

Lecture 9

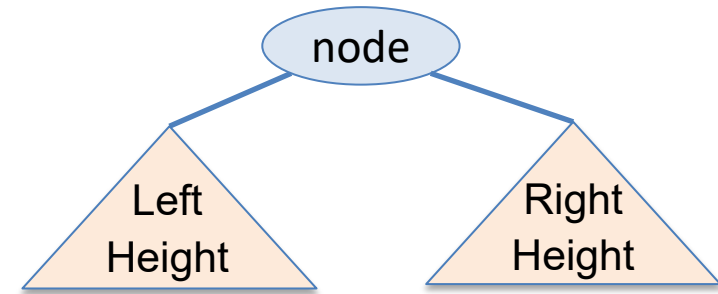
Self-Balancing Trees

Department of Computer Science
Hofstra University

AVL Tree

AVL Tree: A balanced BST that maintains the invariant: $|\text{LeftHeight} - \text{RightHeight}| \leq 1$ for all nodes in the tree.

- Named after Adelson-Velsky and Landis
- But also A Very Lovable Tree!

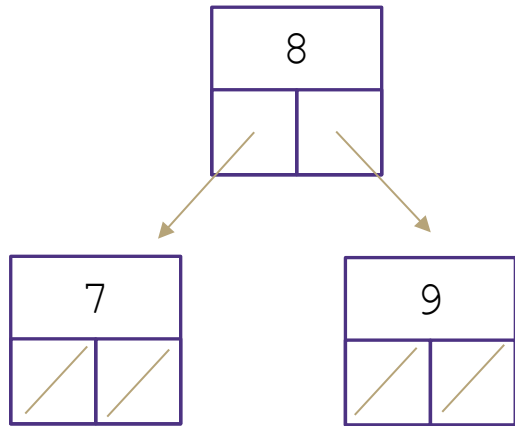


An AVL Tree has height $\approx \log(n)$

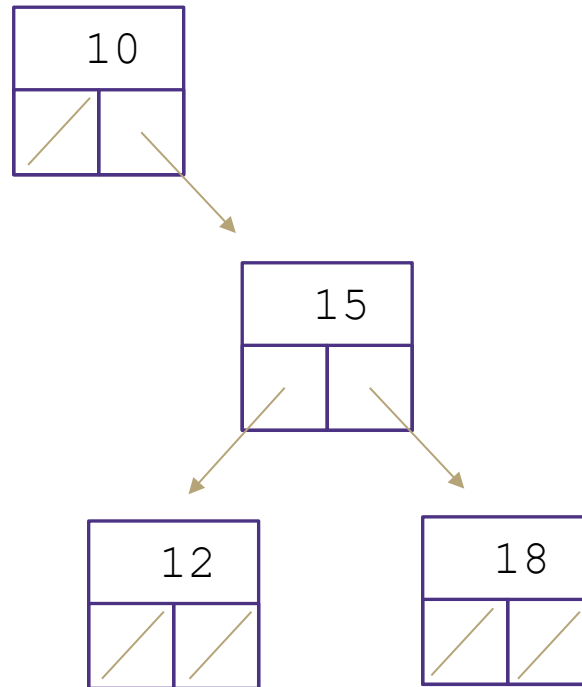
Measuring Balance

Measuring balance:

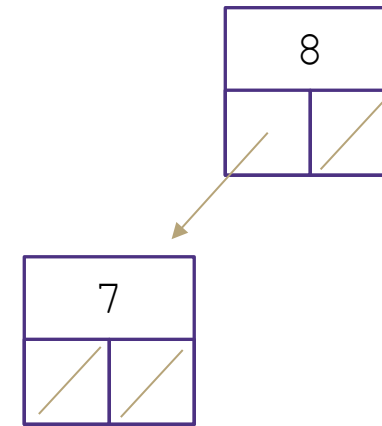
- For each node, compare the heights of its two subtrees
- Balanced when the difference in heights between subtrees is no greater than 1



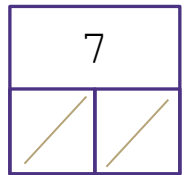
Balanced



Unbalanced



Balanced

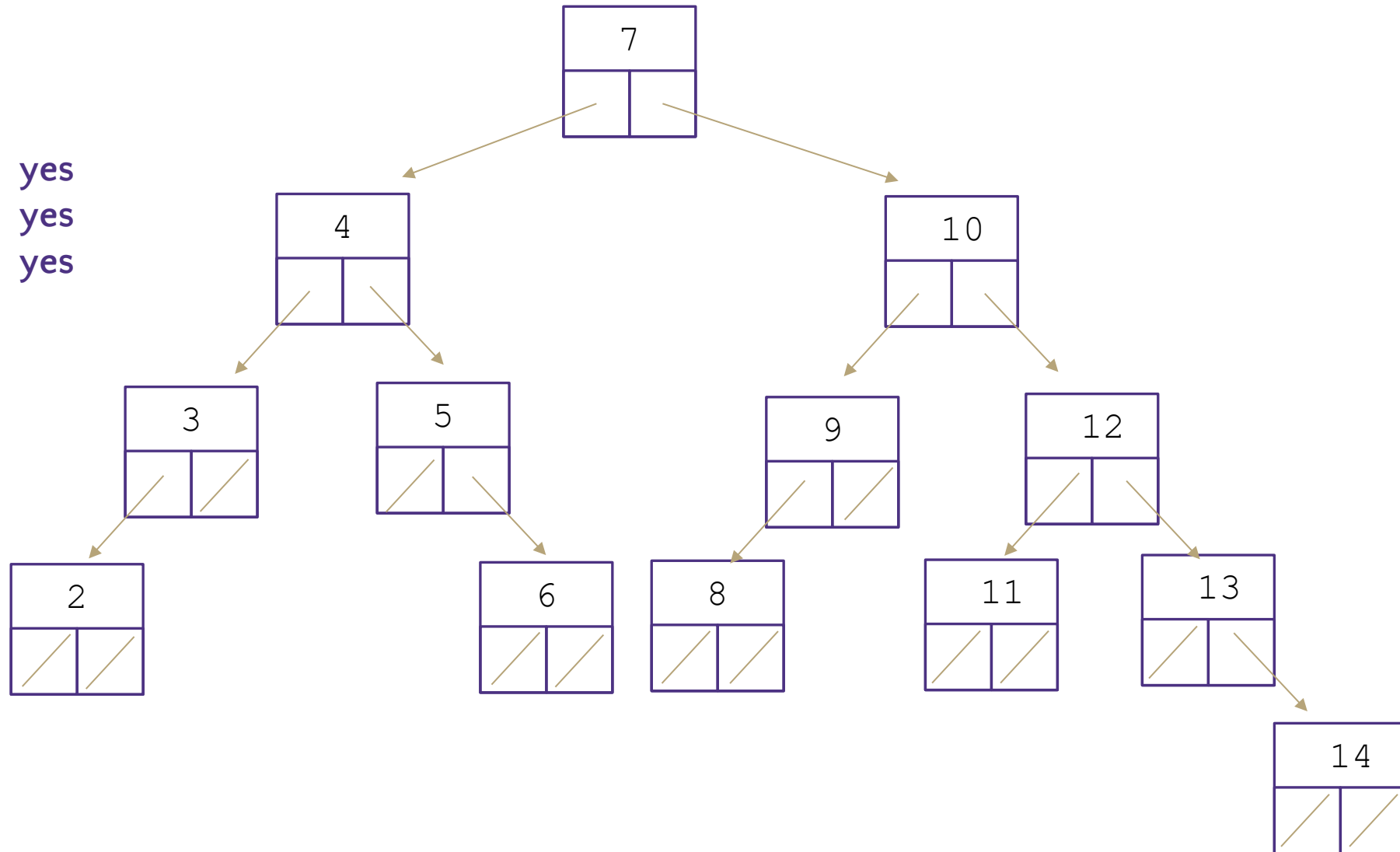


Balanced

Is this a valid AVL tree?

Is it...

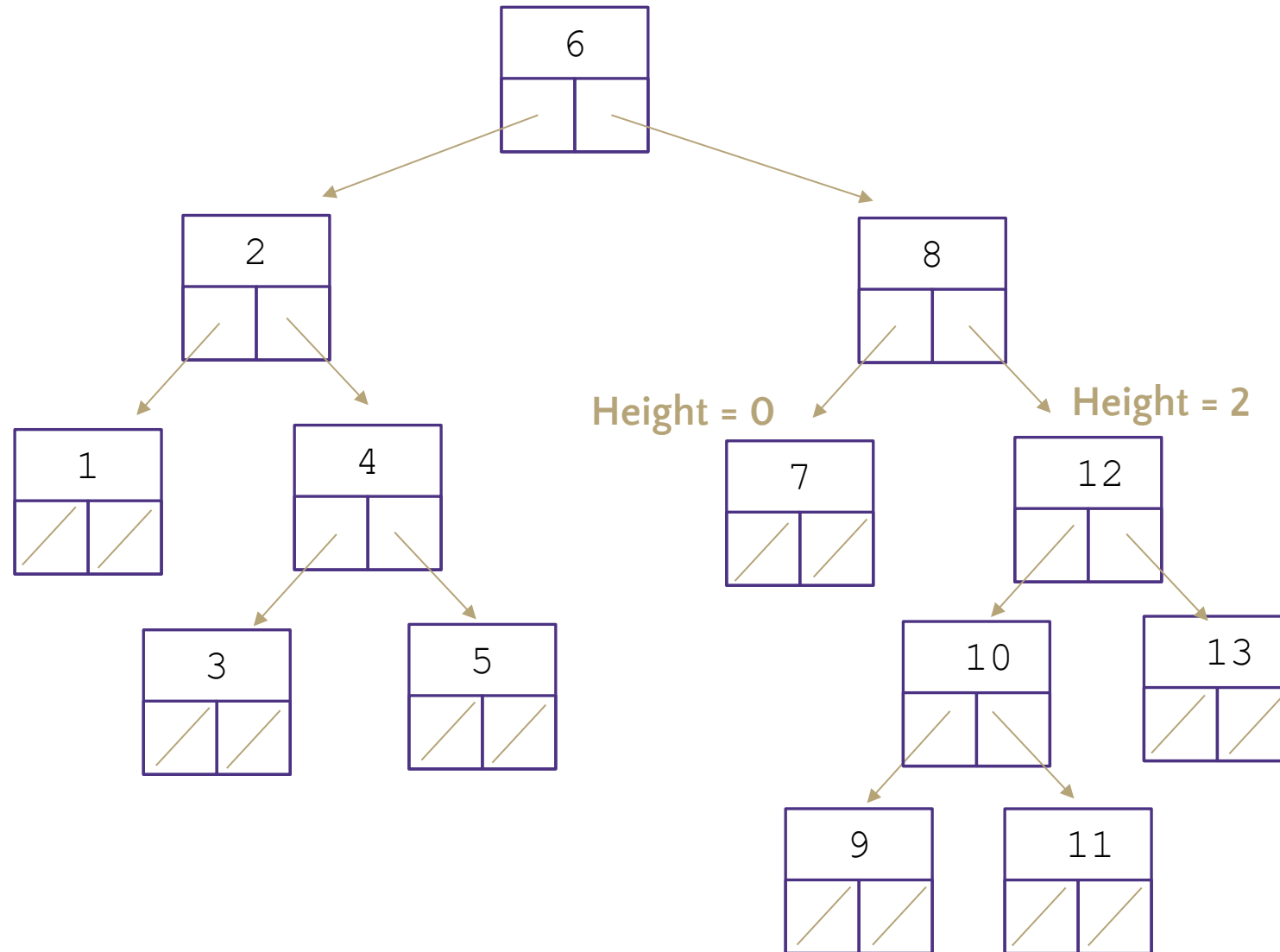
- Binary **yes**
- BST **yes**
- Balanced? **yes**



Is this a valid AVL tree?

Is it...

- Binary **yes**
- BST **yes**
- Balanced? **no**



Time Complexity



INVARIANT

```
public boolean containsKey(node, key) {  
    // find key  
}
```



INVARIANT

containsKey benefits from invariant:
worst-case $O(\log n)$ time



INVARIANT

```
public boolean insert(node, key) {  
    // find where key would go  
    // insert  
}
```



INVARIANT

Insert benefits from invariant:
worst-case $O(\log n)$ time to find location for
key

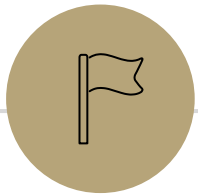
How to maintain the invariant?

- Track heights of subtrees
- Detect any imbalance
- Restore balance

BST containsKey()

The AVL Invariant

Rotations

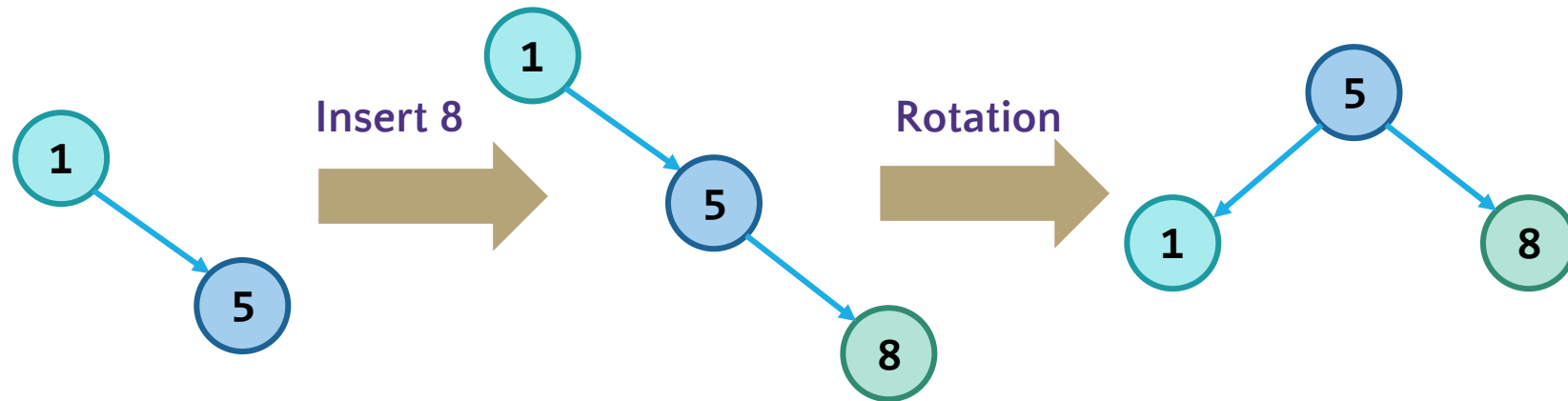


Insertion

What happens if an insertion breaks the AVL invariant?

