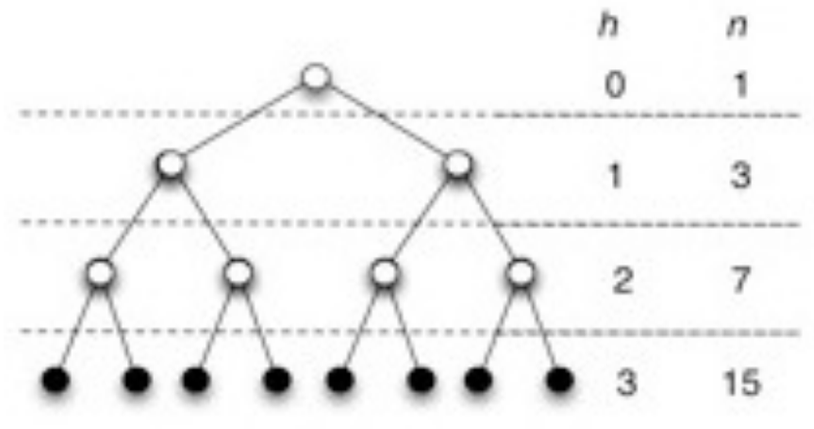
Balance:
2-3-4 Trees.
Red-Black Trees.


A BUG IN MY CODE?
:)
, IT'S A FEATURE
imgflip.com


YOUR MOMMA'S SO FAT
SHE CAN SIT ON A BINARY TREE AND FLATTEN IT TO A LINKED LIST IN O(1) TIME
imgflip.com

Reorienting: Binary Search Trees

- BSTs consist of a collection of *ordered key/value nodes*. We use the keys to traverse the BST and perform map operations (insert/delete/search/etc).
- Ideally, a *random* series of insertions/deletions in BST leads to O(log n) *expected running time* for basic map operations (insert/delete/search).
- However, we can't claim better than O(n) *worst case running time* because it is possible that some seq. of operations could lead to an ***unbalanced tree***
    - Recall that ***balance*** suggests that all the leaves in the tree are approximately at the same depth in the tree w.r.t. the root.
        - In the extreme case, this could be a linear sequence of nodes (i.e., a linked list).
- We really want trees to be *balanced* in order to make strong guarantees about operation performance …

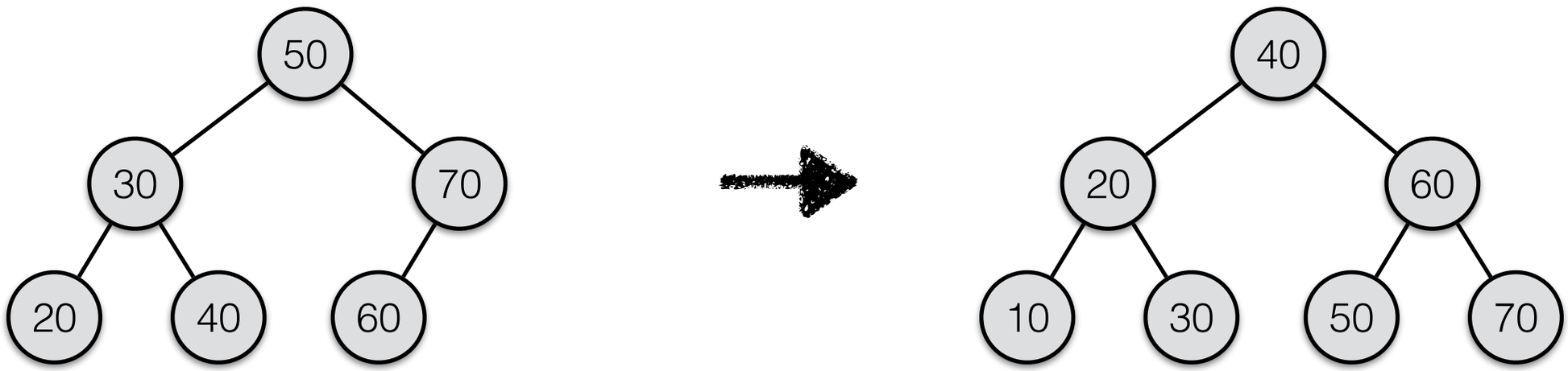


www2.hawaii.edu

> ***Note:*** *running times are often related to the* ***height*** *of the tree; i.e., if the height of a tree is O(h), then operations take O(h). In the worst case, h == n. In the more ideal case, h is O(lg(n))*

Balanced Binary Search Trees

- **Q:** So how do we insure “balance”?
- **Idea:** Each time you insert/delete, “fix up” the tree so that it is always as close to being balanced as possible.
- **Problem:** This isn’t so easy…
  - ex. Consider the below tree…
    - This tree is balanced…
    - Insert 10
    - **Notice:** not a single parent-child relationship is preserved between the original tree and the new, balanced tree — it is possible that balancing could take O(n) time!
  - ex. Consider deleting 70 and inserting 0.
  - ex. Then consider deleting 60 and inserting -10.



Balanced Binary Search Trees

Today, we will look at two known “solutions” (compromises):

1. Give up “binary” — allow nodes to have variable # of children (**2-3-4 Trees**, B-Trees).
2. Give up “perfect” — keep tree “close” to perfectly balanced (AVL Trees, **Red-Black Trees**).

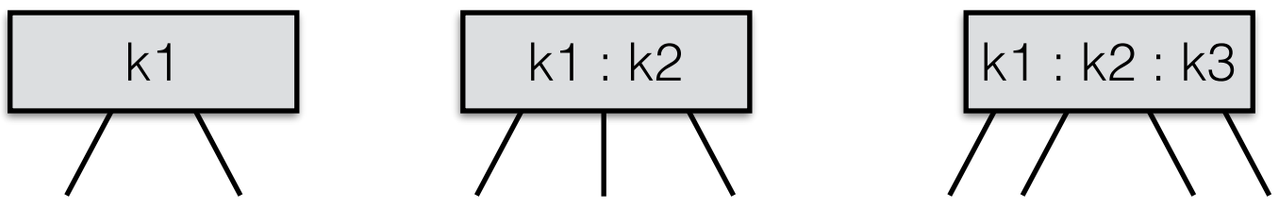
2-3-4 Trees

2-3-4 Trees

- Also known as (2,4) Trees.
- Intuition:
  - Allow multiple keys to be stored at each node
  - A node will have one more child than it has keys:
    - leftmost child — all keys ***less than*** the first key
    - next child — all keys ***between*** the first and second keys
    - … etc …
    - last child — all keys ***greater than*** the last key.
- This general intuition is really more for multiway search trees; we will work with 2-3-4 trees.

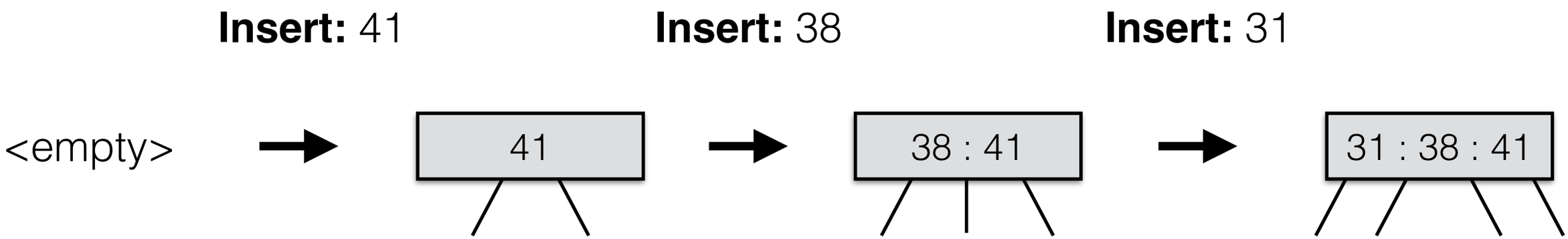
2-3-4 Trees: “Rules”

- Since we are working with 2-3-4 trees, the “rules” we will follow are:
  - Every node has either 2, 3, or 4 children (1, 2, or 3 keys)
    - a.k.a. the ***size property***
  - All the leaves of the tree (either external nodes or null pointers) are on the same level.
    - a.k.a. the ***depth property***



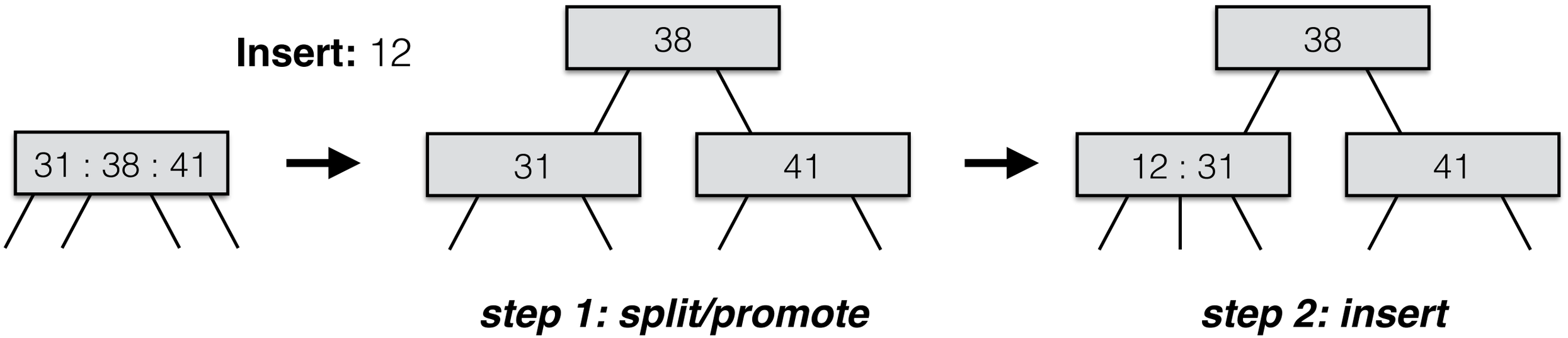
2-3-4 Trees: “Rules” (Inserting)

- To insert, go as far down to a node w/ external children…
- **If the node you get down to is a 2- or 3- node**, expand to a 3- or 4- node by adding the key in the correct location.



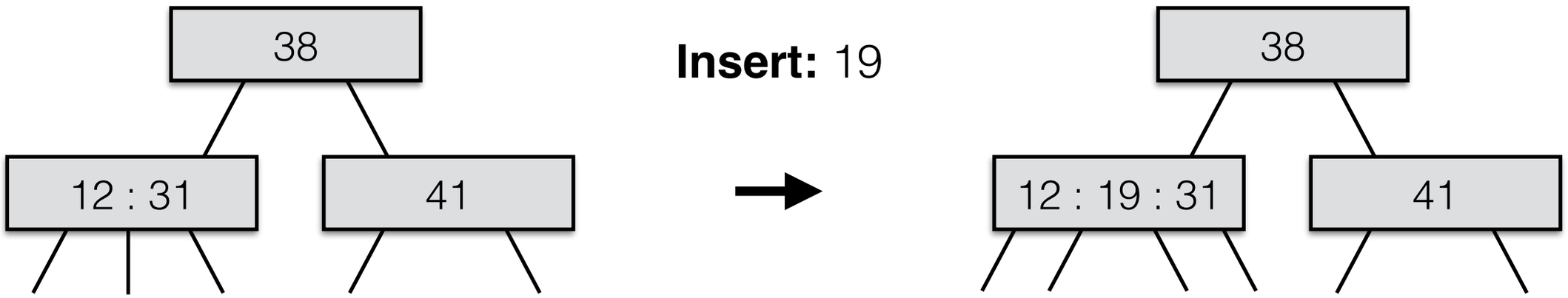
2-3-4 Trees: “Rules” (Inserting)

- To insert, go as far down to a node w/ external children…
- **If the node you get down to is a 2- or 3- node**, expand to a 3- or 4- node by adding the key in the correct location.
- **If the node you get down to is a 4-node**, SPLIT into two 2-nodes and promote the *middle key* to join the parent node.
    - Note 1: Create a new root if the current root is the 4-node being split.
    - Note 2: Promoting the middle key to the parent may add a 4th key to the parent, meaning it will also have to split, etc.
- Now, continue following down into the correct 2-node where you will insert the new key.



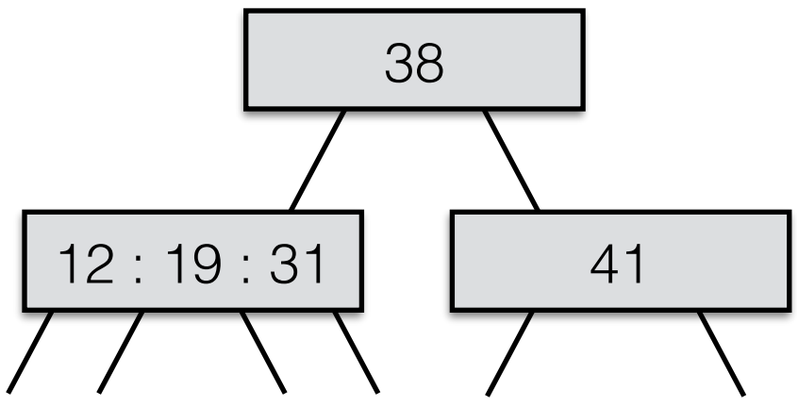
2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

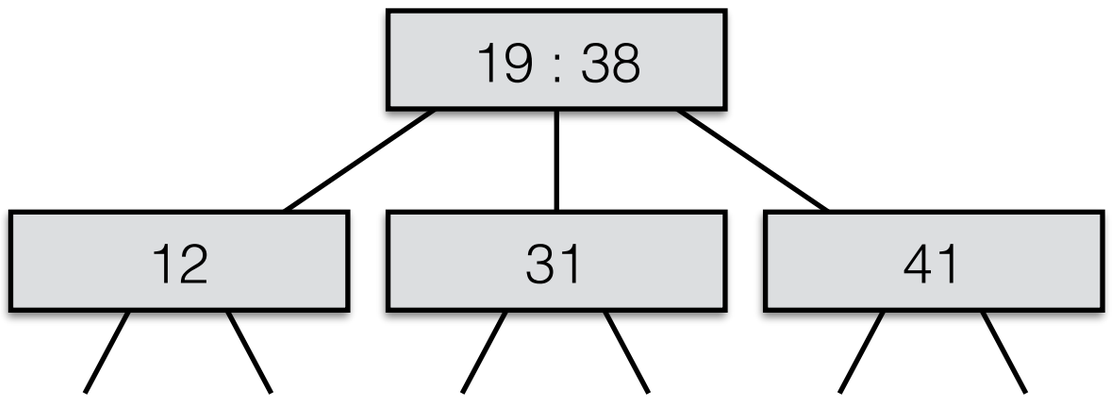


2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```



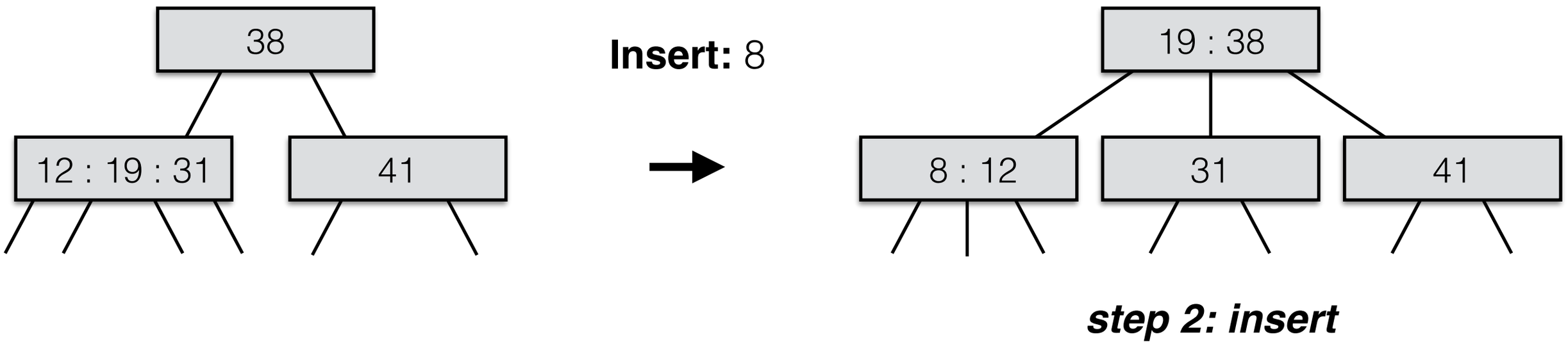
Insert: 8



step 1: split/promote

2-3-4 Trees: “Rules” (Inserting)

