CS211FZ: Data Structures and Algorithms II

Trees

Red-Black Trees and AVL Trees

LECTURER: CHRIS ROADKNIGHT

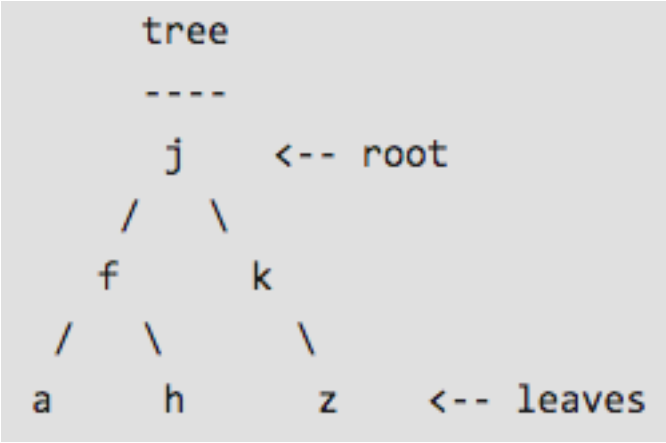
CHRIS.ROADKNIGHT@MU.IE

Trees…a recap

Arrays, linked lists, stacks and queues are linear data structures.

Trees are hierarchical data structures.

Trees can provide **faster search** than linked lists, and **faster insert or delete** than arrays.



Source: https://www.geeksforgeeks.org/binary-tree-set-1-introduction/

Tree Traversals

Linear data structures have only one logical way to traverse them, a tree can be traversed in many different ways, for example

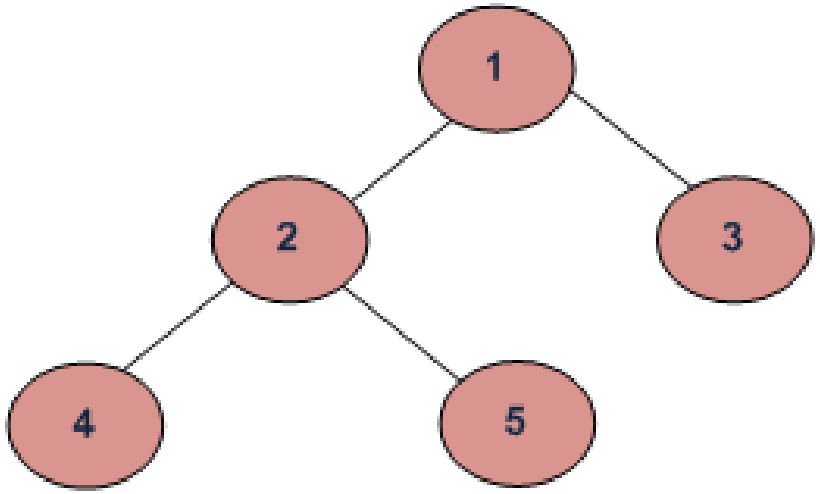
Depth First Traversals:

- Inorder (Left, Root, Right) - 4 2 5 1 3
- Preorder (Root, Left, Right) - 1 2 4 5 3
- Postorder (Left, Right, Root) - 4 5 2 3 1

Breadth First Traversals

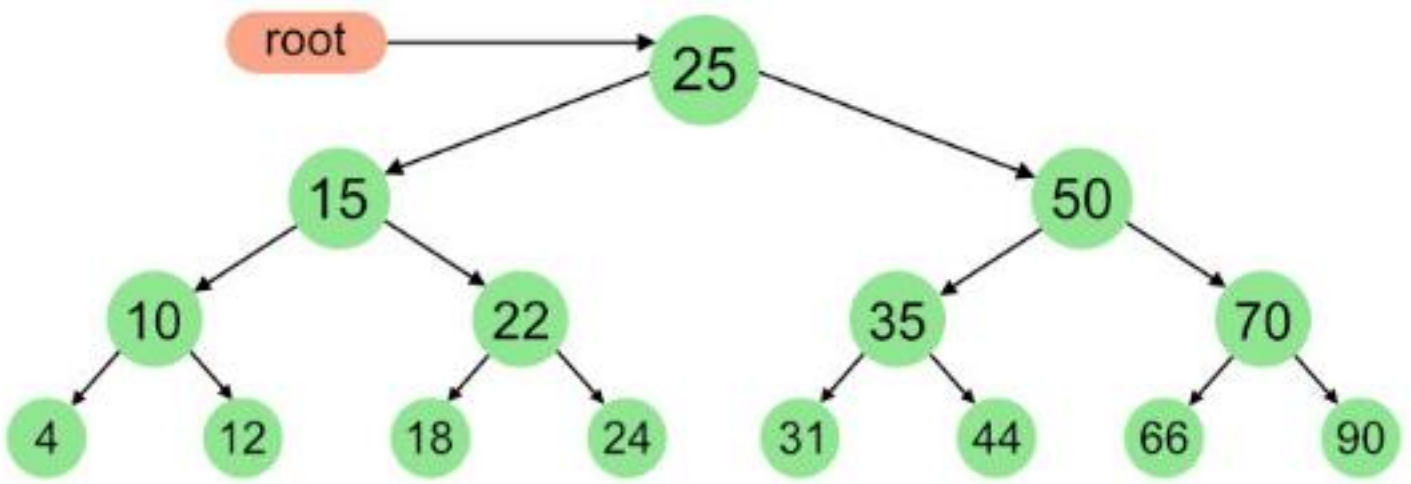
- Level Order -1 2 3 4 5

:



