A graphic explanation of Red-black trees

Matthias Goerner

Red-black trees

A red-black tree is

one of the easiest and most important
self-balancing
binary search trees.

(implemented by, e.g., C++'s std::map, Java's TreeMap)

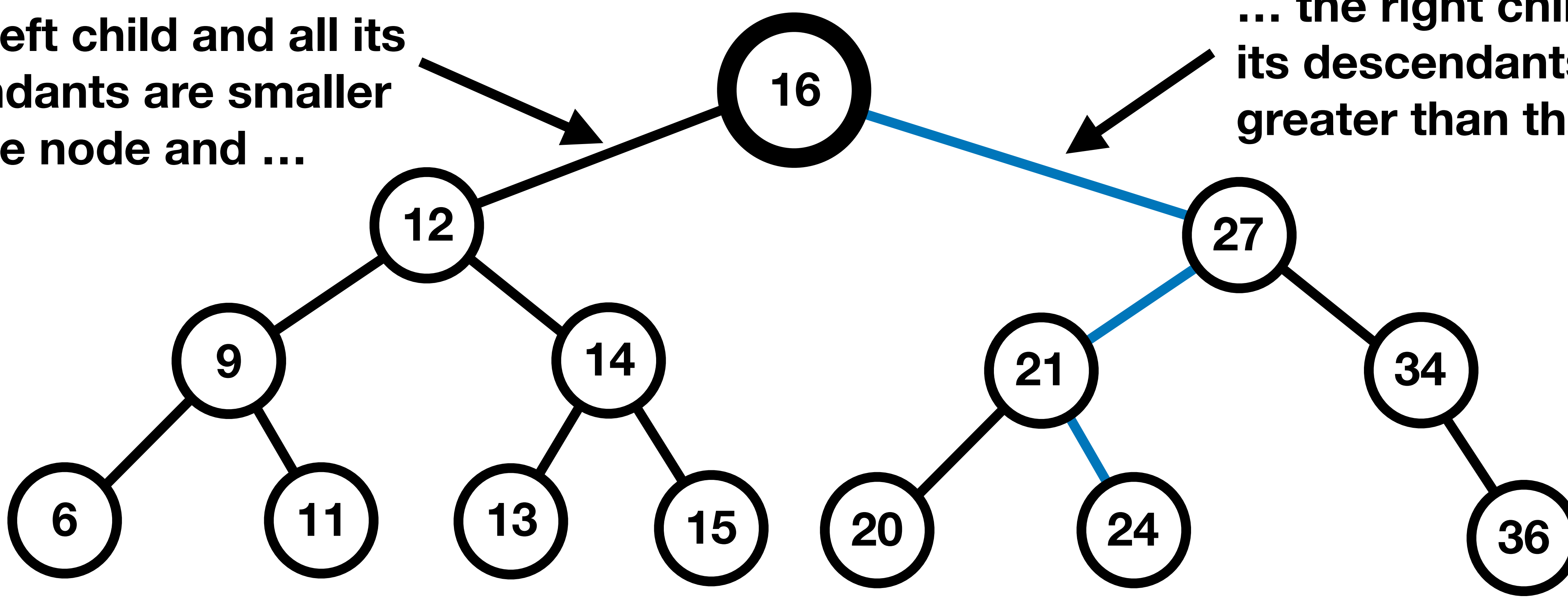
Binary search trees

Each node has:

- **some orderable piece of data (key), say a number**
- **can have a left child and**
- **can have a right child such that …**

… the left child and all its descendants are smaller than the node and …

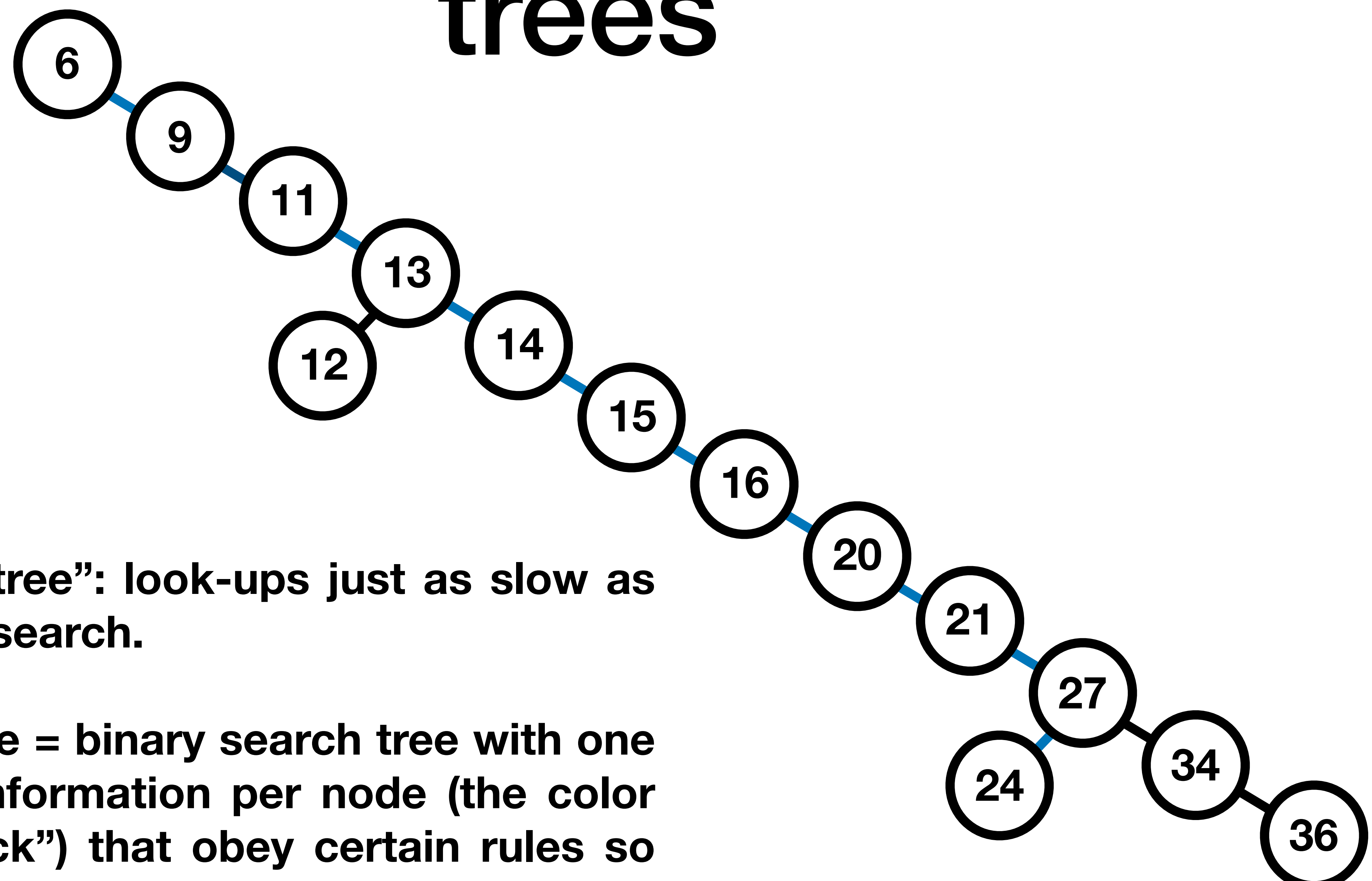
… the right child and all its descendants are greater than the node.



Tree "balanced": fast look-ups, e.g., if tree had 1 million nodes, only traverse 20.

Similar to opening a phone book or dictionary in the middle and deciding whether to continue searching in the left or right half.

Unbalanced binary search trees



However:
"unbalanced tree": look-ups just as slow as a brute force search.

Red-black tree = binary search tree with one extra bit of information per node (the color "red" or "black") that obey certain rules so that they never become unbalanced, no matter in what order the elements were inserted when creating the tree.

A graphic way to think about red-black trees

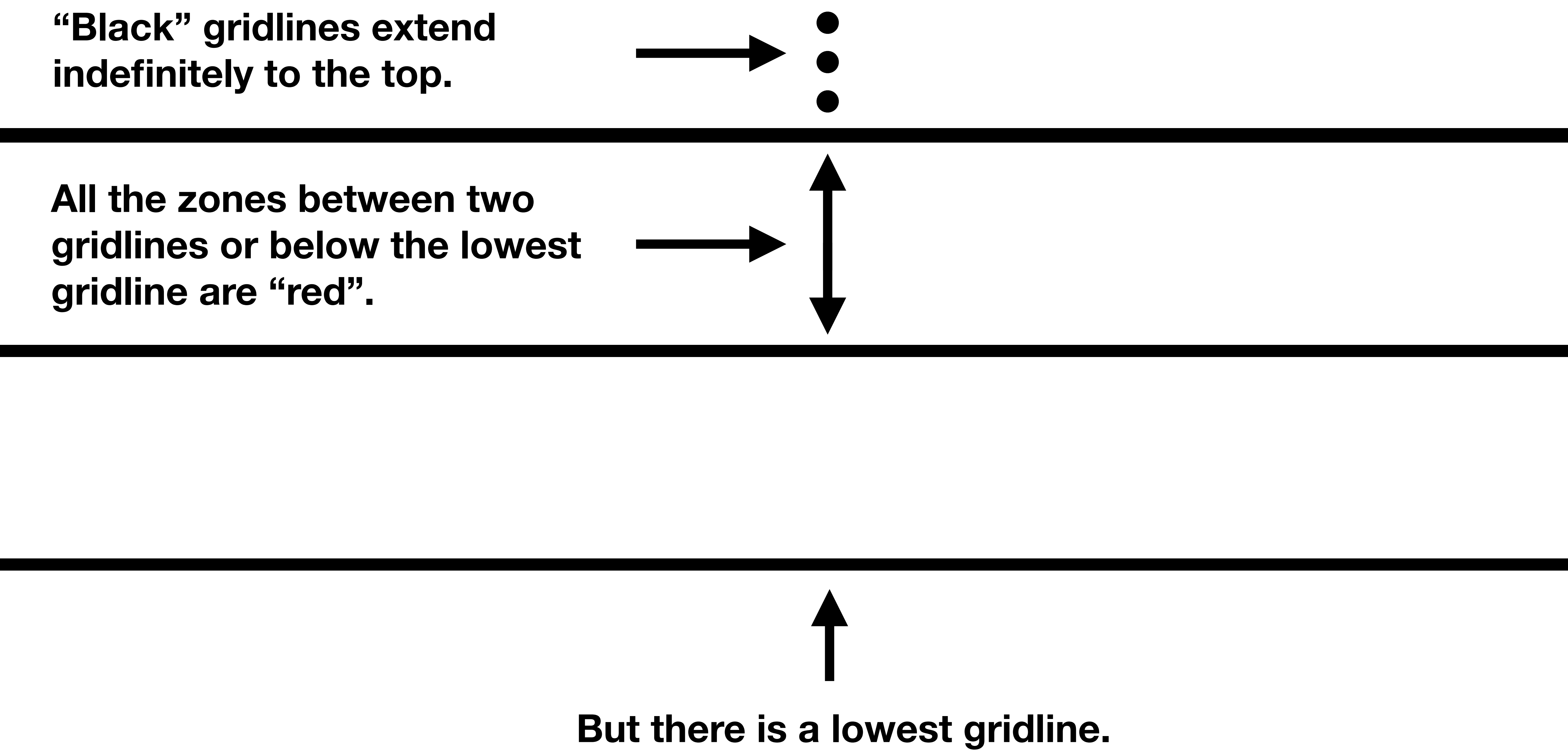
A red-black tree is a tree drawn on red ruled paper obeying certain rules.

Draw tree on red ruled paper

⋮

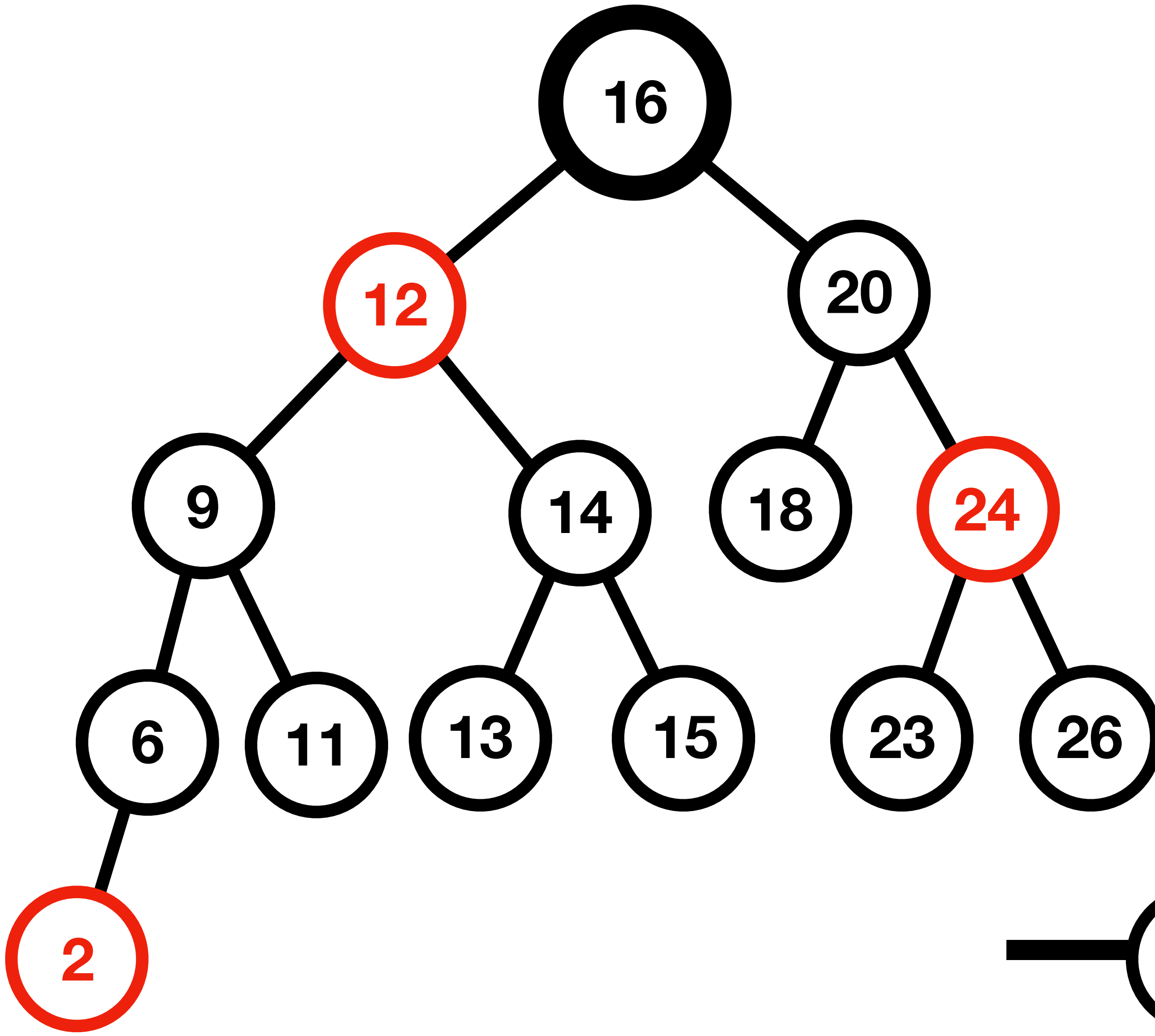
But drawing all the red zones in red actually hurts the eyes too much, so just imagine it.

