Balanced Trees

Part Two

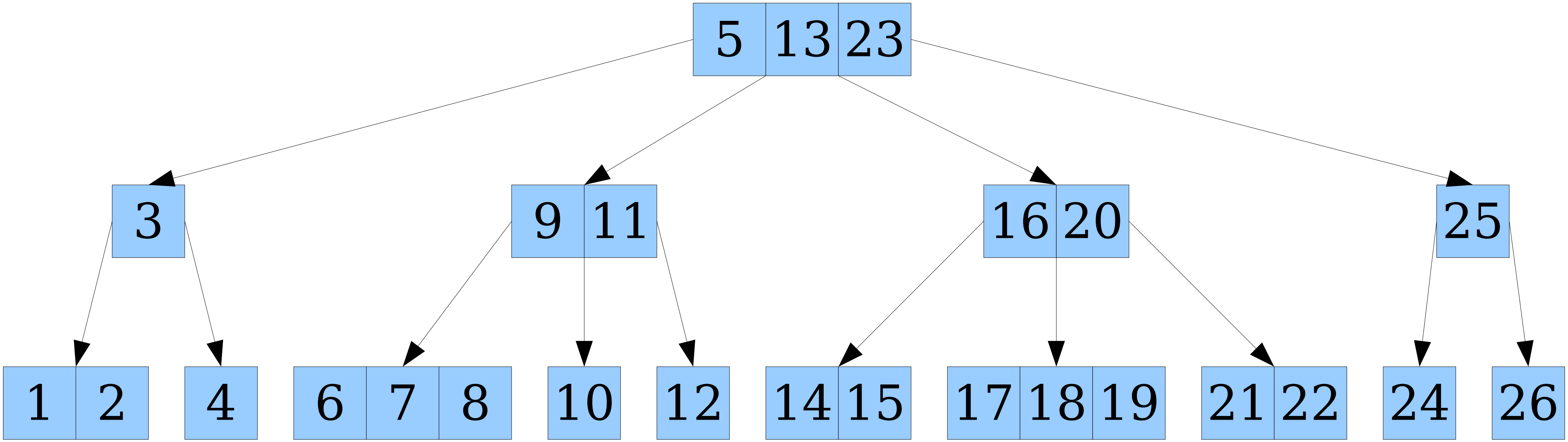
Outline for Today

- ***Red/Black Trees***
  - Using our isometry!
- ***Tree Rotations***
  - A key primitive in restructuring trees.
- ***Augmented Binary Search Trees***
  - Leveraging red/black trees.

Recap from Last Time

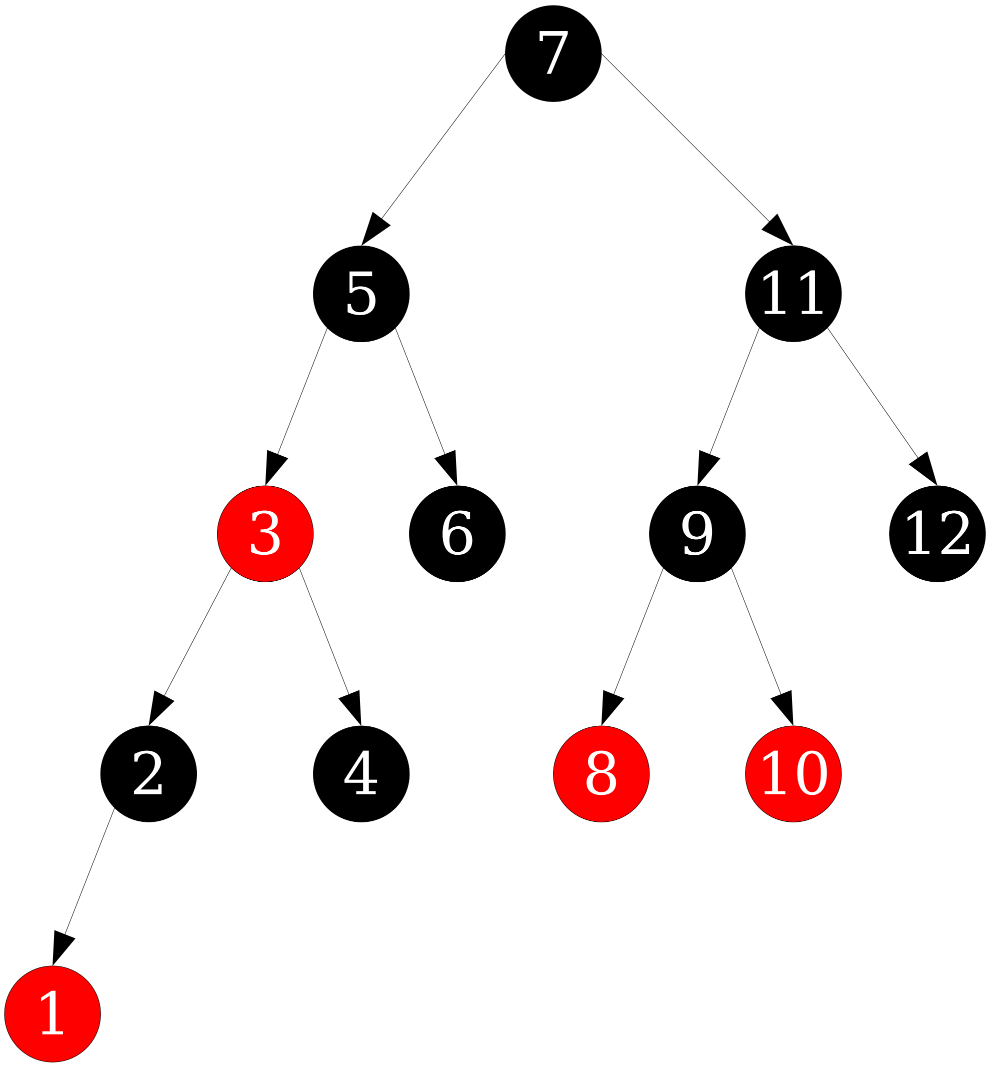
2-3-4 Trees

- A ***2-3-4 tree*** is a multiway search tree where
  - every node has 1, 2, or 3 keys,
  - any non-leaf node with *k* keys has exactly *k*+1 children, and
  - all leaves are at the same depth.
- To insert a key, place it in a leaf. If out of space, split the leaf and kick the median key one level higher, repeating this process.



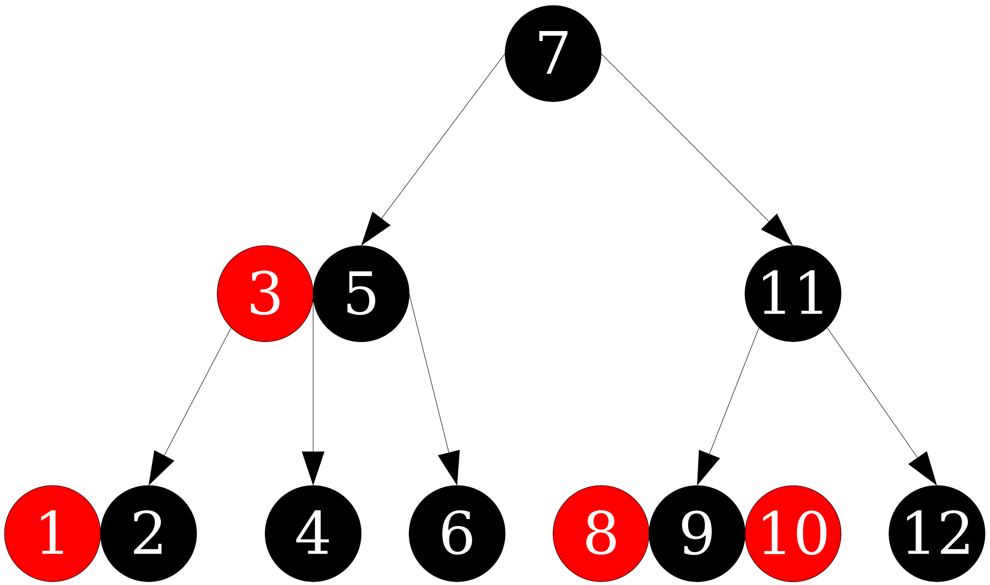
Red/Black Trees

- A ***red/black tree*** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.



Red/Black Trees

- A ***red/black tree*** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
- After we hoist red nodes into their parents:
  - Each “meta node” has 1, 2, or 3 keys in it. (No red node has a red child.)
  - Each “meta node” is either a leaf or has one more child than key. (Root-null path property.)
  - Each “meta leaf” is at the same depth. (Root-null path property.)



This is a 2-3-4 tree!

New Stuff!

Data Structure Isometries

- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- That gives us some really easy theorems basically for free.
- ***Theorem:*** The maximum height of a red/black tree with n nodes is O(log n).

Data Structure Isometries

- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- That gives us some really easy theorems basically for free.
- ***Theorem:*** The maximum height of a red/black tree with n nodes is O(log n).

Explain why, using the isometry.

Formulate a hypothesis!

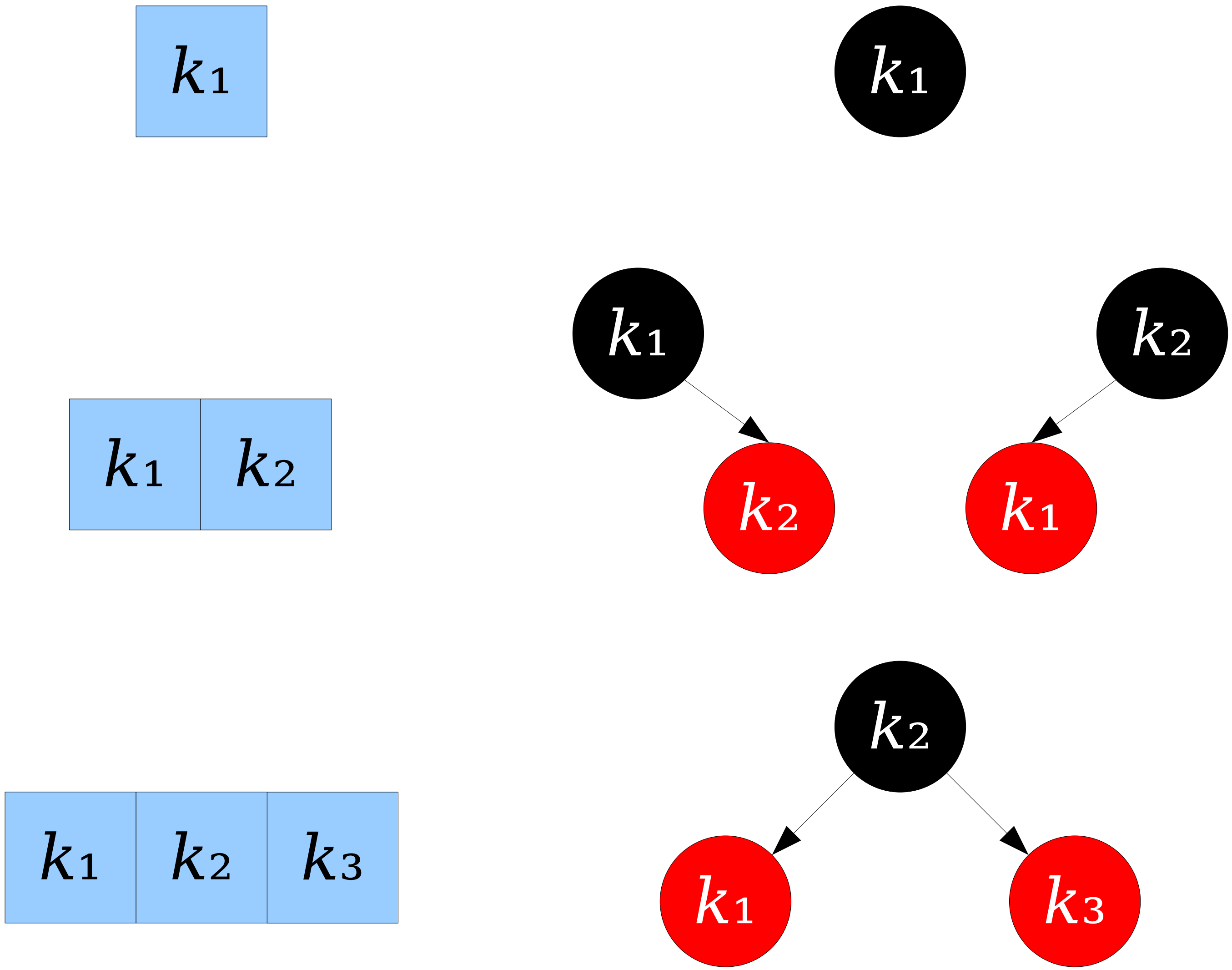
Data Structure Isometries

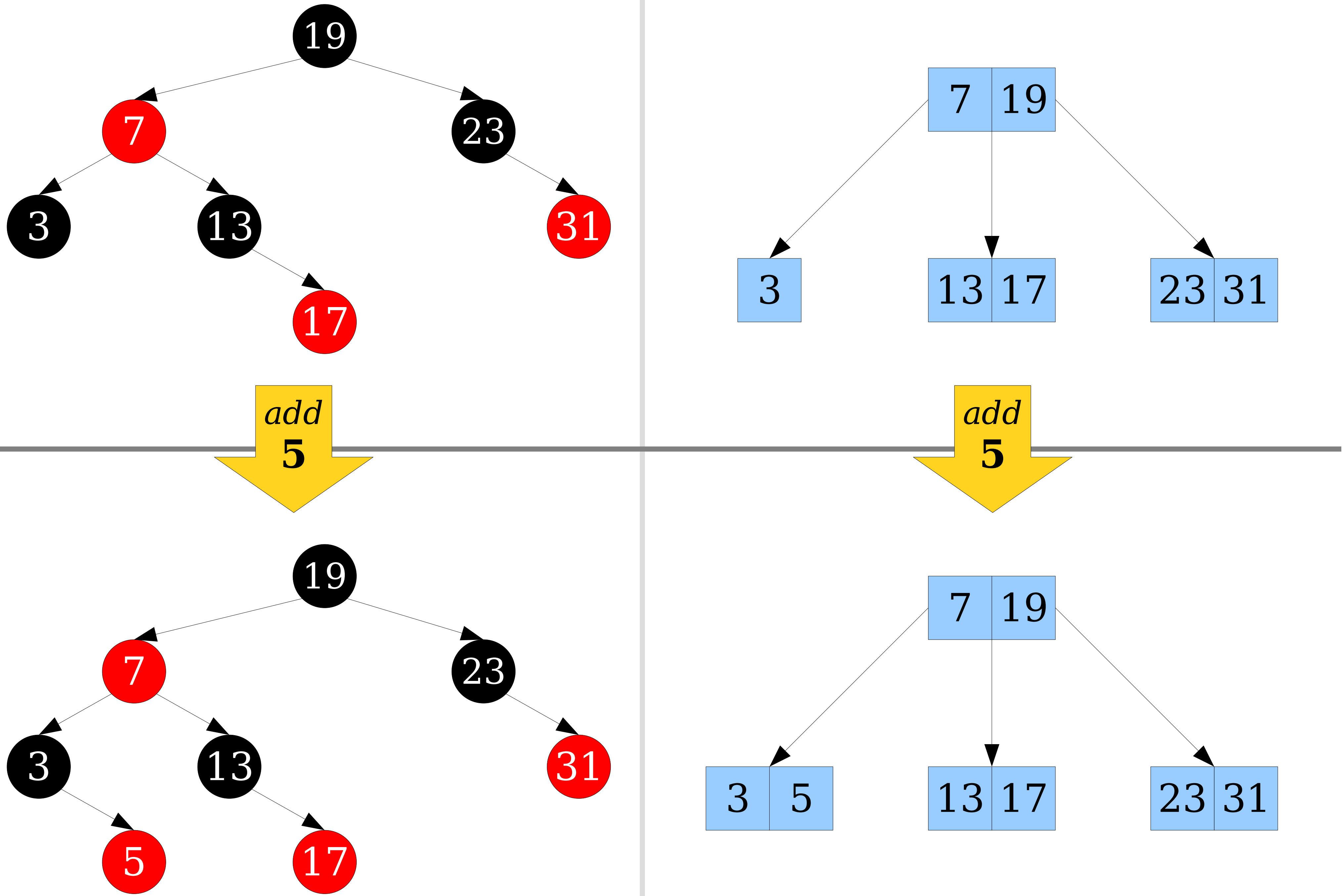
- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- That gives us some really easy theorems basically for free.
- ***Theorem:*** The maximum height of a red/black tree with n nodes is O(log n).
- ***Proof idea:*** Pulling red nodes into their parents forms a 2-3-4 tree with n keys, which has height O(log n). Undoing this at most doubles the height of the tree. ■-*ish*

Exploring the Isometry

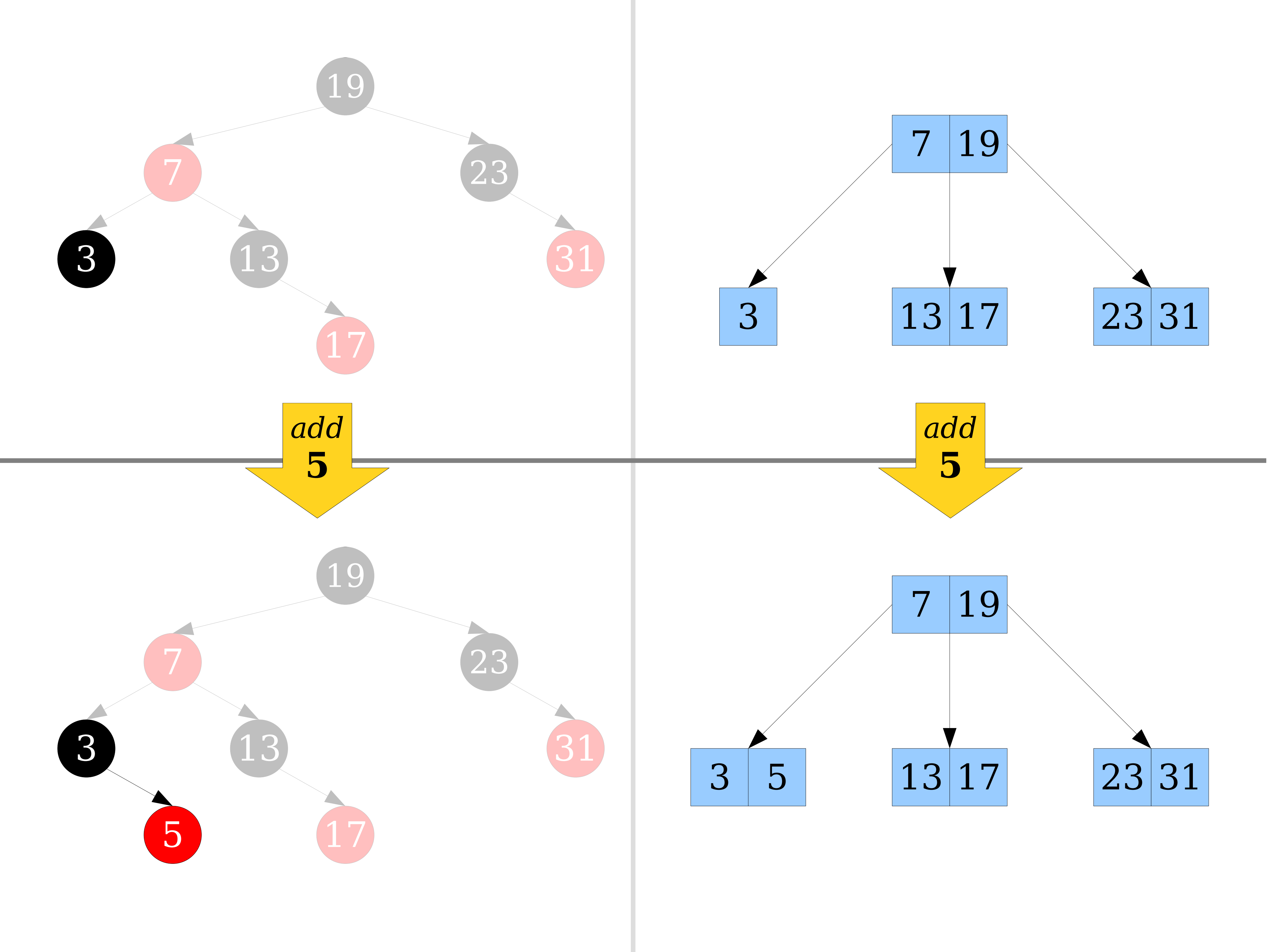
- Nodes in a 2-3-4 tree are classified into types based on the number of children they can have.
  - ***2-nodes*** have one key (two children).
  - ***3-nodes*** have two keys (three children).
  - ***4-nodes*** have three keys (four children).
- How might these nodes be represented?

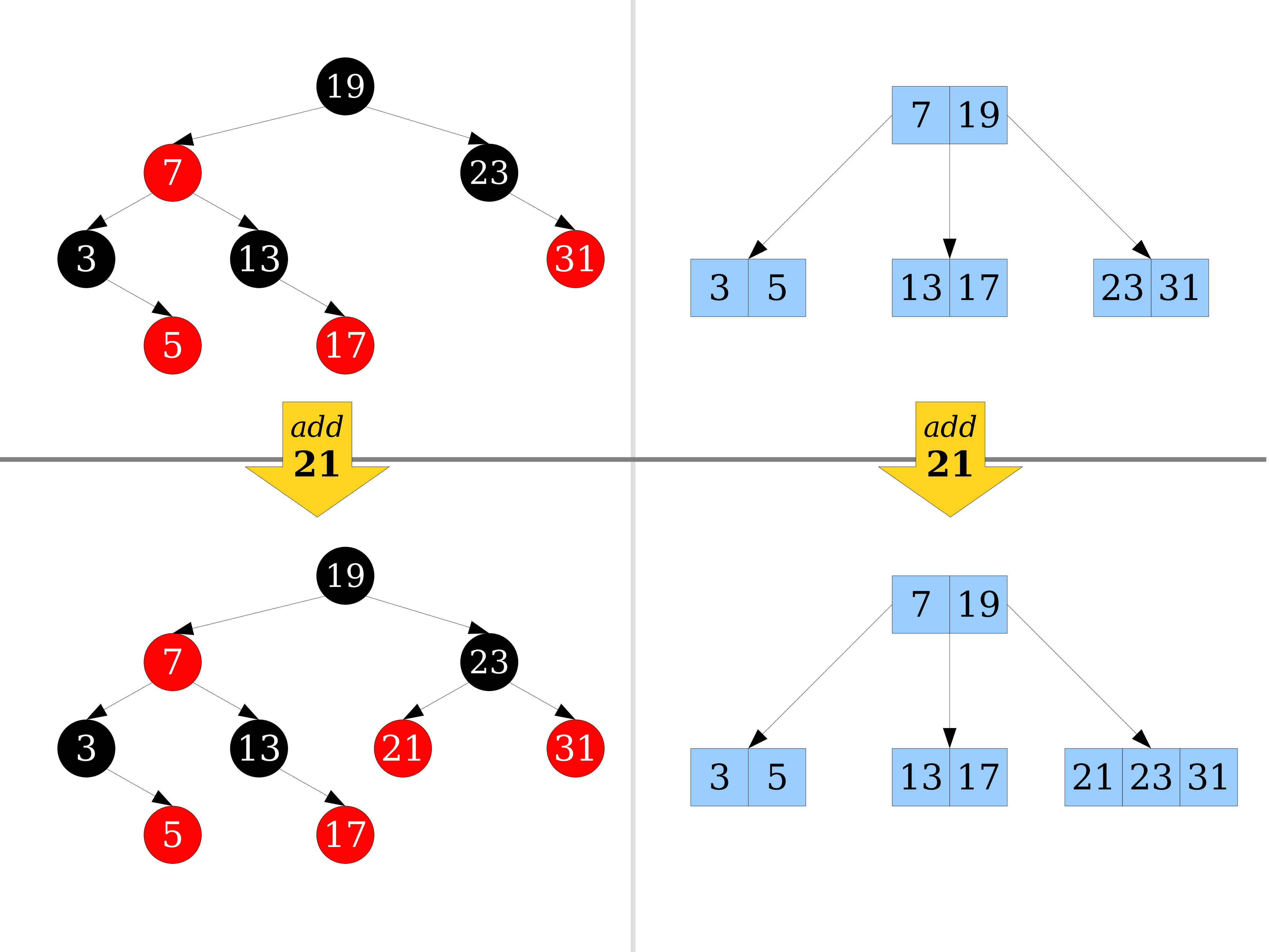
Exploring the Isometry



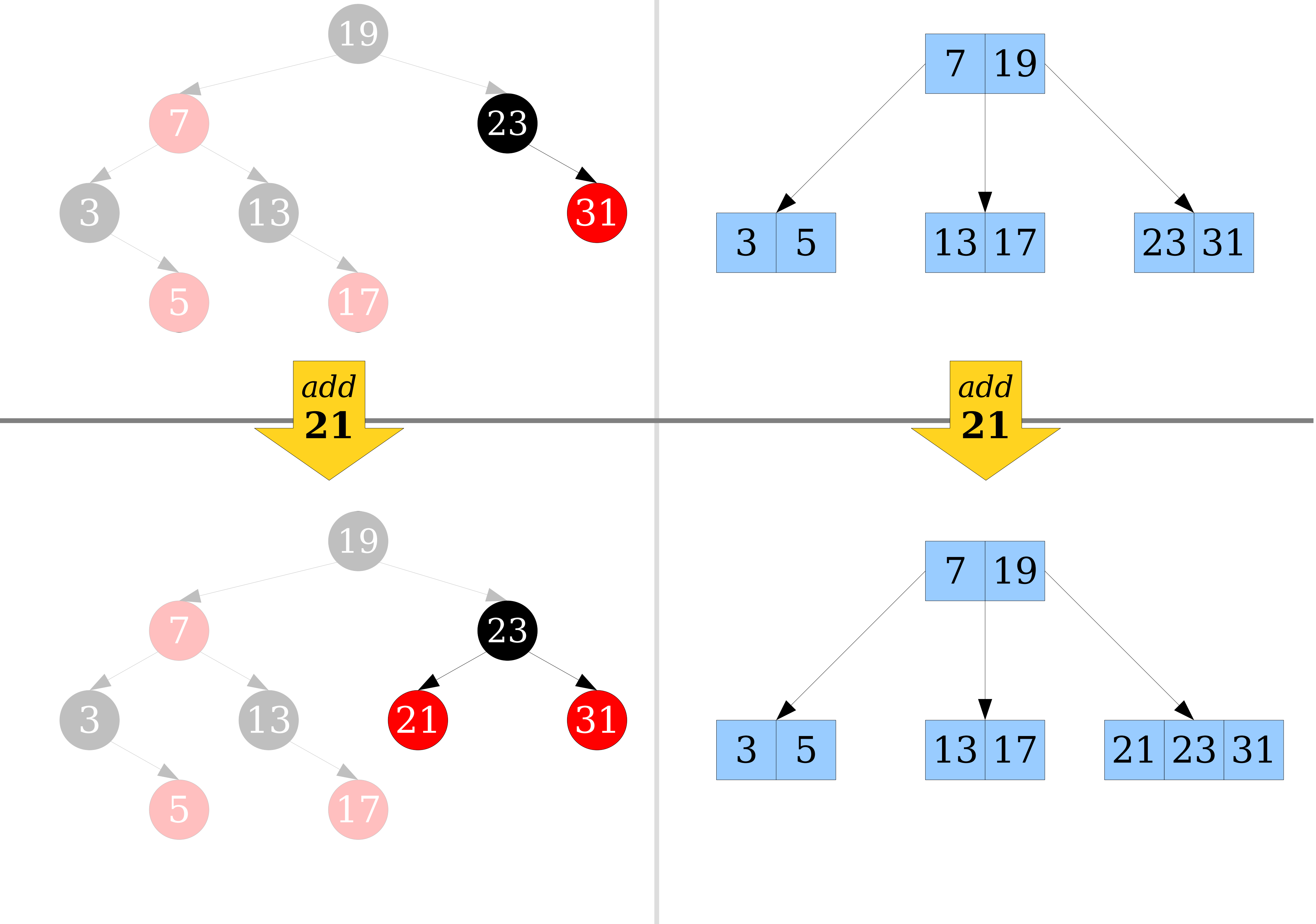


19
7
23
3
13
31
17
7
19
3
13
17
23
31
add
5
add
5
19
7
23
3
13
31
5
17
7
19
3
5
13
17
23
31