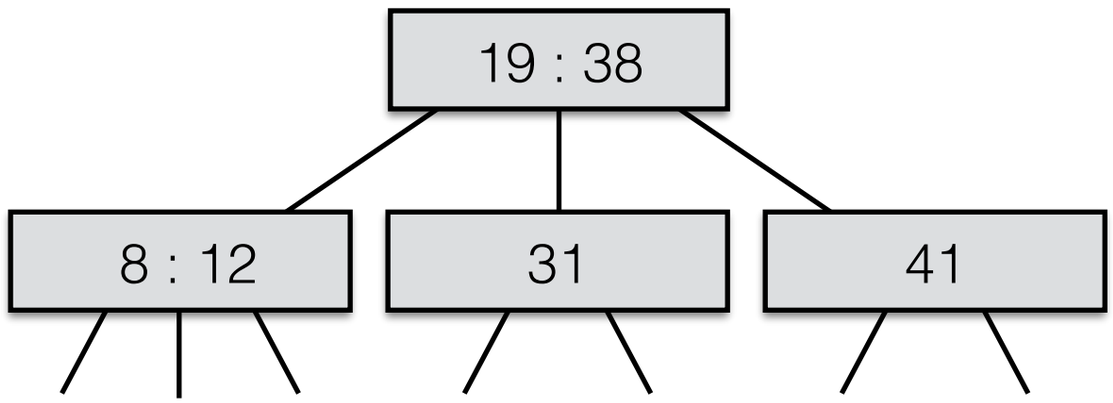
```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

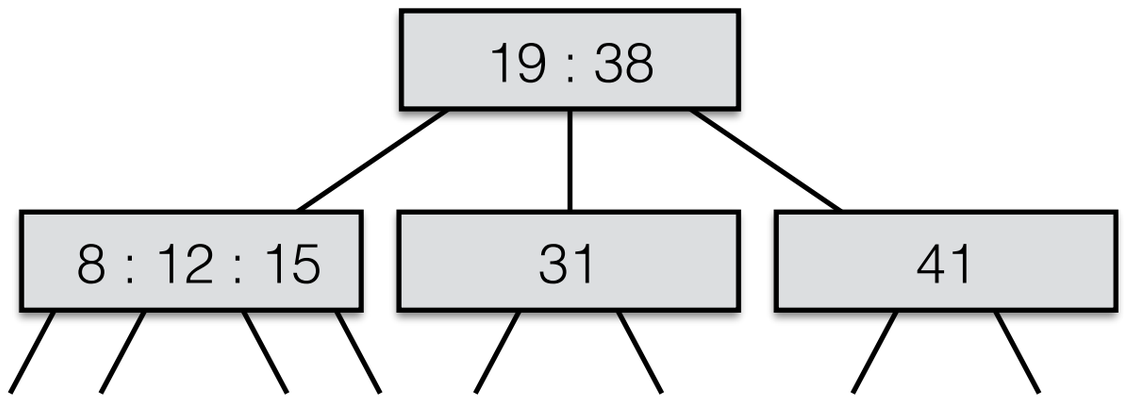
Insert: 15 (before)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

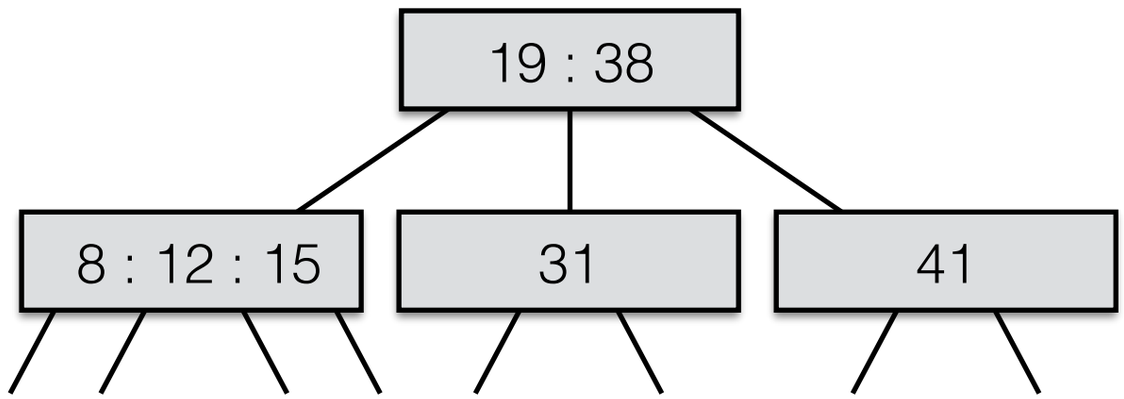
Insert: 15 (after)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

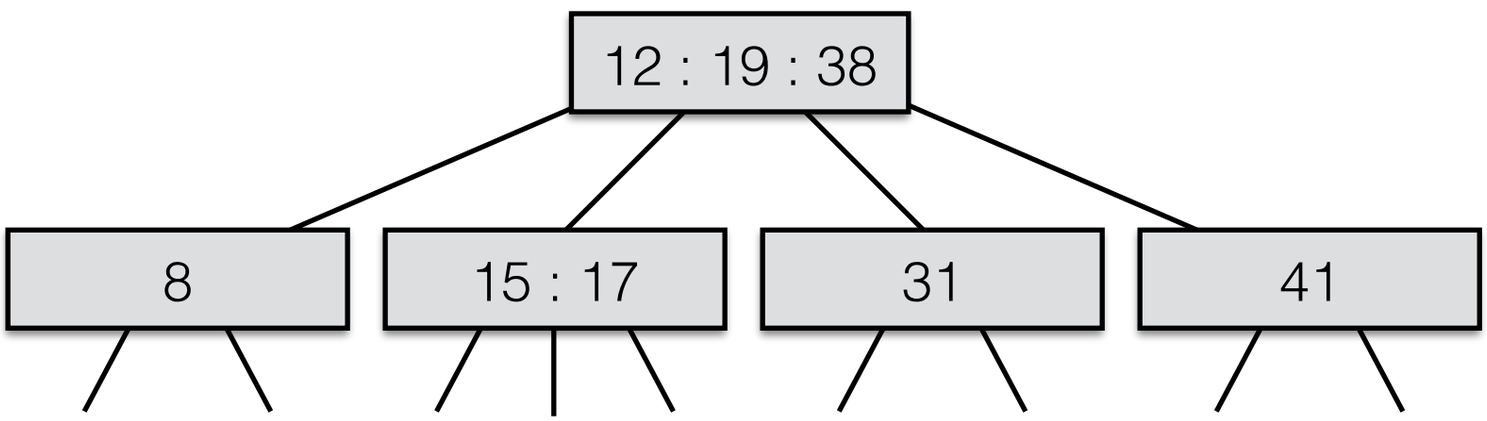
Insert: 17 (before)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

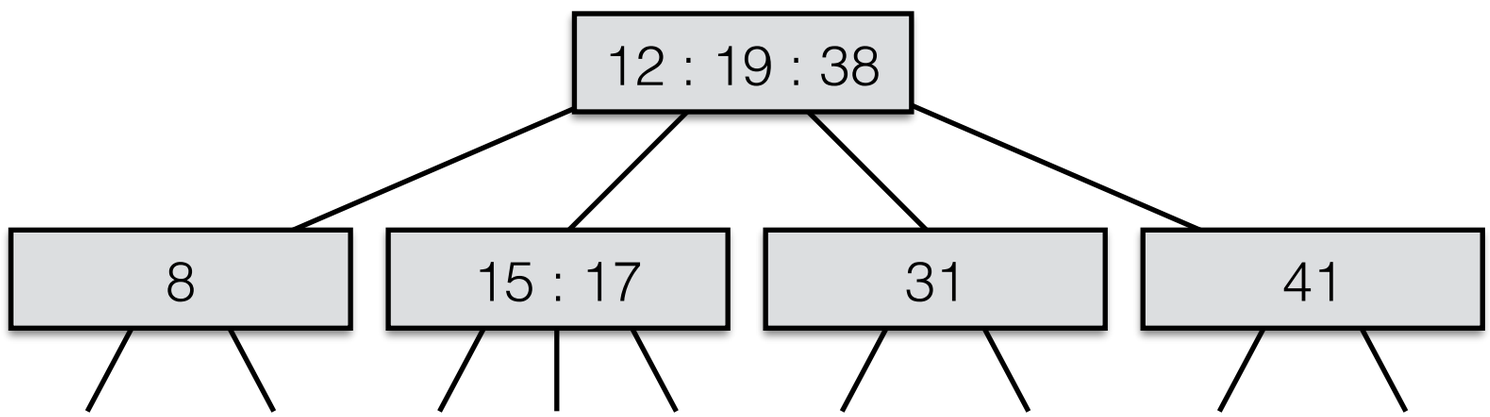
Insert: 17 (after)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

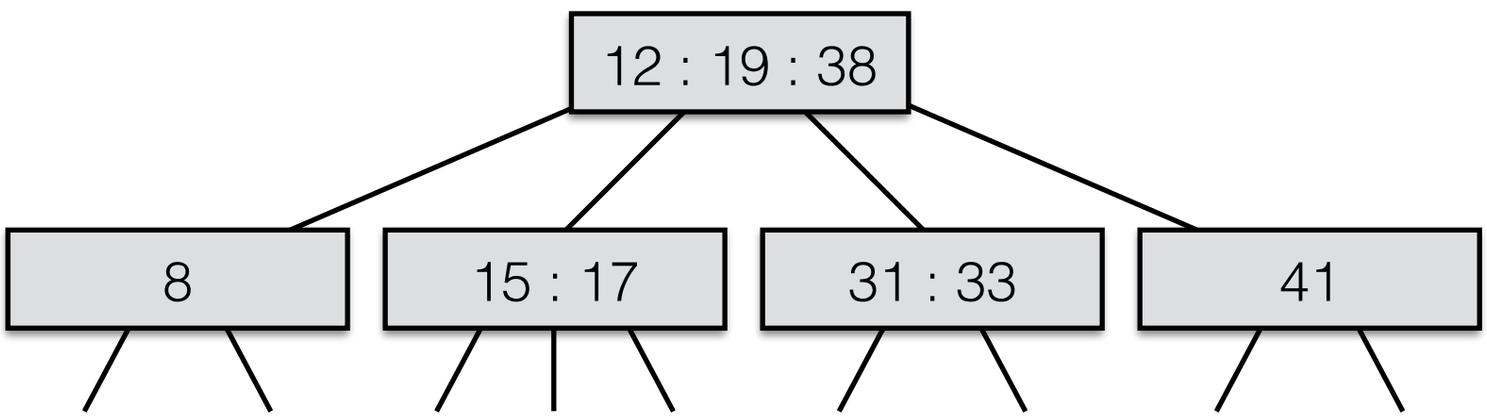
Insert: 33 (before)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

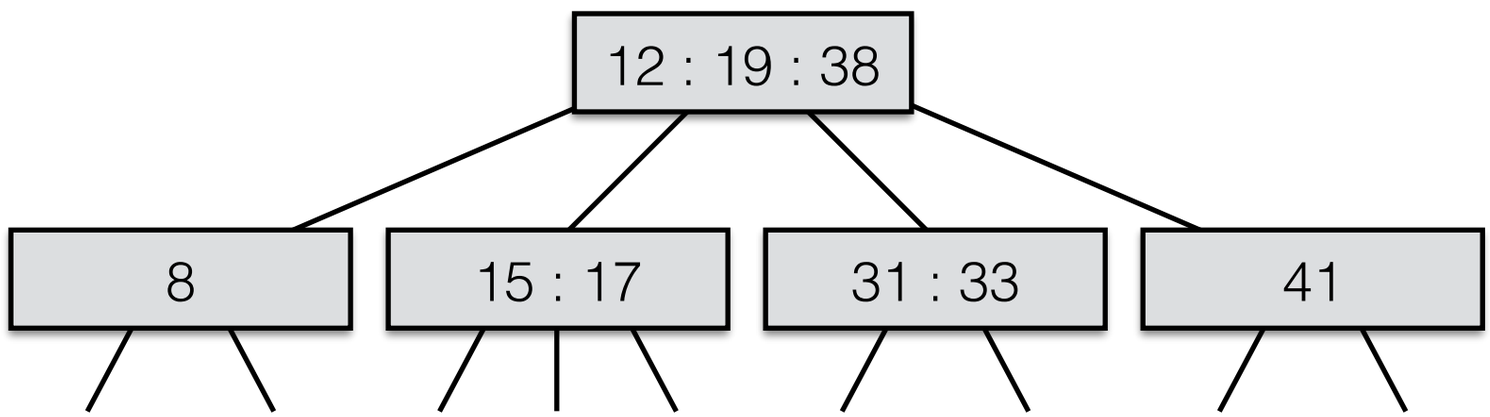
Insert: 33 (after)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

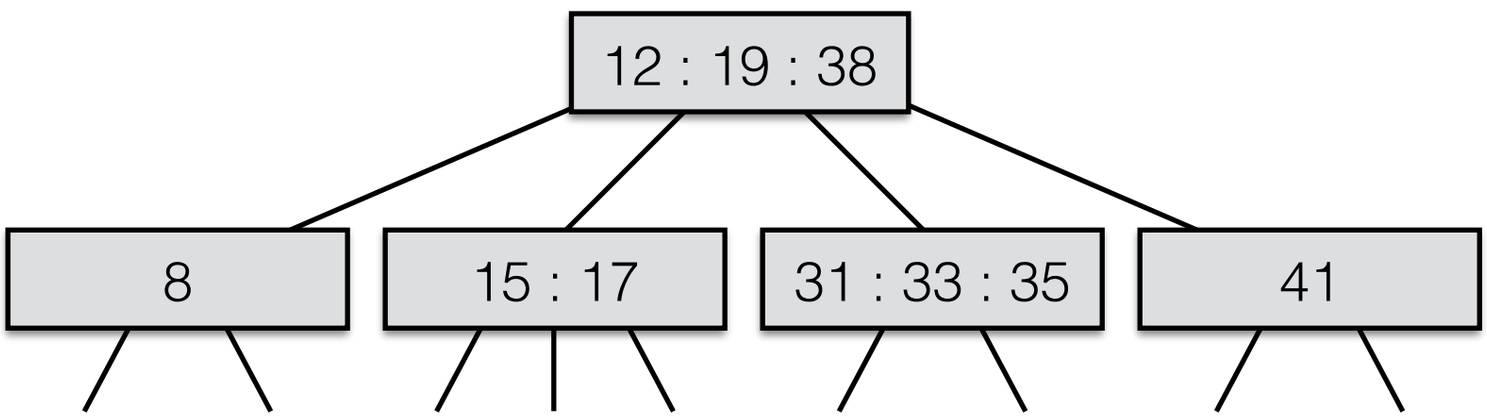
Insert: 35 (before)



2-3-4 Trees: “Rules” (Inserting)

```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

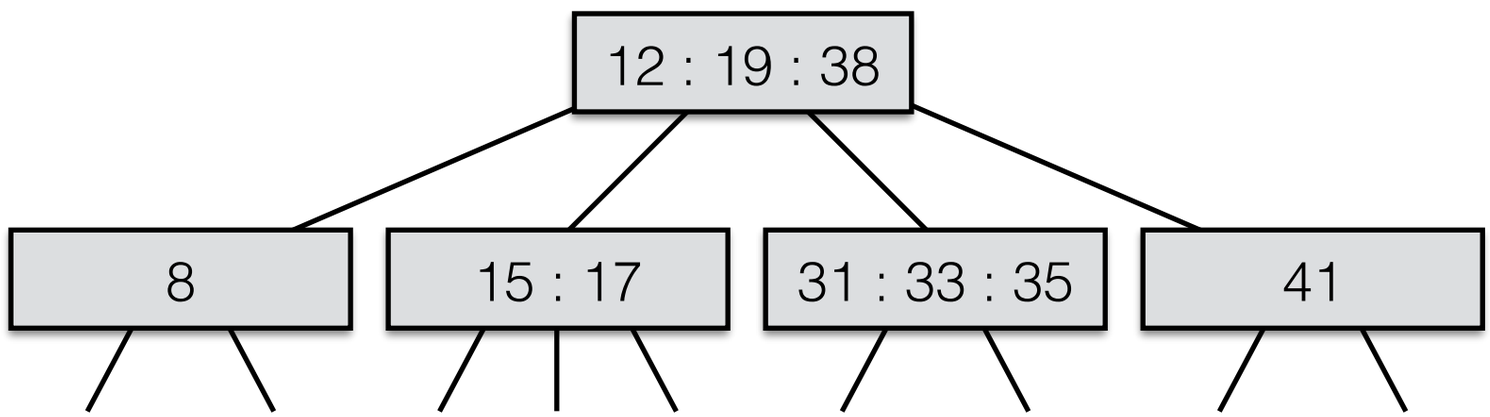
Insert: 35 (after)



