

CS 10:


Problem solving via Object Oriented Programming

Balance

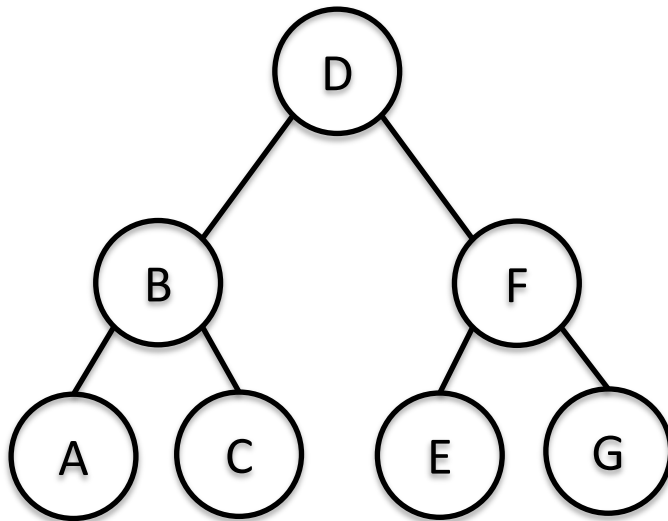
Main goals

- Conceptually implement trees and insertion/deletion operations for balanced BST

Agenda

- 
1. Balanced Binary Trees
 2. 2-3-4 Trees
 3. Red-Black Trees

Review: Binary Search Trees (BSTs) are an ordered collection of Key/Value nodes

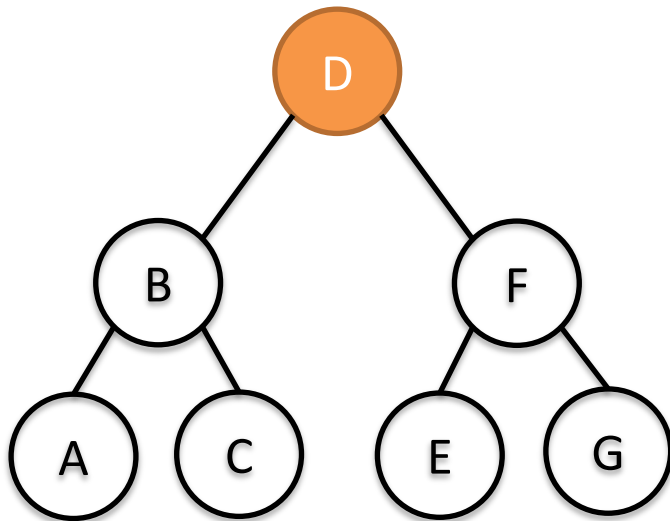


Binary Search Tree property

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

- $\text{left.key} < x.\text{key}$
- $\text{right.key} > x.\text{key}$

Review: Binary Search Trees (BSTs) are an ordered collection of Key/Value nodes

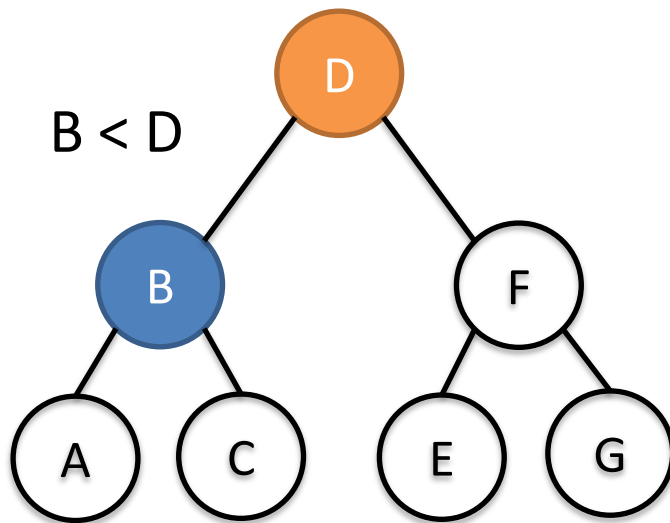


Binary Search Tree property

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

- $\text{left.key} < x.\text{key}$
- $\text{right.key} > x.\text{key}$

Review: Binary Search Trees (BSTs) are an ordered collection of Key/Value nodes

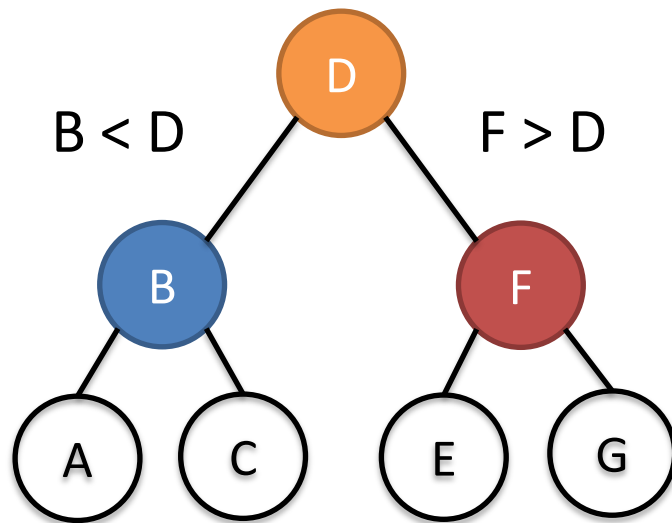


Binary Search Tree property

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

- $\text{left.key} < x.\text{key}$
- $\text{right.key} > x.\text{key}$

Review: Binary Search Trees (BSTs) are an ordered collection of Key/Value nodes



Binary Search Tree property

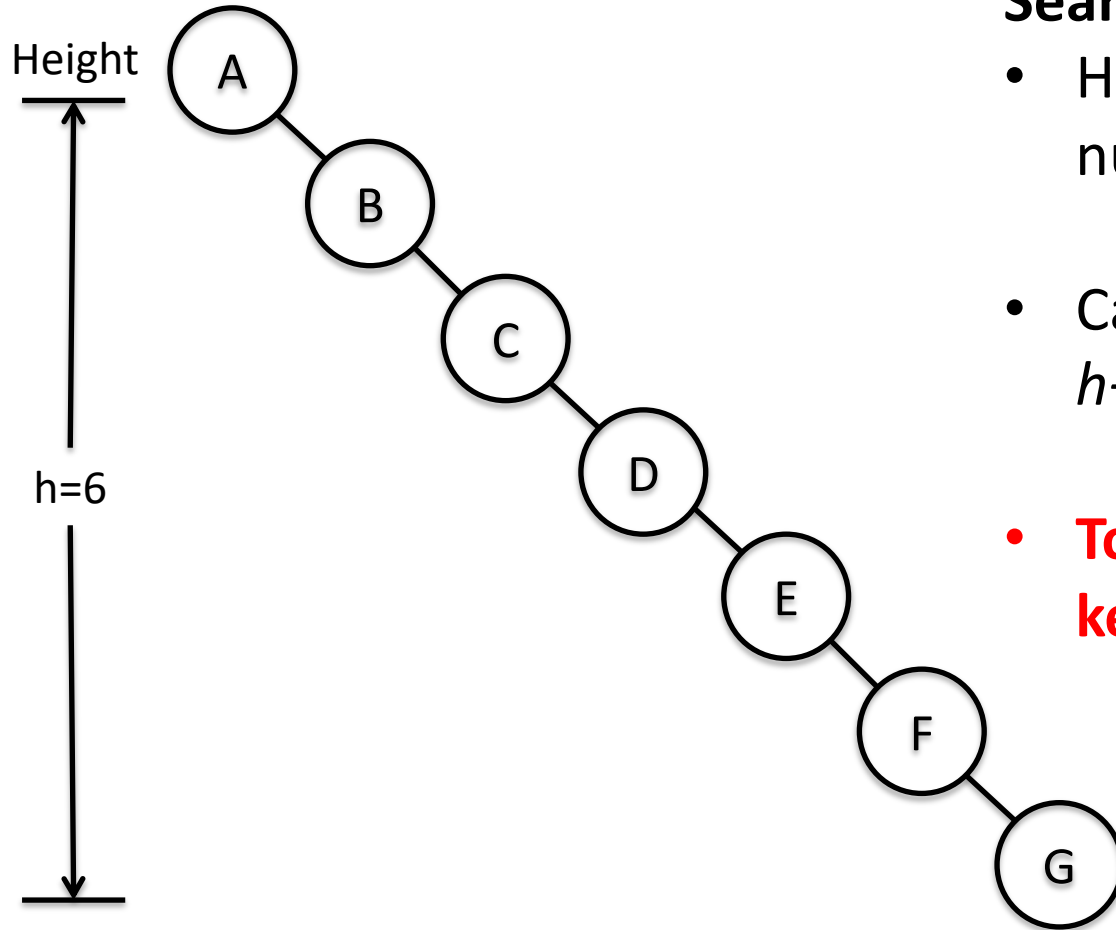
Let x be a node in a binary search tree s.t.:

- $\text{left.key} < x.\text{key}$
- $\text{right.key} > x.\text{key}$

Remember, I'm showing the Keys for each node, but there is also a Value for each node that is not shown

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

Find Key "G"

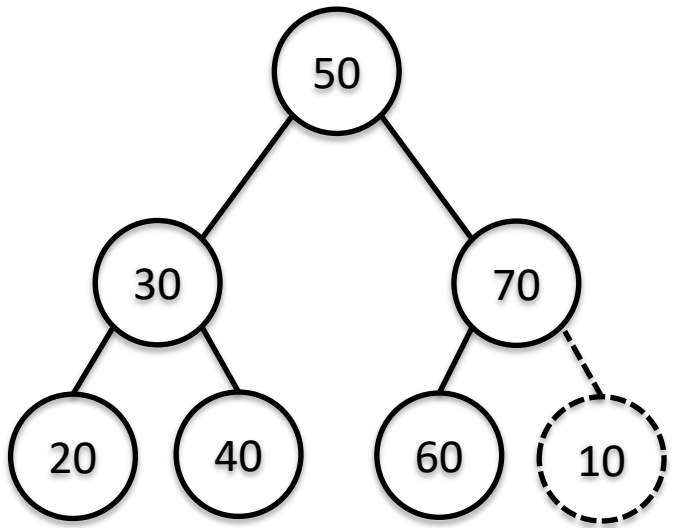


Search process

- 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"**

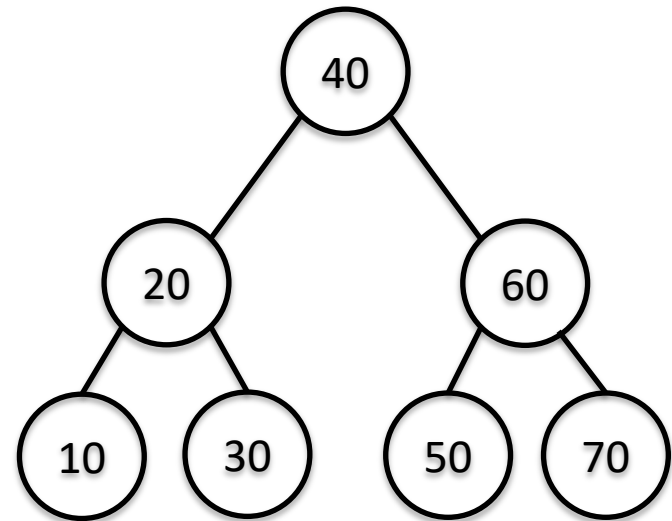
Could try to “fix up” tree to keep balance as nodes are added/removed

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 “perfectly 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

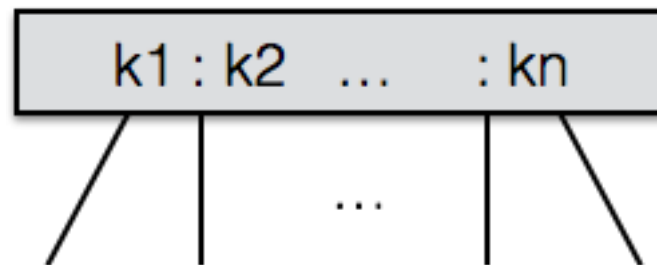
 2. 2-3-4 Trees

3. Red-Black Trees

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 the number of 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)

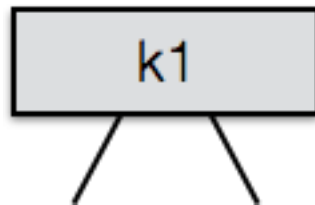
Each node type named after number of children, not keys

Depth property

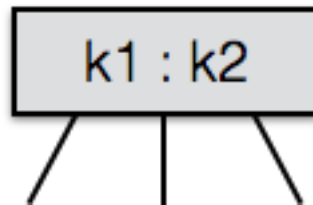
All leaves of the tree (external nodes) are on the same level

It can be shown that the height of the Tree is $O(\log n)$ if these properties are maintained (see book)