19
7
23
3
13
31
17
7 19
3
13 17
23 31
add
5
add
5
19
7
23
3
13
31
5
17
7 19
3 5
13 17
23 31

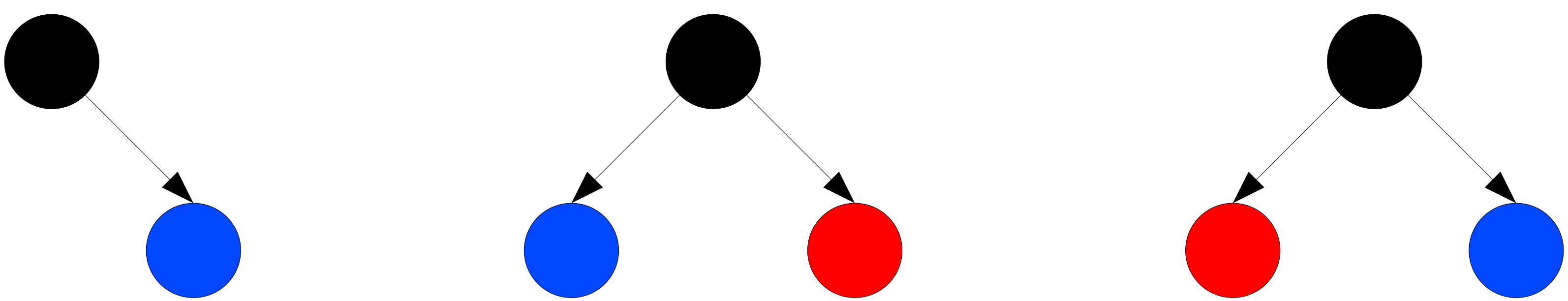


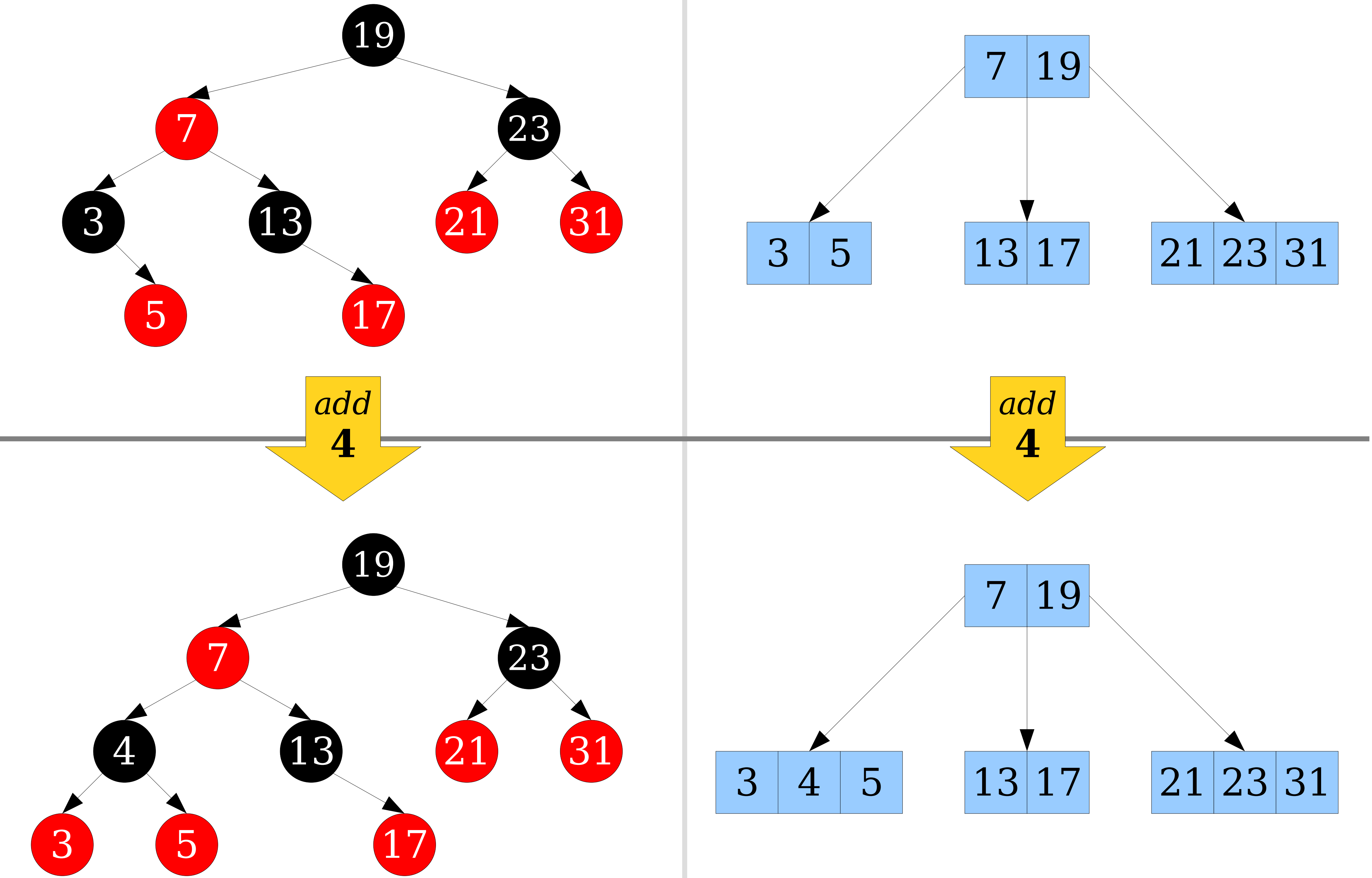
19
7
23
3
13
31
5
17
7 19
3 5
13 17
23 31
add 21
add 21
19
7
23
3
13
21
31
5
17
7 19
3 5
13 17
21 23 31

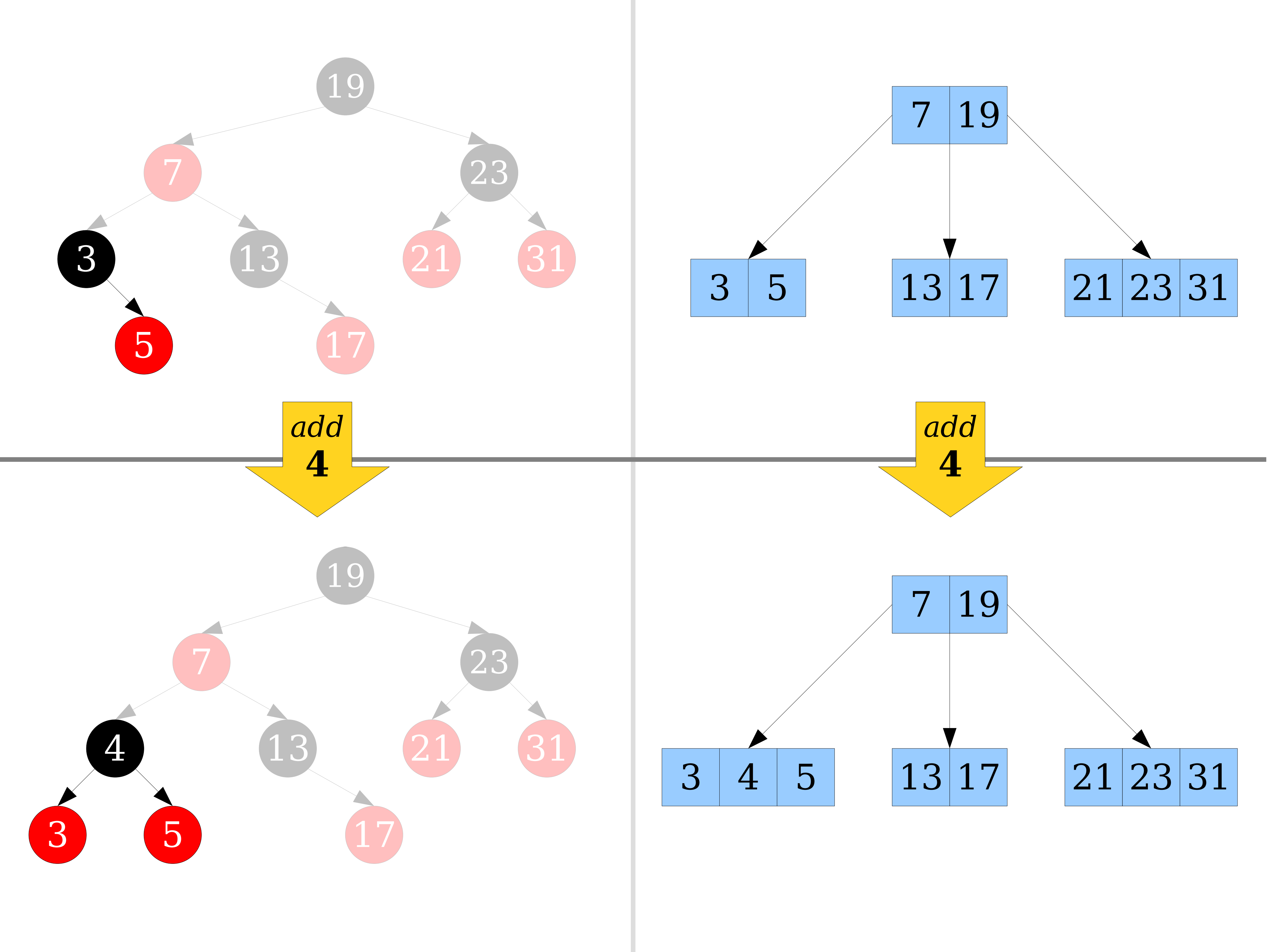

19
7
23
3
13
31
5
17
7 19
3 5
13 17
23 31
add 21
add 21
19
7
23
3
13
21
31
5
17
7 19
3 5
13 17
21 23 31

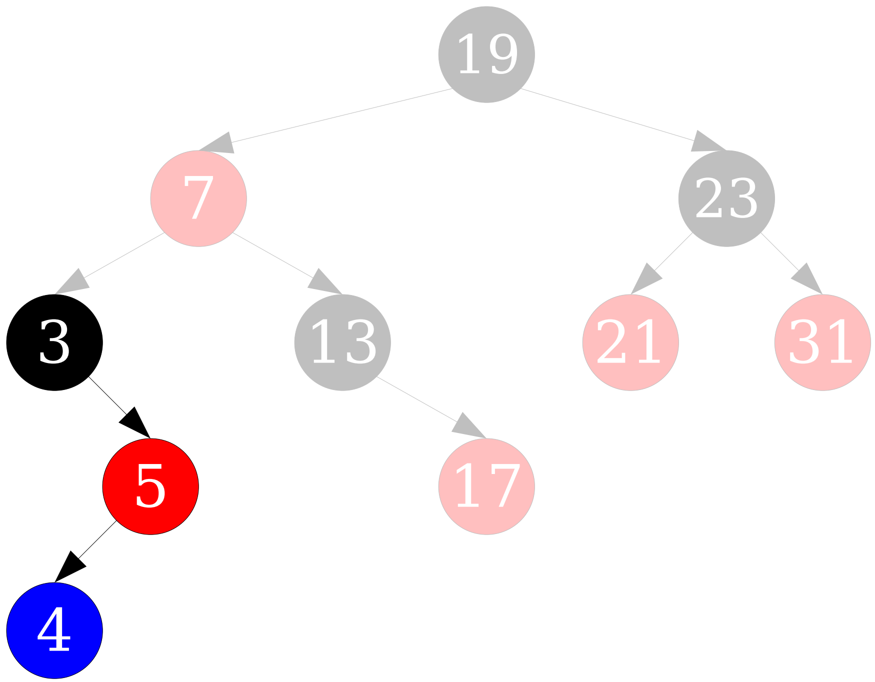
Red/Black Tree Insertion

- ***Rule #1:*** When inserting a node, if its parent is black, make the node red and stop.
- ***Justification:*** This simulates inserting a key into an existing 2-node or 3-node.

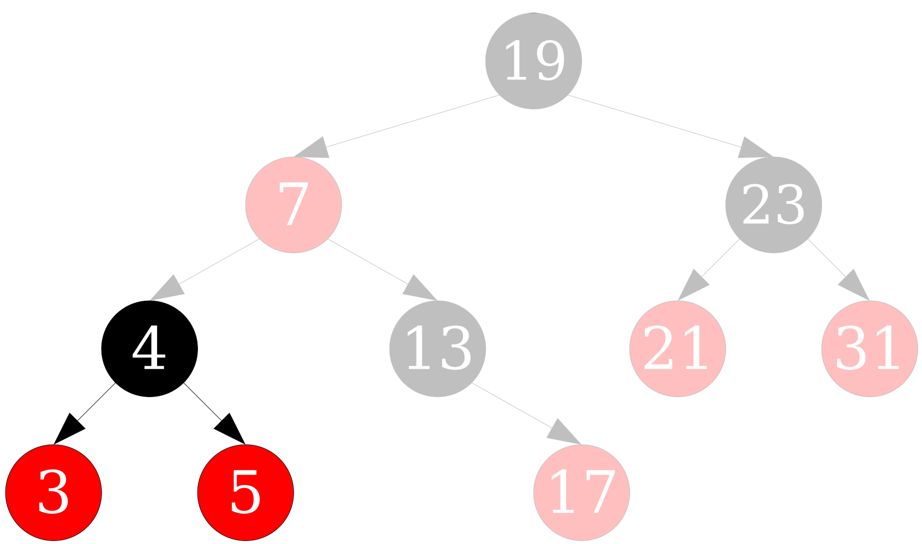



19
7
23
3
13
21
31
5
17
7 19
3 5
13 17
21 23 31
add
4
add
4
19
7
23
4
13
21
31
3
5
17
7 19
3 4 5
13 17
21 23 31


19
7
23
3
13
21
31
5
17
7 19
3 5
13 17
21 23 31
add
4
add
4
19
7
23
4
13
21
31
3
5
17
7 19
3 4 5
13 17
21 23 31

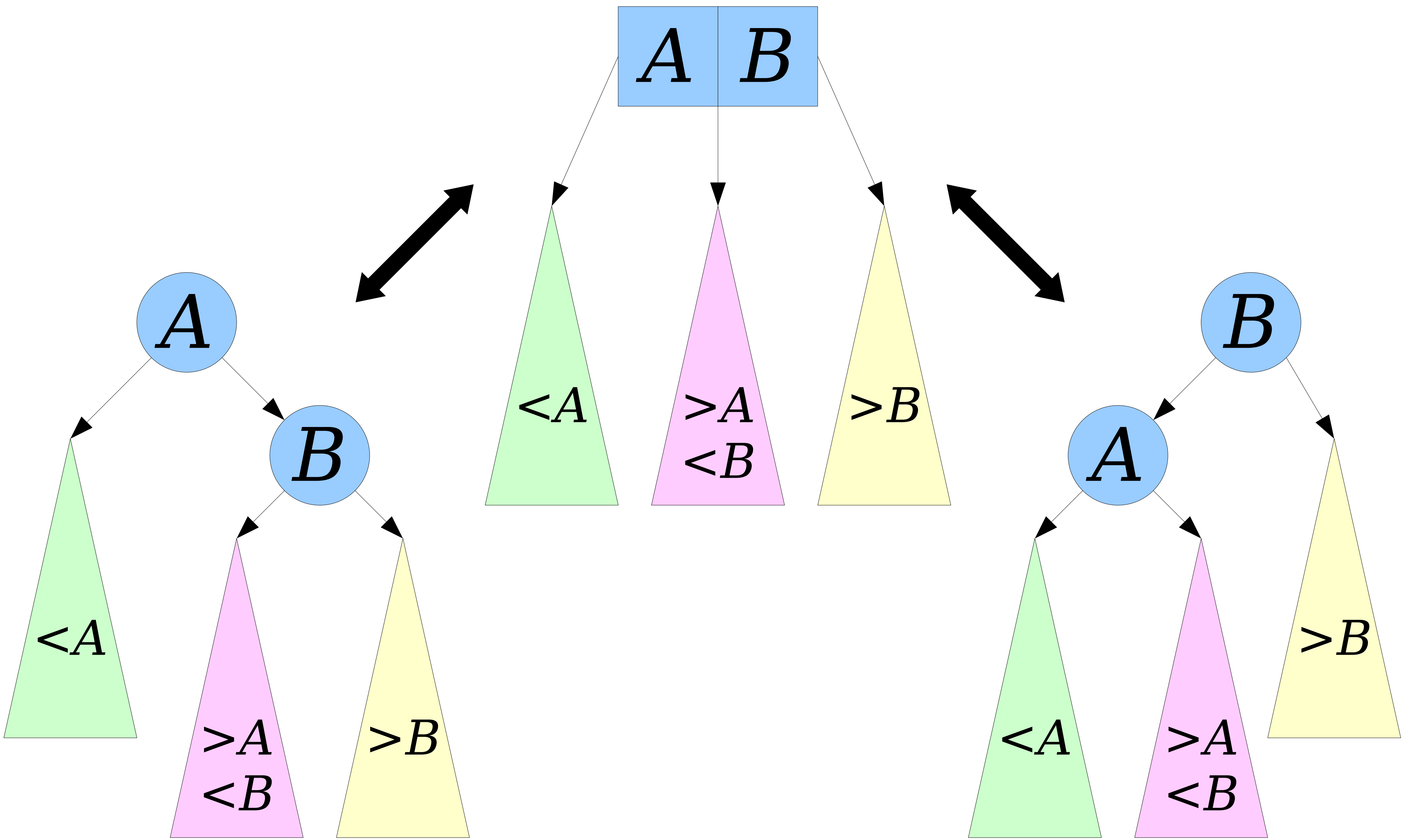


We need to move nodes around in a binary search tree. How do we do this?

