CYDEO

Data Structures and Algorithms Course

Trees Review

Today’s Agenda

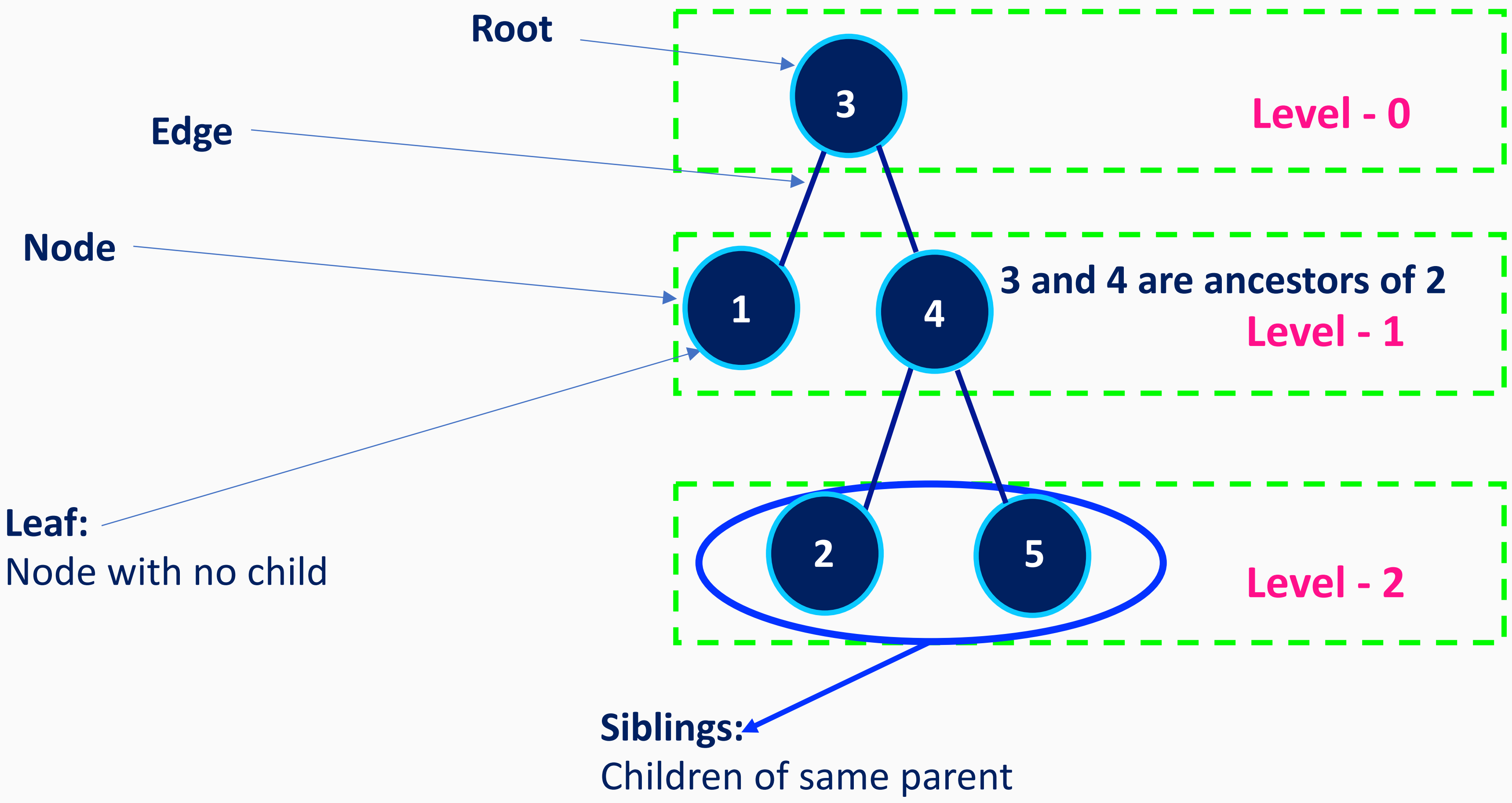
Quick Review

1. Review tree terminology/properties.
2. Review basic implementation of Trees in core Java.(Insertion + Traversals)
3. Review AVL trees.
4. Sample tasks on trees.

Trees Prerequisites

1. Knowledge of Linked Lists.
2. Knowledge of Recursion.
3. Knowledge of Stacks.
4. Knowledge of Queues.

Trees-Terminology



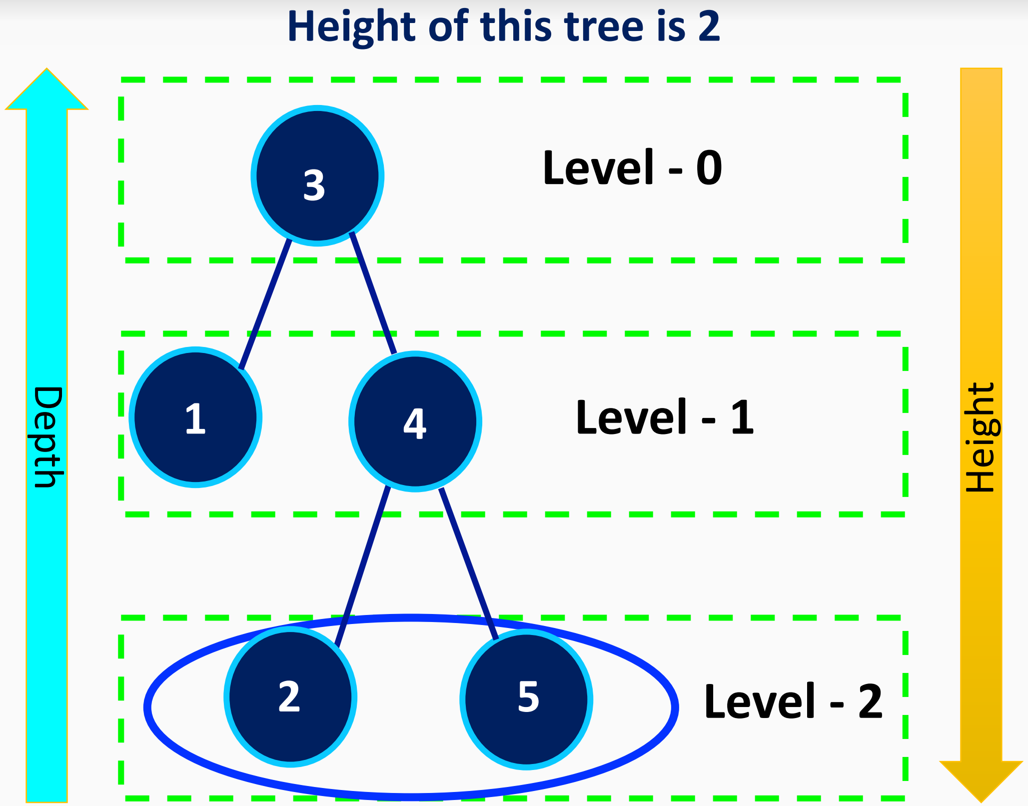
Trees-Depth & Height

The ***depth*** of node ***p*** is the number of ancestors of *p*, other than *p* itself.

- For example Depth of Node with value "5" is 2 since it has two ancestors.
- Depth of root is zero.

Height of a tree is equal to the maximum of the depths of its positions (or zero, if the tree is empty).

- For example Height of tree is 2 since it has max depth of descendants is 2.



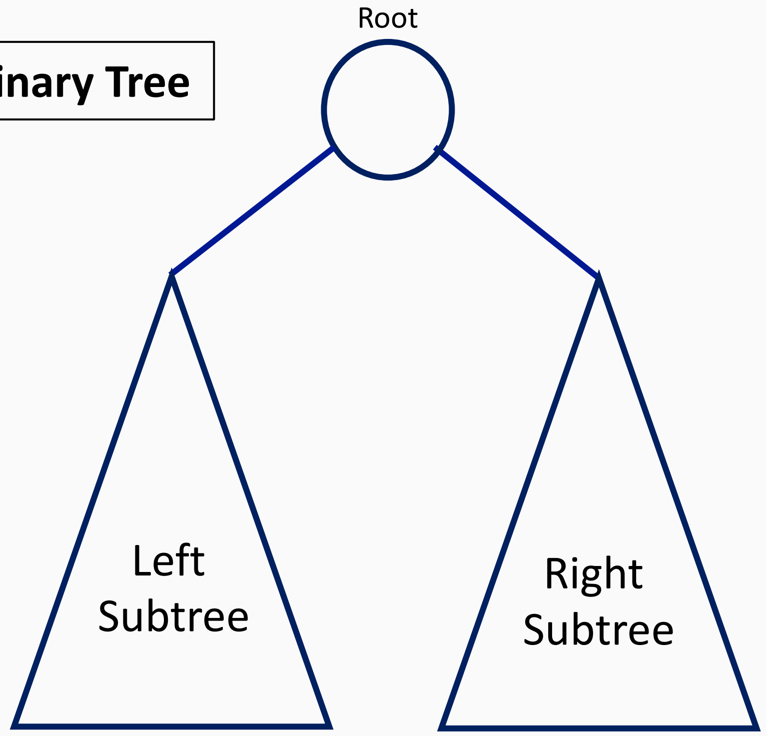
We define the ***height*** of a position *p* in a tree *T* as follows:

- If *p* is a leaf, then the height of *p* is 0.
- Otherwise, the height of *p* is one more than the maximum of the heights of *p*'s children.

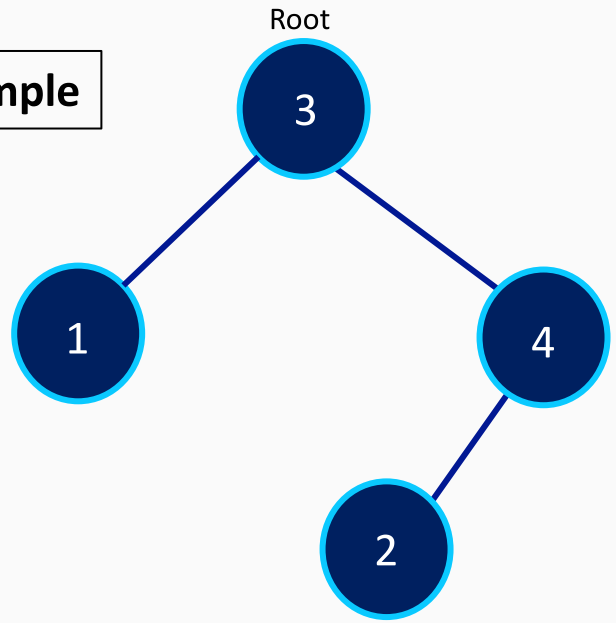
Binary Trees

- A tree is called **binary tree** if each node has **zero child**, **one child** or **two children**.
- Empty tree is also a valid binary tree.

Generic Binary Tree



Example



Binary Trees

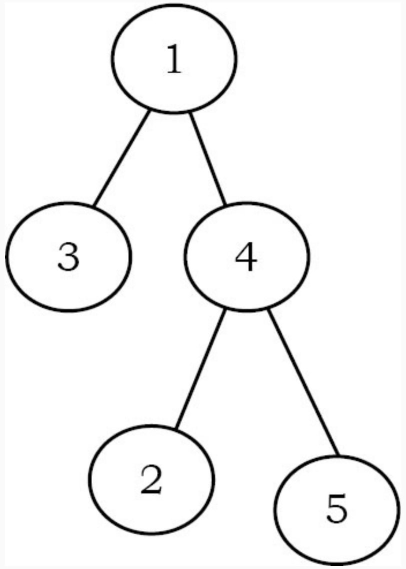
- **Types of Binary Trees:**

Strict Binary Tree: Each node has exactly two children or no children.

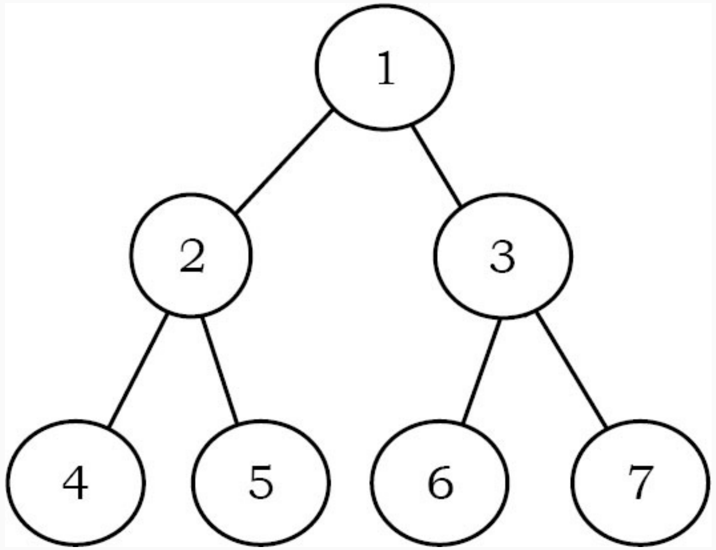
Full Binary Tree: Each node has exactly two children and all leaf nodes are at the same level.

Complete Binary Tree: Every level except the last is completely filled and levels are complete from left to right.