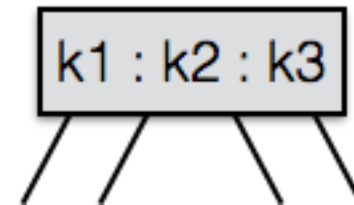
Node types



2 node



3 node



4 node

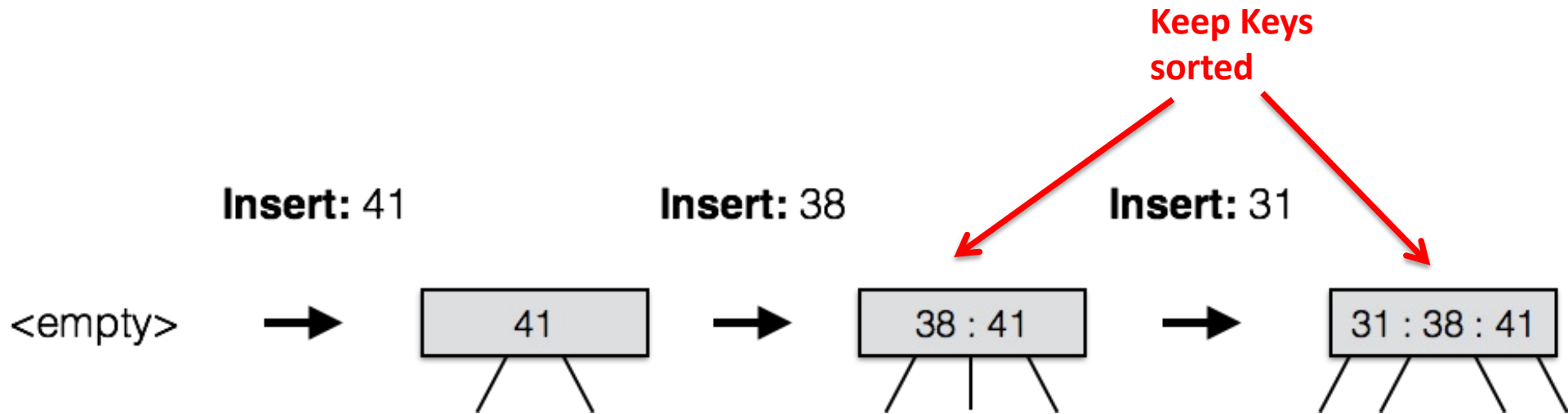
Inserting into a 2-3-4 Tree must maintain Size and Depth properties

Insertion:

1. Begin by searching for Key in 2-3-4 Tree
2. If found, update Value
3. If not found, search terminates at a leaf
4. Do an insert at the leaf
5. Maintain the Size and Depth properties (next slides)

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

Inserting into 2 or 3 node



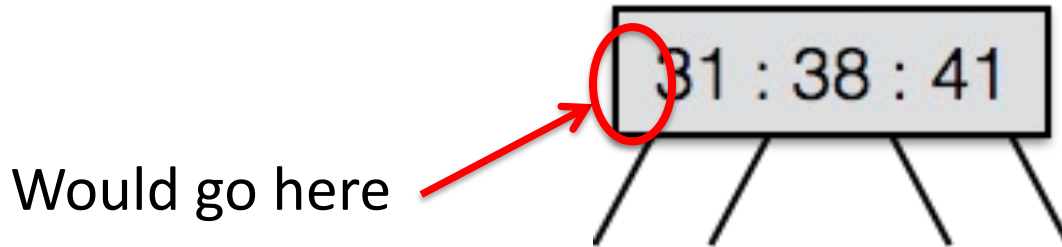
Inserting into a 2 or 3 node:

- Keep keys ordered inside each node
- Can insert key inside a *node* in $O(1)$ because there are only three places where Key could go
- So, we can update a node in constant, $O(1)$, time

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

Inserting into 4 node

Insert: 12

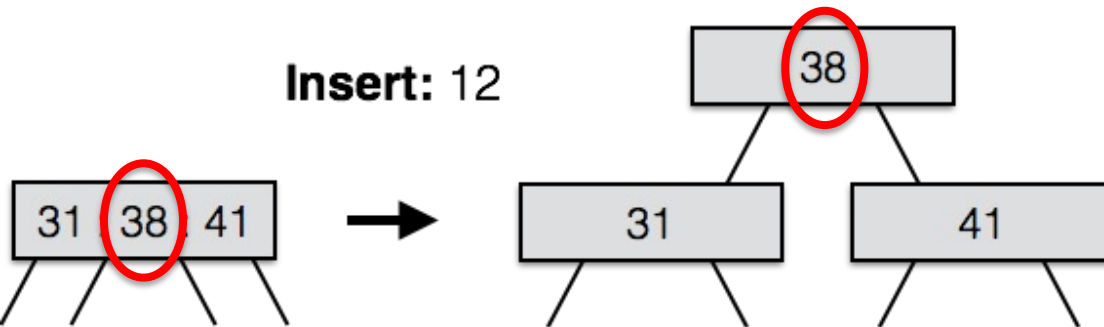


Insert would cause size violation for this node

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



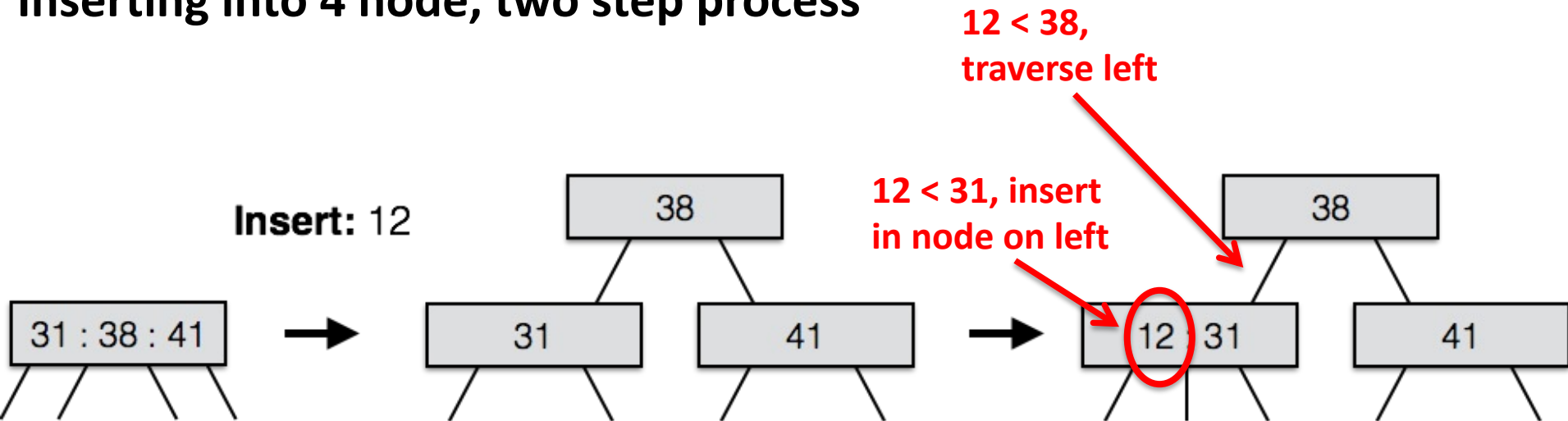
Step 1: split/promote

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

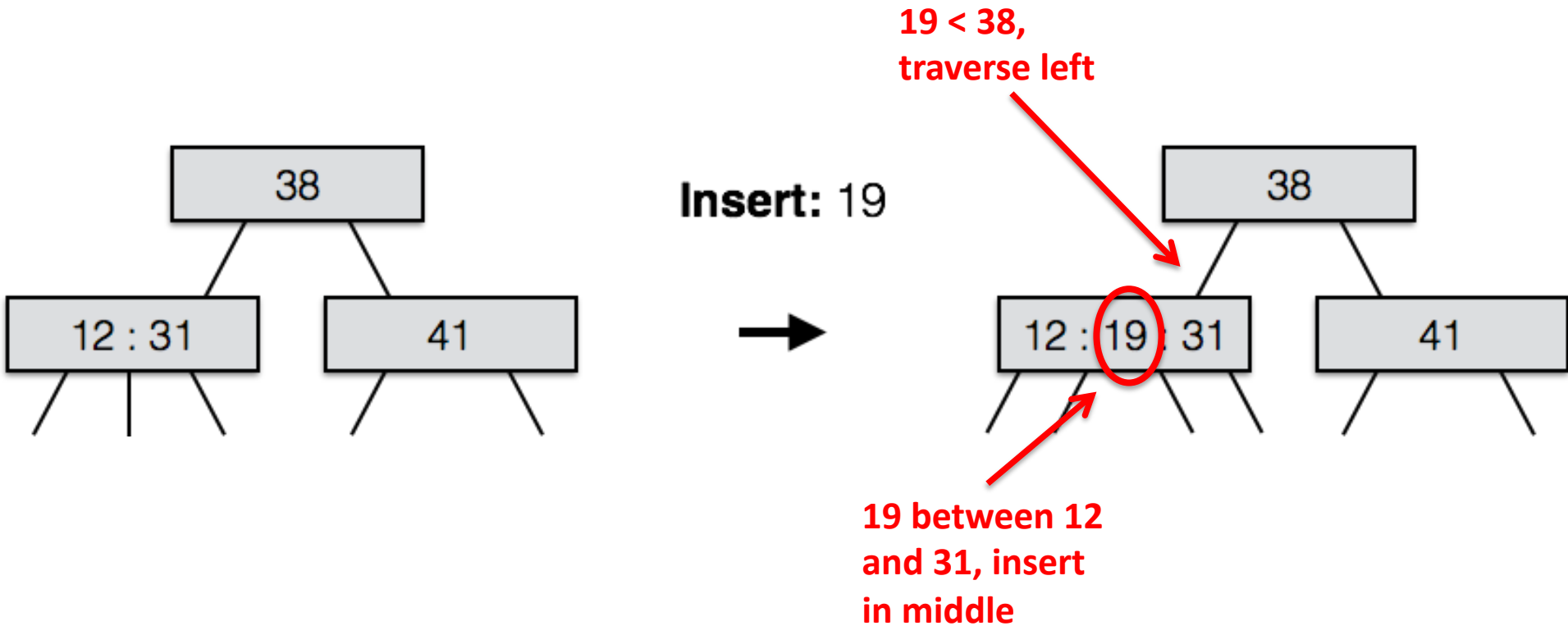
- May become new root
- Parent may have to be split also!

Step 2: insert

Insert 12 into appropriate node at lowest level

Continue inserting until need to split nodes

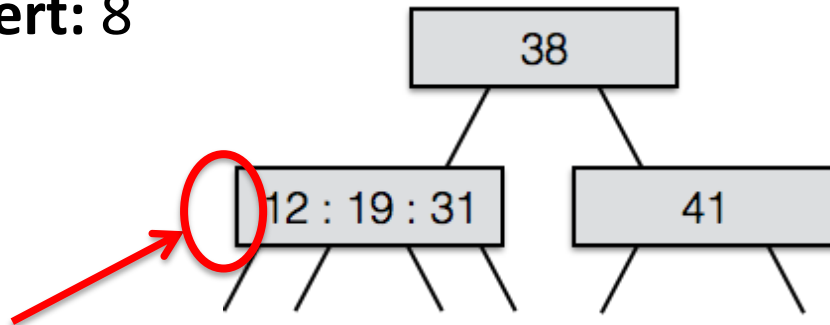
Insert process



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

Insert process

Insert: 8

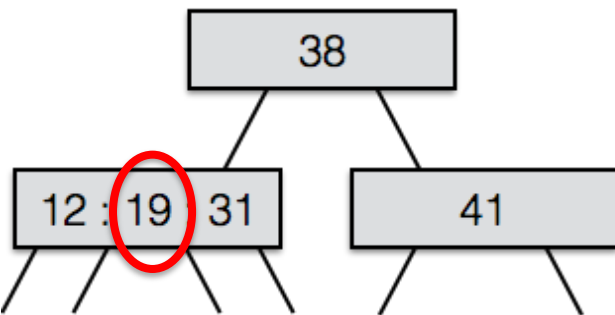


Would go here

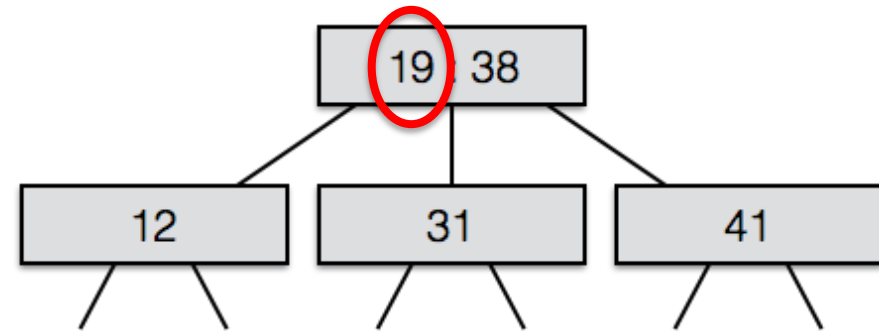
Insert would cause size violation for this node

