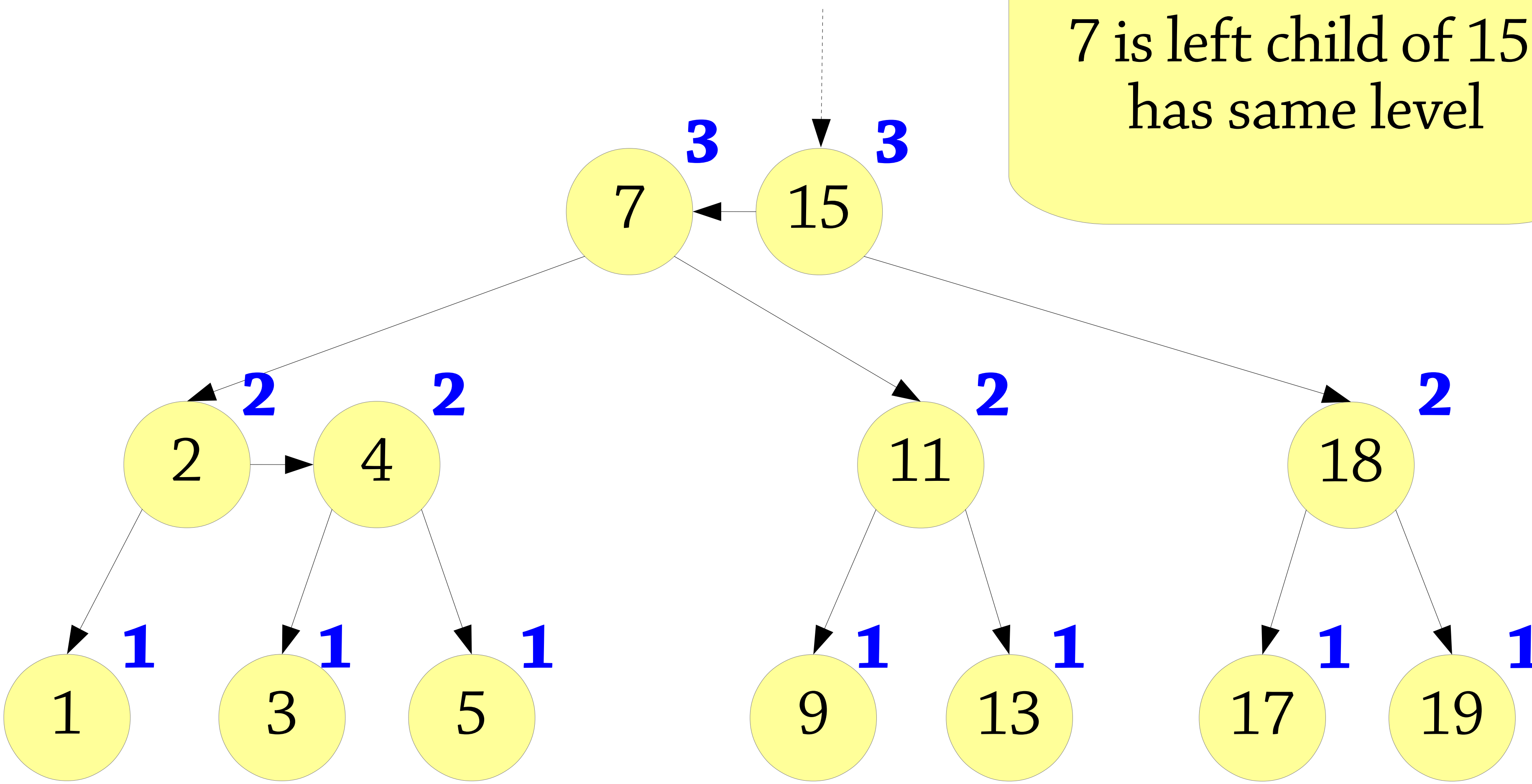
We also have the normal BST invariant!

Why is this not an AA tree?