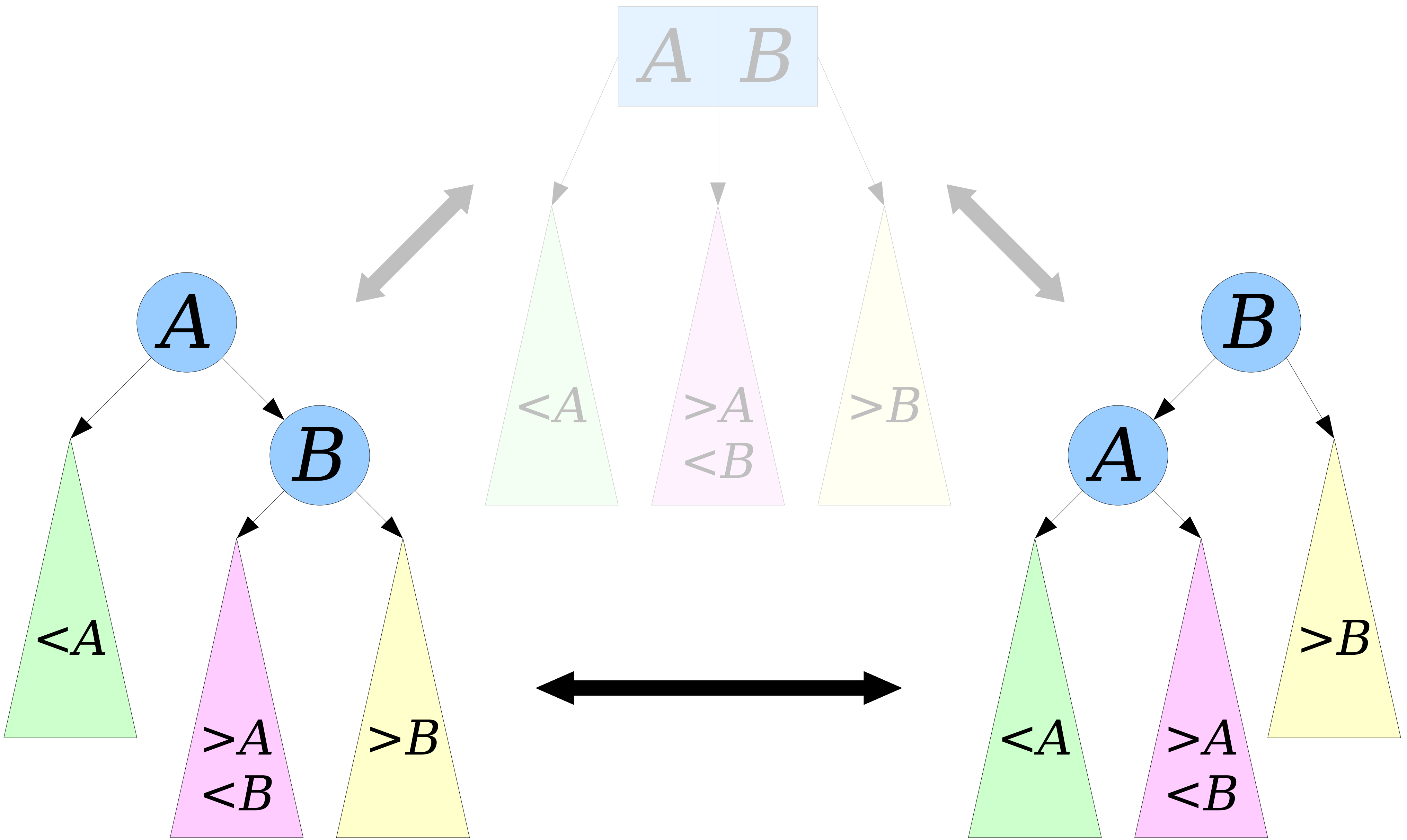


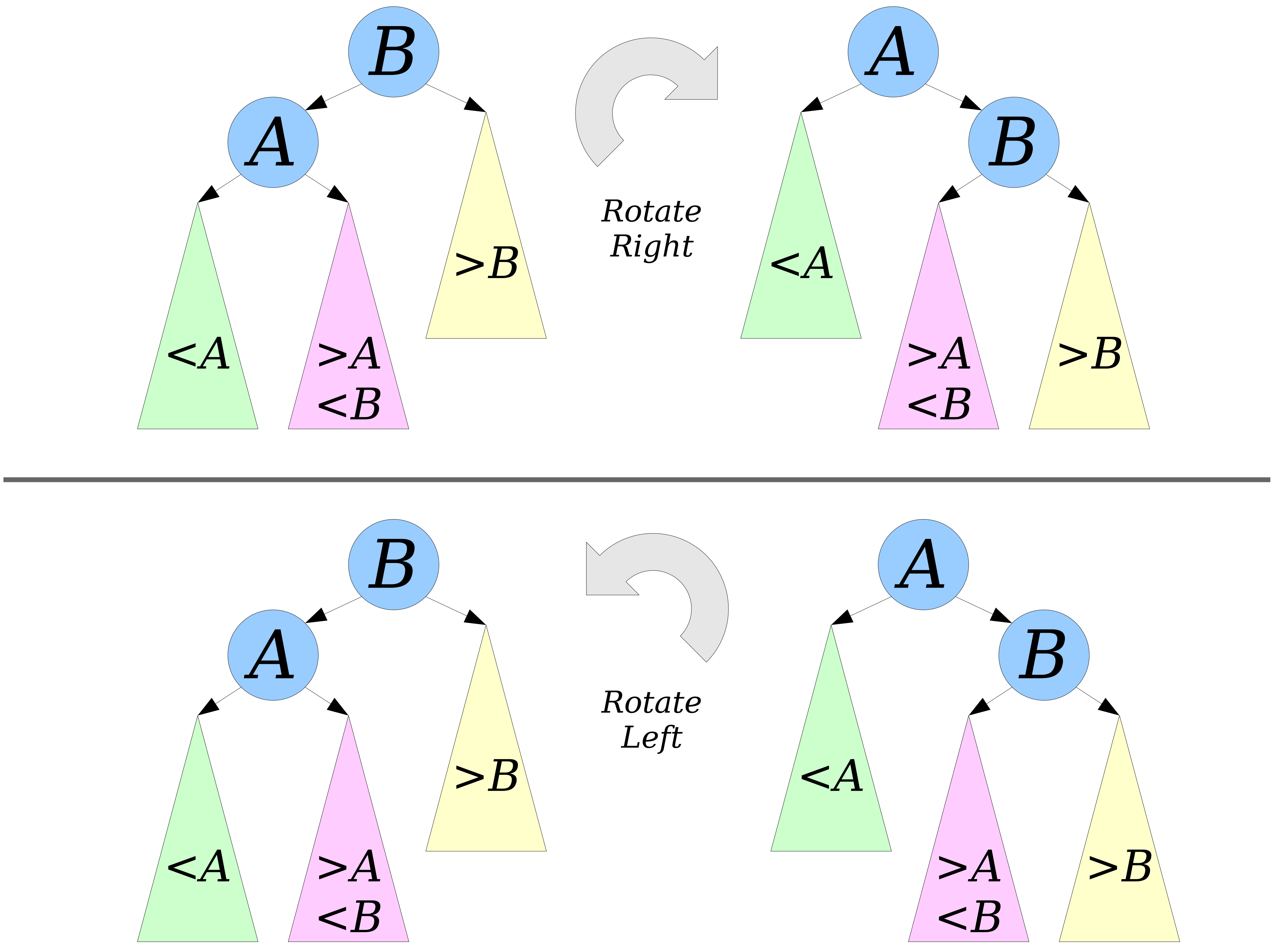
Goal

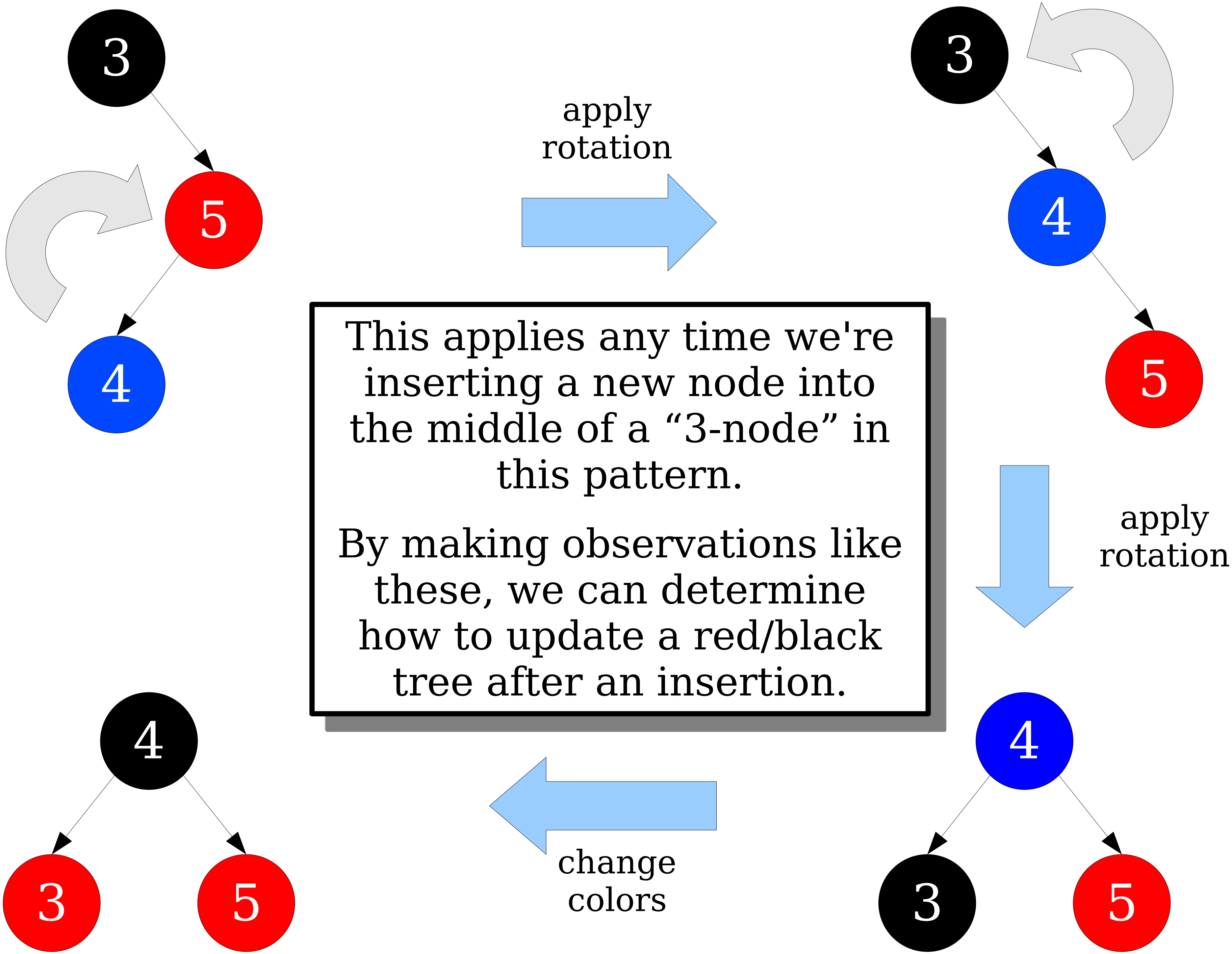
Tree Rotations



Tree Rotations

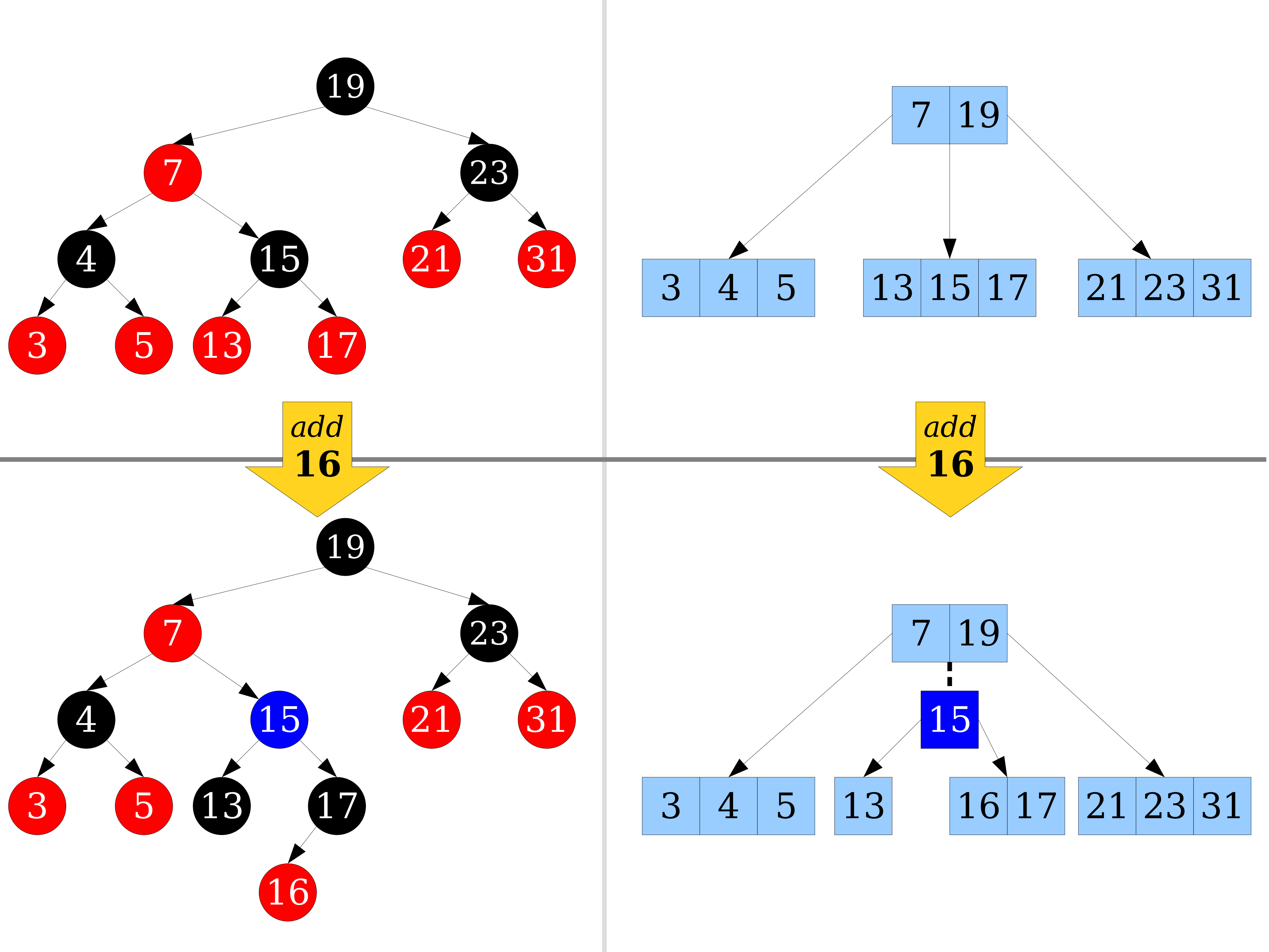


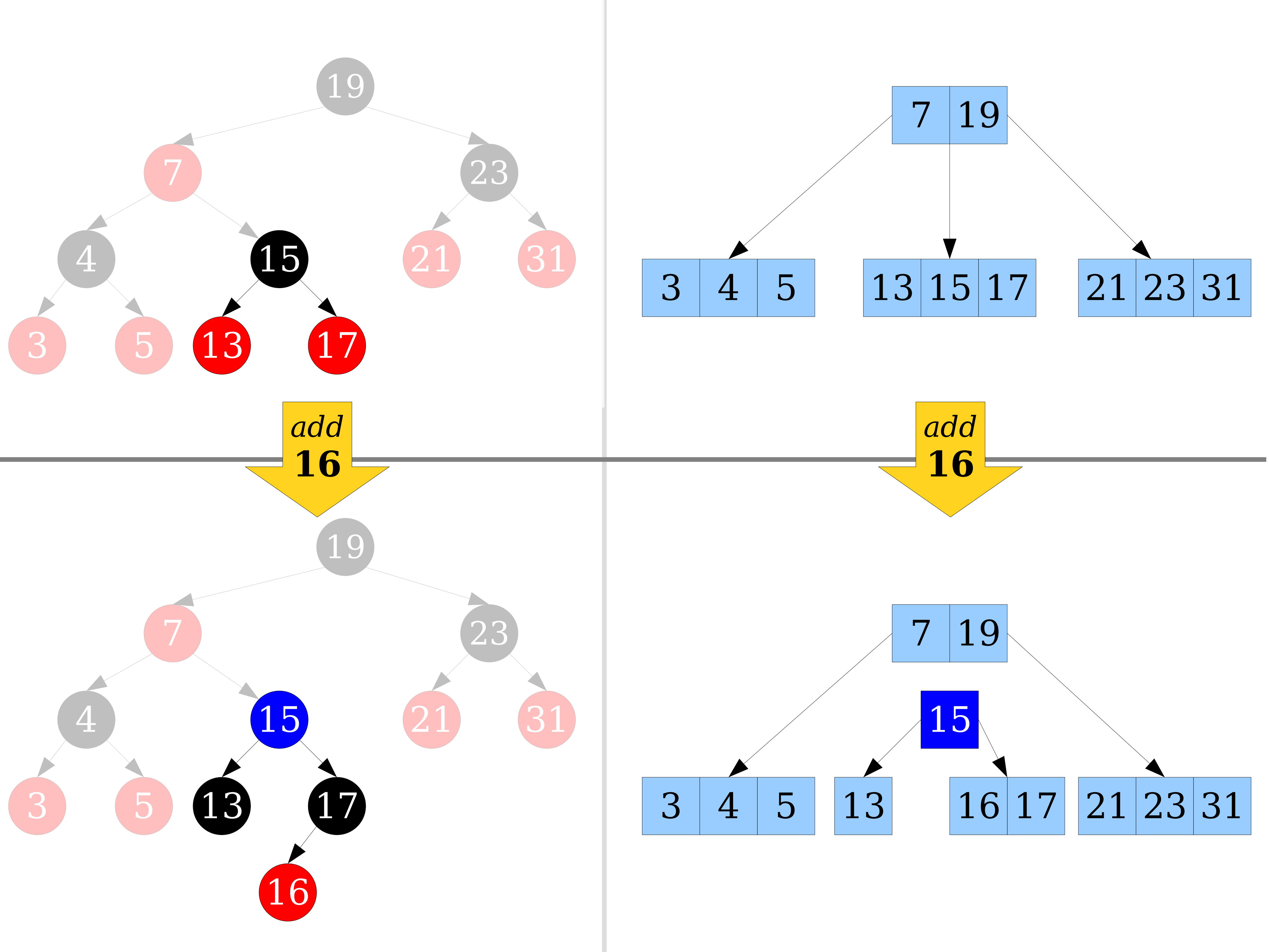

B
A
<A
>A
<B
>B
Rotate
Right
A
B
<A
>A
<B
>B
B
A
<A
>A
<B
>B
Rotate
Left
A
B
<A
>A
<B
>B



This applies any time we're inserting a new node into the middle of a “3-node” in this pattern.

By making observations like these, we can determine how to update a red/black tree after an insertion.


19
7
23
4
15
21
31
3
5
13
17
7 19
3 4 5
13 15 17
21 23 31
add
16
add
16
19
7
23
4
15
21
31
3
5
13
17
16
7 19
15
3 4 5
13
16 17
21 23 31


19
7
23
4
15
21
31
3
5
13
17
7 19
3 4 5
13 15 17
21 23 31
add
16
add
16
19
7
23
4
15
21
31
3
5
13
17
16
7 19
15
3 4 5
13
16 17
21 23 31

Building Up Rules

- The complex rules on red/black trees make perfect sense if you connect it back to 2-3-4 trees.
- There are lots of cases to consider because there are many different ways you can insert into a red/black tree.
- ***Main point:*** Simulating the insertion of a key into a node takes time O(1) in all cases. Therefore, since 2-3-4 trees support O(log n) insertions, red/black trees support O(log n) insertions.
- The same is true of deletions.

My Advice

- ***Do*** know how to do B-tree insertions and searches.
  - You can derive these easily if you remember to split nodes.
- ***Do*** remember the rules for red/black trees and B-trees.
  - These are useful for proving bounds and deriving results.
- ***Do*** remember the isometry between red/black trees and 2-3-4 trees.
  - Gives immediate intuition for all the red/black tree operations.
- ***Don't*** memorize the red/black rotations and color flips.
  - This is rarely useful. If you're coding up a red/black tree, just flip open CLRS and translate the pseudocode. ☺

Dynamic Problems

Classical Algorithms

- The "classical" algorithms model goes something like this:

Given some input X, compute some interesting function f(X).

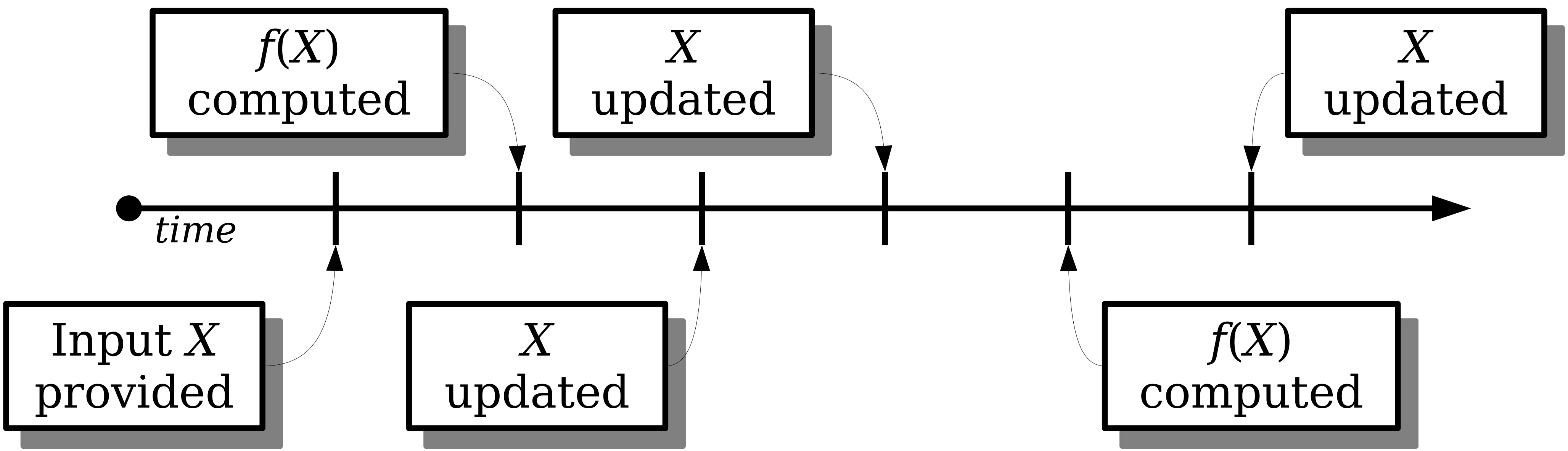
- The input X is provided up front, and only a single answer is produced.

Dynamic Problems

- ***Dynamic versions*** of problems are framed like this:

Given an input X that can change in fixed ways, maintain X while being able to compute $f(X)$ efficiently at any point in time.

- These problems are typically harder to solve efficiently than the "classical" static versions.



Dynamic Selection

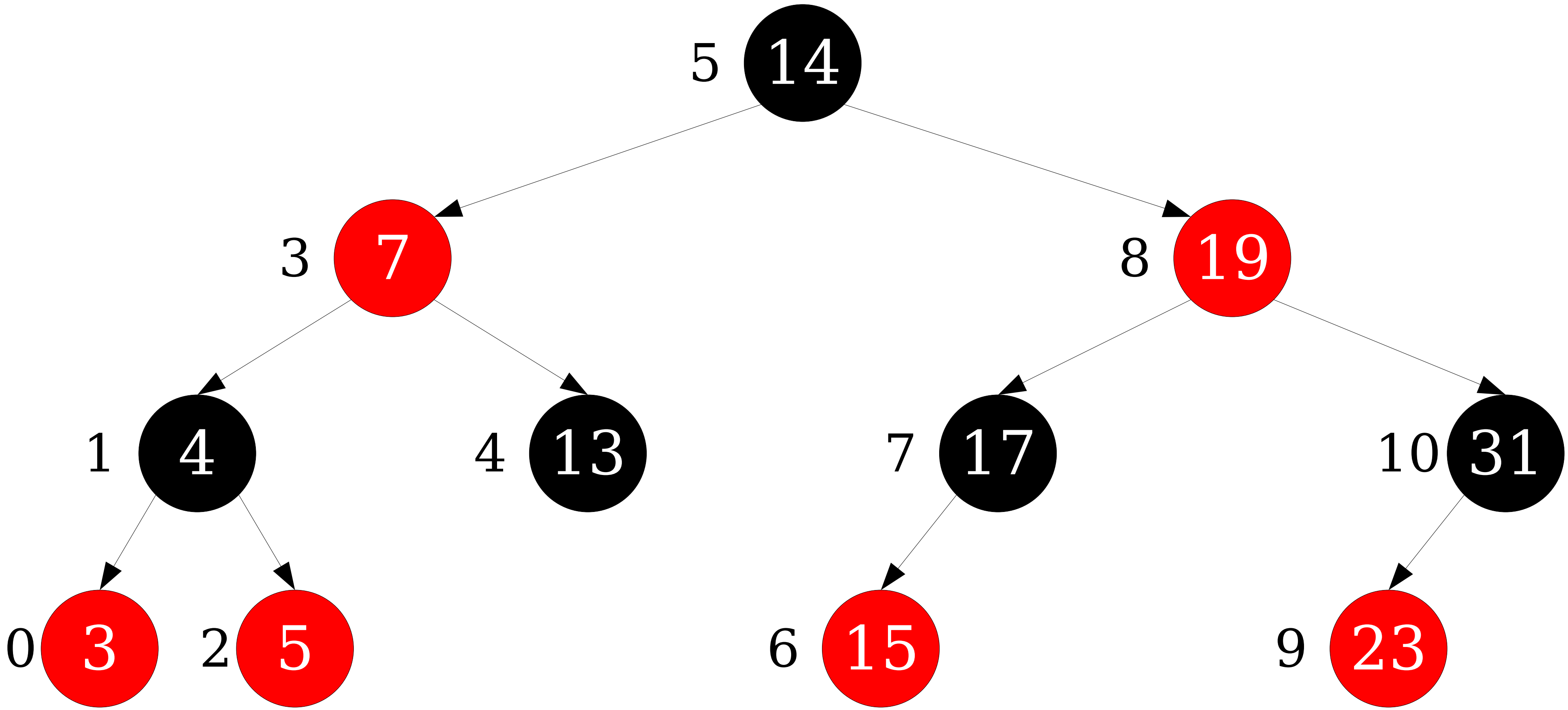
- The ***selection*** problem is the following:

Given a list of distinct values and a number k, return the kth-smallest value.

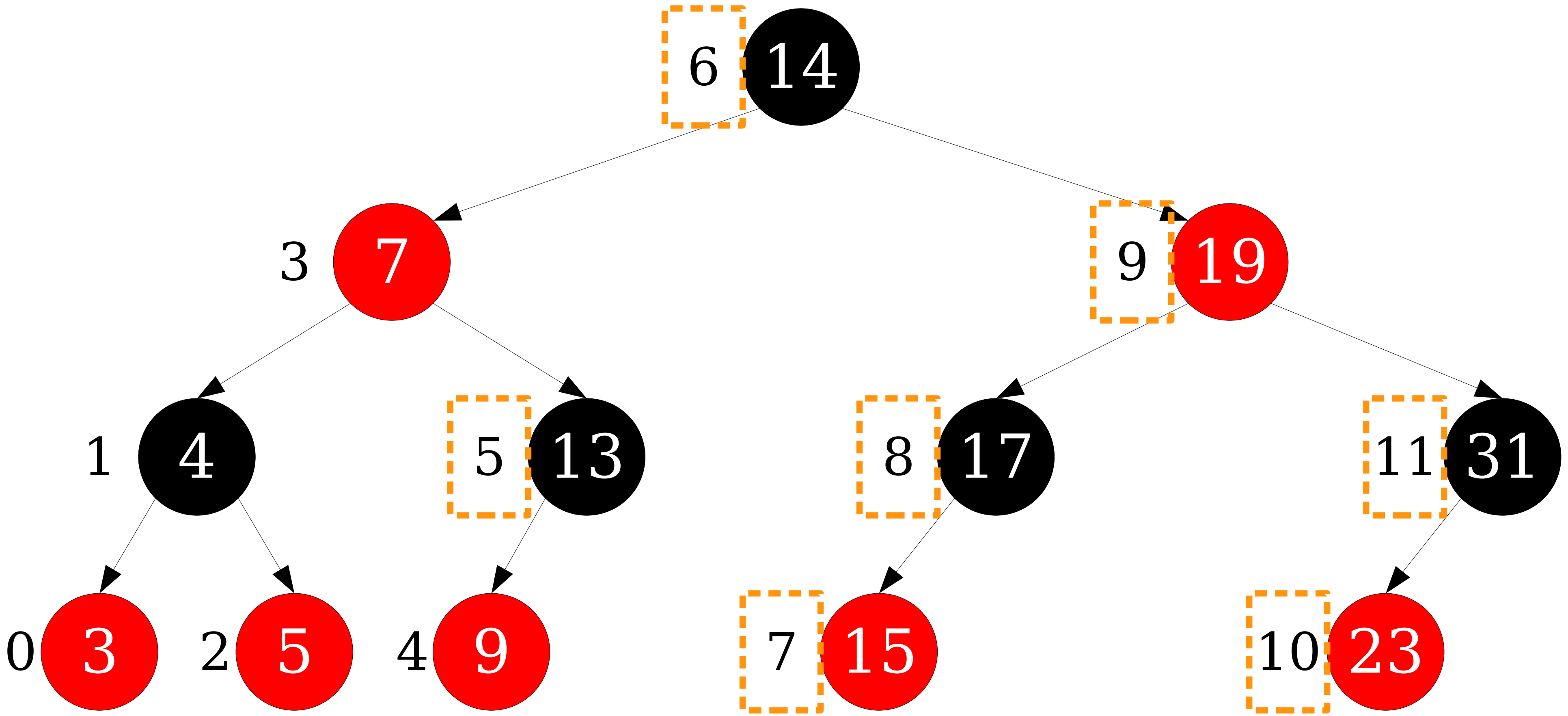
- In the static case, where the data set is fixed in advance and k is known, we can solve this in time $O(n)$ using quickselect or the median-of-medians algorithm.
- ***Goal:*** Solve this problem efficiently when the data set is changing – that is, the underlying set of elements can have insertions and deletions intermixed with queries.

