

04-630

Data Structures and Algorithms for Engineers

Lecture 13: Height-Balanced Trees: Red Black Trees

Height-balanced Trees

Red-Black Trees

- The goal of **height-balancing** is to ensure that the tree is as **complete** as possible and that, consequently, it has **minimal height for the number of nodes in the tree**
- As a result, the **number of probes it takes to search the tree** (and the time it takes) **is minimized**.

Red-Black Trees

- A **perfect or a complete tree** with n nodes has height $O(\log_2 n)$
 - So the time it takes to search a perfect or a complete tree with n nodes is $O(\log_2 n)$
- A **skinny tree** could have height $O(n)$
 - So the time it takes to search a skinny tree can be $O(n)$
- Red-Black trees are similar to AVL trees in that they allow us to construct trees which have a guaranteed search time $O(\log_2 n)$

Red-Black Trees

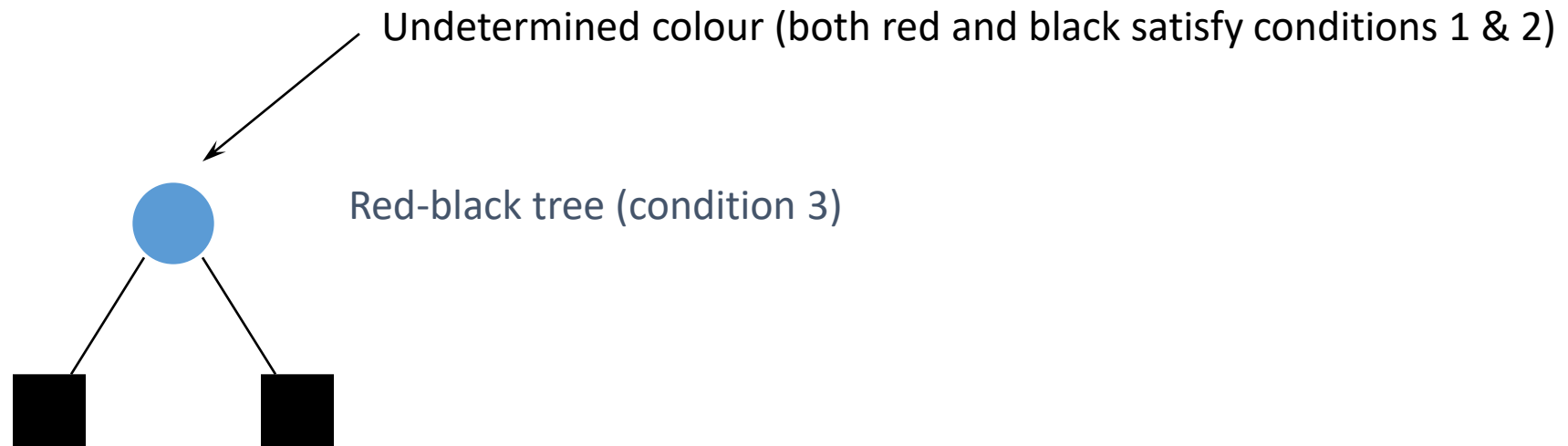
- A red-black tree is a binary tree whose nodes can be coloured either red or black to satisfy the following conditions:
 1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
 2. **Red condition:** *Each red node* that is not the root *has a black parent* (If a node is red, both children are black)
 3. Each external node (leaf node) is **black**.
 4. Every node must have one color –**red** or **black**.
 5. Root node is **black**.

Red-Black Trees



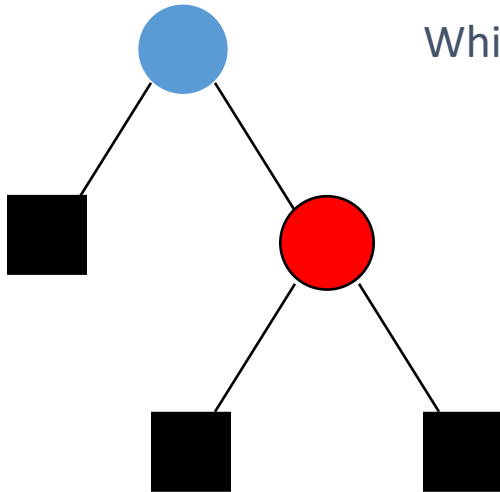
Red-black tree (condition 3)

Red-Black Trees



1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** *Each red node that is not the root has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color – **red** or **black**.
5. Root node is **black**.

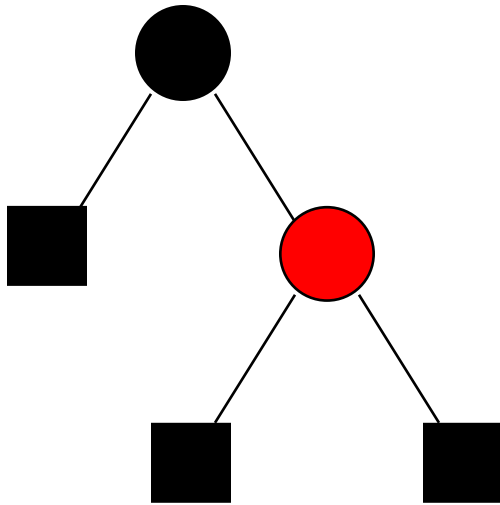
Red-Black Trees



Which color would you assign to the root to make the tree a red-black tree?

1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each red node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color – **red or black**.
5. Root node is **black**.

Red-Black Trees

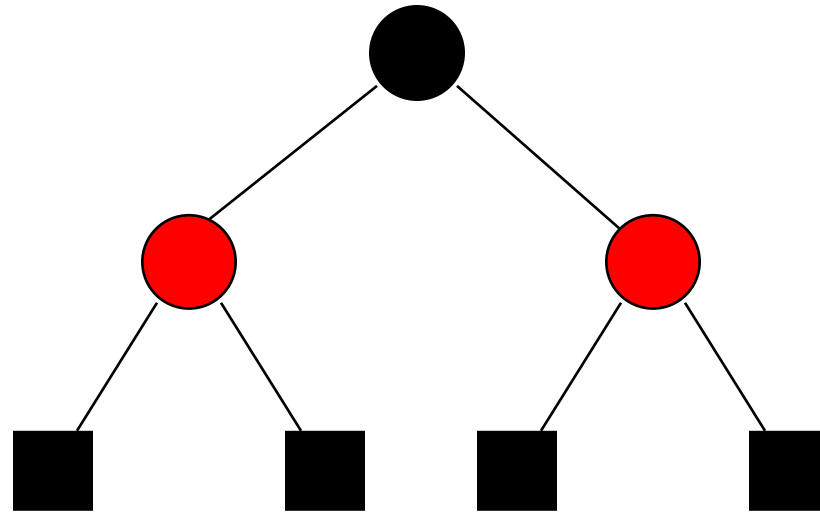


If root was **red**, then right child would have to be **black** (because if it was red, by Condition 2 it would have to have a black parent) but then Condition 1, the black condition, would be violated ... so the root **can't be red** in this case.

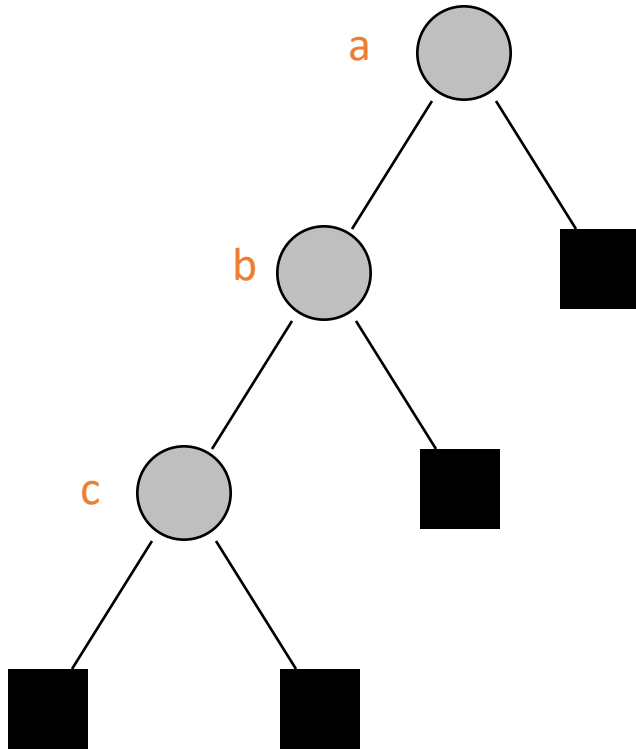
1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each red node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color —**red** or **black**.
5. Root node is **black**.

Red-Black Trees

1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each **red** node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color – **red** or **black**.
5. Root node is **black**.

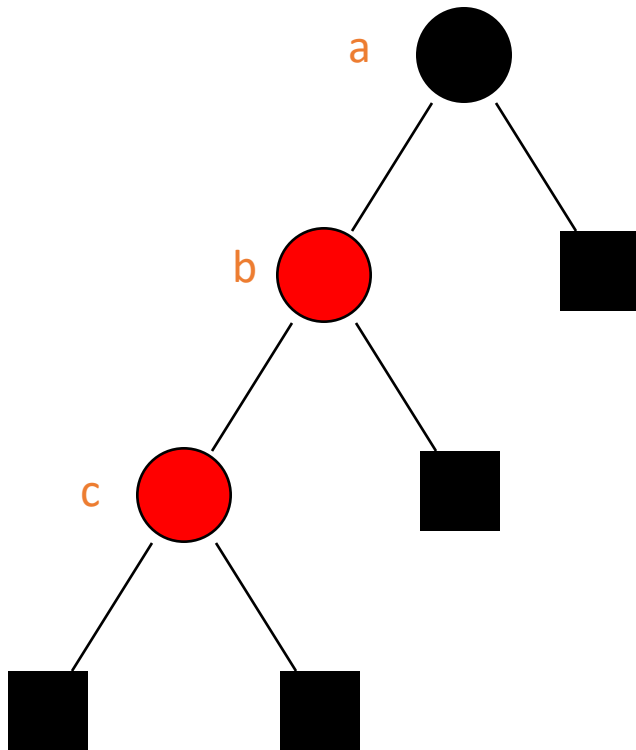


Red-Black Trees



Consider the tree shown. Can the tree be colored to make it a red black tree?

Red-Black Trees



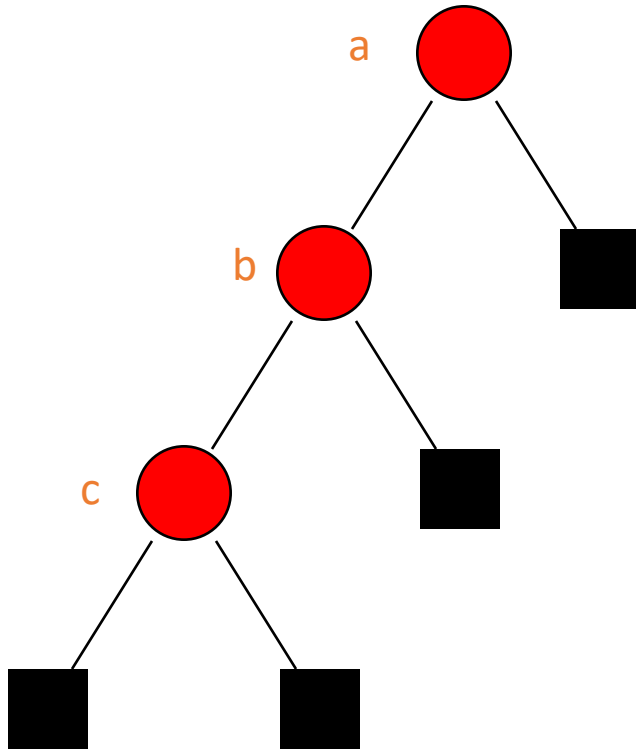
To satisfy **black condition**, either

(1) node a is black and nodes b and c are red, or

(2) nodes a, b, and c are red.

