Red Black Trees

CLRS 13.1 – 13.3

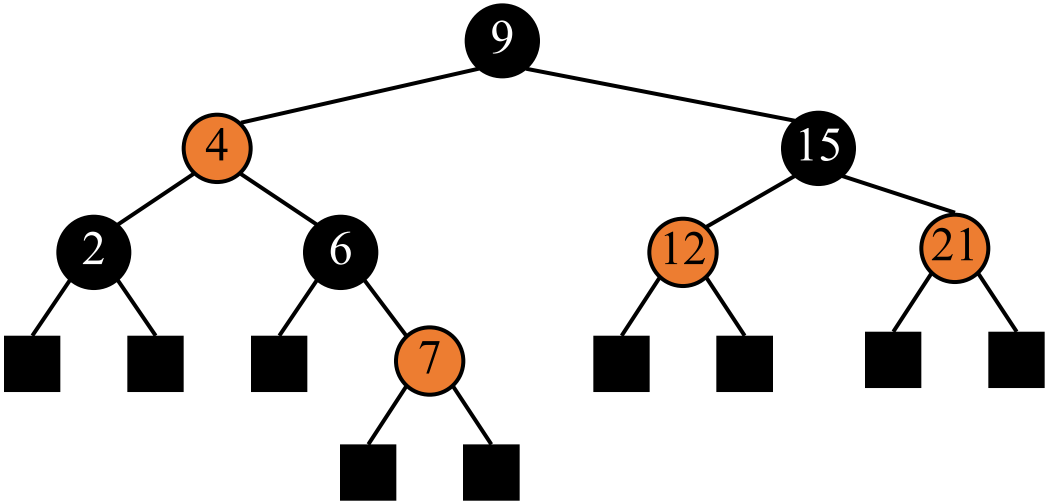
(+ some supplemental material)

includes variation of RBT insert described differently than CLRS

Red Black Tree (RBT)

A **binary search tree** that satisfies the following red-black properties:

1. Every node is either **red** or **black**
2. The **root** is black
3. Every **leaf** (NIL) is black
4. If a node is **red**, then both its **children** are black
5. For each node, all simple paths from the node to the descendant leaves contain the same number of black nodes (i.e., all leaves have the same *black depth)*

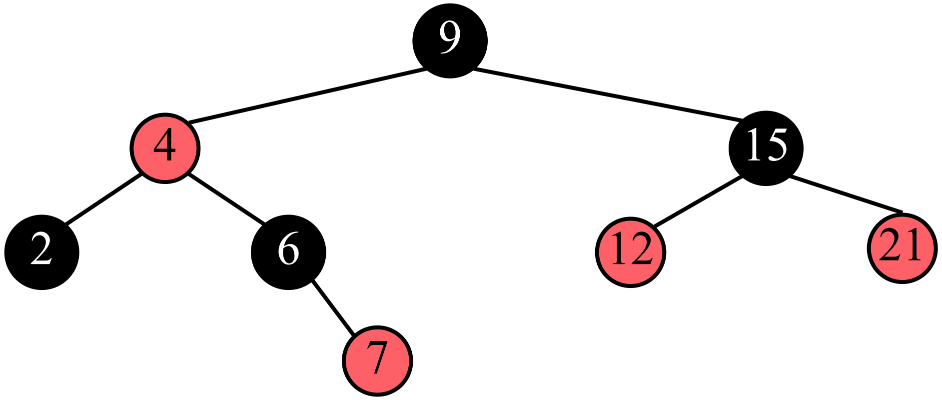


When we visualize, we often omit the black leaves (NILs).

