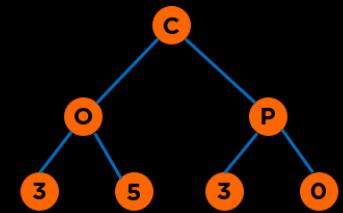


# Balanced Trees



# Categories of Data Structures

Linear Ordered

Lists

Stacks

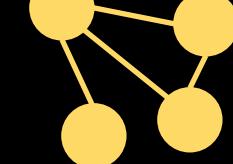
Queues



Non-linear Ordered

Trees

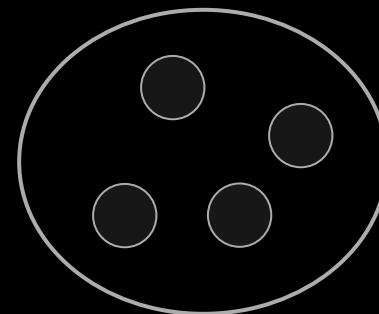
Graphs



Not Ordered

Sets

Tables/Maps



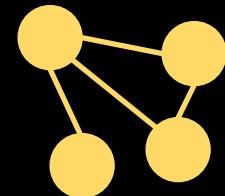
# Recap

Non-linear Ordered

Trees

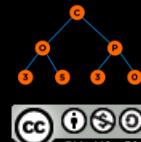
- **Binary Search Trees**

- **Operations**
- **Traversals**

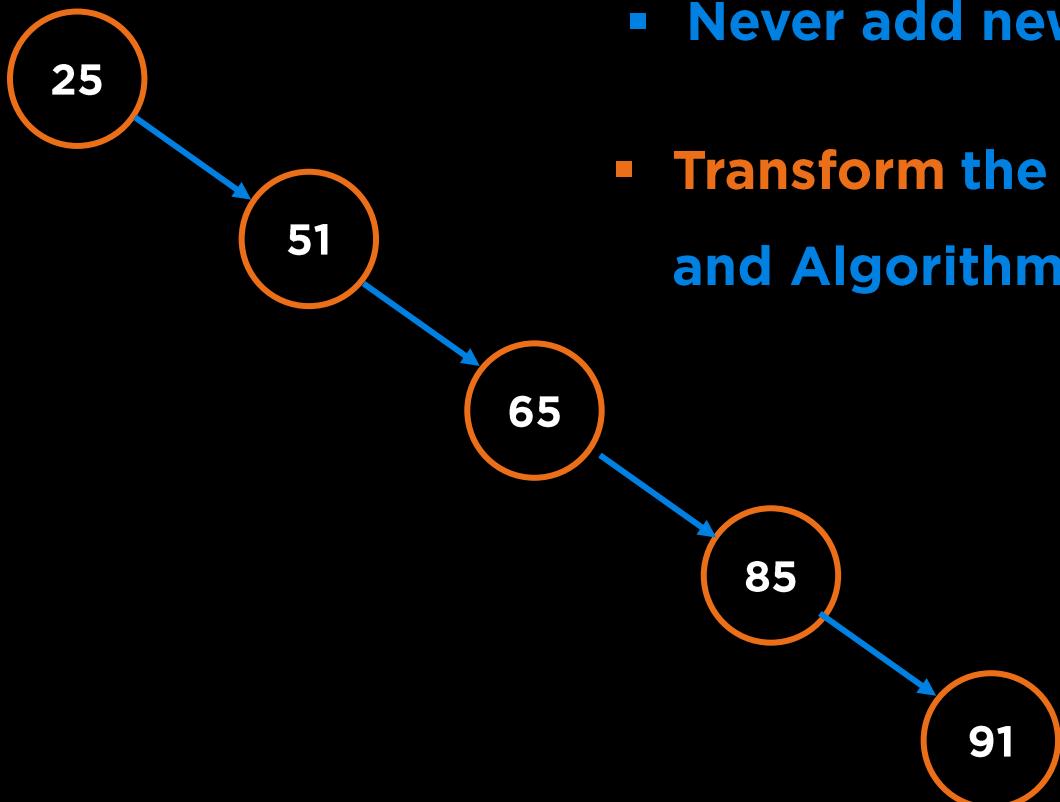


# Agenda

- **Trees**
  - **More Properties Related to Height**
- **Binary Search Tree Performance**
- **Rotations**
- **Balanced Trees: AVL Trees, Red Black Trees, Splay Trees, B/B+ Trees/Tries**
  - **Properties**
  - **Common operations e.g. Search/Insertion/Deletion**
  - **Performance**

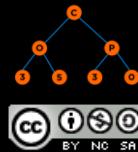


# How do we fix the Worst Case?



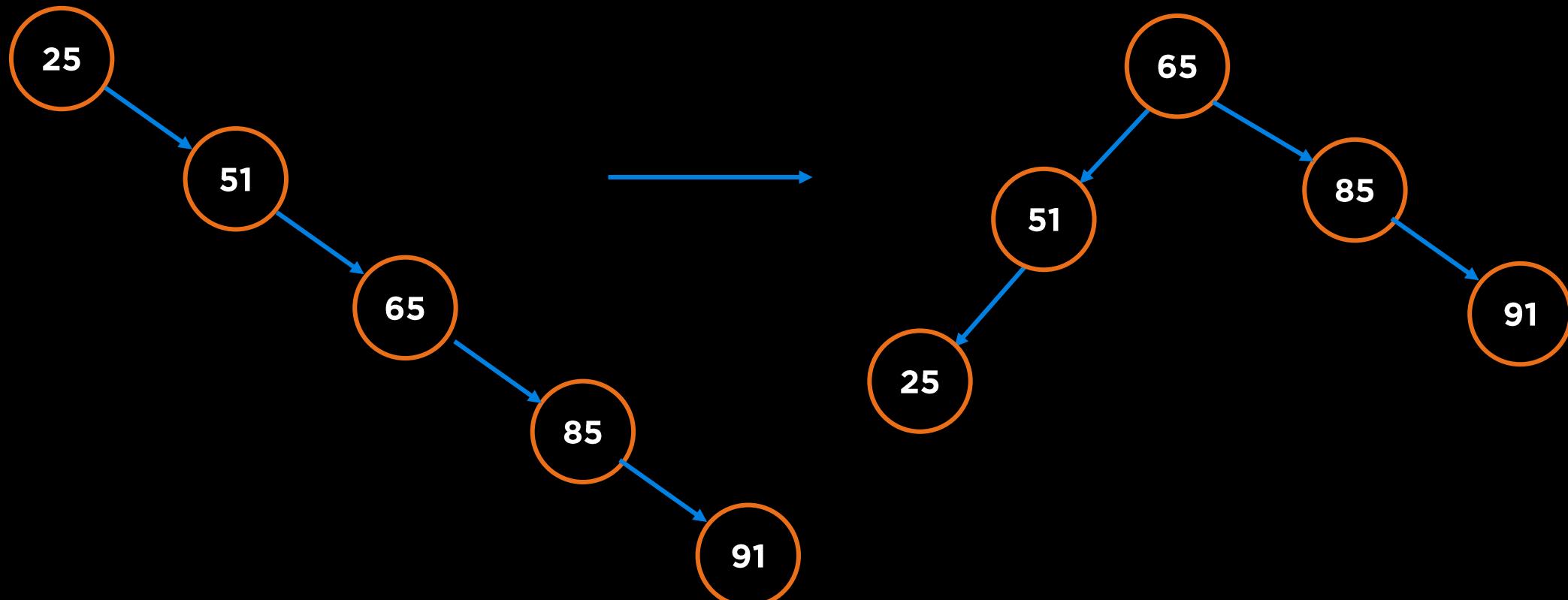
- Never add new leaves at the bottom: Increase size of node
- Transform the “Spindly” Tree to “Bushy Tree” using Tools and Algorithms