2-3-4 Trees: “Rules” (Inserting)


```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

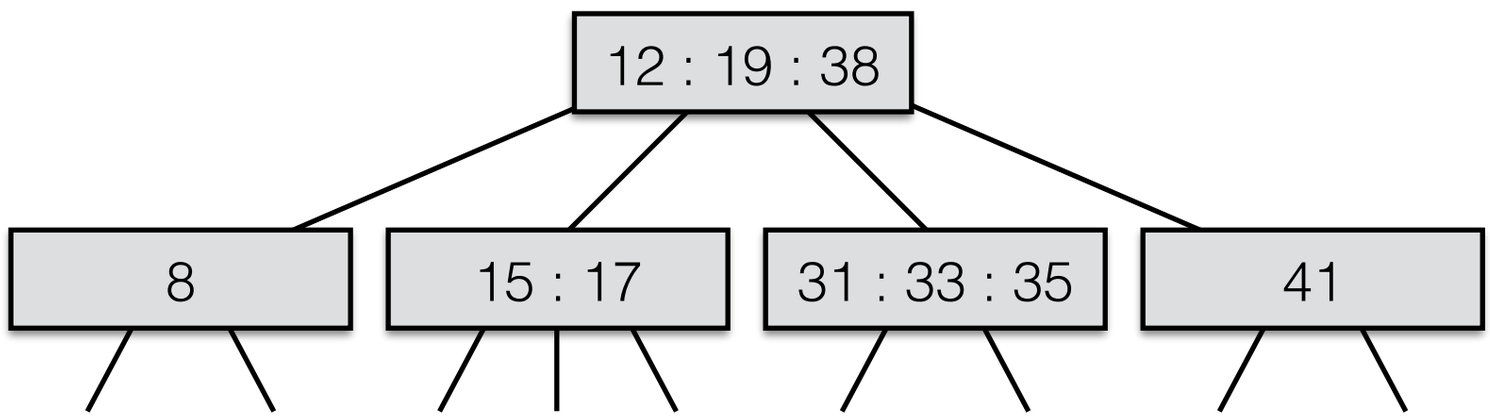
Insert: 20 (before)



2-3-4 Trees: “Rules” (Inserting)


```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

Insert: 20 (before)

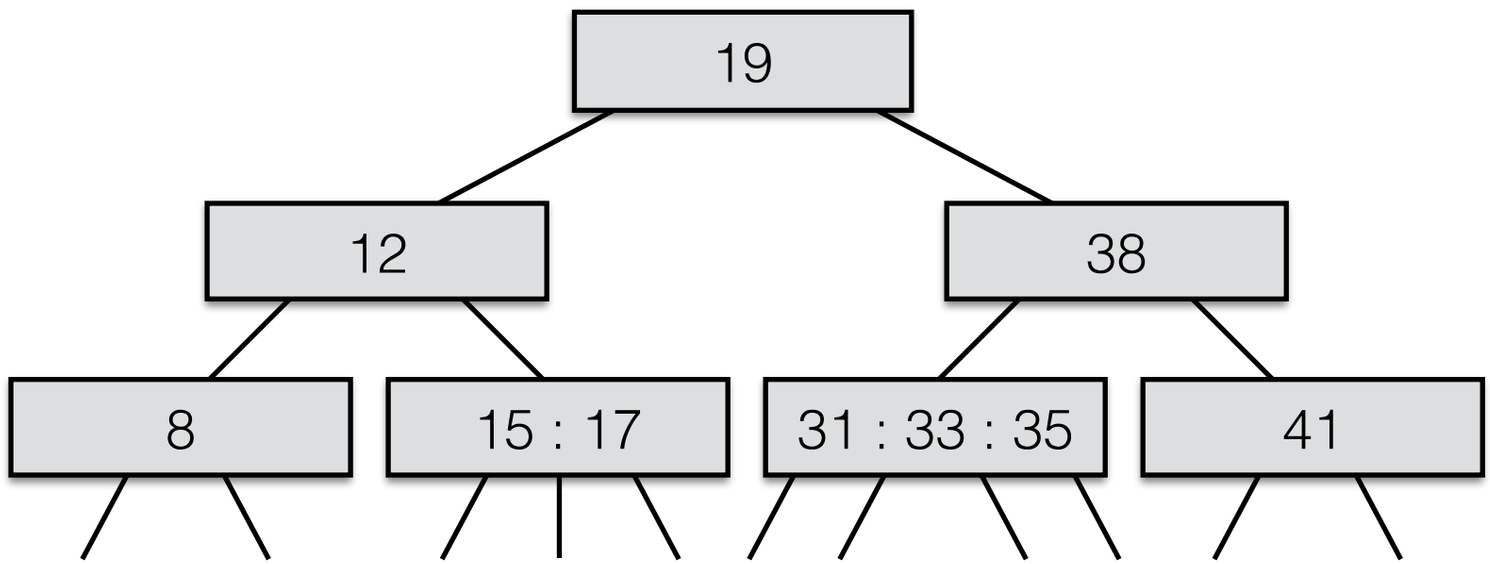


2-3-4 Trees: “Rules” (Inserting)


```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

Insert: 20

[after first split]

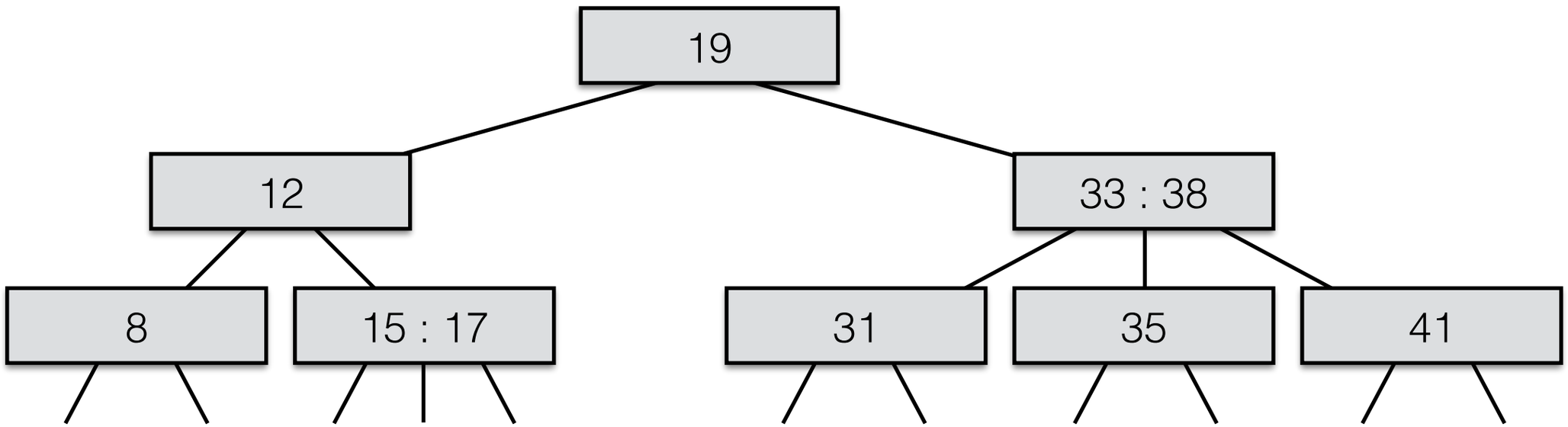


2-3-4 Trees: “Rules” (Inserting)


```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

Insert: 20

[after second split]

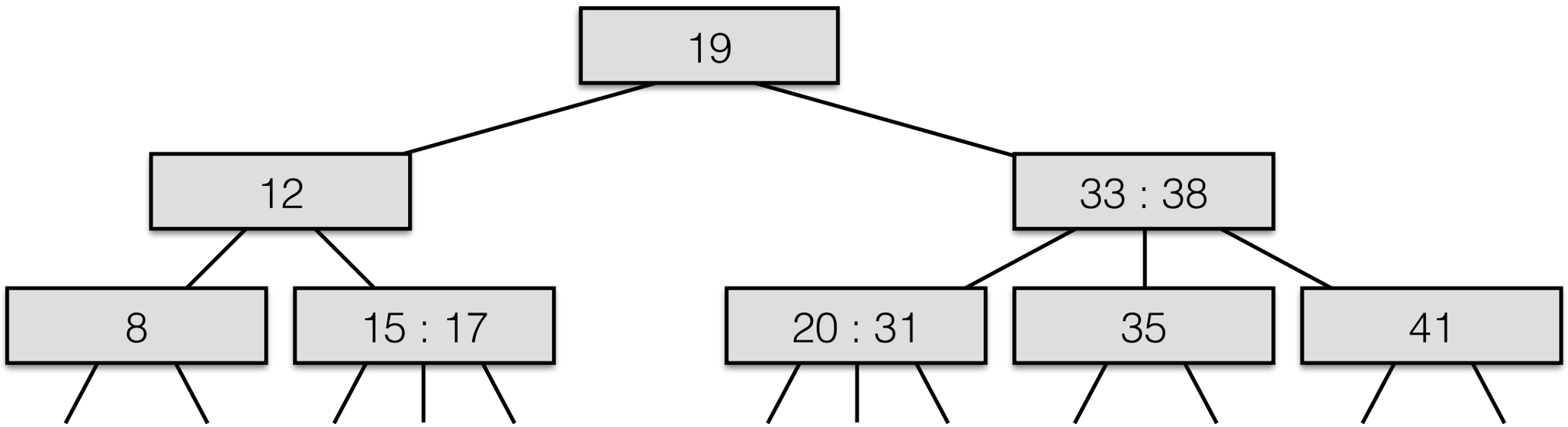


2-3-4 Trees: “Rules” (Inserting)


```
insert(tree, k_new):
     Search to position where k_new should be inserted.
     If node == (2-node || 3-node)
          insert k_new at correct position in node
     If node == (4-node)
          // assume parent != null except for root
          if node == root
               parent = create new root node as 2-node with k_2
          else
               promote k_2 up to join its parent node
          create two 2-nodes w/ k_1 and k_3, respectively.
          insert k_new into correct 2-node.
```

Insert: 20 (after)

[20 finally inserted!]



2-3-4 Trees: Problems…

- Tricky to implement… I guess you could:
  - Have 3 different types of nodes, and keep creating new ones as you need
    - —> complex to code; a lot of allocations; etc.
  - Have each node big enough to hold a 4-node
    - —> wastes a lot of space!
- Binary nodes are nice because:
  - they are simple and small
  - you always need at least TWO children anyway, so you can keep track of what's on either side of a particular key.
- The book has more info. about insertion as well as deletion. Generally speaking, there is a better way to do all of this (i.e., in a binary representation), so we will focus our remaining time on that.

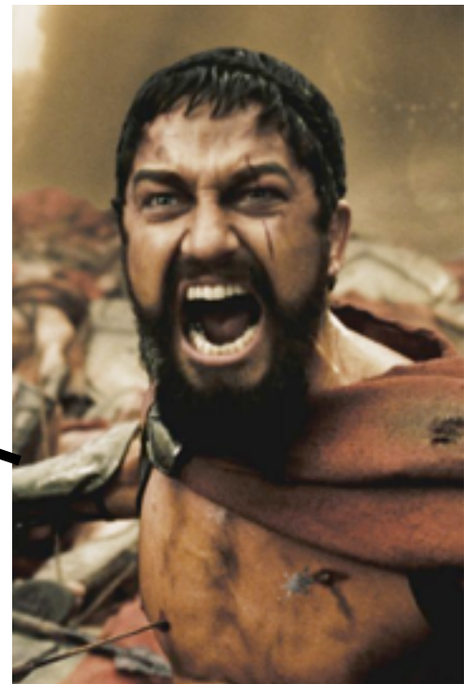
Question: *So how can we have nice balance of 2-3-4 trees, but w/ only binary nodes?!*

Red-Black Trees

Red-Black Trees

- Around 1984, Guibas and Sedgewick came up with a clever idea:
  - Translate each 2-, 3-, or 4-node into a miniature binary tree.
  - "Color" each vertex, so that we can tell which nodes belong together as part of a larger 2-3-4 tree node.

?!?! :)

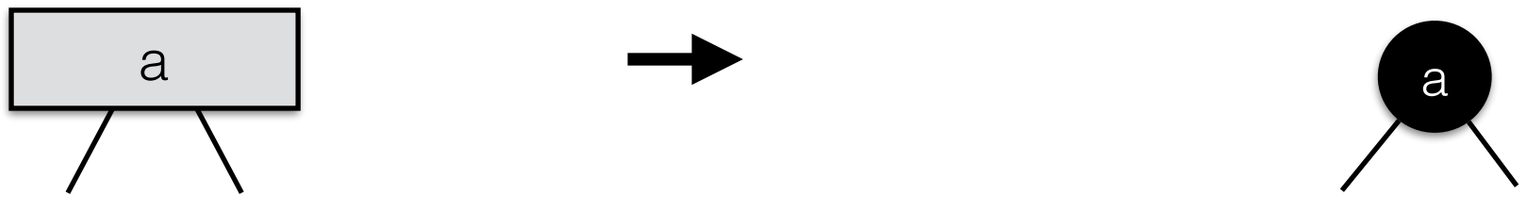


Fun Fact: The color "red" was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC

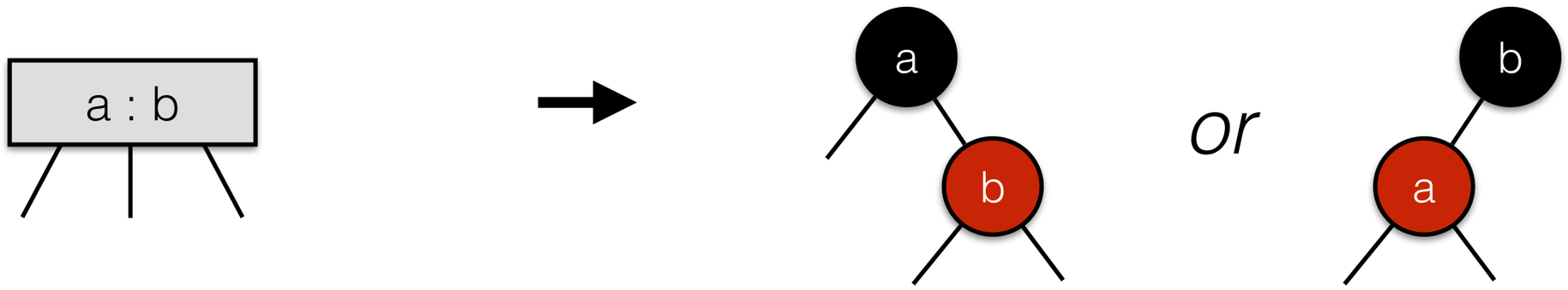
Red-Black Trees

- Translate each 2-, 3-, or 4-node into a miniature binary tree.
- “Color” each vertex, indicating which nodes belong together as part of a larger 2-3-4 node.