Red Black Tree (RBT)

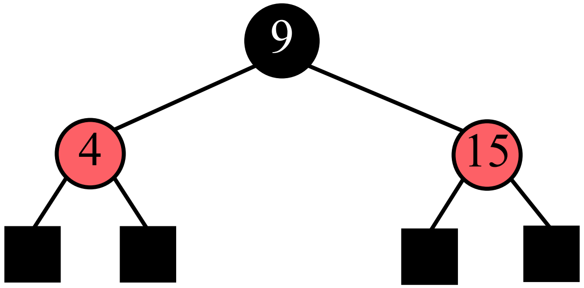
A **binary search tree** that satisfies the following red-black properties:

1. Every node is either **red** or **black**
2. The **root** is black
3. Every **leaf** (NIL) is black
4. If a node is **red**, then both its **children** are black
5. For each node, all simple paths from the node to the descendant leaves contain the same number of black nodes (i.e., all leaves have the same *black depth)*

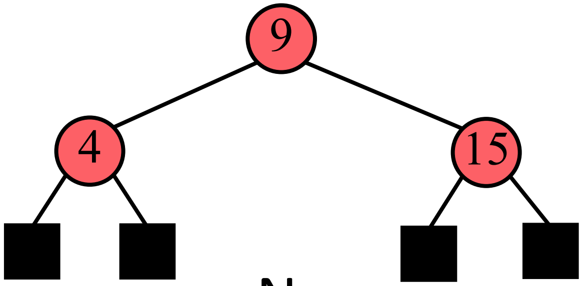


When we visualize, we often omit the black leaves (NILs).

Ex: Is it a red black tree?