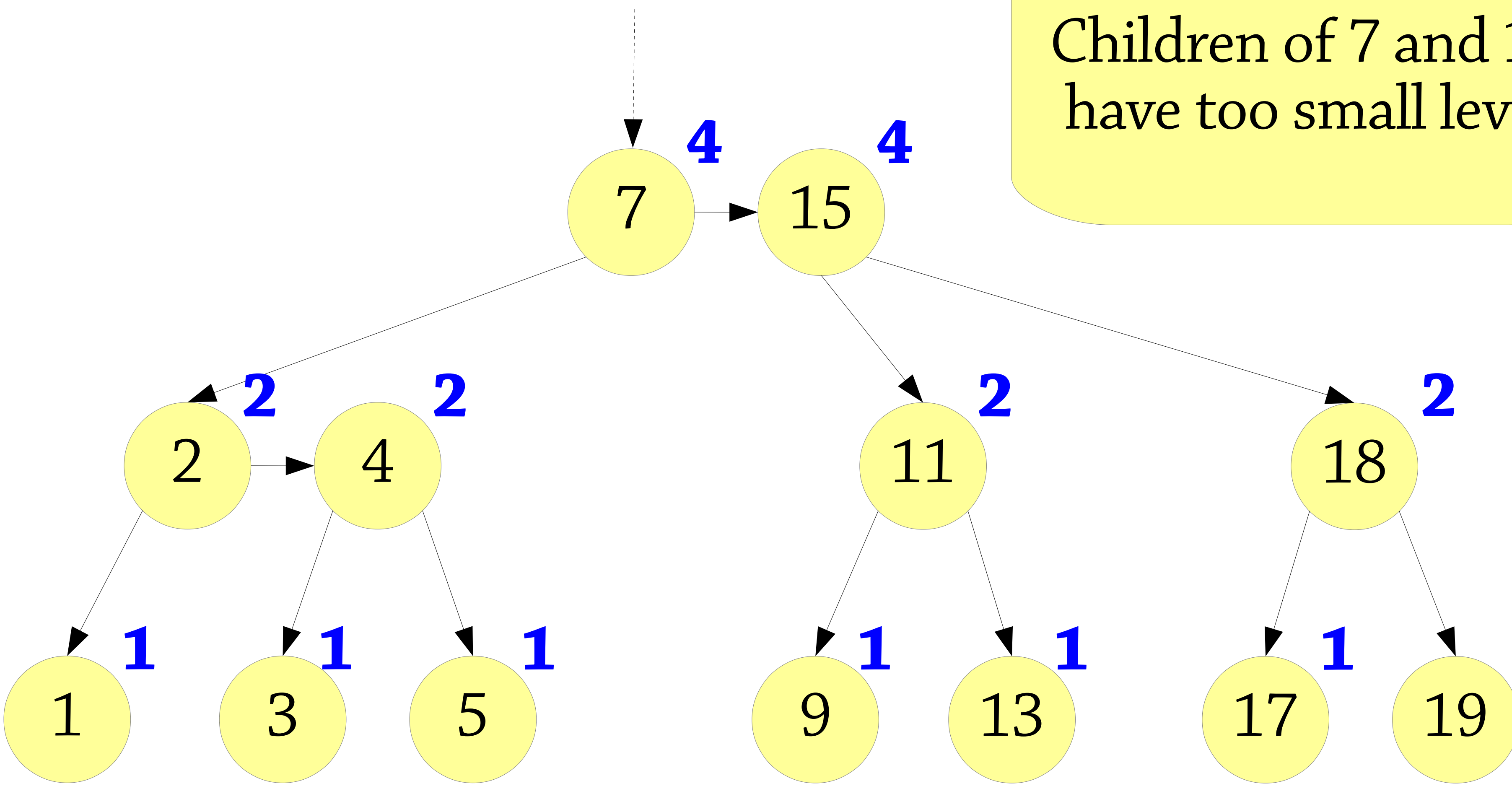
11's right child (null) has level 0, should be 1 or 2

Why is this not an AA tree?



7 is left child of 15,
has same level

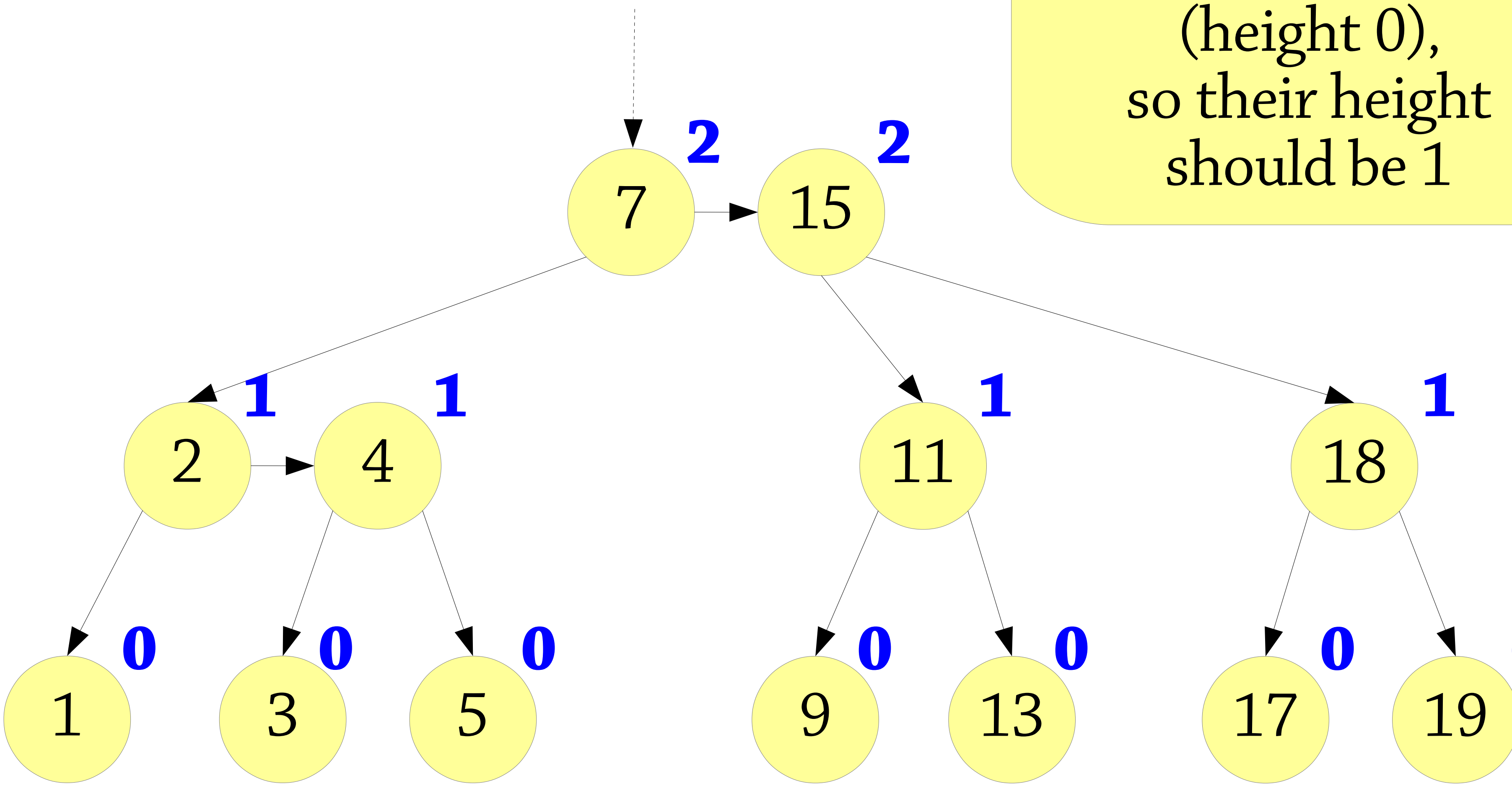
Why is this not an AA tree?



