CS 10: Problem solving via Object Oriented Programming

Winter 2017

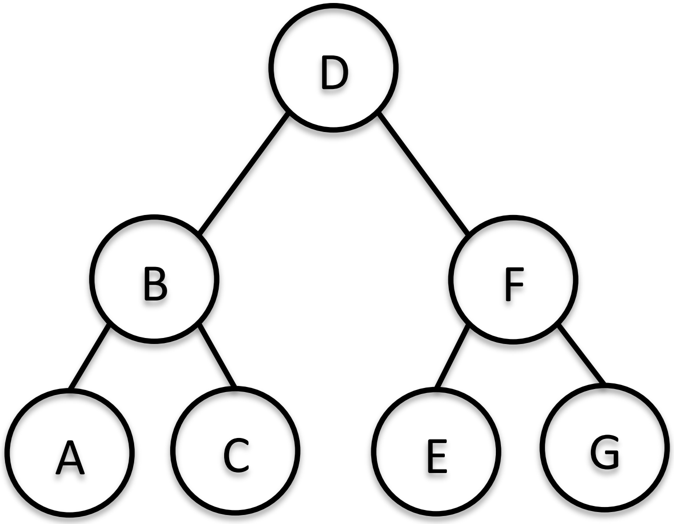
Tim Pierson
260 (255) Sudikoff

Balance

Agenda

1. Balanced Binary Trees
2. 2-3-4 Trees
3. Red-Black Trees
4. Deletion in 2-3-4 and Red-Black trees

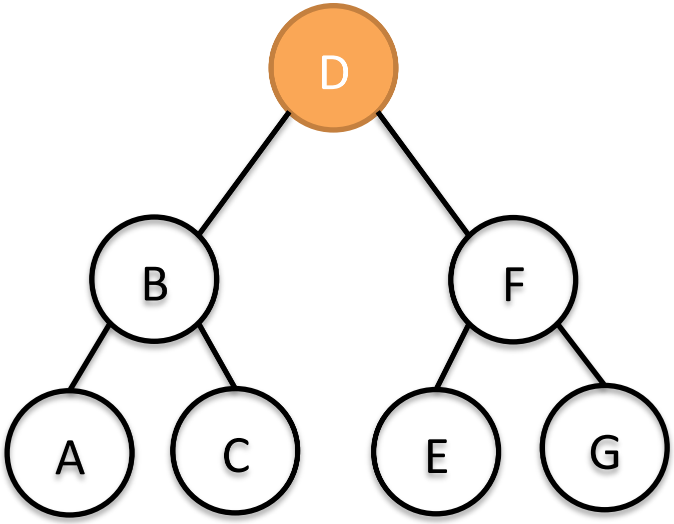
Review: Binary Search Trees (BSTS) are an ordered collection of key/value nodes



Binary Search Tree property
Let x be a node in a binary search tree s.t.:

- left.key < x.key
- right.key > x.key

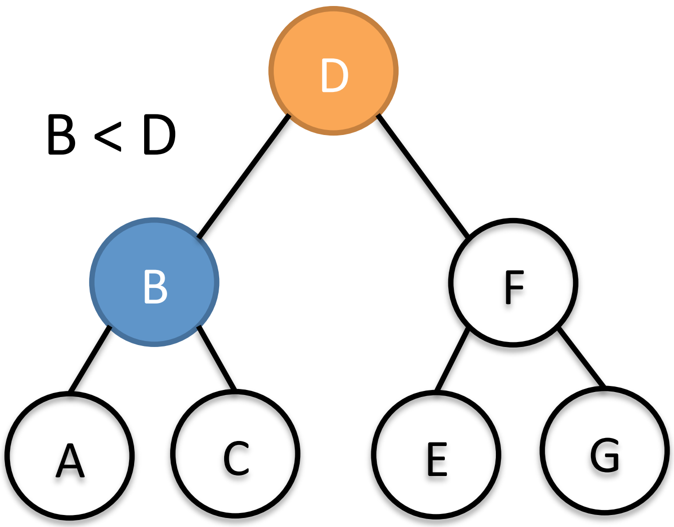
Review: Binary Search Trees (BSTS) are an ordered collection of key/value nodes



Binary Search Tree property
Let x be a node in a binary search tree s.t.:

- left.key < x.key
- right.key > x.key

Review: Binary Search Trees (BSTS) are an ordered collection of key/value nodes

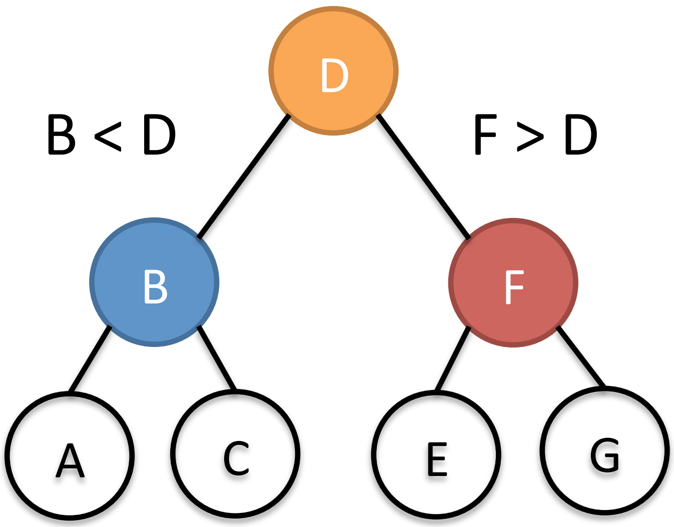


Binary Search Tree property

Let x be a node in a binary search tree s.t.:

- left.key < x.key
- right.key > x.key

Review: Binary Search Trees (BSTS) are an ordered collection of key/value nodes



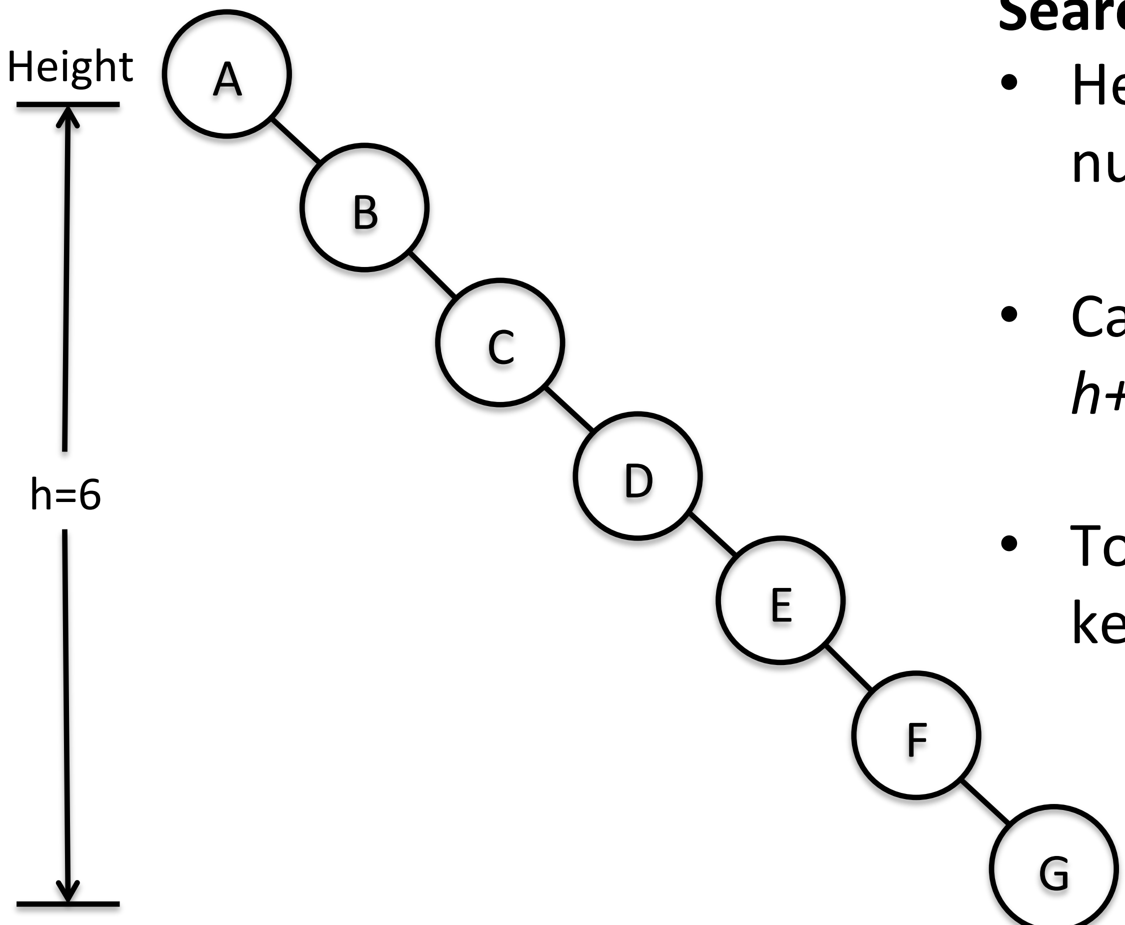
Binary Search Tree property

Let x be a node in a binary search tree s.t.:

- left.key < x.key
- right.key > x.key

BSTs do not have to be balanced! Can not make tight bound assumptions

Find Key "G"

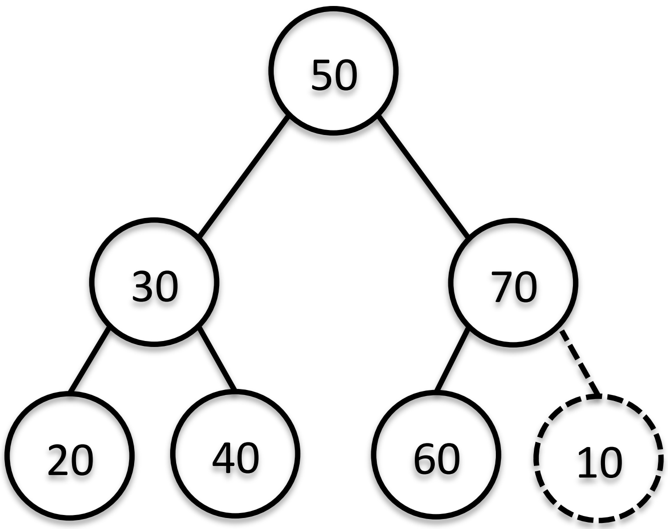


Search process

- Height *h = 6* (count number of edges to leaf)
- Can take no more than *h+1* checks, O(h)
- Today we will see how to keep trees balanced

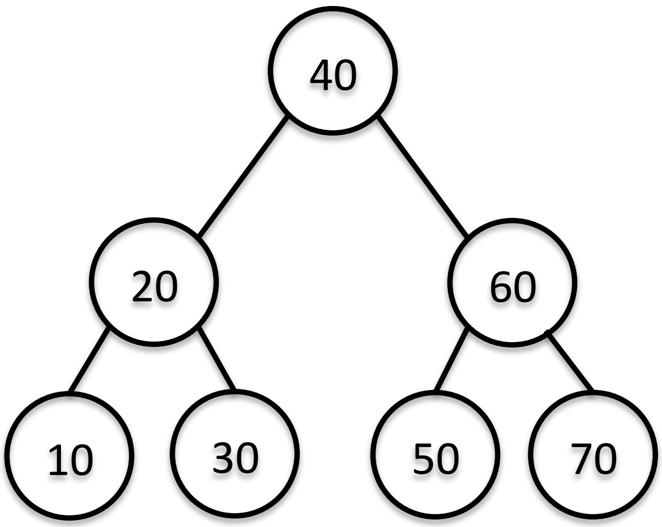
Could try to "fix up" tree to keep balance as nodes are added/removed