The AVL rebalances itself by rotations. AVL is a type of “Self-Balancing Tree”

- A rotation alters the structure of a tree by rearranging subtrees.
- Goal is to decrease the height of the tree to maximum height of $O(\log n)$.
- Larger subtrees up, smaller subtrees down
- Does not affect the order of elements
- Time complexity $O(1)$

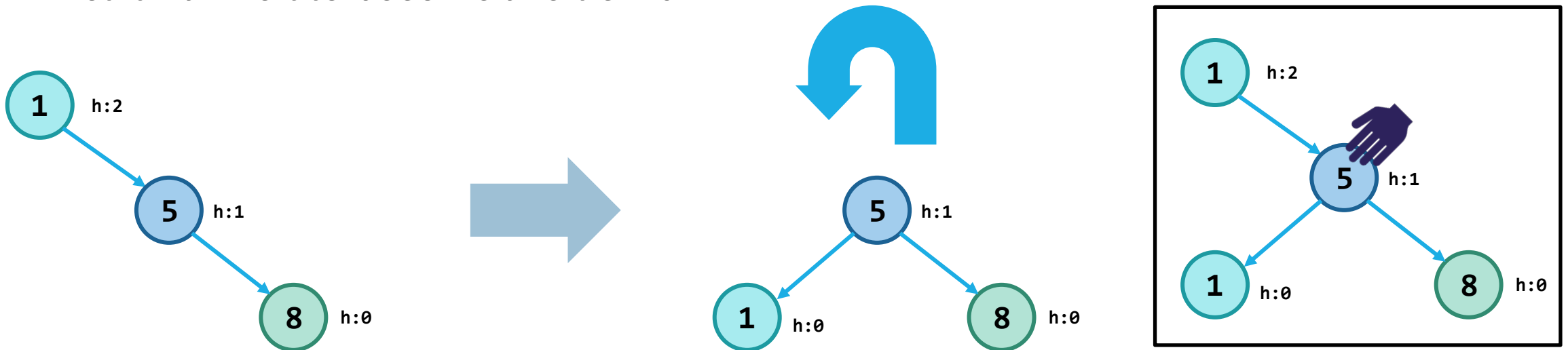


Fixing AVL Invariant: Left Rotation

We can fix the AVL invariant by performing rotations wherever an imbalance was created

Left Rotation

- Find the node that is violating the invariant (here, 1)
- Let it “fall” left to become a left child

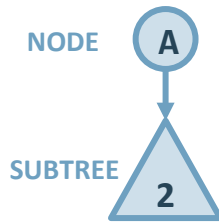
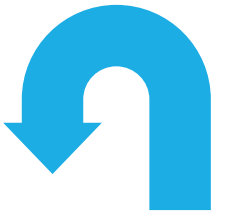
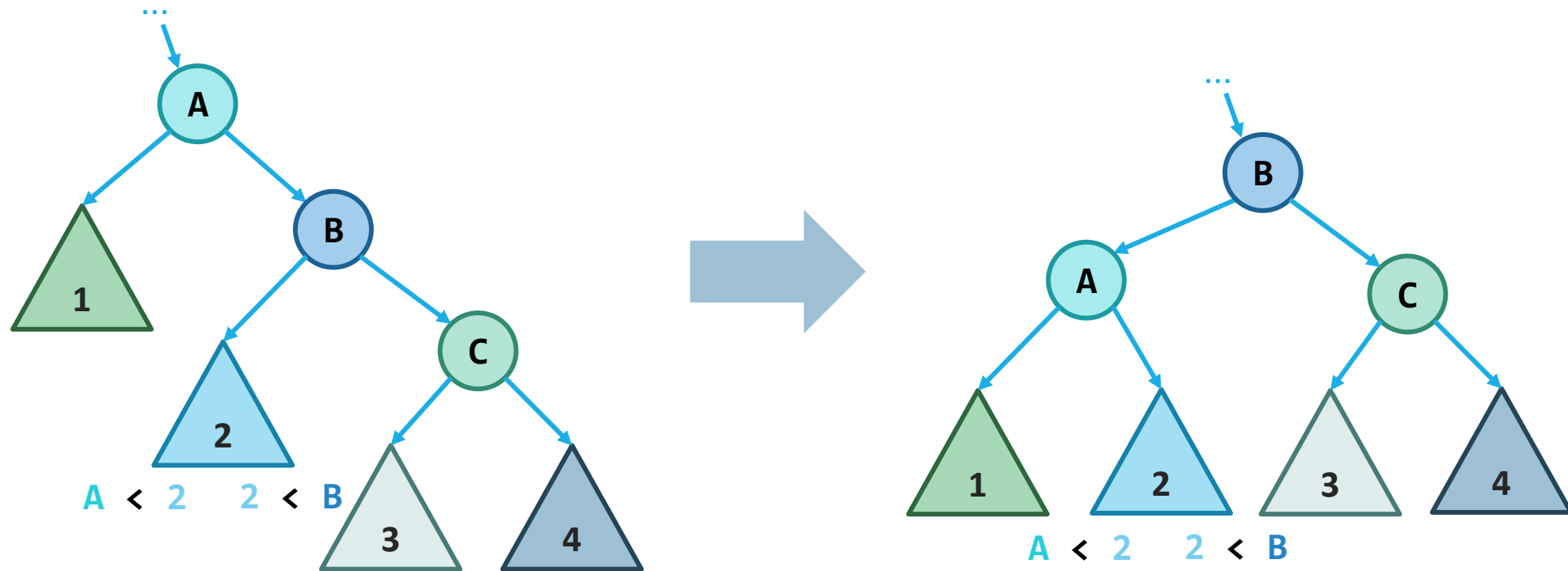


Apply a left rotation whenever the newly inserted node is located under the **right child of the right child**

Left Rotation: More Precisely

Subtrees are okay! They just come along for the ride.

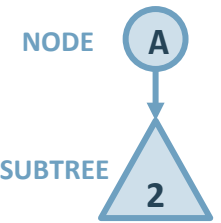
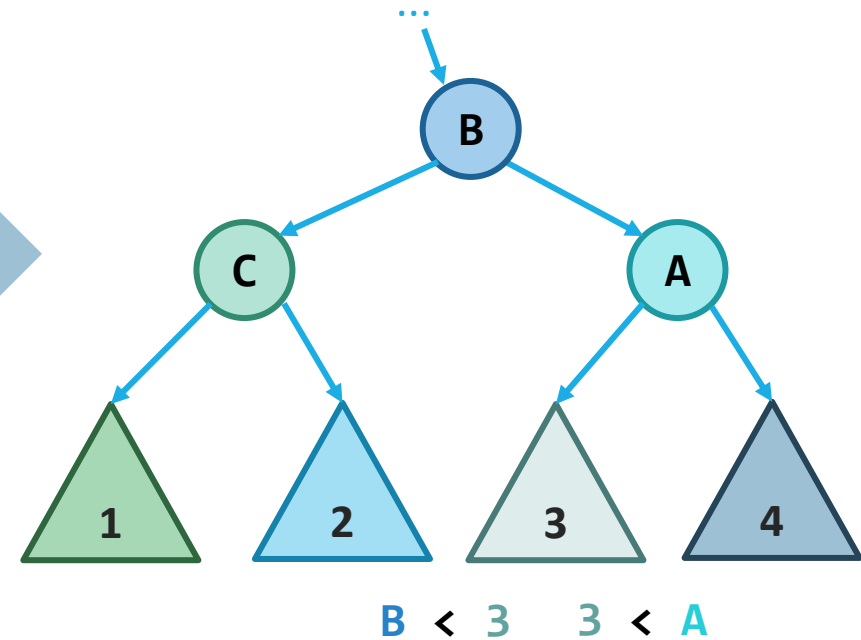
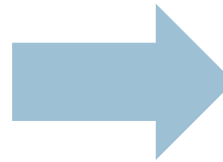
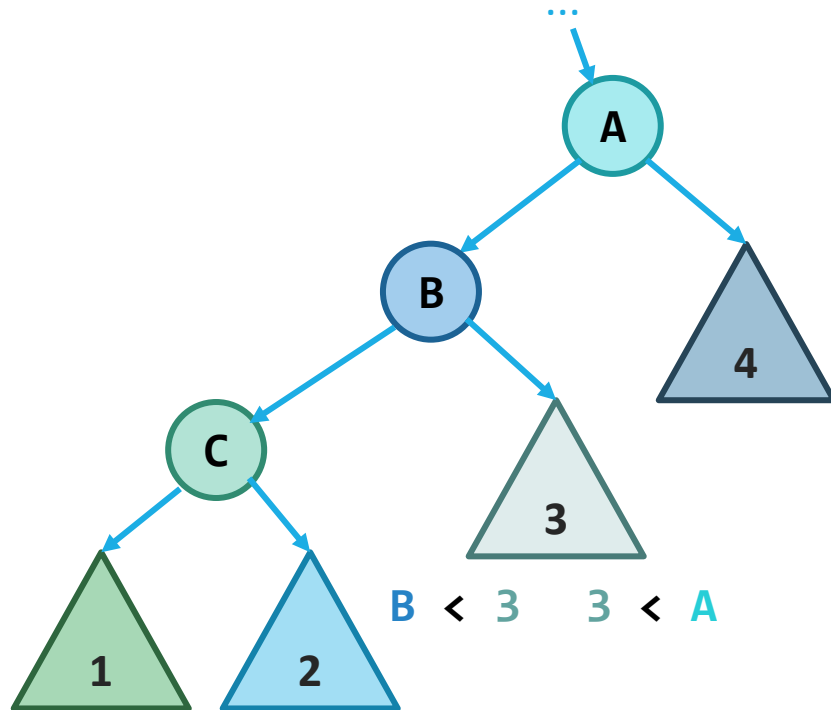
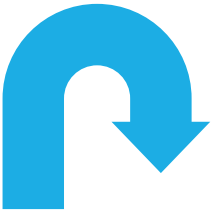
- Subtree 2 changes from left child of B to right child of A – but notice that its relationship with nodes A and B doesn't change in the new position!



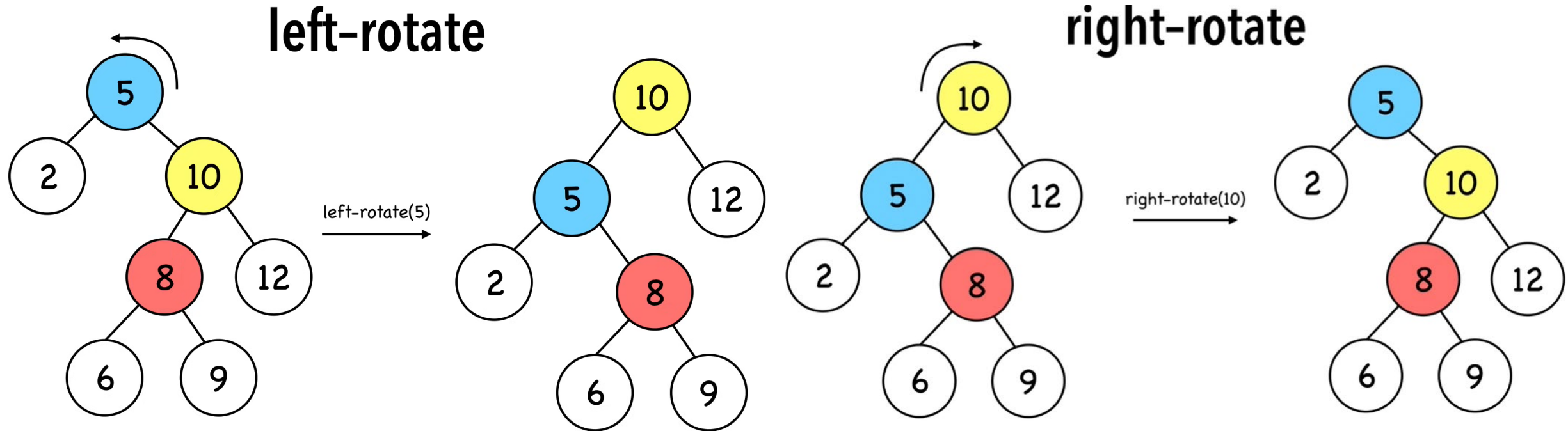
Right Rotation

Right Rotation

- Mirror image of Left Rotation!