Example #2: Pre-order Traversal

Root, Left, Right

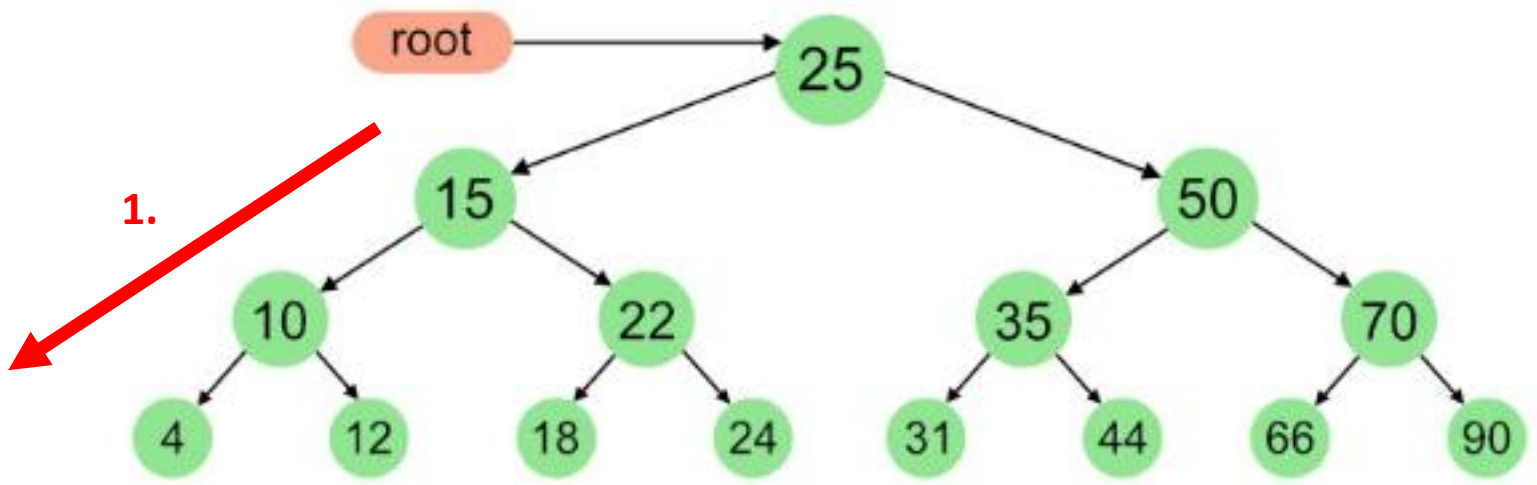


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4

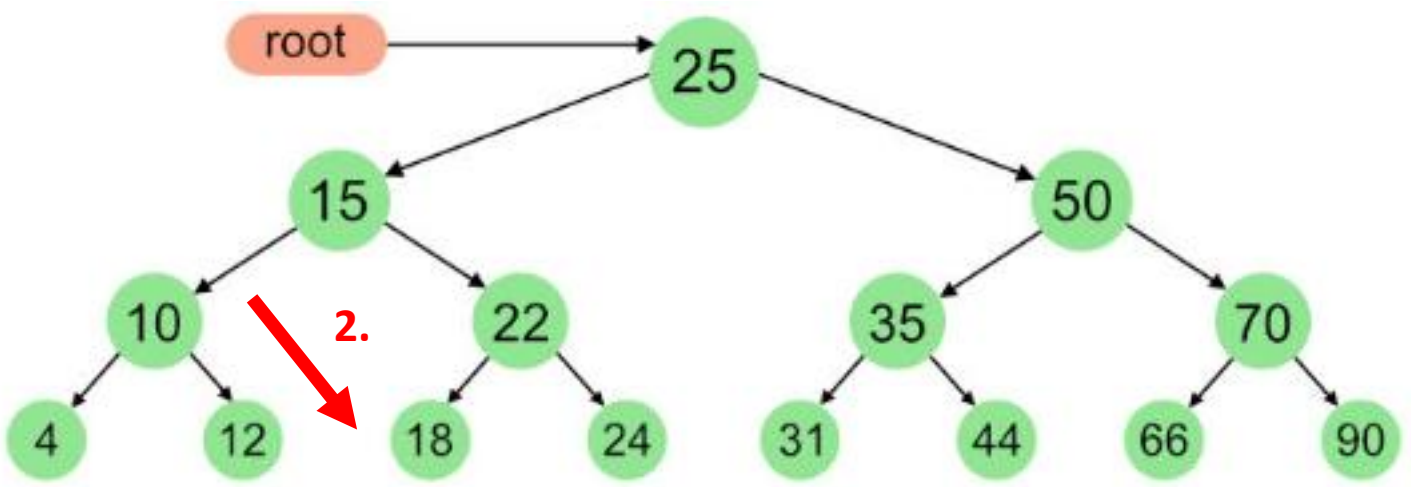


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12

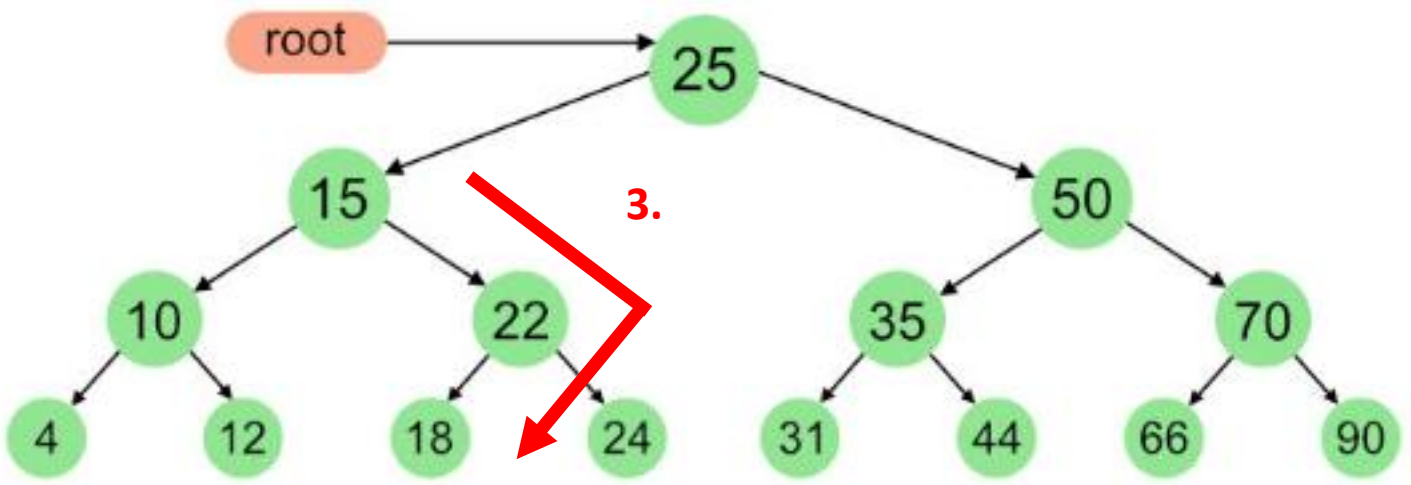


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12 22 18

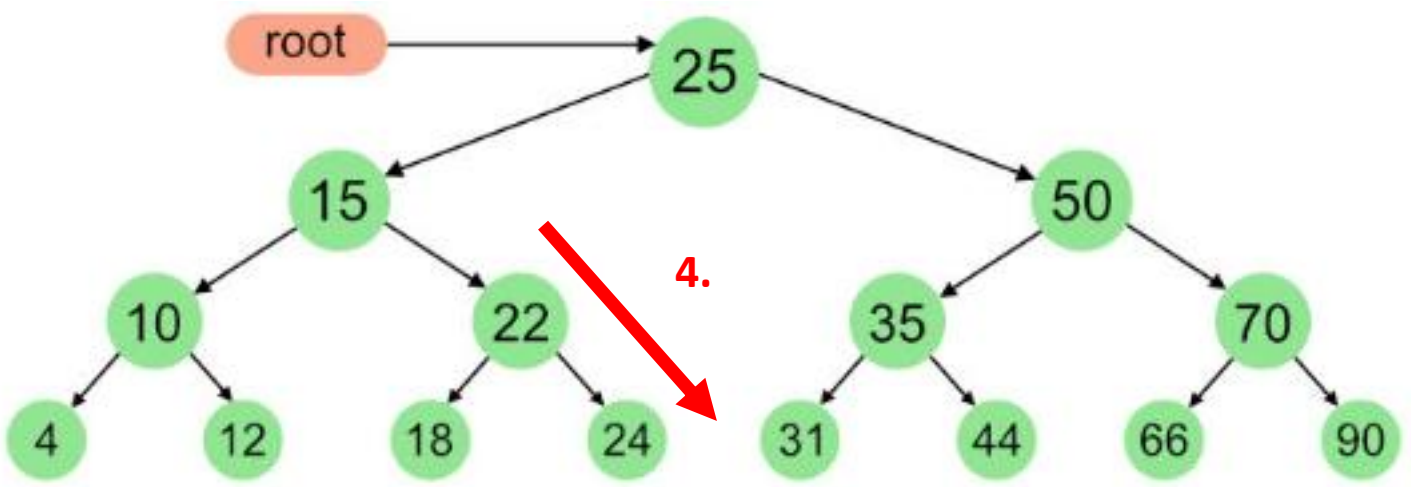


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12 22 18 24

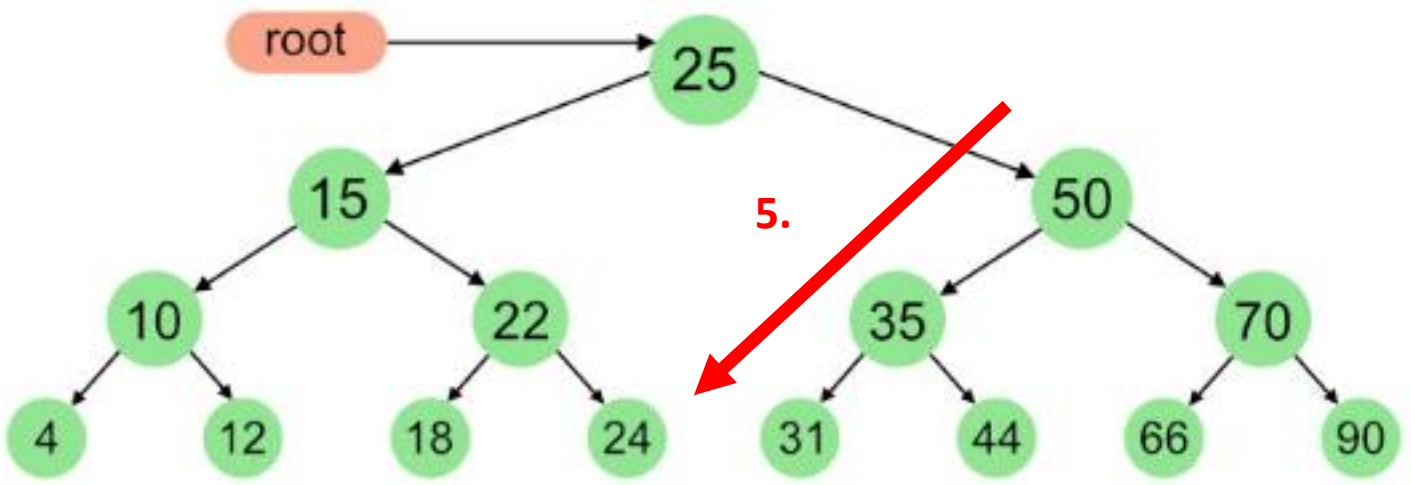


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12 22 18 24 50 35 31

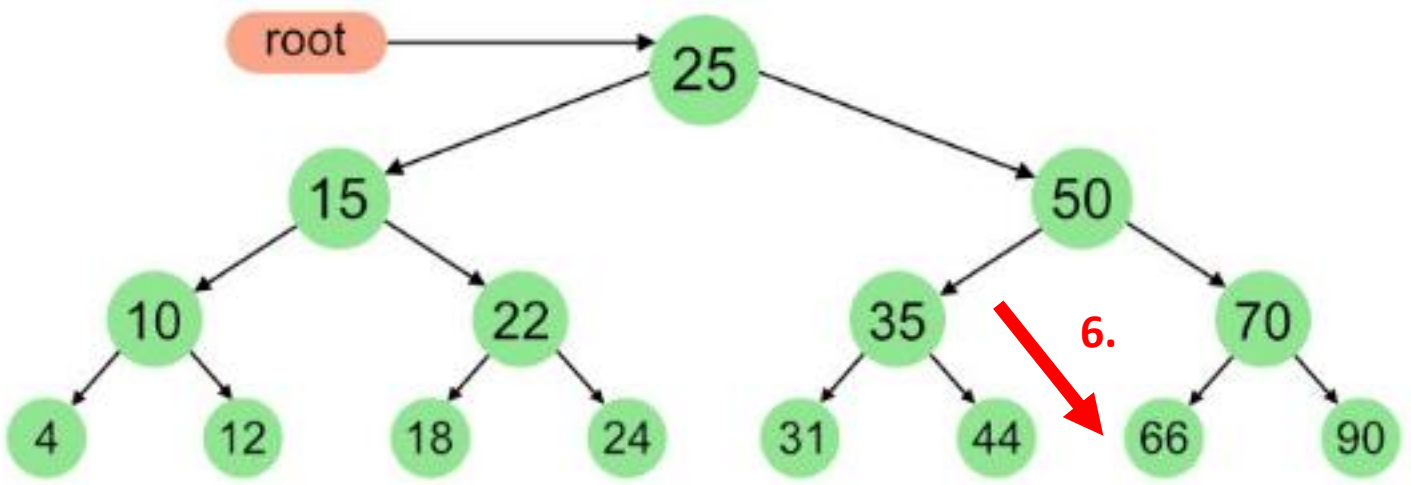


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12 22 18 24 50 35 31 44

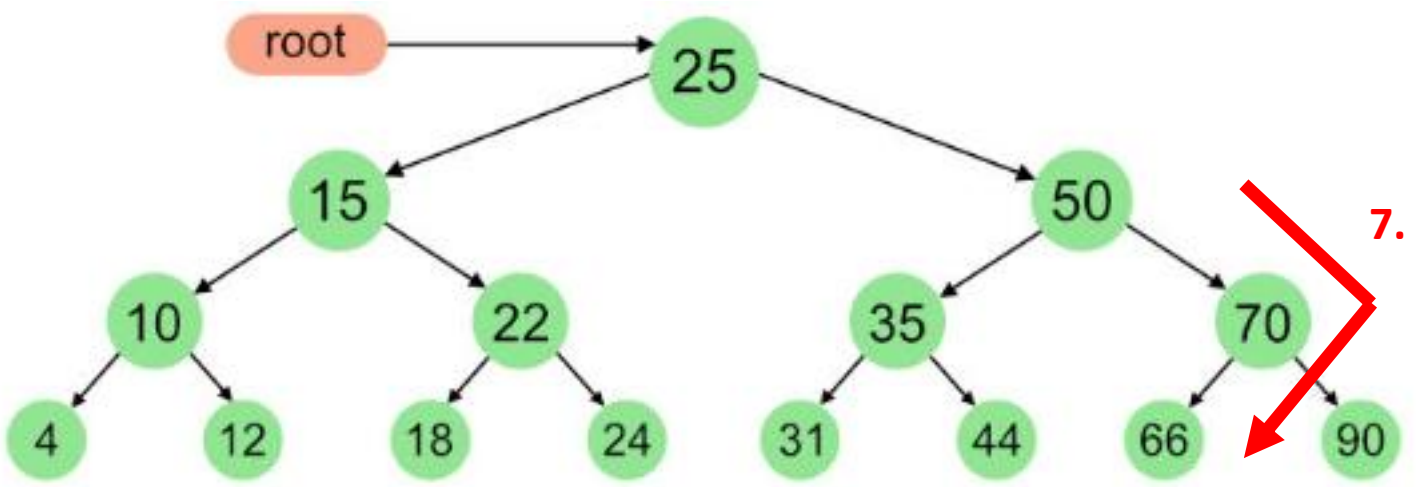


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

25 15 10 4 12 22 18 24 50 35 31 44 70 66

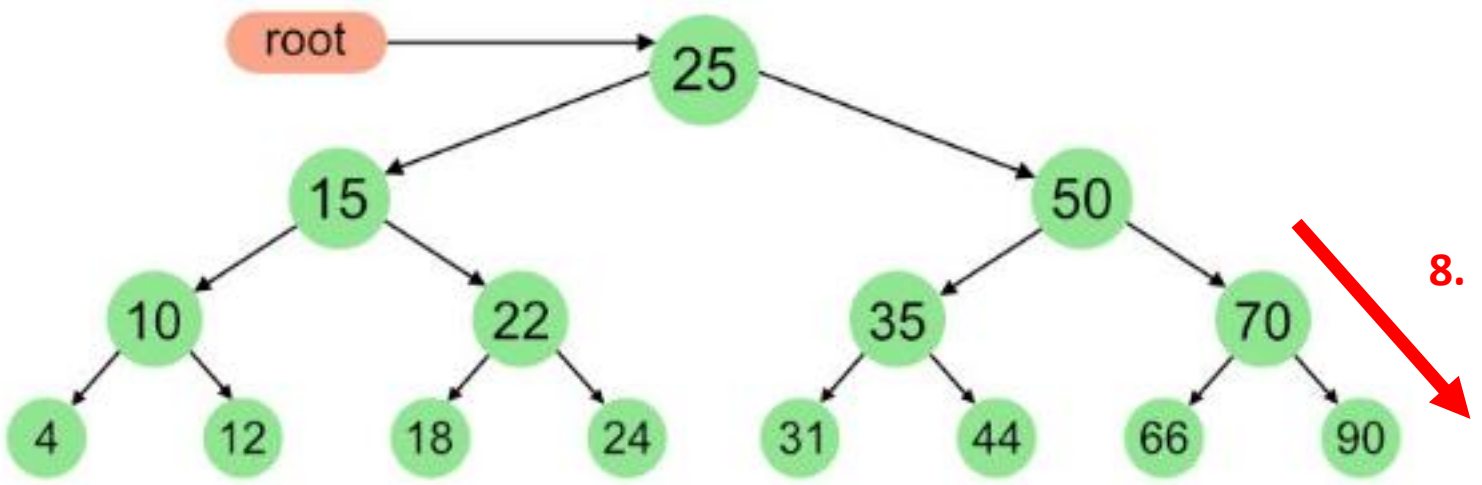


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #2: Pre-order Traversal

Root, Left, Right

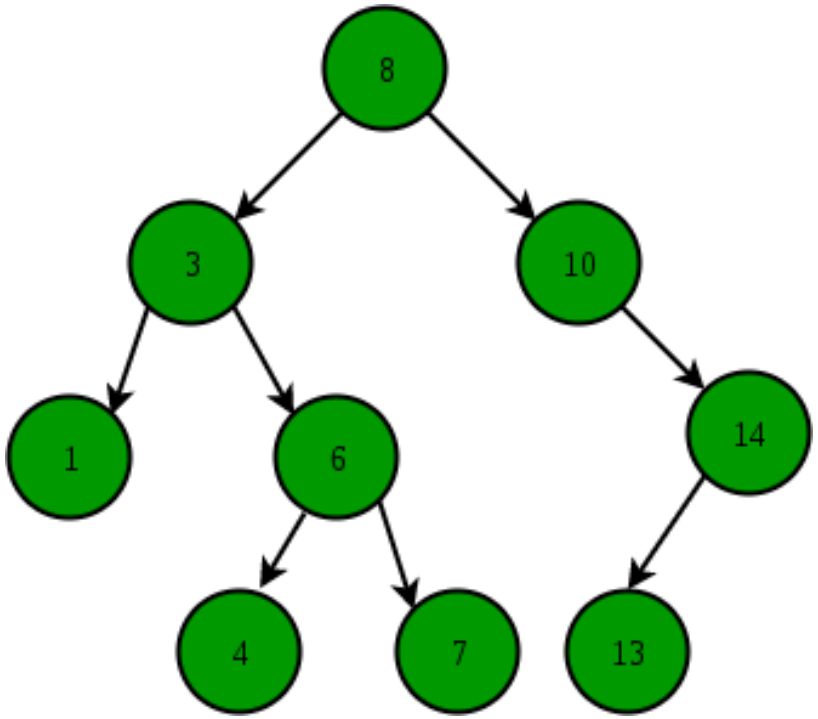
25 15 10 4 12 22 18 24 50 35 31 44 70 66 90



Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Binary Search Tree - recap

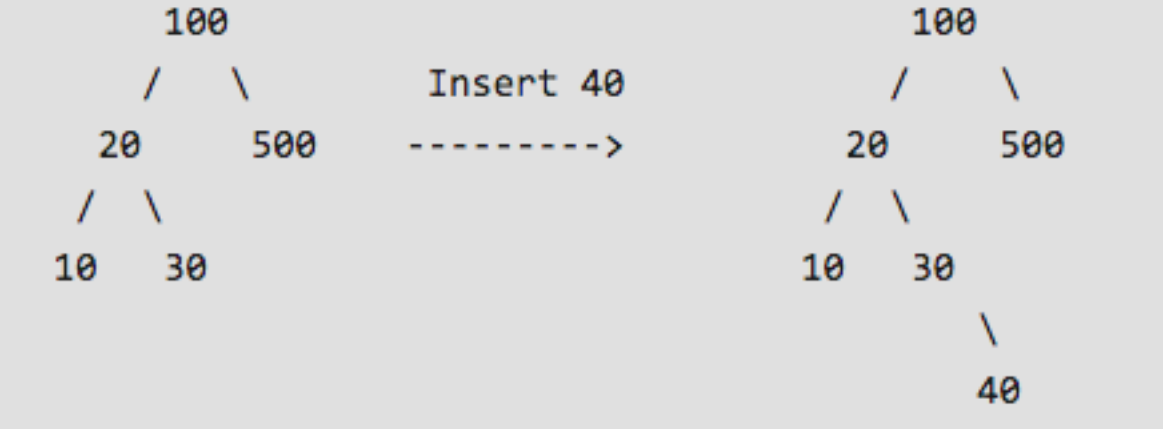
A binary tree in which a node has a value greater than all values in its left subtree and smaller than all values in its right subtree.



Binary Search Tree: Insertion

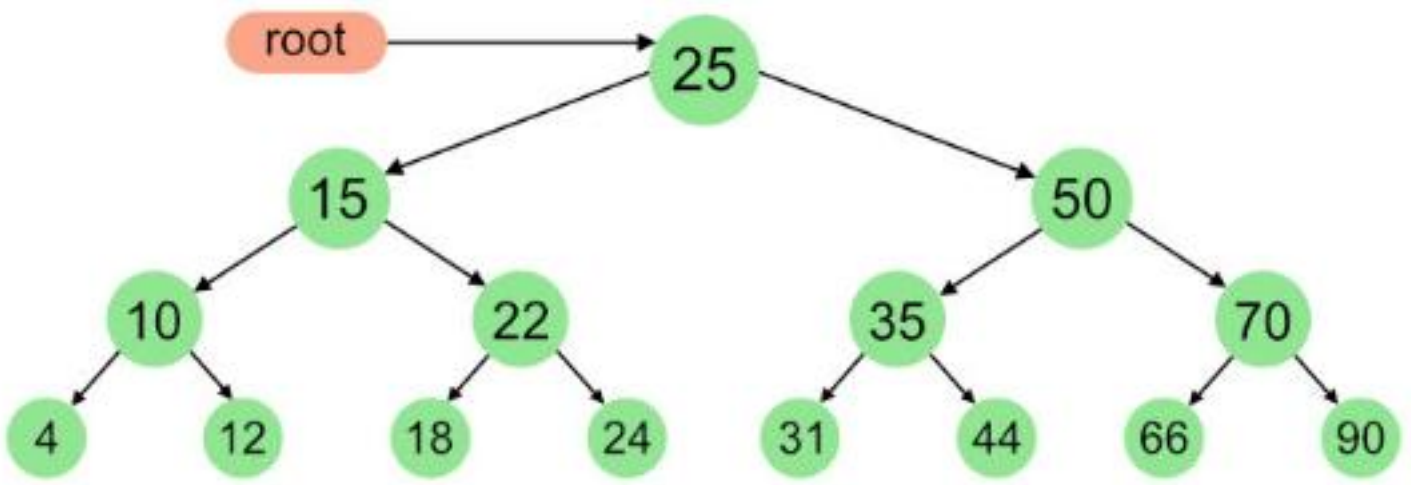
Compare the inserting element with root, if less than root, then recur for left, else recur for right.

After reaching a leaf node, just insert that node at left (if less than current) else right.



Example #3: In-order Traversal

Left, Root, Right

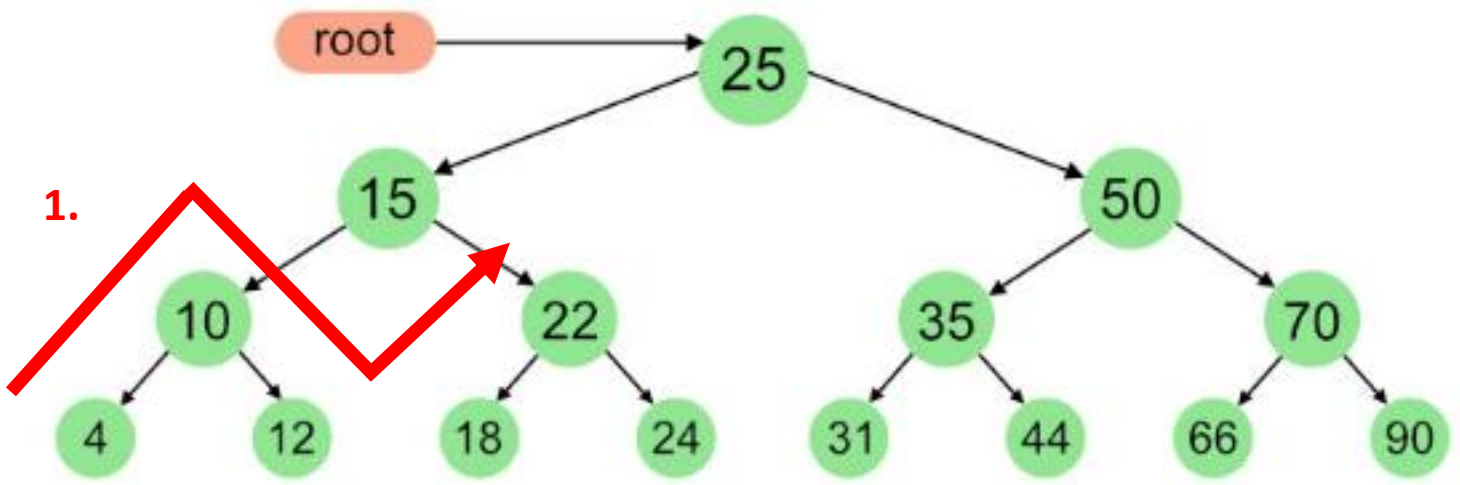


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #3: In-order Traversal

Left, Root, Right

4 10 12 15

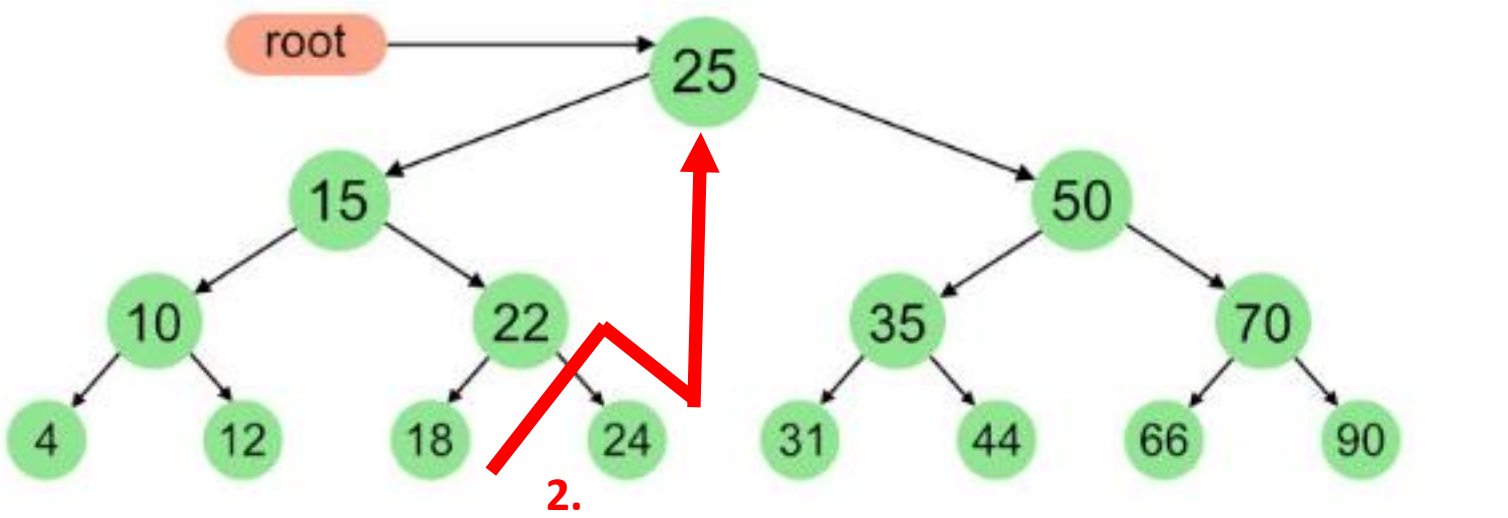


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #3: In-order Traversal

Left, Root, Right

4 10 12 15 18 22 24 25

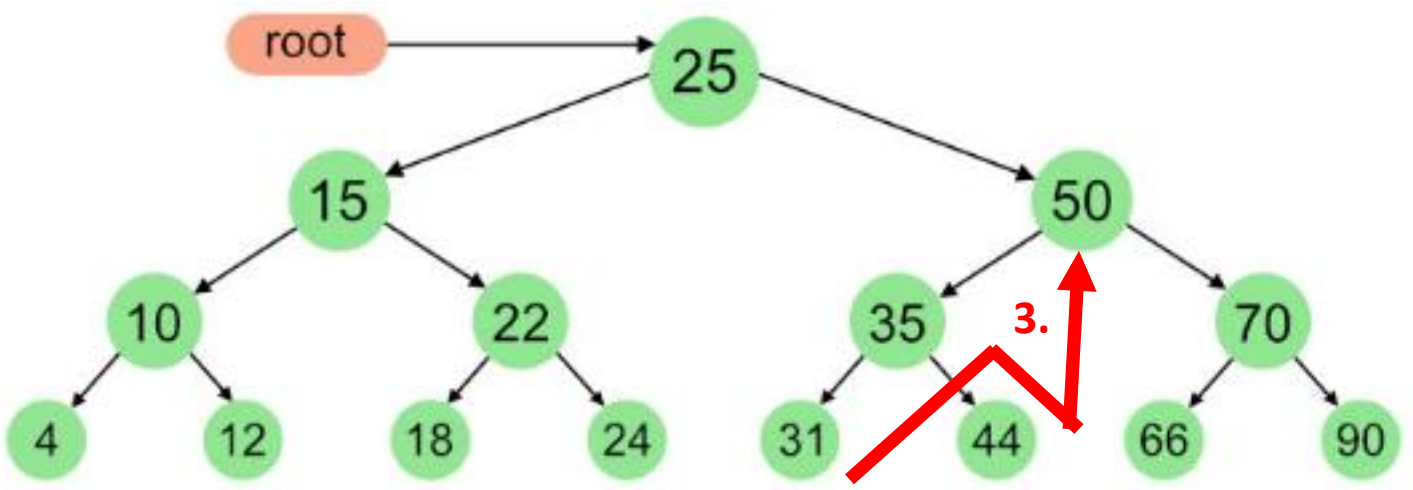


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #3: In-order Traversal

Left, Root, Right

4 10 12 15 18 22 24 25 31 35 44 50

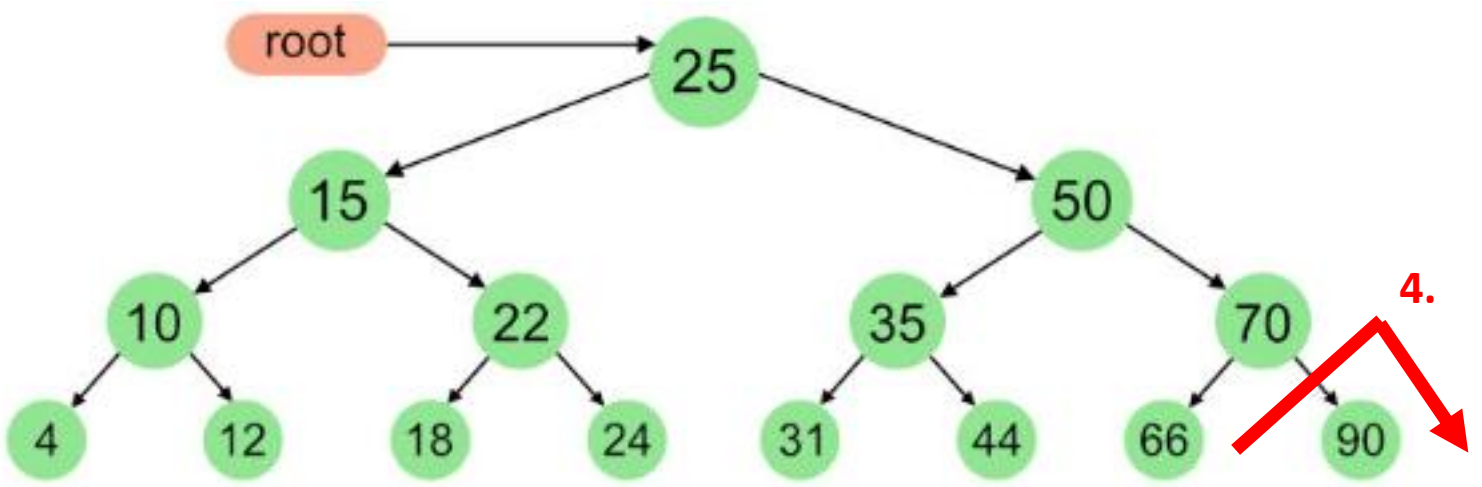


Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

Example #3: In-order Traversal

Left, Root, Right

4 10 12 15 18 22 24 25 31 35 44 50 66 70 90



Source: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/?ref=rp

AVL (Adelson-Velsky and Landis) tree

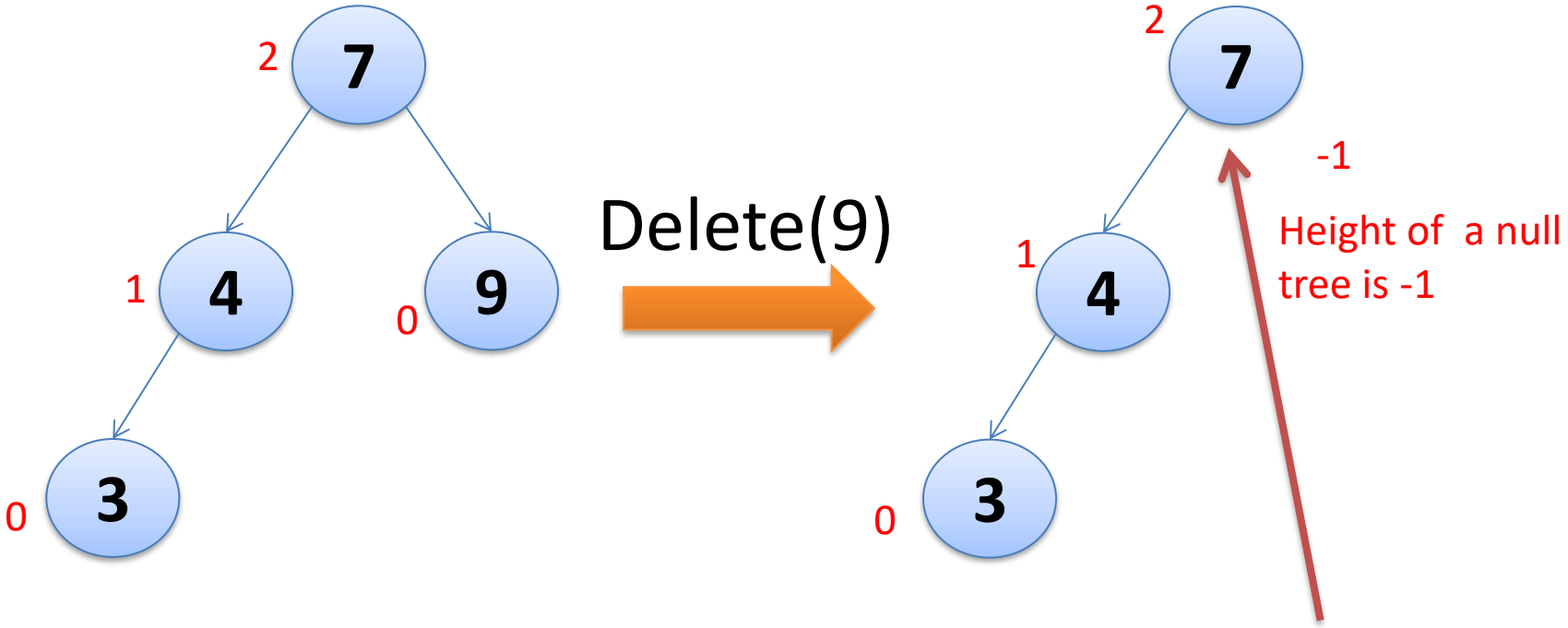
- Is a binary search tree
- Has an additional height constraint:
  - For each node x in the tree, Height(x.left) differs from Height(x.right) by at most 1
  - If you satisfy this height constraint, then the height of the tree is O(log (n)).
  - Therefore insert, delete and find are all O(log(n))

AVL tree

- To be an AVL tree, the tree must:
  - (1) Be a binary search tree
  - (2) Satisfy the height constraint

- Example. start with an AVL tree, then delete as if we're in a regular BST.
- Will the tree be an AVL tree after the delete?
  - (1) It will still be a BST…
  - (2) Will it satisfy the height constraint?