Keeping balance is tricky



Insert 10

"Fix up"



All nodes changed position
O(n) possible on many updates!
Need another way

We consider two other options to keep "binary" trees "balanced"

1. Give up on "binary" – allow nodes to have multiple keys (2-3-4 trees)

2. Give up on "perfect" – keep tree "close" to perfectly balanced (Red-Black trees)

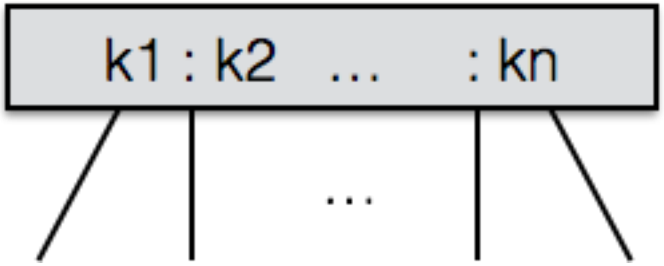
Agenda

1. Balanced Binary Trees
2. 2-3-4 Trees
3. Red-Black Trees
4. Deletion in 2-3-4 and Red-Black trees

2-3-4 trees (aka 2,4 trees) give up on binary but keep tree balanced

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
- We will work with nodes that have 2, 3, or 4 children (nodes are named after number of children, not keys)



2-3-4 trees maintain two properties: size and depth

Size property

Each node has either 2, 3, or 4 children (1, 2, or 3 keys per node)

Each node type named after number of children, not keys

Depth property

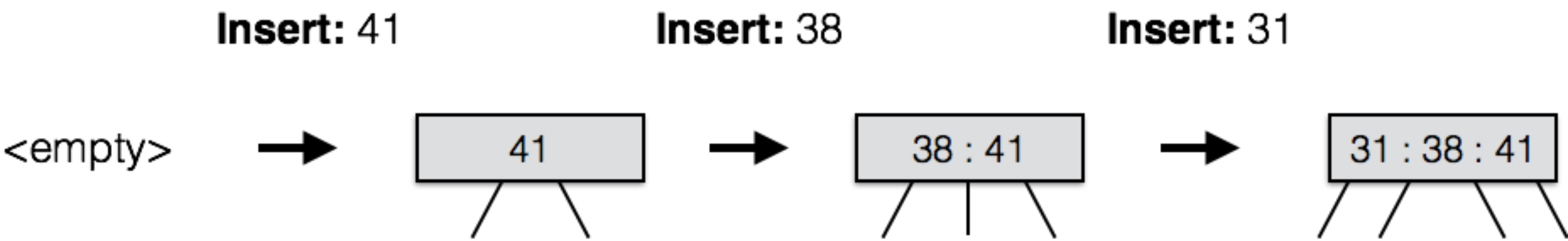
All leaves of the tree (either external nodes or null pointers) are on the same level

Node types

2 node 3 node 4 node

Insert into the lowest node, but do not violate the size property

Inserting into 2 or 3 node

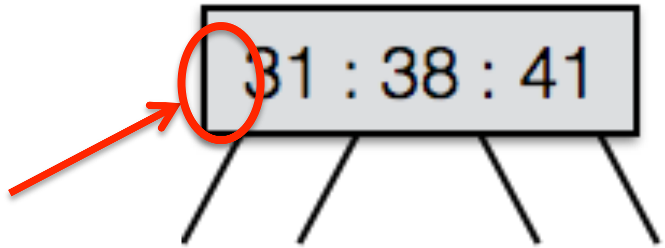


- Keep keys ordered inside each node
- Can insert key in *node* in O(1)
- Can keep node ordered in O(1) time because there at most 3 keys in each node

If insert would violate size rule, split 4 node into two 2 nodes, then insert new object

Inserting into 4 node

Insert: 12



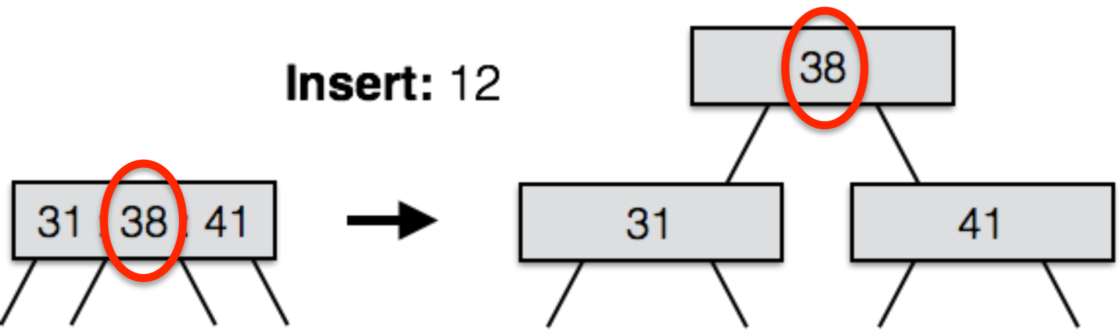
Would go here

Insert would cause size violation for this node

Insert in a two step process

If insert would violate size rule, split 4 node into two 2 nodes, then insert new object

Inserting into 4 node, two step process



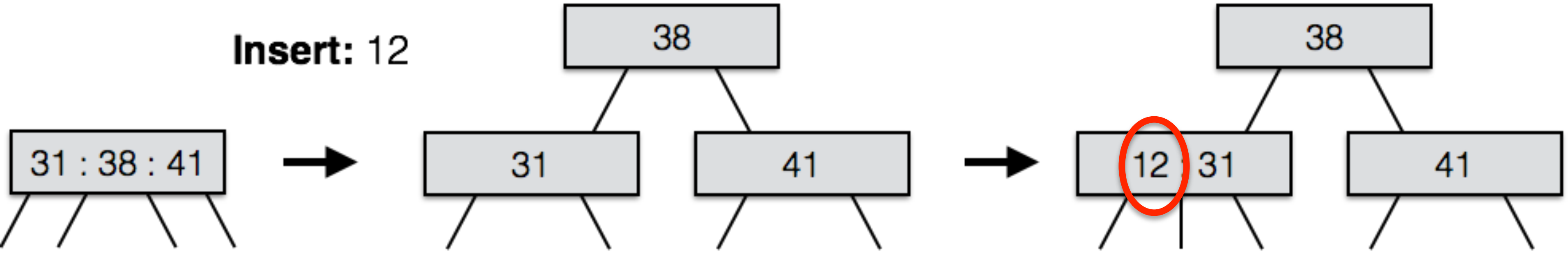
Step 1: split/promote

Promote middle key to higher level

- May become new root
- Parent may have to be split also!

If insert would violate size rule, split 4 node into two 2 nodes, then insert new object

Inserting into 4 node, two step process



Step 1: split/promote
Promote middle key to higher level
- May become new root
- Parent may have to be split also!

Step 2: insert
Insert 12 into appropriate node at lowest level

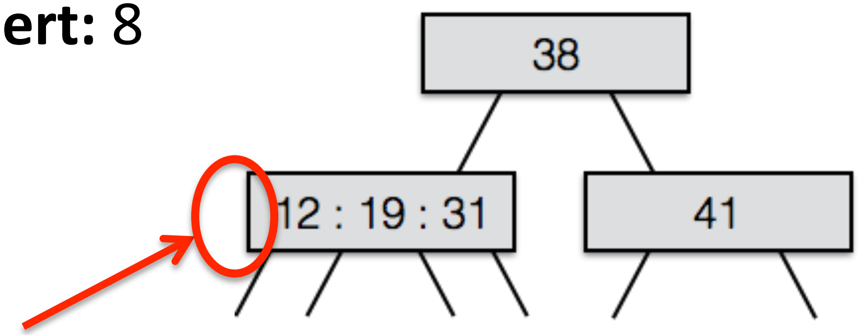
Continue inserting until need to split nodes