Left or Right Rotation Examples

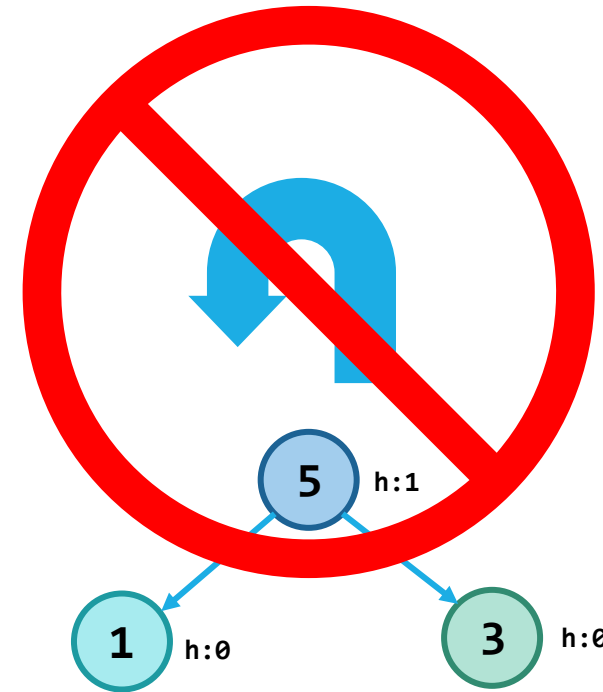
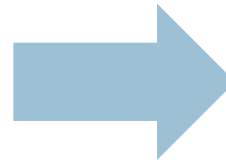
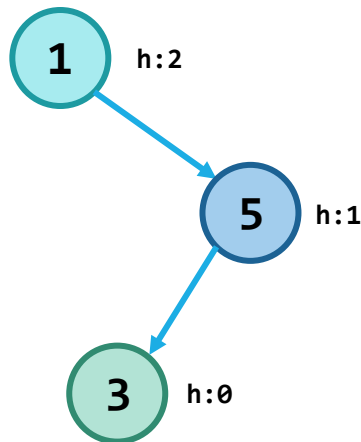


Not Quite as Straightforward

What if there's a "kink" in the tree where the insertion happened?

Can we apply a Left Rotation?

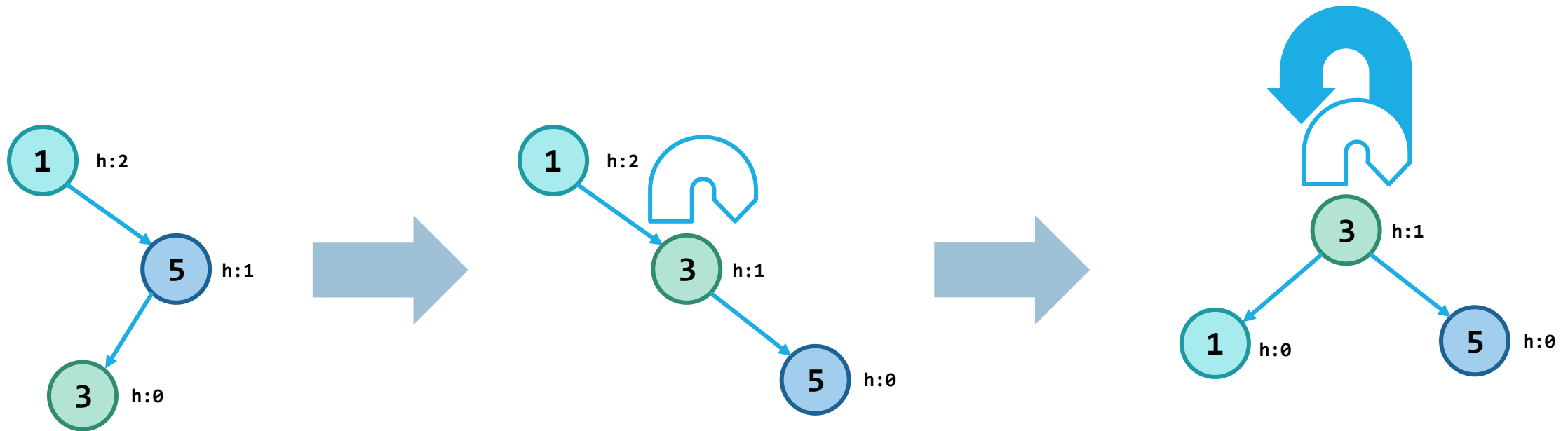
- No, violates the BST invariant!



Right/Left Rotation

Solution: **Right/Left Rotation**

- Two steps: First do a right rotation for the right two nodes. then do a left rotation for the three nodes.
- Preserves BST invariant!

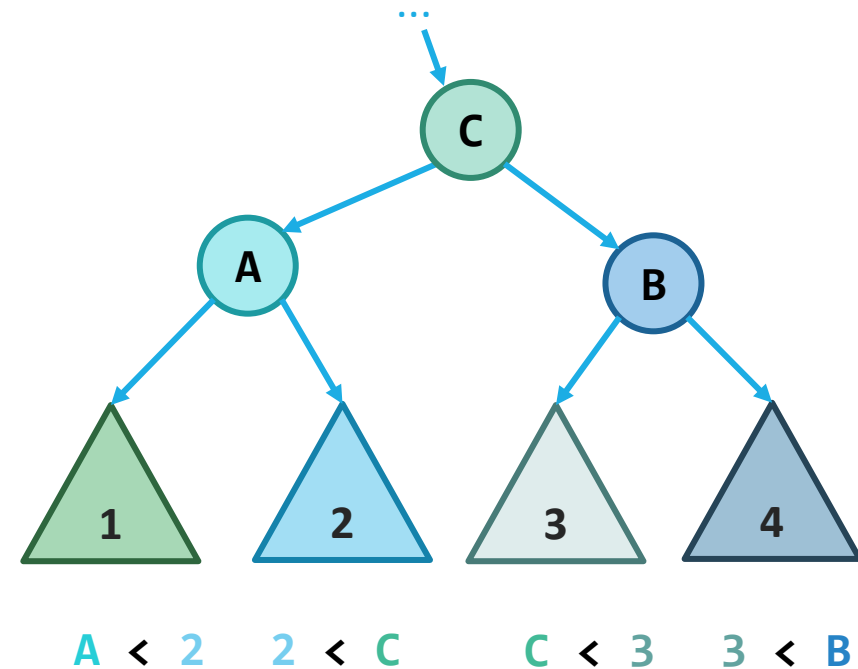
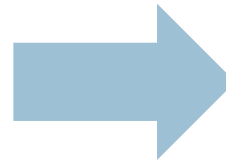
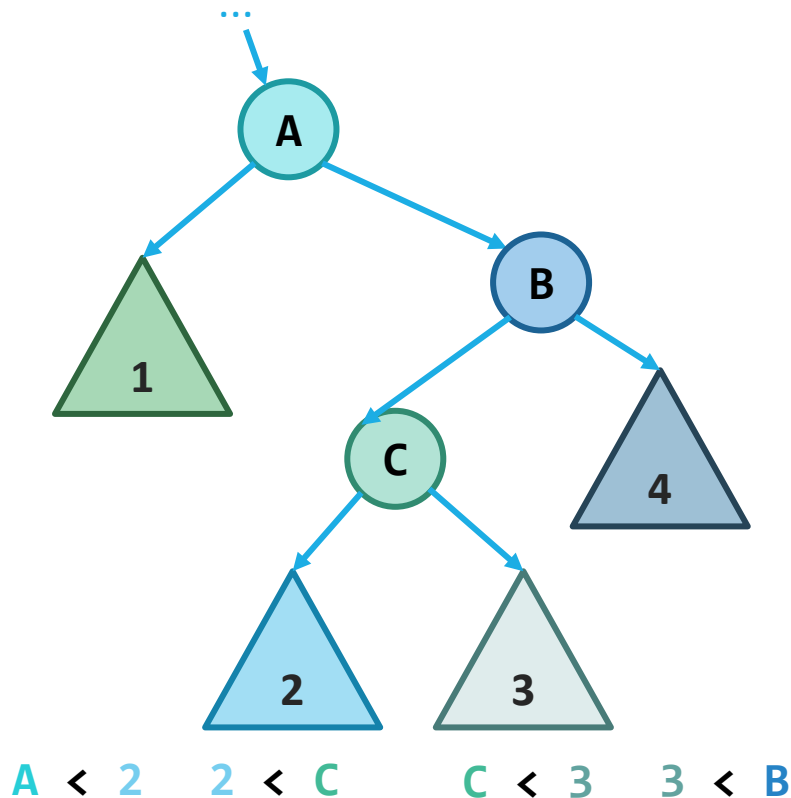
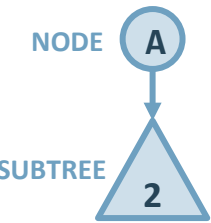


Right/Left Rotation: More Precisely



Again, subtrees are invited to come with

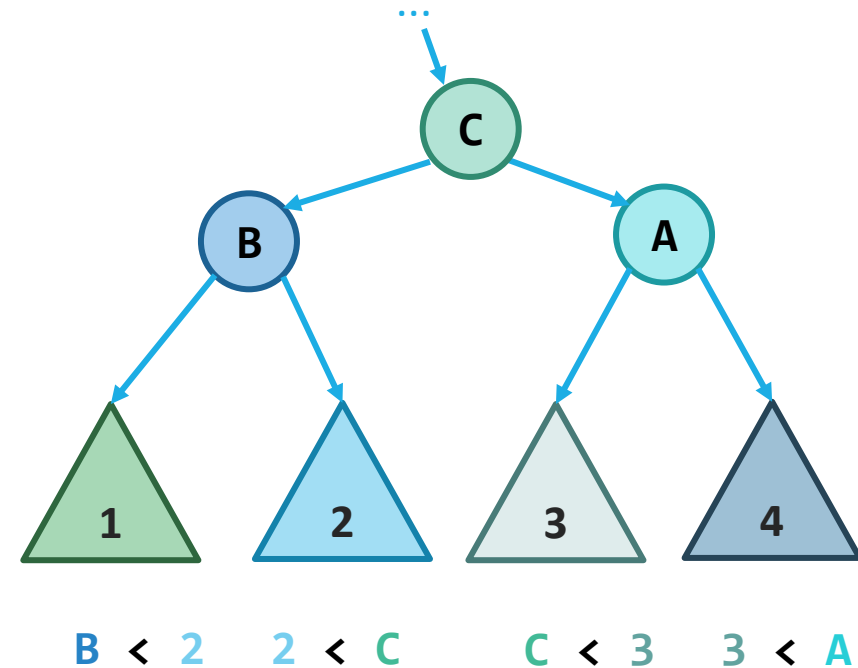
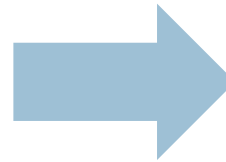
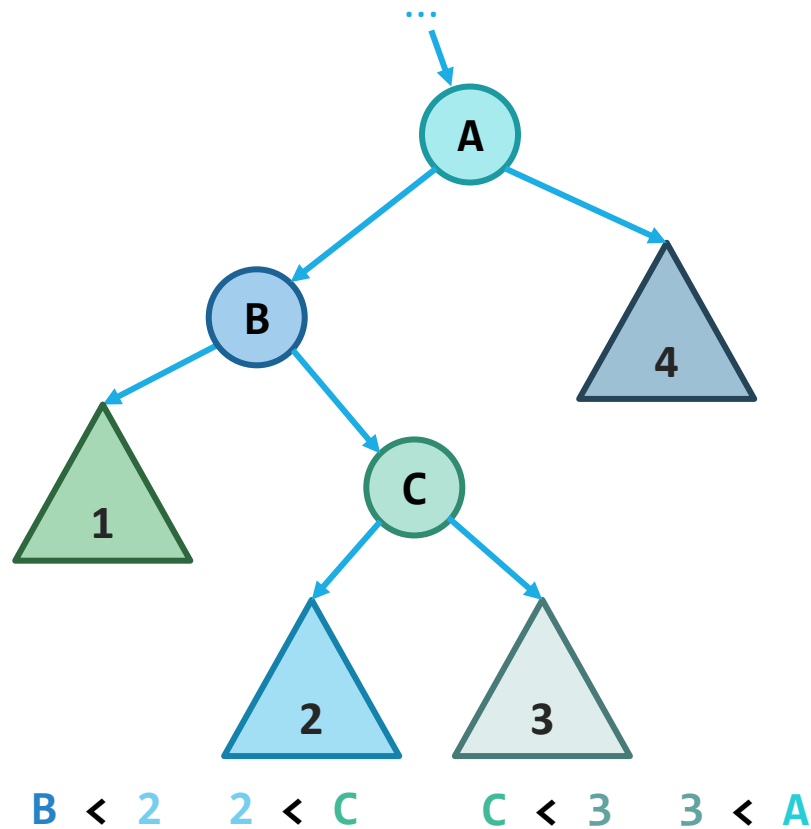
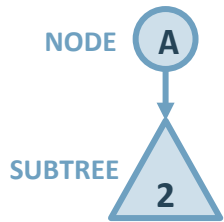
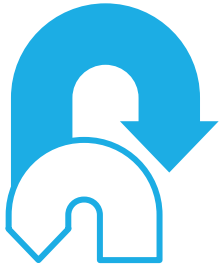
- Now 2 and 3 both have to hop, but all BST ordering properties are still preserved
- (Note that A, B and C denote some numerical value, not letters 'A', 'B', 'C')