BST Delete breaks an AVL tree



Reminder, h = the length of the longest path from that node down to a leaf node

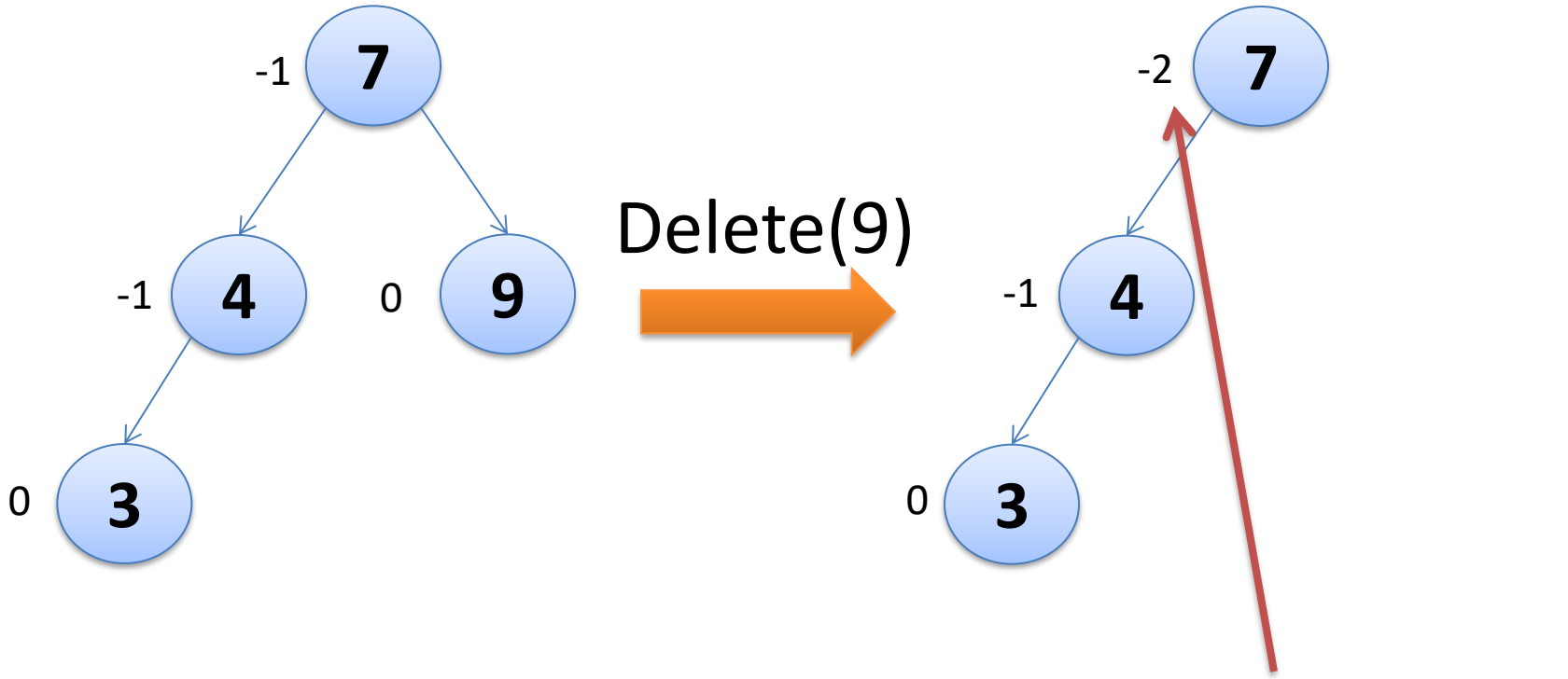
h(left) > h(right)+1

so **NOT** an AVL tree!

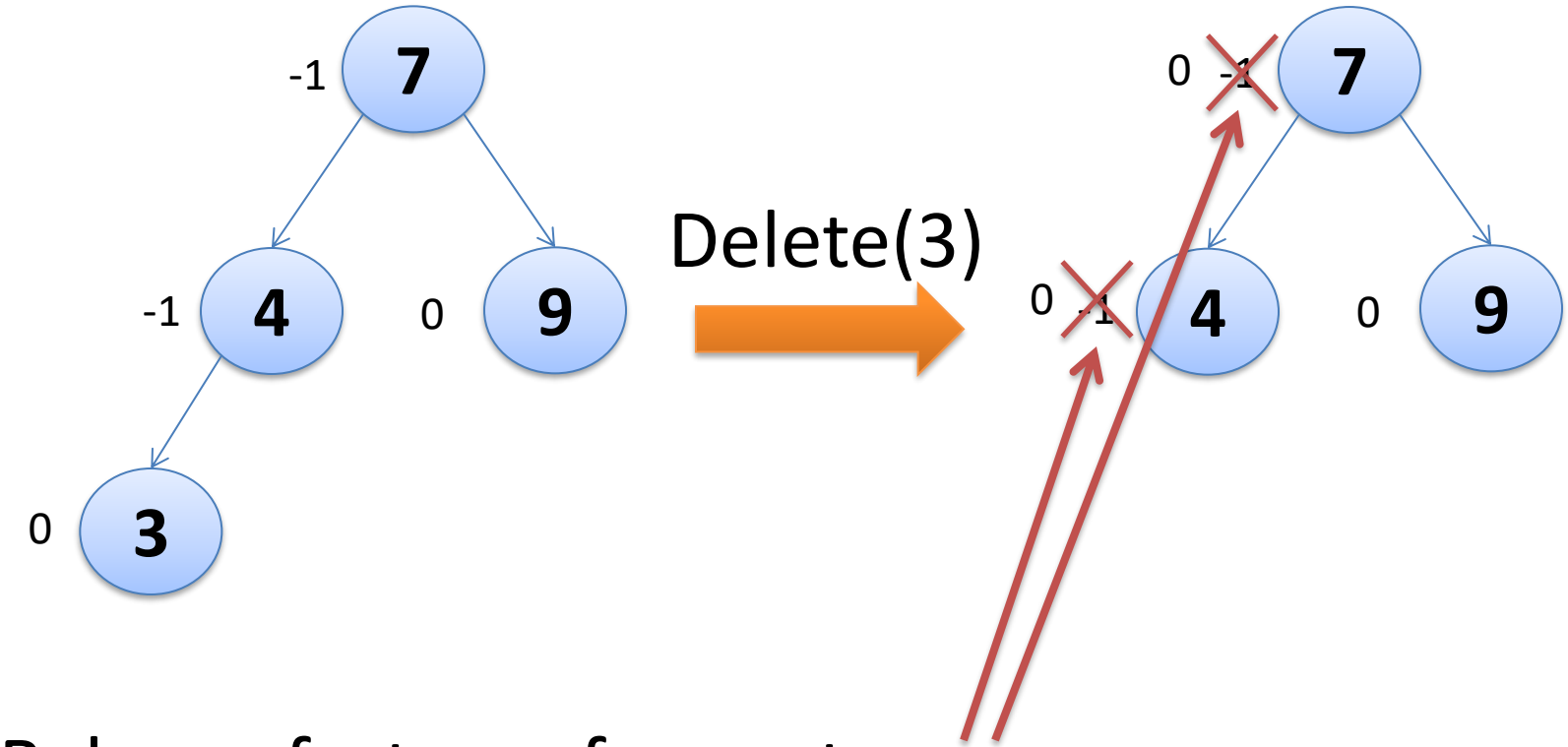
Replacing height differential with balance factors

- Instead of thinking about the heights of nodes, it is helpful to think in terms of *balance factors*
- The balance factor *bf(x) = h(x.right) – h(x.left)*
  - bf(x) values -1, 0, and 1 are allowed
  - If bf(x) < -1 or bf(x) > 1 then tree is NOT AVL

Same example with **bf(x)**



What else can BST Delete break?



- Balance factors of ancestors…

Delete in an AVL tree

- Step 1: do BST delete.
  - This maintains the BST property, but can cause the balance factors of ancestors to change
- Step 2: fix the height constraint and update balance factors.
  - Update any invalid balance factors affected by delete.
    - After updating them, they can be 0, < -1 or > 1.
  - Do rotations to fix any balance factors that are too small or large while maintaining the BST property.

Balance factor Violations

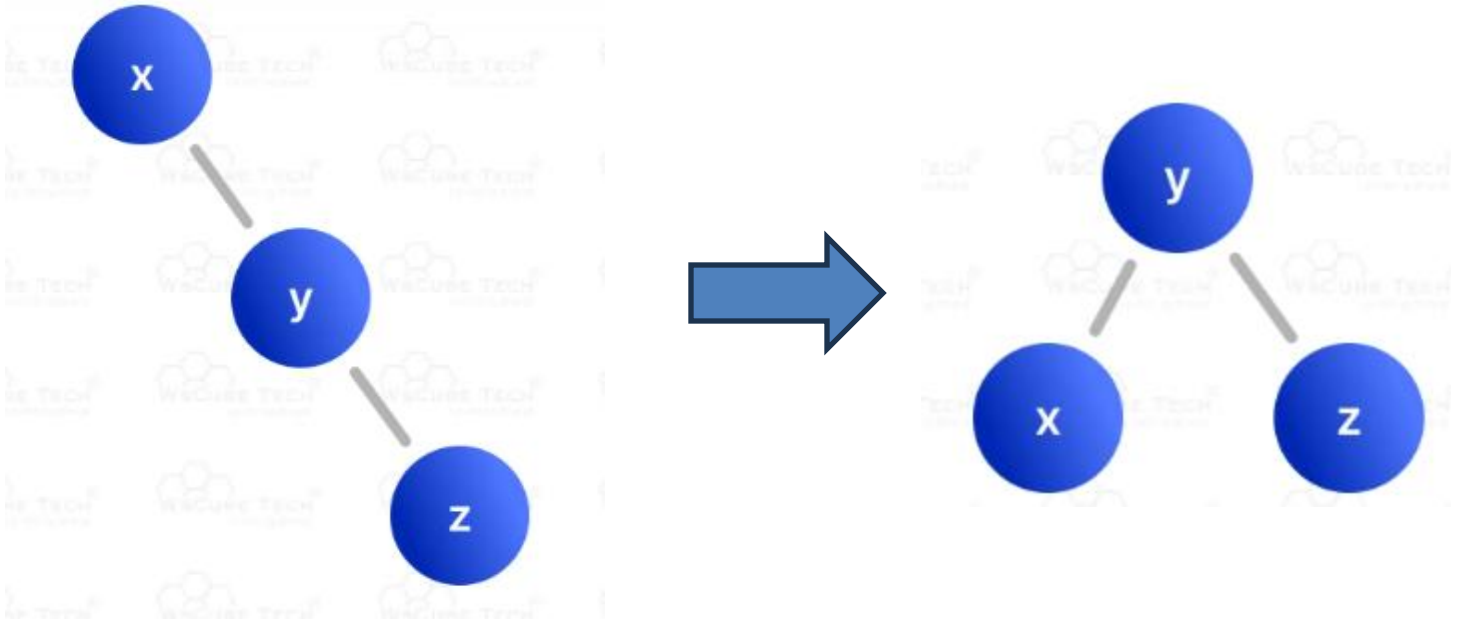
- Start with an AVL tree, then do a BST Delete.
- What bad values can bf(x) take on?
  - Delete can reduce a subtree's height by 1.
  - So, it might increase or decrease h(x.right) – h(x.left) by 1.
  - So, bf(x) might increase or decrease by 1.
  - This means:
    - if bf(x) = 1 before Delete, it might become 2. BAD.
    - If bf(x) = -1 before Delete, it might become -2. BAD.
    - If bf(x) = 0 before Delete, then it is still -1, 0 or 1. OK.

Rotations - insertions

- AVL tree rotations are operations performed to rebalance the tree after insertions or deletions, ensuring the binary search tree (BST) property is maintained.
- There are four main types of AVL rotations:
  - left rotation
  - right rotation
  - left-right rotation
  - right-left rotation.

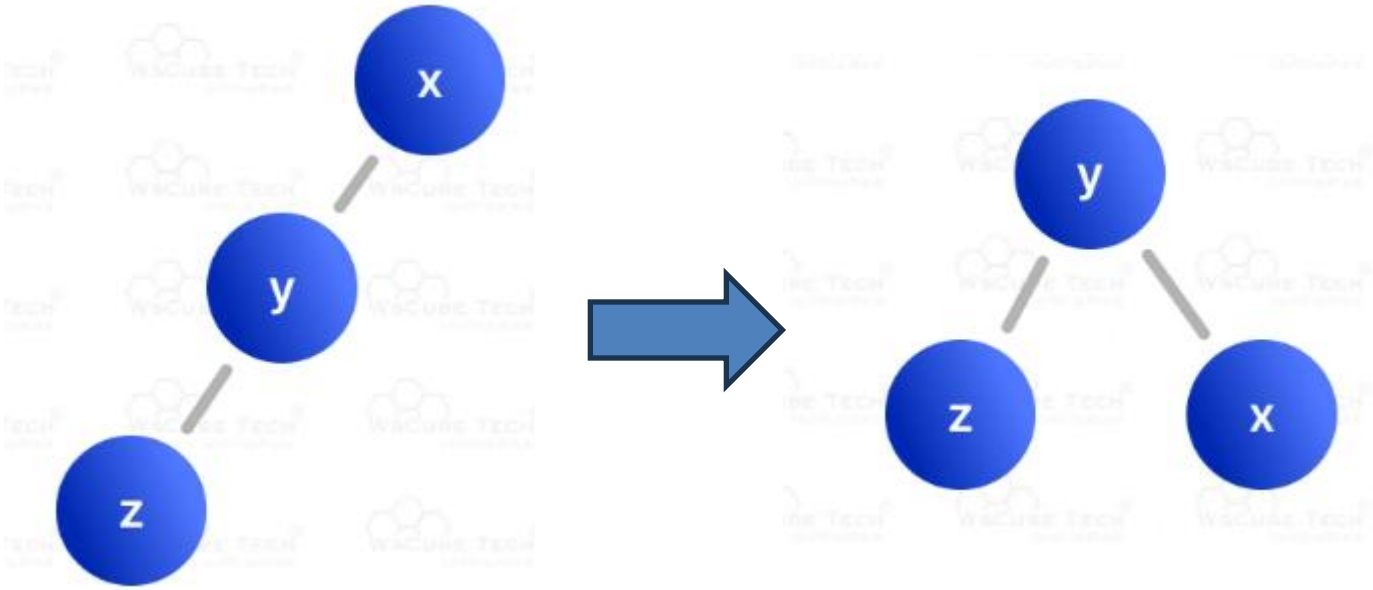
Left Rotation

- When a node is inserted into the right subtree of the right subtree, causing an imbalance.
- The unbalanced node's **right child** becomes the new root, and the original root becomes the new left child of the right child.



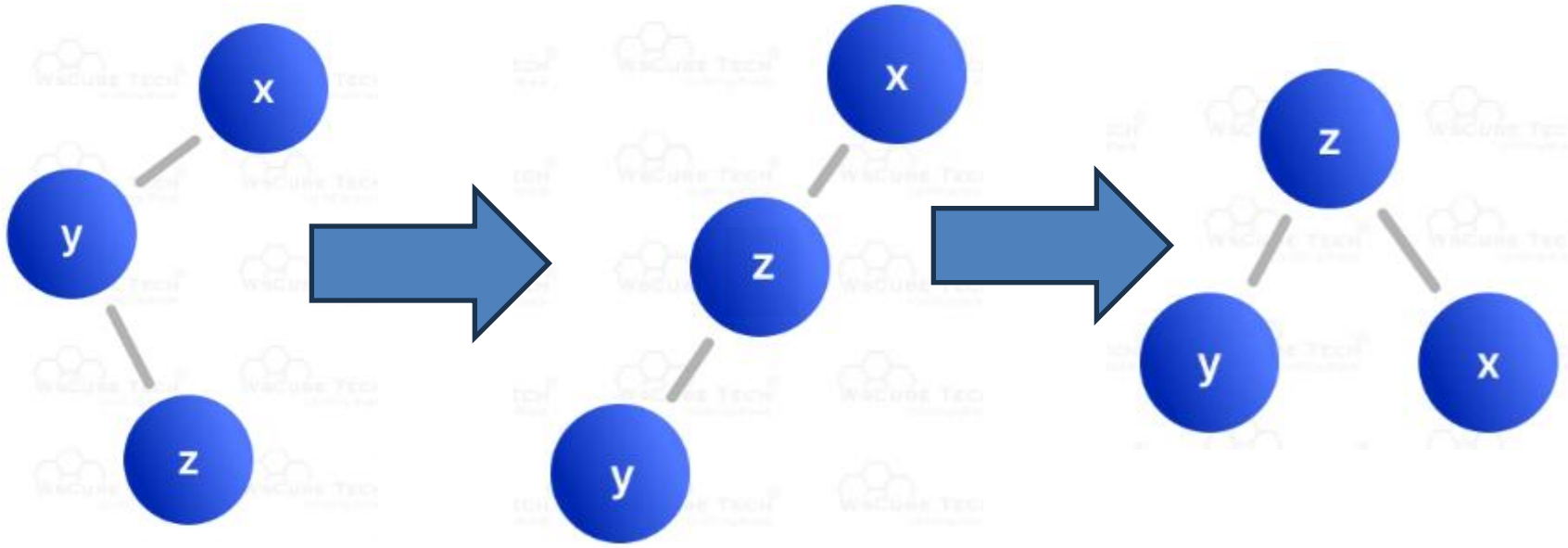
Right Rotation

- When a node is inserted into the left subtree of the left subtree, causing an imbalance.
- The unbalanced node's **left child** becomes the new root, and the original root becomes the new right child of the left child.



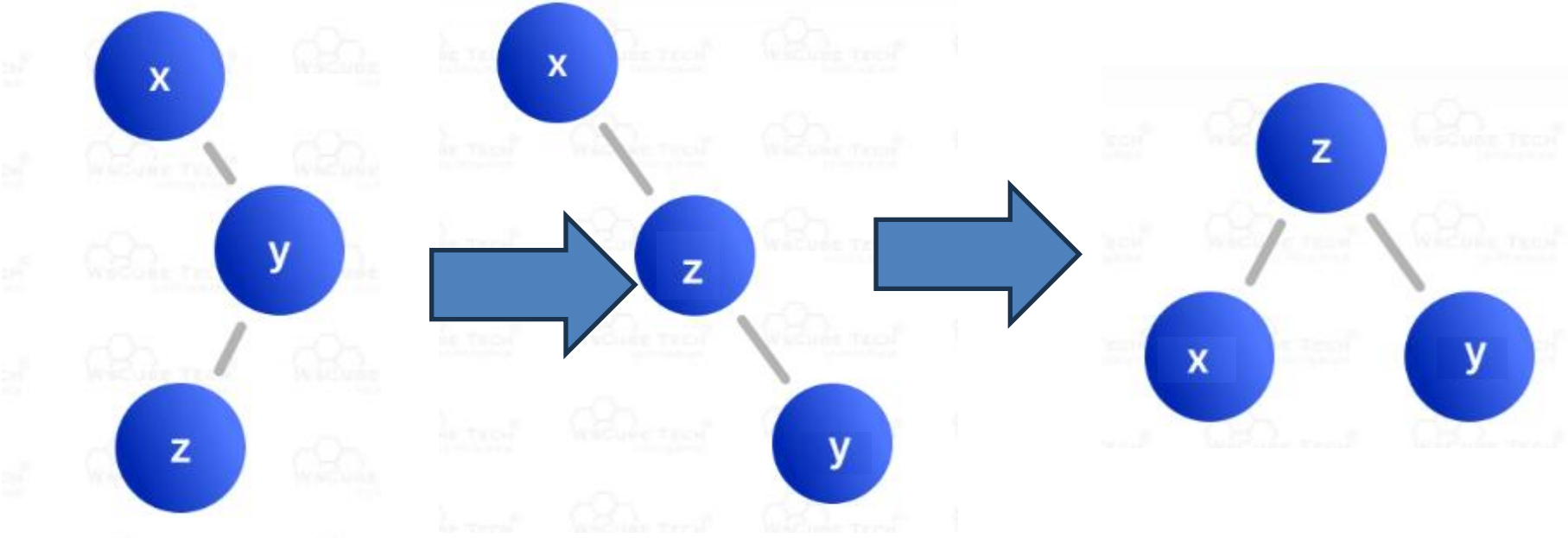
Left-Right Rotation (LR Rotation)

- When a node is inserted into the right subtree of the left subtree, causing an imbalance.
- First, a left rotation is performed on the left child of the unbalanced node, then a right rotation is performed on the unbalanced node itself.