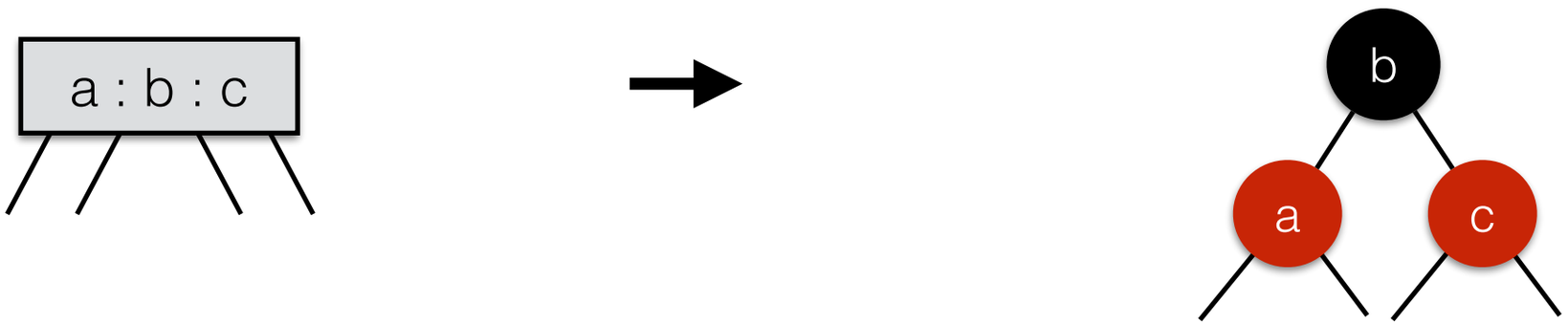
2-node



3-node

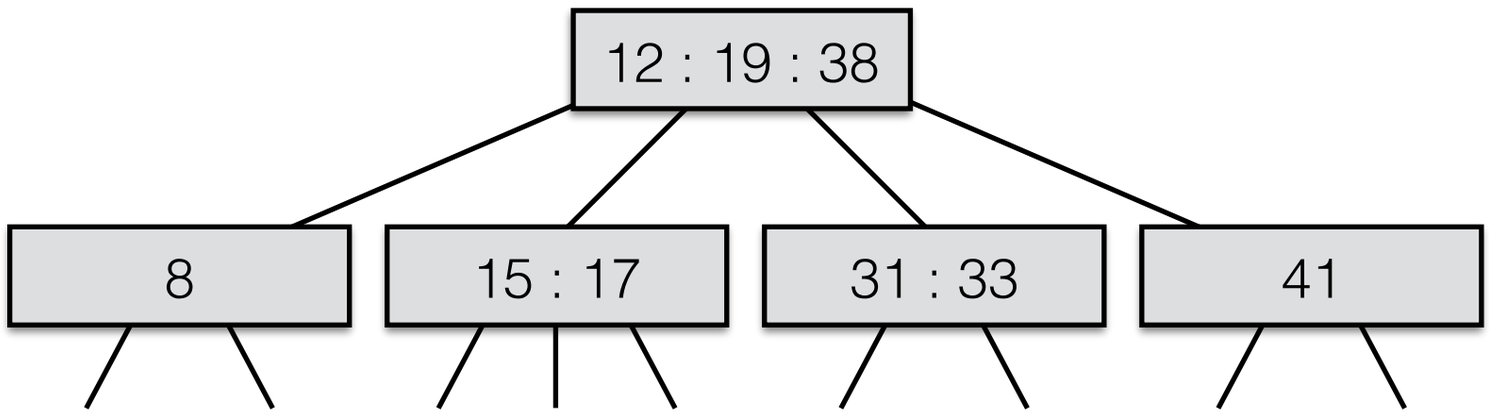


4-node

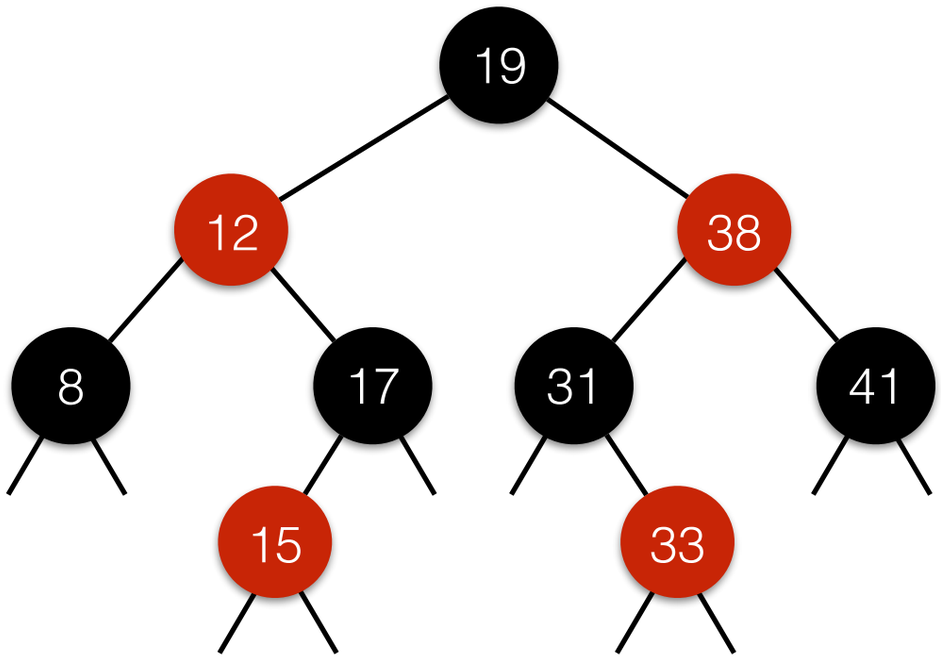


Here, we’ve basically encoded each 2-, 3-, and 4- node using only binary nodes, but we’ve painted each node with a “color” (red/black). We paint a node red if it’s part of a “bigger” node. This is called a Red-Black Tree.

Red-Black Trees



One simple way to think about Red-Black Trees is to just apply the 2-3-4 Tree rules (bearing in mind this new encoding)



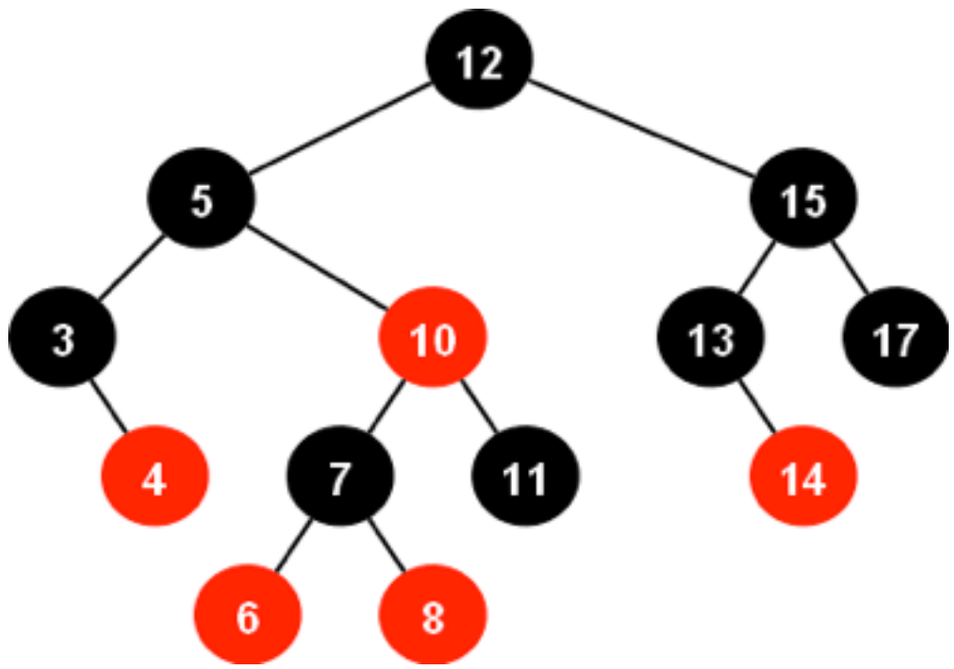
Red-Black Trees: Properties

You can also think of a Red-Black Tree as a structure separate from a 2-3-4 Tree.

<u>The rules (properties) are:</u>

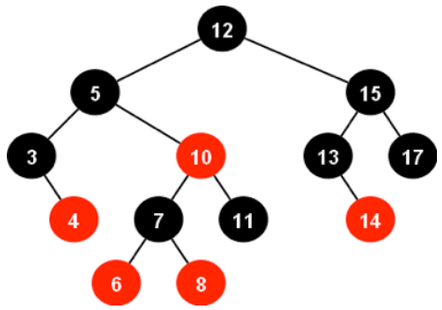
1. Every node is either **red** or **black**; by convention, the empty tree is a leaf, and is always considered **black**
2. The root is black. If an action colors it red, change it back to black.
3. For any node which is **red**, both of its children are **black** (this means: No two consecutive red nodes along any path!)
4. Every path from any node to a descendant leaf has the same number of black nodes.

[Write properties on board]



Red-Black Trees: Properties

Claim: *The Red-Black properties (1-4) mean that the depth of the tree is O(lg(n)) if there are n nodes in the tree.*



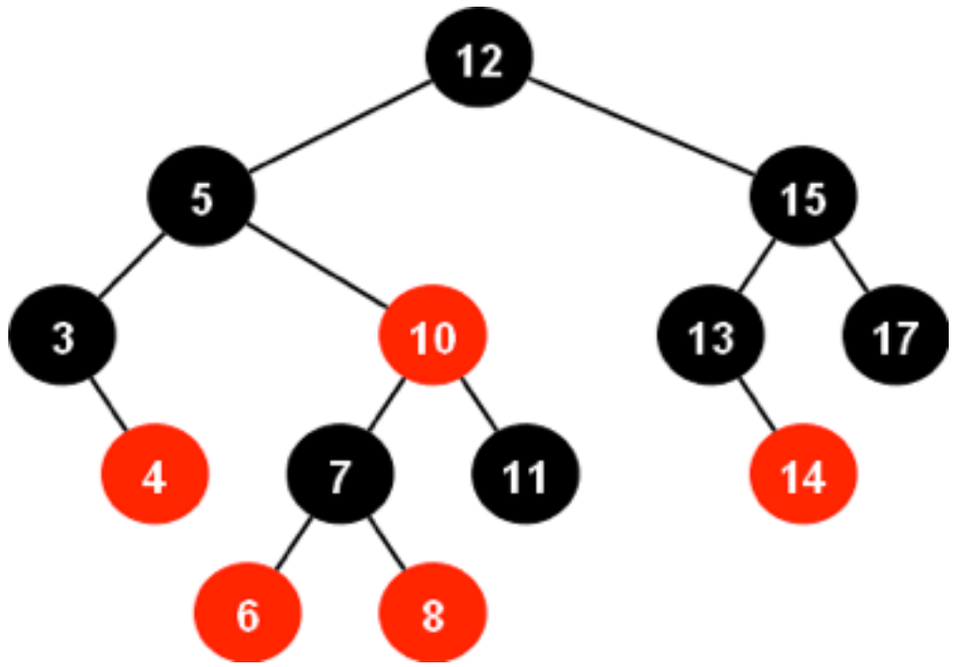
Informal justification:

- Since every path from the root to a leaf has the same number of black nodes (by property 4), ***the shortest possible path*** would be one which has NO red nodes in it.
- Suppose *k* is the number of black nodes along any path from the root to a leaf.
- How many red nodes could there be? At most *k*. By property 3, anytime you get a red node, your next node ***must*** be black. You can only do that *k* times before you run out of black nodes. So, the LONGEST possible path is at most 2 times the length of the shortest, or $h \leq 2k$.
- It can be shown that if each path from the root to a leaf has *k* black nodes, there must be ***at least*** $2^k - 1$ nodes in the tree. (1 node at root, 2 nodes at level 1, 4 nodes at level 2, etc. Add them up.) Since $h \leq 2k$, i.e. $k \geq h/2$, there must be at least $2^{(h/2)} - 1$ nodes in the tree. If there are *n* nodes in the tree, that means $n \geq 2^{(h/2)} - 1$. Adding 1 to both sides: $n + 1 \geq 2^{(h/2)}$ and taking the log (base 2) of both sides: $lg(n+1) \geq h/2 \text{ —> } 2\,lg(n + 1) \geq h$, which is $O(lg(n))$.

Thus the time complexity of the `lookup' operation is $O(h)$, which we just argued is $O(lg(n))$ in the worst case.

Red-Black Trees: Searching

- A Red-Black Tree *is a* Binary Search Tree
- Thus, searching takes time proportional to the height of the tree — O(lg(n))
- Done!
- Easy, right? =)
- Actually, no… The hard part is maintaining the tree…
- Let's consider inserting a key/value pair into a Red-Black Tree…



R/B Insertion


YOUR MOMMA'S SO FAT
SHE CAN SIT ON A BINARY TREE AND FLATTEN IT TO A LINKED LIST IN O(1) TIME
imgflip.com

Red-Black Trees: Insertion

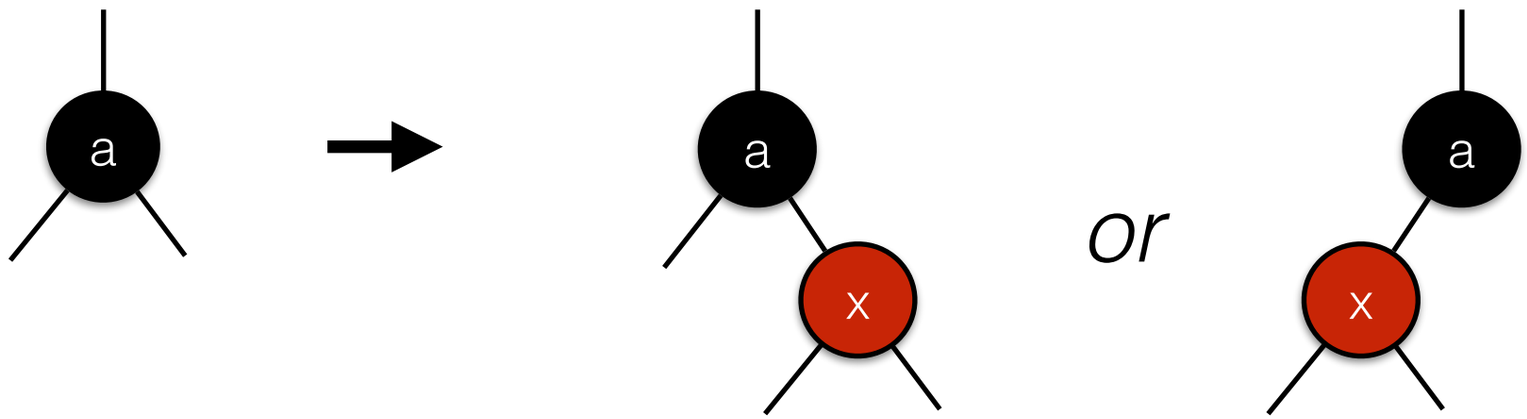
- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

R/B insert: case 1

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 1: *A 2-node (black node w/ both child nodes colored black) becomes a 3-node — this is okay (no violation of rule. 3)*

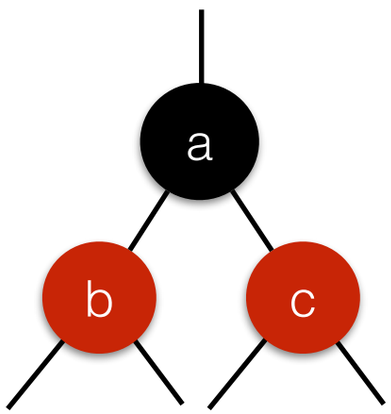


R/B insert: case 2

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 2: *A 4-node (black node with red children)*

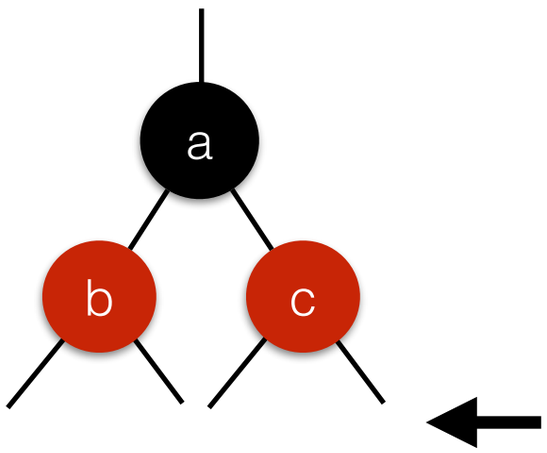


x to be inserted at one of these locations…
but we can't have consecutive red nodes!

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 2: *A 4-node (black node with red children) — must split the node, "promoting" the middle key up in the tree. We could "join" (a) w/ its parent and "unjoin" (b) and (c) from (a) — This amounts to a "color flip" …*

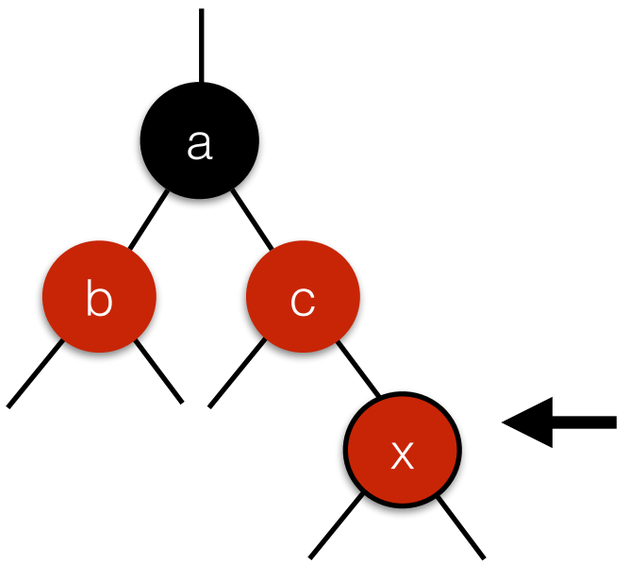


x to be inserted at one of these locations… but we can't have consecutive red nodes!