Insert process

Promote middle key to higher level and insert new key into proper position

Insert process

Insert: 8

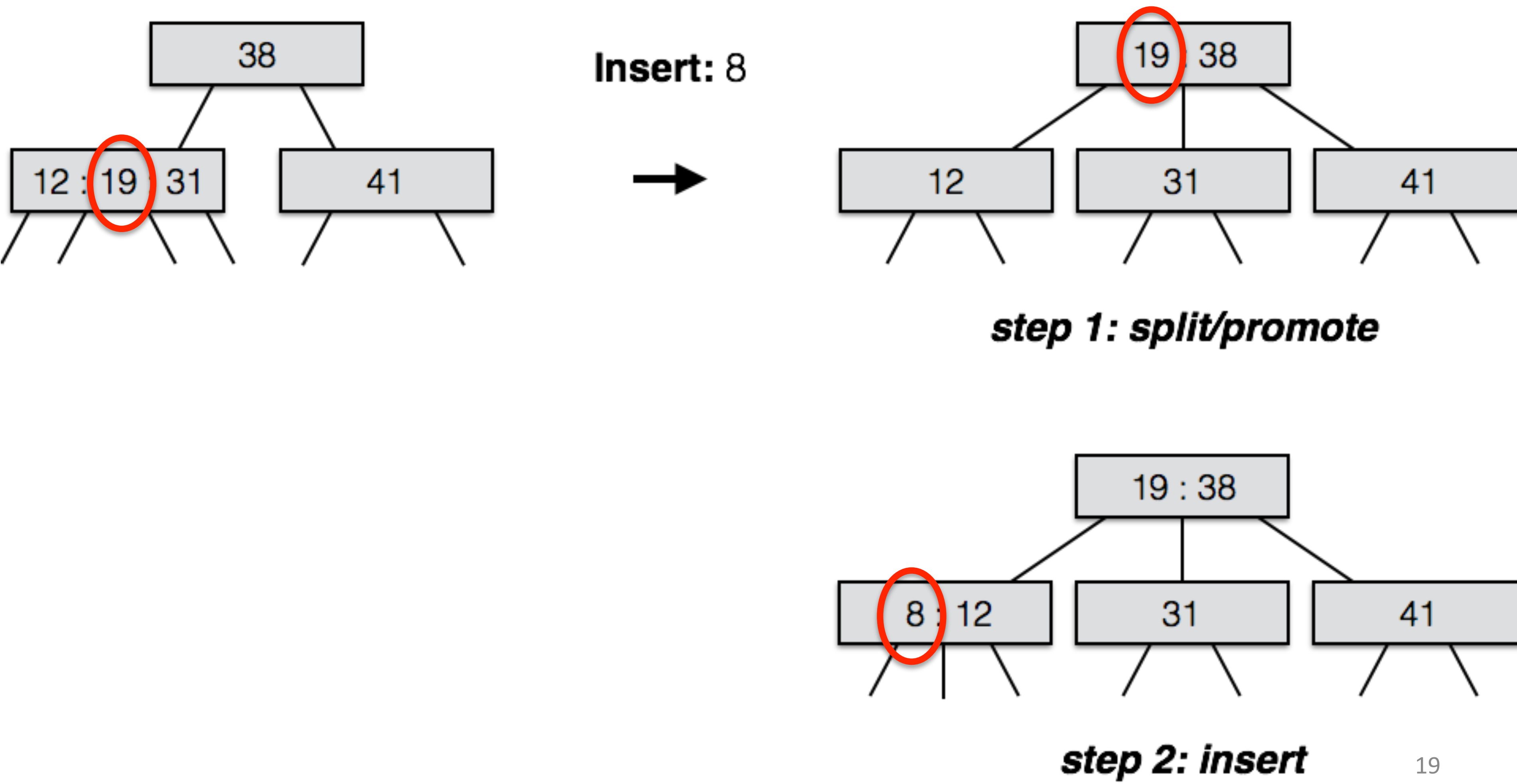


Would go here

Insert would cause size violation for this node

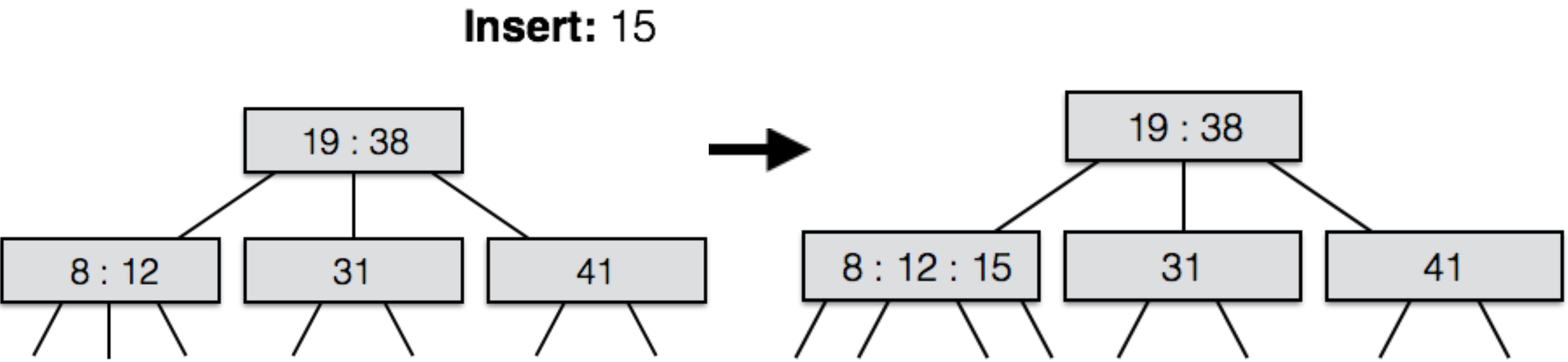
Promote middle key to higher level and insert new key into proper position

Insert process



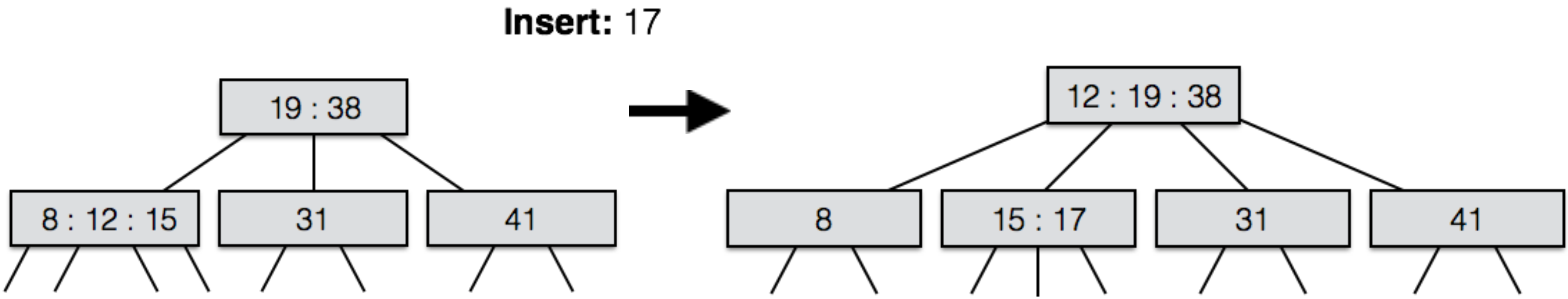
Insert new key in lowest level

Insert process



Insert new key in lowest level

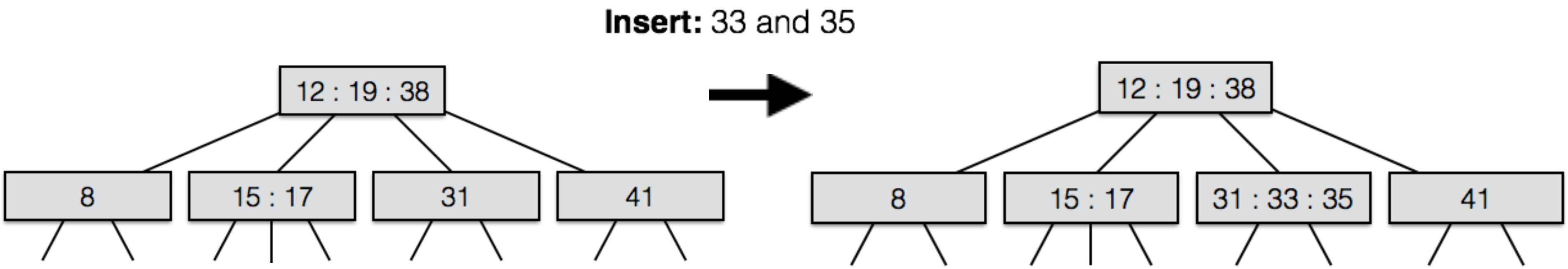
Insert process



Step 1: Split and promote 12
Step 2: Insert 17

Insert new key in lowest level

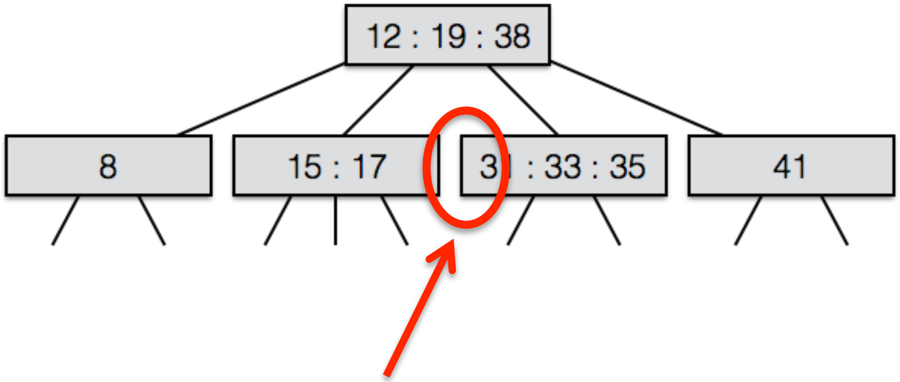
Insert process



Might have to split multiple nodes to ensure parent size property is not violated

Insert process

Insert: 20



Would go here

Insert would cause size violation for this node

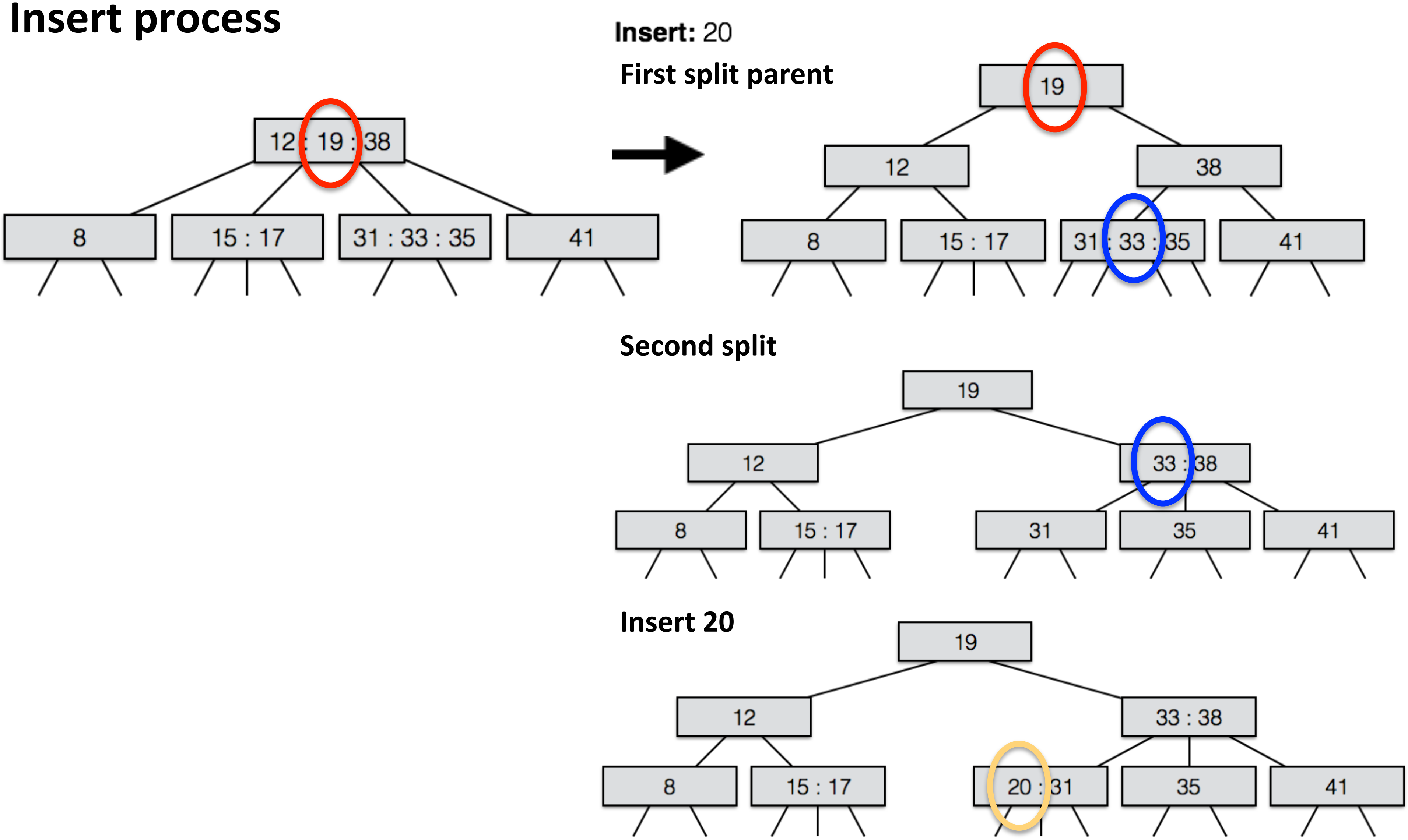
Promoting would cause parent size violation

Split parent first, then split child, then insert

Could bubble up all the way to the root

Might have to split multiple nodes to ensure parent size property is not violated

Insert process



2-3-4 work, but are tricky to implement

- Need three different types of nodes
- Create new ones as you need them
- Can waste space if nodes have few keys
- Copy information from old node to new node
- Book has more info on insertion and deletion
- There are generally easier ways to implement as a binary tree