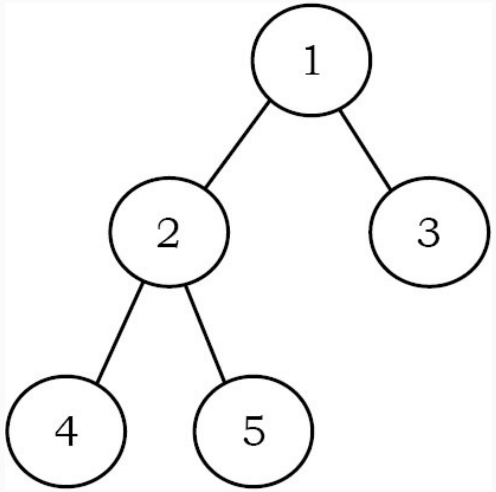
Strict Binary Tree



Full Binary Tree



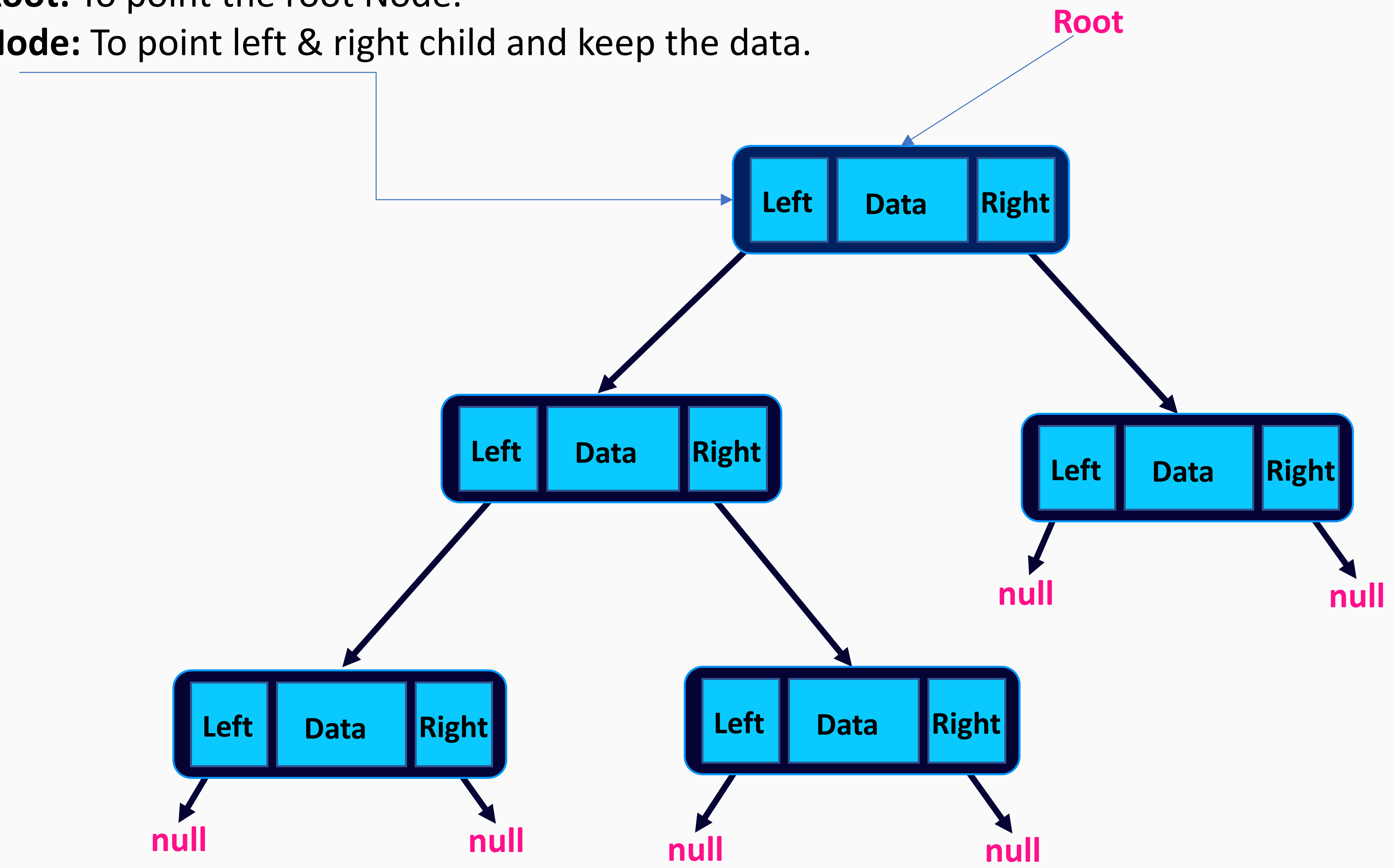
Complete Binary Tree



Implementation of Binary Trees

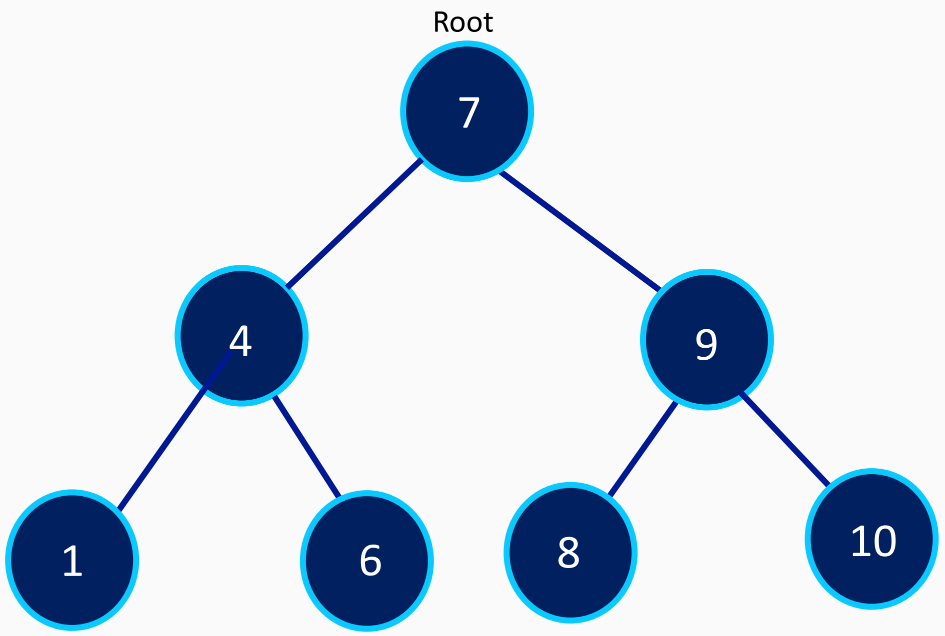
Root: To point the root Node.

Node: To point left & right child and keep the data.

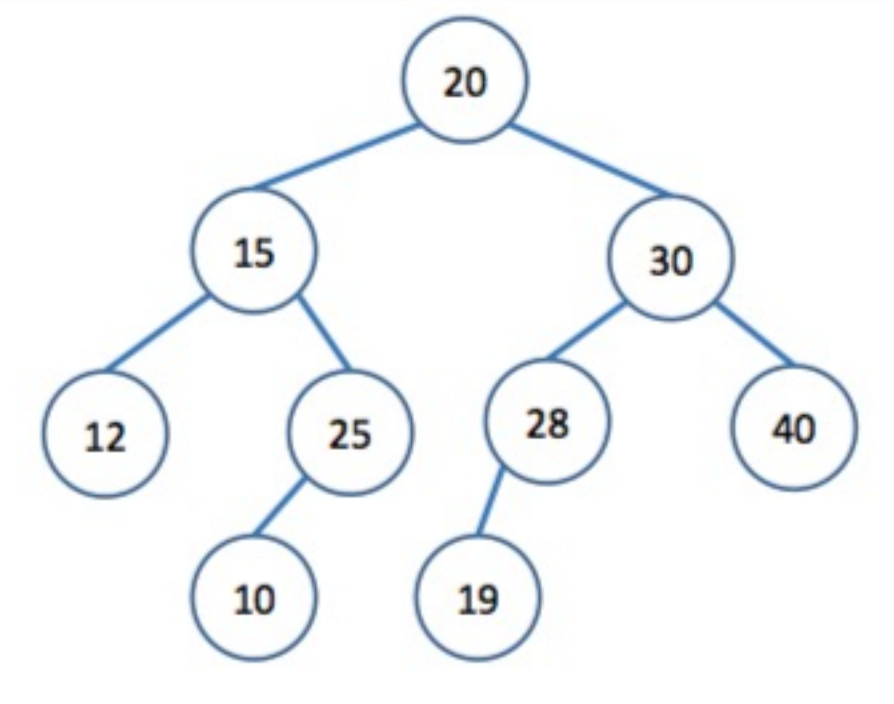


Binary Search Tree

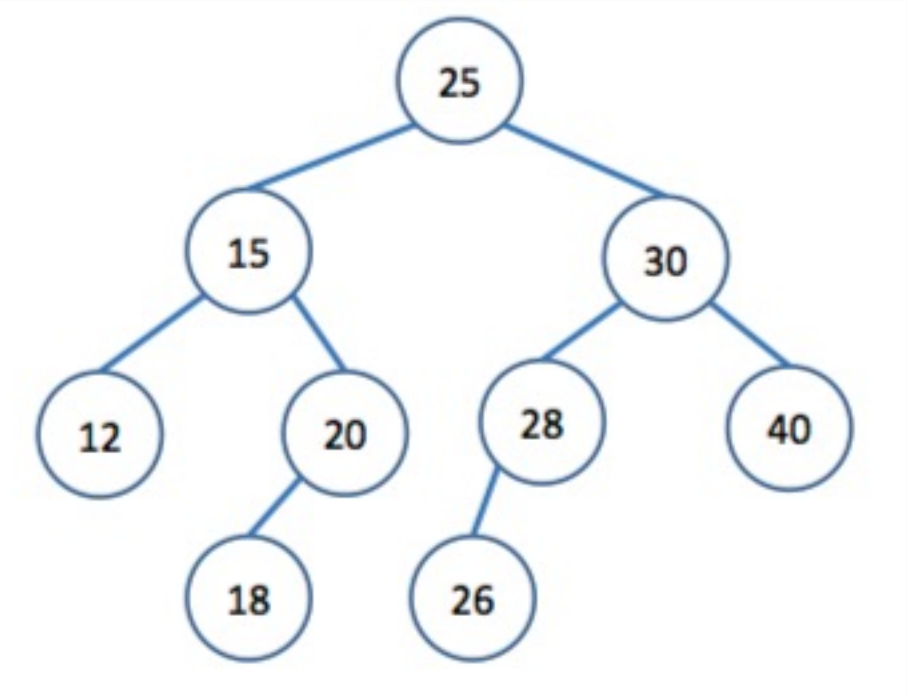
- **Binary Search Tree** has the following properties:
  - ✓ The left subtree of a node contains only nodes with keys lesser than the node’s key.
  - ✓ The right subtree of a node contains only nodes with keys greater than the node’s key.
  - ✓ The left and right subtree each must also be a binary search tree.



BST or Not?



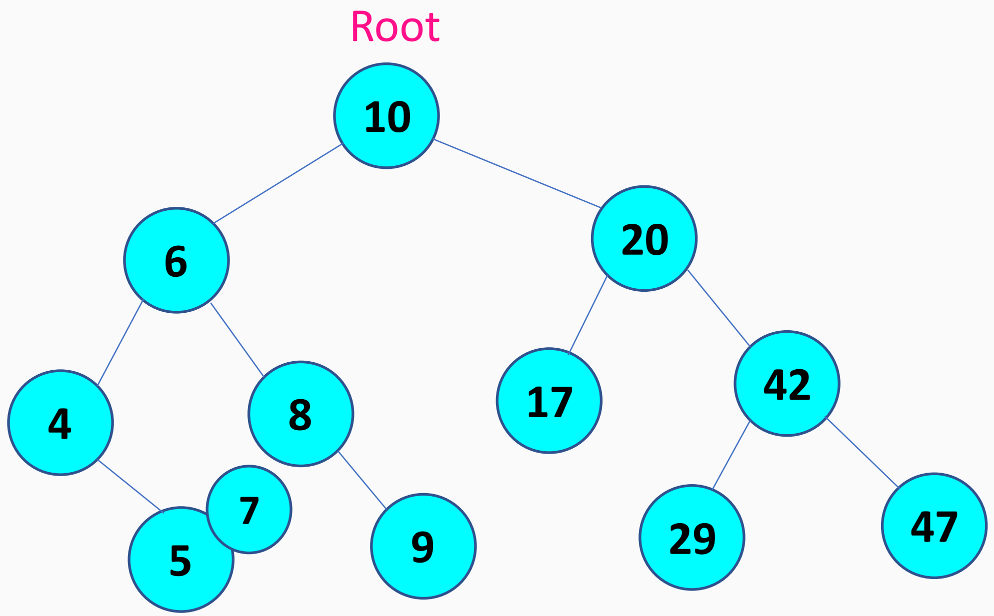
Not Valid BST



Valid BST

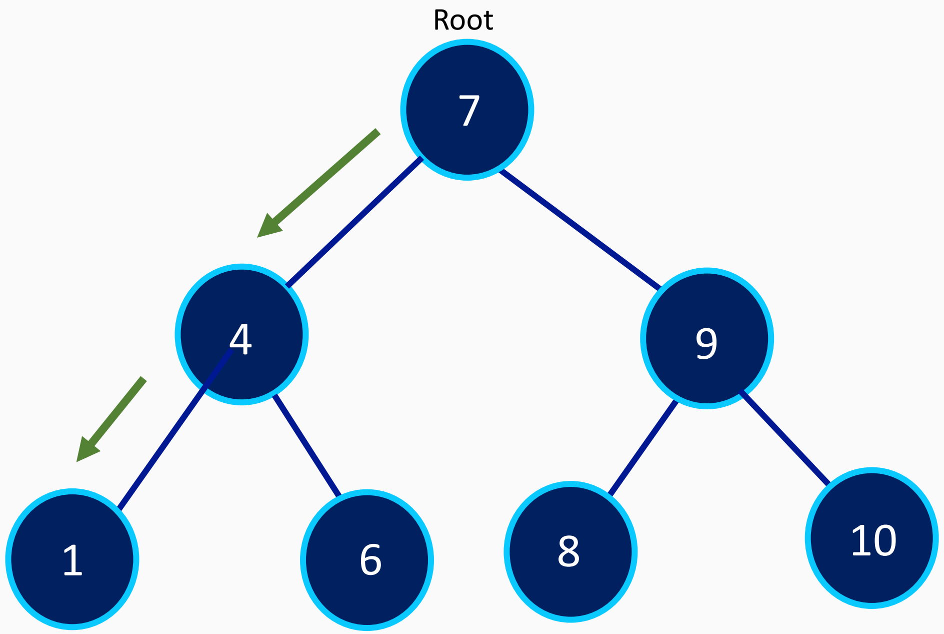
How to Build a BST?