31	41	59	26	53	58	79

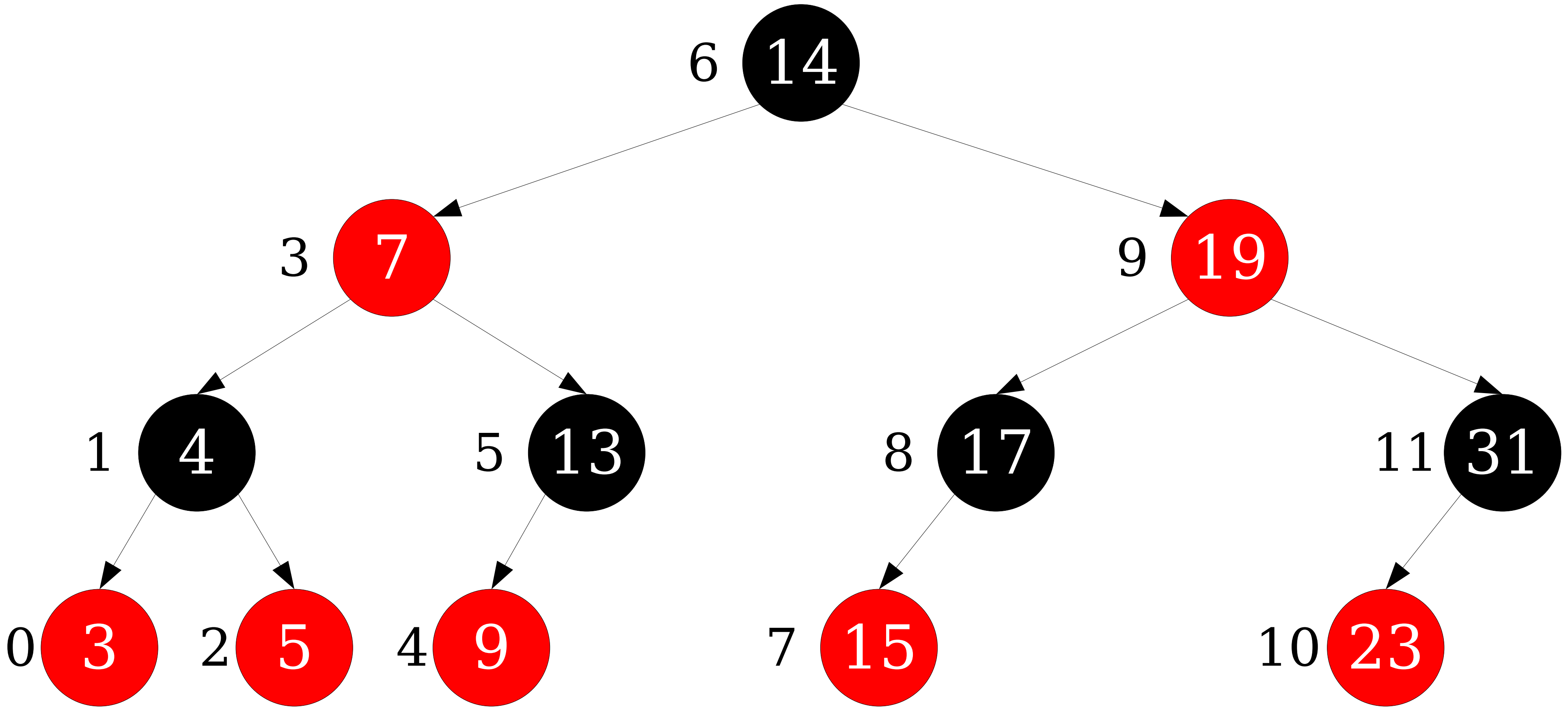
Dynamic Selection



Dynamic Selection



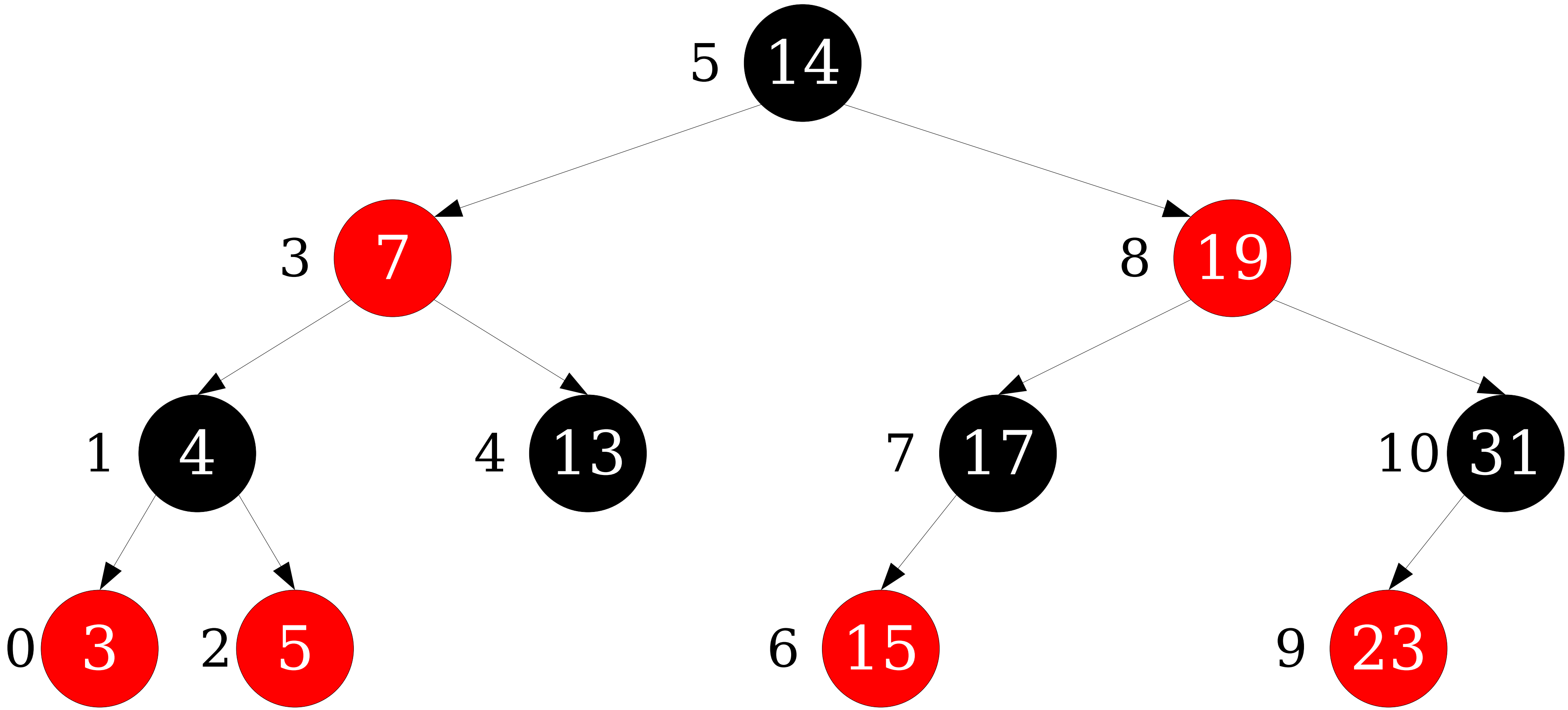
Dynamic Selection



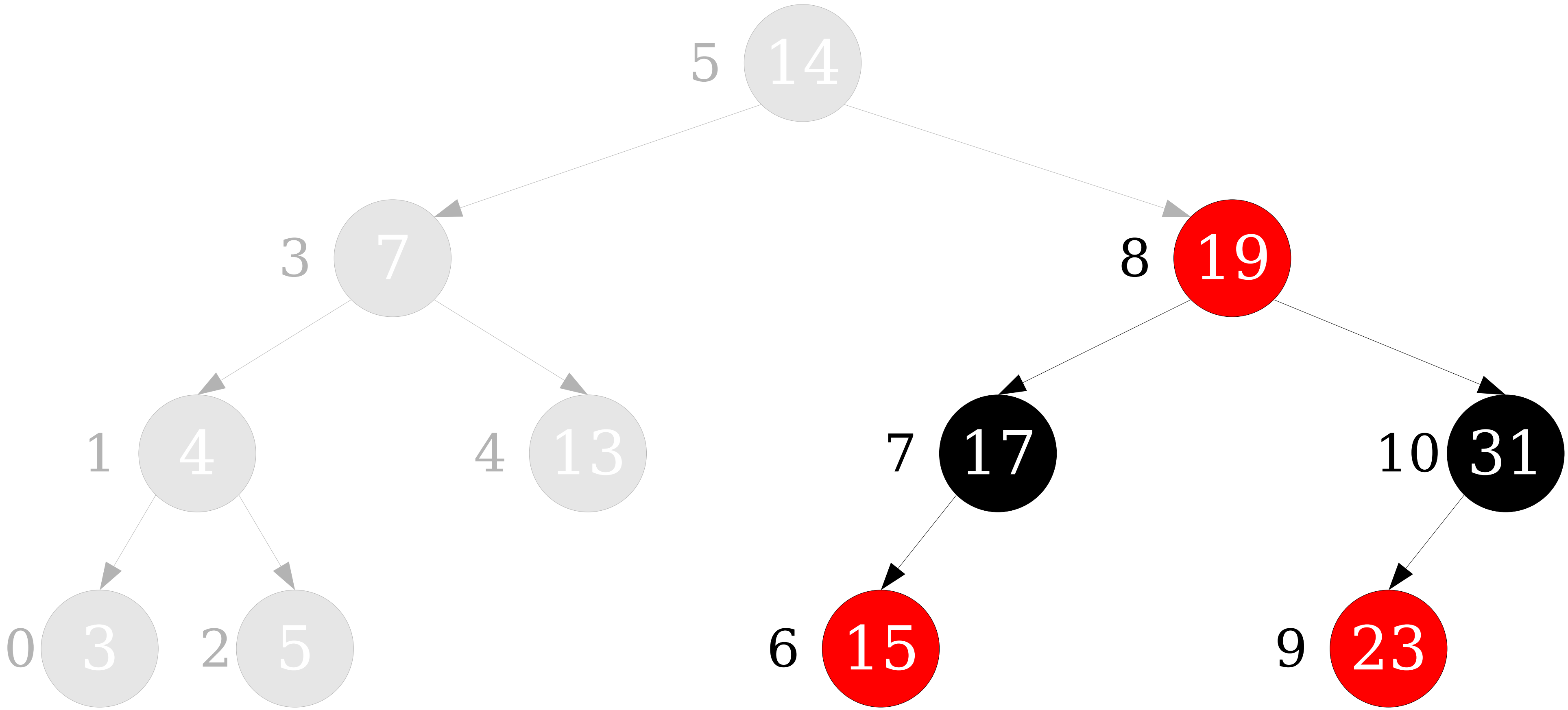
Problem: After inserting a new value, we may have to update $\Theta(n)$ values.

This is inherent in this solution route. These numbers track *global* properties of the tree.

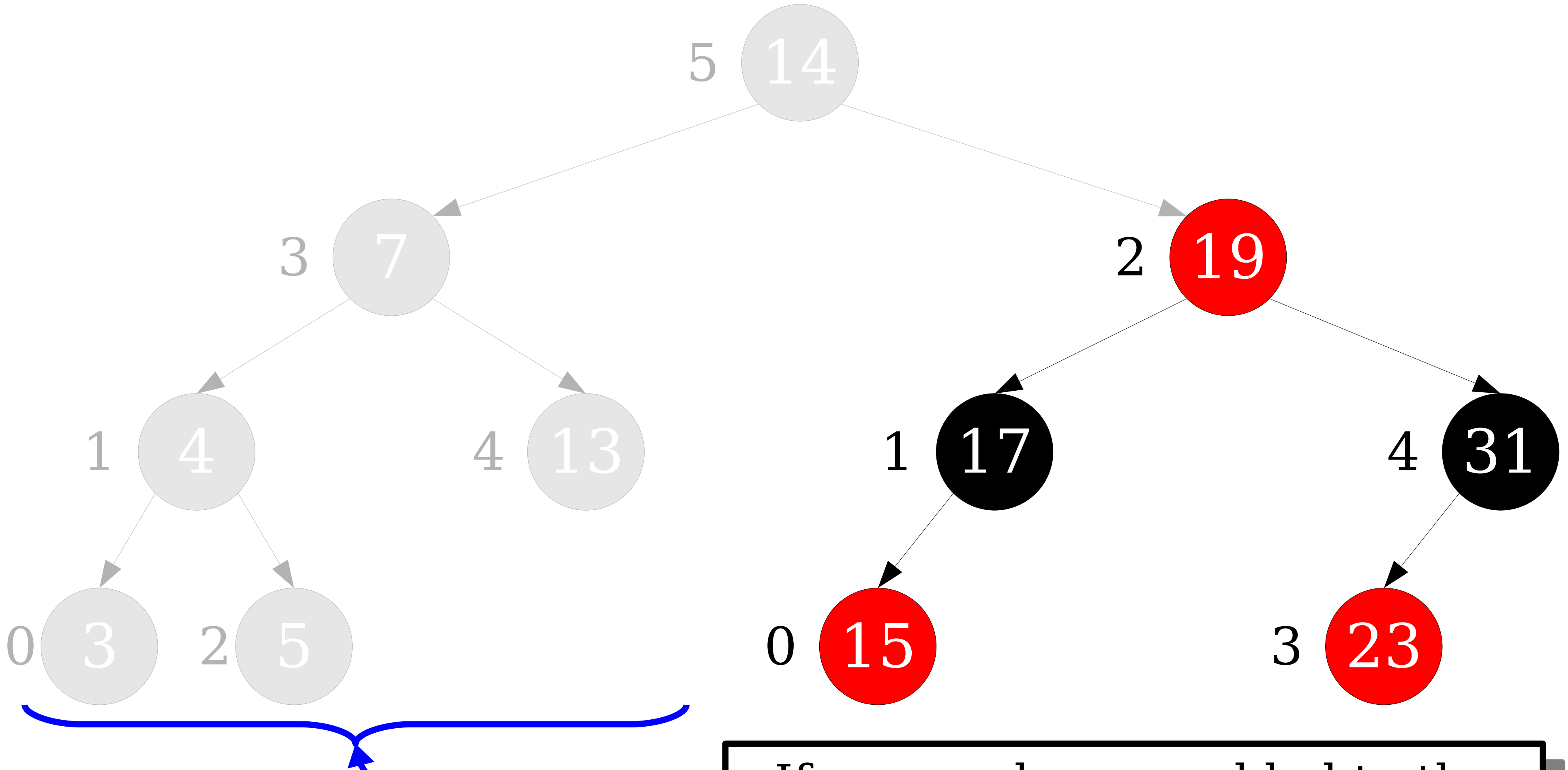
Dynamic Selection



Dynamic Selection

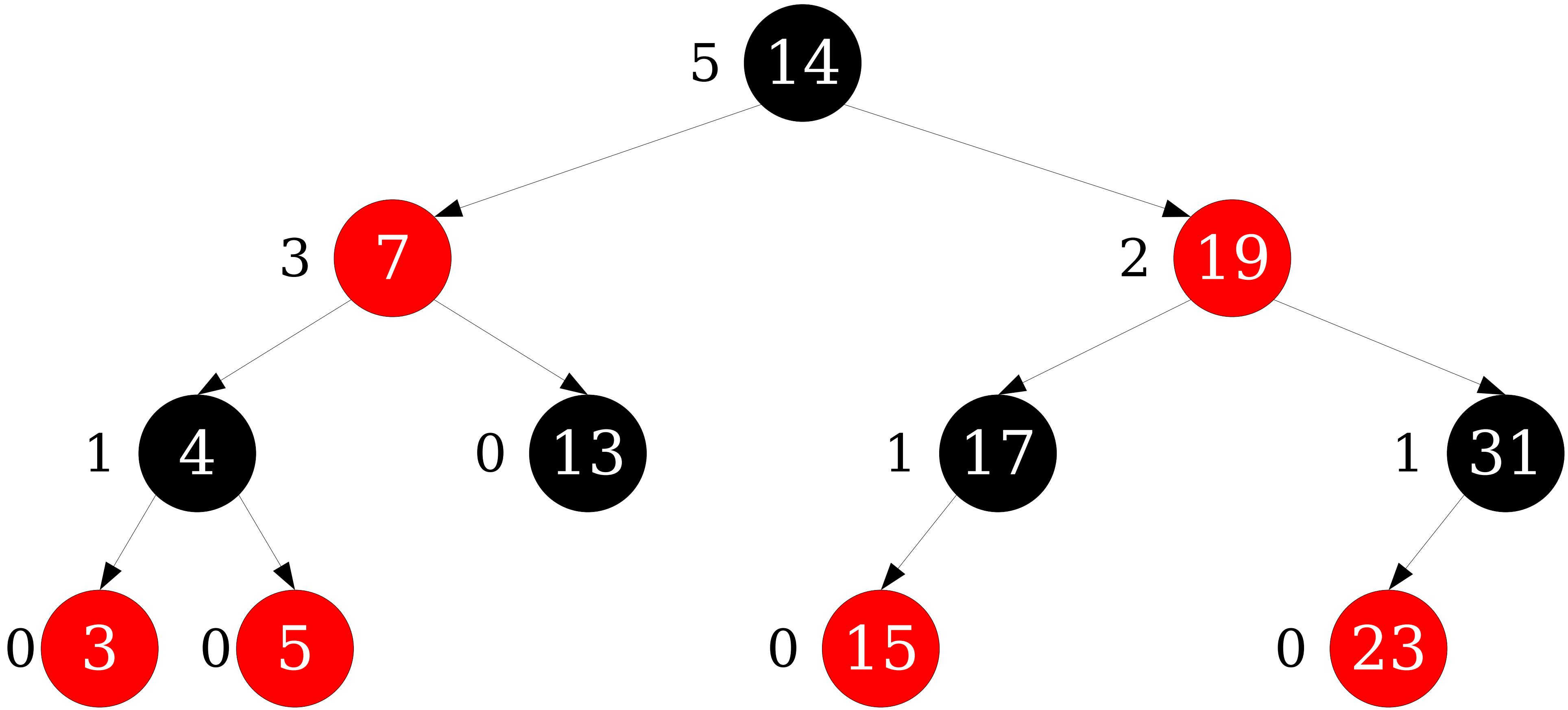


Dynamic Selection



If new nodes are added to the the left subtree, the numbers on the right don't need to update.

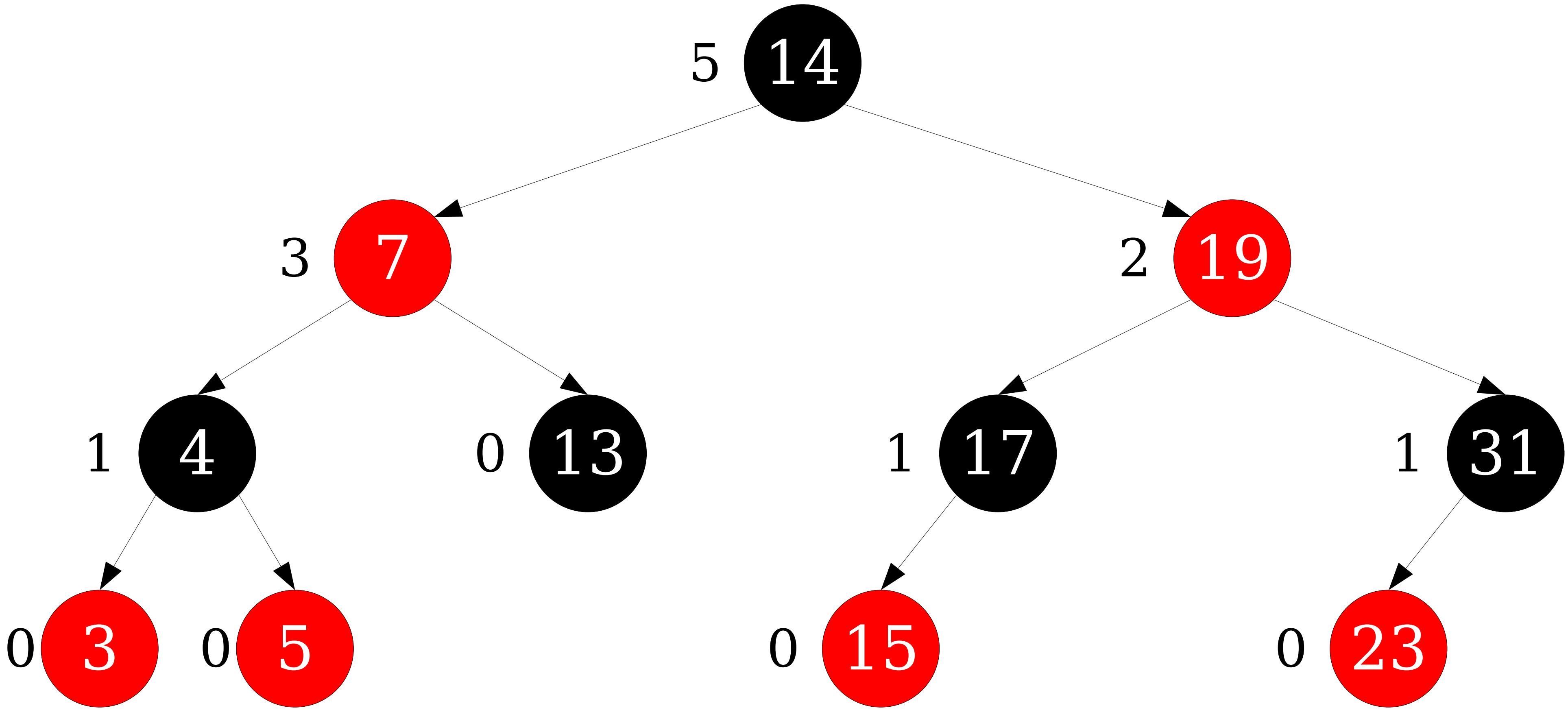
Dynamic Selection



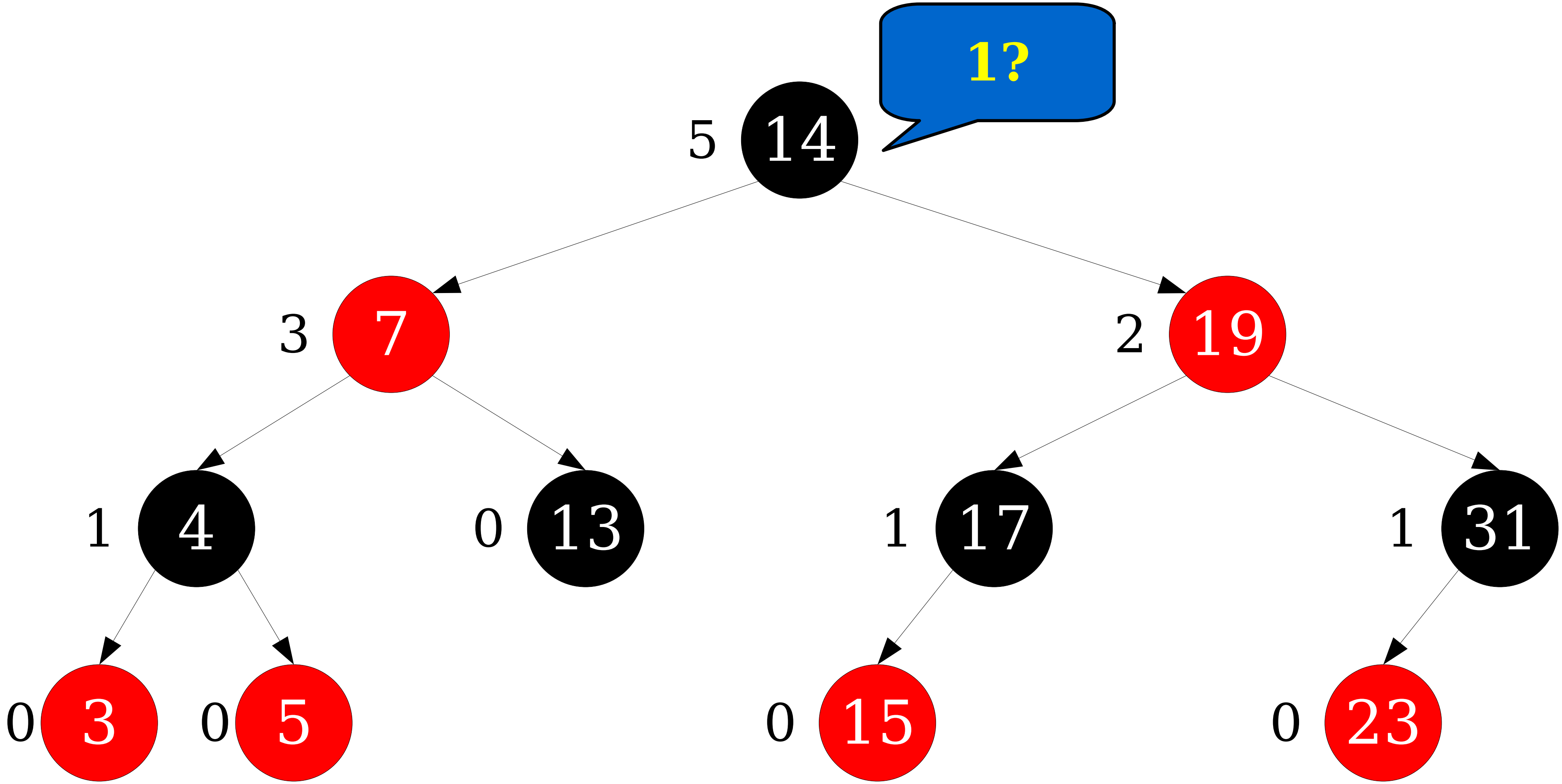
Mechanically: Number each key so that it only stores its order statistic in the subtree rooted at itself.

Operationally: Annotate each key with the number of keys in its left subtree.