Draw tree on ruled paper

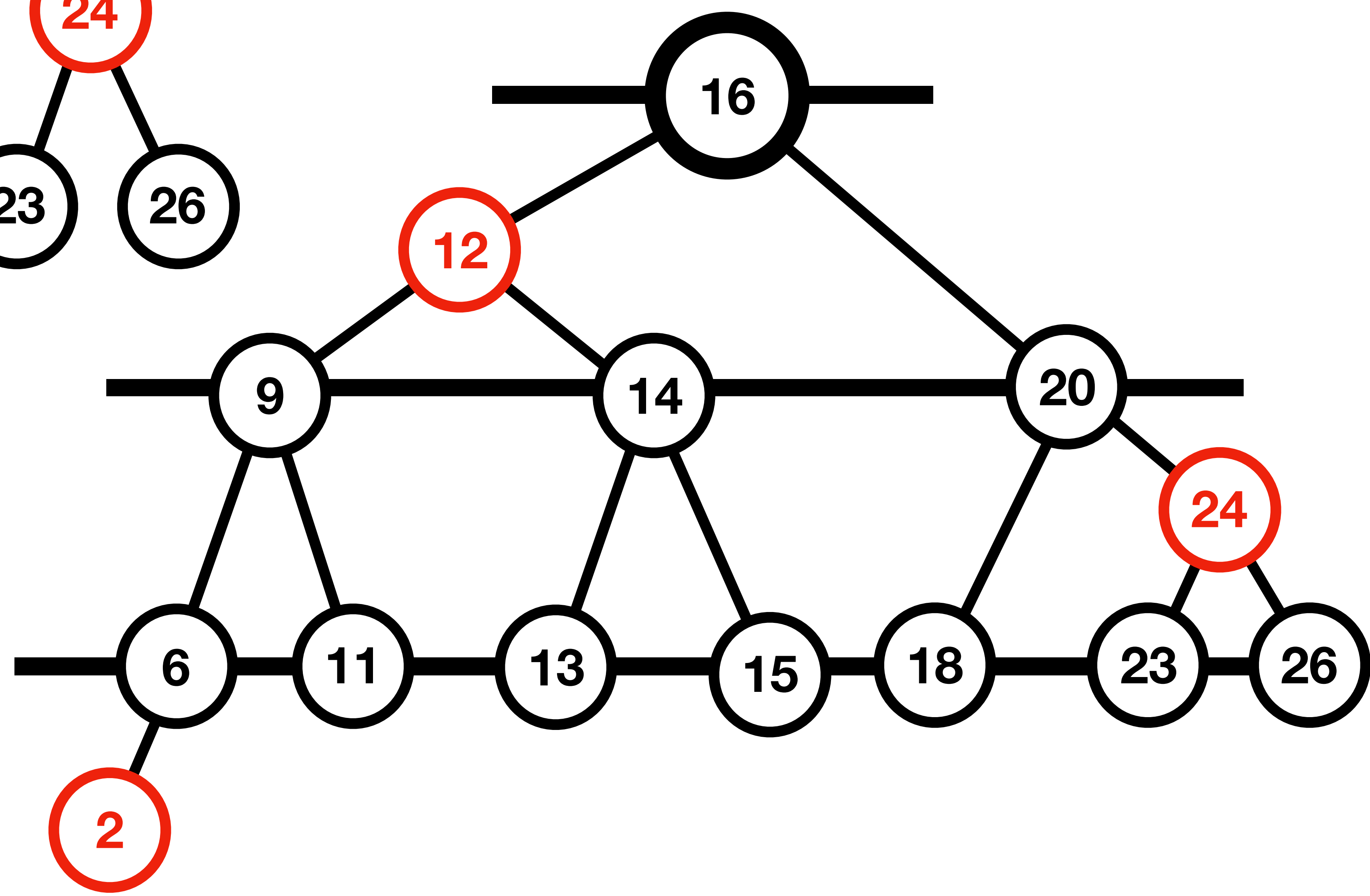


Traditional textbook:
Red-black tree
as colored tree

Unfortunately, most traditional textbooks do not draw red-black trees on ruled paper - which makes them harder to understand.



This presentation:
Red-black tree
on ruled paper



The rules

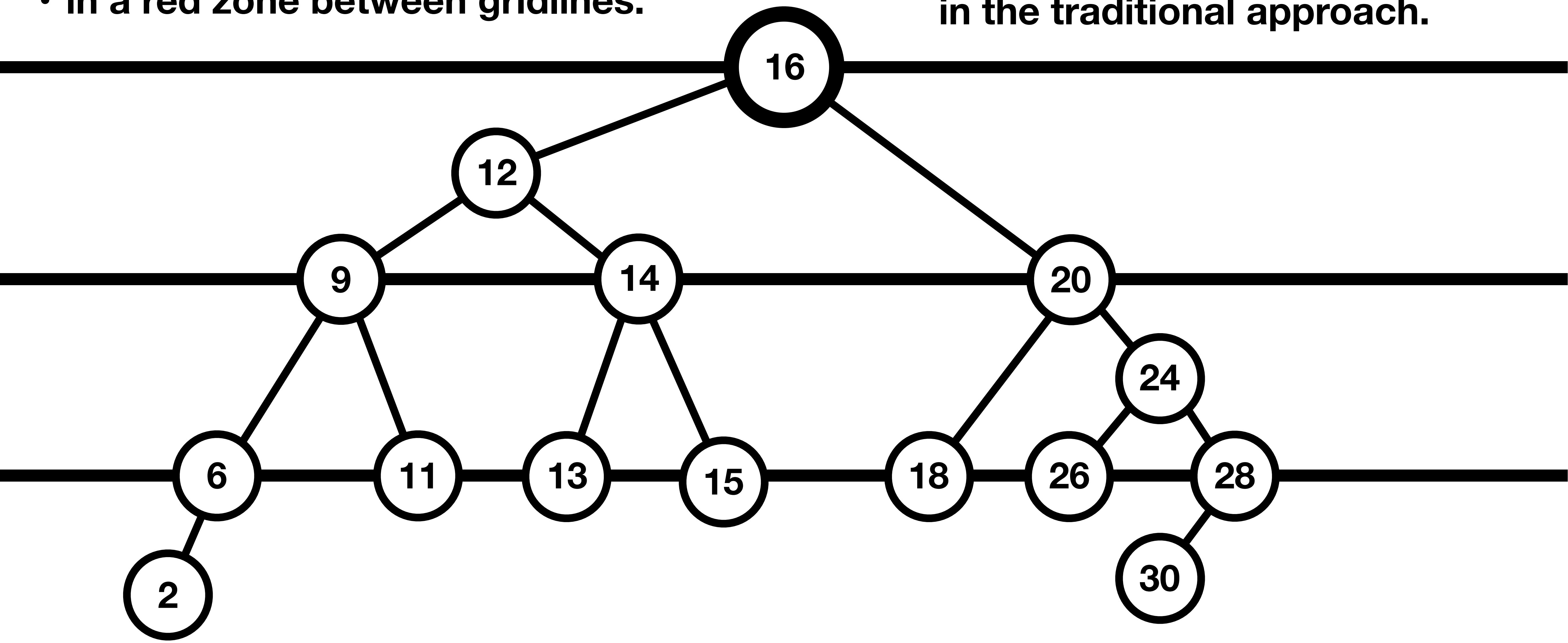
The rules for red-black trees now become:

(compare to, e.g.,
Cormen et al “Introduction to Algorithms”)

Rule 1

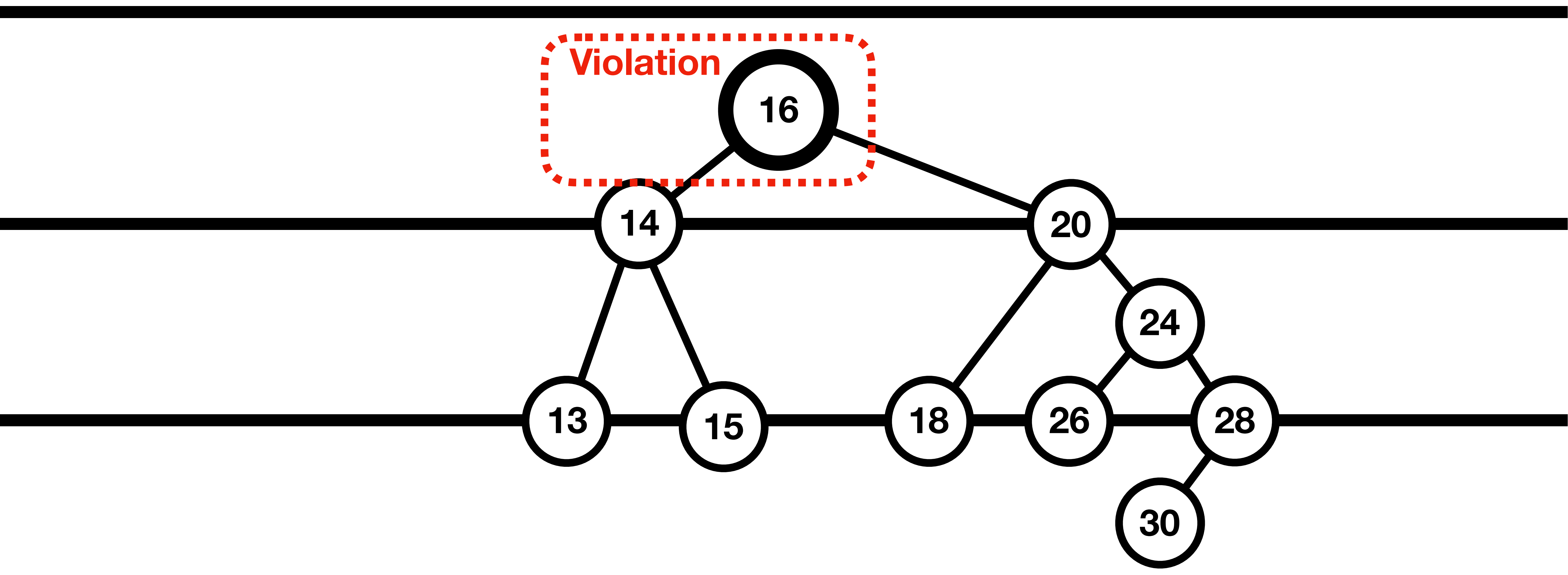
Each node is considered to be either
- on a black gridline or
- in a red zone between gridlines.

This corresponds to the node being colored “black” or “red” in the traditional approach.



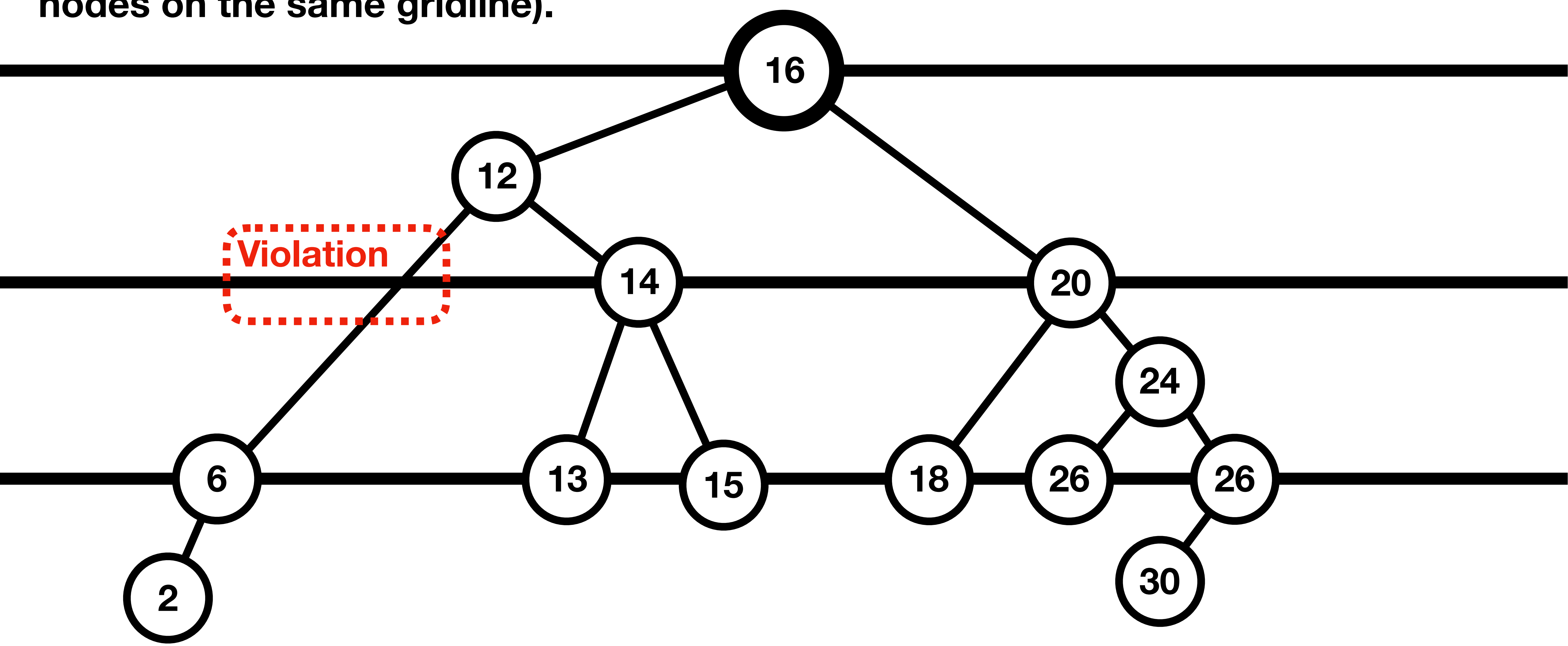
Rule 2

The root has to be on a gridline.