- Given array of integers { 10, 6, 8, 20, 4, 9, 5, 17, 42, 47, 29}, build a BST



Performance of Operations on Binary Trees

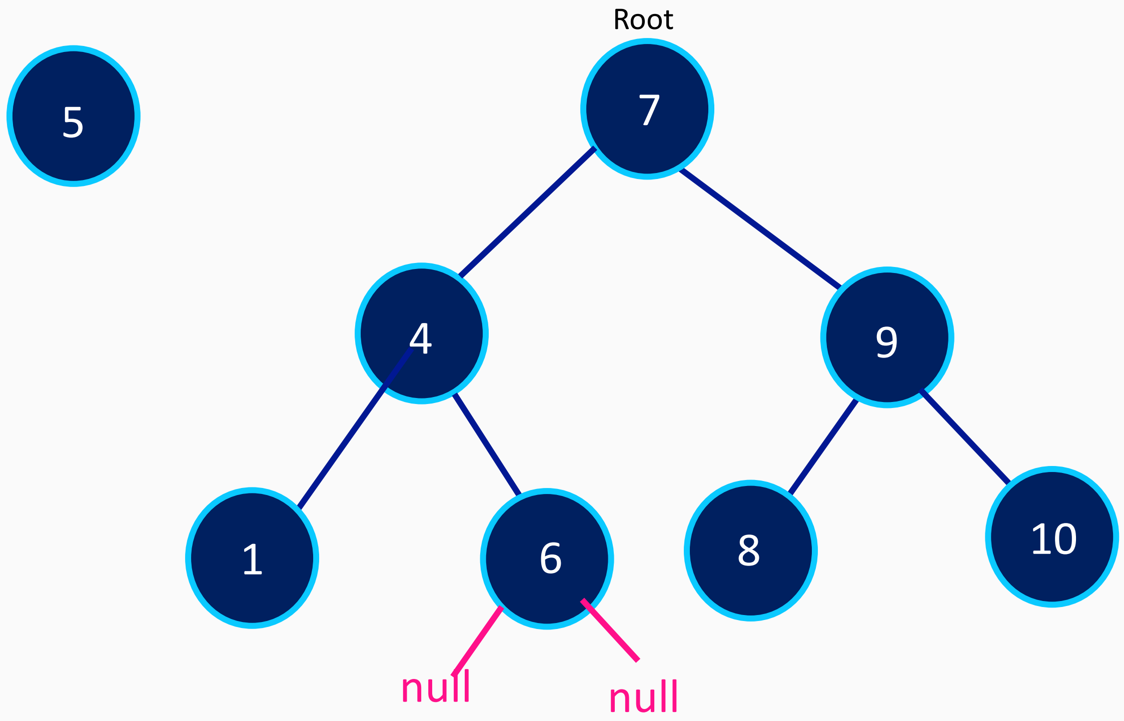
Binary Search Tree



Lookup	O(log n)
Insert	O(log n)
Delete	O(log n)

Assume you are searching smallest value ‘1’. You can reach the value with 3 comparisons. This is what we call as LOGARITHMIC time complexity. So Lookup an item is **O(log n).**

Insertion into a Binary Tree



Traversing trees

Two main types of Traversals:

1. **Breadth First** – Level Order
2. **Depth First**
   - Pre-Order
   - In-Order
   - Post-Order

Traversing trees

DEPTH FIRST

Pre-order	Root, Left, Right
In-order	Left, Root, Right
Post-order	Left, Right, Root

For each sub-tree

Traversing trees

Depth First (Pre Order)

PRE-ORDER

Root, Left, Right

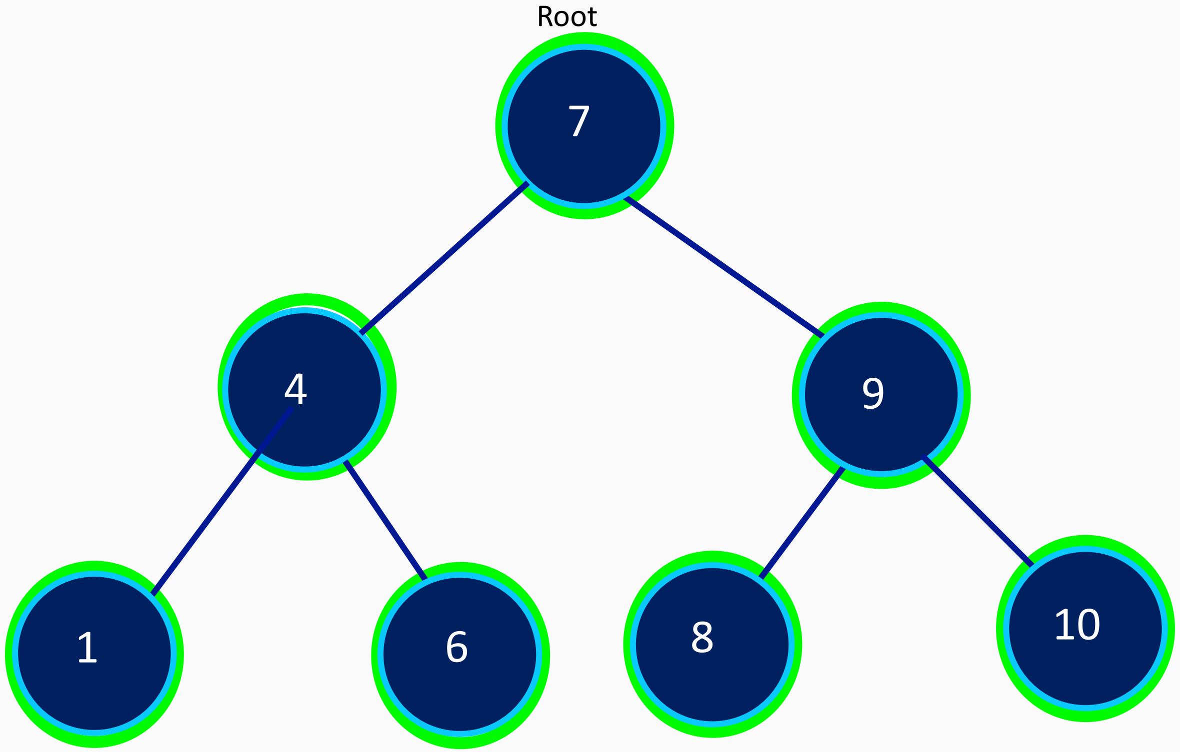
(For each Subtree)

<u>Algorithm</u> <u>preorder(*p*):</u>

1. perform the "visit" action for position *p*. // this happens before any recursion
2. for each child *c* in children(*p*) **do** // from left to right
   - preorder(*c*) // recursively traverse the subtree rooted at *c*

Traversing trees

Depth First (Pre Order)



PRE-ORDER

Root, Left, Right

For each Subtree

Visit order : 7, 4, 1, 6, 9, 8,10

Traversing trees

Depth First (In-Order)

IN-ORDER

Left, Root, Right

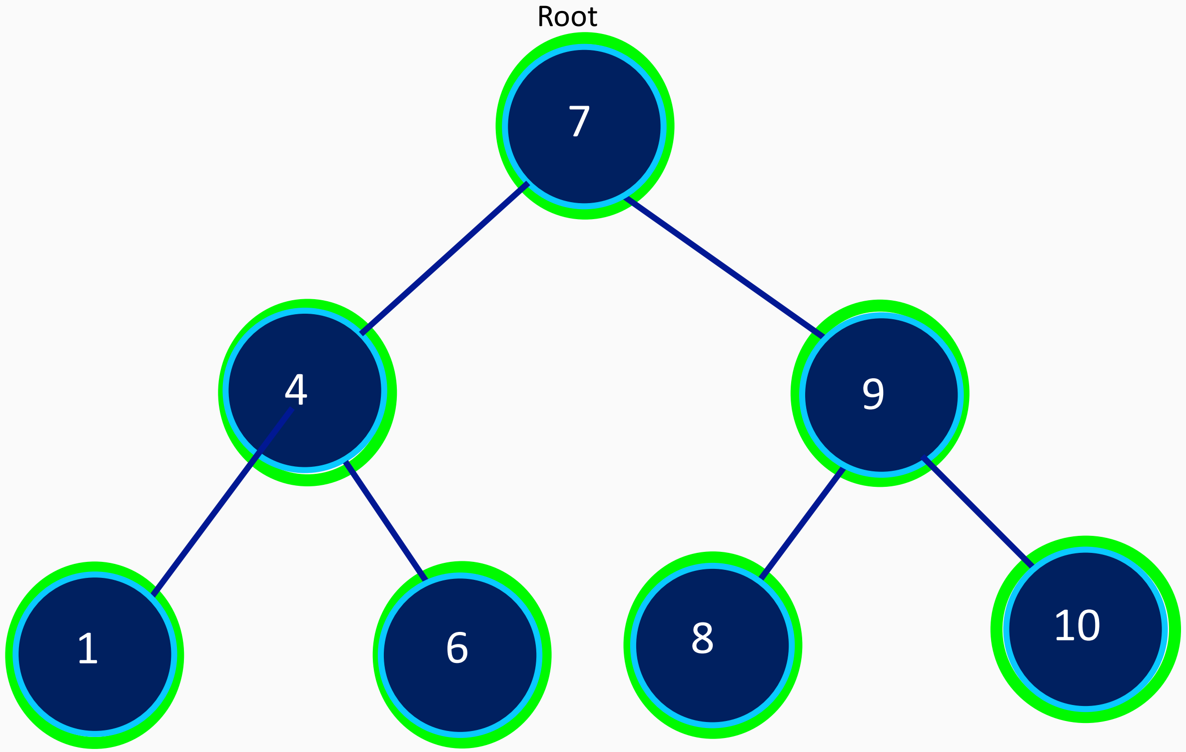
(For each Subtree)

An in-order traversal on a tree performs the following steps starting from the root:

1) Traverse the left subtree by recursively calling the in-order function.
2) Return the root node value.
3) Traverse the right subtree by recursively calling the in-order function.

Traversing trees

Depth First (In-Order)



IN-ORDER

Left, Root, Right

For each Subtree

Visit order : 1, 4, 6 ,7 ,8, 9, 10

Ascending order !

Traversing Trees

Depth First (Post Order)

POST-ORDER

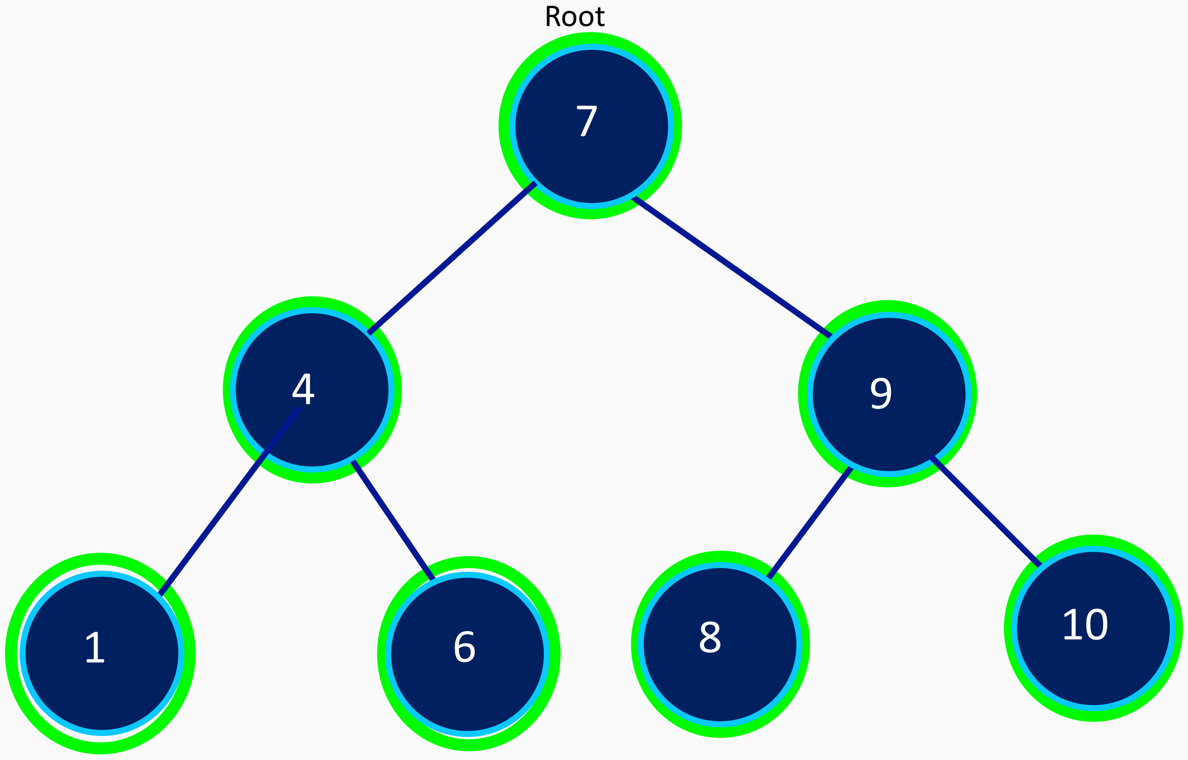
Left, Right, Root

For each Subtree

Algorithm postorder(*p*):
1. for each child *c* in children(*p*) **do** // from left to right
  -postorder(*c*) // recursively traverse the subtree rooted at *c*
2. perform the "visit" action for position *p* // this happens after any recursion