Agenda

1. Balanced Binary Trees
2. 2-3-4 Trees
3. Red-Black Trees
4. Deletion in 2-3-4 and Red-Black trees

Red-Black trees are binary trees related to 2-3-4 trees

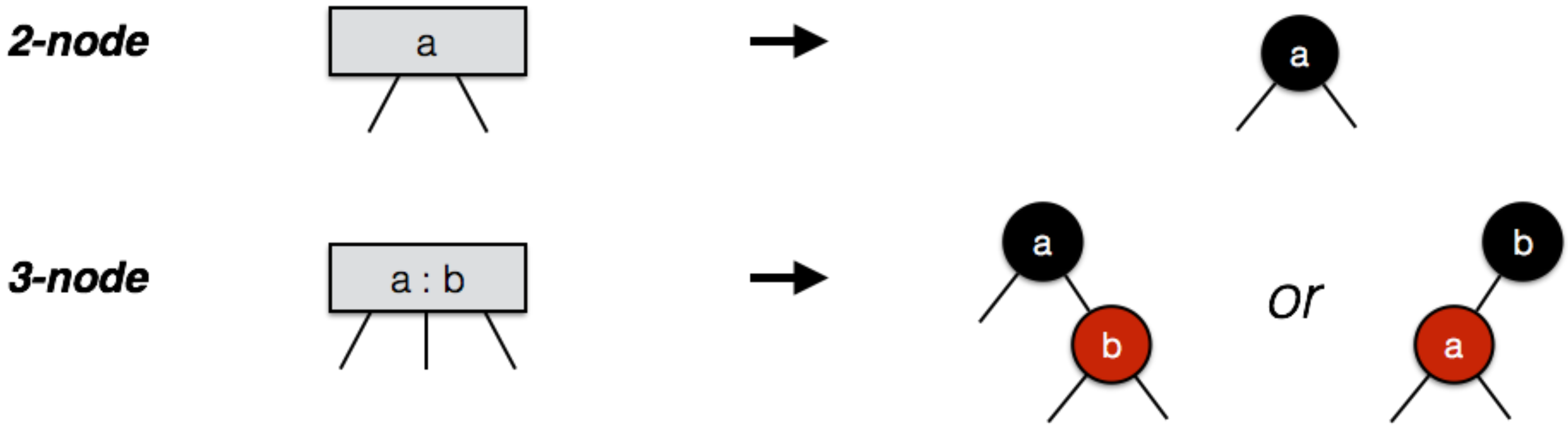
Overview

- Translate each 2, 3, or 4 node into 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
- Paint node red if would be part of a 2-3-4 node

Red-Black trees are binary trees related to 2-3-4 trees

Overview

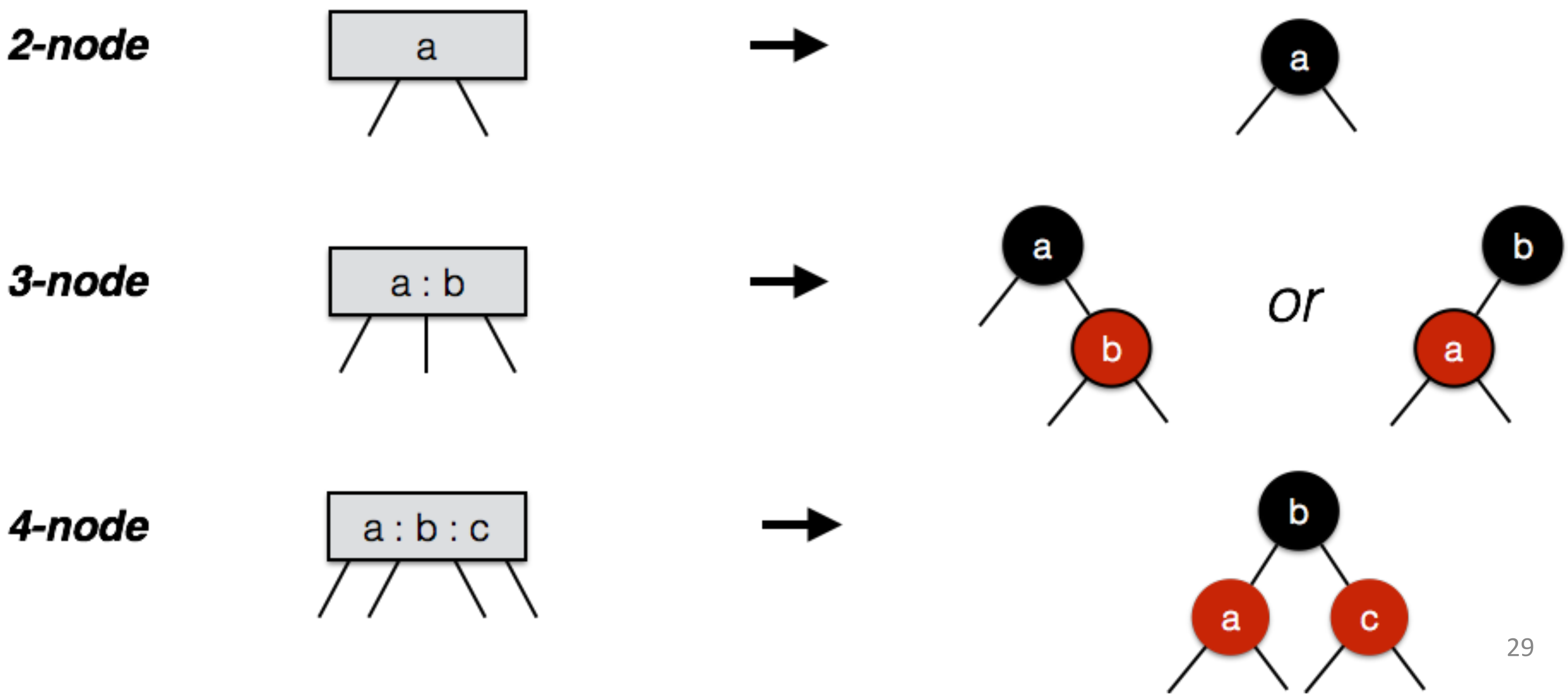
- Translate each 2, 3, or 4 node into miniature binary tree
- "Color" each vertex so that we can tell which nodes belong together as part of a larger 2-3-4 tree node
- Paint node red if would be part of a 2-3-4 node



Red-Black trees are binary trees related to 2-3-4 trees

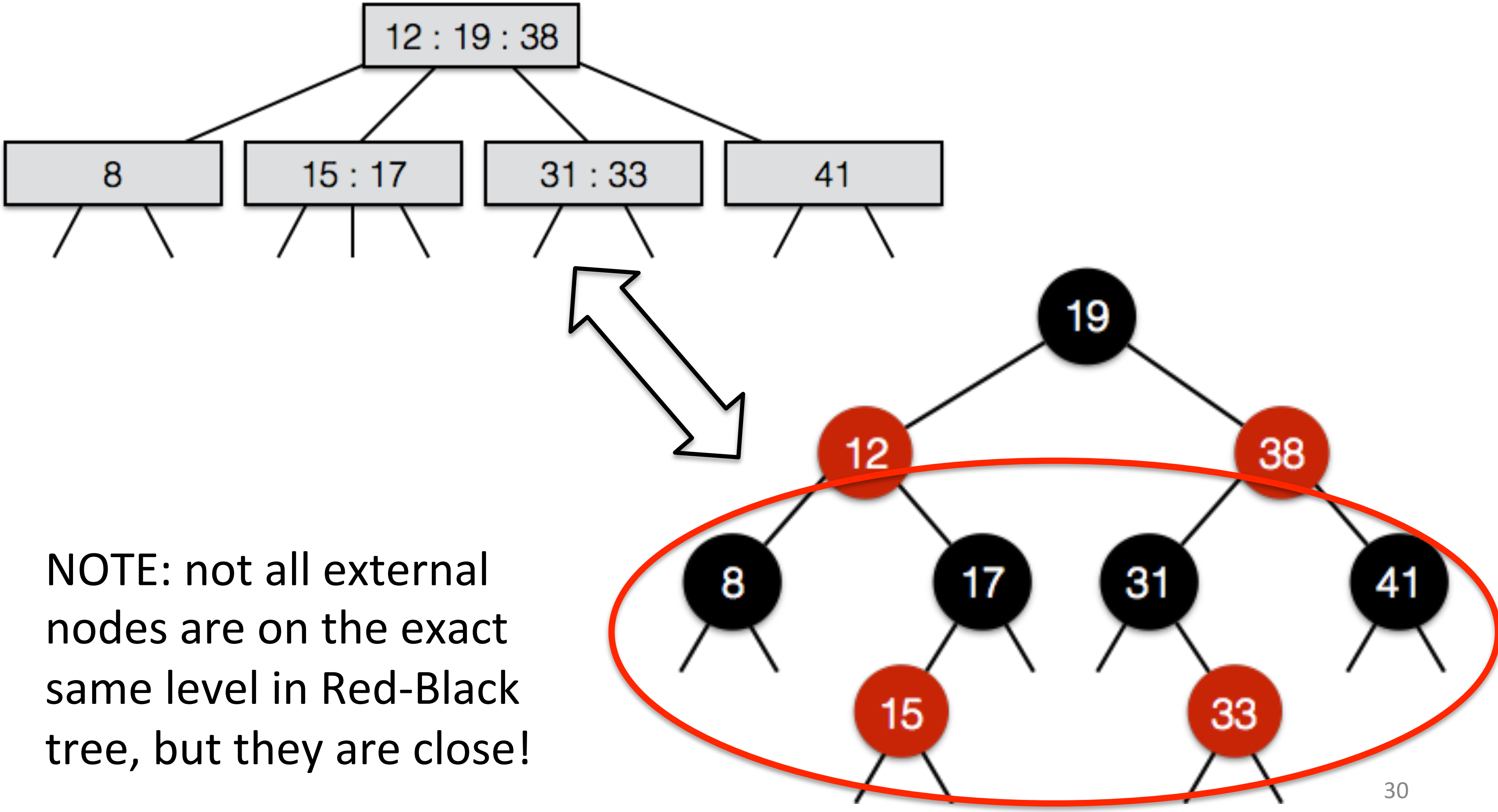
Overview

- Translate each 2, 3, or 4 node into 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
- Paint node red if would be part of a 2-3-4 node



You can convert between 2-3-4 trees and Red-Black trees and vice versa

Red-Black as related to 2-3-4 trees



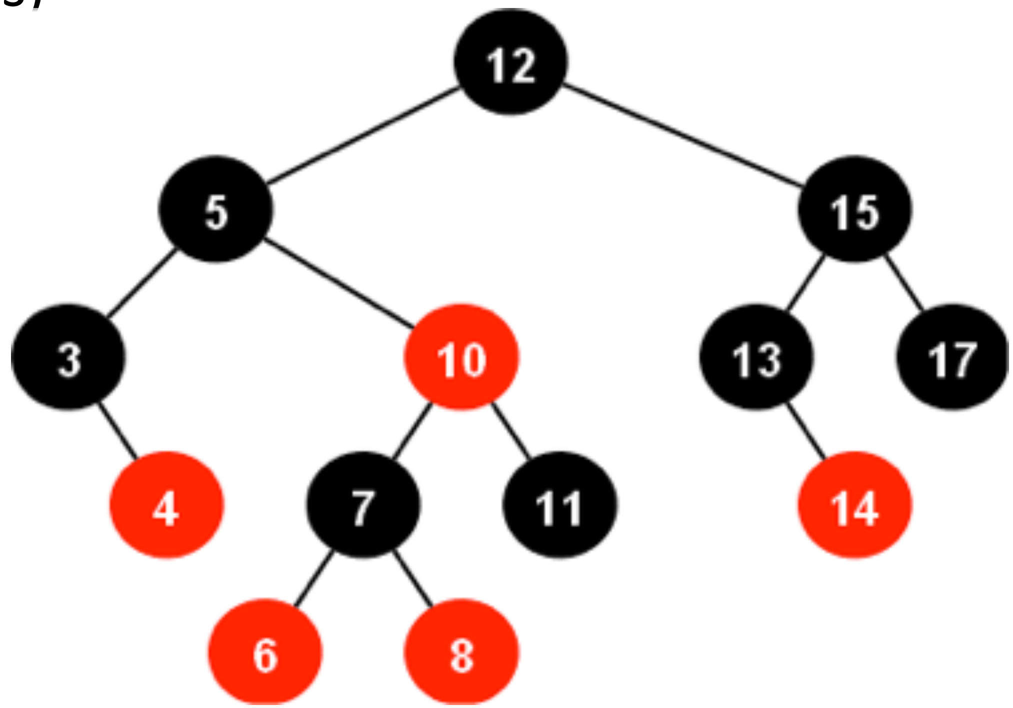
NOTE: not all external nodes are on the exact same level in Red-Black tree, but they are close!