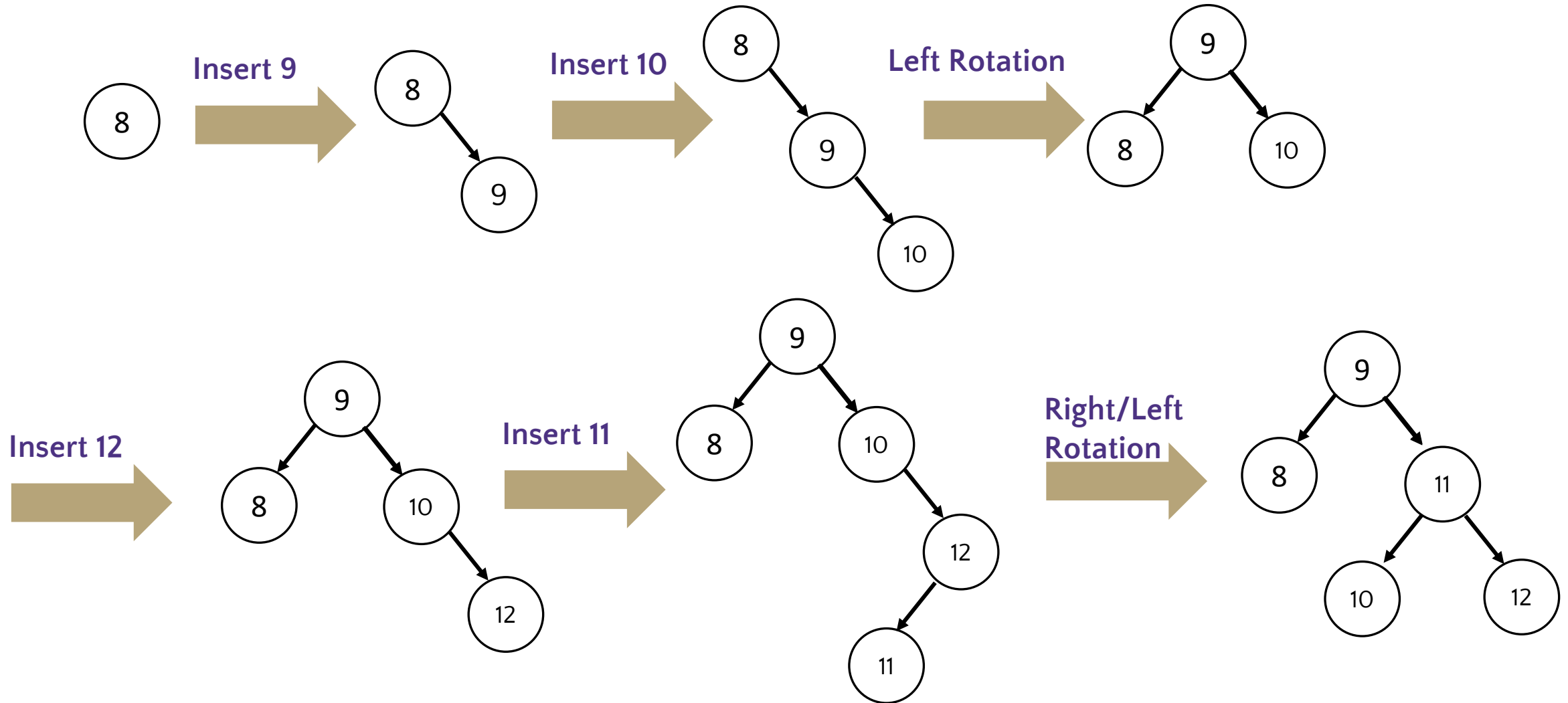
Left/Right Rotation

Left/Right Rotation

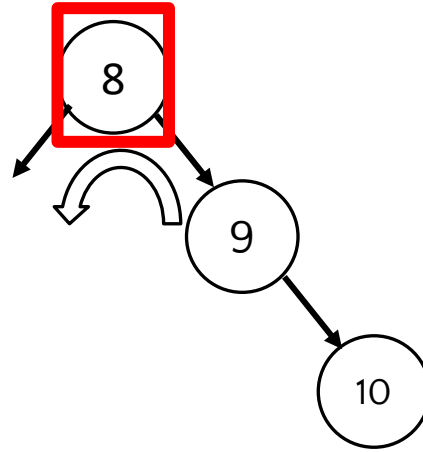
- Mirror image of Right/Left Rotation!



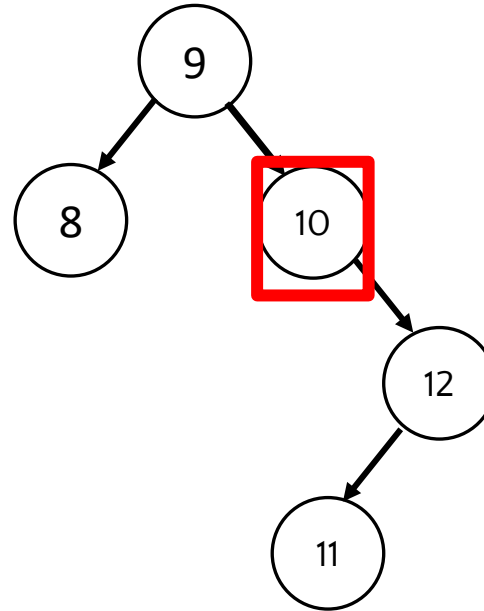
AVL Example: 8,9,10,12,11



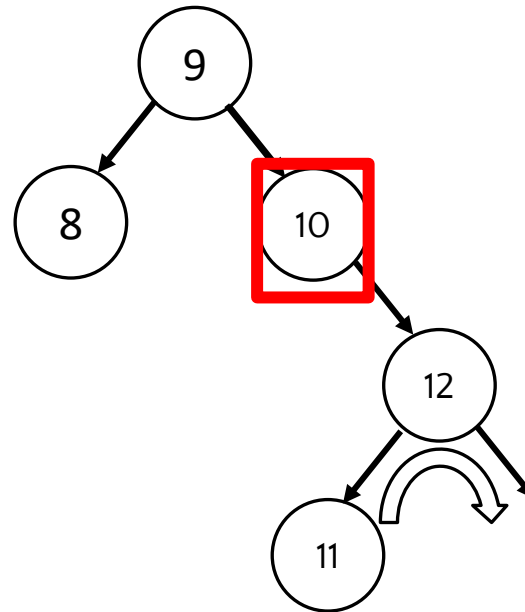
AVL Example: 8,9,10,12,11



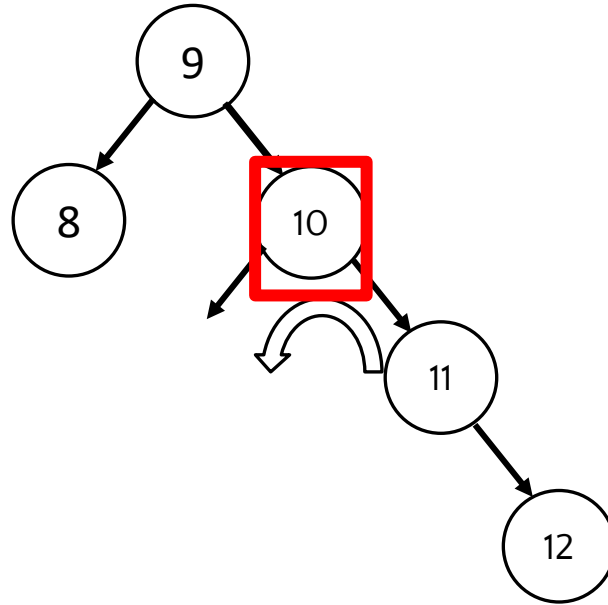
AVL Example: 8,9,10,12,11



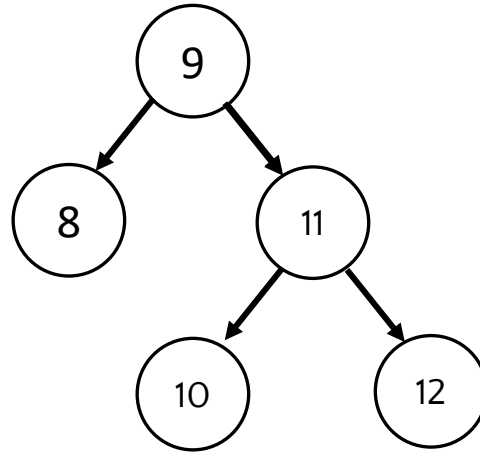
AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



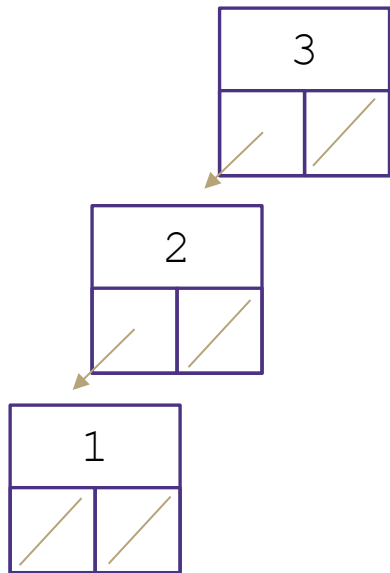
AVL Example: 8,9,10,12,11



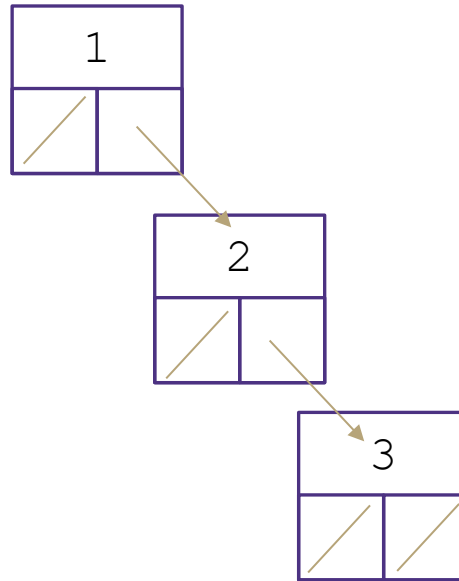
Two AVL Cases

Line Case

Solve with **1** rotation



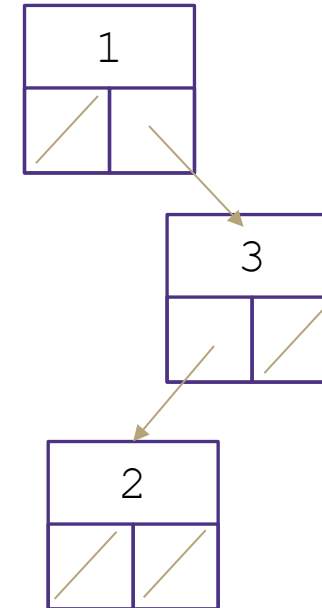
Rotate Right



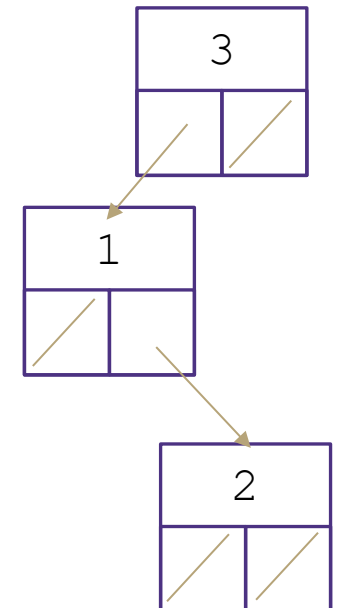
Rotate Left

Kink Case

Solve with **2** rotations



Right Kink Resolution
Right/Left Rotation



Left Kink Resolution
Left/Right Rotation

How Long Does Rebalancing Take?

- Assume we store in each node the height of its subtree.
 - How do we find an unbalanced node?
 - Go back up the tree from where we inserted.
- How many rotations might we have to do?
 - Just a single or double rotation on the lowest unbalanced node.
 - A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion
 - $O(\log n)$ time to traverse to a leaf of the tree
 - $O(\log n)$ time to find the imbalanced node
 - $O(1)$ constant time to do the rotation(s)
 - Overall complexity: $O(\log n)$ time for adding a node.

AVL insertion: Approach

Overall algorithm:

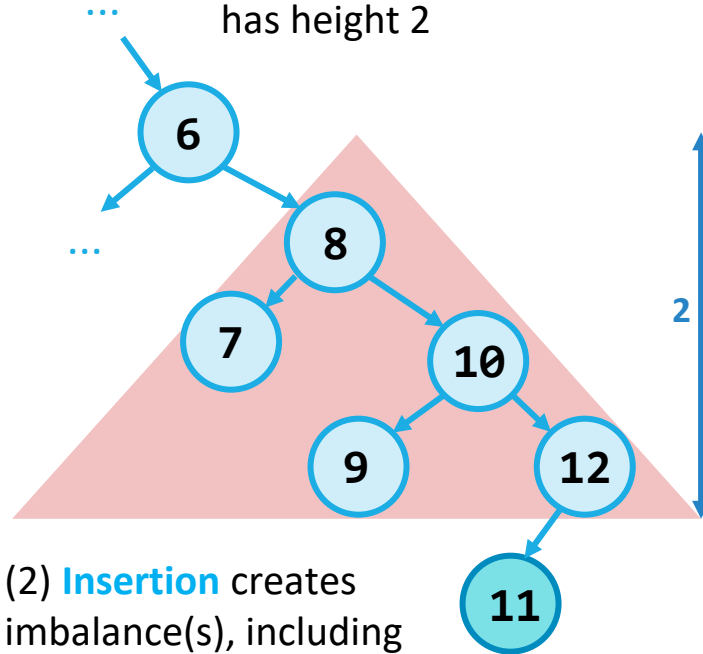
1. Insert the new node as in a BST (a new leaf)
2. For each node *on the path from the root to the new leaf*:
 - The insertion may (or may not) have changed the node's height
 - Detect height imbalance and perform a *rotation* to restore balance

Facts that make this easier:

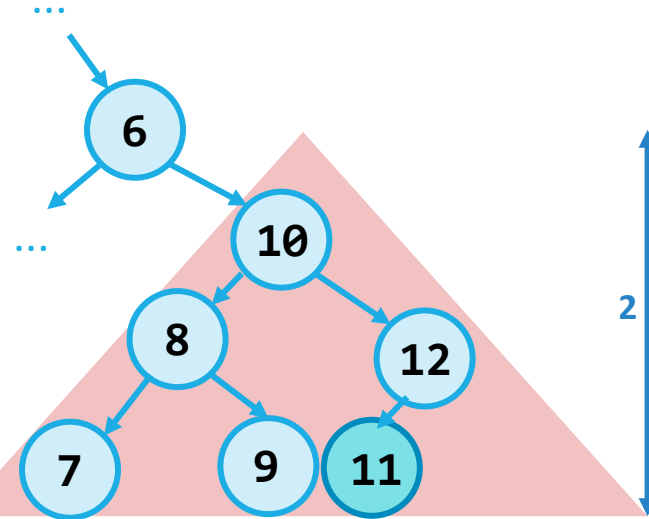
- Imbalances can only occur along the path from the new leaf to the root
- We only have to address the lowest unbalanced node
- Applying a rotation (or double rotation), restores the height of the subtree before the insertion -- when everything was balanced!
- Therefore, we need ***at most one rebalancing operation***

AVL insertion: Example

(1) Originally, whole tree balanced, and **this subtree** has height 2



(2) **Insertion** creates imbalance(s), including **the subtree** (8 is lowest unbalanced node)



(3) Left rotation on 8 will restore **the subtree** to height 2, whole tree balanced again!