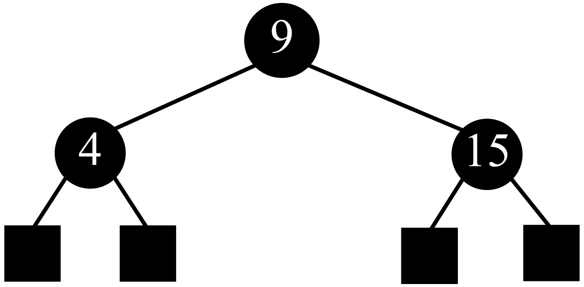


Yes

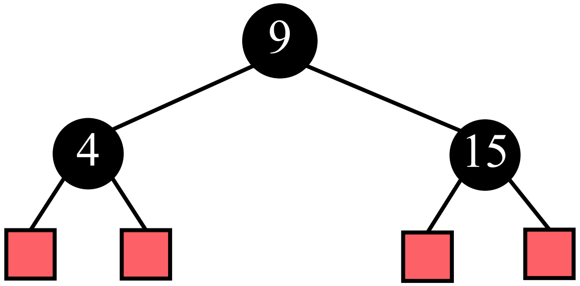


No

Root isn’t black & red node has red child



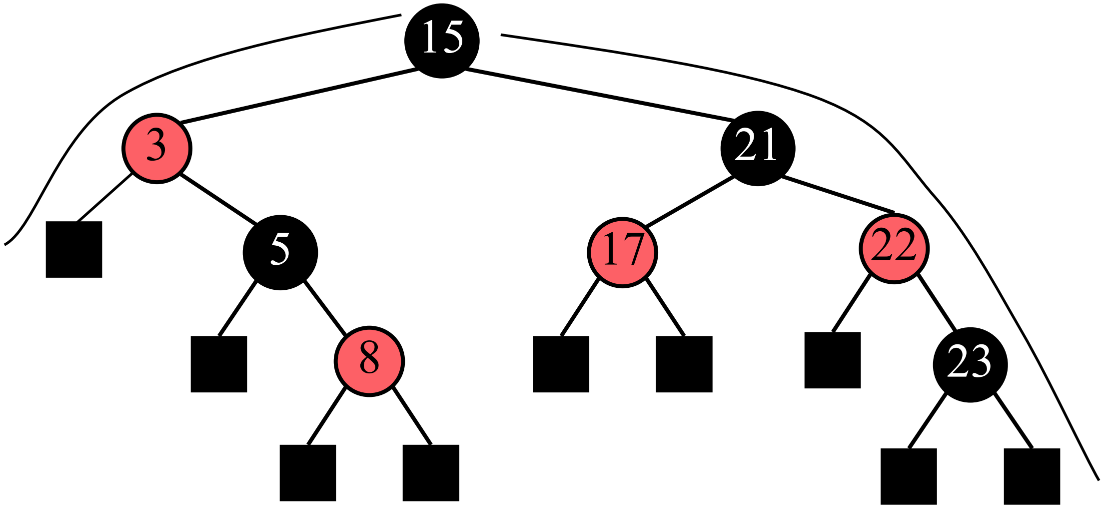
Yes



No

NIL leaves aren’t black

Ex: Is it a red black tree?



Black depth = 2

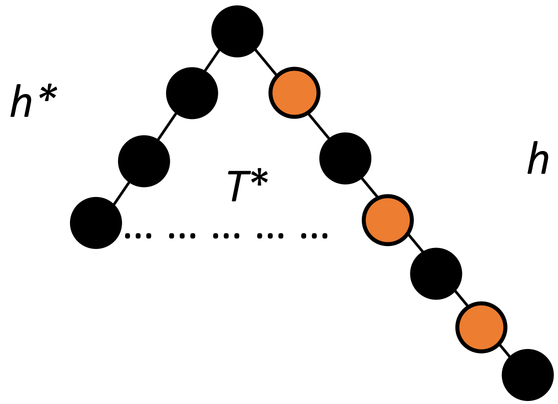
Black depth = 4

No

Violates depth property

Height of a Red Black Tree

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



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 $\le h^*$

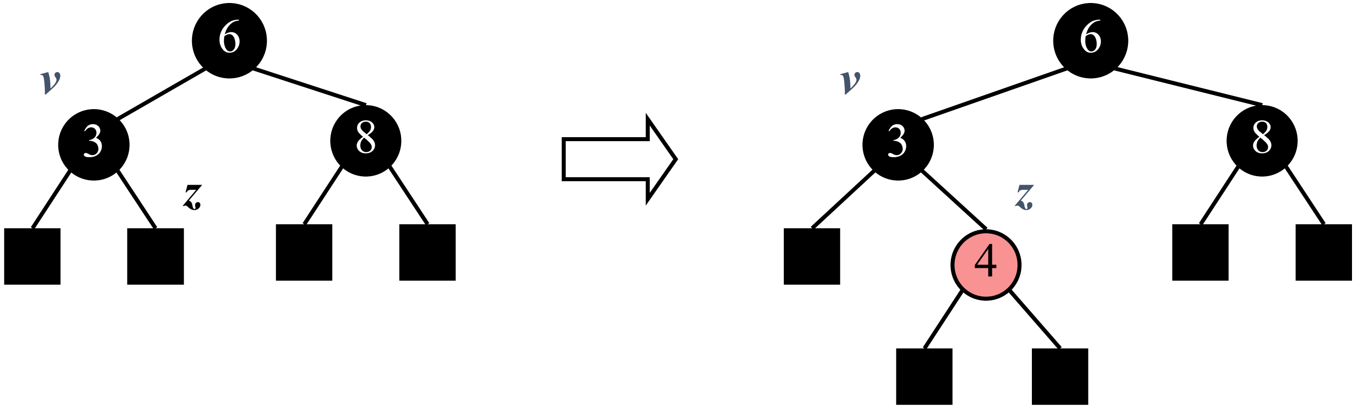
T^* is complete. Thus, $h^* \le \log n$.

Because $h \le 2h^*$, $h \le 2\log n \in O(\log n)$.