Children of 7 and 15 have too small level

Why is this not an AA tree?

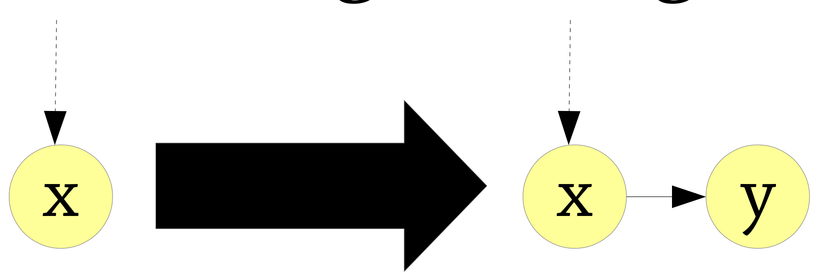
Leaf nodes have left child null (height 0), so their height should be 1



AA tree insertion

To insert into an AA tree, we start with a normal BST insertion. The new node is a leaf so we give it a level of 1. Note that its parent also has level 1 (why?)

If we are lucky the parent was a 2-node and we insert into the right of it, giving a 3-node:

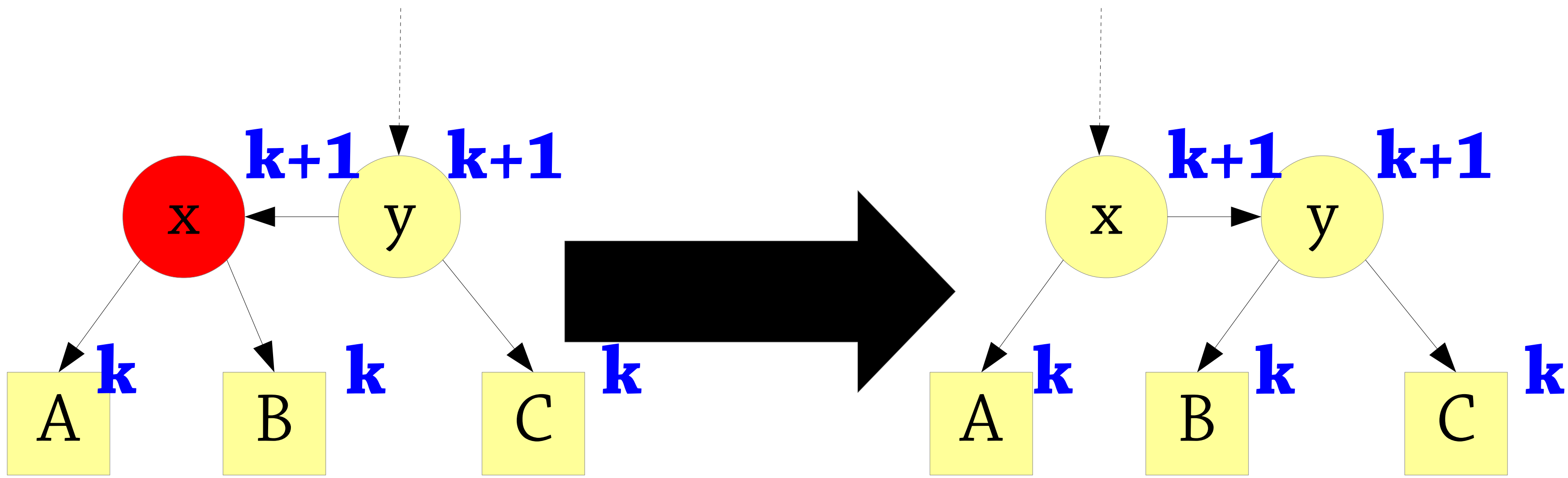


Otherwise, the invariant is broken.
But there are only two ways it can break!

Case 1: skew

Here, we have inserted into the *left* of a 2-node, breaking the invariant.

We can fix it by doing a right rotation!



This operation is called *skew*.
We do it whenever the new node is the left child of its parent.

