Balanced Trees

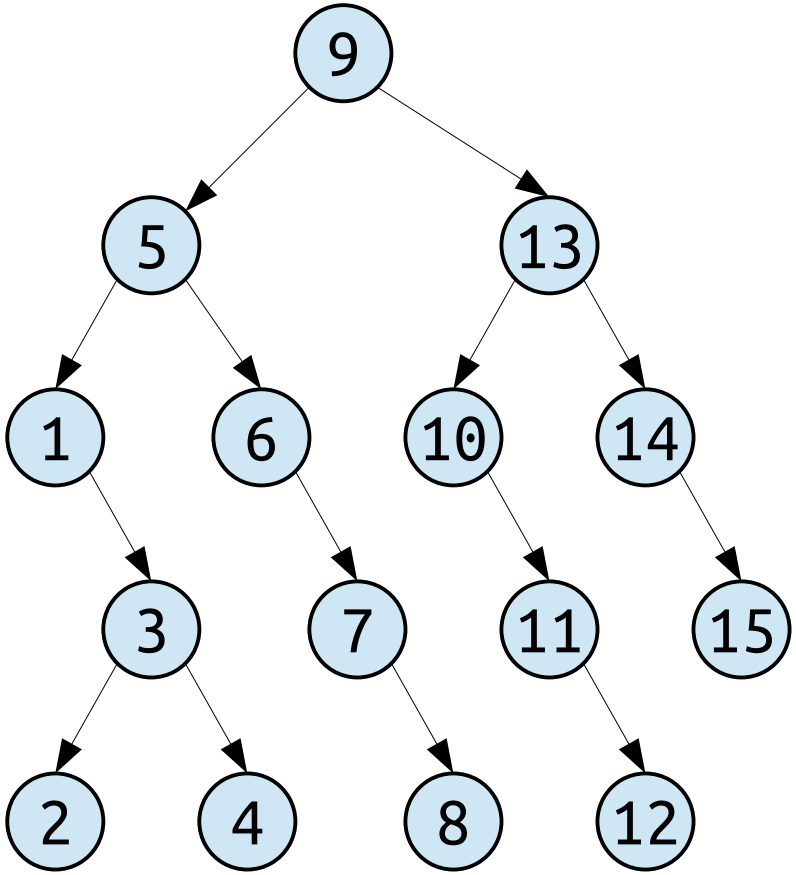
Outline for Today

- ***BST Review***
  - Refresher on basic BST concepts and runtimes.
- ***Overview of Red/Black Trees***
  - What we're building toward.
- ***B-Trees and 2-3-4 Trees***
  - Simple balanced trees, in depth.
- ***Intuiting Red/Black Trees***
  - A much better feel for red/black trees.

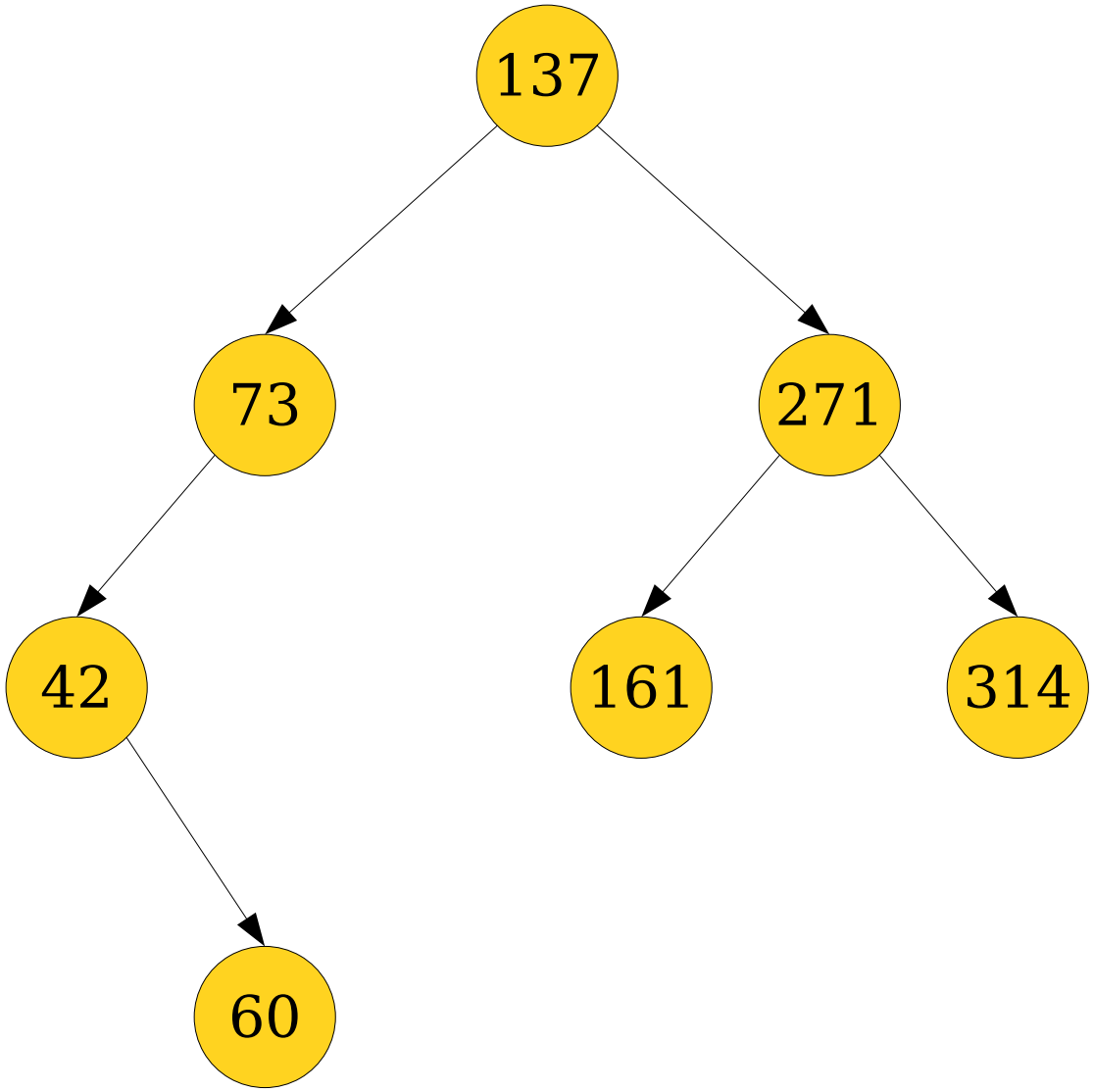
A Quick BST Review

Binary Search Trees

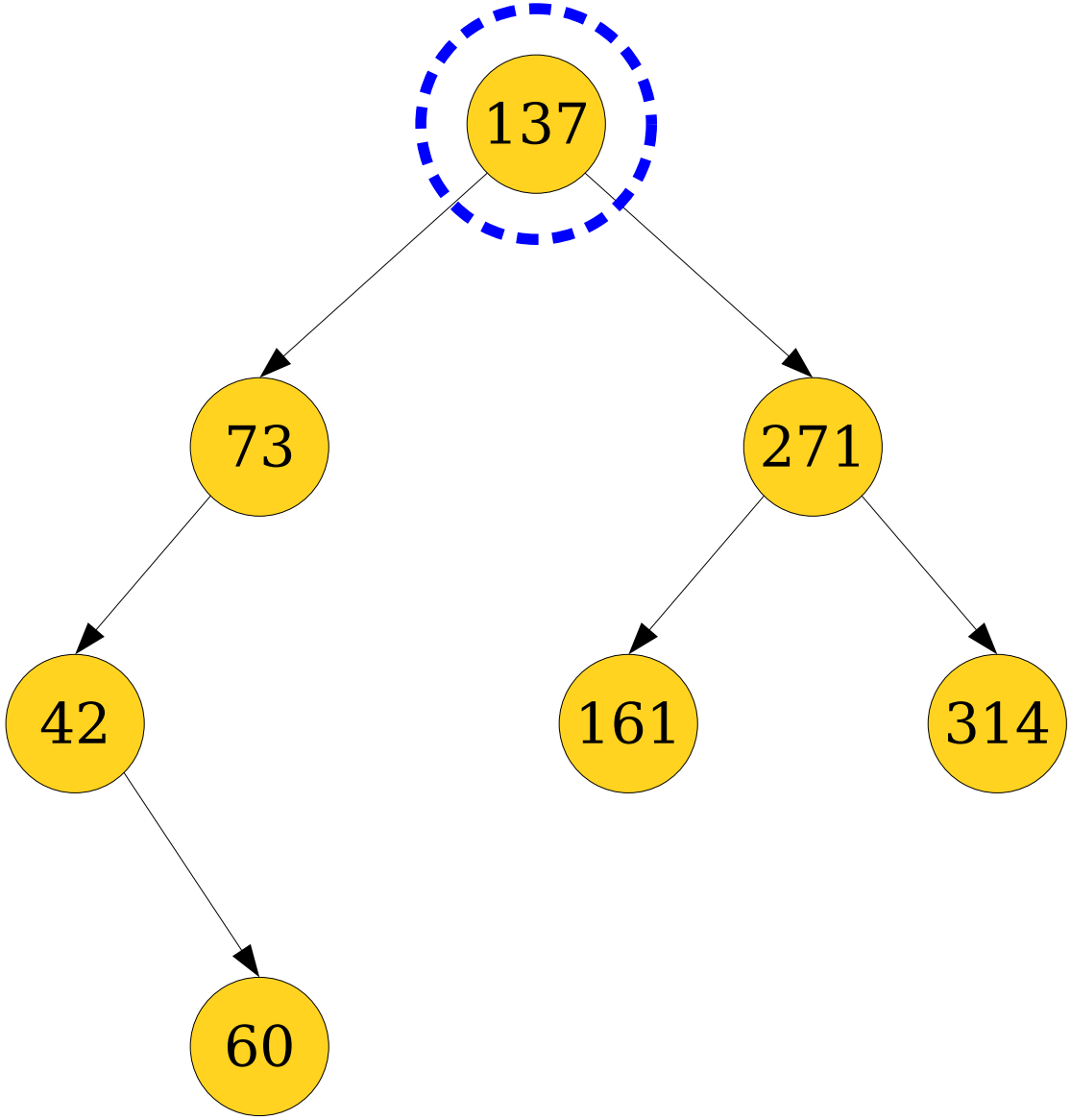
- A ***binary search tree*** is a binary tree with the following properties:
  - Each node in the BST stores a ***key***, and optionally, some auxiliary information.
  - The key of every node in a BST is strictly greater than all keys to its left and strictly smaller than all keys to its right.