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 2: *A 4-node (black node with red children) — must split the node, "promoting" the middle key up in the tree. We could "join" (a) w/ its parent and "unjoin" (b) and (c) from (a) — This amounts to a "color flip" …*

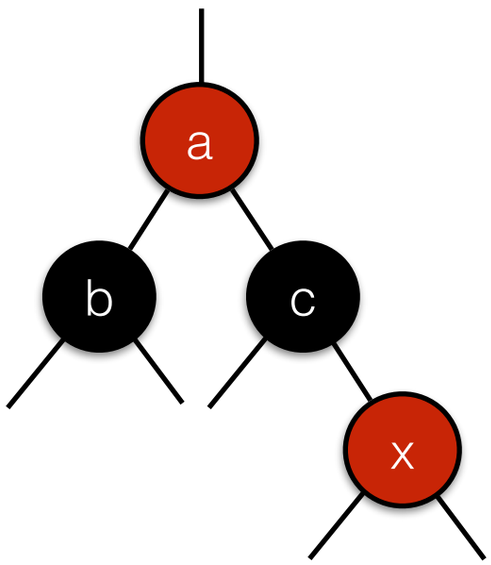


x to be inserted at one of these locations… but we can't have consecutive red nodes!

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 2: *A 4-node (black node with red children) — must split the node, "promoting" the middle key up in the tree. We could "join" (a) w/ its parent and "unjoin" (b) and (c) from (a) — This amounts to a "color flip" …*

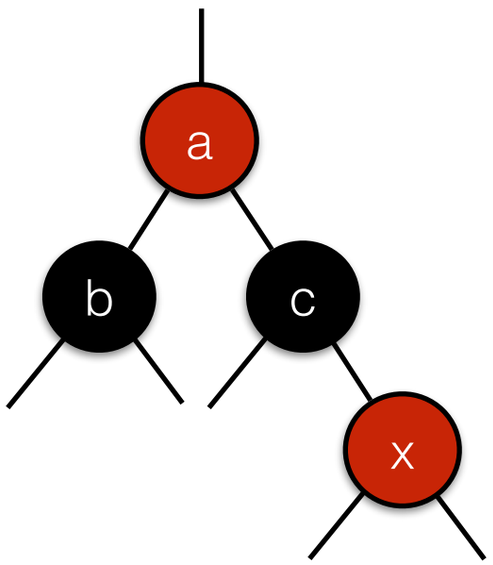


x can be inserted anywhere and its okay (haven't changed the "black height" of anything; the new node is red; just switched order of red/black in the subtree.

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

Case 2: *A 4-node (black node with red children) — must split the node, "promoting" the middle key up in the tree. We could "join" (a) w/ its parent and "unjoin" (b) and (c) from (a) — This amounts to a "color flip" …*



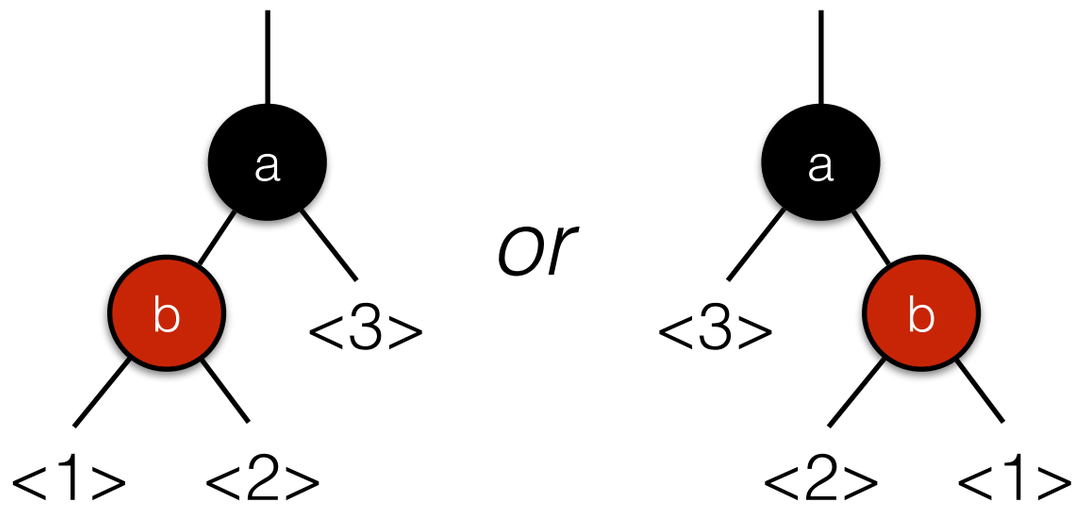
Haven't broken other properties, and rule 3 is okay here — but, we have to make sure we did not violate any properties higher up in the tree!

R/B insert: case 3

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*



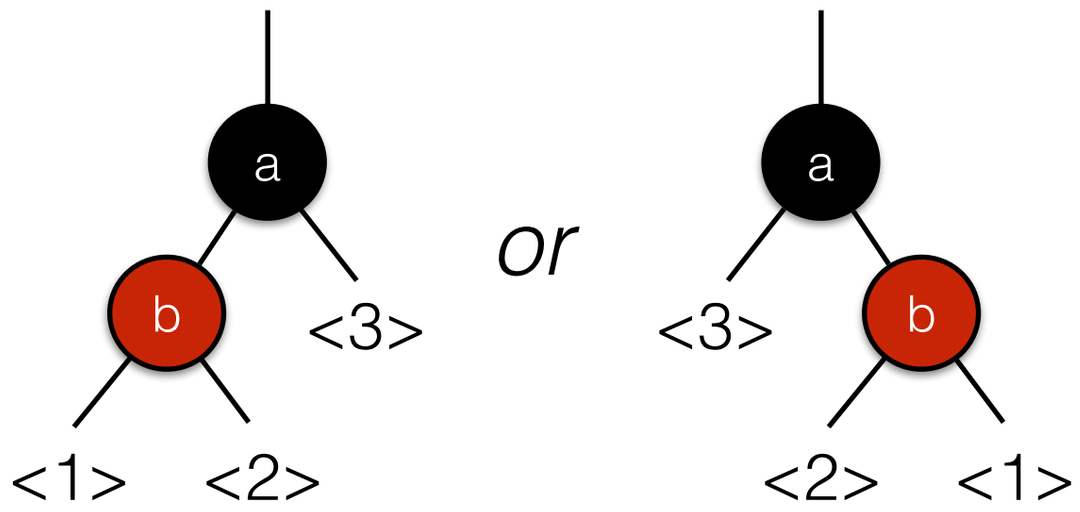
R/B insert: case 1 (a)

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(a) *[before] if inserting at #3, no problem (basically making 4-node)*

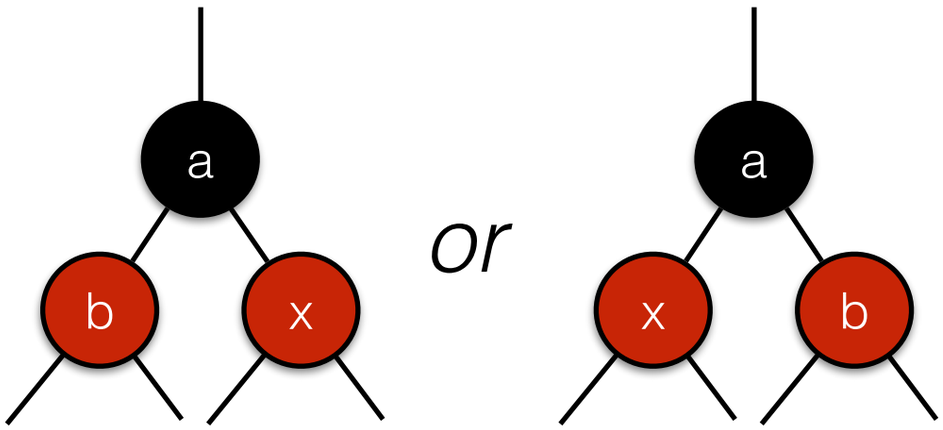


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(a) *[before] if inserting at #3, no problem (basically making 4-node)*



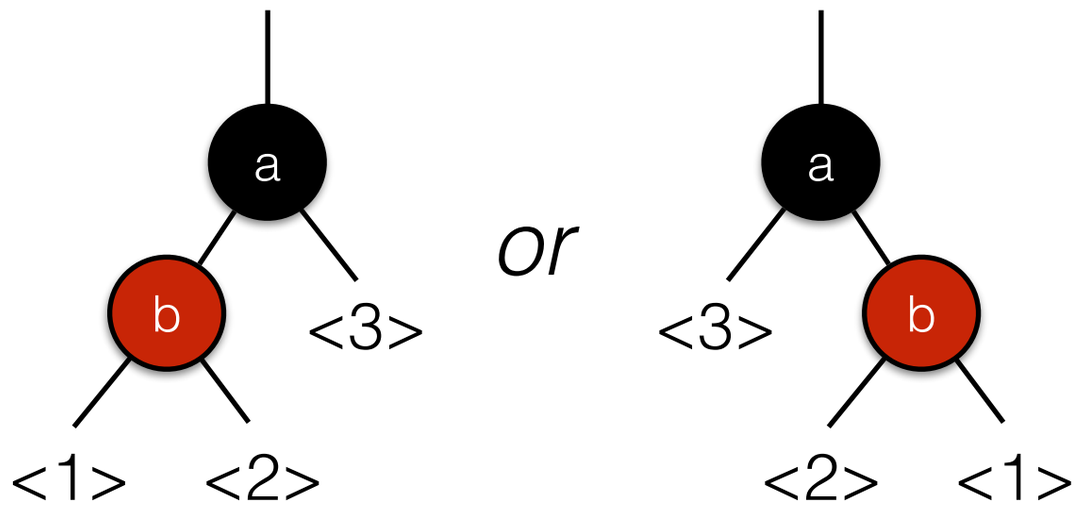
R/B insert: case 1 (b)

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(b) *[before] if inserting at #1…*

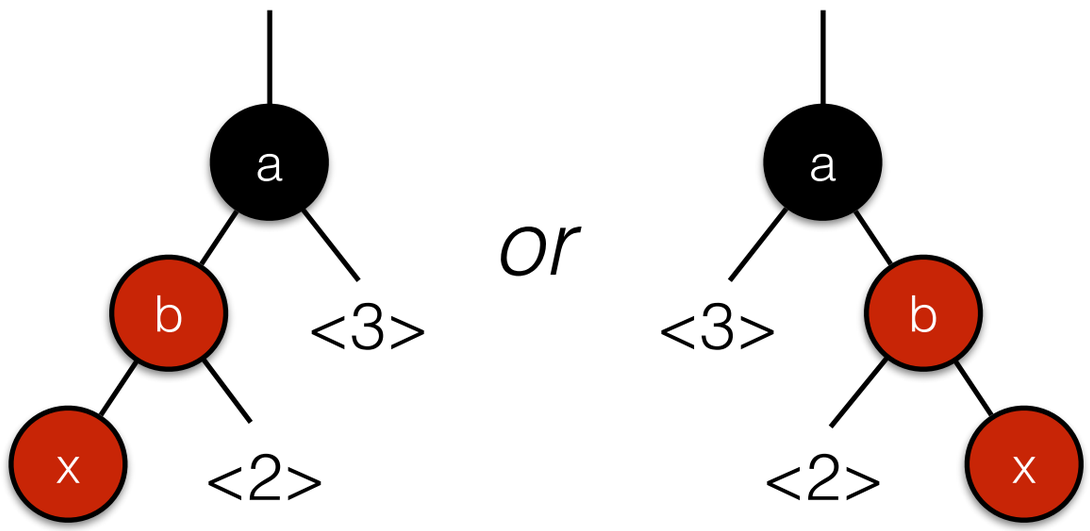


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(b) *[before] if inserting at #1, then you get 2 red nodes in a row!*

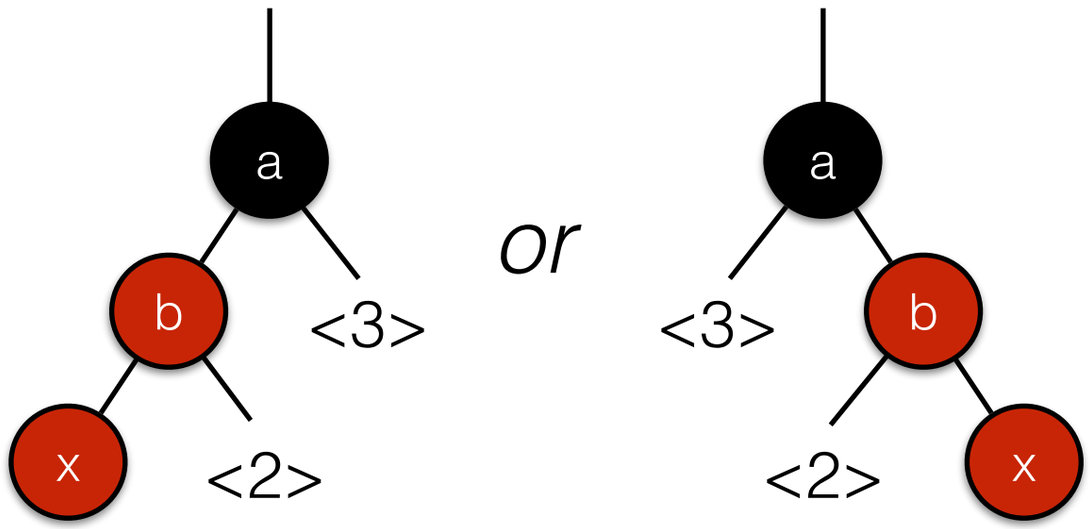


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(b) *[before] if inserting at #1, then you get 2 red nodes in a row! Since x < b < a [left], we could fix this operation by "rotating" this whole structure — "lift" b up to the root of this structure (colored black), while dropping a down to be the right child of b (colored red), and leaving x as the left child (colored red).*

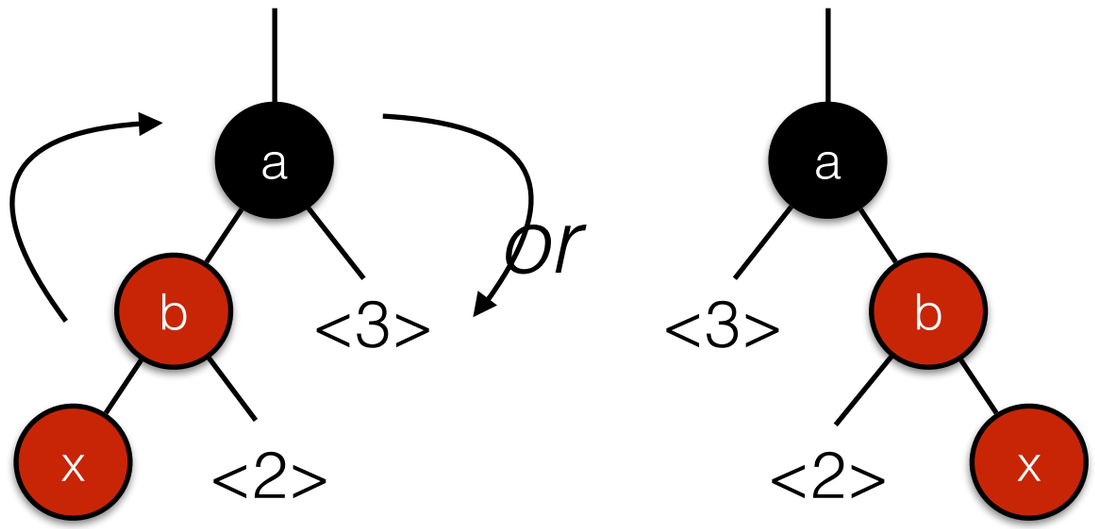


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(b) *[before] if inserting at #1, then you get 2 red nodes in a row! Since x < b < a [left], we could fix this operation by "rotating" this whole structure — "lift" b up to the root of this structure (colored black), while dropping a down to be the right child of b (colored red), and leaving x as the left child (colored red).*