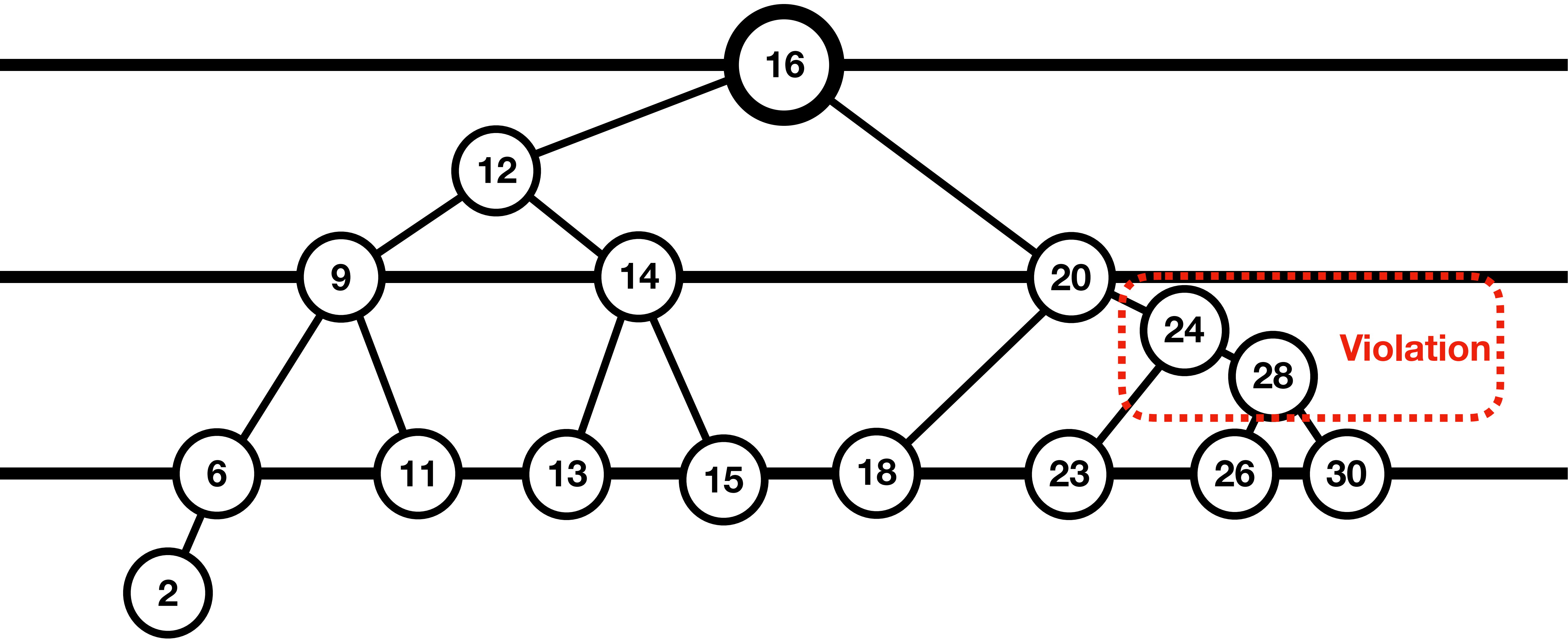
Rule 3

An edge cannot cross a gridline (and, in particular, not connect two nodes on the same gridline).



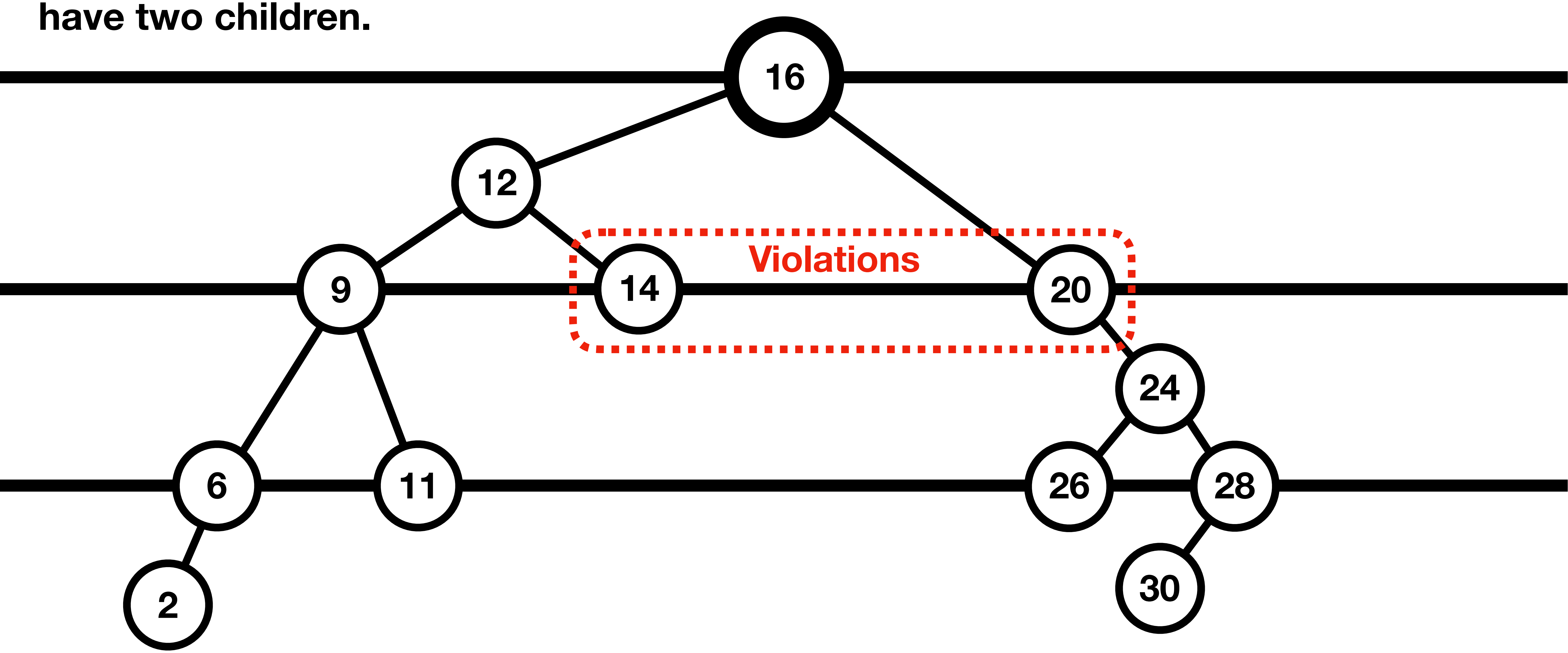
Rule 4

There cannot be two or more connected nodes in a red zone.



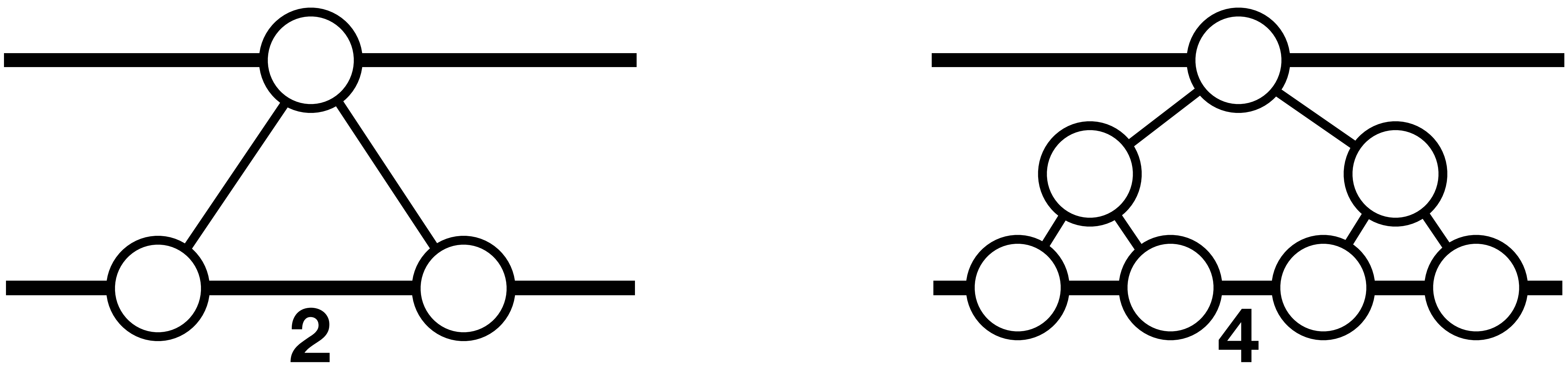
Rule 5

Each node not on or below the lowest gridline must have two children.



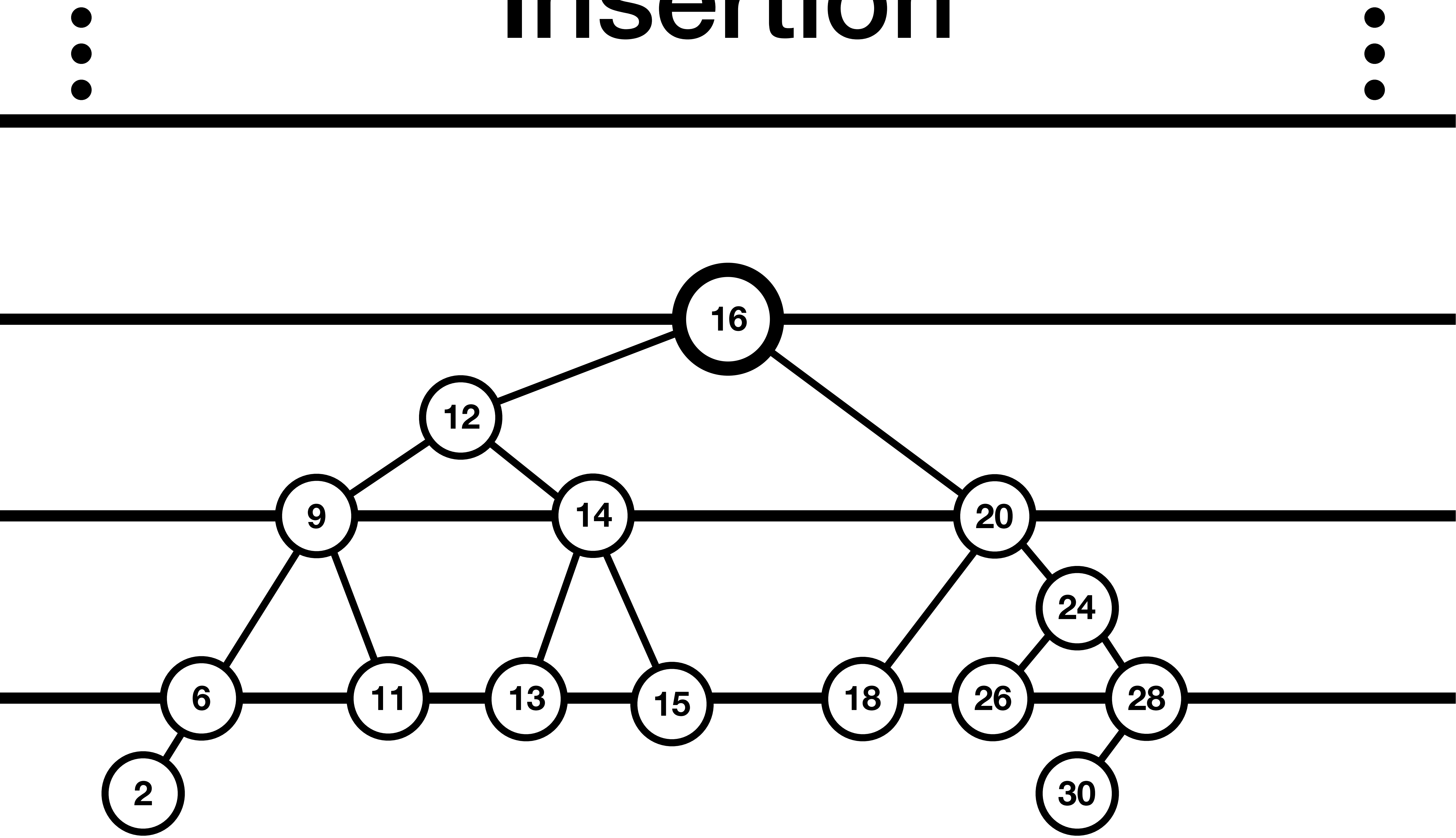
Search time

The rules enforce that (starting from the root),
for each gridline down,
the number of nodes doubles to quadruples,
so the tree grows *exponentially*:



