Red-Black Trees

Outline

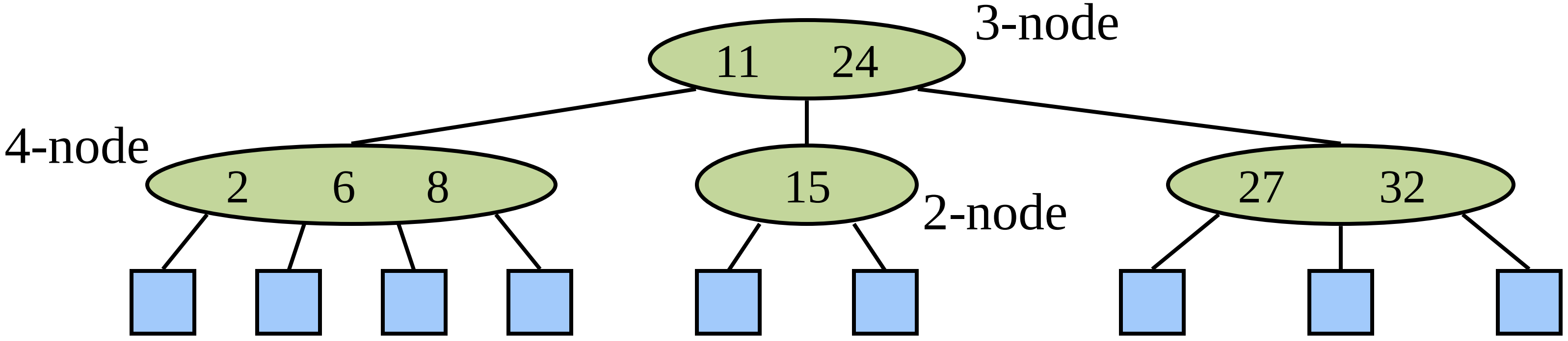
- From (2,4) trees to Red-Black trees
- Definition and height
- Search
- Insertion
  - Restructuring
  - Recoloring
- Deletion
  - Restructuring
  - Recoloring
  - Adjustment

(2,4) Trees

A multi-way search tree, where an internal node has *k* children and stores *k-1* elements, and it has the following additional properties:

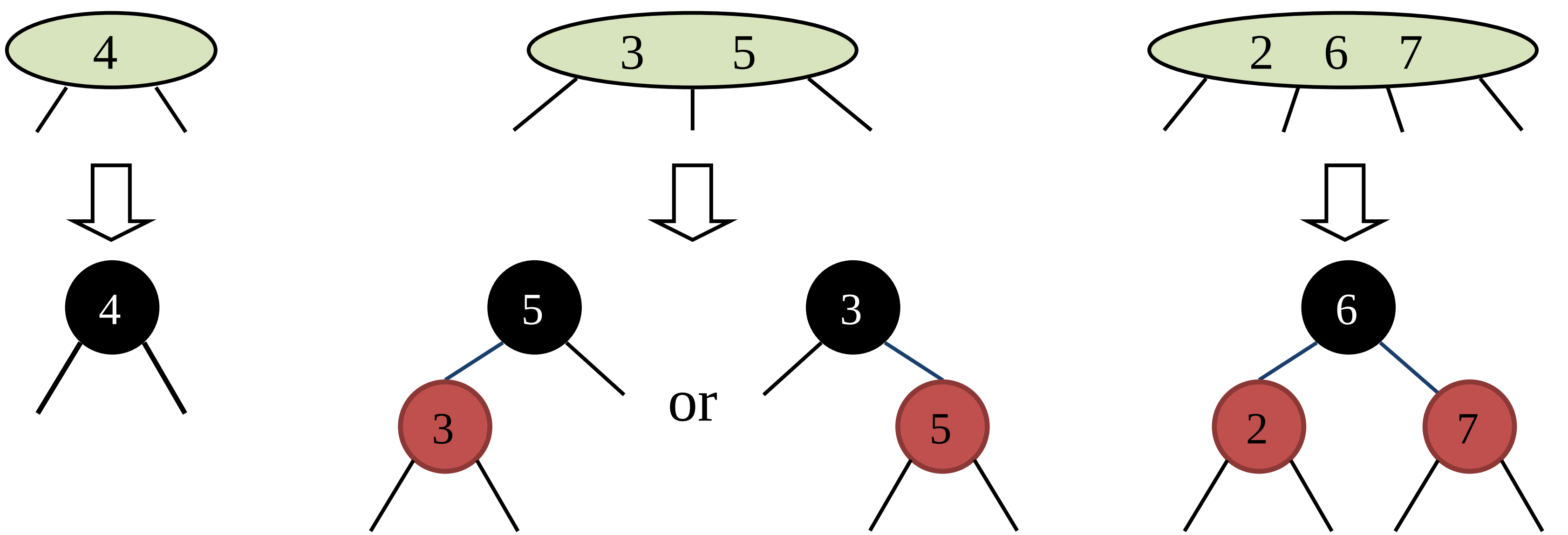
- Node-Size property: all internal nodes have at most four children (i.e., $k = 2,3,4$)
- Depth property: all external nodes have the same depth