# Rotations



# Rotations

Tools to Rearrange the Tree Without affecting its Semantics

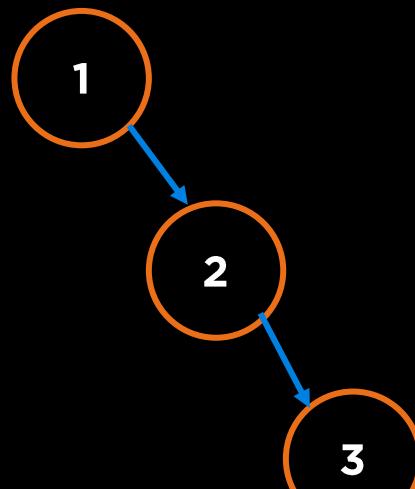


# BST Insertion: Inventing the Tool

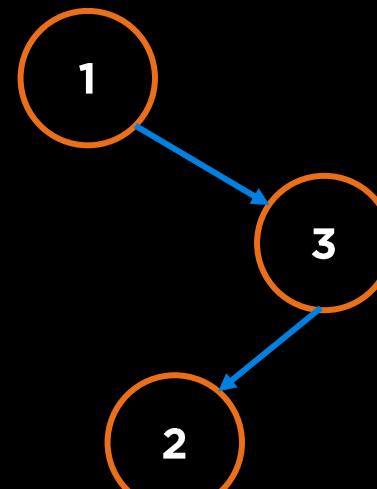
**n! different ways to insert n elements,**