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

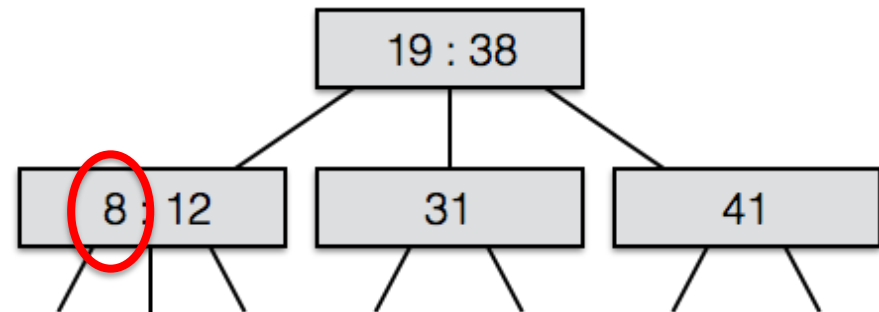
Insert process



Insert: 8



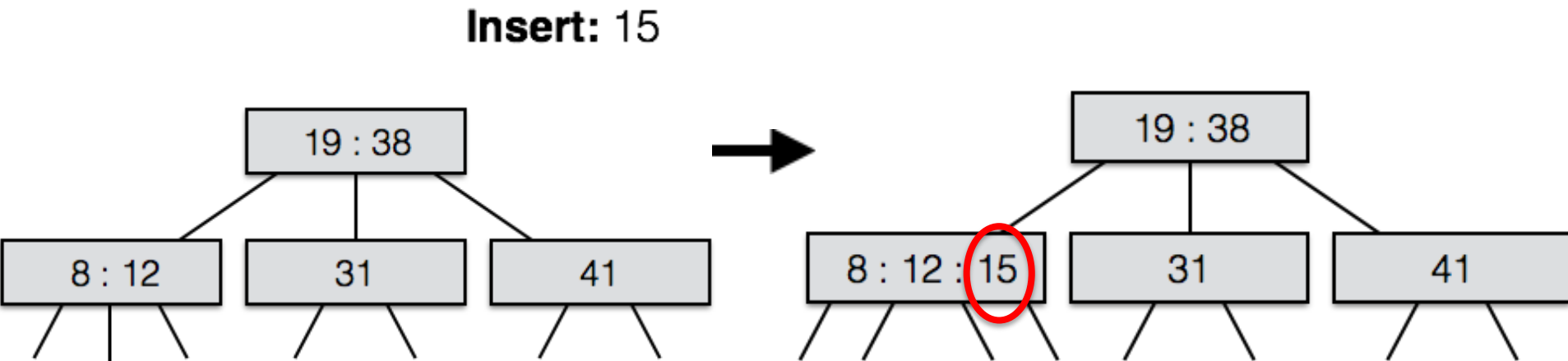
step 1: split/promote



step 2: insert

Always insert new key in lowest level

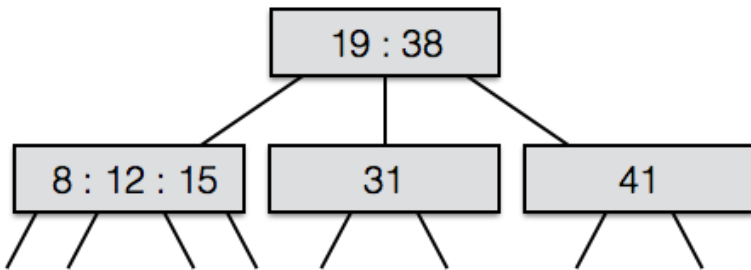
Insert process



Always insert new key in lowest level

Insert process

Insert: 17

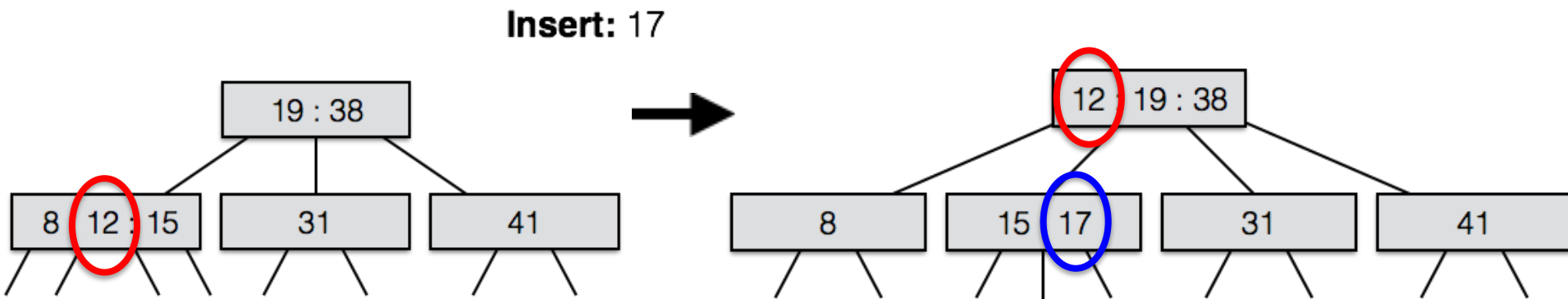


Step 1: Split and promote 12

Step 2: Insert 17

Always insert new key in lowest level

Insert process



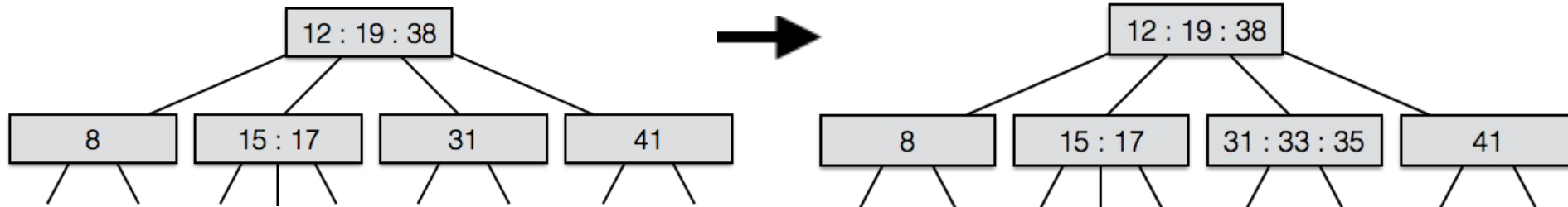
Step 1: Split and promote 12

Step 2: Insert 17

Always insert new key in lowest level

Insert process

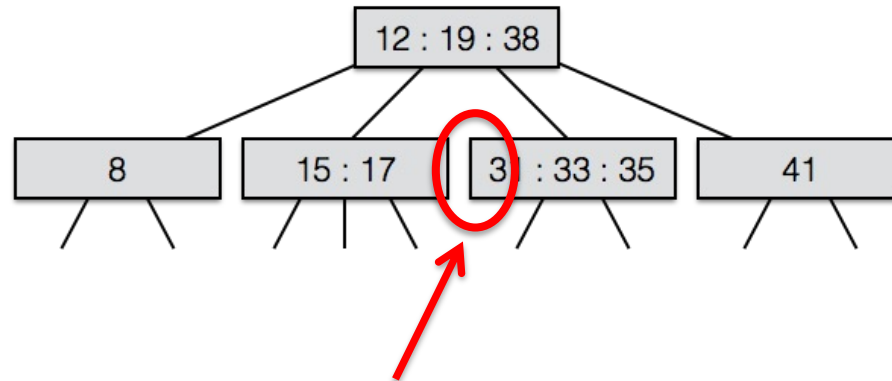
Insert: 33 and 35



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

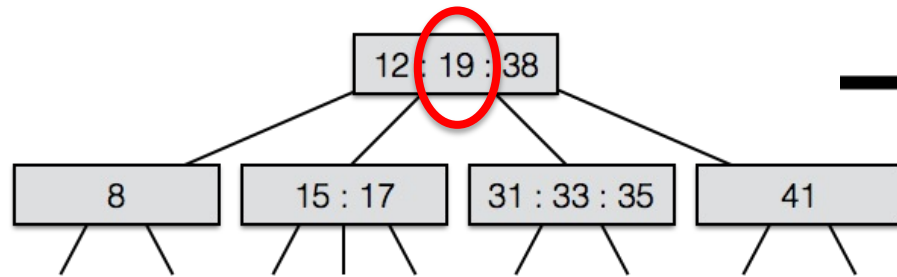
Promoting would cause parent size violation

Split parent first, then split child, then insert

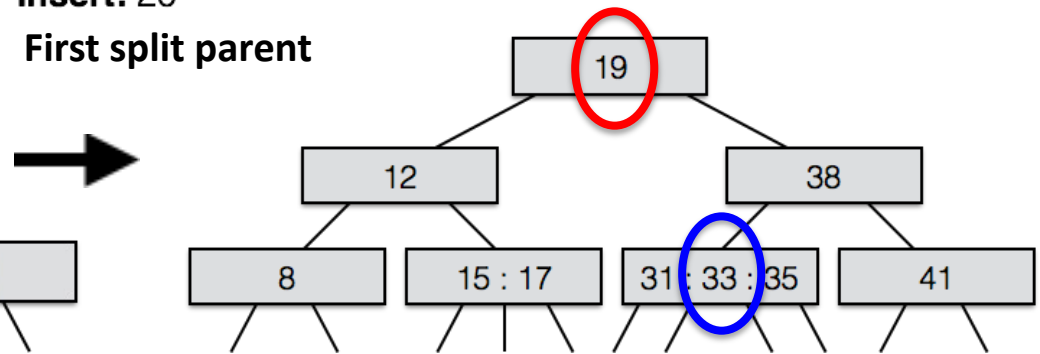
Could bubble up all the way to the root

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

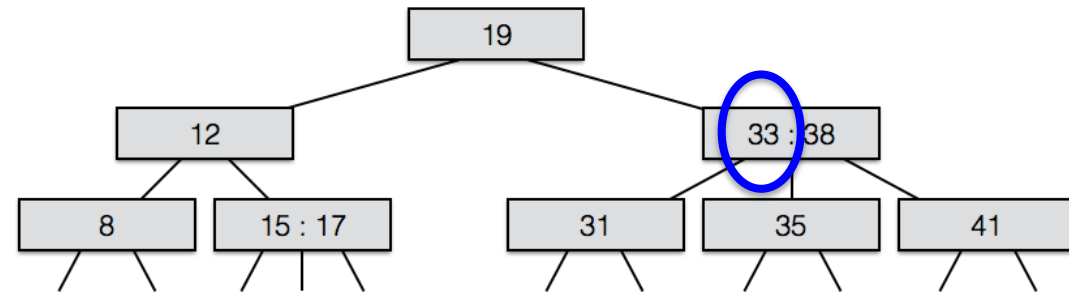
Insert process



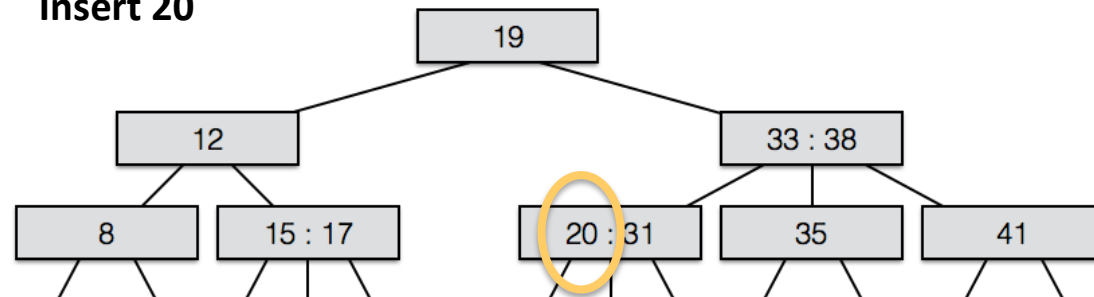
Insert: 20
First split parent



Second split



Insert 20



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

- Need three different types of nodes
- Create new nodes as you need them, then copy information from old node to new node
- Can waste space if nodes have few keys
- 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

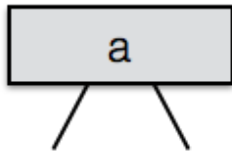
 3. Red-Black Trees

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

Overview

- Can think of each 2, 3, or 4 node as 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 with parent

2-node

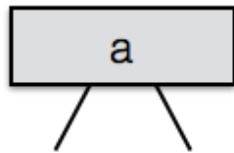


Red-Black trees are binary trees conceptually related to 2-3-4 trees

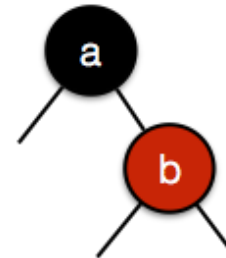
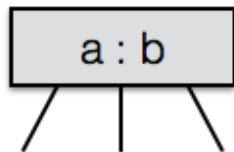
Overview

- Can think of each 2, 3, or 4 node as 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 with parent

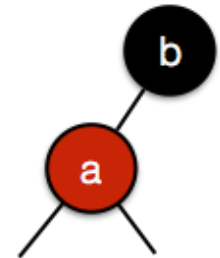
2-node



3-node



or



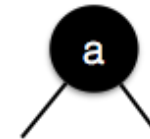
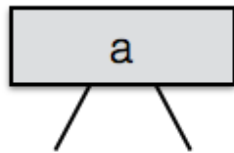
Red node would be in the same node as black parent in a 2-3-4 Tree