AVL deletion

- Deletion involves a similar set of rotations that let you rebalance an AVL tree after deleting an element
 - Omitted since it is beyond scope of this course
- In the worst case, takes $O(\log n)$ time to rebalance after a deletion
 - Finding the node to delete is also $O(\log n)$, so total complexity is $O(\log n)$

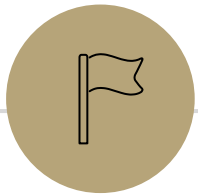
AVL Trees

PROS

- All operations on an AVL Tree (search, insertion, deletion) have worst-case complexity $O(\log n)$
- Because the tree is always balanced!

CONS

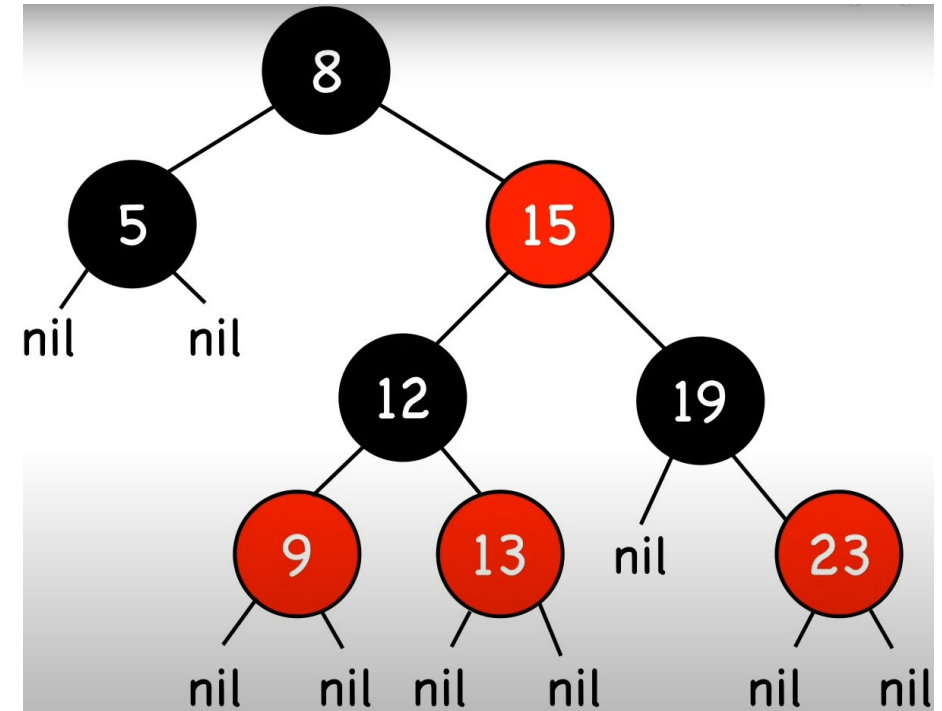
- Additional space for the height field
- Rebalancing does incur some overhead
 - May not be important depending on the application



Red Black Trees

Red-Black Tree

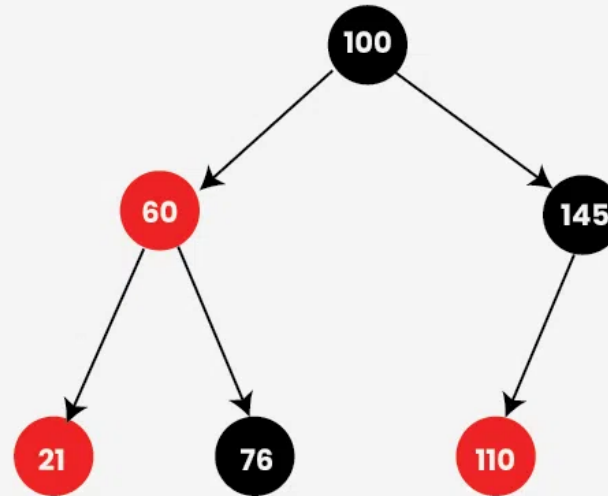
- **Red-Black Tree:** A balanced BST that maintains the invariant: the longest path (root to farthest NIL) is no more than twice the length of the shortest path (root to nearest NIL).
 - Right figure: Shortest path: all black nodes (=2); Longest path: alternating red and black (=4)
- RBT uses color coding to maintain balance and reduce the number of node re-arrangements needed via rotations. It has four properties:
 - Node Color: A node is either red or black.
 - Root Property: The root and leaves (NIL) are black.
 - Red Property: If a node is red, then its children are black.
 - Black Property: All paths from a node to its NIL descendants contain the same number of black nodes.
- Node insertion and deletion may result in violation of these properties.
 - Use recoloring and rotations to maintain these properties.



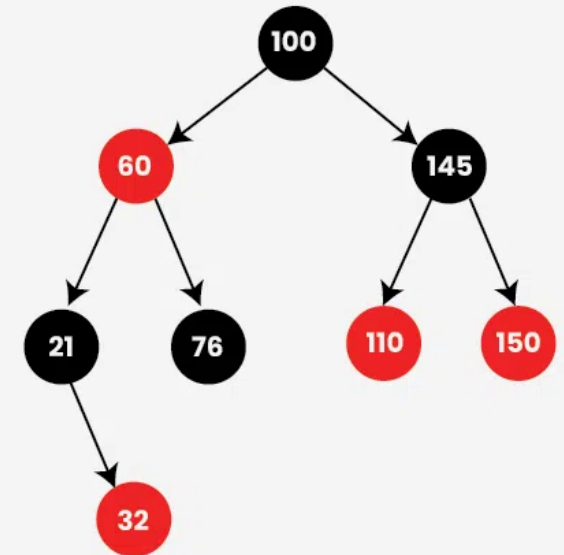
Examples

- Tree on the left: Incorrect Red Black Tree.
 - Two red nodes are adjacent to each other.
 - One of the paths to a leaf node has zero black nodes, whereas the other two paths contain 1 black node each.

Example of Red-black Tree



A incorrect Red-black Tree

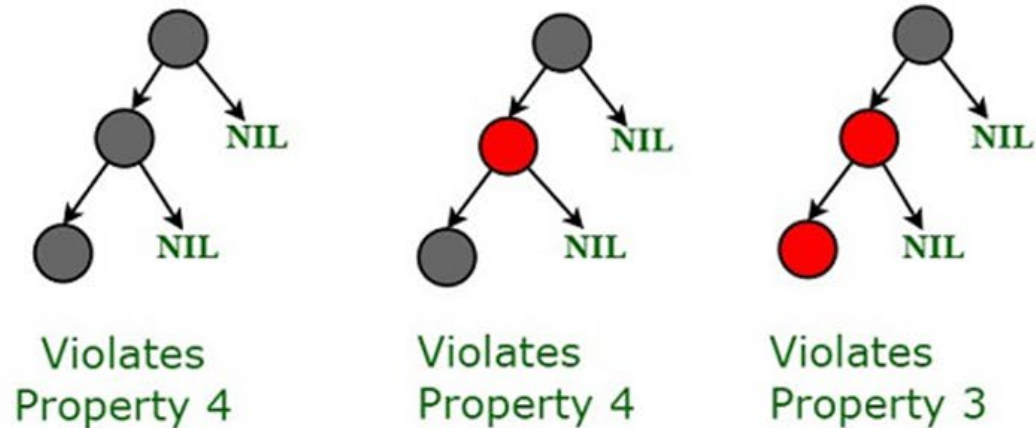


A correct Red-black Tree

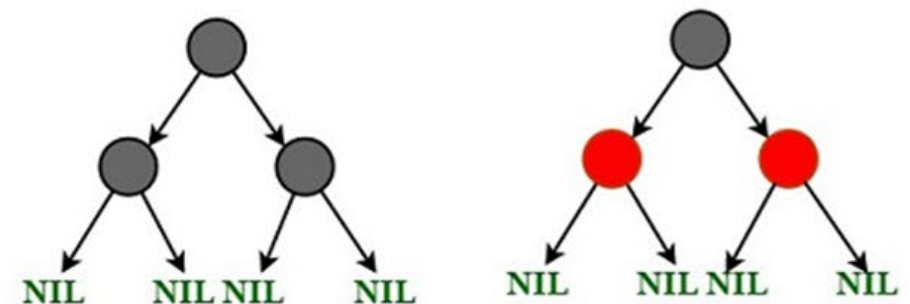
Red-Black tree ensures balancing

- A linear chain of 3 nodes is not possible in a Red-Black tree

Following are NOT possible 3-noded Red-Black Trees

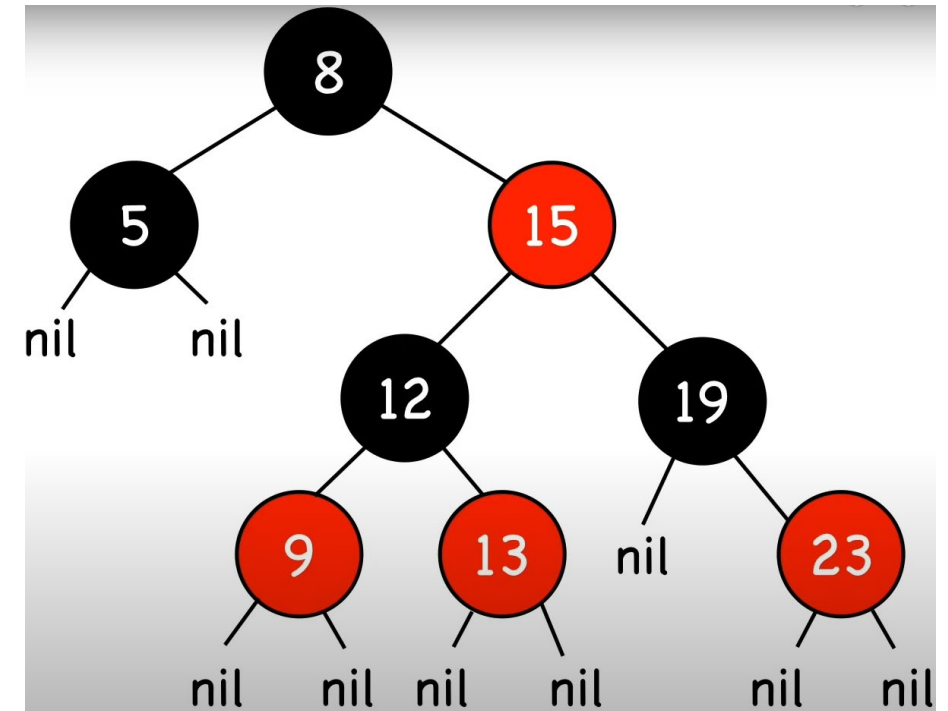


Following are possible Red-Black Trees with 3 nodes



Additional Properties

- **AVL Tree:** A balanced BST that maintains the invariant $|\text{LeftHeight} - \text{RightHeight}| \leq 1$ for all nodes in the tree.

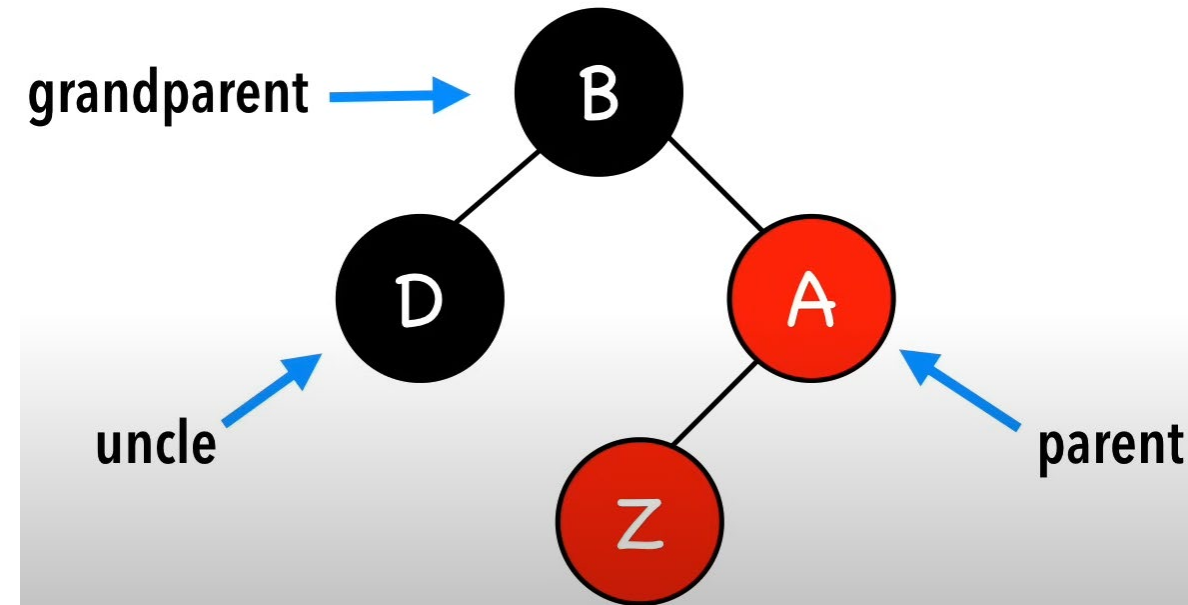


Insertion

- Inserting a new node in a Red-Black Tree involves a two-step process: performing a standard binary search tree (BST) insertion, followed by fixing any violations of Red-Black properties.
- **Insertion Steps**
 - 1. BST Insert:** Insert the new node into BST and color it red.
 - 2. Fix Violations:**
 2. If the parent of the new node is **black**, no properties are violated.
 3. If the parent is **red**, the tree might violate the Red Property, requiring fixes.