Black condition: Each root-to-frontier path contains exactly the *same number of black nodes*.

Red-Black Trees



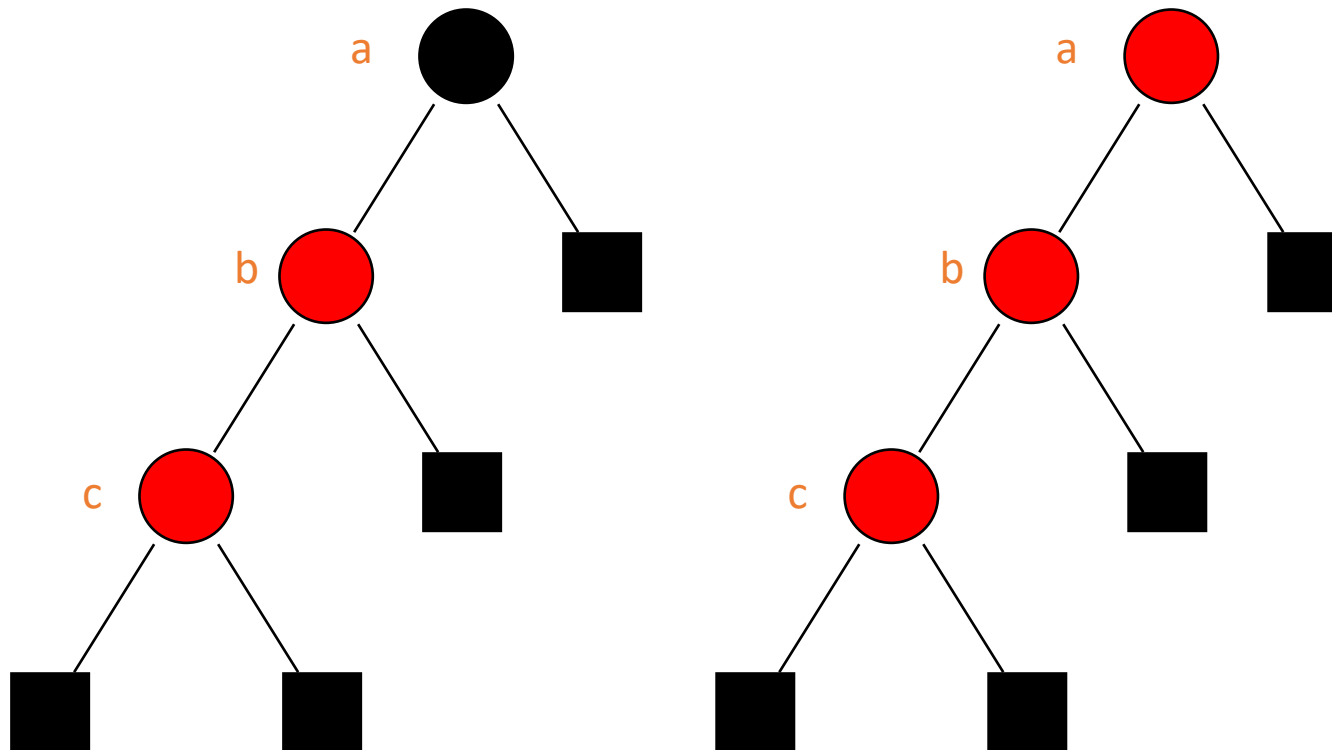
To satisfy **black condition**, either

(1) node a is black and nodes b and c are red, or

(2) nodes a, b, and c are red.

Black condition: Each root-to-frontier path contains exactly the *same number of black nodes*.

Red-Black Trees



To satisfy **black condition**, either

(1) node a is black and nodes b and c are red, or

(2) nodes a, b, and c are red.

In both cases, a **red condition is violated**.

Therefore, this is not a red-black tree (i.e., it cannot be coloured in a way that satisfies all three conditions)

Red condition: *Each red node that is not the root has a black parent*

Red-Black Trees

- For all $n \geq 1$, every red-black tree of size n has height $O(\log_2 n)$
- Thus, red-black trees provide a guaranteed worst-case search time of $O(\log_2 n)$

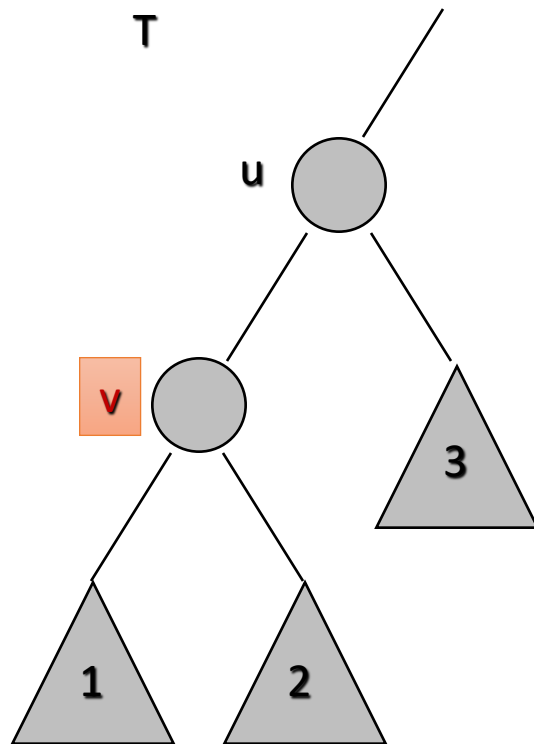
Red-Black Trees

- Insertions and deletions can cause red and black conditions to be violated
- Trees then have to be restructured
- Restructuring is called a promotion (or rotation)
 - Single promotion
 - 2 promotion

Red-Black Trees

- Single promotion
- Also referred to as
 - single (left) rotation
 - single (right) rotation
- Promotes a node one level

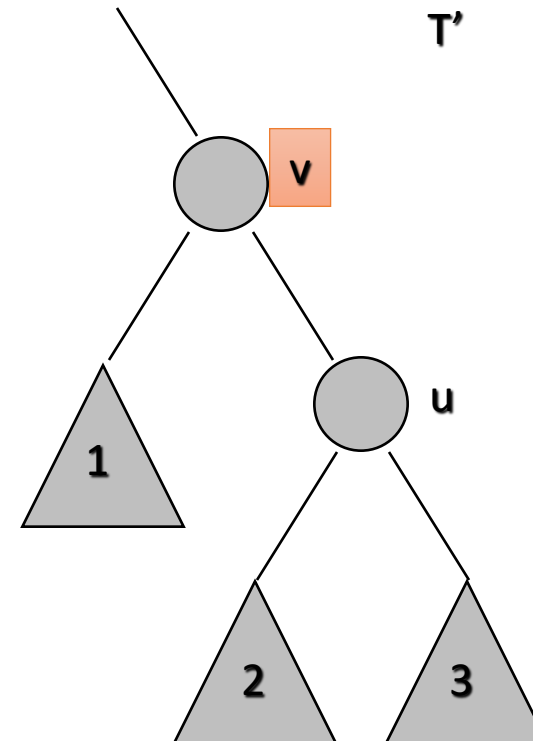
Red-Black Trees



Promote **v**
(Right Rotation)



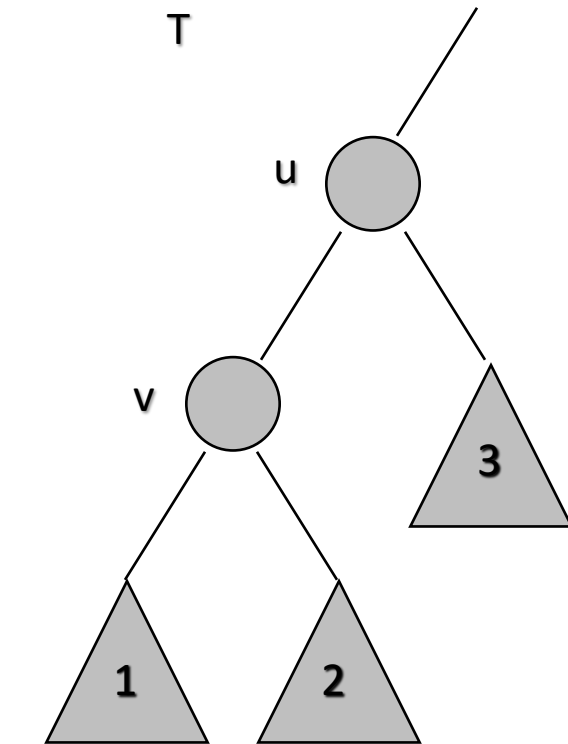
Promote **u**
(Left Rotation)



Red-Black Trees

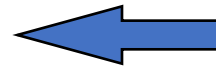
- A single promotion (Left Rotation or Right Rotation) preserves the binary-search condition
- Same manner as an AVL rotation (*earlier*)

Red-Black Trees

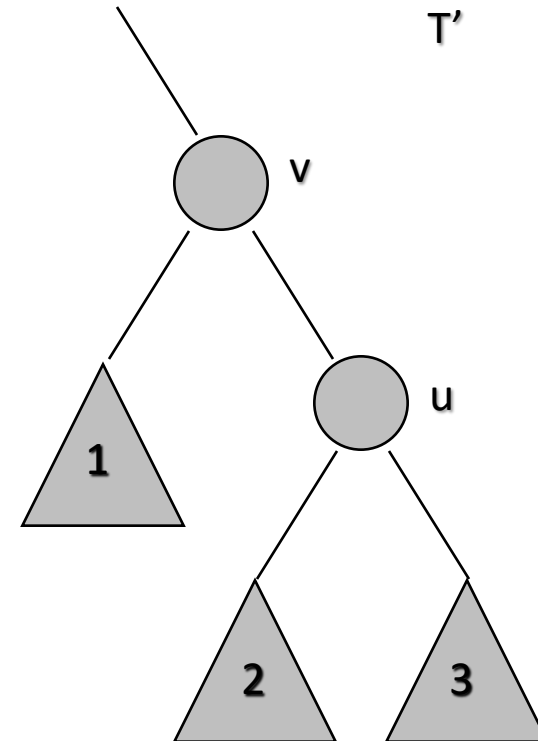


$\text{keys}(1) < \text{key}(v) < \text{key}(u)$
 $\text{key}(v) < \text{keys}(2) < \text{key}(u)$
 $\text{key}(u) < \text{keys}(3)$

Promote v
(Right Rotation)



Promote u
(Left Rotation)