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

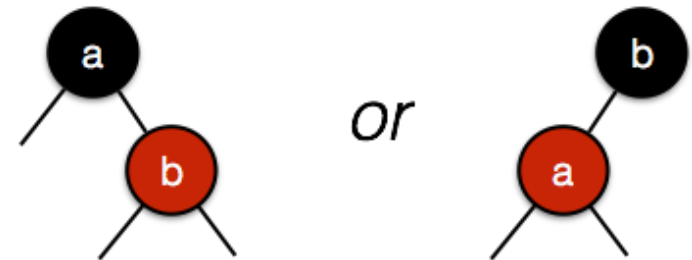
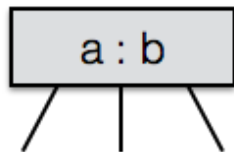
Overview

- Can think of each 2, 3, or 4 node as 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 with parent

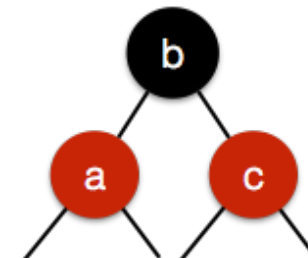
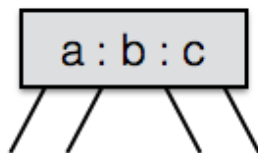
2-node



3-node



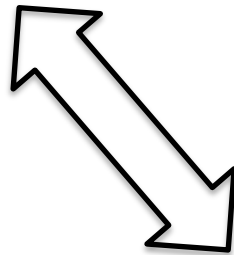
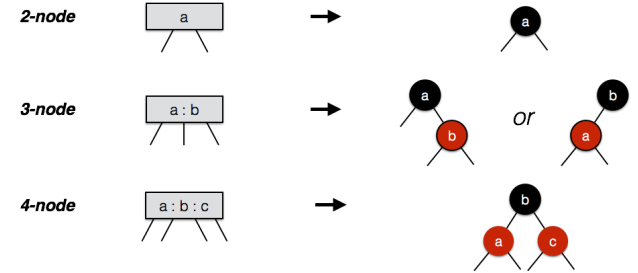
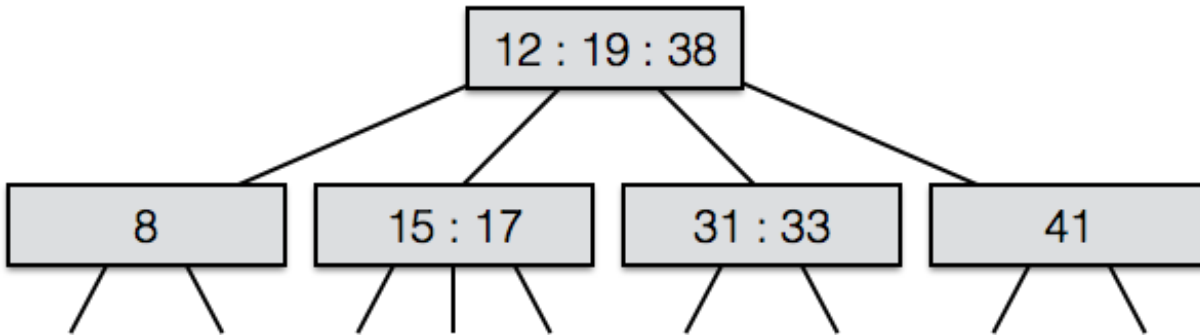
4-node



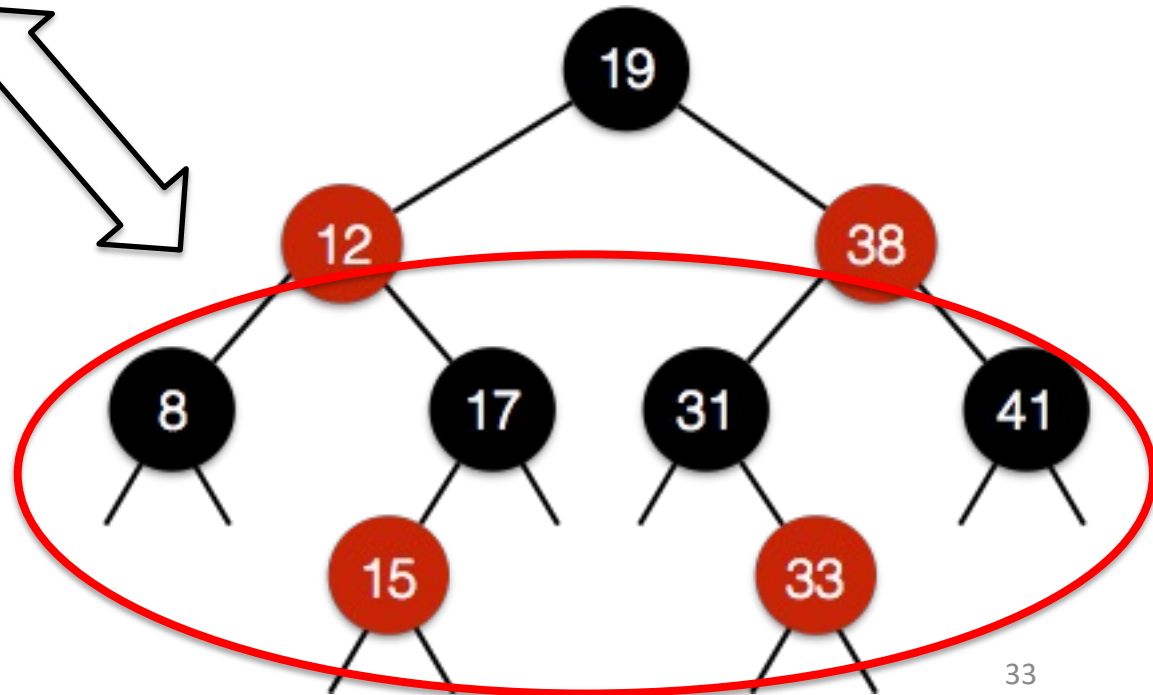
**NOTE: Red-Black trees
are binary trees!**

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!



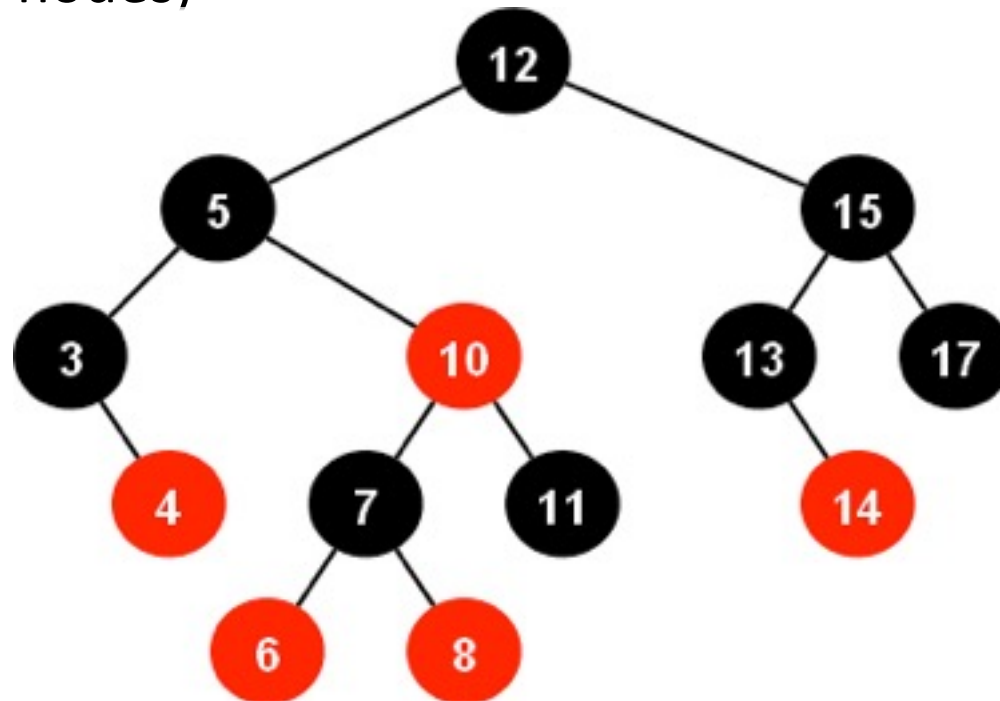
Red-Black trees maintain four properties

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 ancestor nodes)

Black depth: 3

No node more than 3 black nodes away from root



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
- What is the longest possible path?
 - It would have alternating black and red nodes
 - Since there can't be two red nodes in a row (property 3) and root is black (property 2), the longest path given k black nodes is $2k$ or $h \leq 2k$, where h is Tree height
- It can be shown that if *each* path from root to leaf has k black nodes, there must be at least 2^{k-1} nodes in the tree
- Since $h \leq 2k$, then $k \geq h/2$, so there must be at least $2^{(h/2)} - 1$ nodes in the tree
- If there are n nodes in the tree then:
 - $n \geq 2^{(h/2)} - 1$
 - 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 means h is upper bound by $2\log(n+1) = O(\log n)$

Run time complexity of a search operation is $O(h)$ in a Binary Tree, which we just argued is $O(\log n)$ in the worst case here

Searching a Red-Black Tree is $O(\log n)$

- Red-Black tree is a Binary Search Tree with search time proportional to height
- Search time takes $O(\log n)$ since h is $O(\log n)$
- Hard part is maintaining the tree with inserts and deletes

Insertion into Red-Black trees must deal with several cases

Four Red-Black Tree properties:

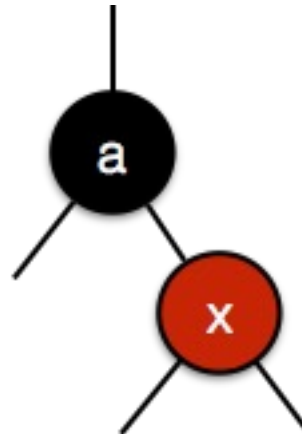
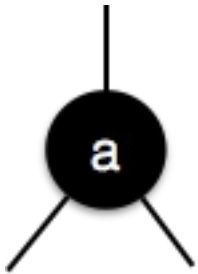
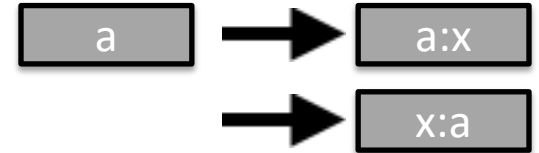
1. Every nodes is either red or black
2. Root is always black, if operation changes it red, turn it black again
3. Children of a red node are black (no consecutive red nodes)
4. All external nodes have the same black depth (same number of black ancestors)

Insert procedures

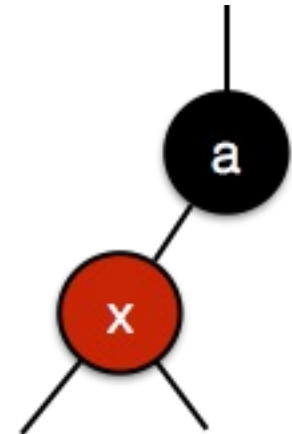
- As with BSTs, find location in tree where new element goes and insert
- Color new node red – ensures rules 1, 2 and 4 are preserved
- Rule 3 might be violated (red node must have black children)
- Three cases can arise on insert (equivalent to 2, 3, or 4 node inserts)
- Inserting into a 2 or 4 node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