Depending on the number of children, an internal node is called either a 2-node, 3-node, or 4-node



From (2,4) to Red-Black Trees

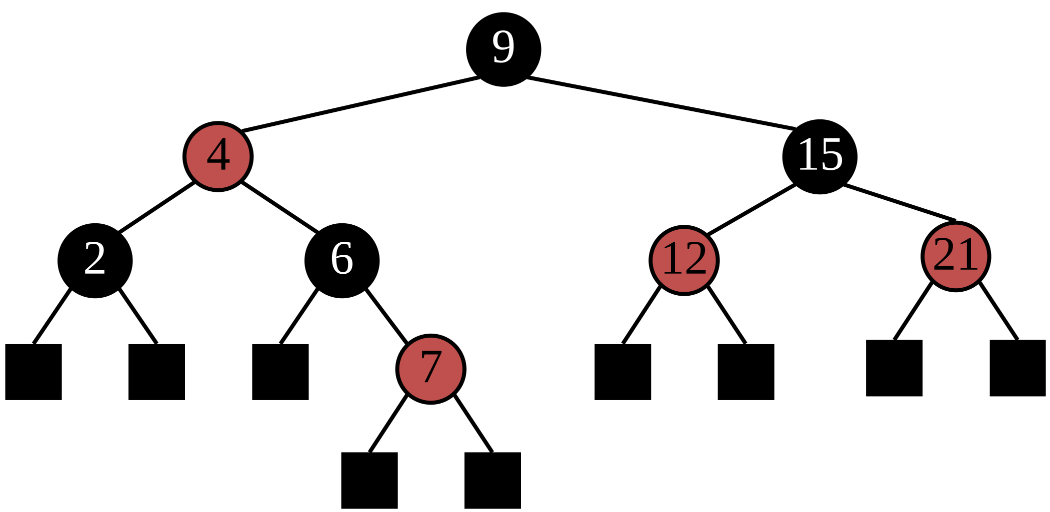
- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or black.
- In comparison with a (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type



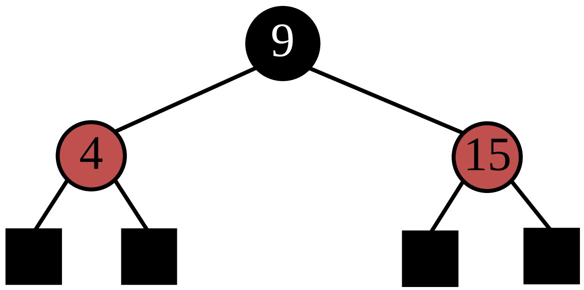
Red-Black Trees

A **binary search tree** with nodes colored red and black in a way that satisfies the following color properties:

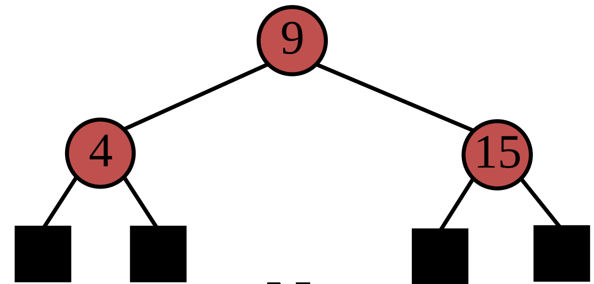
1. Root property: the root is black.
2. External property: every leaf is black.
3. Internal property: the children of a red node are black.
4. Depth property: all leaves have the same black depth.



Ex: Is it a Red-Black Tree?