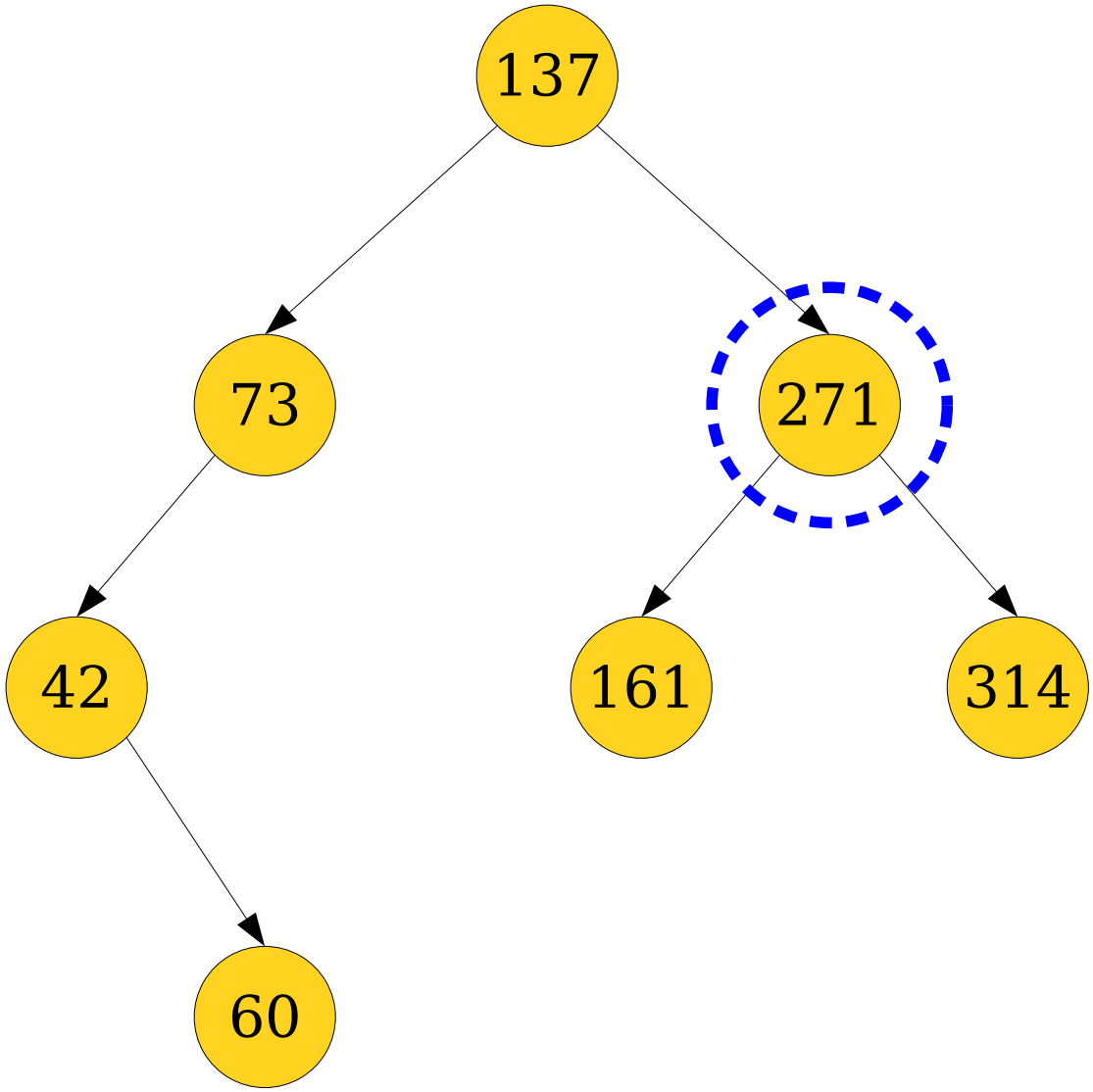
Searching a BST



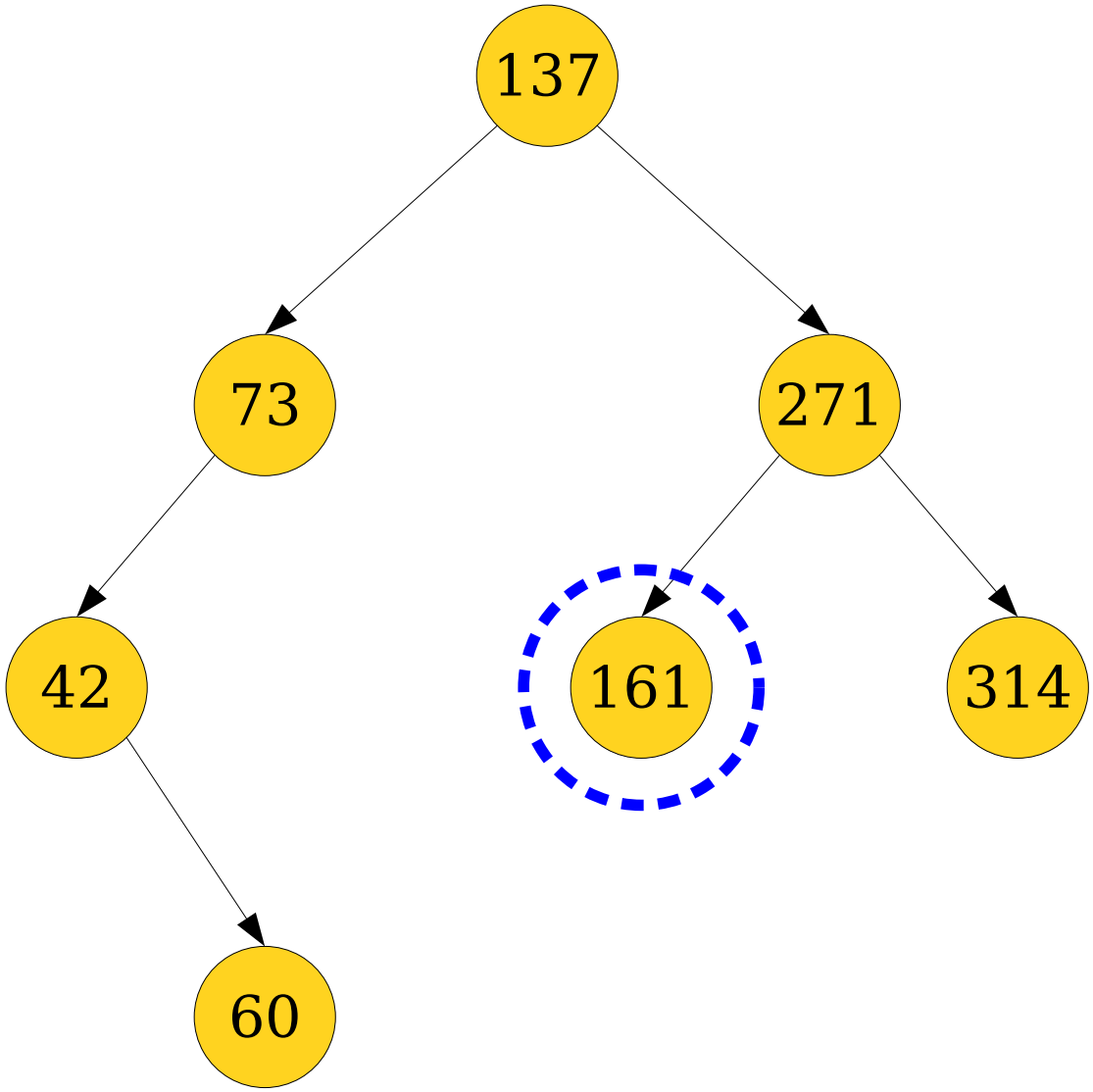
Searching a BST



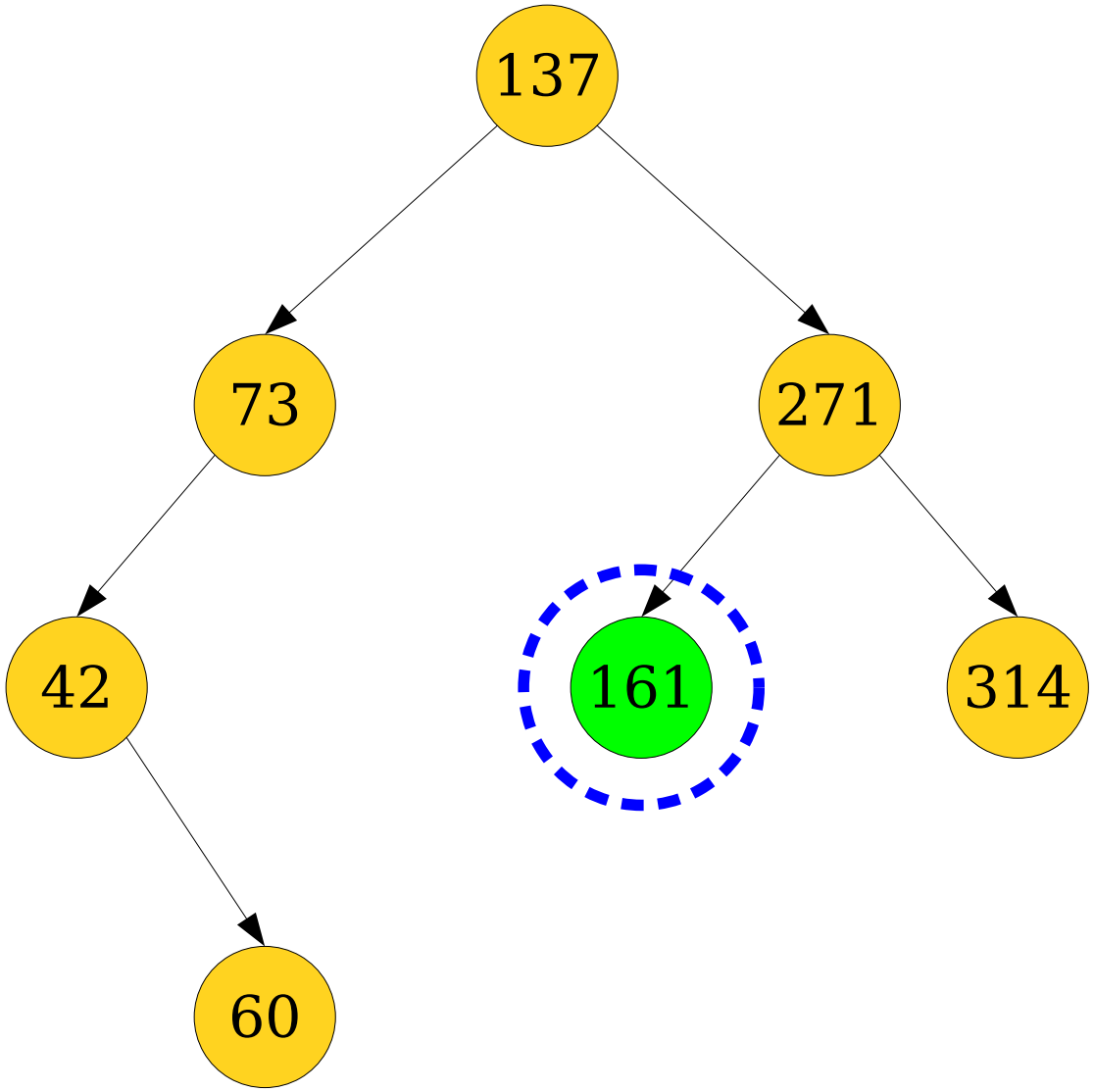
Searching a BST



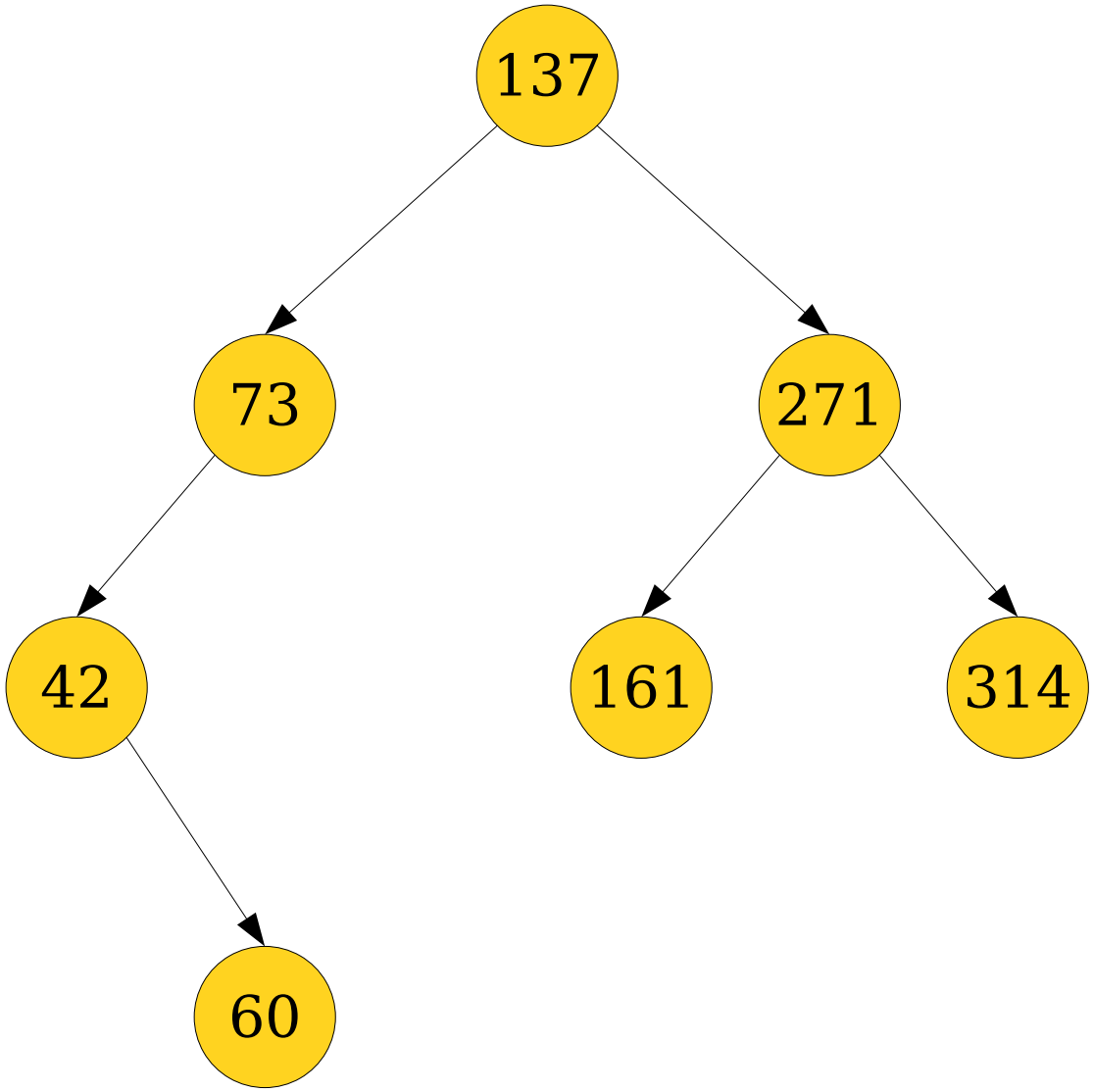
Searching a BST



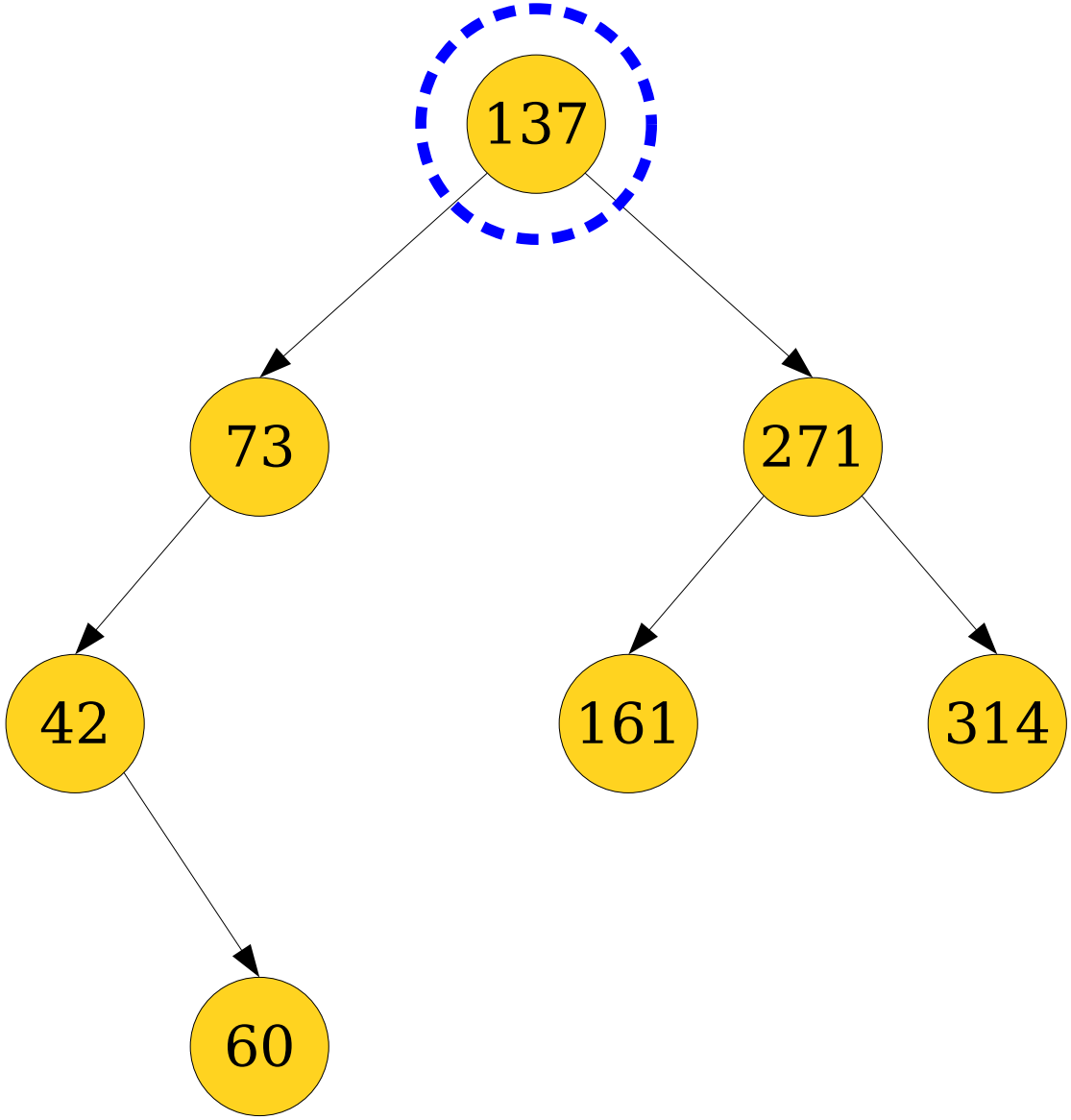
Searching a BST



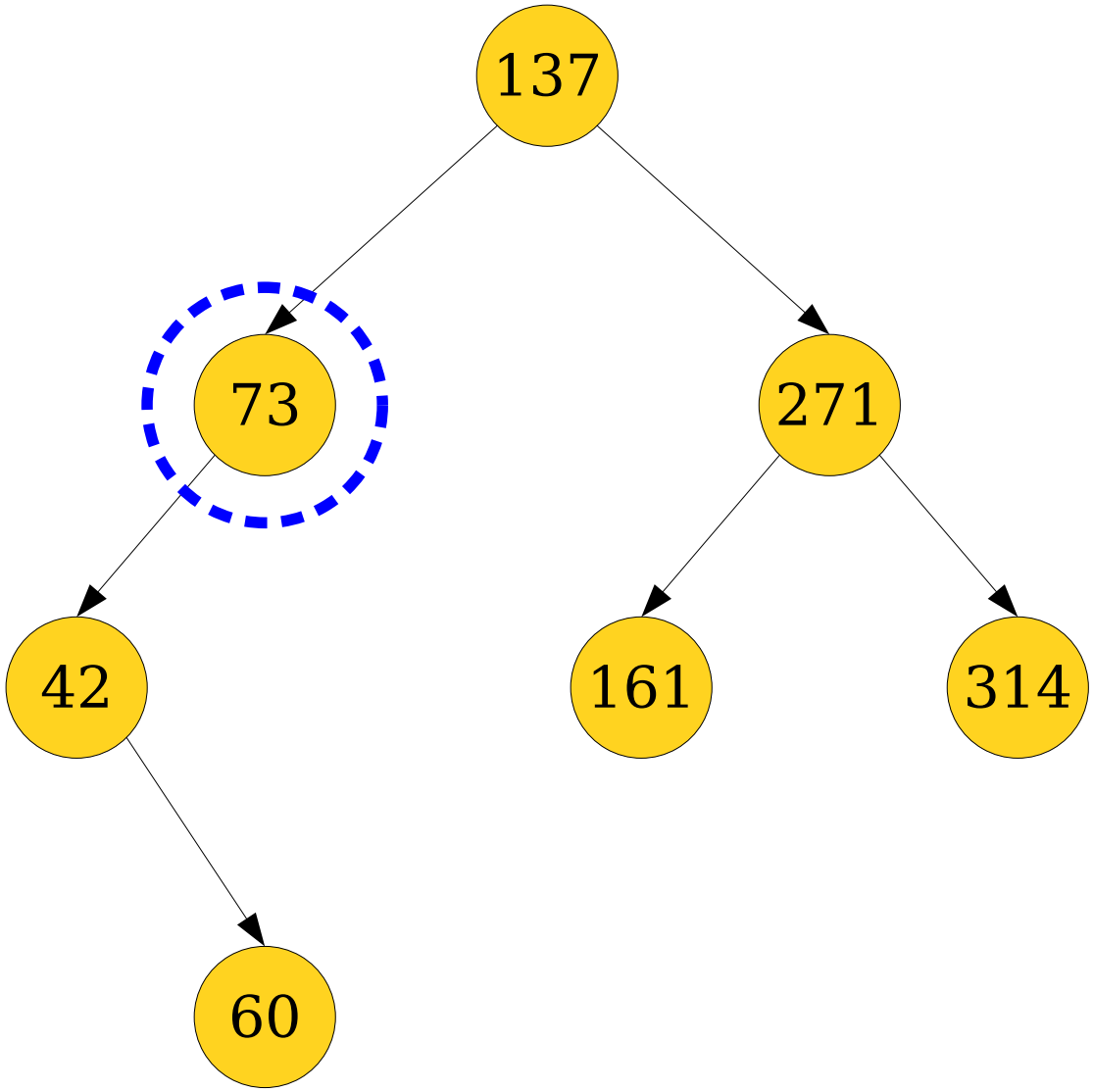
Searching a BST



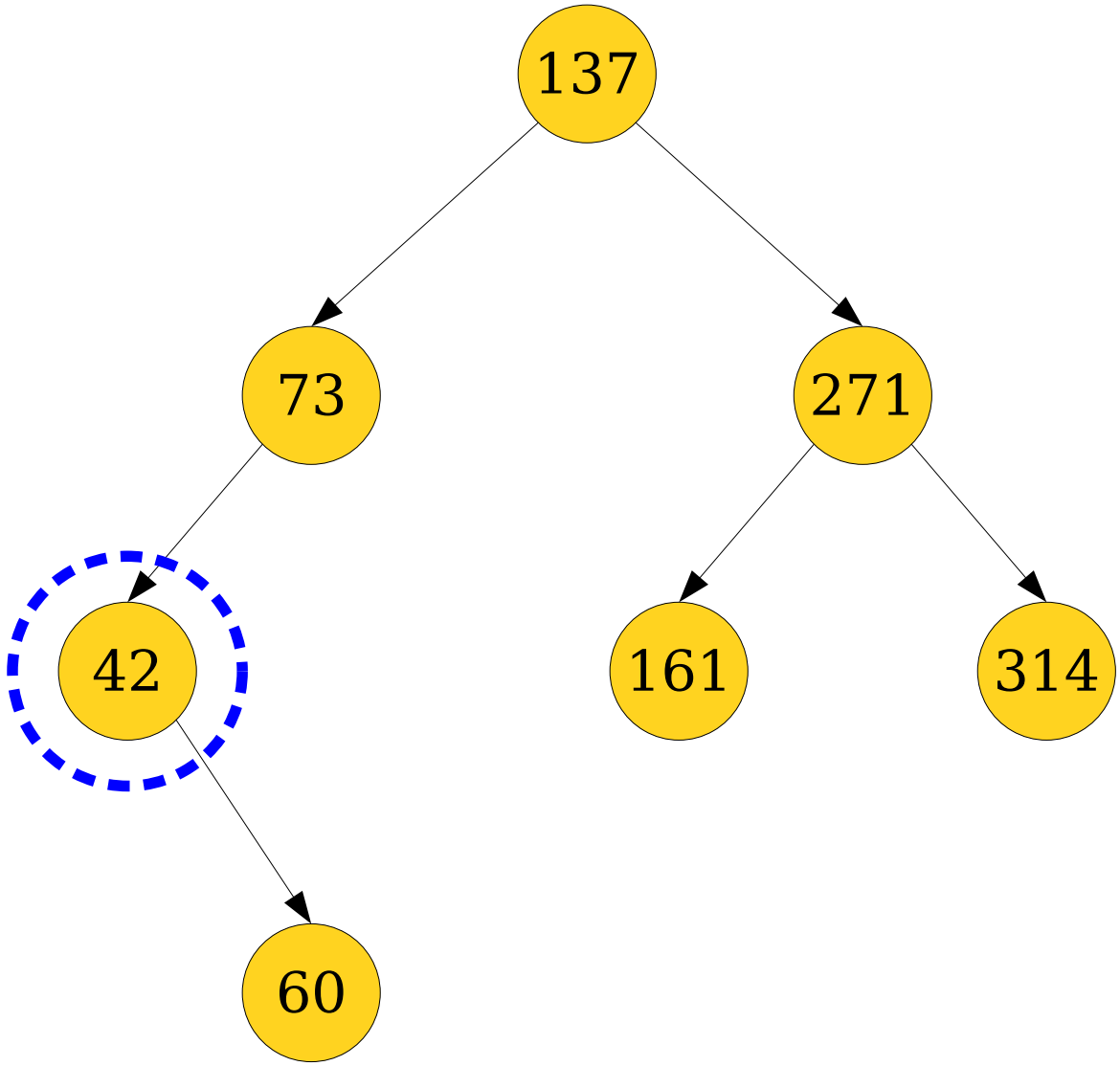
Searching a BST



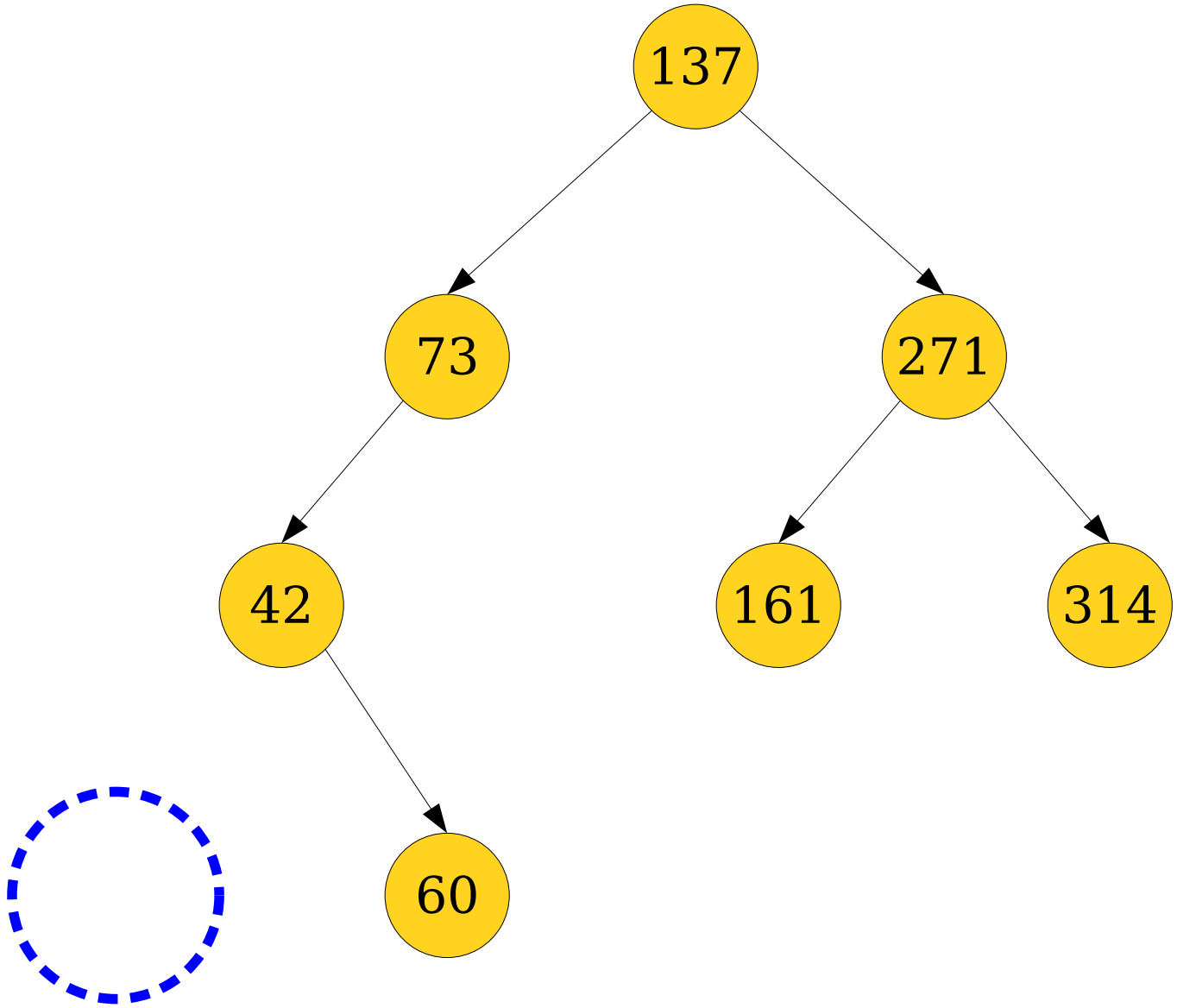
Searching a BST



Searching a BST

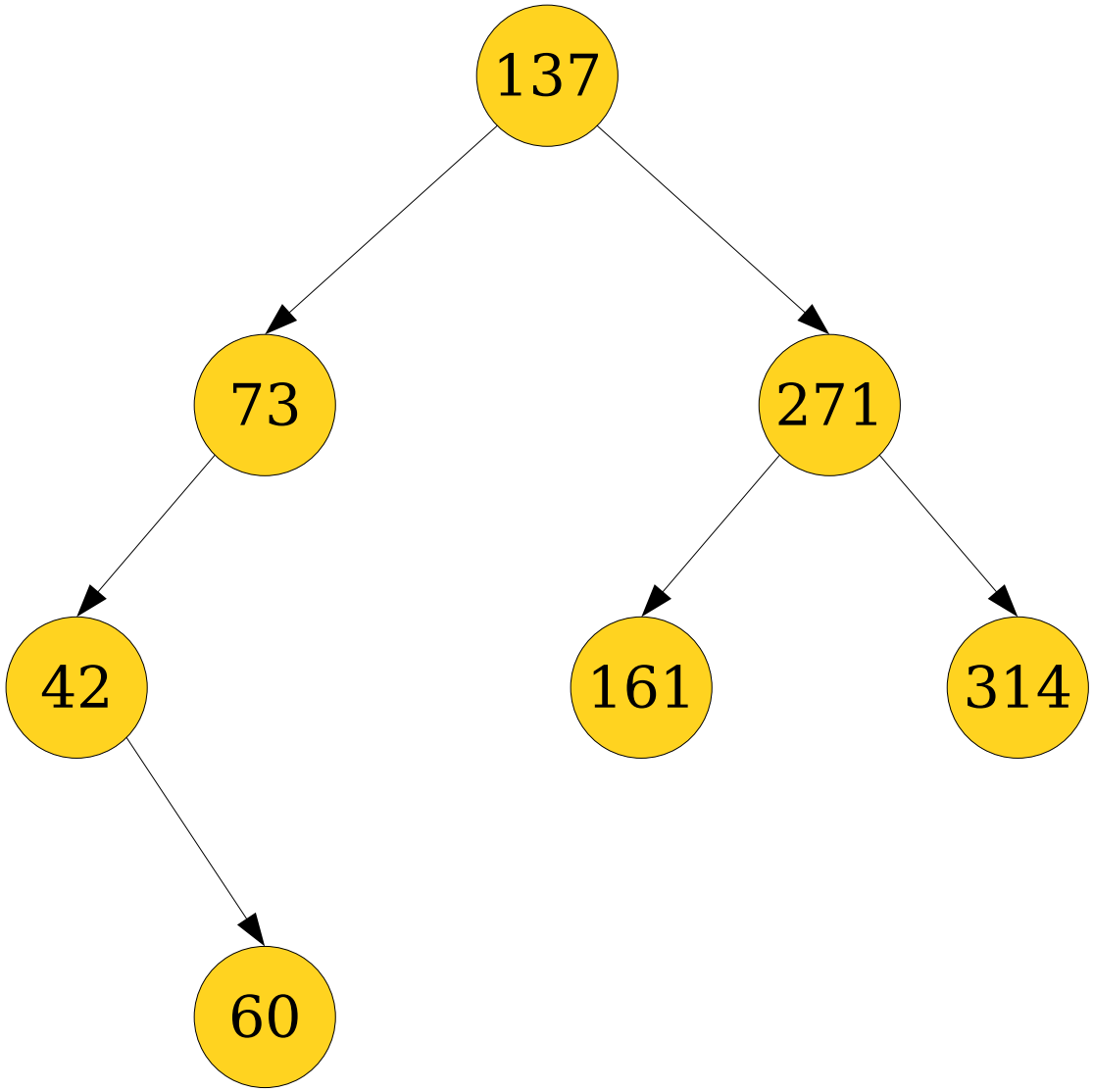


Searching a BST

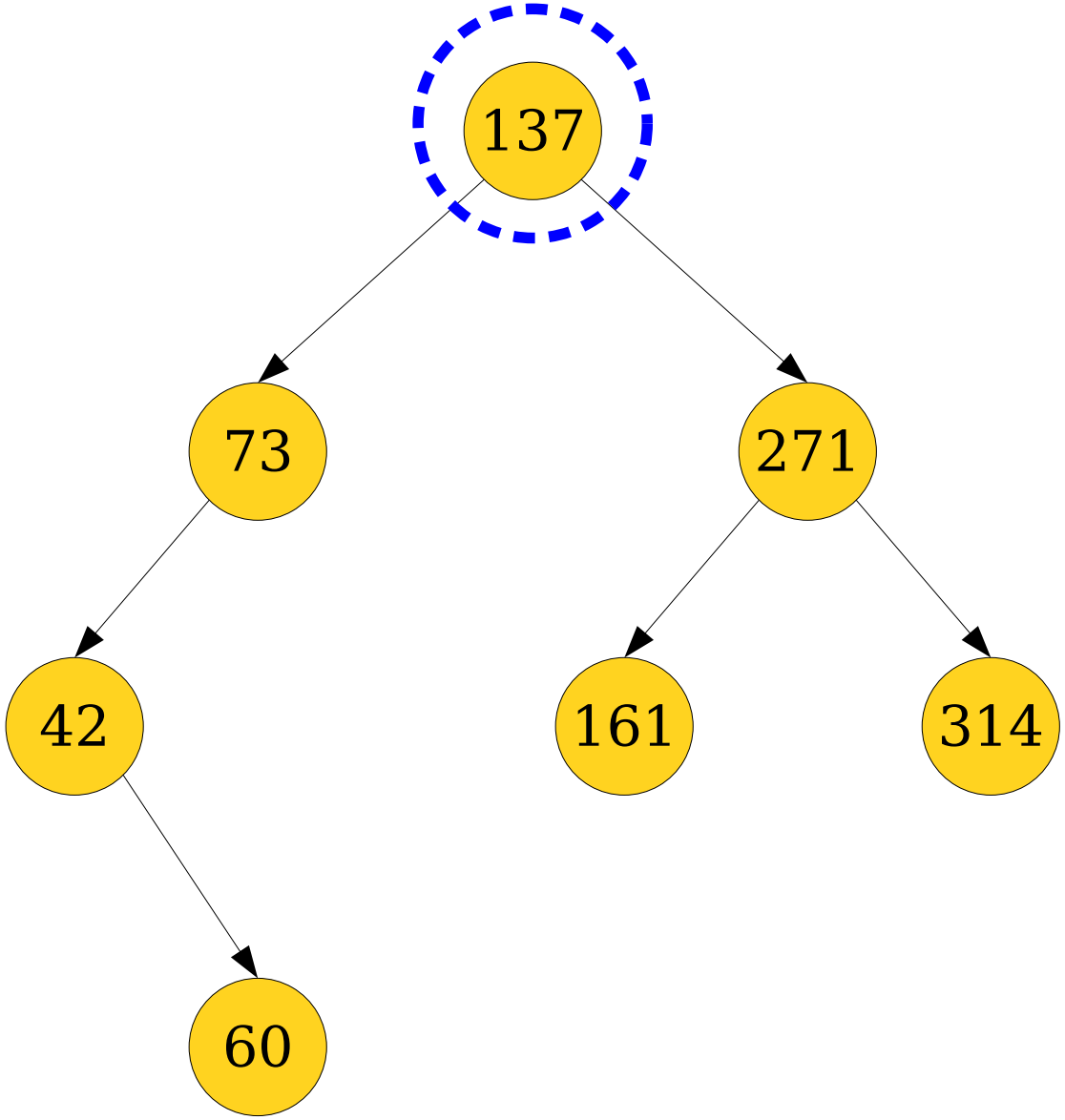


Inserting into a BST

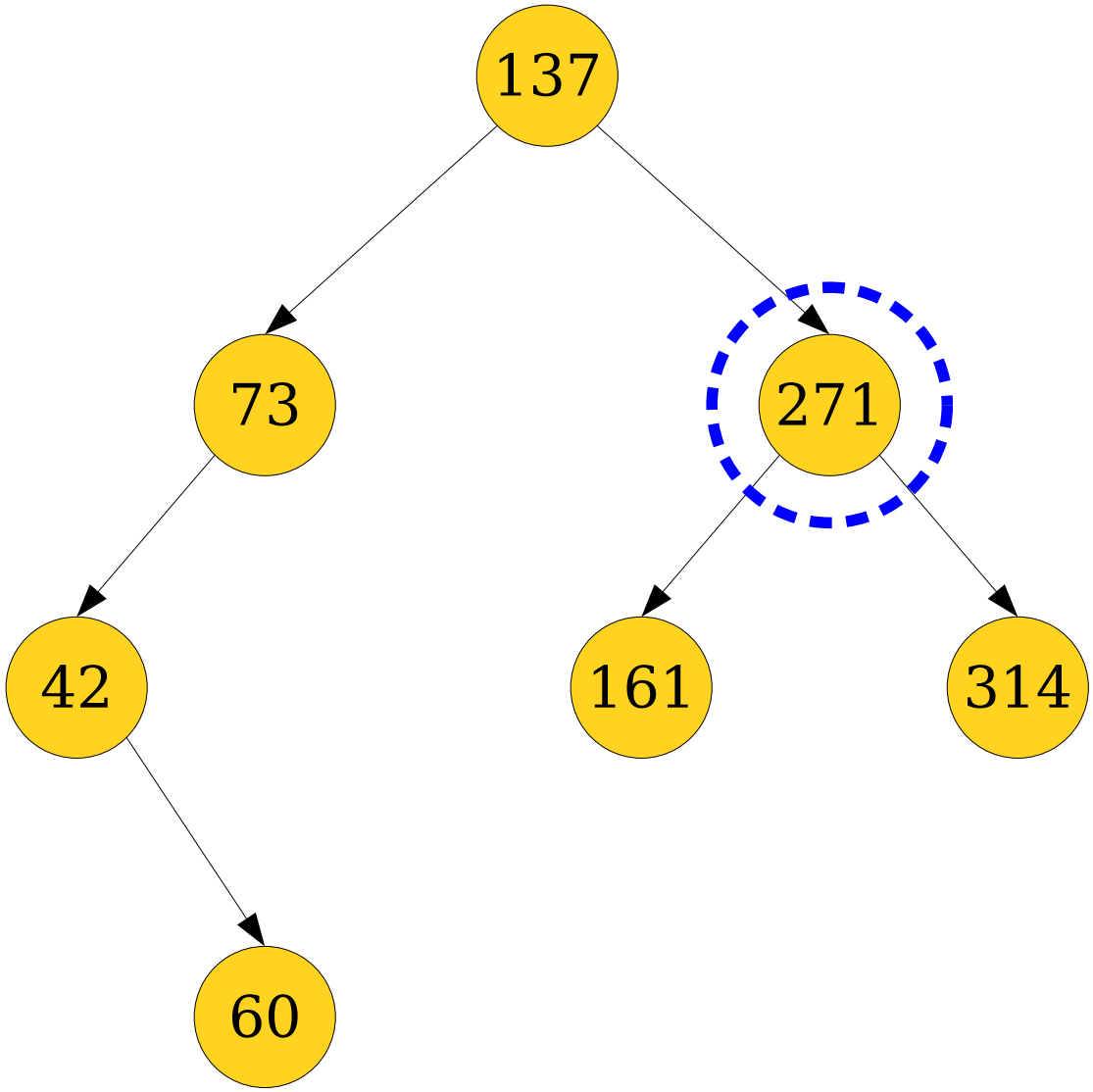
Inserting into a BST



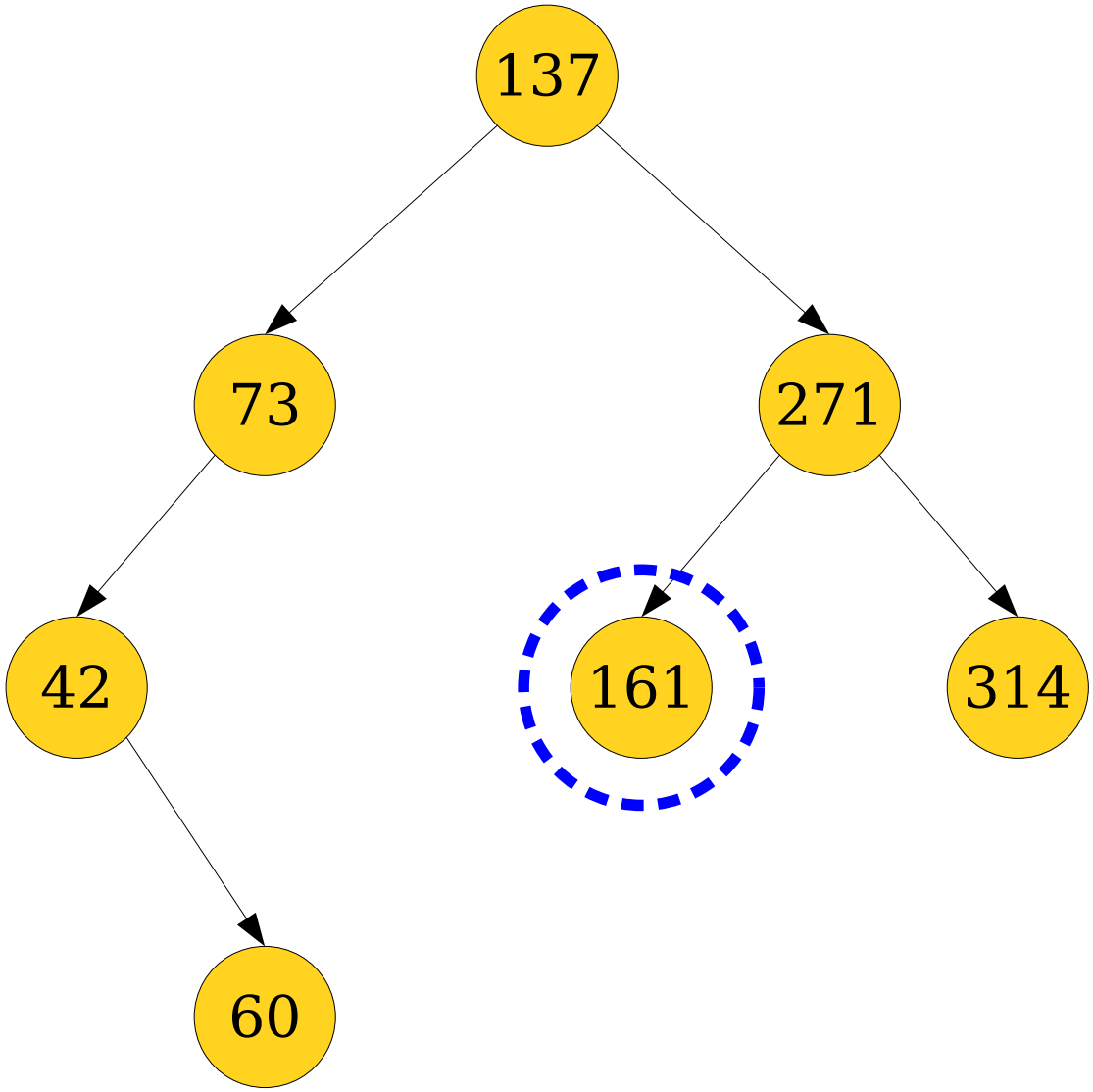
Inserting into a BST



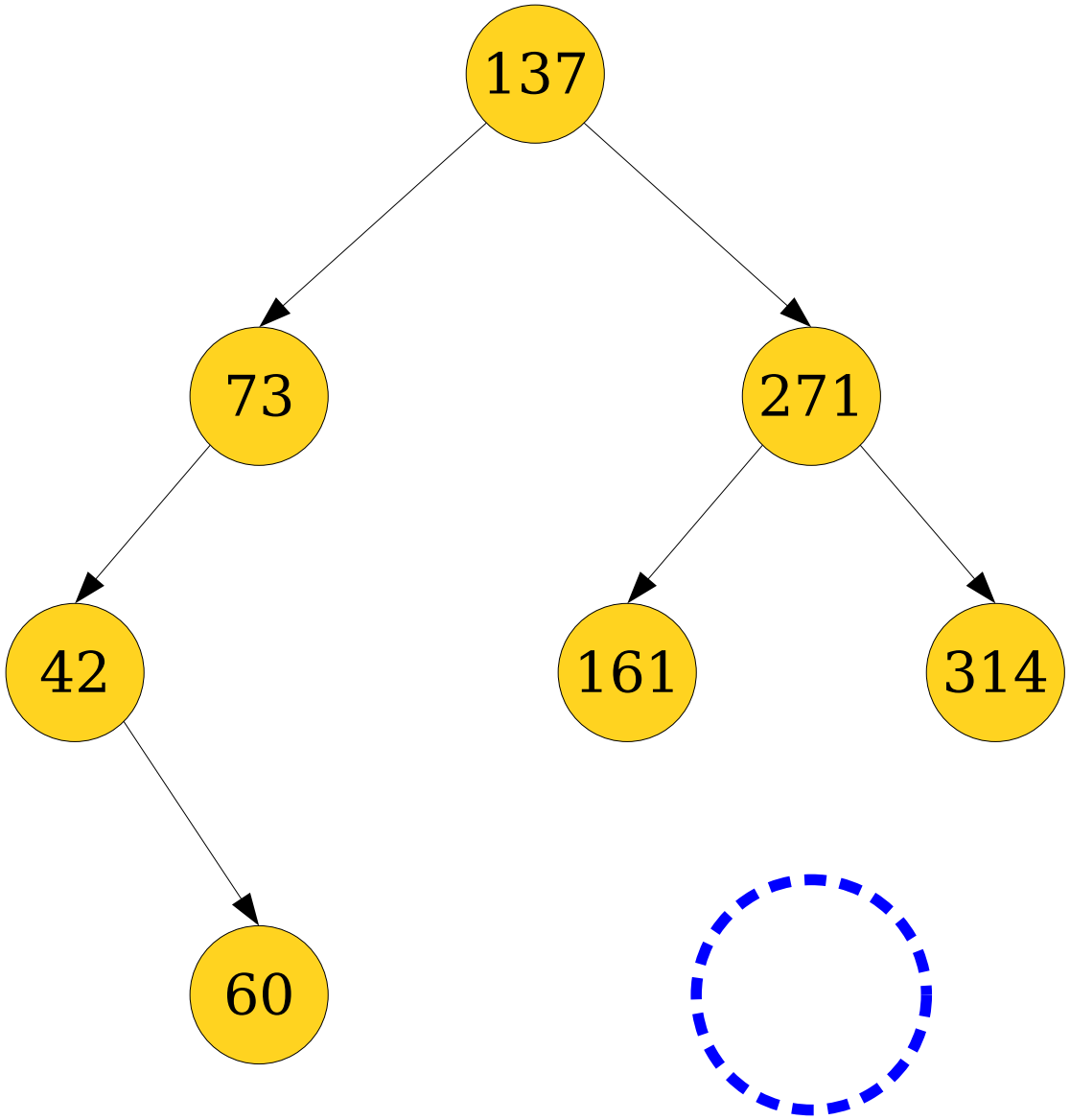
Inserting into a BST



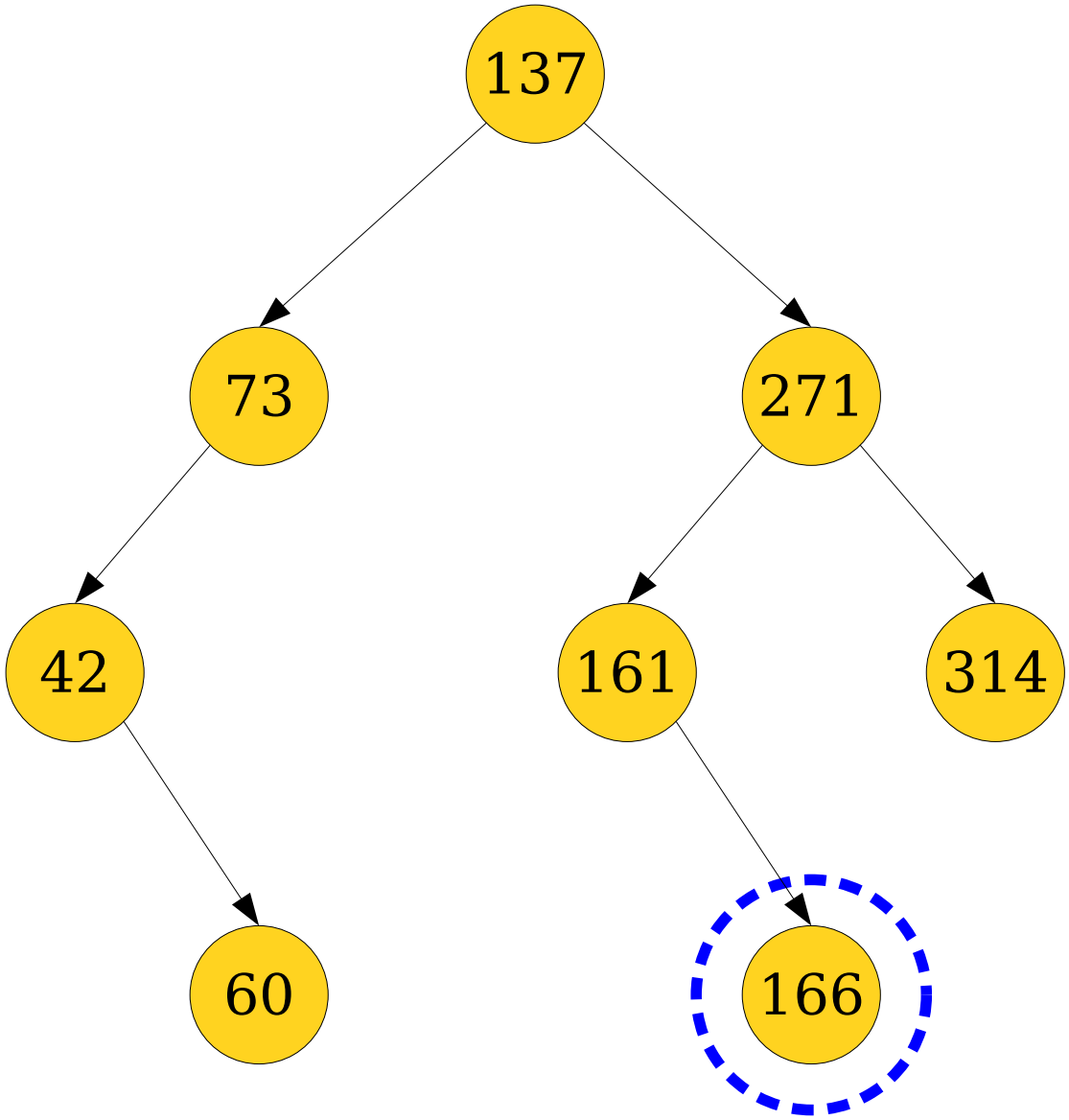
Inserting into a BST



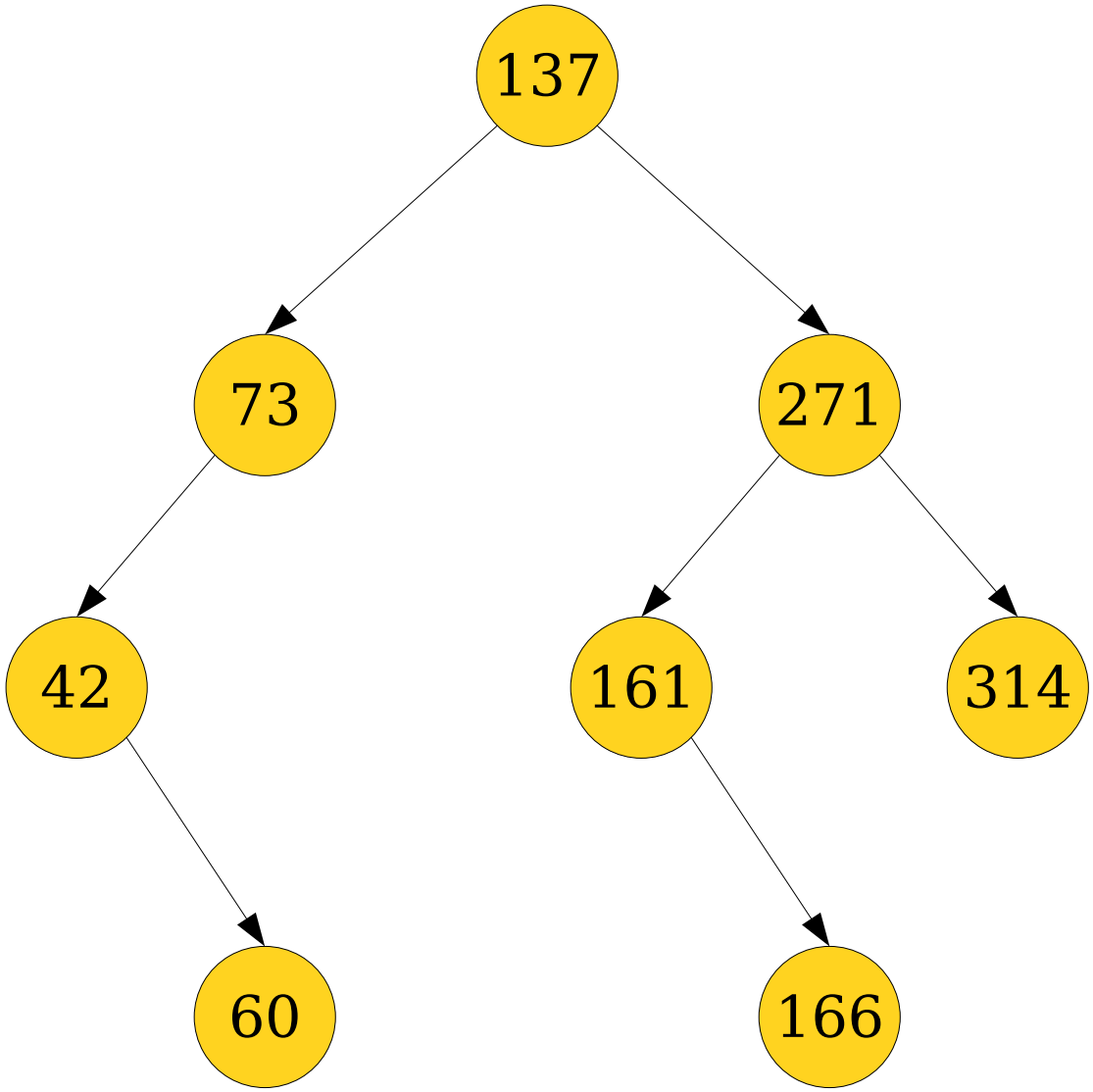
Inserting into a BST



Inserting into a BST



Inserting into a BST

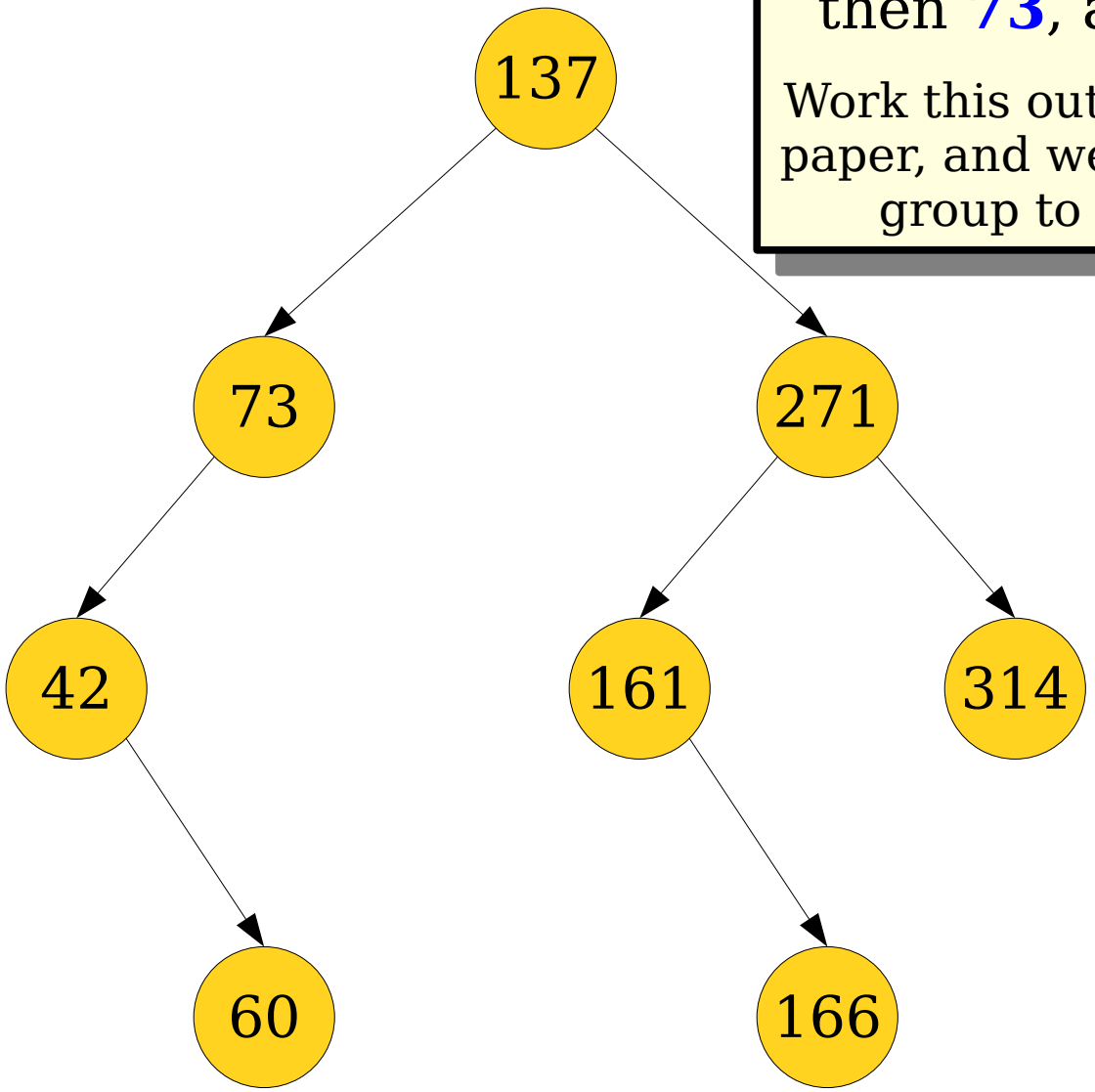


Deleting from a BST

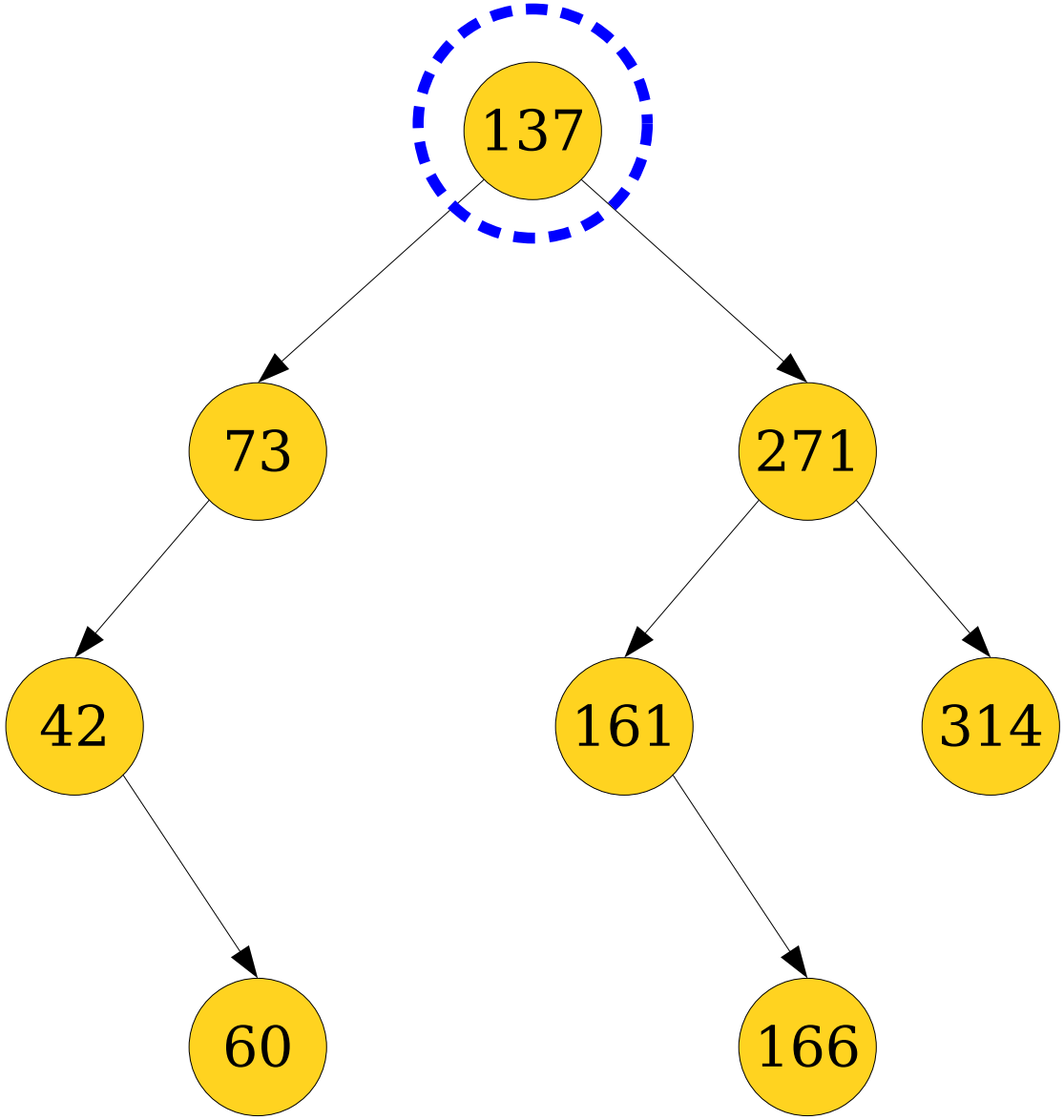
Deleting from a BST

Delete **60** from this tree, then **73**, and then **137**.

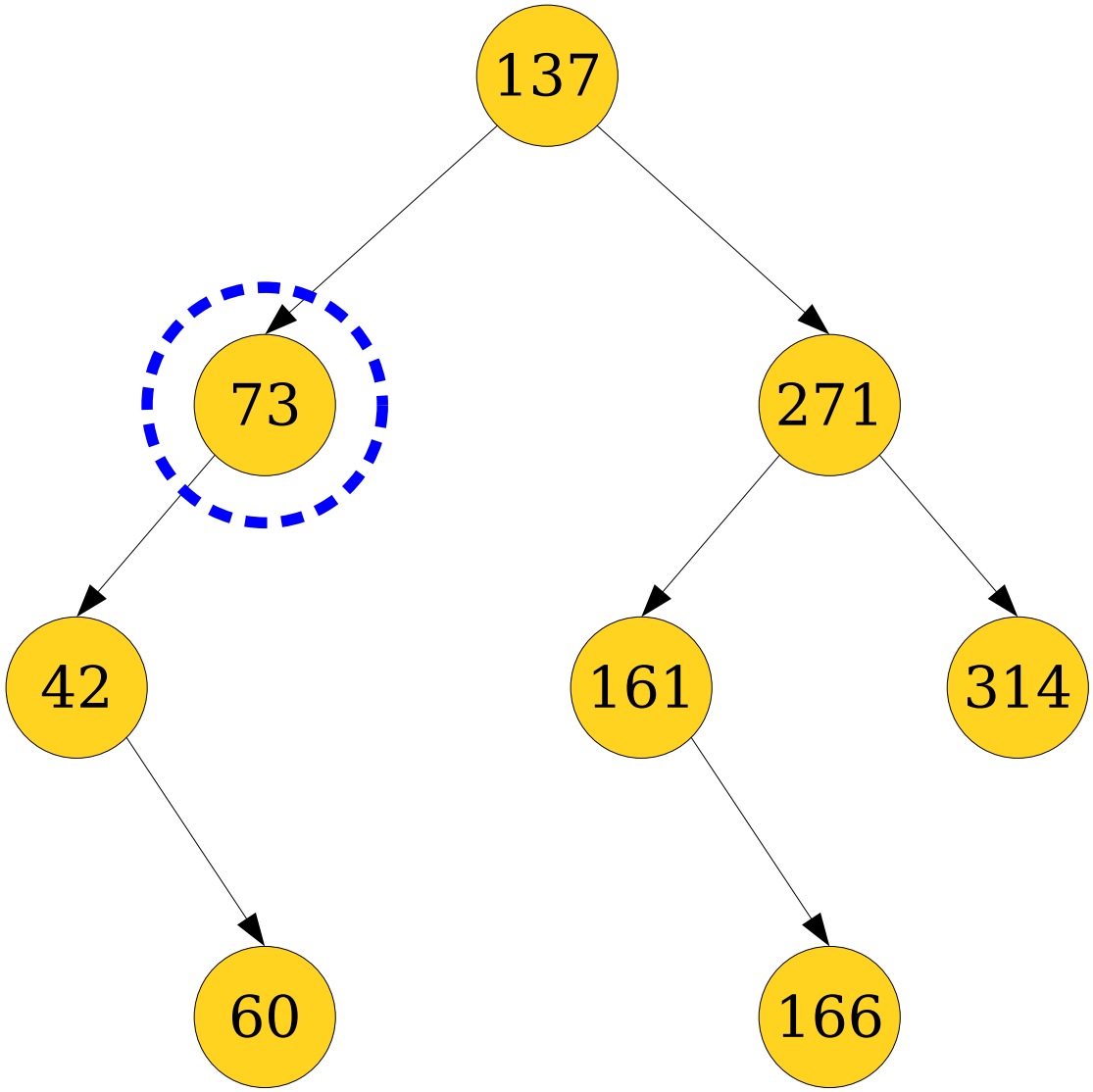
Work this out with a pencil and paper, and we'll reconvene as a group to do it together.



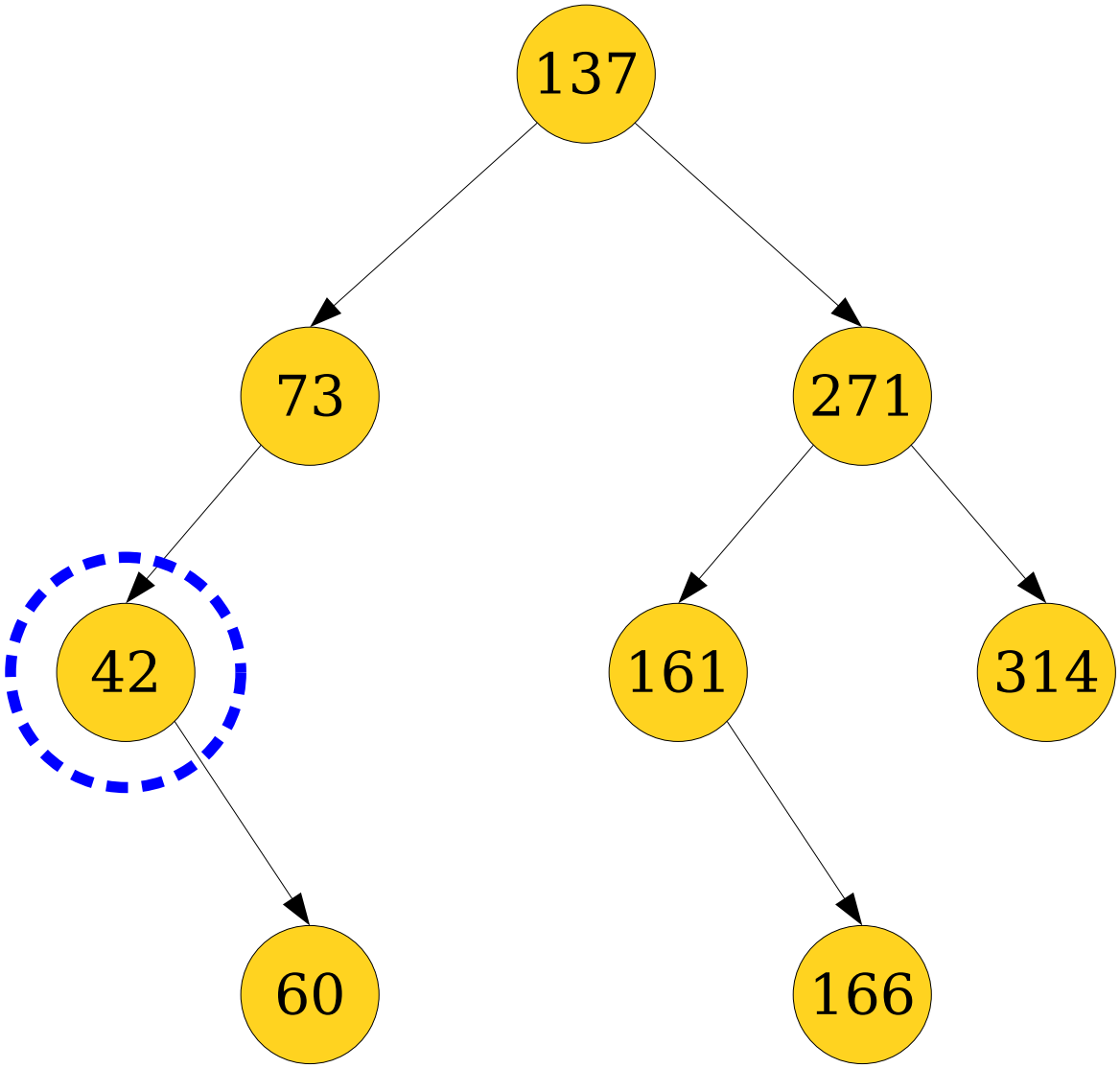
Deleting from a BST



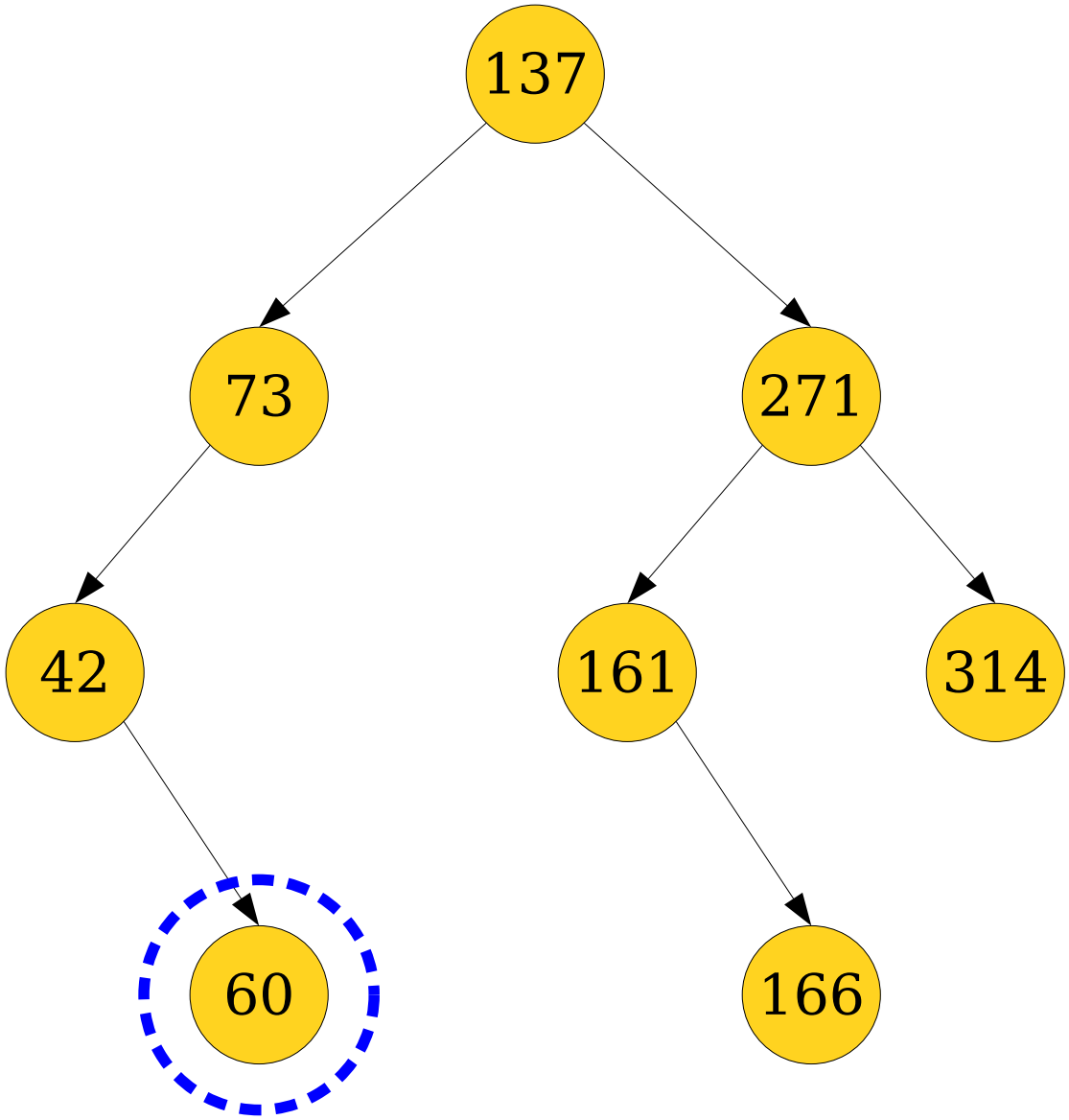
Deleting from a BST



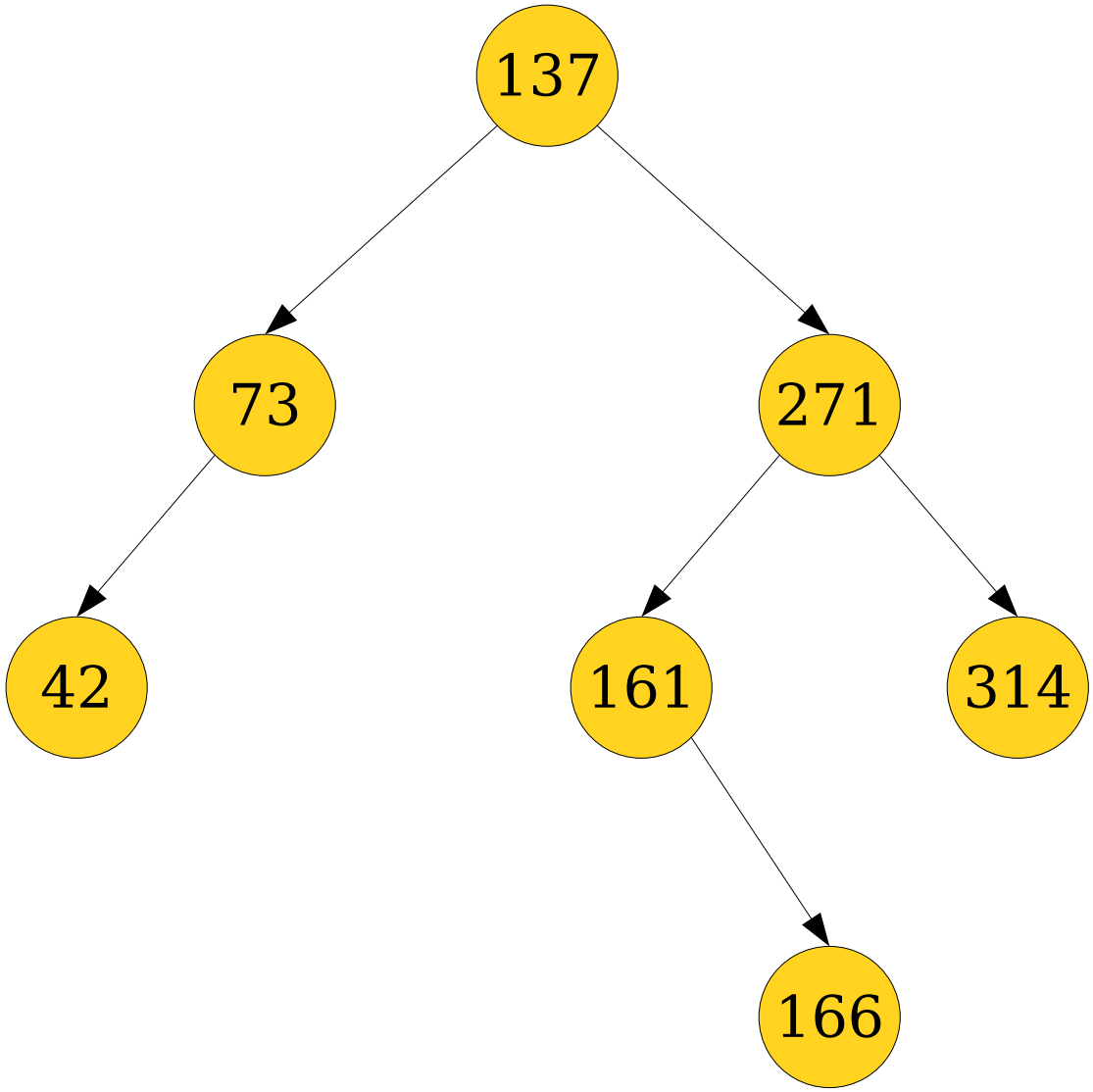
Deleting from a BST



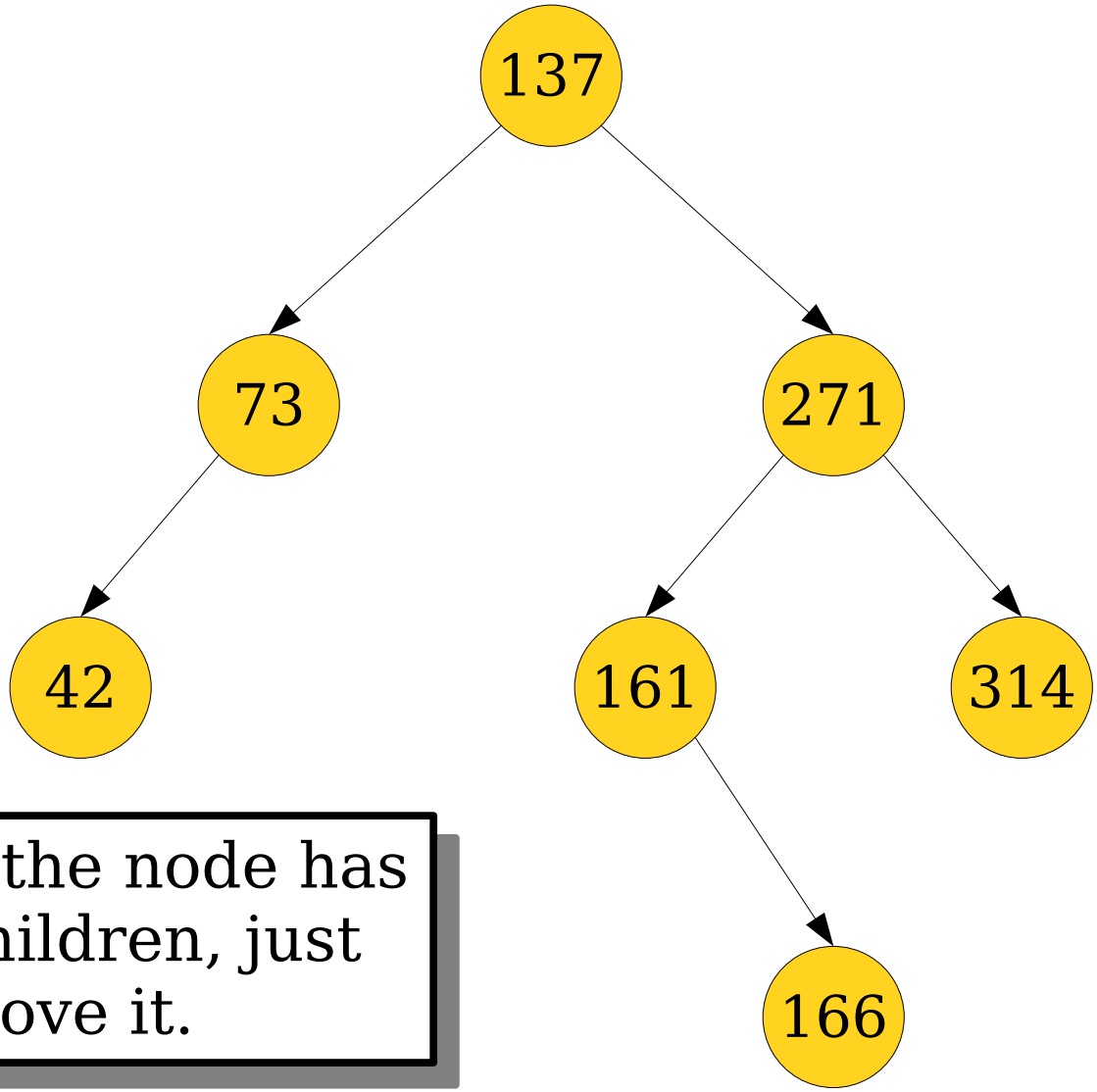
Deleting from a BST



Deleting from a BST

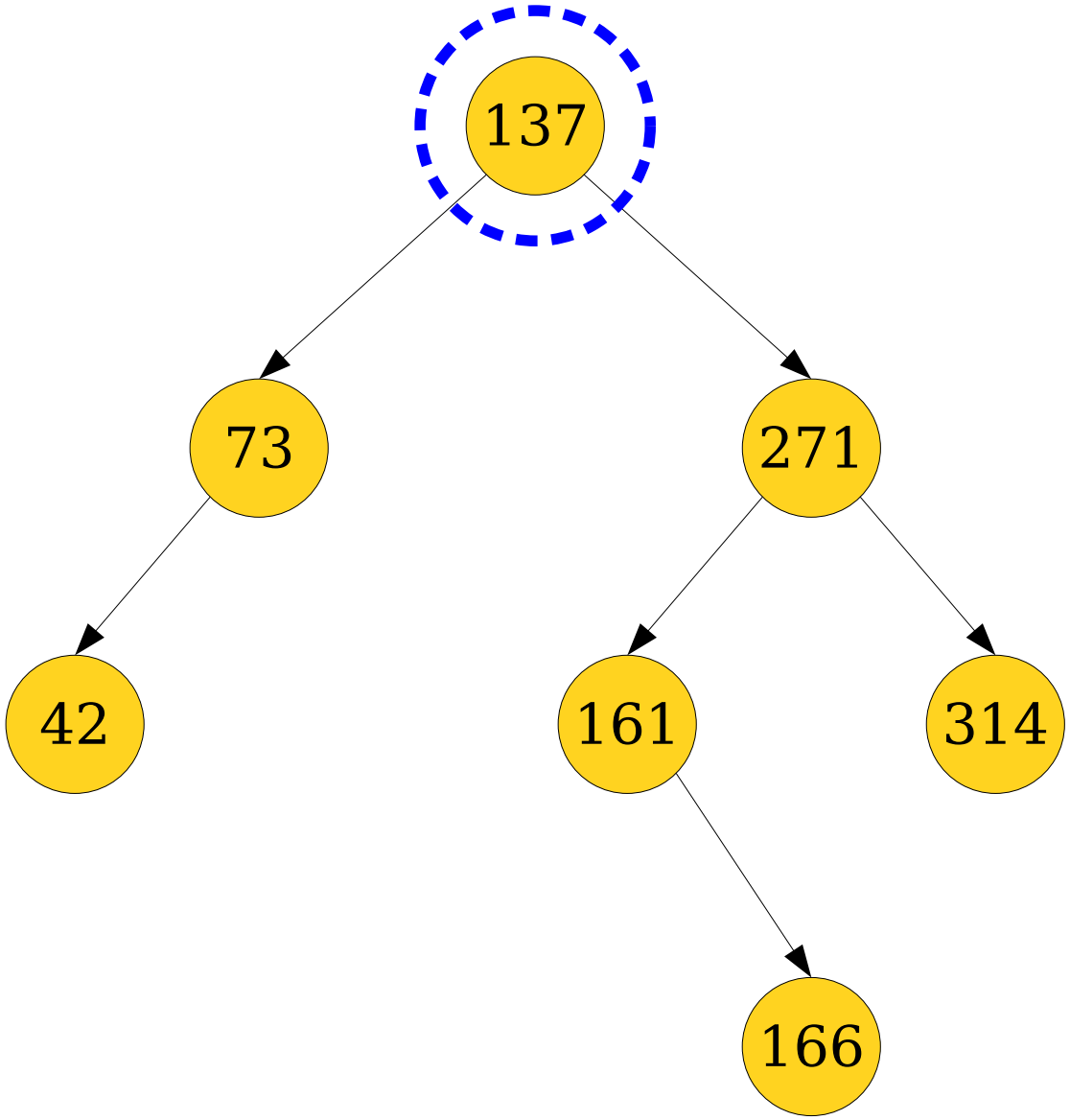


Deleting from a BST

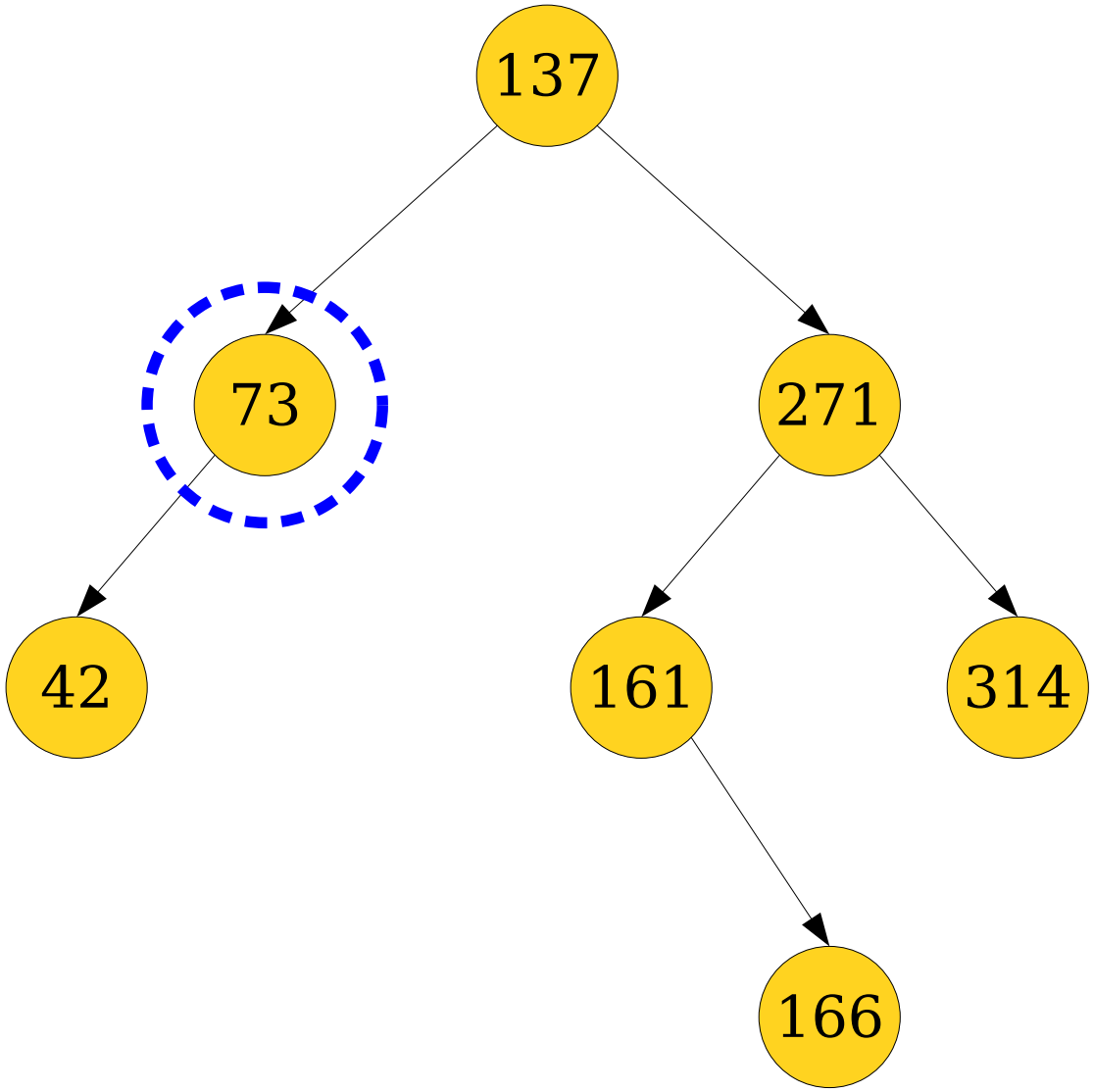


Case 0: If the node has just no children, just remove it.

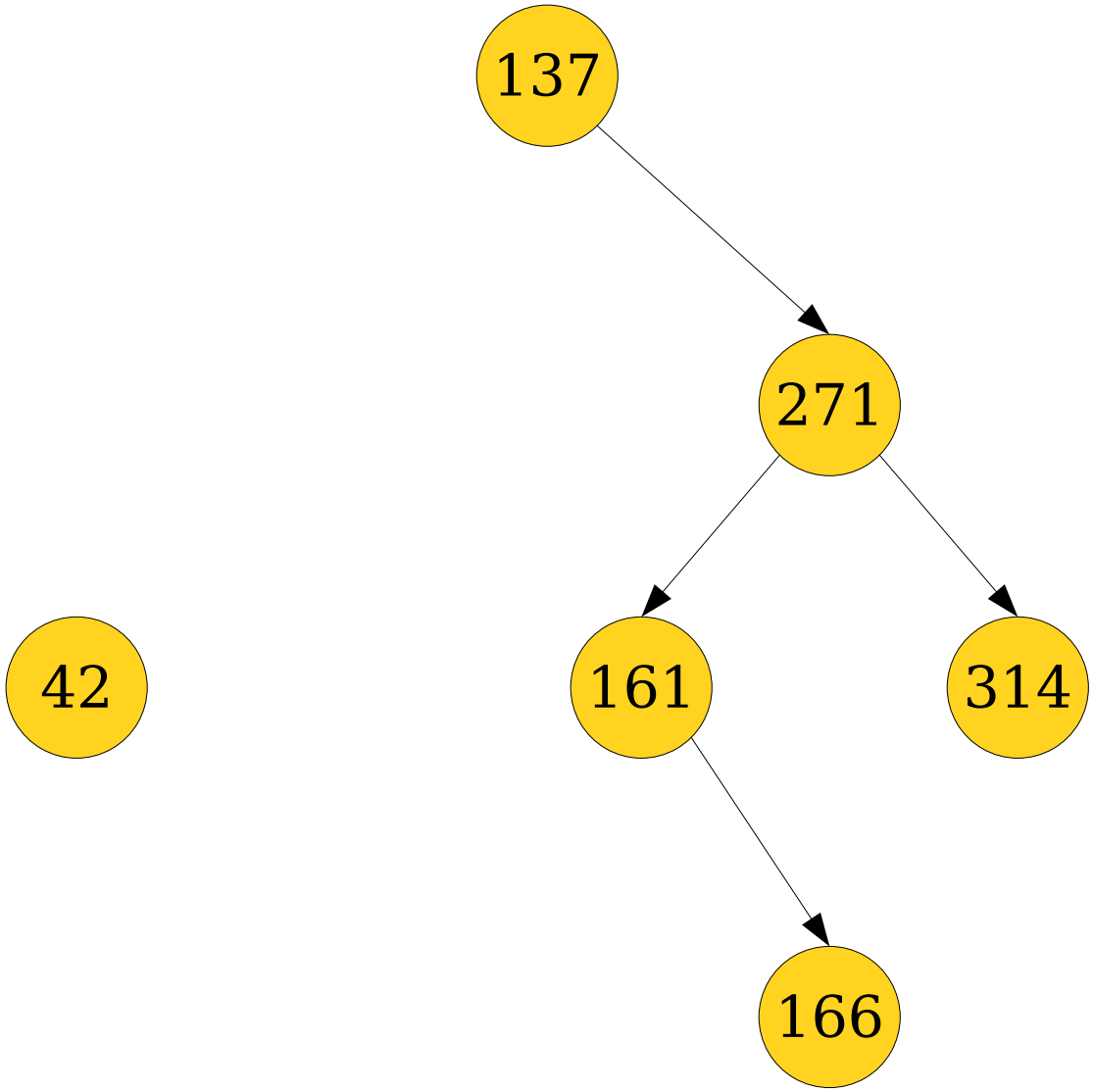
Deleting from a BST



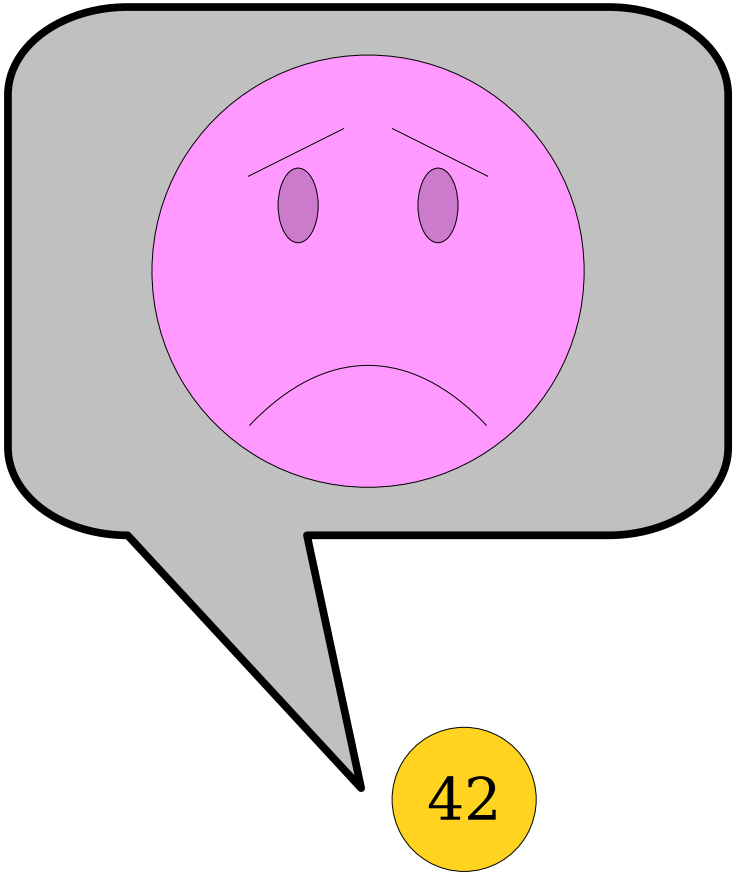
Deleting from a BST



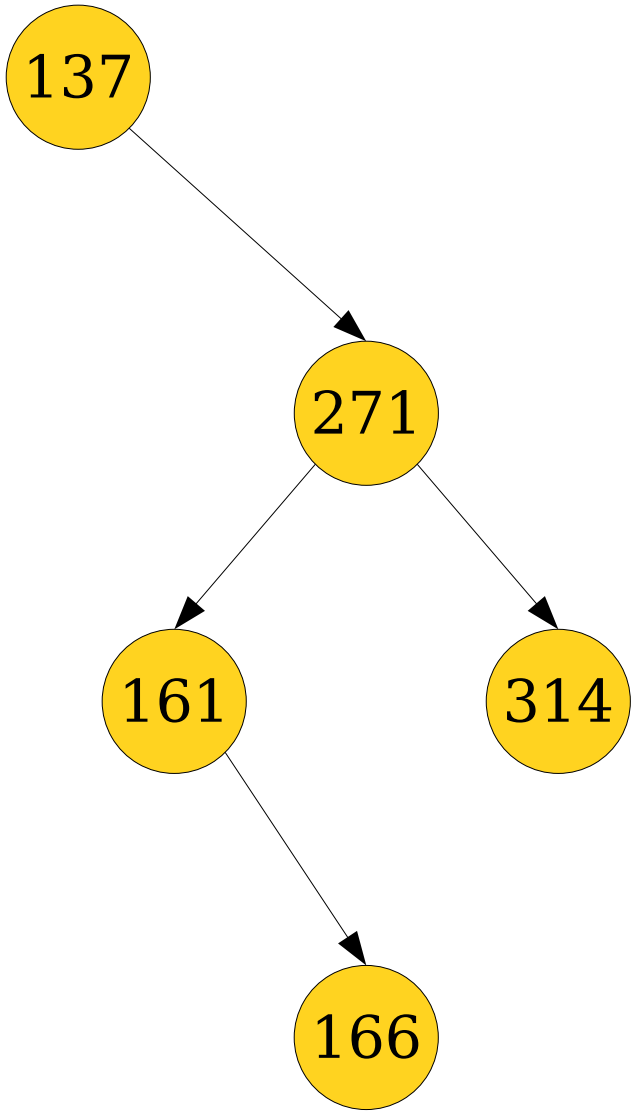
Deleting from a BST



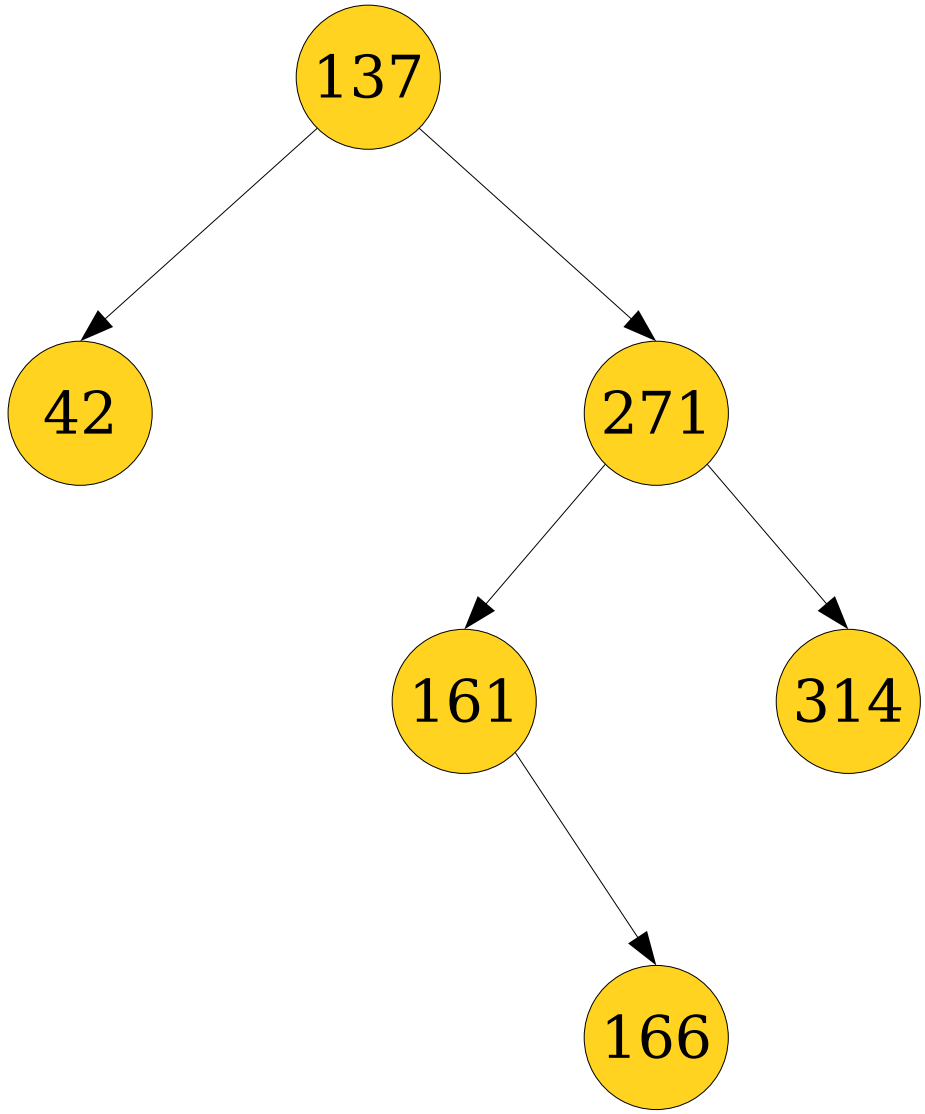
Deleting from a BST



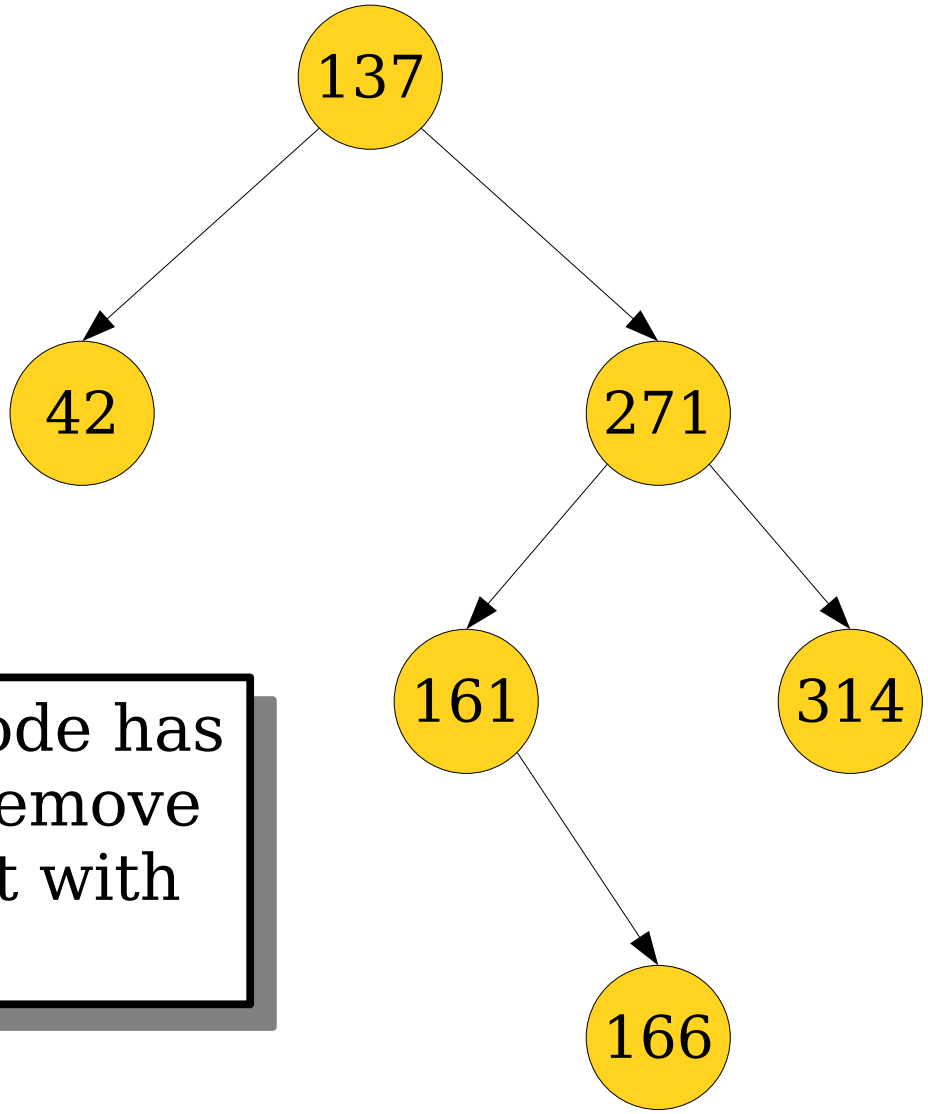
42



Deleting from a BST

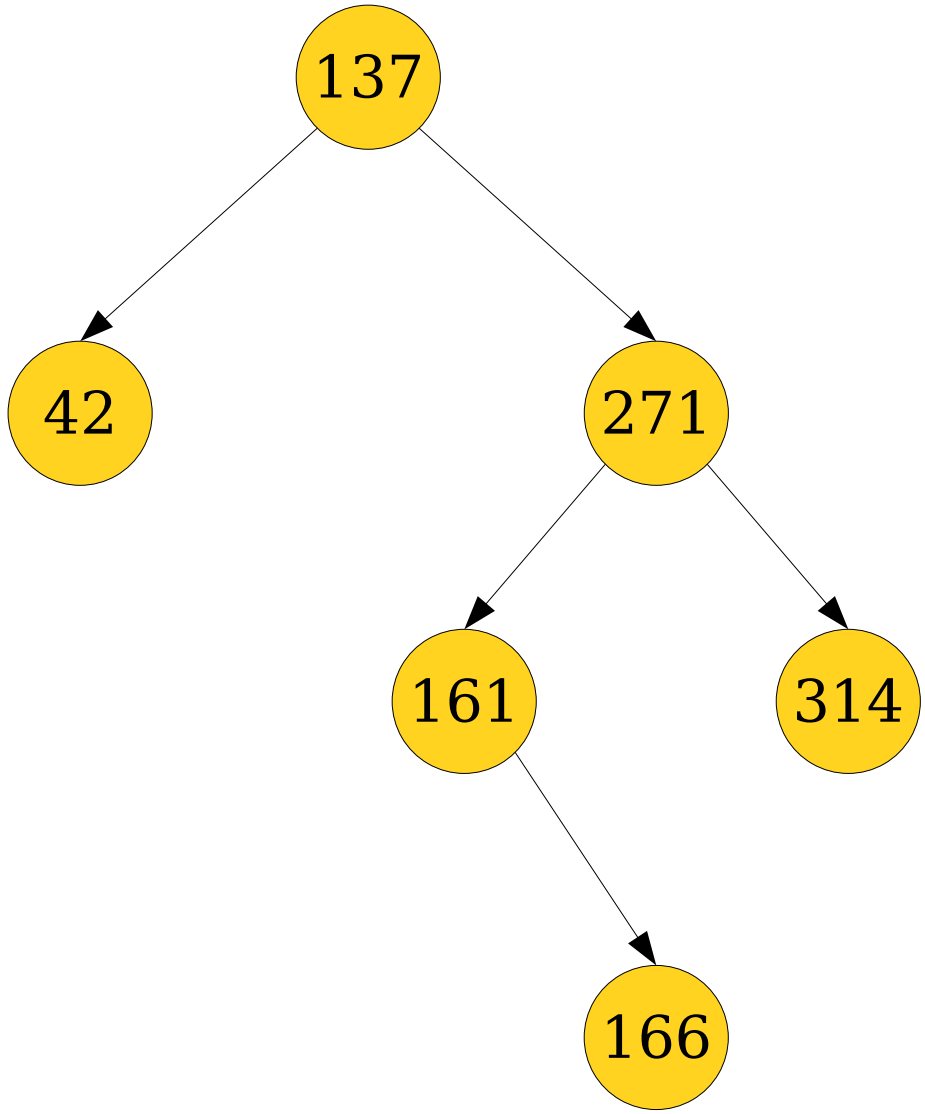


Deleting from a BST

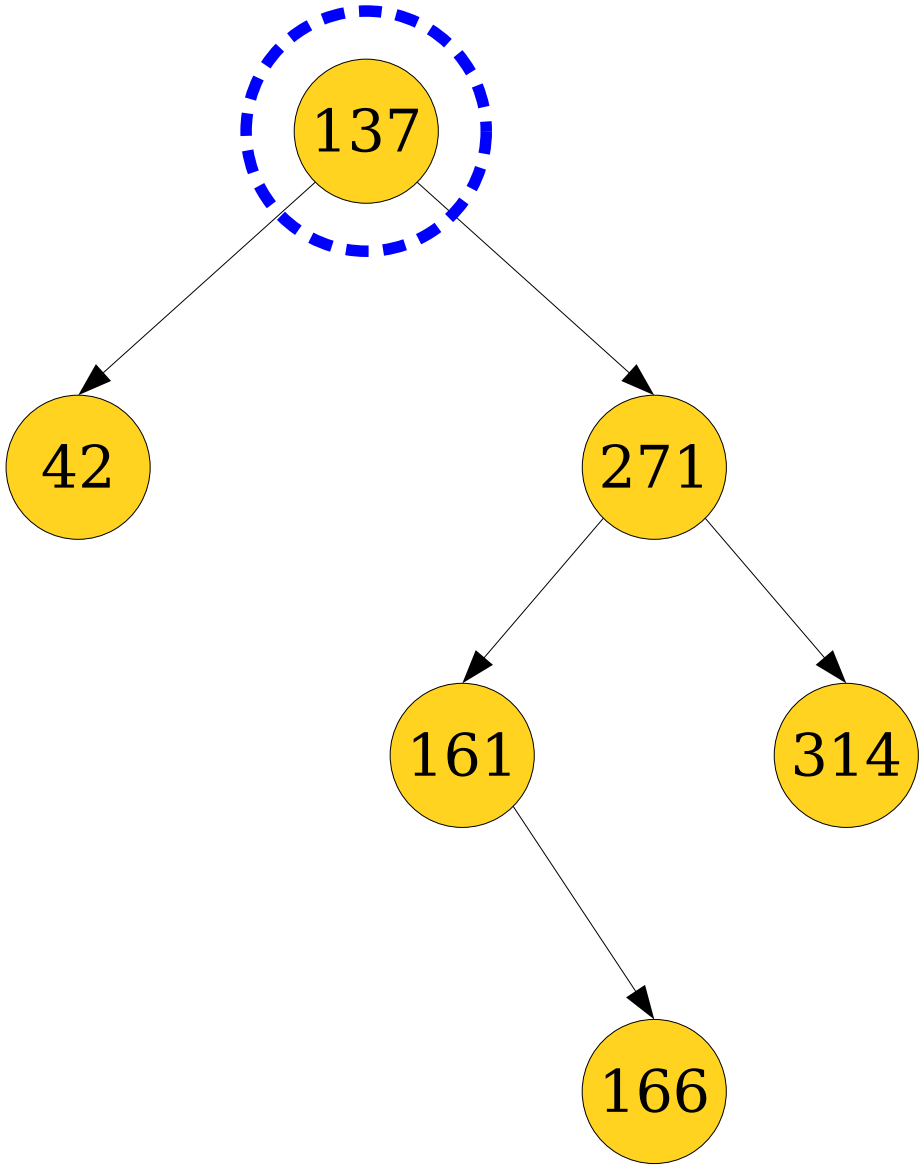


Case 1: If the node has just one child, remove it and replace it with its child.