Case 1: Insert into 2 node, no violation

Insert into 2 node causes no violation



or



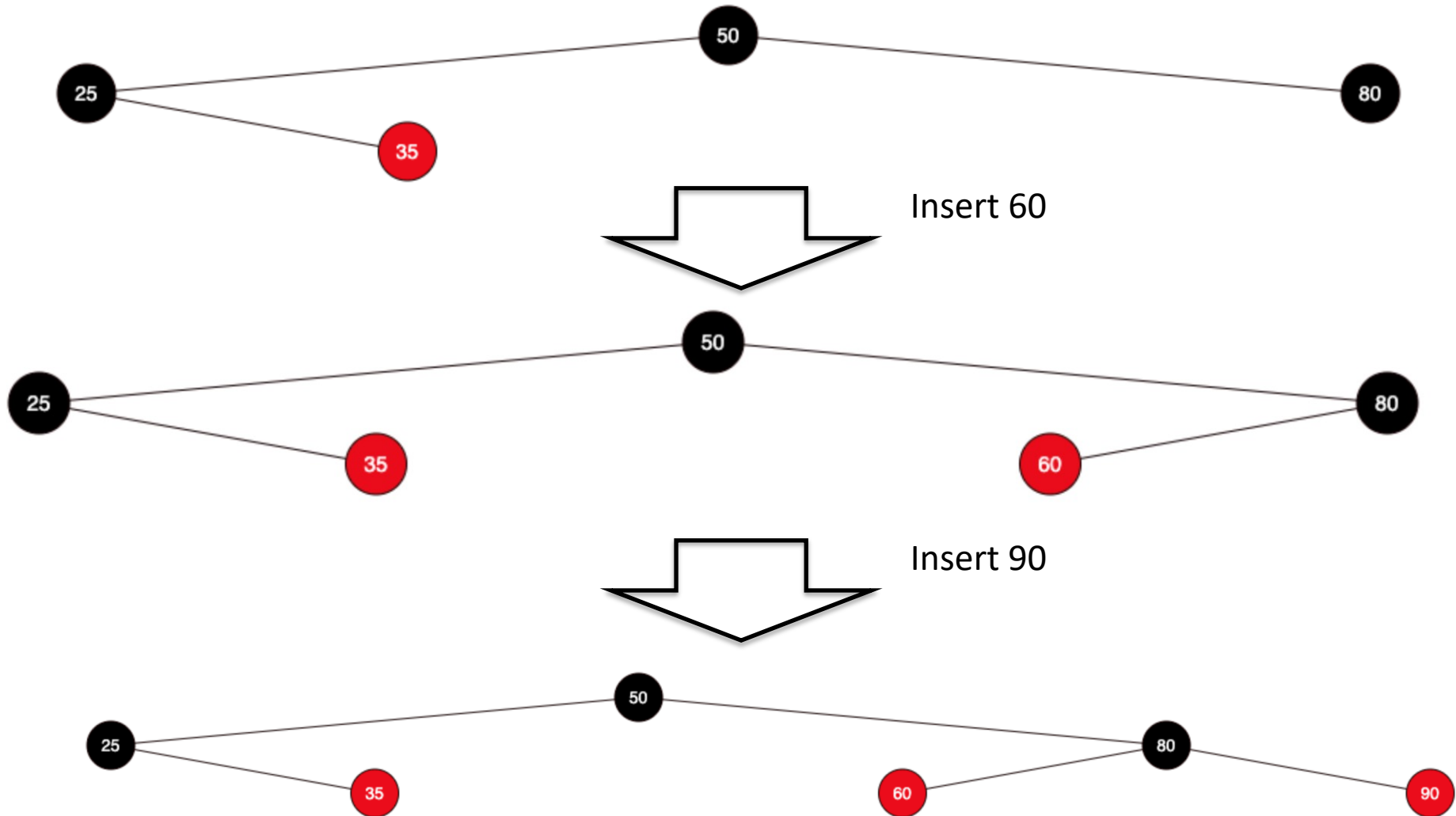
Insert new node $\langle x \rangle$ as child of $\langle a \rangle$

Color $\langle x \rangle$ red

No violations

Each of these Trees are possible depending on the value of $\langle x \rangle$

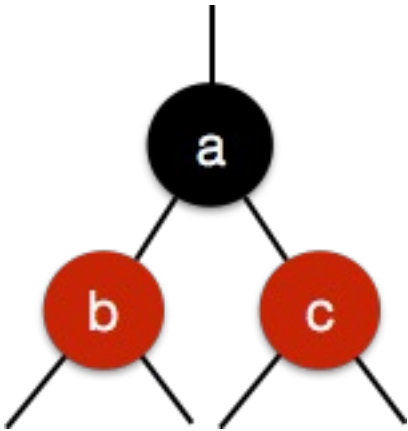
Example Case 1



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

4 nodes are black with red children

b:a:c

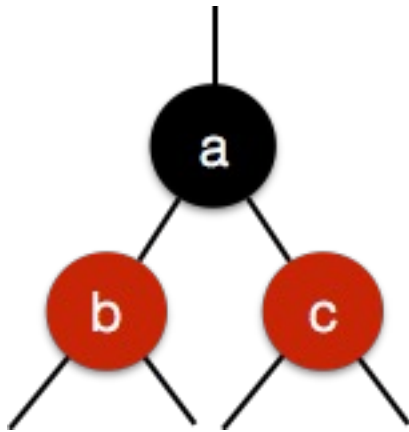


Insert new node <x> as
child of or <c>
would cause two red
nodes in a row

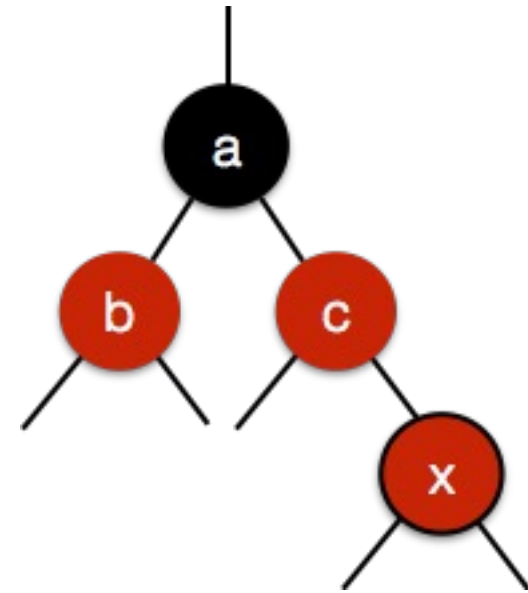
Violates rule 3

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

4 nodes are black with red children



b:a:c



Insert new node <x> as child of or <c> would cause two red nodes in a row

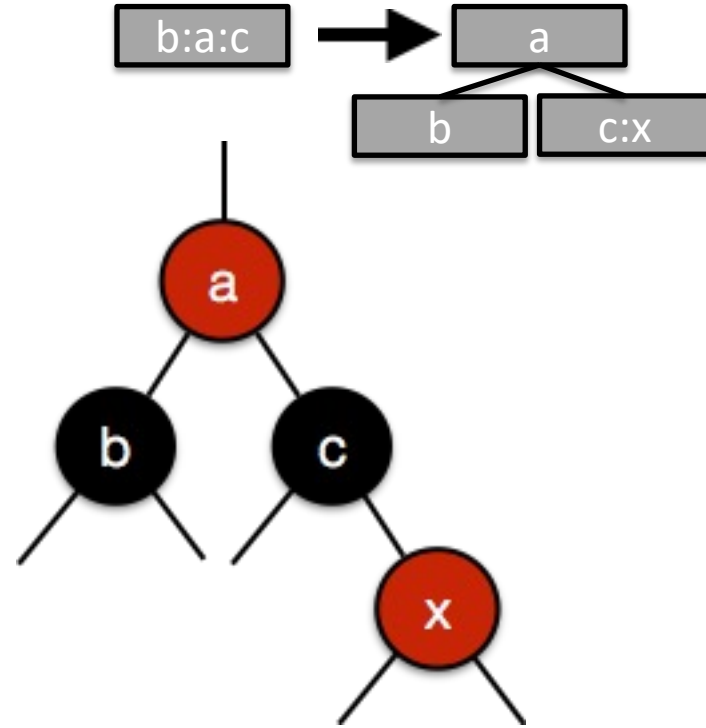
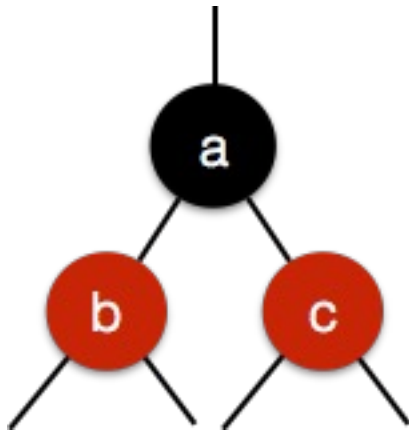
Violates rule 3

Must split node, promoting middle key

- Could promote <a> to parent, and unjoin 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 <x> as child of or <c> would cause two red nodes in a row

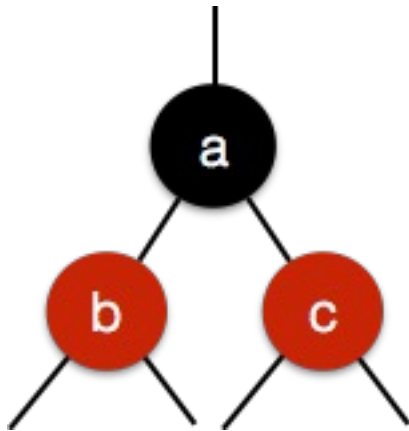
Violates rule 3

Must split node, promoting middle key

- Could promote <a> to parent, and unjoin 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



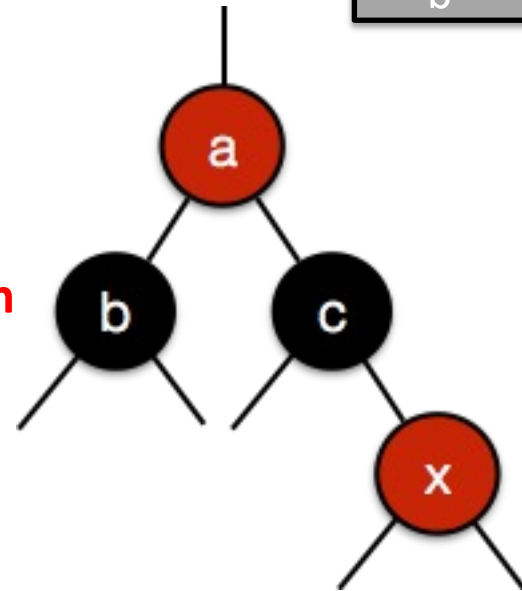
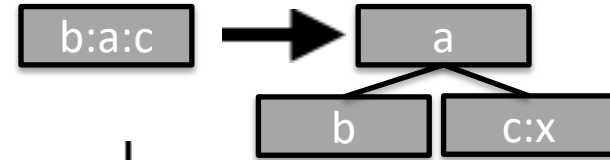
Insert new node <x> as child of or <c> would cause two red nodes in a row

Violates rule 3

Black length not changed

Must check <a> doesn't violate parent two reds in a row

Might bubble up color flips to root

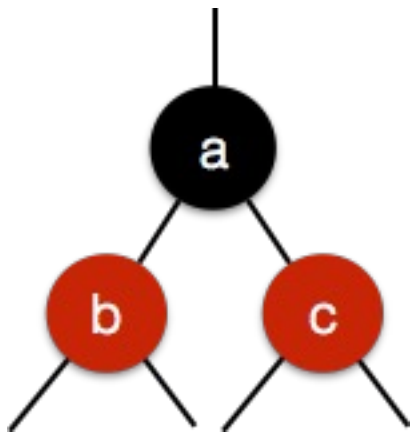


Must split node, promoting middle key

- Could promote <a> to parent, and unjoin 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



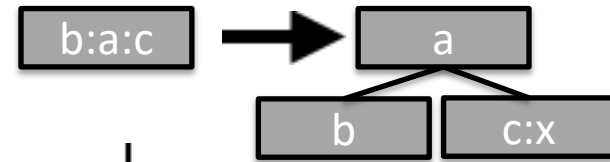
Insert new node <x> as child of or <c> would cause two red nodes in a row

Violates rule 3

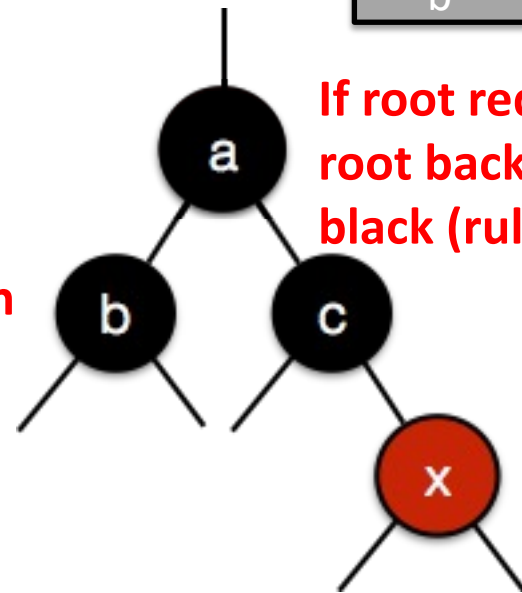
Black length not changed

Must check <a> doesn't violate parent two reds in a row

Might bubble up color flips to root



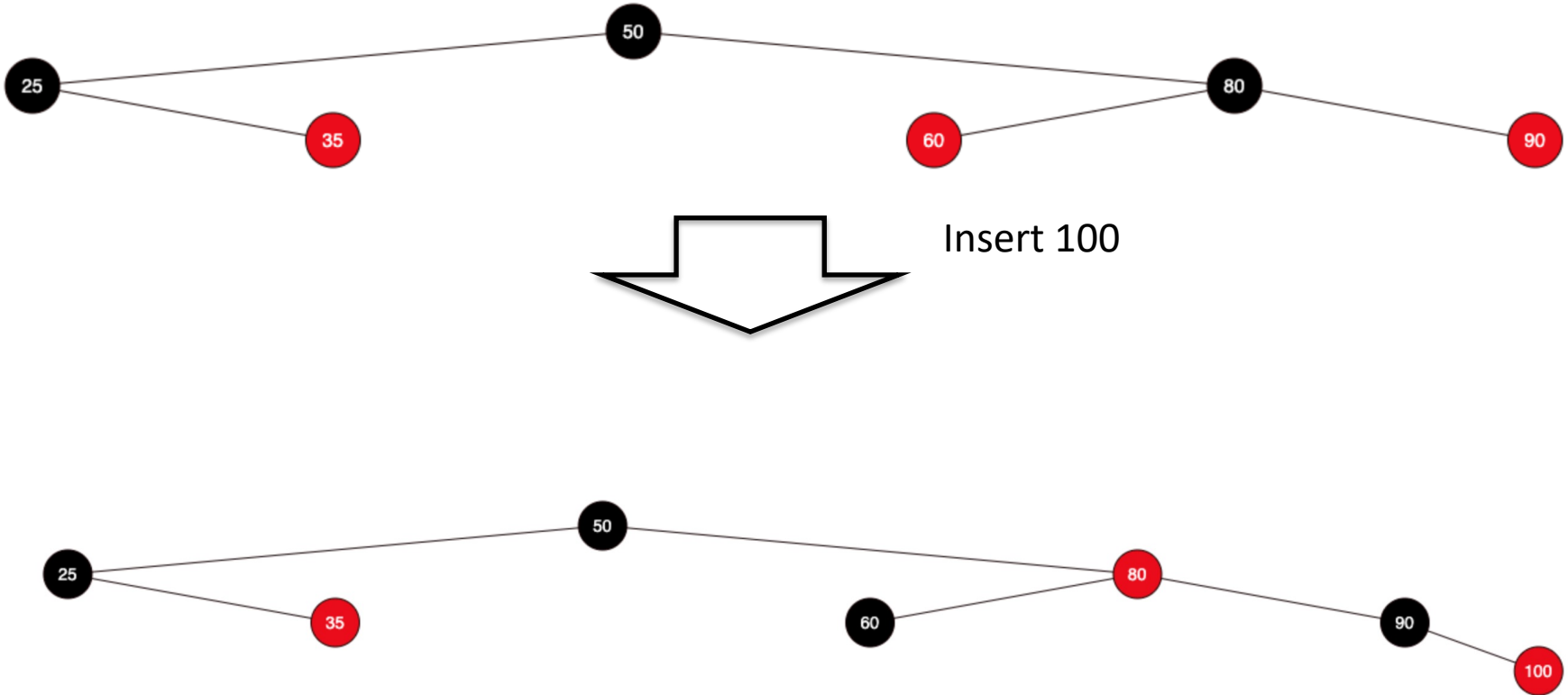
If root red, flip root back to black (rule 2)