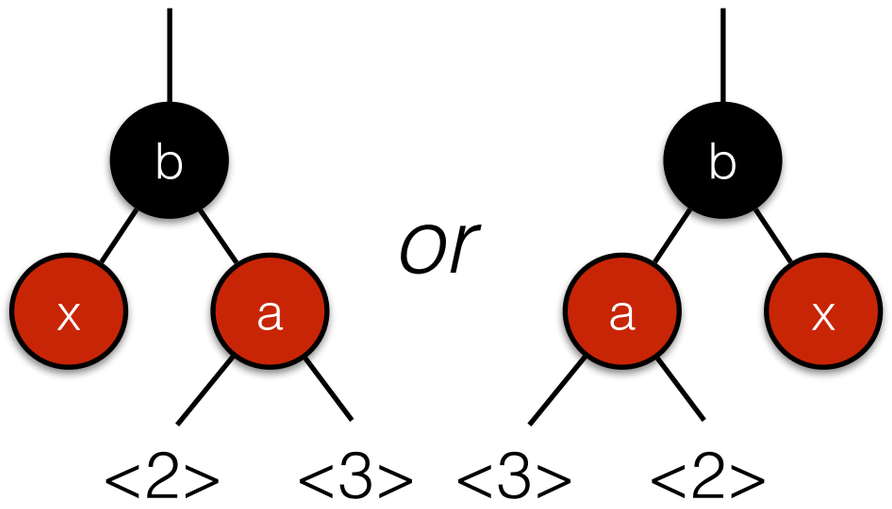


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(b) *[after] if inserting at #1, then you get 2 red nodes in a row! Since x < b < a [left], we could fix this operation by "rotating" this whole structure — "lift" b up to the root of this structure (colored black), while dropping a down to be the right child of b (colored red), and leaving x as the left child (colored red).*

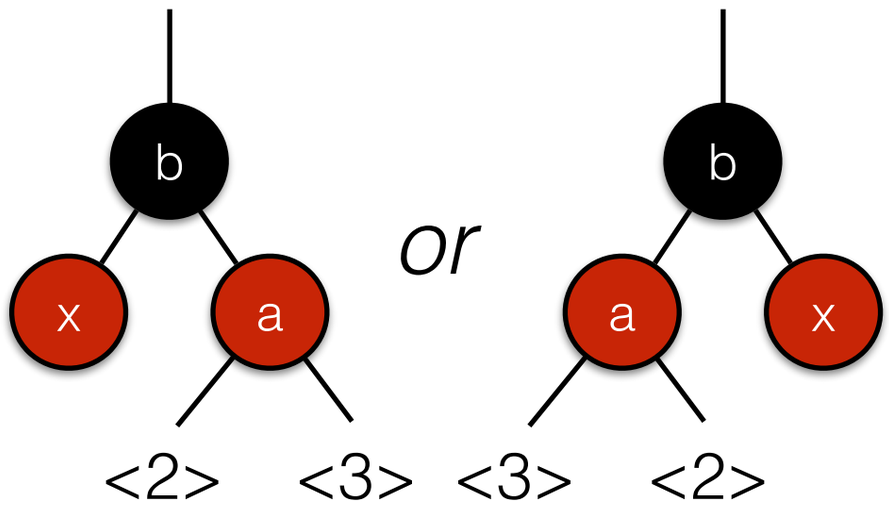


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

Note: *we haven't changed the number of black nodes along any path by doing this, and we've fixed the double red. And sub-trees #2 and #3 are still ordered properly. This is called a* ***single rotation****.*



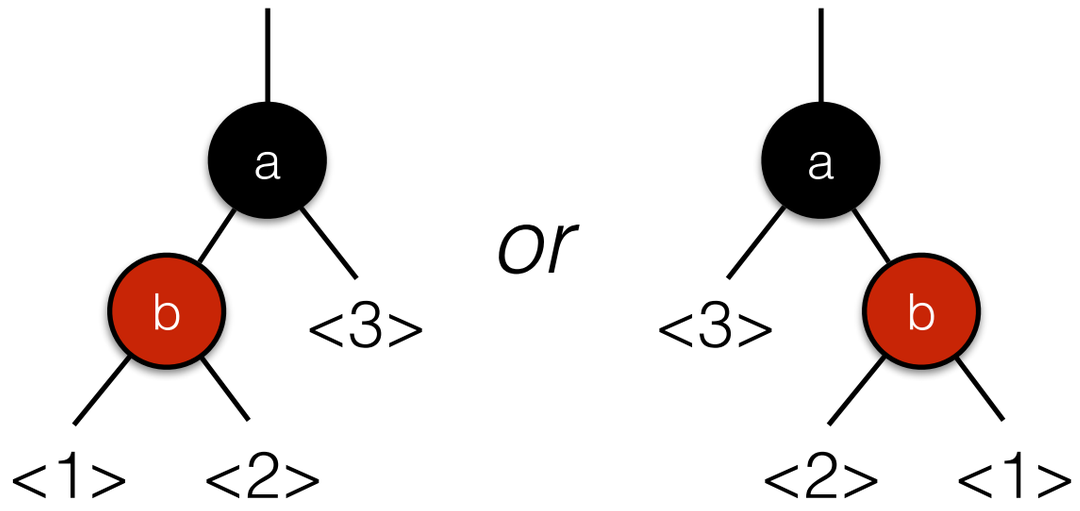
R/B insert: case 1 (c)

Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(c) *[before] if inserting at #2 …*

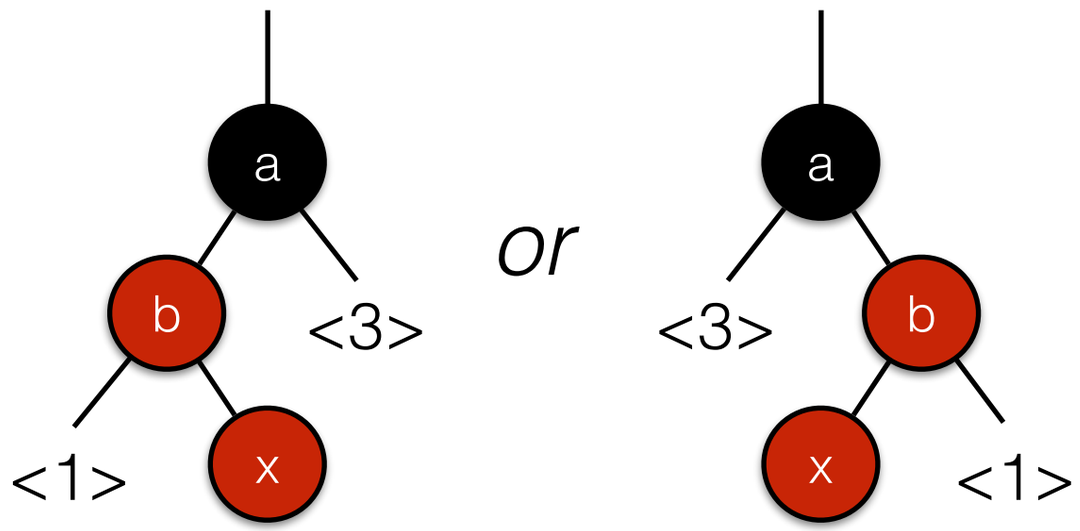


Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(c) *[before] if inserting at #2, then have two red nodes in a row, but in a "zig-zag" pattern.*



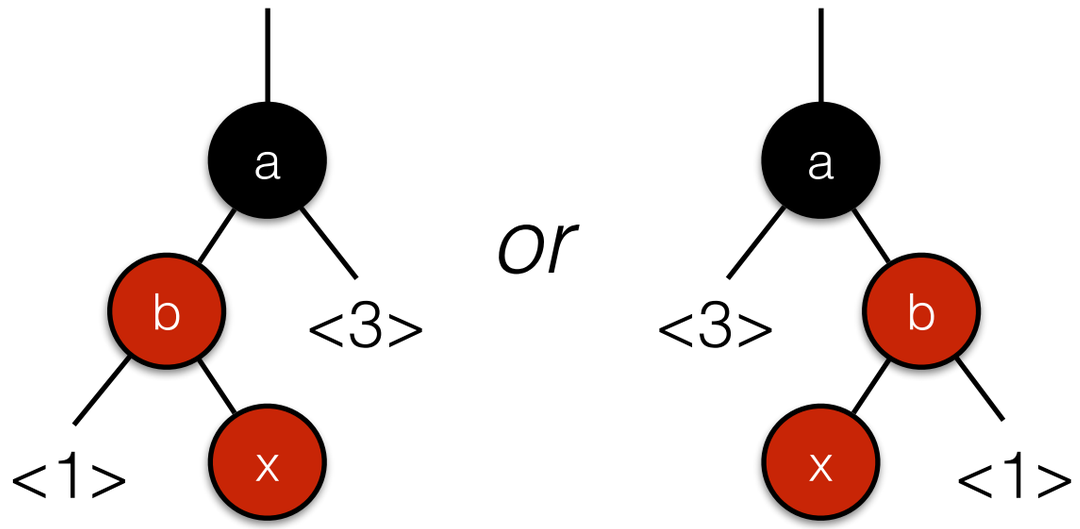
Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

(c) *[before] if inserting at #2, then have two red nodes in a row, but in a "zig-zag" pattern.*

Now, the "middle" element is x, so we want to "lift" x up to be the root (colored black), and have a and b fall to be the children (colored red)



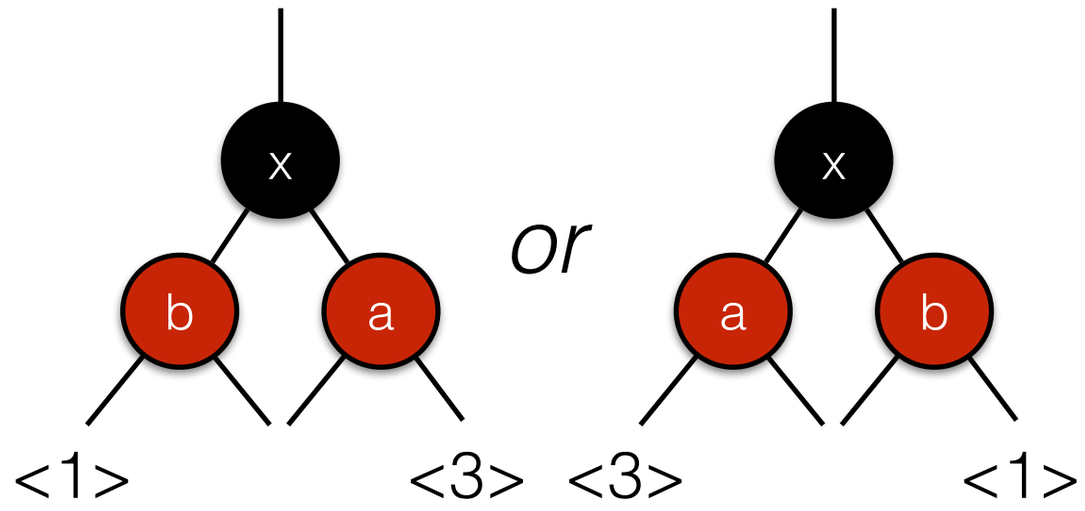
Red-Black Trees: Insertion

- As usual for a binary search tree, find the location where the new element goes, and insert it there.
- Initially, color it RED. This insures that rules 1, 2 and 4 are preserved.
- But rule 3 might be violated (a red node has black children). There are several cases (assume we're inserting a key x).

> ***Case 3:*** *A 3-node (black root with one red child) — can be tricky depending on where you are inserting…*

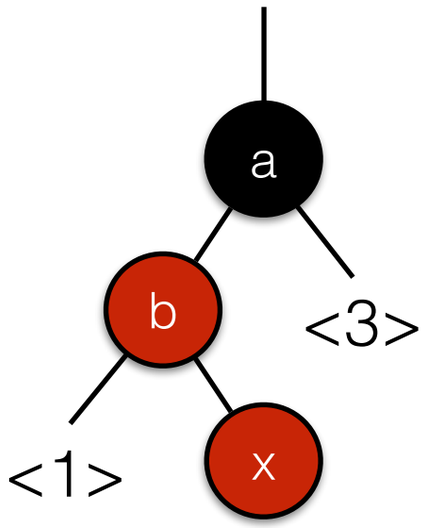
(c) *[before] if inserting at #2, then have two red nodes in a row, but in a "zig-zag" pattern.*

Now, the "middle" element is x, so we want to "lift" x up to be the root (colored black), and have a and b fall to be the children (colored red)



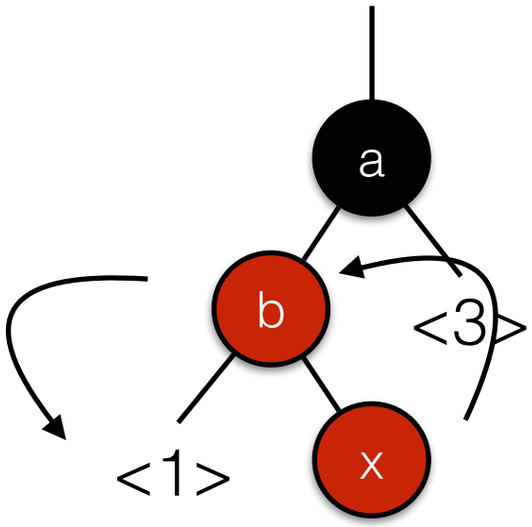
Red-Black Trees: Insertion

Note: *You can think about this as two single rotations if that is helpful — once "around" b and then around x. Here it is shown only for the left case, but the other case is symmetric.*



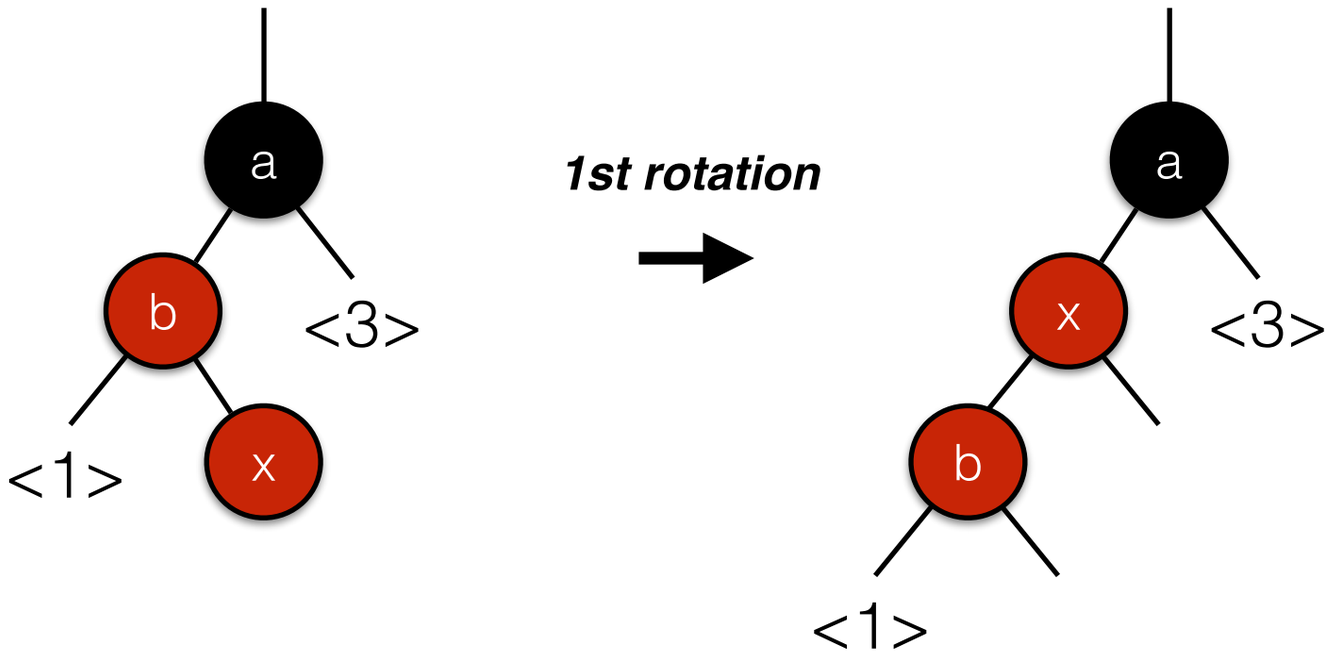
Red-Black Trees: Insertion

Note: *You can think about this as two single rotations if that is helpful — once "around" b and then around x. Here it is shown only for the left case, but the other case is symmetric.*