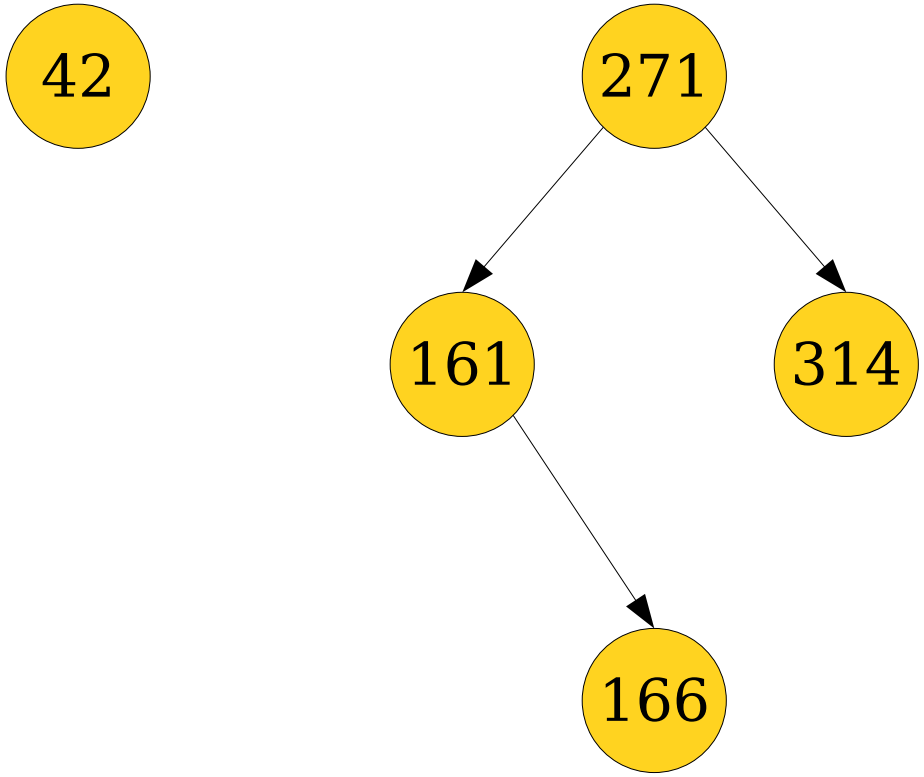
Deleting from a BST



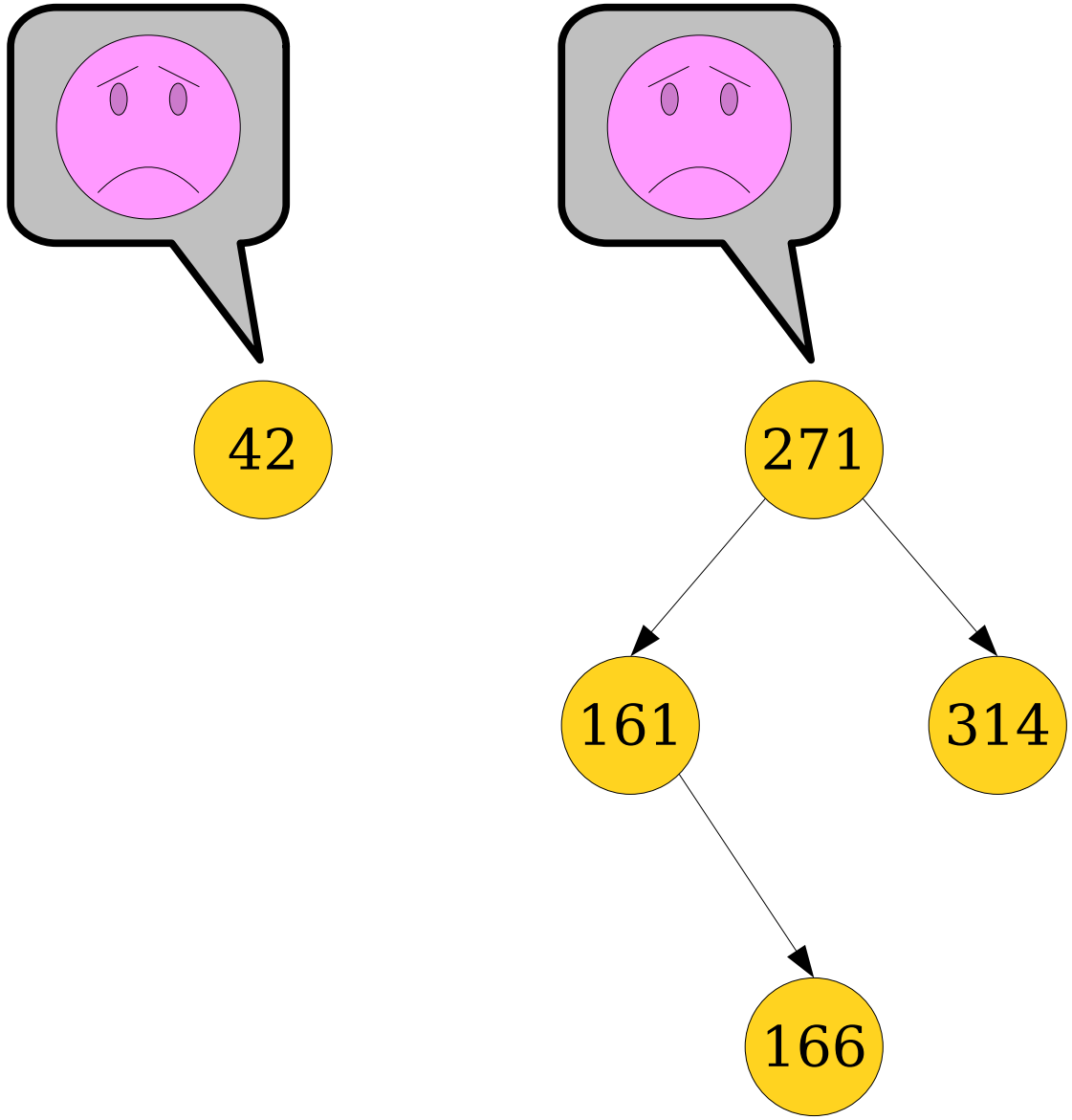
Deleting from a BST



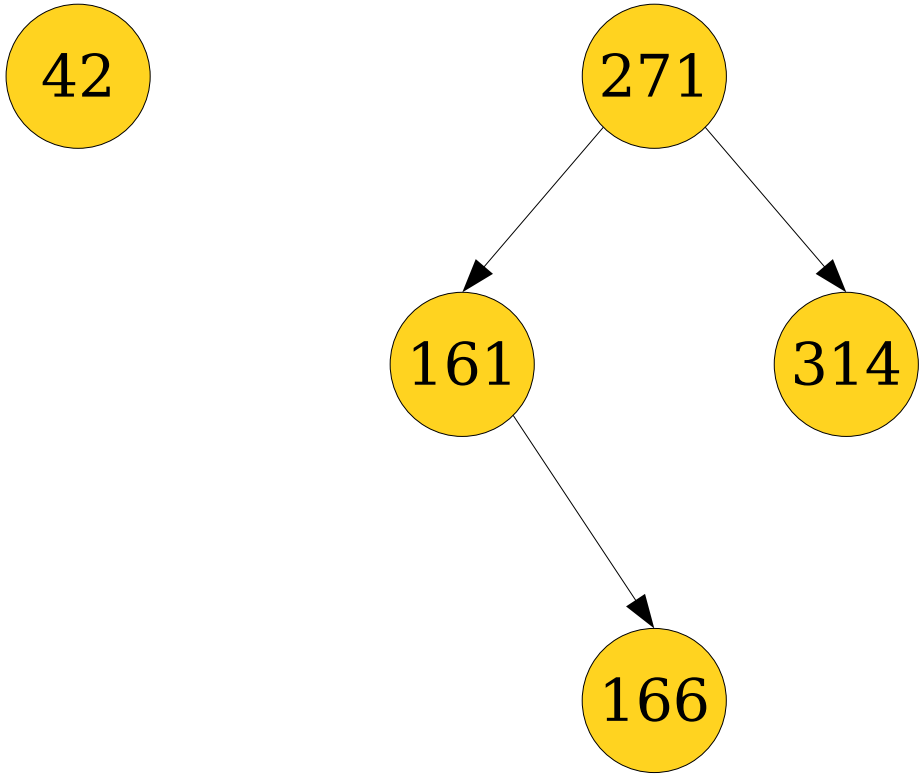
Deleting from a BST



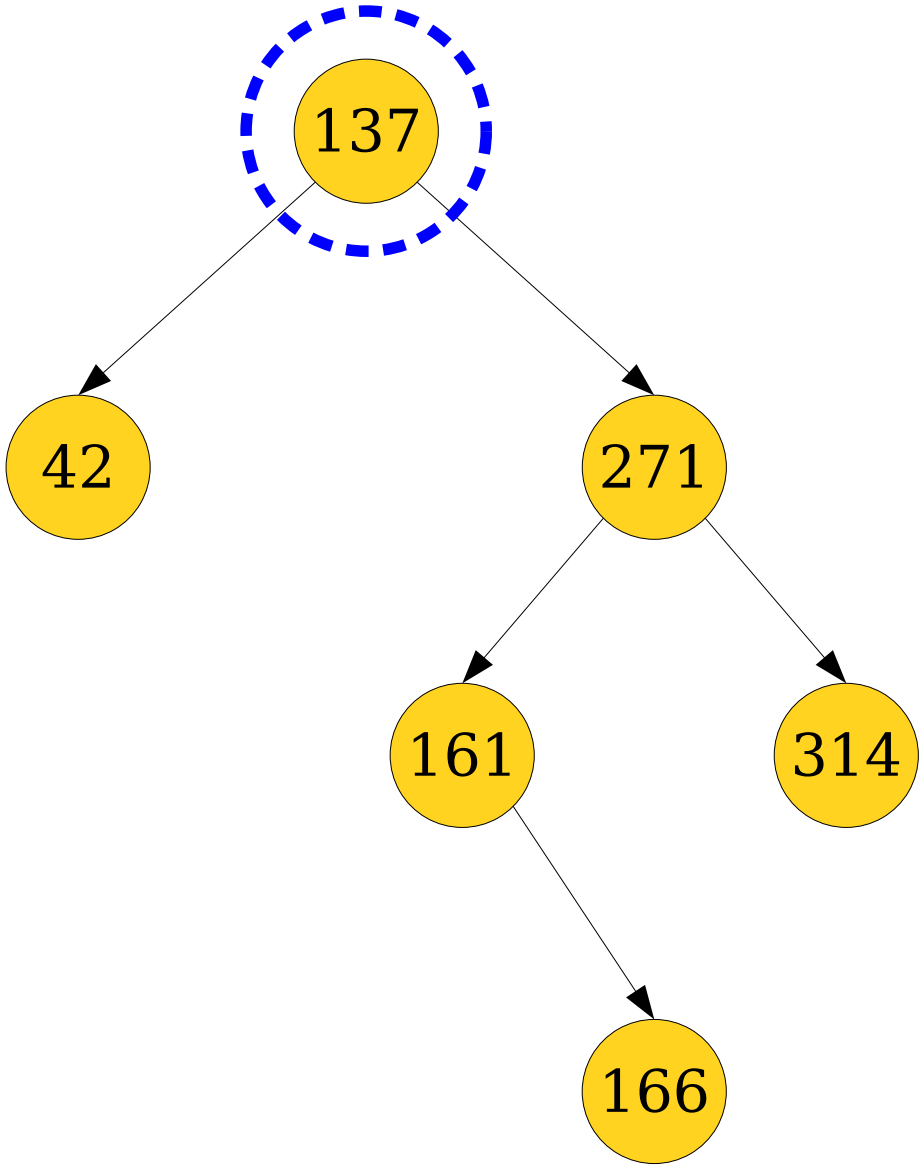
Deleting from a BST



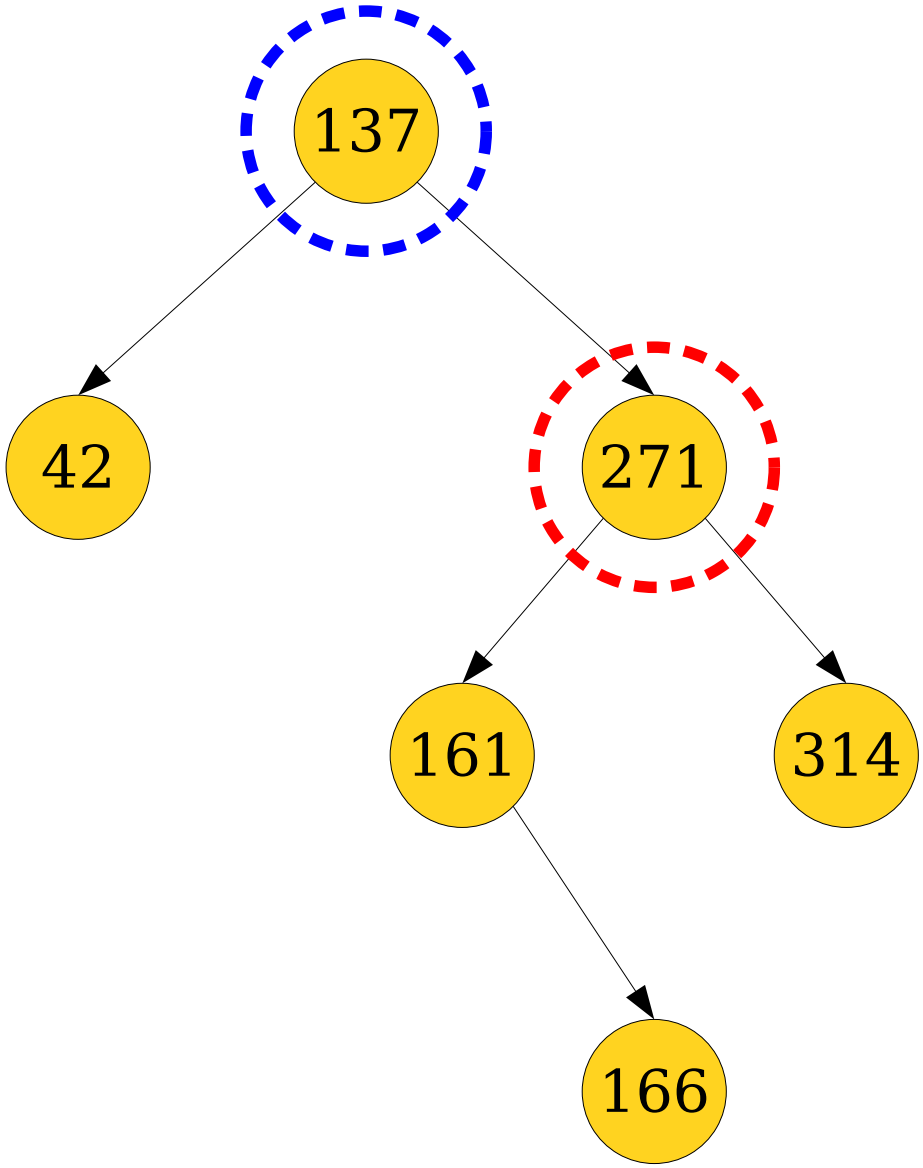
Deleting from a BST



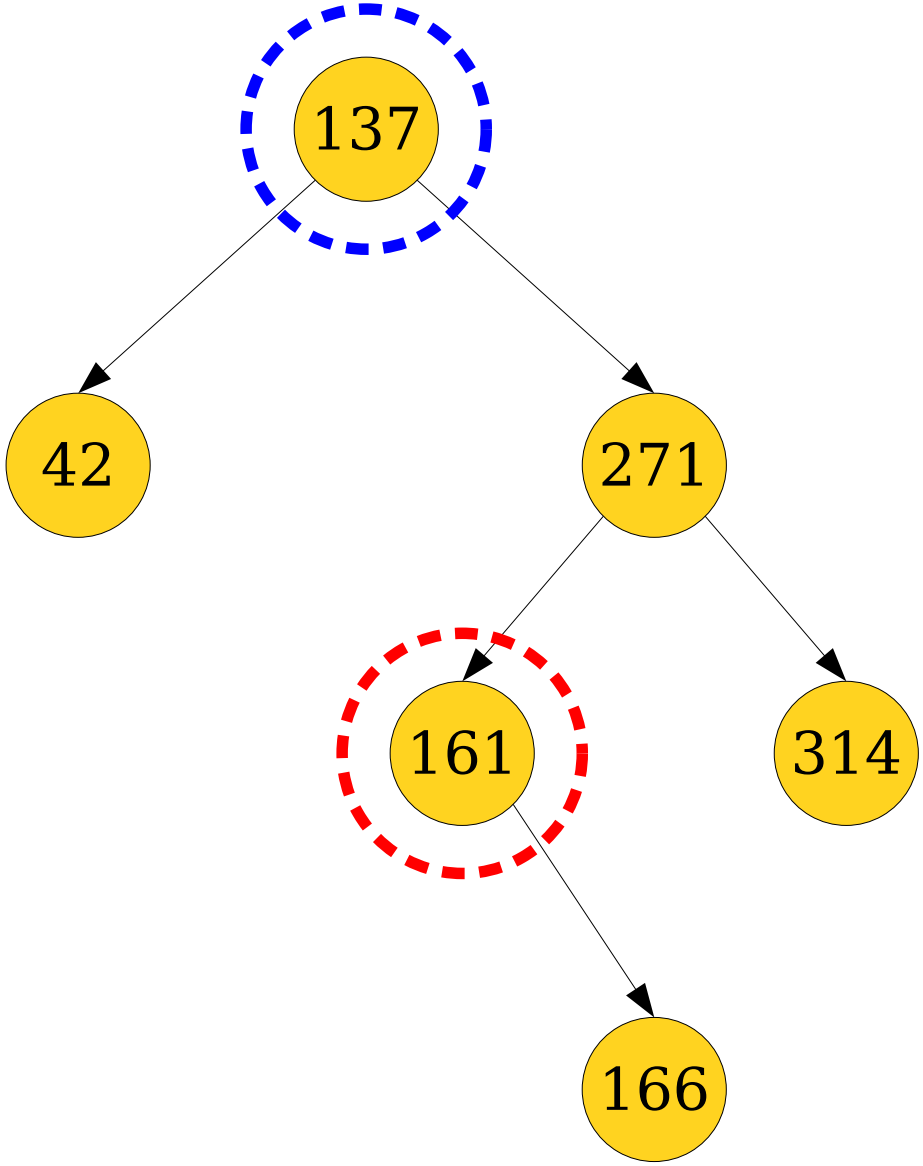
Deleting from a BST



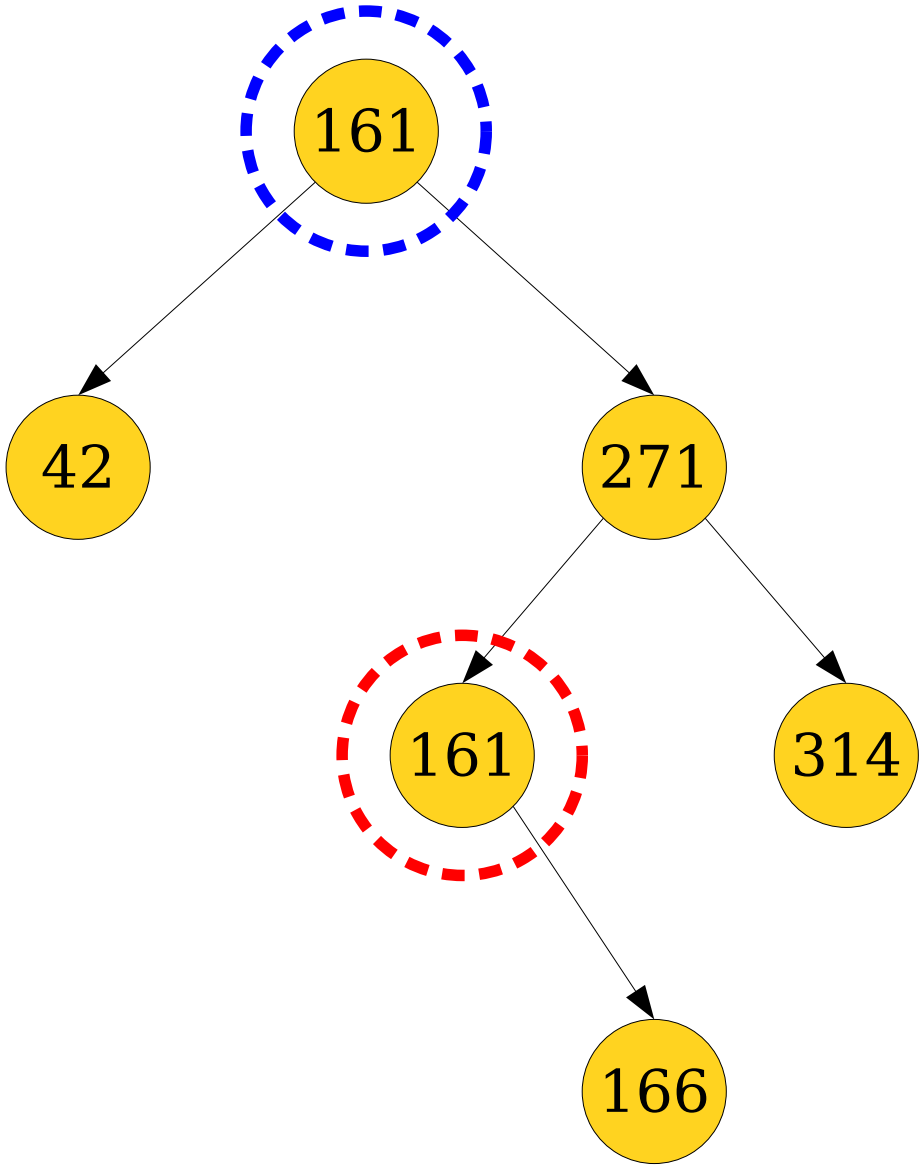
Deleting from a BST



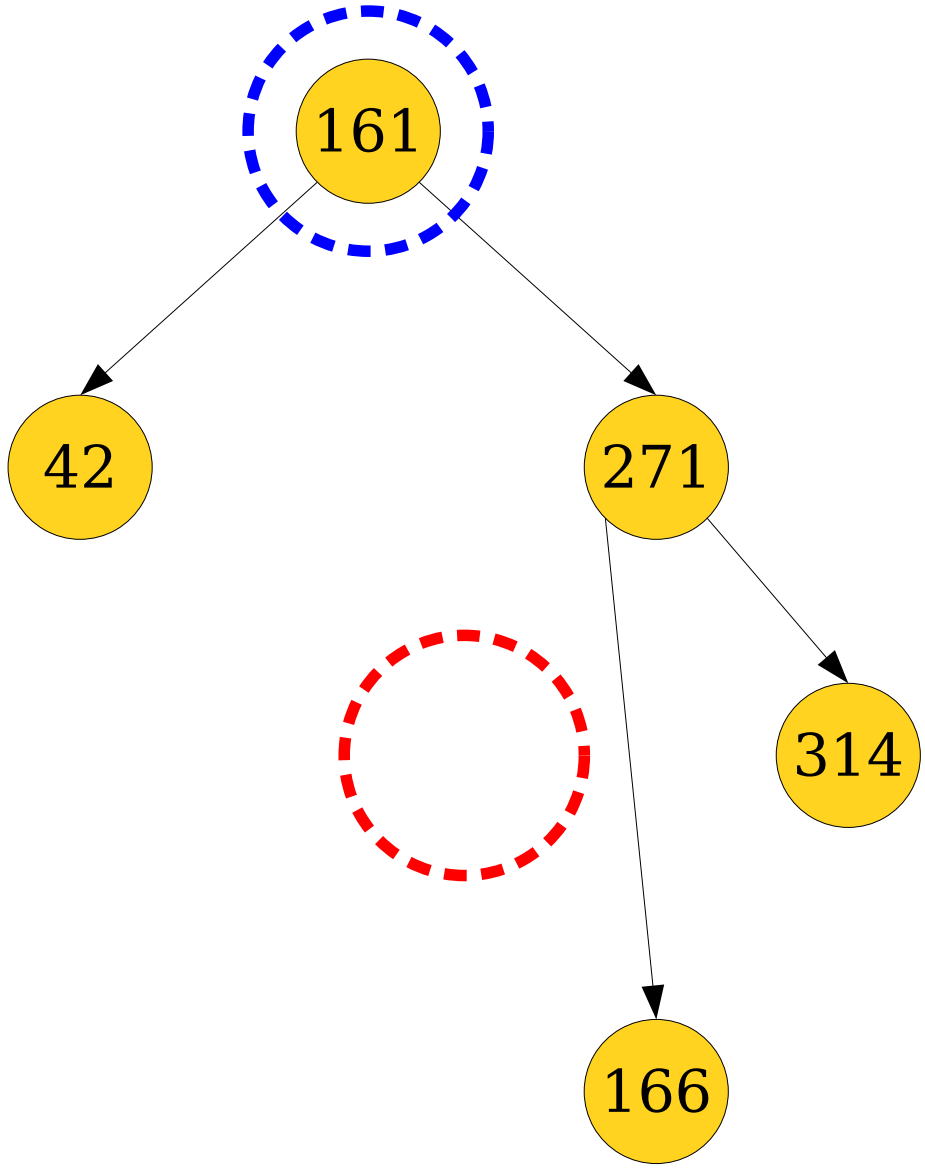
Deleting from a BST



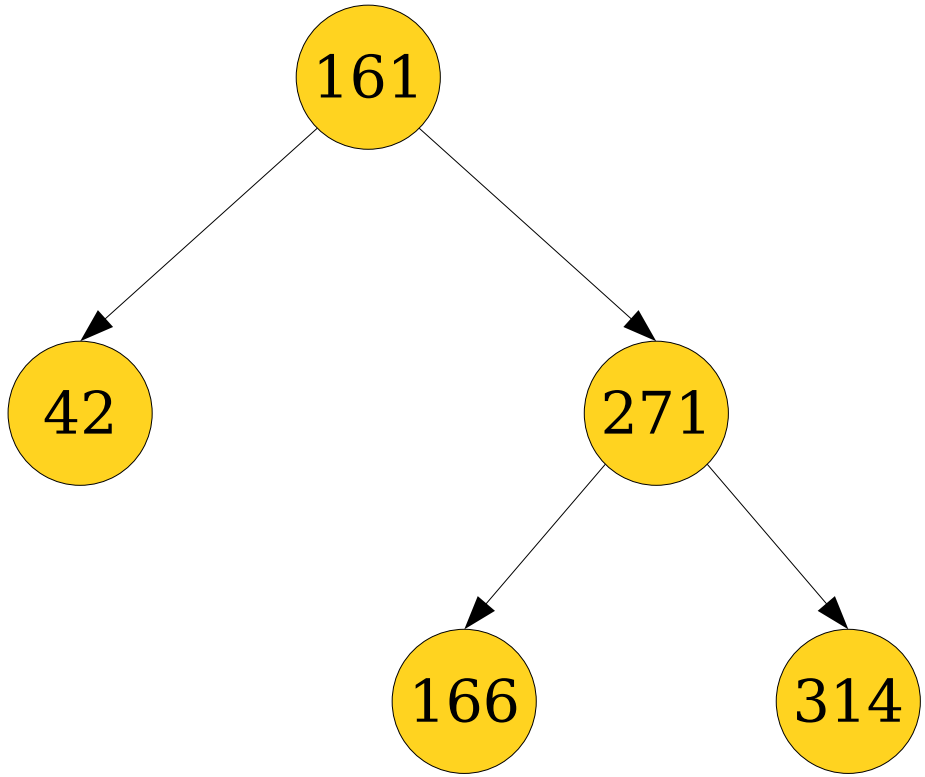
Deleting from a BST



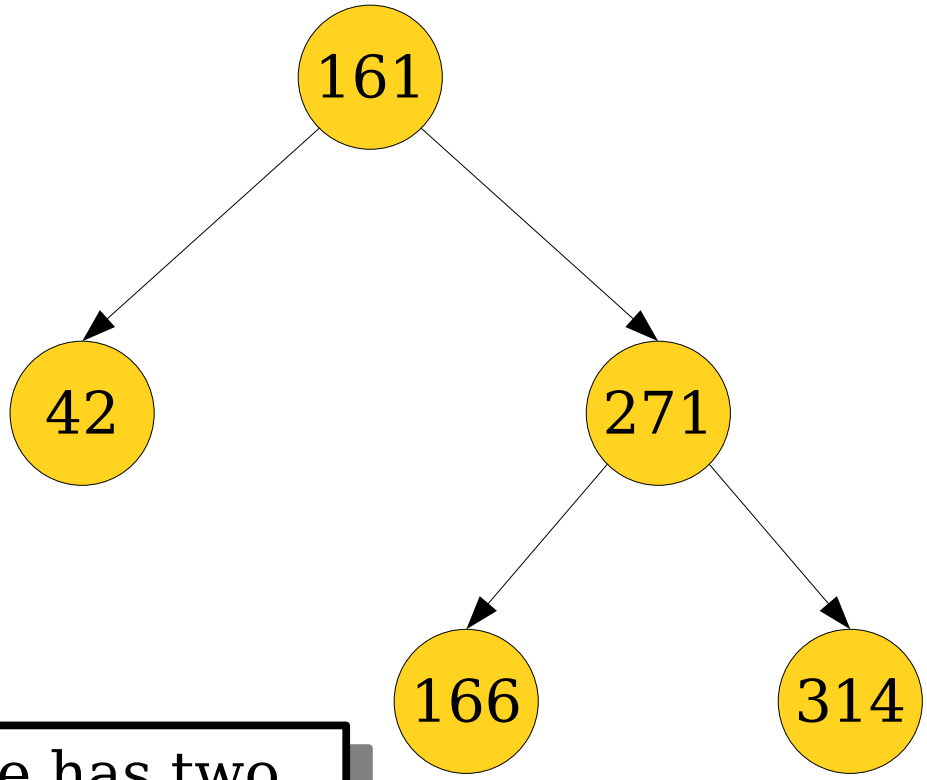
Deleting from a BST



Deleting from a BST



Deleting from a BST



Case 2: If the node has two children, find its inorder successor (which has zero or one child), replace the node's key with its successor's key, then delete its successor.

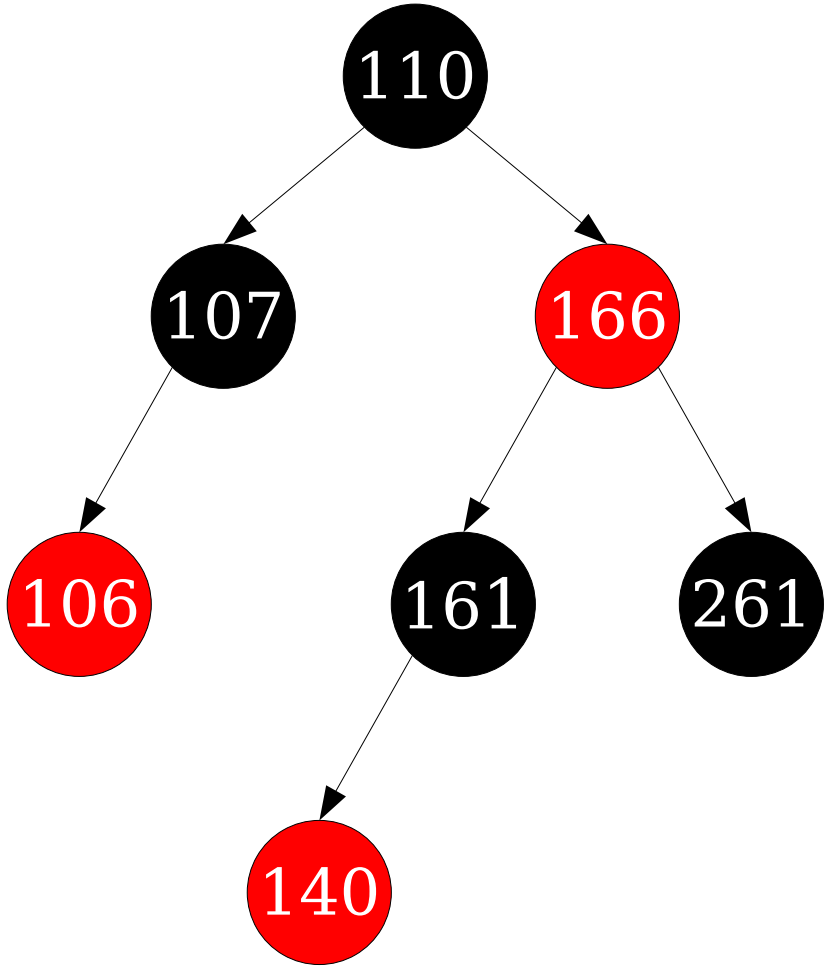
Runtime Analysis

- The time complexity of all these operations is $O(h)$, where h is the height of the tree.
  - That's the longest path we can take.
- In the best case, $h = O(\log n)$ and all operations take time $O(\log n)$.
- In the worst case, $h = \Theta(n)$ and some operations will take time $\Theta(n)$.
- ***Challenge:*** How do you efficiently keep the height of a tree low?

A Glimpse of Red/Black Trees

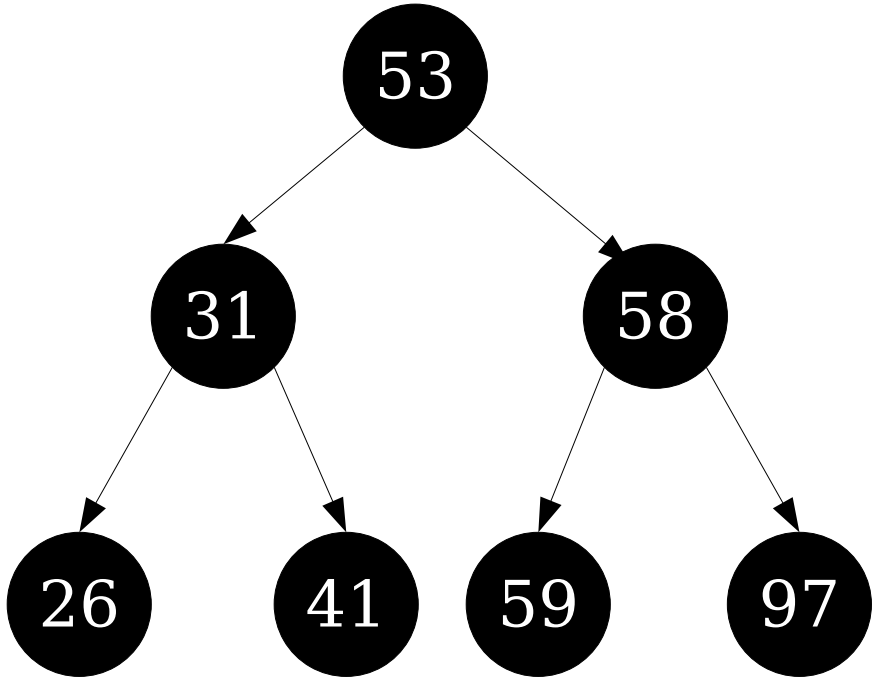
Red/Black Trees

- 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.



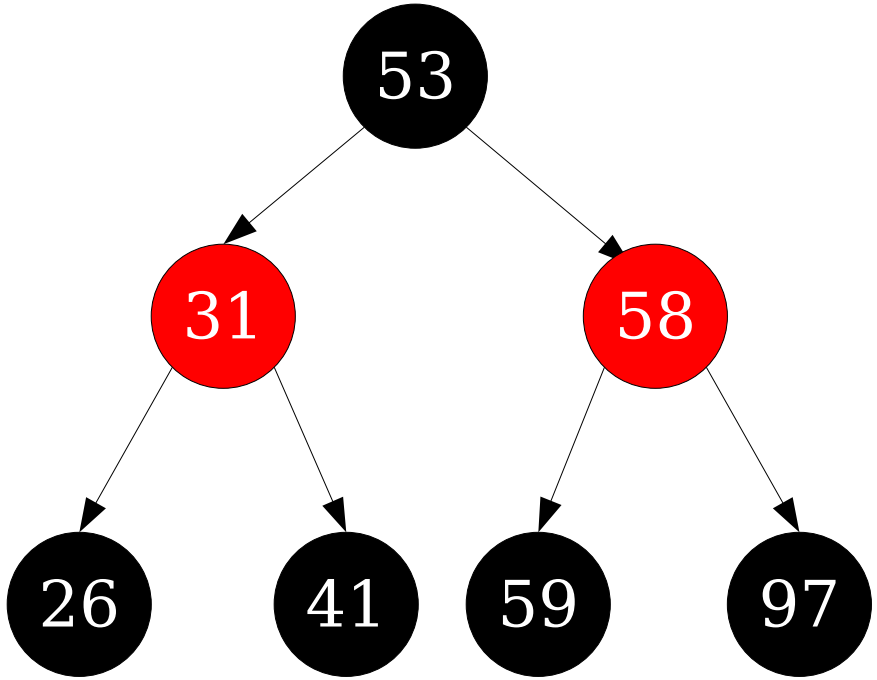
Red/Black Trees

- 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.



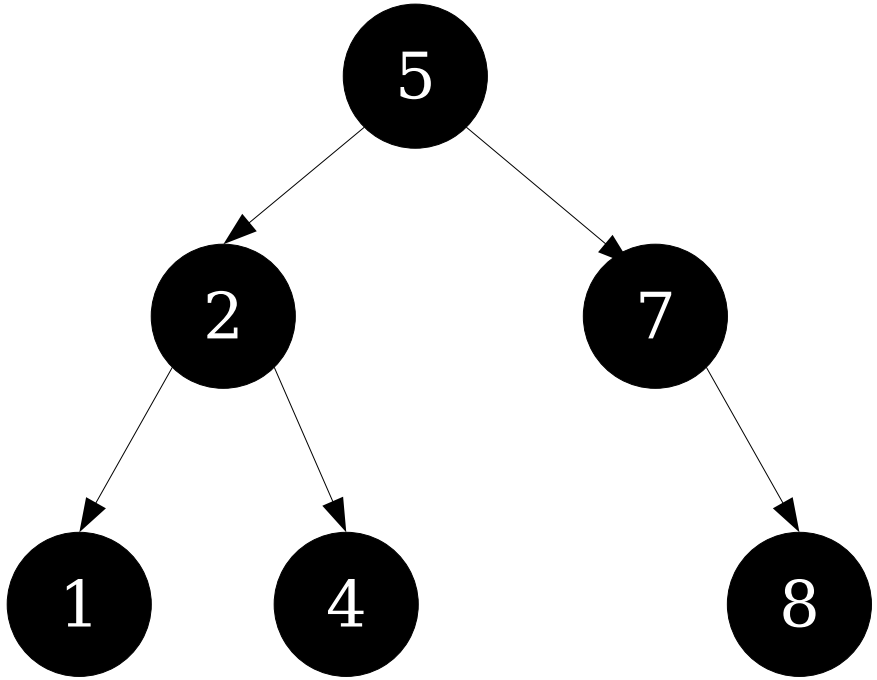
Red/Black Trees

- 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.



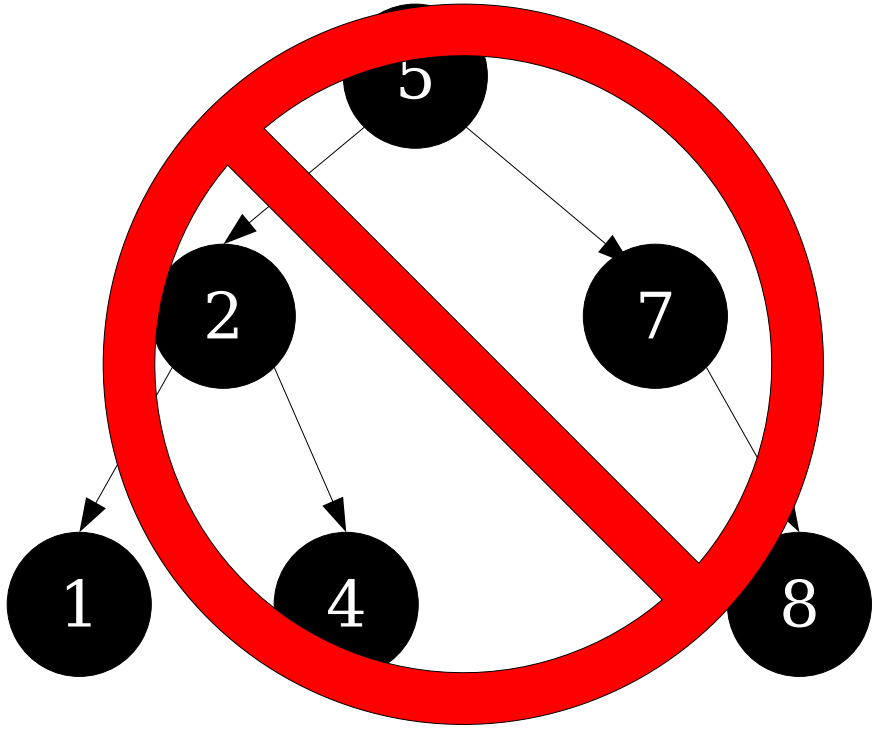
Red/Black Trees

- 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.



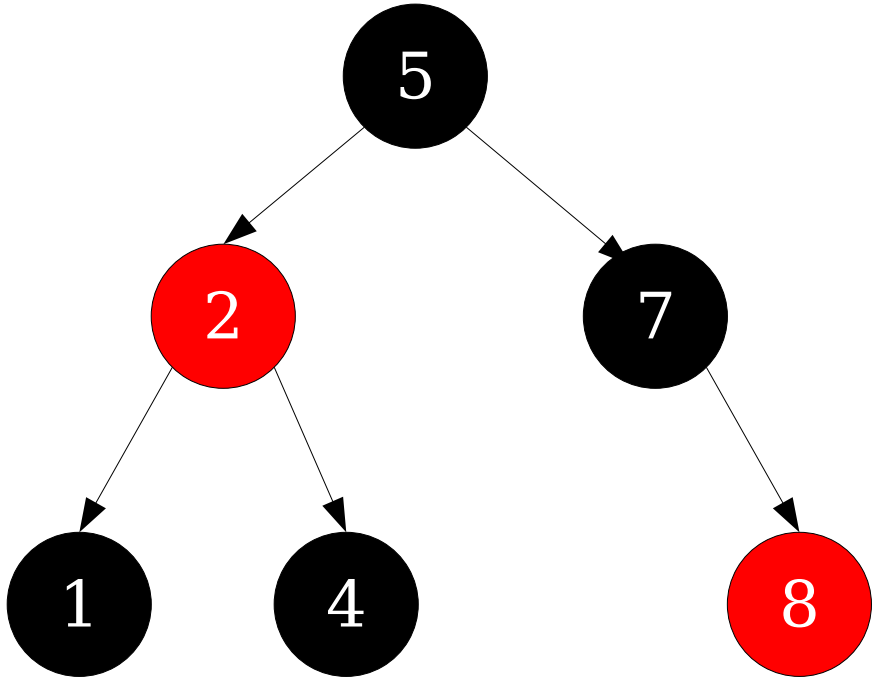
Red/Black Trees

- 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.



Red/Black Trees

- ***Theorem:*** Any red/black tree with n nodes has height O(log n).
  - We could prove this now, but there's a *much* simpler proof of this we'll see later on.
- Given a fixed red/black tree, lookups can be done in time O(log n).

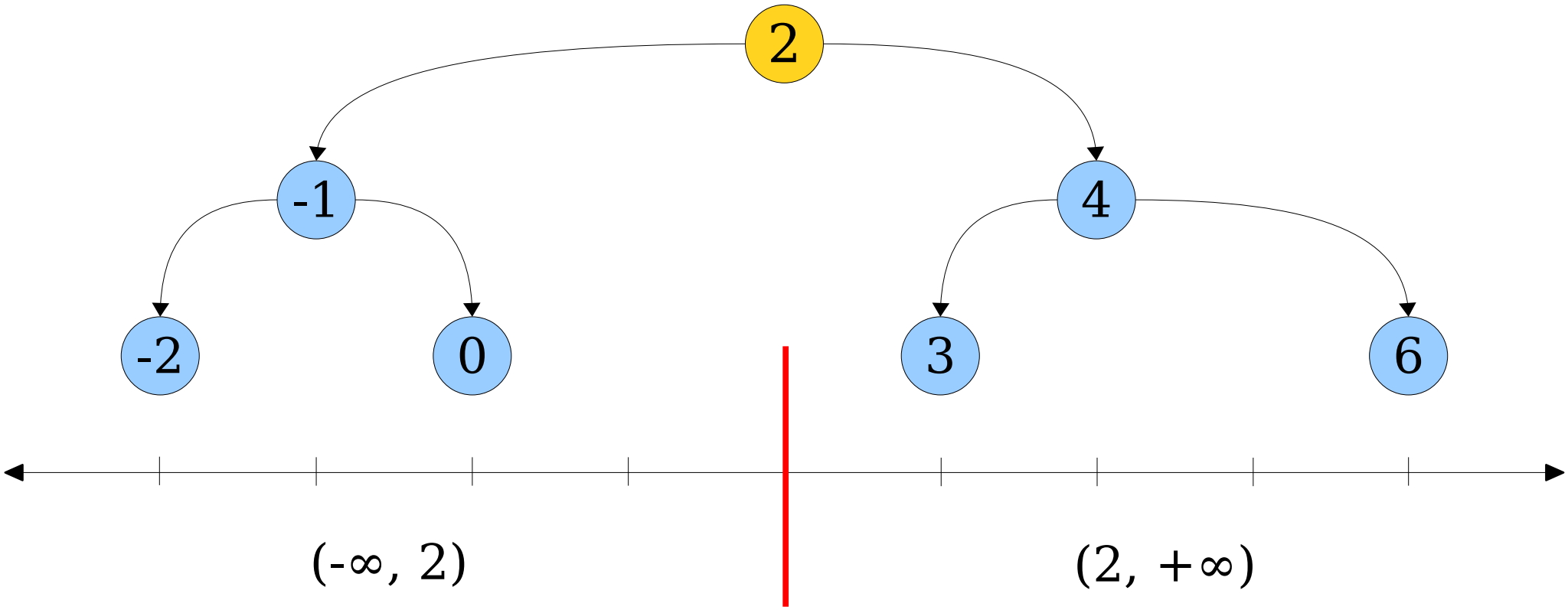
Fixing Up Red/Black Trees

- ***The Good News:*** After doing an insertion or deletion, we can locally modify a red/black tree in time O(log n) to fix up the red/black properties.
- ***The Bad News:*** There are a *lot* of cases to consider and they're not trivial.
- Some questions:
  - How do you memorize / remember all the rules for fixing up the tree?
  - How on earth did anyone come up with red/black trees in the first place?

B-Trees

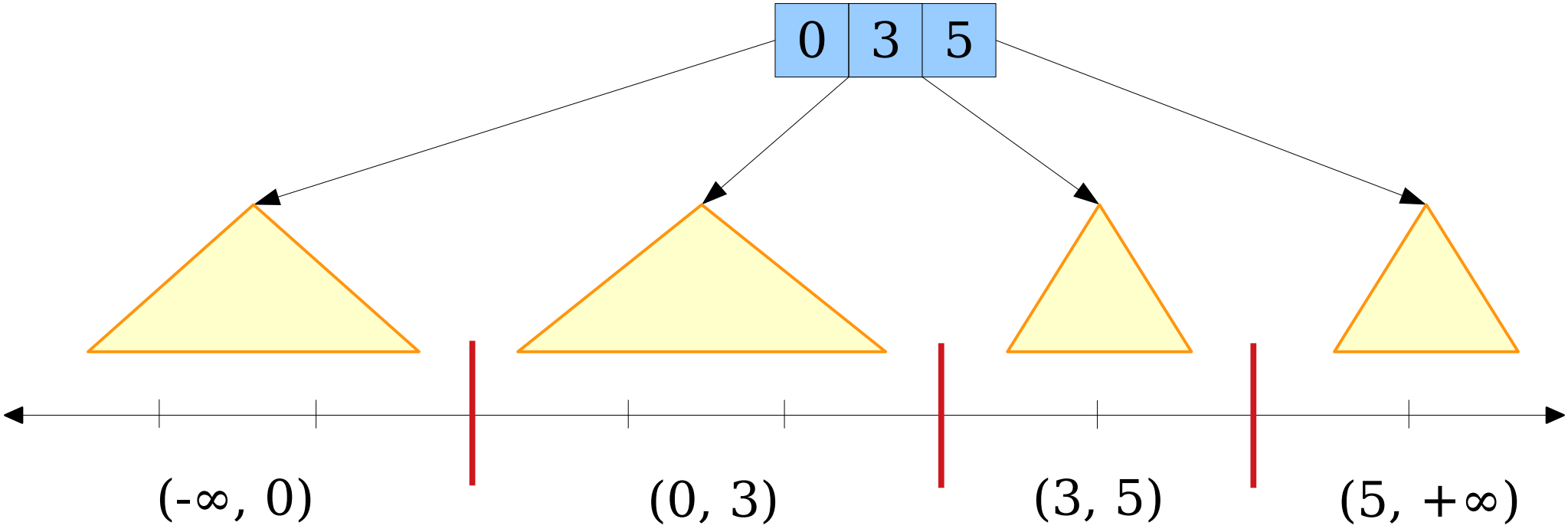
Generalizing BSTs

- In a binary search tree, each node stores a single key.
- That key splits the “key space” into two pieces, and each subtree stores the keys in those halves.



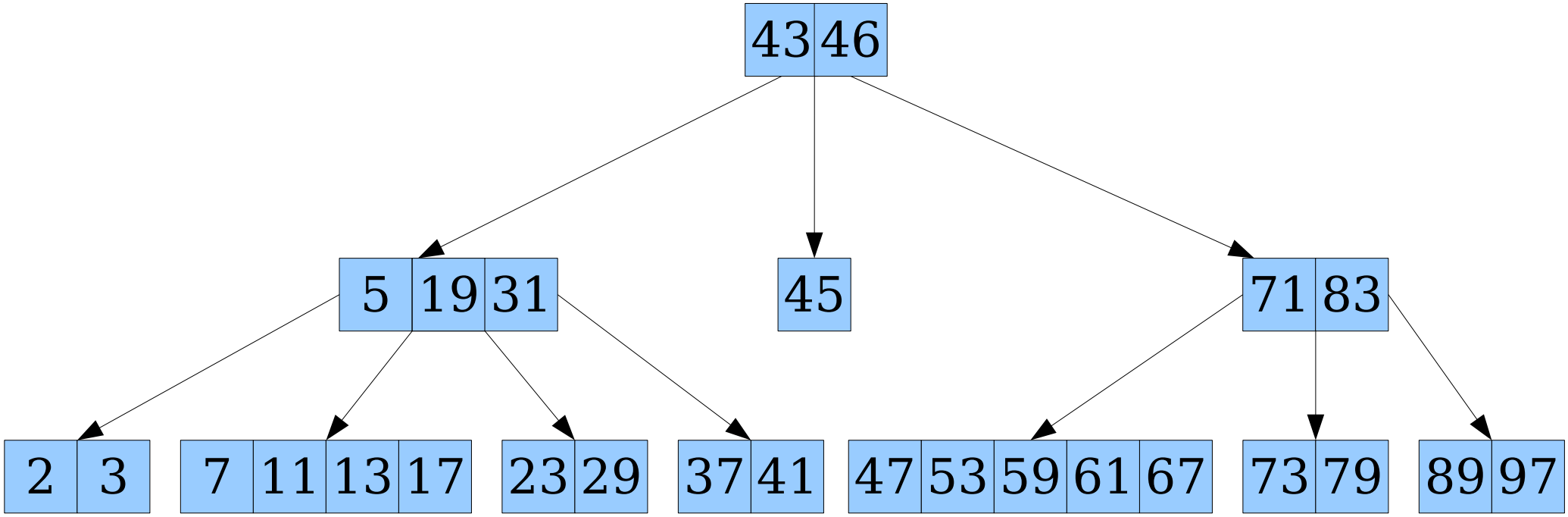
Generalizing BSTs

- In a ***multiway search tree***, each node stores an arbitrary number of keys in sorted order.
- A node with k keys splits the key space into $k+1$ regions, with subtrees for keys in each region.



Generalizing BSTs

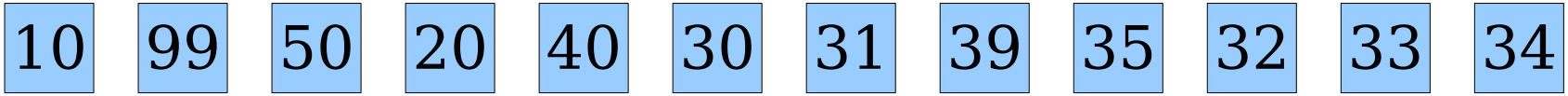
- In a ***multiway search tree***, each node stores an arbitrary number of keys in sorted order.



- Surprisingly, it's a bit easier to build a balanced multiway tree than it is to build a balanced BST. Let's see how.