Must split node, promoting middle key

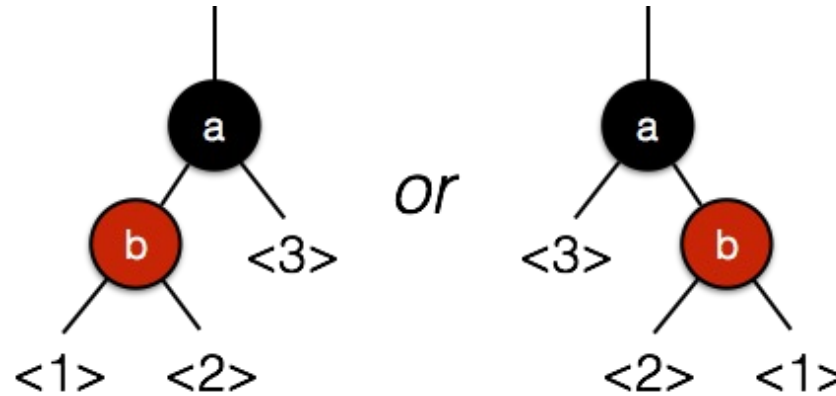
- Could promote <a> to parent, and unjoin and <c> from <a>
- Amounts to a “color flip”

Example Case 2



Case 3: Insert into 3 node, might be violation

3 nodes are black with one red child



With a 3 node there are three places where node could be added: <1>, <2>, or <3>

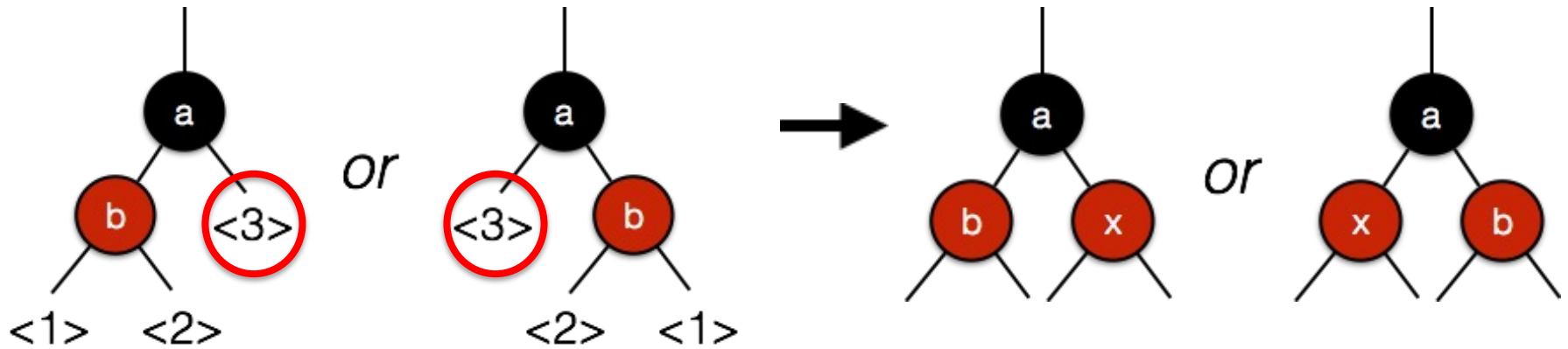
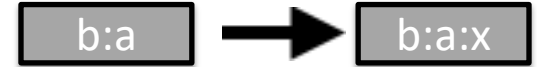
<3> is easy

<1> involves a single rotation (2 reds in straight line)

<2> involves a double rotation (2 reds in zig-zag)

Case 3: Inserting at position <3> is easy

3 nodes are black with one red child



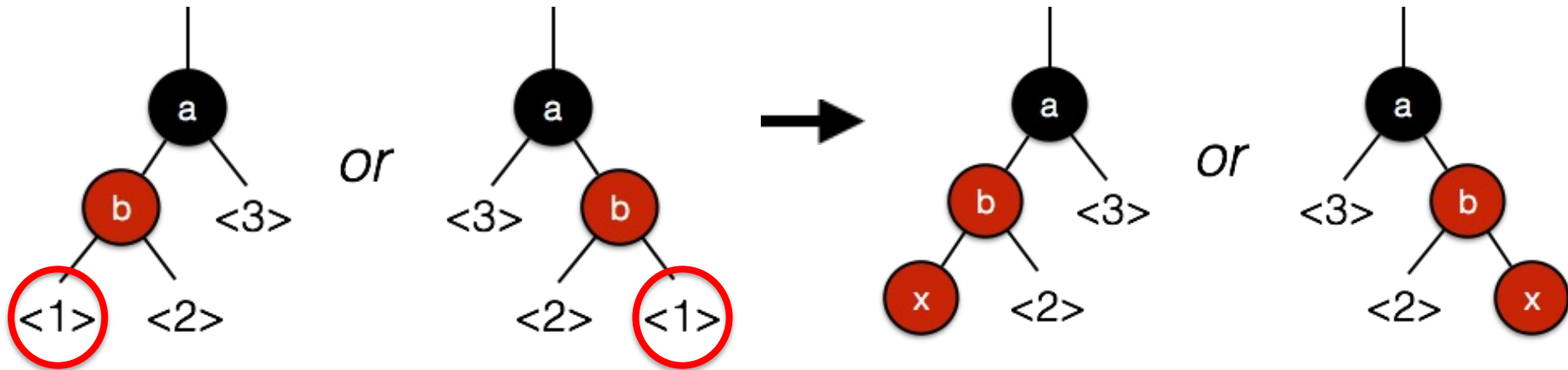
Inserting into position <3> makes a 4 node

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

Case 3: Inserting at position <1> (two red in straight line) causes single rotation

3 nodes are black with one red child

b:a

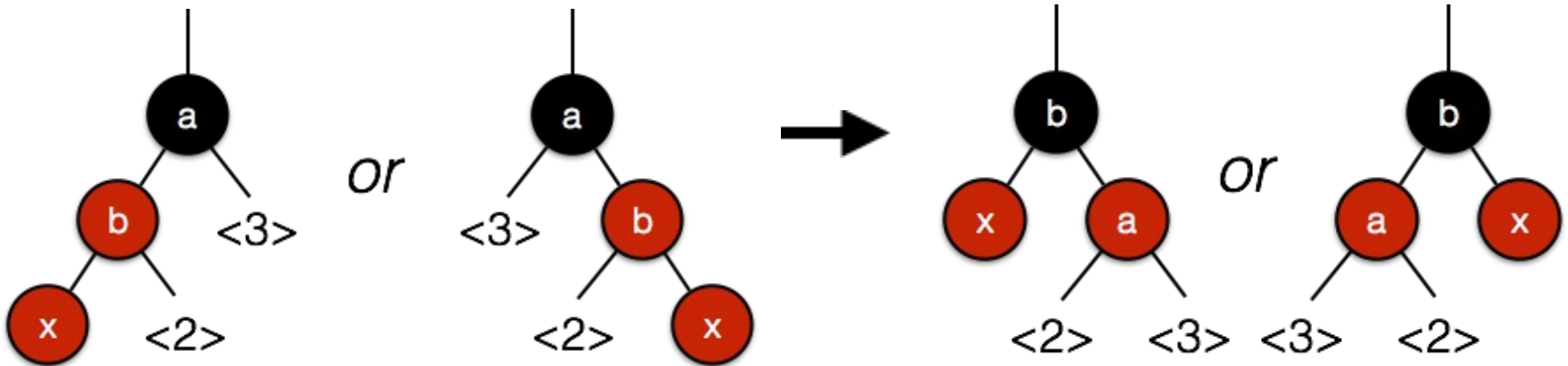
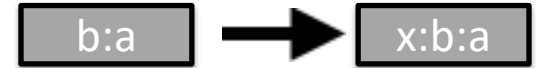


Inserting at <1> do a single rotation

- 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 to root (color black), while dropping down <a> (color red) to be child of

Case 3: Inserting at position <1> (two red in straight line) causes single rotation

3 nodes are black with one red child



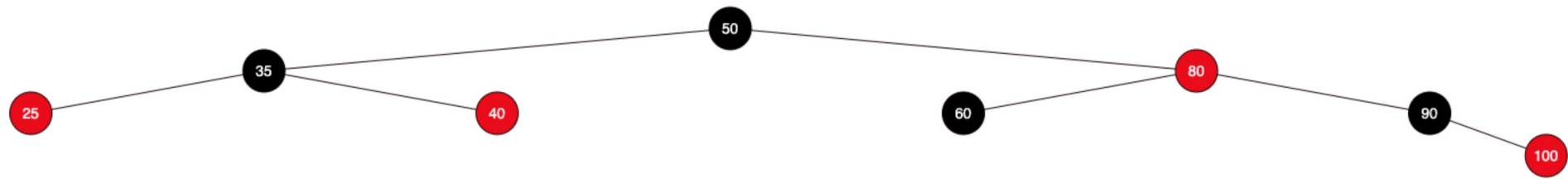
Inserting at <1> do a single rotation

- 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 to root (color black), while dropping down <a> (color red) to be child of
- Still maintains ordered property
- Called a **single rotation**

Example Case 3 <1>



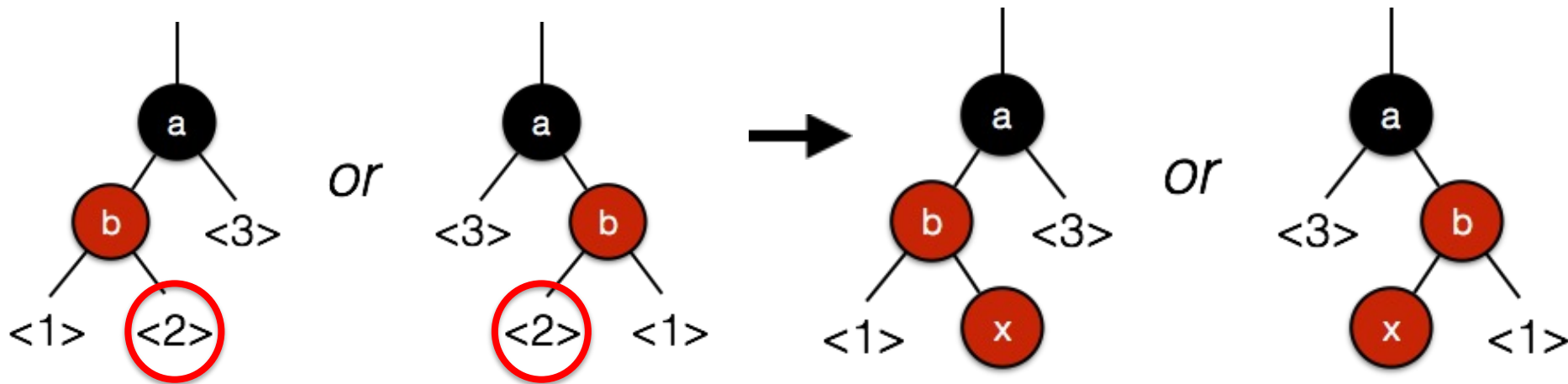
Insert 40



Case 3: Inserting at position <2> (two red in zig-zag) causes double rotation

3 nodes are black with one red child

b:a

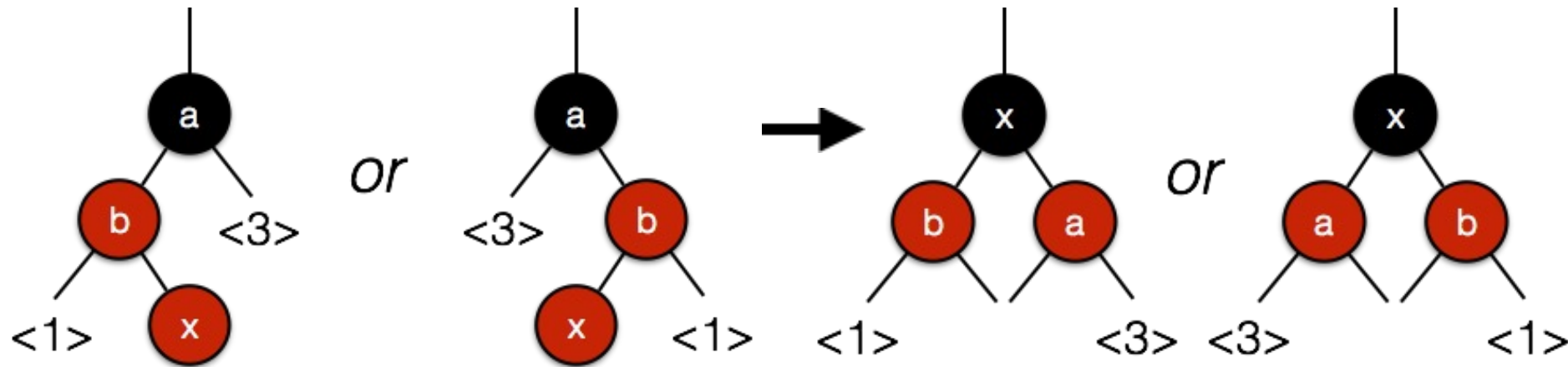
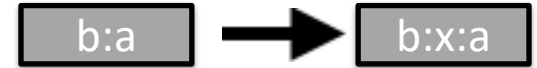