You can also think of Red-Black trees as structure maintaining four properties

Red-Black trees properties

1. Every nodes is either red or black
2. Root is always black, if operation changes it red, turn it black again
3. Children of a red node are black (no consecutive red nodes)
4. All external nodes have the same black depth (same number of black ancestors)

Black depth: 3



Red-Black properties ensure depth of tree is O(log n), given n nodes in 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.)
- 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 gives: $n + 1 \geq 2^{(h/2)}$
  - Taking the log (base 2) of both sides gives:
    - $\log(n+1) \geq h/2 = 2 \log(n + 1) \geq h$, which is O(log `n`)

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

Searching a Red-Black is O(log n)

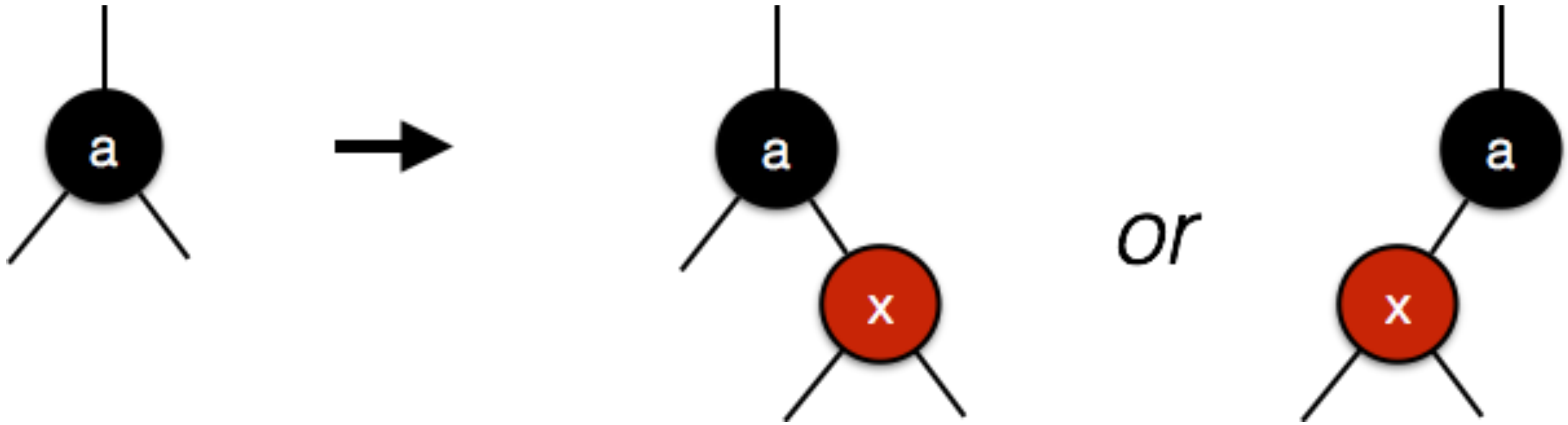
- Red-Black tree *is* a Binary Search Tree with time proportional to height
- Search time then takes O(log n)
- Hard part is maintaining the tree with inserts and deletes

Insertion into Red-Black trees must deal with several cases

- As with BSTs, find location in tree where new element goes and insert there
- Color new node red. This ensures rules 1, 2 and 4 are preserved
- Rule 3 might be violated (red node must have black children)
- Three different cases can arise on insert
- Inserting into a 2 or 4 node fairly straightforward; 3 node is more complex

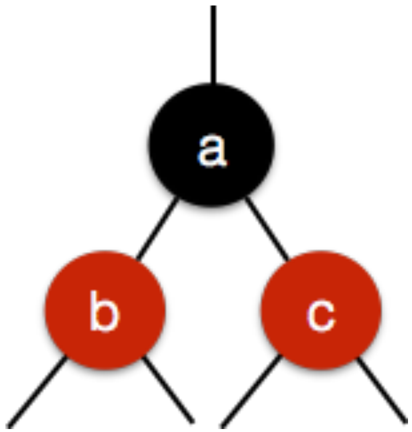
Case 1: Insert into 2 node, no violation

Insert into 2 node causes no violation



Case 2: Insert into 4 node is a violation, resolve with "color flip"

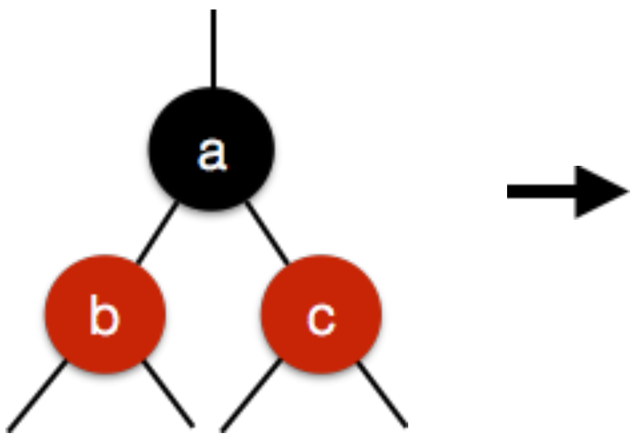
4 nodes are black with red children

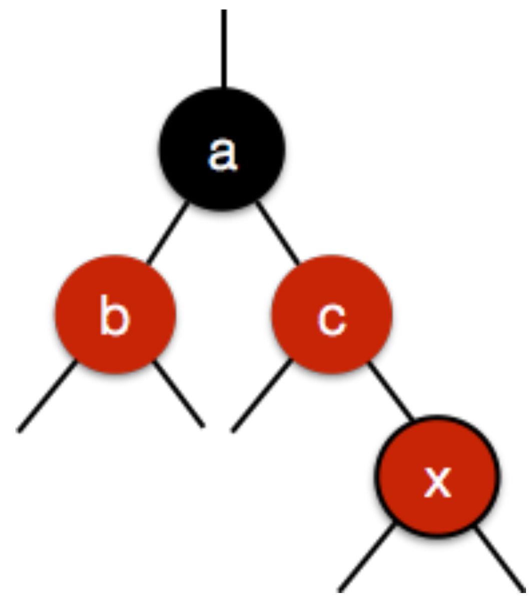


Insert new node as child of <b> or <c>, now have two red nodes in a row

Case 2: Insert into 4 node is a violation, resolve with "color flip"

4 nodes are black with red children





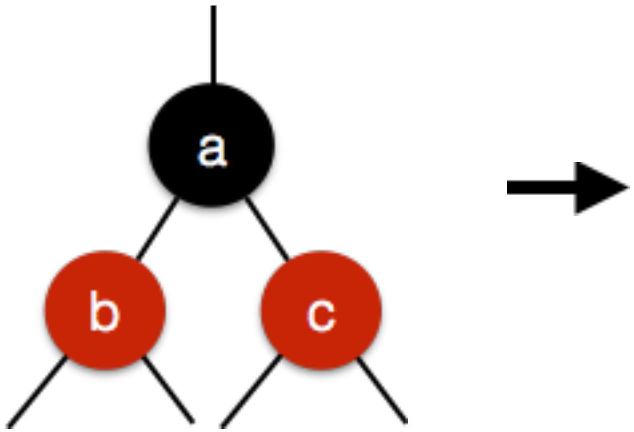
Insert new node as child of <b> or <c>, now have two red nodes in a row

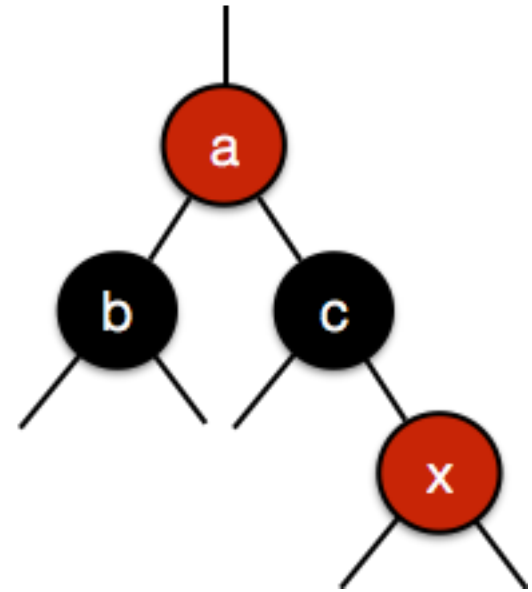
Must split node, promoting middle key

- Could promote <a> to parent, and unjoin <b> and <c> from <a>
- Amounts to a "color flip"

Case 2: Insert into 4 node is a violation, resolve with “color flip”

4 nodes are black with red children





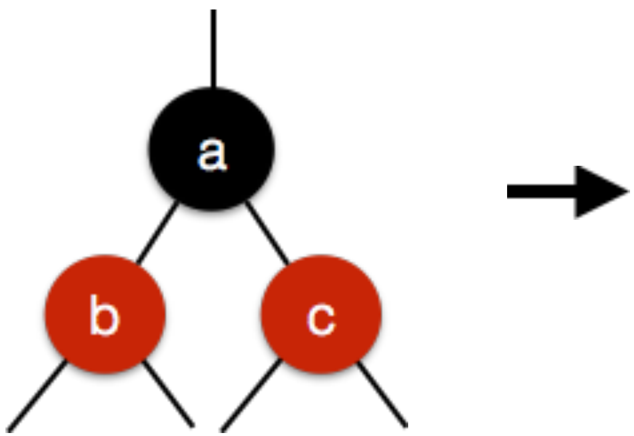
Insert new node as child of <b> or <c>, now have two red nodes in a row

Must split node, promoting middle key

- Could promote <a> to parent, and unjoin <b> and <c> from <a>
- Amounts to a “color flip”

Case 2: Insert into 4 node is a violation, resolve with “color flip”

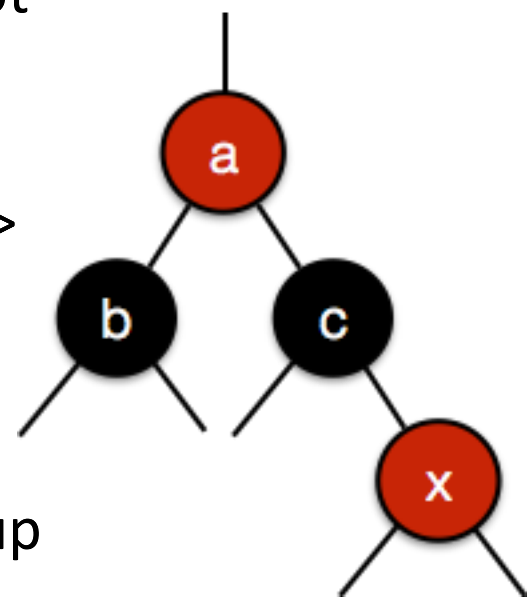
4 nodes are black with red children



Black length not changed

Must check <a> doesn’t violate parent

Might bubble up to root



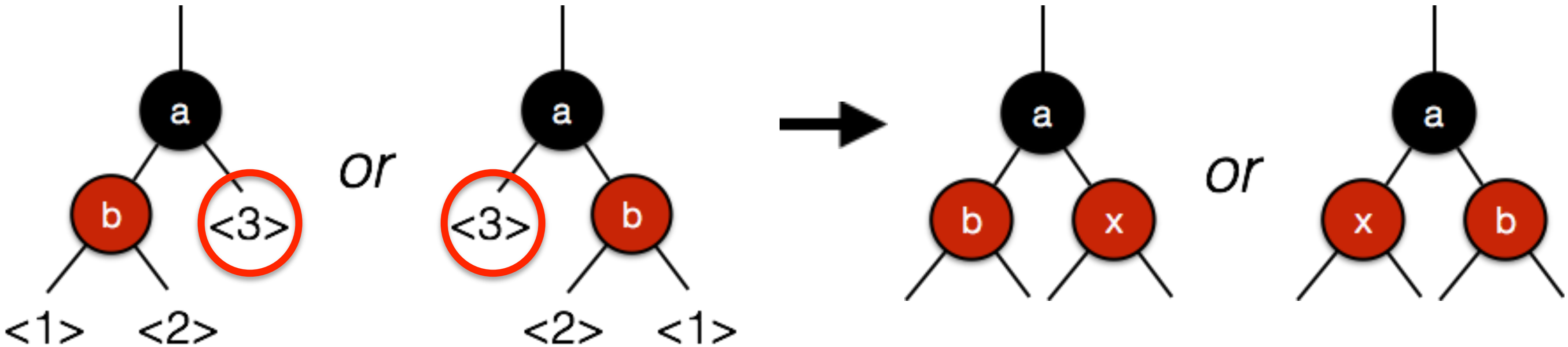
Insert new node as child of <b> or <c>, now have two red nodes in a row