Traversing Trees

Depth First (Post Order)



Visit order : 1, 6, 4, 8, 10, 9, 7

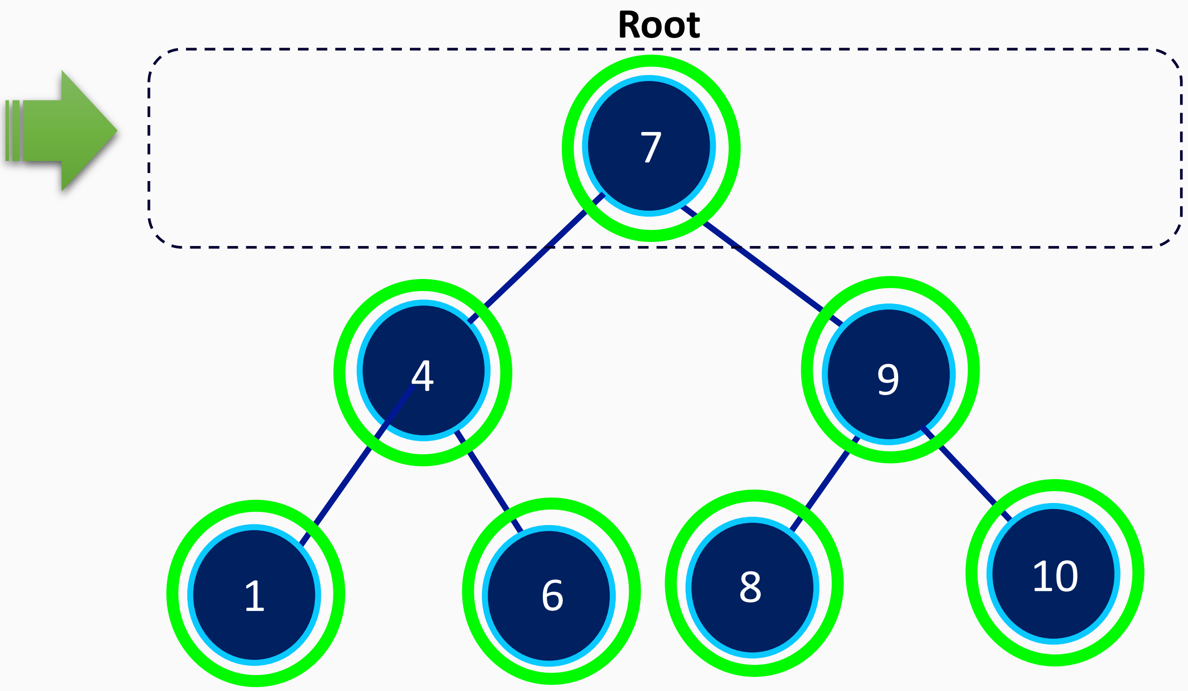
POST-ORDER

Left, Right, Root

For each Subtree

Traversing trees

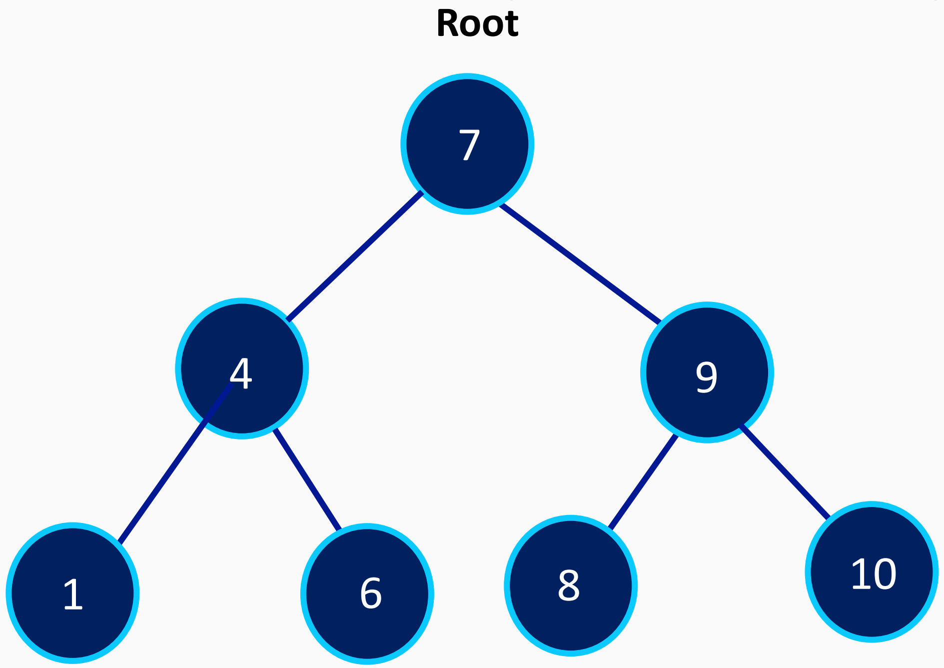
Breadth First (Level Order)



Visit order : 7, 4, 9, 1, 6, 8,10

Traversing trees

Breadth First (Level Order)



Algorithm breadthfirst():
Initialize queue *Q* to contain root()
while *Q* not empty **do**
    p = *Q*.dequeue()
    perform the "visit" action for position *p*
 for each child *c* in children(*p*) **do**
      Q.enqueue(c)

Queue

7 4	9	1	6	8	10	

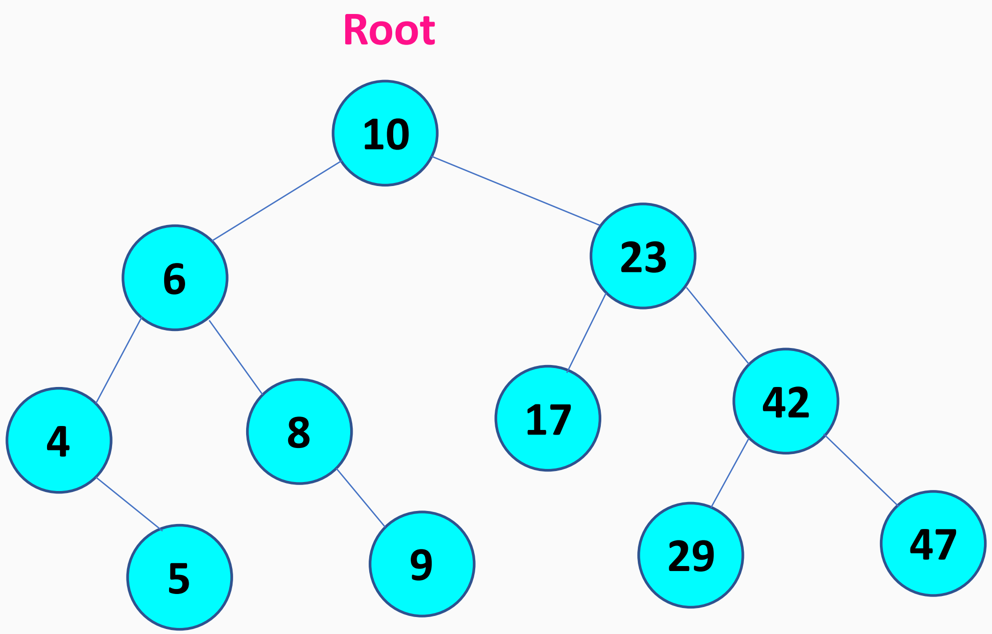
front back

Visited:

Deletion from a BST

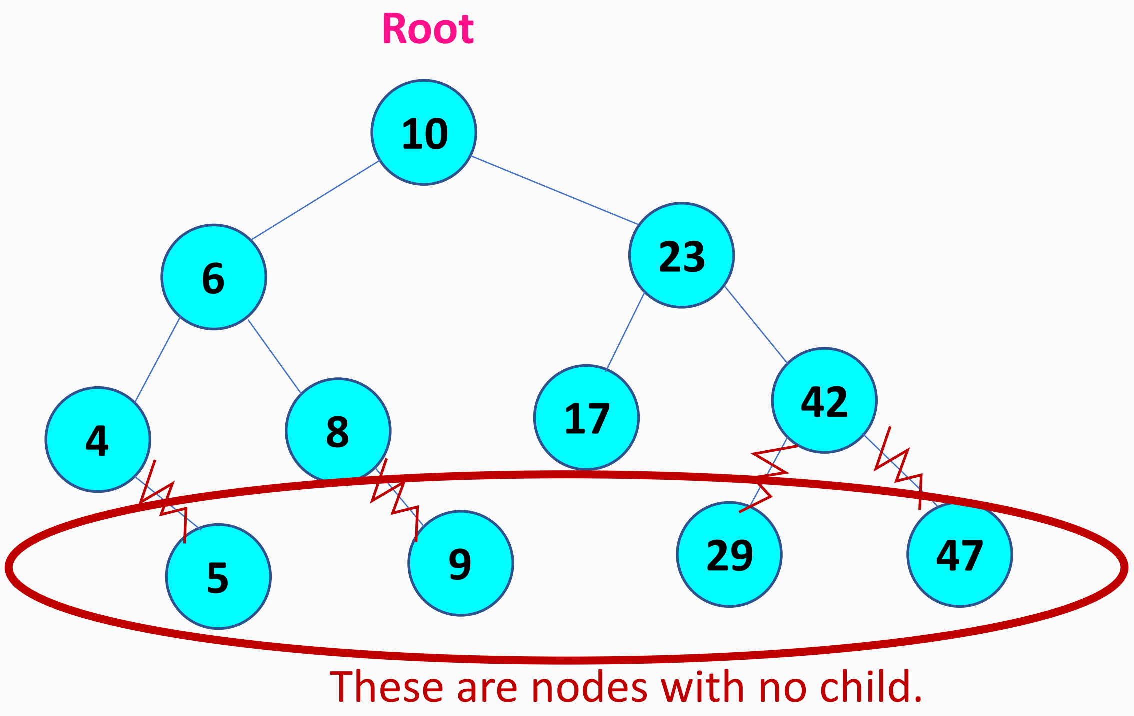
Cases:

1. No child
2. One Child
3. Two Children
   - In-Order Predecessor
   - In-Order Successor



Deletion from a BST

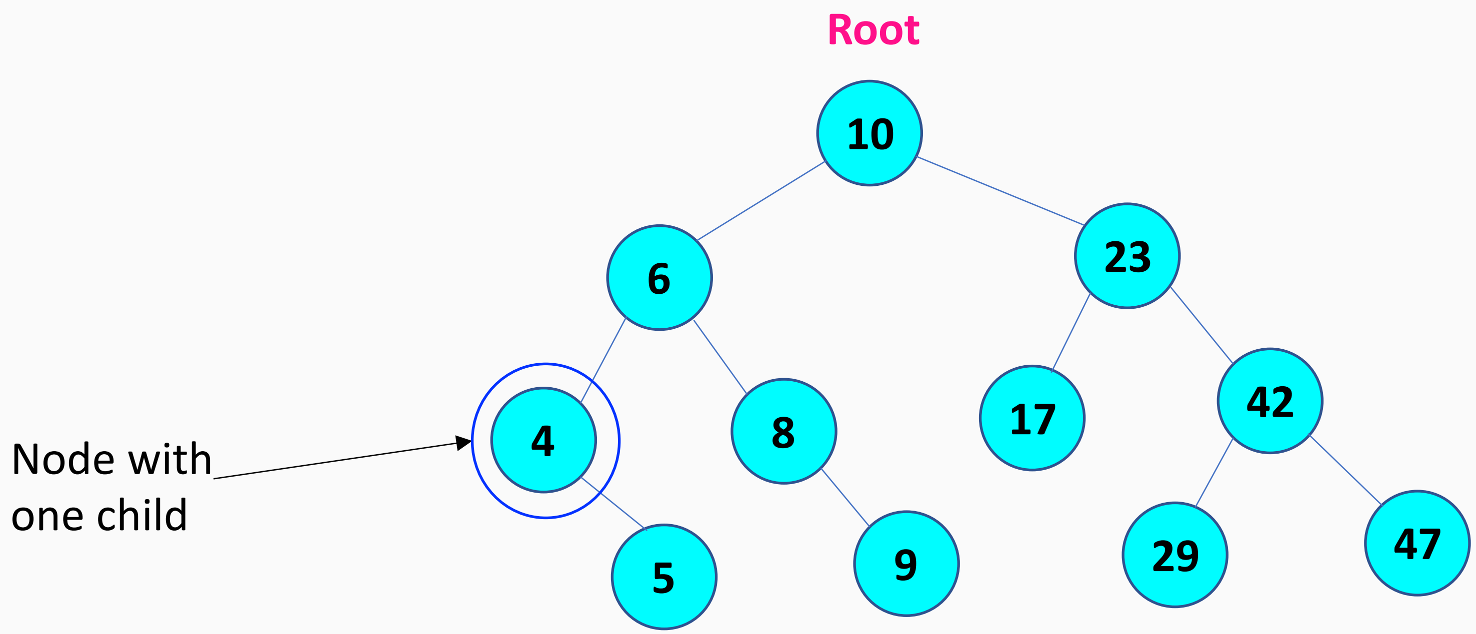
Case 1: **No child**



- Just remove the the link between node to be deleted and its ancestor.