Catalan (n) different BSTs

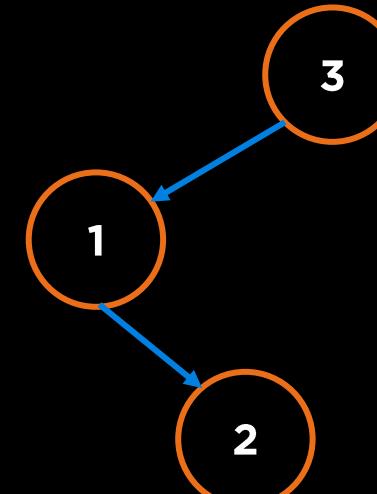
1, 2, 3



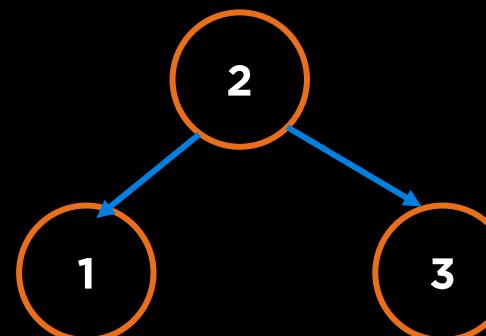
1, 3, 2



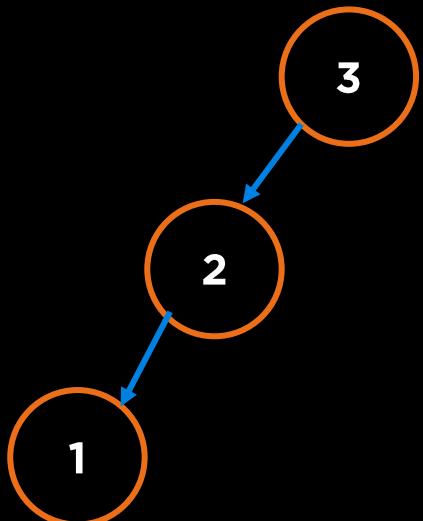
3, 1, 2



2, 3, 1 or 2, 1, 3

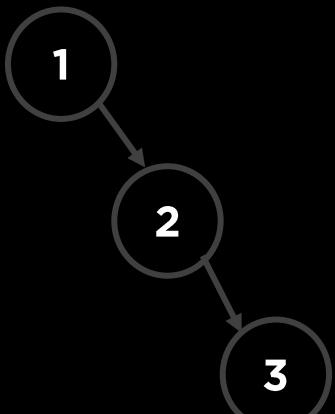


3, 2, 1

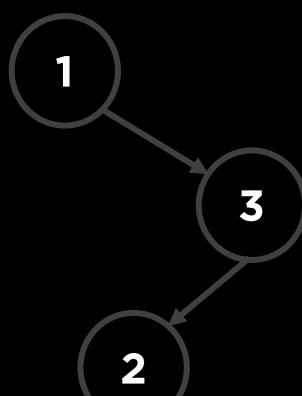


# Goal

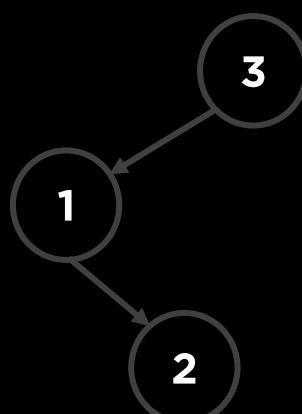
1, 2, 3



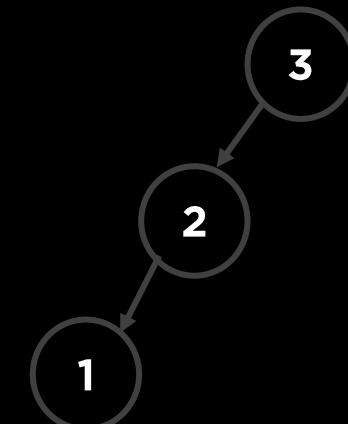
1, 3, 2



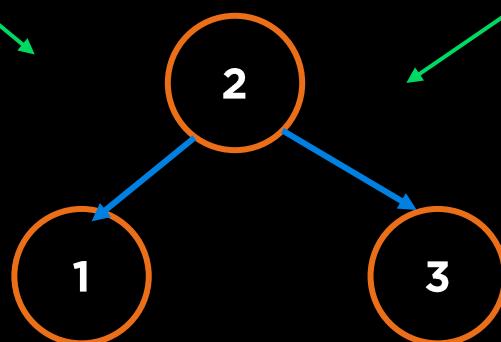
3, 1, 2



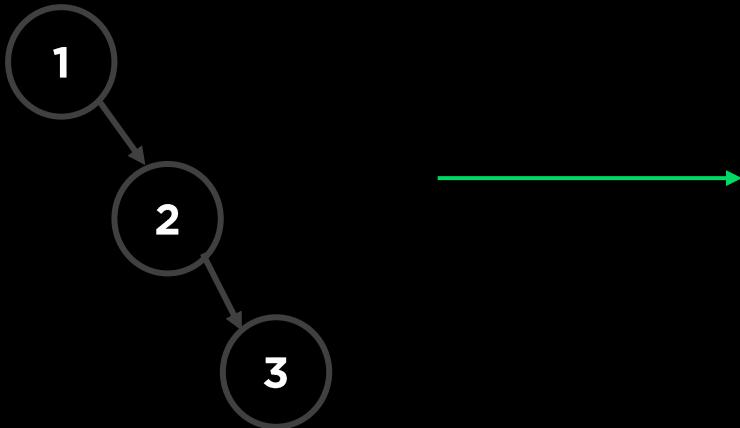
3, 2, 1



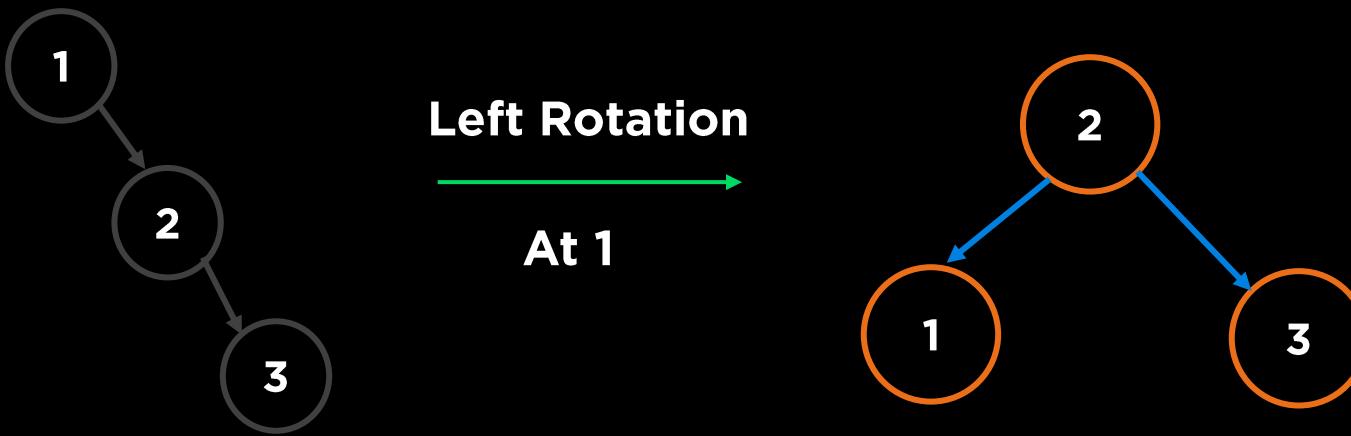
2, 3, 1 or 2, 1, 3



# Left Rotation: Right Right Case



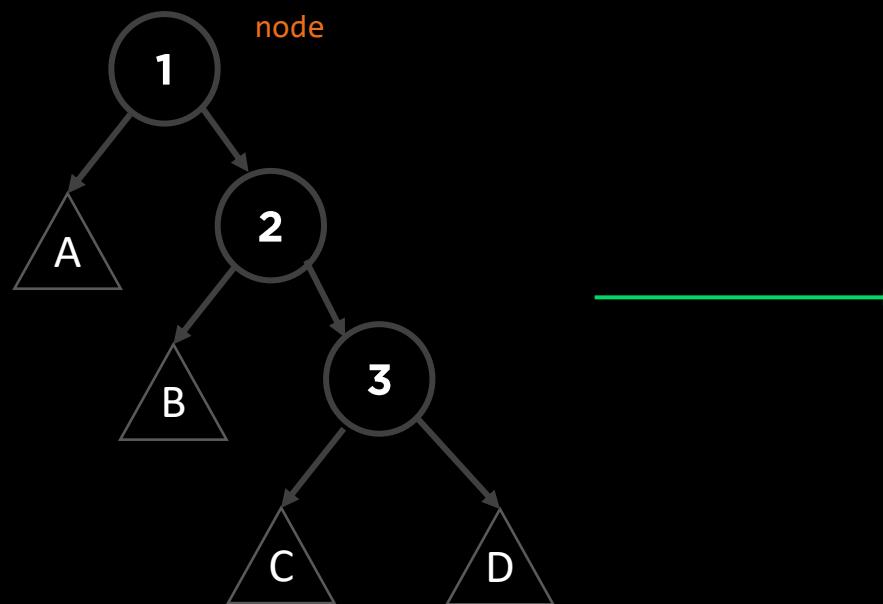
# Left Rotation: Right Right Case



Single Rotation

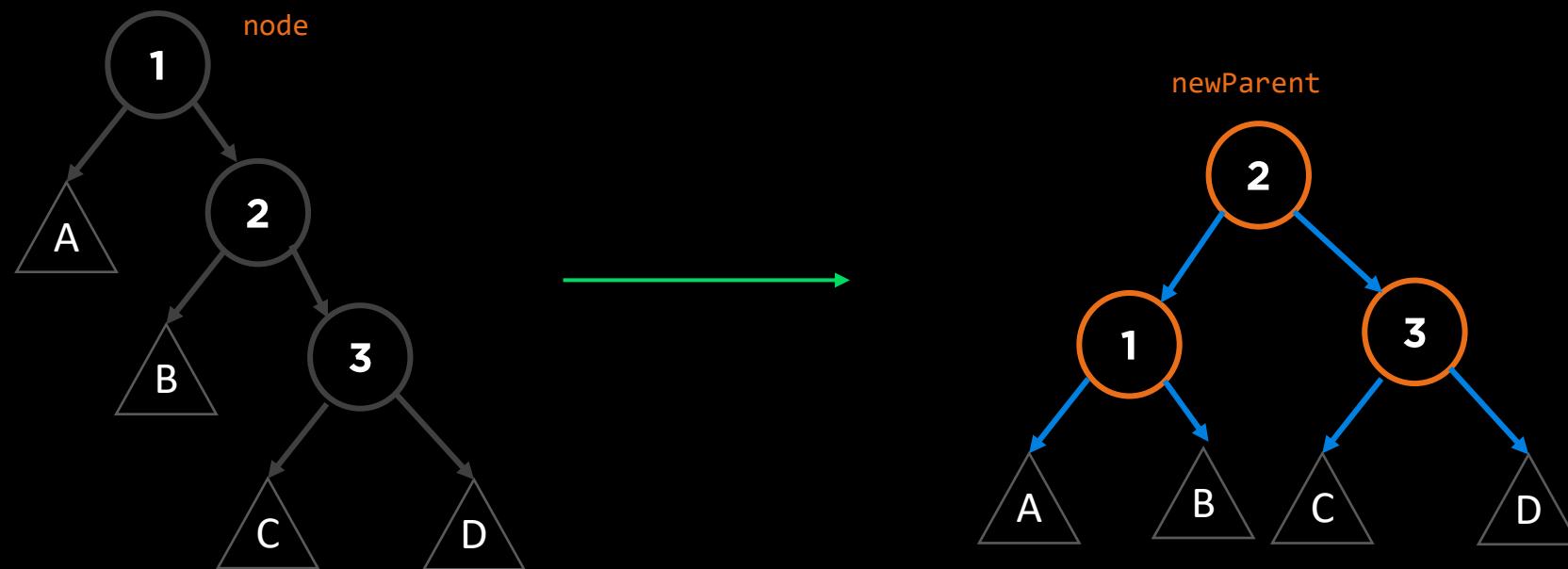
# Left Rotation: Right Right Case

```
rotateLeft (node)
{
    grandchild = node->right->left;
    newParent = node->right;
    newParent->left = node;
    node->right = grandchild;
    return newParent;
}
```