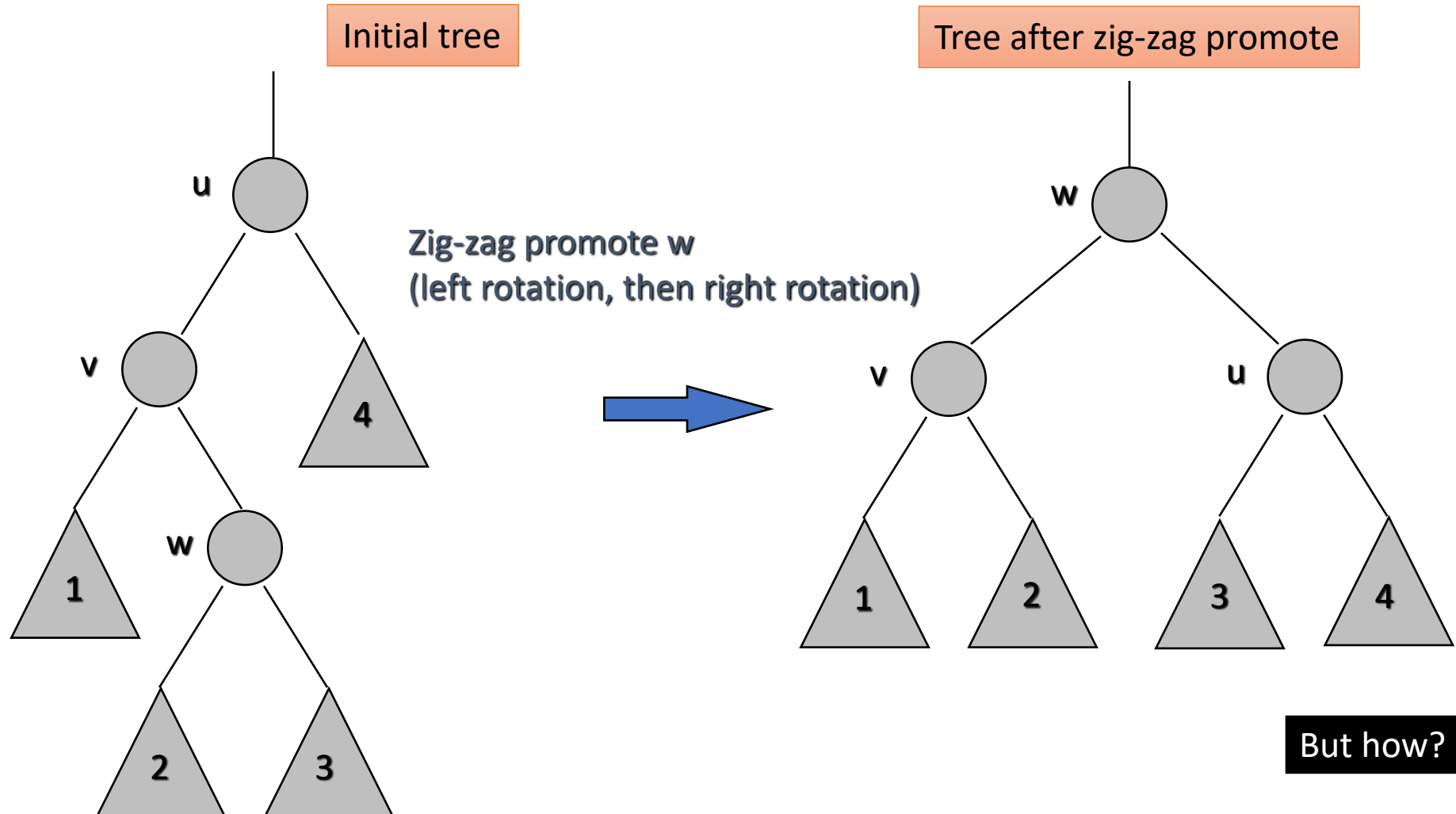
$\text{keys}(1) < \text{key}(v)$
 $\text{key}(v) < \text{keys}(2) < \text{key}(u)$
 $\text{key}(v) < \text{key}(u) < \text{keys}(3)$

preserves the binary-search condition

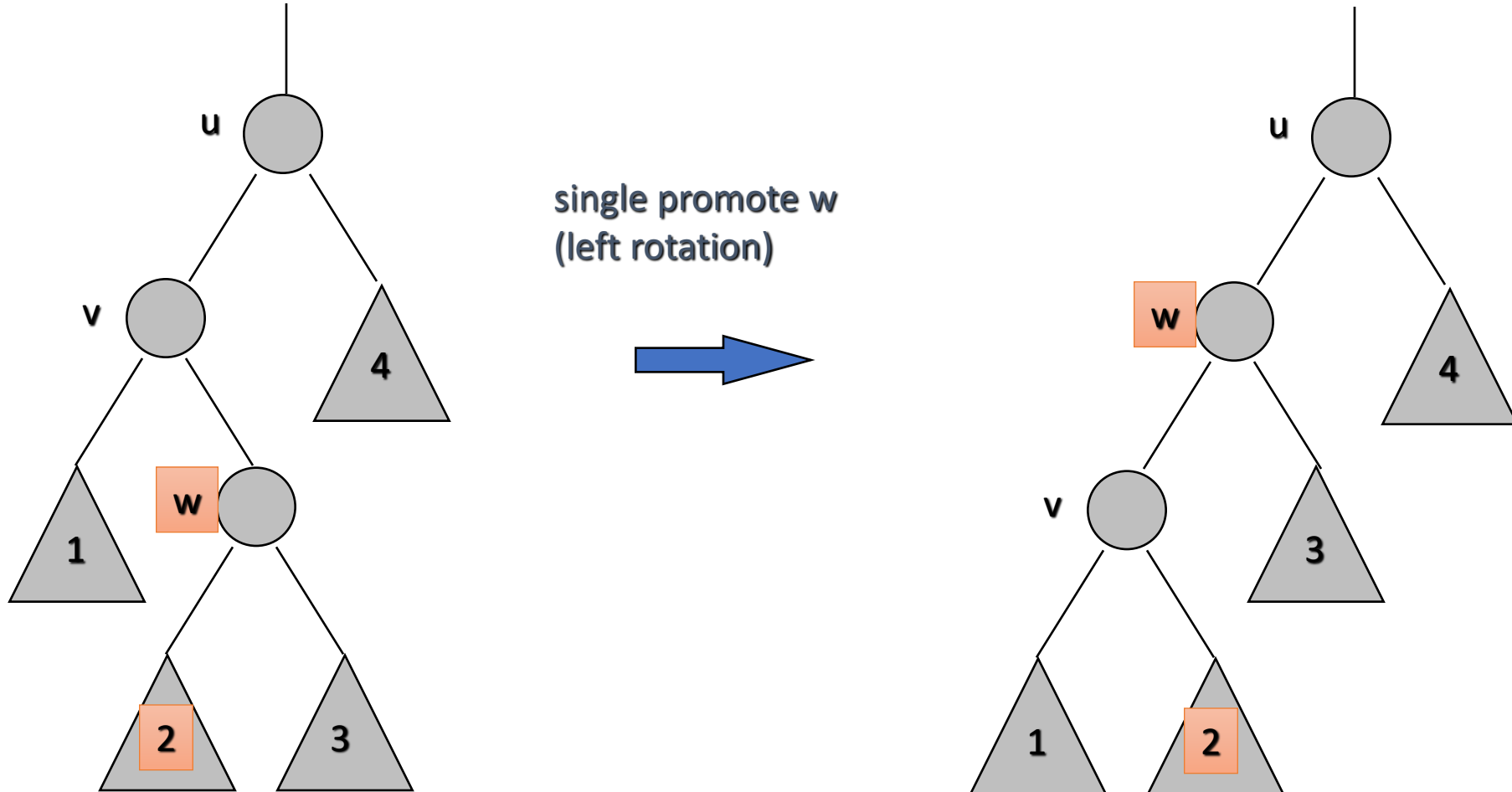
Red-Black Trees

- 2-Promotion
- Zig-zag promotion
- Composed of two single promotions
- And hence preserves the binary-search condition

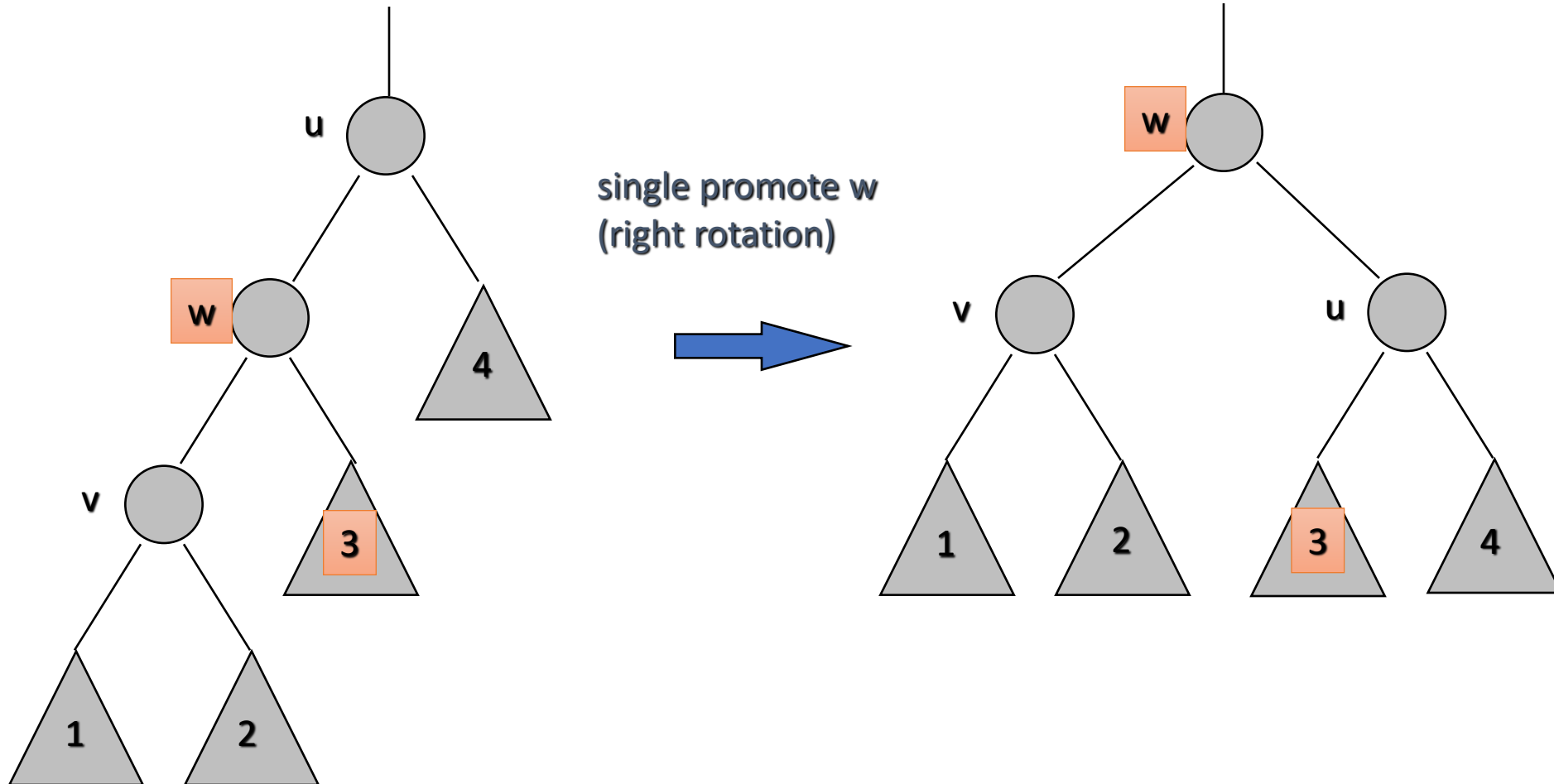
Red-Black Trees



Red-Black Trees: zig-zag promote—left rotate (Step 1)