Deletion from a BST-One Child

Case 2 : One Child

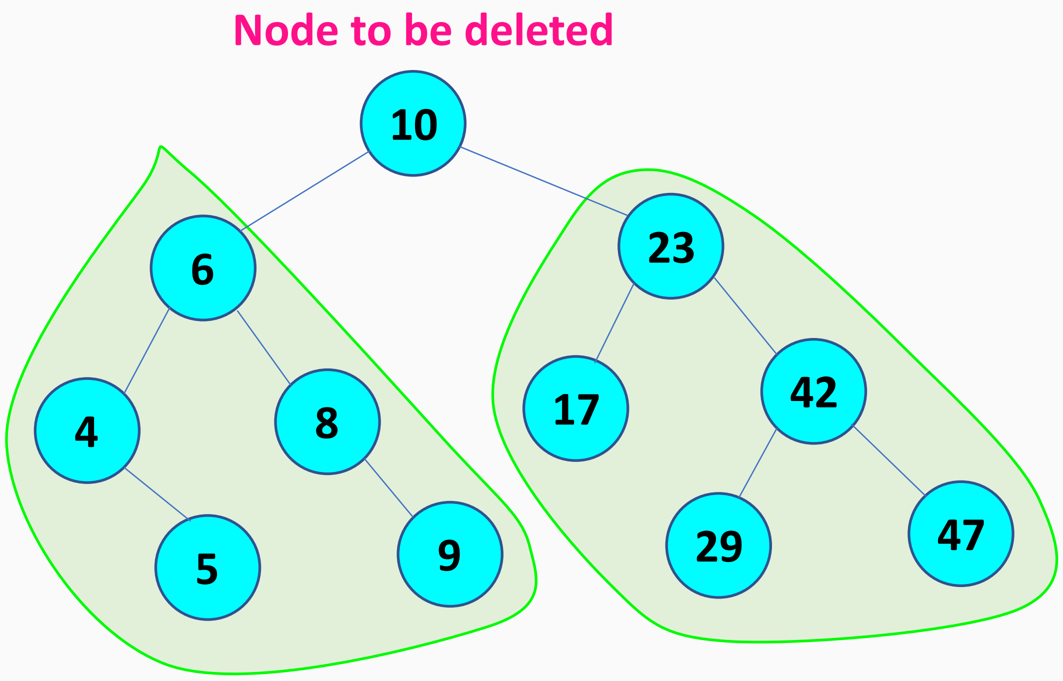


- Remove link of node to be deleted (NTBD) and link ancestor with NTBD's child

Deletion from a BST- Two Children

Case 3: Two Children

- In-Order Predecessor (Maximum of Left Subtree)
- In-Order Successor (Minimum of Right Subtree)



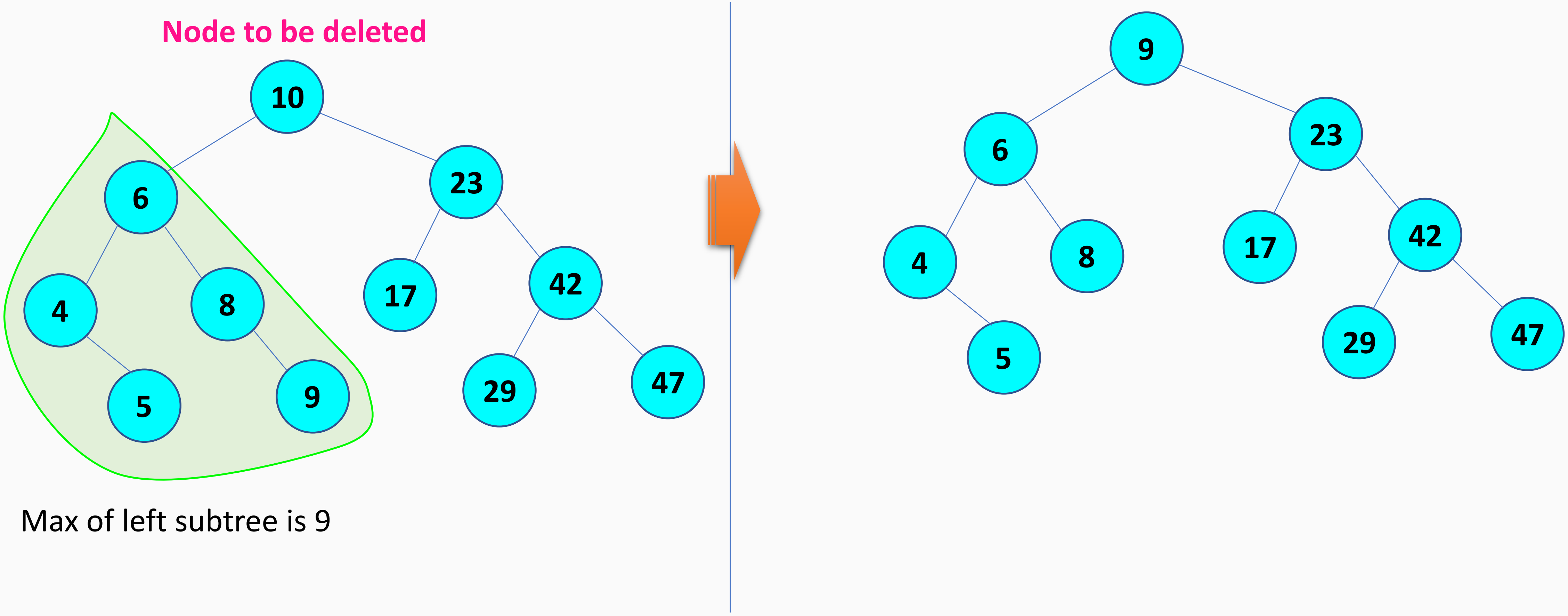
Max of left subtree is 9

Min of right subtree is 17

Deletion from a BST with In-Order Predecessor

<u>Case 3:</u> Two Children

- In-Order Predecessor (Maximum of Left Subtree)

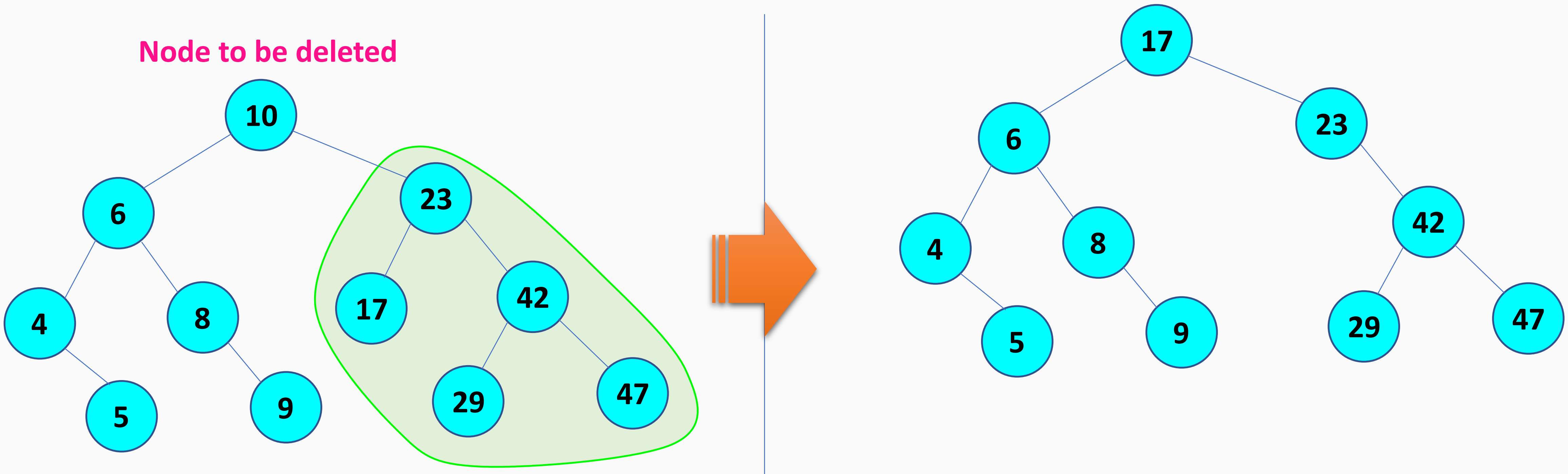


Max of left subtree is 9

Deletion from a BST with In-Order Successor

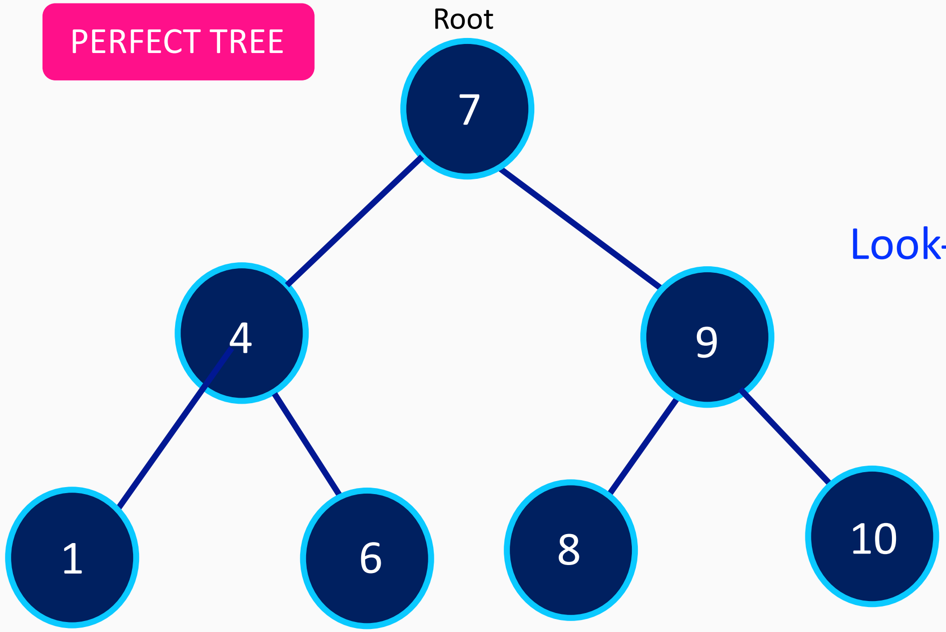
Case 3: Two Children

- In-Order Successor (Minimum of Right Subtree)



Min of right subtree is 17

Balanced and Unbalanced Trees

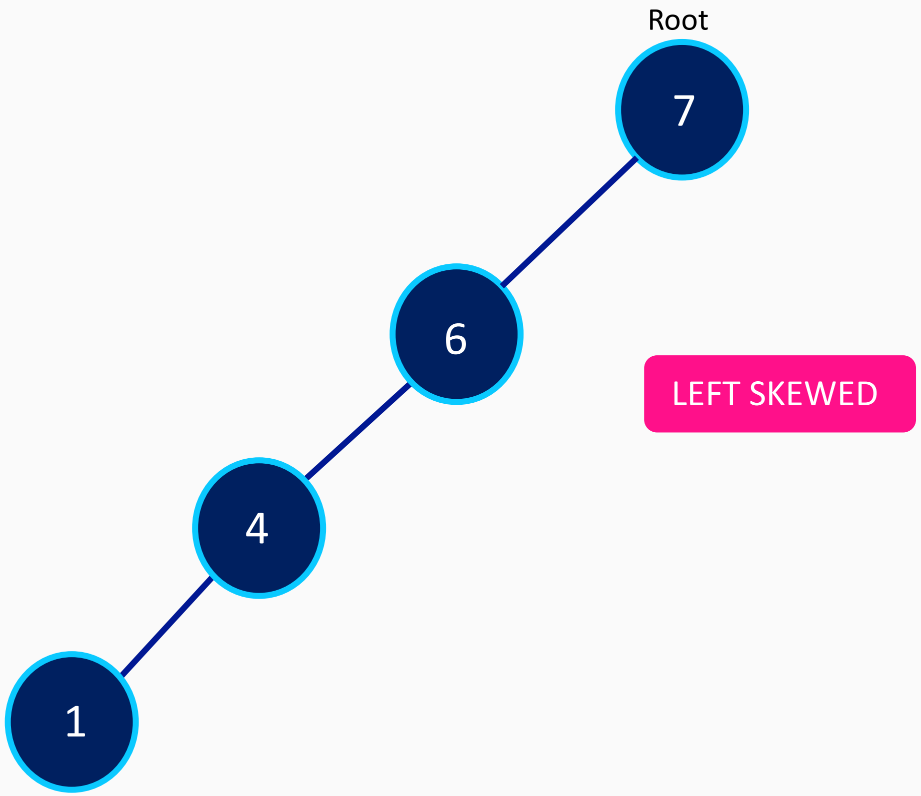


Look-up / Access a value: O (log n)

Insertion Order: 7, 4,9,1,6,8,10

- **Why do we need to balance trees?**
- **We should keep BST property while balancing!**

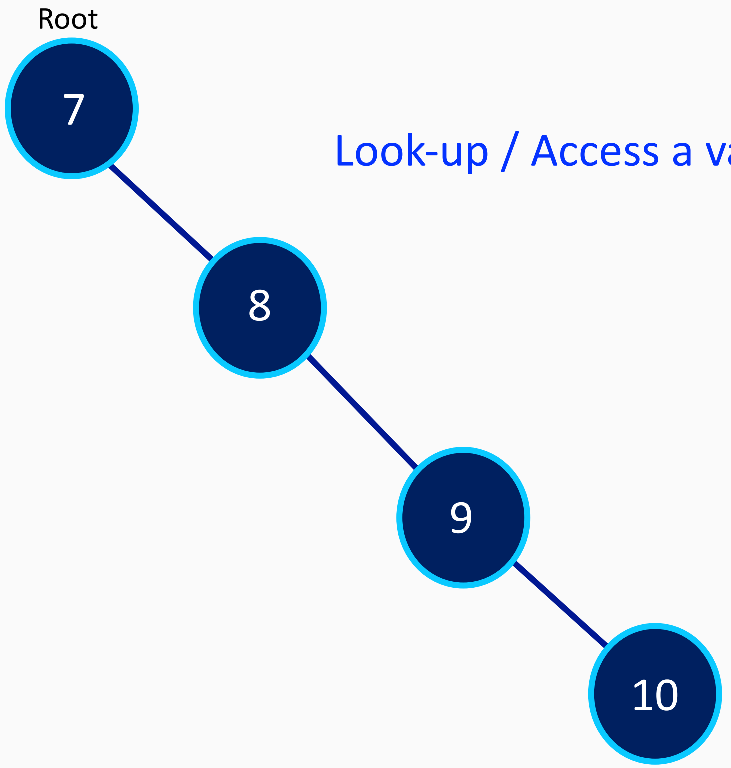
Balanced and Unbalanced Trees



Look-up / Access a value: O (n) !