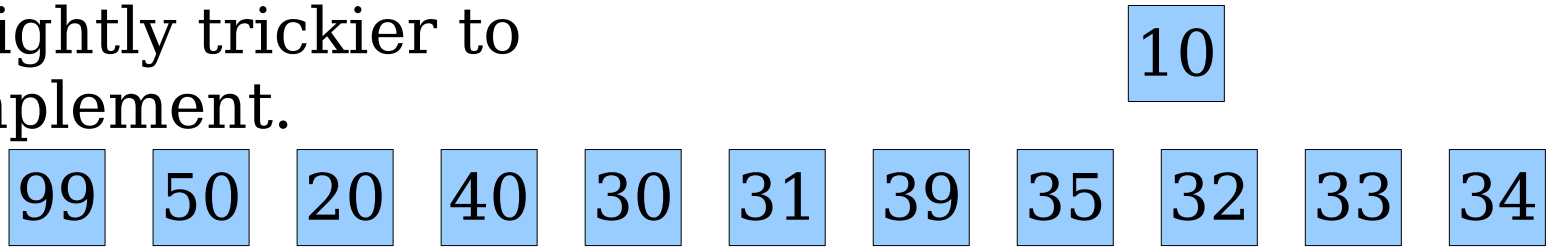
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



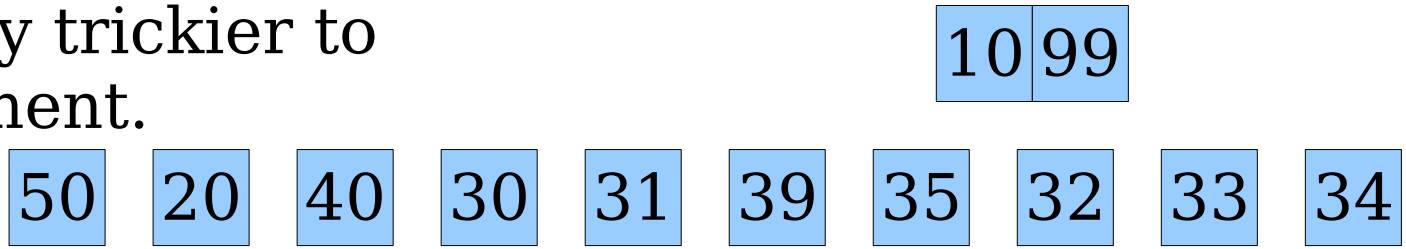
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



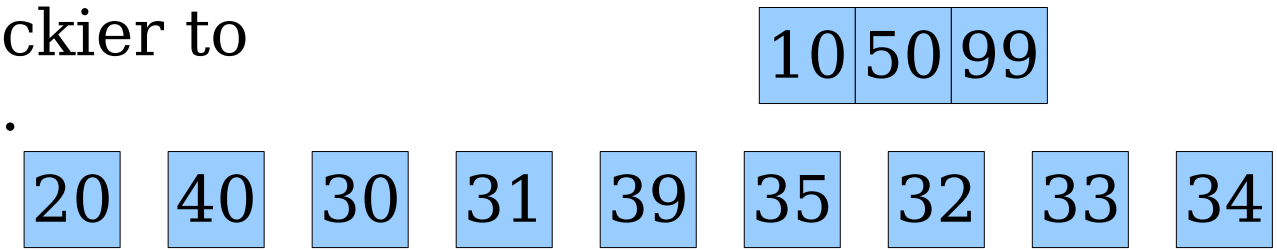
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



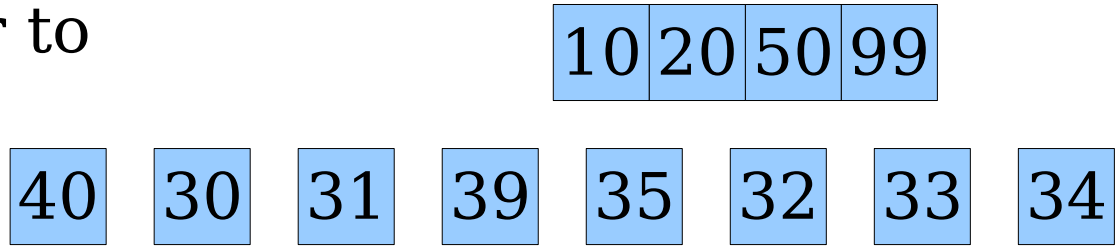
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



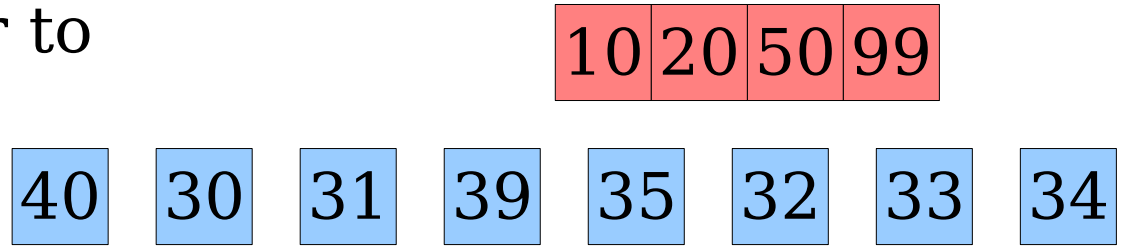
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



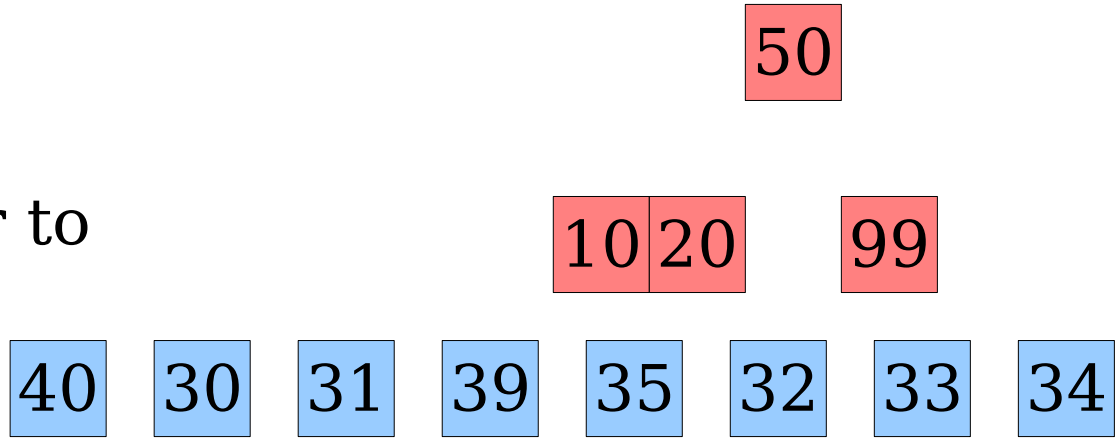
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



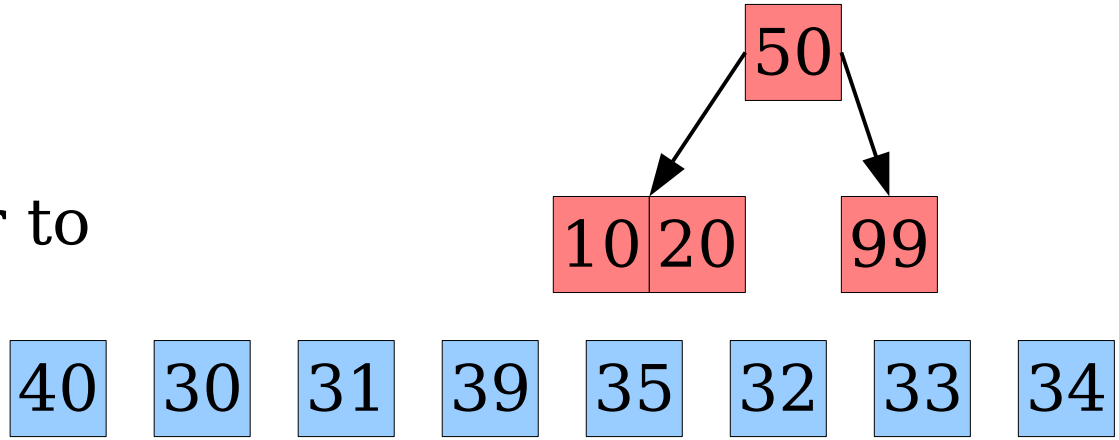
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



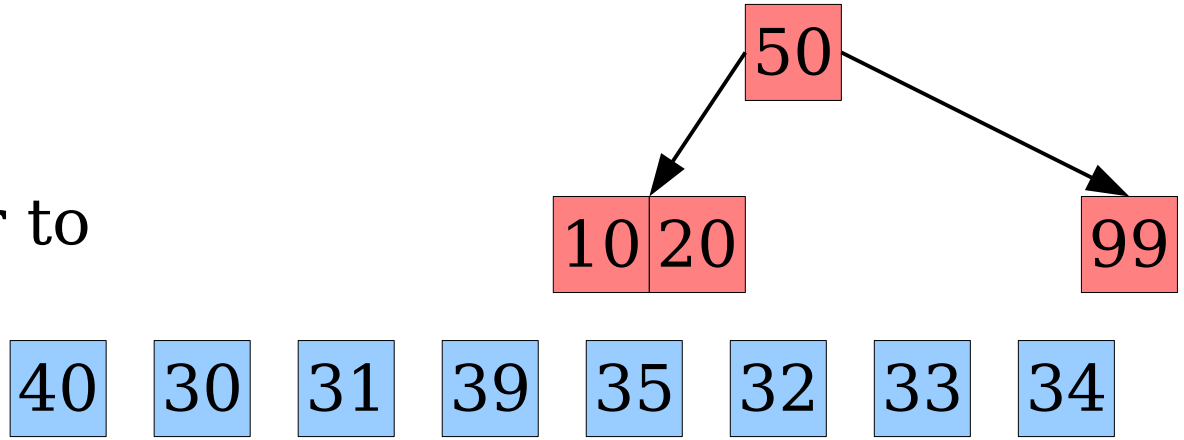
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



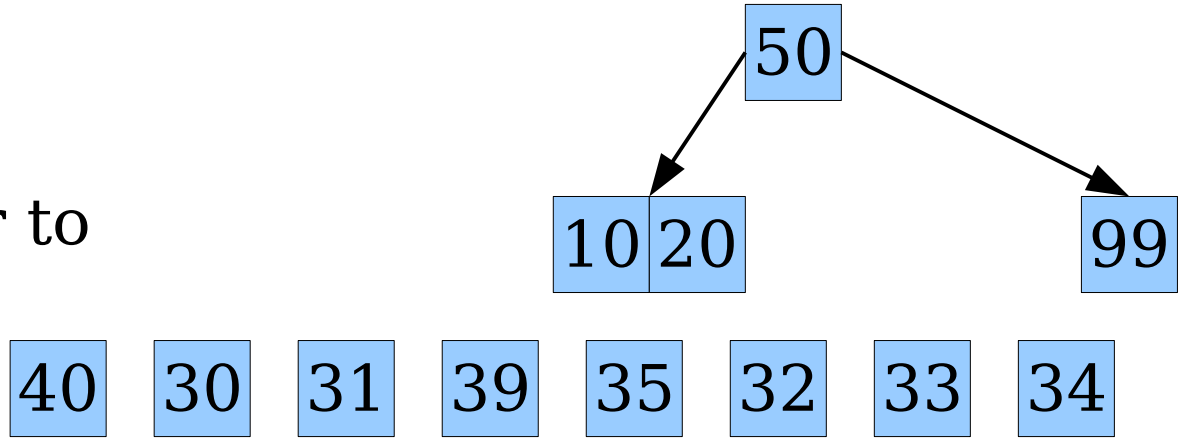
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



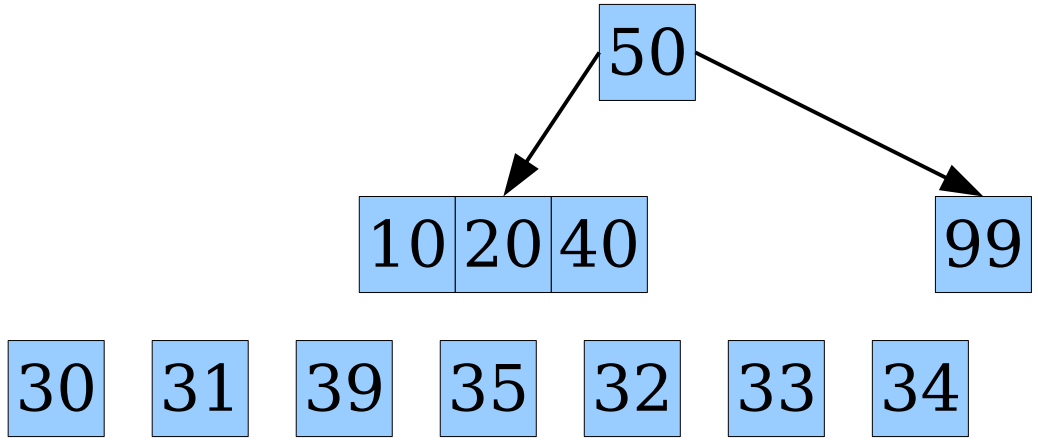
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



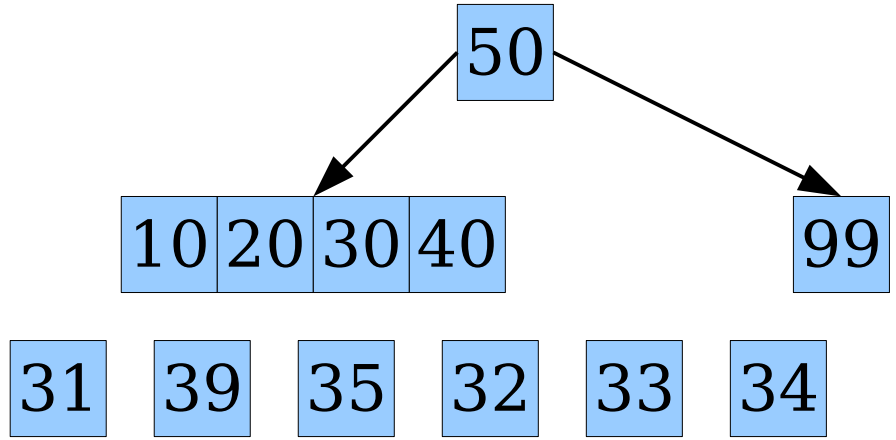
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



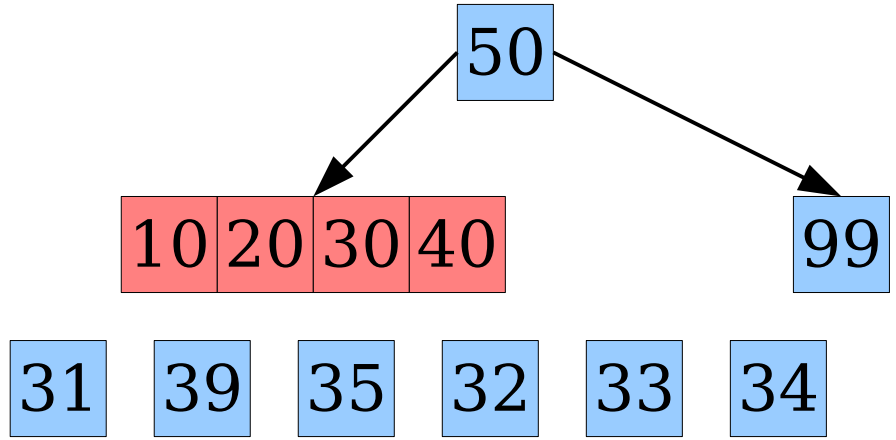
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



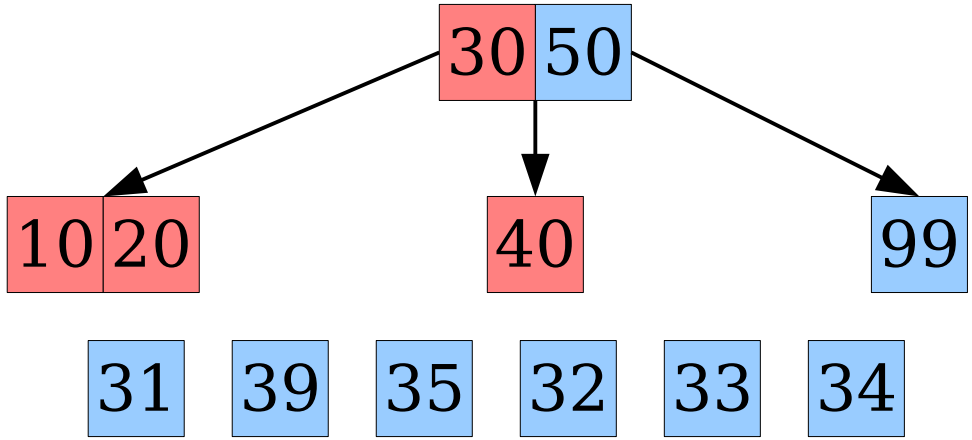
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



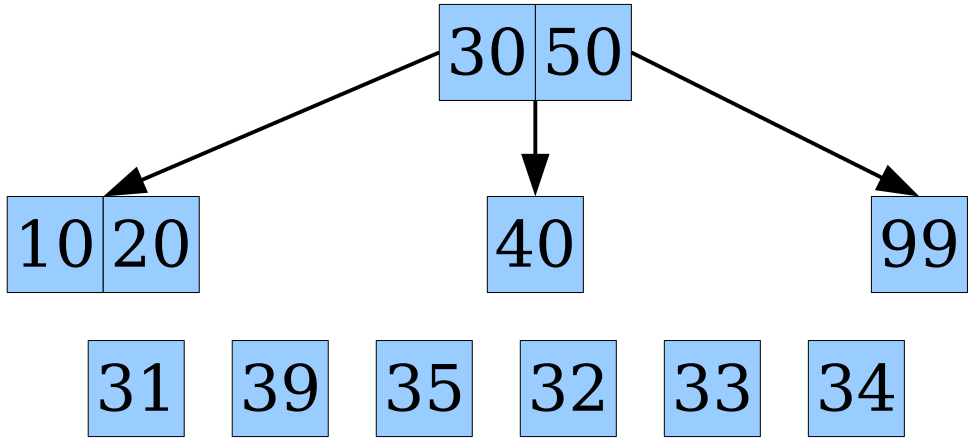
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



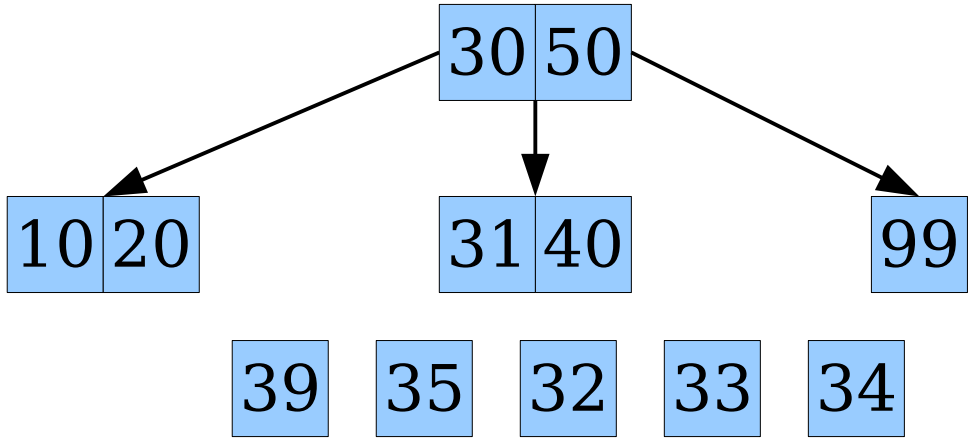
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



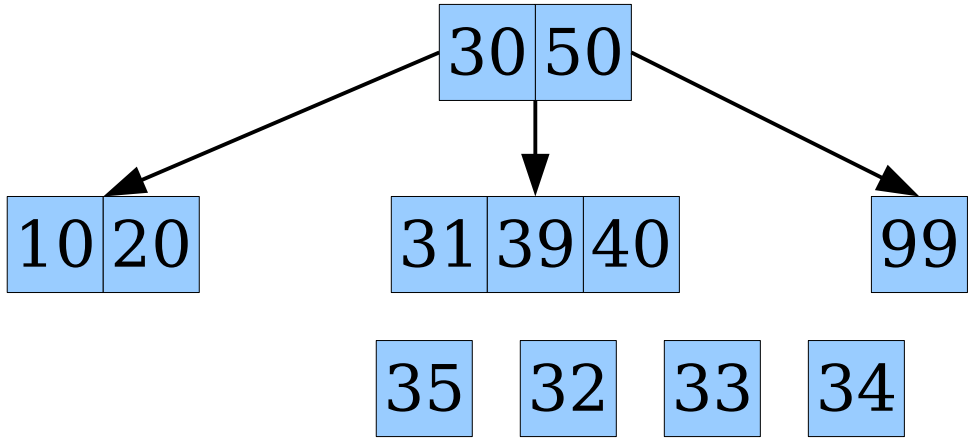
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



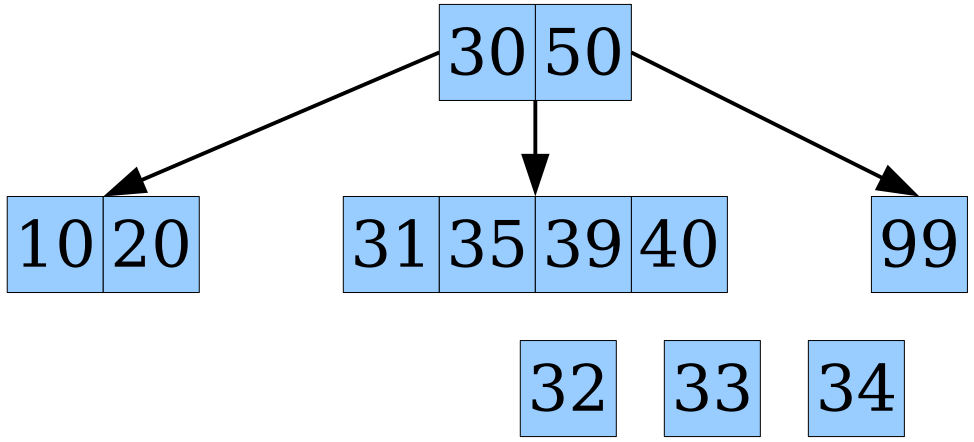
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



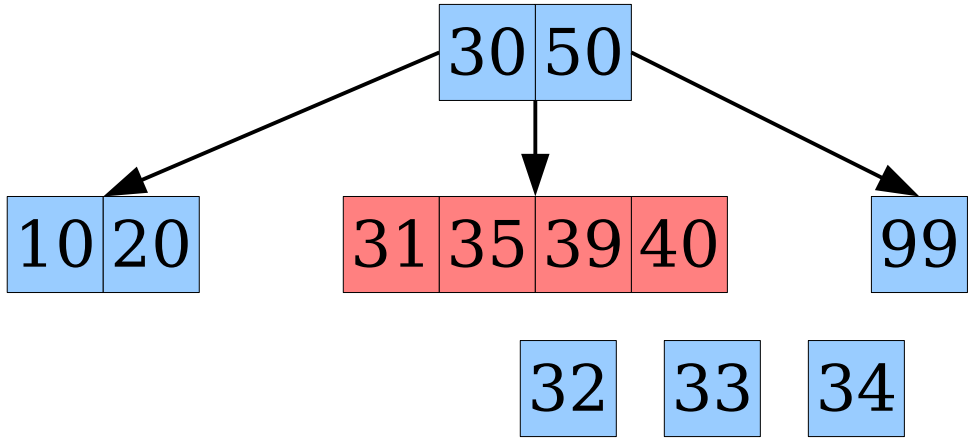
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



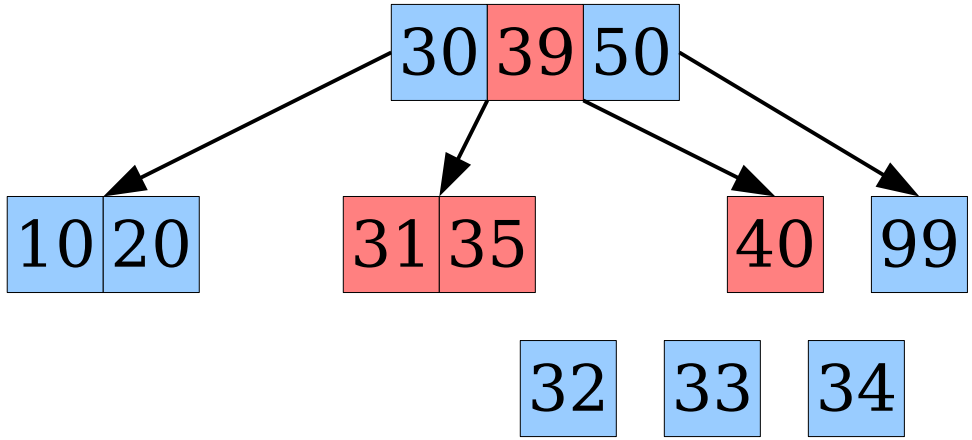
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



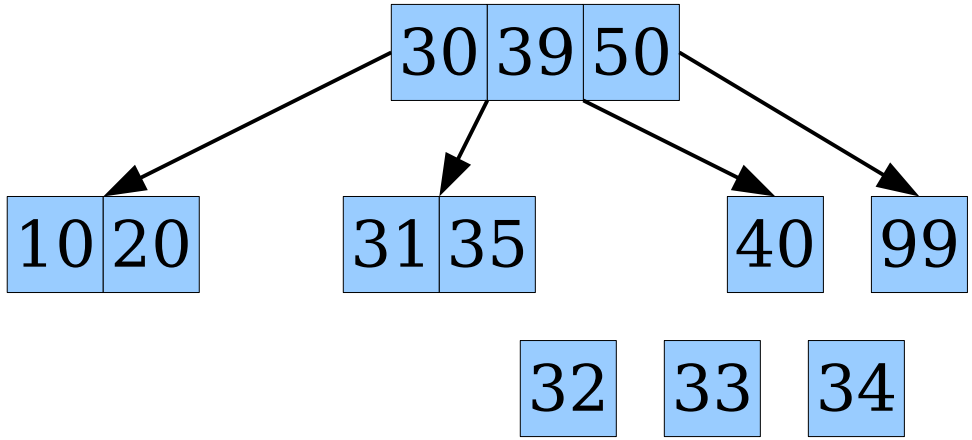
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



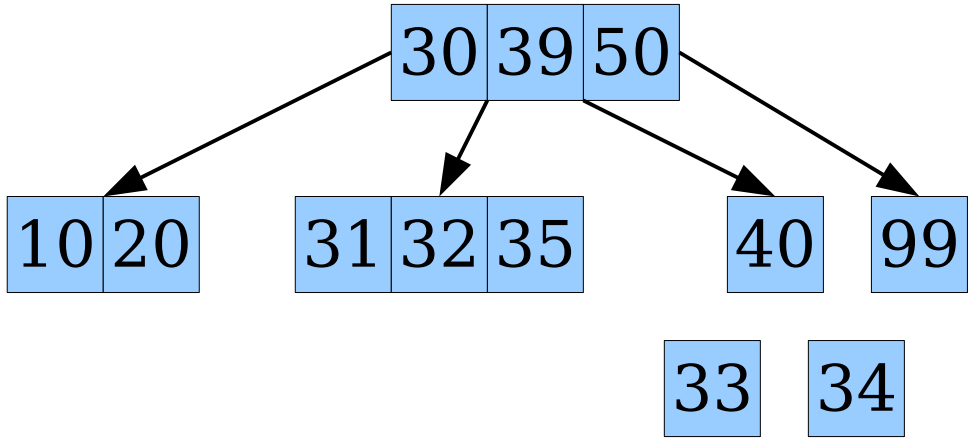
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



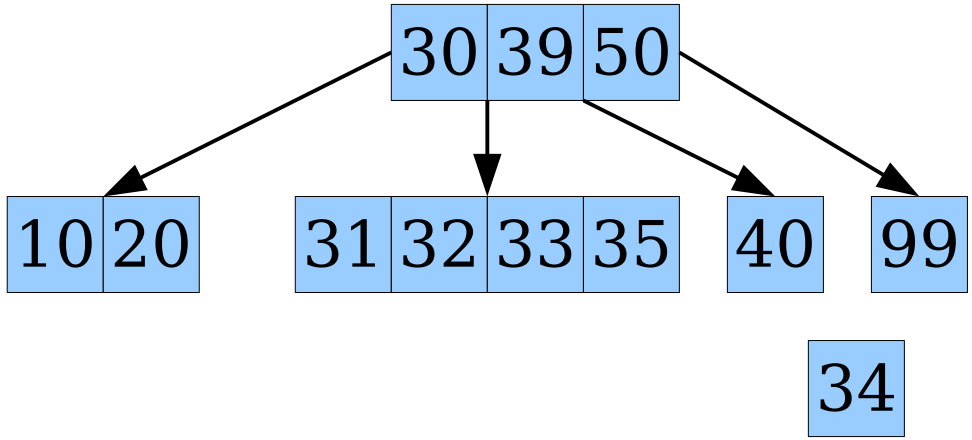
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



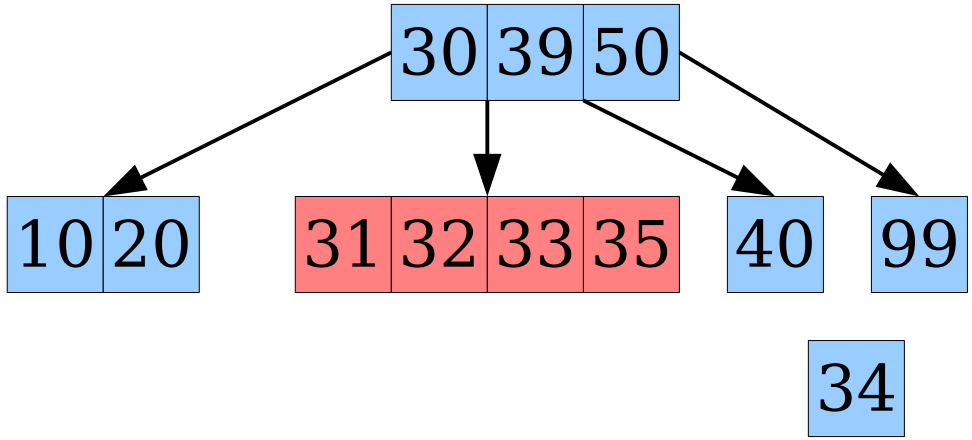
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



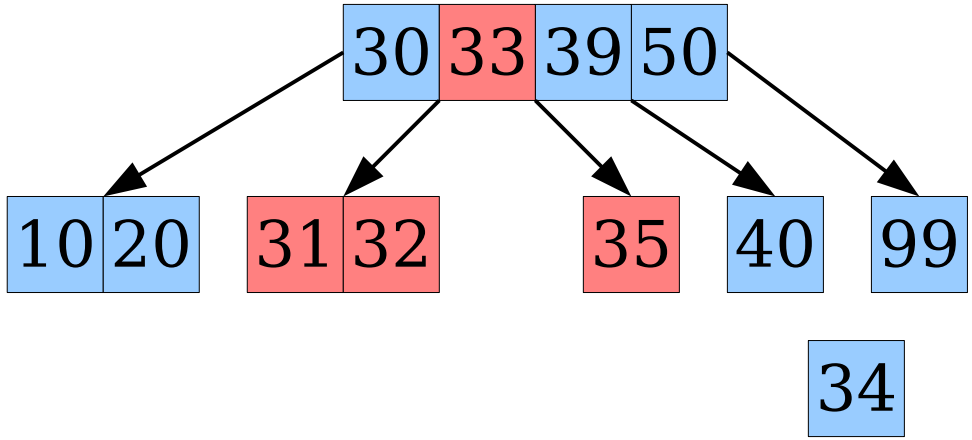
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



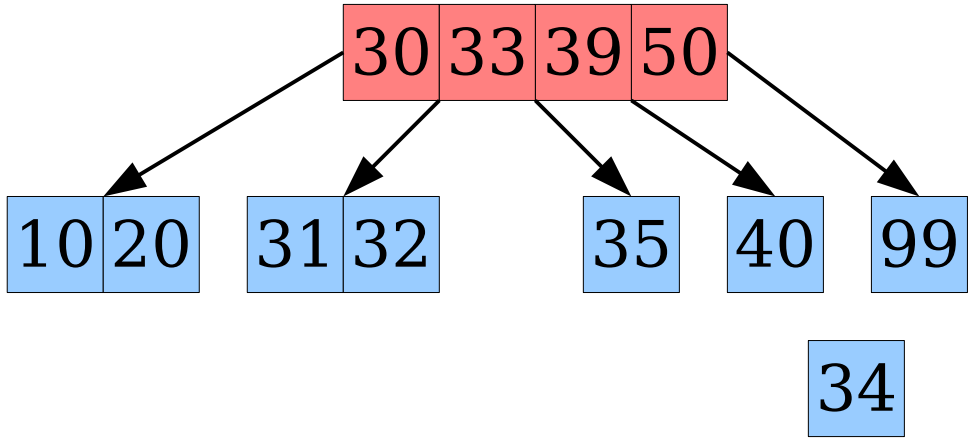
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



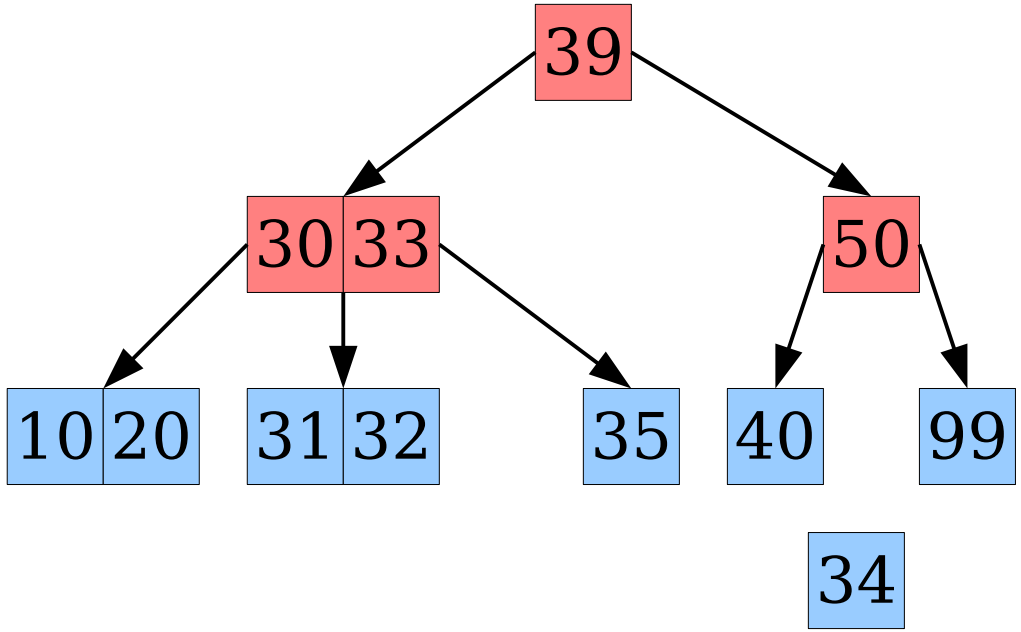
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



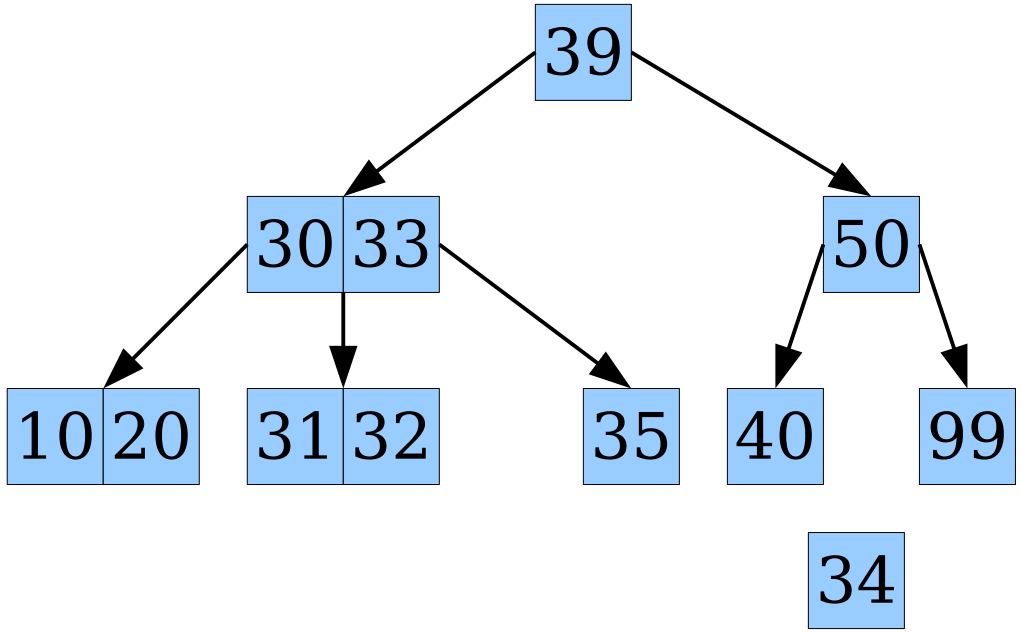
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



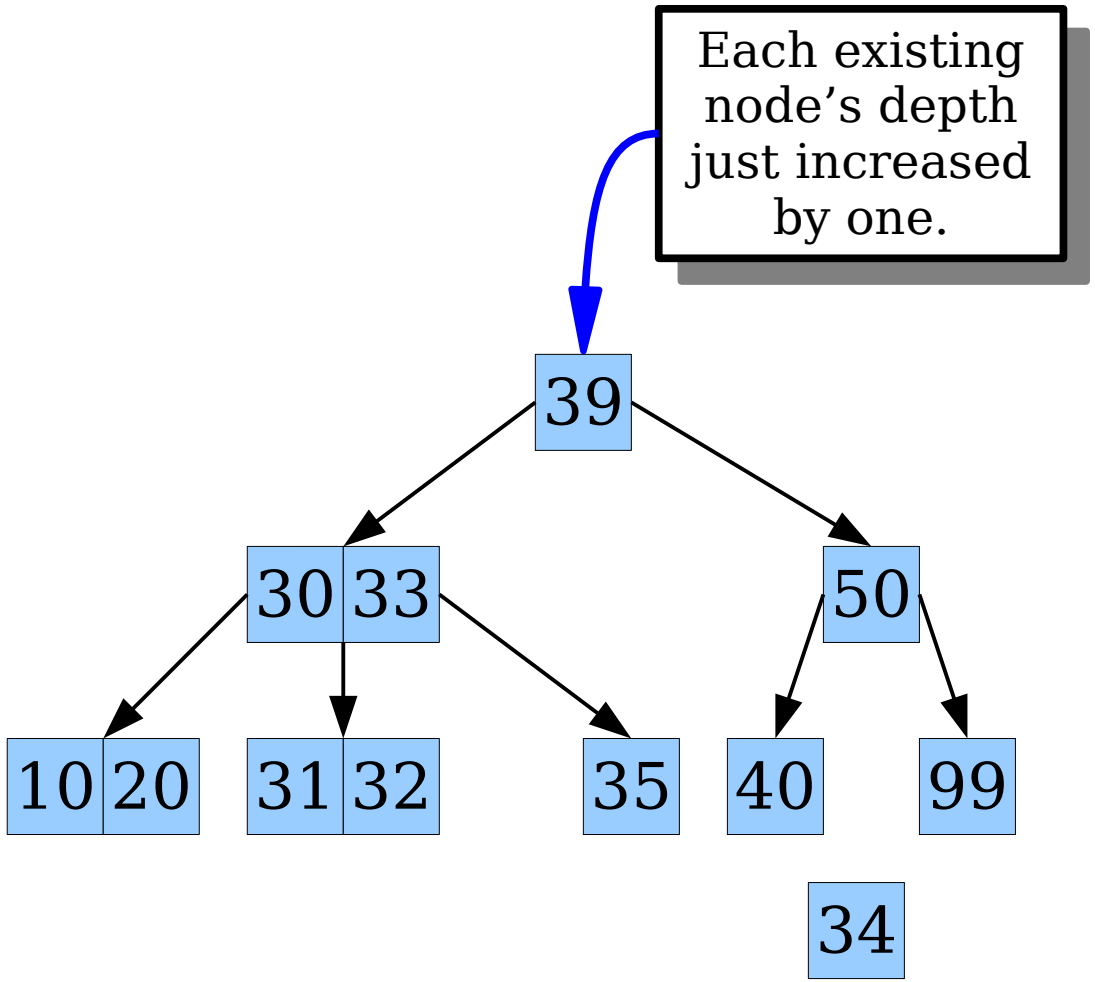
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



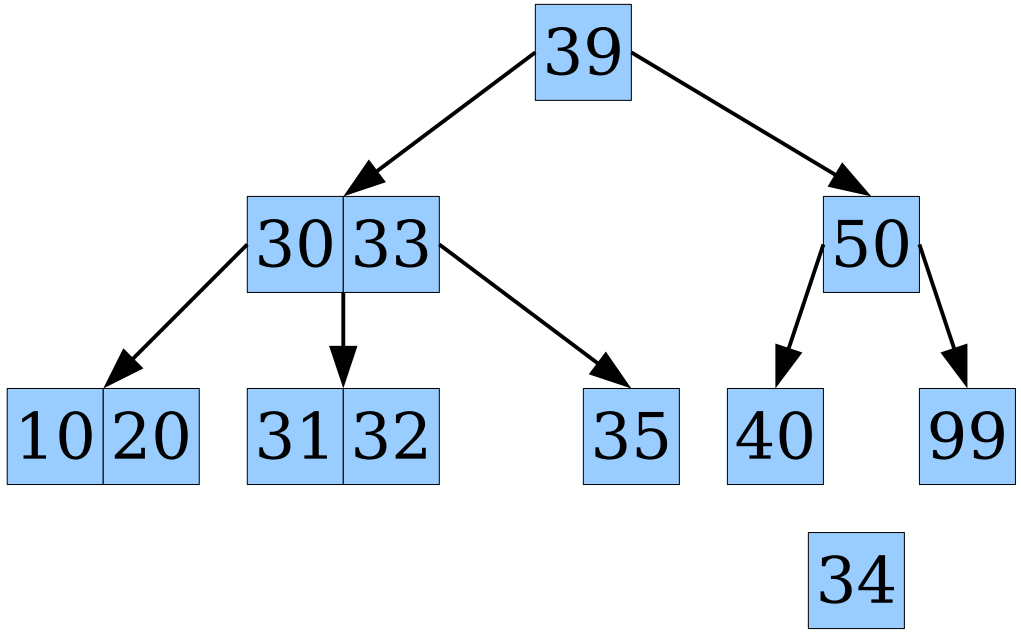
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



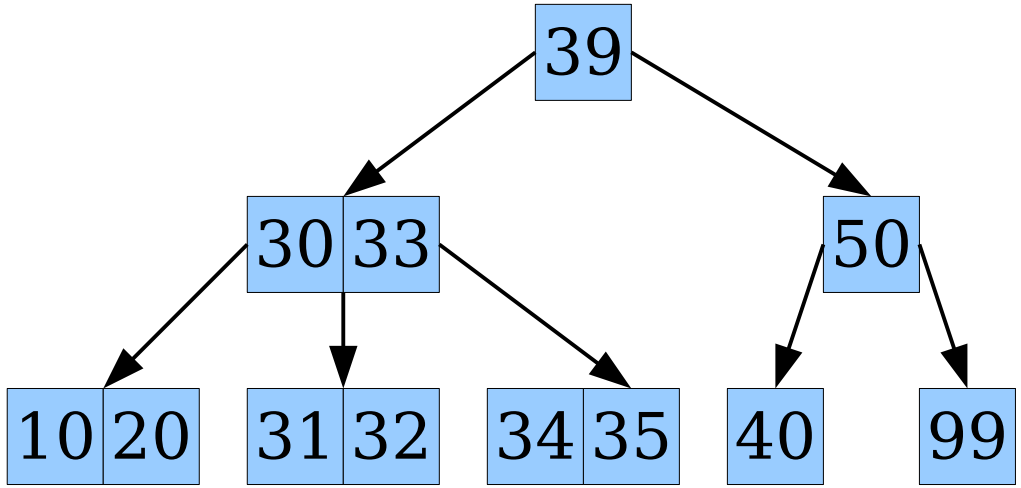
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



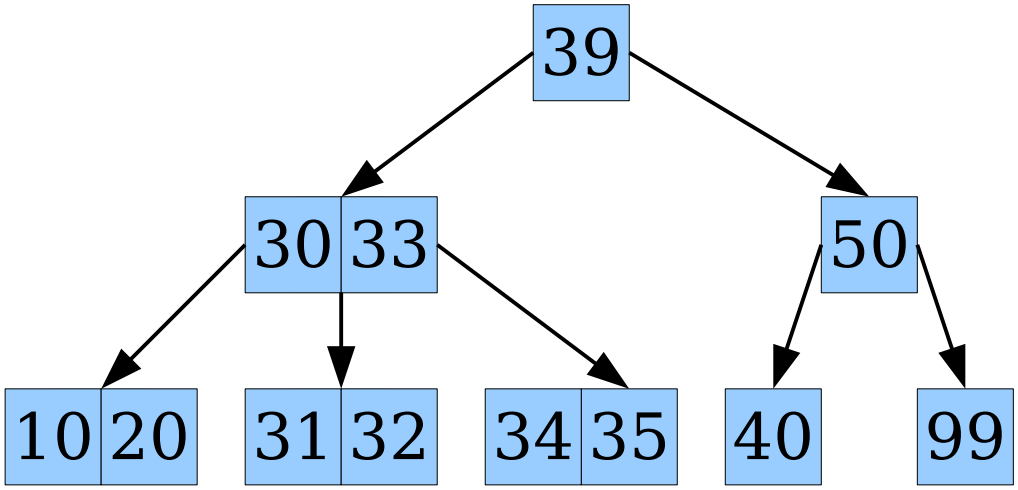
Balanced Multiway Trees

- ***Option 1:*** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- ***Option 2:*** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.



Balanced Multiway Trees

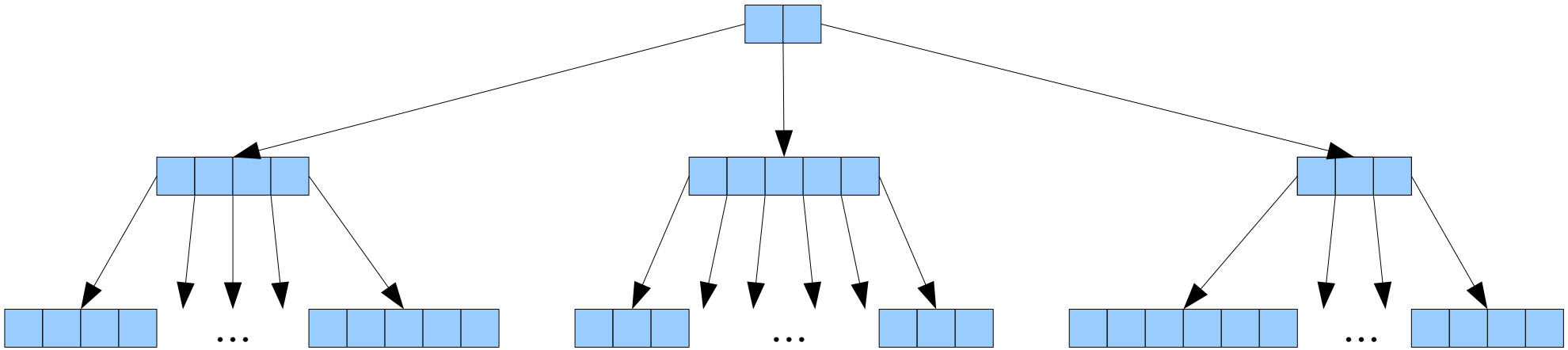
- ***General idea:*** Cap the maximum number of keys in a node. Add keys into leaves. Whenever a node gets too big, split it and kick one key higher up the tree.



- ***Advantage 1:*** The tree is always balanced.
- ***Advantage 2:*** Insertions and lookups are pretty fast.

B-Trees

- A ***B-tree of order b*** is a multiway search tree where
  - each node has between b-1 and $2b$-1 keys, except the root, which may only have between 1 and $2b$-1 keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
- Different authors give different bounds on how many keys can be in each node. The ranges are often [b–1, $2b$–1] or [b, $2b$]. For the purposes of today's lecture, we'll use the range [b-1, $2b$-1] for the key limits, just for simplicity.



Analyzing B-Trees

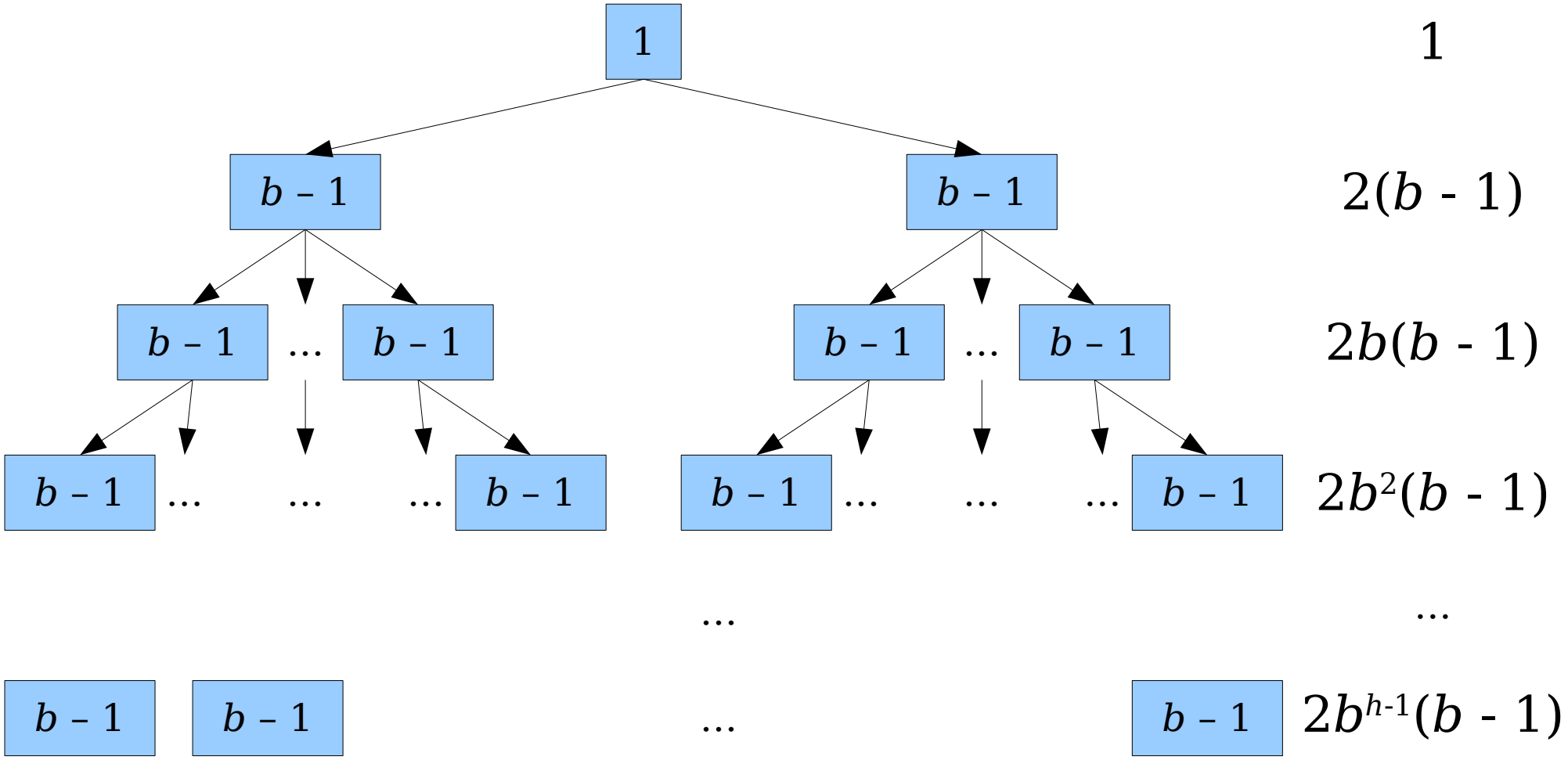
The Height of a B-Tree

- What is the maximum possible height of a B-tree of order b that holds n keys?

Intuition: The branching factor of the tree is at least b, so the number of keys per level grows exponentially in b. Therefore, we'd expect something along the lines of $O(\log_b n)$.

The Height of a B-Tree

- What is the maximum possible height of a B-tree of order b that holds n keys?



The Height of a B-Tree

- ***Theorem:*** The maximum height of a B-tree of order b containing n keys is $O(\log_b n)$.
- ***Proof:*** Number of keys n in a B-tree of height h is guaranteed to be at least

$$1 + 2\mathbf{(b-1)} + 2\mathbf{b(b-1)} + 2\mathbf{b^2(b-1)} + \ldots + 2\mathbf{b^{h-1}(b-1)}$$

$$= 1 + 2\mathbf{(b-1)}(1 + \mathbf{b} + \mathbf{b^2} + \ldots + \mathbf{b^{h-1}})$$

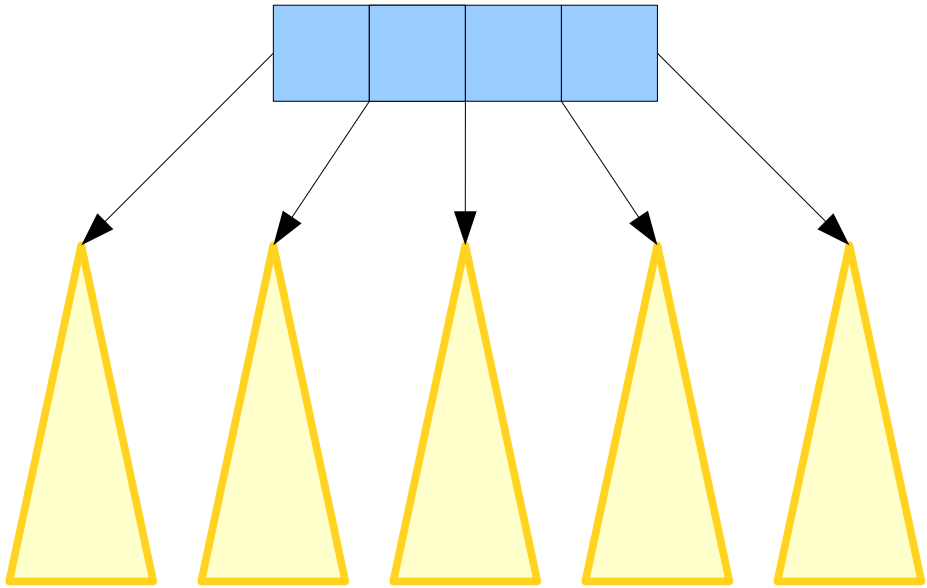
$$= 1 + 2\mathbf{(b-1)}((\mathbf{b}^h - 1) / \mathbf{(b-1)})$$

$$= 1 + 2(\mathbf{b}^h - 1) = 2\mathbf{b}^h - 1.$$

Solving $n = 2b^h - 1$ yields $h = \log_b ((n + 1) / 2)$, so the height is $O(\log_b n)$. ■

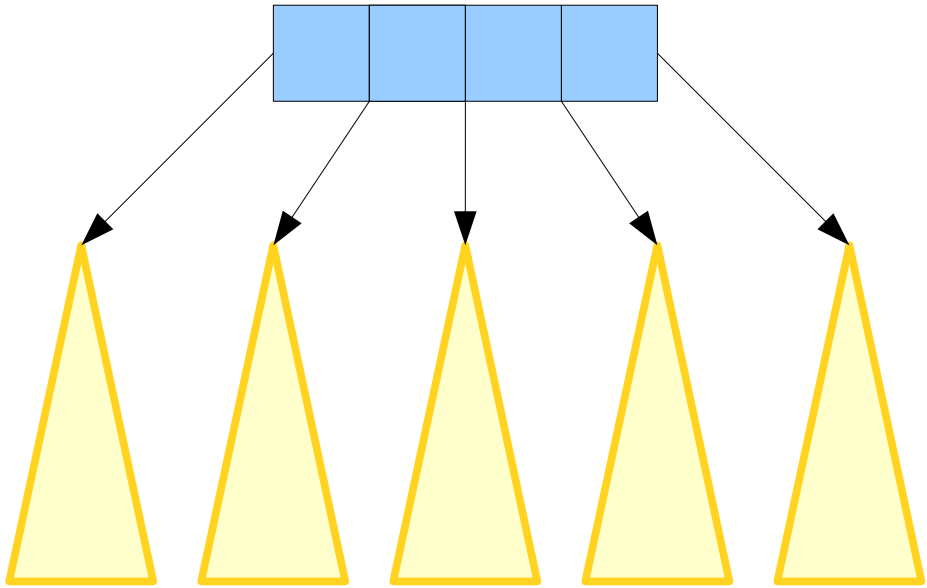
Analyzing Efficiency

- Suppose we have a B-tree of order b.
- What is the worst-case runtime of looking up a key in the B-tree?
- ***Answer:*** It depends on how we do the search!



Analyzing Efficiency

- To do a lookup in a B-tree, we need to determine which child tree to descend into.
- This means we need to compare our query key against the keys in the node.
- ***Question:*** How should we do this?

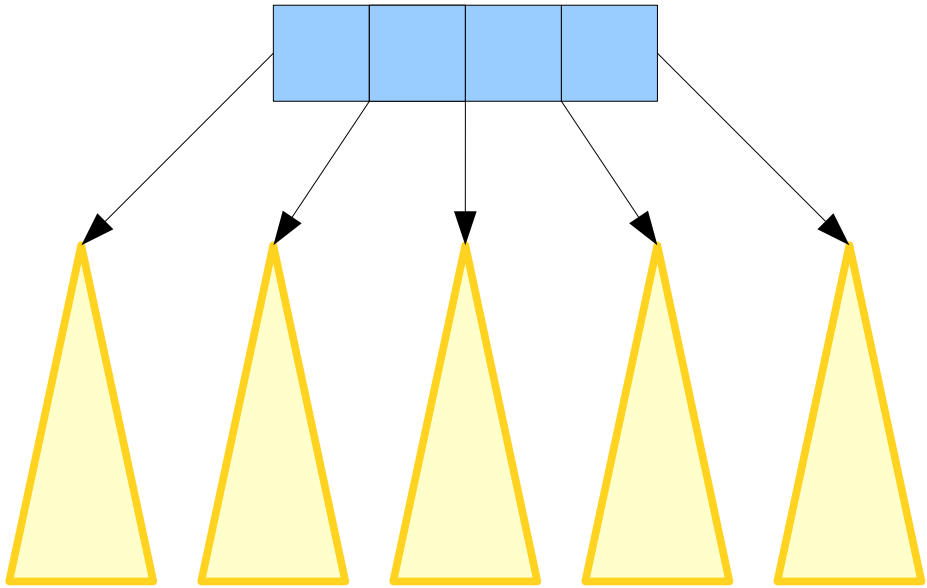


Analyzing Efficiency

- ***Option 1:*** Use a linear search!
- Cost per node: $O(b)$.
- Nodes visited: $O(\log_b n)$.
- Total cost:

$$O(b) \cdot O(\log_b n)$$

$$= \mathbf{O(b \log_b n)}$$



Analyzing Efficiency

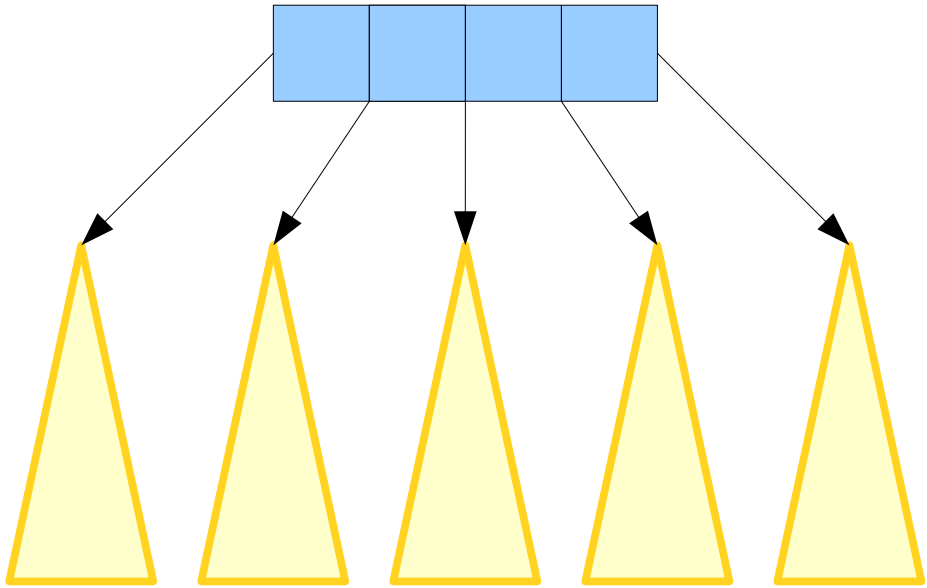
- ***Option 2:*** Use a binary search!
- Cost per node: O(log b).
- Nodes visited: O($\log_b n$).
- Total cost:

$$O(\log b) \cdot O(\log_b n)$$

$$= O(\log b \cdot \log_b n)$$

$$= O(\log b \cdot (\log n) / (\log b))$$

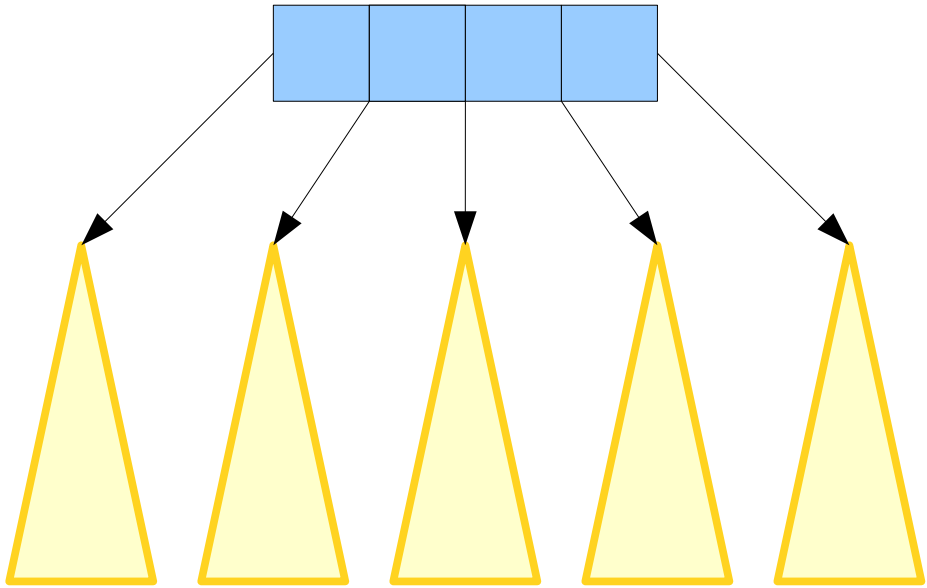
$$= \mathbf{O(\log n)}.$$



Intuition: We can't do better than O(log n) for arbitrary data, because it's the information-theoretic minimum number of comparisons needed to find something in a sorted collection!

Analyzing Efficiency

- Suppose we have a B-tree of order b.
- What is the worst-case runtime of inserting a key into the B-tree?
- Each insertion visits $O(\log_b n)$ nodes, and in the worst case we have to split every node we see.
- ***Answer:*** $O(b \log_b n)$.