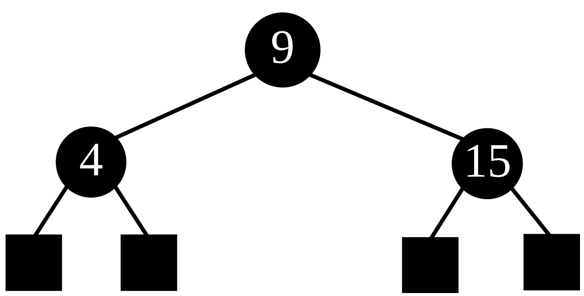


Yes

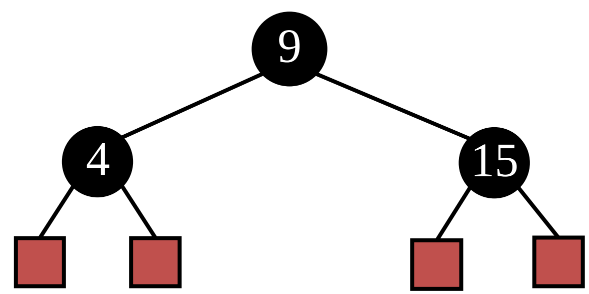


No

Violates root & internal property



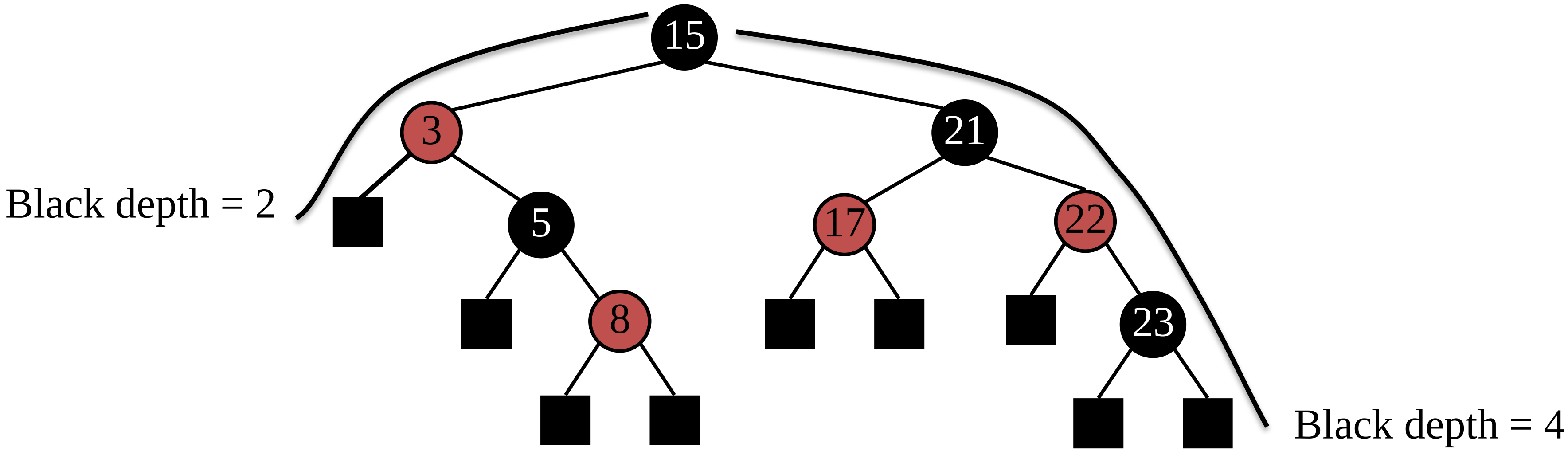
Yes



No

Violates external property

Ex: Is it a Red-Black Tree?

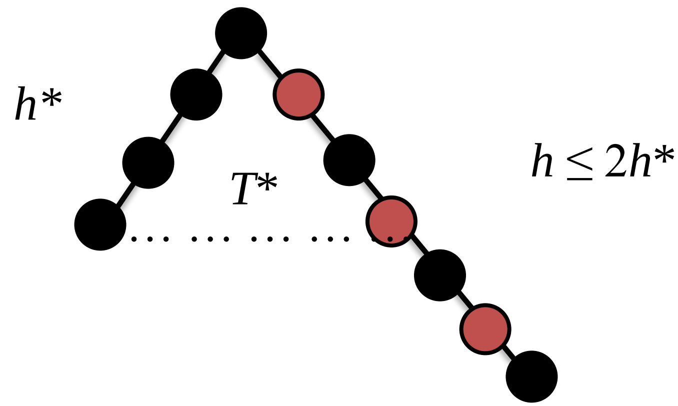


No
Violates depth property

Height of a Red-Black Tree

Theorem: A red-black tree storing $\boldsymbol{n}$ items has height $\boldsymbol{O}(\log \boldsymbol{n})$

Proof:



Consider the shortest path (left) and longest path (right) from the root to an external node.

Let T^* be the portion of the tree T consisting of all nodes with depth $\leq h^*$

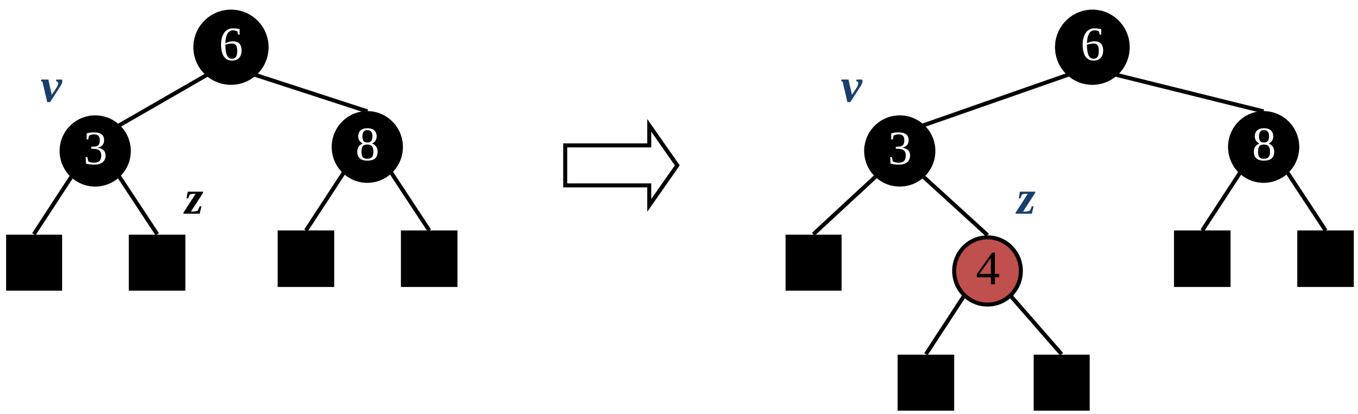
T^* is complete. Thus, $h^* \leq \log \boldsymbol{n}$.