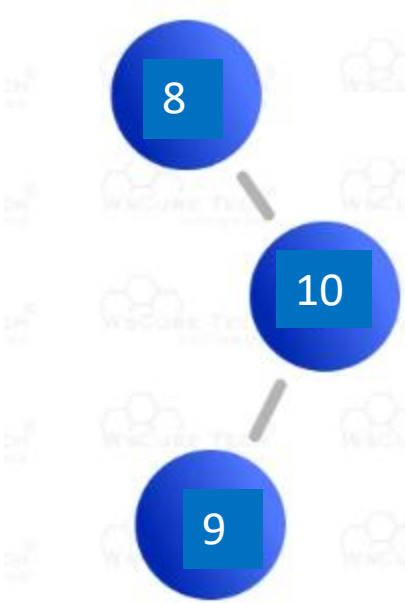


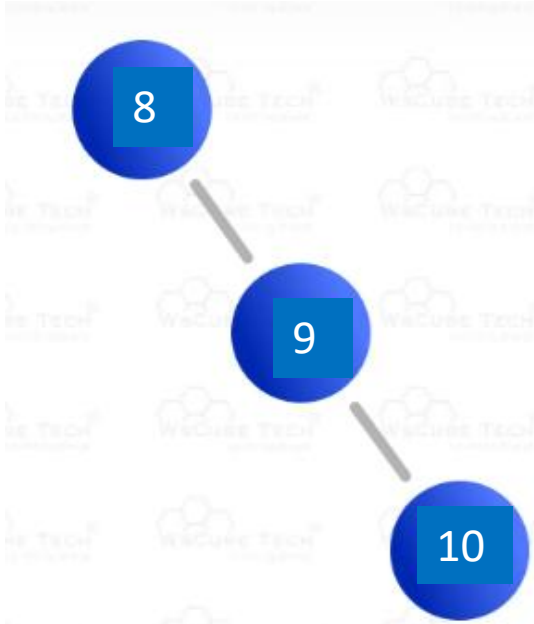
Right-Left Rotation (RL Rotation)

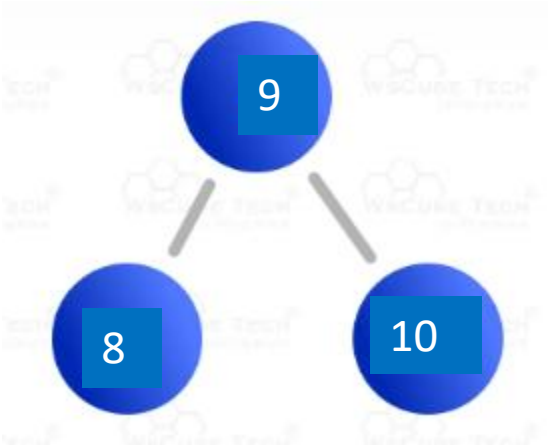
- When a node is inserted into the left subtree of the right subtree, causing an imbalance.
- First, a right rotation is performed on the right child of the unbalanced node, then a left rotation is performed on the unbalanced node itself.



example







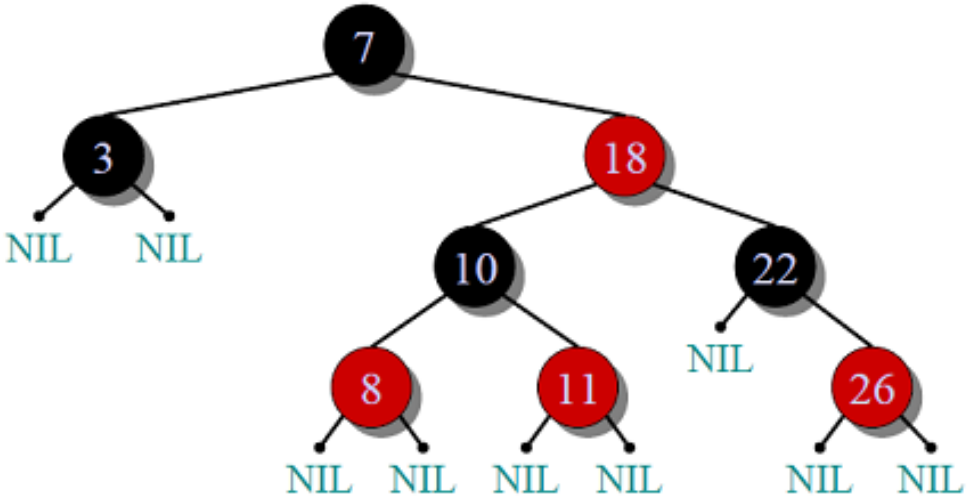
Red-Black Trees

A binary tree with one extra attribute for each node: the colour i.e. red or black.

Root of the tree and leaves (NIL) are black.

There are no two adjacent red nodes i.e. red nodes cannot have a red parent or red child.

Any path from a node (including root) to any of its descending NIL node has the same number of black nodes.



Source: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

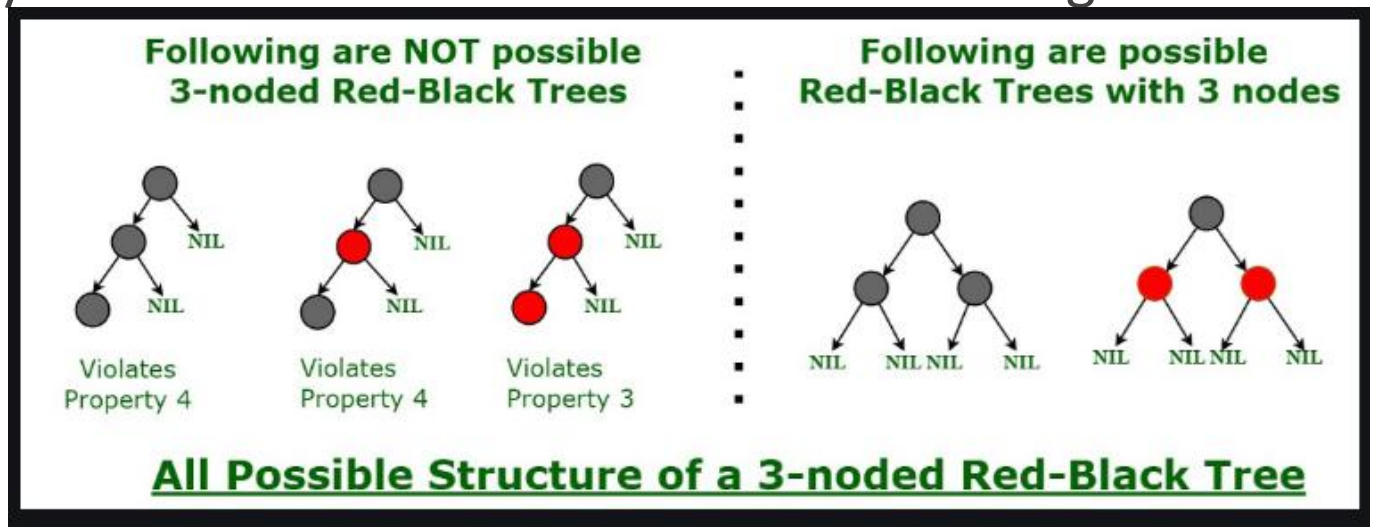
Red Black trees

Each node of the tree now contains the attributes $colour$, key, $left$, $right$, and p.

If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. Think of these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree.

Properties of red-black trees (1)

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

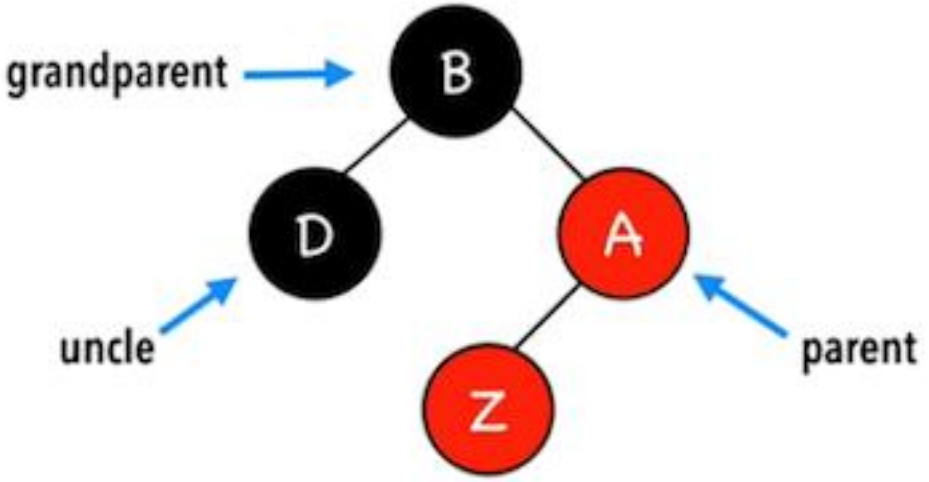


1. The root is **black**.
2. Every leaf (NIL) is **black**
3. If a node is **red**, then both its children are **black**.
4. For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

Red-Black Trees: Insertion

Insert Z and colour it red

Recolour and rotate nodes to fix violation



4 possible scenarios:

Z = root

Z's uncle = red

Z's uncle = black (triangle)

Z's uncle = black (line)

Insertion in RB Tree

New node must be inserted with the color red.

After every insertion operation, we need to check all the properties of red black tree.

If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it a red black tree.

1. Recolor
2. Rotation
3. Rotation followed by recolor.

Pseudocode for RB tree

Check whether tree is empty.

- If tree is empty then insert the new node as root node with colour black and exit

If tree is not empty then insert the new node as leaf node with colour red.

- If the parent of new node is black then exit

If the parent of new node is red then change the colour of uncle of new node.

- If it is coloured black or null then make suitable rotation and recolour it.
- If it is coloured red then perform recolour.

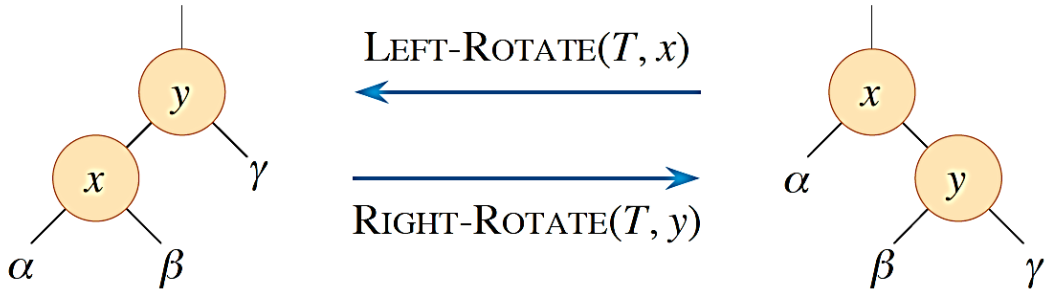
Rotations

The next slide shows the two kinds of rotations: left rotations and right rotations. Let's look at a left rotation on a node x, which transforms the structure on the right side of the figure to the structure on the left. Node x has a right child y, which must not be $T.nil$.

The left rotation changes the subtree originally rooted at x by "twisting" the link between x and y to the left. The new root of the subtree is node y, with x as y's left child and y's original left child (the subtree represented by β in the figure) as x's right child.

The rotation operations on a binary search tree

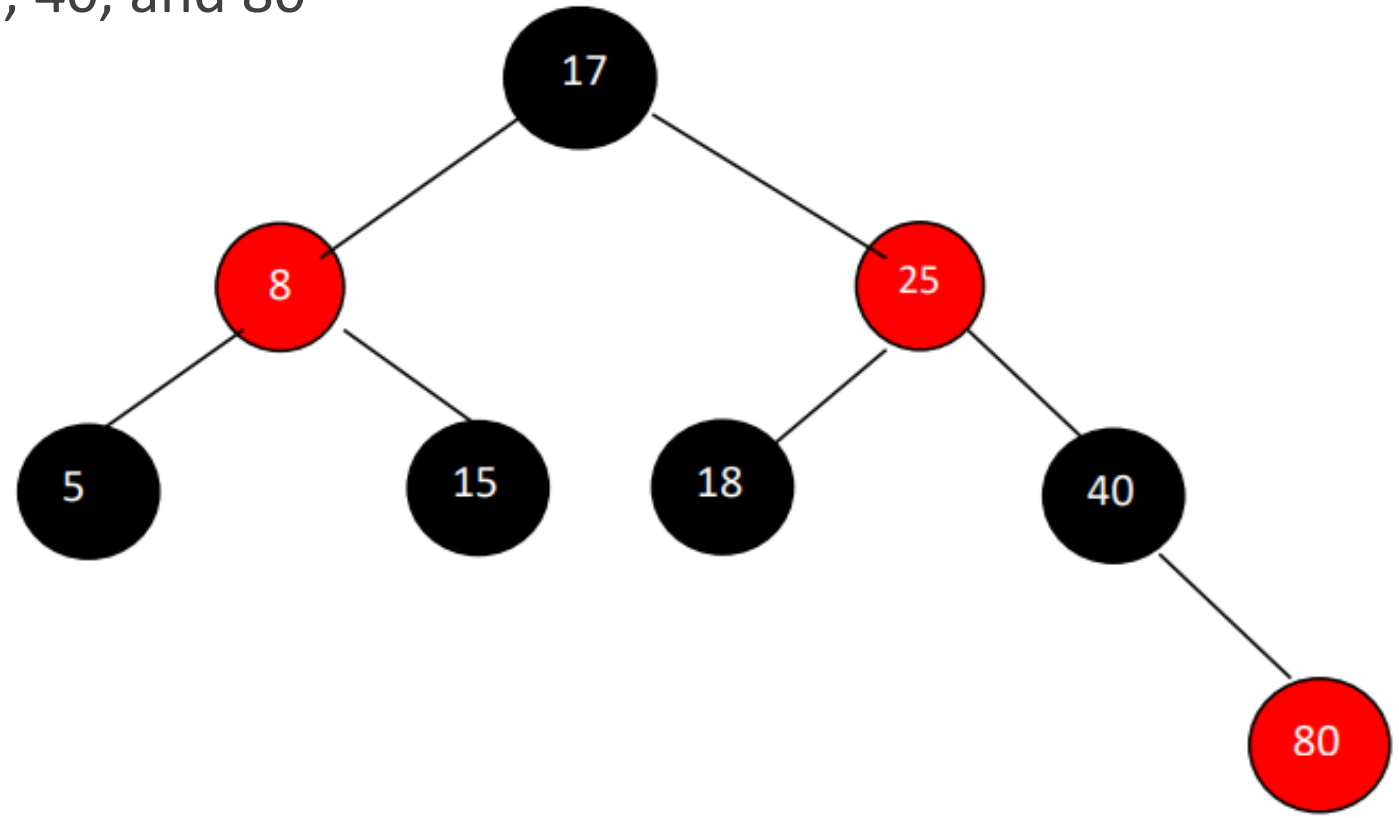
The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.



Example (based on https://rnlkwc.ac.in/pdf/studymaterial/comsc/Design.pdf)

Create a red black tree by inserting following sequence of number :-

8, 18, 5, 15, 17, 25, 40, and 80

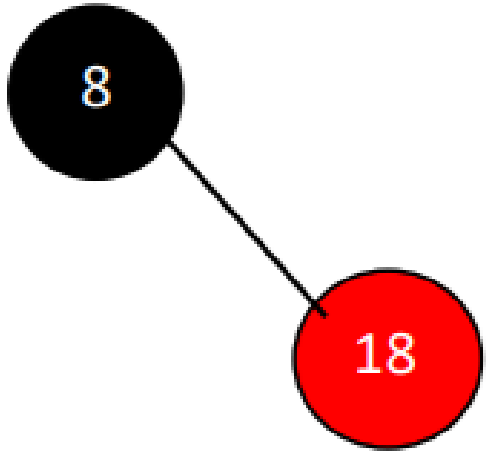


Insert (8)

Tree is empty. So insert newnode as root node with black color.

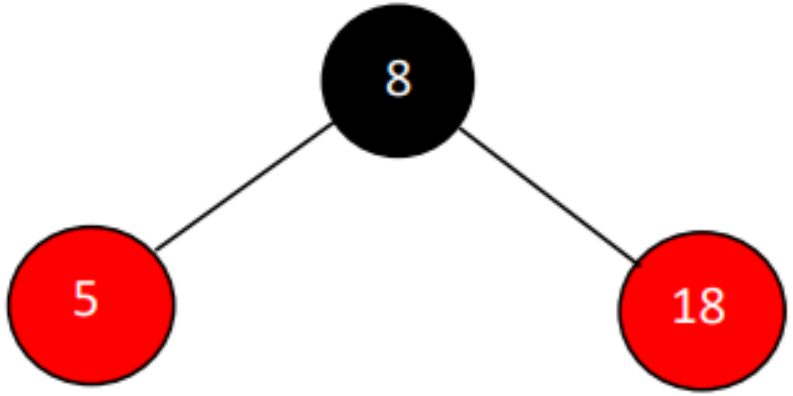
Insert (18)

Tree is not empty. So insert newnode with red color.



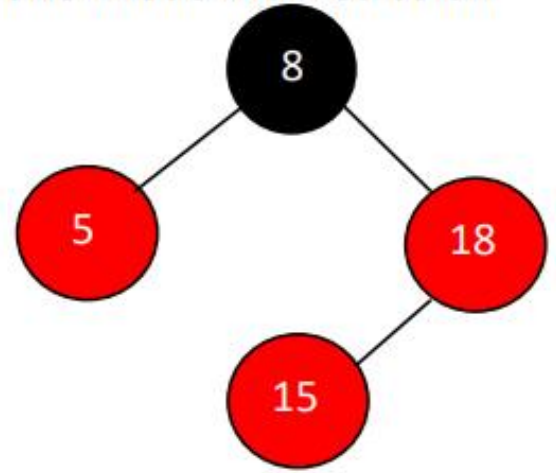
Insert (5)

Tree is not empty. So insert newnode with red color.



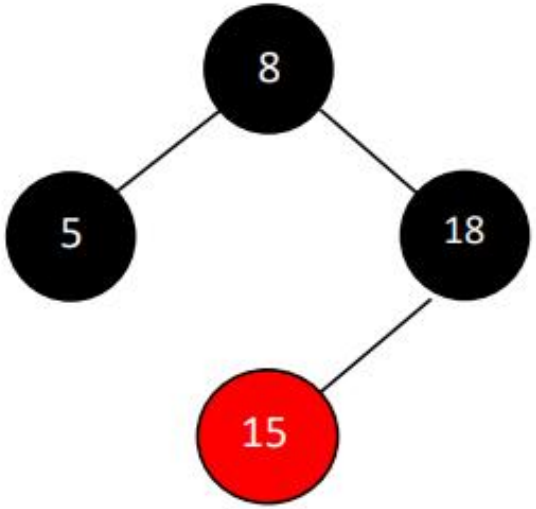
Insert (15)

Tree is not empty. So insert newnode with red color.



Here there are two consecutive red nodes 18 and 15. The newnode's parent sibling color is red and parent's parent is root node. So we use recolor to make it red black tree.

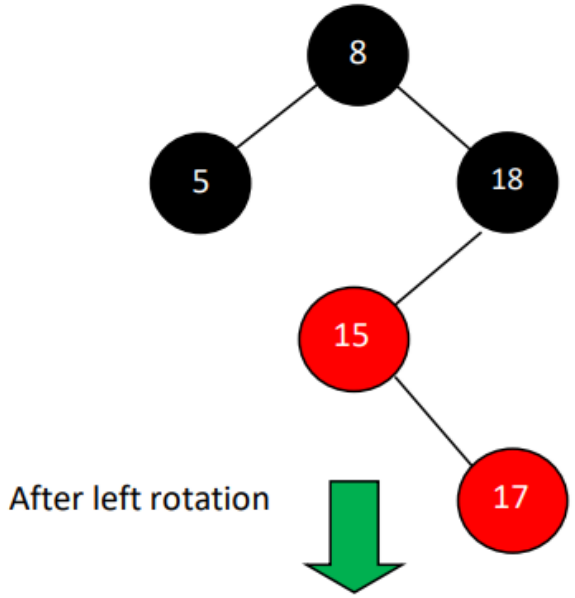
After recolor



After recolor operation, the trees satisfying all red black tree properties.

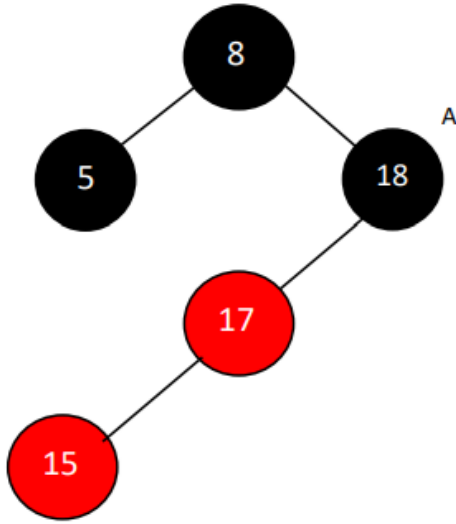
Insert (17)

The tree is not empty. So insert newnode with red color.

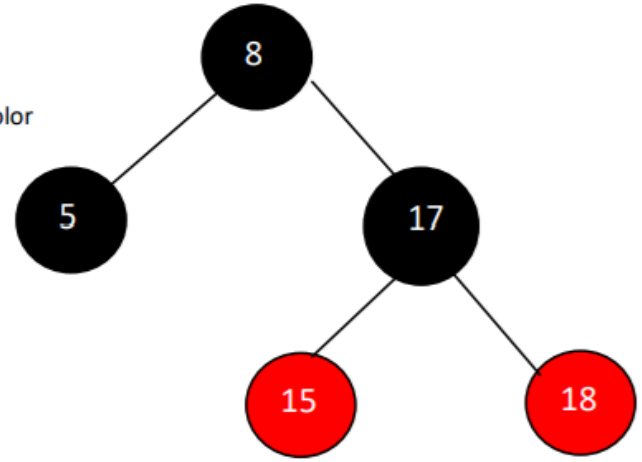


After left rotation

Here there two consecutive red nodes 15 and 17.The newnode's parent sibling is null. So we need rotation. Here we need LR rotation and recolor.