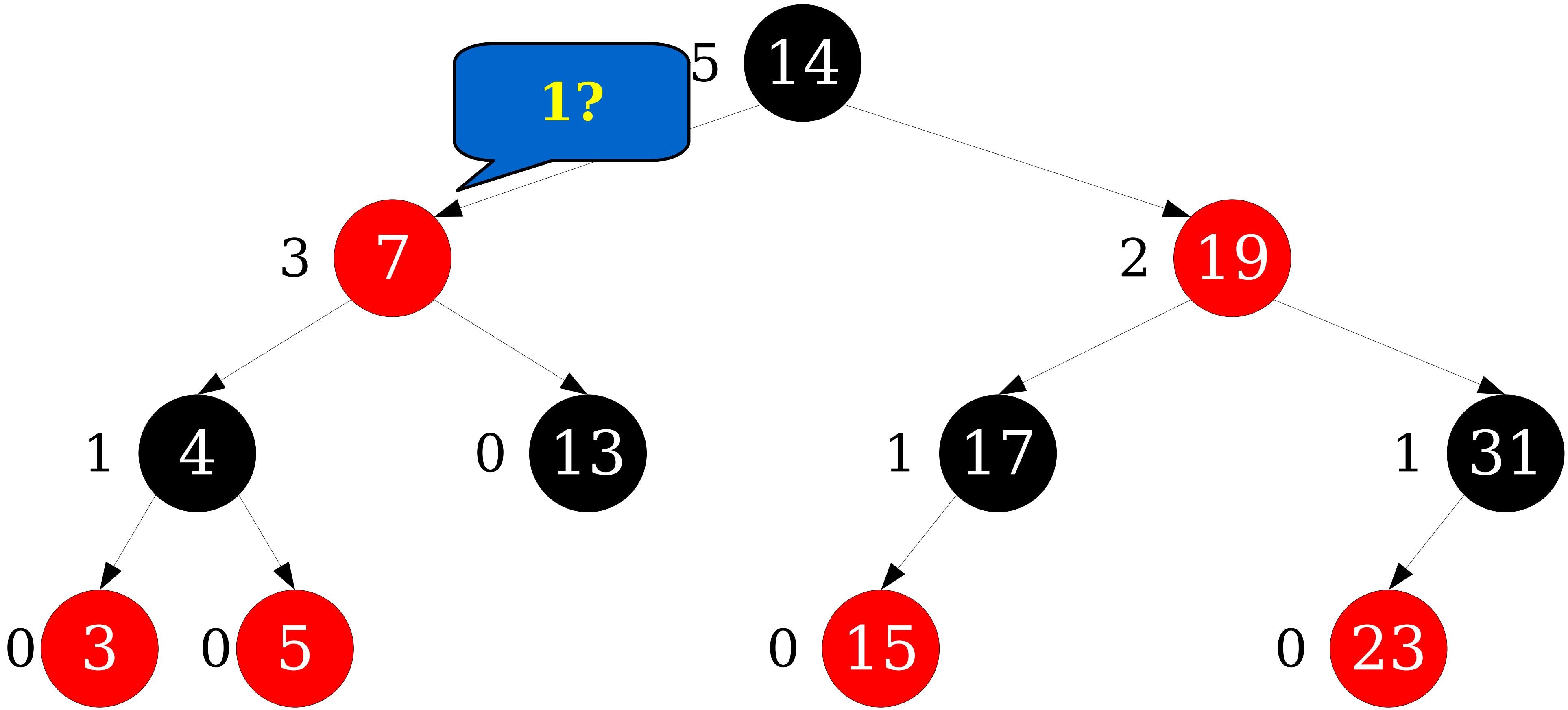
Dynamic Selection



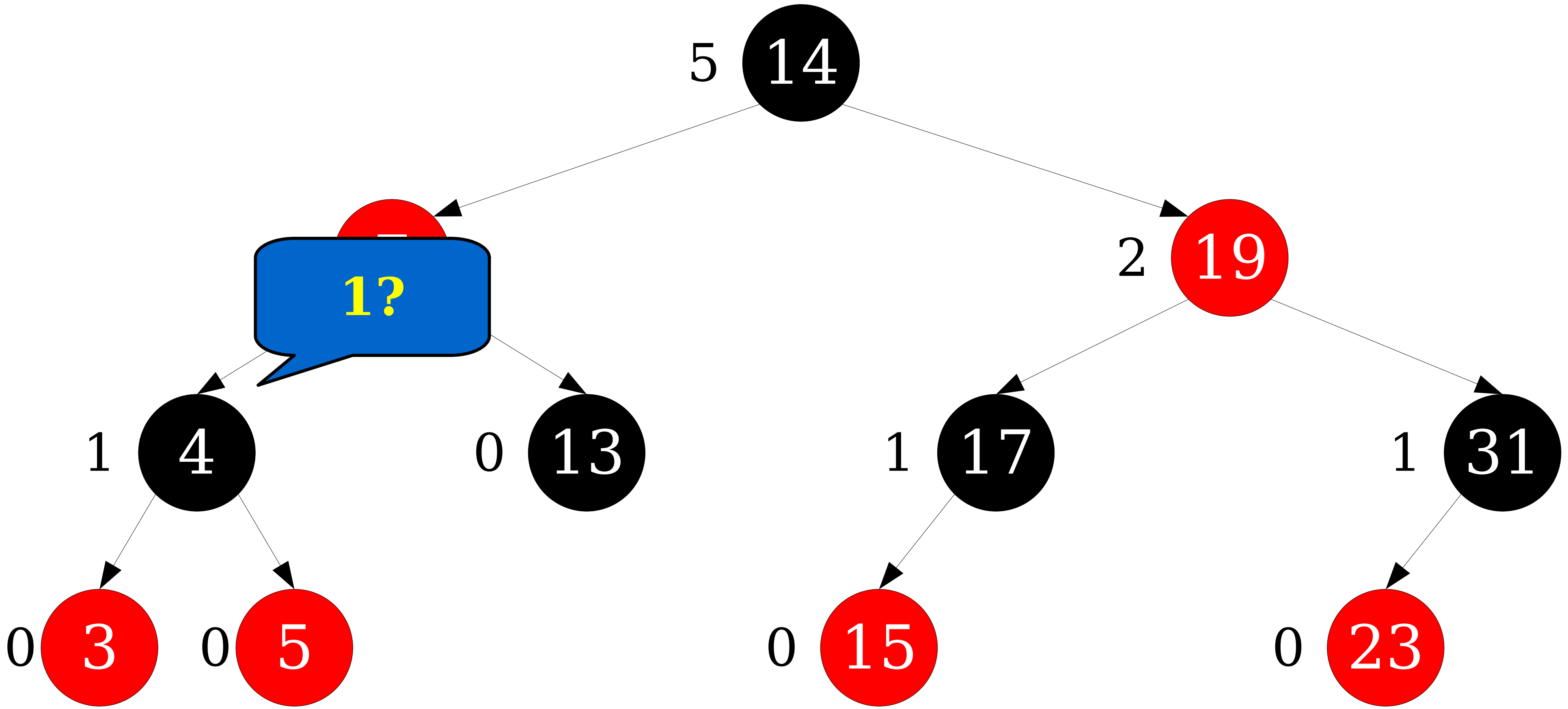
Dynamic Selection



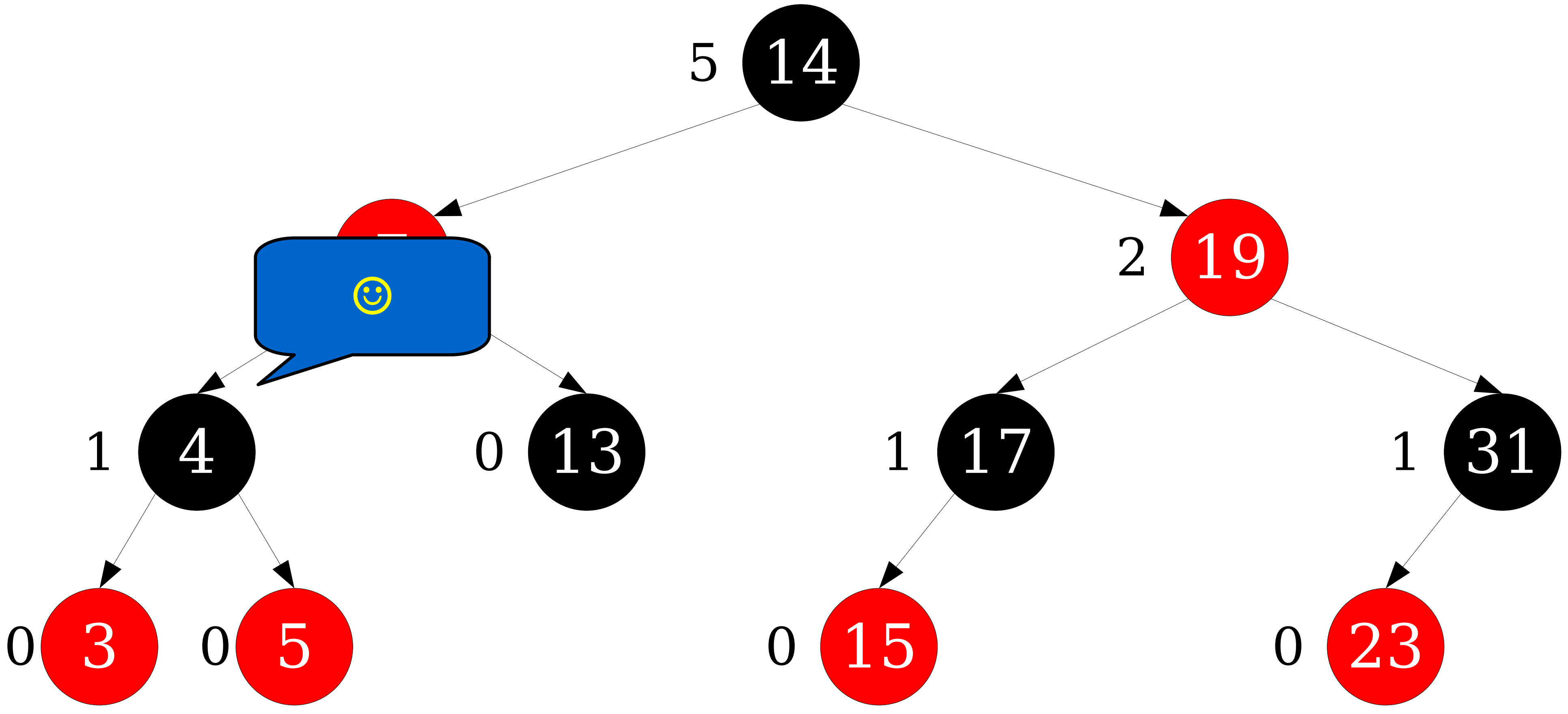
Dynamic Selection



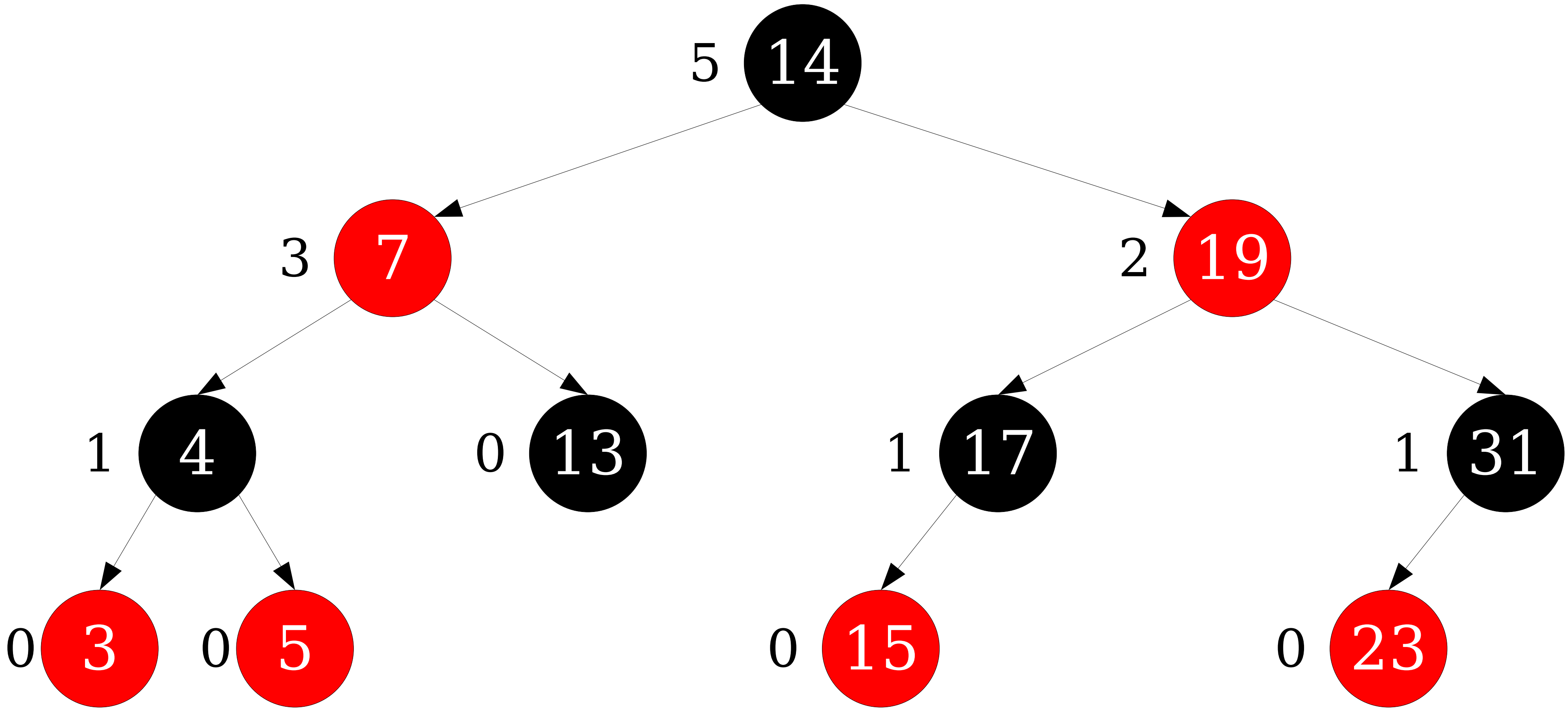
Dynamic Selection



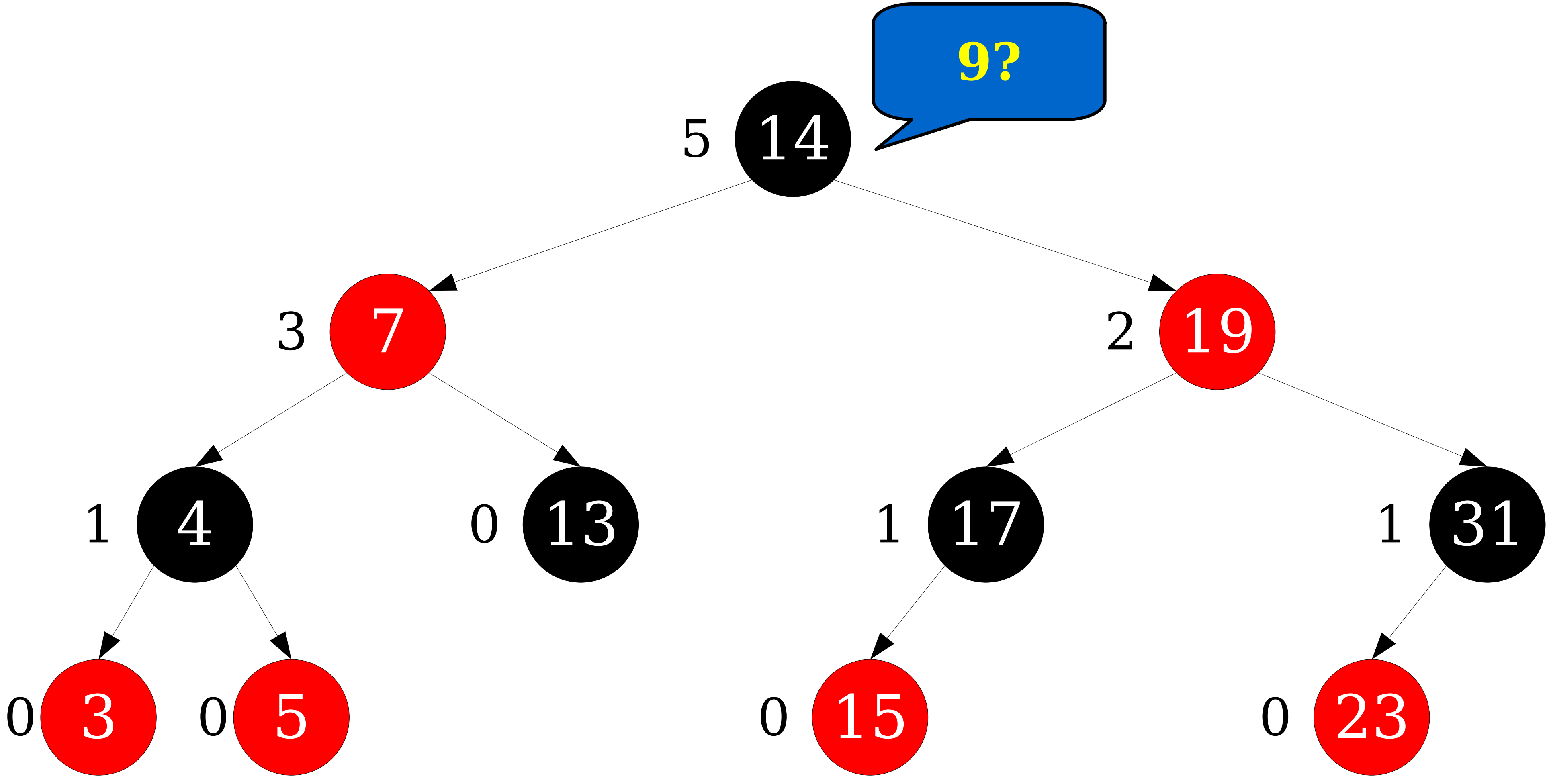
Dynamic Selection



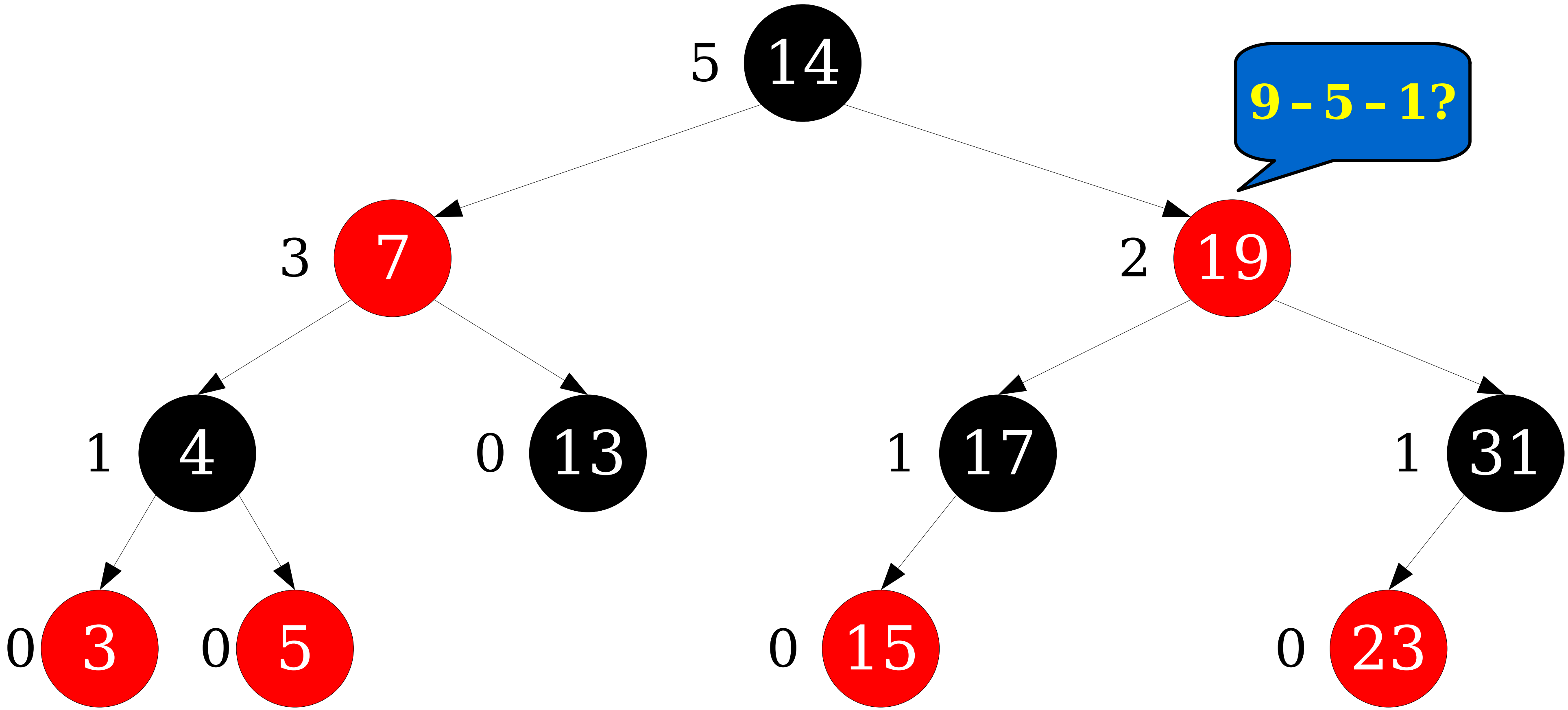
Dynamic Selection



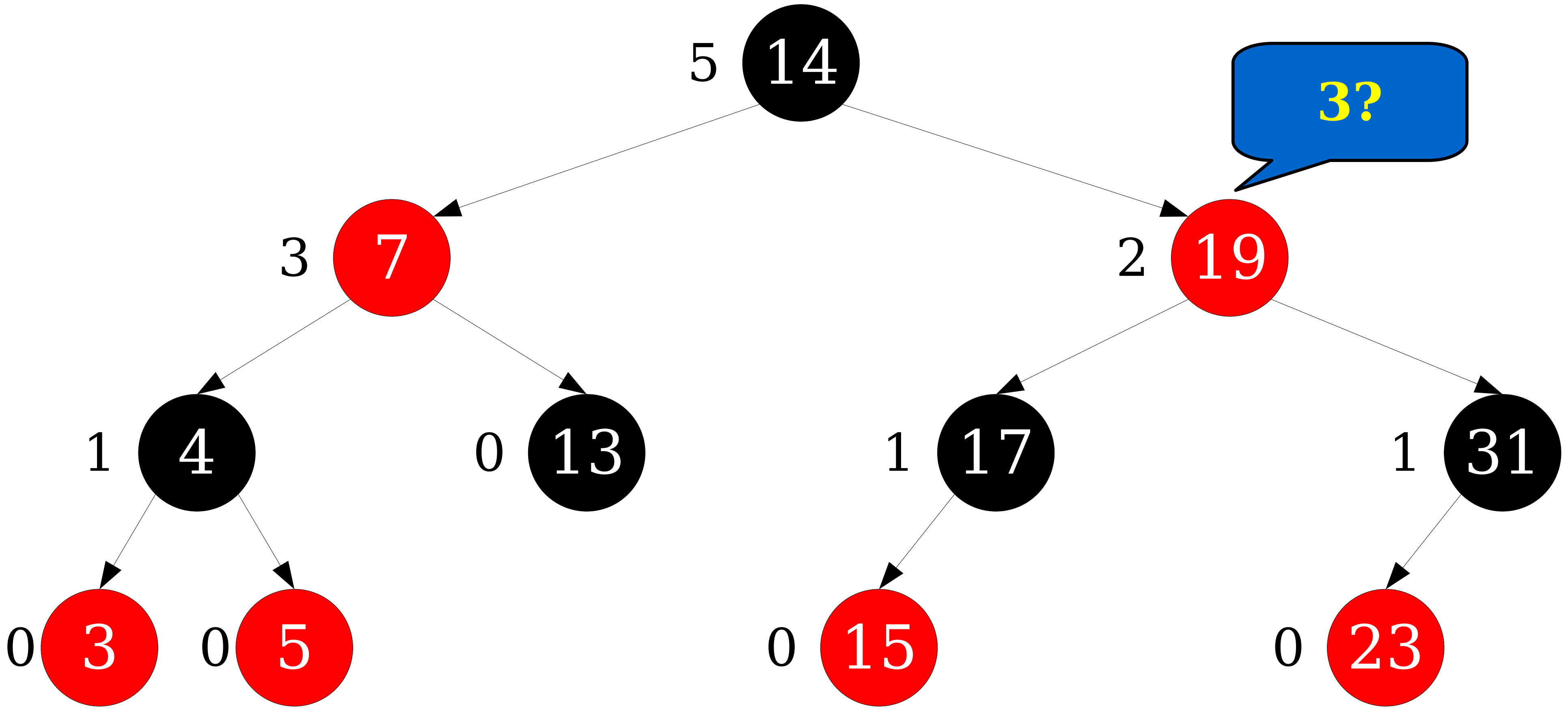
Dynamic Selection



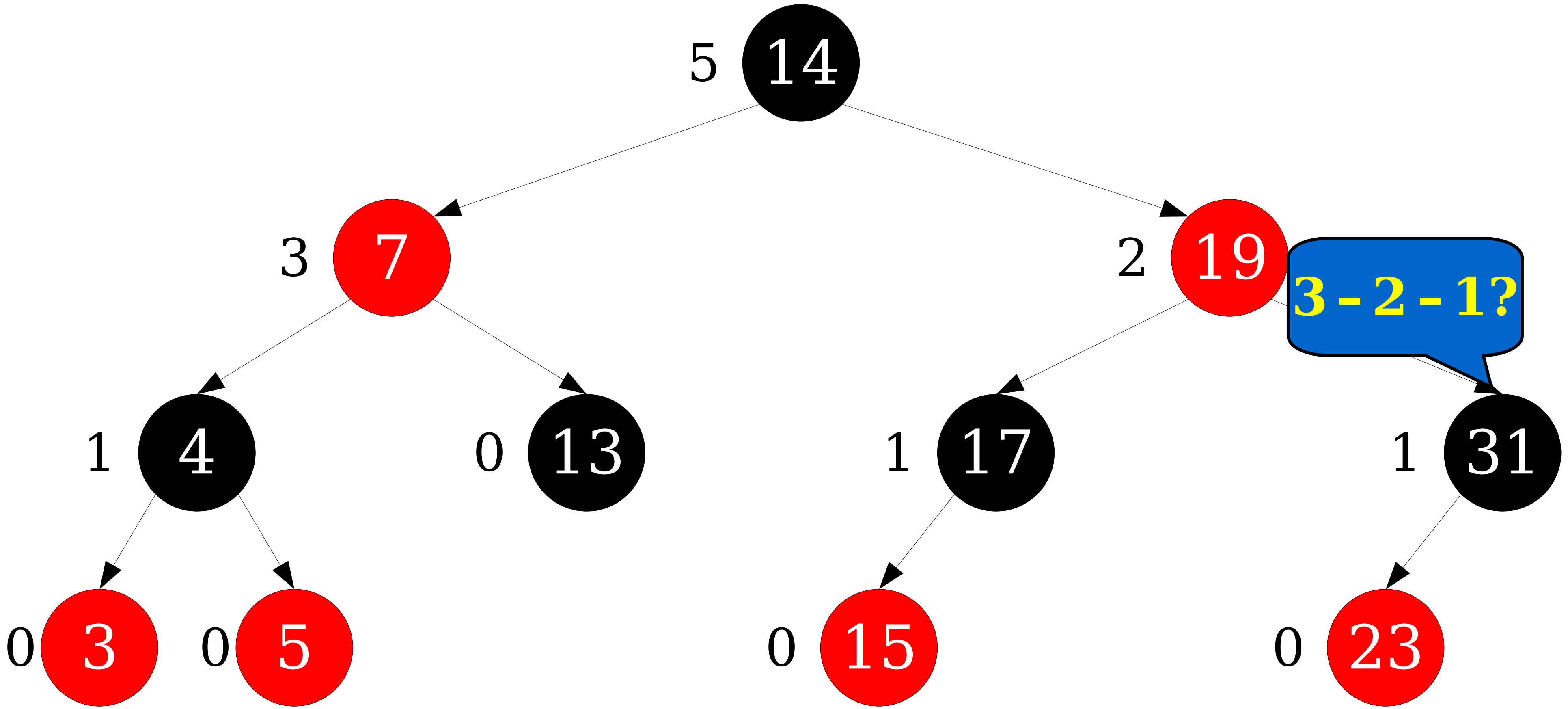
Dynamic Selection



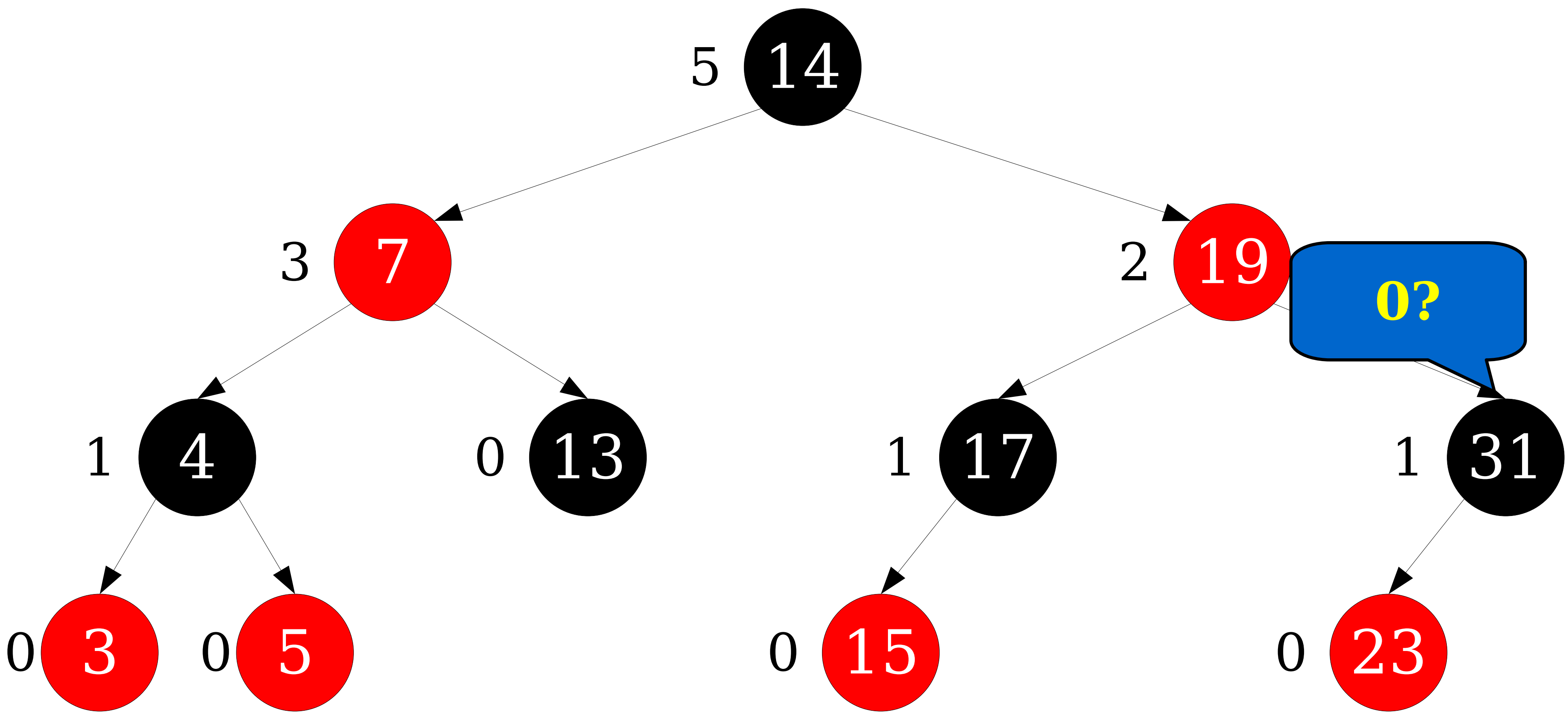
Dynamic Selection