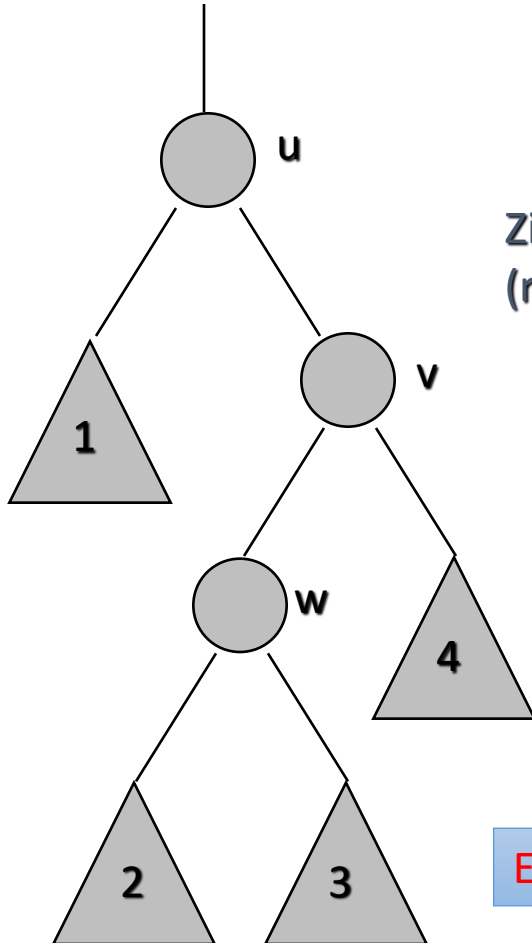


Red-Black Trees: zig-zag promote—right rotate (Step 2)

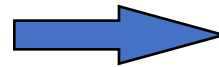


Red-Black Trees

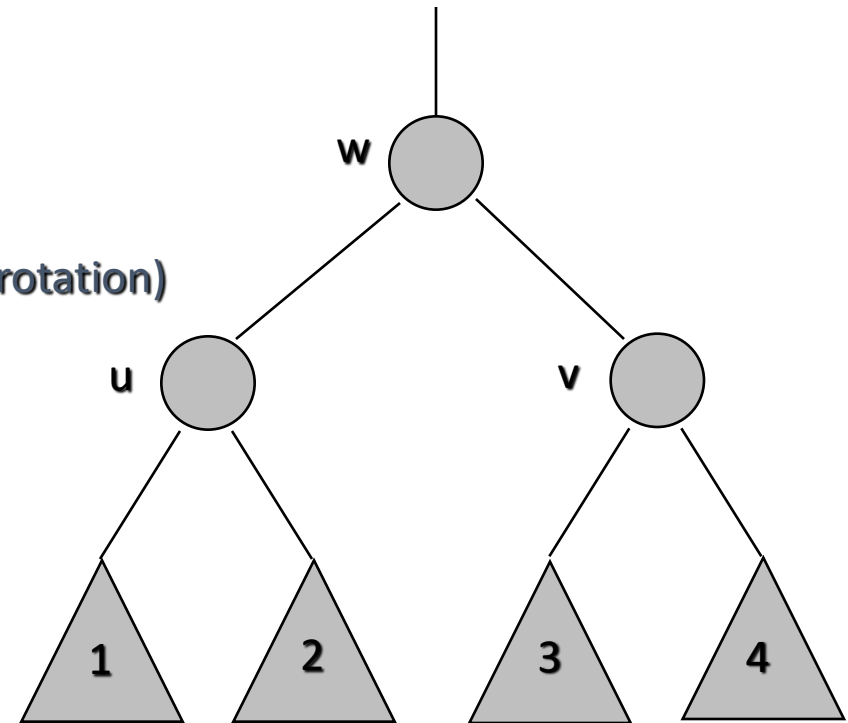
Initial tree



Zig-zag promote w
(right rotation, then left rotation)



Tree after zig-zag promote



Exercise: Show the sequence of right rotation, left rotation steps.

Red-Black Trees

Insertions

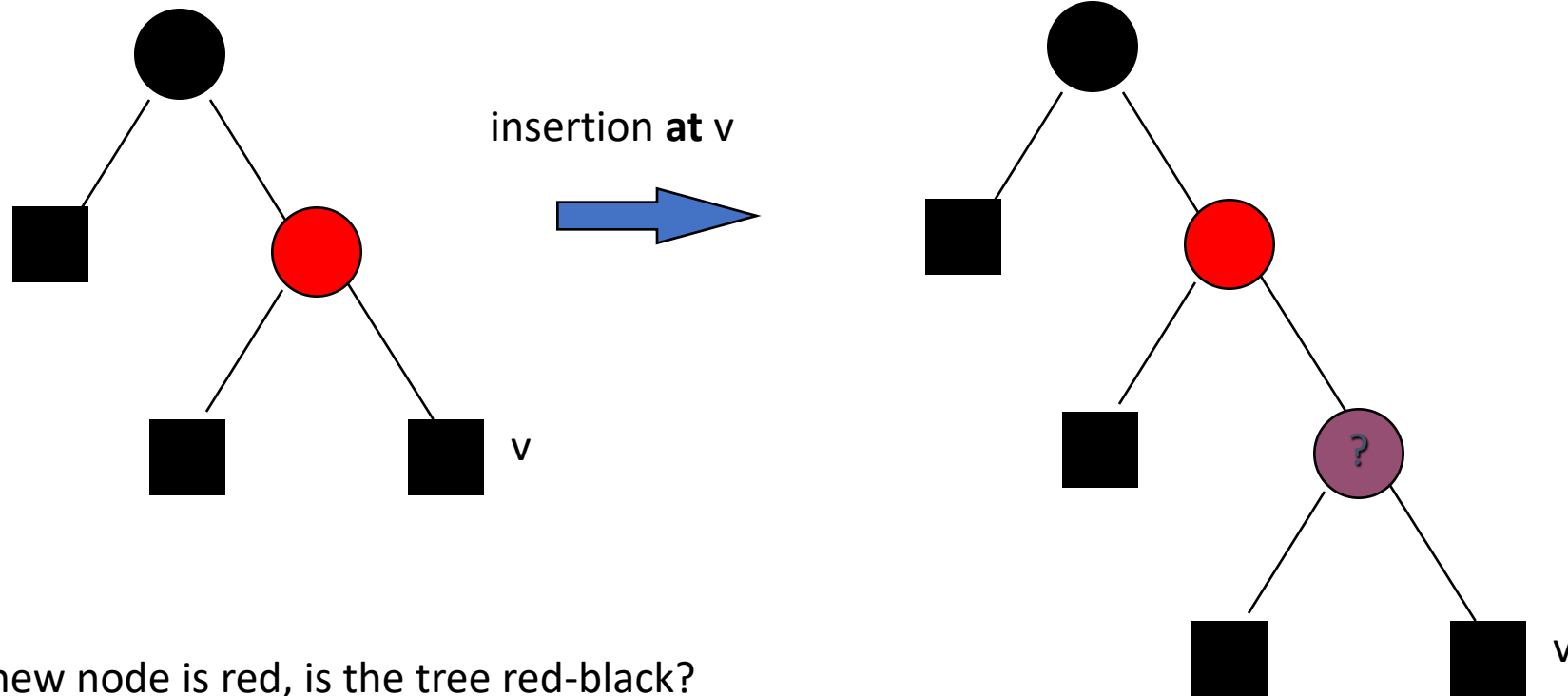
- A red-black tree can be searched in **logarithmic time**, i.e., $O(\log n)$, worst case
- **Insertions may violate the red-black conditions necessitating restructuring**
- This restructuring can also be performed in **logarithmic time**, i.e., $O(\log n)$.
- Thus, an insertion (or a deletion) can be performed in **logarithmic time**, i.e., $O(\log n)$.

Operation	Complexity
search(key)	$O(\log n)$
insert(T, key)	$O(\log n)$
restructure(T)	$O(\log n)$
delete(T, key)	$O(\log n)$

Red-Black Trees

- Just as with AVL trees (*earlier*), we perform the insertion by
 - first searching the tree until an **external** node is reached (if the key is not already in the tree)
 - then inserting the new (**internal**) node
- We then have to **recolour** and **restructure, if necessary**.

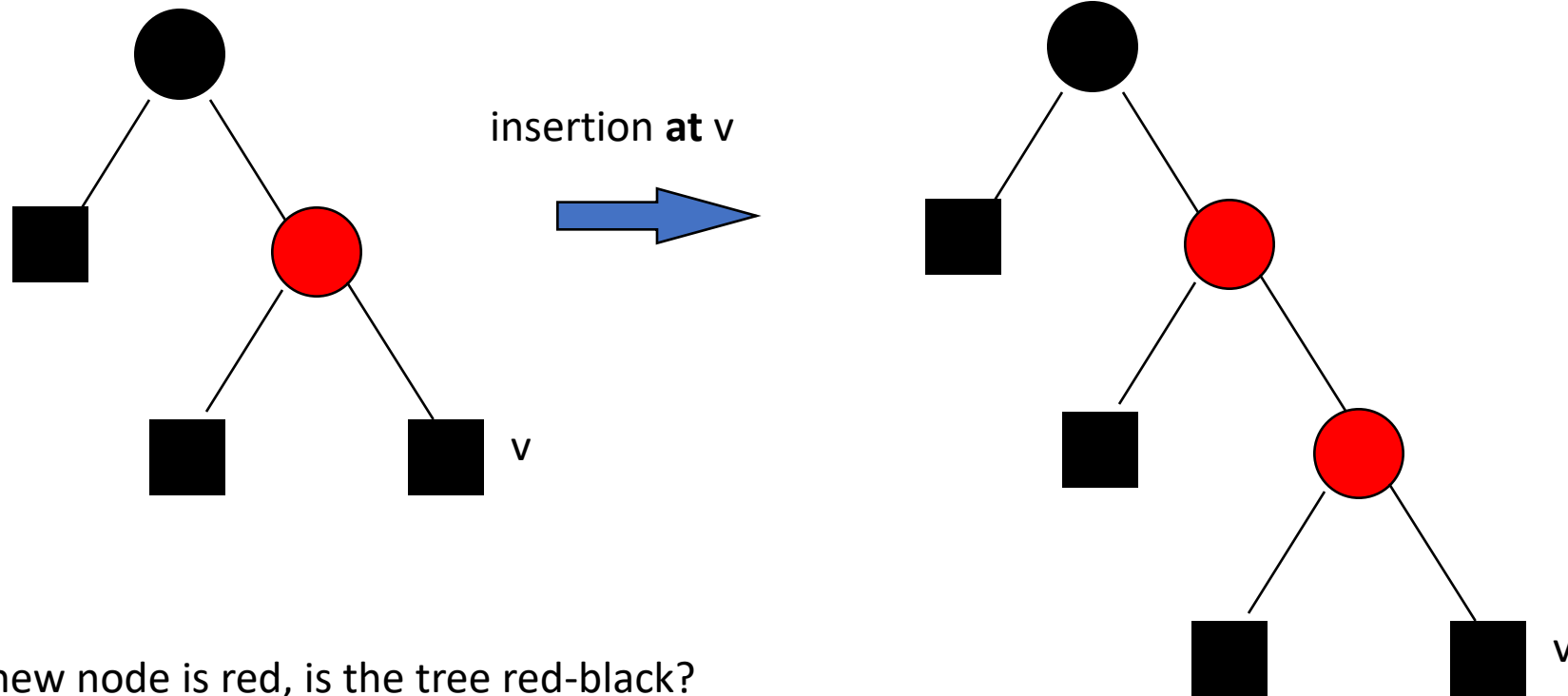
Red-Black Trees



If new node is red, is the tree red-black?

If the new node is black, is the tree red-black?