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

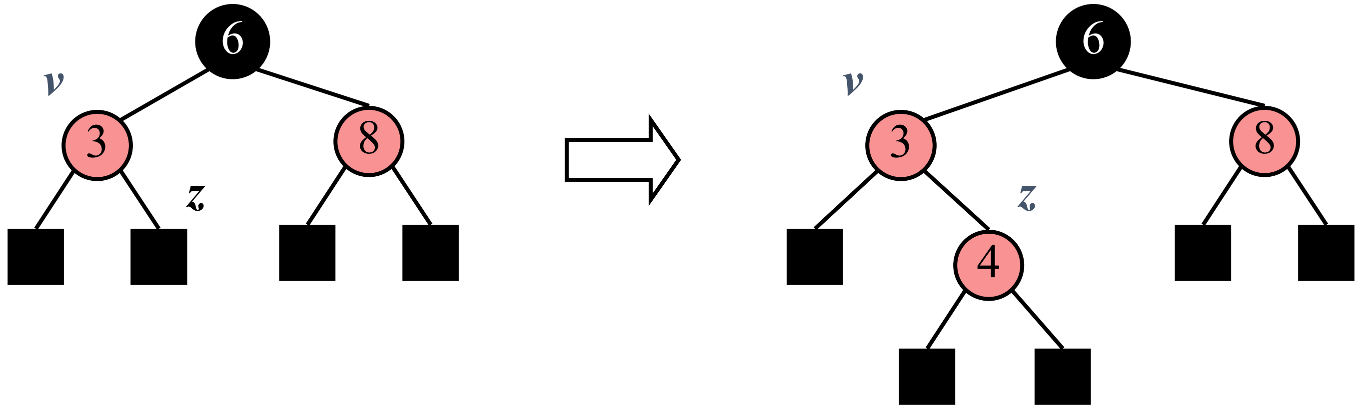
RBT - Insert

- Use insertion algorithm for binary search trees and color **red** the newly inserted node *z*, unless it’s the root.
  - we preserve properties 1, 2, 3, 5.
  - **if the parent *v* of *z* is black**, we also preserve property 4 and we are done



RBT - Insert

- Use insertion algorithm for binary search trees and color **red** the newly inserted node *z*, unless it's the root.
  - we preserve properties 1, 2, 3, 5.
  - if the parent *v* of *z* is black, we also preserve property 4 and we are done
  - **if the parent *v* of *z* is red**, we have a **double red** (a violation of property 4), which requires a reorganization of the tree
- Ex: Insert 4 causes a double red

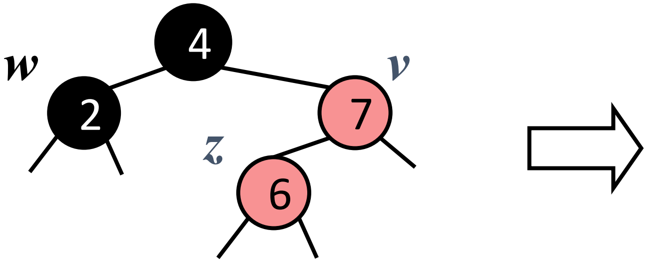


Not a valid RBT!

Fixing a double red

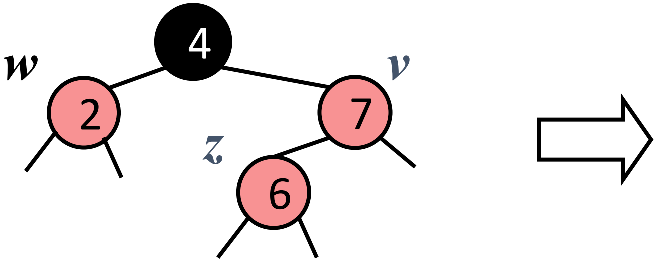
Consider a double red with child ***z*** and parent ***v***, and let ***w*** be the sibling of ***v***