Because $h \leq 2h^*$, $h \leq 2\log \boldsymbol{n} \in \boldsymbol{O}(\log \boldsymbol{n})$.

- The search algorithm for a red-black tree is the same as that for a binary search tree.
- By the above theorem, searching takes $\boldsymbol{O}(\log \boldsymbol{n})$ time

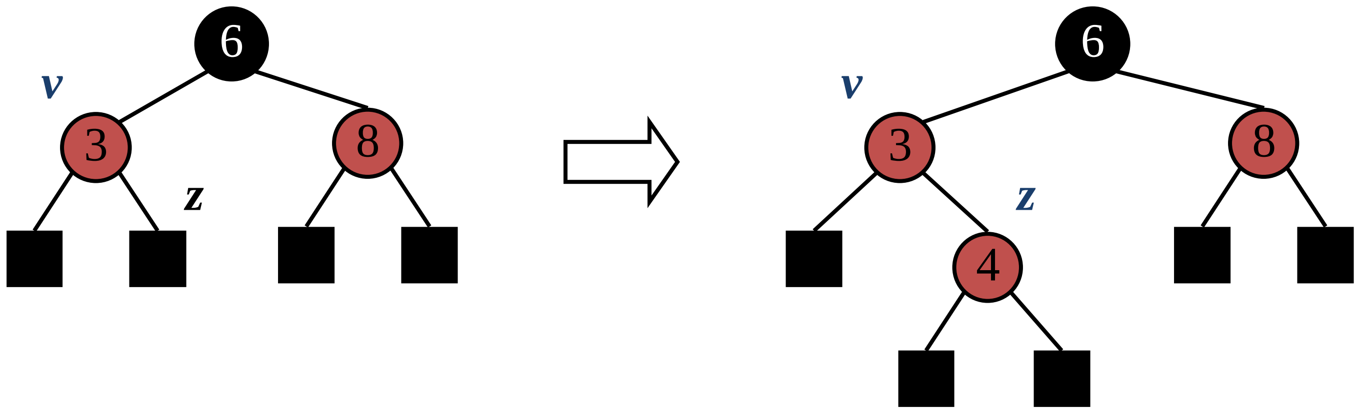
Insertion

- Use insertion algorithm for binary search trees and color red the newly inserted node *z*, unless it's the root.
  - we preserve the root, external, and depth properties
  - **if the parent *v* of *z* is black**, we also preserve the internal property and we are done



Insertion

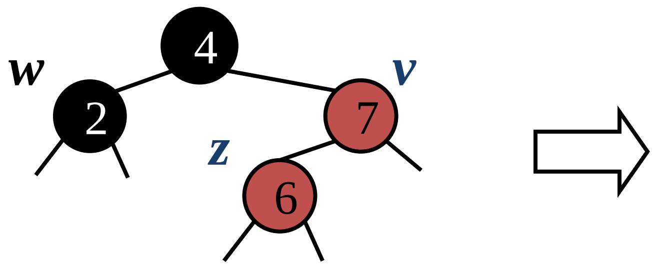
- Use insertion algorithm for binary search trees and color red the newly inserted node *z*, unless it's the root.
  - we preserve the root, external, and depth properties
  - if the parent *v* of *z* is black, we also preserve the internal property and we are done
  - **if the parent *v* of *z* is red**, we have a double red (a violation of the internal property), which requires a reorganization of the tree
- Ex: Insert 4 causes a double red



Fixing a Double Red

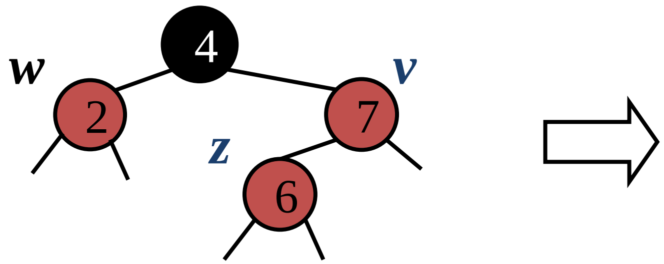
Consider a double red with child ***z*** and parent ***v***, and let ***w*** be the sibling of ***v***