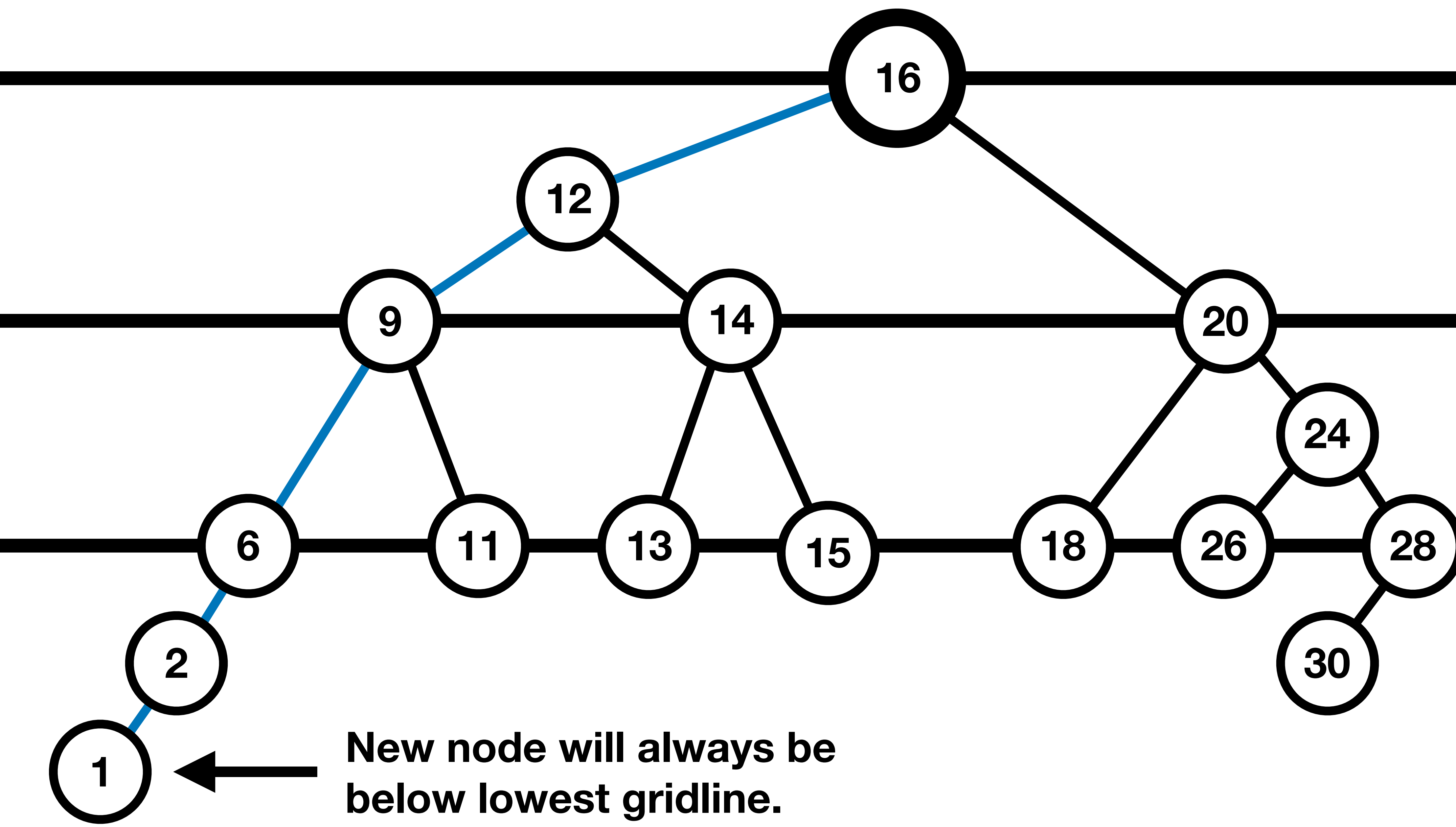
Thus, the number of gridlines occupied by the tree
and, hence, the search time grows
***logarithmically* in the number of nodes.**

Insertion



Insertion

1. Insert as for ordinary binary search tree.

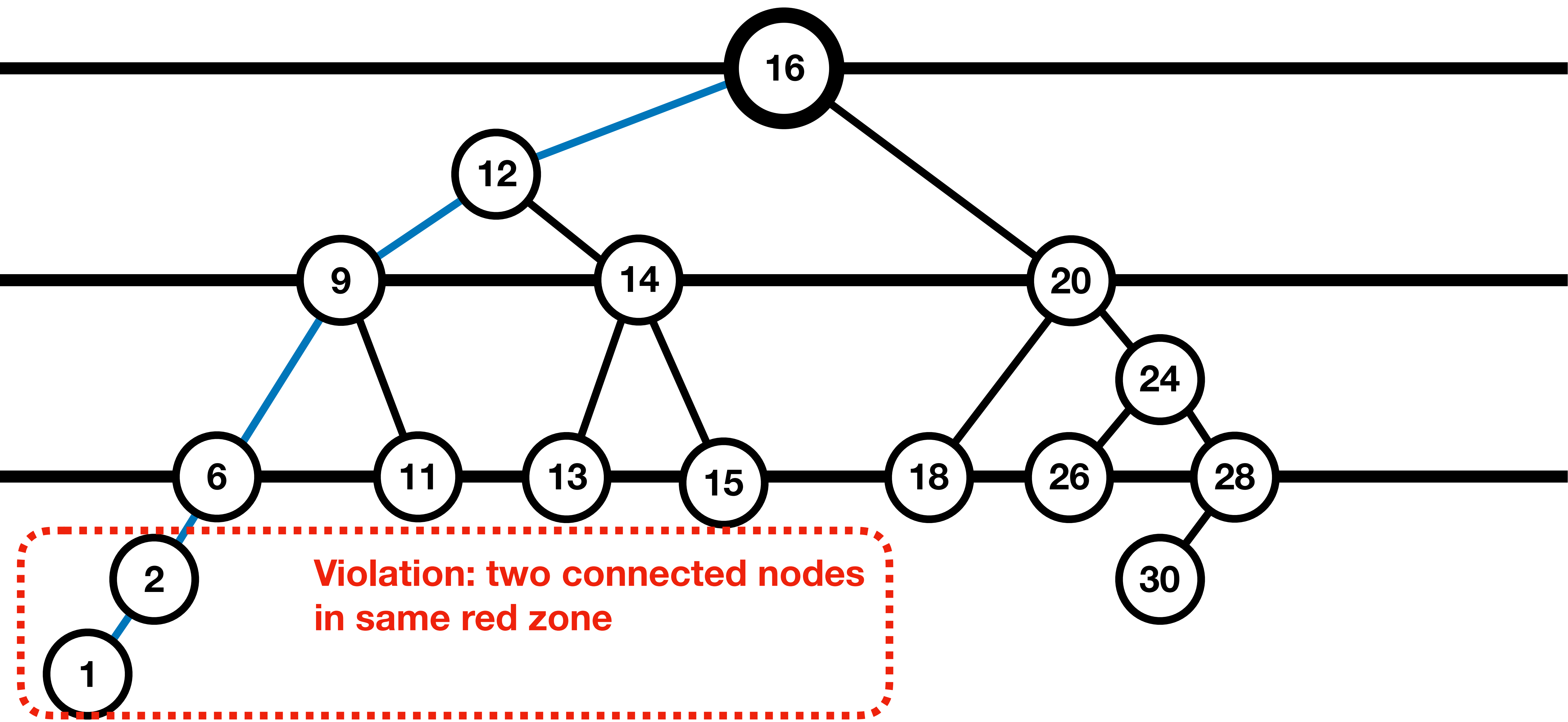


New node will always be below lowest gridline.

Insertion

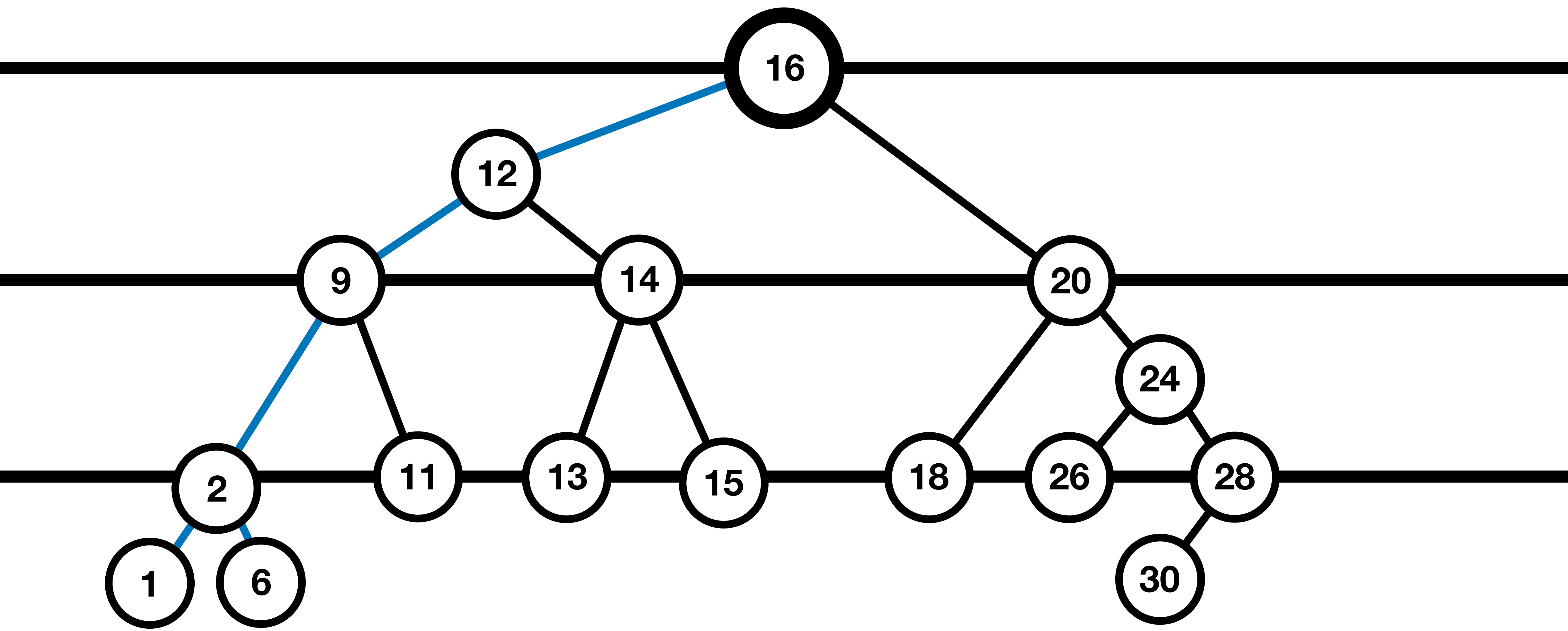
1. Insert as for ordinary binary search tree.

This might cause a temporary violation of rule 4!



Insertion

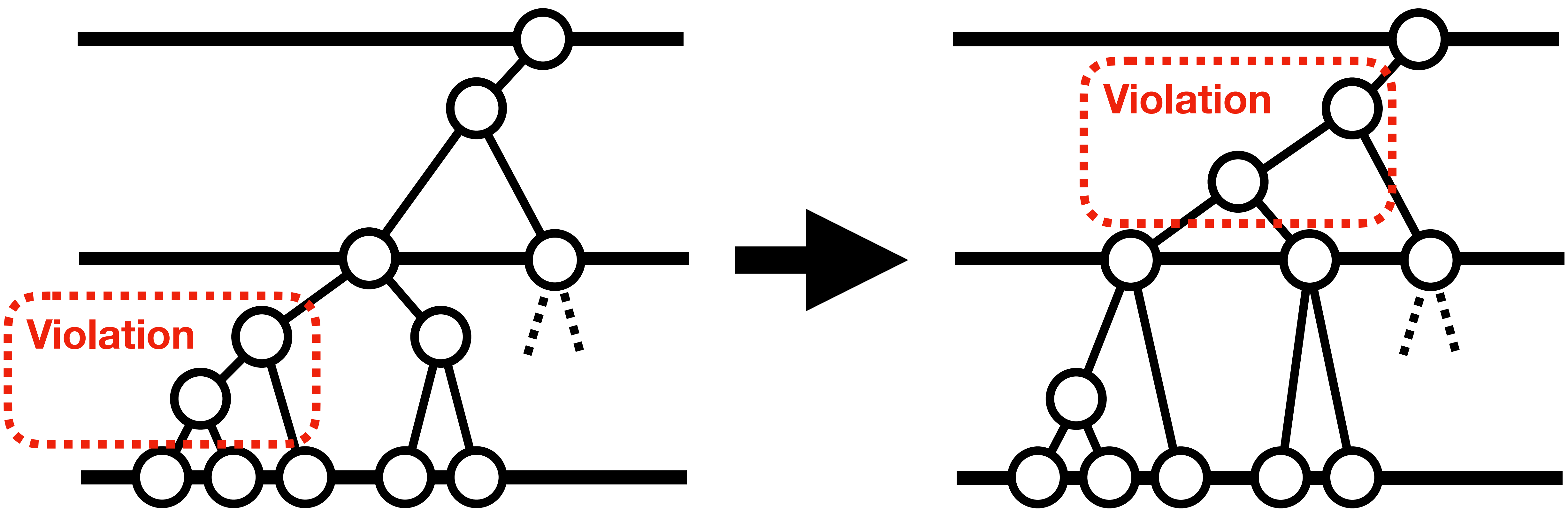
1. Insert as for ordinary binary search tree.
2. Follow insertion path from bottom up to fix rule violation.



Fixing the rule violation

The previous example was easy, just one "right-rotation".