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



Restructuring

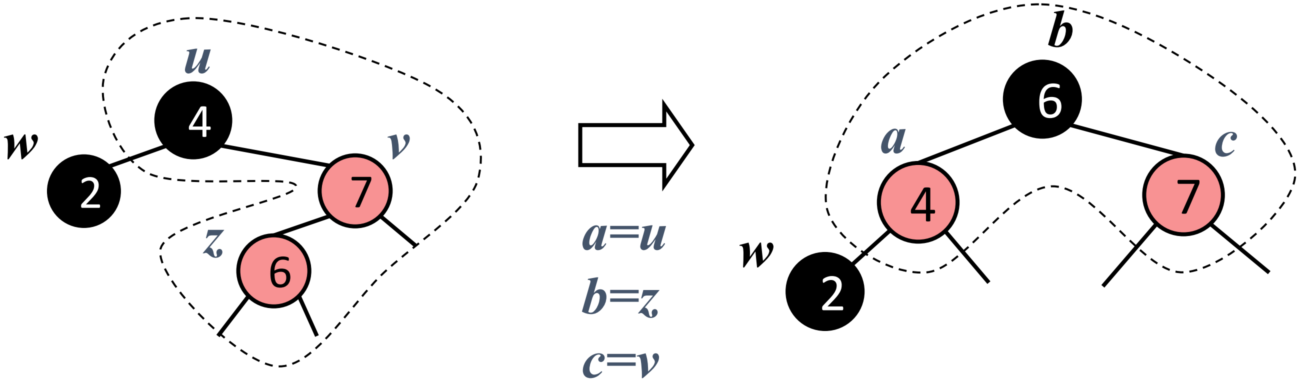
- Case 2: ***w*** is **red**



Recoloring

Fixing a double red: restructuring

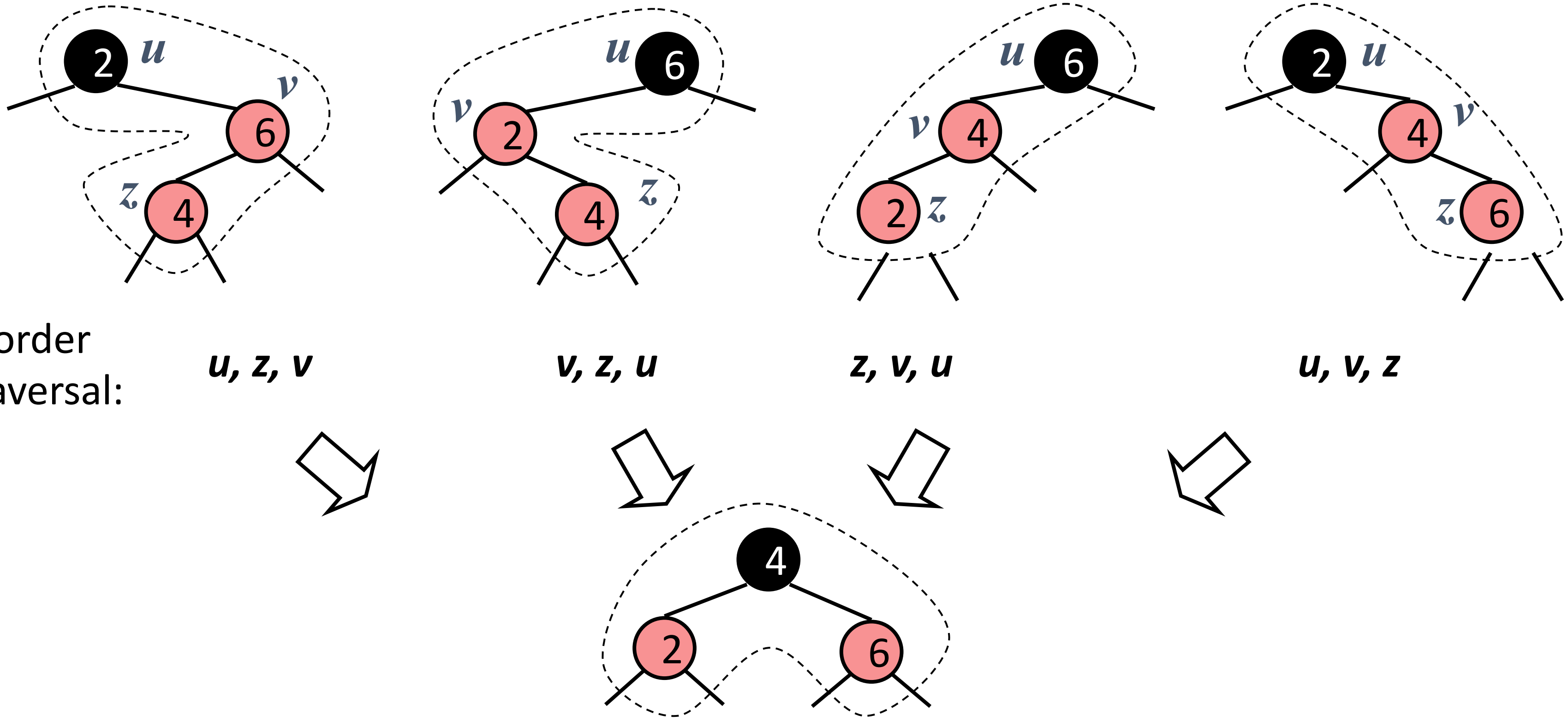
Consider a double red with child ***z*** and parent ***v*** and let ***w*** be the sibling of ***v*.** Let ***u*** be the parent of ***v***.