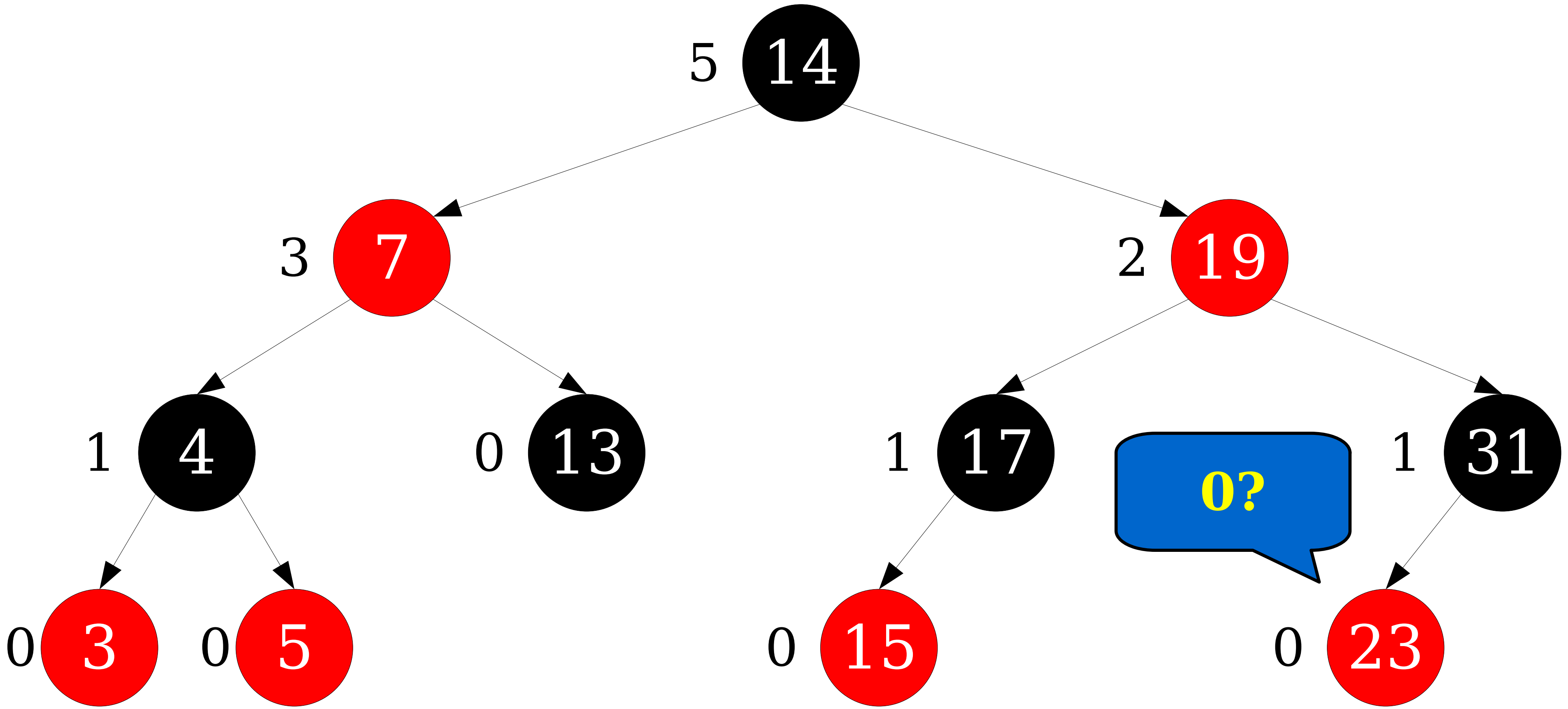
Dynamic Selection



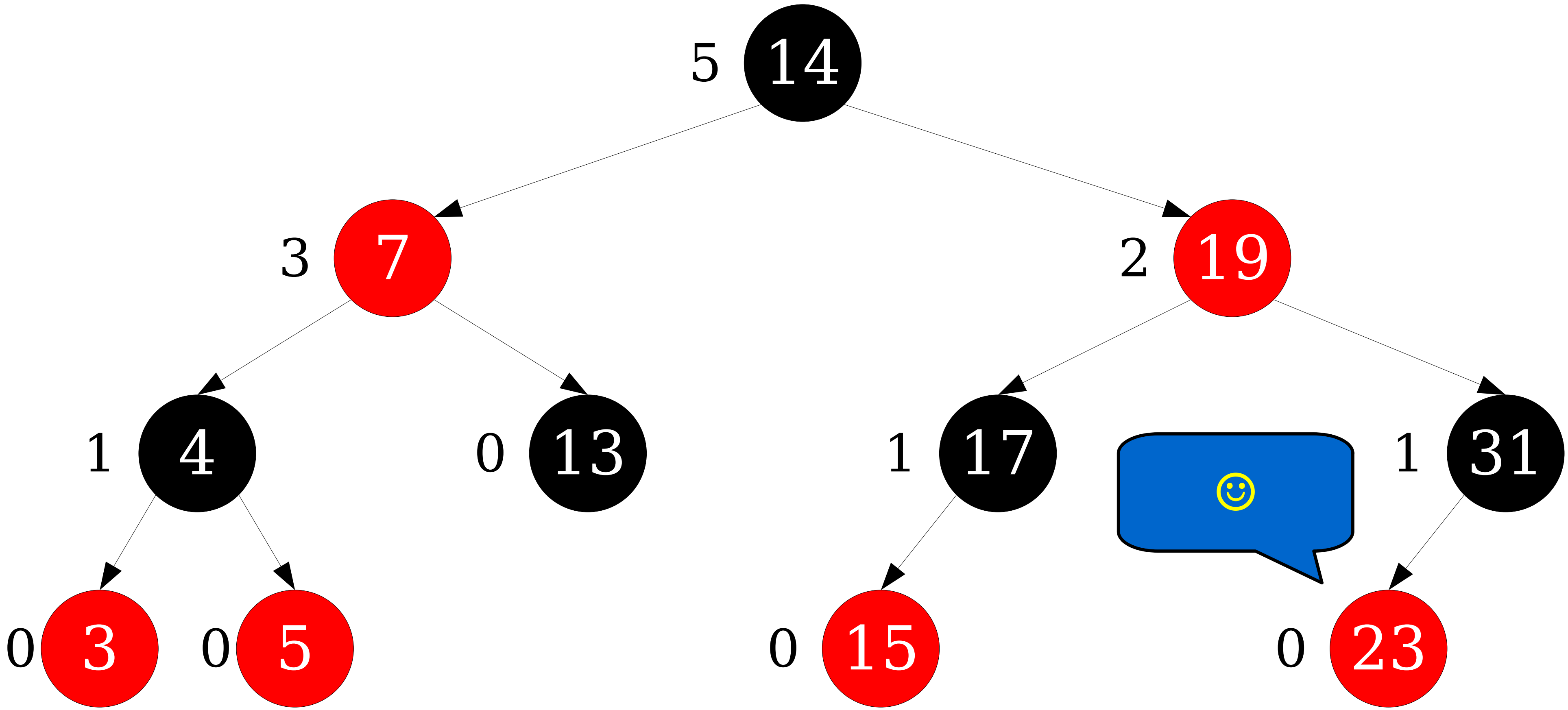
Dynamic Selection



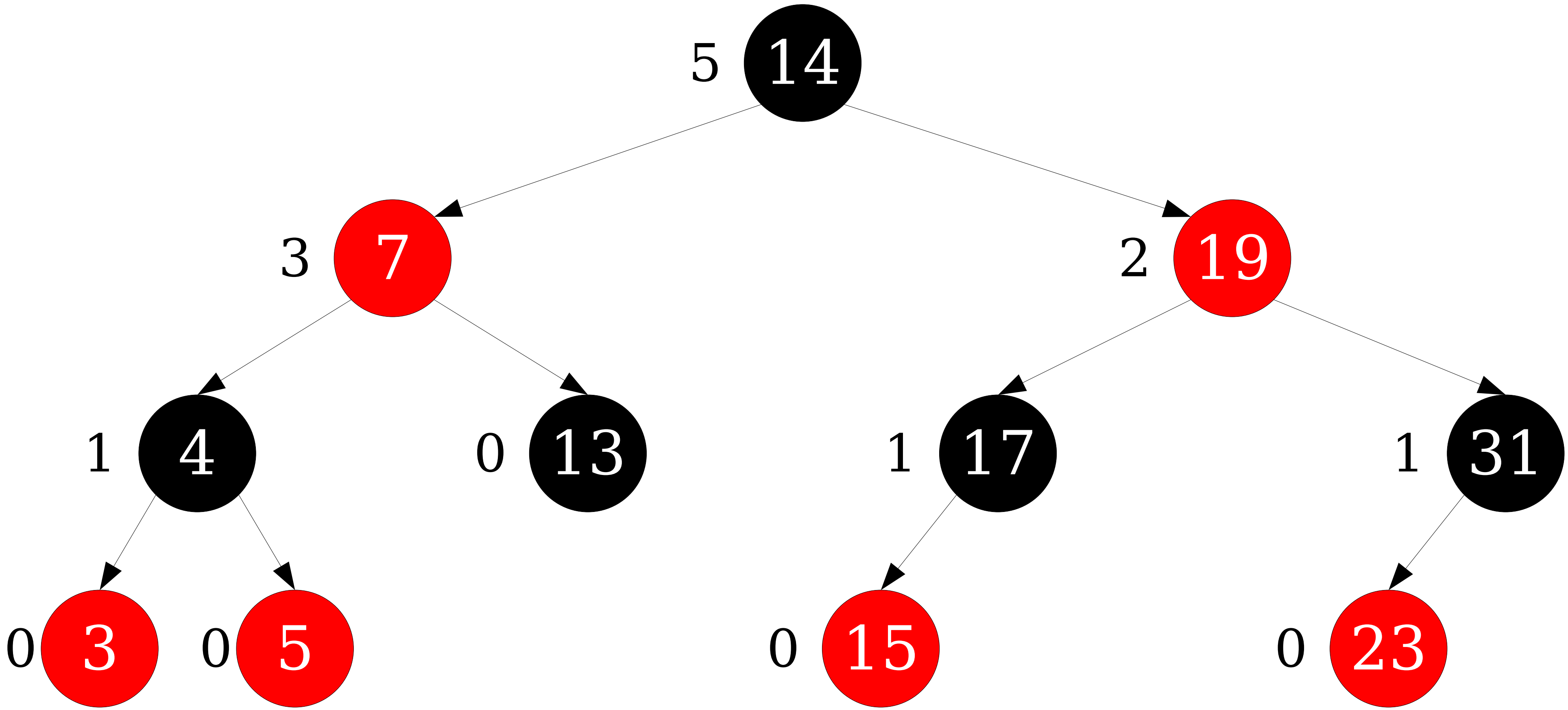
Dynamic Selection



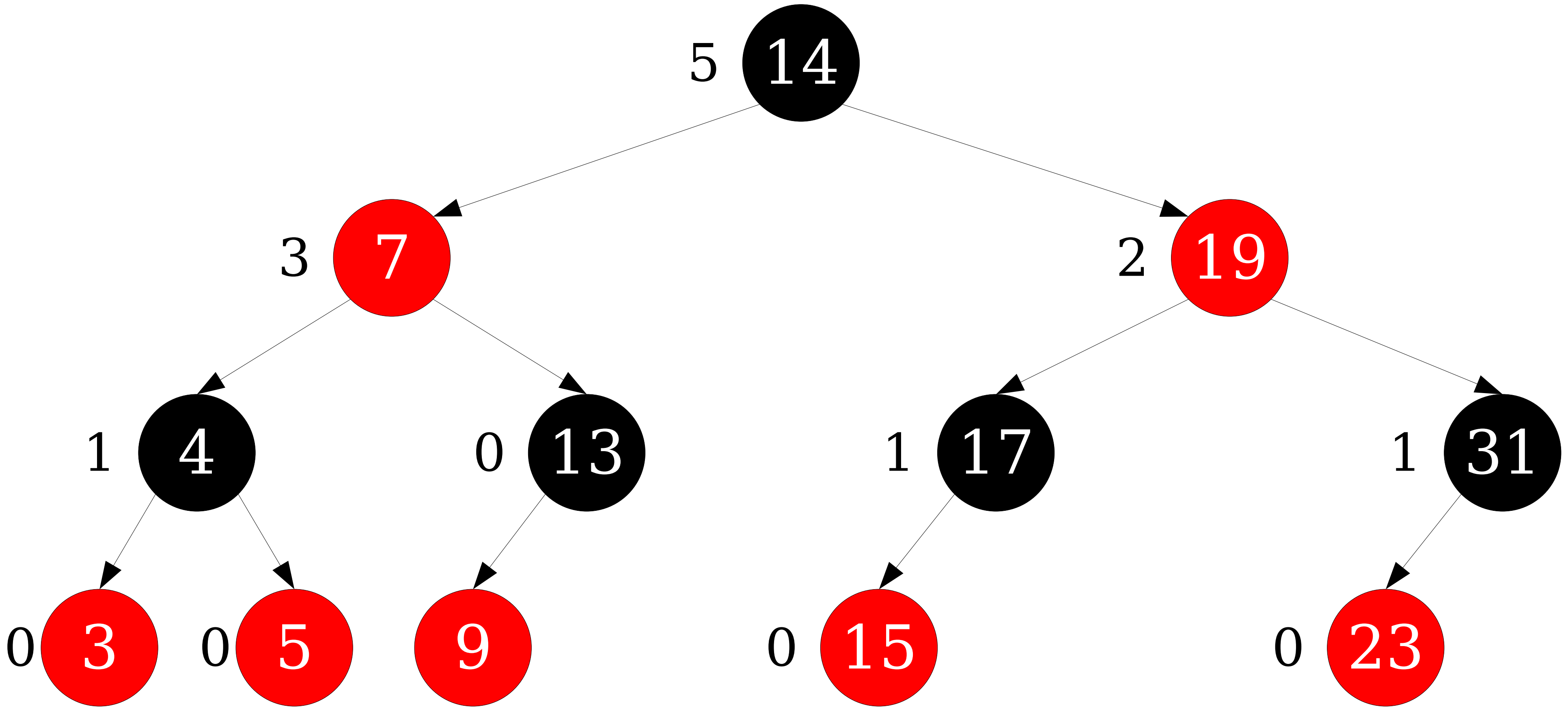
Dynamic Selection



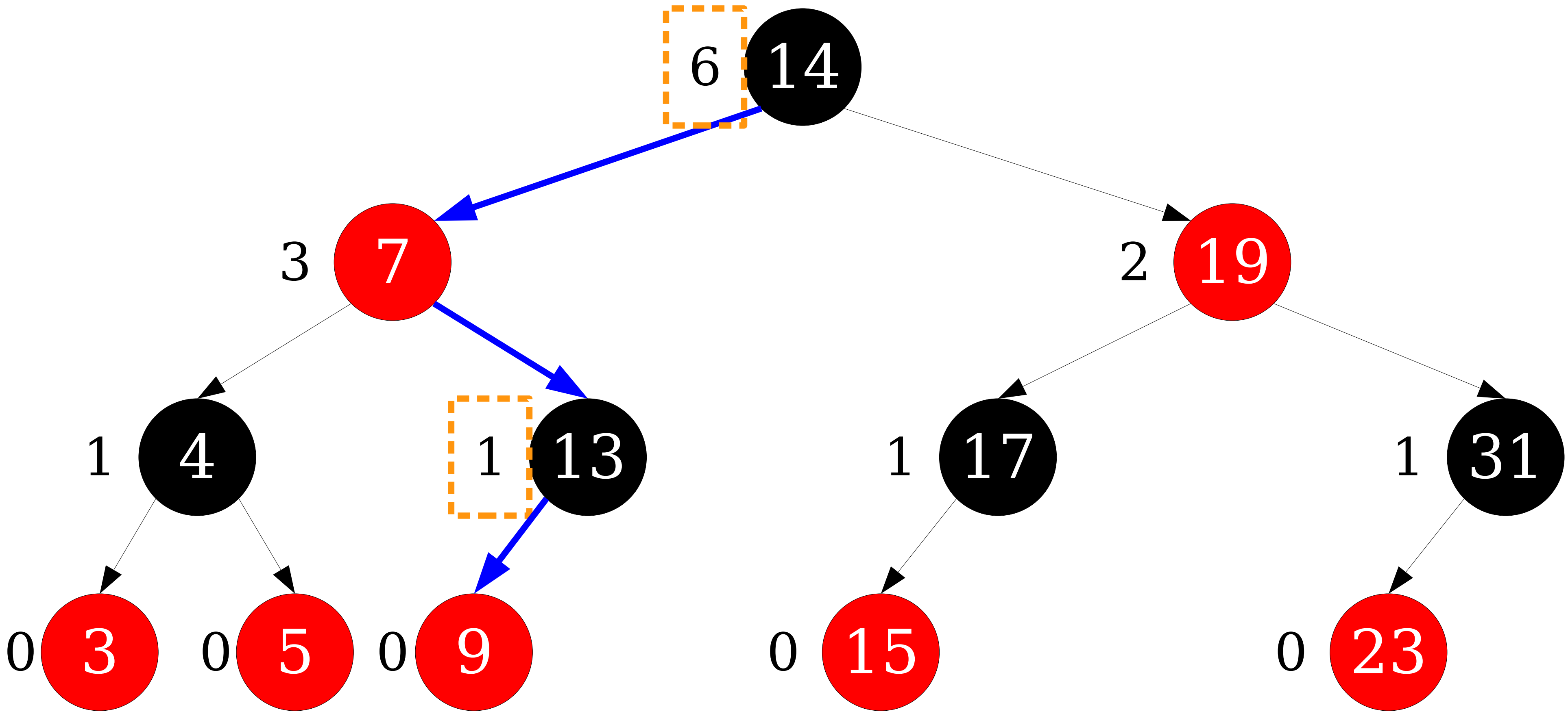
Dynamic Selection



Dynamic Selection

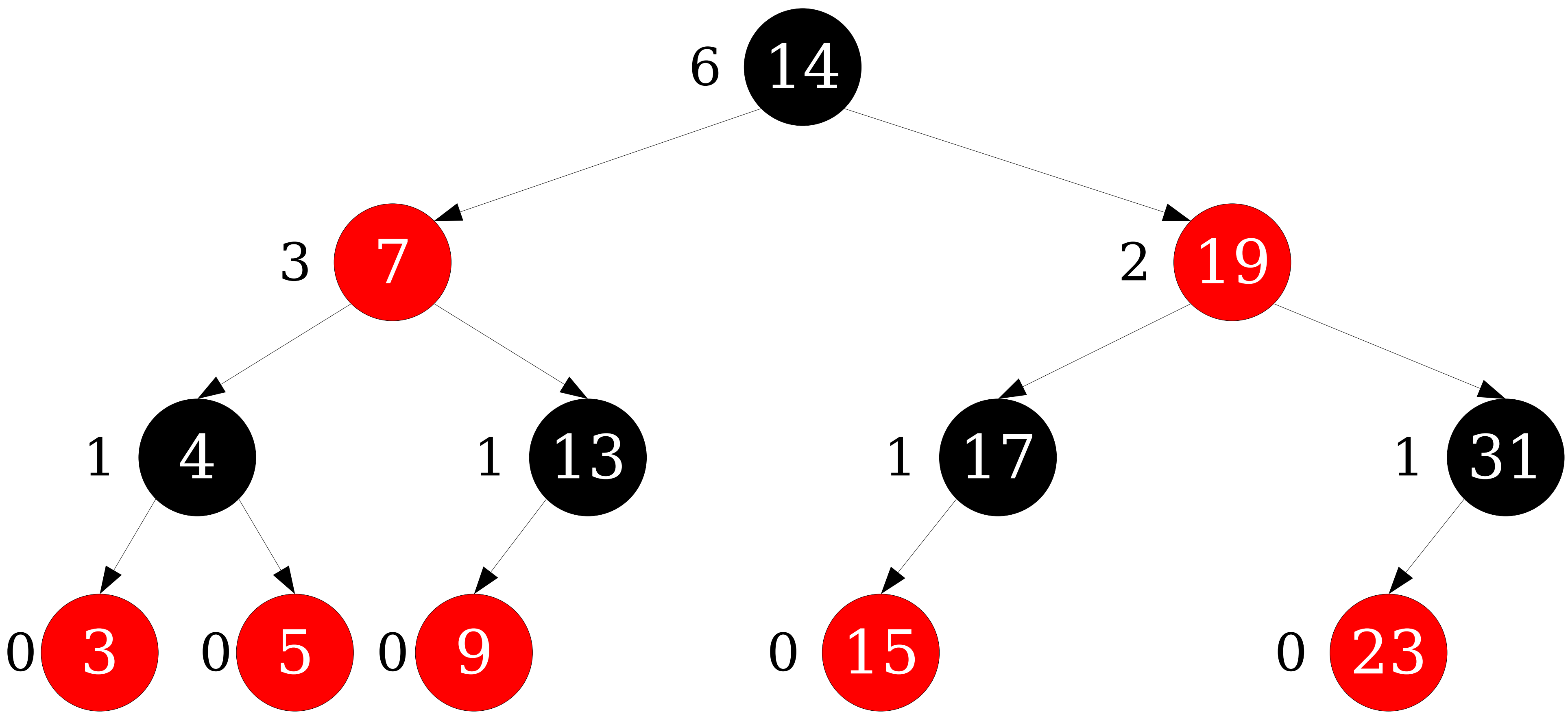


Dynamic Selection

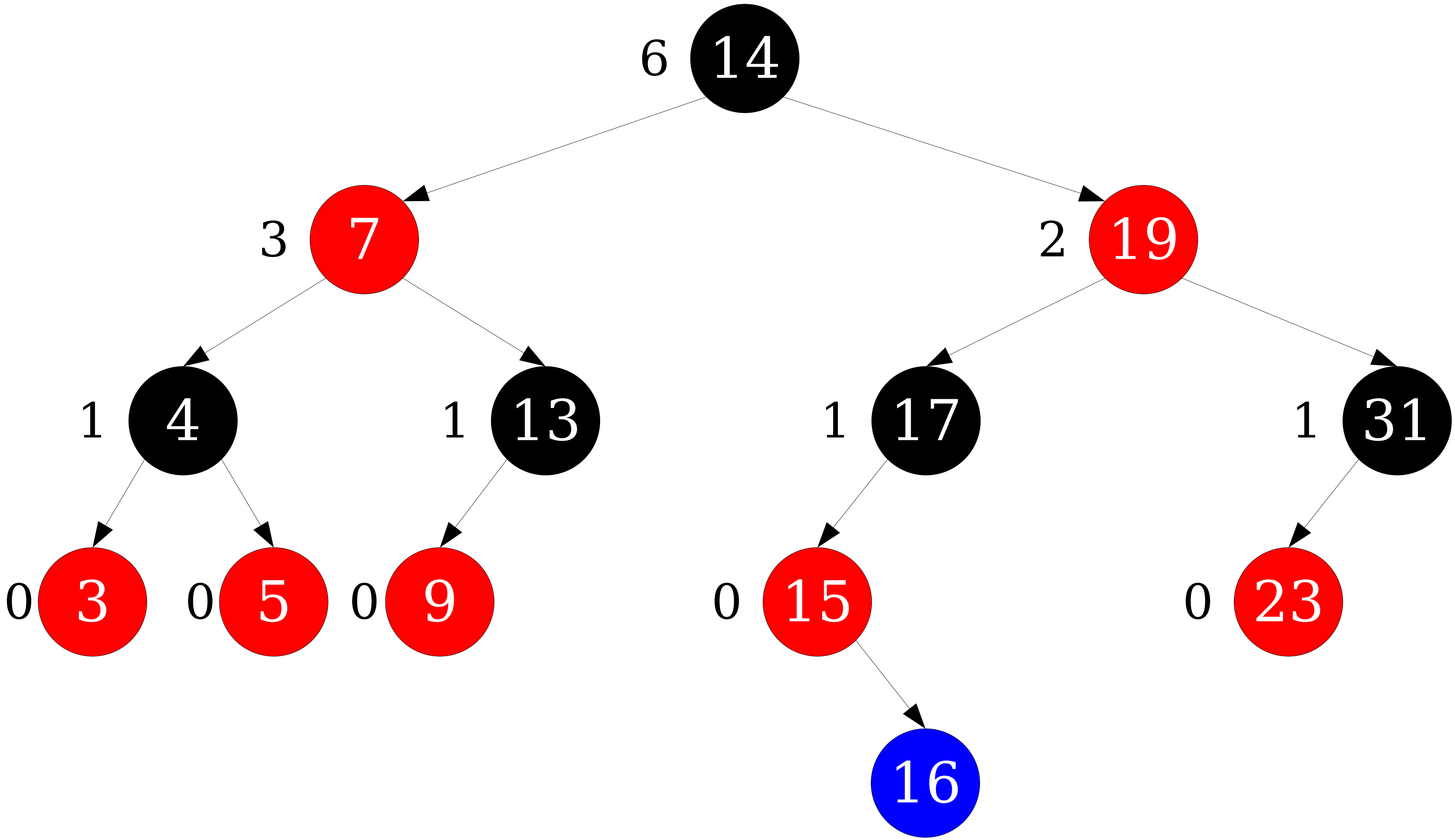


We only update values on nodes that gained a new key in their left subtree. And there are only O(log *n*) of these!