Analyzing Efficiency

- The cost of an insertion in a B-tree of order b is $O(b \log_b n)$.
- What's the best choice of b to use here?
- Note that

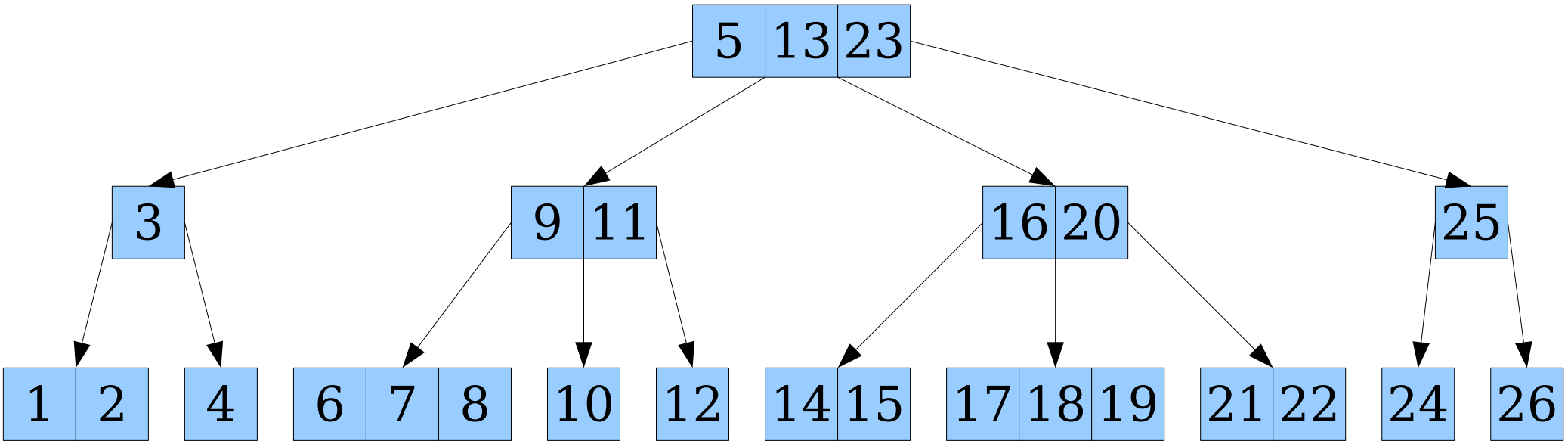
$$b \log_b n$$
$$= b (\log n / \log b)$$
$$= (b / \log b) \log n.$$

Fun fact: This is the same time bound you'd get if you used a b-ary heap instead of a binary heap for a priority queue.

- What choice of b minimizes $b / \log b$?
- ***Answer:*** Pick $b = e$.

2-3-4 Trees

- A ***2-3-4 tree*** is a B-tree of order 2. Specifically:
  - each node has between 1 and 3 keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
- You actually saw this B-tree earlier! It's the type of tree from our insertion example.



The Story So Far

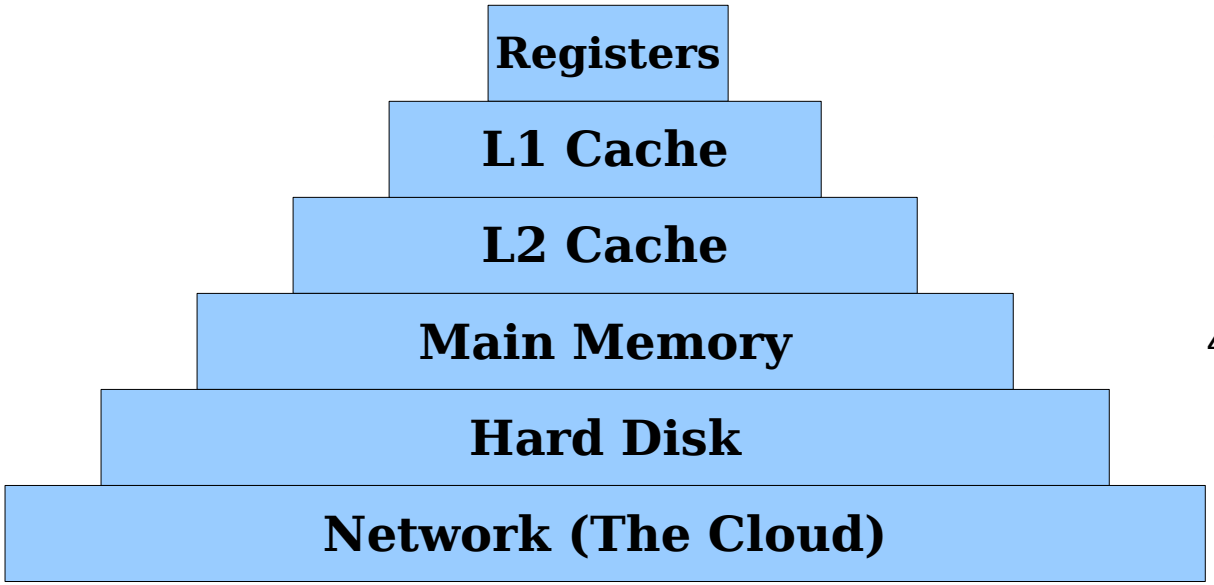
- A B-tree supports
  - lookups in time O(log n), and
  - insertions in time O($b \log_b n$).
- Picking b to be around 2 or 3 makes this optimal in Theoryland.
  - The 2-3-4 tree is great for that reason.
- ***Plot Twist:*** In practice, you most often see choices of b like 1,024 or 4,096.
- ***Question:*** Why would anyone do that?


Theoryland
IRL

The Memory Hierarchy

The Memory Hierarchy

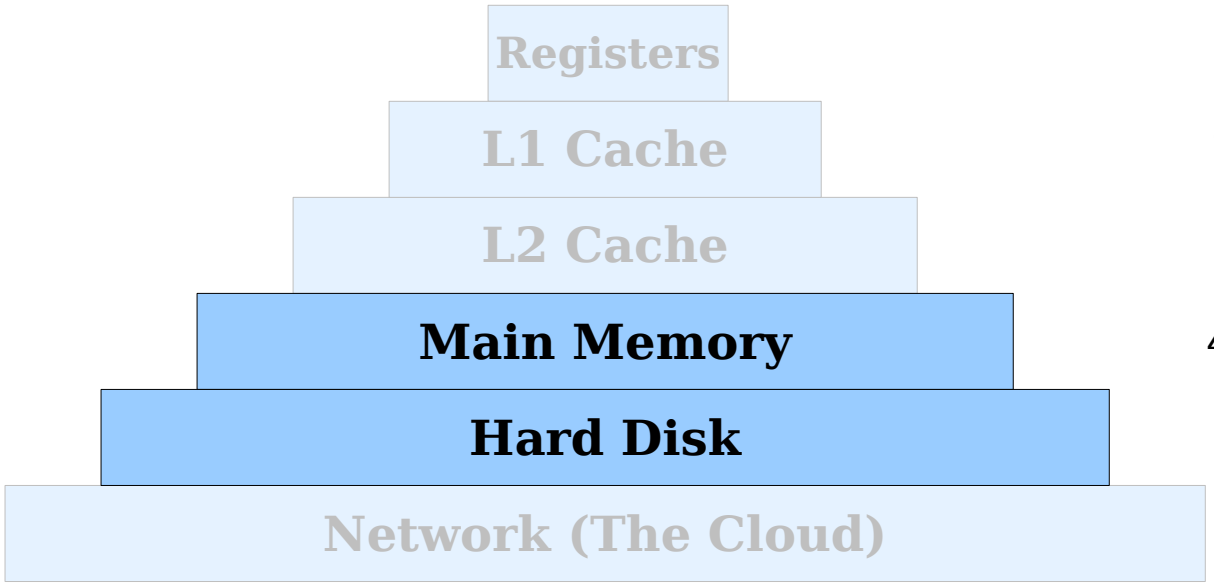
- ***Idea:*** Try to get the best of all worlds by using multiple types of memory.



	256B - 8KB	0.25 – 1ns
	16KB – 64KB	1ns – 5ns
	1MB - 4MB	5ns – 25ns
	4GB – 256GB	25ns – 100ns
	1TB+	3 – 10ms
	Lots	10 – 2000ms

The Memory Hierarchy

- ***Idea:*** Try to get the best of all worlds by using multiple types of memory.



Registers	256B - 8KB	0.25 – 1ns
L1 Cache	16KB – 64KB	1ns – 5ns
L2 Cache	1MB - 4MB	5ns – 25ns
Main Memory	4GB – 256GB	25ns – 100ns
Hard Disk	1TB+	3 – 10ms
Network (The Cloud)	*Lots*	10 – 2000ms

External Data Structures

- Suppose you have a data set that's *way* too big to fit in RAM.
- The data structure is on disk and read into RAM as needed.
- Data from disk doesn't come back one *byte* at a time, but rather one *page* at a time.
- ***Goal:*** Minimize the number of disk reads and writes, not the number of instructions executed.

"Please give me 4KB starting at location *addr2*"

00110101000101000101000101…

Analyzing B-Trees

- Suppose we tune b so that each node in the B-tree fits inside a single disk page.
- We *only* care about the number of disk pages read or written.
  - It's so much slower than RAM that it'll dominate the runtime.
- ***Question:*** What is the cost of a lookup in a B-tree in this model?

- ***Question:*** What is the cost of inserting into a B-tree in this model?

Analyzing B-Trees

- Suppose we tune b so that each node in the B-tree fits inside a single disk page.
- We *only* care about the number of disk pages read or written.
  - It's so much slower than RAM that it'll dominate the runtime.
- ***Question:*** What is the cost of a lookup in a B-tree in this model?
  - Answer: The height of the tree, $O(\log_b n)$.
- ***Question:*** What is the cost of inserting into a B-tree in this model?
  - Answer: The height of the tree, $O(\log_b n)$.

External Data Structures

- Because B-trees have a huge branching factor, they're great for on-disk storage.
  - Disk block reads/writes are slow compared to CPU operations.
  - The high branching factor minimizes the number of blocks to read during a lookup.
  - Extra work scanning inside a block offset by these savings.
- Major use cases for B-trees and their variants (B^+-trees, H-trees, etc.) include
  - databases (huge amount of data stored on disk);
  - file systems (ext4, NTFS, ReFS); and, recently,
  - in-memory data structures (due to cache effects).

Analyzing B-Trees

- The cost model we use will change our overall analysis.
- Cost is number of operations:

  $O(\log n)$ per lookup, **$O(b \log_b n)$** per insertion.

- Cost is number of blocks accessed:

  $O(\log_b n)$ per lookup, **$O(\log_b n)$** per insertion.

- Going forward, we'll use operation counts as our cost model, though looking at caching effects of data structures would make for an awesome final project!

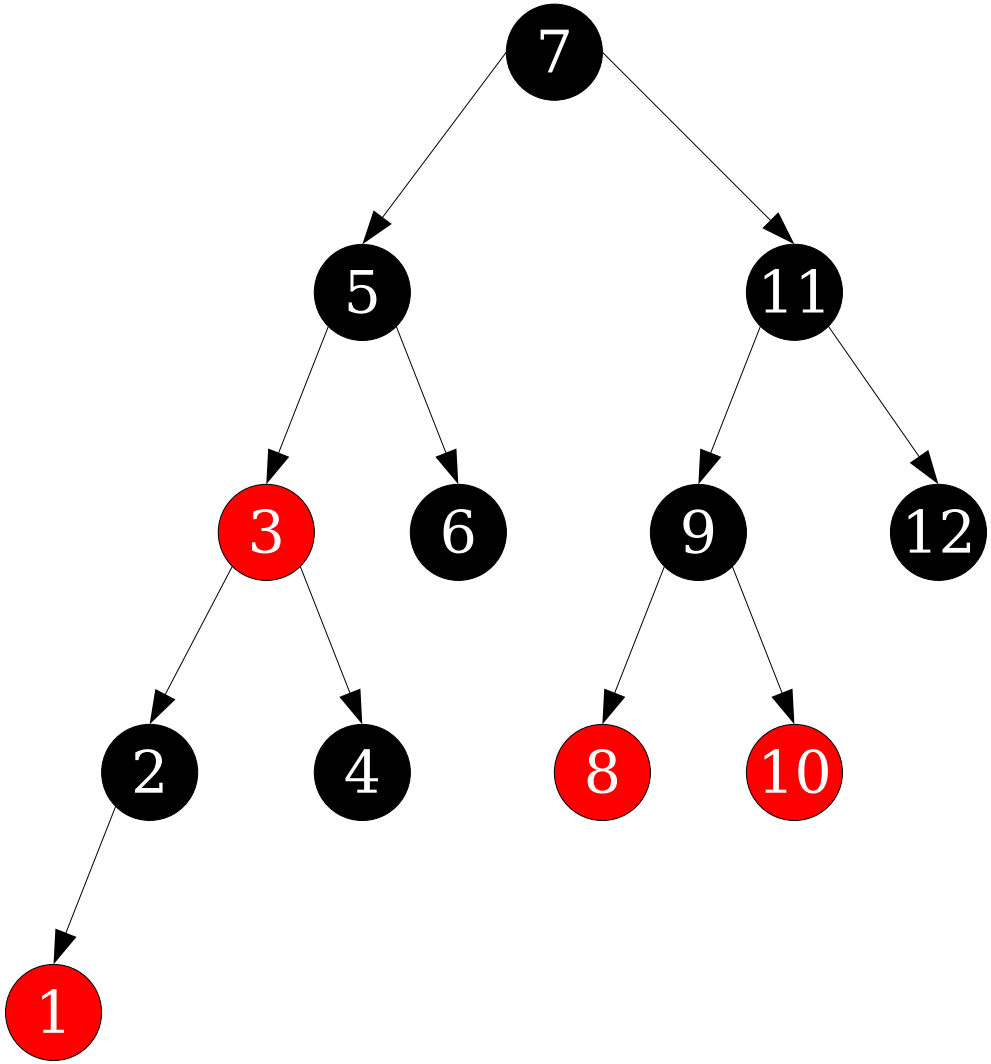
The Story So Far

- We've just built a simple, elegant, balanced multiway tree structure.
- We can use them as balanced trees in main memory (2-3-4 trees).
- We can use them to store huge quantities of information on disk (B-trees).
- We've seen that different cost models are appropriate in different situations.

So… red/black trees?

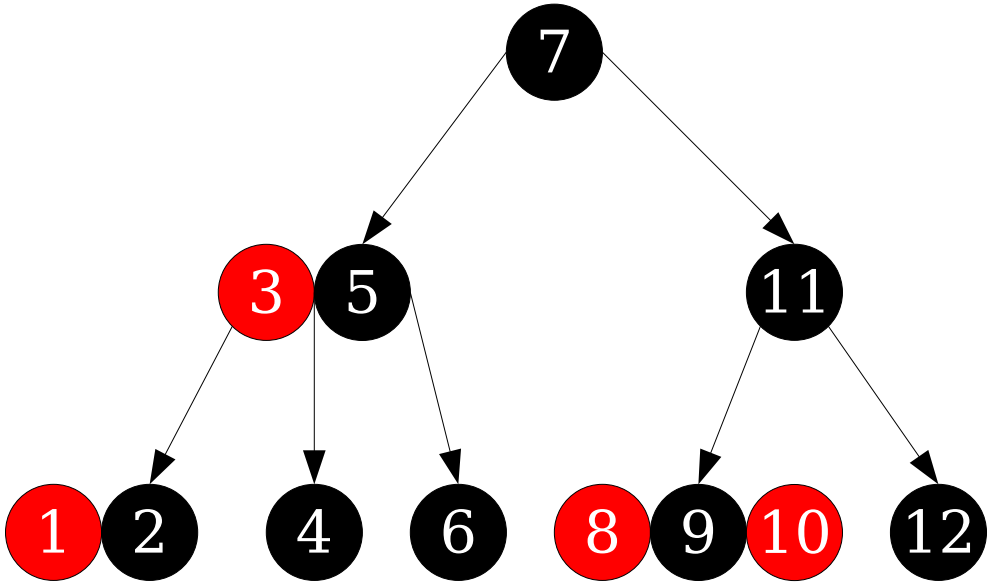
Red/Black Trees

- 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.)

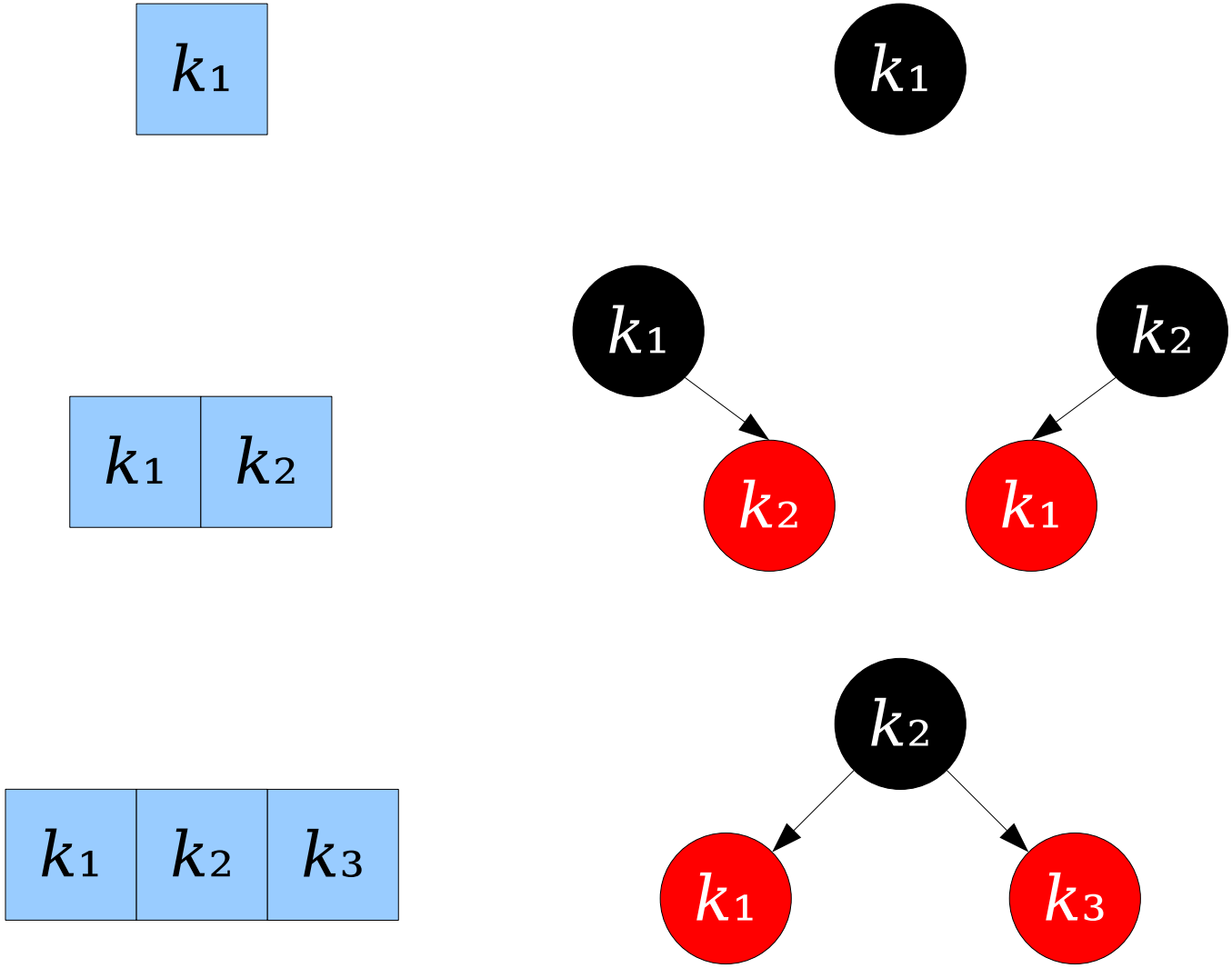


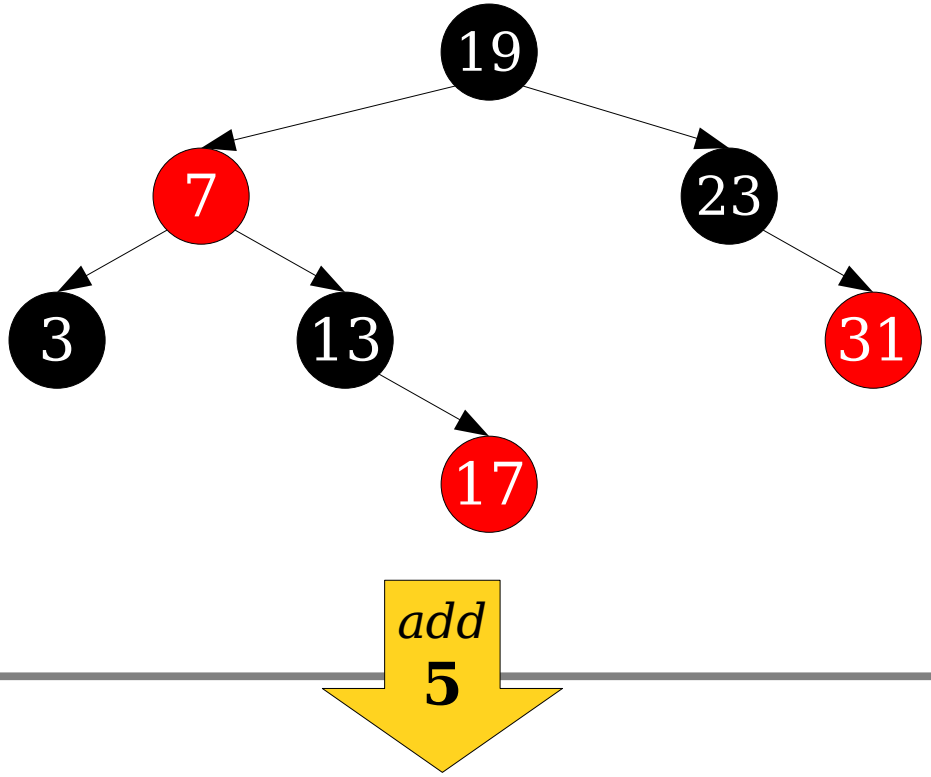
This is a 2-3-4 tree!

Data Structure Isometries

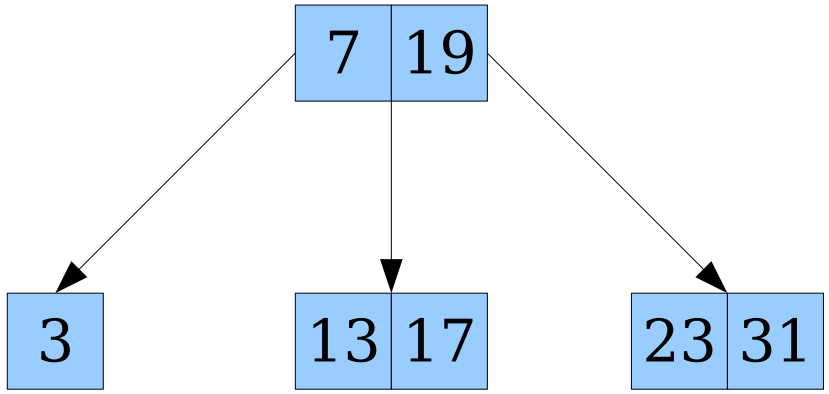
- Red/black trees are an ***isometry*** of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.
- Many data structures can be designed and analyzed in the same way.
- ***Huge advantage:*** Rather than memorizing a complex list of red/black tree rules, just think about what the equivalent operation on the corresponding 2-3-4 tree would be and simulate it with BST operations.