Must split node, promoting middle key

- Could promote <a> to parent, and unjoin <b> and <c> from <a>
- Amounts to a “color flip”

Case 3: Insert into 3 node, might be violation

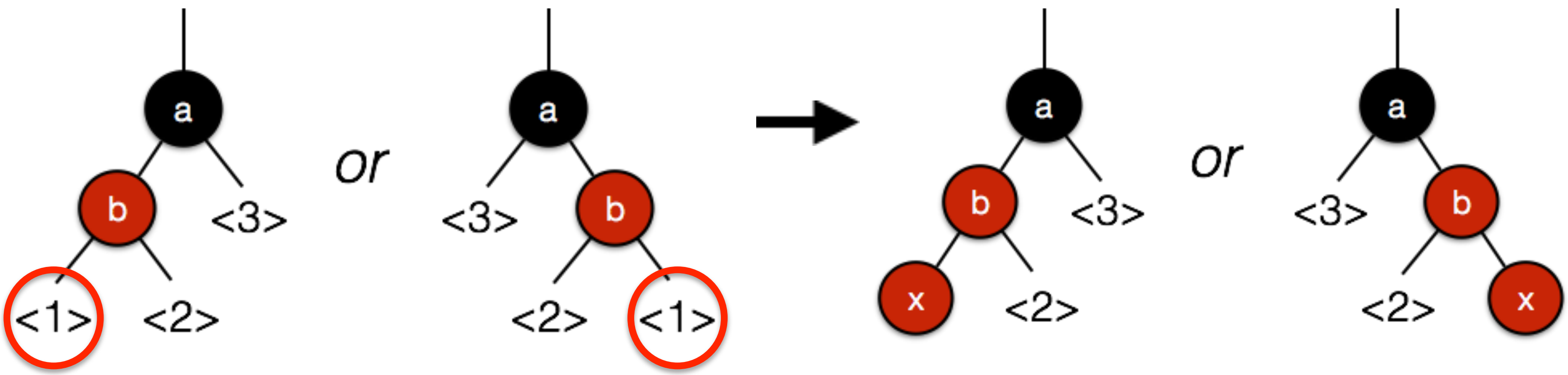
3 nodes are black with red children, insert at <3>, just insert



- No problem if inserting at position <3>
- Makes a 4 node

Case 3: Insert into 3 node, might be violation

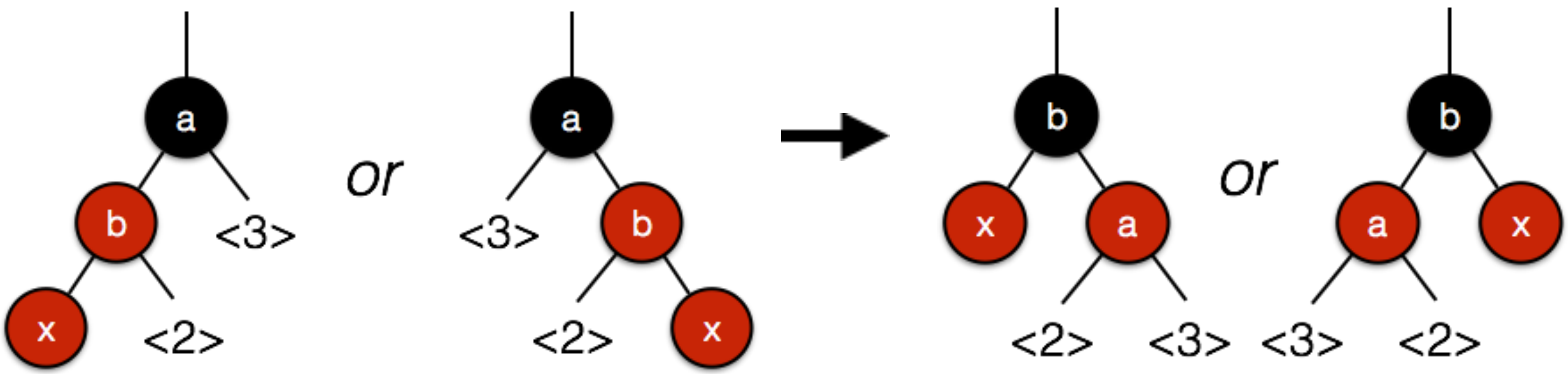
Black 3 node with red children, inserting at <1>, do single rotation



- Violation of no two red nodes in a straight line
- Since x < b < a or x > b > a, could fix by rotating whole structure
- Lift <b> to root (color black), while dropping down <a> (color red) to be child of <b>

Case 3: Insert into 3 node, might be violation

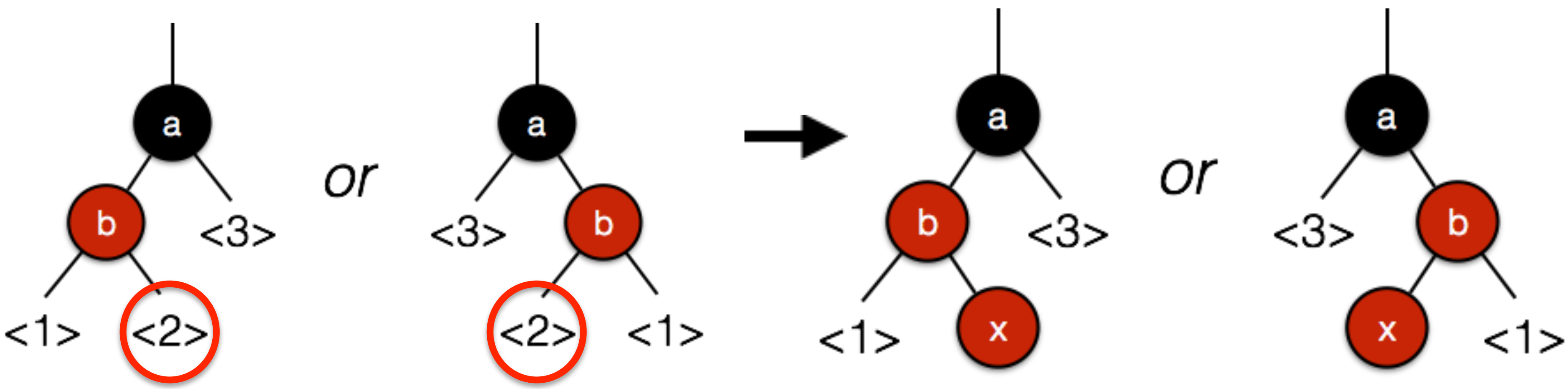
Black 3 node with red children, inserting at <1>, do single rotation



- Violation of no two red nodes in a straight line
- Since x < b < a or x > b > a, could fix by rotating whole structure
- Lift <b> to root (color black), while dropping down <a> (color red) to be child of <b>
- Still maintains ordered property
- Called a ***single rotation***

Case 3: Insert into 3 node, might be violation

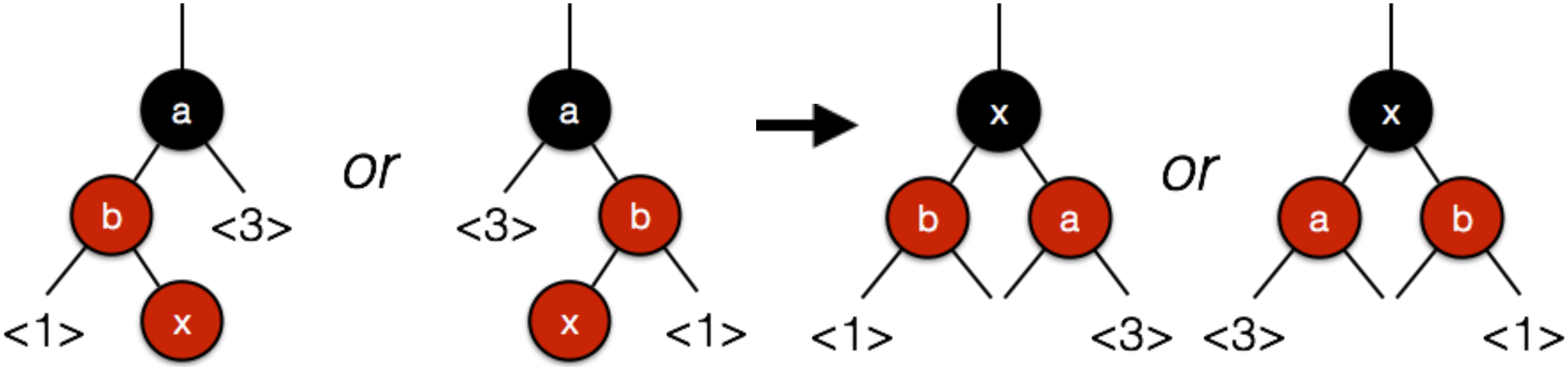
Black 3 node with red children, inserting at <2>, do double rotation



- Two red nodes in zig-zag pattern
- Lift <x> to root (color black) and have <a> and <b> as children (colored red)
- Called a ***double rotation***

Case 3: Insert into 3 node, might be violation

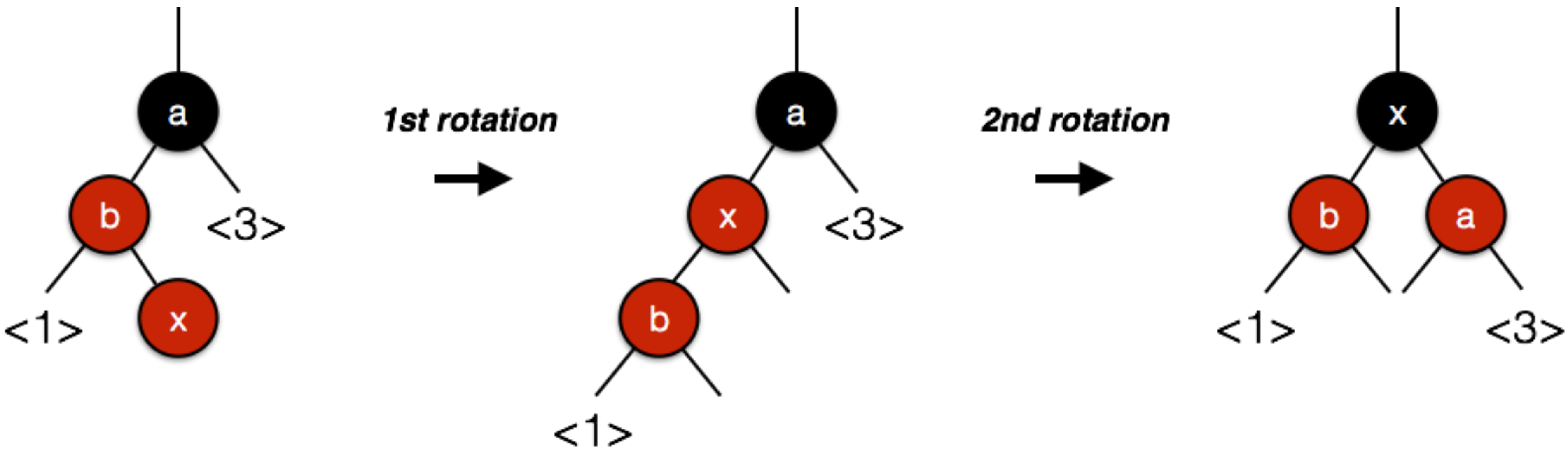
Black 3 node with red children, inserting at <2>, do double rotation