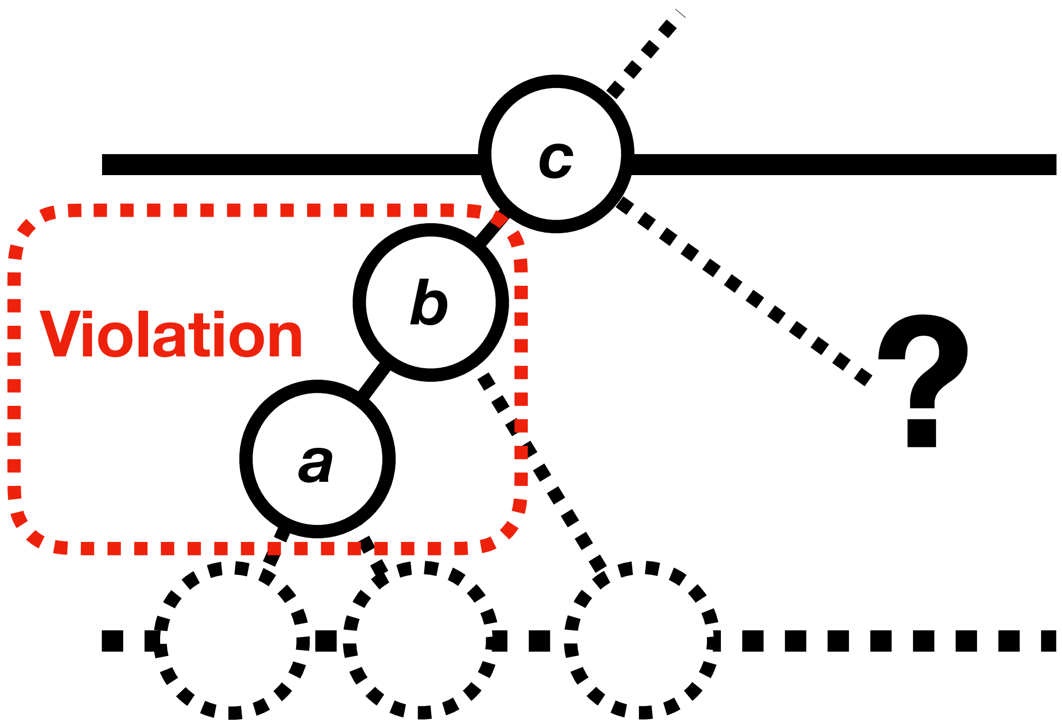
In general, while fixing a violation in one red zone, we might introduce another violation in the zone above which we then need to fix and so on…



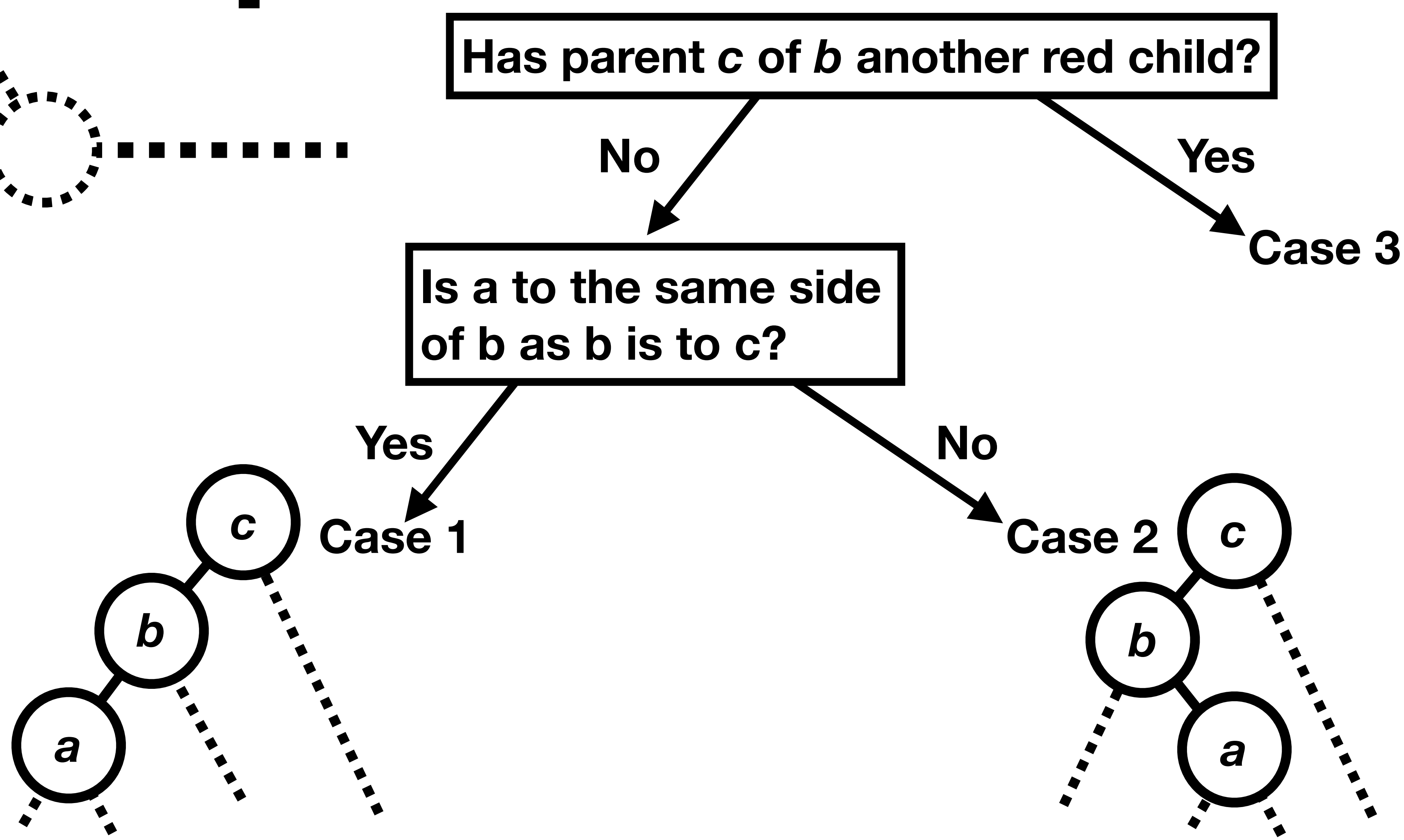
… until we hit the root. A that point, we can fix the tree by changing the root or moving it half a grid line up.

This might sound complicated, but it really boils down to:
only *3* cases (up to mirroring)!

Case analysis: overview



Violation:
***a* is a child of *b* and both are in the (same) red zone.**



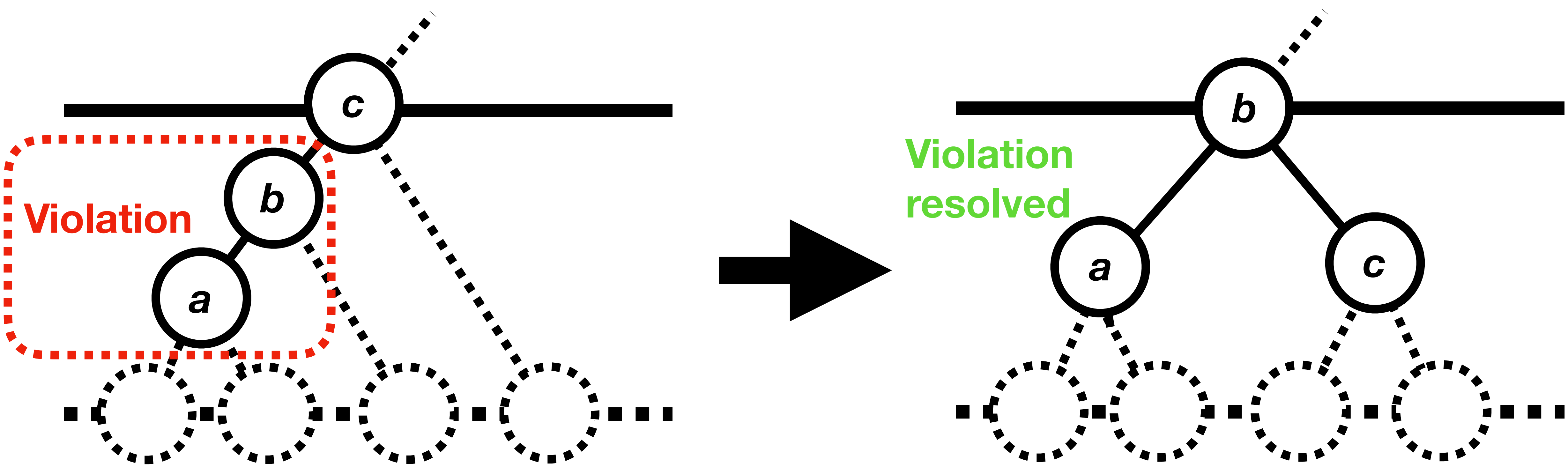
Case 1

A node *b* in a red zone

- **has a left child *a* in the (same) red zone and**
- **is the left child of a node *c* and *c* has no right child in the red zone.**

Resolution:

- **Right-rotate about *c*.**
- **If *c* was a root, set *b* as new root.**



Note that this works the same when the red zone is the lowest one and when interchanging left and right everywhere.

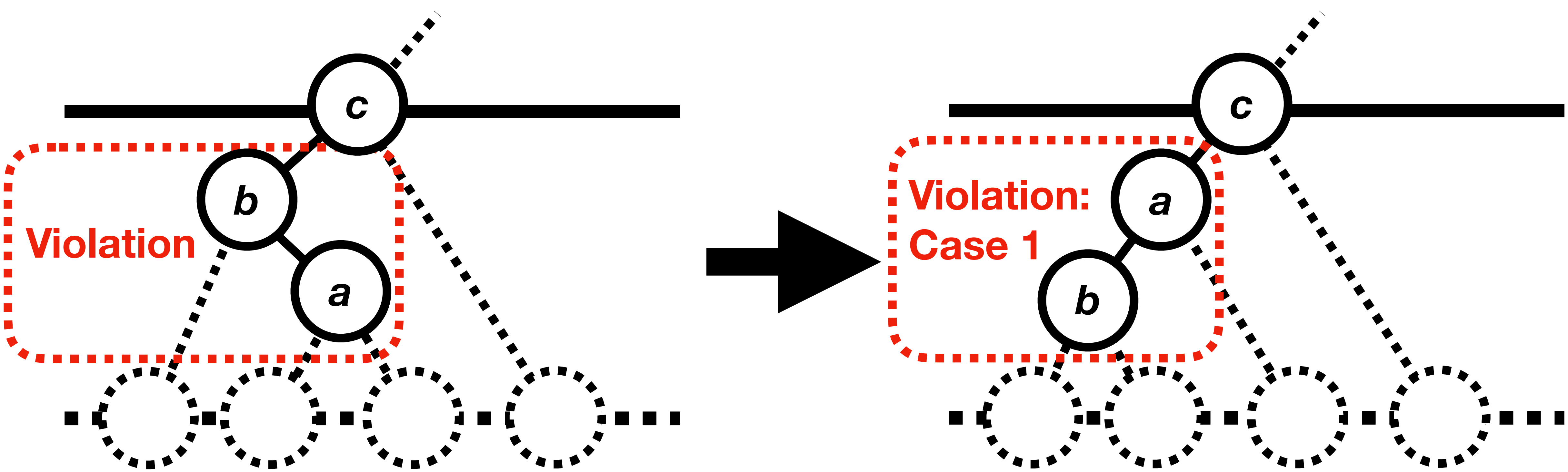
Case 2

A node *b* in a red zone

- has a right child *a* in the (same) red zone and
- is the left child of a node *c* and *c* has no right child in the red zone.

Resolution:

- Left-rotate about *b* to reduce to case 1.



Note that this works the same when the red zone is the lowest one and when interchanging left and right everywhere.

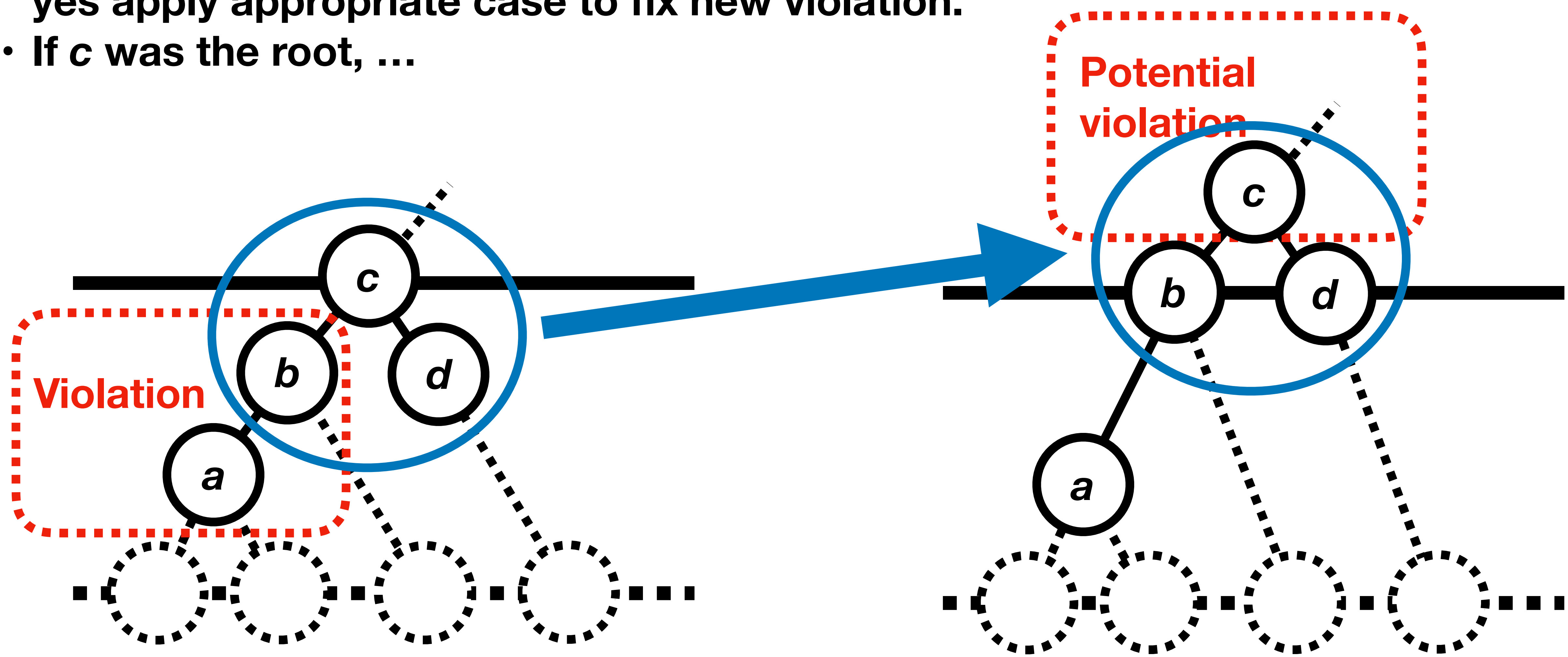
Case 3

A node *b* in a red zone

- has a child *a* in the (same) red zone and
- its parent *c* has two children in the red zone, call the other one *d*.