Inserting at <2>, do double rotation

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

Case 3: Inserting at position <2> (two red in zig-zag) causes double rotation

3 nodes are black with one red child

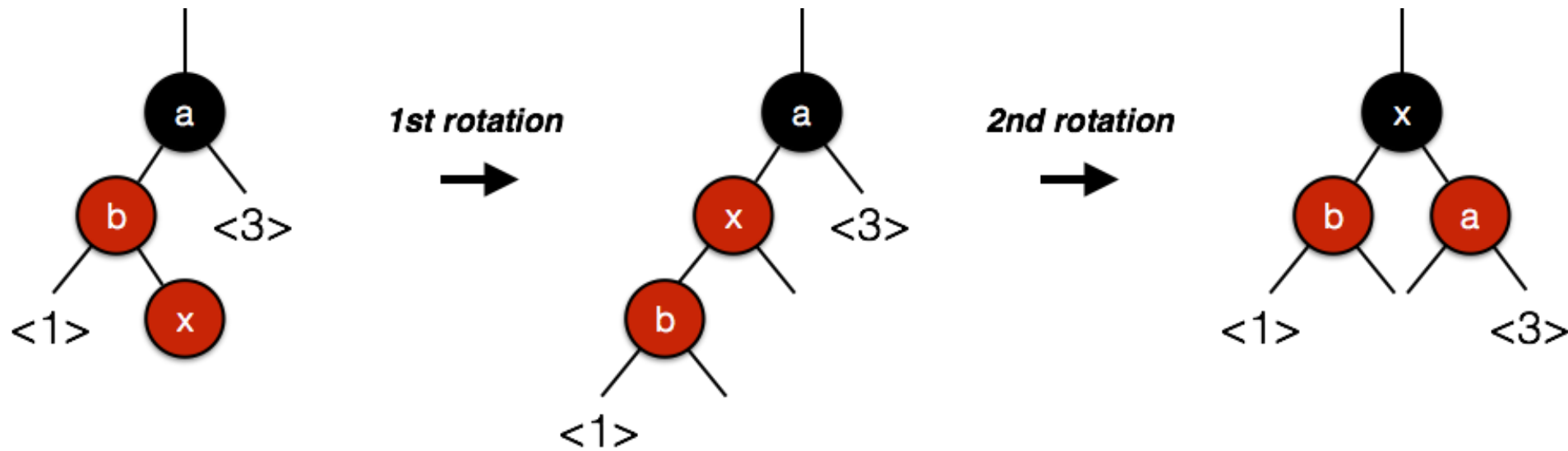
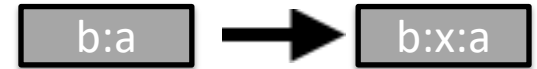


Inserting at <2>, do double rotation

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

Case 3: Inserting at position <2> (two red in zig-zag) causes double rotation

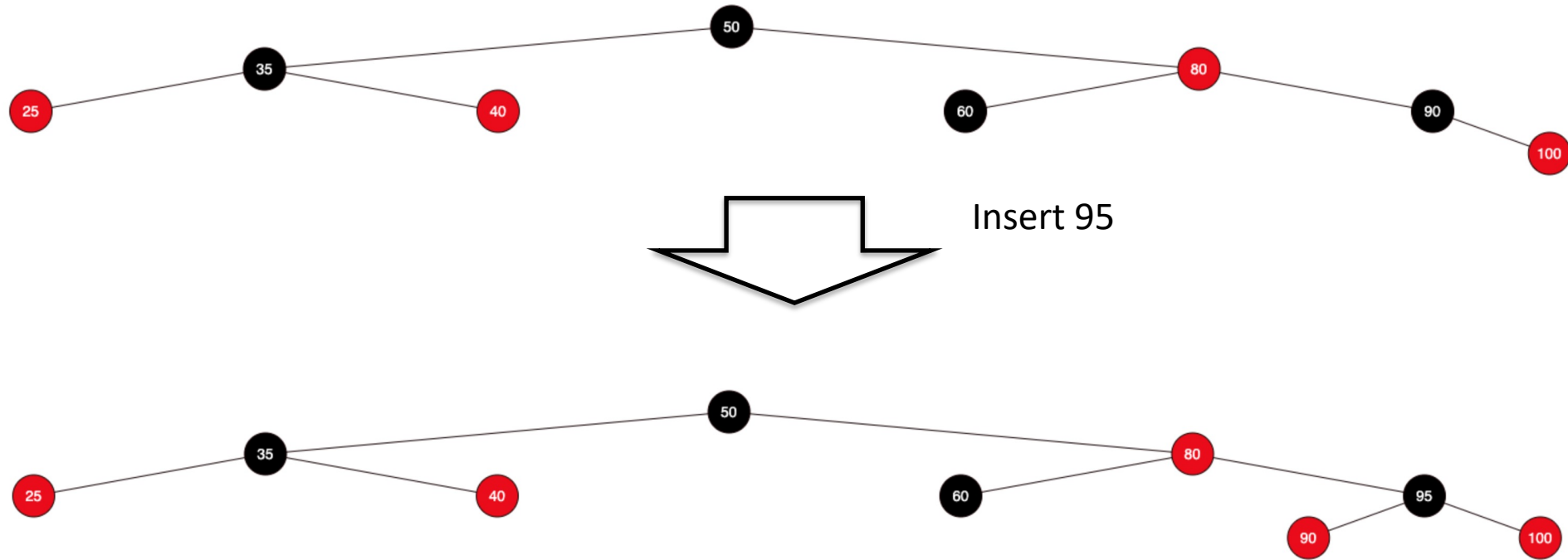
3 nodes are black with one red child



Inserting at <2>, do double rotation

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

Example Case 3 <2>



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 constant time
 - 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, $O(1)$
 - But, might have to traverse up to root
- Leads to $O(\log n)$ insert run-time complexity

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 perfectly balanced**
 - 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 Trees for TreeMap

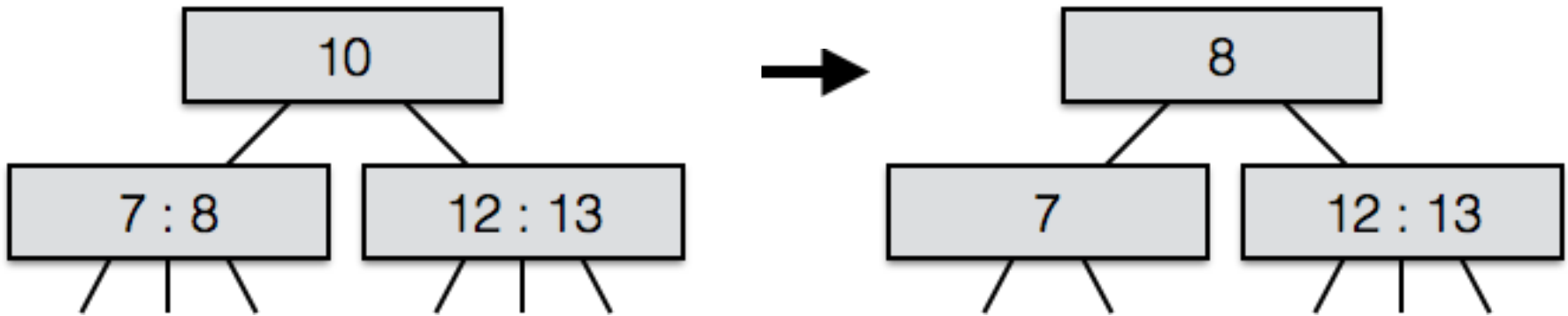
Extra on deletion

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 is easy in 2-3-4 tree

Delete 10

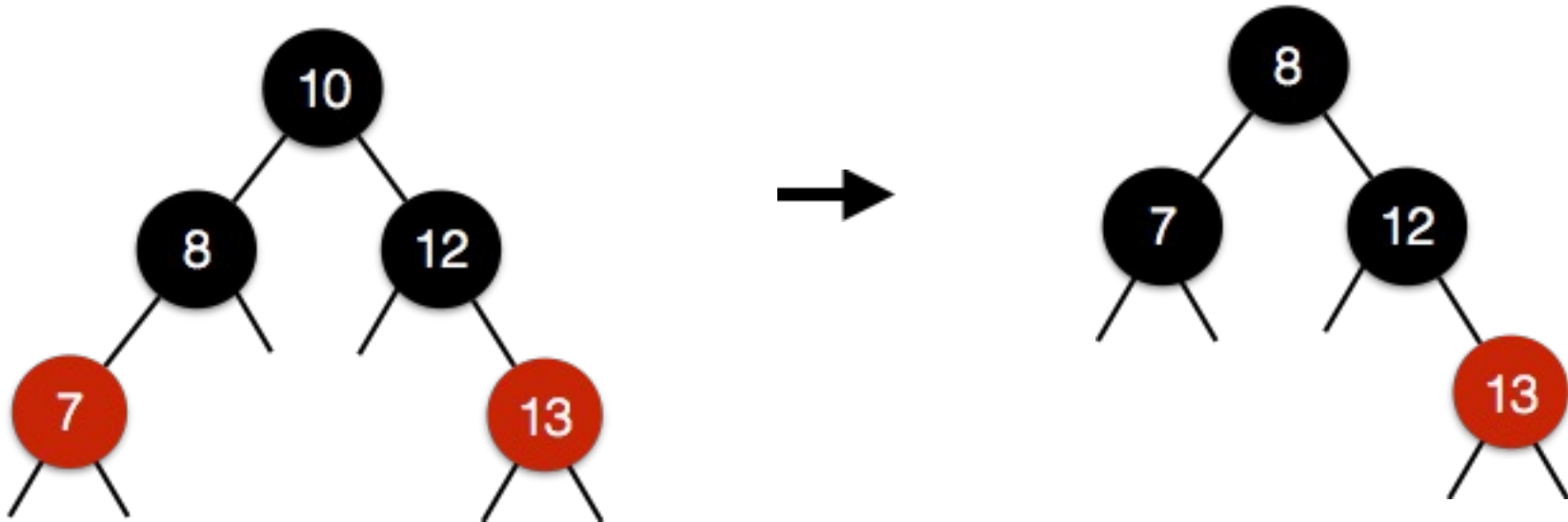


Delete 3 or 4 node in 2-3-4 Tree

1. Find immediate predecessor (largest on left, or successor, smallest on right), 8 here which is in a 3 node
2. Copy Key 8 and Value into space occupied by 10
3. Delete 8 from the 3 node it currently belongs to

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

Delete 10

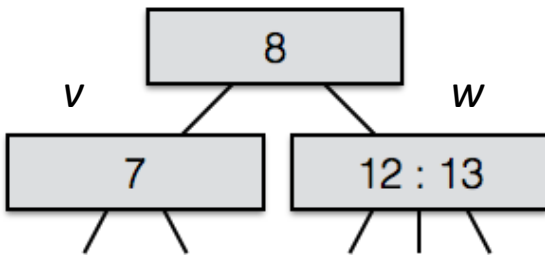


Delete 3 or 4 node in Red-Black Tree

1. Find immediate predecessor, 8 (or successor if no predecessor) in a 3 node (so 8 is black with red children)
2. Replace 10 with 8
3. Delete 8
4. Color 8's children black (does not change black length)