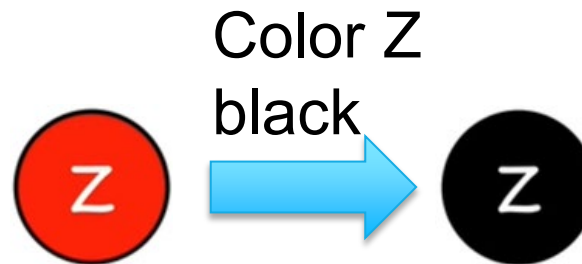
Insertion

- Step 1. Insert Z and color it **red**
- Step 2. Recolor and rotate nodes to fix violations
- 4 scenarios after inserting node Z
- Case 0. Z = root
 - Color Z black
- Case 1. Z.uncle = **red**
 - Recolor Z's parents and grandparent
- Case 2. Z.uncle = black (triangle)
 - Rotate Z.parent, turns into Case 3
- Case 3. Z.uncle = black (line)
 - Rotate Z.grandparent & Recolor Z's parents and grandparent



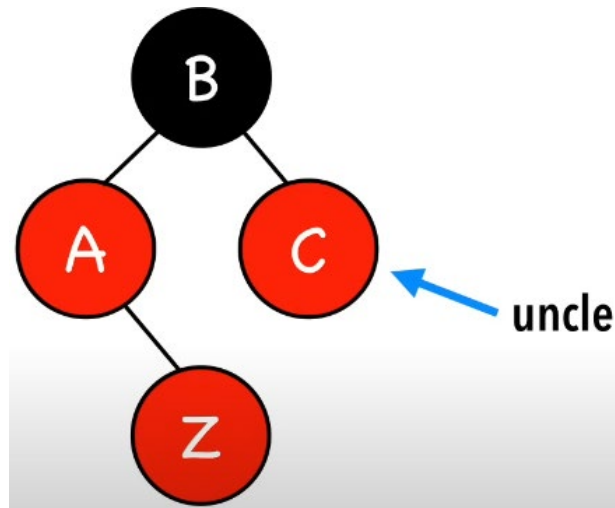
Case 0. $Z = \text{root}$

- Color Z black

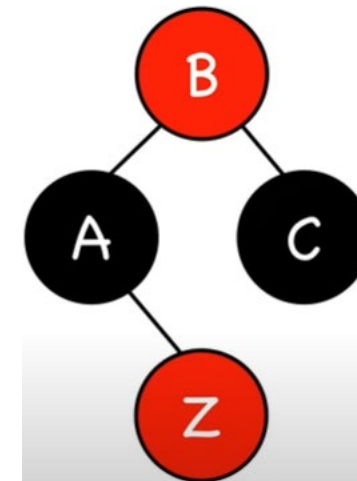


Case 1. Z.uncle = red

- Recolor Z's parent, uncle, and grandparent

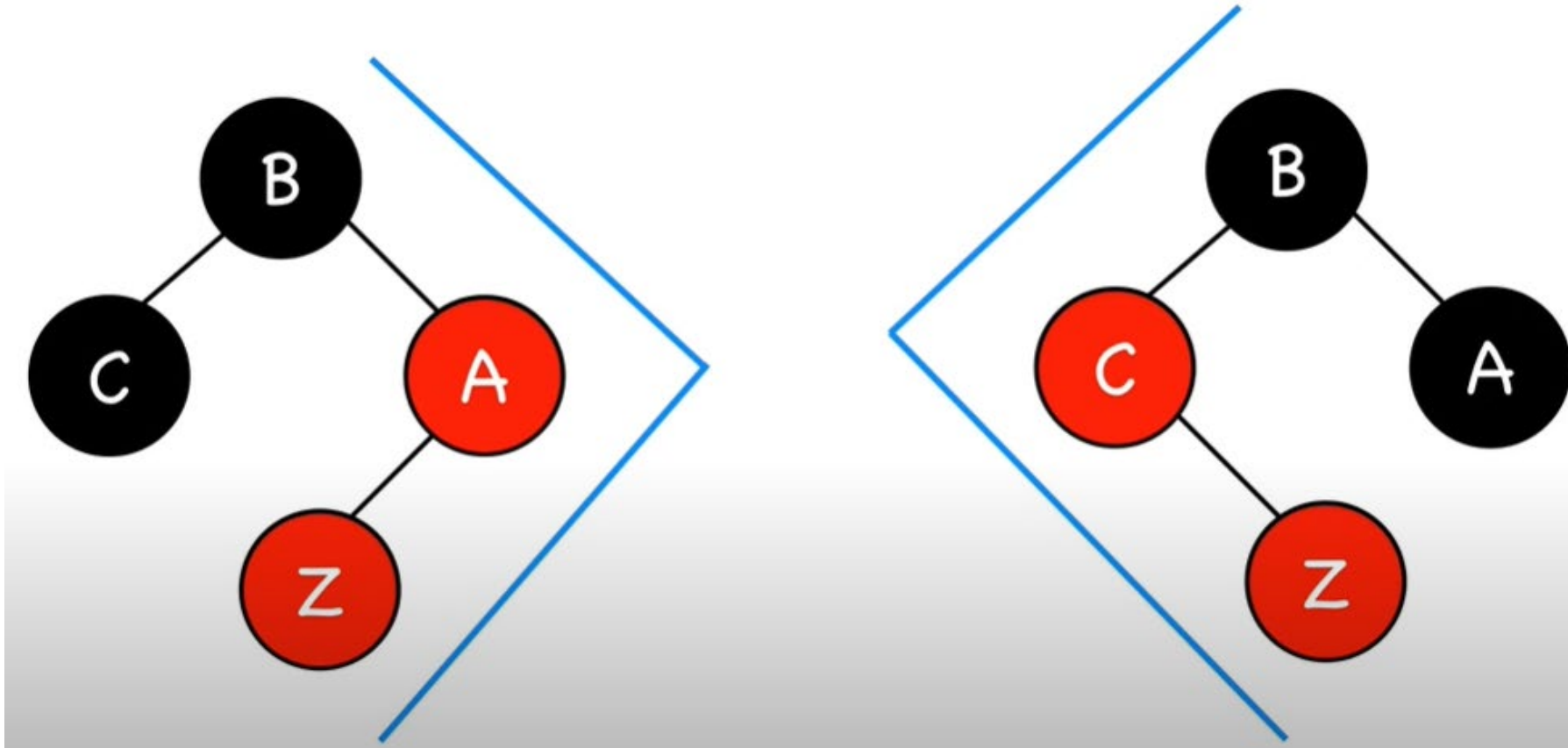


recolor



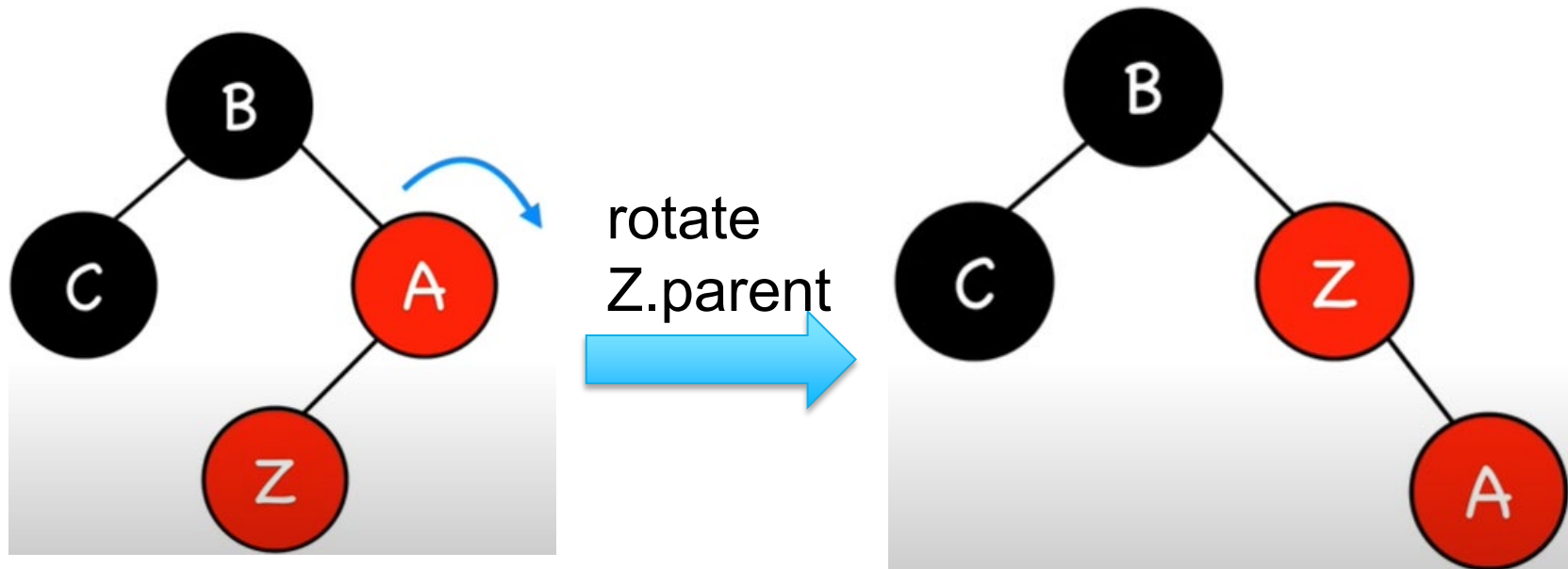
Case 2. Z.uncle = black (triangle)

case 2 : Z.uncle = black (triangle)



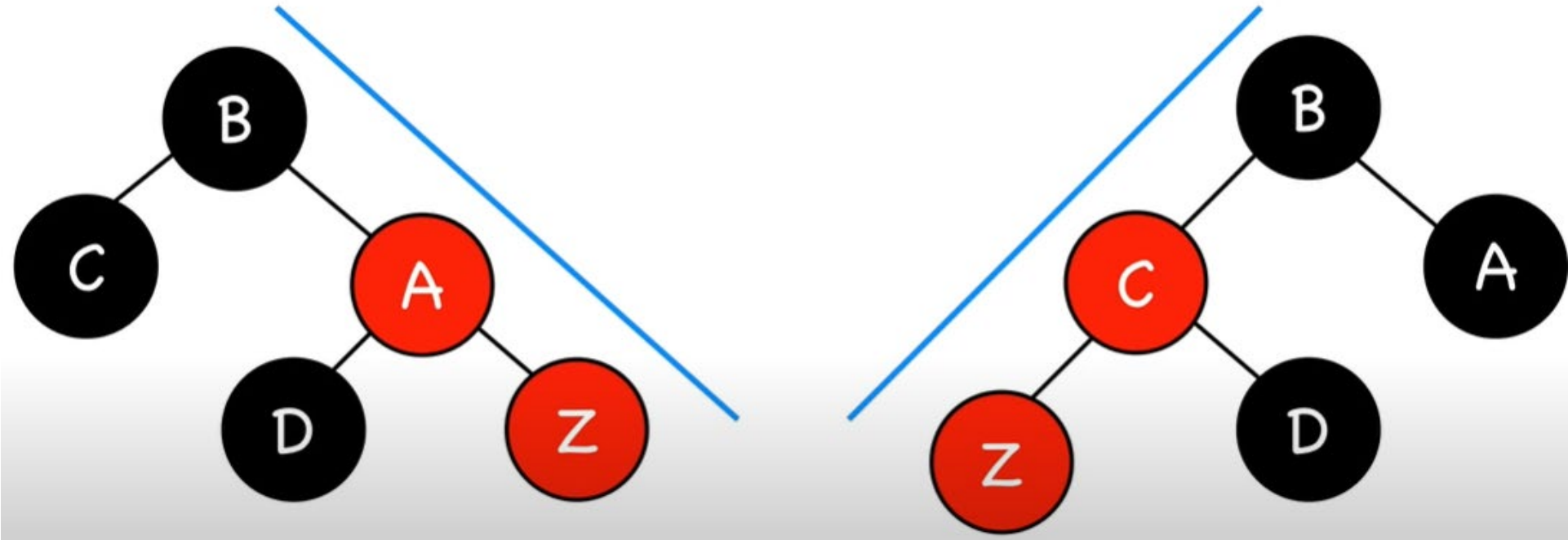
Case 2. Z.uncle = black (triangle)

- Rotate Z.parent
- Turns into Case 3



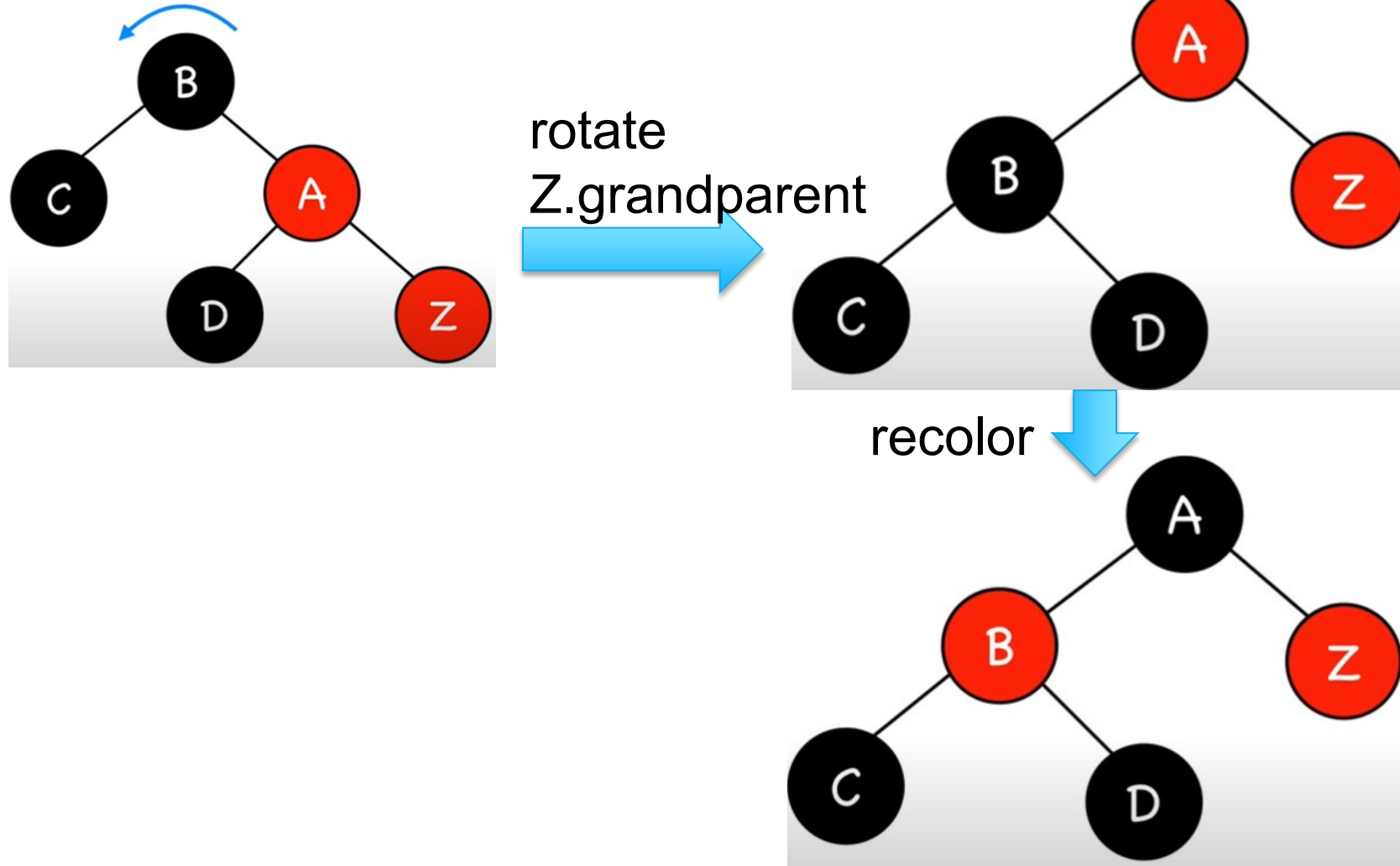
Case 3 Z.uncle = black (line)

case 3 : Z.uncle = black (line)