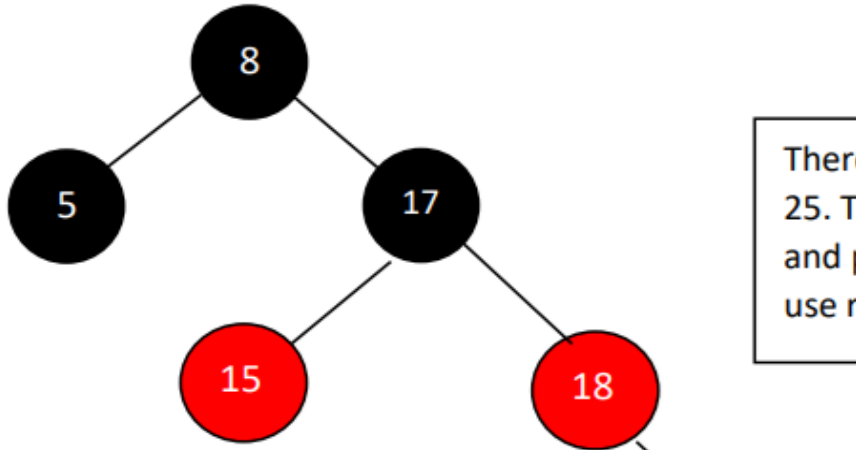


After right rotation and recolor



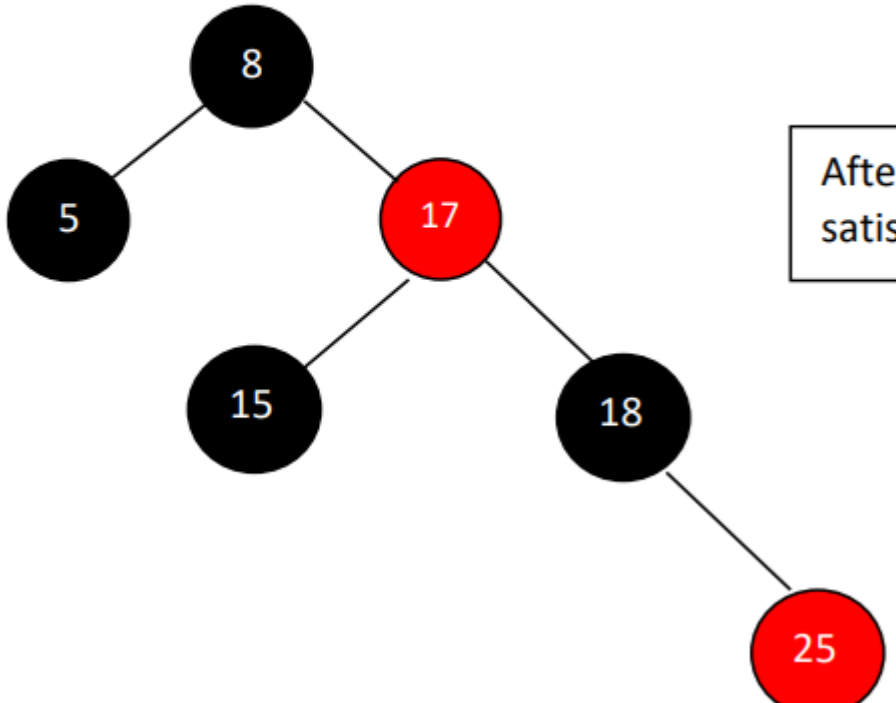
Insert (25)

Tree is not empty. So insert newnode with red color.



There are two consecutive red nodes 18 an
25. The newnode's parent sibling color is re
and parent's parent is not root node. So we
use recolor and recheck.

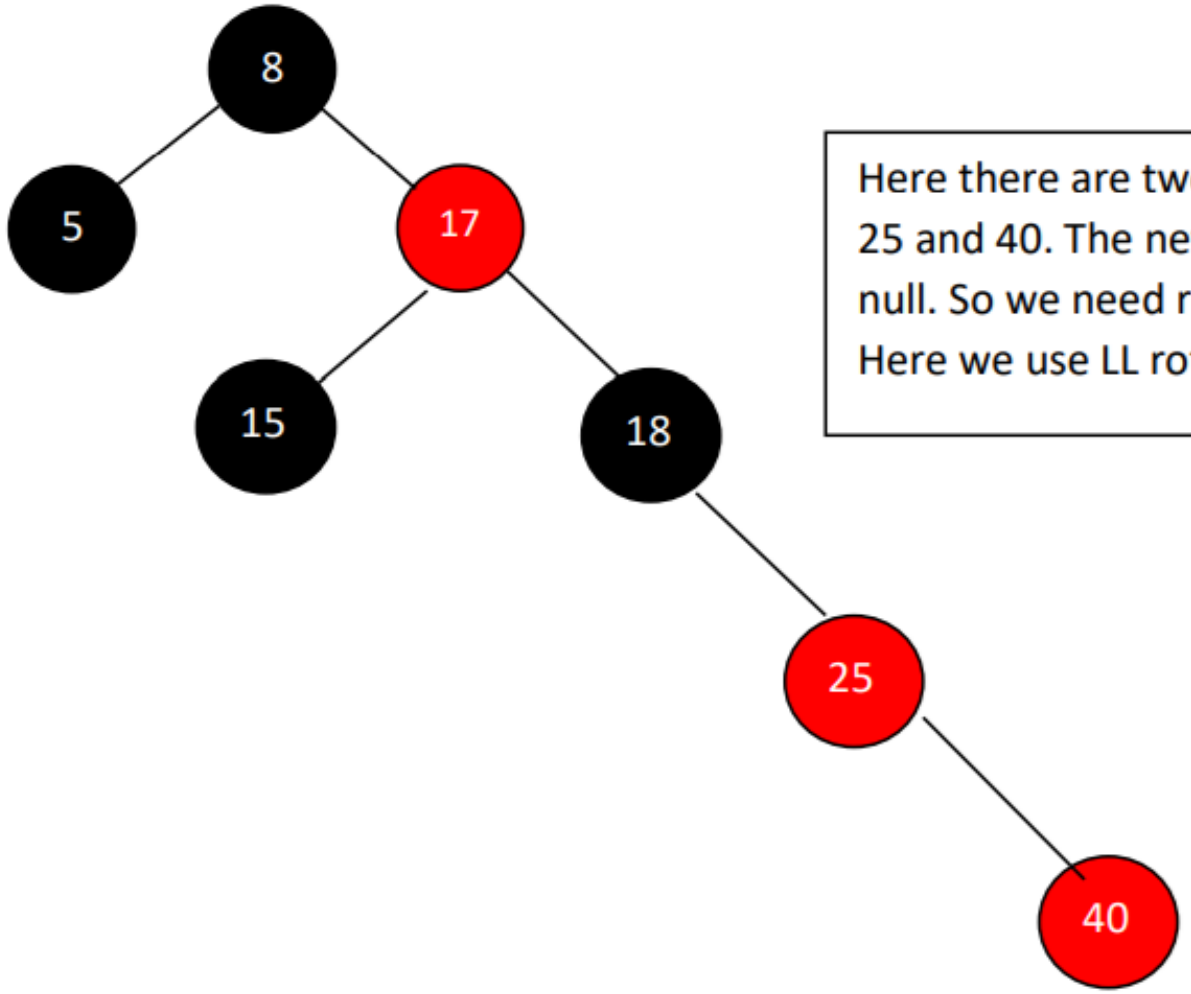
After recolor



After recolor operation, the tree is satisfying all red black tree properties.

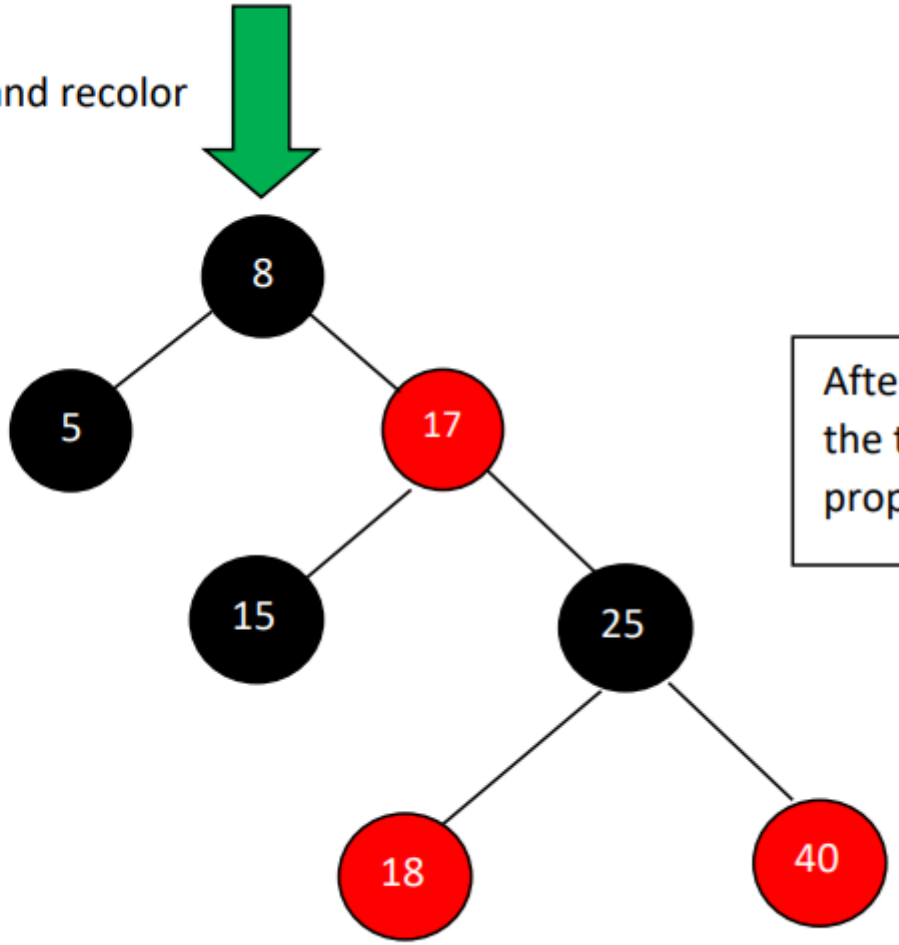
Insert(40)

Tree is not empty. So insert newnode with red color.



Here there are two consecutive red nodes 25 and 40. The newnode's parent sibling is null. So we need rotation and recolor. Here we use LL rotation and recheck.

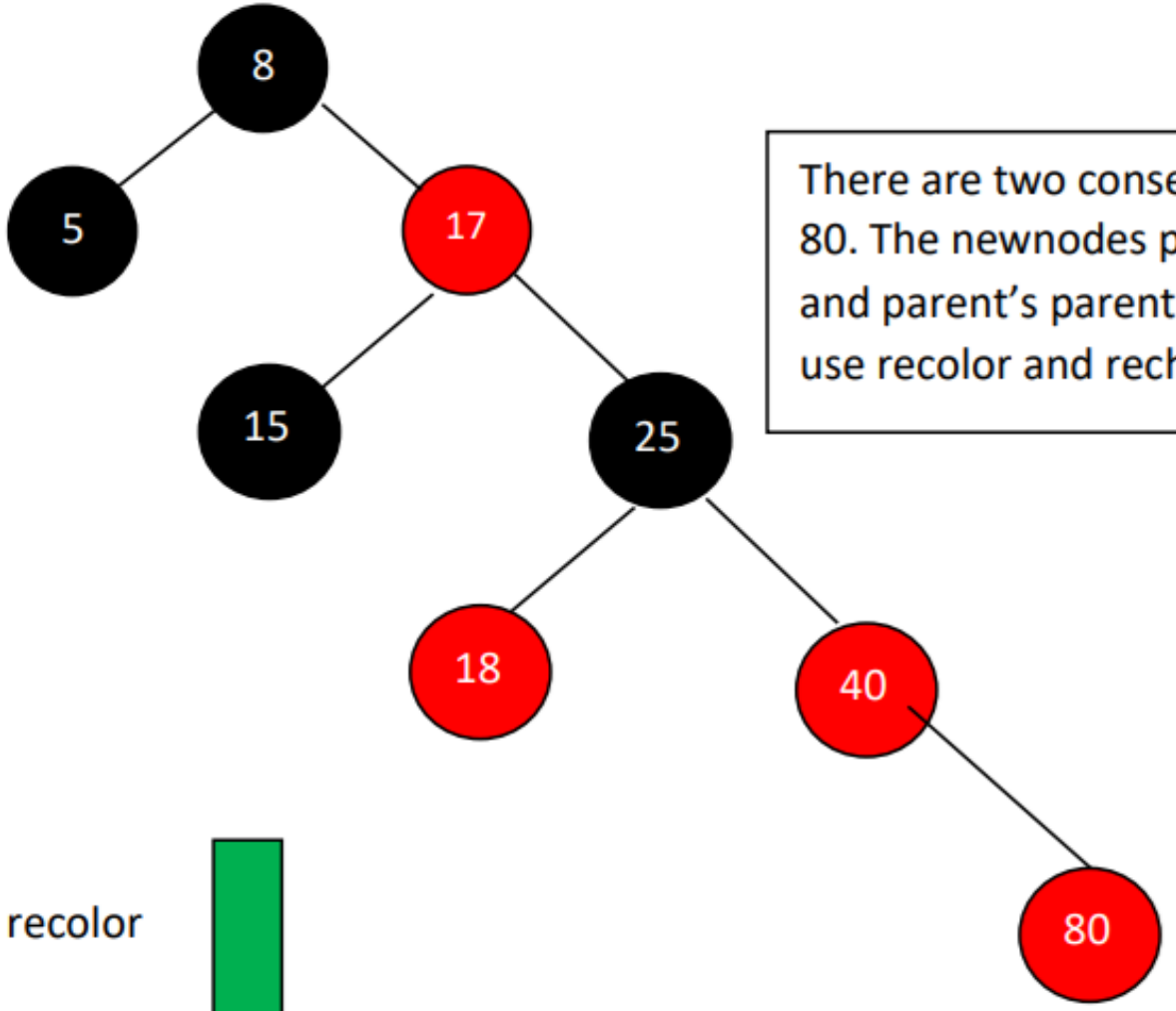
After LL rotation and recolor



After LL rotation and recolor operation, the tree satisfying all red black tree properties.

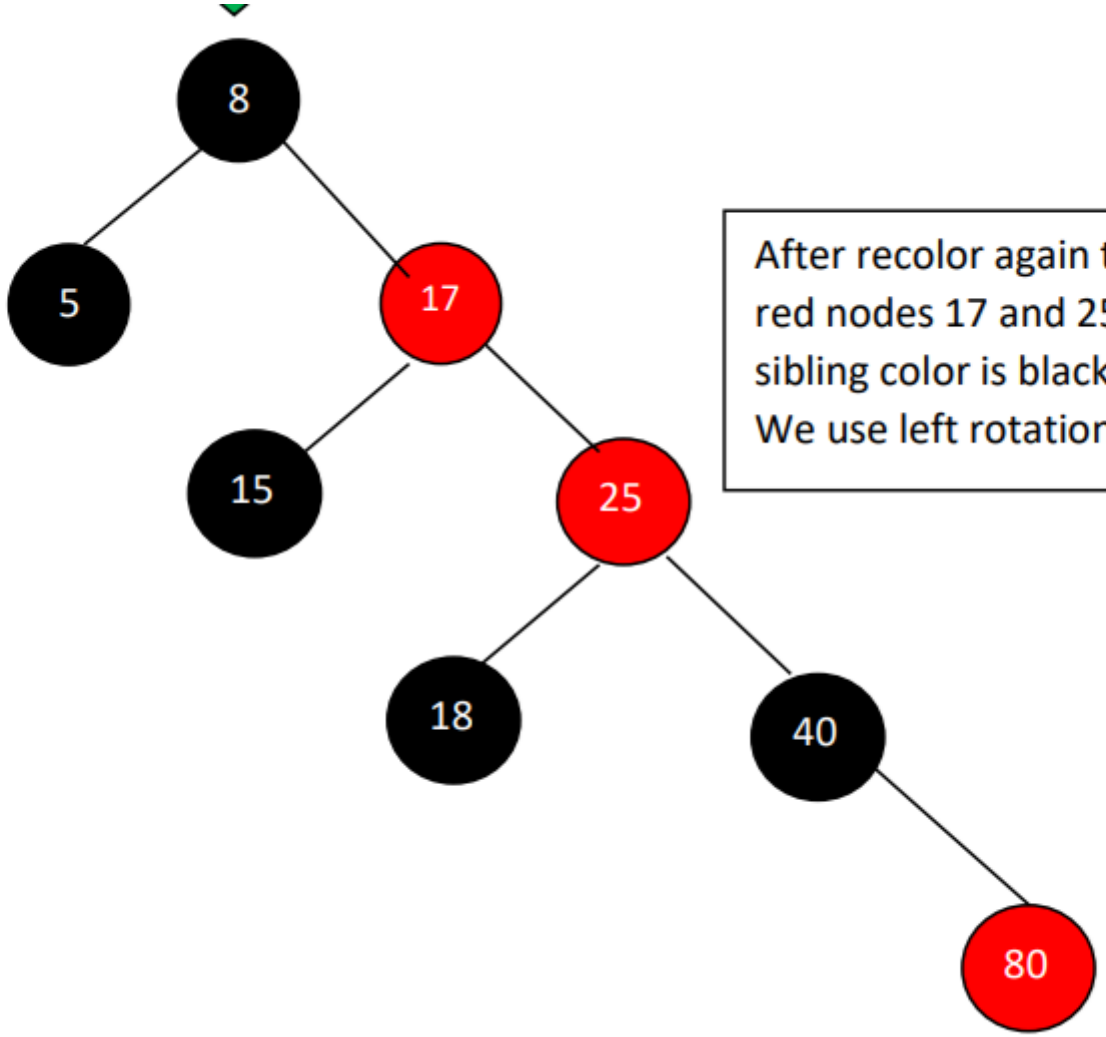
Insert(80)

Tree is not empty. So insert newnode with red color.



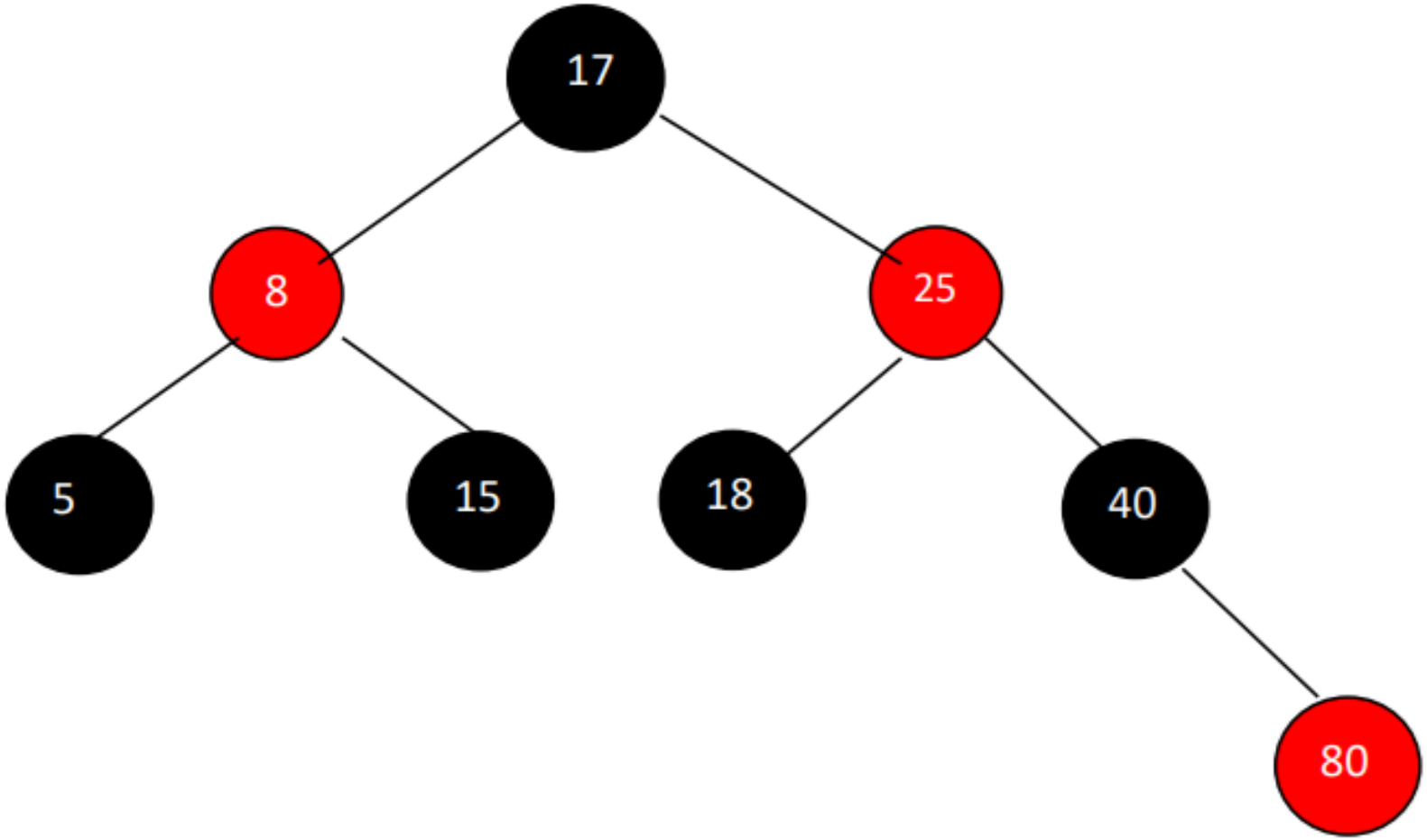
There are two consecutive red nodes 40 and 80. The newnodes parent sibling color is red and parent's parent is not root node. So we use recolor and recheck.