Red-Black Trees

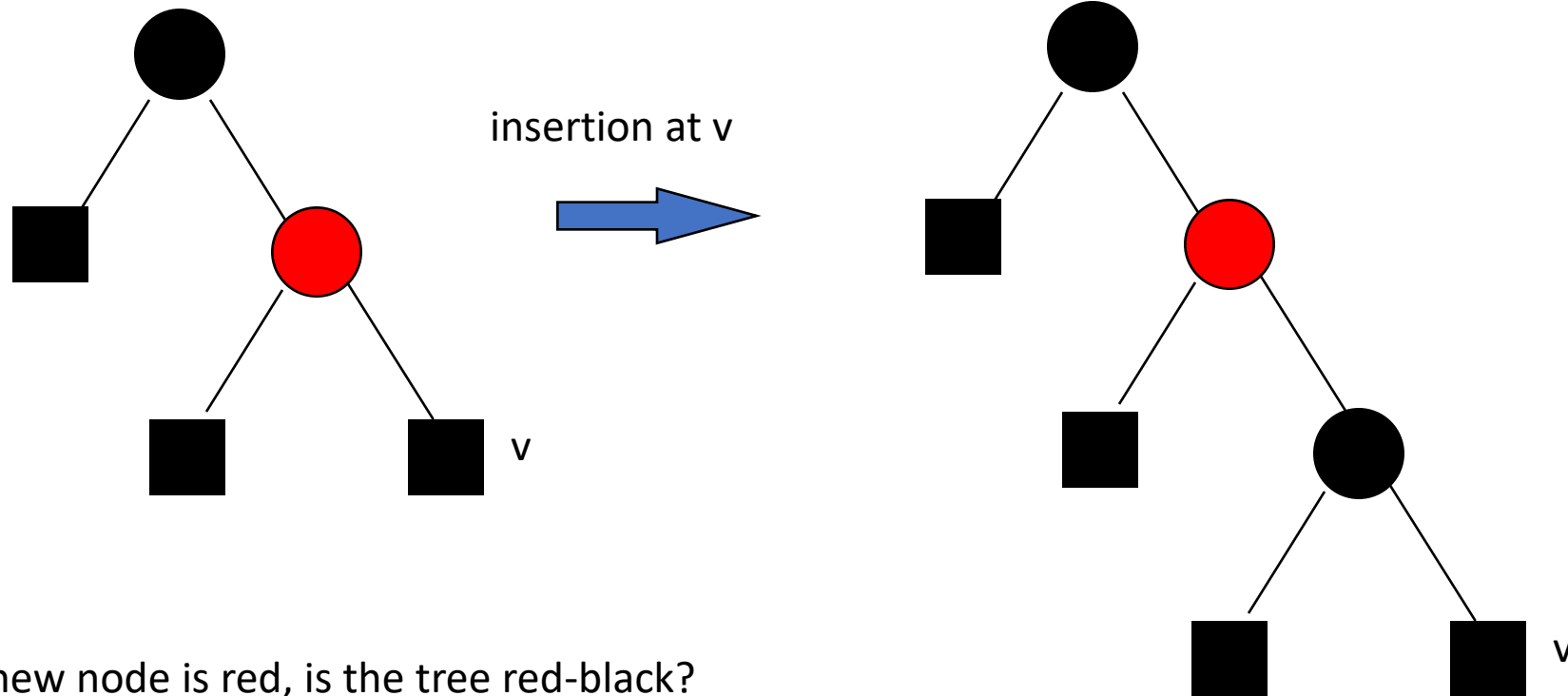


If new node is red, is the tree red-black?

No. Violates red condition---non-root red node has red parent.

If the new node is black, is the tree red-black?

Red-Black Trees



If new node is red, is the tree red-black?

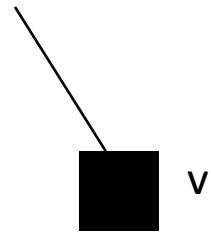
If the new node is black, is the tree red-black?

No. Violates black condition---some paths have 2 black nodes, others have 3.

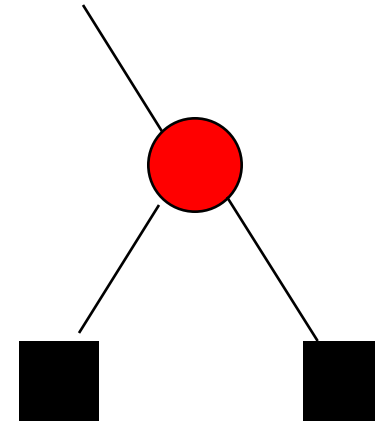
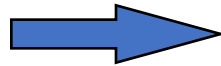
Red-Black Trees

- Recolouring:
 - Colour new node red.
 - This preserves the black condition.
 - but may violate the red condition.
- Red condition can be violated only if the parent of an internal node is also red.
- Must transform this 'almost red-black tree' into a red-black tree.

Red-Black Trees



insertion at v

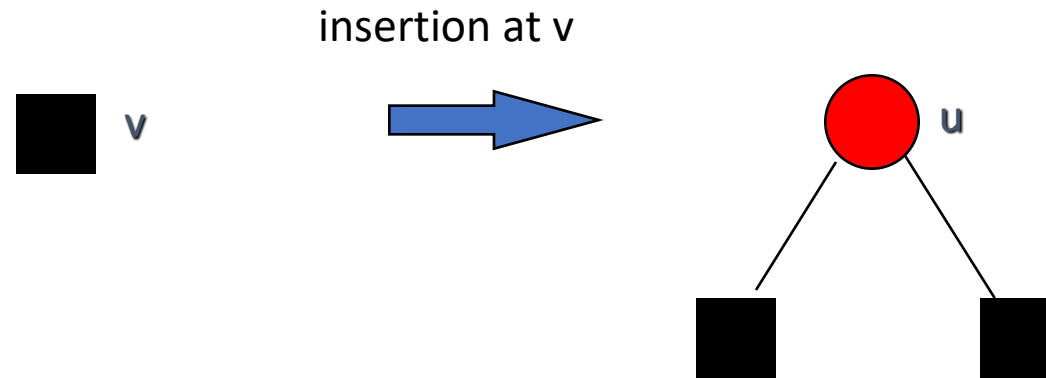


Red-Black Trees

- Recolouring and restructuring algorithm
 - The node u is a red node in a BST, T
 - u is the only candidate violating node
 - Apart from u , the tree T is red-black

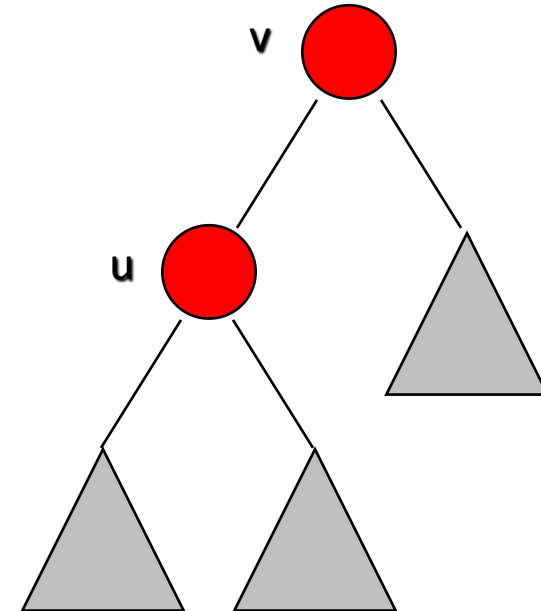
Red-Black Trees

- Case 1:
 - u is the root
 - T is red-black

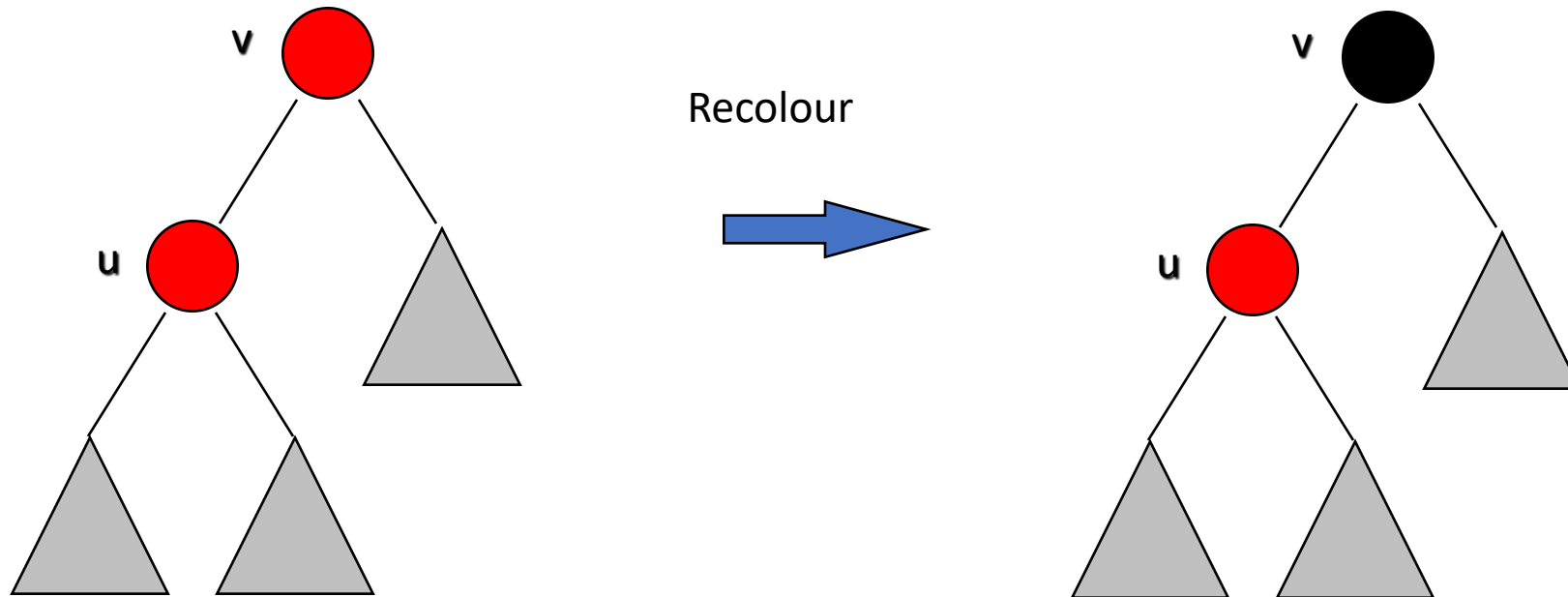


Red-Black Trees

- Case 2:
 - u is not the root
 - its parent v is the root
 - **Colour v black**
 - Since v is the parent and the root, it is on the path to all external nodes and therefore, the black condition is satisfied

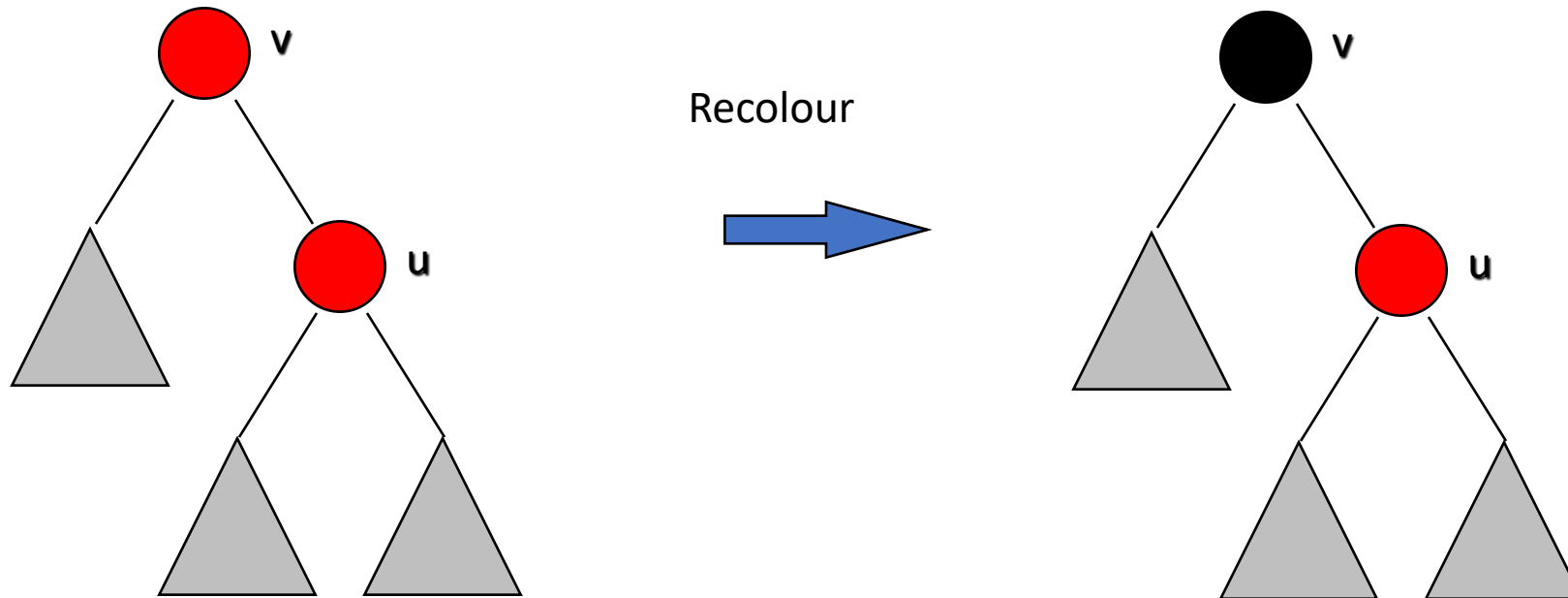


Red-Black Trees



Is there anything unexpected about this figure?