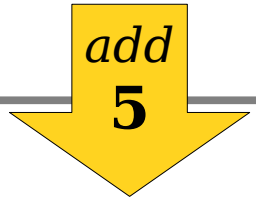
Exploring the Isometry

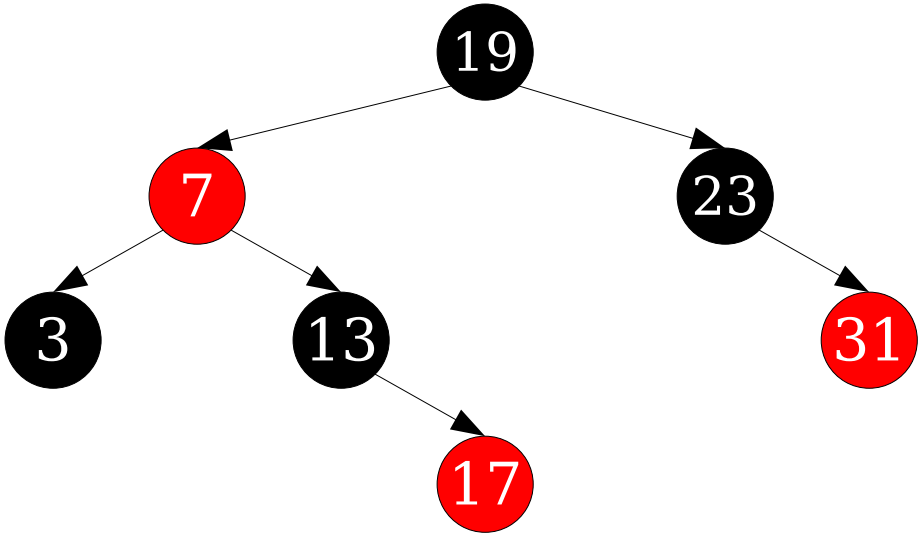


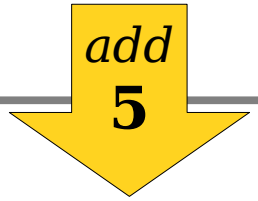

19
7
23
3
13
31
17

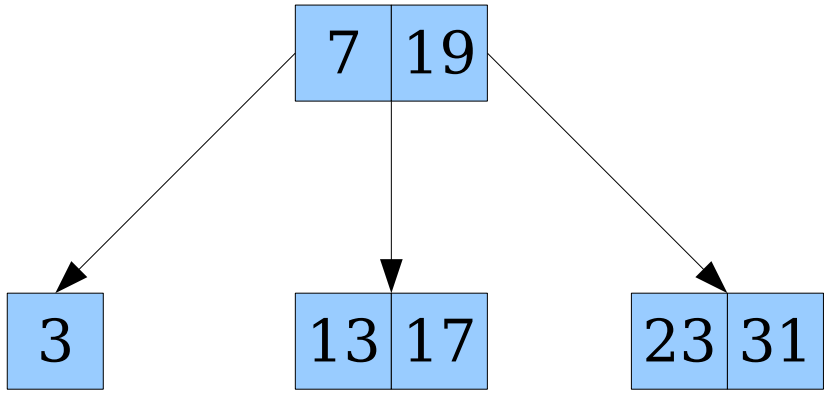
add
5

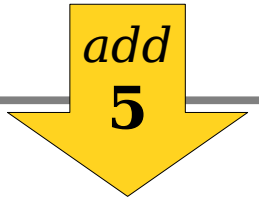

7
19
3
13
17
23
31

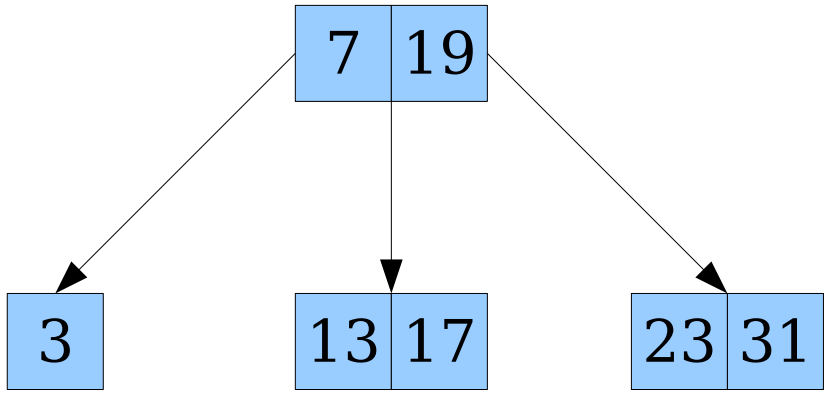

add
5

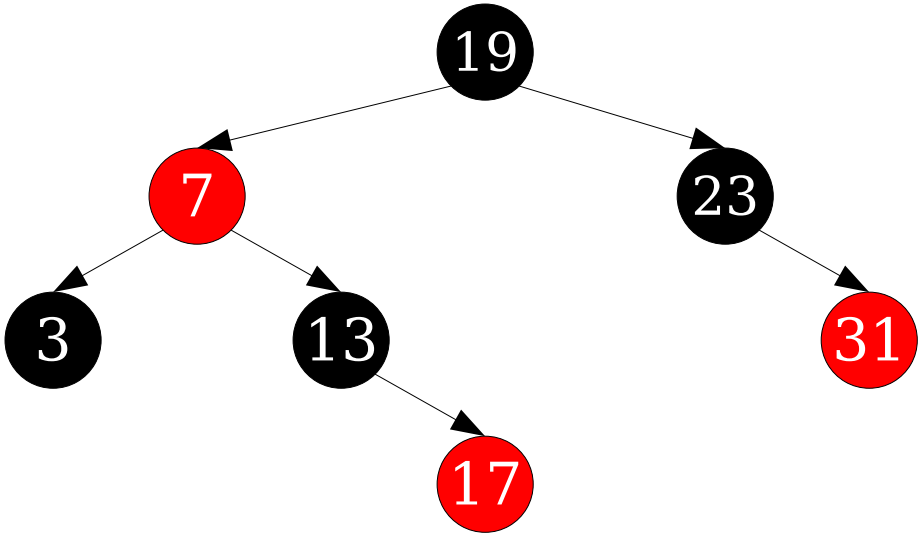

19
7
23
3
13
31
17

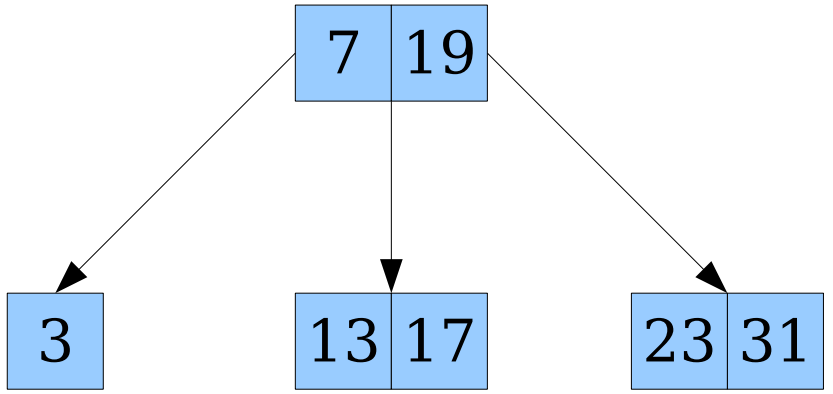

add
5

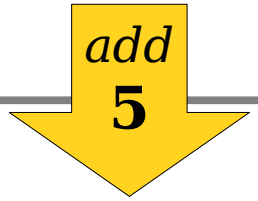

7
19
3
13
17
23
31

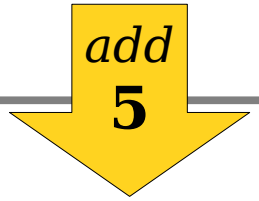

add
5

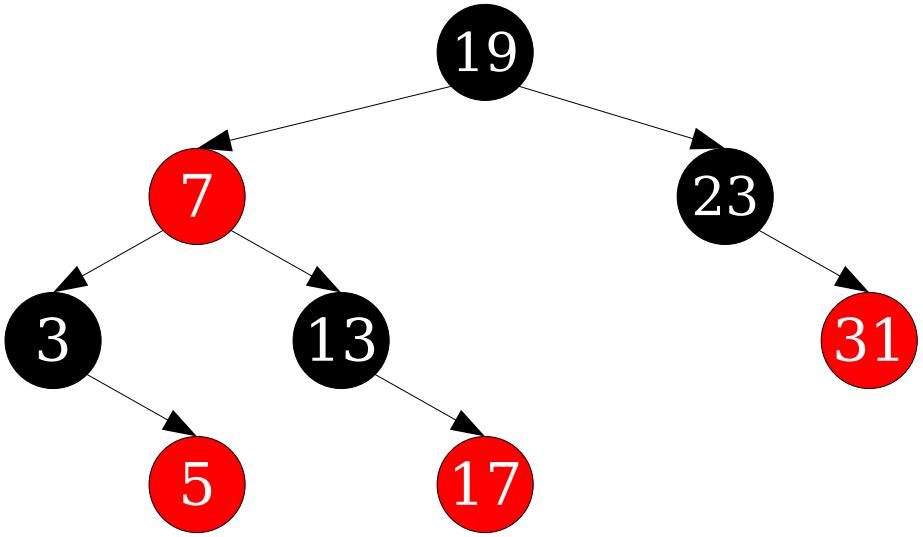

7
19
3
13
17
23
31

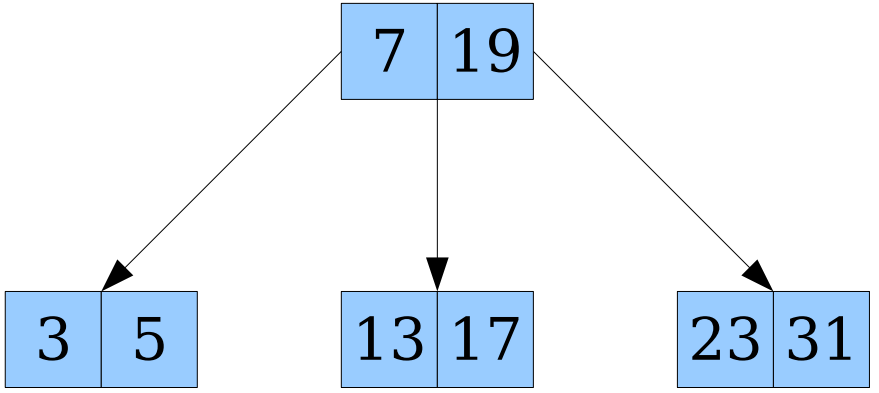

19
7
23
3
13
31
17

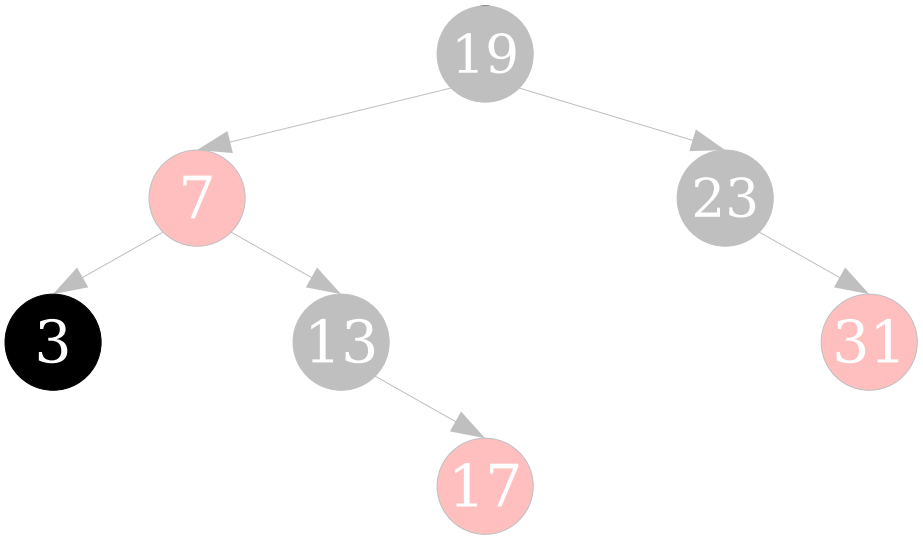

7
19
3
13
17
23
31

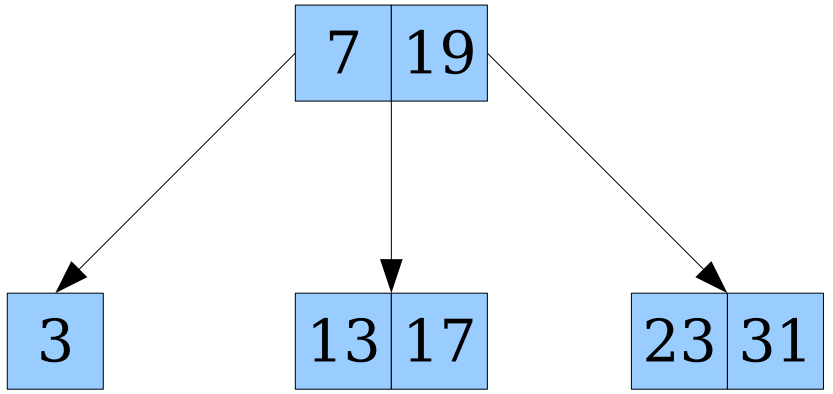

add
5

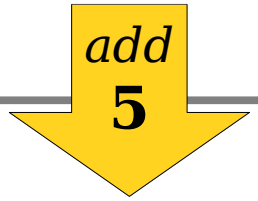

add
5

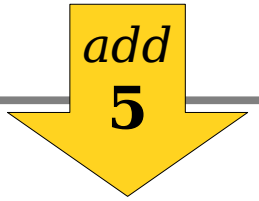

19
7
23
3
13
31
5
17

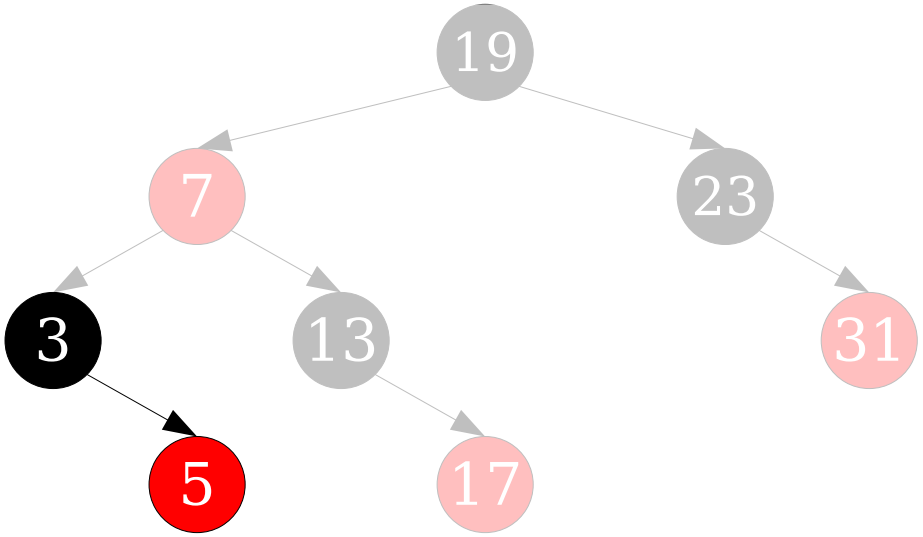

7
19
3
5
13
17
23
31

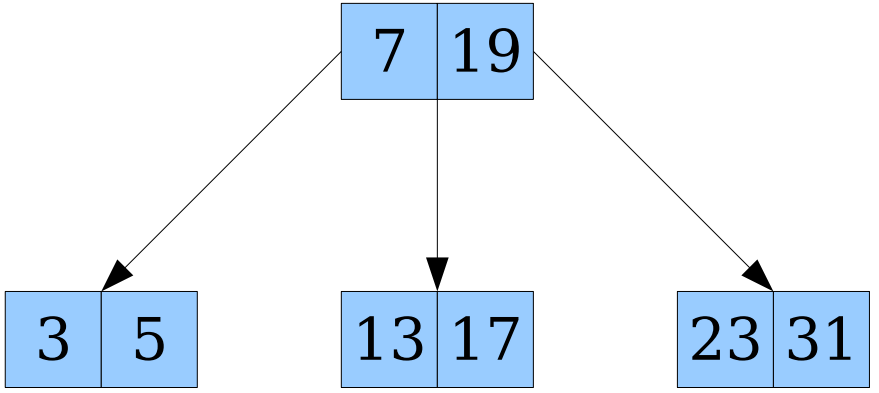

19
7
23
3
13
31
17

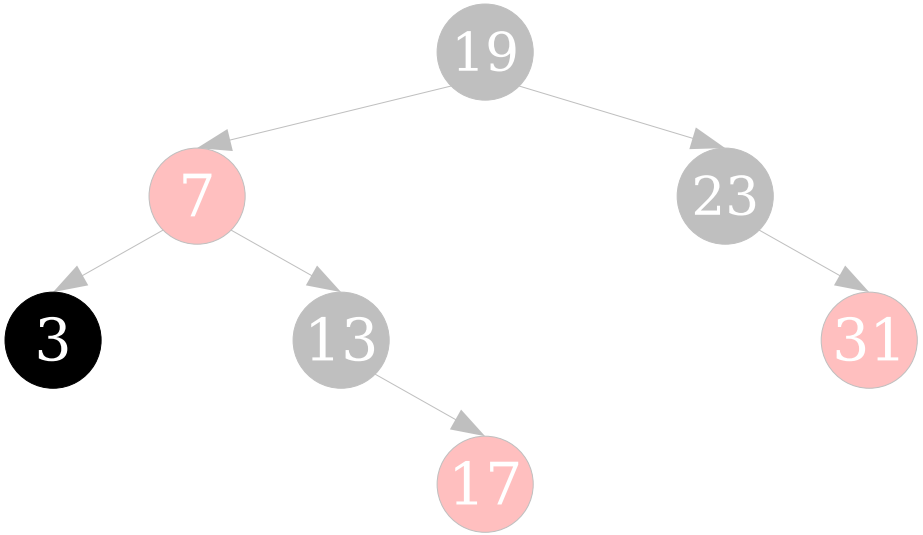

7
19
3
13
17
23
31

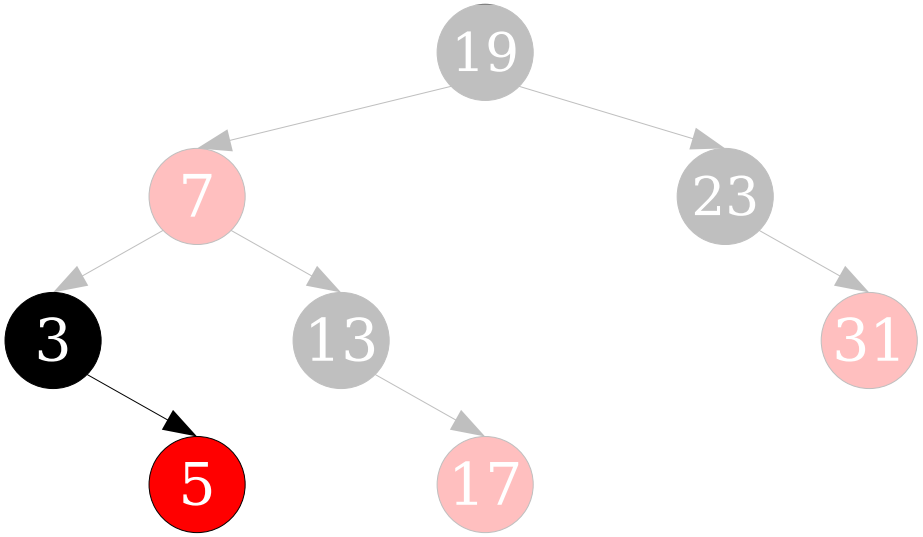

add
5


add
5

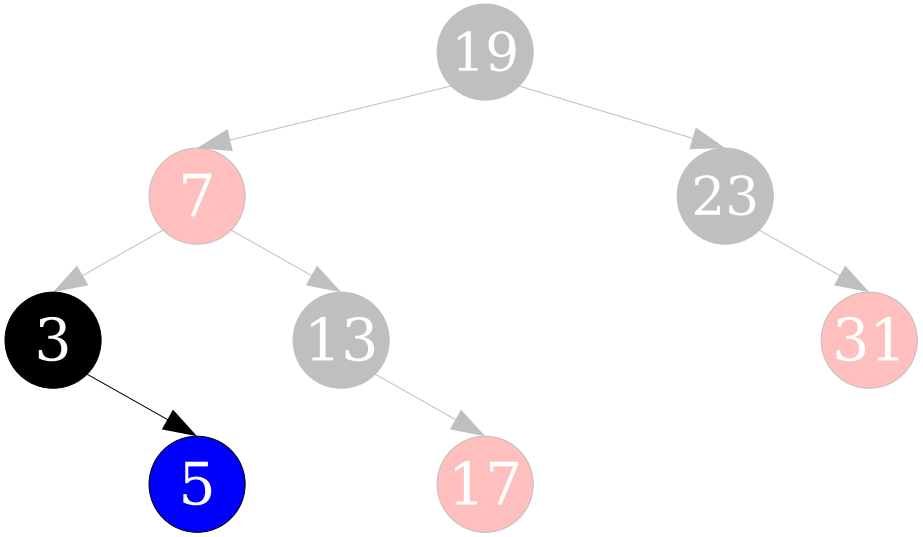

19
7
23
3
13
31
5
17

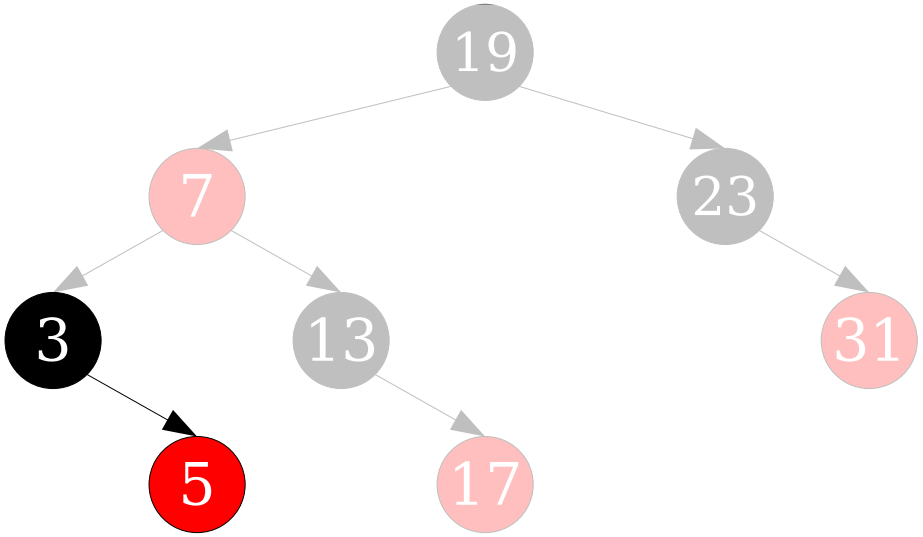

7
19
3
5
13
17
23
31



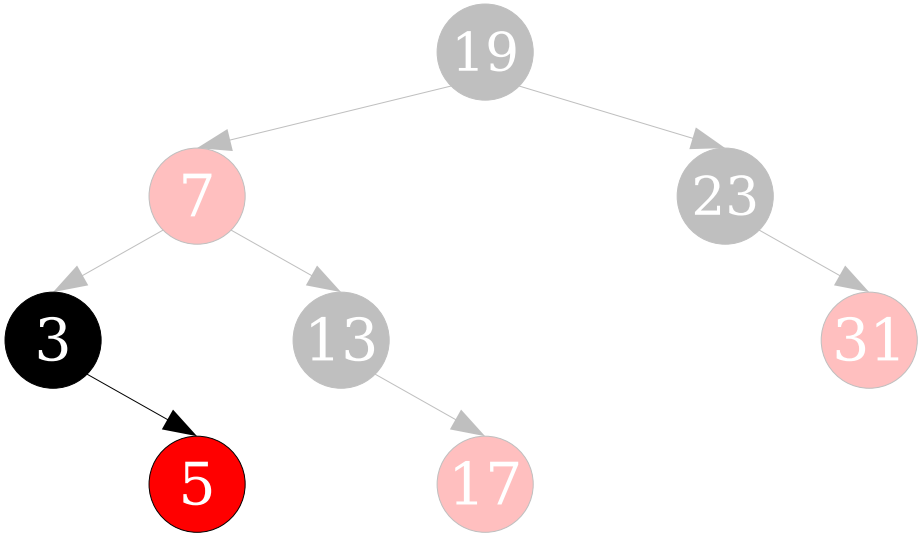


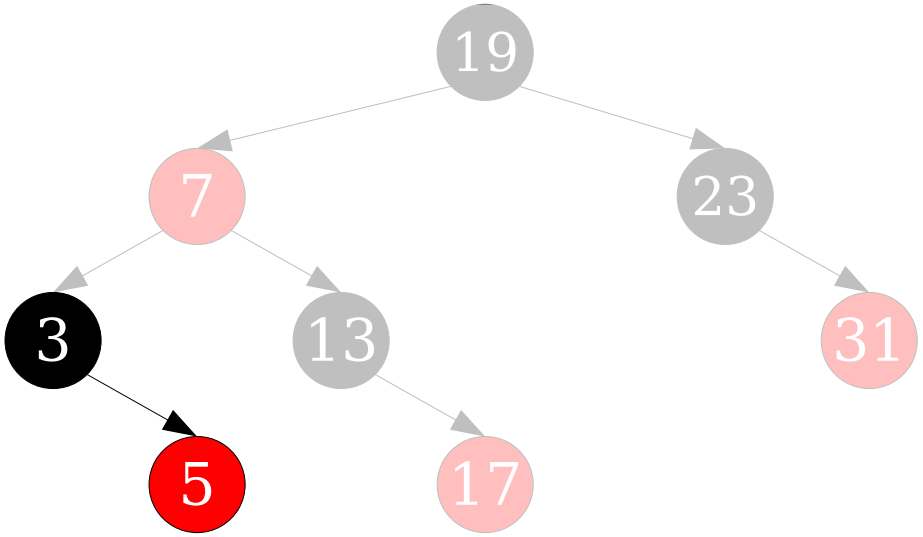
Goal



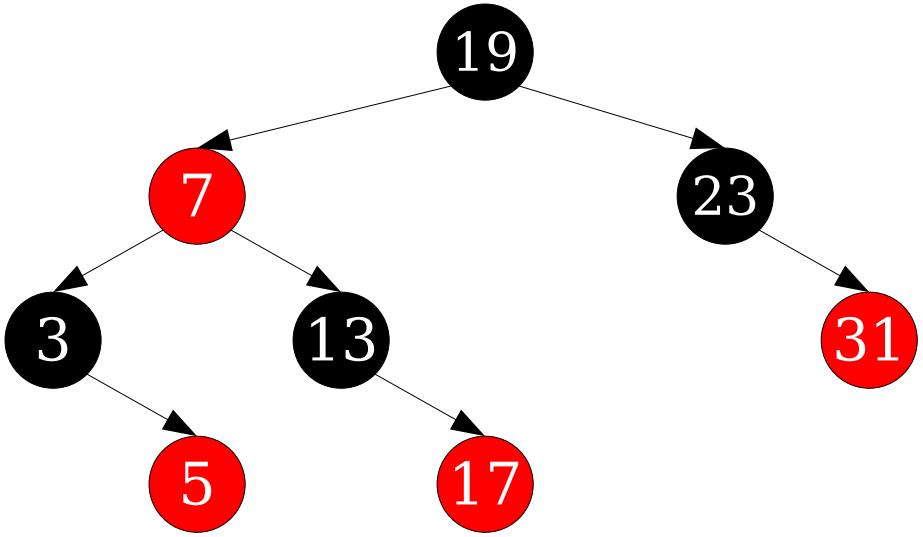


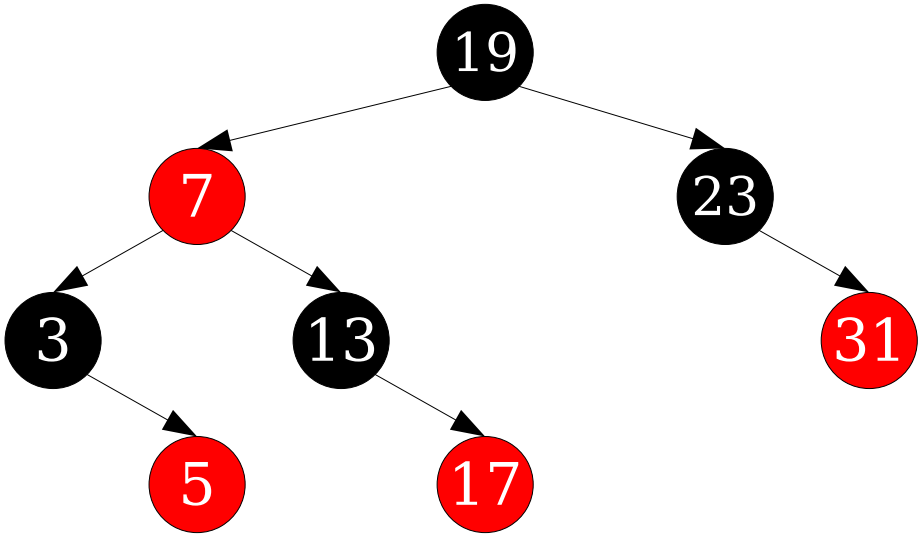
Goal



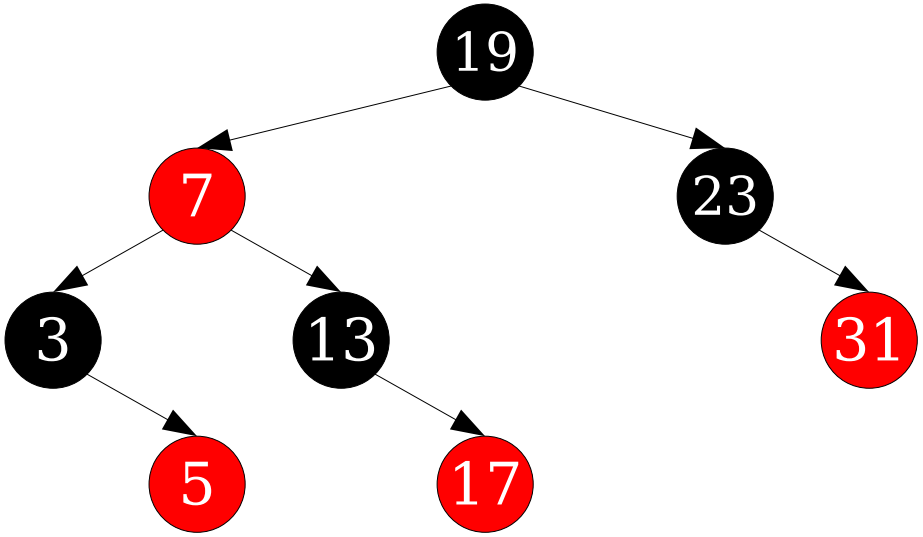


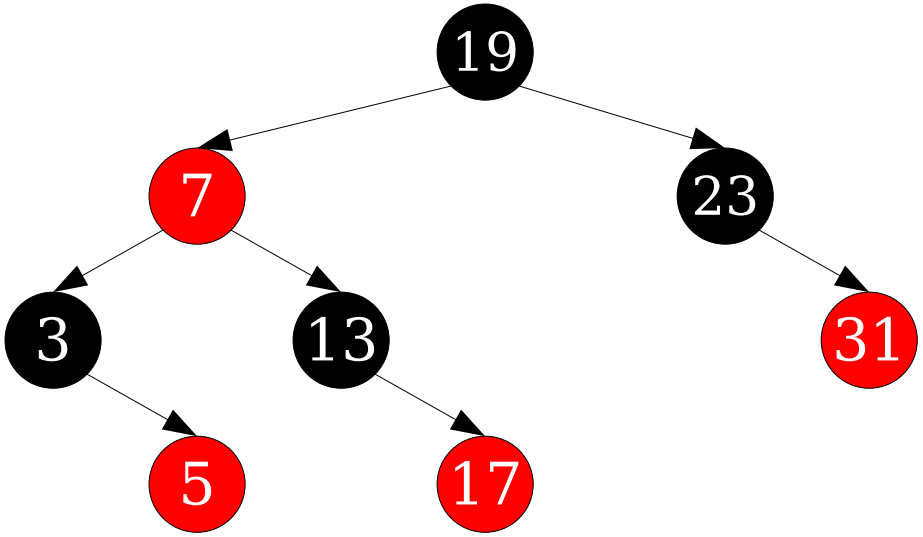
Goal

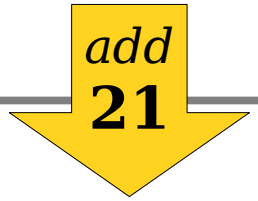


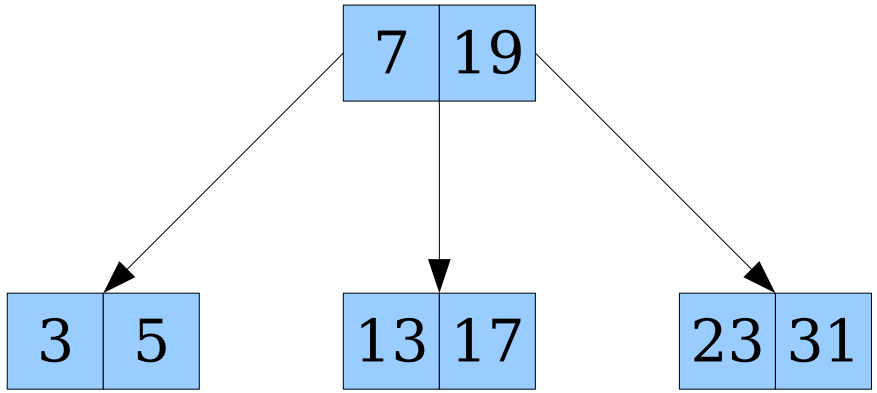


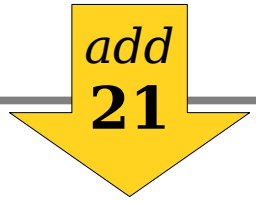
Goal

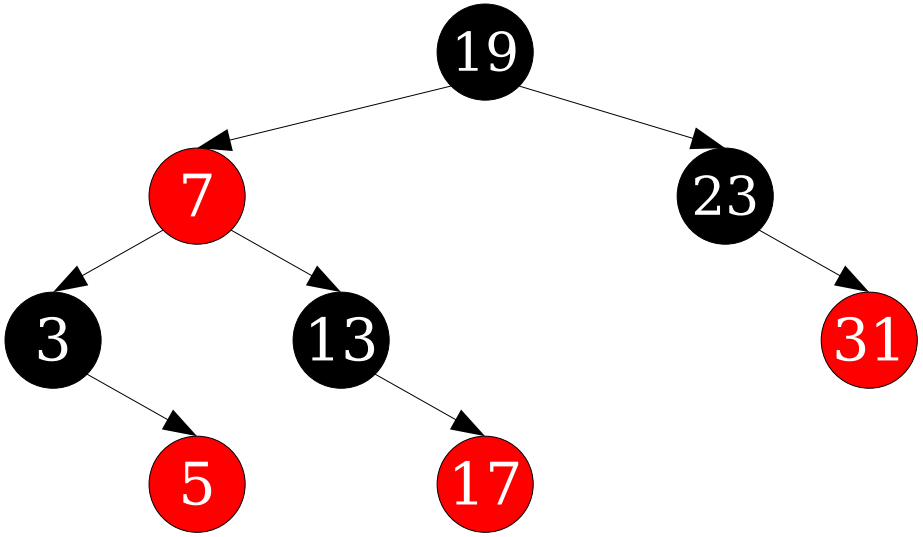

19
7
23
3
13
31
5
17

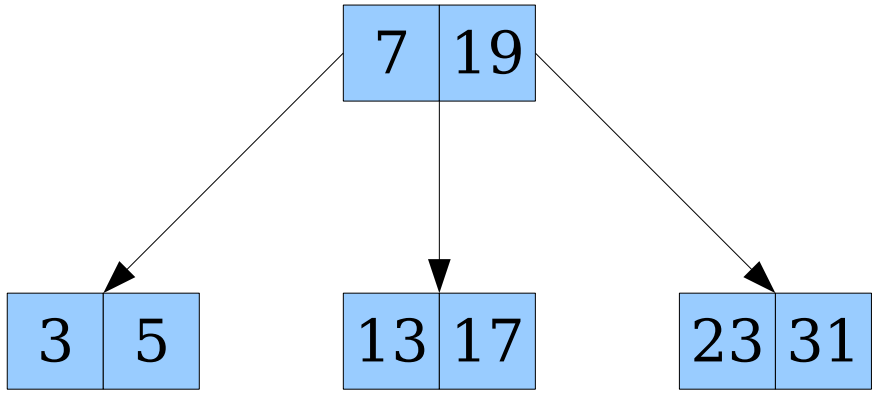

19
7
23
3
13
31
5
17

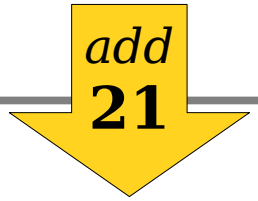

add
21

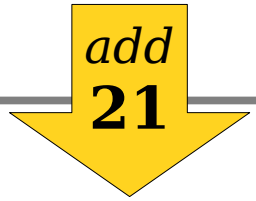

7
19
3
5
13
17
23
31

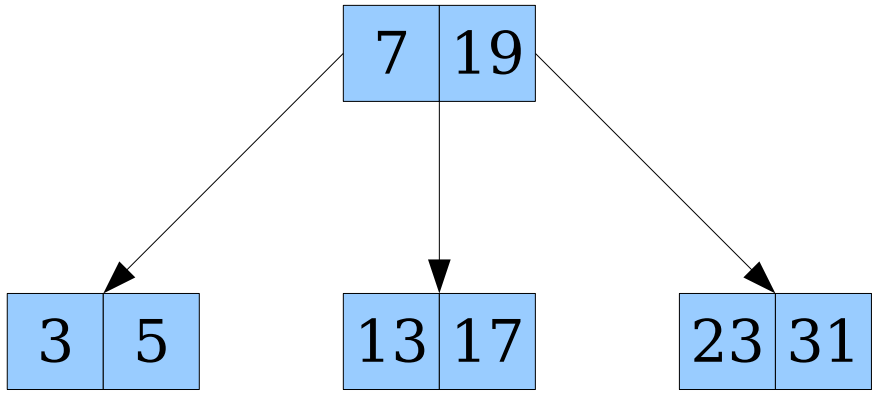

add
21

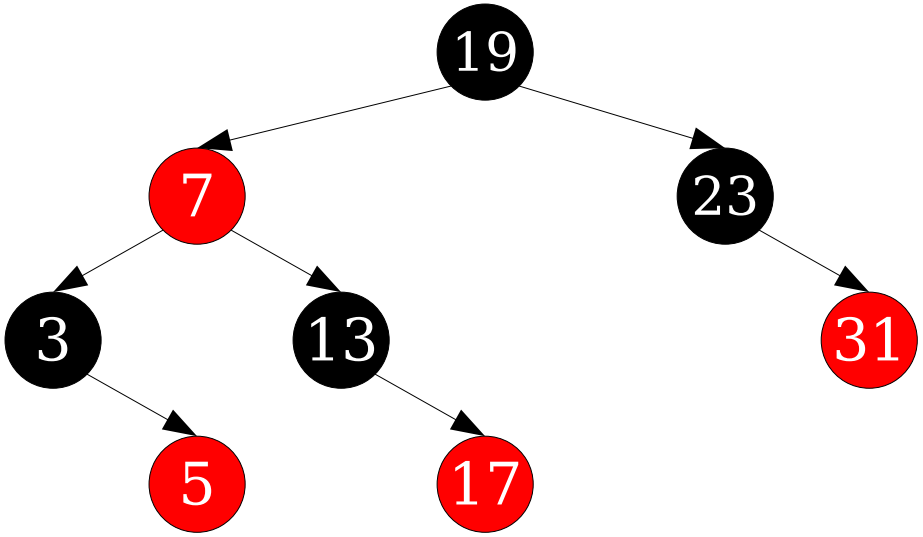

19
7
23
3
13
31
5
17

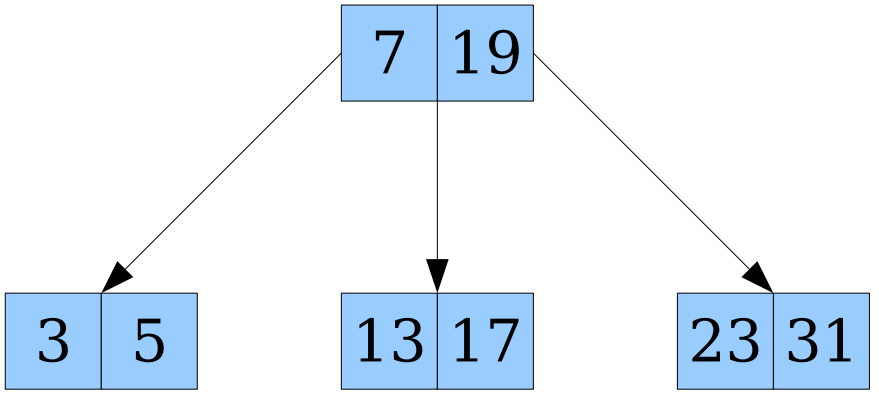

7
19
3
5
13
17
23
31

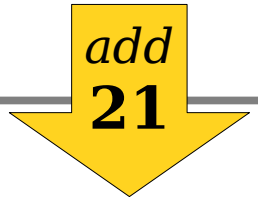

add
21

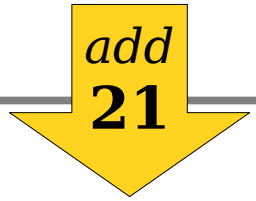

add
21

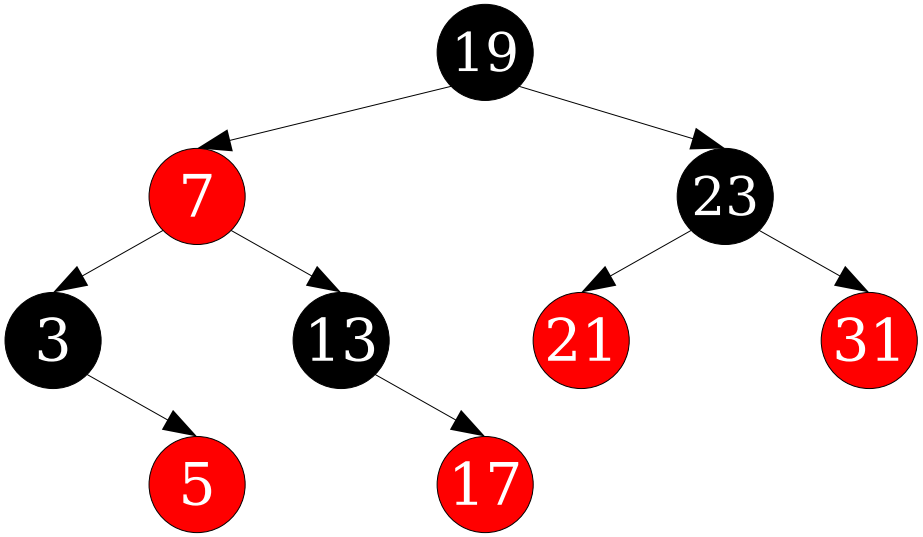

7
19
3
5
13
17
23
31

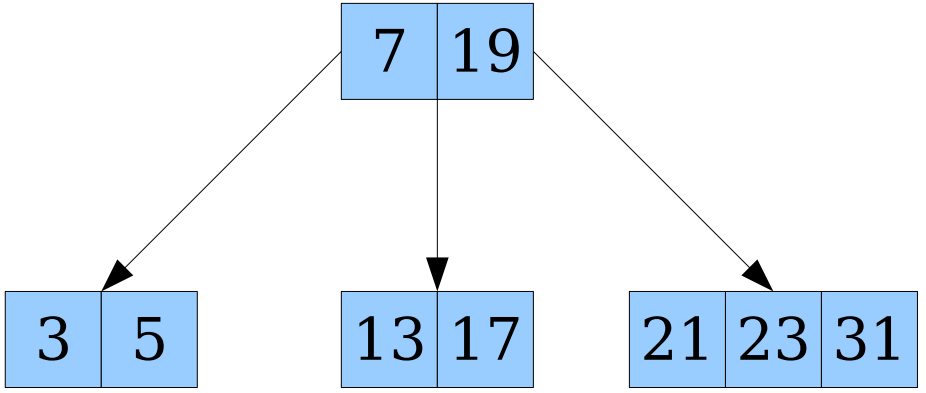

19
7
23
3
13
31
5
17

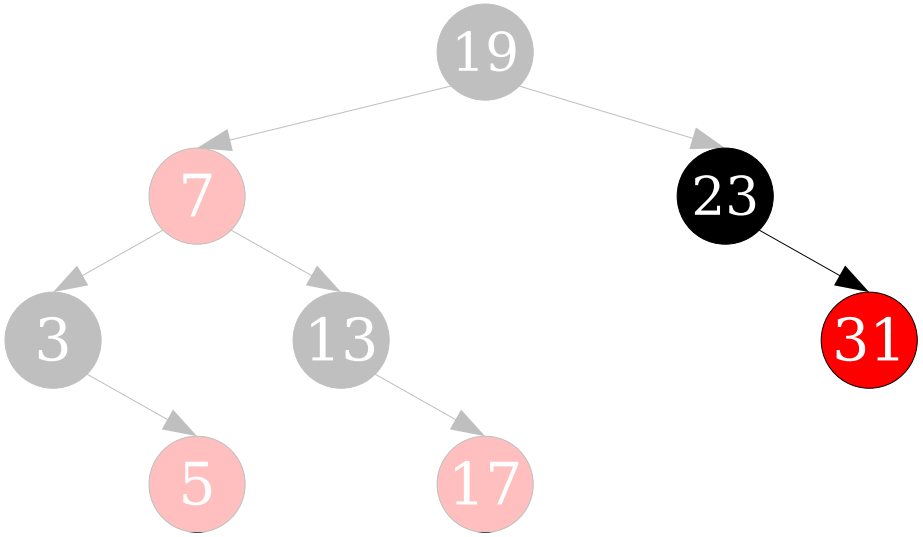

7
19
3
5
13
17
23
31

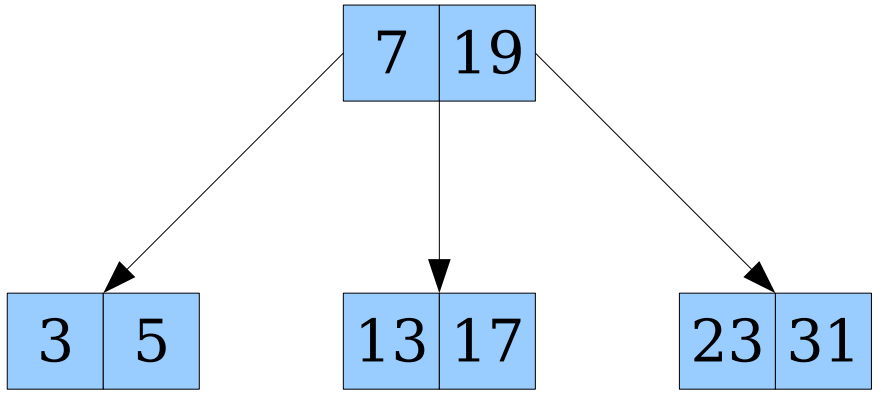

add
21

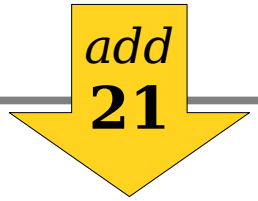

add
21

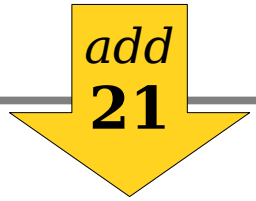

19
7
23
3
13
21
31
5
17

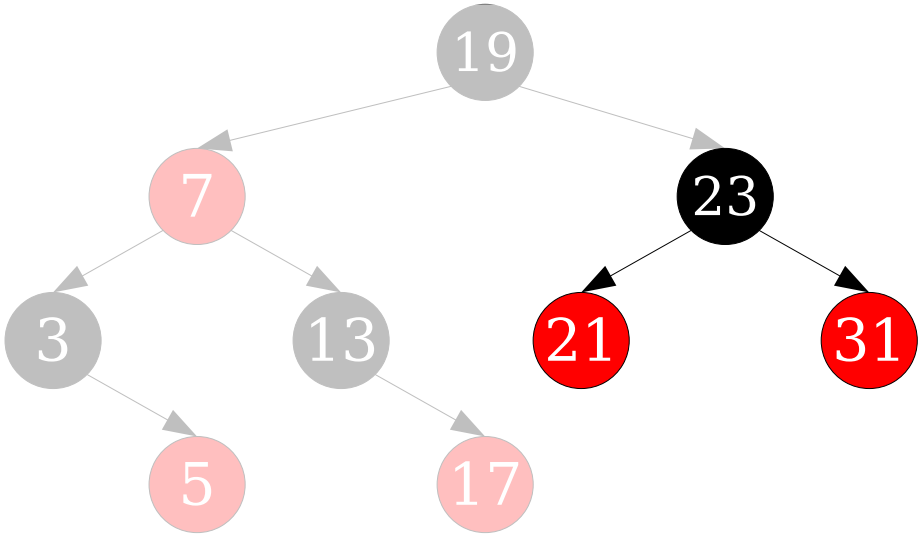

7
19
3
5
13
17
21
23
31

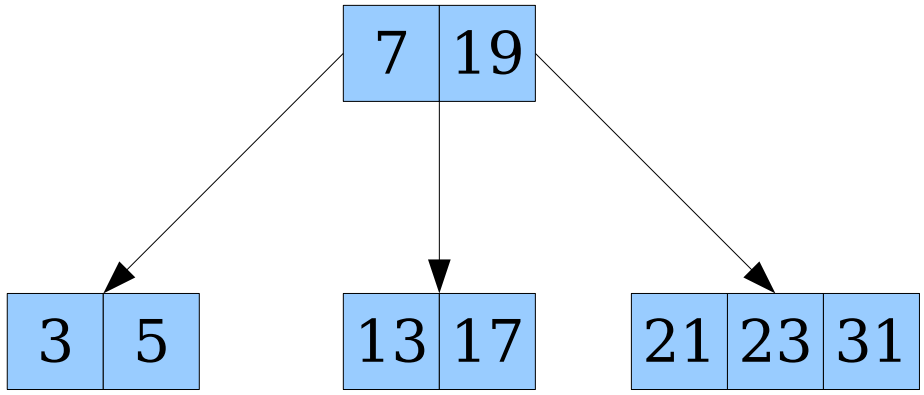

19
7
23
3
13
31
5
17

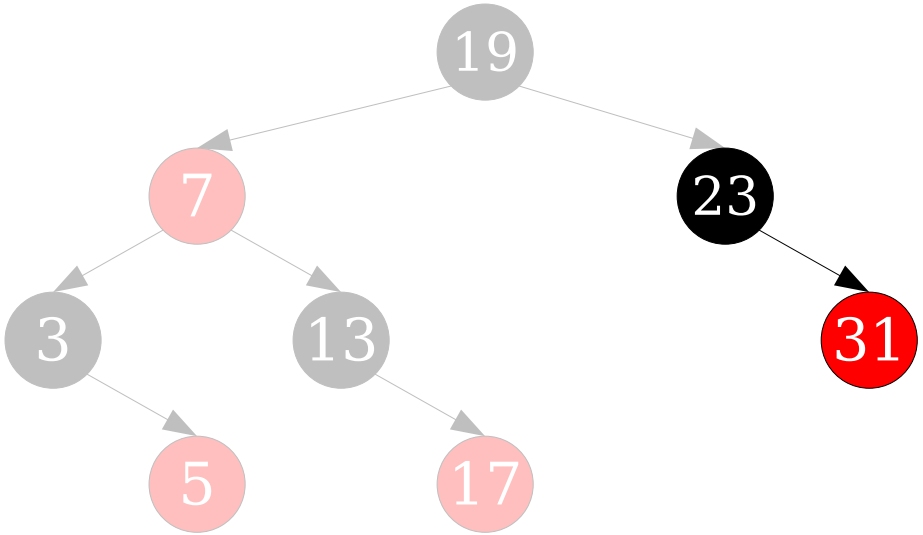

7
19
3
5
13
17
23
31

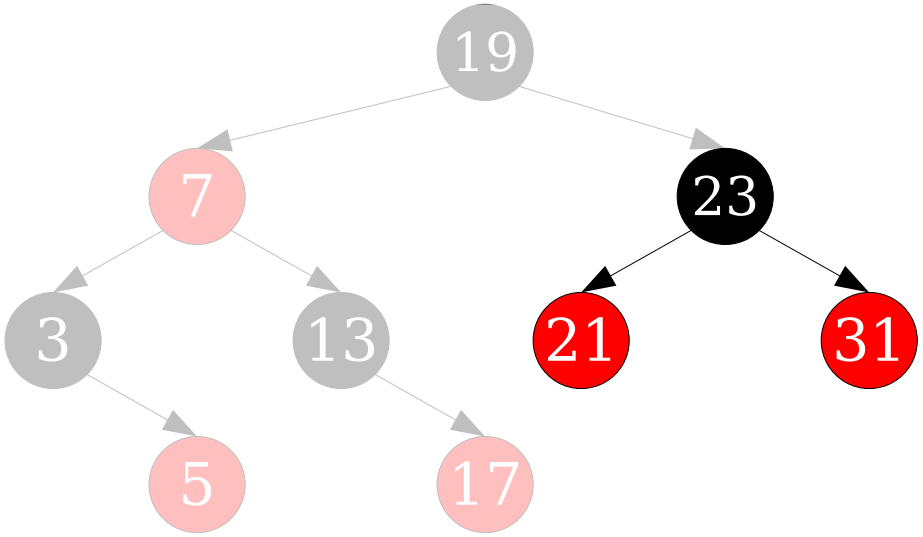

add
21


add
21

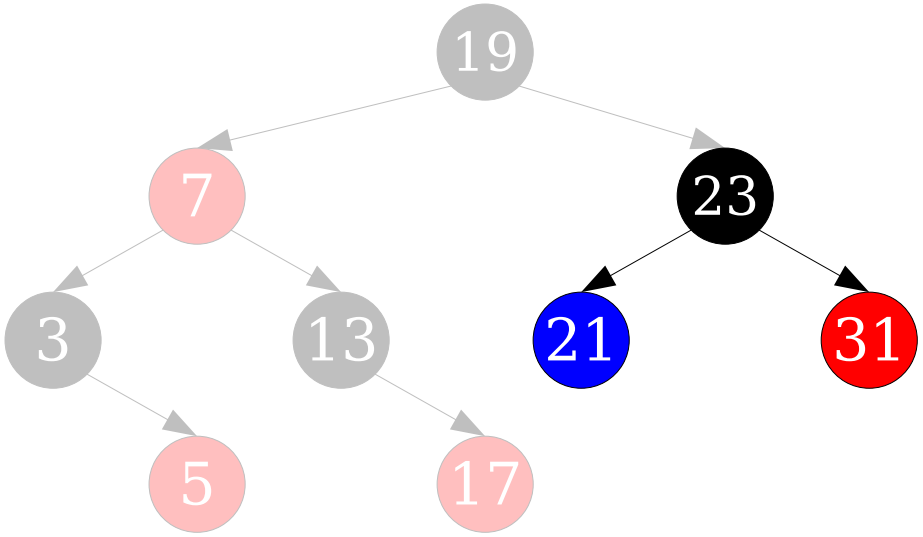

19
7
23
3
13
21
31
5
17

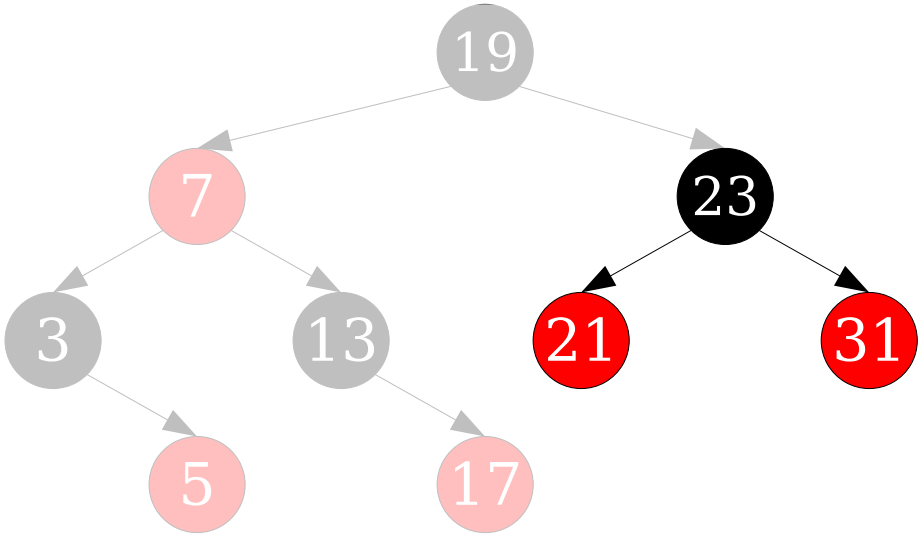

7
19
3
5
13
17
21
23
31



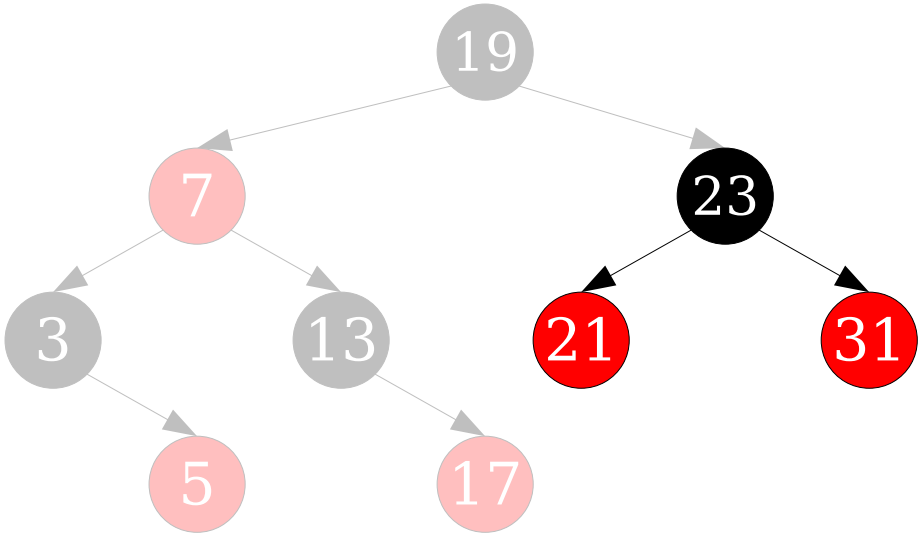


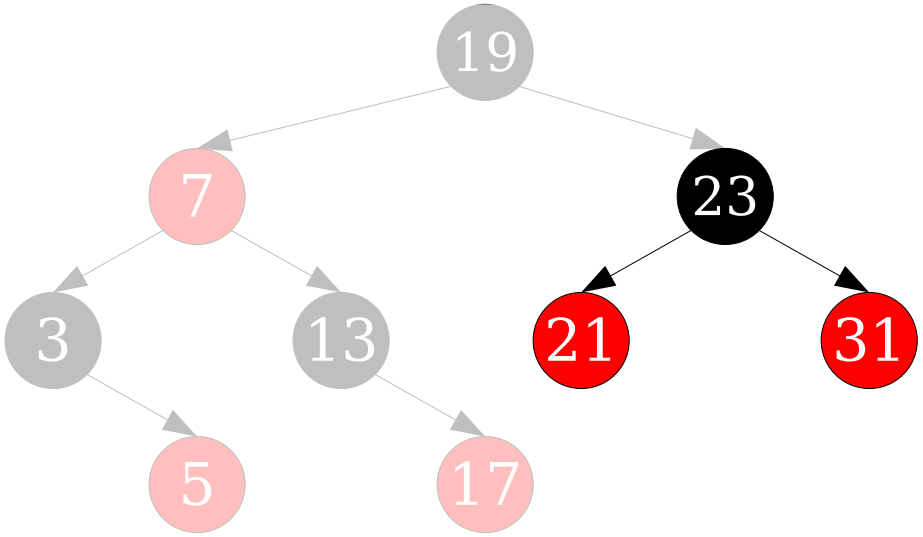
Goal



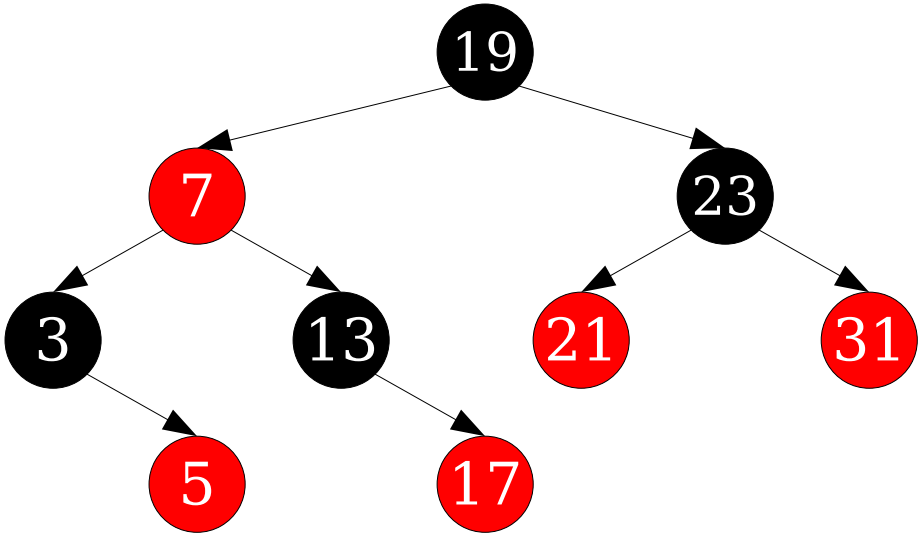


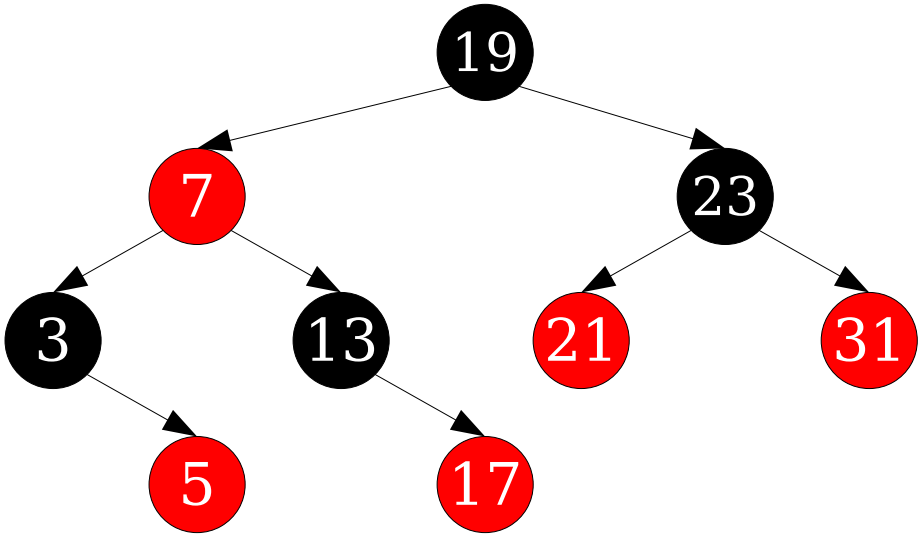
Goal





Goal

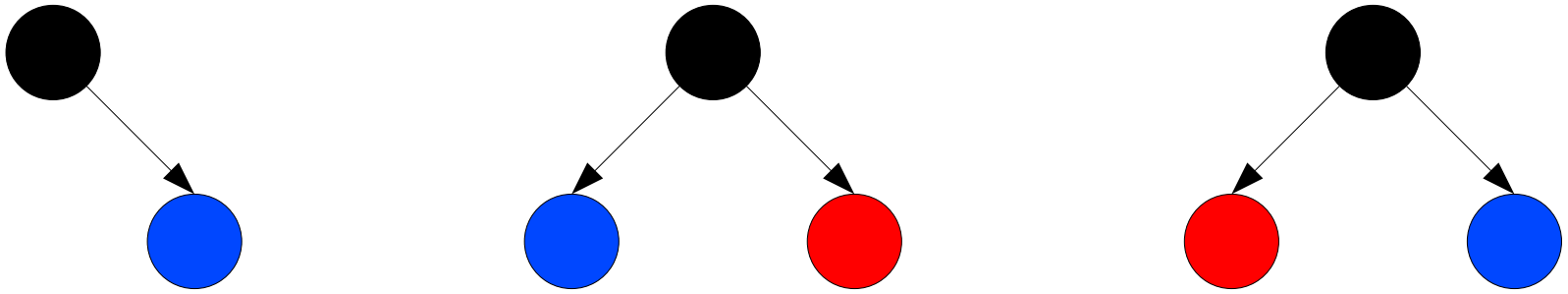


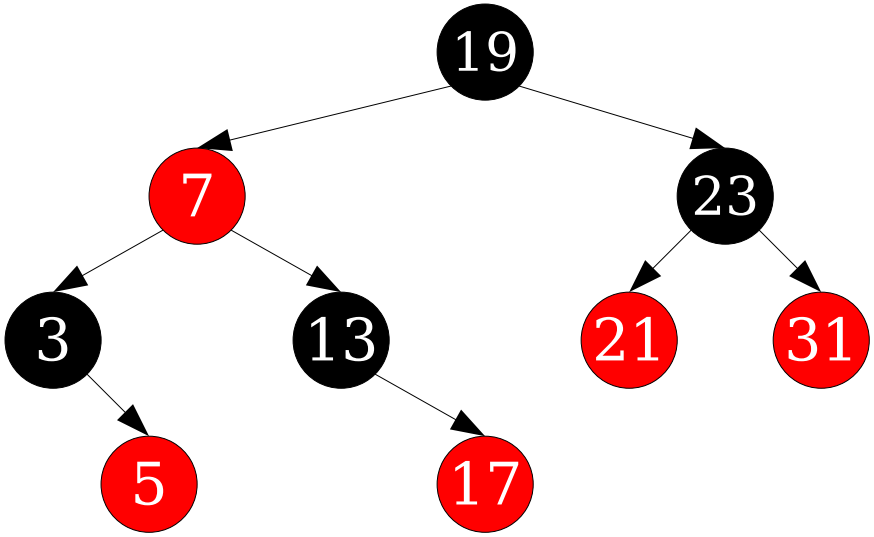


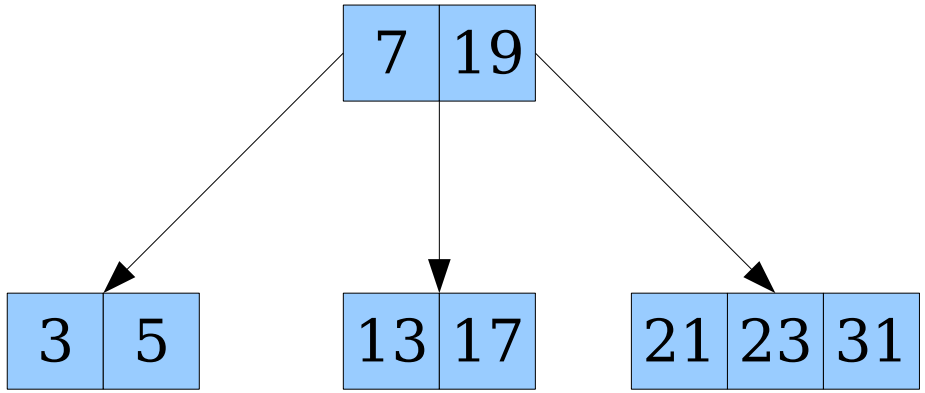
Goal

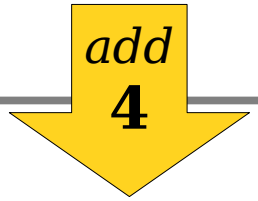
Red/Black Tree Insertion

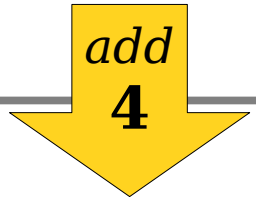
- ***Rule #1:*** When inserting a node, if its parent is black, make the node red and stop.
- ***Justification:*** This simulates inserting a key into an existing 2-node or 3-node.

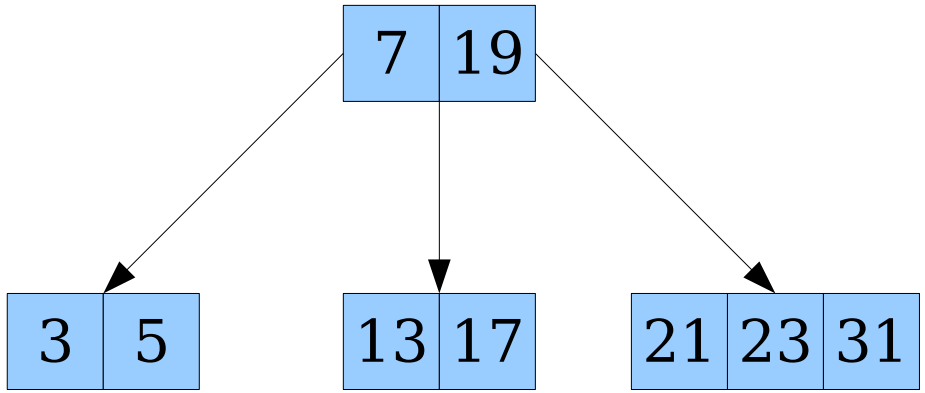


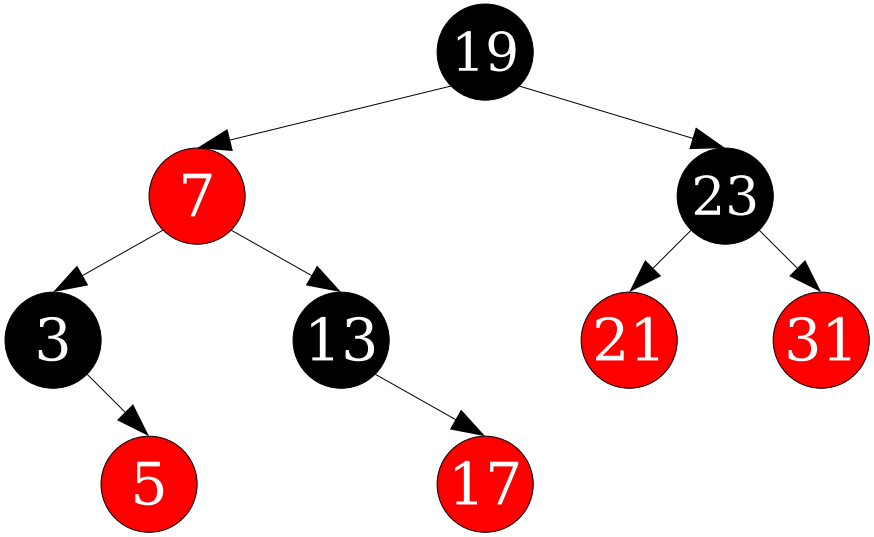

19
7
23
3
13
21
31
5
17

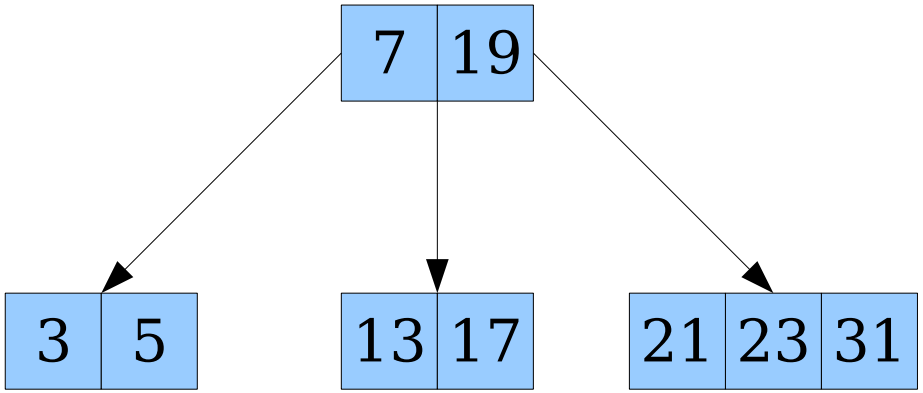

7
19
3
5
13
17
21
23
31

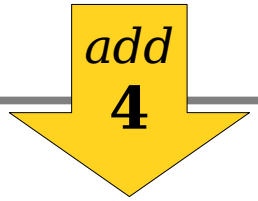

add
4

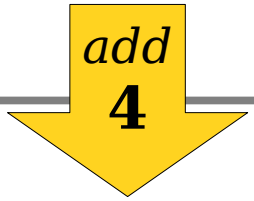

add
4

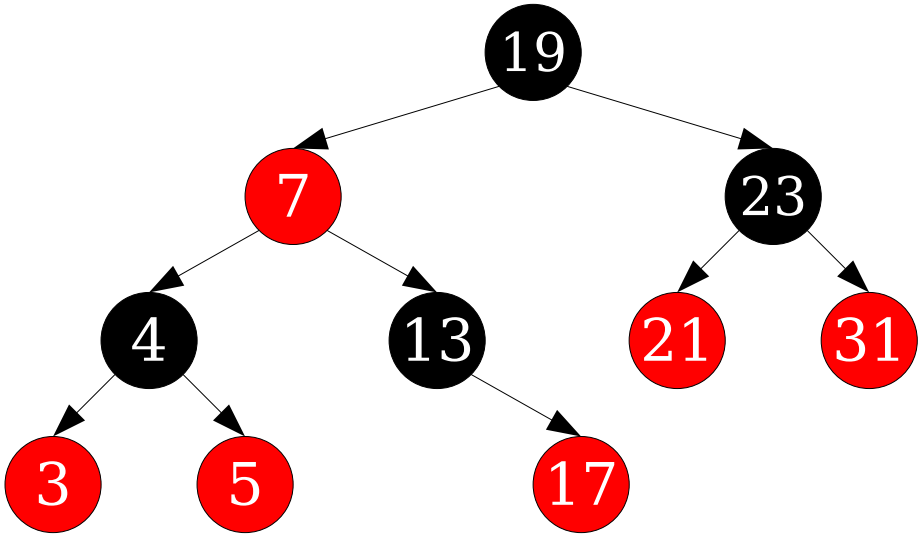

7
19
3
5
13
17
21
23
31

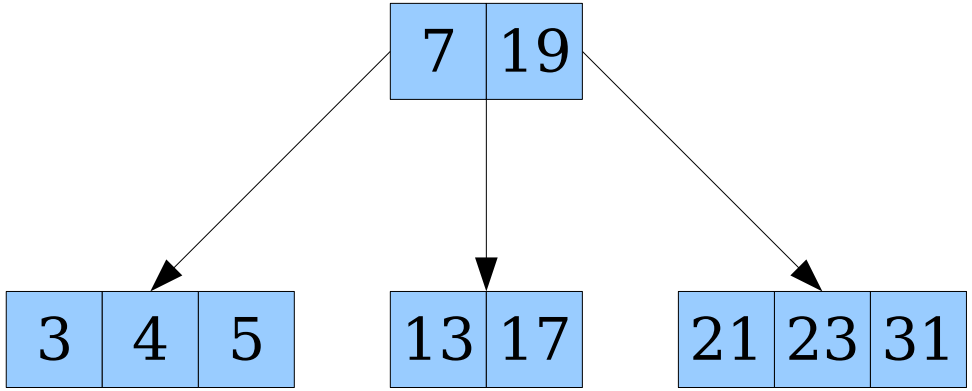

19
7
23
3
13
21
31
5
17

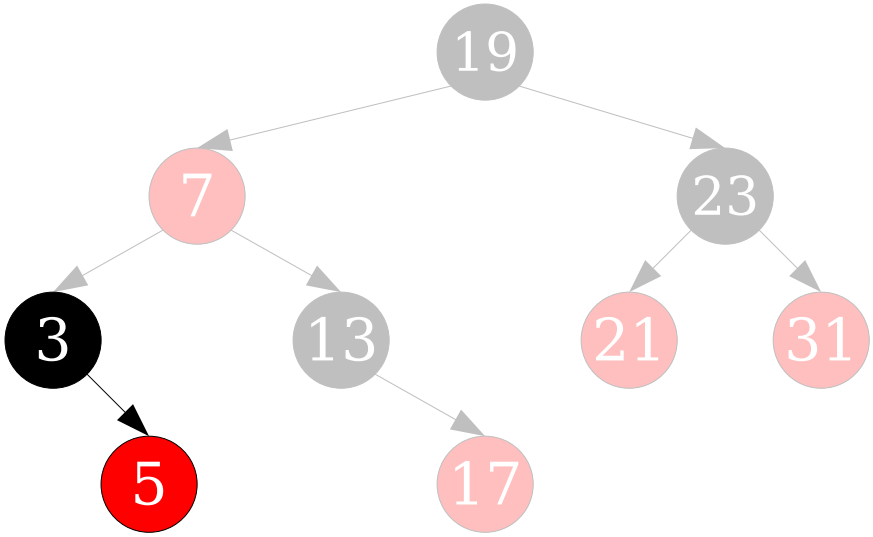

7
19
3
5
13
17
21
23
31

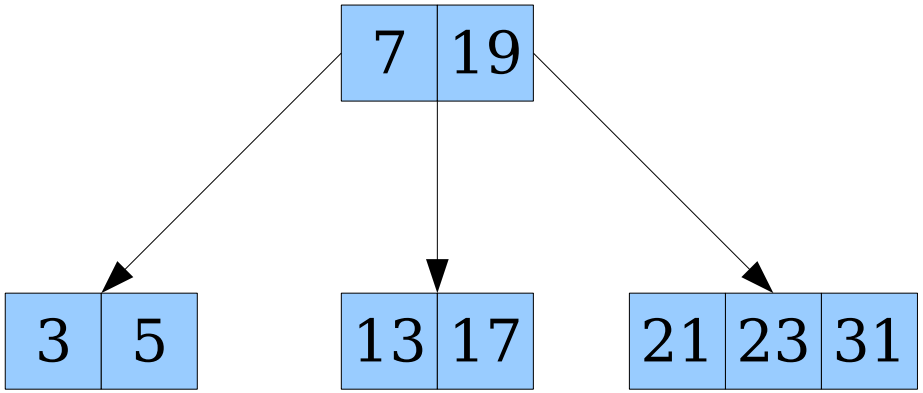

add
4

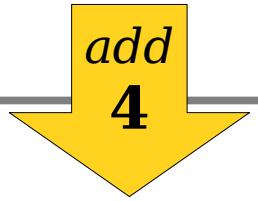

add
4

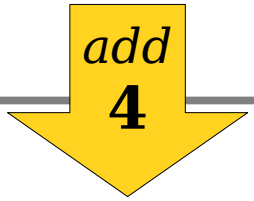

19
7
23
4
13
21
31
3
5
17

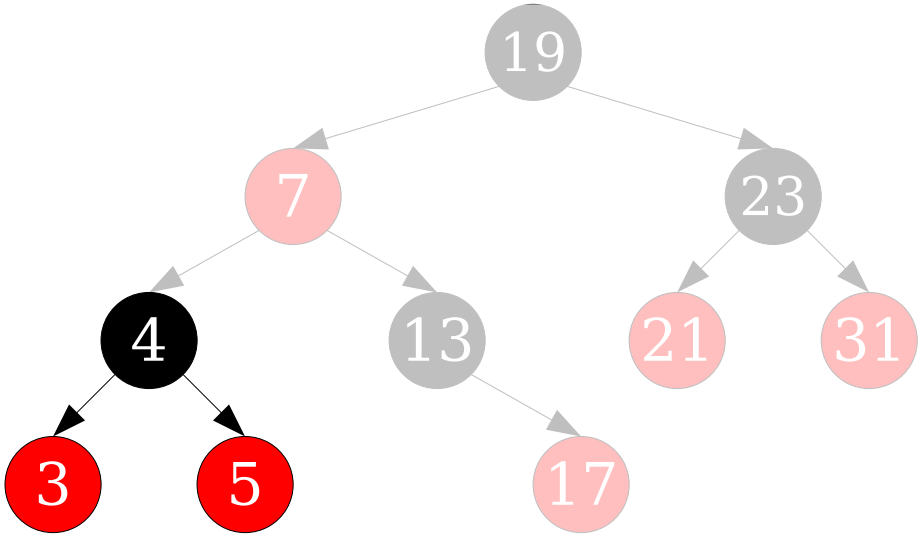

7
19
3
4
5
13
17
21
23
31

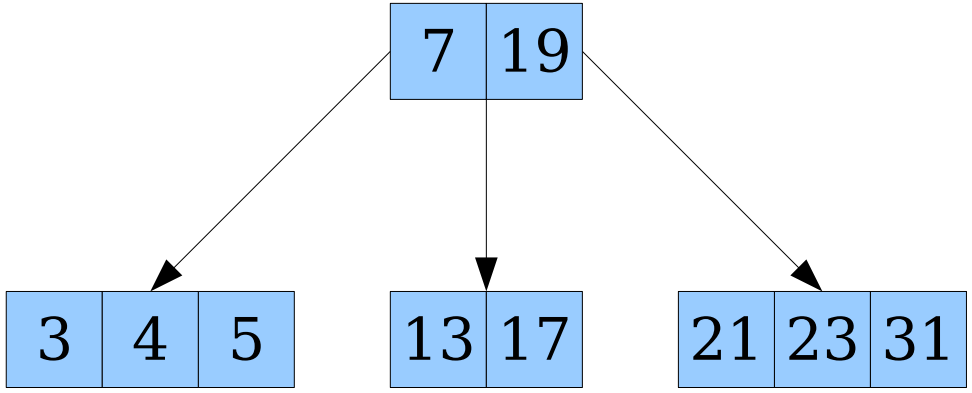

19
7
23
3
13
21
31
5
17

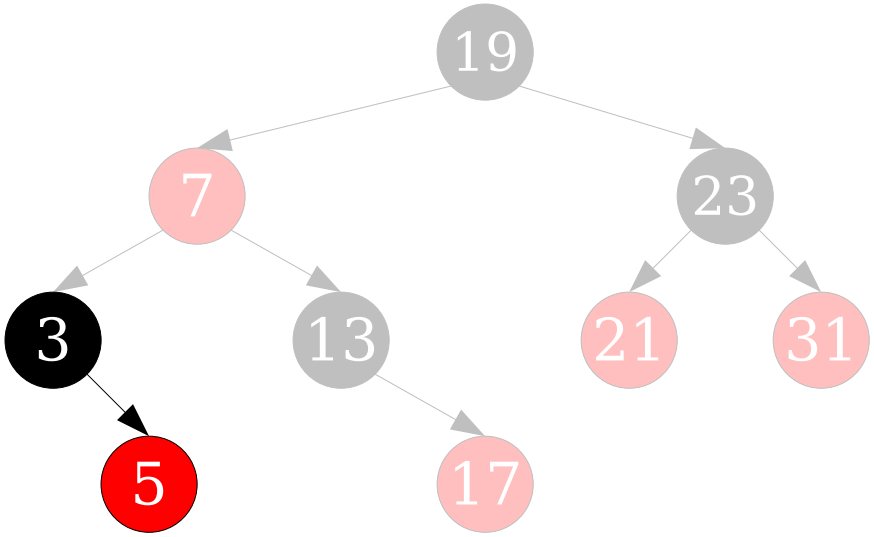

7
19
3
5
13
17
21
23
31

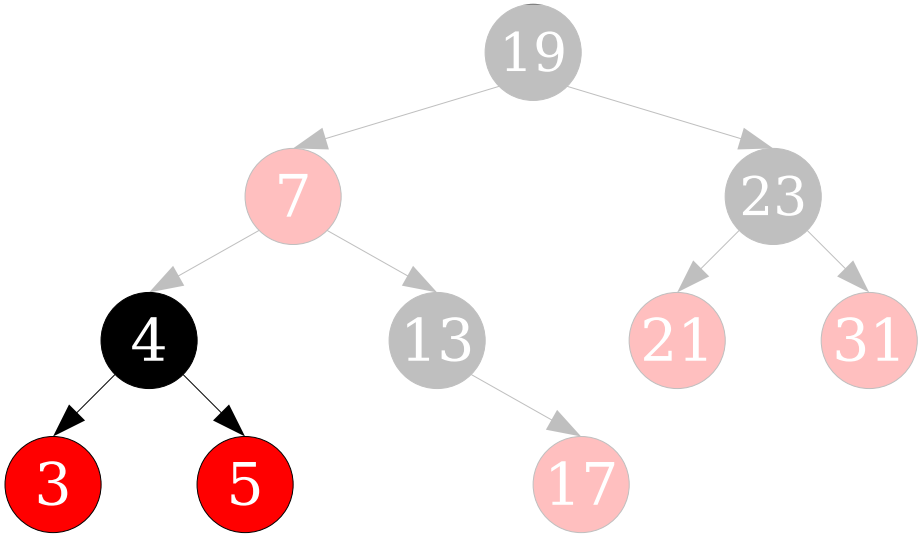

add
4


add
4

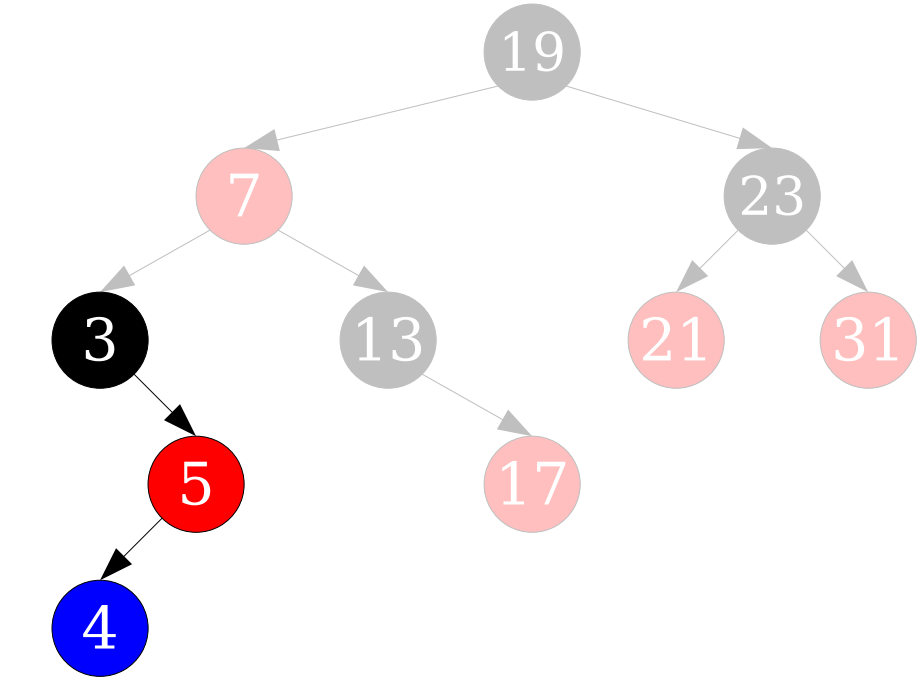

19
7
23
4
13
21
31
3
5
17


7
19
3
4
5
13
17
21
23
31

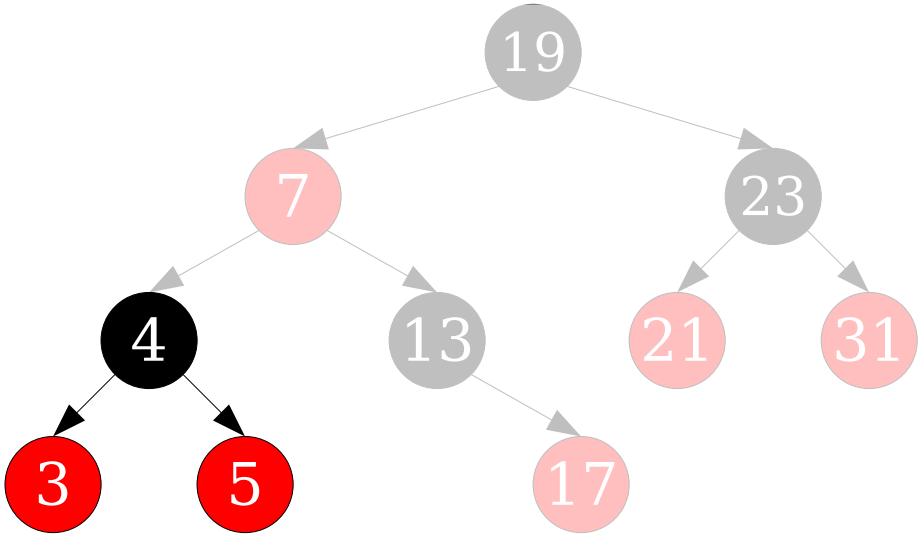




Goal

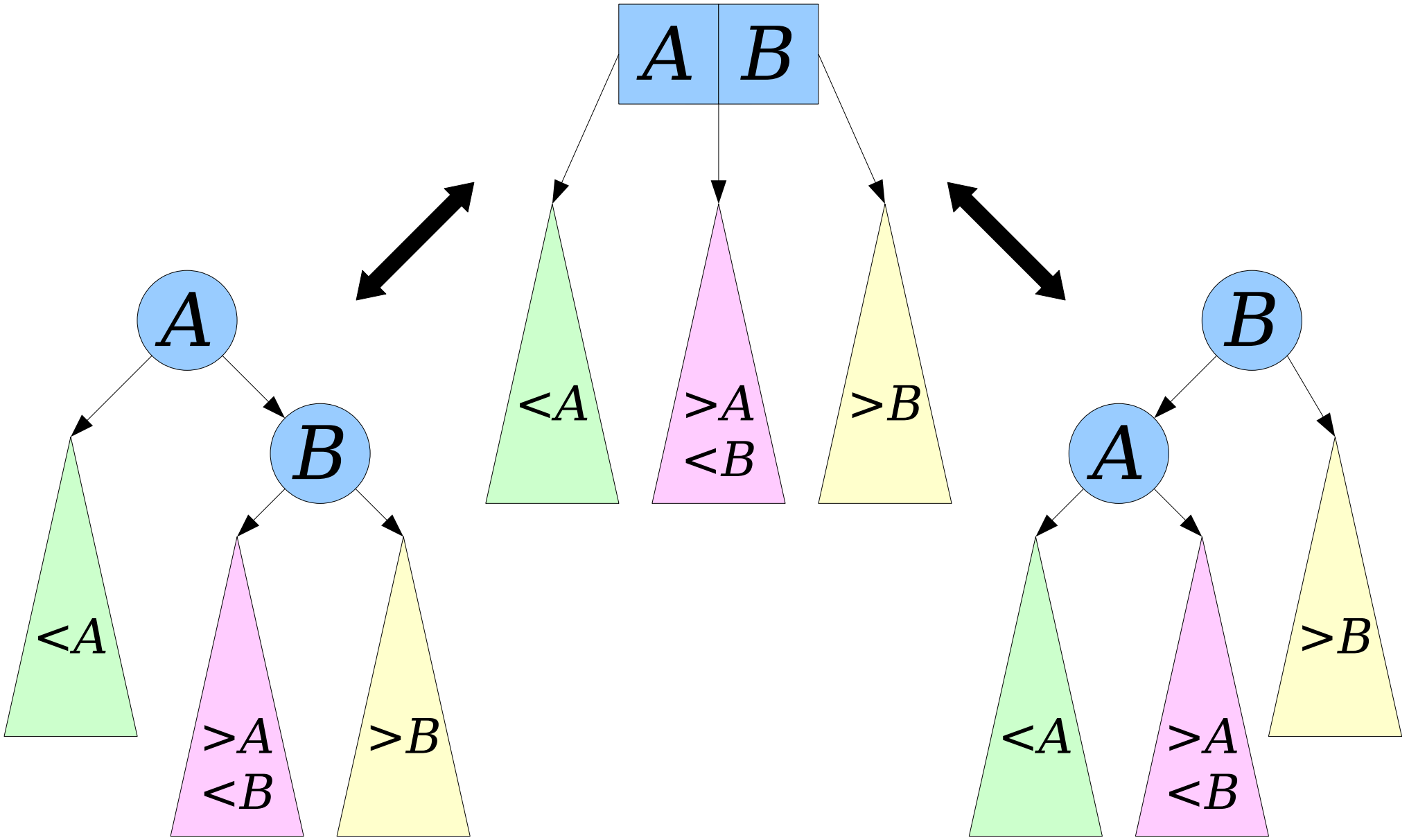


We need to move nodes around in a binary search tree. How do we do this?