- Case 1: ***w*** is **black**



Restructuring

- Case 2: ***w*** is **red**

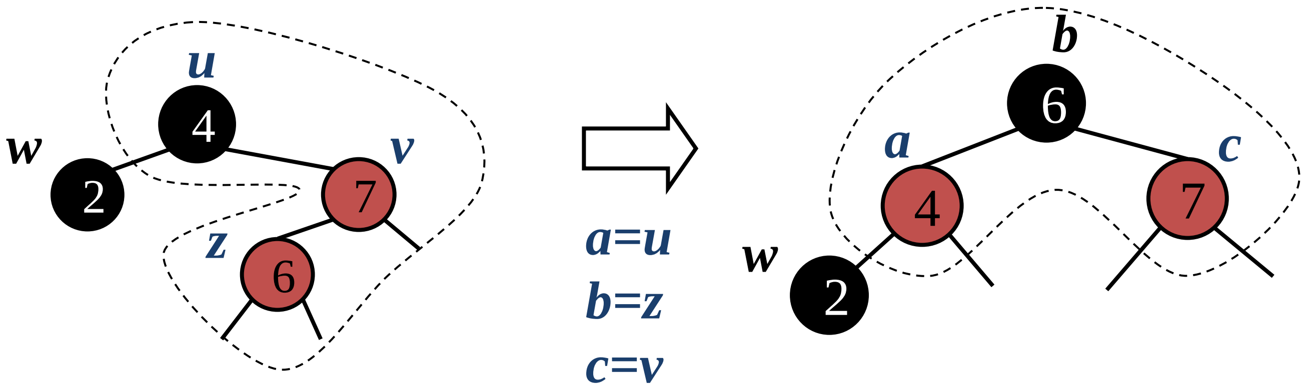


Recoloring

Note: pictures with dangling edges are a visualization of a small portion of larger tree

Restructuring

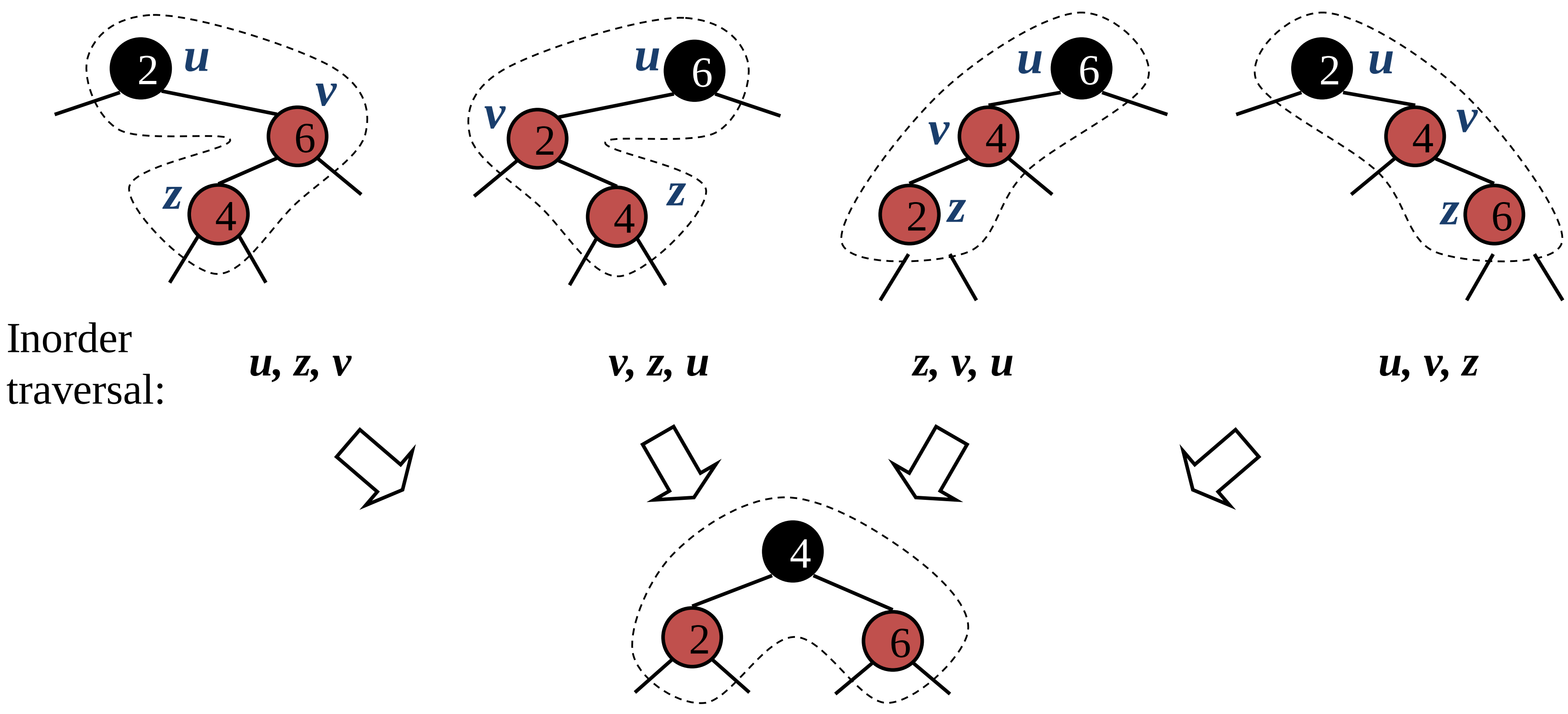
Consider a double red with child ***z*** and parent ***v*** and let ***w*** be the sibling of ***v***. Let ***u*** be the parent of ***v***.