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**)**.**

Fixing a double red: restructuring

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



Inorder traversal:

u, z, v

v, z, u

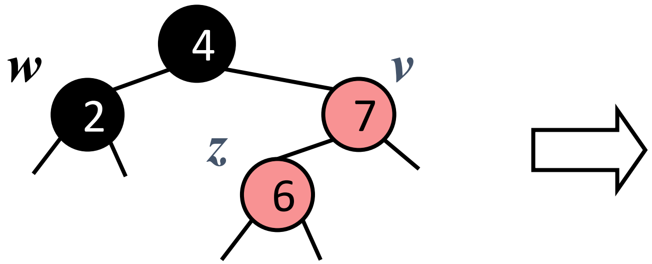
z, v, u

u, v, z

Fixing a double red

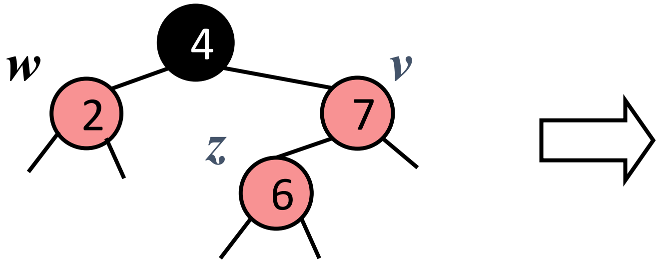
Consider a double red with child ***z*** and parent ***v***, and let ***w*** be the sibling of ***v***