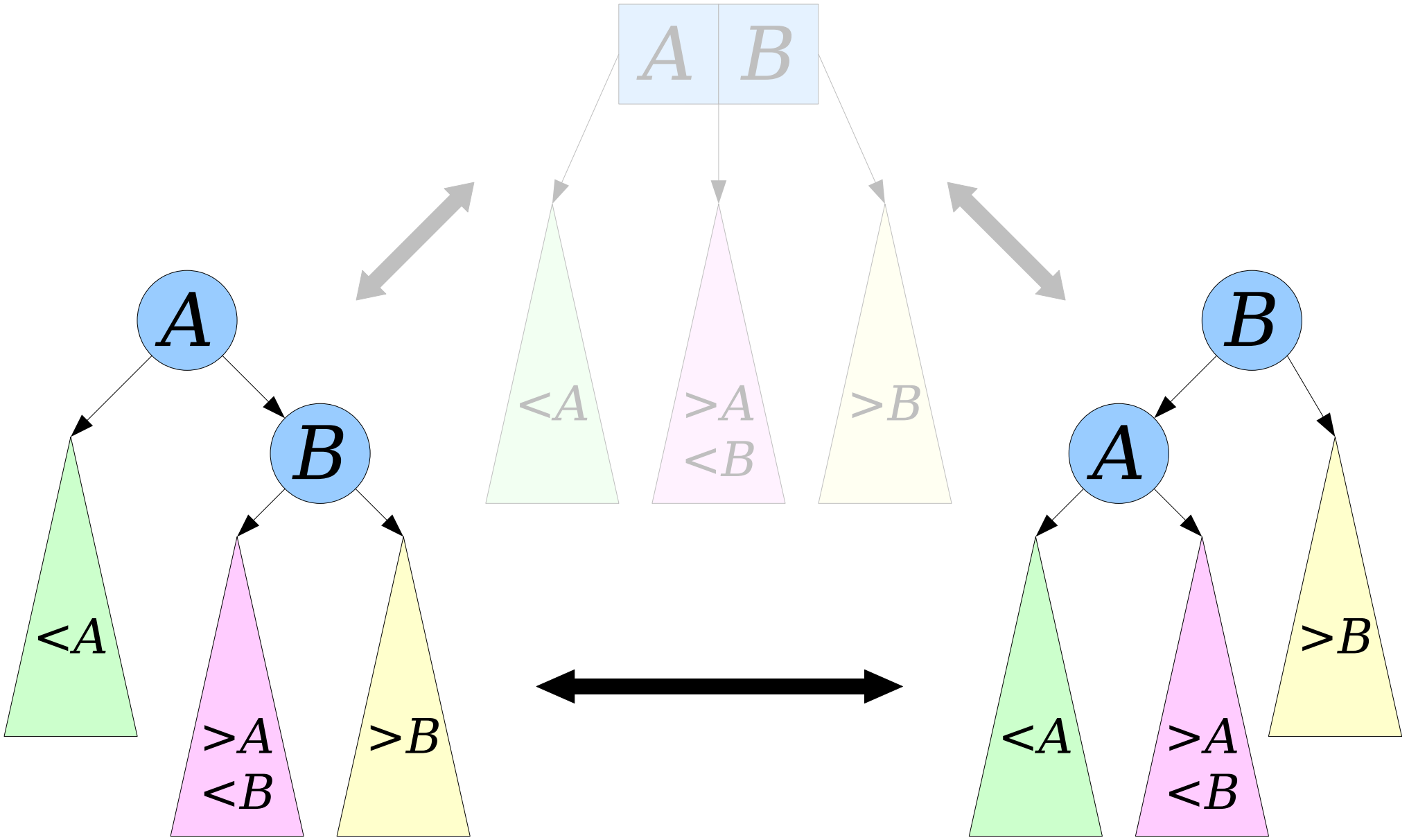


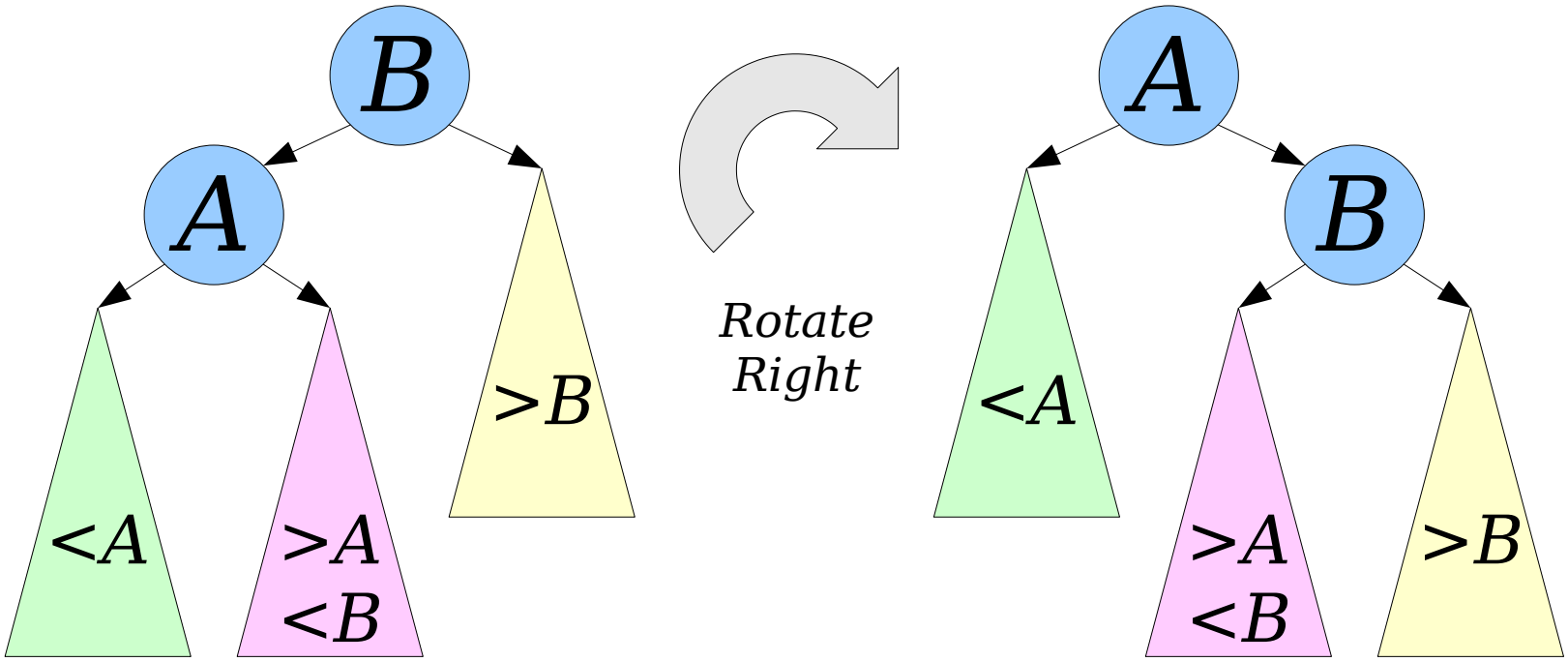
Goal

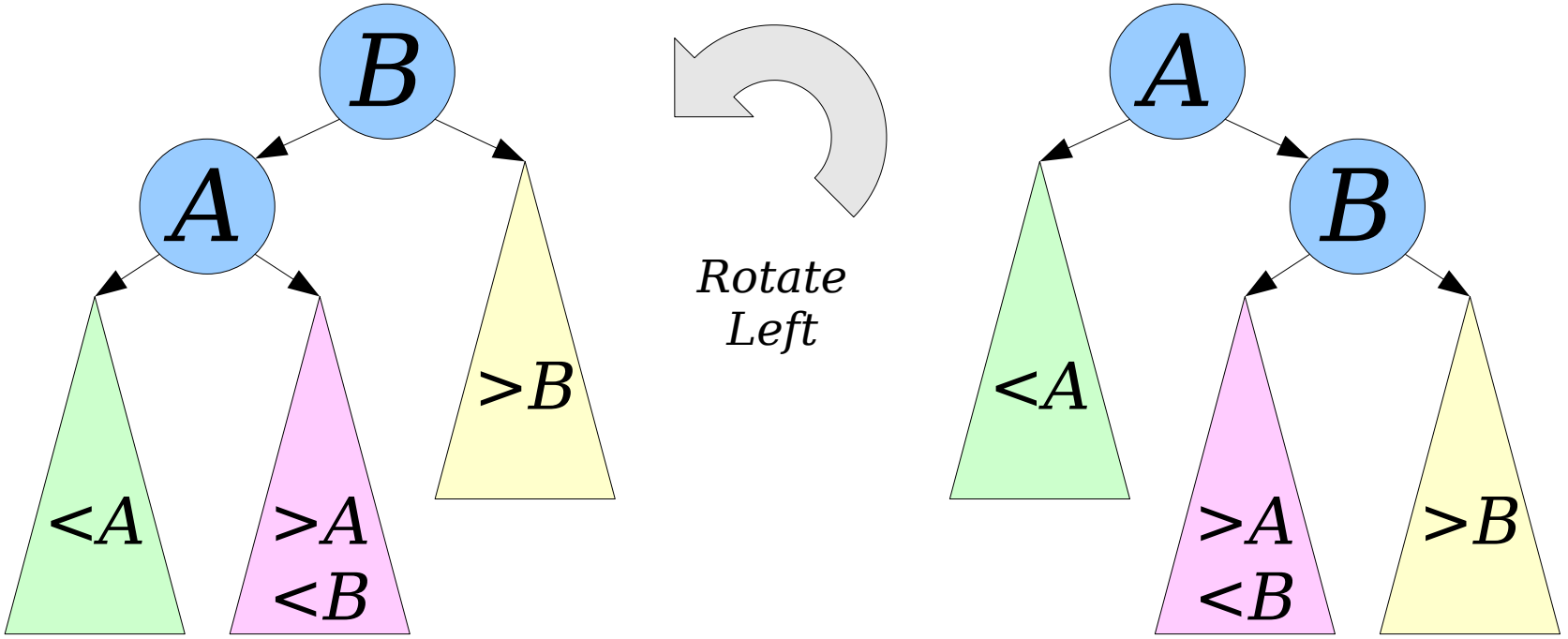
Tree Rotations

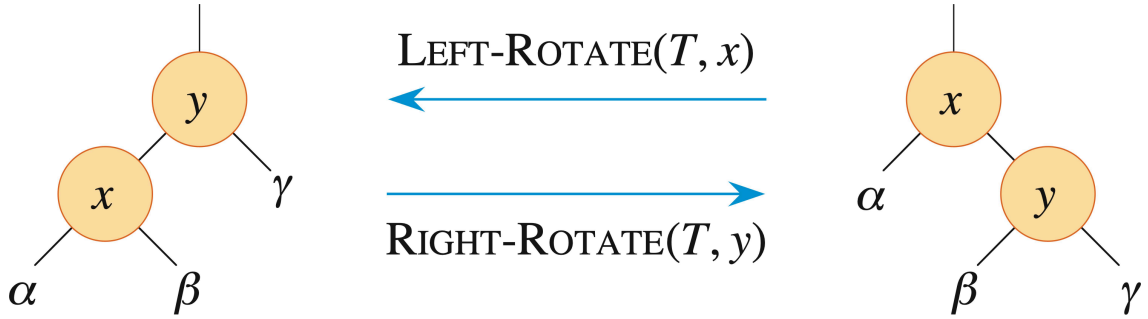


Tree Rotations



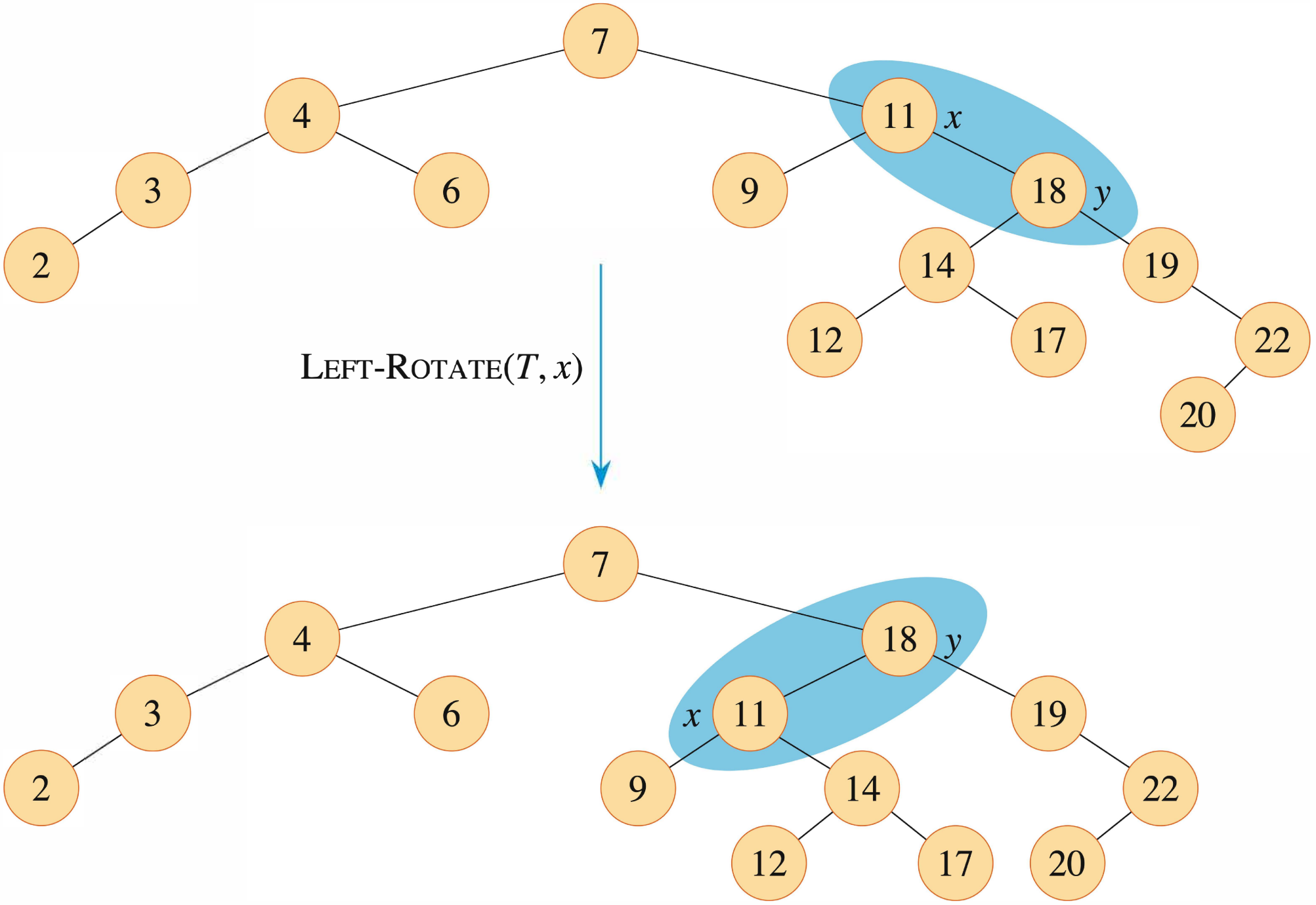

B
A
<A
>A
<B
>B
Rotate
Right
A
B
<A
>A
<B
>B

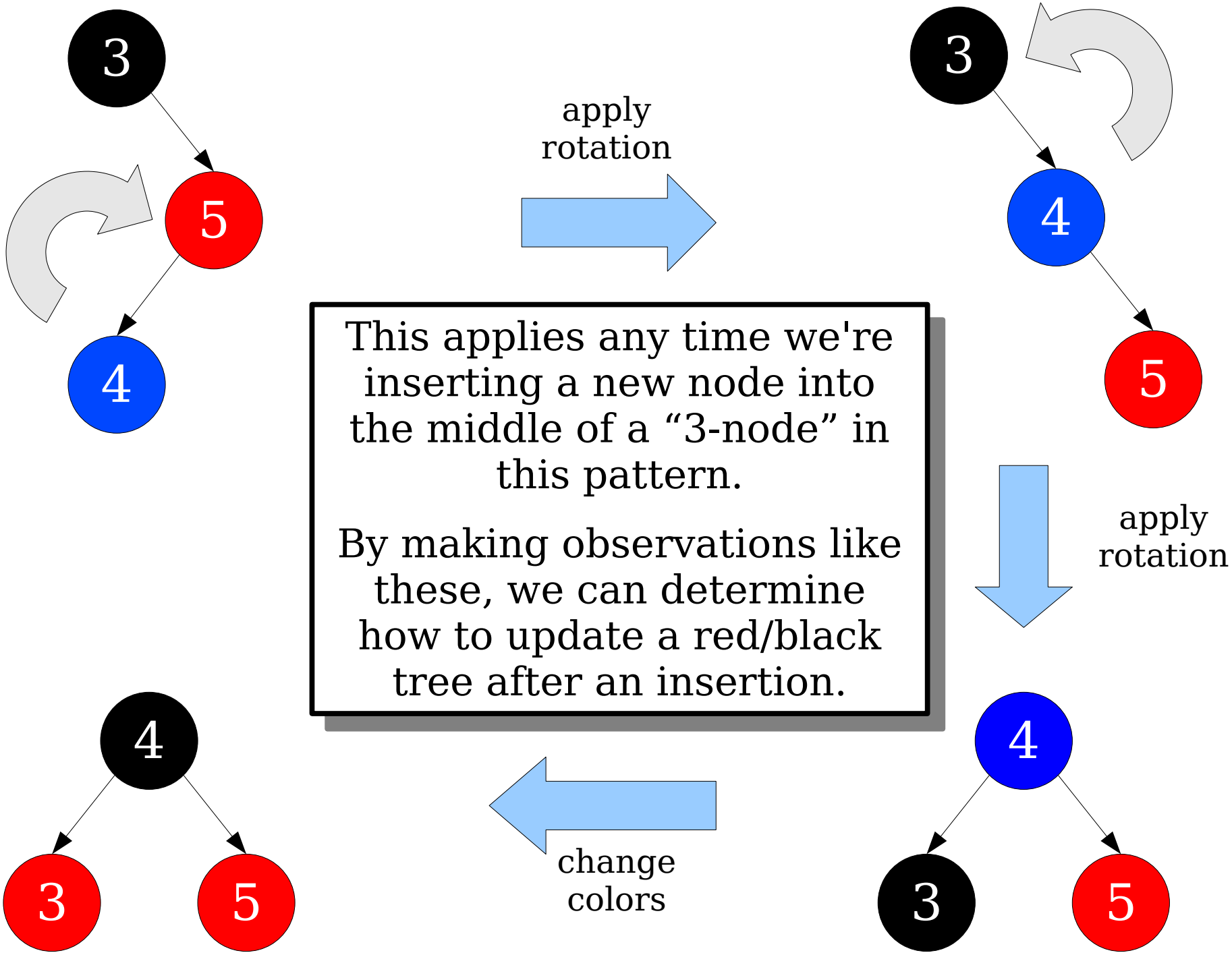

B
A
<A
>A
<B
>B
Rotate
Left
A
B
<A
>A
<B
>B

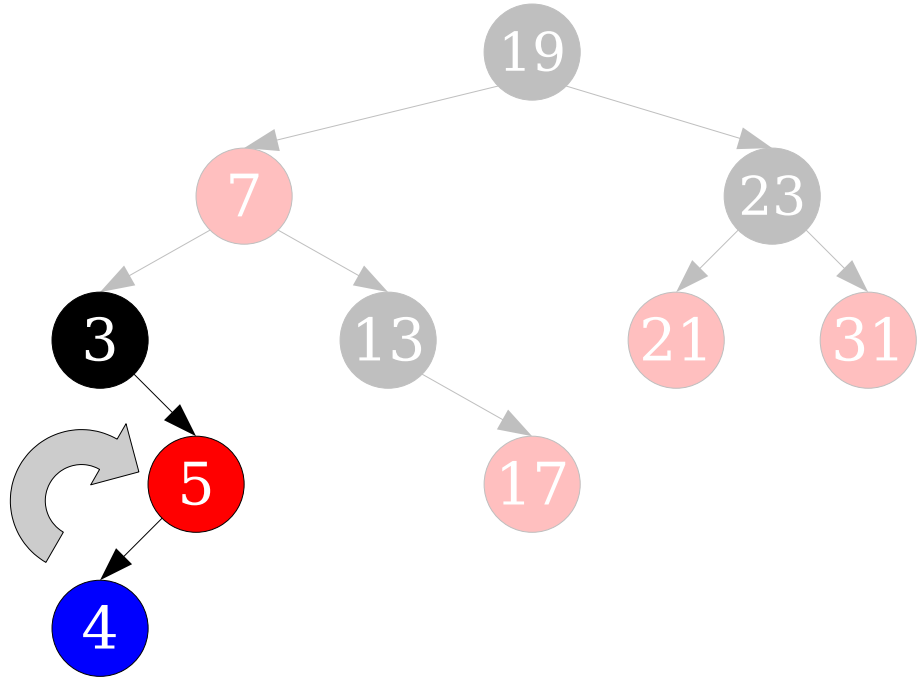


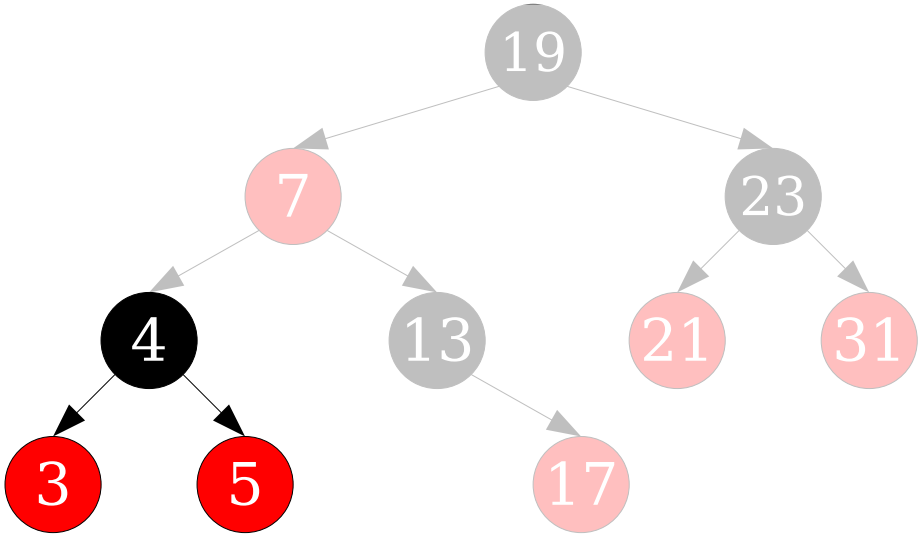
LEFT-ROTATE(T, x)

```
 1   y = x.right
 2   x.right = y.left          // turn y's left subtree into x's right subtree
 3   if y.left ≠ T.nil         // if y's left subtree is not empty ...
 4         y.left.p = x        // ... then x becomes the parent of the subtree's root
 5   y.p = x.p                 // x's parent becomes y's parent
 6   if x.p == T.nil           // if x was the root ...
 7         T.root = y          // ... then y becomes the root
 8   elseif x == x.p.left      // otherwise, if x was a left child ...
 9         x.p.left = y        // ... then y becomes a left child
10   else x.p.right = y        // otherwise, x was a right child, and now y is
11   y.left = x                // make x become y's left child
12   x.p = y
```

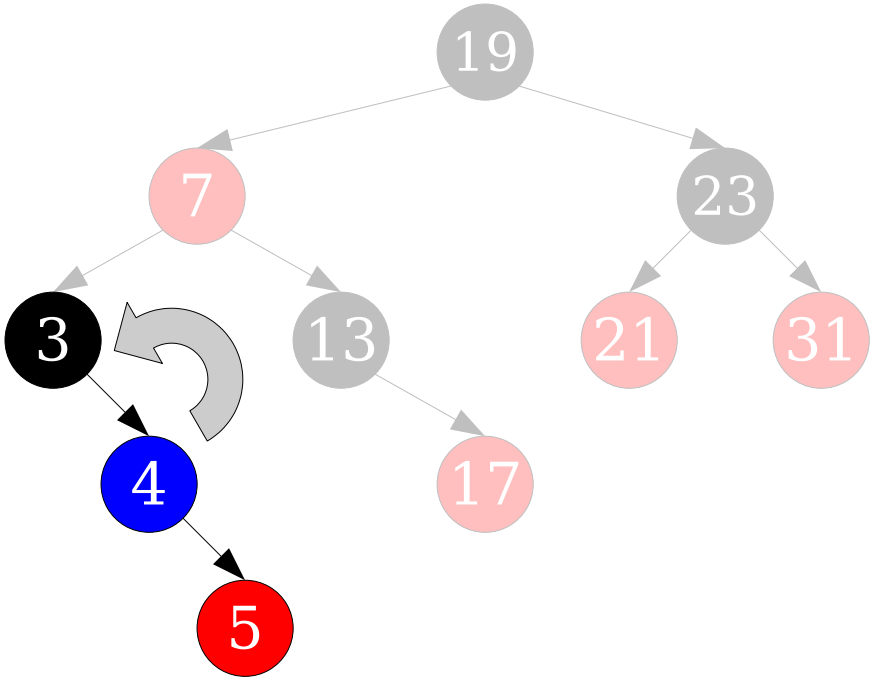

7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

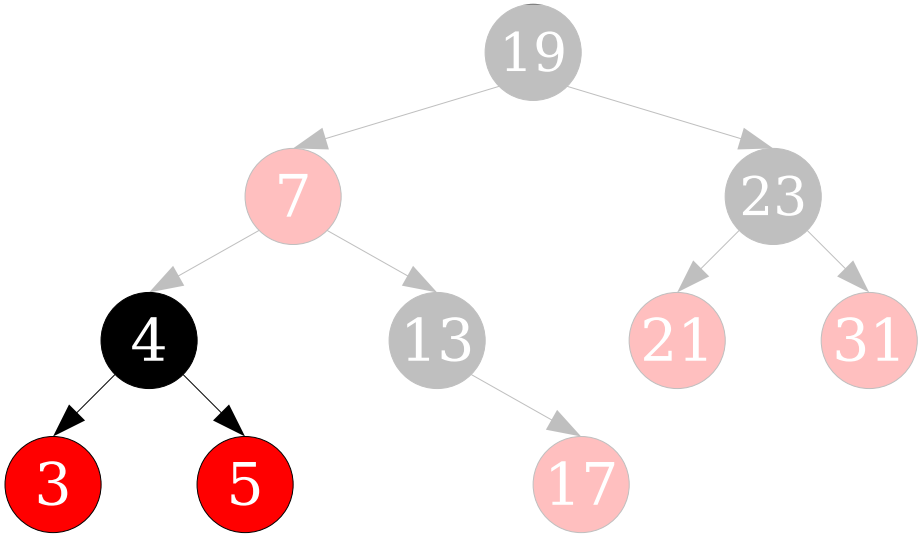

3
5
4
apply
rotation
3
4
5
This applies any time we're inserting a new node into the middle of a “3-node” in this pattern.
By making observations like these, we can determine how to update a red/black tree after an insertion.
apply
rotation
4
3
5
change
colors
4
3
5