# Left Rotation: Right Right Case

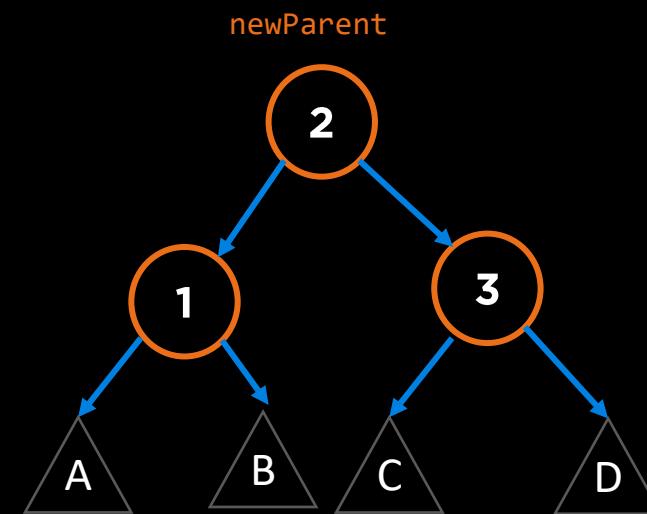
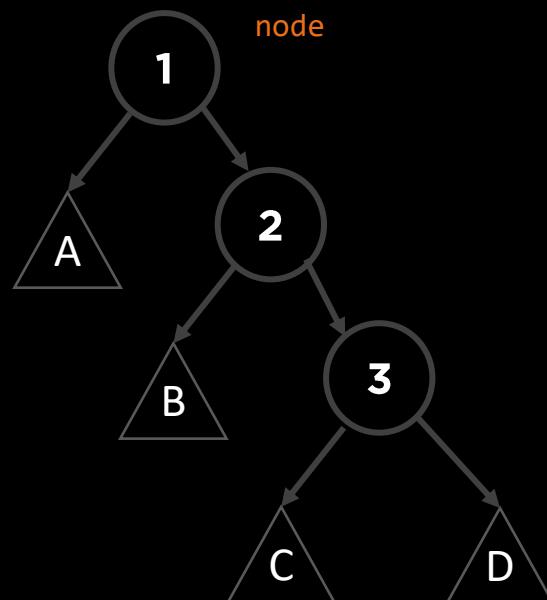
```
rotateLeft (node)
{
    grandchild = node->right->left;
    newParent = node->right;
    newParent->left = node;
    node->right = grandchild;
    return newParent;
}
```



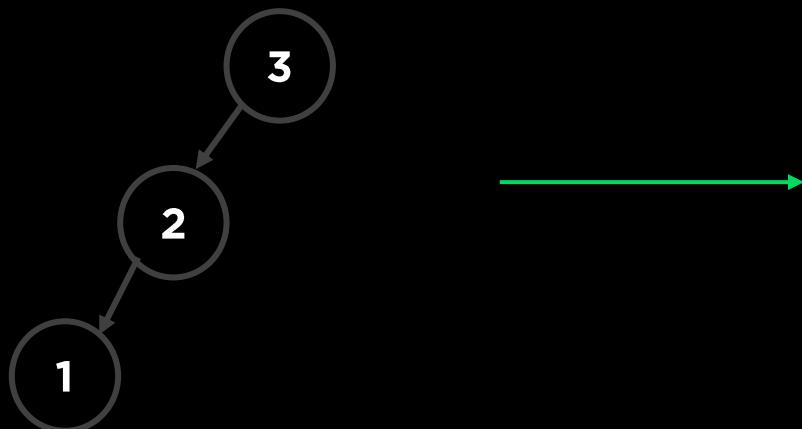
# Left Rotation: Right Right Case

```
rotateLeft (node)
{
    grandchild = node->right->left;
    newParent = node->right;
    newParent->left = node;
    node->right = grandchild;
    return newParent;
}
```

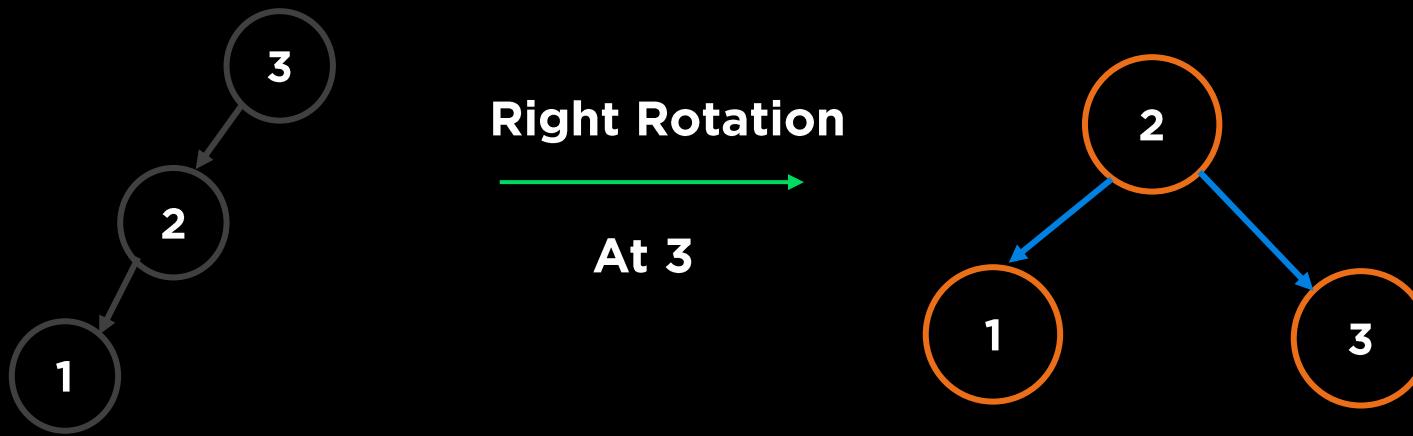
Take Constant Time, O(1)