After recolor



After recolor again there are two consecutive red nodes 17 and 25. The newnode's parent sibling color is black. So we need rotation. We use left rotation And recolor.

After left rotation And recolor



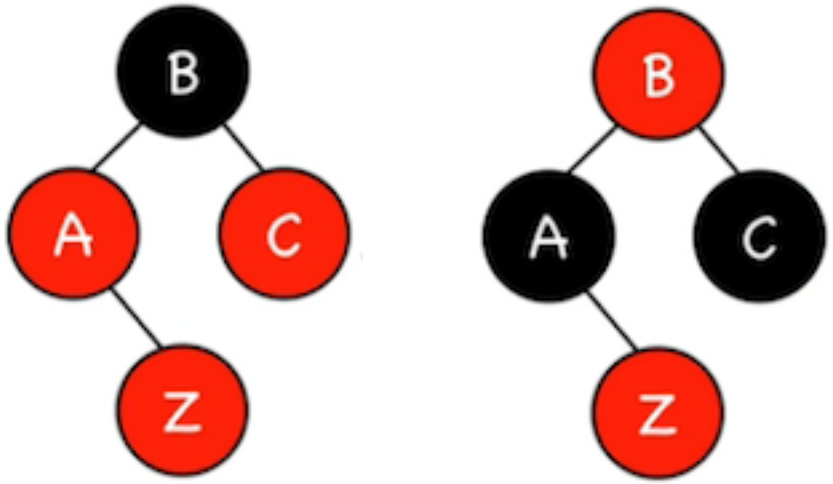
More RB slides

Possible Scenarios

Case 1: Z = root

Recolour Z to black.

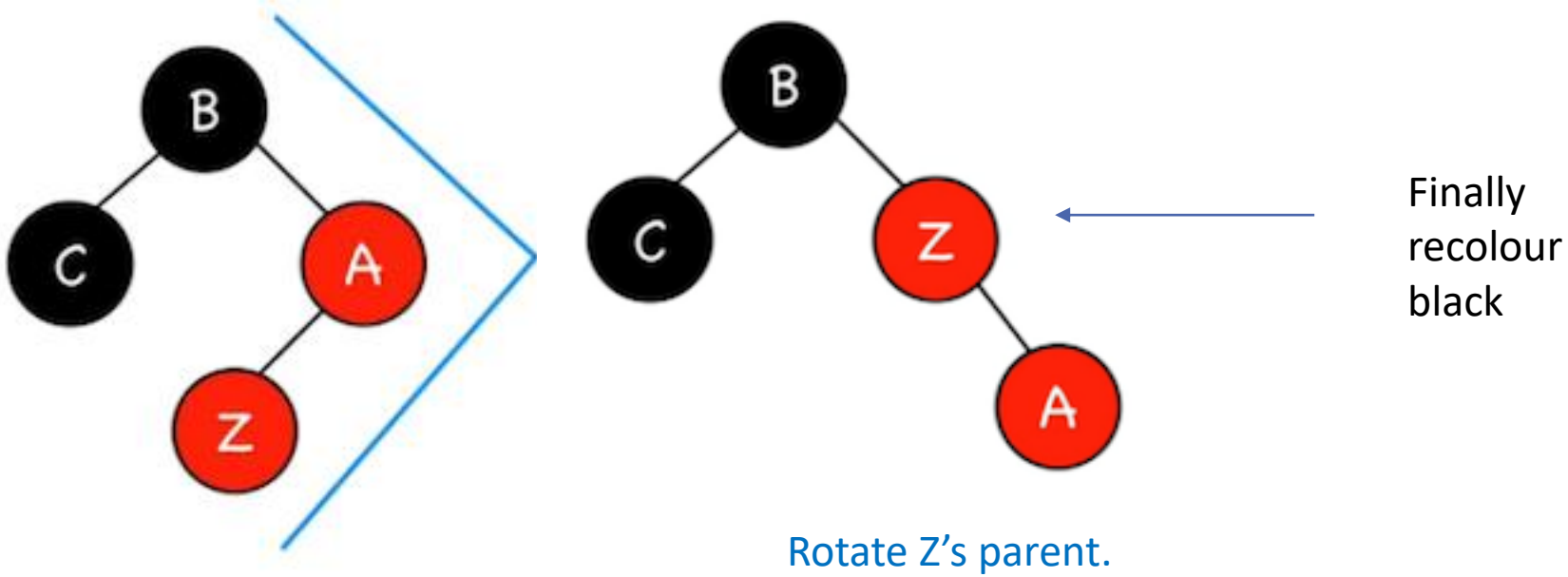
- Case 2: Z's uncle = red



Recolour Z's parent, uncle to black and grandparent to red

Four Possible Scenarios (2)

Case 3: Z's uncle = black (triangle)



Example #4

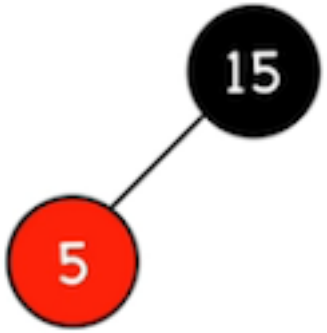
Add 15, and colour it red.

Another example!!

Example #4

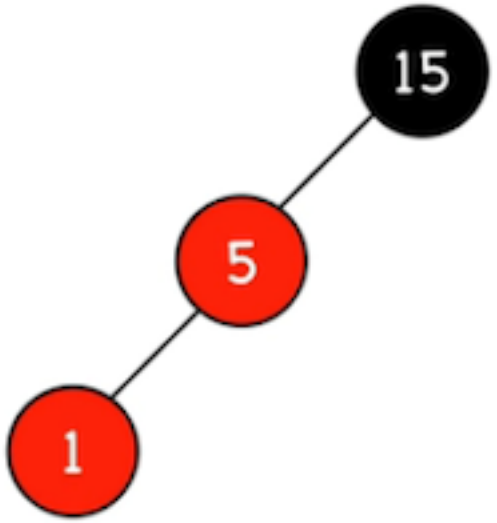
- Add 15, and colour it red.
- Recolour 15 to black because it is root.

Example #4



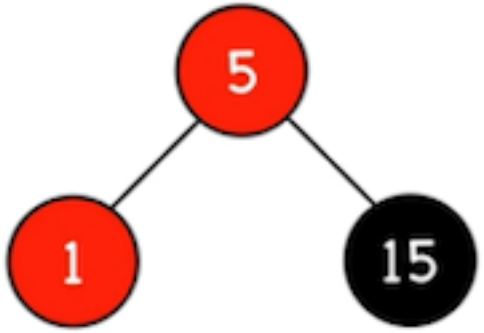
- Add 15, and colour it red.
- Recolour 15 to black because it is root.
- Add 5, and colour it red (no violation).

Example #4



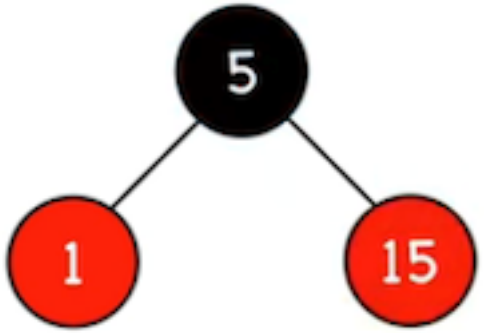
- Add 15, and colour it red.
- Recolour 15 to black because it is root.
- Add 5, and colour it red (no violation).
- Add 1, and colour it red (violation!!).
- Uncle of 1 (nil) is black, so rotate the grandparent.

Example #4



- Add 15, and colour it red.
- Recolour 15 to black because it is root.
- Add 5, and colour it red (no violation).
- Add 1, and colour it red (violation!!).
- Uncle of 1 (nil) is black, so rotate the grandparent.
- 5 is now root, but it is red (violation!!!).

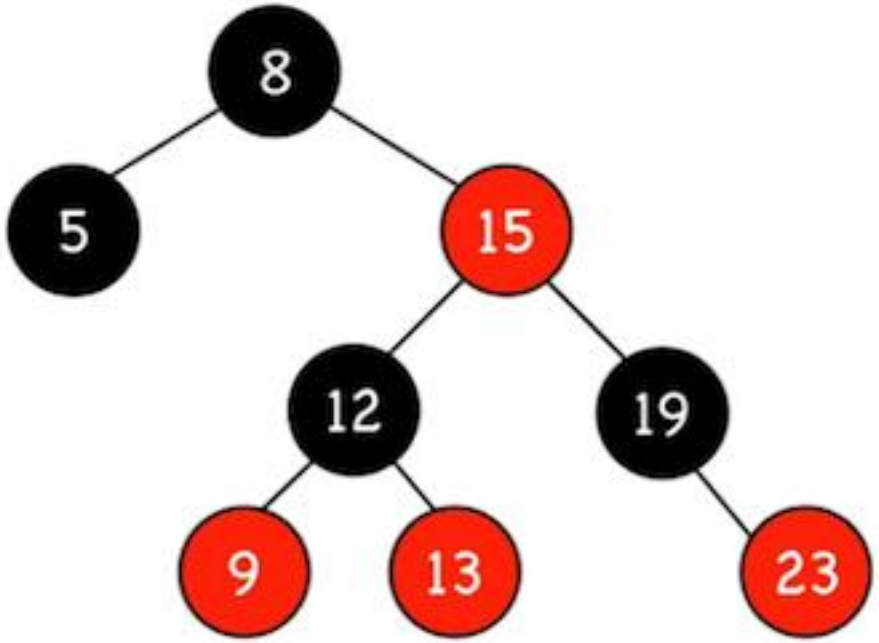
Example #4

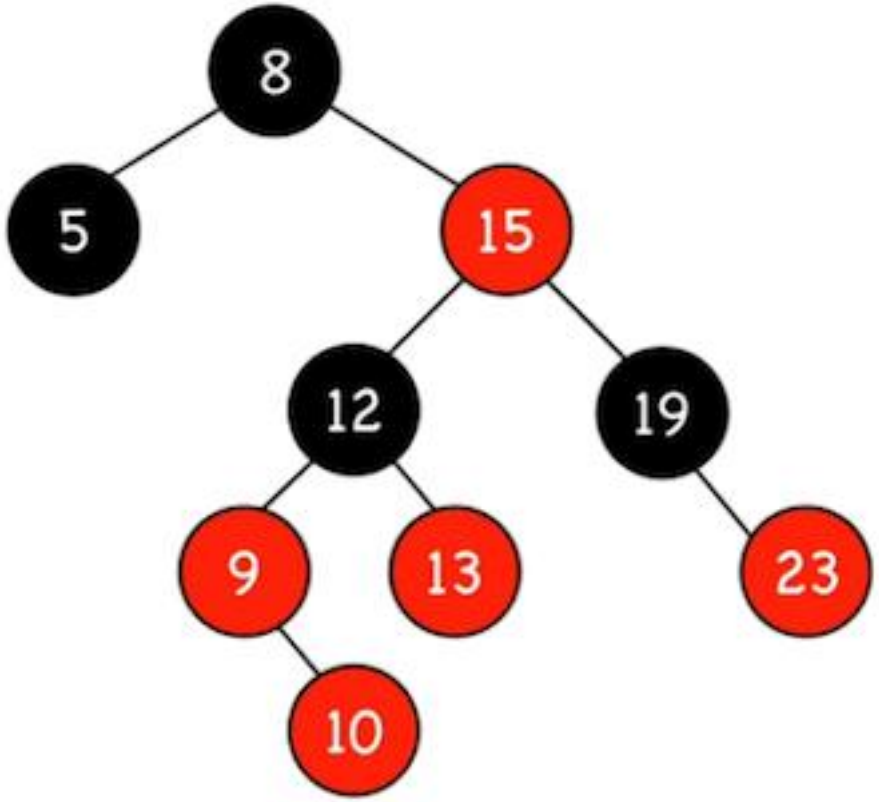


- Add 15, and colour it red.
- Recolour 15 to black because it is root.
- Add 5, and colour it red (no violation).
- Add 1, and colour it red (violation!!).
- Uncle of 1 (nil) is black, so rotate the grandparent.
- 5 is now root, but it is red (violation!!!).
- Recolour the grandparent 15 and the parent 5.

Example #5

Add 10 to the red-black tree.



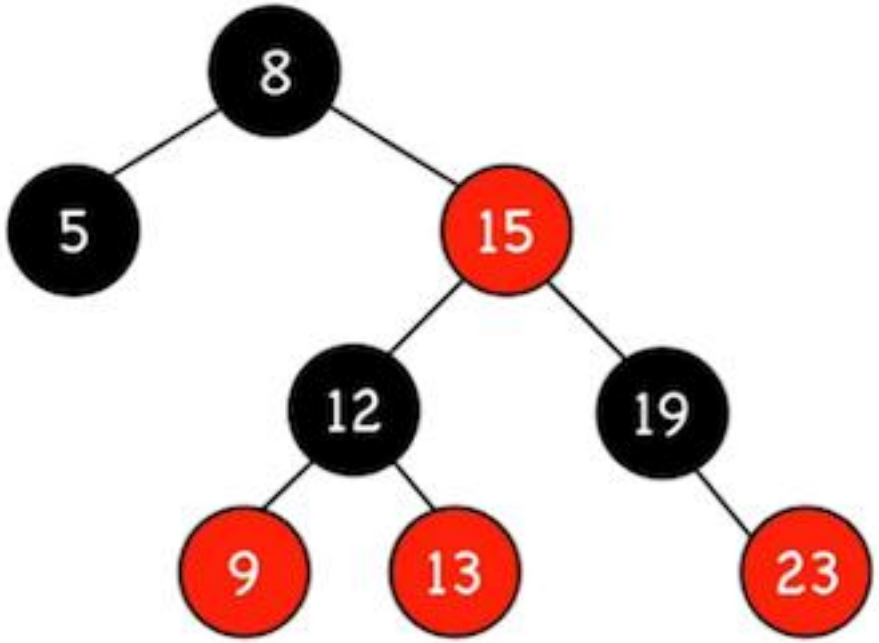


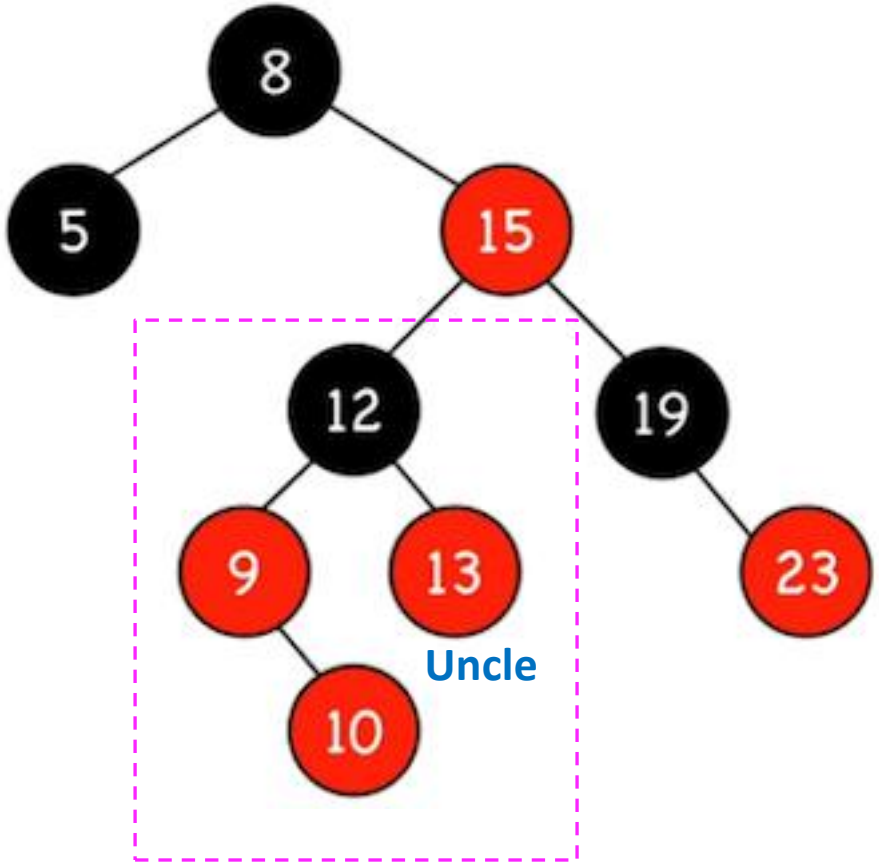
#1

#2

Example #5

Add 10 to the red-black tree.



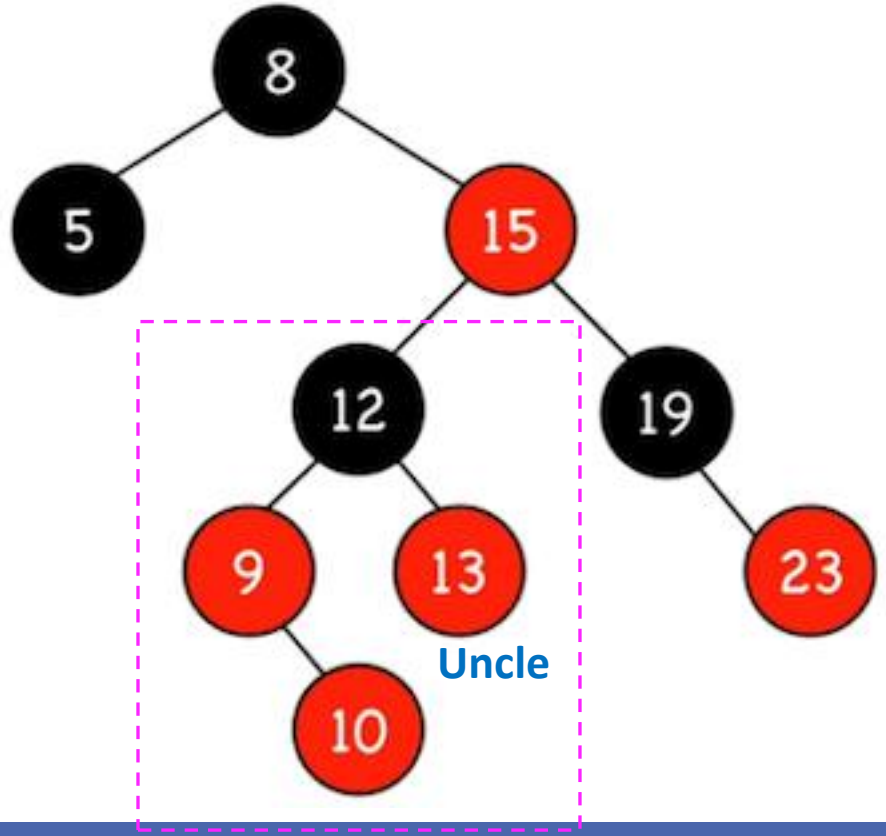


#1

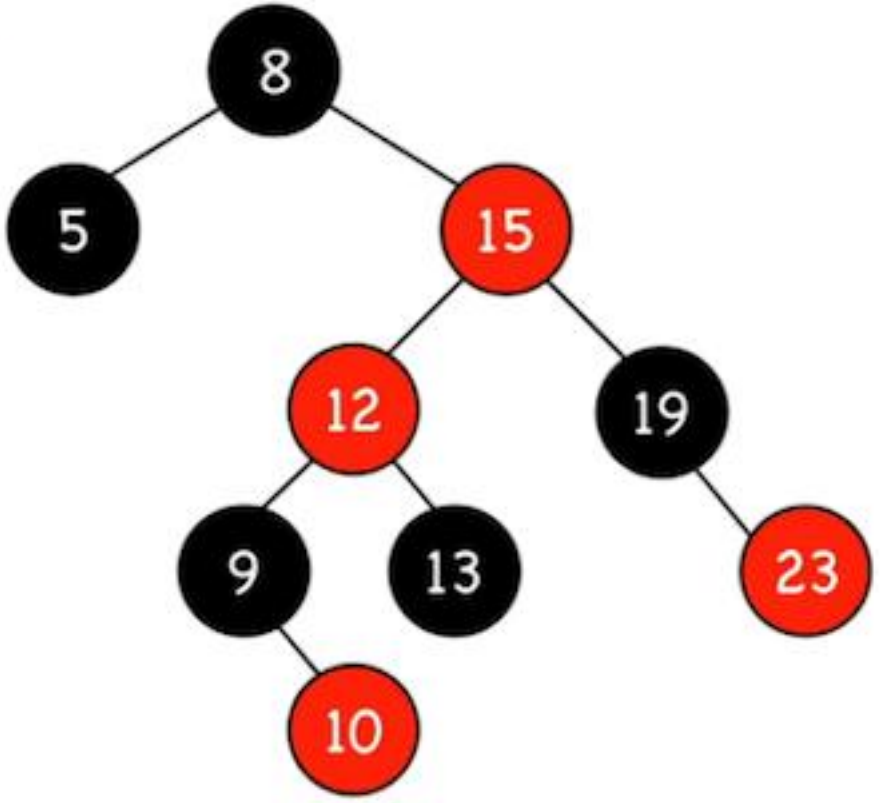
#2

Example #5

Uncle was red (violation) -> recolour.