Insertion Order: 7, 6,4,1

Balanced and Unbalanced Trees

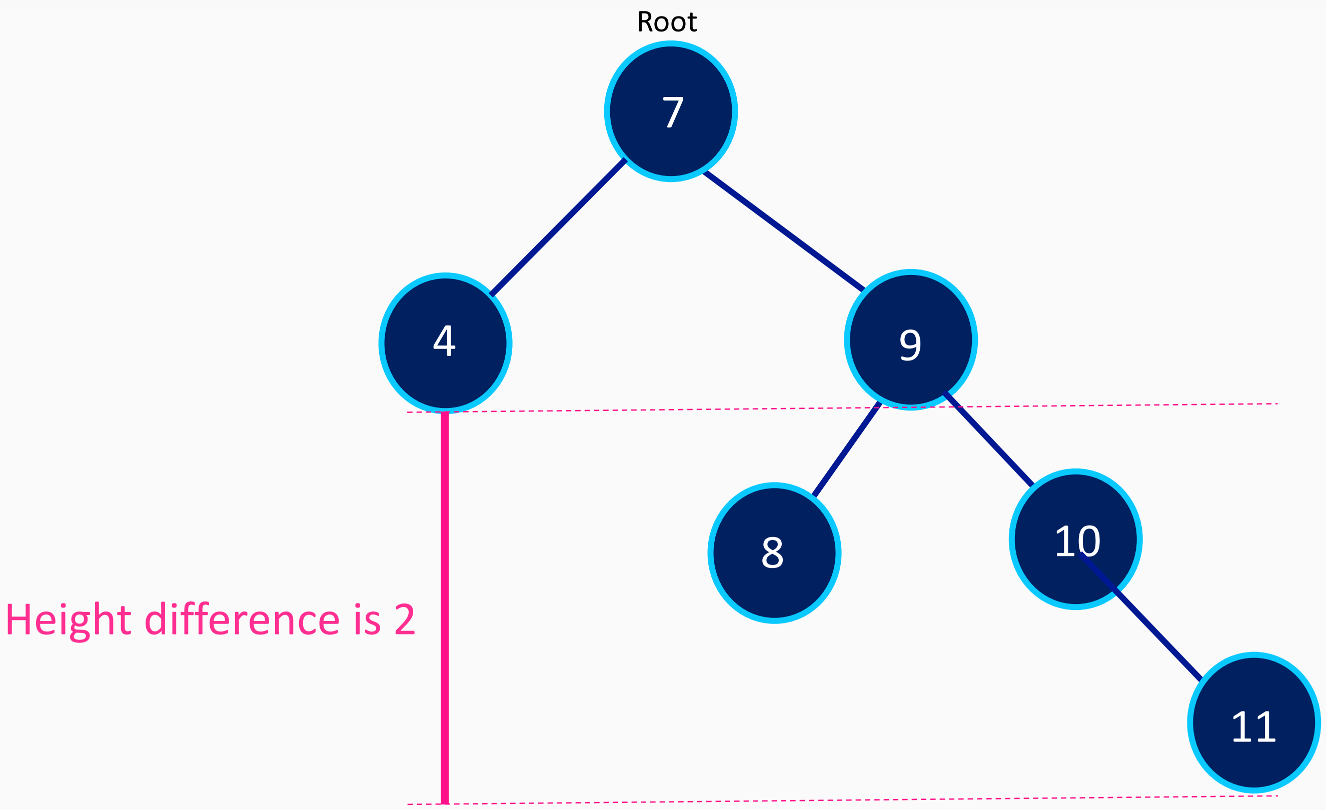


Look-up / Access a value: O (n) !

RIGHT SKEWED

Insertion Order: 7, 8,9,10

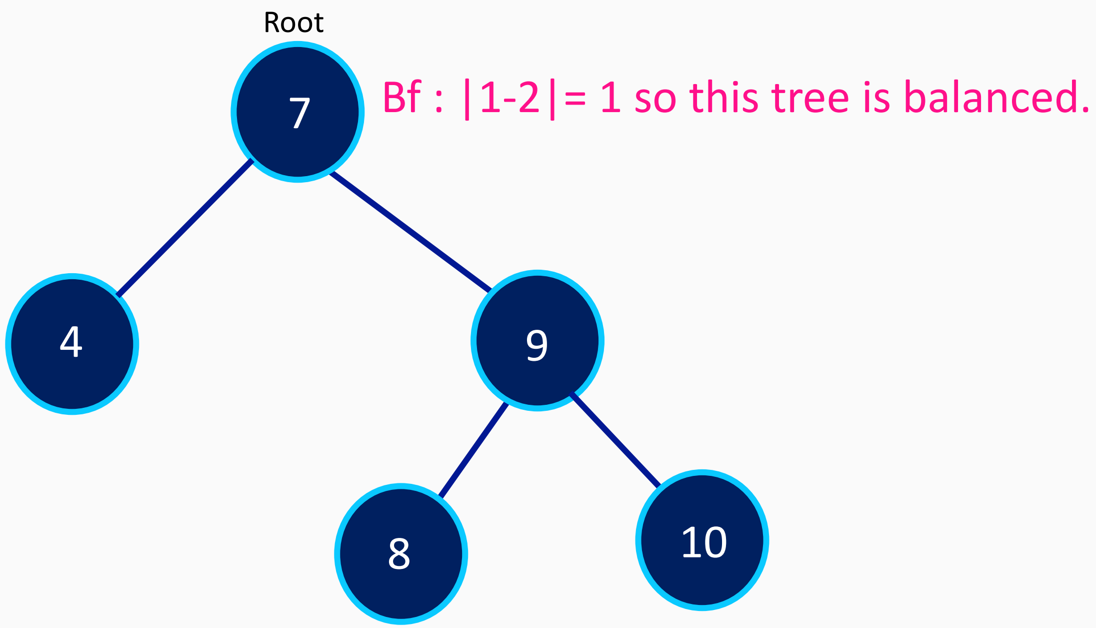
What is imbalance?



How to check the balance of a tree?

Formula:

Balance Factor: |height(left)-height(right)| <=1 then balanced



Height of a Node : Max # of edges from the leaves to that Node

AVL Trees

- **AVL tree** is a self-balancing Binary Search **Tree** (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- After every Insertion/Deletion an auto balance check is executed.
- Other types of self balancing trees are:

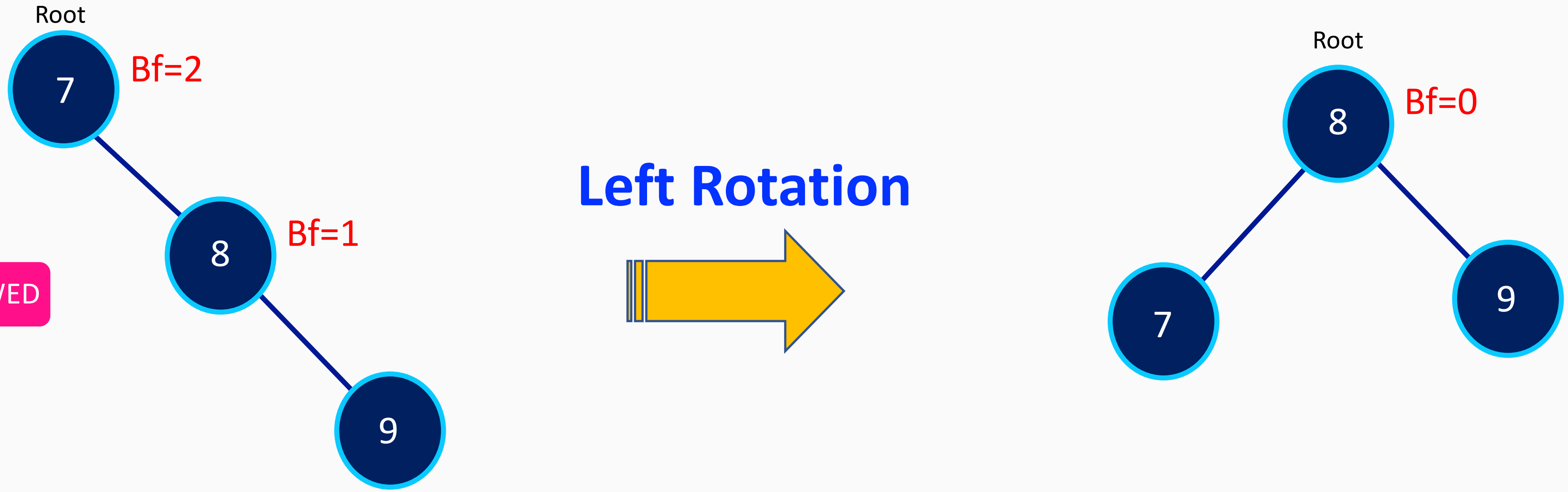
  - 2–3 tree
  - AA tree
  - B-tree
  - Red–black tree
  - Scapegoat tree
  - Splay tree
  - Treap
  - Weight-balanced tree

Balancing Trees

- If there is imbalance we make rotations to balance a tree.
- We check balance after every insertion/deletion.
- It should be a valid BST after any balancing operation.
- There may be 4 cases and 4 kinds of rotation:
  - **Left Rotation** (Right heavy)
  - **Right Rotation** (Left heavy)
  - **Left-Right Rotation** (LR)
  - **Right-Left Rotation** (RL)