# Right Rotation: Left Left Case

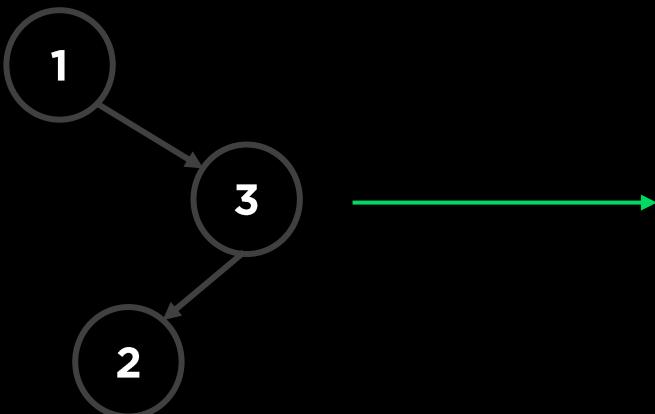


# Right Rotation: Left Left Case

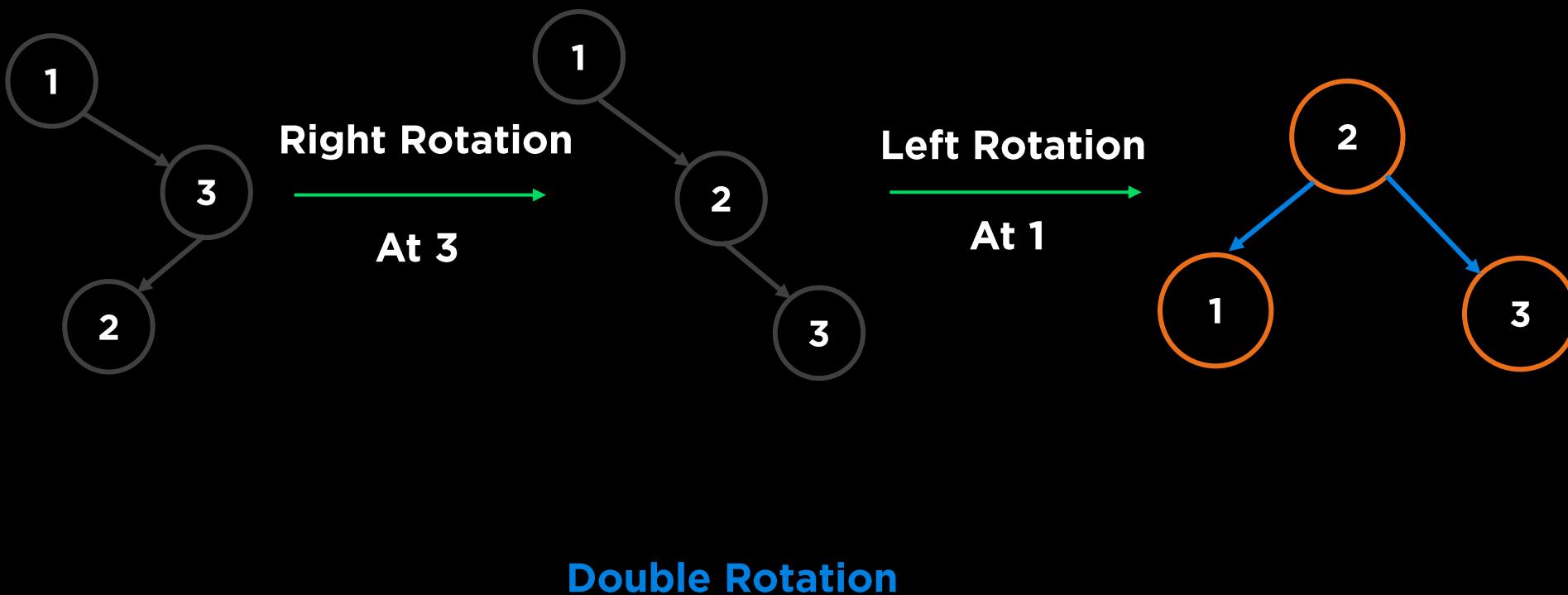


Single Rotation

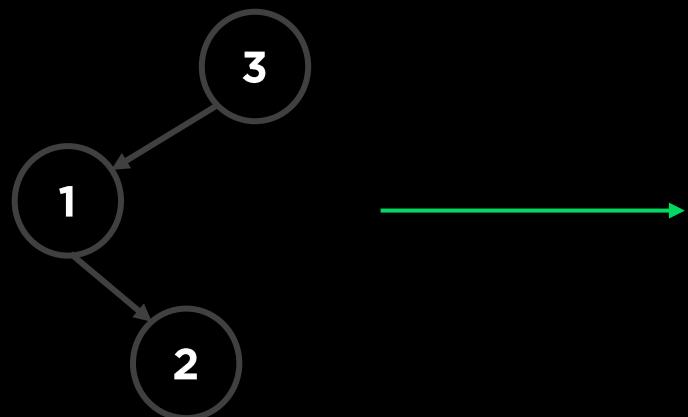
# Right Left Rotation: Right Left Case



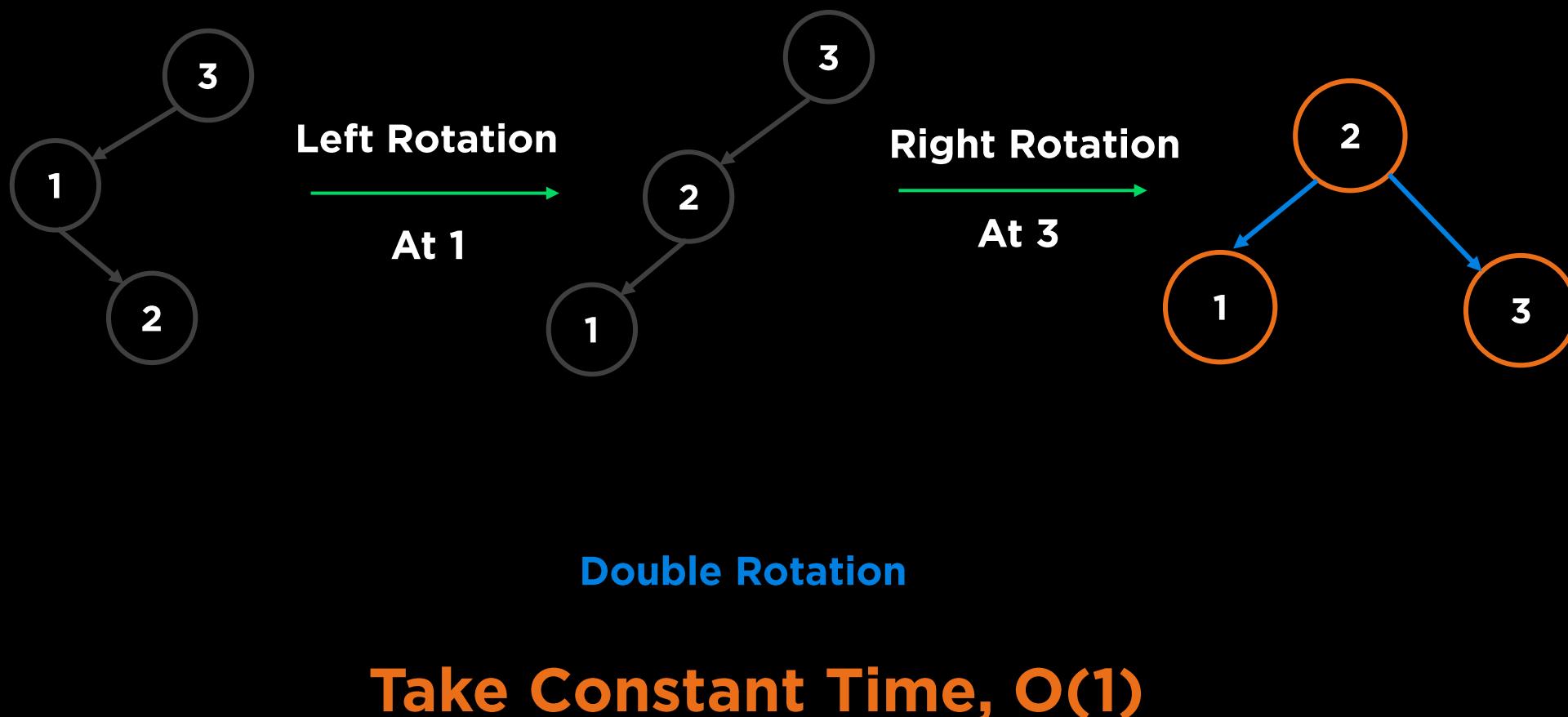