1. Relabel nodes ***z, v, u*** temporarily as ***a, b, c*** so that ***a***, ***b***, ***c*** will be visited in this order by an inorder tree traversal.
2. Replace ***u*** with the node labeled ***b*** (colored **black**). Make nodes ***a*** and ***c*** the left and right child of ***b*** (each colored **red**).

Restructuring

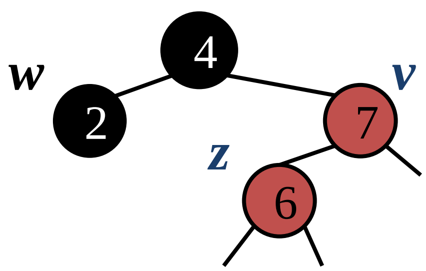
There are four restructuring configurations depending on the in-order traversal of nodes ***z, v, u***



Fixing a Double Red

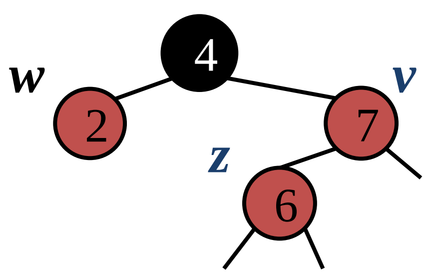
Consider a double red with child ***z*** and parent ***v***, and let ***w*** be the sibling of ***v***

- Case 1: ***w*** is **black**



Restructuring

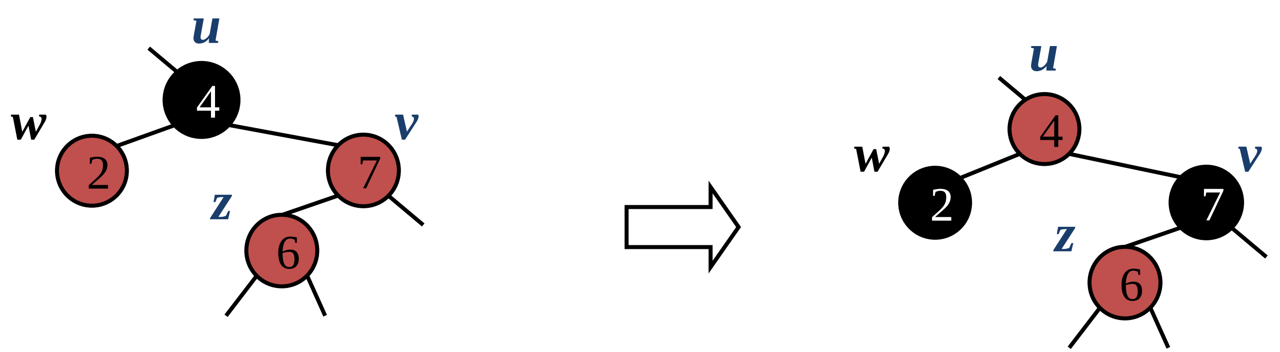
- Case 2: ***w*** is **red**



Recoloring

Recoloring

Consider a double red with child ***z*** and parent ***v***, and let ***w*** be the sibling of ***v***. Let ***u*** be the parent of ***v***.



1. Color ***v*** and ***w*** **black**.
2. Color ***u*** **red**, unless it's the root.
3. If the double-red problem reappears at ***u***, then repeat the process for fixing two reds at ***u*** (either with restructuring or recoloring).

Fixes problem locally, but can propagate double-red problem up the tree.