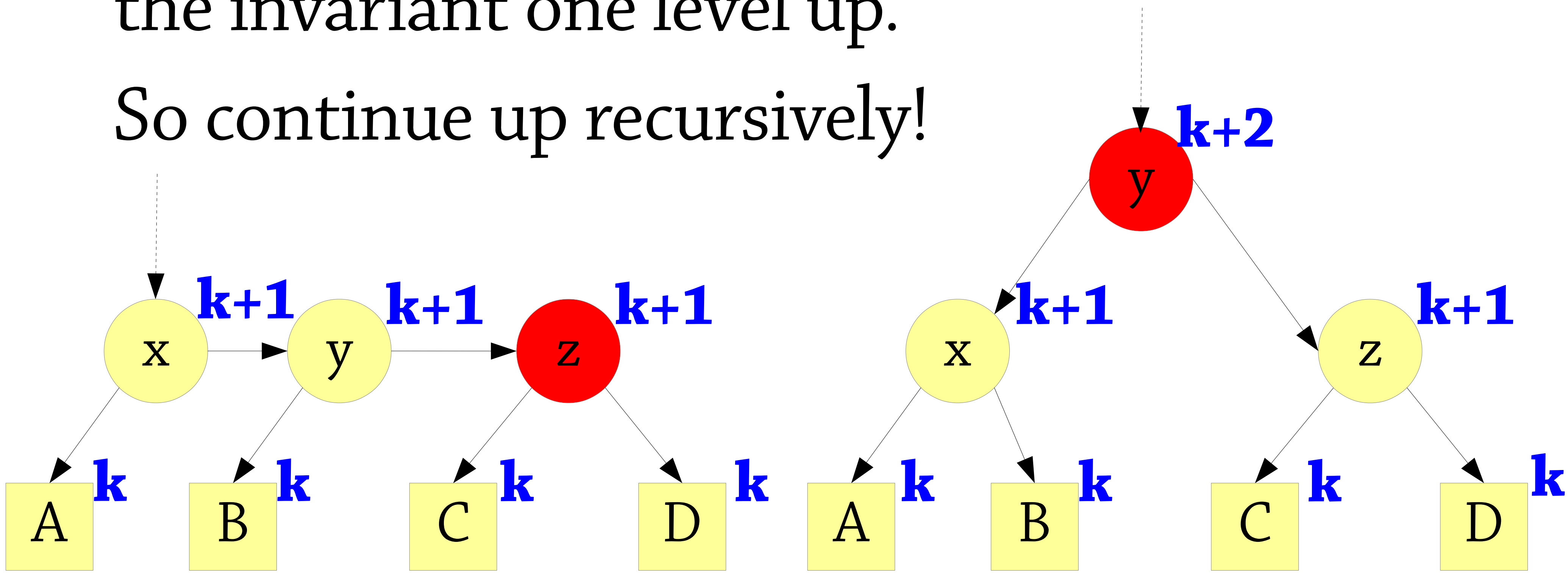
Case 2: split

Here, insertion created a 4-node.

We can split it into 2-nodes!

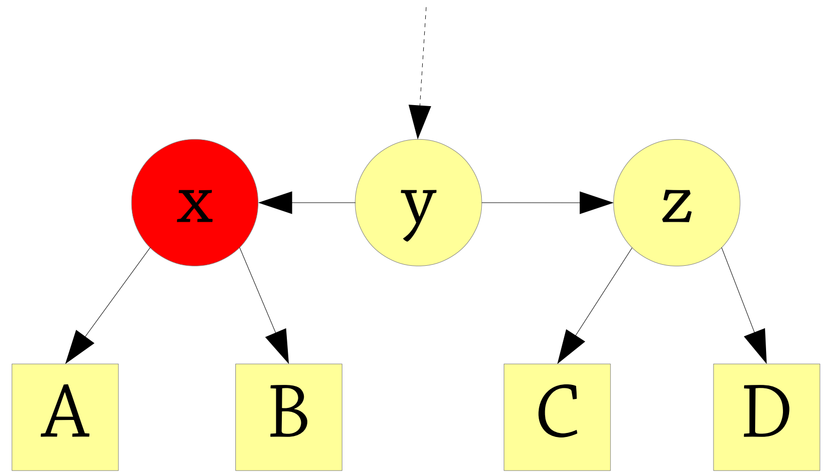
Notice that y's level increases – may break the invariant one level up.

So continue up recursively!



All other cases: skew and split

Insertion can also create a 4-node like this:

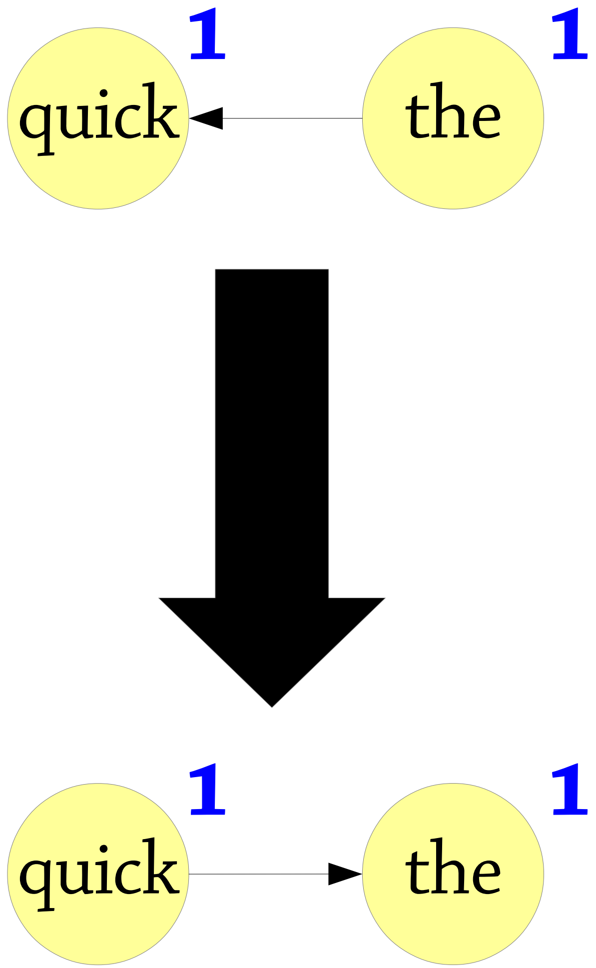


But, if we do a skew, this turns into the previous kind of 4-node!