- Two red nodes in zig-zag pattern
- Lift <x> to root (color black) and have <a> and <b> as children (colored red)
- Called a ***double rotation***

Case 3: Insert into 3 node, might be violation

Black 3 node with red children, inserting at <2>, do double rotation



- Two red nodes in zig-zag pattern
- Lift <x> to root (color black) and have <a> and <b> as children (colored red)
- Called a ***double rotation***
- Rotate once around <b>, then again around <x>

Insert run time is O(log n)

- Worse case we only have to fix colors along the path between new node and root, O(log n) path length

- Each operation is a constant factor of work
  - It can be shown we only need to do at most one single-rotation or one double-rotation to fix the tree, O(1)
  - All other changes done with color flips, O(1)

- Leads to O(log n) insert run time complexity

Agenda

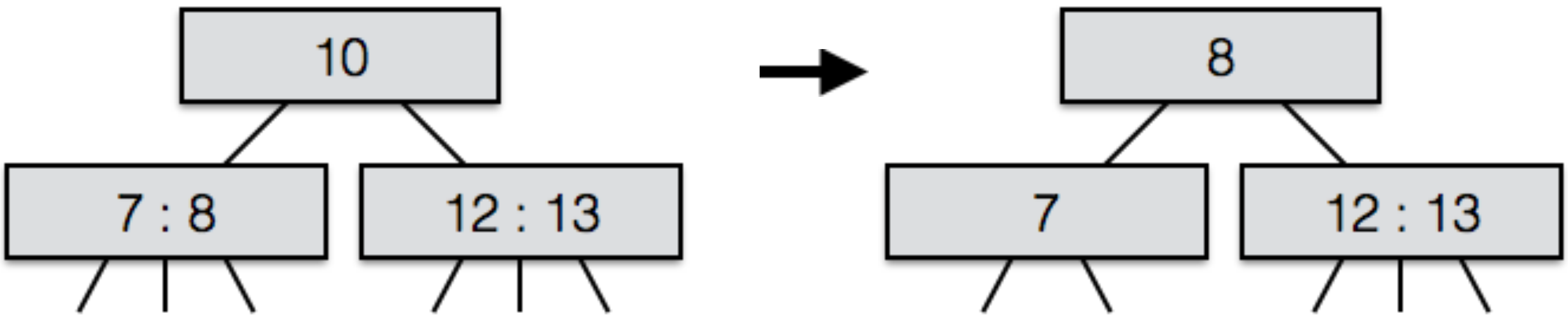
1. Balanced Binary Trees
2. 2-3-4 Trees
3. Red-Black Trees
4. Deletion in 2-3-4 and Red-Black trees

Deletion is O(log n)

- Key idea: make it so we simply have to delete a node at the bottom of the tree
- If node is internal, find a predecessor or successor at the bottom of the tree and use its key as a replacement for the one we want to delete (like BSTs)
- Then have to delete the predecessor or successor at the bottom of the tree

Case 1: Delete in 3 or 4 node, easy in 2-3-4 tree

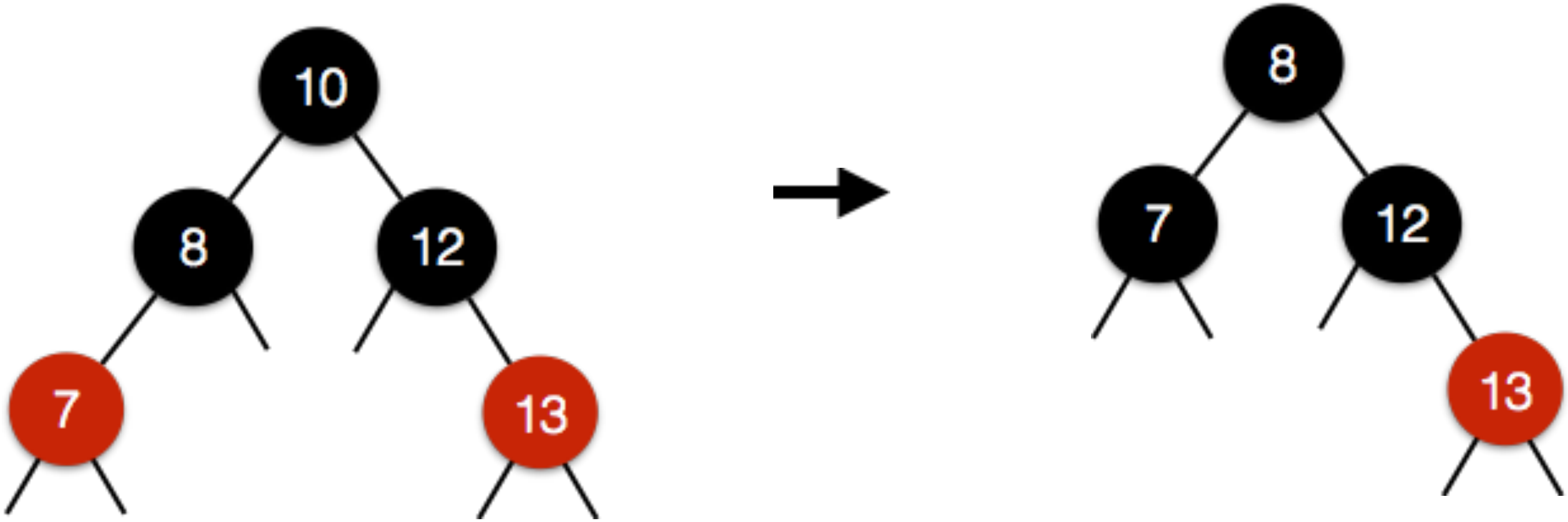
Delete 10



- Find immediate predecessor (or successor), 8
- Copy 8 key and value into 10
- Now delete 8 from 3 node it currently belongs to

Case 1: Delete in 3 or 4 node in Red-Black tree

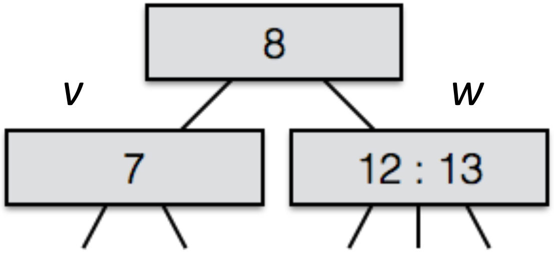
Delete 10



- Find immediate predecessor, 8 (or successor if no predecessor)
- Replace 10 with 8
- Delete 8
- Color children black (does not change black length)

Case 2: Delete 2 node in 2-3-4 tree

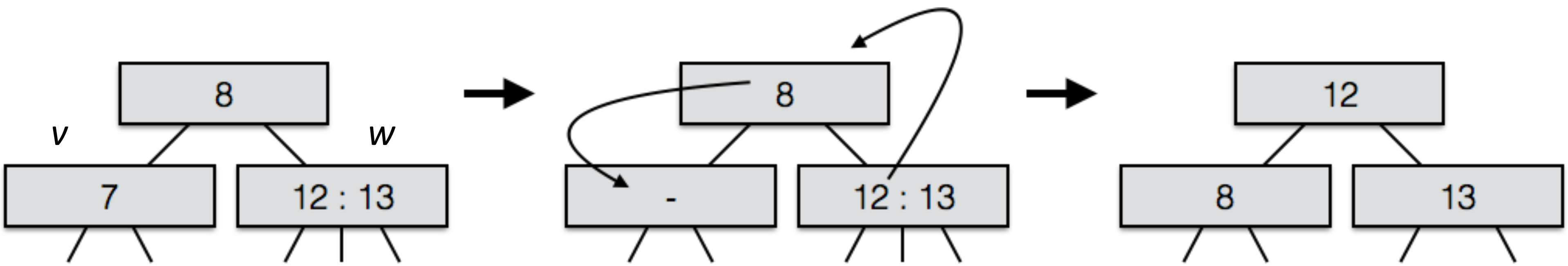
Delete 7



- If *w* is an adjacent sibling node of *v* to be deleted
- Move key up from *w* to parent and key from parent down to *v*

Case 2: Delete 2 node in 2-3-4 tree

Delete 7



- If *w* is an adjacent sibling node of *v* to be deleted
- Move key up from *w* to parent and key from parent down to *v*

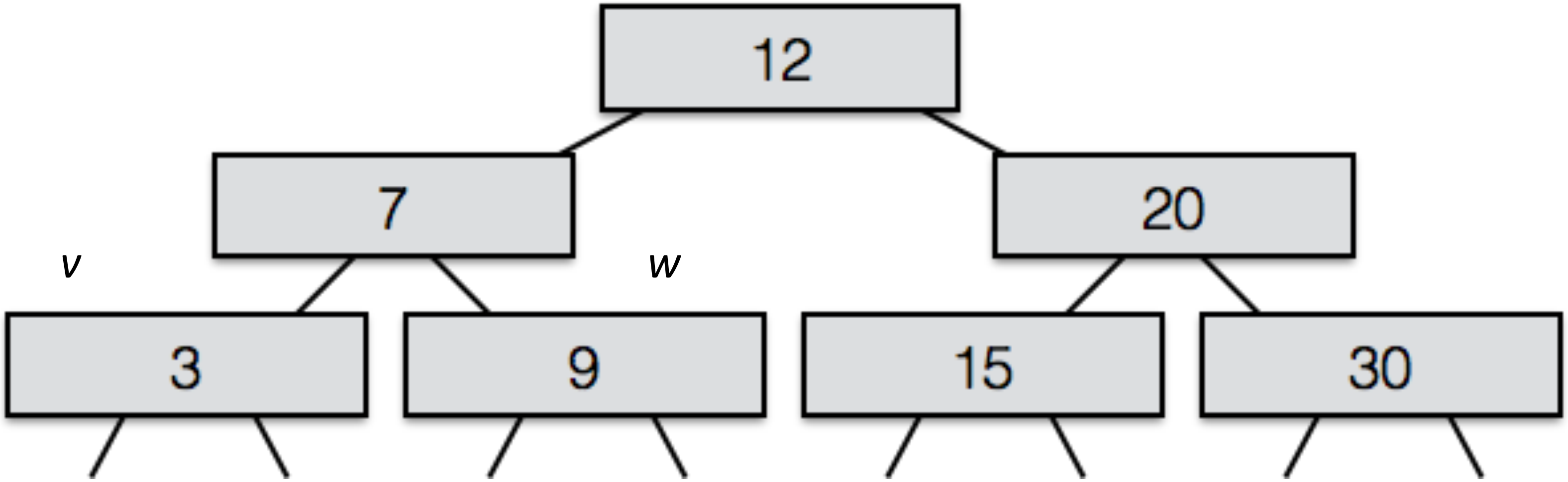
Case 2: Delete 2 node in Red-Black tree

Delete 7

- Deleting 7 and stopping would violate black depth property
- Trinode reconstruction
- Book has more details