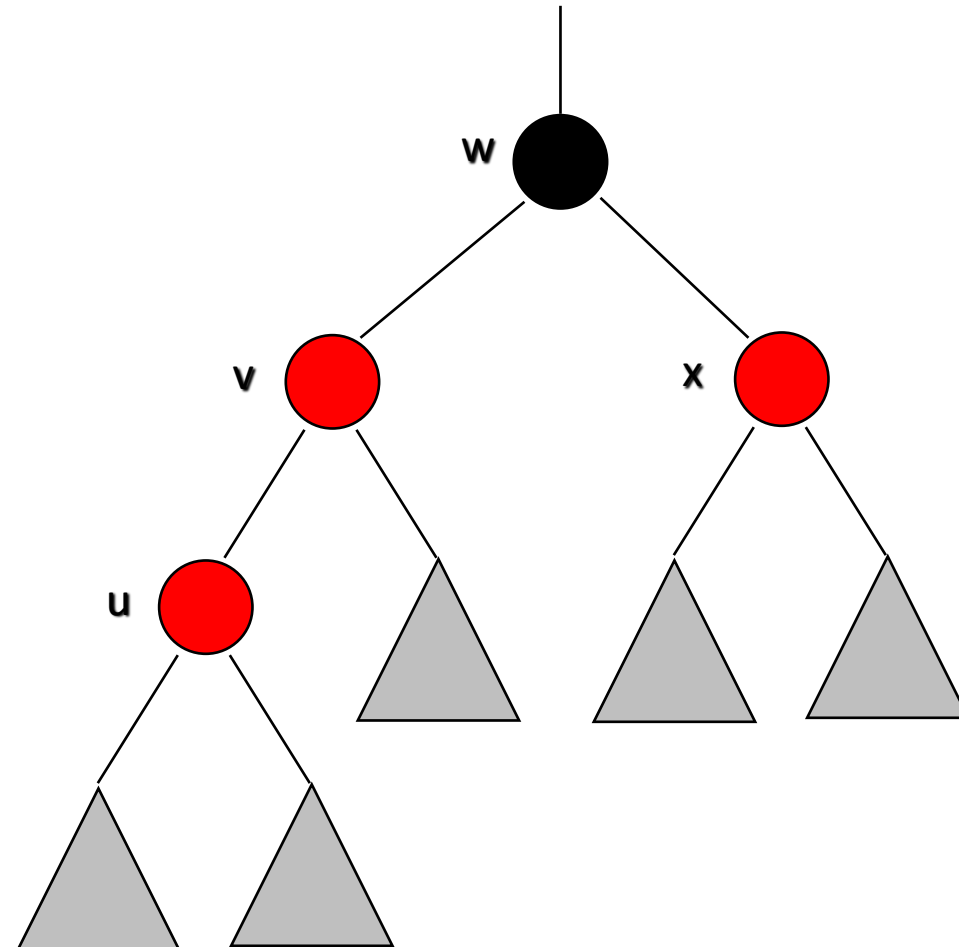
Red-Black Trees



Is there anything unexpected about this figure?

Red-Black Trees

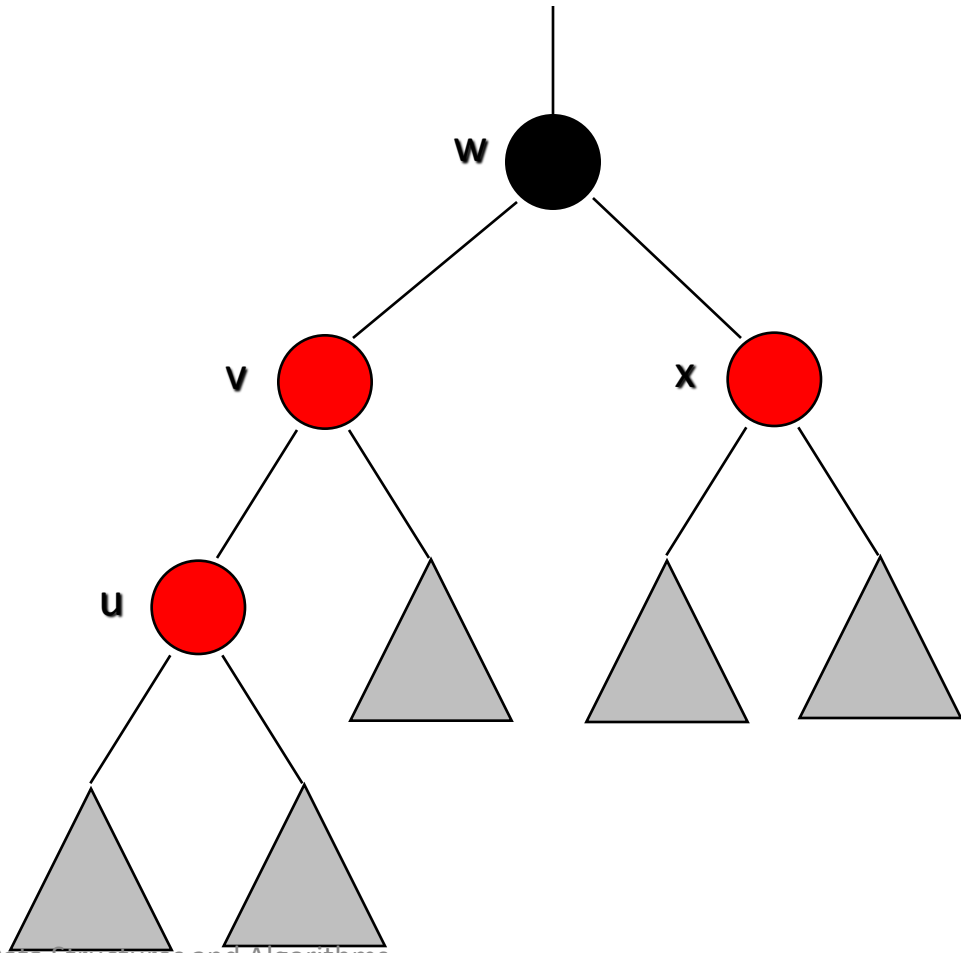
- Case 3:
 - u is not the root,
 - its parent v is not the root,
 - v is the left child of its parent w
 - (x is the right child of w, i.e., x is v's sibling)



Red-Black Trees

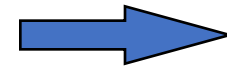
- Case 3.1:
 - x is red
 - Colour v and x black and w red
 - Now repeat the restructuring with $u := w$
(since the recolouring of w to red may cause a **red violation**)

Red-Black Trees



Note:
w must be black,
v must be red,
u must be red.
Why?