To cover all the cases we just have to:
first skew if the left child is bad,
then split if the right grandchild is bad

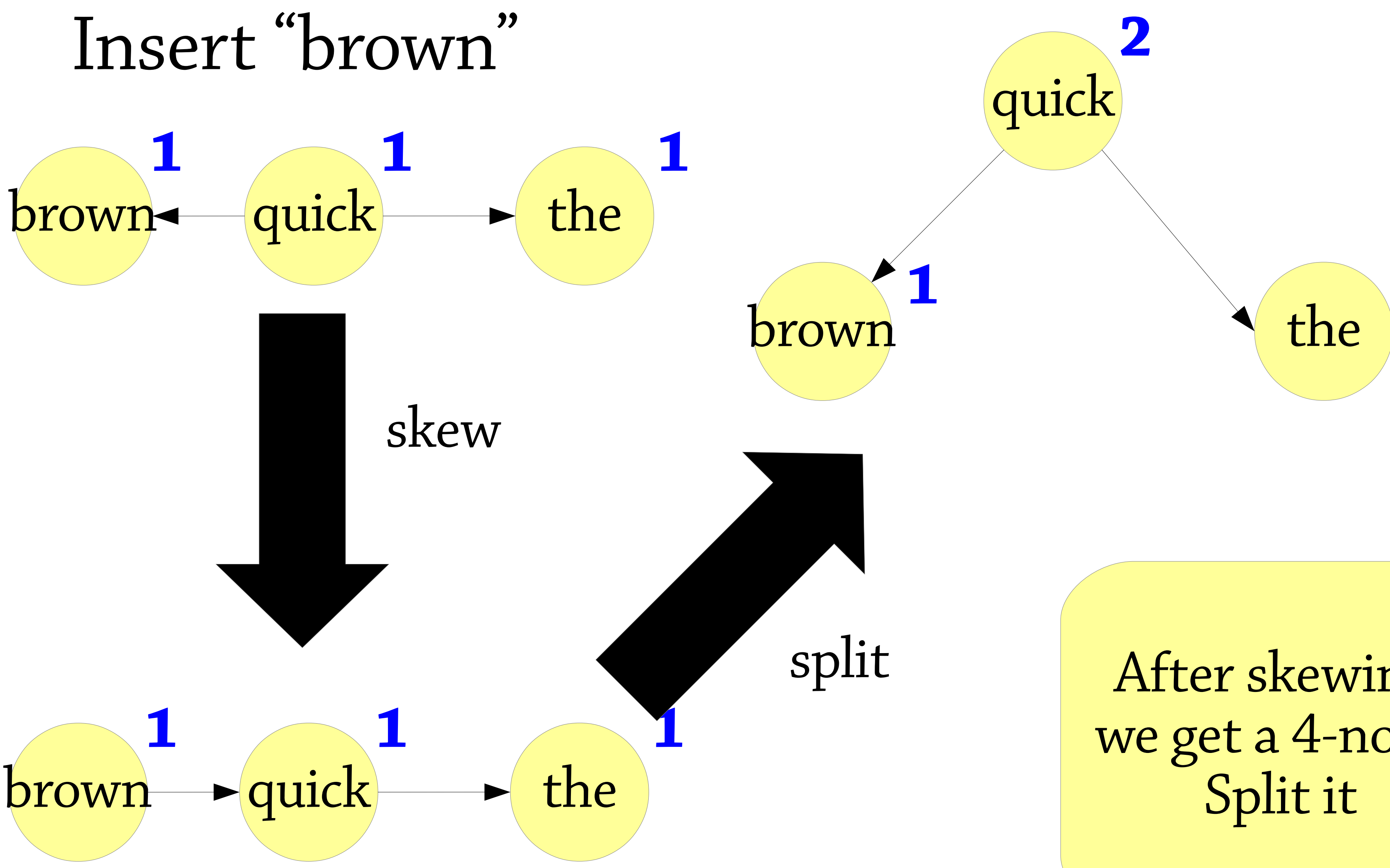
Example: the quick brown fox...

Insert “quick” into “the”



Left child at
same level!
Skew to fix it
(rotate right)

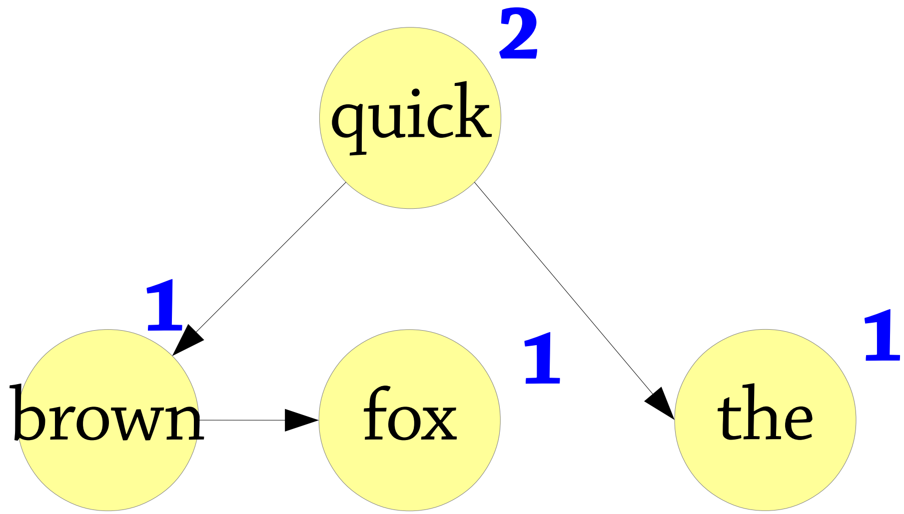
Example: the quick brown fox...