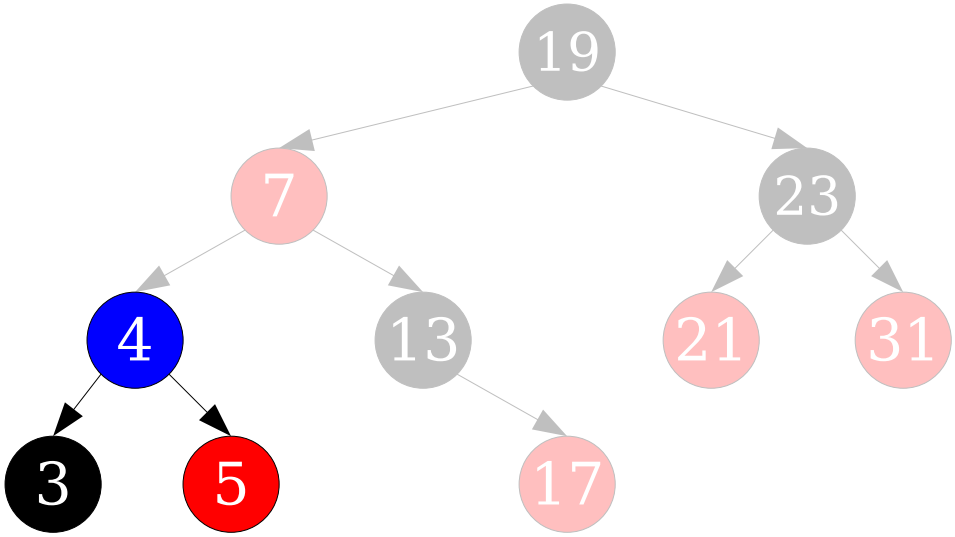


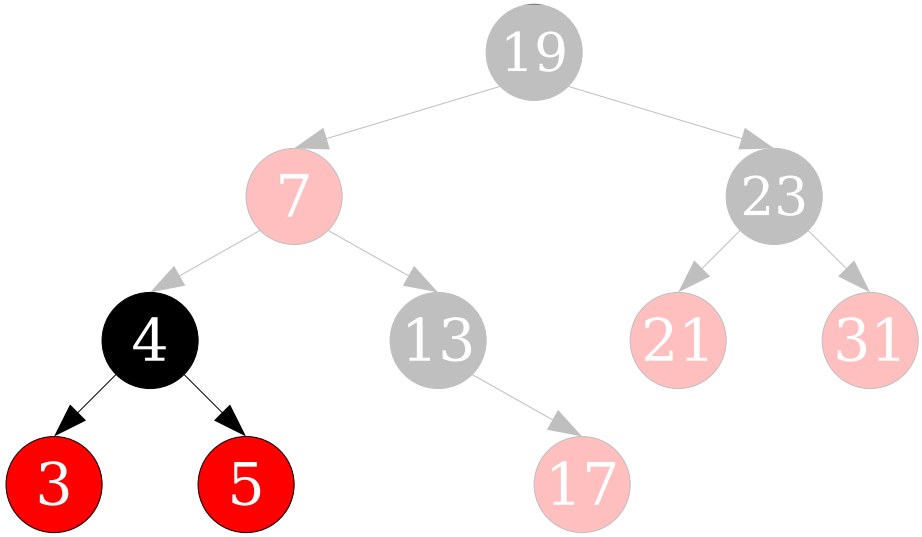
Goal



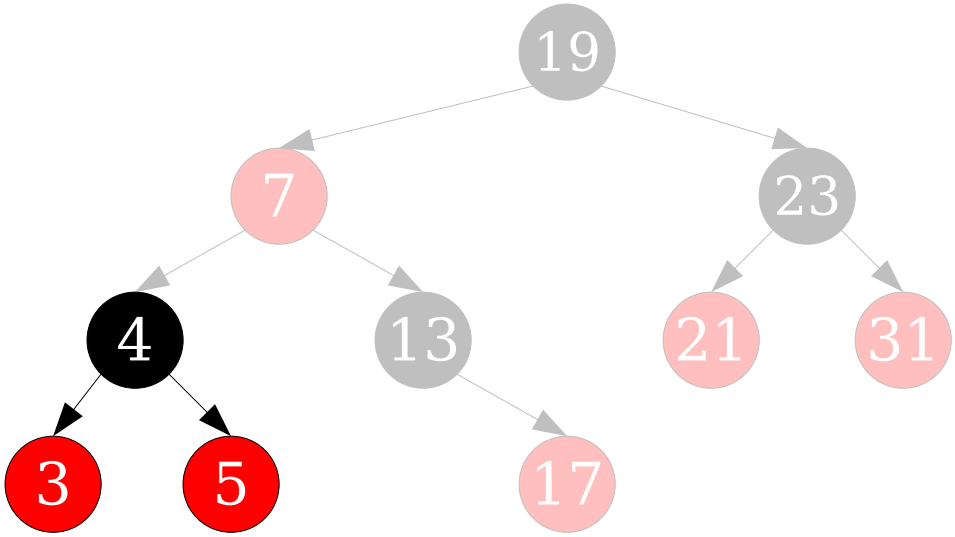


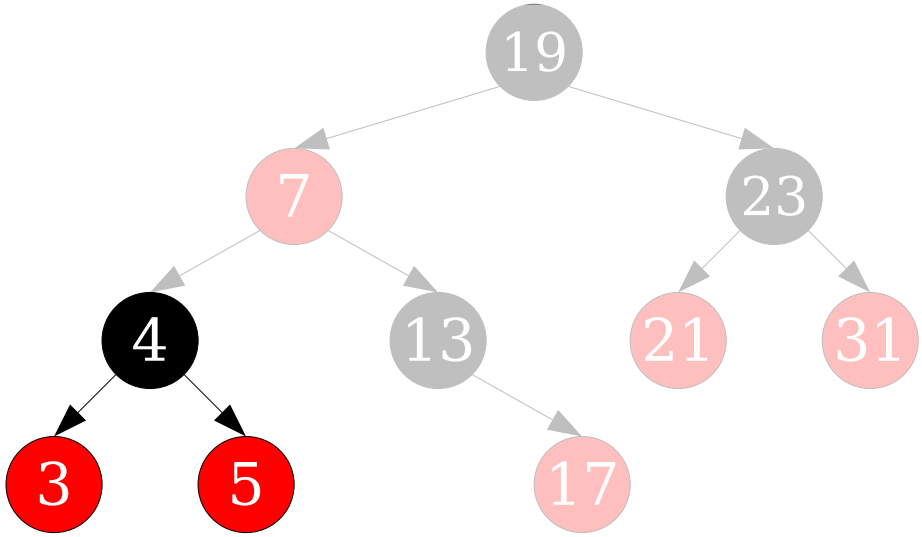
Goal



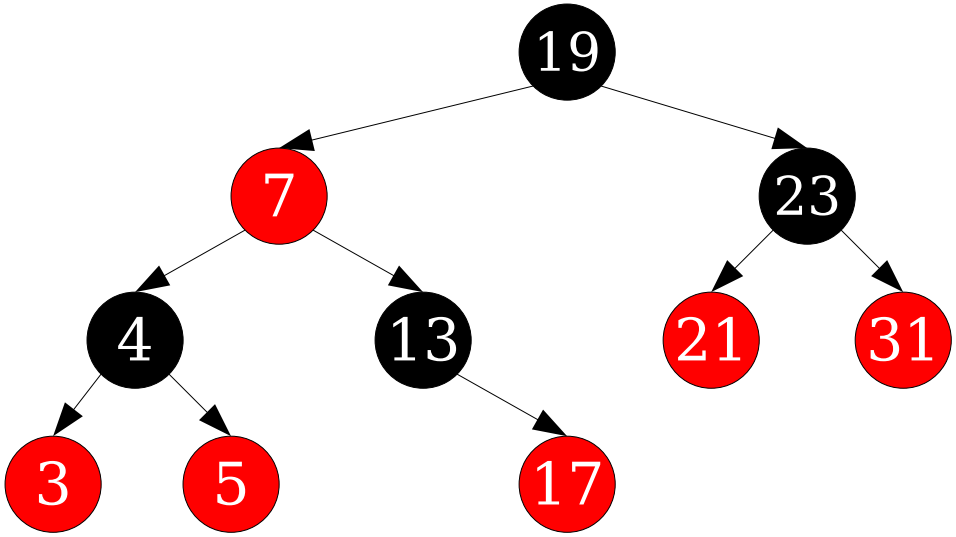


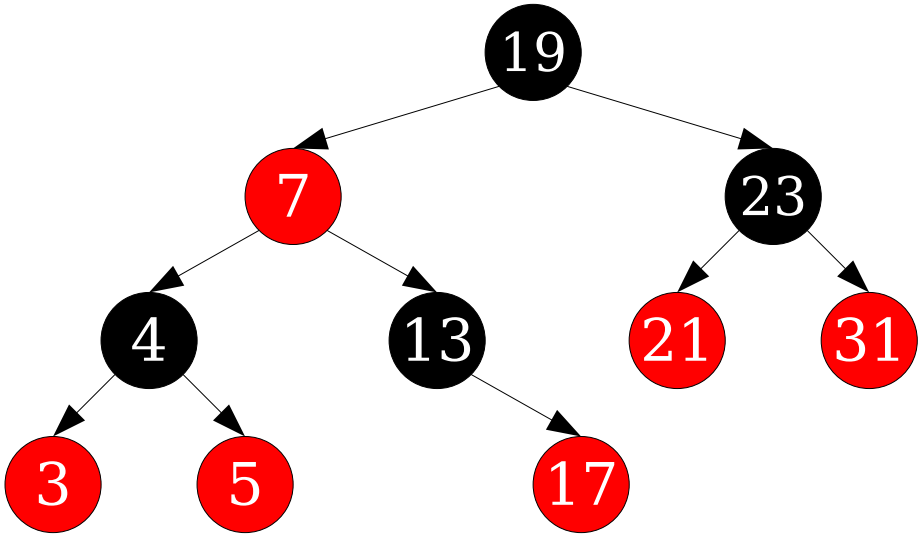
Goal



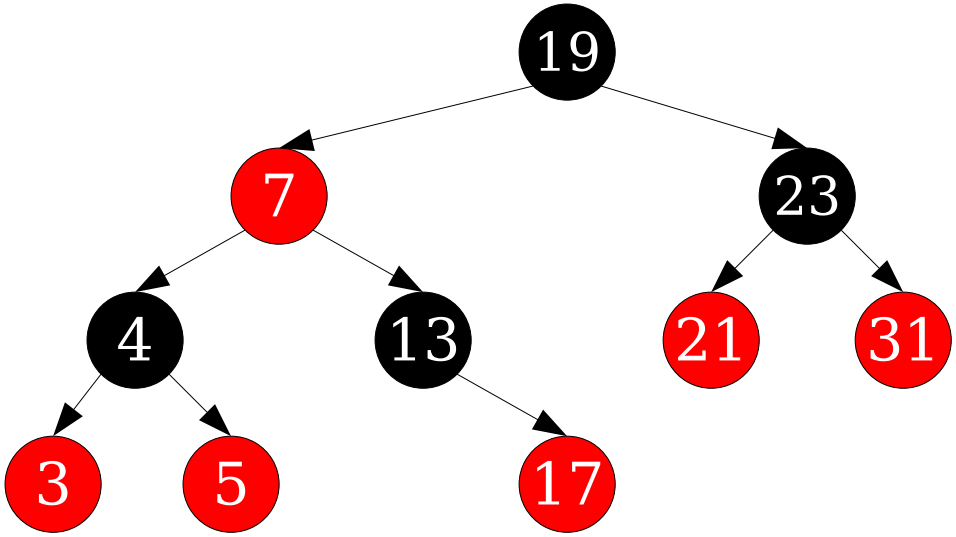


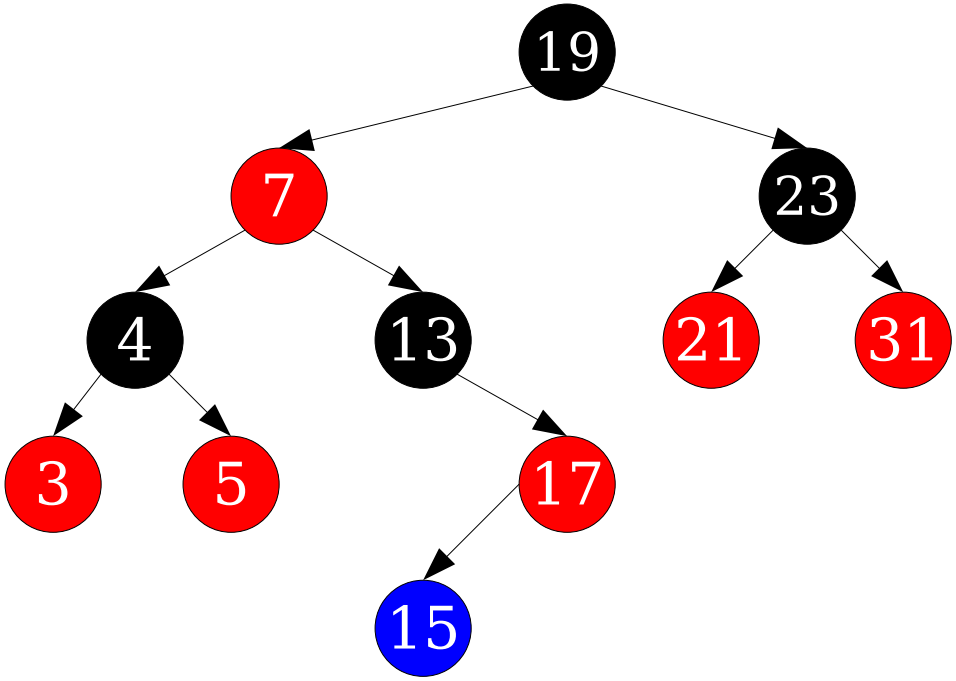
Goal

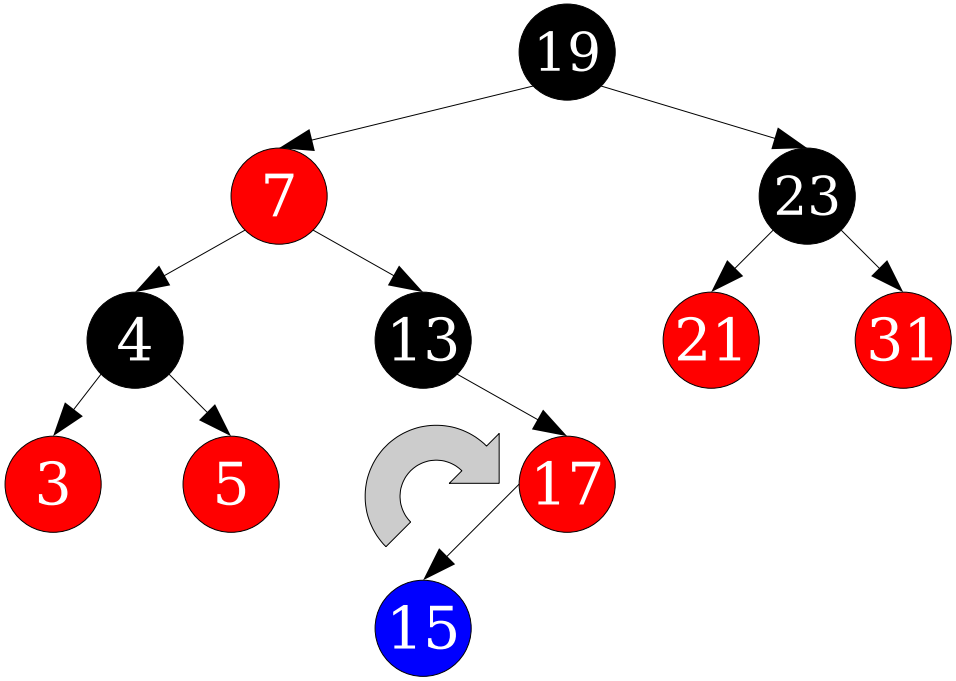


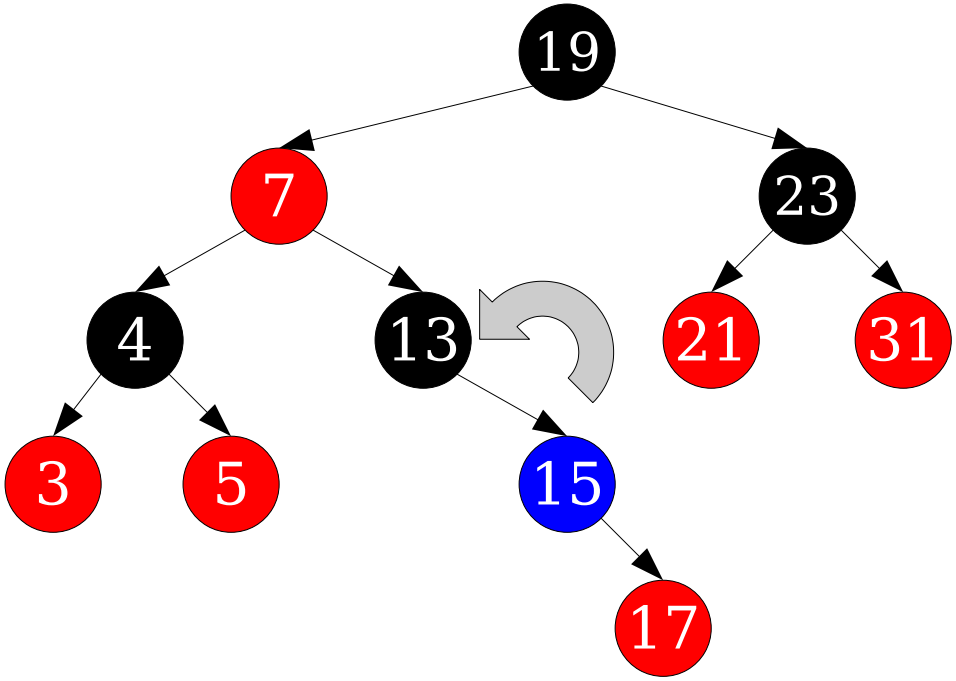


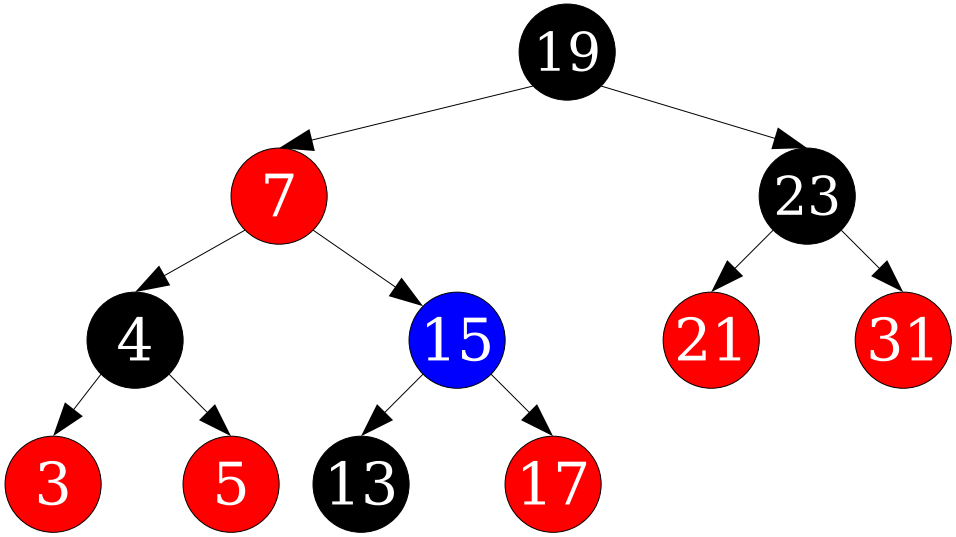
Goal

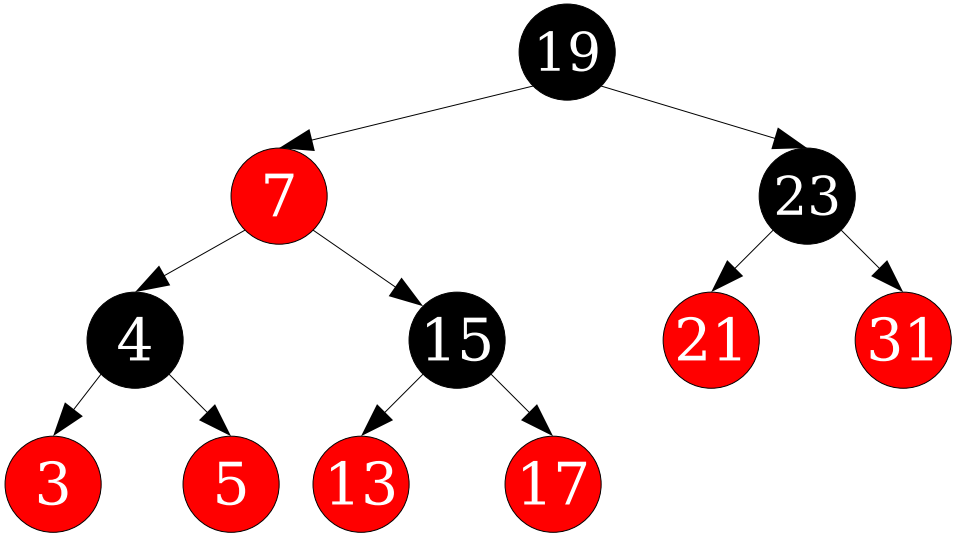

19
7
23
4
13
21
31
3
5
17

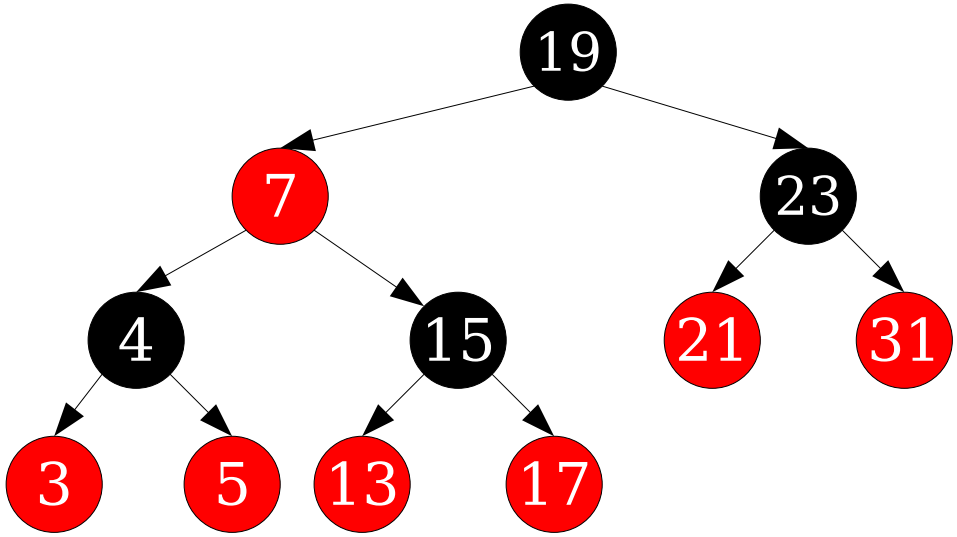

19
7
23
4
13
21
31
3
5
17
15


19
7
23
4
13
21
31
3
5
17
15

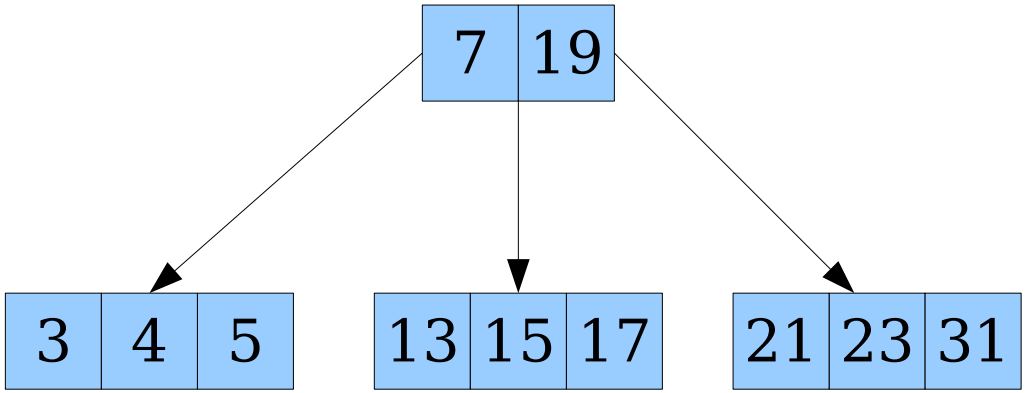

19
7
23
4
13
21
31
3
5
15
17


19
7
23
4
15
21
31
3
5
13
17

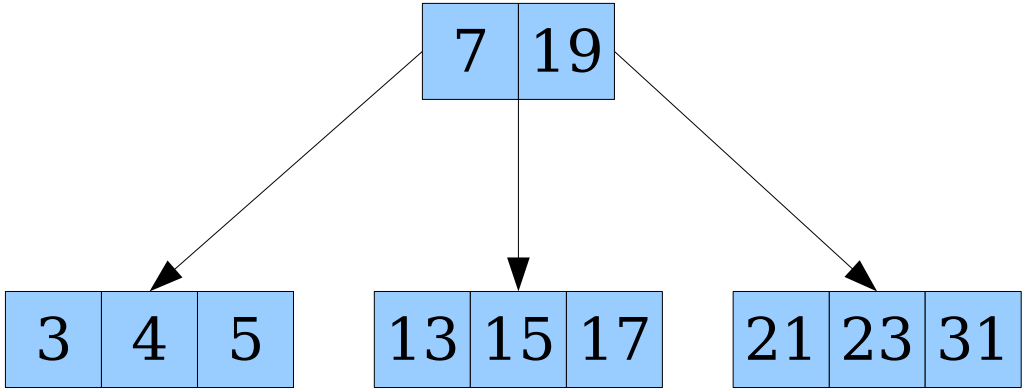

19
7
23
4
15
21
31
3
5
13
17

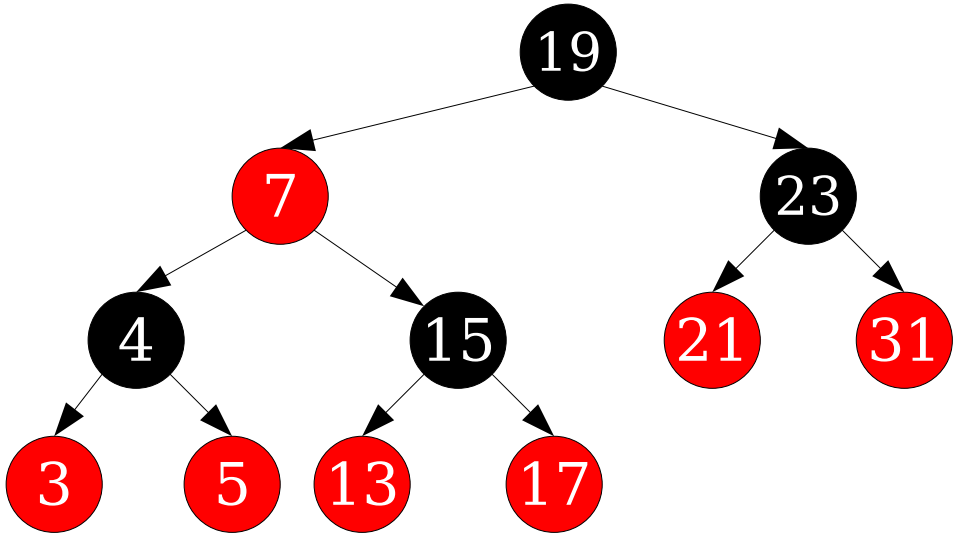

19
7
23
4
15
21
31
3
5
13
17

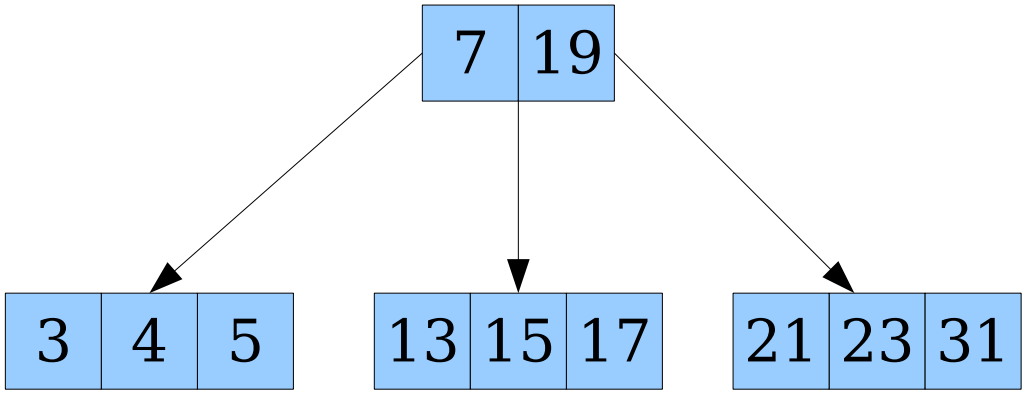
add **16**

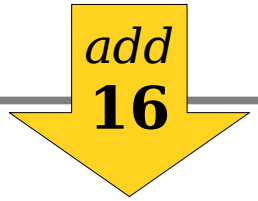

7
19
3
4
5
13
15
17
21
23
31

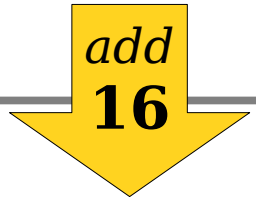
add **16**

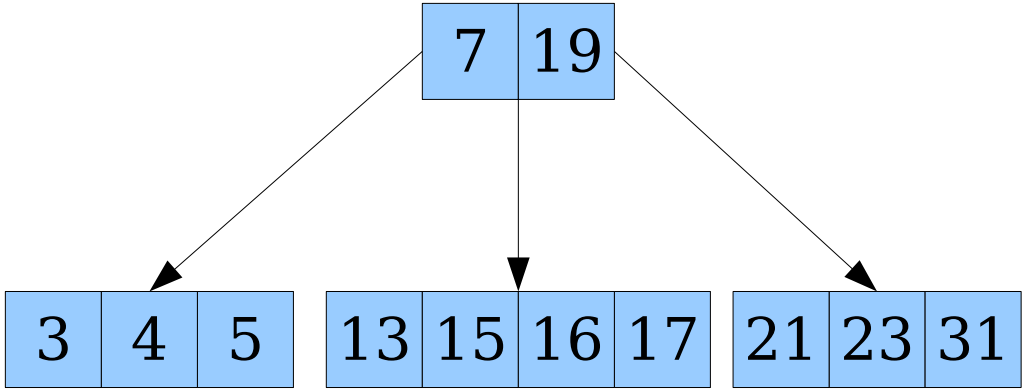

7
19
3
4
5
13
15
17
21
23
31

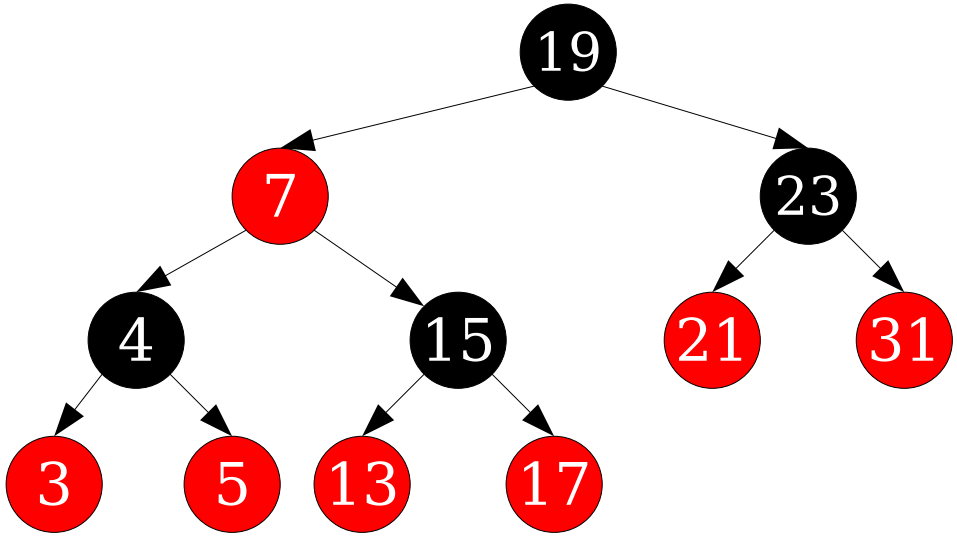

19
7
23
4
15
21
31
3
5
13
17


7
19
3
4
5
13
15
17
21
23
31

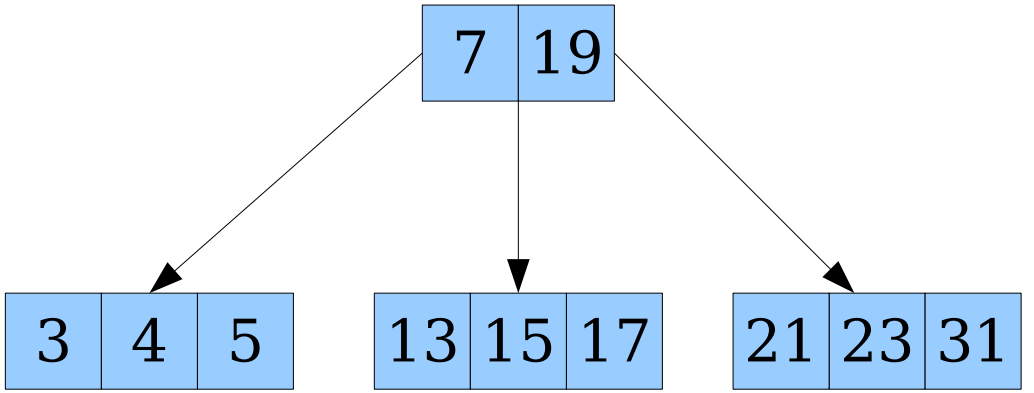

add
16


add
16

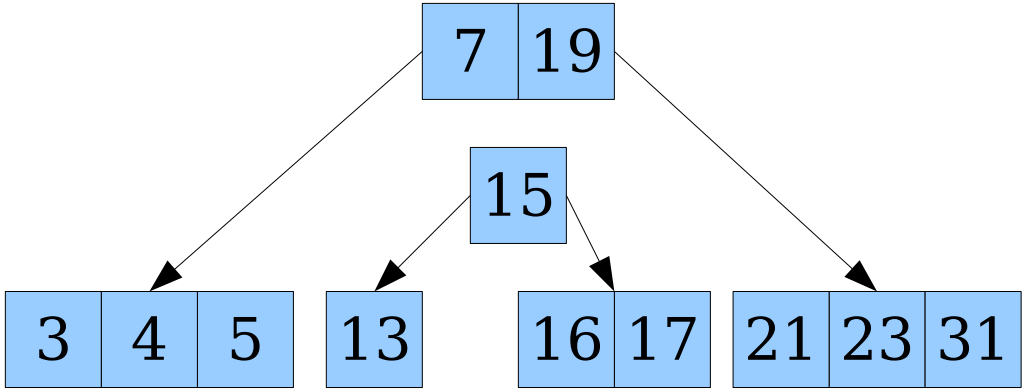

7
19
3
4
5
13
15
16
17
21
23
31

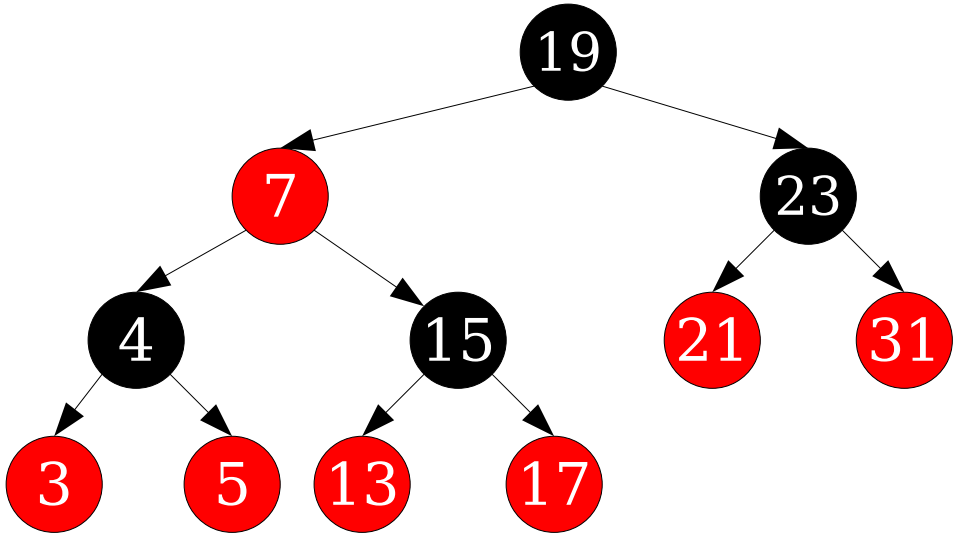

19
7
23
4
15
21
31
3
5
13
17

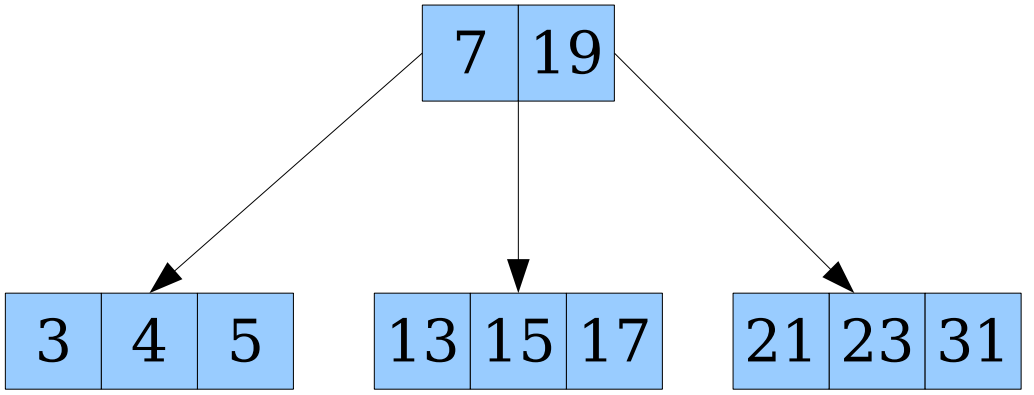
add **16**


7
19
3
4
5
13
15
17
21
23
31

add **16**

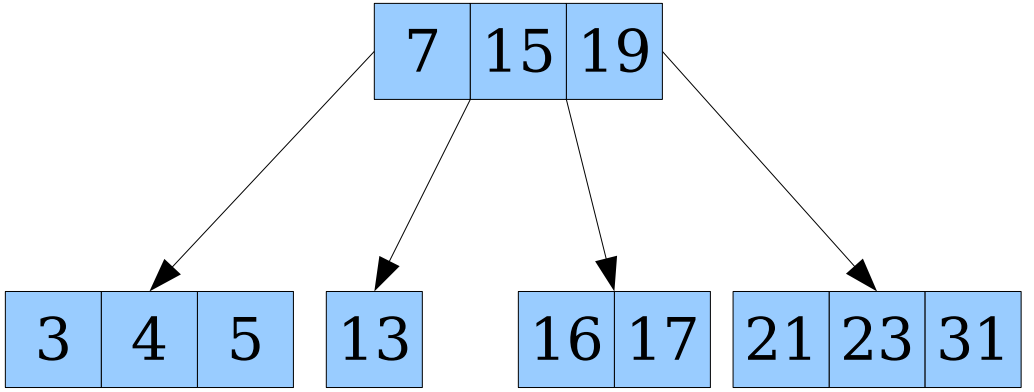

7
19
15
3
4
5
13
16
17
21
23
31

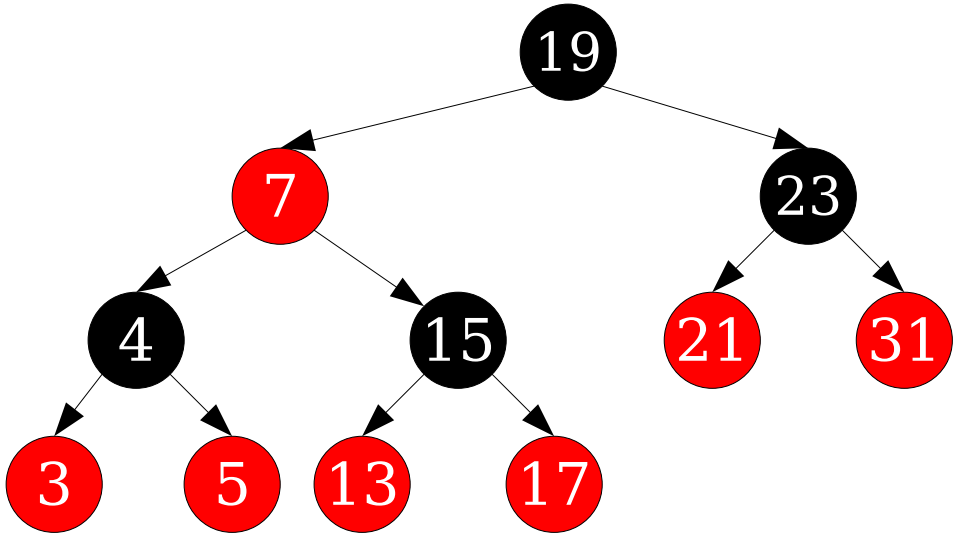

19
7
23
4
15
21
31
3
5
13
17

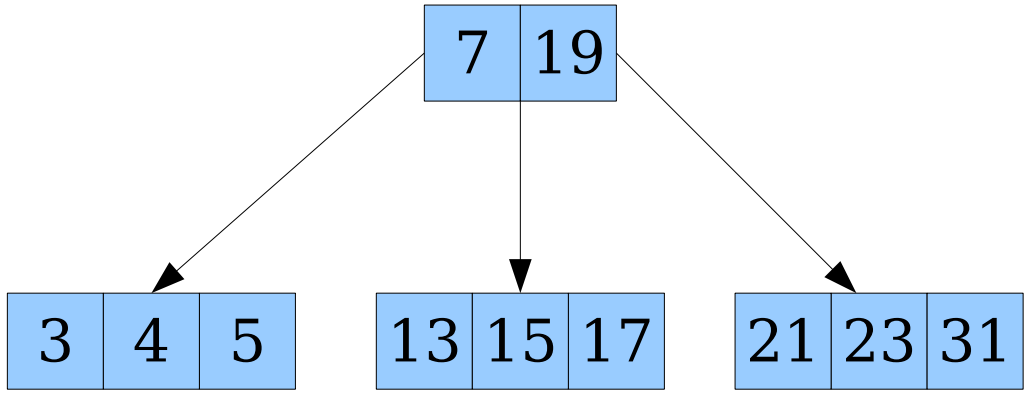

7
19
3
4
5
13
15
17
21
23
31

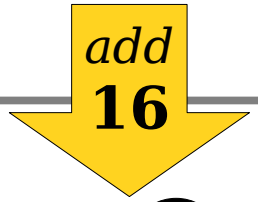
add **16**

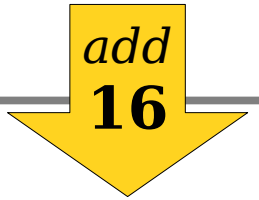
add **16**

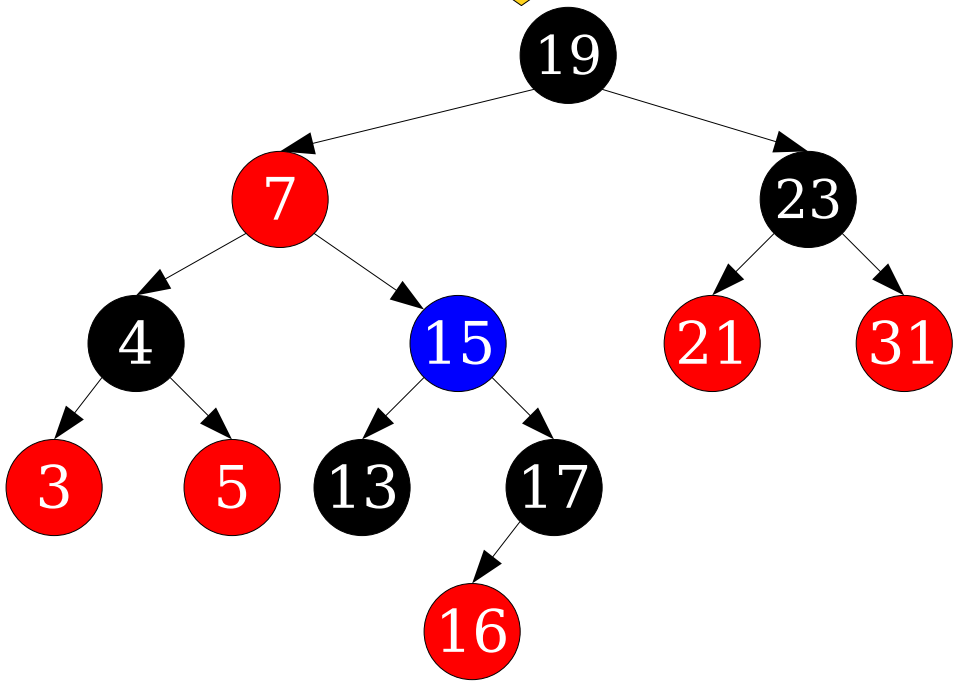

7
15
19
3
4
5
13
16
17
21
23
31

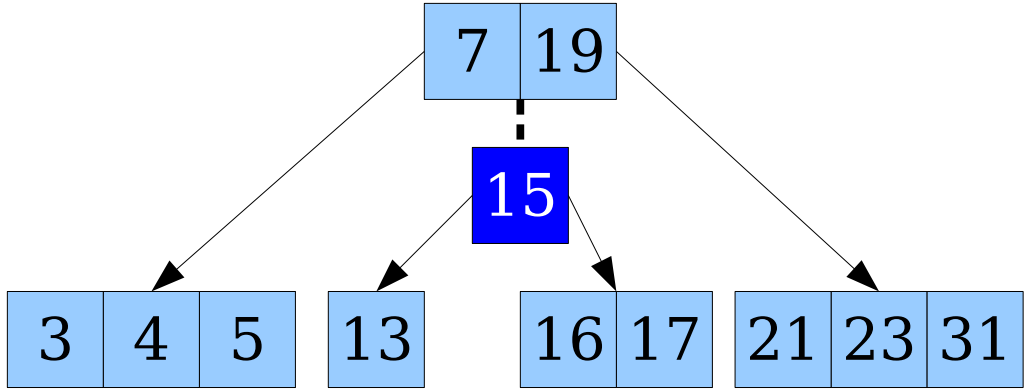

19
7
23
4
15
21
31
3
5
13
17

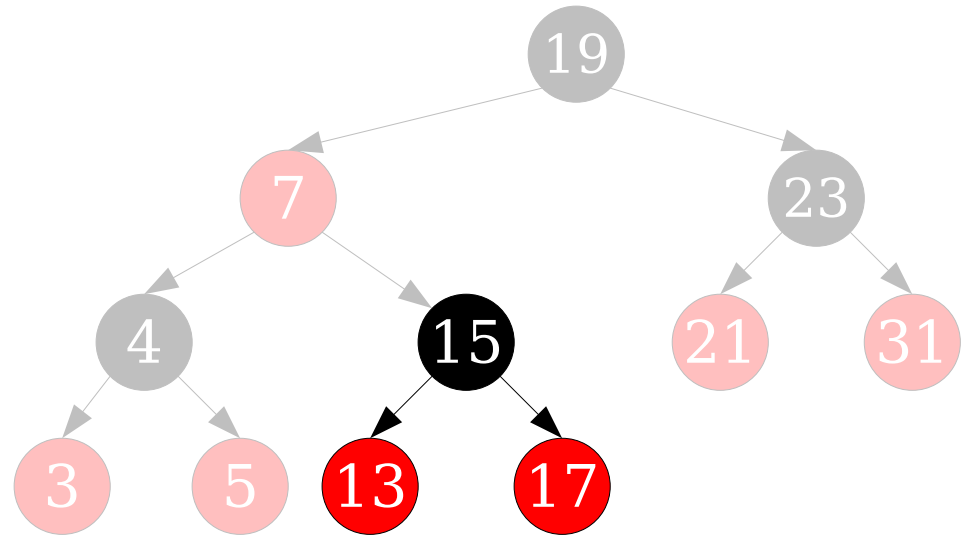

7
19
3
4
5
13
15
17
21
23
31

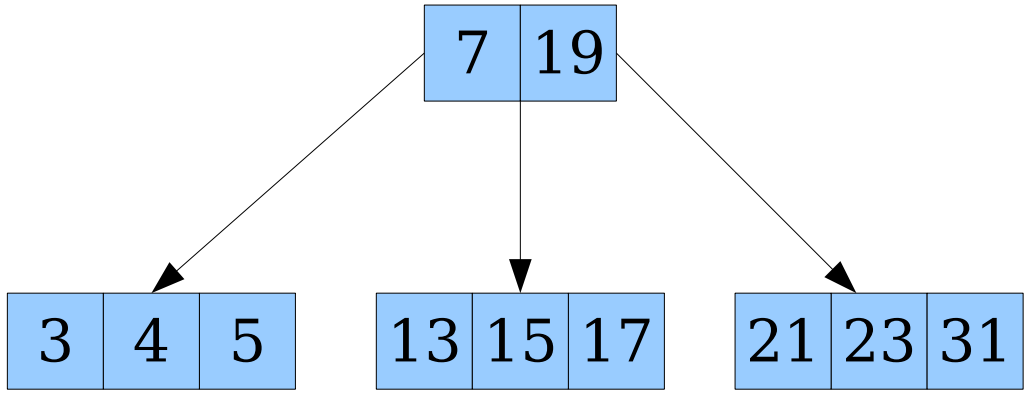

add
16

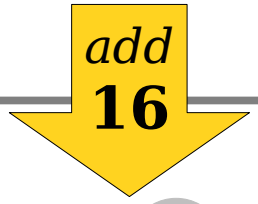

add
16

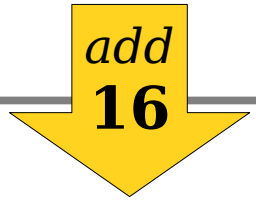

19
7
23
4
15
21
31
3
5
13
17
16

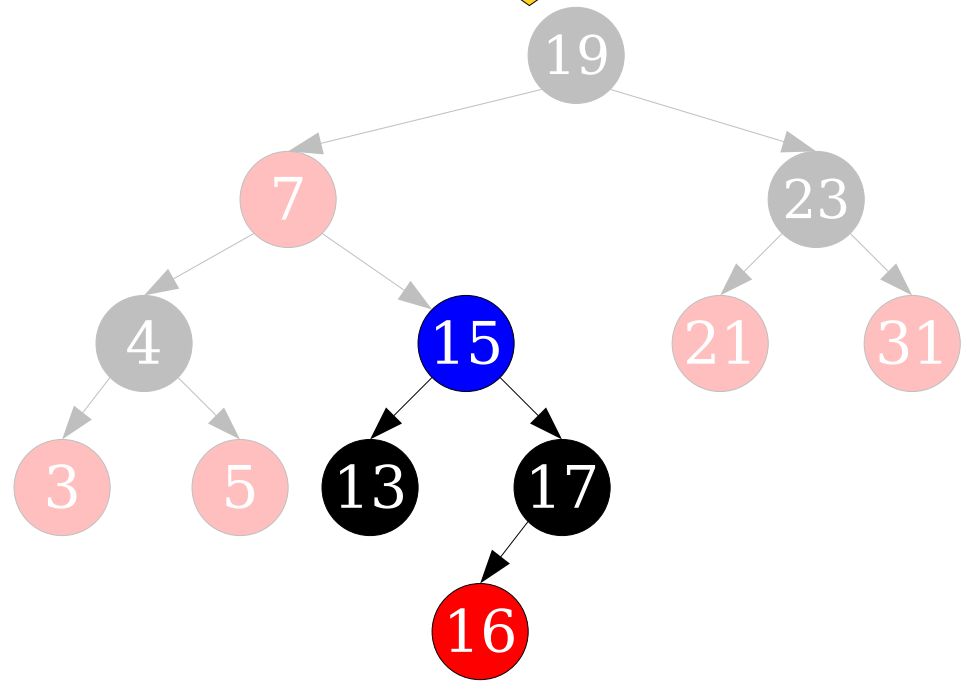

7
19
15
3
4
5
13
16
17
21
23
31

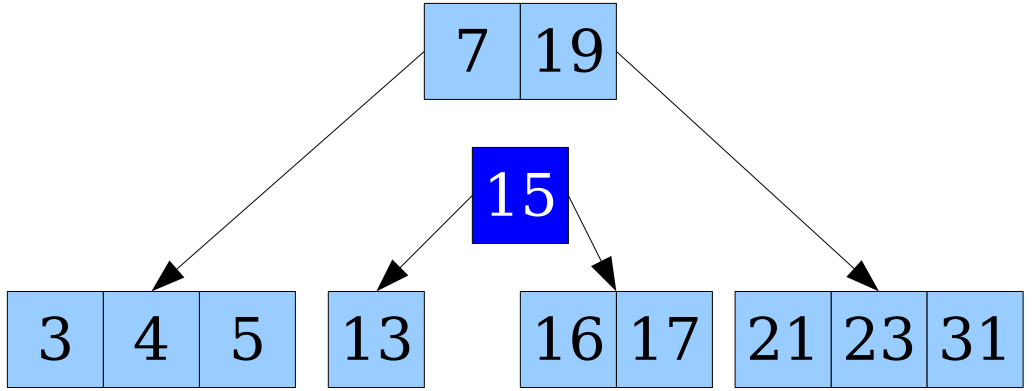

19
7
23
4
15
21
31
3
5
13
17

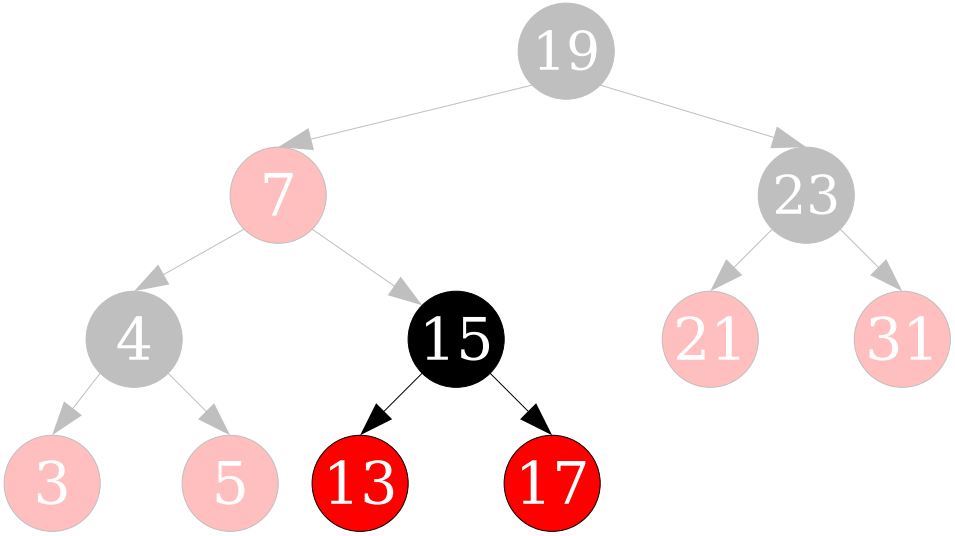

7
19
3
4
5
13
15
17
21
23
31

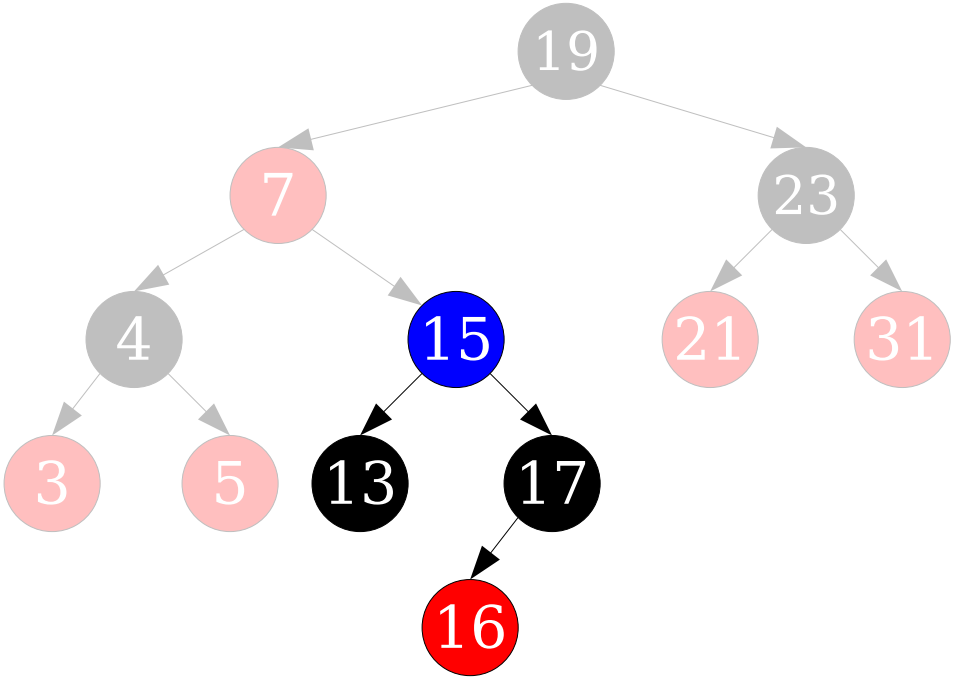

add
16

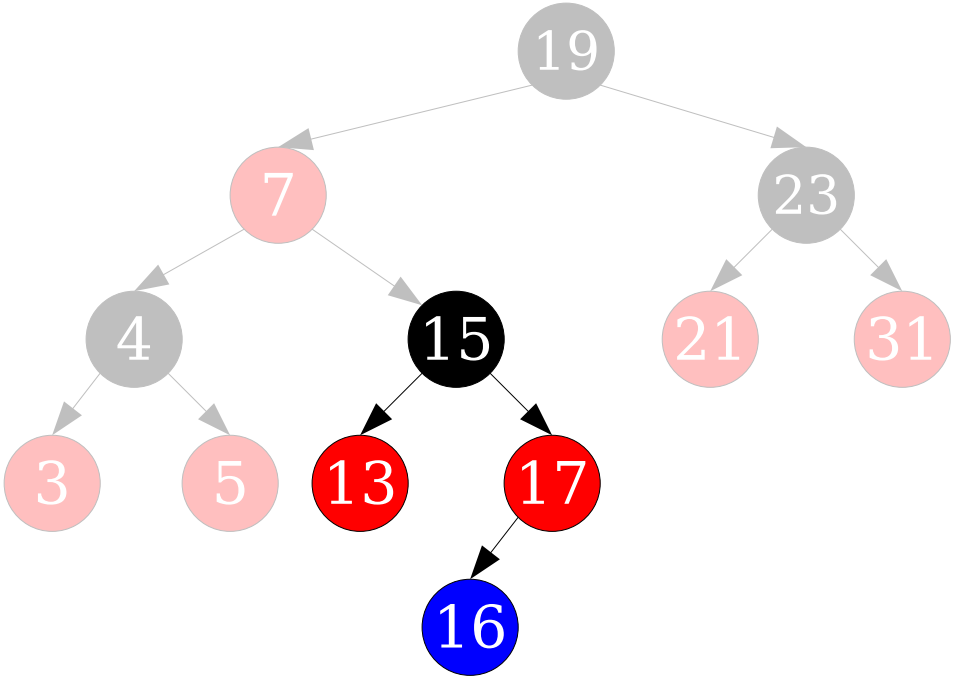

add
16

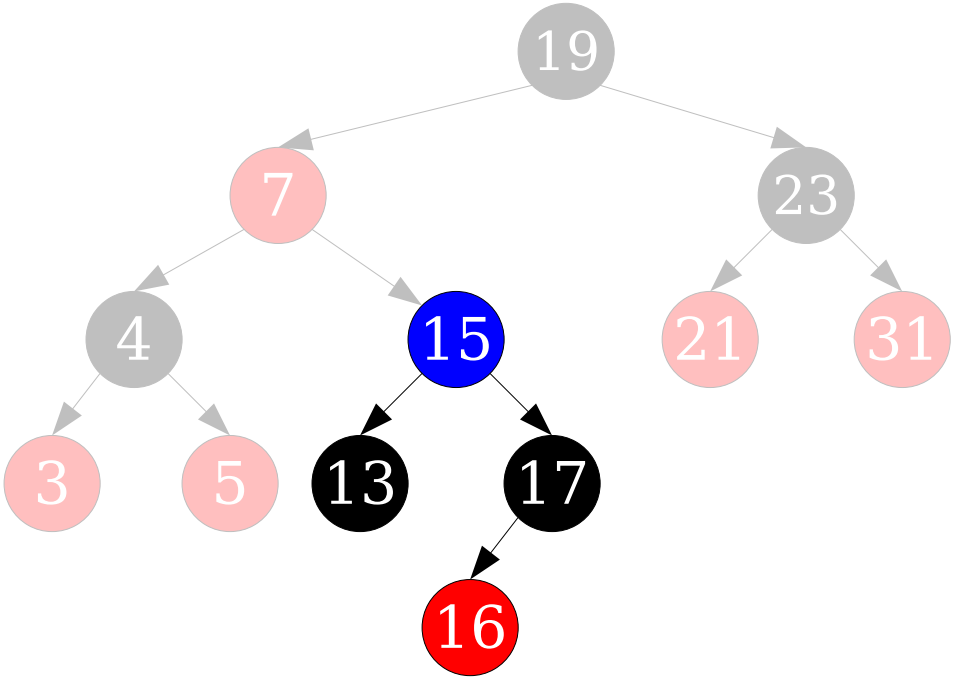

19
7
23
4
15
21
31
3
5
13
17
16

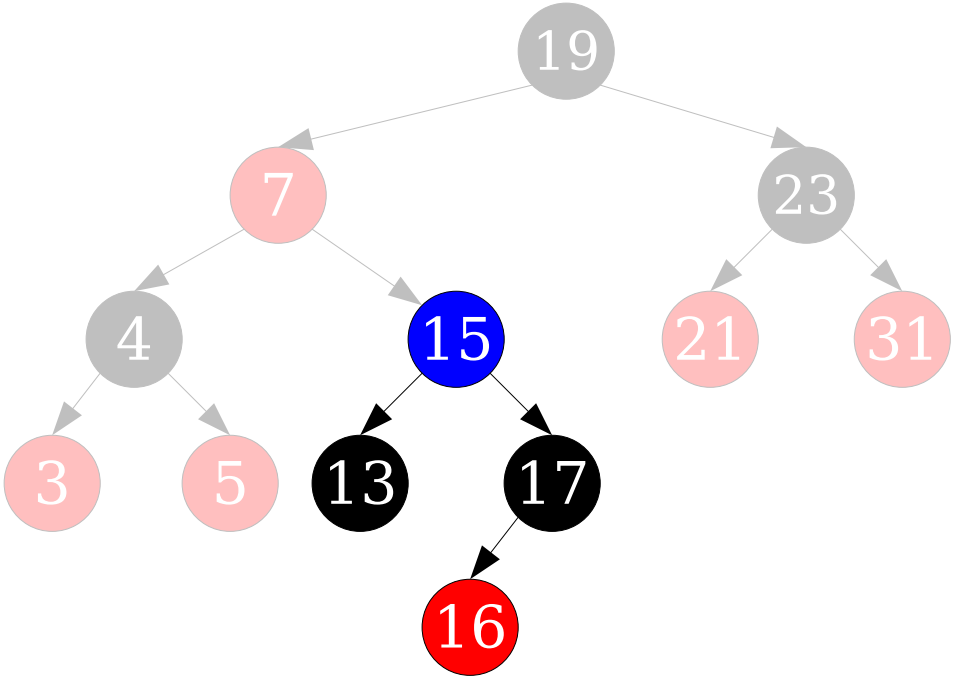

7
19
15
3
4
5
13
16
17
21
23
31

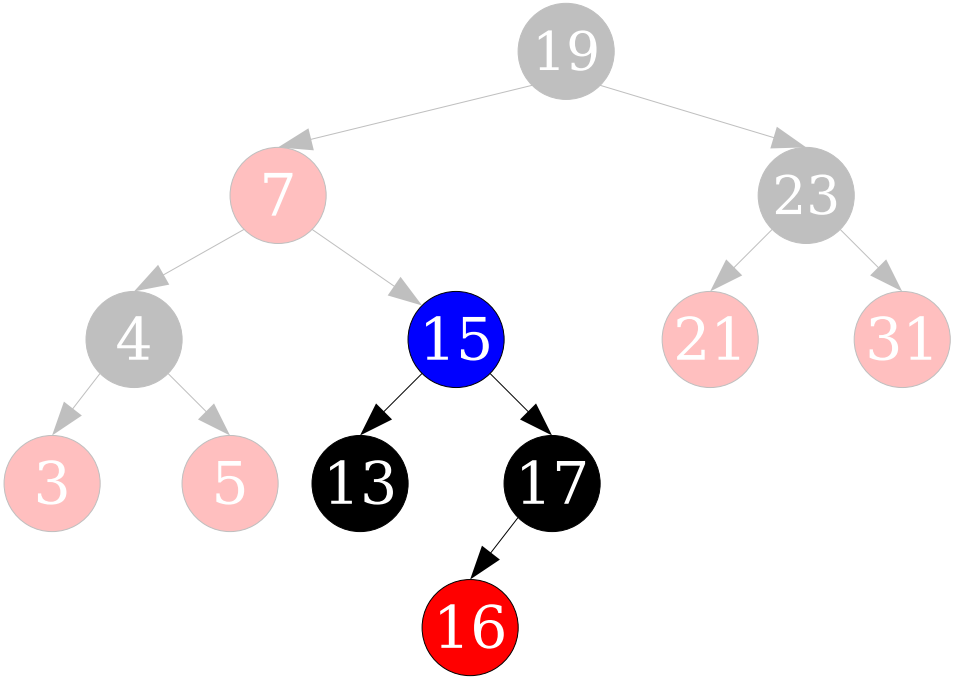

19
7
23
4
15
21
31
3
5
13
17

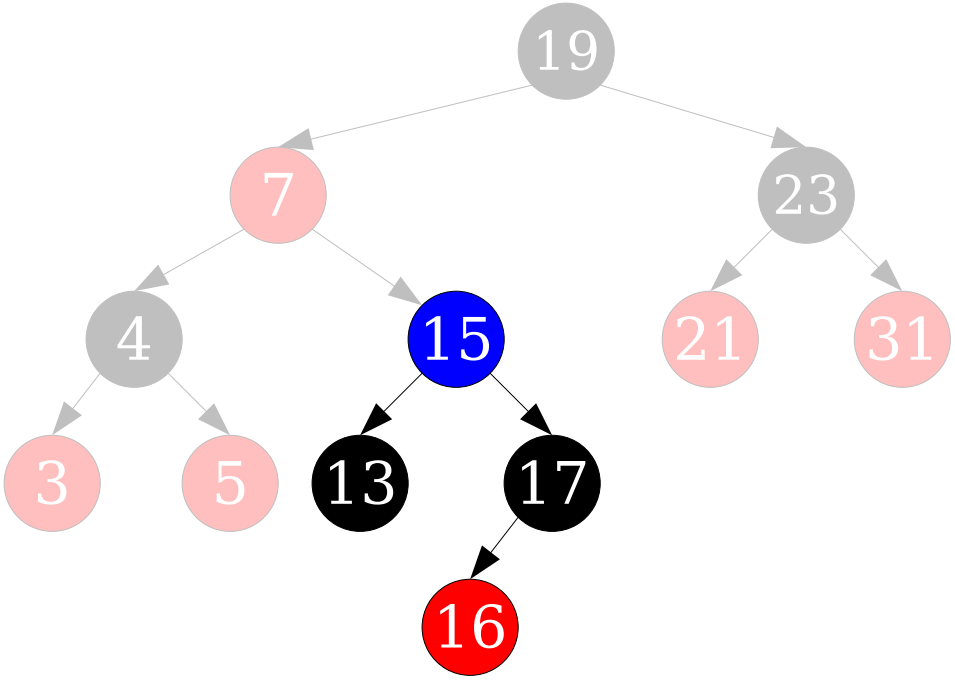

19
7
23
4
15
21
31
3
5
13
17
16

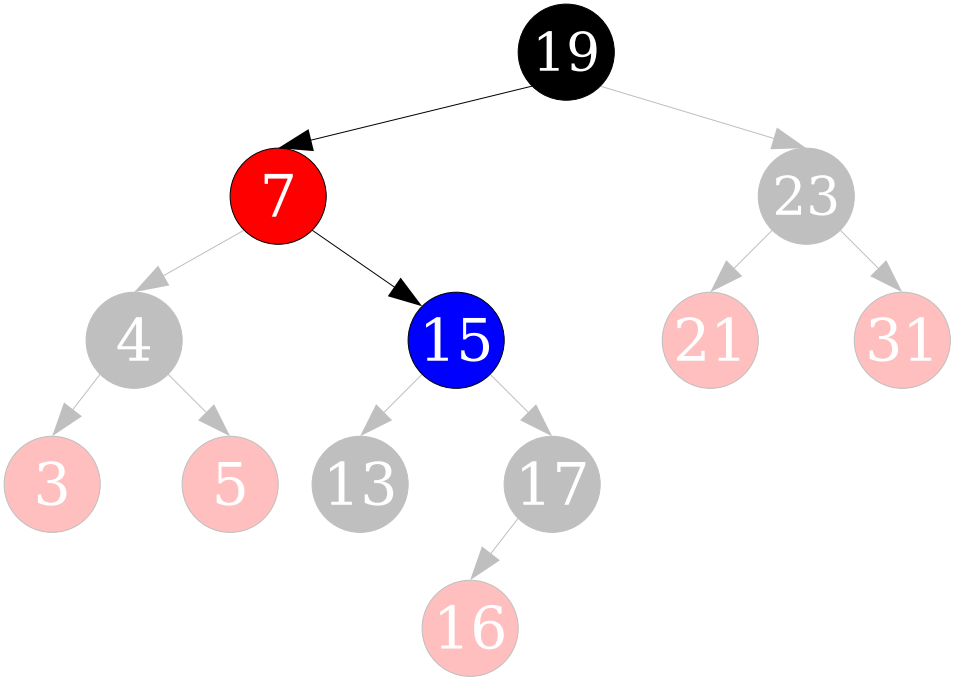

19
7
23
4
15
21
31
3
5
13
17
16

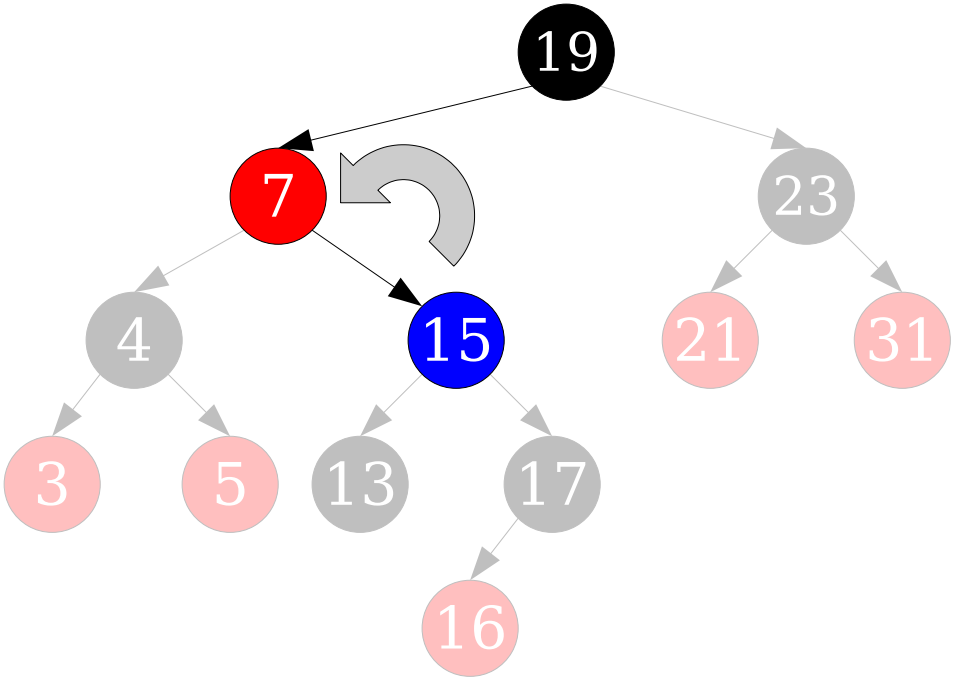

19
7
23
4
15
21
31
3
5
13
17
16

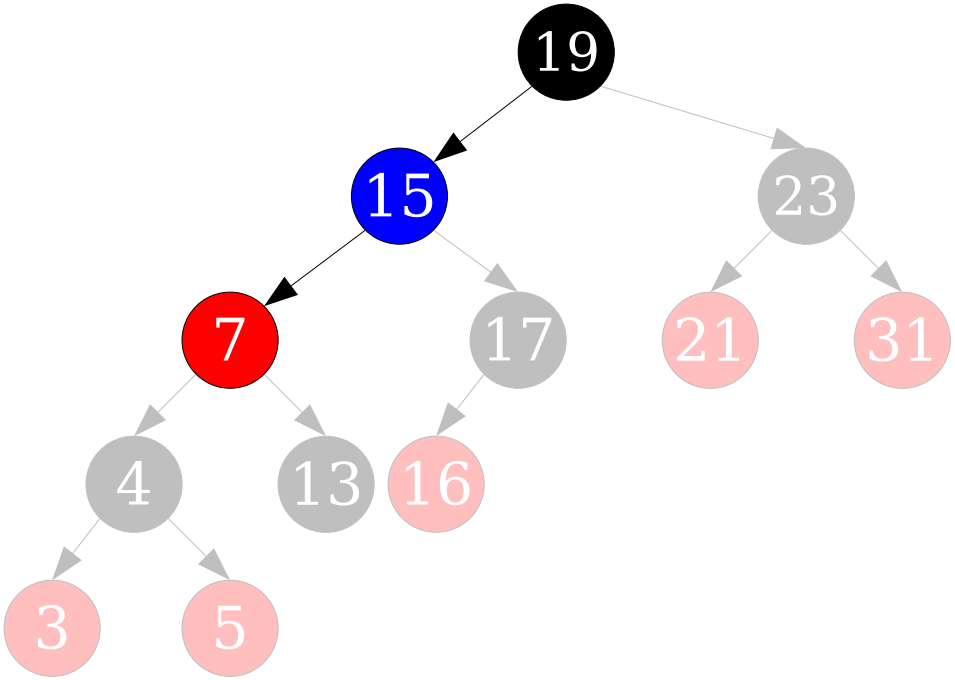

19
7
23
4
15
21
31
3
5
13
17
16

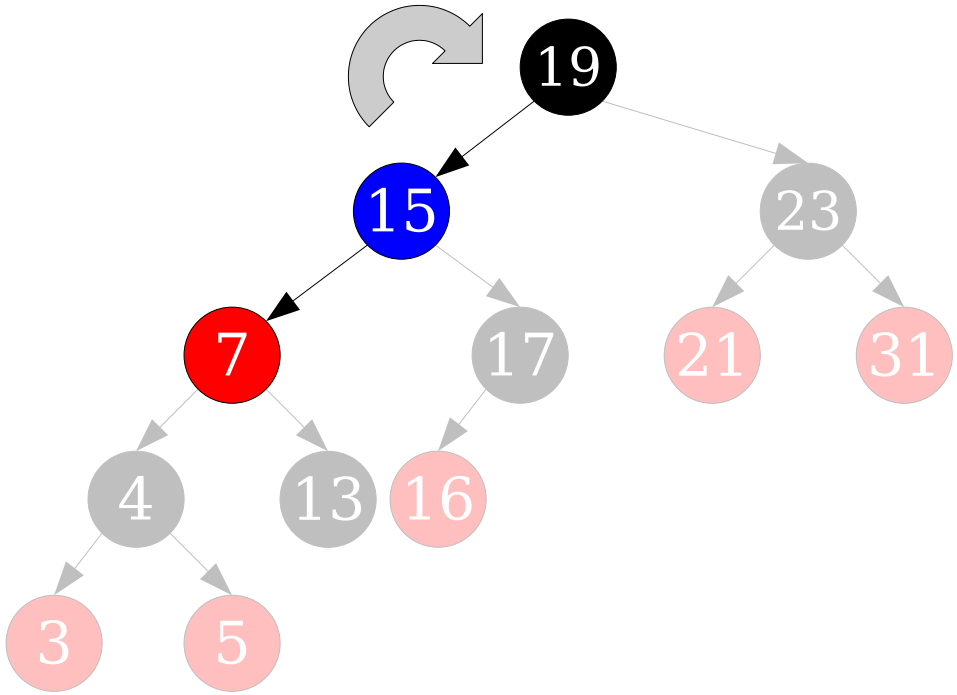

19
7
23
4
15
21
31
3
5
13
17
16

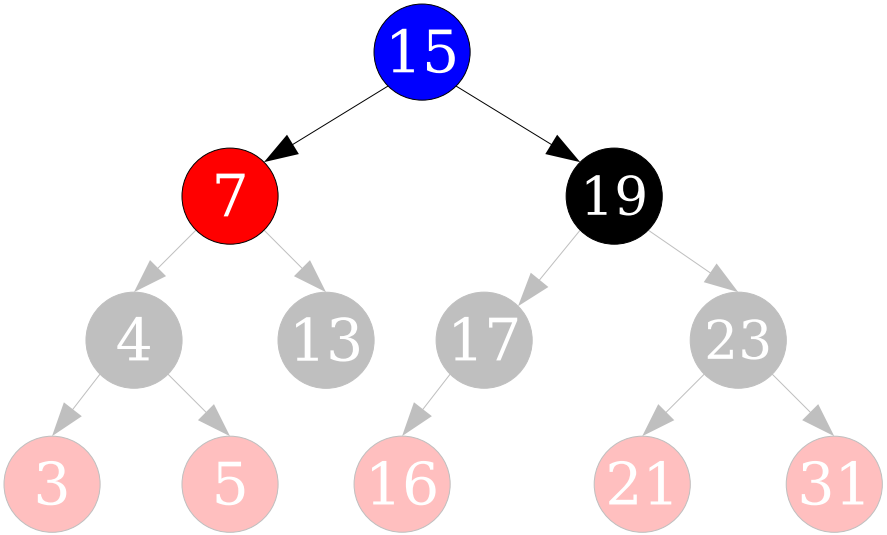

19
7
23
4
15
21
31
3
5
13
17
16

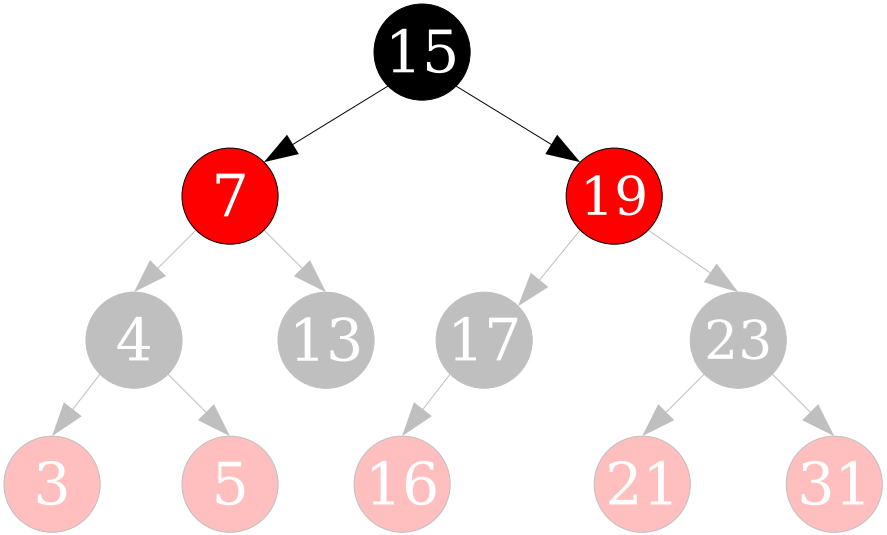

19
7
23
4
15
21
31
3
5
13
17
16

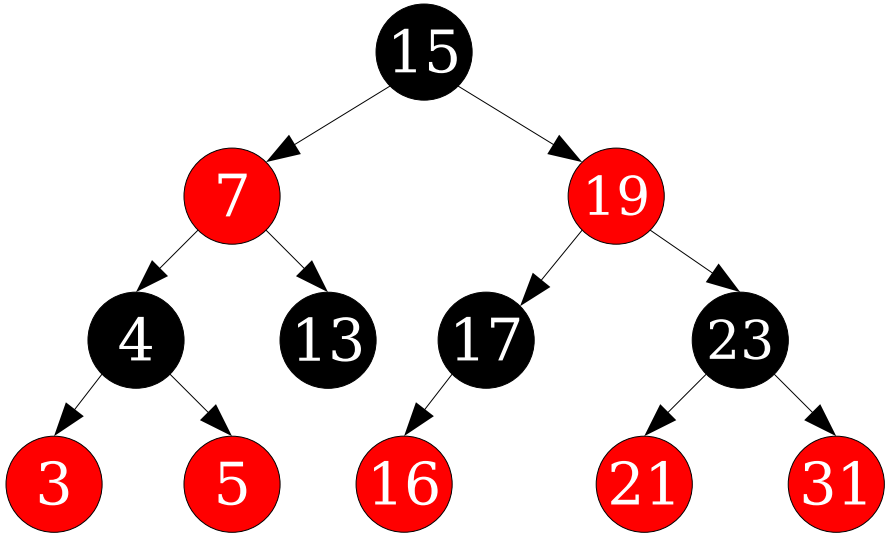

19
7
23
4
15
21
31
3
5
13
17
16


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


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

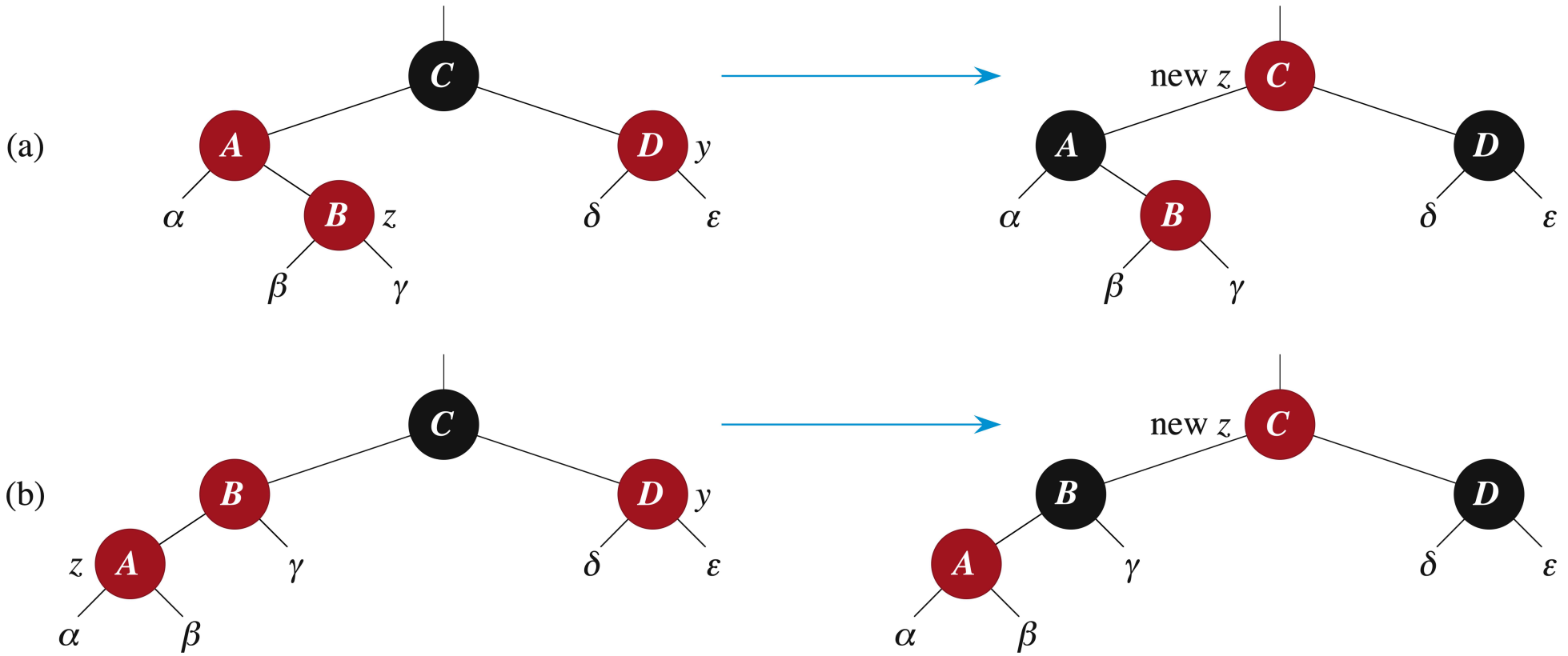

15
7
19
4
13
17
23
3
5
16
21
31

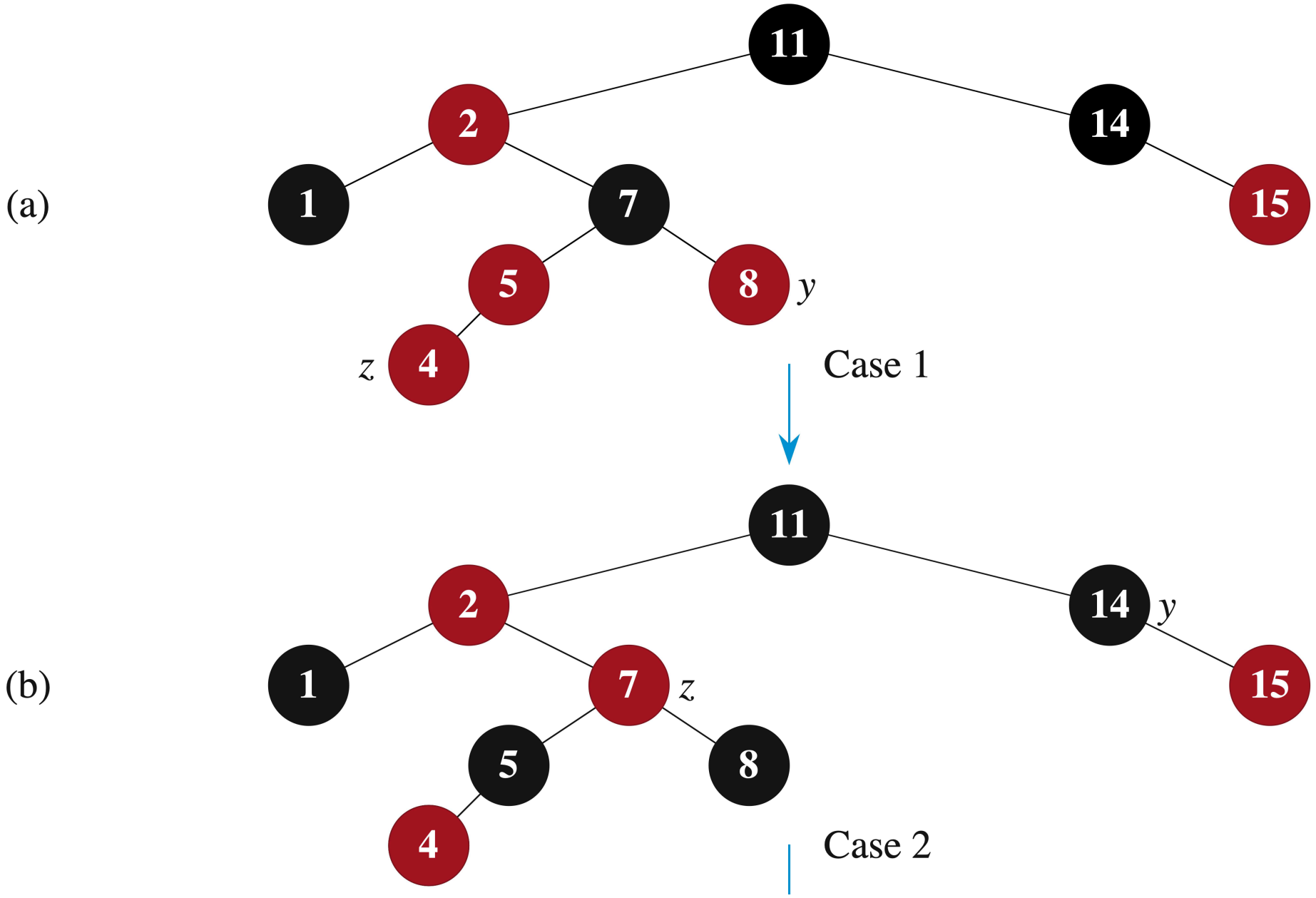

15
7
19
4
13
17
23
3
5
16
21
31

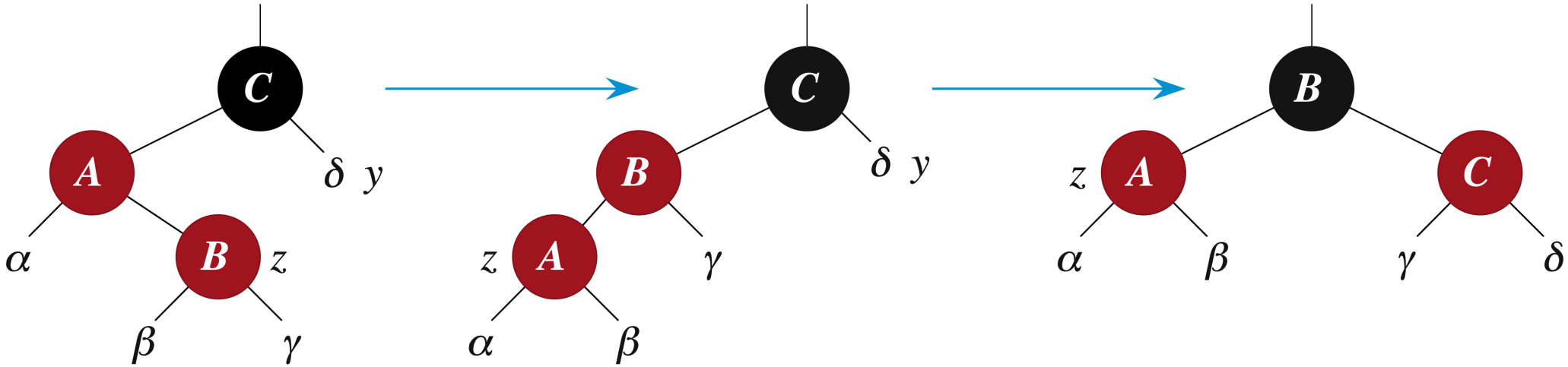

15
7
19
4
13
17
23
3
5
16
21
31

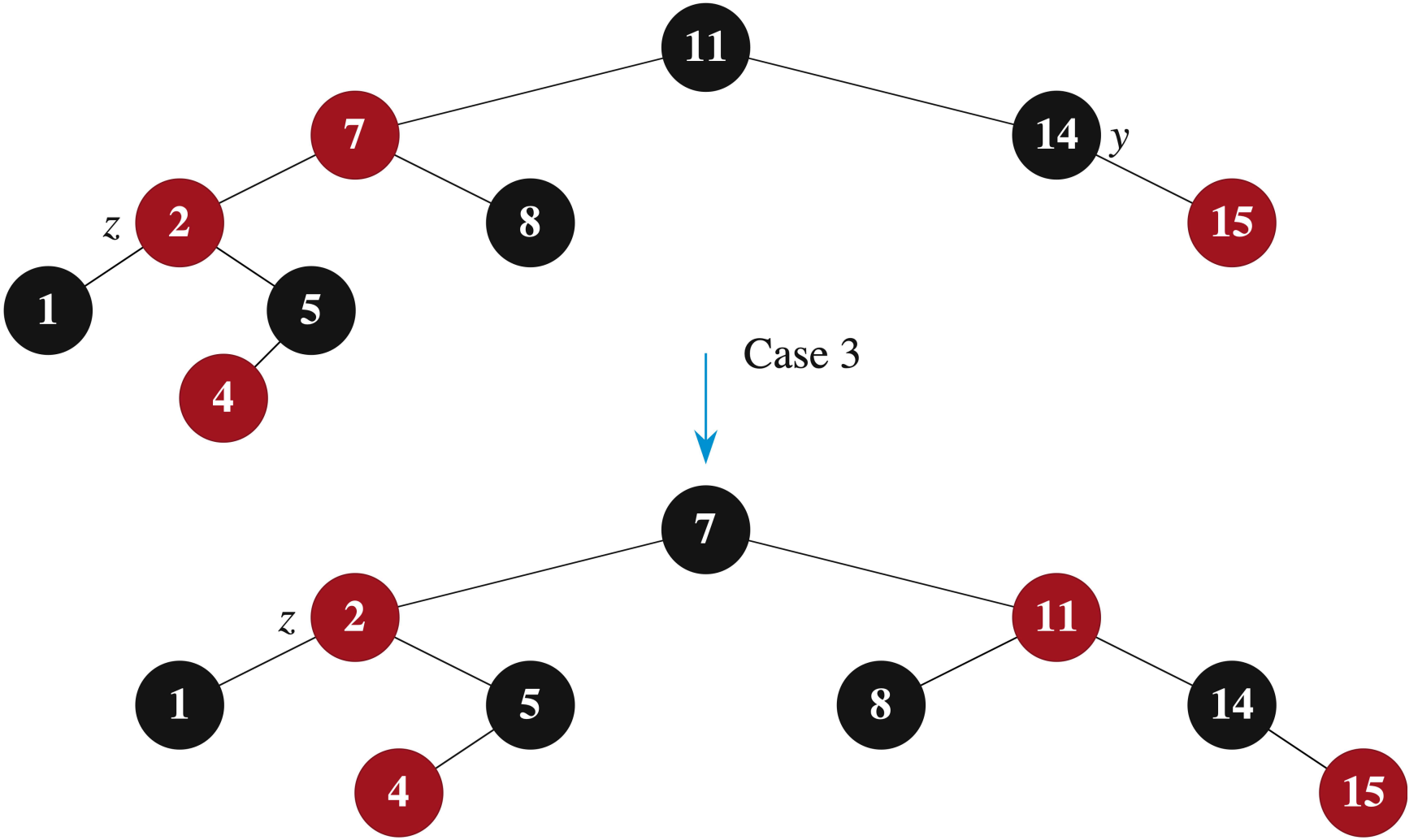
Insert Into Red Black Tree

```
RB-INSERT(T, z)
 1  x = T.root                      // node being compared with z
 2  y = T.nil                       // y will be parent of z
 3  while x ≠ T.nil                 // descend until reaching the sentinel
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y                         // found the location—insert z with parent y
 9  if y == T.nil
10        T.root = z                // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil                  // both of z's children are the sentinel
15  z.right = T.nil
16  z.color = RED                   // the new node starts out red
17  RB-INSERT-FIXUP(T, z)           // correct any violations of red-black properties
```


(a)
C
A
D
y
α
B
z
δ
ε
β
γ
new z
C
A
D
α
B
δ
ε
β
γ
(b)
C
B
D
y
z
A
γ
δ
ε
α
β
new z
C
B
D
A
γ
δ
ε
α
β


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2


C
A
δ y
α
B
z
β
γ
Case 2
C
B
δ y
z
A
γ
α
β
Case 3
B
z
A
C
α
β
γ
δ


11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left                  // is z's parent a left child?
 3          y = z.p.p.right                   // y is z's uncle
 4          if y.color == RED                 // are z's parent and uncle both red?
 5              z.p.color = BLACK             \
 6              y.color = BLACK               |  case 1
 7              z.p.p.color = RED             |
 8              z = z.p.p                     /
 9          else
10              if z == z.p.right             \
11                  z = z.p                   |  case 2
12                  LEFT-ROTATE(T, z)         /
13              z.p.color = BLACK             \
14              z.p.p.color = RED             |  case 3
15              RIGHT-ROTATE(T, z.p.p)        /
```