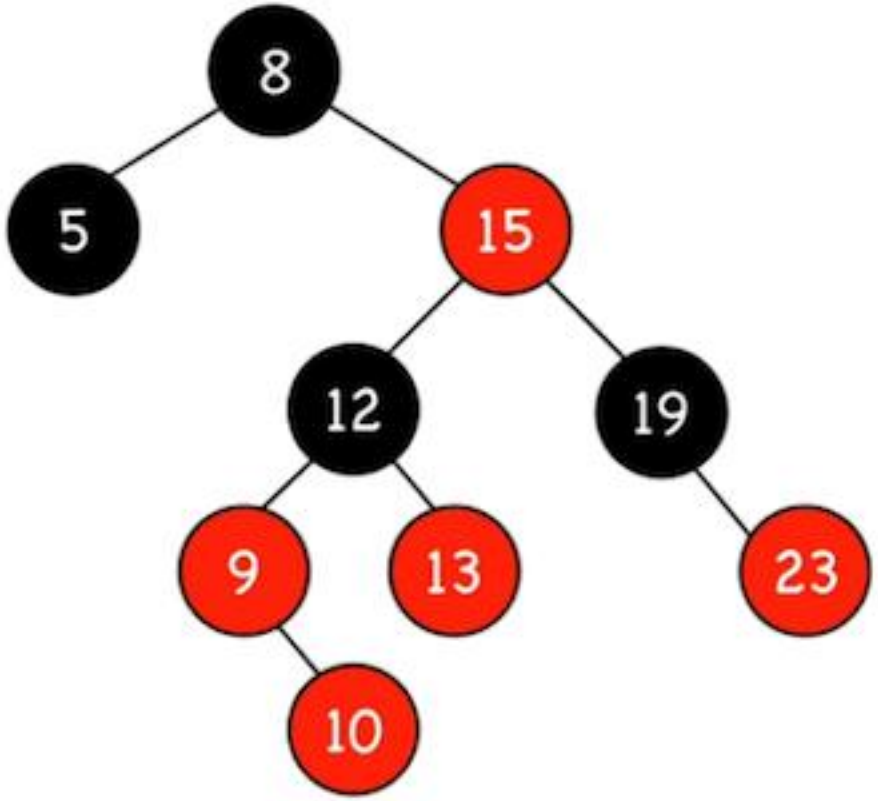
#2



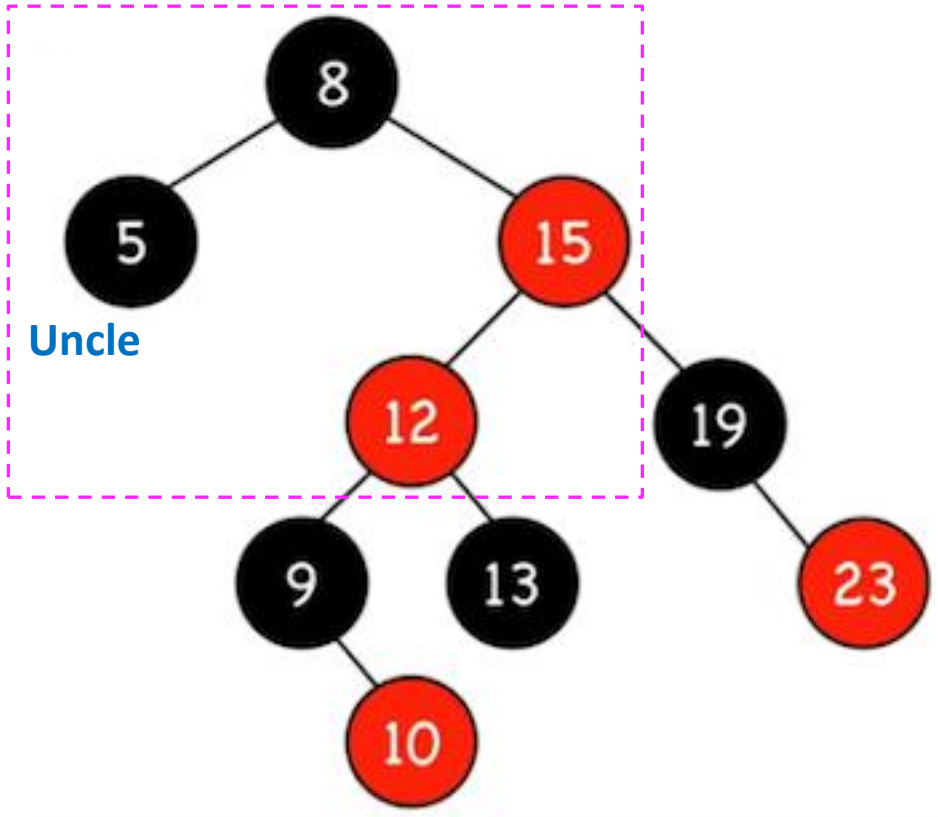
#3

Example #5

Uncle was red (violation) -> recolour.



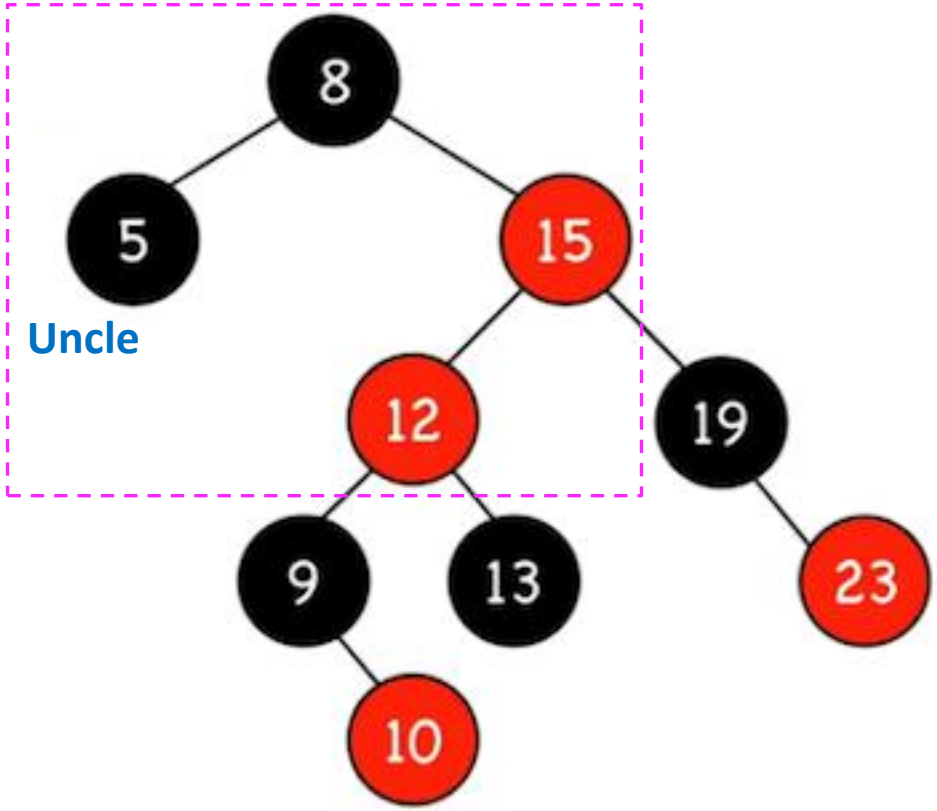
#2

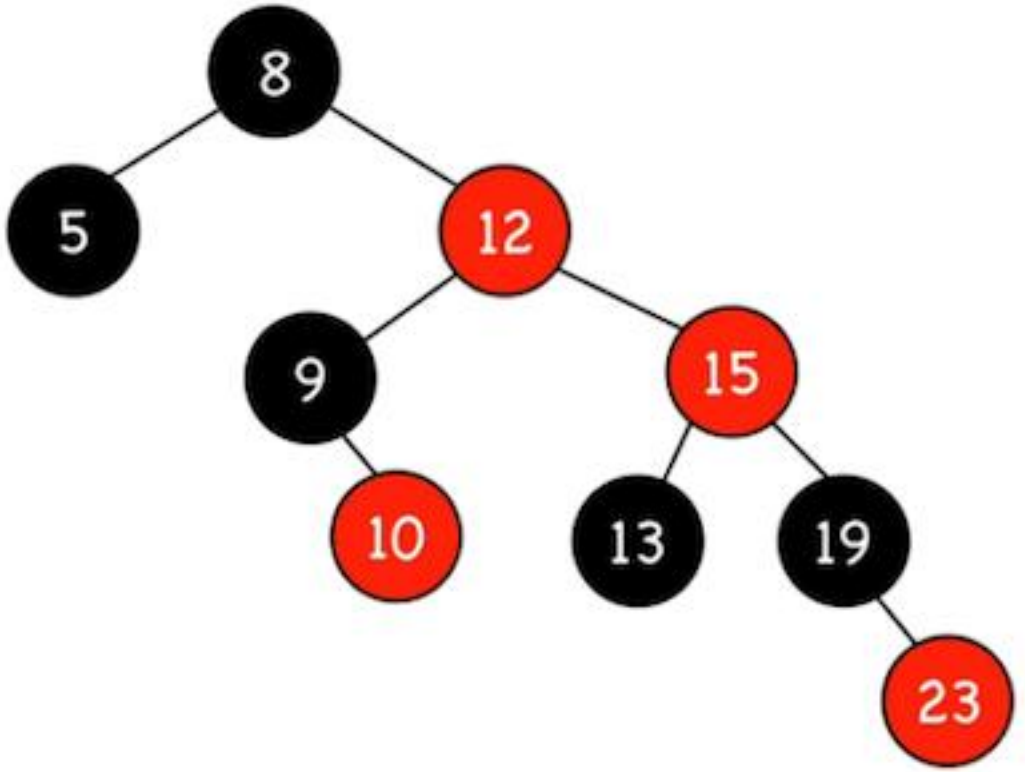


#3

Example #5

Uncle was black (violation) -> rotate.



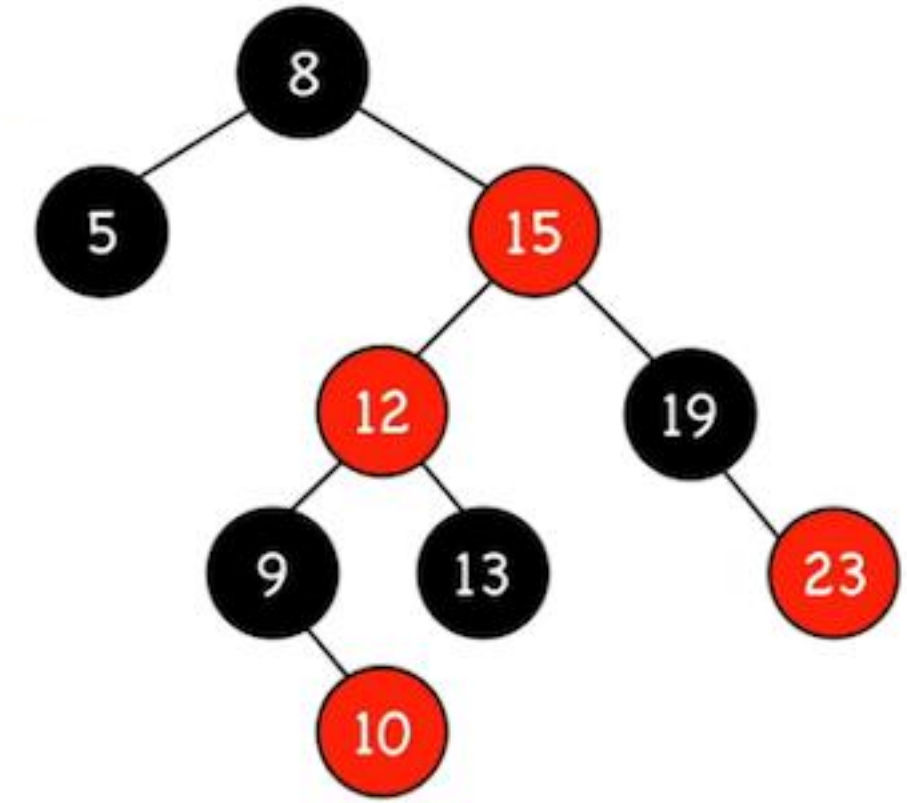


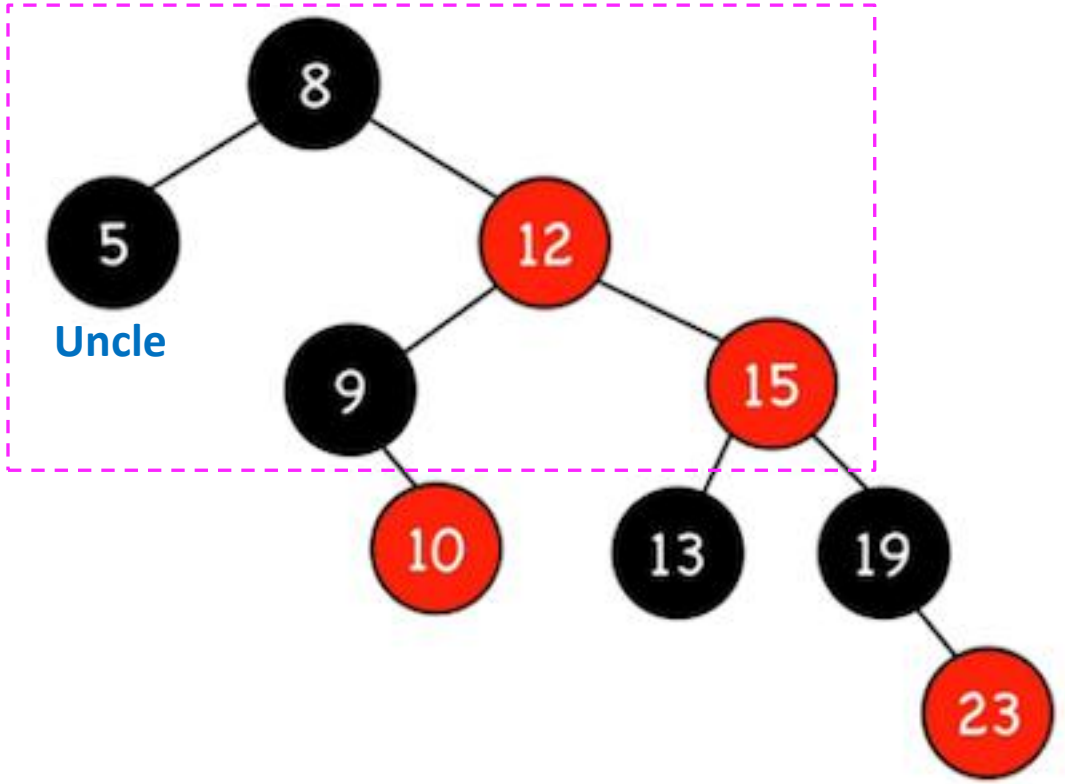
#3

#4

Example #5

Uncle was black (violation) -> rotate.



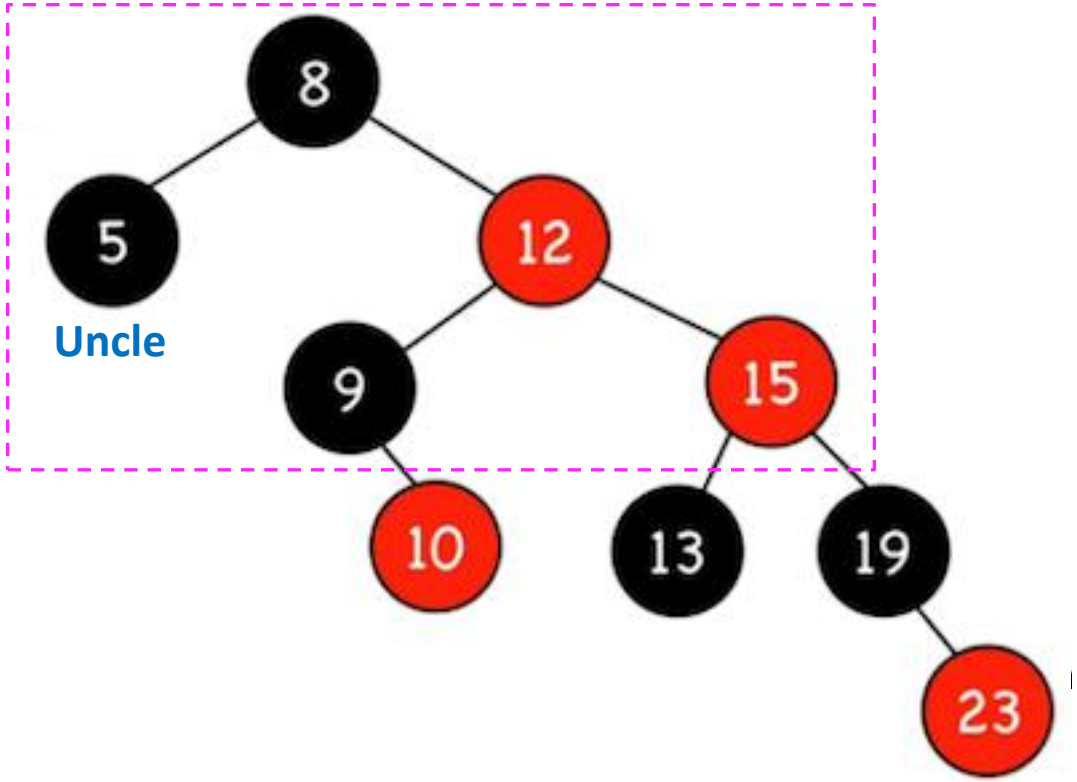


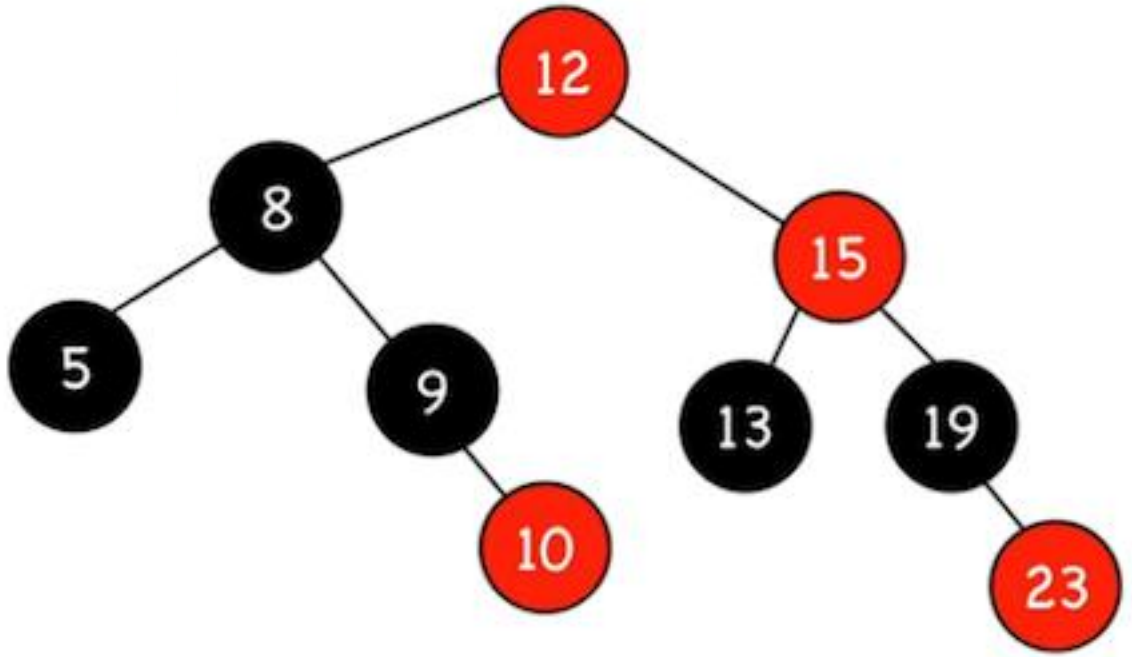
#3

#4

Example #5

Uncle was black (violation) -> rotate.