# Right Left Rotation: Right Left Case



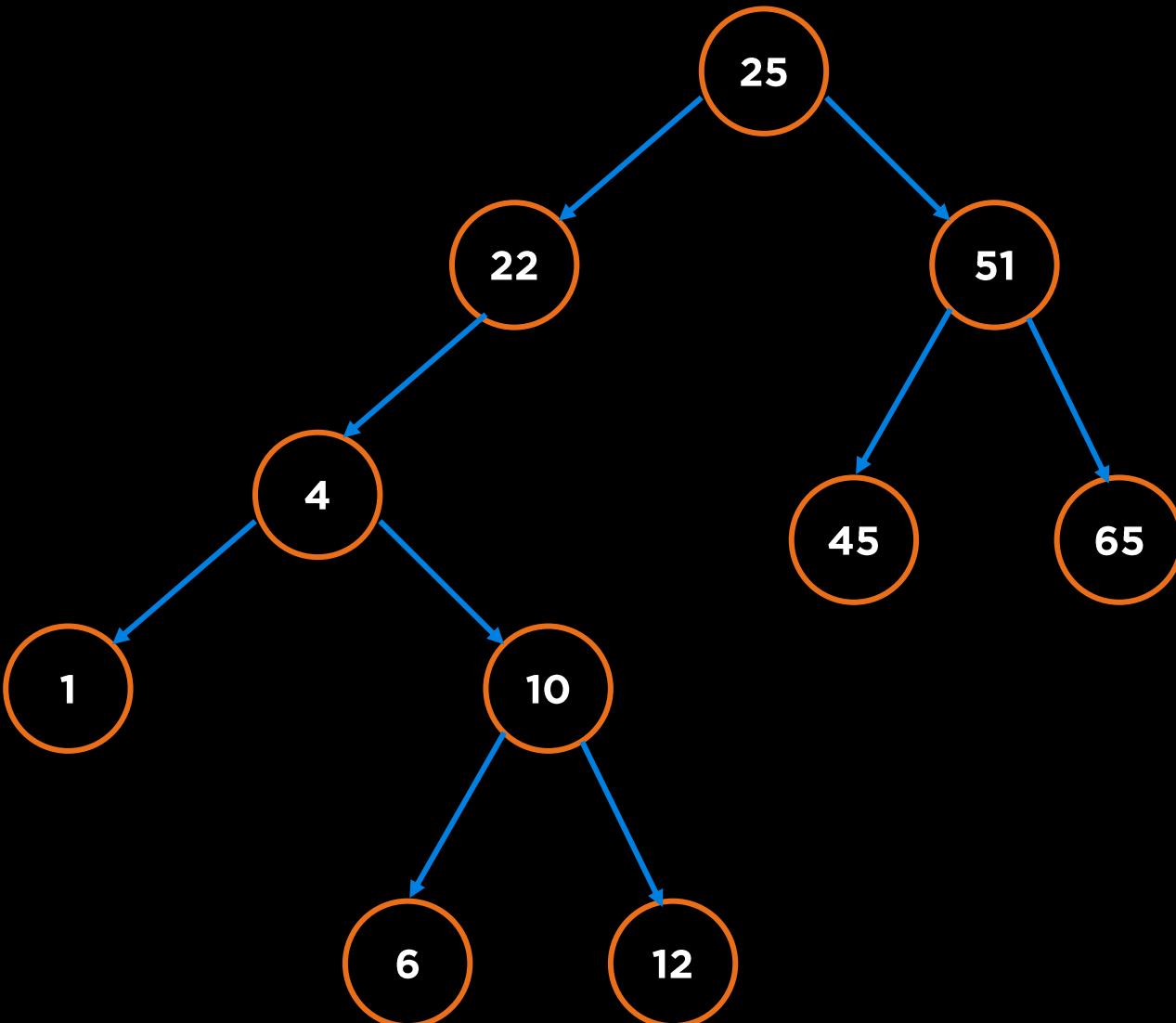
# Left Right Rotation: Left Right Case



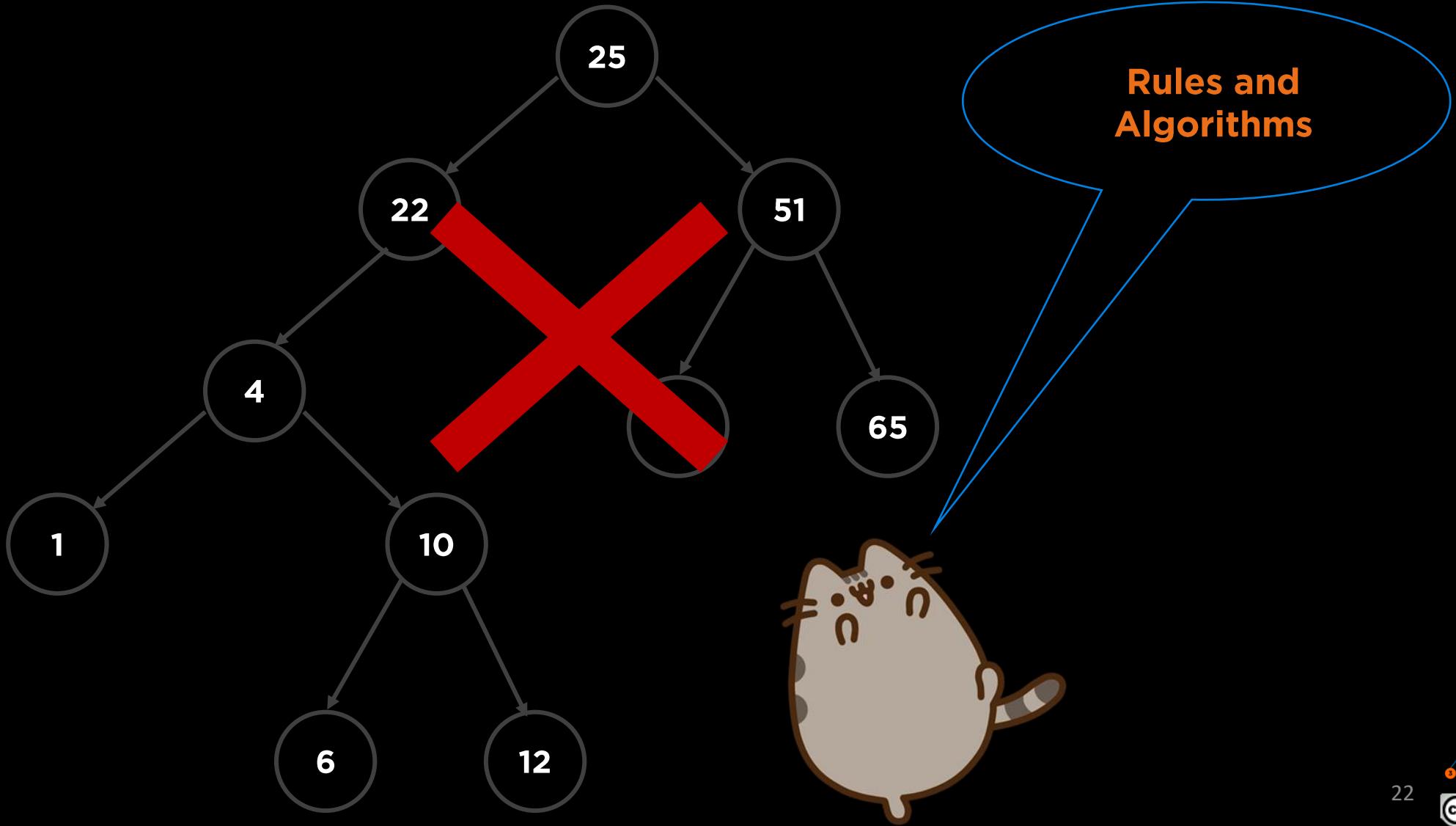
# Left Right Rotation: Left Right Case



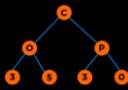
# Tool Issue: Can Get Messy on a Prebuilt Tree