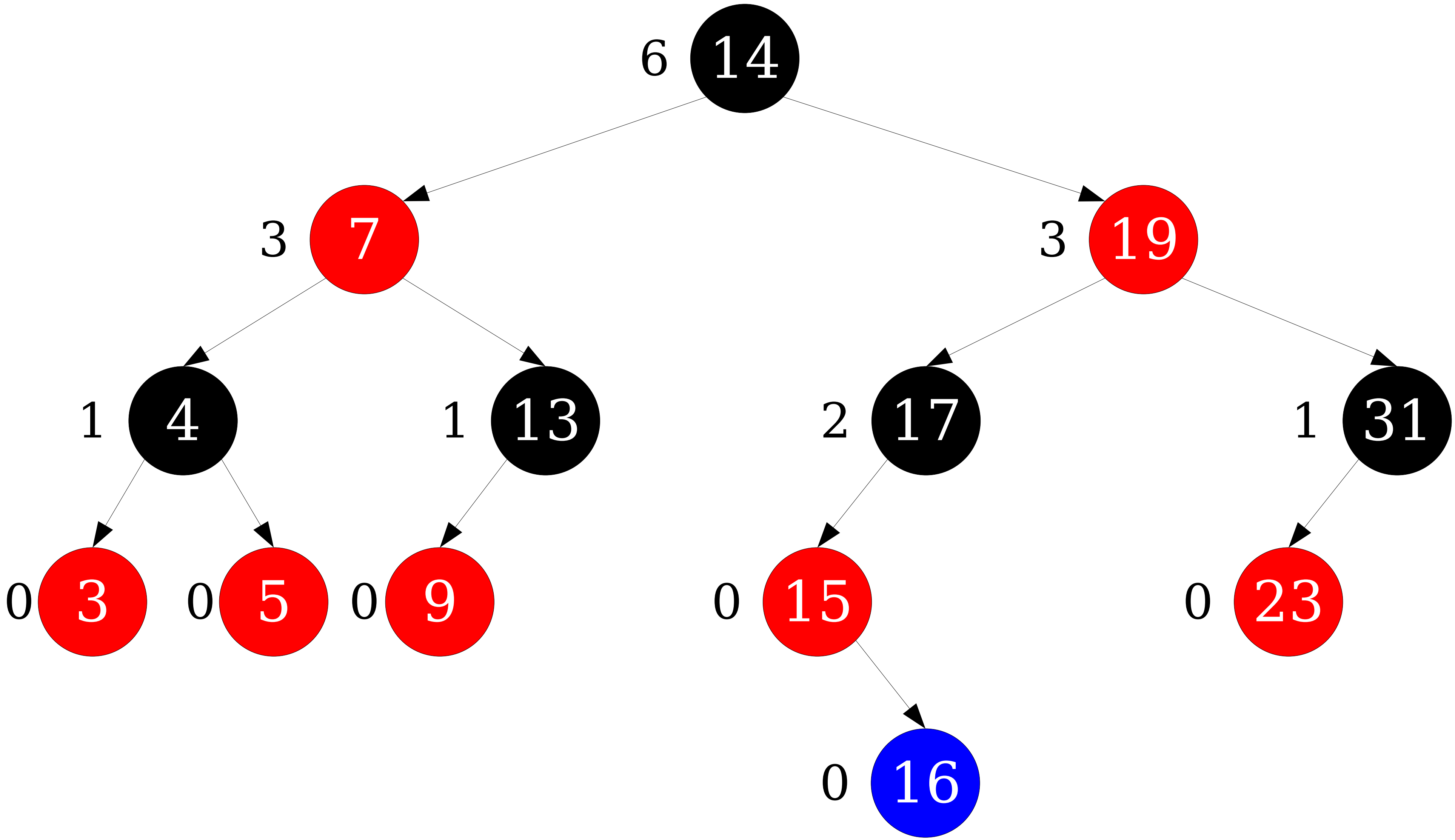
Dynamic Selection



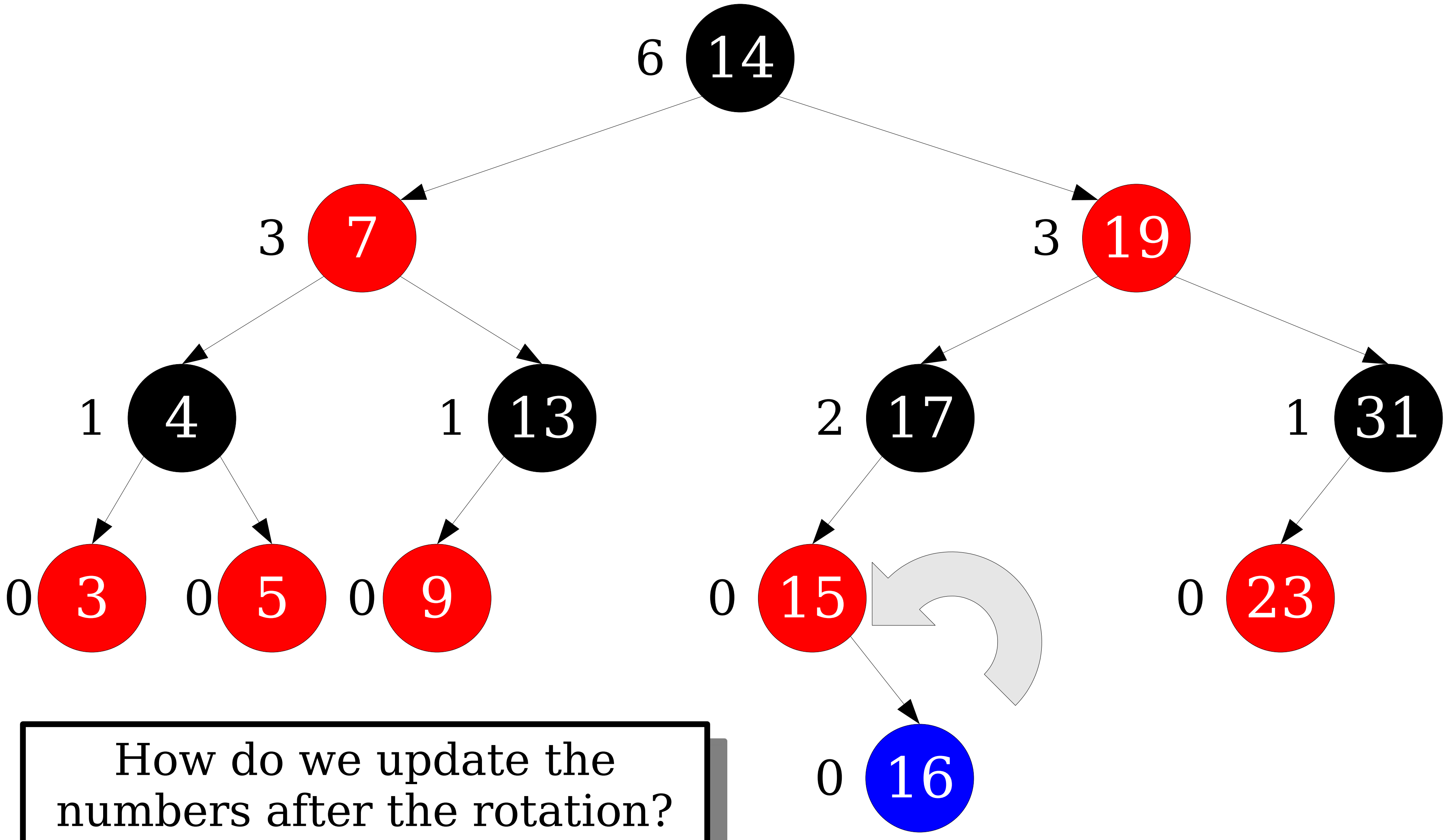
Dynamic Selection



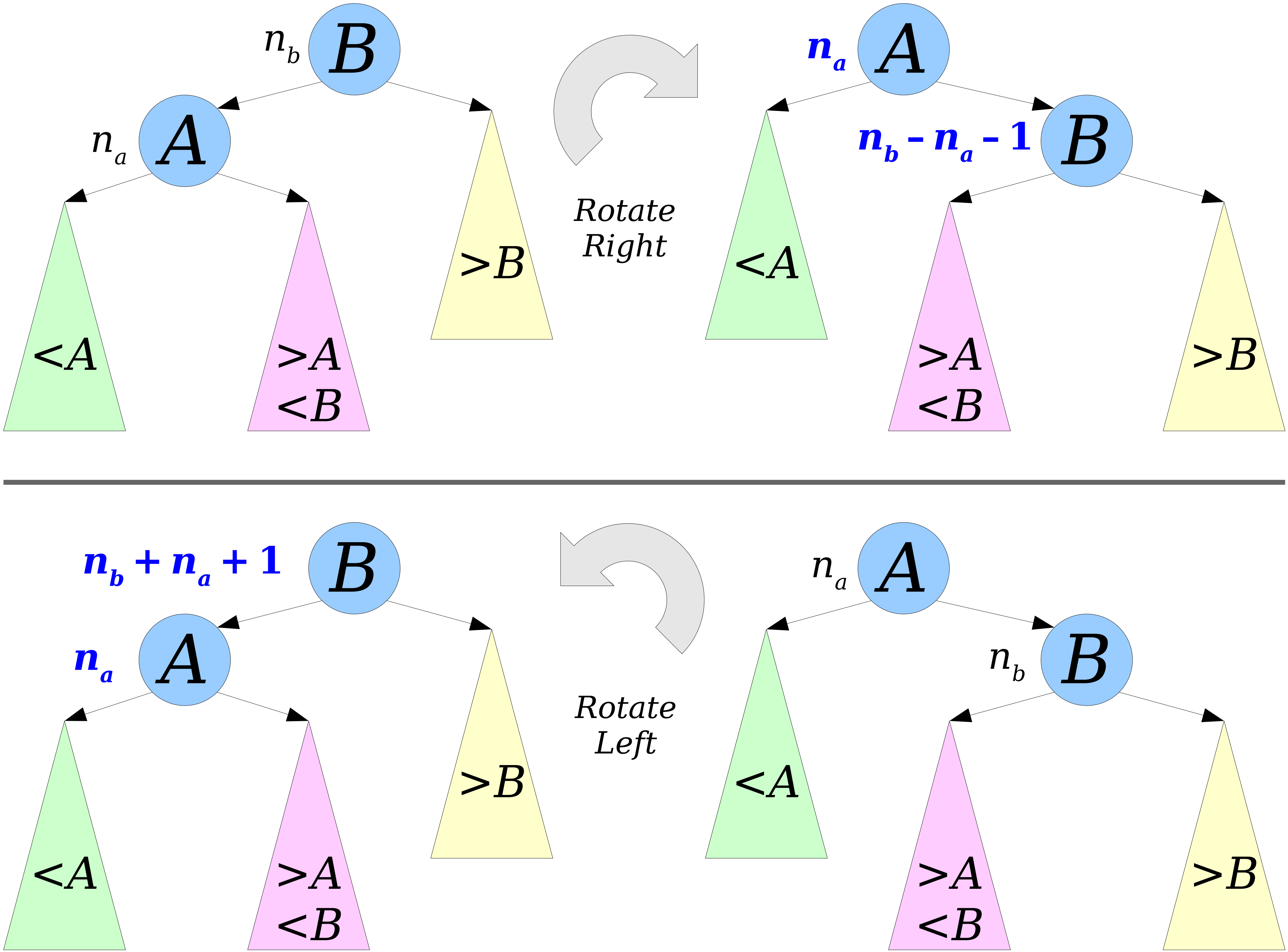
Dynamic Selection



Dynamic Selection



How do we update the numbers after the rotation?

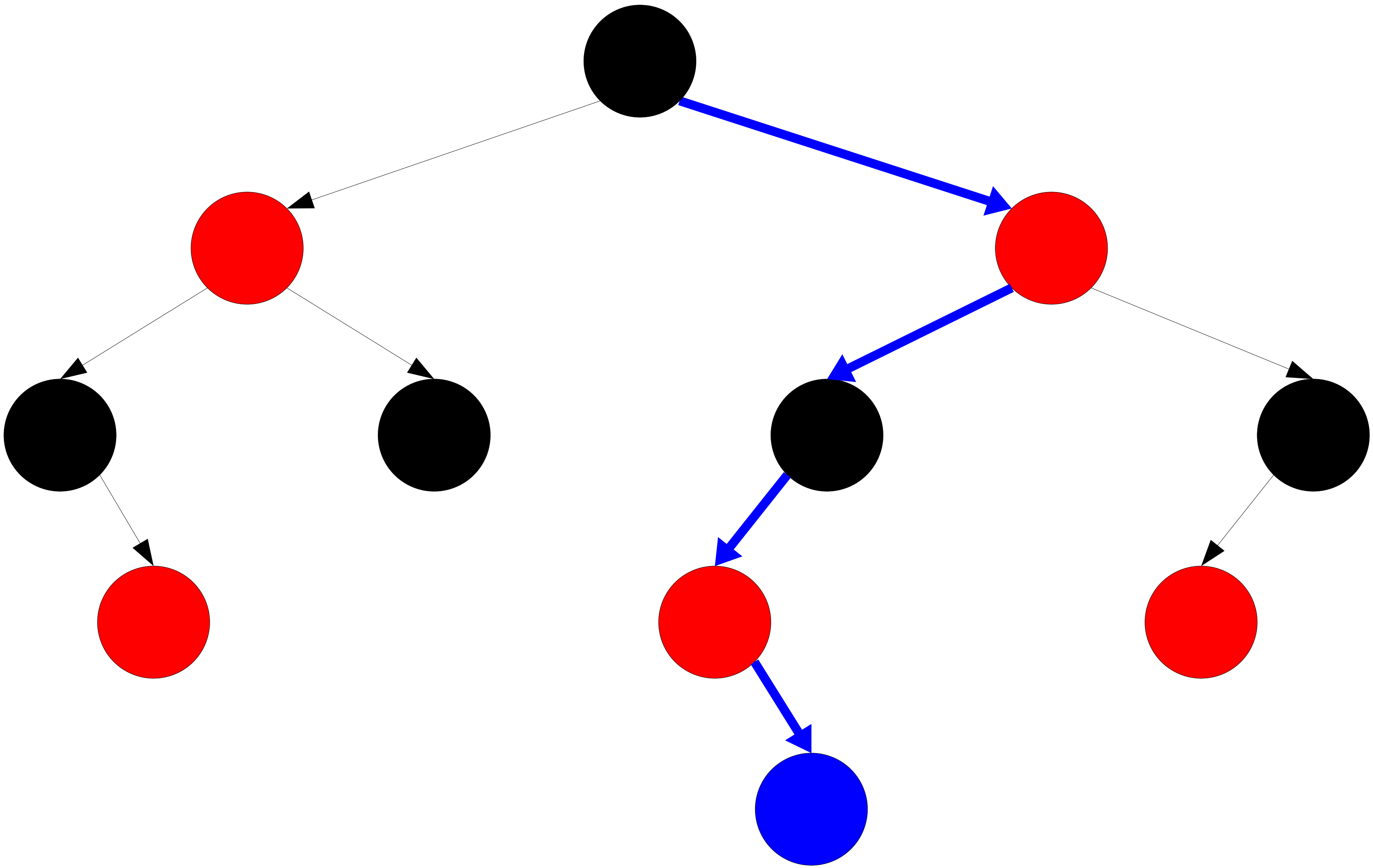


n_b
B
n_a
A
<A
>A
<B
>B
Rotate
Right
n_a
A
n_b - n_a - 1
B
<A
>A
<B
>B
n_b + n_a + 1
B
n_a
A
<A
>A
<B
>B
Rotate
Left
n_a
A
n_b
B
<A
>A
<B
>B

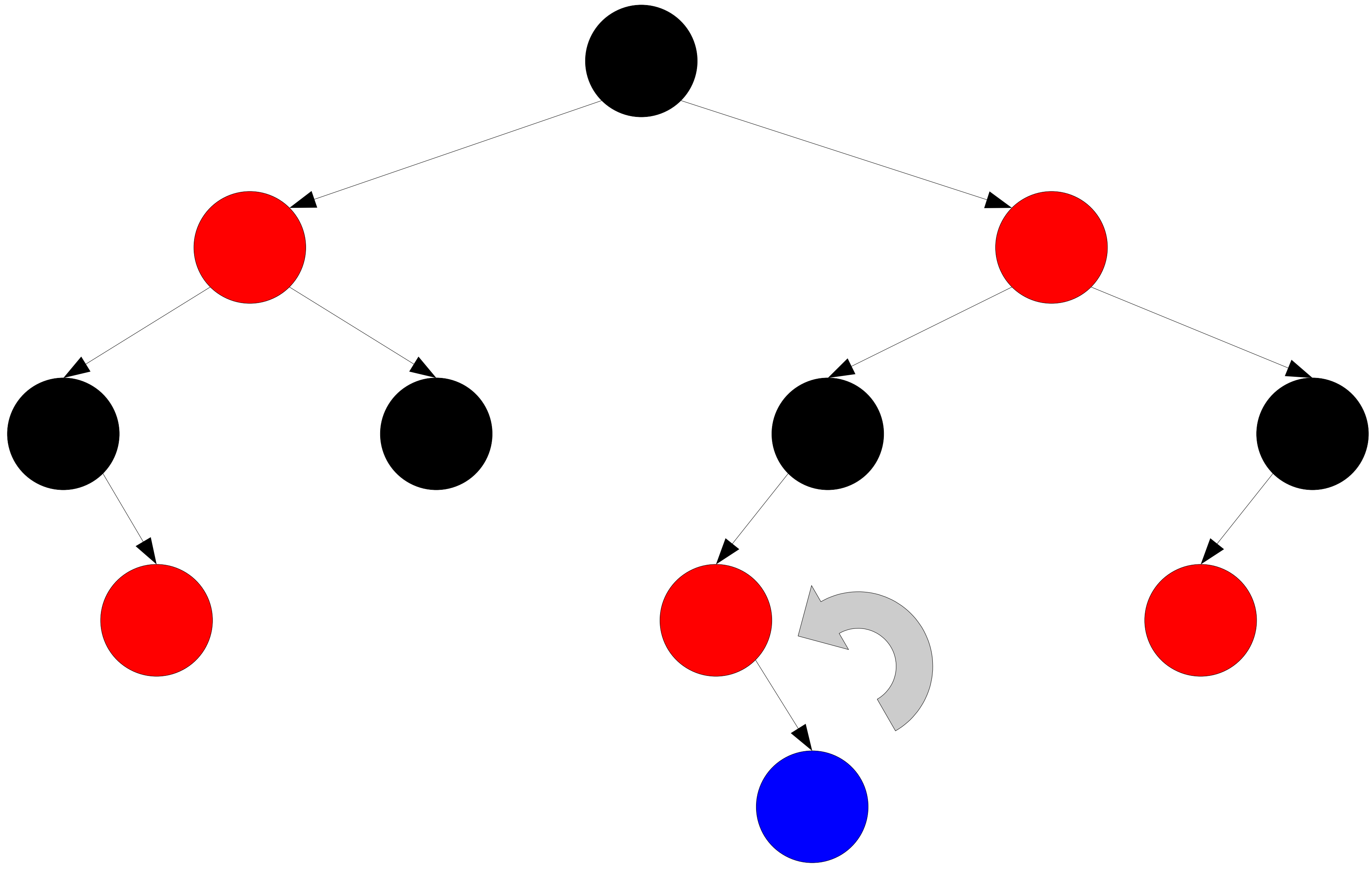
Order Statistic Trees

- This modified red/black tree is called an ***order statistics tree***.
  - Start with a red/black tree.
  - Tag each node with the number of nodes in its left subtree.
  - Use the preceding update rules to preserve values during rotations.
  - Propagate other changes up to the root of the tree.
- Only O(log n) values must be updated on an insertion or deletion and each can be updated in time O(1).
- Supports all BST operations plus ***select*** (find kth order statistic) and ***rank*** (given a key, report its order statistic) in time O(log n).

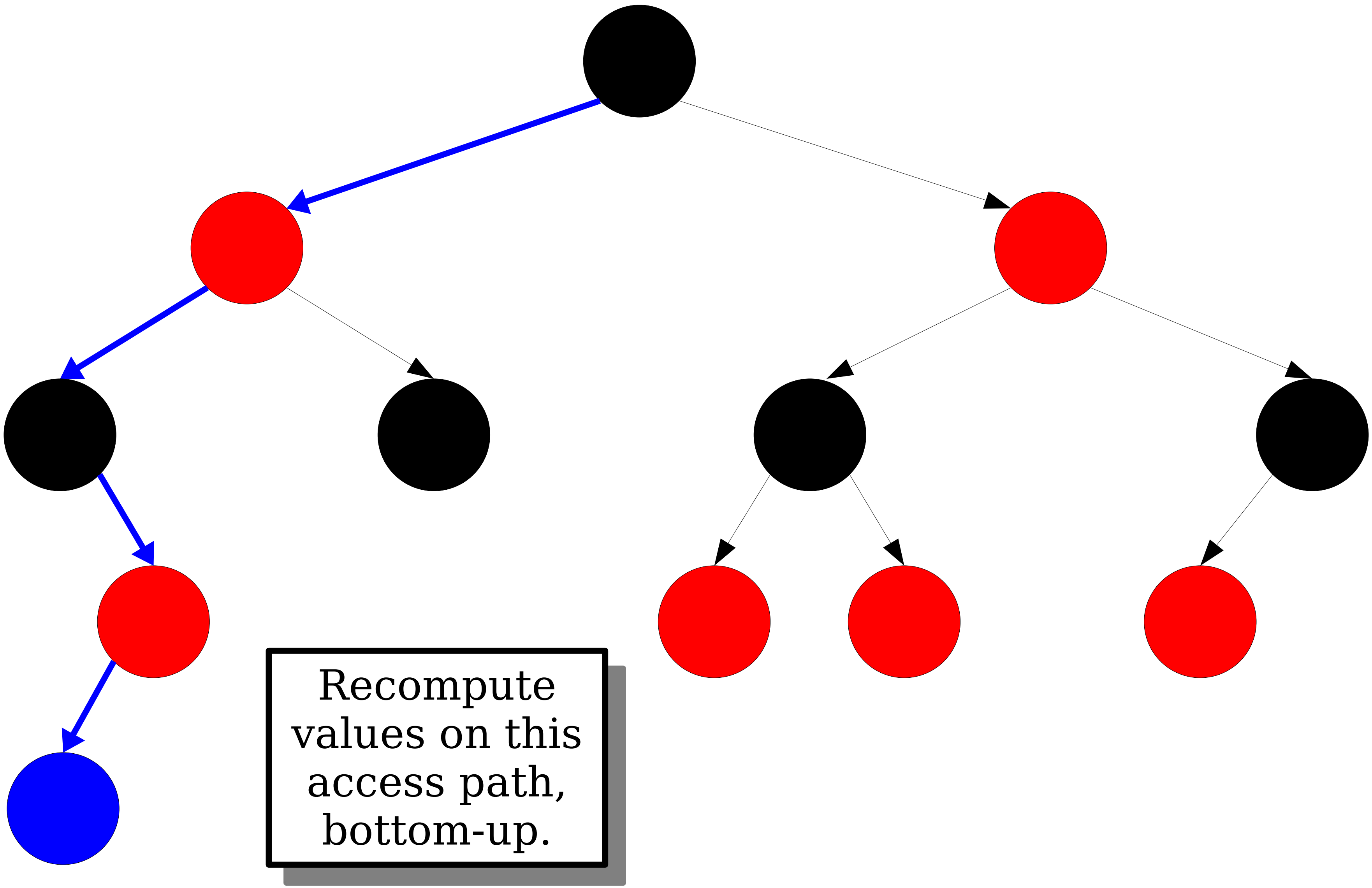
Generalizing our Idea



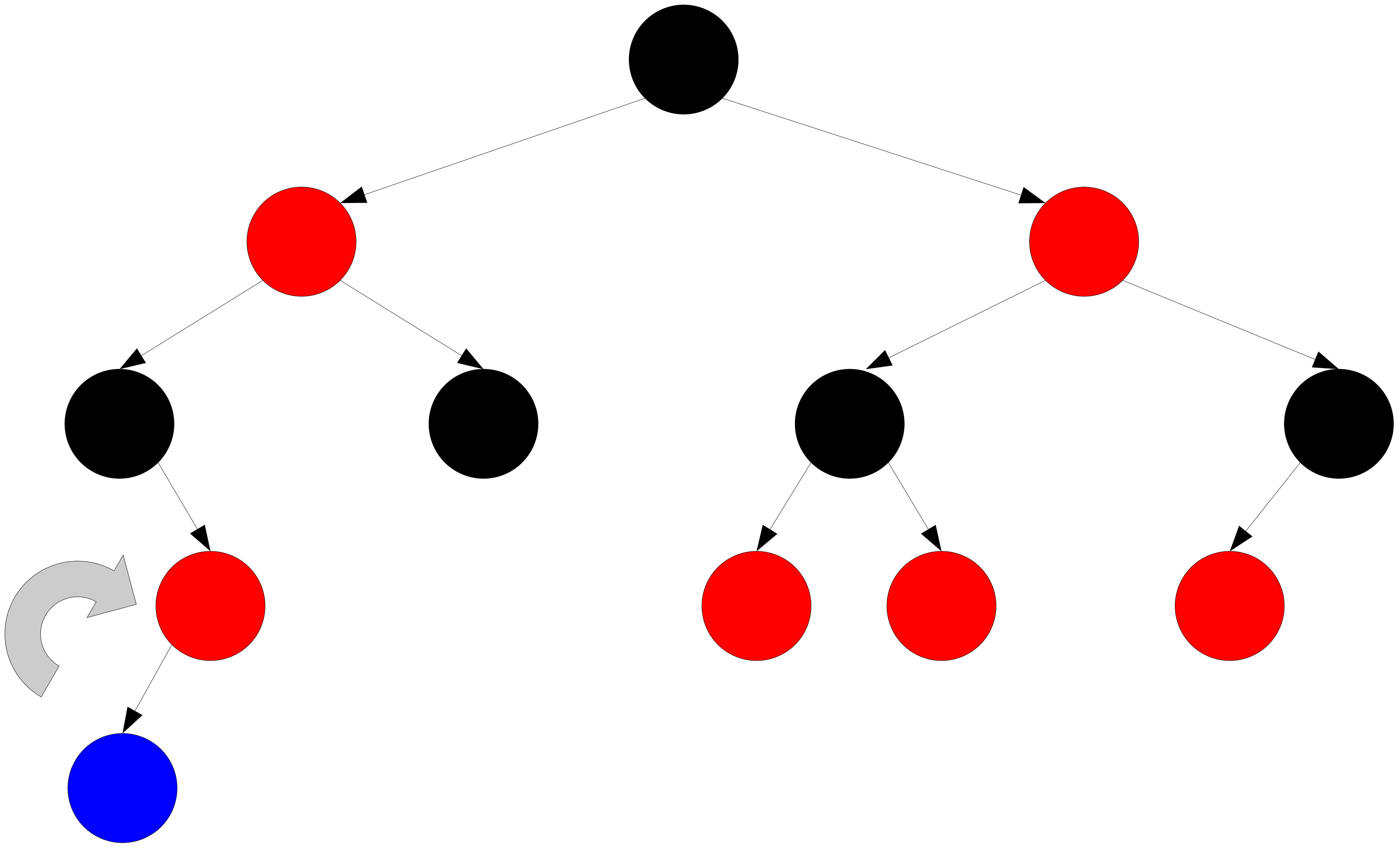
Edits to values are localized along the access path.