Analysis of Insertion

Algorithm ***insertItem*(*k*, *o*)**

1. We search for key ***k*** to locate the insertion node ***z***

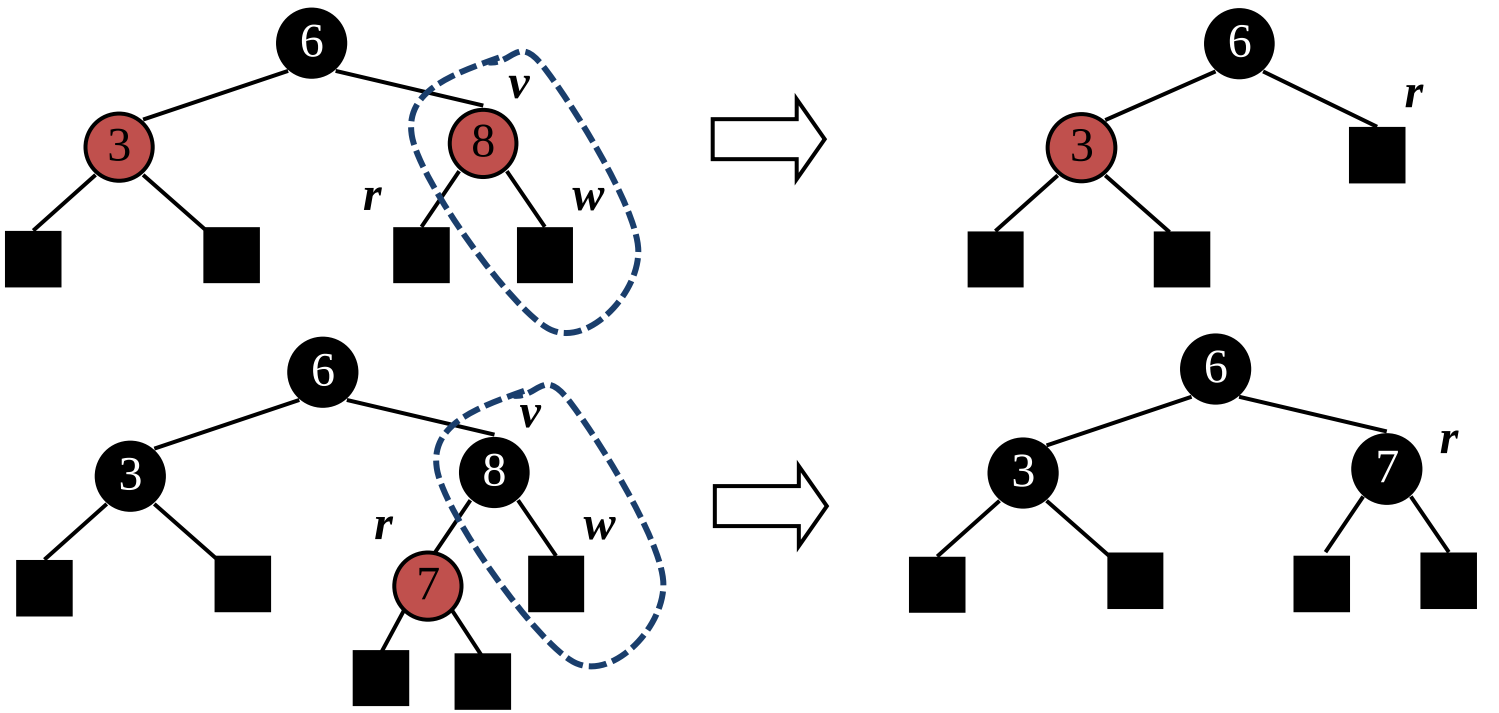
2. We add the new item (***k***, ***o***) at node ***z*** and color ***z*** red

3. **while** ***doubleRed*(*z*)**
   if ***isBlack*(*sibling*(*parent*(*z*)))**
      ***restructure*(*z*)**
      return
   else { ***sibling*(*parent*(*z*)** is red }
      z ← ***recolor*(*z*)**

- Recall that a red-black tree has ***O***(log ***n***) height
- Step 1 takes ***O***(log ***n***) time because we visit ***O***(log ***n***) nodes
- Step 2 takes ***O***(1) time
- Step 3 takes ***O***(log ***n***) time because we perform
  - ***O***(log ***n***) recolorings, each taking ***O***(1) time, and
  - at most one restructuring taking ***O***(1) time
- Thus, an insertion in a red-black tree takes ***O***(log ***n***) time

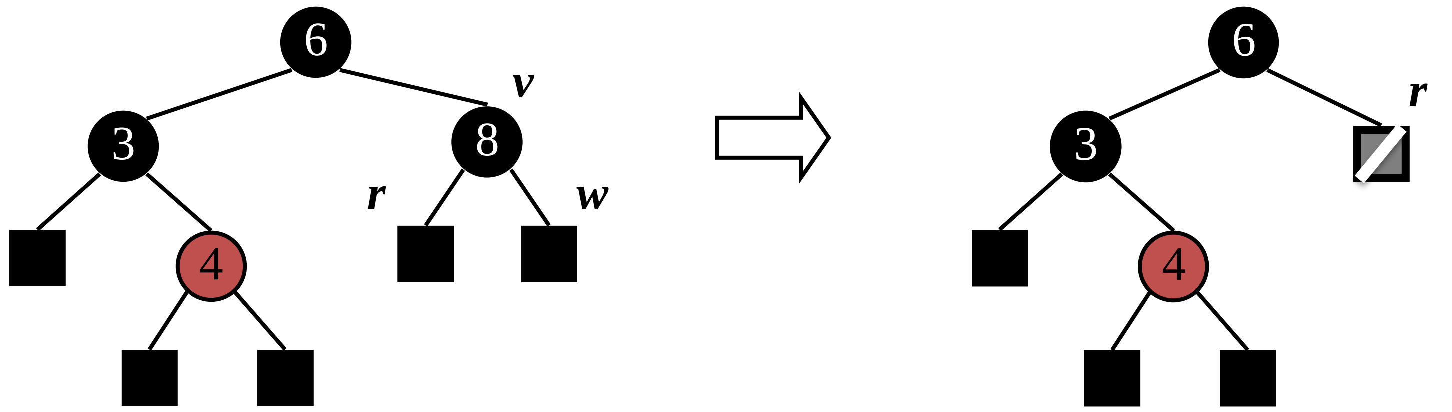
Deletion

- Use deletion algorithm for binary search trees so as to delete internal node ***v*** and its external child ***w***. Let ***r*** be the sibling of ***w.***
  - if ***v*** is red or ***r*** is red, then color ***r*** **black** and we are done.