# Fix for Messiness

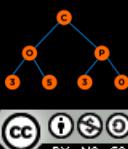


# AVL Trees



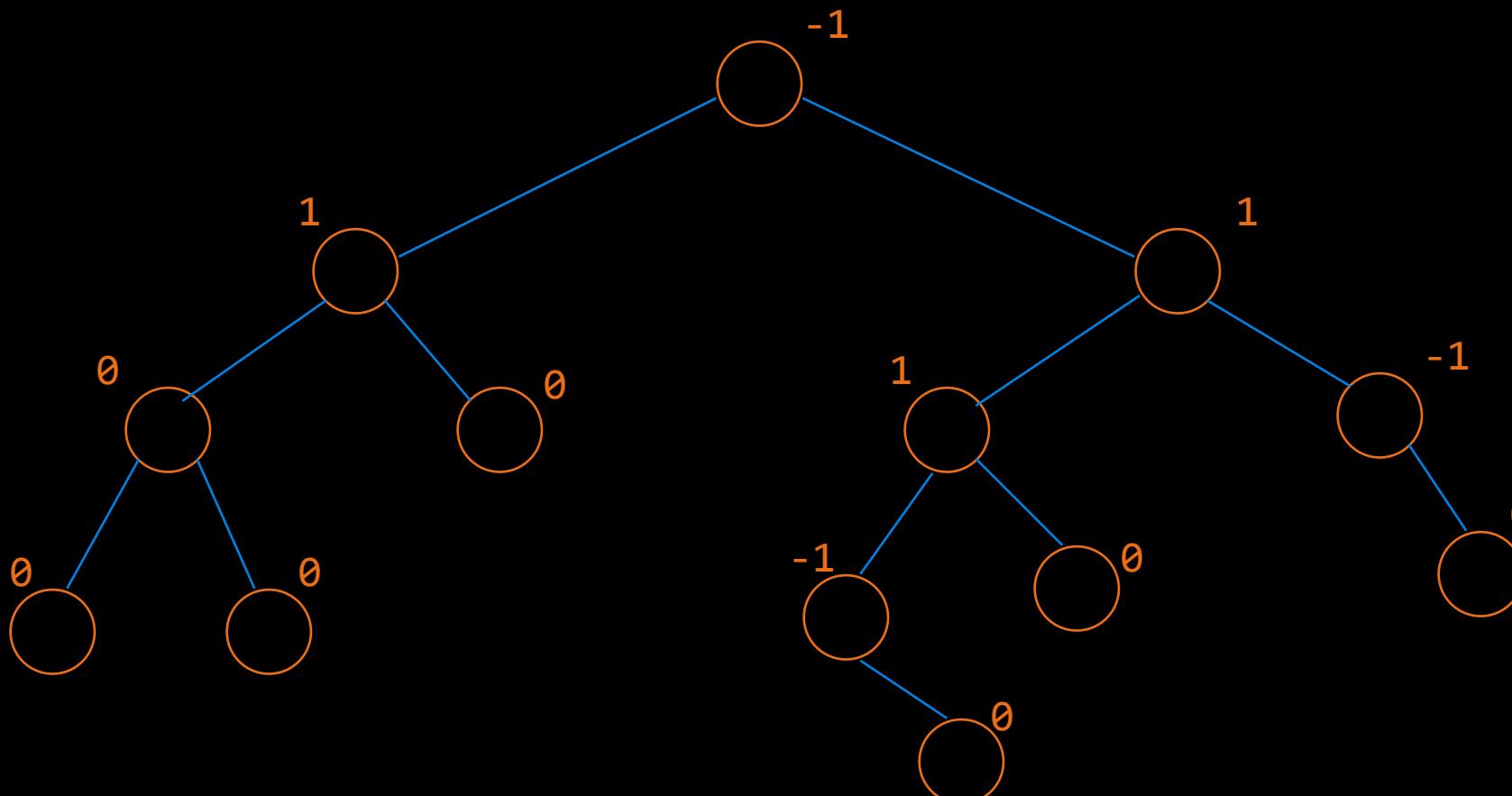
# Adelson-Velsky and Landis Trees

- Height Balanced Binary Search Trees
- Invariants:
  - Maintains BST invariants
    - Every node has 0, 1, or 2 children
    - Every element on the left is smaller and every element on the right is greater than a node.
  - For every node  $x$ , Balance Factor = 0, -1 or 1



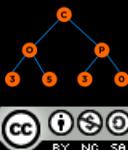
# AVL Tree

Balance Factor of  $x = \text{Height}(\text{left subtree of } x) - \text{Height}(\text{right subtree of } x)$



# AVL Tree : C++ Node Class

```
01 class TreeNode {  
02     public:  
03         int val;  
04         int height; // Or Balance Factor  
05         TreeNode *left;  
06         TreeNode *right;  
07         TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}  
08     };
```



# AVL Trees: Insertion/Deletion

- Same as Binary Search Trees
- Identify deepest node that breaks the Balance Factor rule;  
Start rotating and move further up the search path
- After Insertion/Deletion height of all nodes in Search Path  
may change

# AVL Tree : C++ Insert

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);

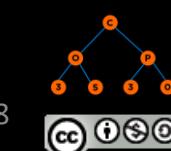
    else if (key < root->val)
        root->left = insert(root->left, key);

    else
        root->right = insert(root->right, key);