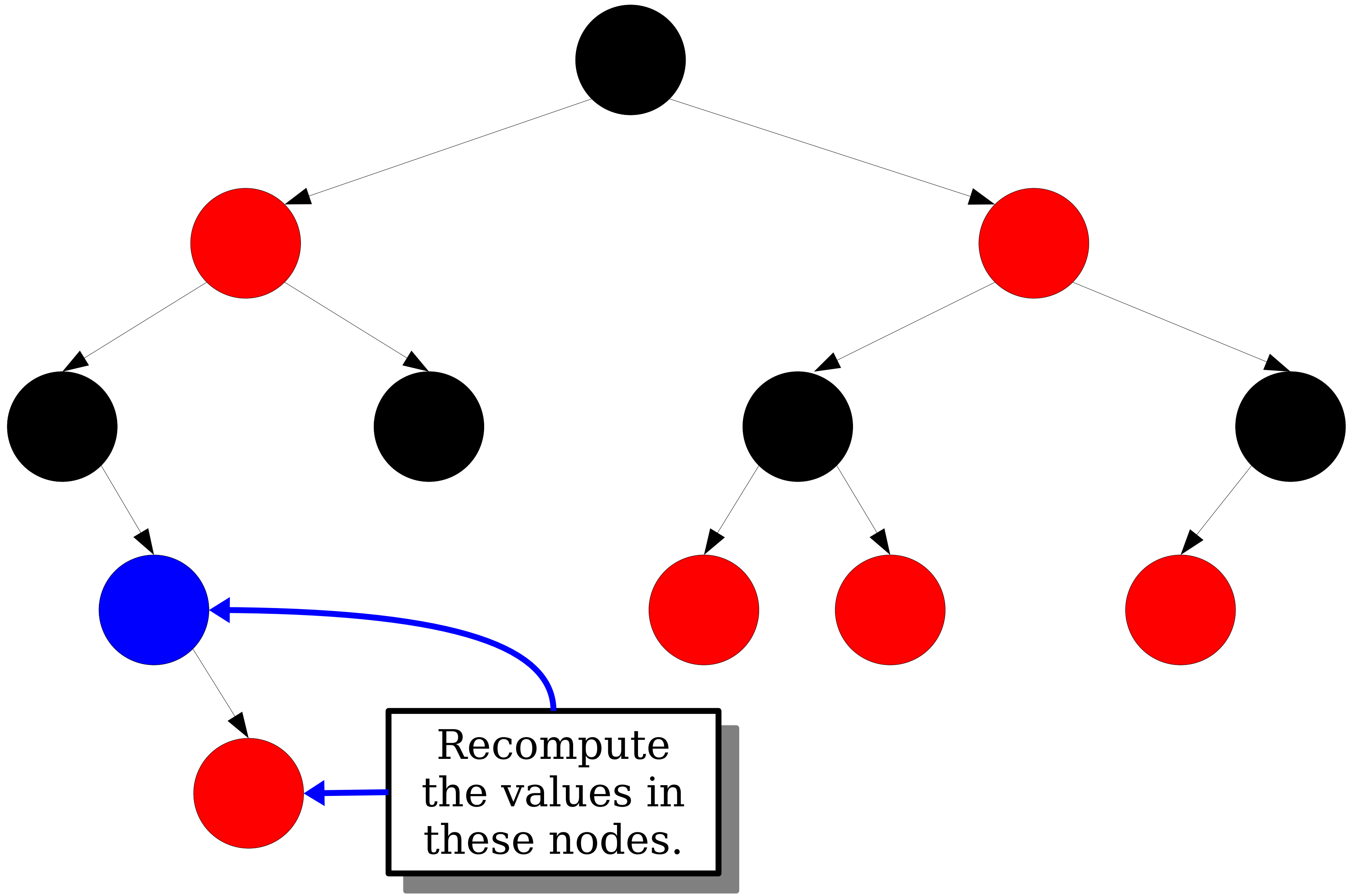
Edits to values are localized along the access path.
We can recompute values after a rotation.



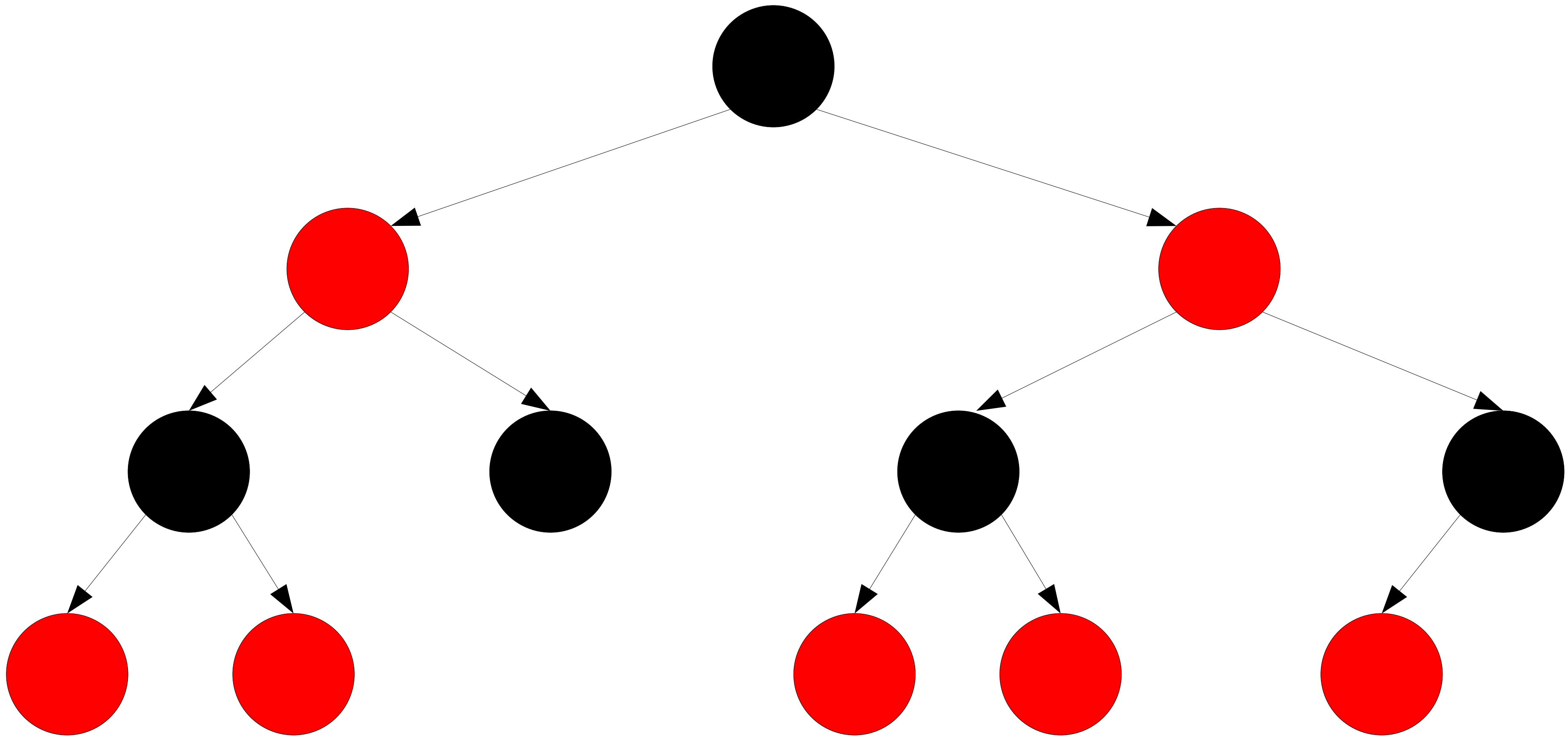
Imagine we cache some value in each node that can be computed just from (1) the node itself and (2) its children's values.



Imagine we cache some value in each node that can be computed just from (1) the node itself and (2) its children's values.



Imagine we cache some value in each node that can be computed just from (1) the node itself and (2) its children's values.



Theorem: Suppose we want to cache some computed value in each node of a red/black tree. Provided that the value can be recomputed purely from the node's value and from it's children's values, and provided that each value can be computed in time O(1), then these values can be cached in each node with insertions, lookups, and deletions still taking time O(log n).

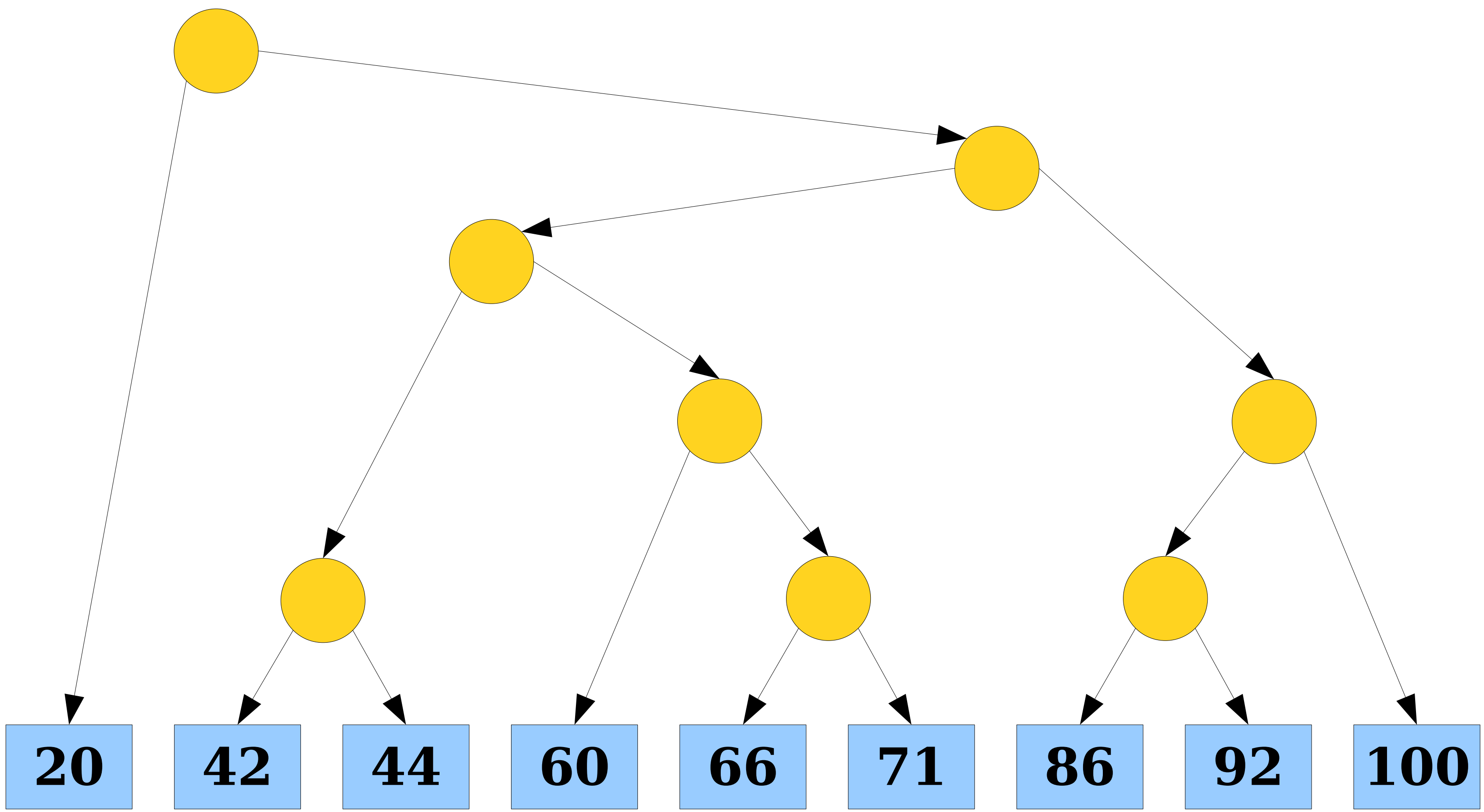
Example: ***Hierarchical Clustering***

1D Hierarchical Clustering

20	**42**	**44**	**60**	**66**	**71**	**86**	**92**	**100**

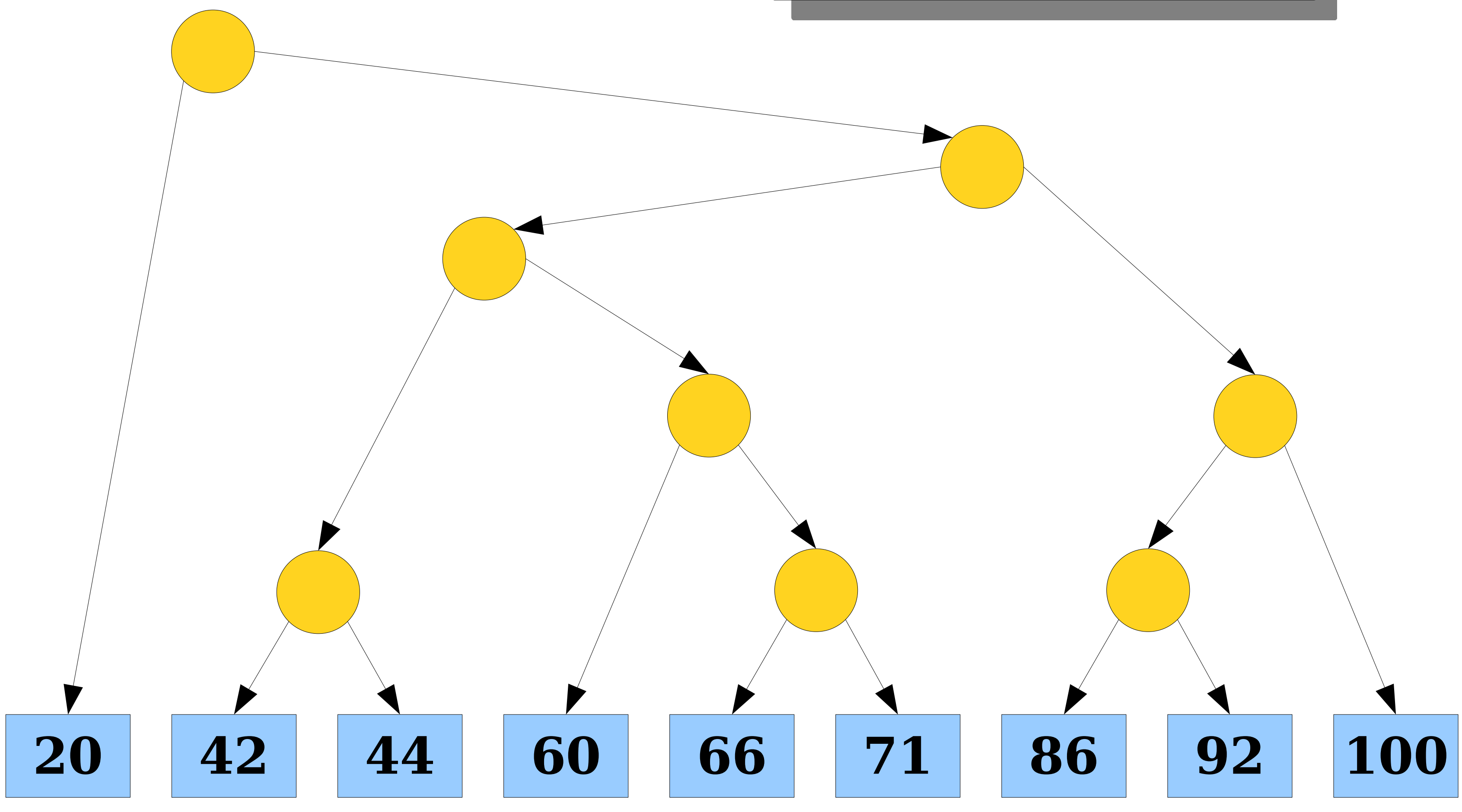
1D Hierarchical Clustering

64.56								
20	**42**	**44**	**60**	**66**	**71**	**86**	**92**	**100**



1D Hierarchical Clustering

This tree is called a ***dendrogram***.



Analyzing the Runtime

- How efficient is this algorithm?
  - Number of rounds: $\Theta(n)$.
  - Work to find closest pair: $O(n)$.
  - Total runtime: $\boldsymbol{\Theta(n^2)}$.
- Can we do better?

Analyzing the Runtime

- How efficient is this algorithm?
  - Number of rounds: $\Theta(n)$.
  - Work to find closest pair: $O(n)$.
  - Total runtime: $\boldsymbol{\Theta(n^2)}$.
- Can we do better?

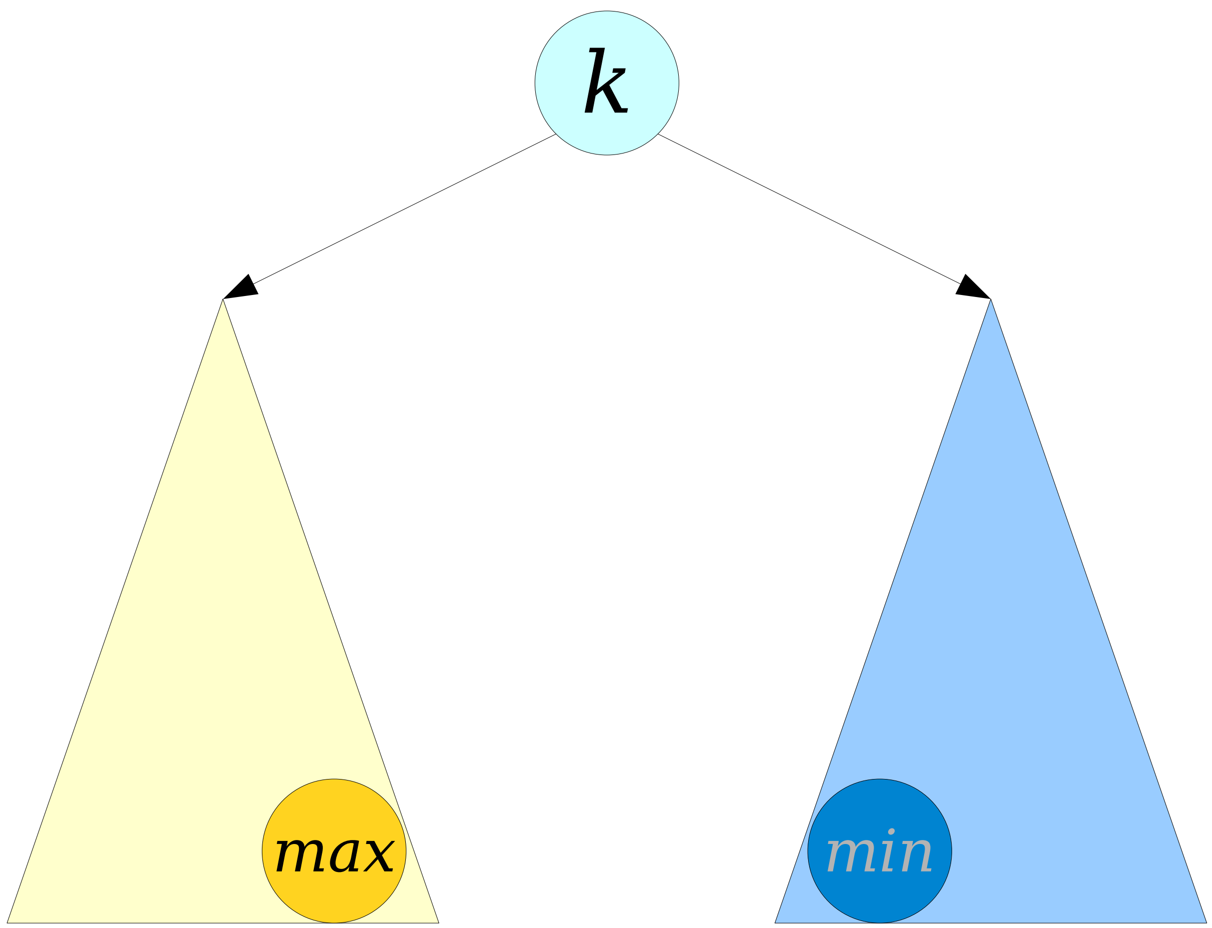
Dynamic 1D Closest Points

- The ***dynamic 1D closest points problem*** is the following:

  Maintain a set of real numbers undergoing insertion and deletion while efficiently supporting queries of the form "what is the closest pair of points?"

- Can we build a better data structure for this?

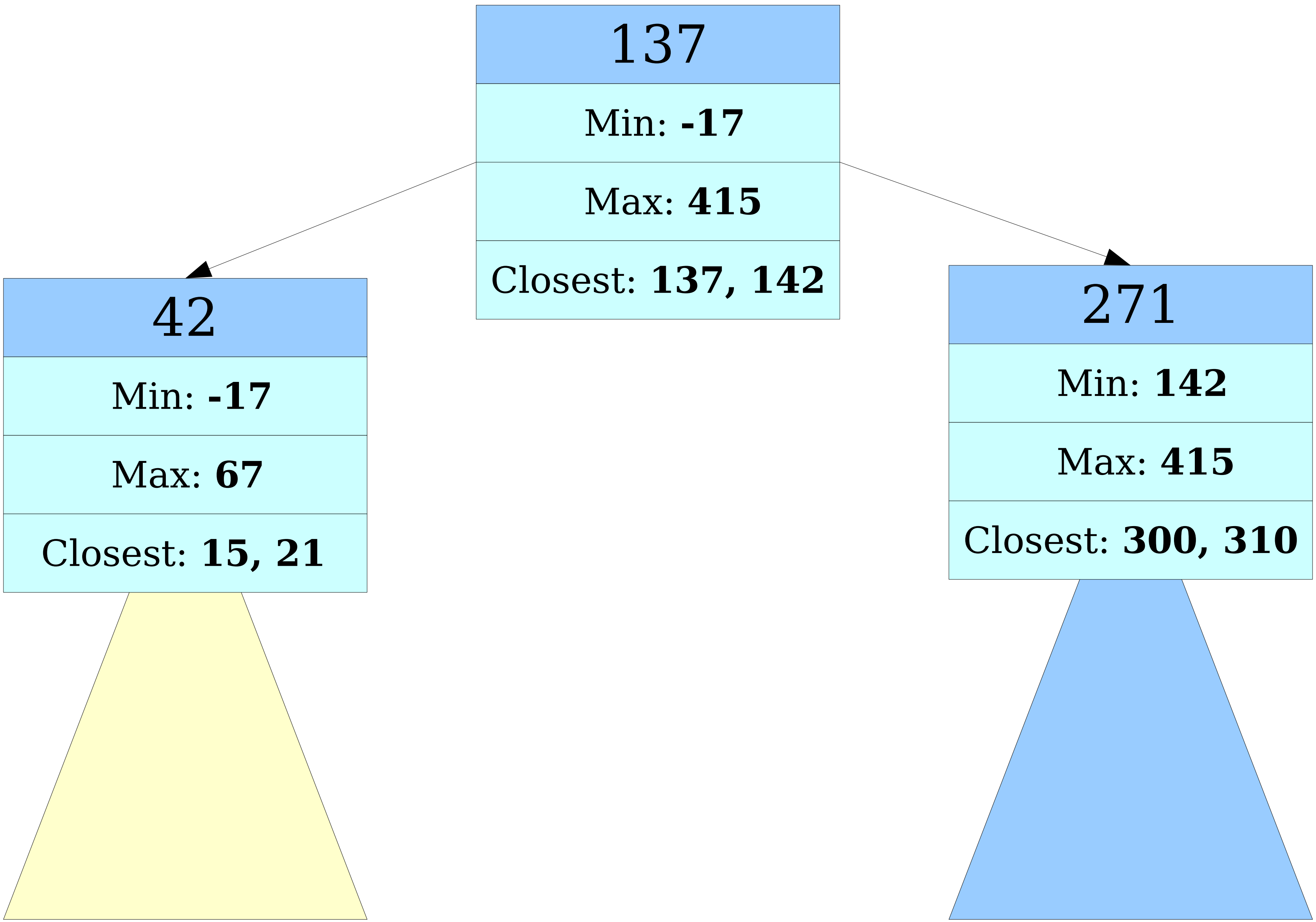
Dynamic 1D Closest Points



A Tree Augmentation

- Augment each node to store the following:
  - The maximum value in the tree.
  - The minimum value in the tree.
  - The closest pair of points in the tree.
- ***Claim:*** Each of these properties can be computed in time O(1) from the left and right subtrees.
- These properties can be augmented into a red/black tree so that insertions and deletions take time O(log n) and "what is the closest pair of points?" can be answered in time O(1).

Dynamic 1D Closest Points



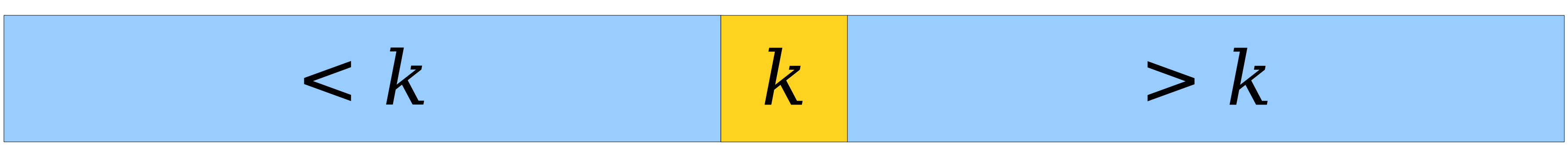
Some Other Questions

- How would you augment this tree so that you can efficiently (in time O(1)) compute the appropriate weighted averages?
- ***Trickier:*** Is this the fastest possible algorithm for this problem?
  - What if you're guaranteed that the keys are all integers in some nice range?

A Helpful Intuition

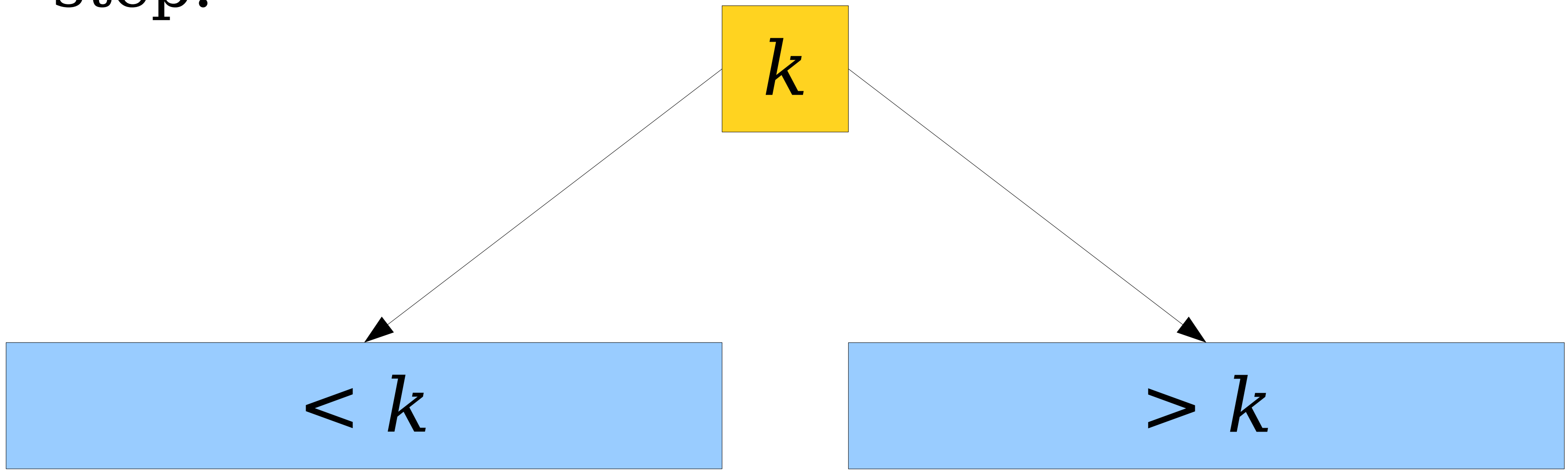
Divide-and-Conquer

- Initially, it can be tricky to come up with the right tree augmentations.
- ***Useful intuition:*** Imagine you're writing a divide-and-conquer algorithm over the elements and have O(1) time per "conquer" step.



Divide-and-Conquer

- Initially, it can be tricky to come up with the right tree augmentations.
- ***Useful intuition:*** Imagine you're writing a divide-and-conquer algorithm over the elements and have O(1) time per “conquer” step.



Next Time

- ***String Data Structures***
  - Storing and manipulating sequences.
- ***Tries and Patricia Trees***
  - Storing a collection of strings efficiently.
- ***Suffix Trees***
  - The Swiss Army Knife of strings.