Case 2: Delete 2 node in 2-3-4 tree

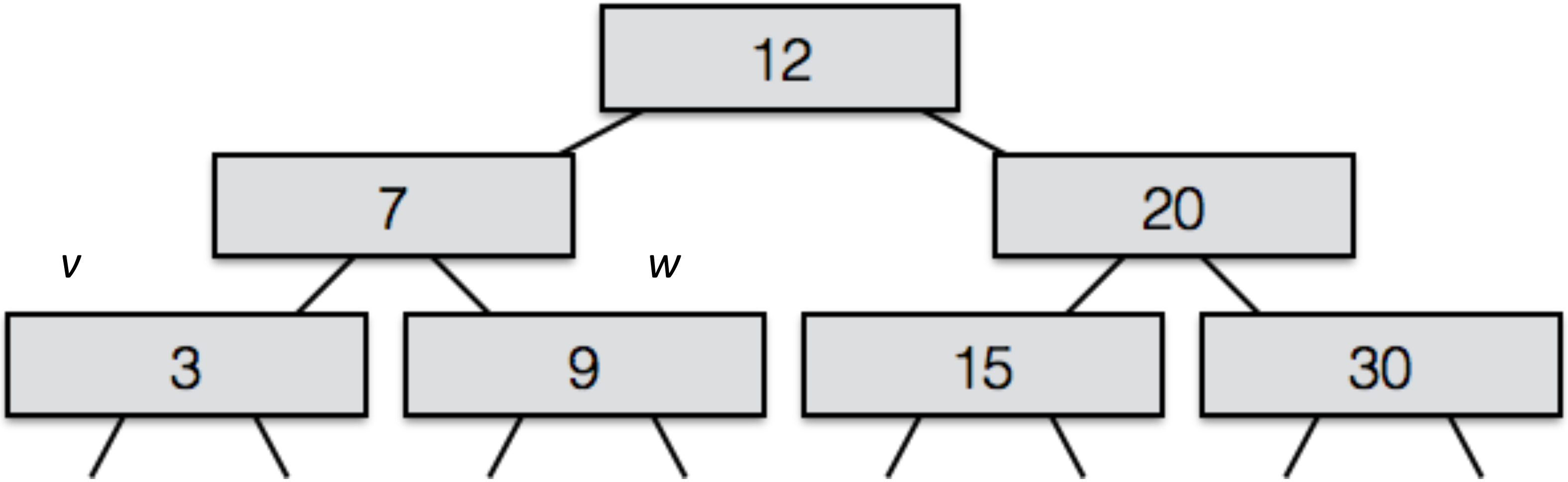
Delete 3



- If *w* is an adjacent sibling node of *v* to be deleted and *w* is 2 node
- Pull key down from parent and fuse with *w*

Case 2: Delete 2 node in 2-3-4 tree

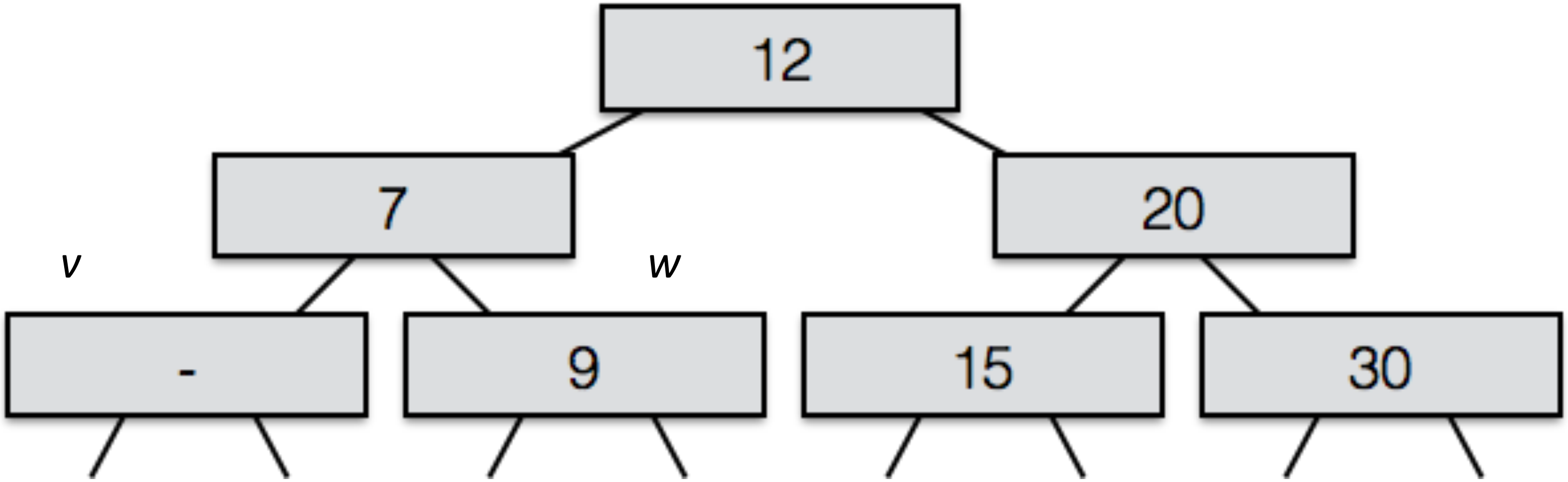
Delete 3



- If *w* is an adjacent sibling node of *v* to be deleted and *w* is 2 node
- Delete 3

Case 2: Delete 2 nodes in 2-3-4 trees

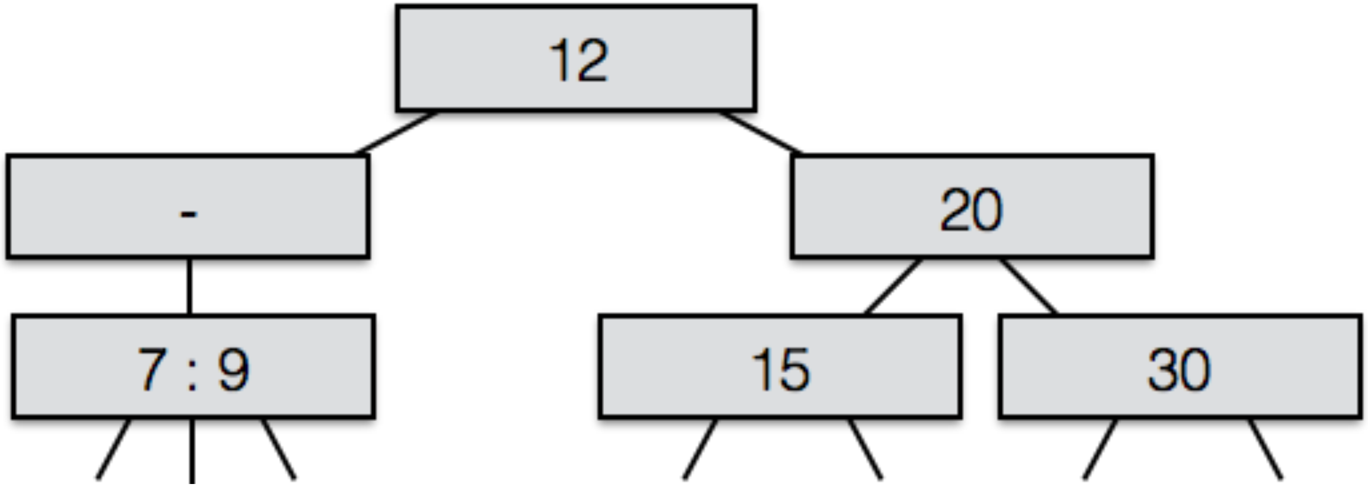
Delete 3



- If *w* is an adjacent sibling node of *v* to be deleted and *w* is 2 node
- Delete 3
- Pull key down from parent and fuse with *w*

Case 2: Delete 2 node in 2-3-4 tree

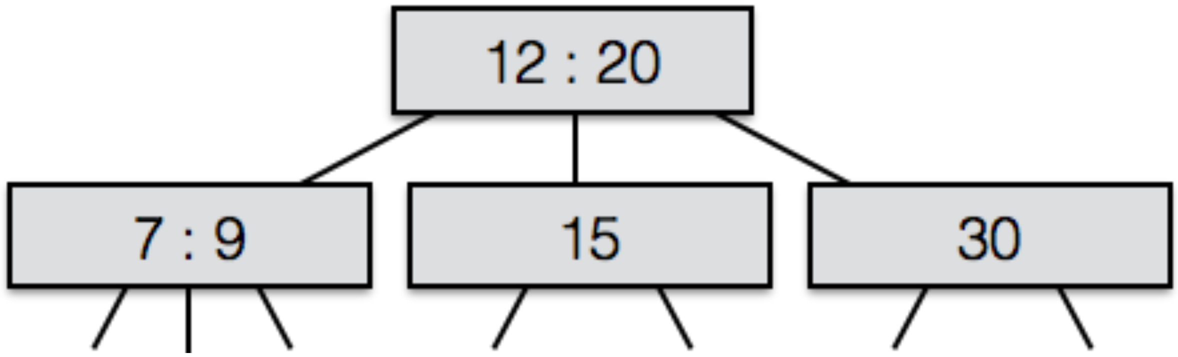
Delete 3



- If *w* is an adjacent sibling node of *v* to be deleted and *w* is 2 node
- Delete 3
- Pull key down from parent and fuse with *w*
- Keep depth property, so fuse again if needed

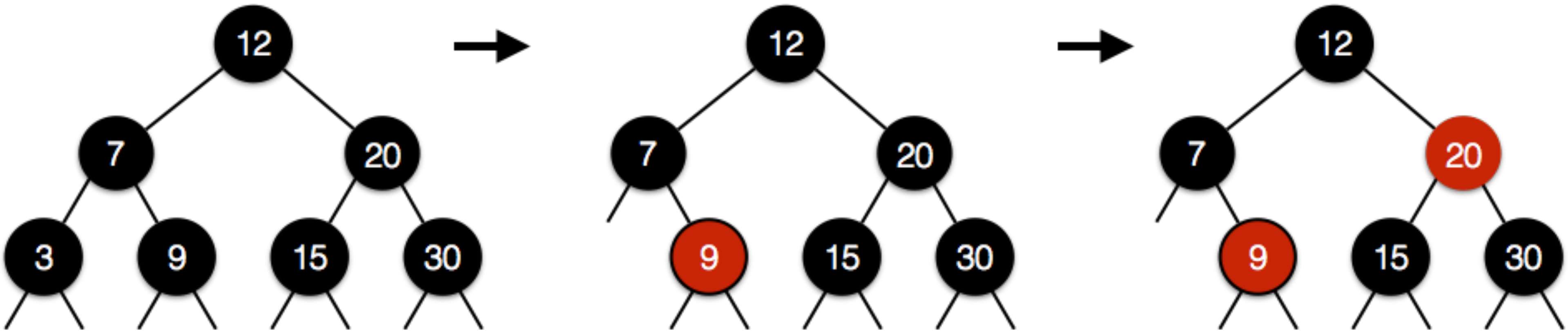
Case 2: Delete 2 node in 2-3-4 tree

Delete 3



- If *w* is an adjacent sibling node of *v* to be deleted and *w* is 2 node
- Delete 3
- Pull key down from parent and fuse with *w*
- Keep depth property, so fuse again if needed
- Tree may loose level if root is fused

In Red-Black trees, deletion causes recoloring to be passed up to parent



Summary

- Binary Search Trees performance suffers if they are unbalanced
- Two options to keep O(log n) find, insert, and delete performance:
  1. 2-3-4 trees – give up on binary
     - All leaves are at the same level, all paths the same length
     - Memory inefficient if nodes have small number of keys
     - Difficult to implement due to different node types
  2. Red-Black trees – give up on binary
     - Encode 2-3-4 nodes as “mini trees”
     - Nodes colored to indicate they are conjoined with their parent
     - Use rotations and color flips to keep tree in approximate balance
     - Find, insert and delete take no more than O(log n)
     - All Map operations O(log n) using Red-Black tree (Java uses for Red-Black for TreeMap)