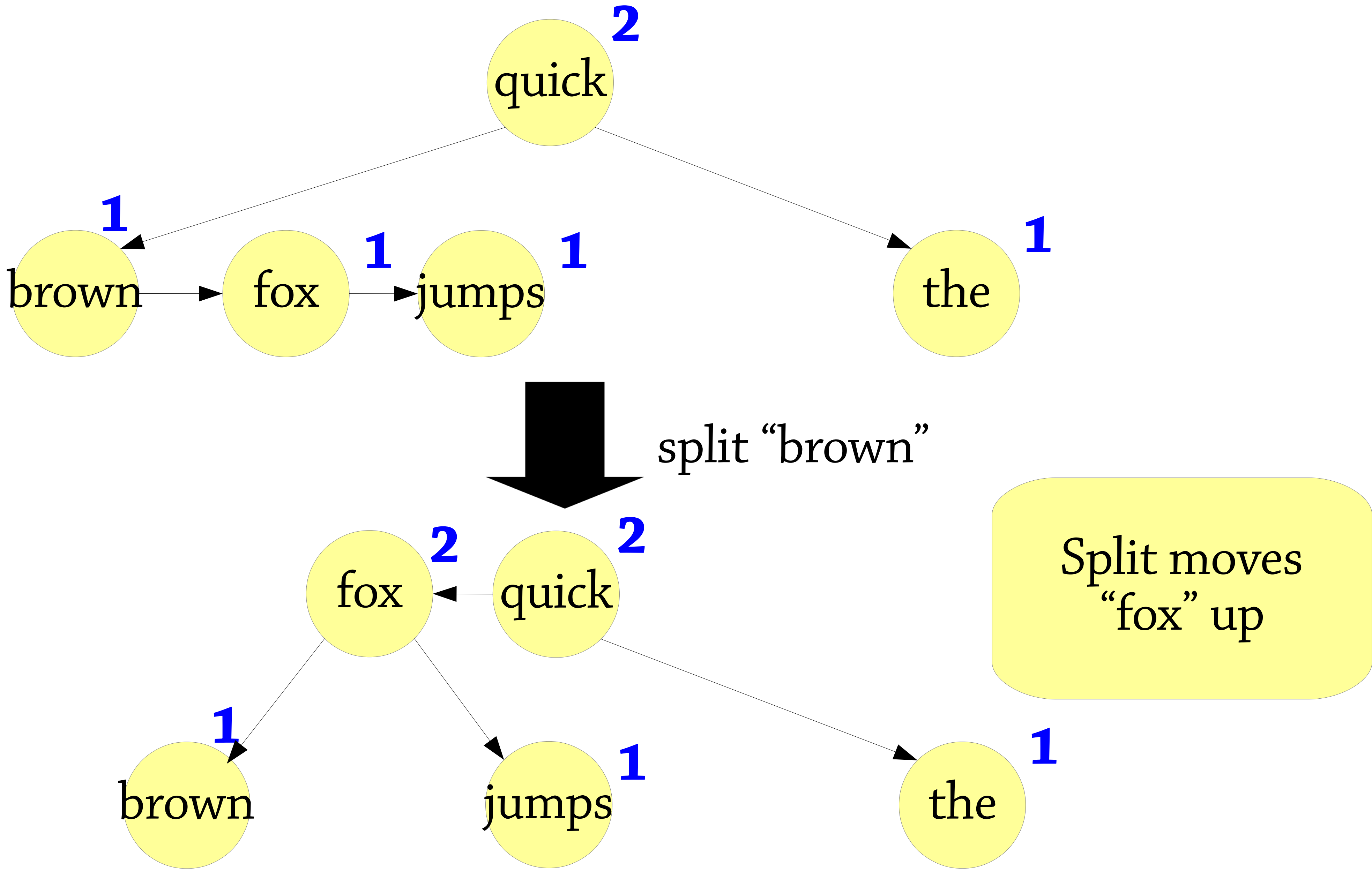
After skewing, we get a 4-node! Split it

Example: the quick brown fox...