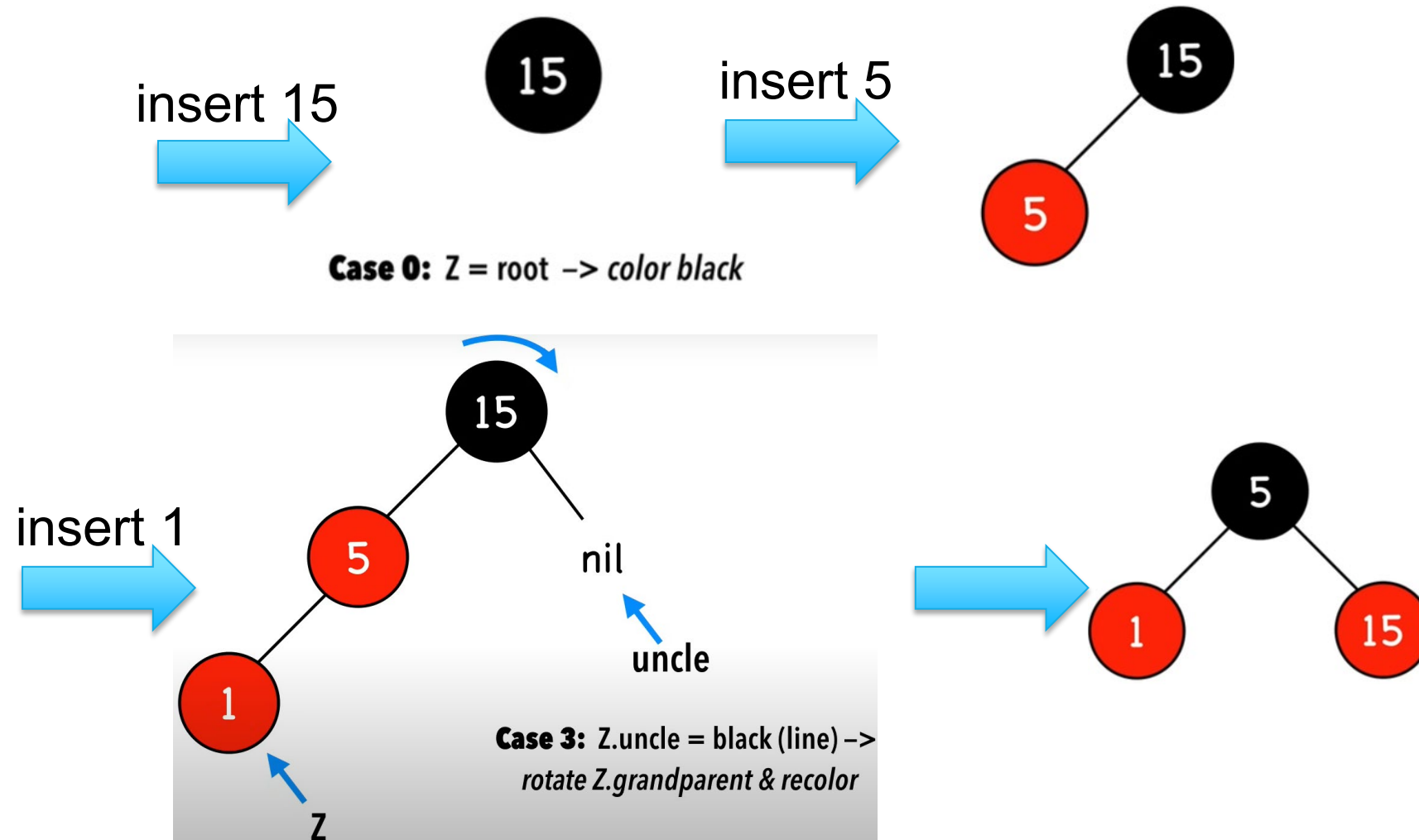


Case 3 Z.uncle = black (line)

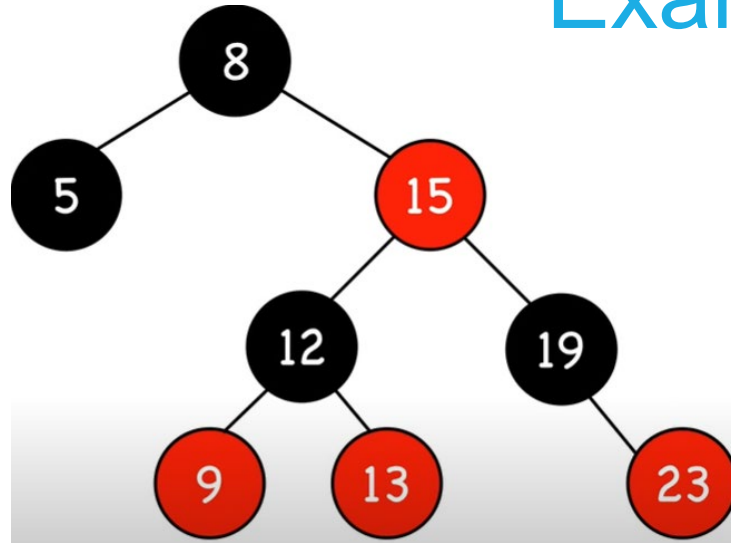
- Rotate Z.grandparent



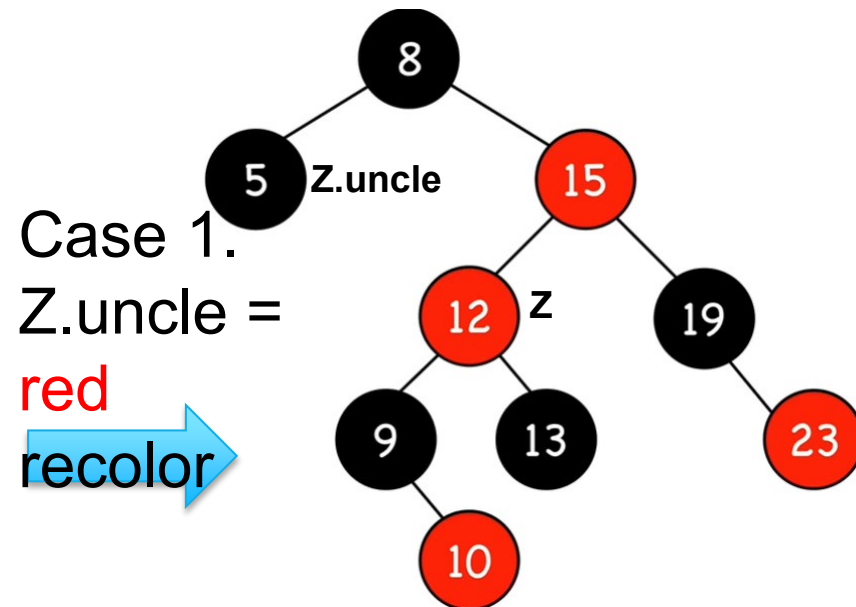
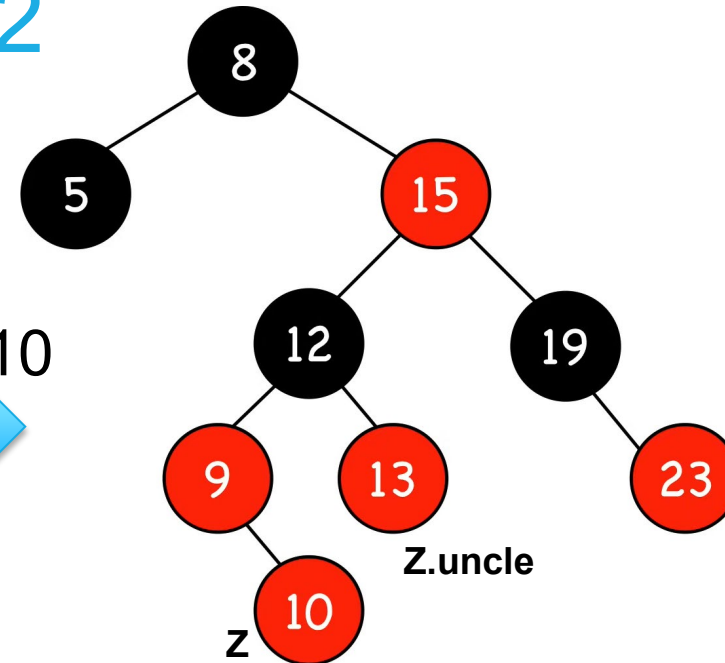
Example 1



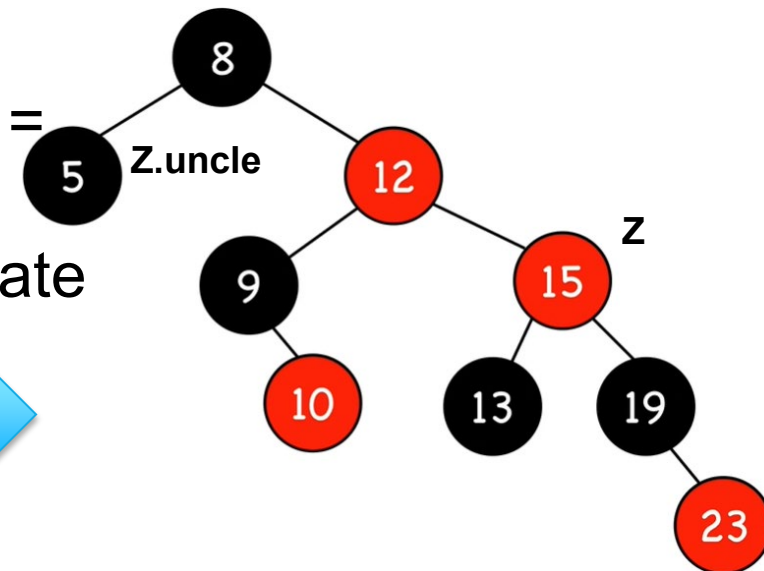
Example 2



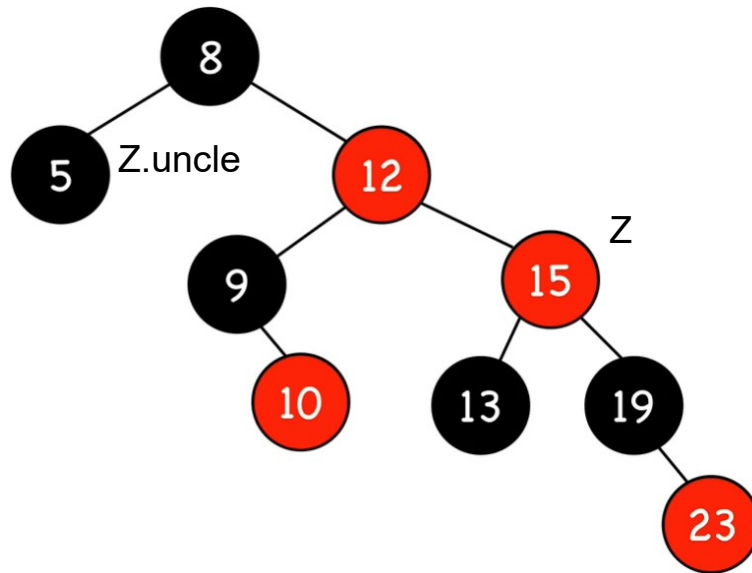
insert 10
→



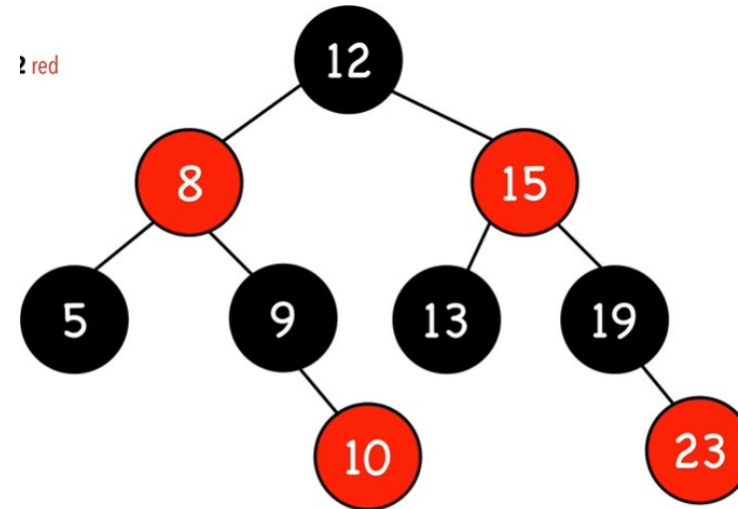
Case 2.
Z.uncle =
black
right rotate
on 15
→