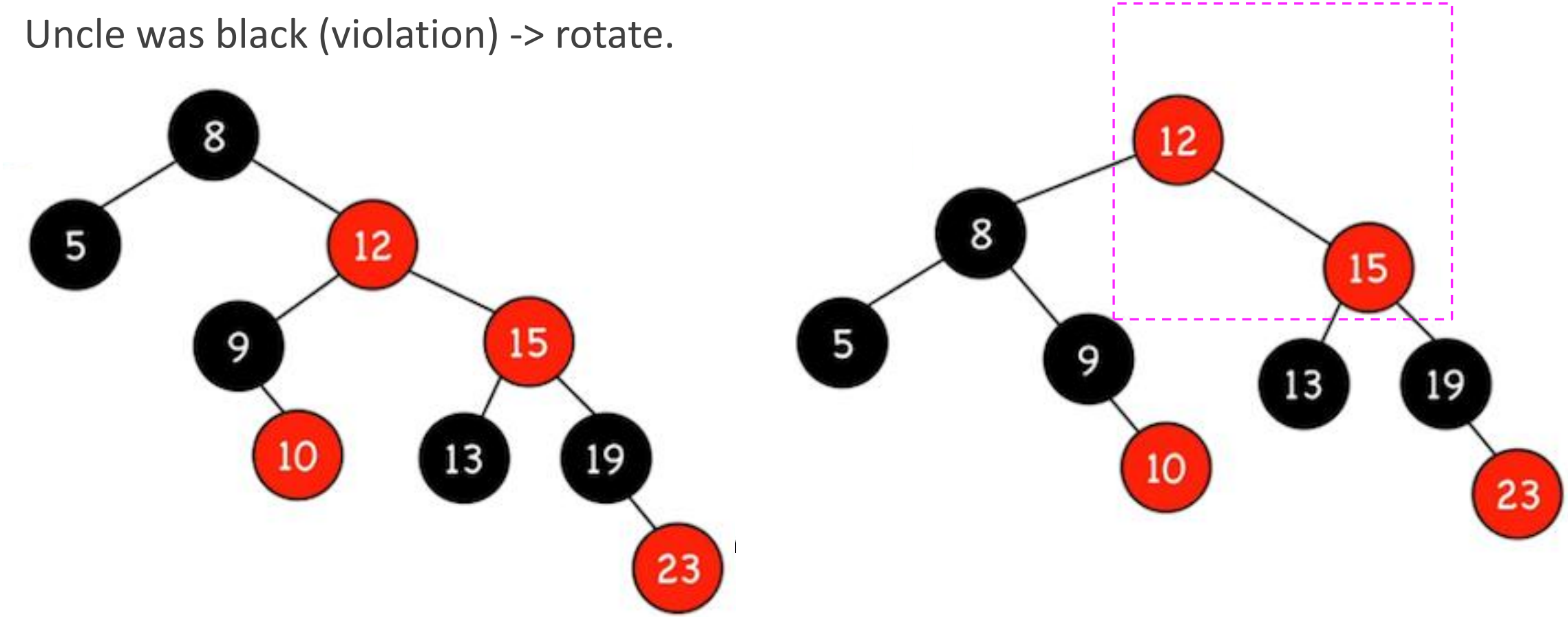


#4

#5

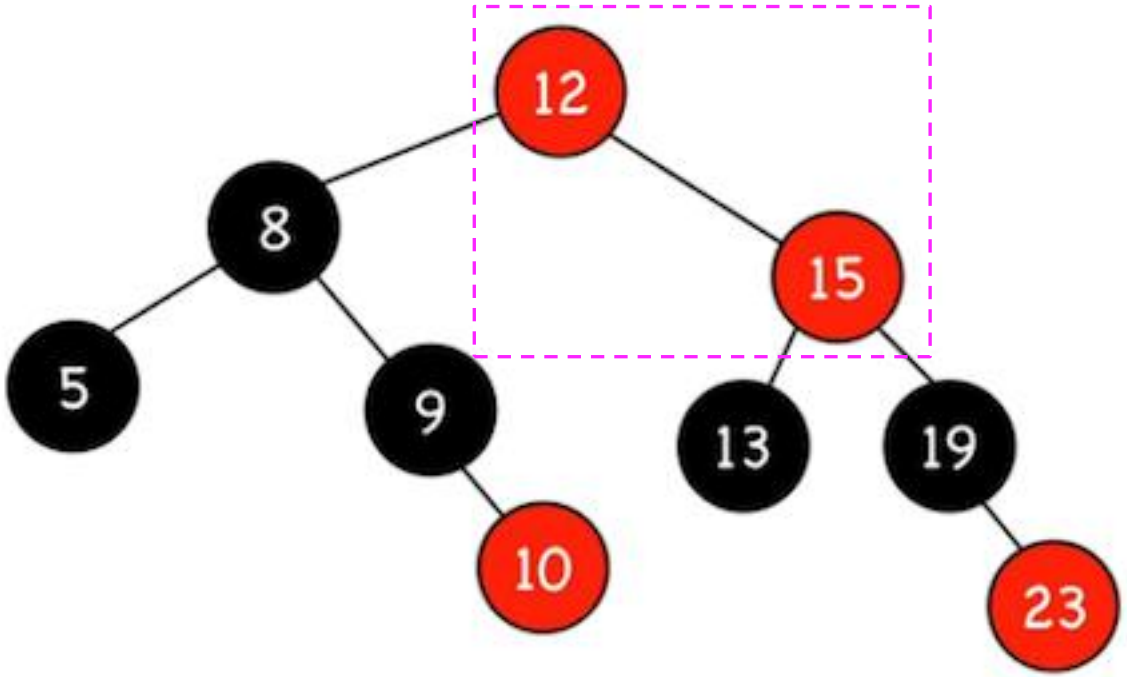
Example #5

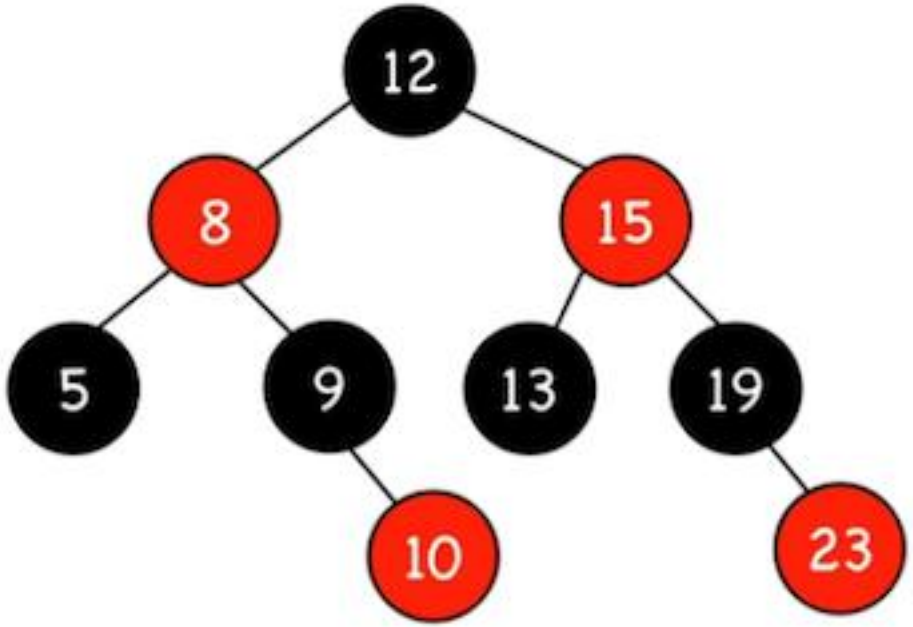
Uncle was black (violation) -> rotate.



Example #5

Adjacent nodes are red -> recolour.





#5

#6

LEFT-ROTATE pseudocode

```
LEFT-ROTATE(T, x)
 1  y = x.right
 2  x.right = y.left        // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil       // if y's left subtree is not empty ...
 4      y.left.p = x        // ... then x becomes the parent of the subtree's root
 5  y.p = x.p               // x's parent becomes y's parent
 6  if x.p == T.nil         // if x was the root ...
 7      T.root = y          // ... then y becomes the root
 8  elseif x == x.p.left    // otherwise, if x was a left child ...
 9      x.p.left = y        // ... then y becomes a left child
10  else x.p.right = y      // otherwise, x was a right child, and now y is
11  y.left = x              // make x become y's left child
12  x.p = y
```

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$.

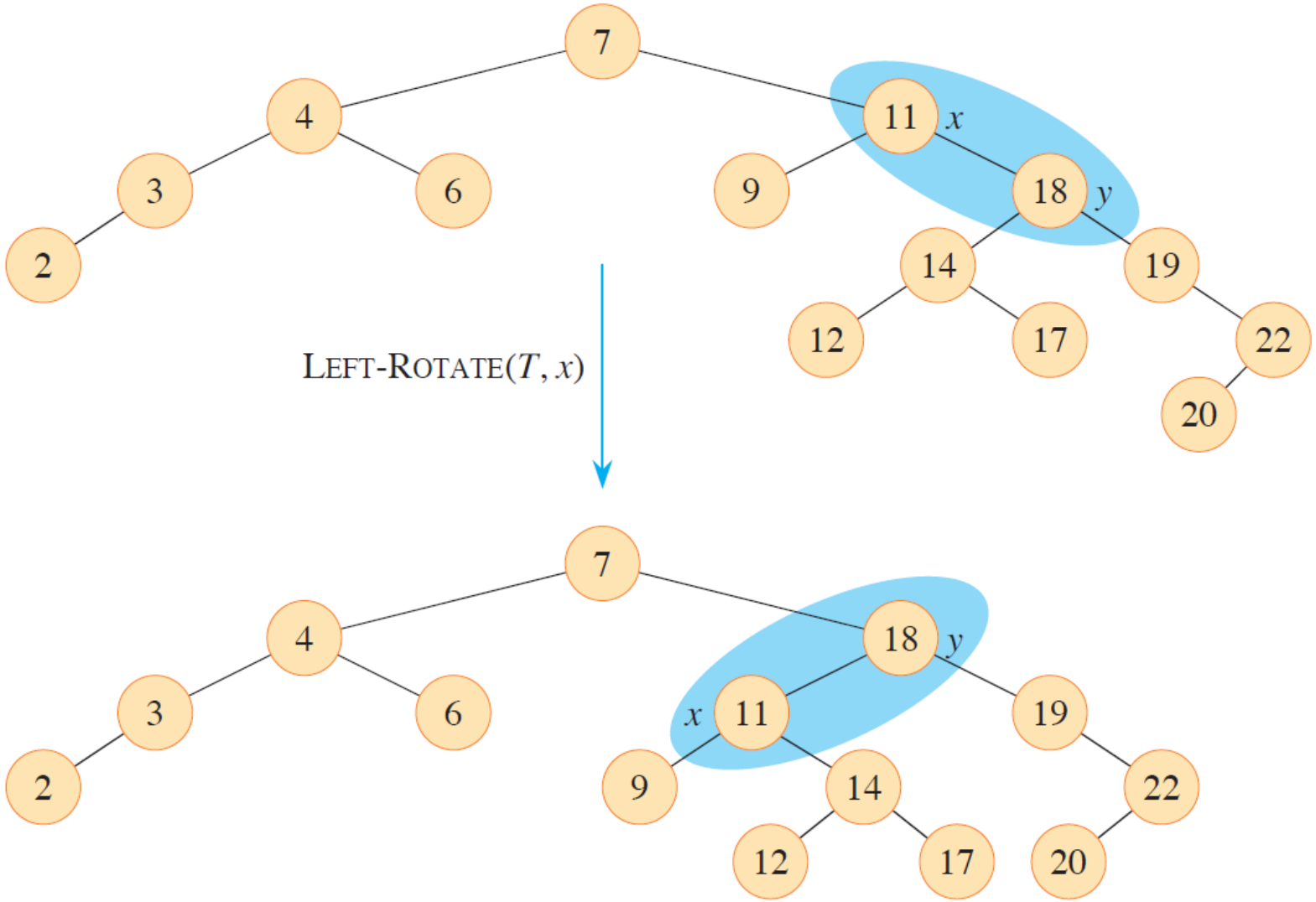
LEFT-ROTATE pseudocode explanation

Next slide shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time.

Only pointers are changed by a rotation, and all other attributes in a node remain the same.

An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree

Inorder tree walks of the input tree and the modified tree produce the same listing of key values.



Properties of red-black trees (2)

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL.

A sentinel is a dummy object that allows us to simplify boundary conditions.

For a red-black tree T, the sentinel $T.nil$ is an object with the same attributes as an ordinary node in the tree. Its colour attribute is **BLACK**, and its other attributes $-p, left, right, and\ key -$ can take on arbitrary values. As slide 9(b) shows, all pointers to NIL are replaced by pointers to the sentinel $T.nil$.

Why use the sentinel?

The sentinel makes it possible to treat a NIL child of a node x as an ordinary node whose parent is x.

An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however.

Instead, just the one sentinel $T.nil$ represents all the NILs - all leaves and the root's parent. The values of the attributes $p, left, right, and\ key$ of the sentinel are immaterial. The red-black tree procedures can place whatever values in the sentinel that yield simpler code.

black-height

We call the number of **black** nodes on any simple path from, but not including, a node x down to a leaf the black-height of the node, denoted $bh(x)$.

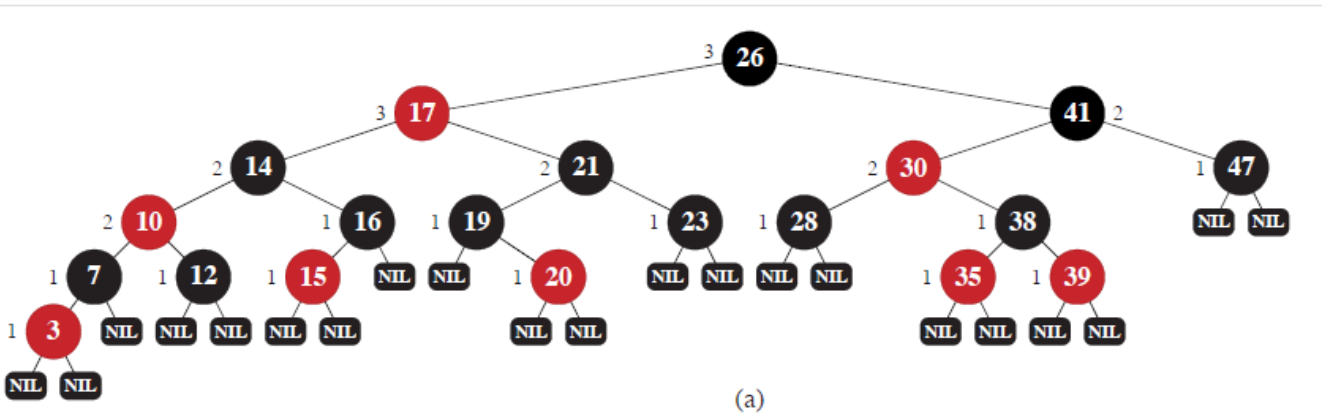
By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of **black** nodes. The **black**-height of a red-black tree is the **black**-height of its root.

A Red-Black tree Example

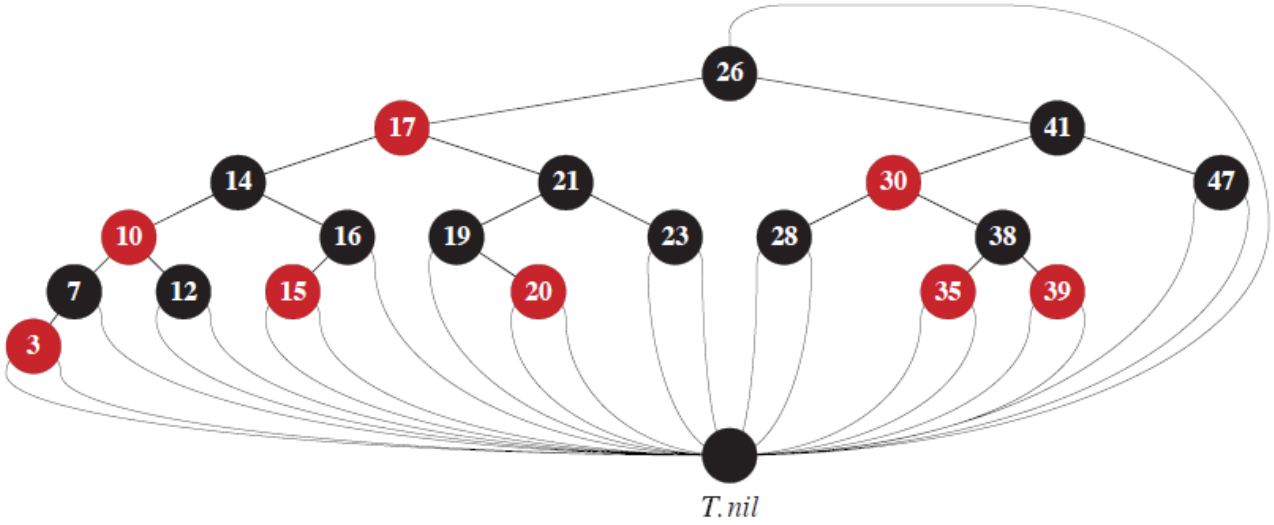
Every node in a red-black tree is either **red** or **black.**

The children of a **red** node are both **black.**

Every simple path from a node to a descendant leaf contains the same number of **black** nodes.

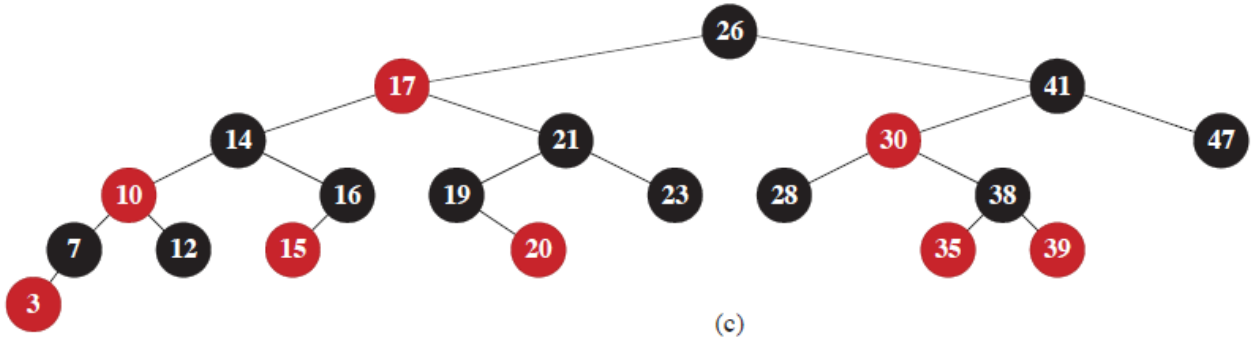


(a)



T.nil

(b)



(c)

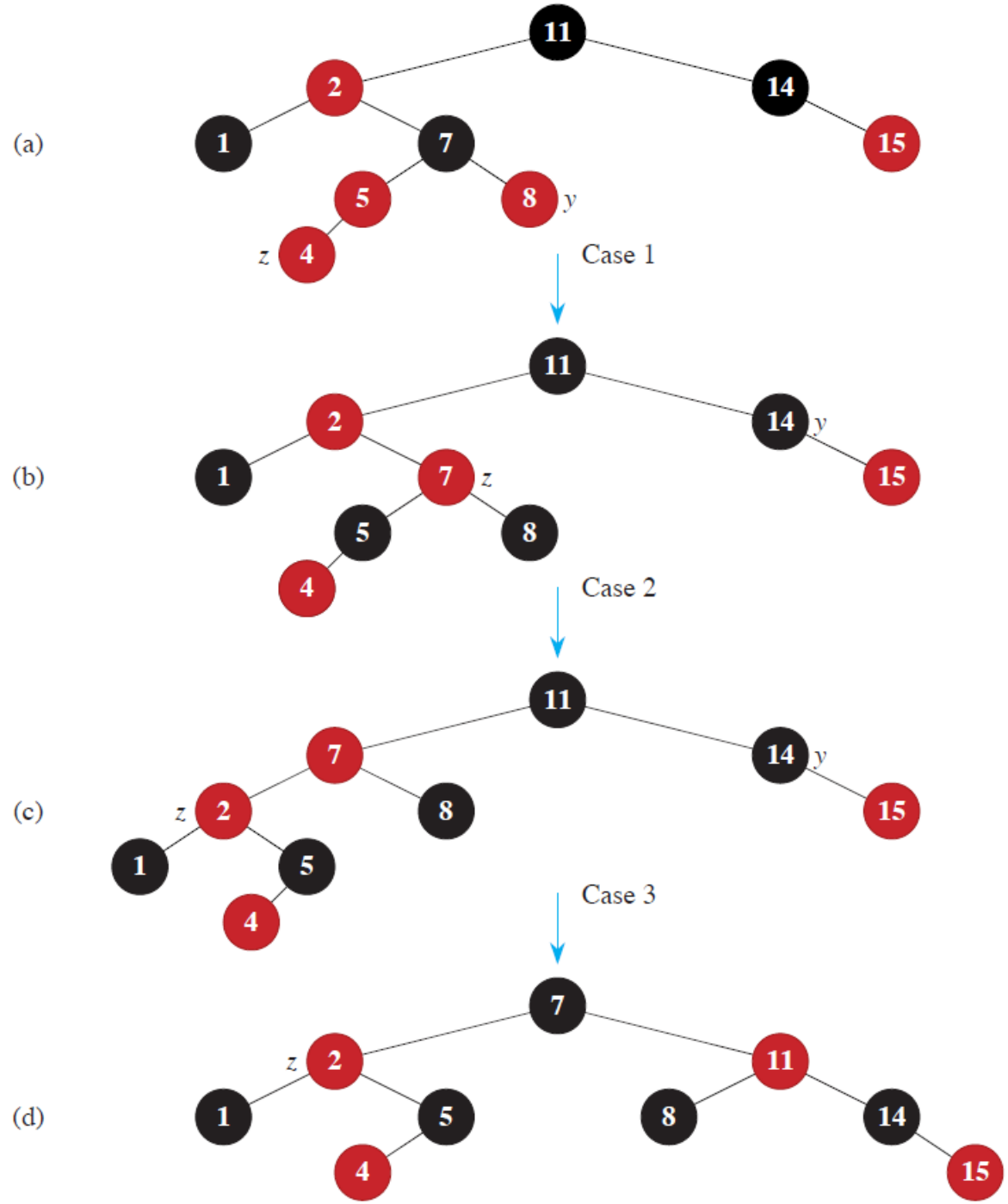
The example's explanation

(a) Every leaf, shown as a NIL, is **black**. Each non-NIL node is marked with its **black**-height, where NILs have **black**-height 0.

(b) The same red-black tree but with each NIL replaced by the single sentinel $T.nil$, which is always **black**, and with **black**-heights omitted. The root's parent is also the sentinel.

(c) The same red-black tree but with leaves and the root's parent omitted entirely.

Another Example of rotations



The example explanation

(a) A node z after insertion. Because both z and its parent $z.p$ are **red**, a violation of property 4 occurs. Since z's uncle y is **red**, case 1 in the code applies. Node z's grandparent $z.p.p$ must be **black**, and its blackness transfers down one level to z's parent and uncle.

Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results.

Once again, z and its parent are both **red**, but this time z's uncle y is **black**. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree in **(c)**.

Now z is the left child of its parent, and case 3 applies. Recolouring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

AVL and Red Black tree Visualisers

https://ds2-iiith.vlabs.ac.in/exp/red-black-tree/red-black-tree-oprations/simulation/redblack.html
https://www.cs.usfca.edu/galles/visualization/RedBlack.html
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html