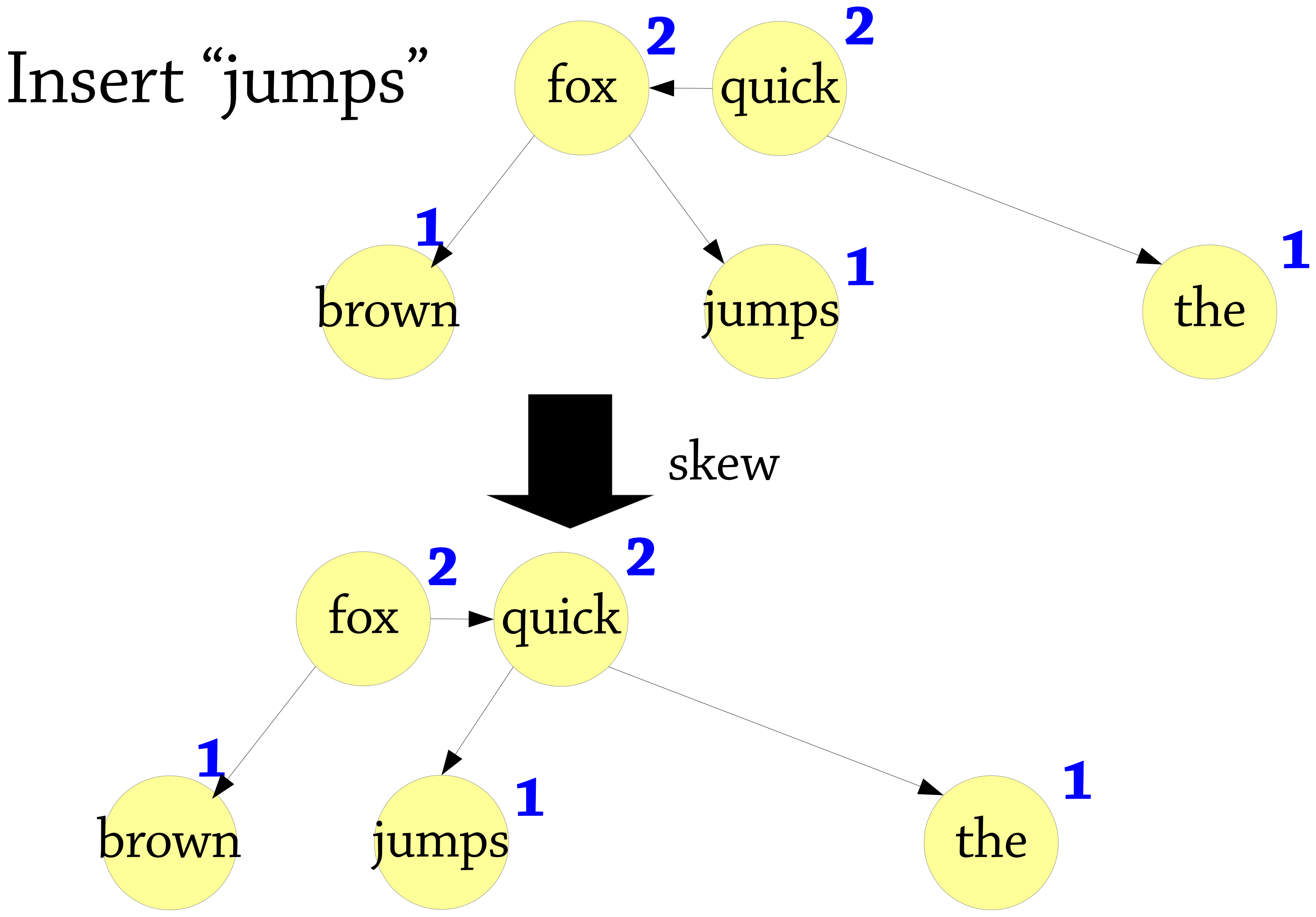
Insert "fox"



Insert “jumps”



Example: the quick brown fox...



AA trees – looking back

There are only two ways that insertion can break the invariant

- Making a left child with the same height as its parent – skew it
- Making a 4-node – split it

Why skew then split? Because skewing ensures there's only one possible way to represent a 4-node

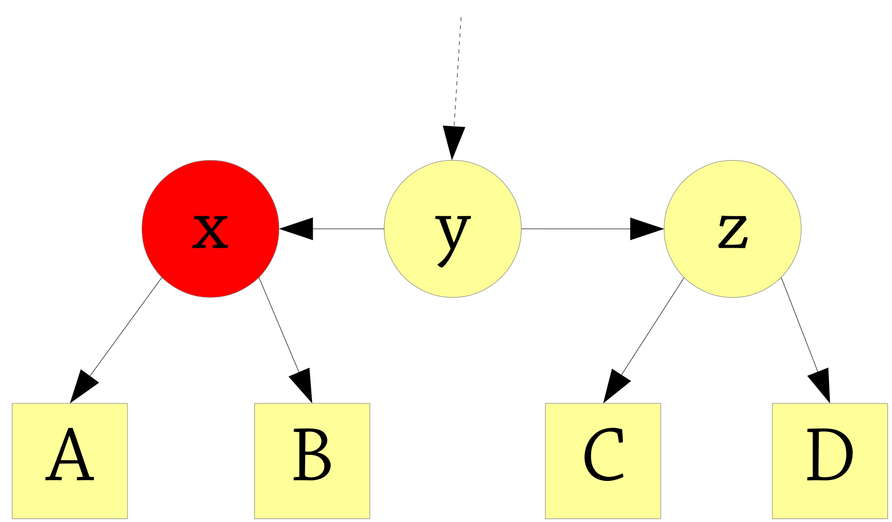
When we split, the level of the top node increases – this corresponds to absorption in a 2-3 tree

AA trees – implementation

The level is stored as part of each node.

Looking at the diagrams, the level changes when you do a *split* – so make sure to do this

Easiest way to implement it:
have separate functions for skew and split,
call them from insert. But first skew then split, to take care
of this case:



AA versus AVL trees

AA 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 no real difference in practice