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



Restructuring

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

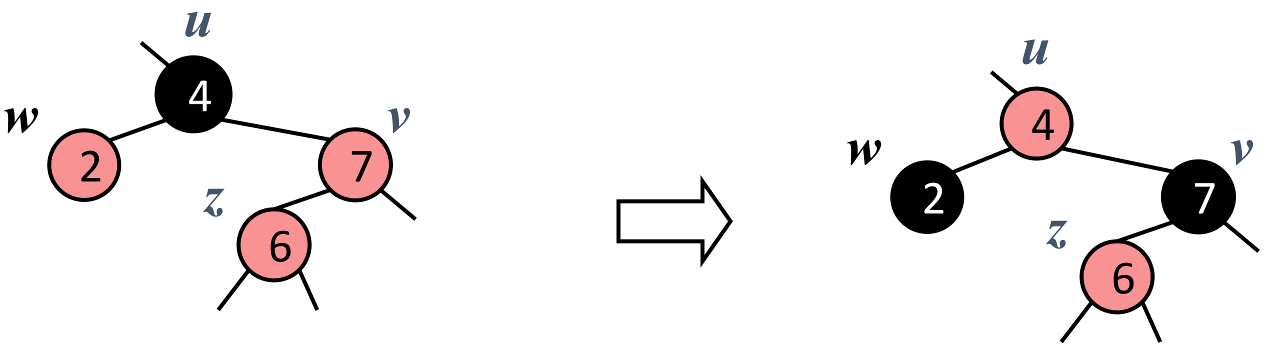


Recoloring

Fixing a double red: recoloring

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

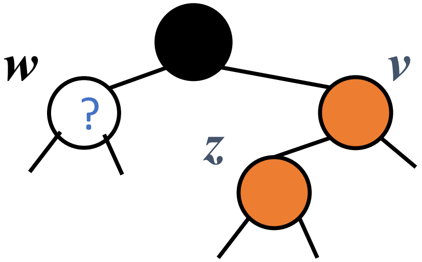
Let ***u*** be the parent of ***v***.



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 insert into RBT



Description:

1. Search for ***k*** to locate the insertion node ***z***
2. Add the new item ***k*** at node ***z*** and color ***z*** red
3. While ***z*** and its parent ***v*** form a double red:
    - If sibling ***w*** of ***v*** is black, do a restructuring once, and we are done
    - If sibling ***w*** of v is red, do a recoloring, and set z to be the parent of u

Analysis:

Recall that RBT has $O(\log n)$ height.

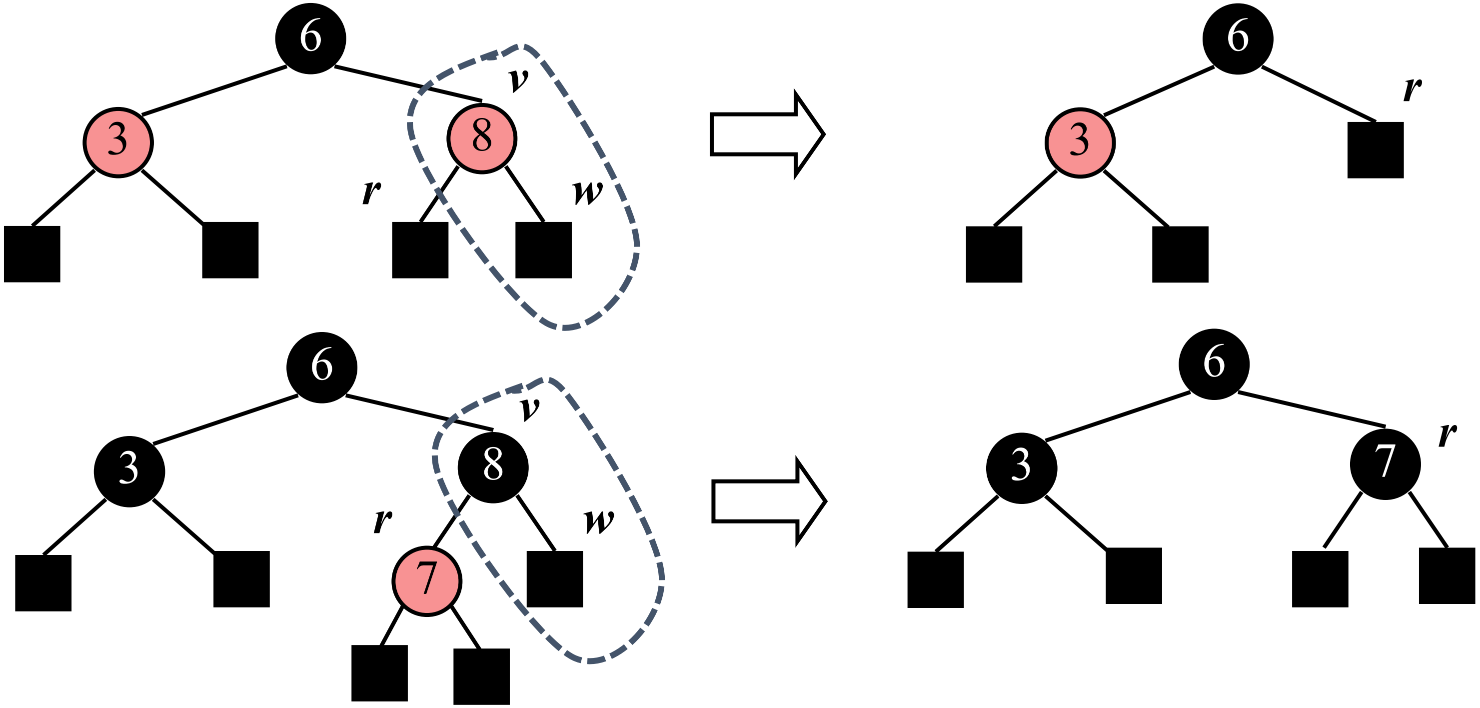
- Searching runs in $O(\log n)$
- Adding a new red node is $O(1)$
- A single restructuring or recoloring is $O(1)$
- While loop repeats at most $O(\log n)$