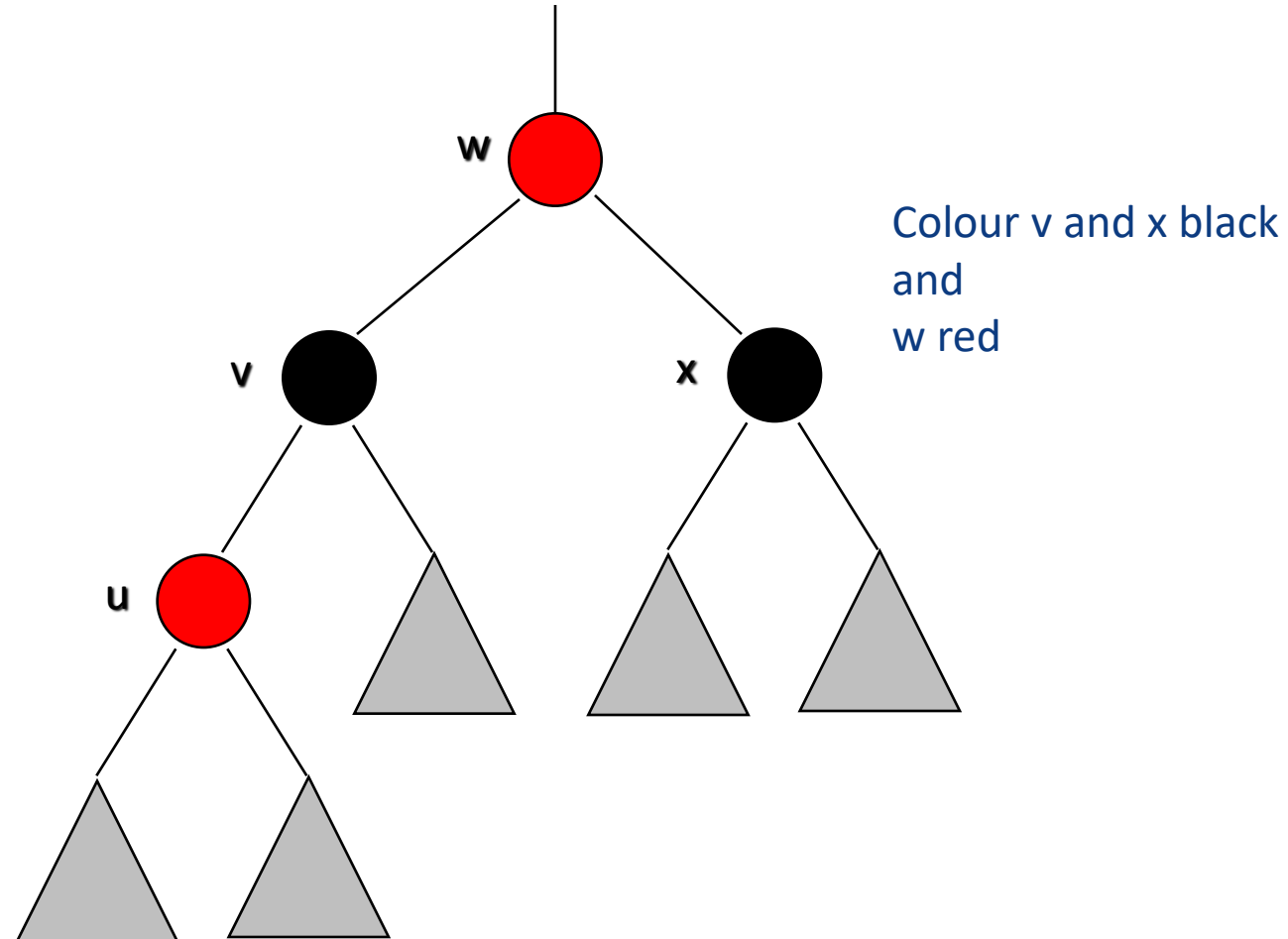
Recolour



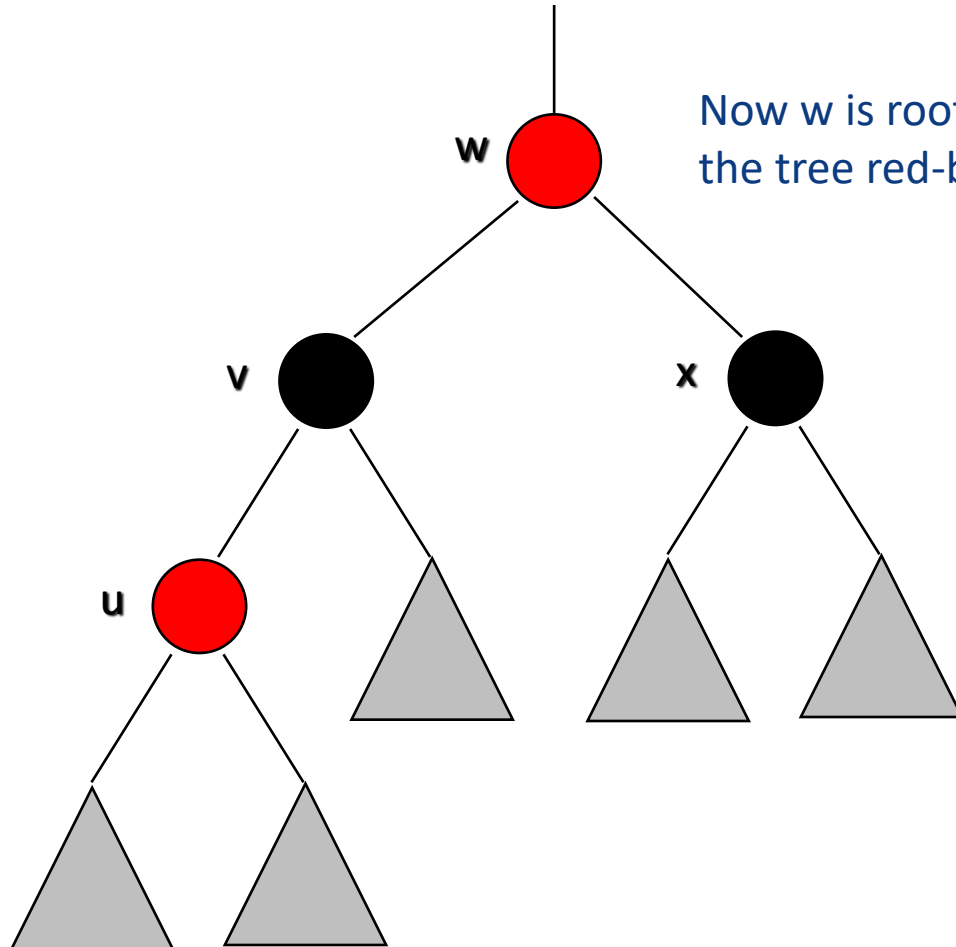
Red-Black Trees

- u must be red because we colour new nodes that way by convention (to preserve the **black condition**).
- v must be red because otherwise it would be black and then we wouldn't have violated the red condition and we wouldn't be restructuring anything!
- w must be black because every red node (that isn't the root) has a black parent (and x is red so w must be black).

Red-Black Trees



Red-Black Trees

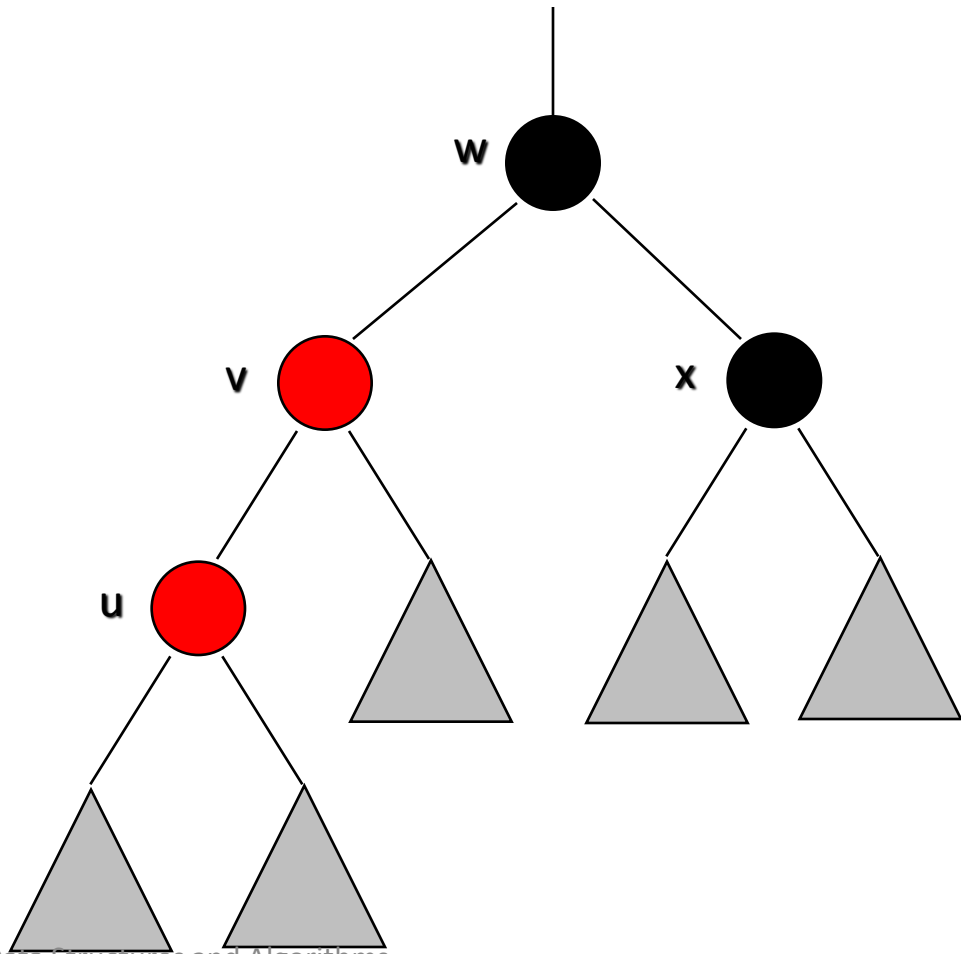


1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each red node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color –**red** or **black**.
5. Root node is **black**.

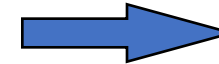
Red-Black Trees

- Case 3.2:
 - x is black
 - u is the left child of v
 - Promote v
 - Colour v black
 - Colour w red

Red-Black Trees

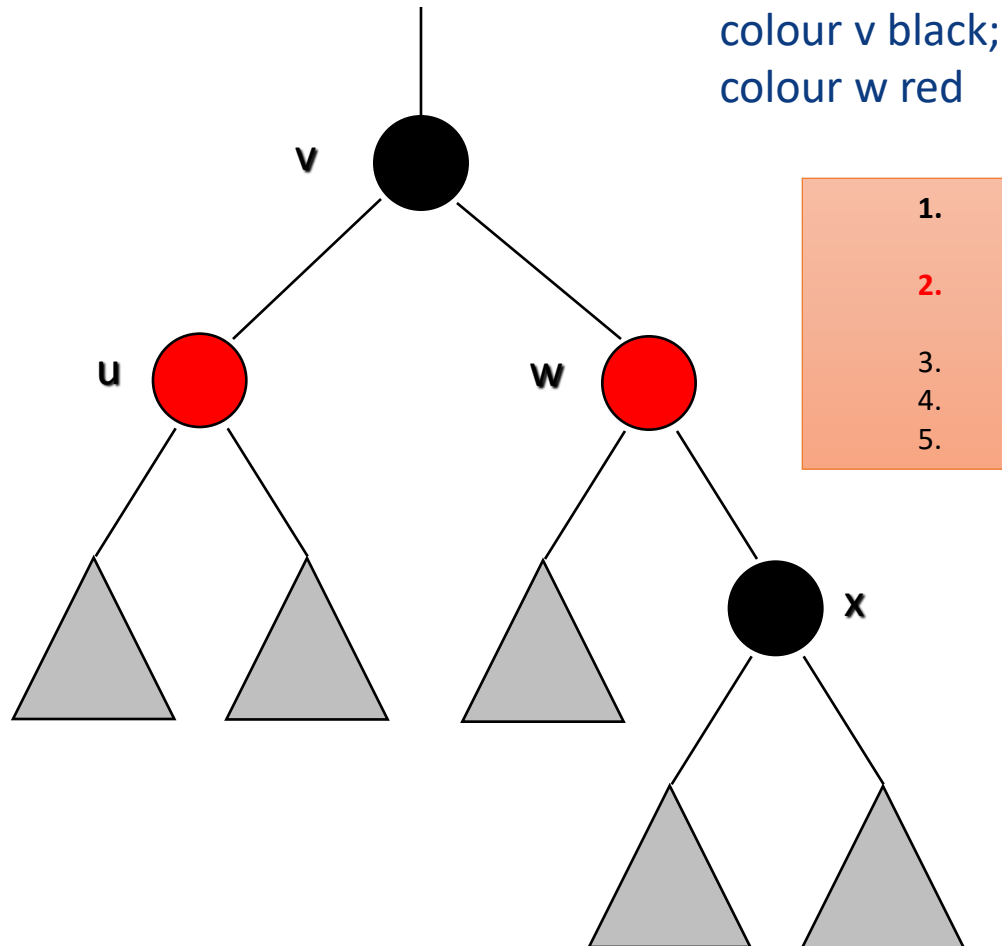


Restructure and recolour



Promote v (right rotation);
colour v black;
colour w red

Red-Black Trees

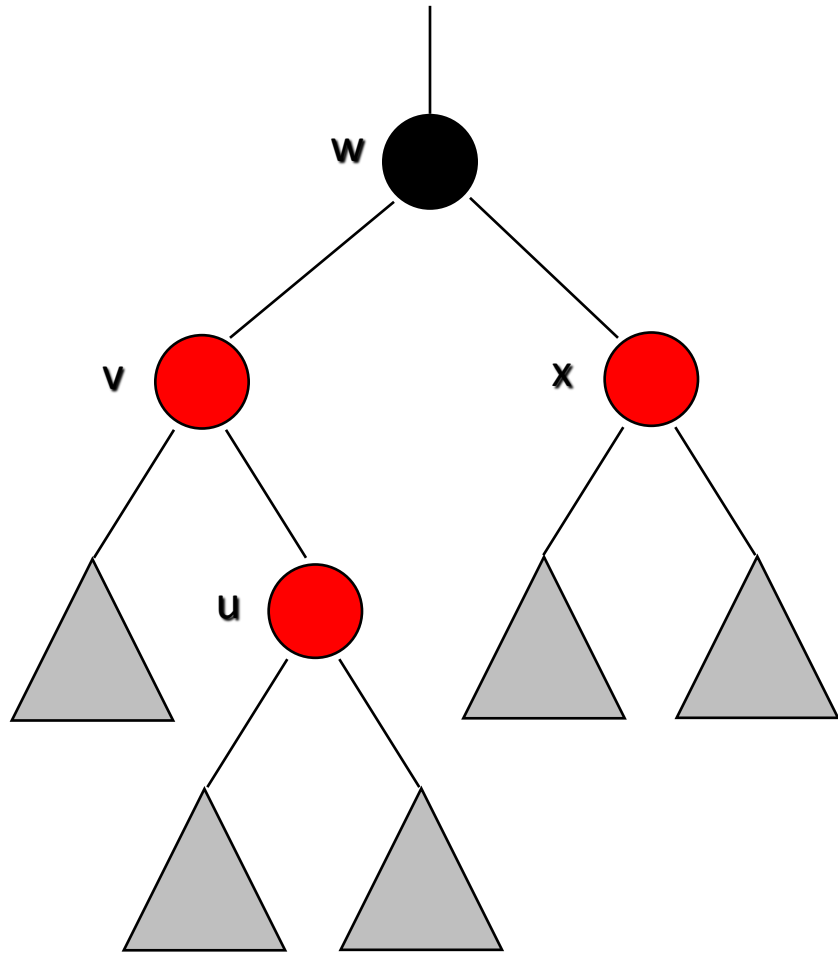


1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each red node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color —**red** or **black**.
5. Root node is **black**.