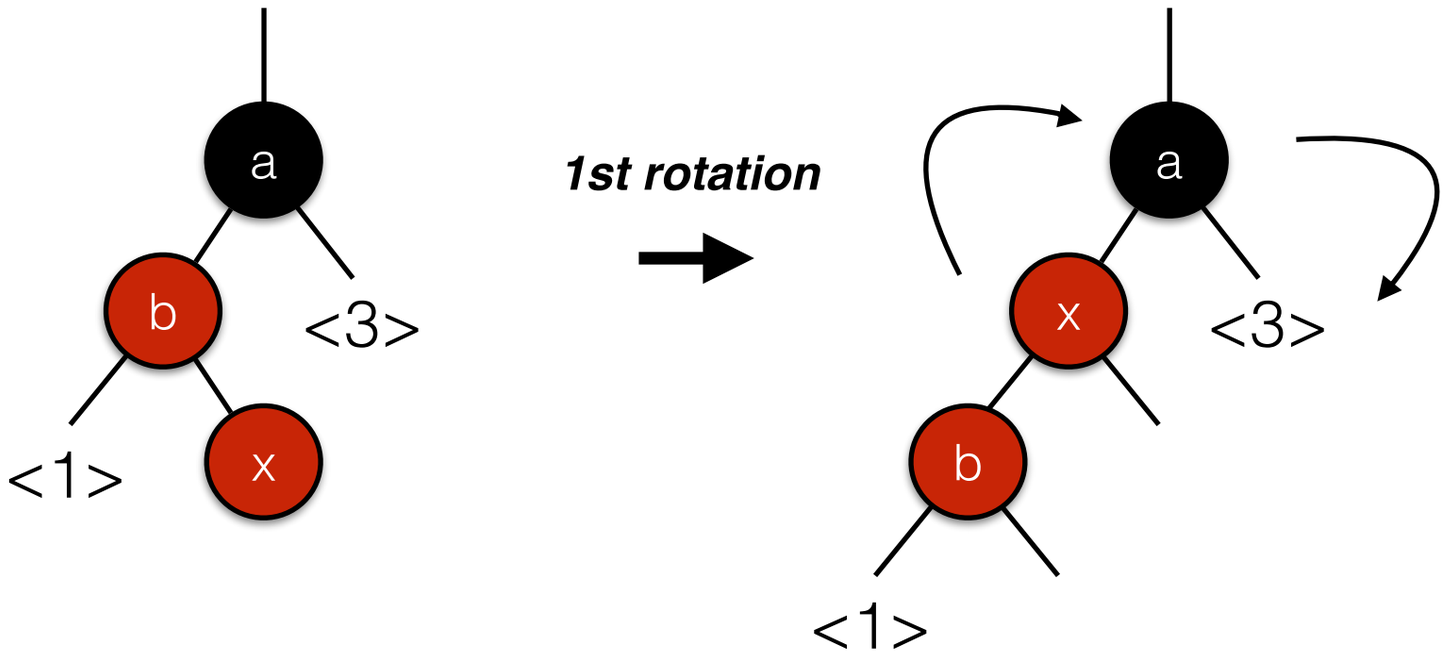
Red-Black Trees: Insertion

Note: *You can think about this as two single rotations if that is helpful — once "around" b and then around x. Here it is shown only for the left case, but the other case is symmetric.*



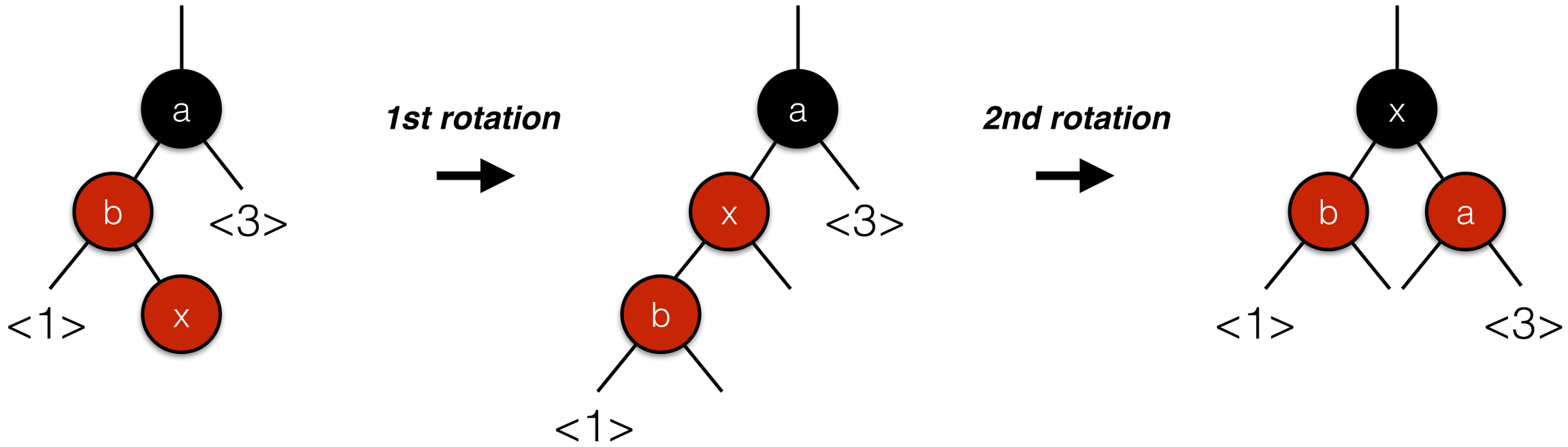
Red-Black Trees: Insertion

Note: *You can think about this as two single rotations if that is helpful — once "around" b and then around x. Here it is shown only for the left case, but the other case is symmetric.*



Red-Black Trees: Insertion

Note: *You can think about this as two single rotations if that is helpful — once "around" b and then around x. Here it is shown only for the left case, but the other case is symmetric.*



*Thus, this operation is known as a **double rotation**. The resulting 4-node will be the same regardless of which 3-node version you start from.*

Red-Black Trees: Insertion

Note: *In the worst case we only have to fix colors along the path between the new node and the root of this structure — that is only a constant factor of work (negligible).*

There is a formal proof to show that we only need to do at most one single-rotation or double-rotation to fix the tree — all other changes can be done w/ color-flips [see the textbook].

—> Insert is O(lg(n)).

R/B Deletion

Red-Black Trees: Deletion

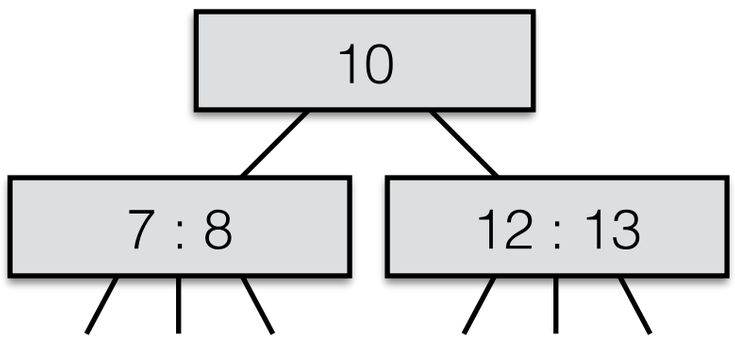
- Even trickier than insertion…
- Still can be done in O(lg(n)) time, worst case!
- More complex because…
  - there is one way to split nodes, but…
  - there are several ways to merge nodes together (which you sometimes have to do when deleting)…
- **The key idea:** make it so we just have to delete a node at the bottom of the tree!
  - If node to delete is *internal* — find immediate pred. or succ. and use its key as a replacement for the one we want to delete (just like regular ol' BSTs).
  - Then we just have to delete the pred./succ. at the bottom of the tree.
  - Again, we have some cases…

R/B delete: case 1 (3- or 4- node)

Red-Black Trees: Deletion

Case 1: 3- or 4- node.

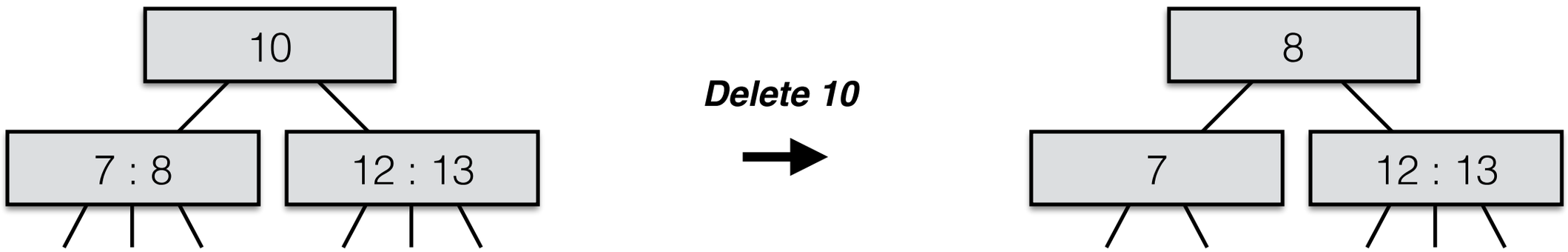
- **2-3-4 Tree:** 3- or 4- node delete it — easy!
  - Ex. delete 10.
  - Find immediate pred./succ.
  - Bring it up to replace 10….
    - This means we have to delete 8 out of the 3 node that it currently belongs to (easy!)



Red-Black Trees: Deletion

Case 1: 3- or 4- node.

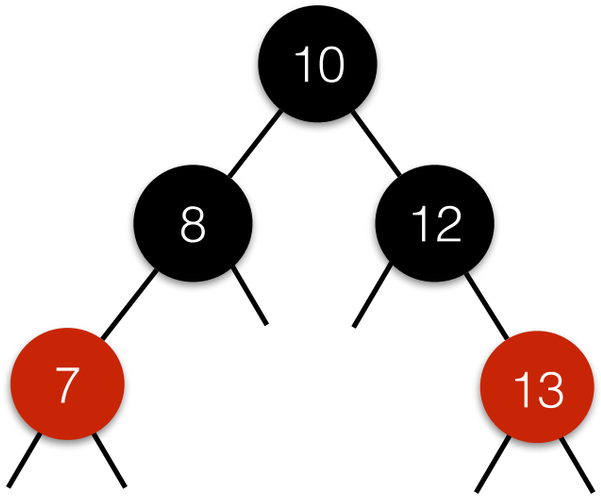
- **2-3-4 Tree:** 3- or 4- node delete it — easy!
  - Ex. delete 10.
  - Find immediate pred./succ.
  - Bring it up to replace 10….
    - This means we have to delete 8 out of the 3 node that it currently belongs to (easy!)



Red-Black Trees: Deletion

Case 1: 3- or 4- node.

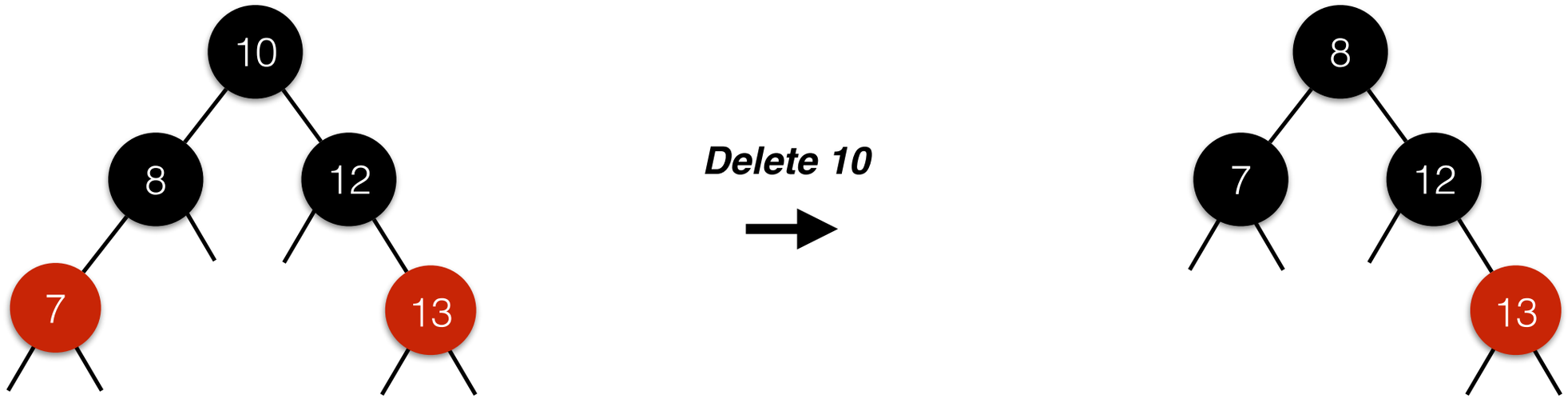
- **Red-Black Tree:** Node to delete is red or its child is red.
  - Replace with 8.
  - Color the child black, and we are done!



Red-Black Trees: Deletion

Case 1: 3- or 4- node.

- **Red-Black Tree:** Node to delete is red or its child is red.
  - Replace with 8.
  - Color the child black, and we are done!

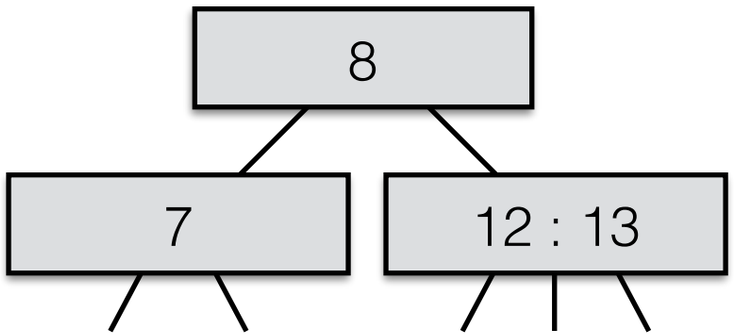


R/B delete: case 1 (2-node)

Red-Black Trees: Deletion

Case 2: 2-node (a)

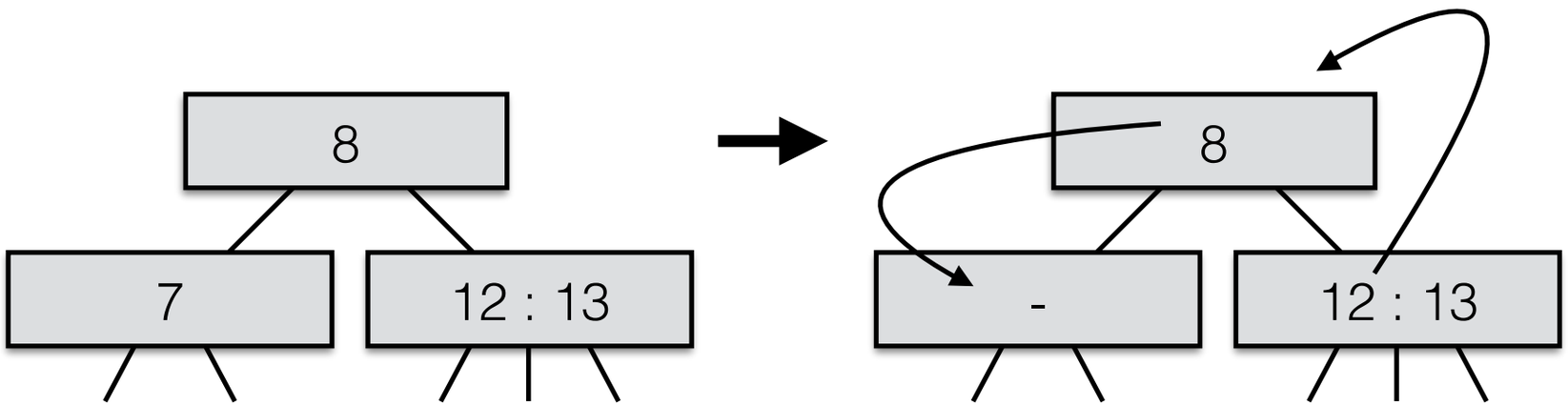
- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If w is an adjacent sibling node of *v* that is a 3- or 4- node, we adopt (steal?) a child from *w* and give it to *v.*
  - To maintain order, we have to move the key from *w* up to parent, and a key from parent down to *v.*
  - Ex. Delete 7.



Red-Black Trees: Deletion

Case 2: 2-node (a)

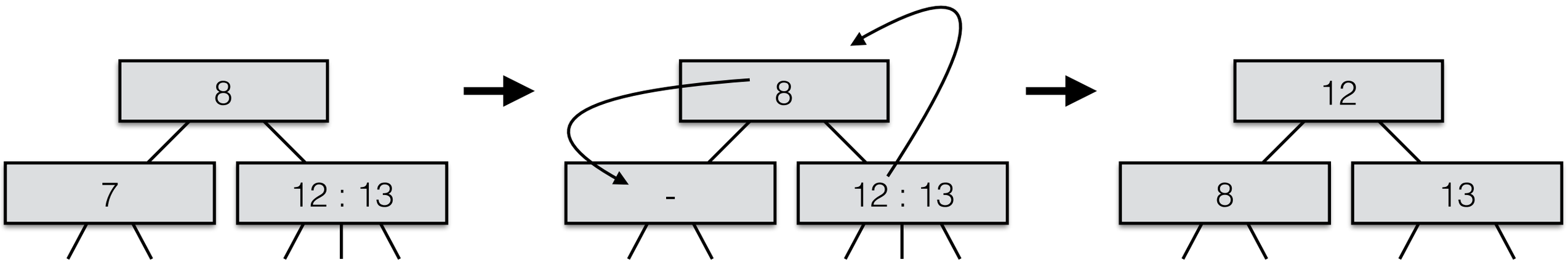
- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If w is an adjacent sibling node of *v* that is a 3- or 4- node, we adopt (steal?) a child from *w* and give it to *v.*
  - To maintain order, we have to move the key from *w* up to parent, and a key from parent down to *v.*
  - Ex. Delete 7.