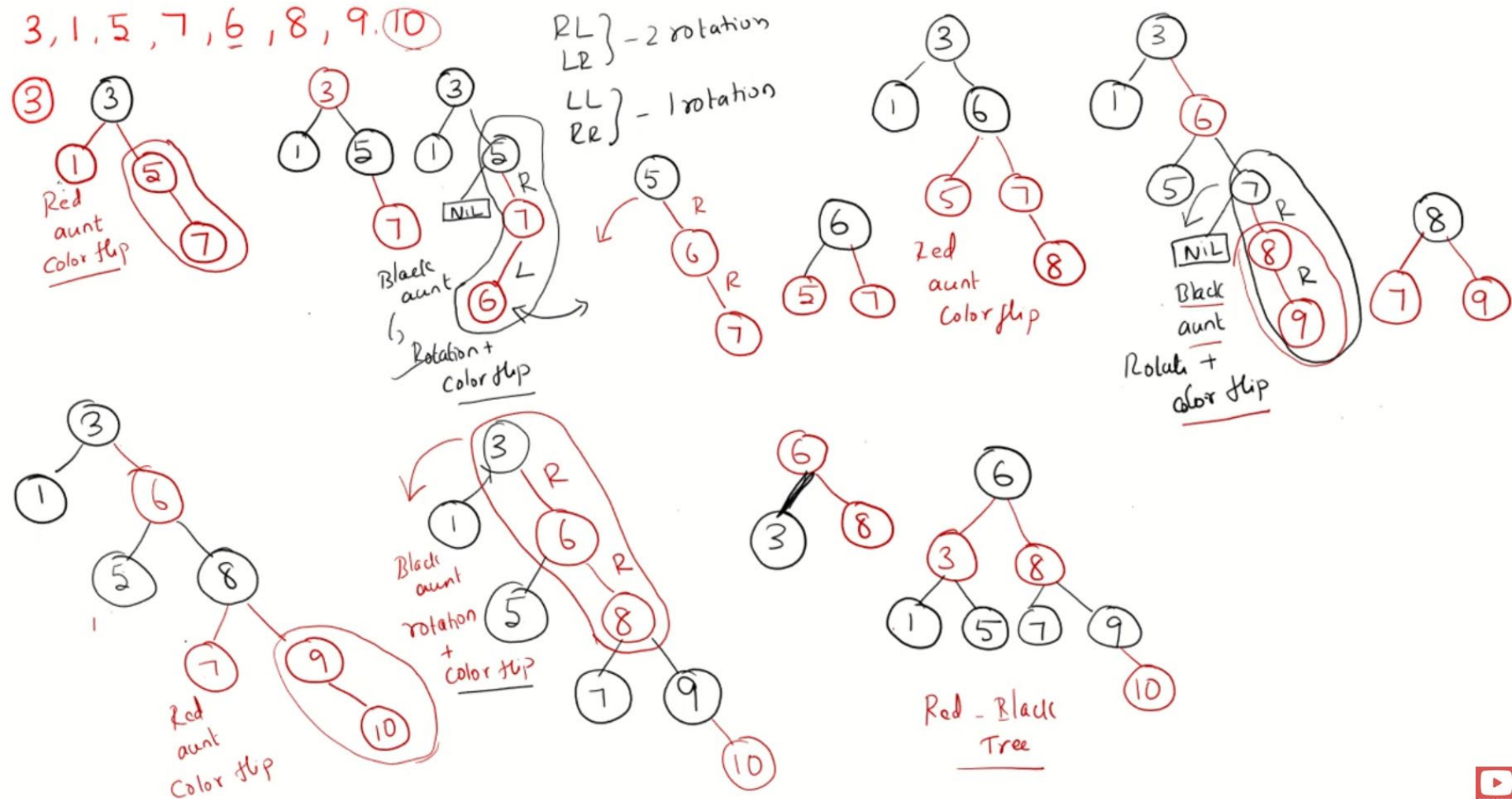
Example 2 Con't



Case 2.
Z.uncle = red
black
left rotate
on 8 &
recolor



Another Example



Red Black Tree – Insertion

<https://www.youtube.com/watch?v=9ubIKipLp>

RU



Time Complexity

- 1. Insert : $O(\log(n))$
 - maximum height of red-black trees
- 2. Color red : $O(1)$
- 3. Fix violations :
 - Constant # of:
 - a. Recolor : $O(1)$
 - b. Rotation: $O(1)$
- Overall time complexity: $O(\log(n))$

AVL vs Red Black Trees

Red Black Tree:

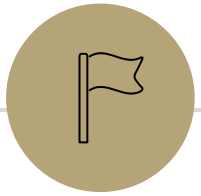
- A balanced BST that maintains the invariant: the longest path (root to farthest NIL) is no more than twice the length of the shortest path (root to nearest NIL).
- Better performance for insertion and deletion because the balancing mechanism is less strict than AVL Tree

AVL Tree:

- A balanced BST that maintains the invariant: $|\text{LeftHeight} - \text{RightHeight}| \leq 1$ for all nodes in the tree.
- Have better look up performance because of their strict balance requirements

Video Tutorials

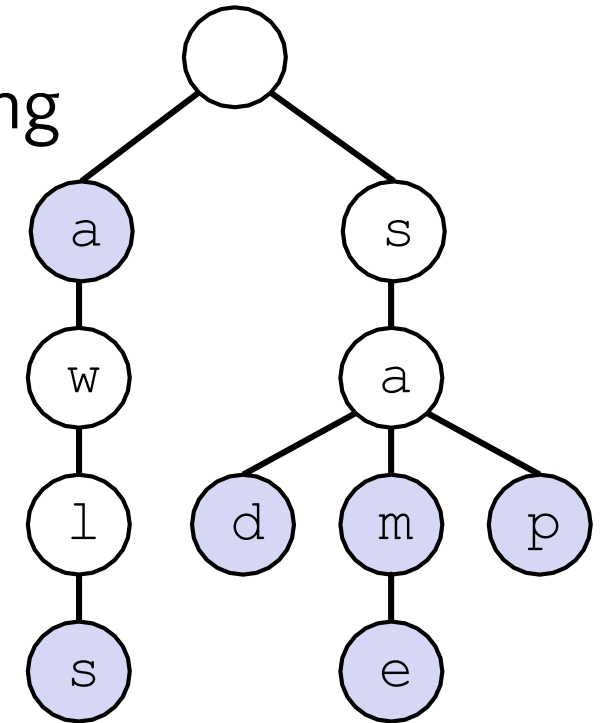
- Red-Black Trees // Michael Sambol
 - https://www.youtube.com/playlist?list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin
 - Lecture slides based in this video series
- Red Black Tree – Insertion
 - <https://www.youtube.com/watch?v=9ubIKipLpRU>
- Introduction to Red-Black Tree
 - <https://www.geeksforgeeks.org/introduction-to-red-black-tree/>



Tries

Trie: An Introduction

- Tries view its keys as:
 - a **sequence of characters**
 - some (hopefully many!) sequences share common prefixes
- Each level of the tree represents an index in the string
 - Children at that level represent possible characters at that index
- This abstract trie stores the set of strings:
 - `awls`, `a`, `sad`, `same`, `sap`, `sam`
- How to deal with `a` and `awls`?
 - Mark which nodes *complete* a string (shown in purple)

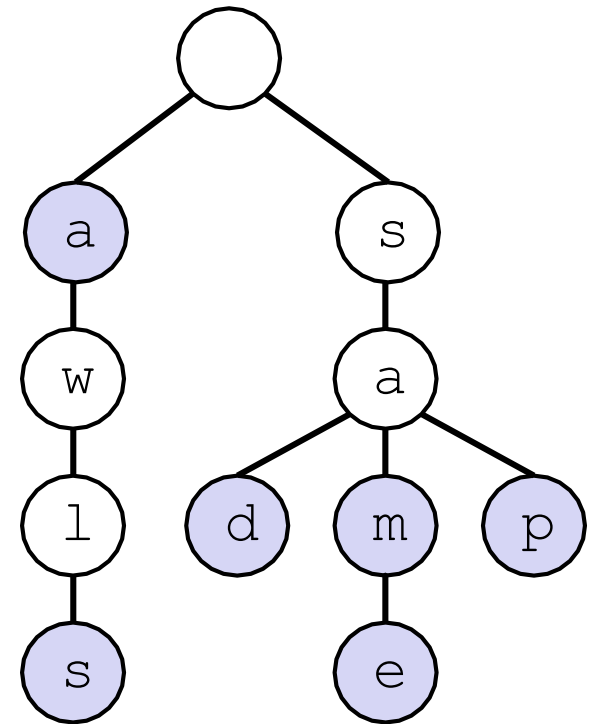


Searching in Tries

Two ways to fail a contains() check:

1. If we fall off the tree
2. If the final node isn't purple (not a key)

<i>Input String</i>	Fall Off? / Is Key?	Result
contains("sam")	hit / purple	True
contains("sa")	hit / white	False
contains("a")	hit / purple	True
contains("saq")	fell off / n/a	False

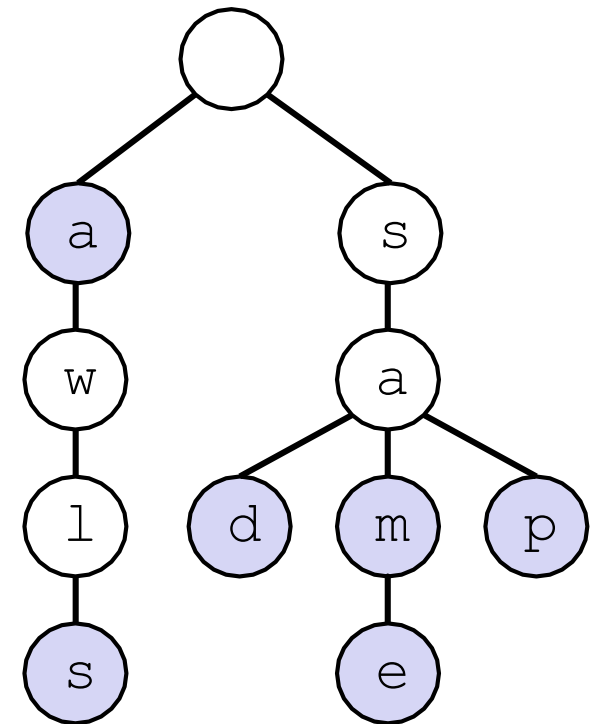


Keys as “a sequence of characters”

- Most dictionaries treat their keys as an “atomic blob”: you can’t disassemble the key into smaller components
- Tries take the opposite view: keys are a **sequence of characters**
 - `Strings` are made of `Characters`
- Tries are defined by 3 types:
 - An “alphabet”: the domain of the characters
 - A “key”: a sequence of “characters” from the alphabet
 - A “value”: the usual Dictionary value

Simple Trie Implementation

```
public class TrieSet {  
    private Node root;  
  
    private static class Node {  
        private char ch;  
        private boolean isKey;  
        private Map<char, Node> next;  
        private Node(char c, boolean b) {  
            ch = c;  
            isKey = b;  
            next = new HashMap();  
        }  
    }  
}
```



Simple Trie Node Implementation

Node

ch	a
isKey	true
next	●

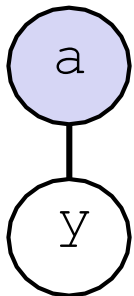
```
private static class Node {  
    private char ch;  
    private boolean isKey;  
    private Map<char, Node> next;  
    ...  
}
```

Map

y	●
---	---

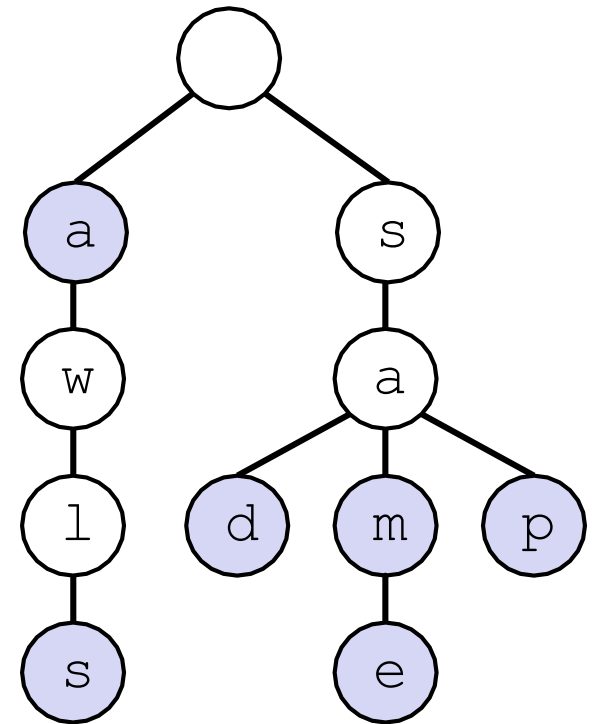
Node

ch	y
isKey	false
next	● → ...



Trie-Specific Operations

- Prefix matching
 - Keys are sequences that can have prefixes
- **Longest prefix**
 - `longestPrefixOf("sample")`
 - Want: {"sam"}
- **Prefix match**
 - `findPrefix("sa")`
 - Want: {"sad", "sam", "same", "sap"}



Summary

- A trie data structure implements the Dictionary and Set
- Tries store sequential keys
 - ... which enables very efficient prefix operations like `findPrefix`
- Tries have many different implementations
 - Could store HashMap/TreeMap/any-dictionary within nodes