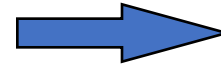
Red-Black Trees

- Case 3.3:
 - x is red
 - u is the right child of v
 - Colour v and x black
 - Colour w red
- Repeat the restructuring with $u := w$
(since the recolouring of w to red may cause a red violation)

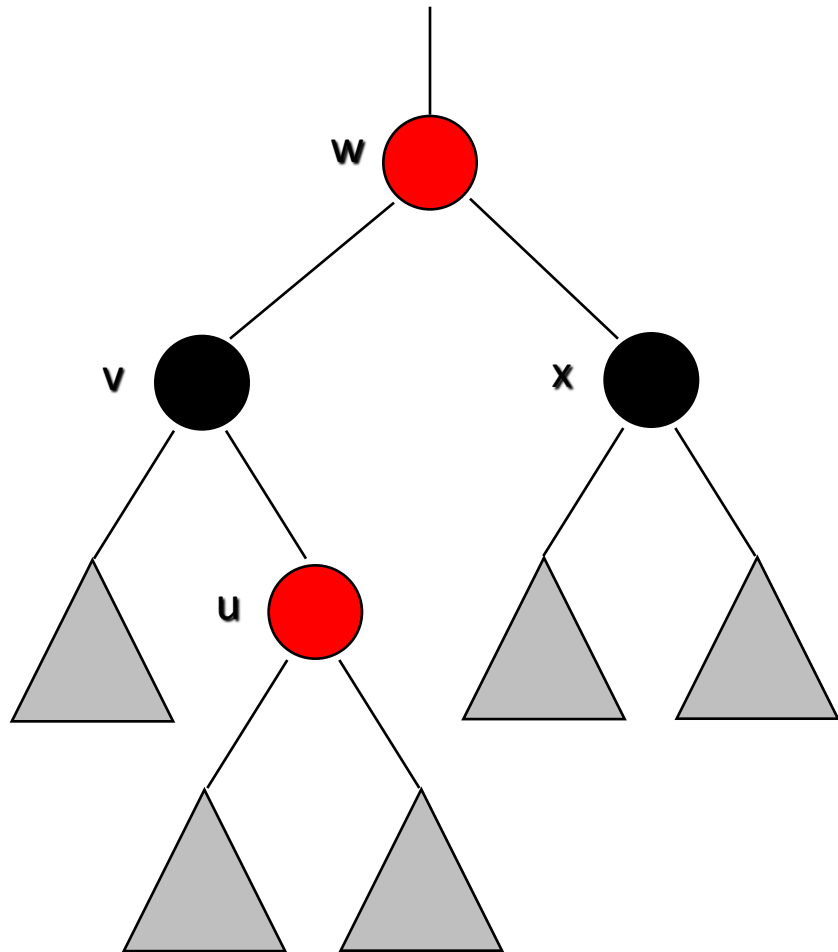
Red-Black Trees



Recolour



Red-Black Trees



Colour v and x black
Colour w red

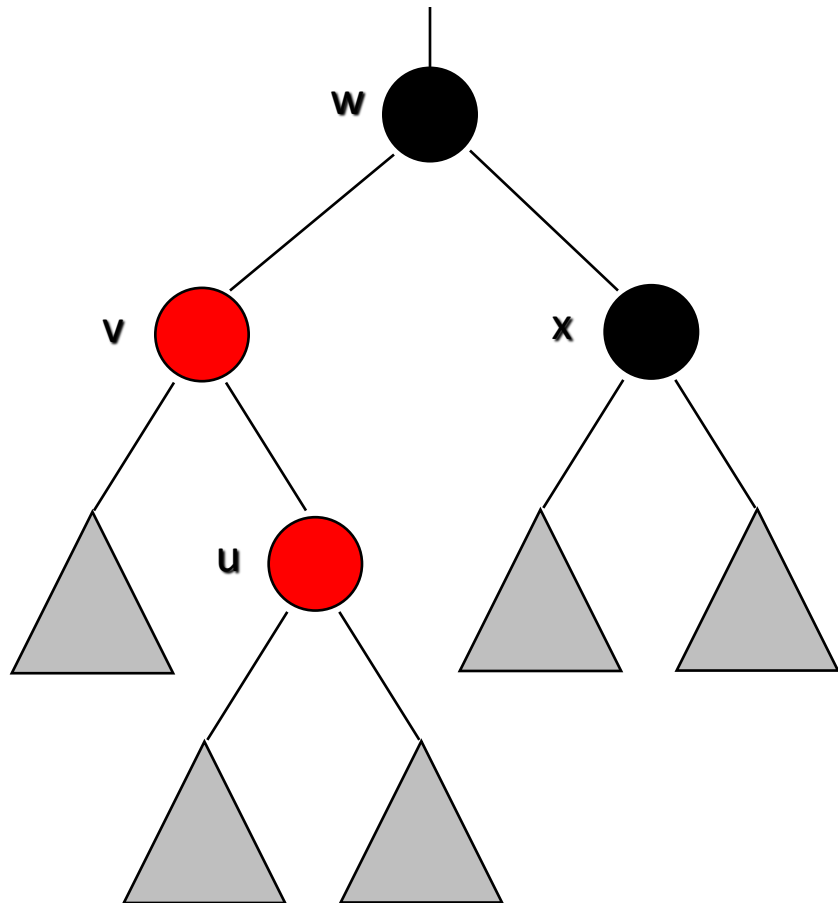
Repeat the restructuring with $u := w$

1. **Black condition:** Each root-to-frontier path contains exactly the *same* number of black nodes
2. **Red condition:** Each red node that is not the root has a black parent (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color –**red** or **black**.
5. Root node is **black**.

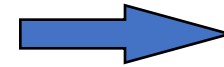
Red-Black Trees

- Case 3.4:
 - x is black
 - u is the right child of v
 - Zig-zag promote u
 - Colour u black
 - Colour w red

Red-Black Trees

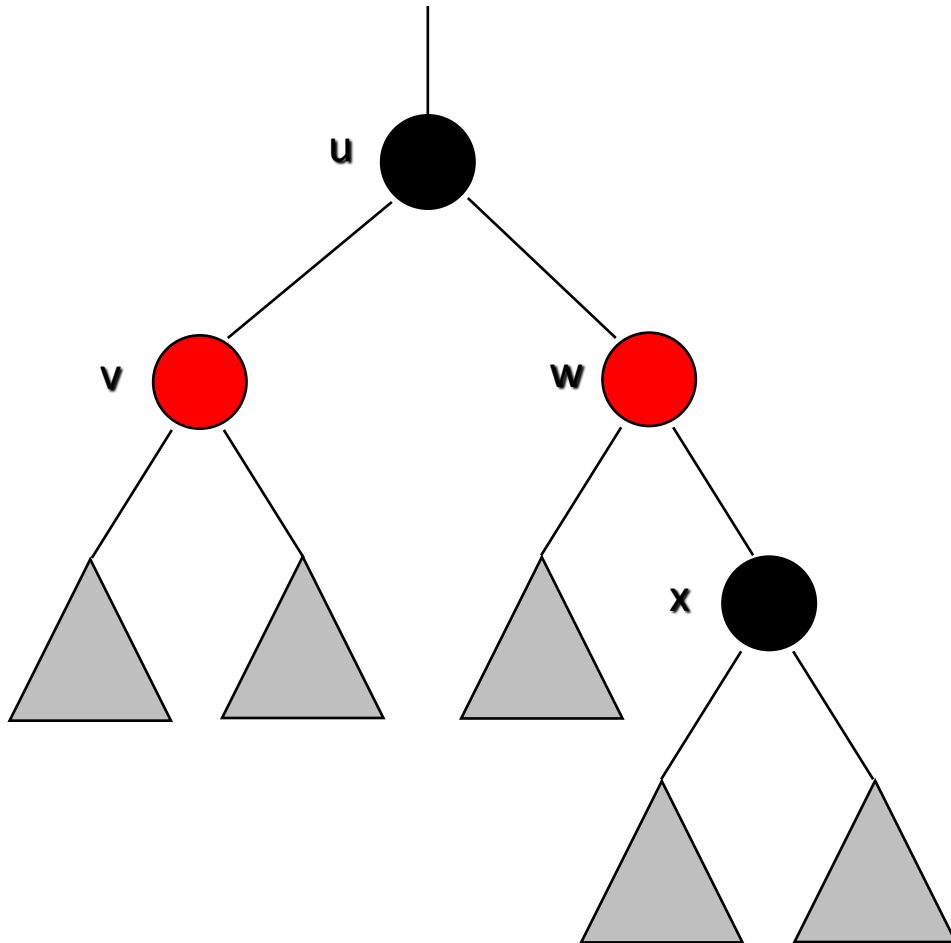


Restructure and recolour



Zig-zag promote u;
colour u black;
colour w red

Red-Black Trees



colour u black;
colour w red

1. **Black condition:** Each root-to-frontier path contains exactly the *same number of black nodes*
2. **Red condition:** Each red node that is not the root *has a black parent* (If a node is red, both children are black)
3. Each external node (leaf node) is **black**.
4. Every node must have one color –**red** or **black**.
5. Root node is **black**.

Red-Black Trees

- Case 4:
 - u is not the root,
 - its parent v is not the root,
 - v is the **right** child of its parent w
 - (x is the **left** child of w, i.e. x is v's sibling)
- This case is symmetric to case 3.

Deletion

- Deletion of node from tree
 - Deleting a node from a Red-Black tree may also need node adjustment
 - Any violation of Red-Black tree properties can also be corrected using a combination of rotation and color change operations.
- *Reading Exercise:* See the 4 cases in Section 13.4 of Introduction to Algorithms 3rd Edition by Cormen et al (2009)

Applications of Red Black Trees(1/2)

- In general, where **sorting, searching , or hashing algorithms** are needed.
- E.g.:
 - Implementation of sets and maps in C++ Standard Templates Library.
 - TreeMap ([TreeMap \(Java Platform SE 7 \) \(oracle.com\)](#)) implemented using Red Black trees, TreeSet—based on TreeMap([TreeSet \(Java Platform SE 7 \) \(oracle.com\)](#)), and HashMap in Java collections.
 - Computational geometry e.g., geometric range searches ([Geometric range searching | ACM Computing Surveys](#))
 - k-means clustering

Applications of Red Black Trees(2/2)

- In general, where **sorting, searching , or hashing algorithms** are needed.
- E.g.:
 - Text-mining.
 - mmap and munmap operations for file/memory mapping in Linux.
 - Searching on the web.
 - Completely Fair Scheduler in Linux.
 - Searching in a dictionary.
 - Databases.

Summary

- Red black trees are height balanced.
- Search, insertion, and deletion have a worst-case complexity of $O(\log n)$.
- Insertion and deletions may lead to restructuring (promotions) and recoloring (as per the case). Insertions can lead to ten (10) distinct cases.
- Red black trees can be used in applications that require searching, sorting, and hashing algorithms.