Resolution:

- Move *b*, *c*, and *d* half a gridline up.
- Check whether *c* has a parent in the red zone (that *c* is now in), if yes apply appropriate case to fix new violation.
- If *c* was the root, …



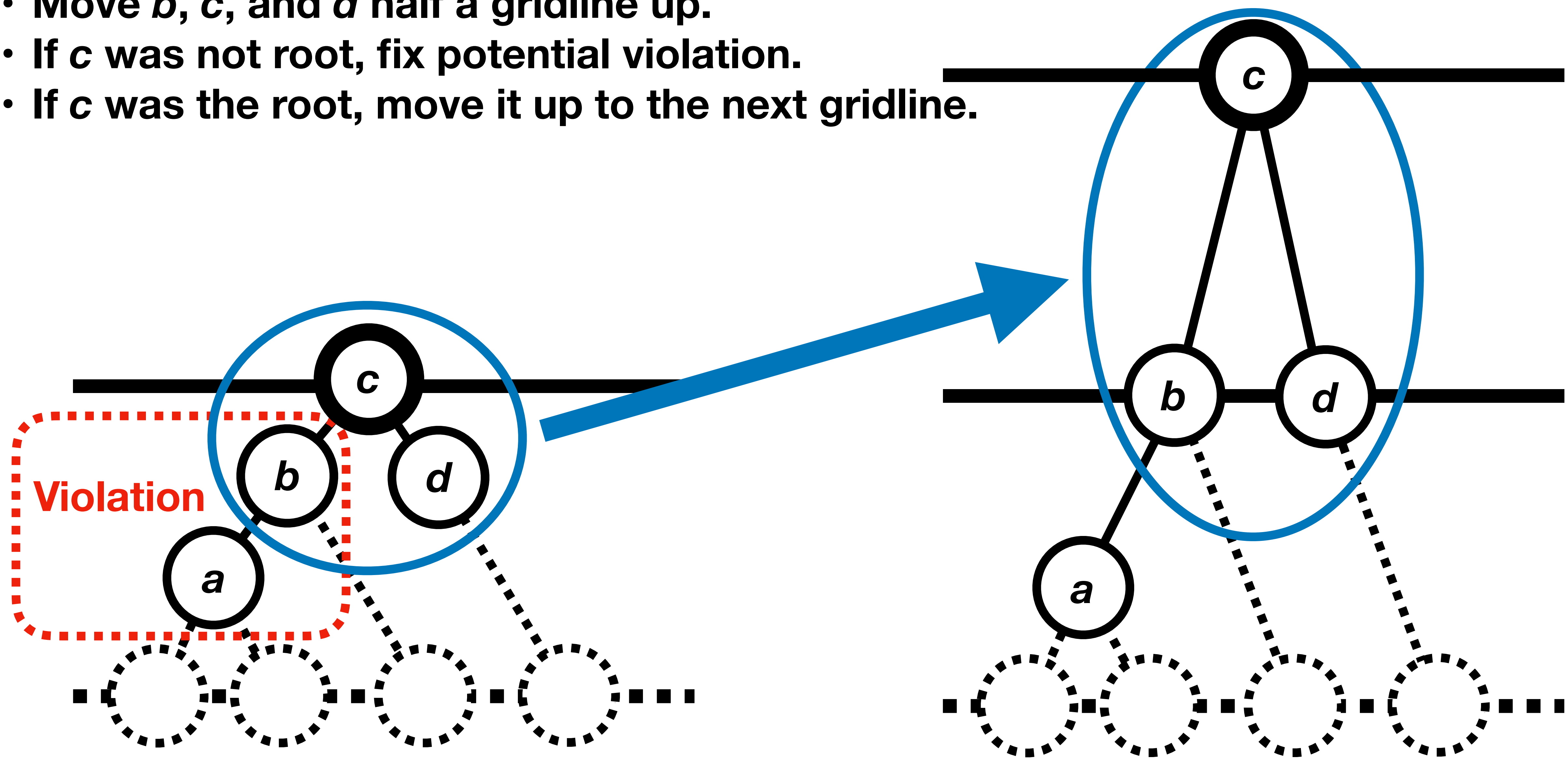
Case 3: root

A node *b* in a red zone

- **has a child *a* in the (same) red zone and**
- **its parent *c* has two children in the red zone, call the other one *d*.**

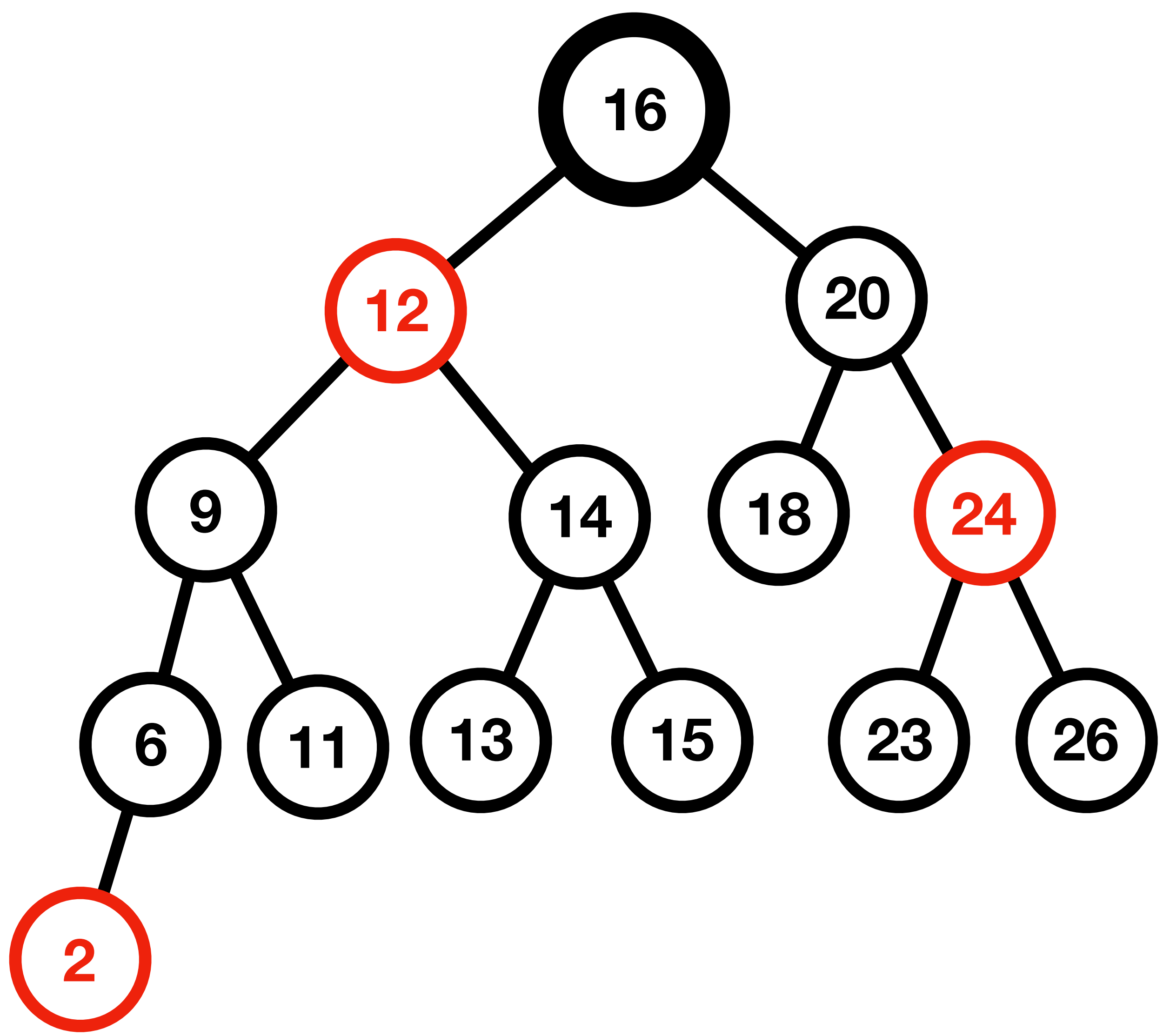
Resolution:

- **Move *b*, *c*, and *d* half a gridline up.**
- **If *c* was not root, fix potential violation.**
- **If *c* was the root, move it up to the next gridline.**



Implementation

Two-colored binary search tree



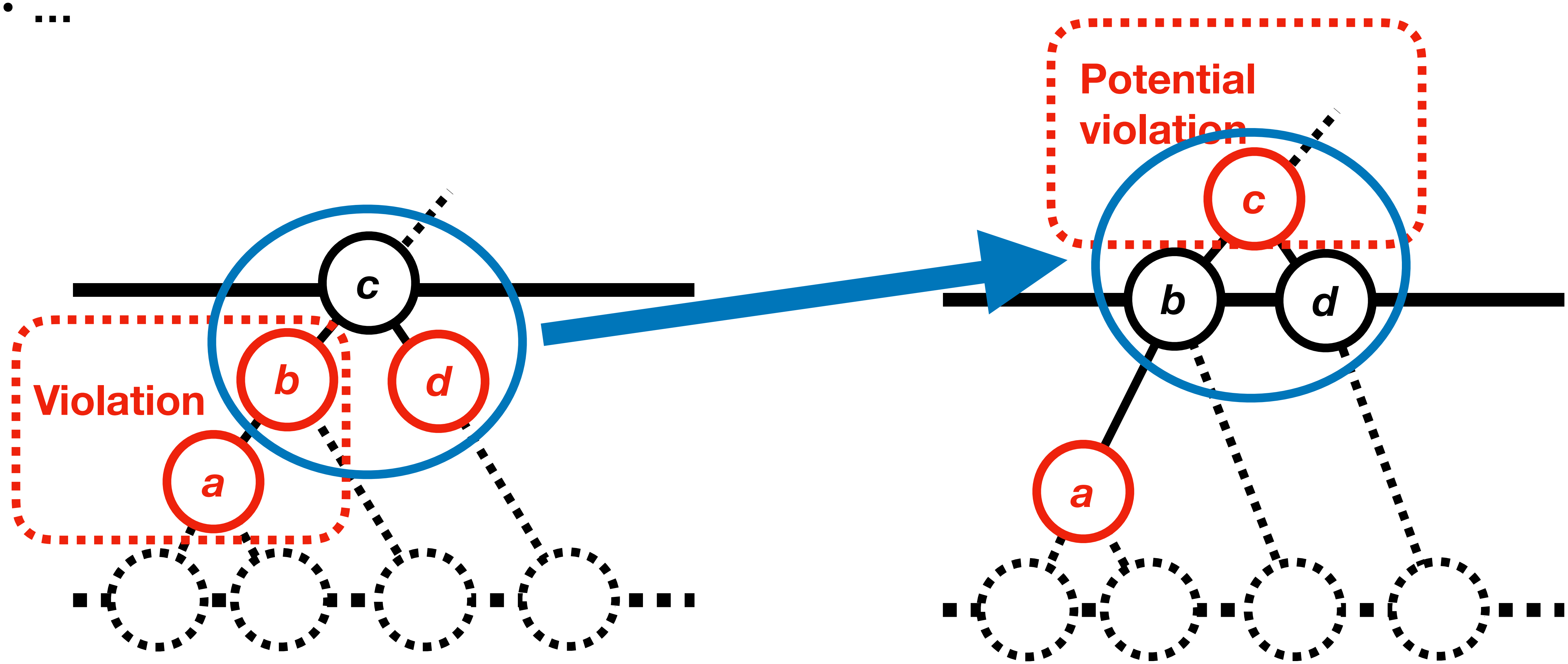
Case 3 graphically

A node *b* in a red zone

- **has a child *a* in the (same) red zone and**
- **its parent *c* has two children in the red zone, call the other one *d*.**

Resolution (*graphically*):

- **Move *b* and *d* up to the next gridline**
- **Move c up to the next red zone**
- **…**