Total run time for insert: $O(\log n)$

RBT - Delete

Use deletion algorithm for binary search trees to delete internal node ***v*** and its external child ***w***. Let ***r*** be the sibling of ***w.***

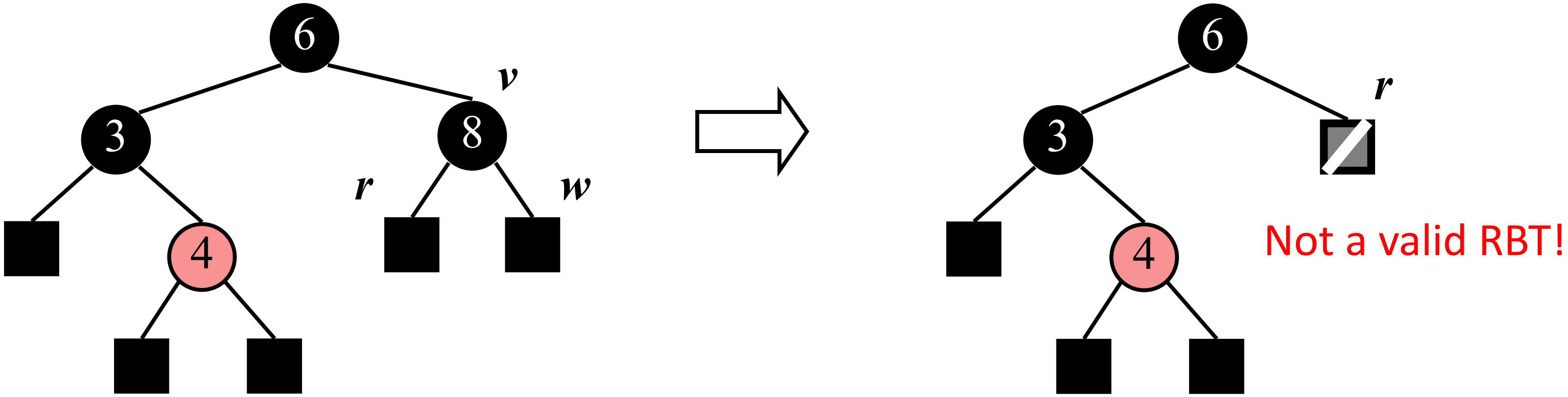
- if ***v*** is **red** or ***r*** is **red**, then color ***r*** **black** and we are done.



RBT - Delete

Use deletion algorithm for binary search trees 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



Not a valid RBT!

How to fix a double black? It's like fixing a double red… requires a recoloring, restructuring, or "adjustment"

- We can delete in $O(\log n)$ time