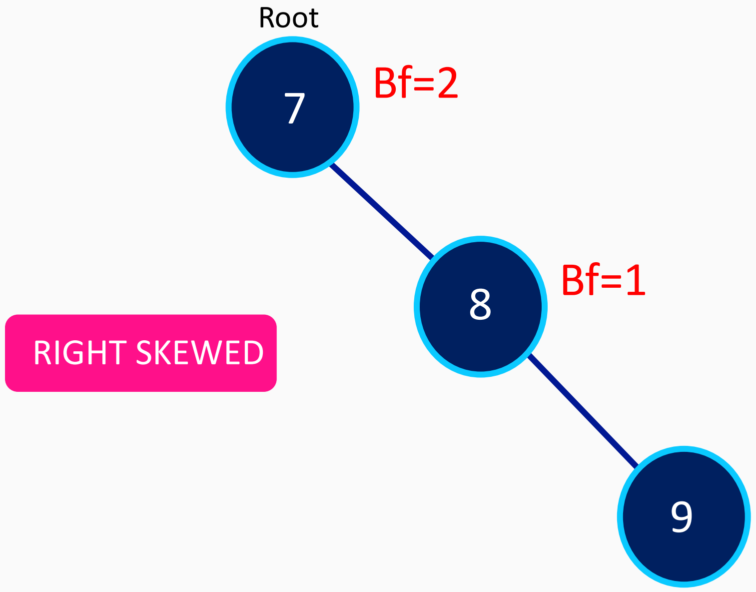
Balancing Trees- Left Rotation

RIGHT SKEWED

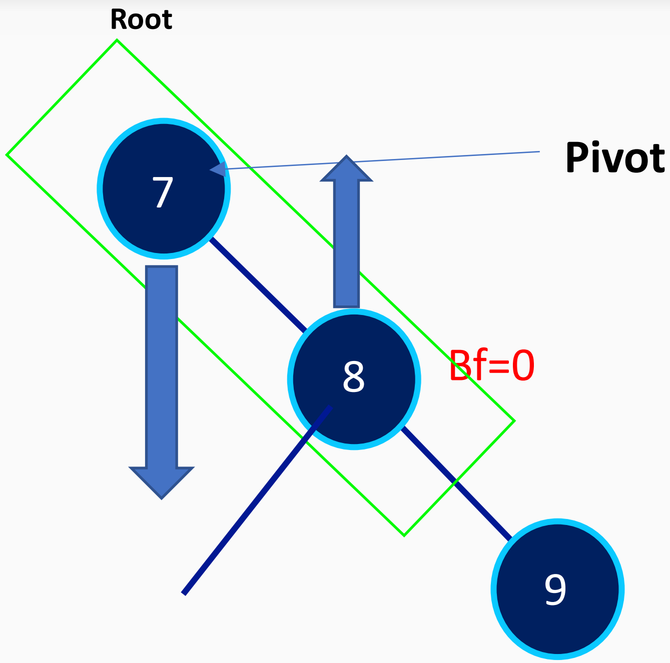


Insertion Order: 7, 8,9

Balancing Trees- Left Rotation



Left Rotation

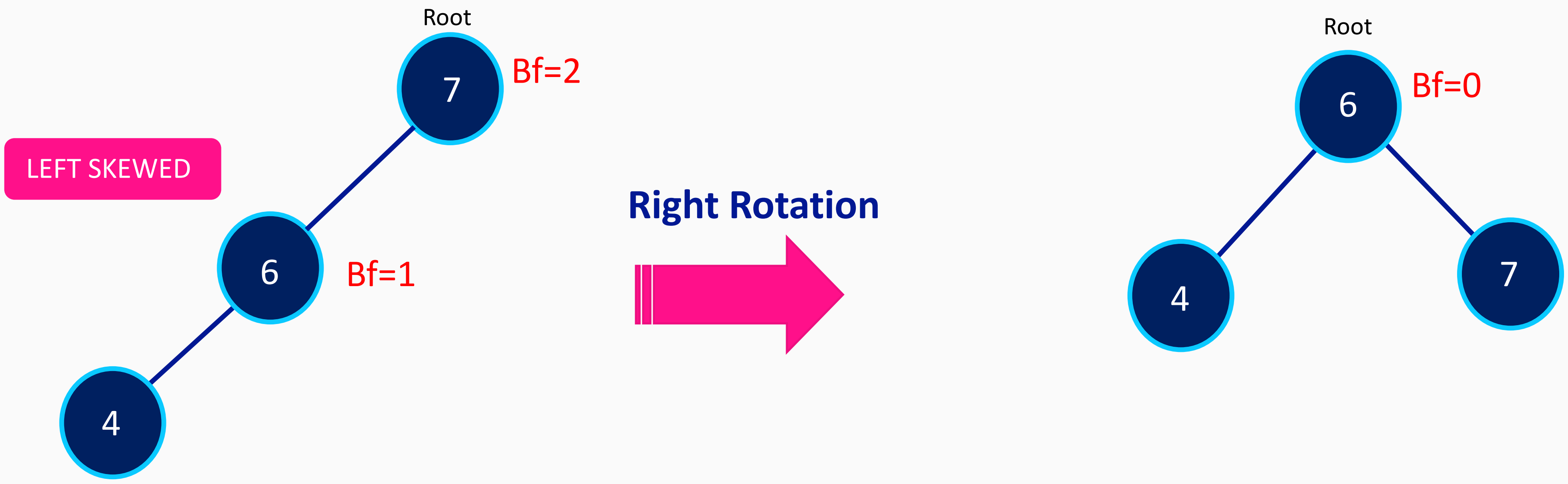


Insertion Order: 7, 8,9

Still BST ?

Only focus on three Nodes!!!!!
But rotate two!!!

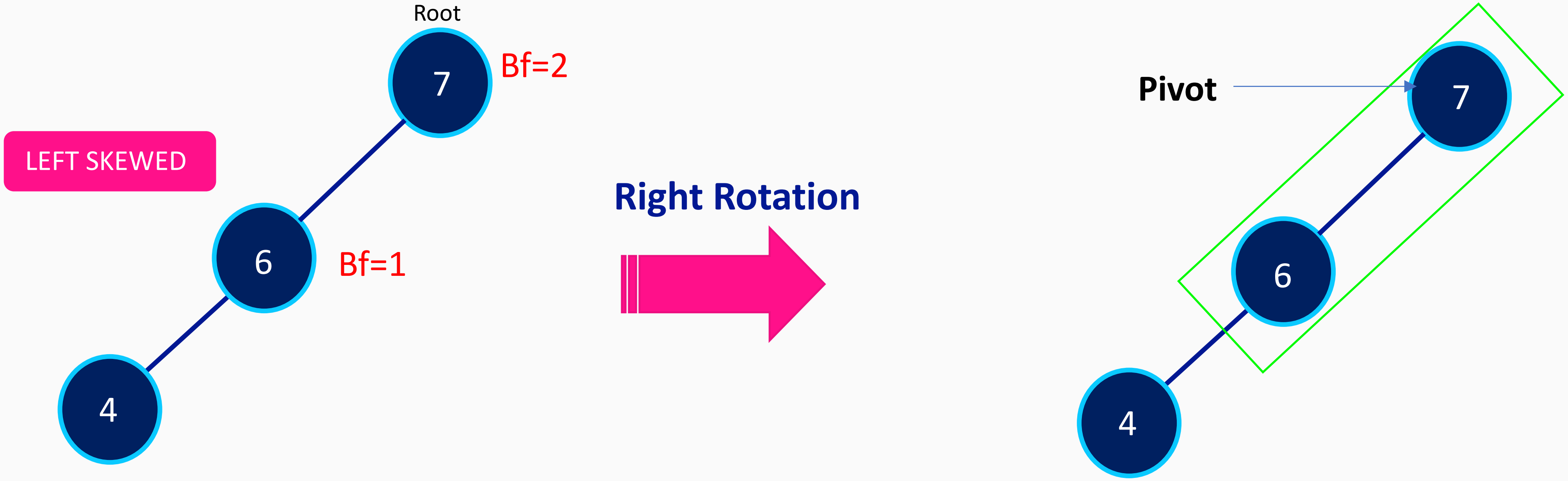
Balancing Trees-Right Rotation



Insertion Order: 7, 6,4

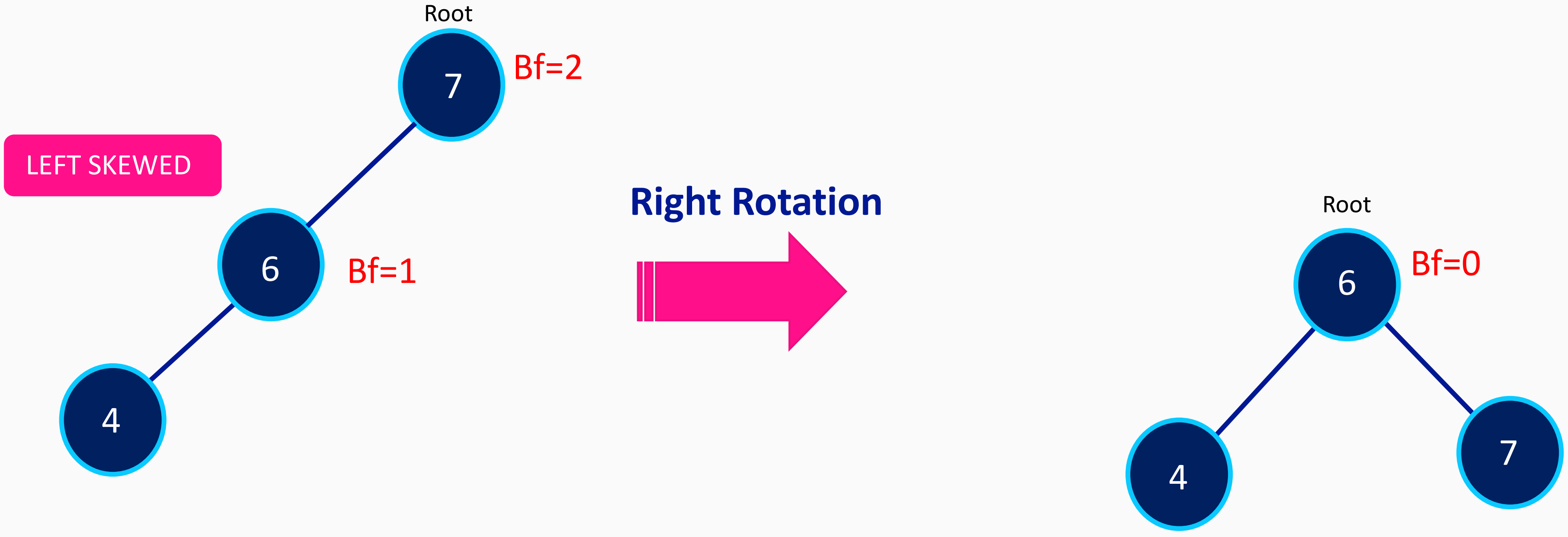
Still BST ?