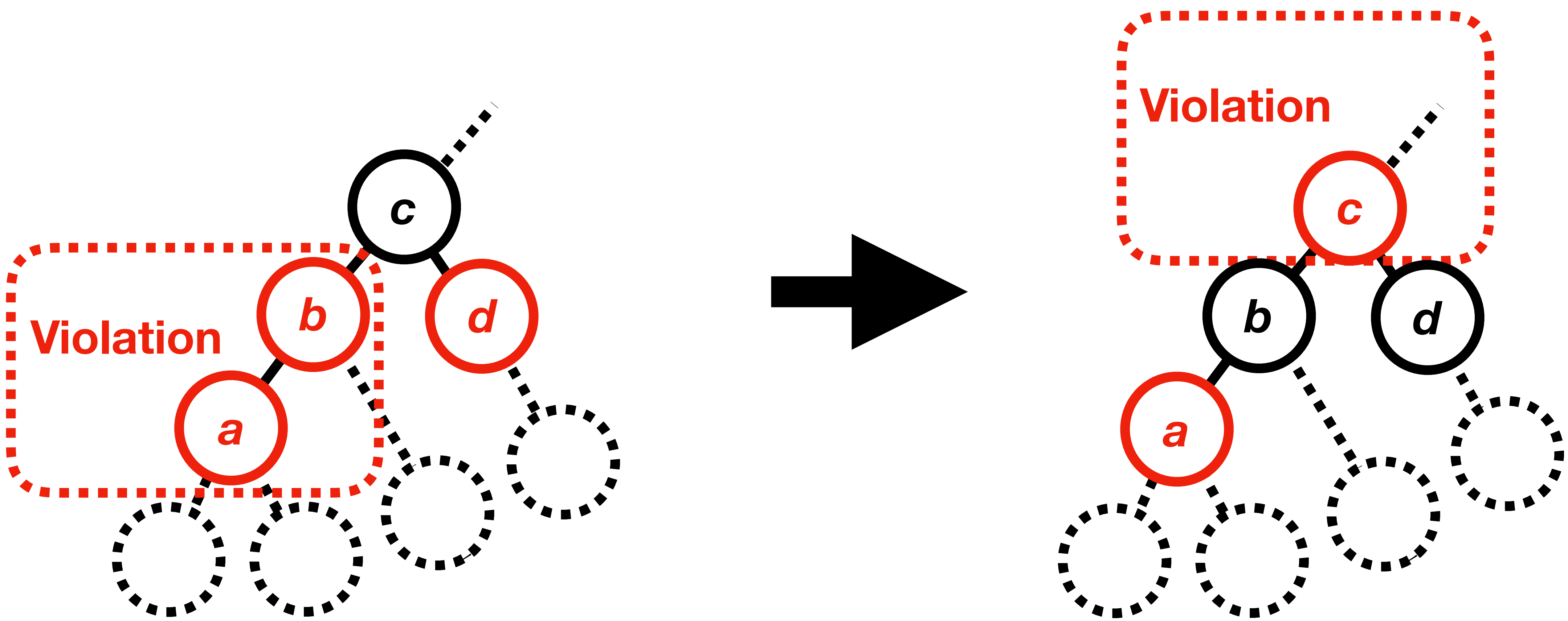
Implementation of Case 3

A node *b* is red and

- **has a red child *a* and**
- **its parent *c* has two red children, call the other one *d*.**

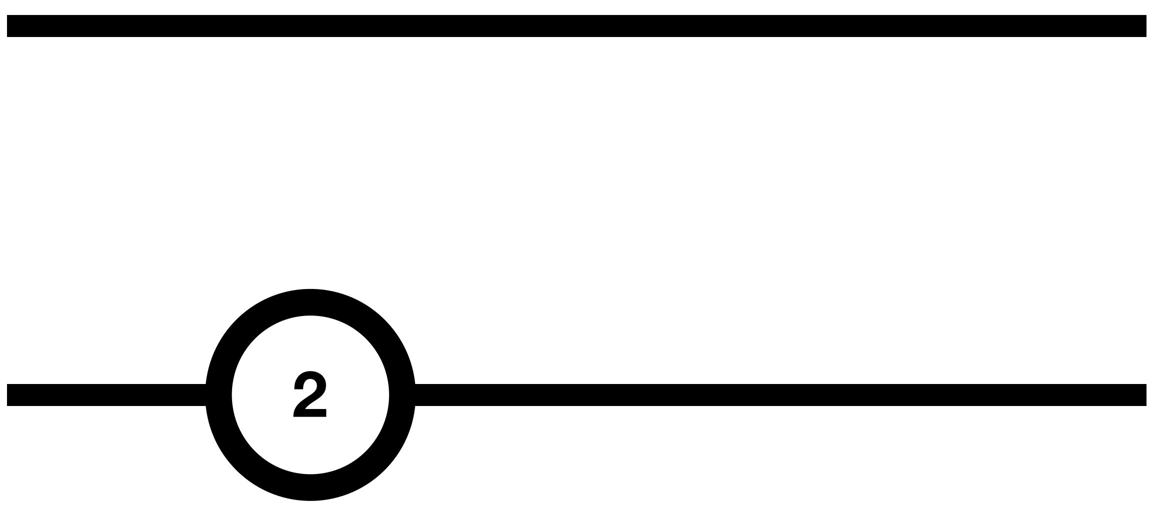
Resolution (*implementation*):

- **Color b and d black (to indicate gridline)**
- **Color c red (to indicate red zone)**
- **…**



Example: Insert 2

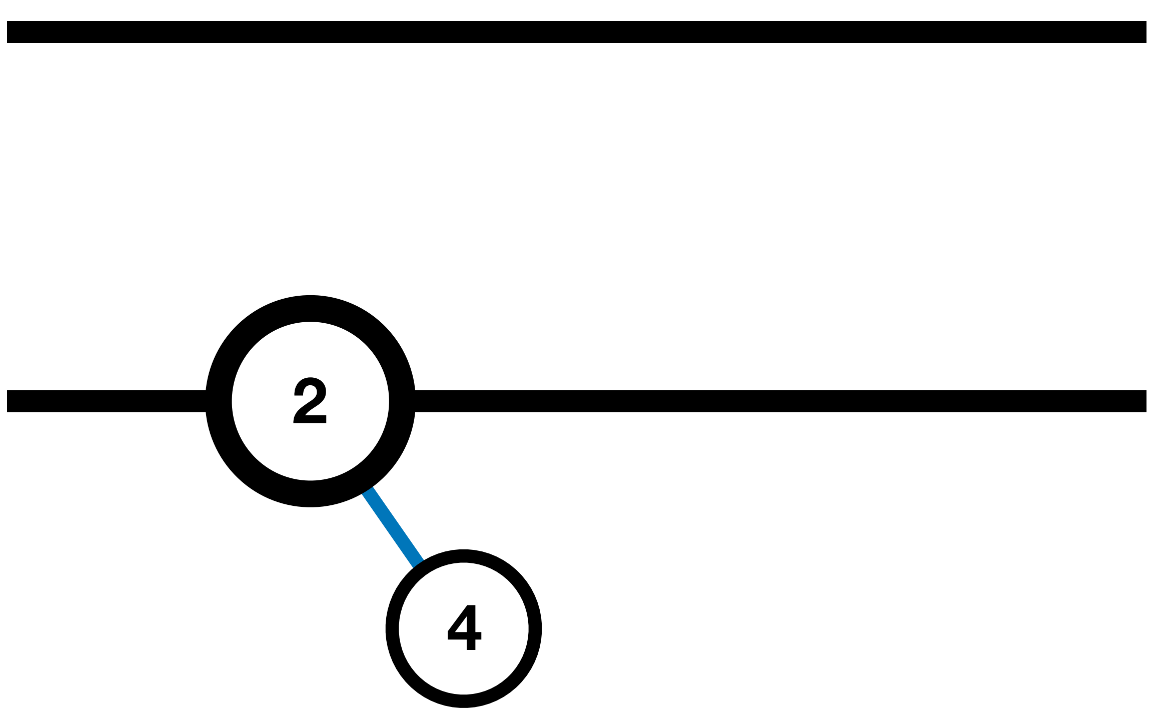
Adding 2 as root to empty tree.



No fix required.

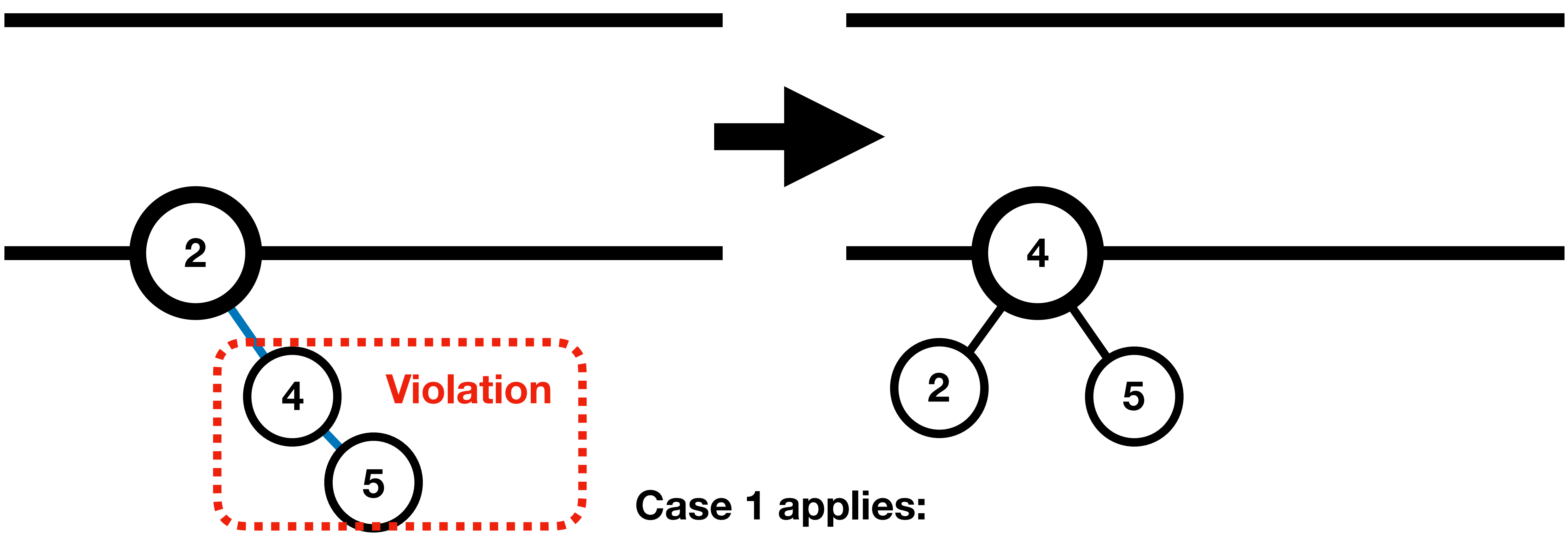
Example: Insert 4

Starting with tree having only one node, insert 4.



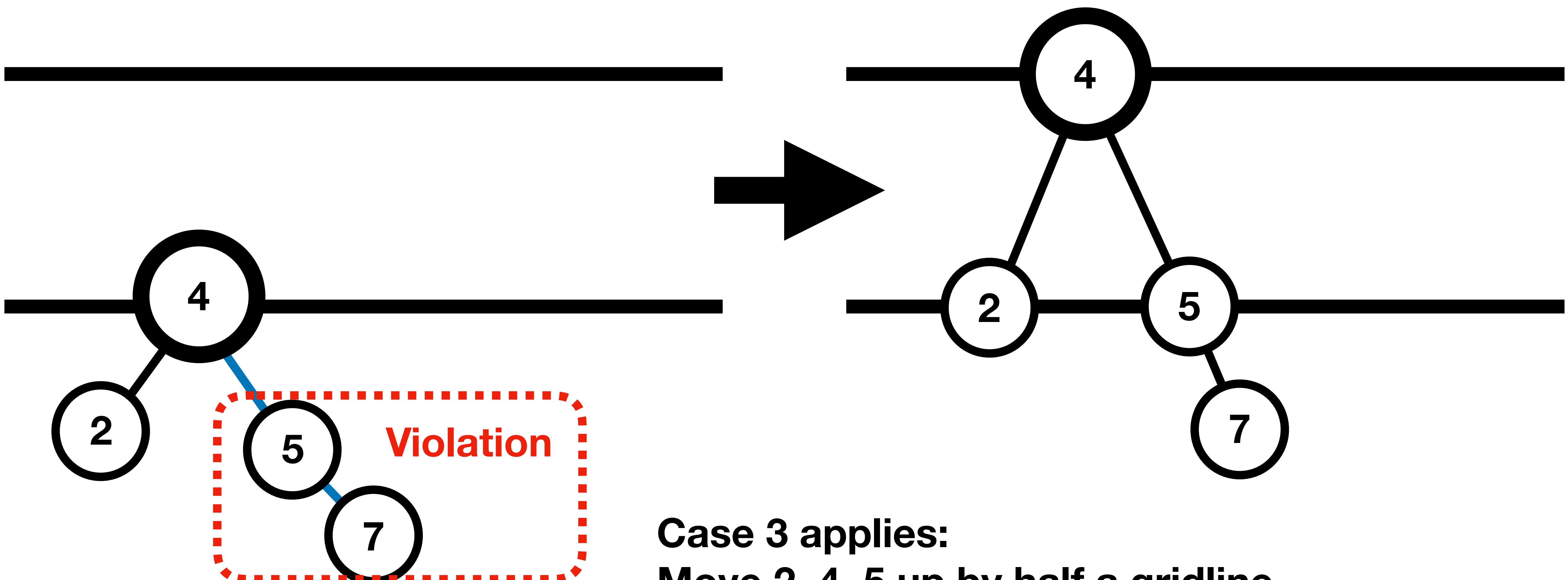
No fix required.