Red-Black Trees: Deletion

Case 2: 2-node (a)

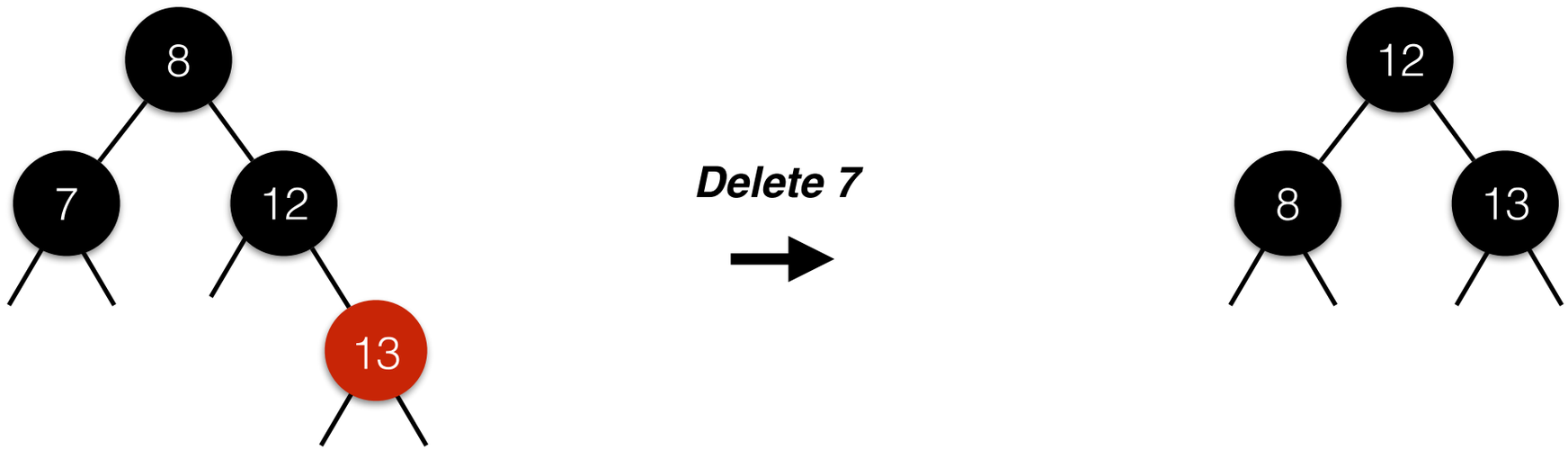
- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If w is an adjacent sibling node of *v* that is a 3- or 4- node, we adopt (steal?) a child from *w* and give it to *v.*
  - To maintain order, we have to move the key from *w* up to parent, and a key from parent down to *v.*
  - Ex. Delete 7.



Red-Black Trees: Deletion

Case 2: 2-node (a)

- **Red-Black Tree:** “trinode reconstruction”
  - The book has more details — check it out!

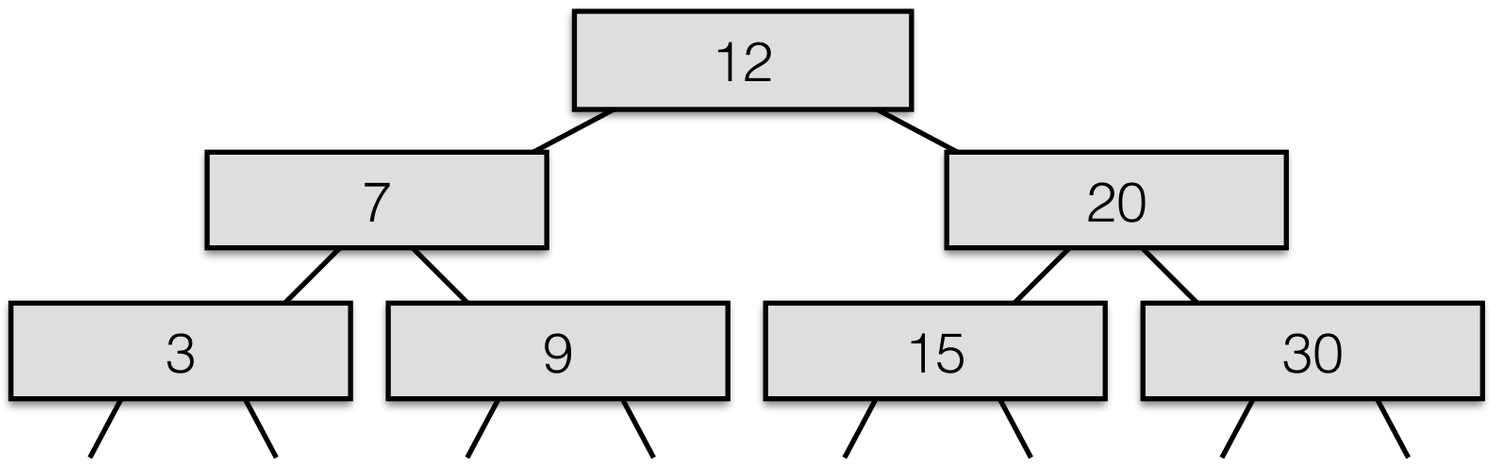


Red-Black Trees: Deletion

Case 2: 2-node (b)

- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If all adjacent sibling nodes of *v* are 2-nodes, we fuse/merge *v* with a sibling.
  - To maintain order, we need a key between them — pull one down from the parent.
  - If parent is 3- or 4- node, easy; if not, we have passed the "underflow" up to the parent, which then must resolve this by adopting or fusing as described previously.
  - Notes:
    - Fusing can be passed up the tree all the way to the root!
    - If the root loses its only key, it disappears and the tree becomes 1 level shorter.
- Ex. Delete 3.

(0)

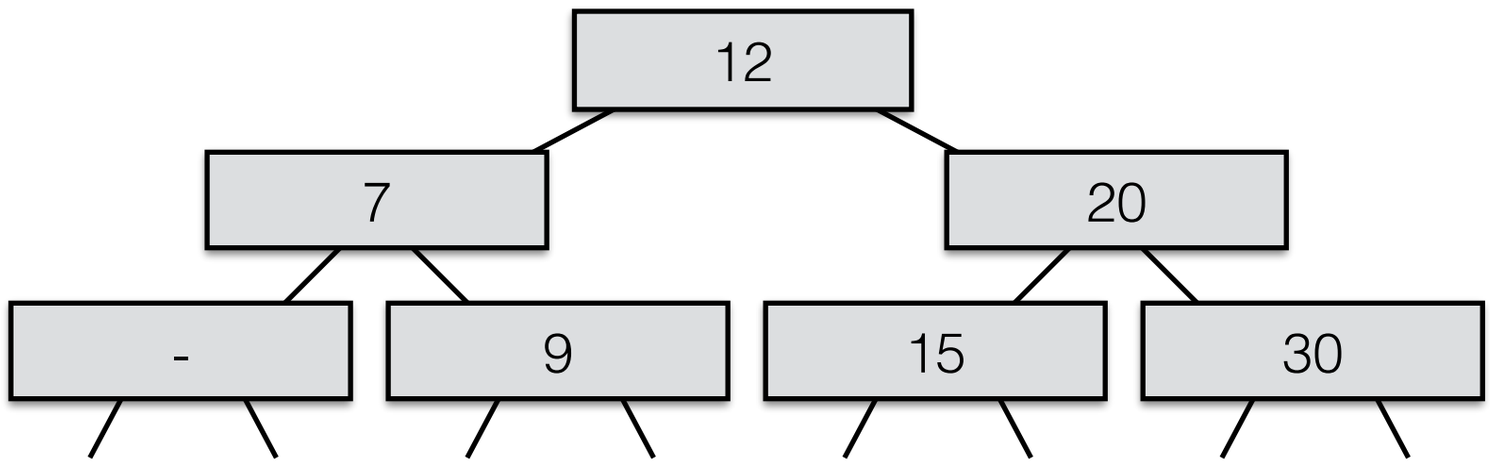


Red-Black Trees: Deletion

Case 2: 2-node (b)

- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If all adjacent sibling nodes of *v* are 2-nodes, we fuse/merge *v* with a sibling.
  - To maintain order, we need a key between them — pull one down from the parent.
  - If parent is 3- or 4- node, easy; if not, we have passed the "underflow" up to the parent, which then must resolve this by adopting or fusing as described previously.
  - Notes:
    - Fusing can be passed up the tree all the way to the root!
    - If the root loses its only key, it disappears and the tree becomes 1 level shorter.
- Ex. Delete 3.

(1)

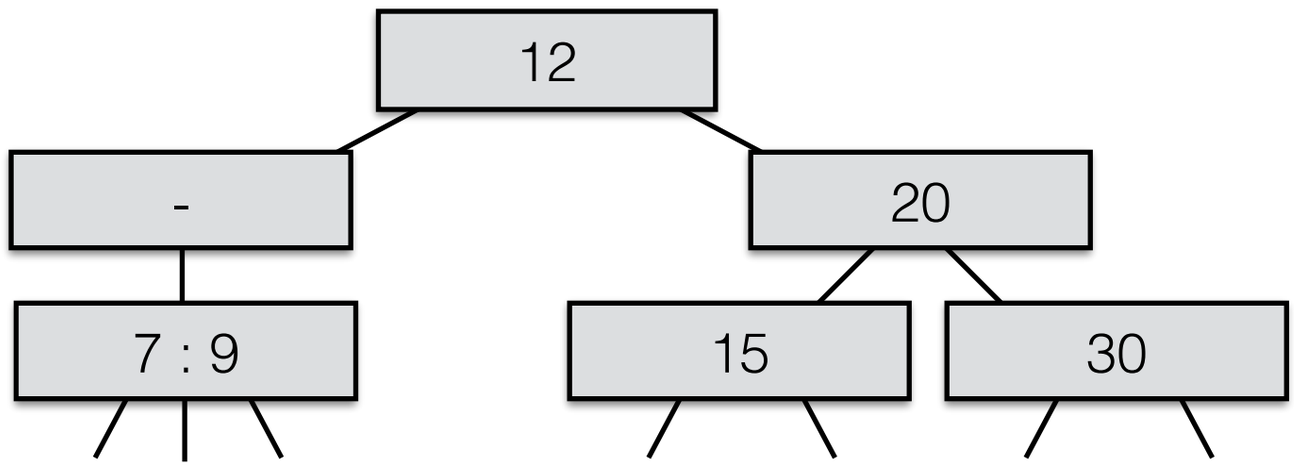


Red-Black Trees: Deletion

Case 2: 2-node (b)

- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If all adjacent sibling nodes of *v* are 2-nodes, we fuse/merge *v* with a sibling.
  - To maintain order, we need a key between them — pull one down from the parent.
  - If parent is 3- or 4- node, easy; if not, we have passed the "underflow" up to the parent, which then must resolve this by adopting or fusing as described previously.
  - Notes:
    - Fusing can be passed up the tree all the way to the root!
    - If the root loses its only key, it disappears and the tree becomes 1 level shorter.
- Ex. Delete 3.

(2)

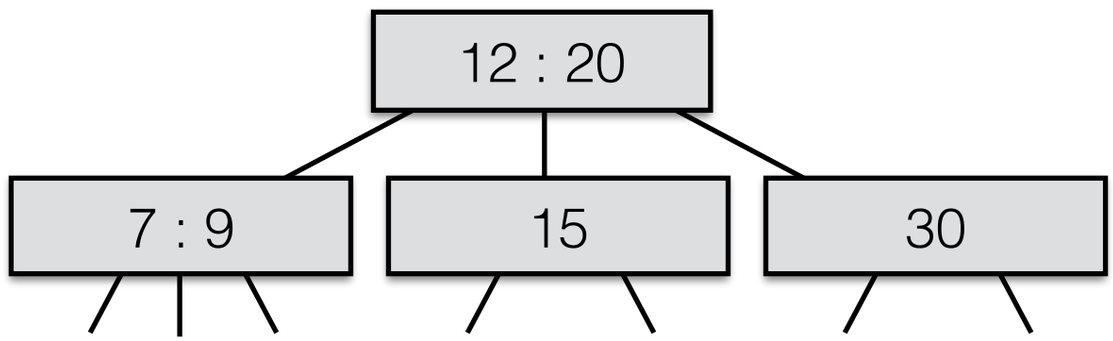


Red-Black Trees: Deletion

Case 2: 2-node (b)

- **2-3-4 Tree:** Deleting a 2-node leaves a 1-node — not valid…
  - If all adjacent sibling nodes of *v* are 2-nodes, we fuse/merge *v* with a sibling.
  - To maintain order, we need a key between them — pull one down from the parent.
  - If parent is 3- or 4- node, easy; if not, we have passed the "underflow" up to the parent, which then must resolve this by adopting or fusing as described previously.
  - Notes:
    - Fusing can be passed up the tree all the way to the root!
    - If the root loses its only key, it disappears and the tree becomes 1 level shorter.
- Ex. Delete 3.

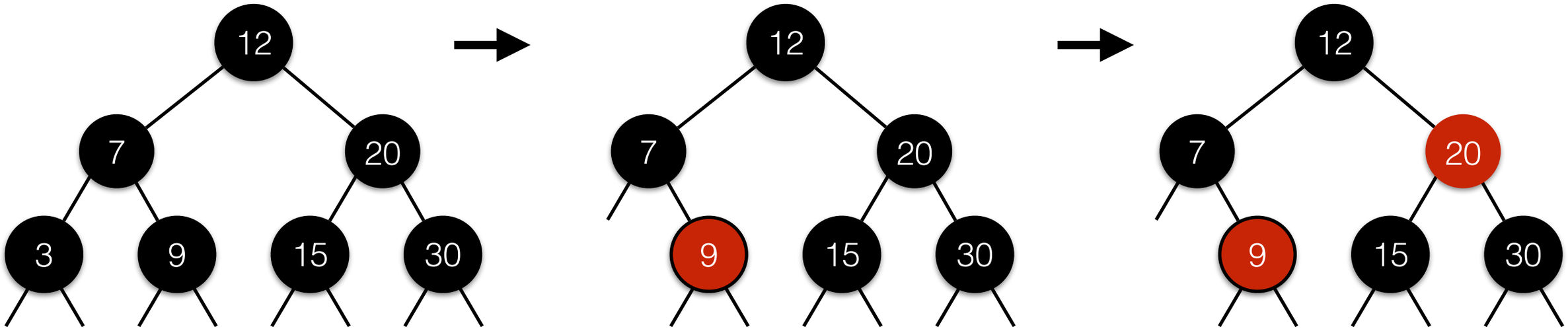
(3)



Red-Black Trees: Deletion

Case 2: 2-node (b)

- **Red-Black Tree:** Essentially involves a recoloring that passes the issue up to the parent where it must be resolved.



The ideas are straightforward, but the implementation can be a bit messy. The book covers it in detail.

Synopsis

BSTs are a good data structure for implementing the Dictionary ADT, but their performance is best when they are kept in balance.

- Keeping "perfect" balance turns out to be too much work.
- But we can make it work if we either give up "binary" (have search trees with variable numbers of children), *or* give up "perfect" (allow trees to be "somewhat unbalanced")
- **2-3-4 trees:** Use the first strategy.
  - All leaves are at the same level, all paths the same length.
  - But it is clumsy to implement due to variable-size nodes.
- **Red-Black trees:** Use the second strategy.
  - Encode 2-3-4 nodes as "mini-trees", nodes colored red to indicate they are conjoined with their parent.
  - Use "rotations" and "color flips" to keep the tree in balance, takes no more than O(lg(n)) time. Also, only do one restructuring adjustment for a insertion or deletion. This is important for some algorithms that you will see if you take CS 31.
  - Doing this, we get guaranteed O(lg(n)) runtime for all the Map operations -- insert, lookup, delete.

Java uses Red-Black Trees for TreeMap — check it out! :)

Announcement: Dartmouth Hackathon