Balancing Trees-Right Rotation



Insertion Order: 7, 6,4,1

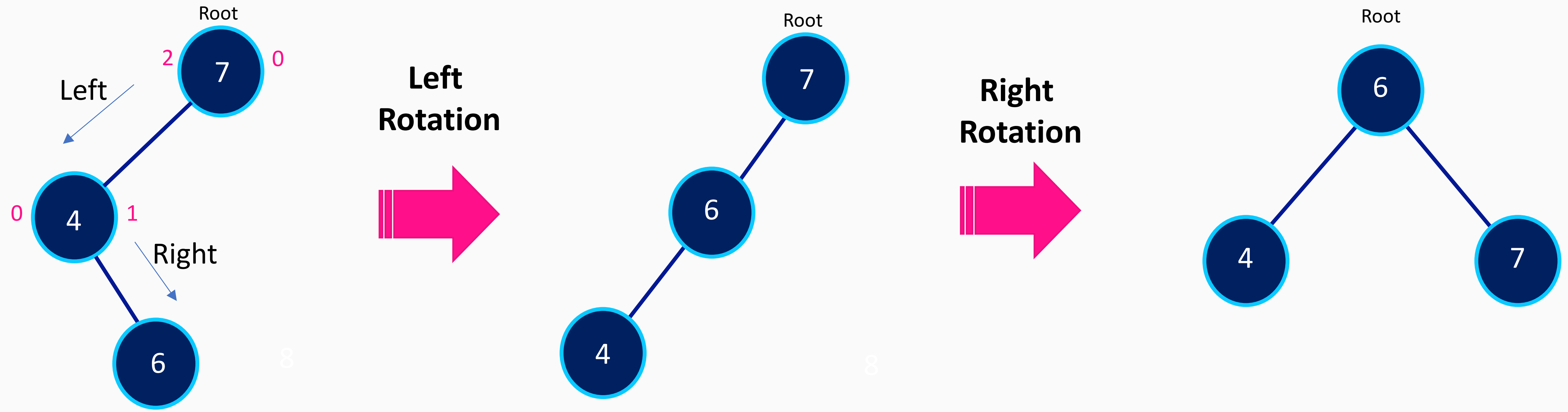
Balancing Trees-Right Rotation



Insertion Order: 7, 6,4

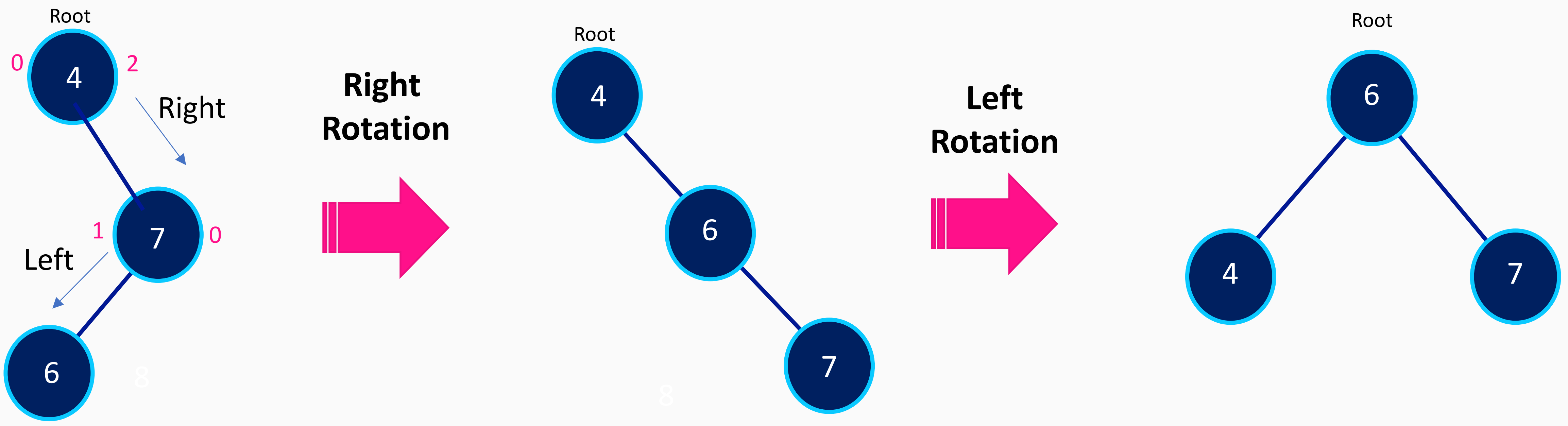
Balancing Trees- Left Right Rotation

LEFT – RIGHT ROTATION



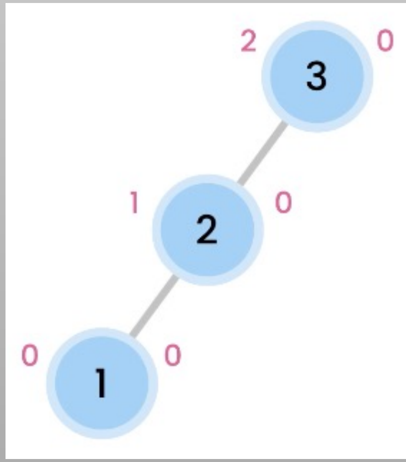
Balancing Trees- Left Right Rotation

RIGHT – LEFT ROTATION

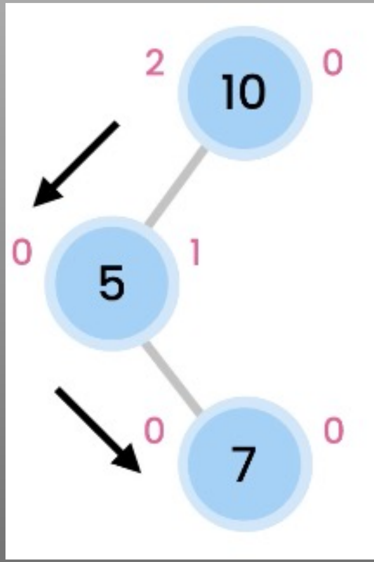


Summary of Rotations

LEFT HEAVY

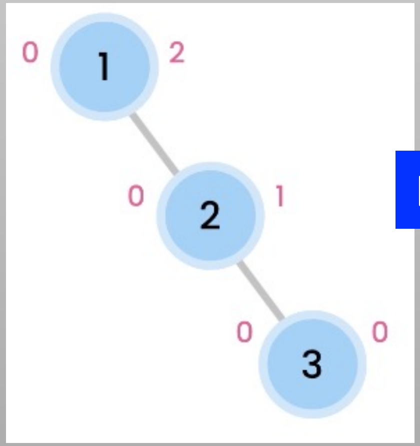


RIGHT ROTATION

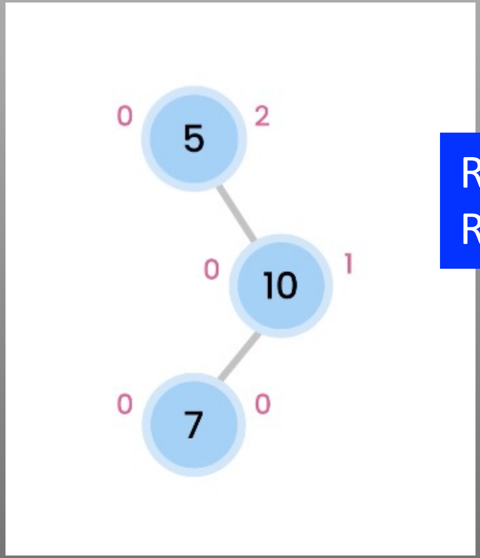


LEFT- RIGHT ROTATION

RIGHT HEAVY



LEFT ROTATION

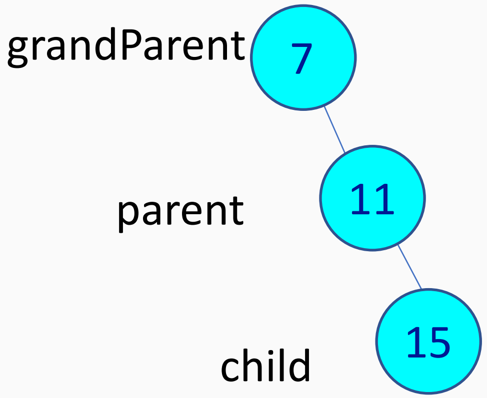


RIGHT - LEFT- ROTATION

Rotation Implementations-Left Rotation

Left Rotation

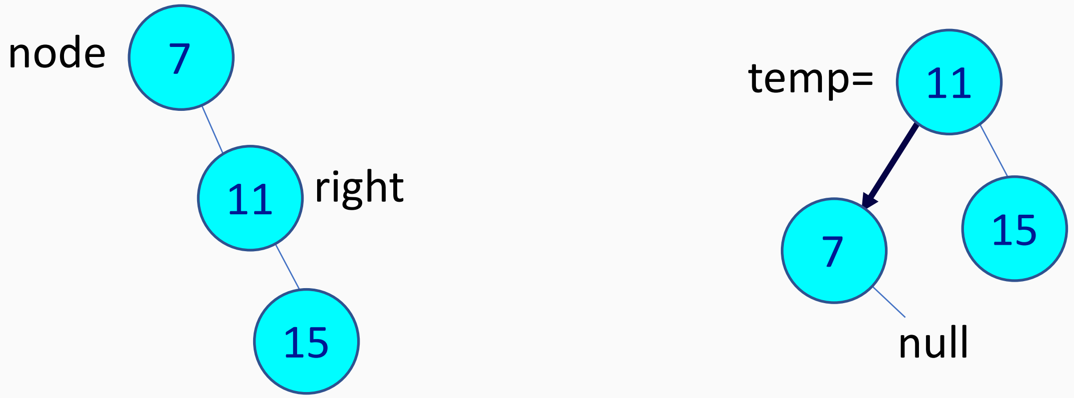
1. Set temp = grandparent’s right child
2. Set grandparent’s right child= temp left child
3. Set temp left child = grandparent
4. Use temp instead of grandparent



Rotation Implementations-Left Rotation

Left Rotation

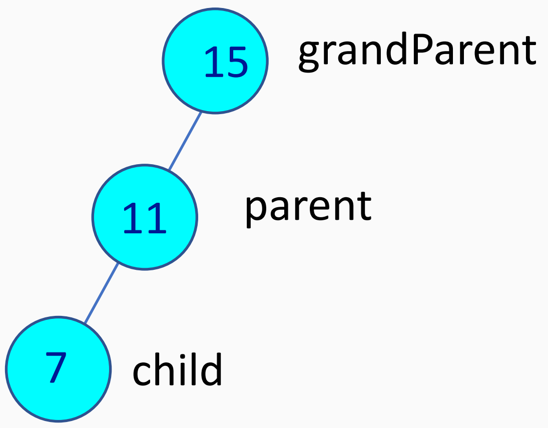
```
Public Node<E> leftRotate(Node<E> node){
Node<E> temp = node.right;
node.right= temp.left;
temp.left=node;
return temp;
}
```



Rotation Implementations-Right Rotation

Right Rotation

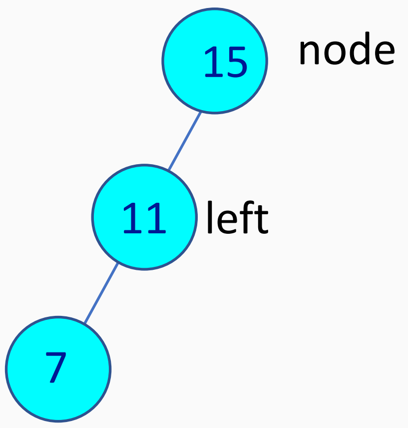
1. Set temp = grandparent's left child
2. Set grandparent's left child= temp right child
3. Set temp right child = grandparent
4. Use temp instead of grandparent

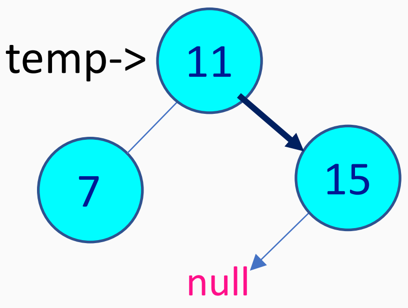


Rotation Implementations-Right Rotation

Right Rotation

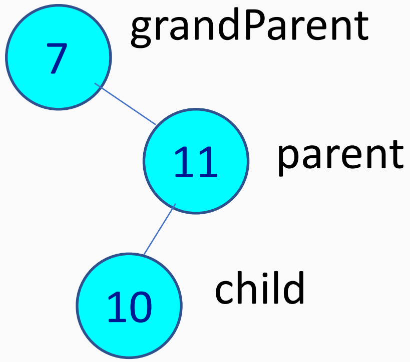
```
Public Node<E> rightRotate(Node<E> node){
Node<E> temp = node.left;
node.left= temp.right;
temp.right=node;
return temp;
}
```





Rotation Implementations

Right –Left Rotation

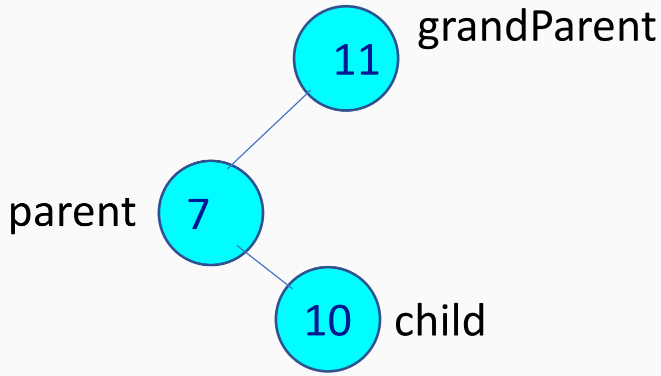



```
Public Node<E> rightLeftRotation (Node<E> node){
node.right=rightRotate(node.right);
Return leftRotate(node);
}
```

1. RightRotation on Parent
2. Return Leftrotation on GrandParent

Rotation Implementations

Left-Right Rotation

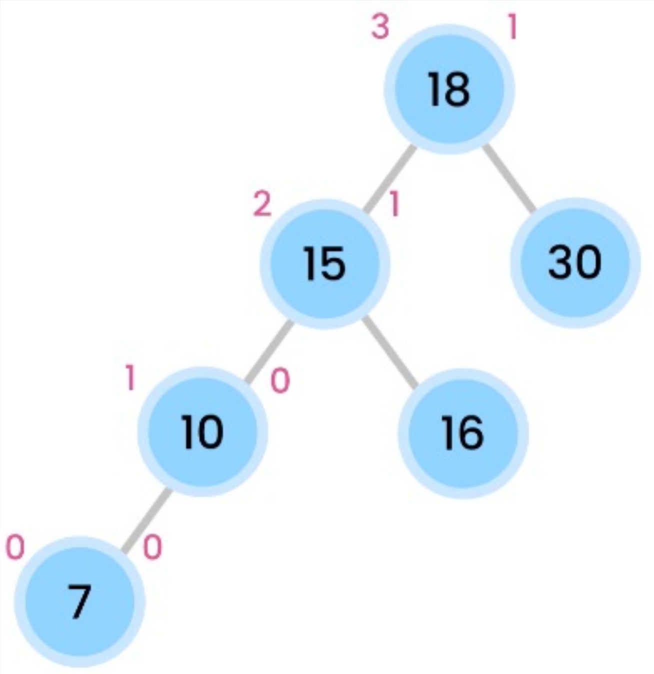


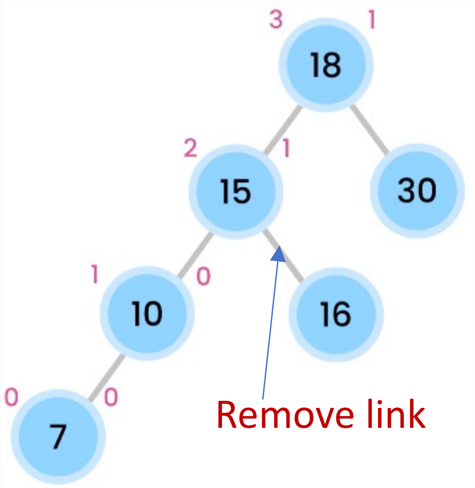

```
Public Node<E> leftRightRotation (Node<E> node){
node.left=leftRotate(node.left);
Return rightRotate(node);
}
```

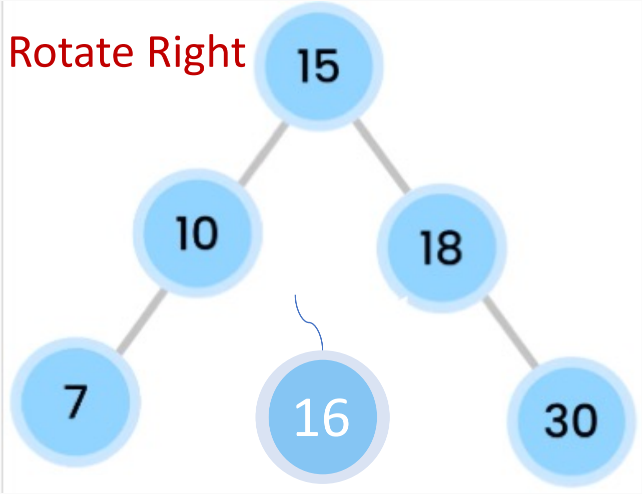
1. leftRotation on Parent
2. Return rightrotation on GrandParent

Balancing Trees

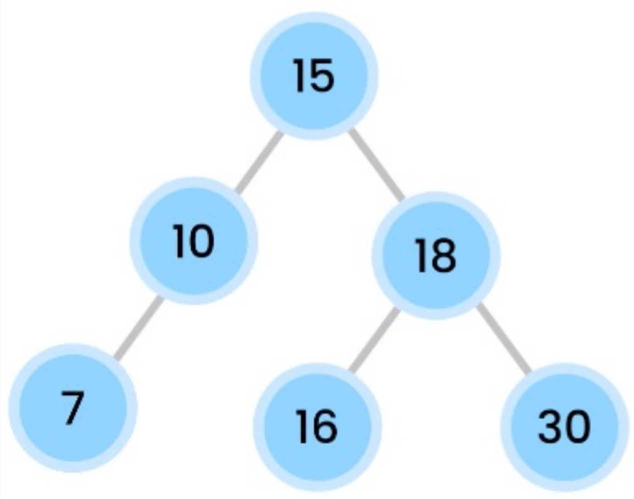
RIGHT- ROTATION







Link / add node ’16’ to left of 18



Right Rotation

```
Public Node<E> rightRotate(Node<E> node){
Node<E> temp = node.left;
node.left= temp.right;
temp.right=node;
return temp;
}
```

Tree Tasks

Task 1: Implement finding an integer value in a BST (Binary Search Tree).

```
boolean contains(int value){}
```

Task 2: Implement a method that returns true if the node is a leaf in a BST.

```
boolean isLeaf(Node node){}
```

Task 3: Implement a method that prints leaves of a BST.

```
void printLeaves(Node root)
```

Tree Tasks

Task 4: Implement a method that calculates height of a Node of a BST.

```
int height(Node root){}
```

Task 5: Implement a method that counts leaves of a BST.

```
int countLeaves(Node root){}
```

Task 6: Implement a method that returns sum of leaf values of a BST.

```
findSumOfLeaves(Node root){}
```

Questions?