Example: Insert 5



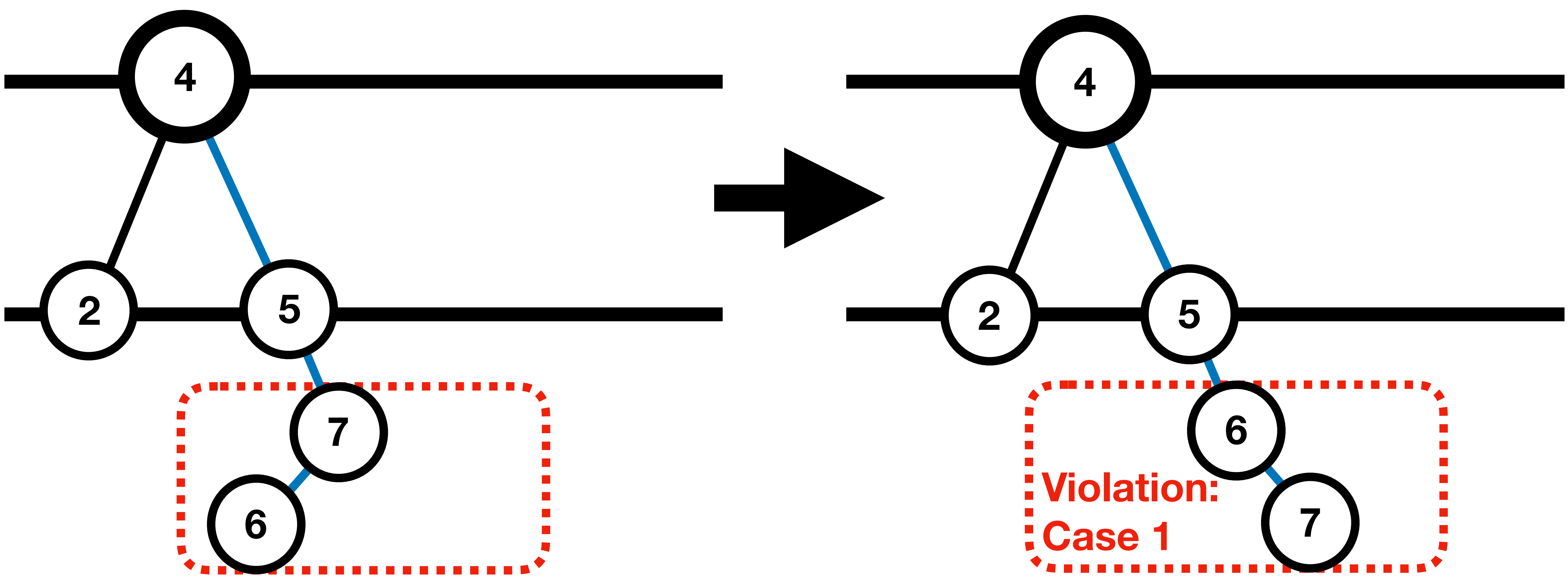
Case 1 applies:
Left-rotate about 2
Note that the root is now 4.

Example: Insert 7



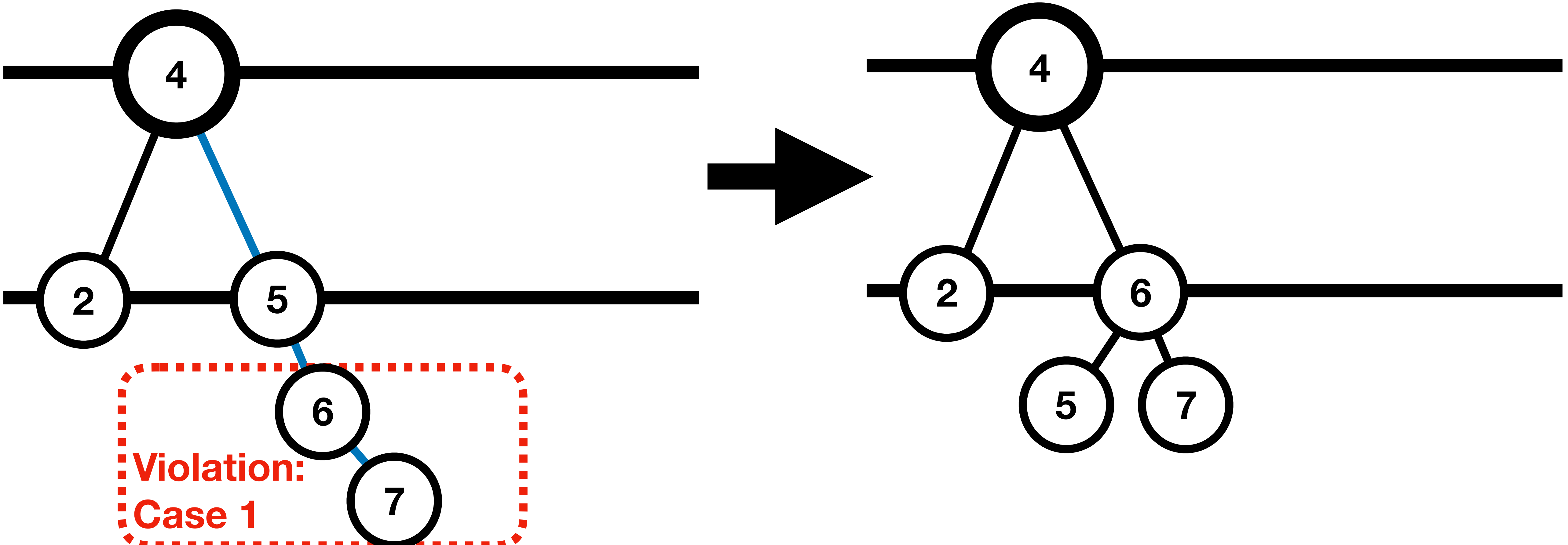
Case 3 applies:
Move 2, 4, 5 up by half a gridline.
Move the root 4 to the next gridline.

Example: Insert 6



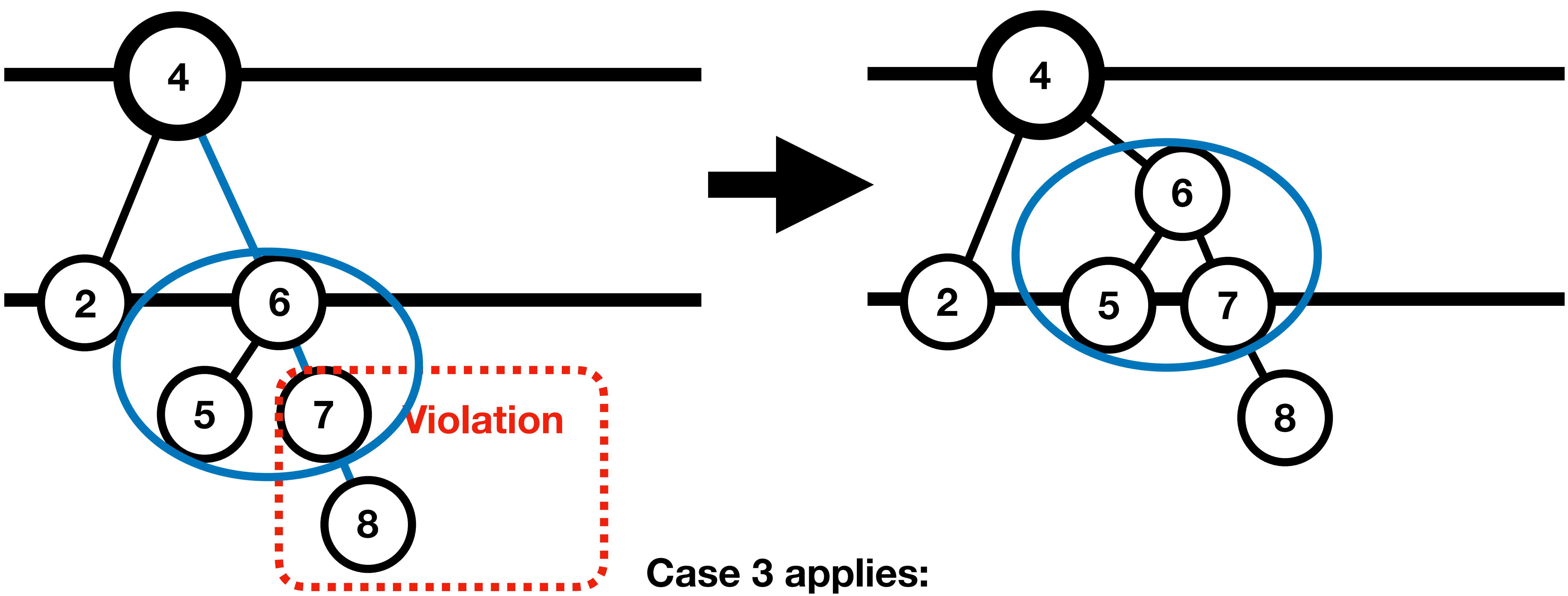
Case 2 applies:
Right-rotate about 7 to reduce to case 1.

Example: Insert 6 (part 2)



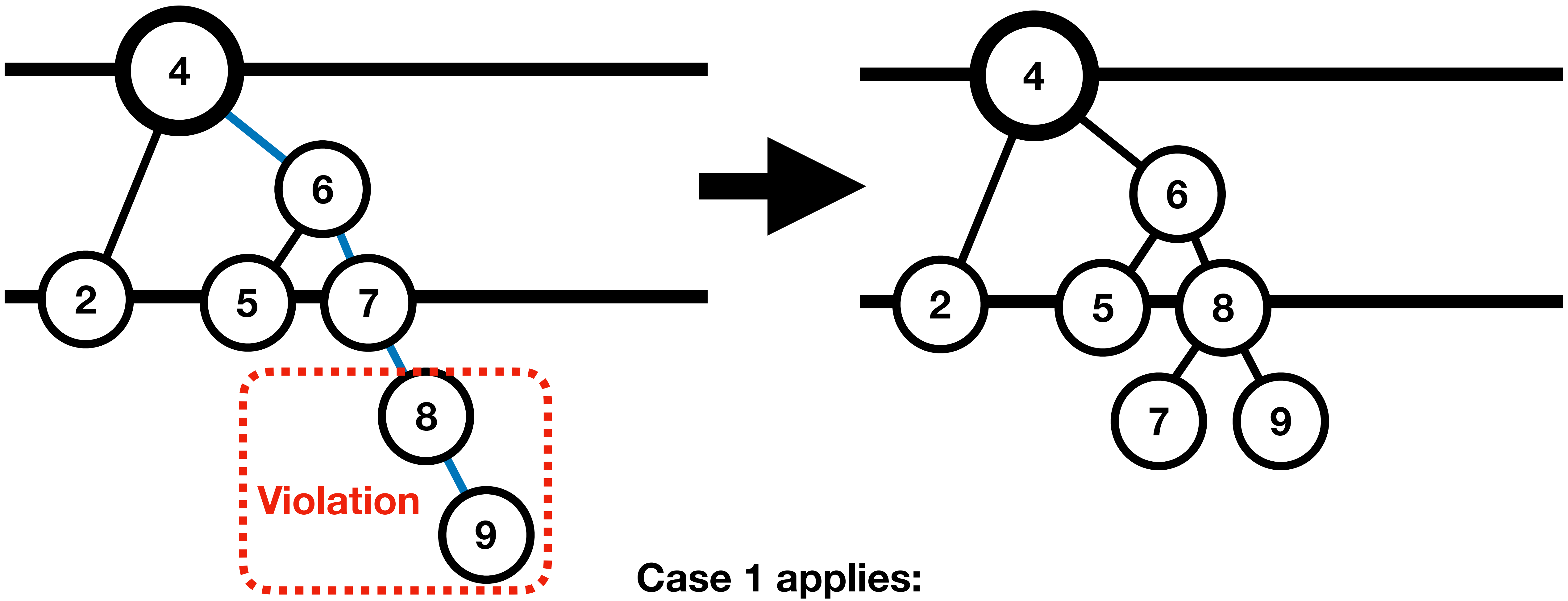
Case 1 applies:
Left-rotate about 5.

Example: Insert 8



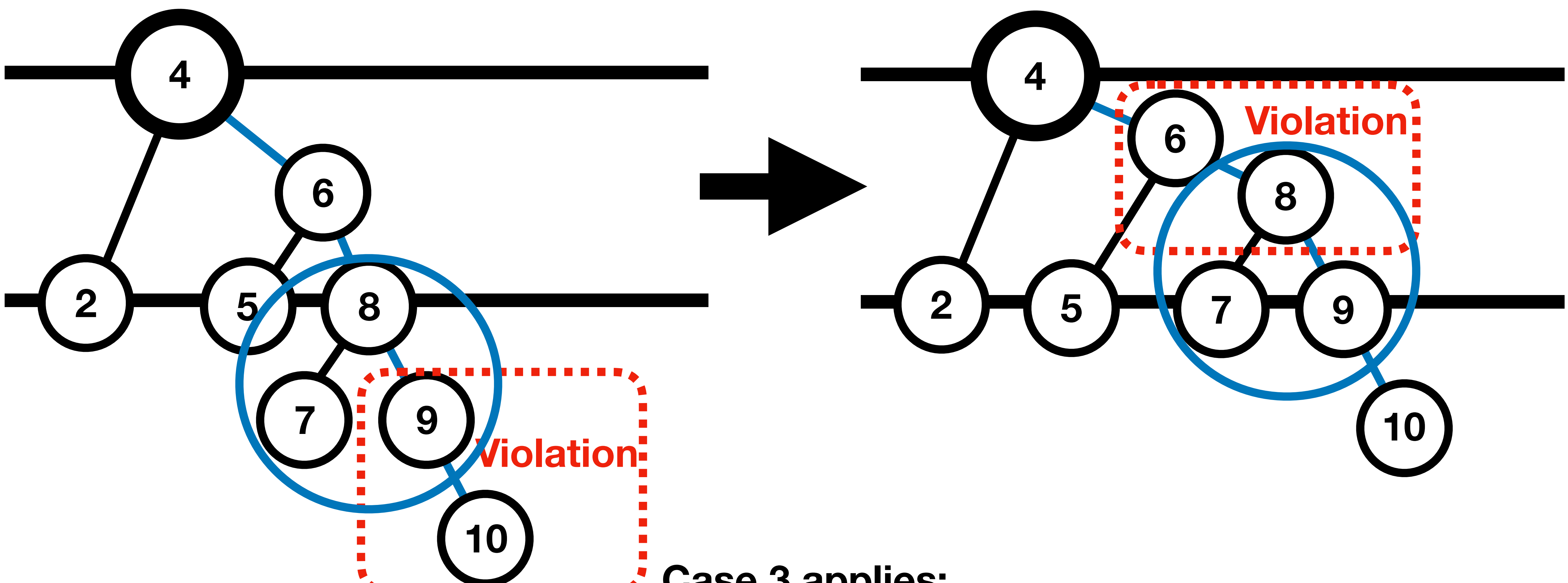
Case 3 applies:
Move 5, 6, 7 by half a gridline.

Example: Insert 9



Case 1 applies:
Left-rotate about 7

Example: Insert 10



Case 3 applies:
Move 7, 8, 9 up half a grid line

Example: Insert 10 (part 2)