Another disadvantage: deletion requires a fair amount of extra work

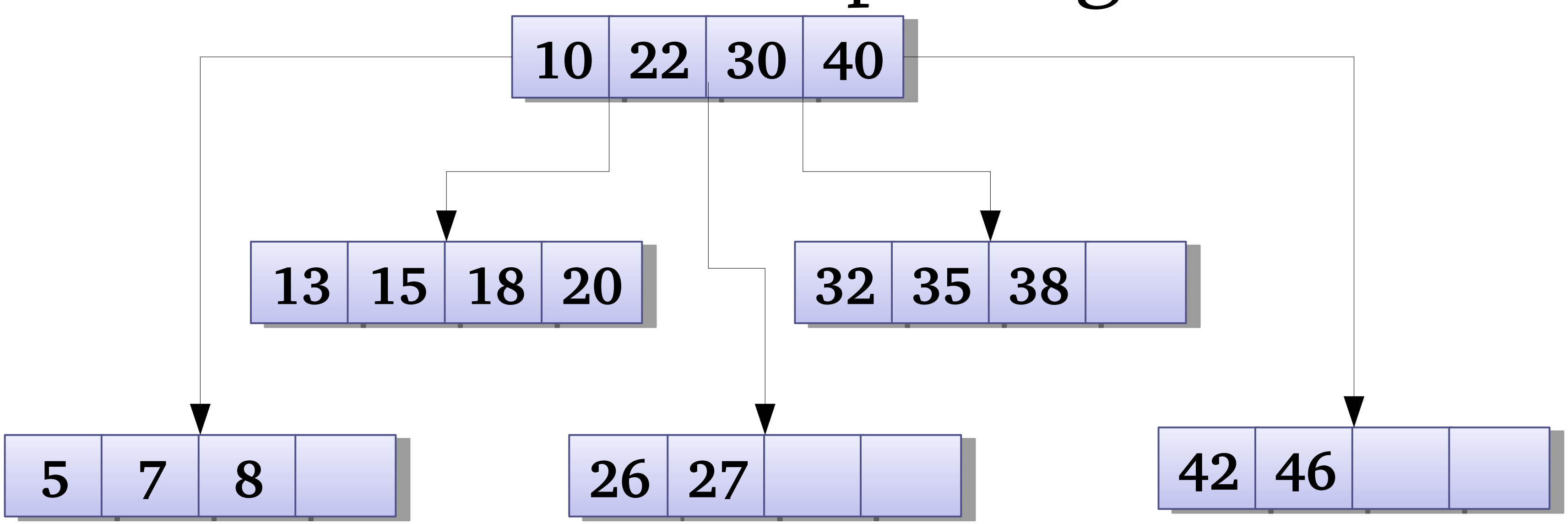
- Still simpler than AVL deletion, but in AVL trees the same balancing code could be used for both insertion and deletion

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

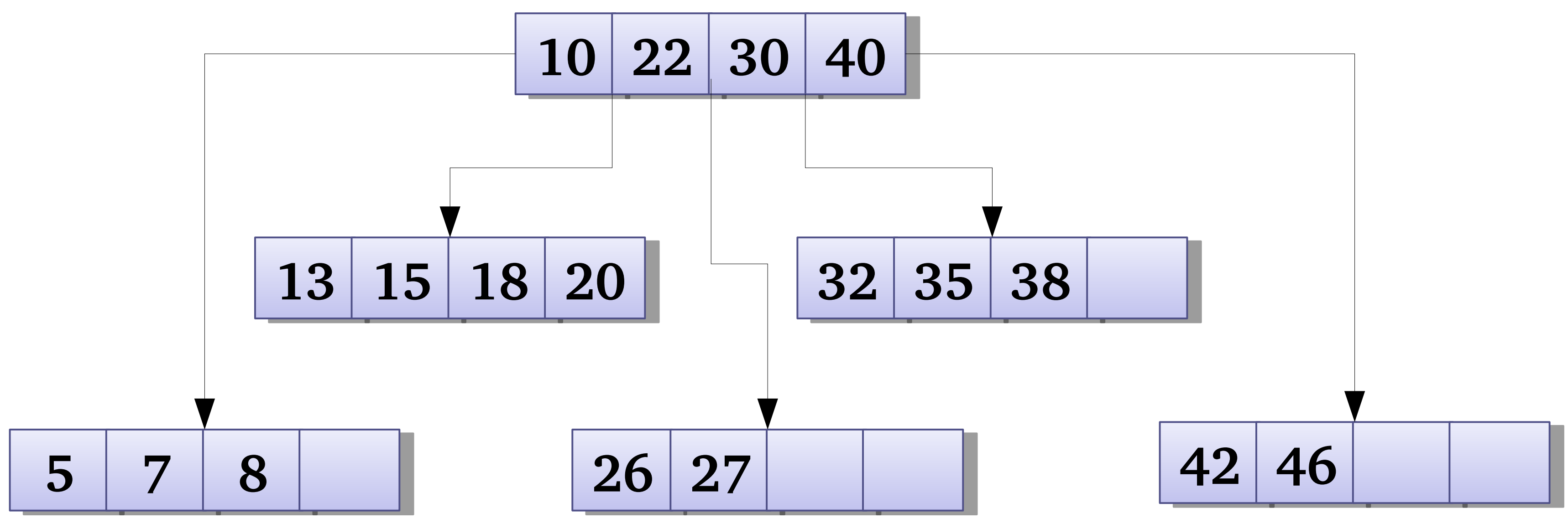
Insertion also based on splitting!



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



Red-black trees (not on exam)

Instead of 2-3 trees, we can use *2-3-4 trees*

- 2-node, 3-nodes and 4-nodes
  (or B-tree with k = 4)

There's a more efficient insertion algorithm for them, called top-down insertion!

- See book section on red-black trees, or Wikipedia page on 2-3-4 trees, for more information

We can implement them using BSTs, using the same ideas as AA trees. This is called a *red-black tree*, the fastest balanced BST

- Gets complicated because of lots of cases

Summary

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

- Possible to keep perfectly balanced
- Slightly annoying to implement

AA trees – 2-3 trees implemented using a BST

- Similar performance to AVL trees, but much simpler
- Fewer cases to consider, because the invariant can only break in two ways

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