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

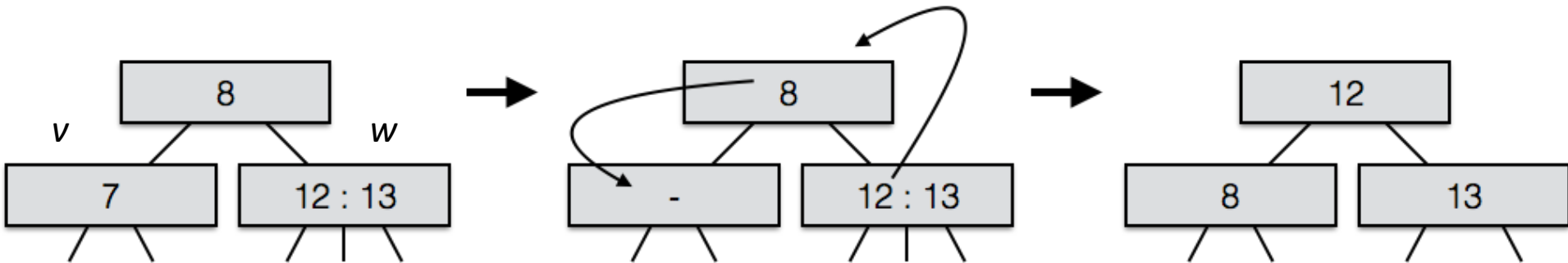
Delete 7



- 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

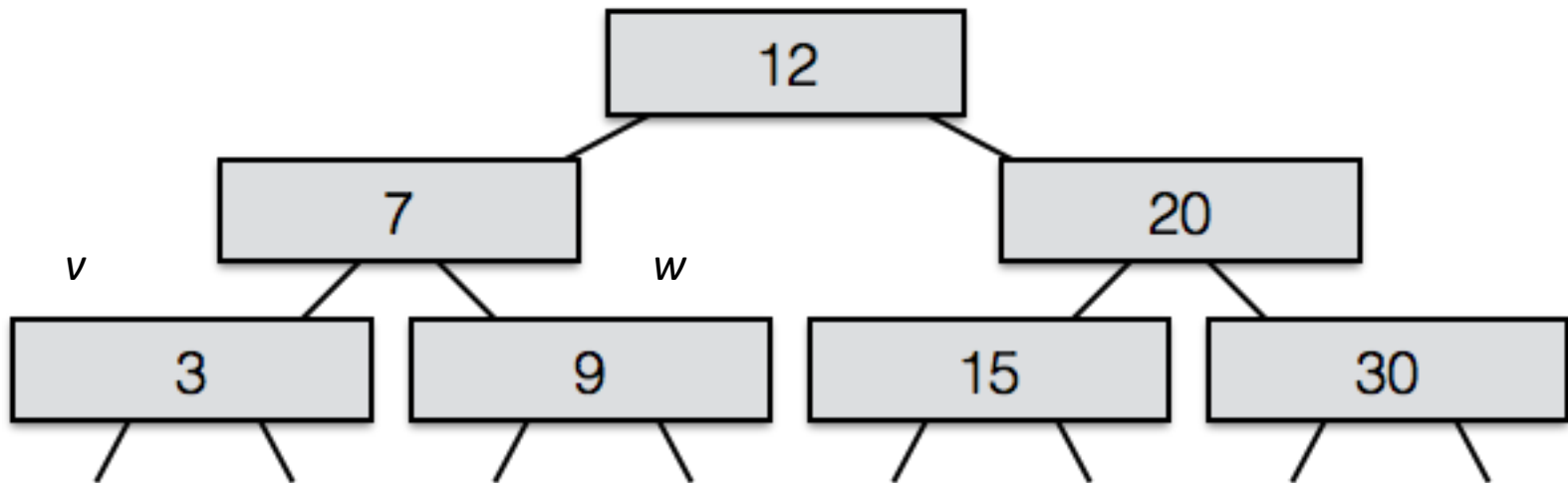
Delete 7



- 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

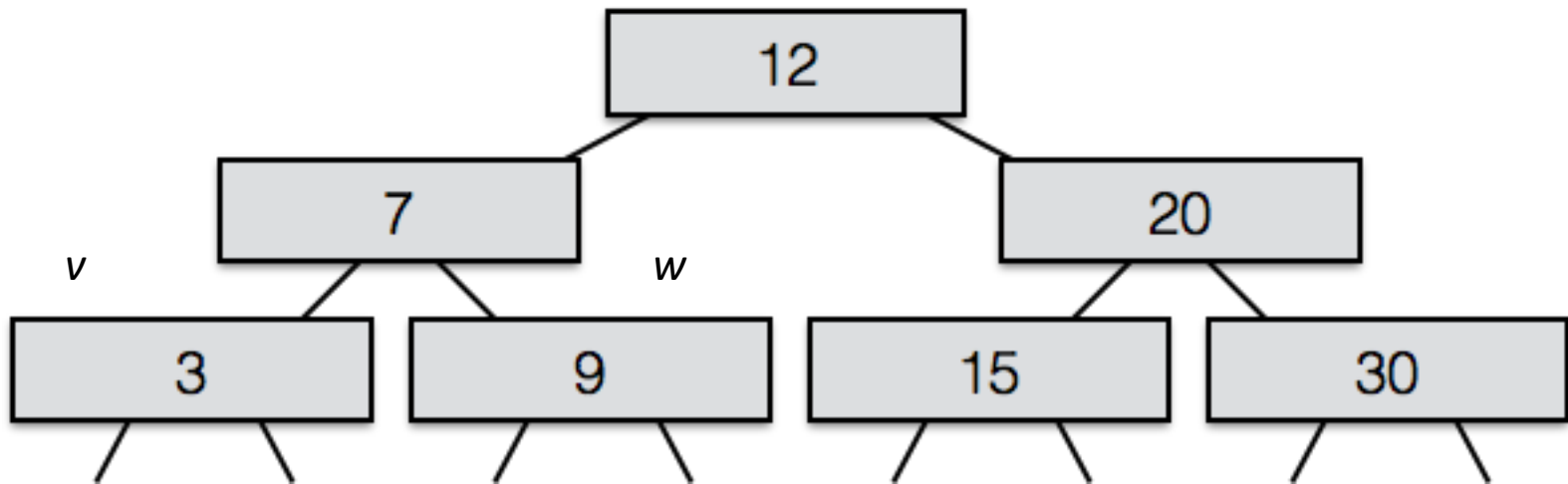
Delete 3



- 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

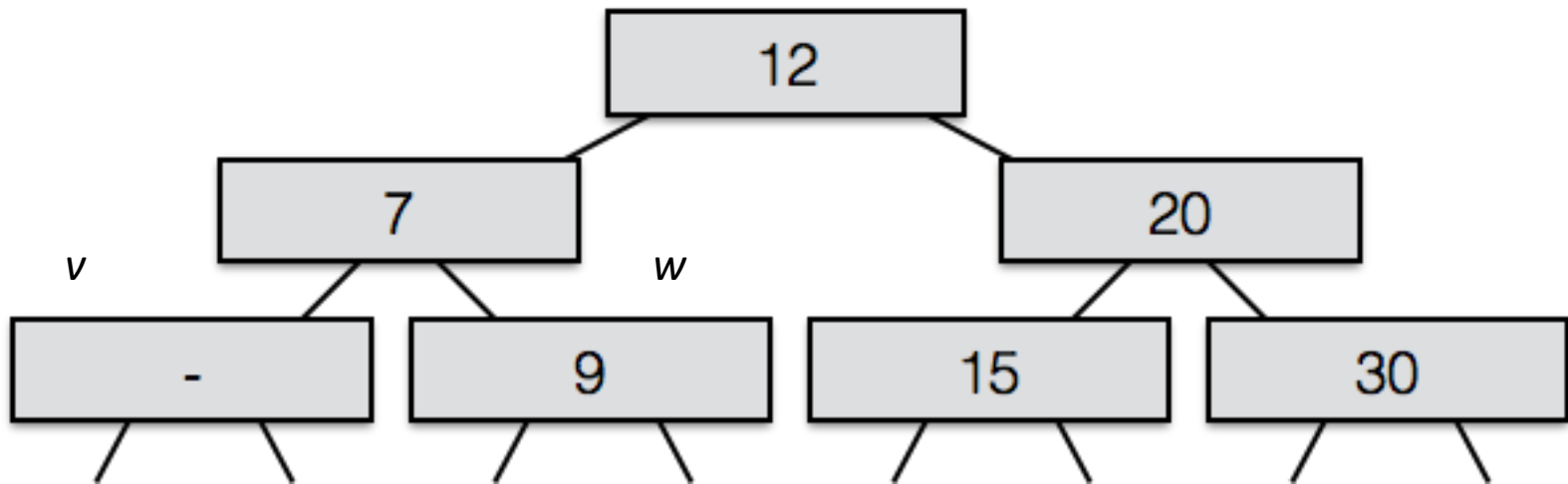
Delete 3



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

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

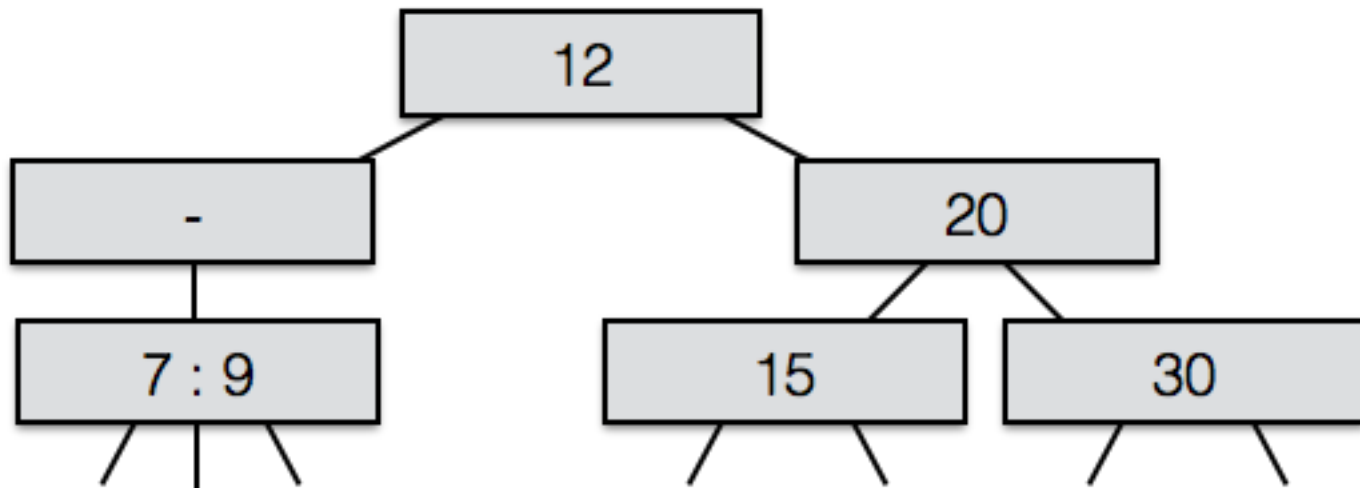
Delete 3



- If w is an adjacent sibling node of v to be deleted and w is 2 node
- Delete node with Key 3
- Pull key down from parent and fuse with w (7 and 9)

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 (7 and 9)
- Keep depth property, so fuse again if needed (12 and 20)

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

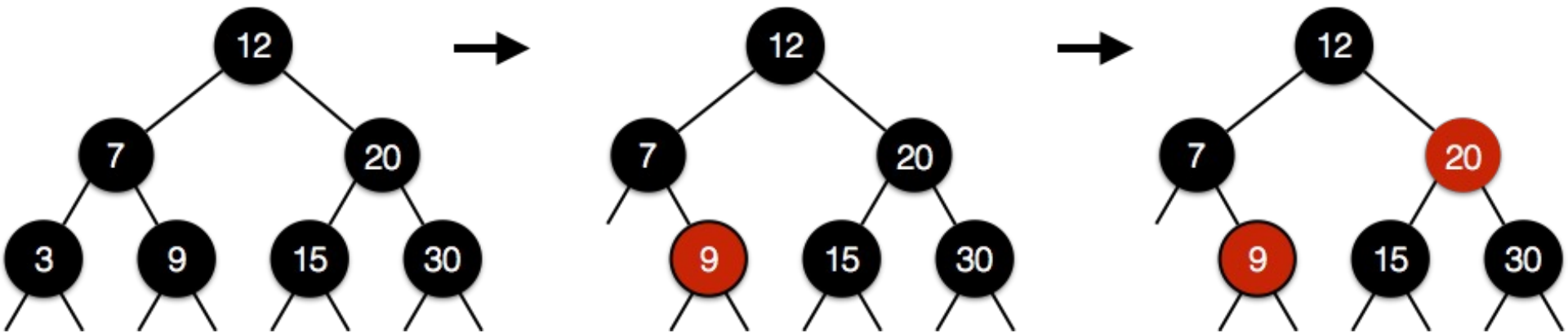
Delete 3



- 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 (7 and 9)
- Keep depth property, so fuse again if needed (12 and 20)
- Tree may lose level if root is fused

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

Delete 3



- Net result is same as 2-3-4 Tree deletion
- See book for more details