Deletion

- Use deletion algorithm for binary search trees so as to delete internal node ***v*** and its external child ***w***. Let ***r*** be the sibling of ***w.***
  - if ***v*** is red or ***r*** is red, then color ***r*** **black** and we are done.
  - otherwise (***v*** and ***r*** are black) we color ***r*** **double black,** which requires a reorganization of the tree
- Ex: Delete 8 causes a double black

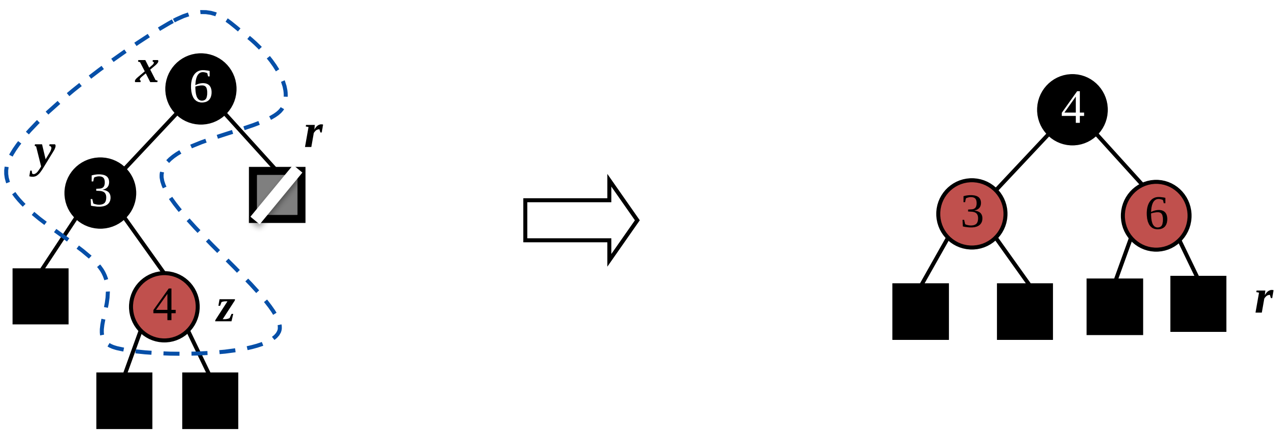


Fixing a Double Black

Let ***y*** be the sibling and ***x*** be the parent of the double black node. The algorithm to fix a double black node considers three cases:

Case 1: ***y*** is black and has a red child ***z***

- We perform a restructuring on ***y, x, z***, and we are done



Fixing a Double Black

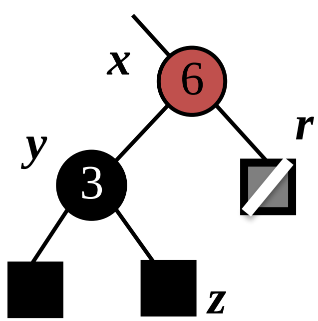
Let ***y*** be the sibling and ***x*** be the parent of the double black node. The algorithm to fix a double black node considers three cases:

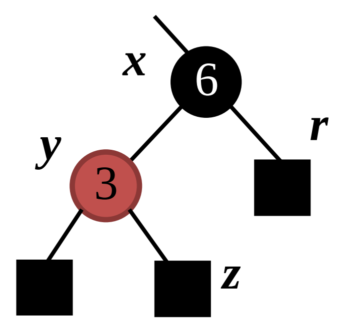
Case 1: ***y*** is black and has a red child ***z***

- We perform a restructuring on ***y, x, z***, and we are done

Case 2: ***y*** is black and its children are both black

- We perform a recoloring. Color ***r*** **black**, and ***y*** **red**.
  - If ***x*** is red, color it **black**. Otherwise, color ***x*** **double-black**.
  - This may propagate up the double black violation





Fixing a Double Black

Let ***y*** be the sibling and ***x*** be the parent of the double black node. The algorithm to fix a double black node considers three cases:

Case 1: ***y*** is black and has a red child ***z***

- We perform a restructuring on ***y, x, z***, and we are done

Case 2: ***y*** is black and its children are both black

- We perform a recoloring. Color ***r*** **black**, and ***y*** **red**.
  - If ***x*** is red, color it **black**. Otherwise, color ***x*** **double-black**.
  - This may propagate up the double black violation

Case 3: ***y*** is red

- We perform an adjustment, after which either Case 1 or Case 2 applies

Deletion in a red-black tree takes ***O***(log ***n***) time.

Red-Black Tree Reorganization

Insertion (fix double red)	result
restructuring	double red removed
recoloring	double red removed or propagated up

Deletion (fix double black)	result
restructuring	double black removed
recoloring	double black removed or propagated up
adjustment	restructuring or recoloring follows