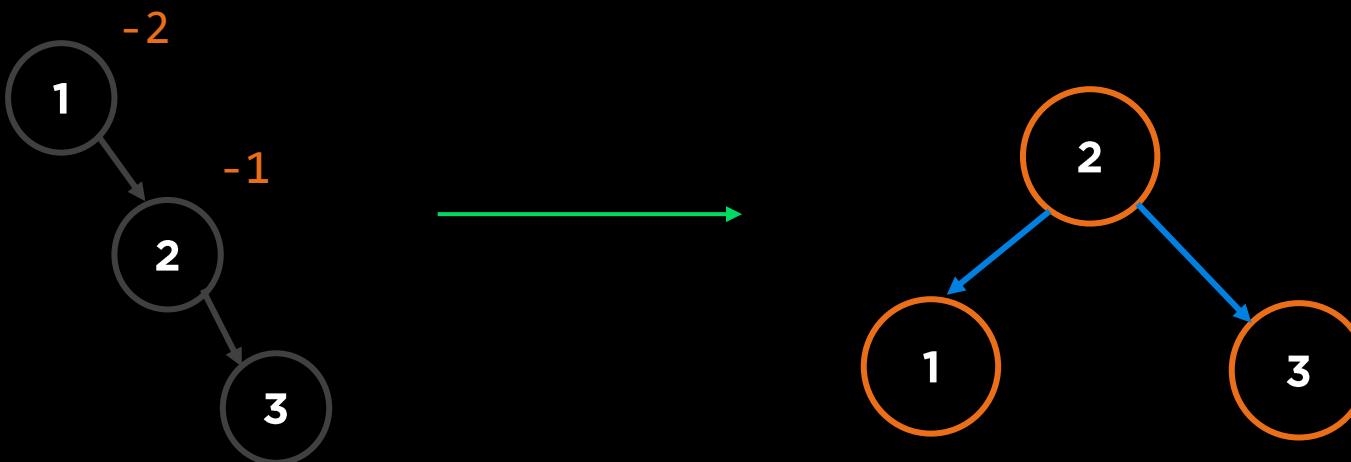
    return root;
}
```

## UPDATE HEIGHTS

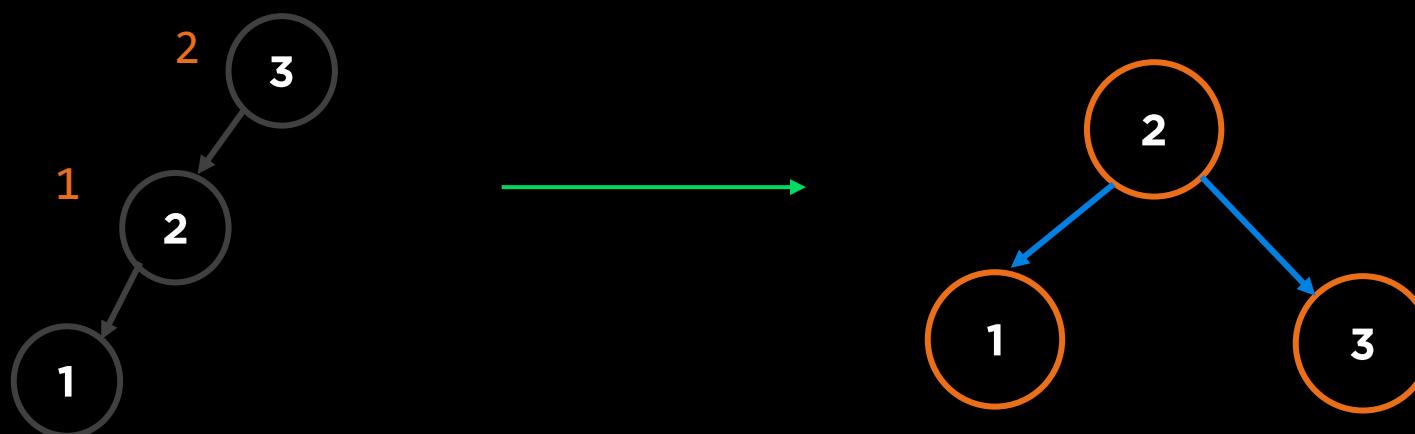
```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update height
    ELSE
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update height
    ELSE
        Perform Right rotation & update height
}
```



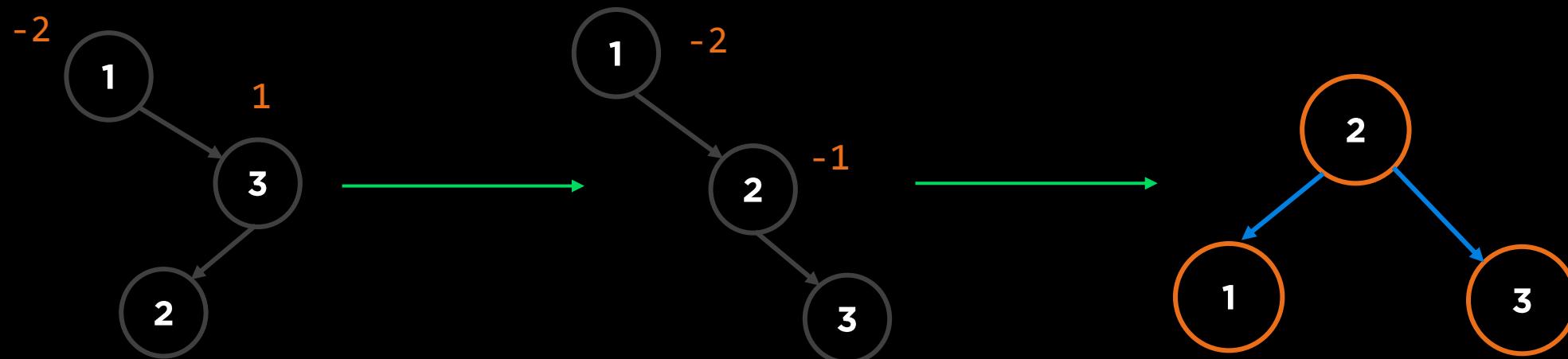
# Left Rotation: Right Right Case



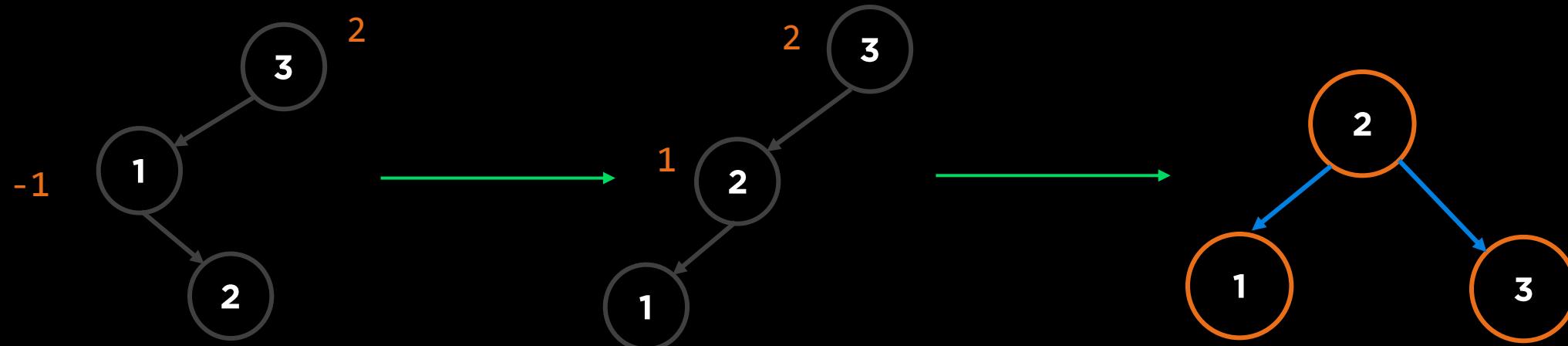
# Right Rotation: Left Left Case



# Right Left Rotation: Right Left Case



# Left Right Rotation: Left Right Case



# AVL Tree Rotations

Balance Factor of  $x$  = Height (left subtree of  $x$ ) - Height (right subtree of  $x$ )

Case (Alignment)	Balance Factor		Rotation
	Parent	Child	
Left Left	+2	+1	Right
Right Right	-2	-1	Left
Left Right	+2	-1	Left Right
Right Left	-2	+1	Right Left

If Balance Factor of  $x$  = Height (right subtree of  $x$ ) - Height (left subtree of  $x$ ),  
Reverse all signs in the table!

# AVL Tree : C++ Insert

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);

    else if (key < root->val)
        root->left = insert(root->left, key);

    else
        root->right = insert(root->right, key);
    // hint: find height
    return root;
}
```

## UPDATE HEIGHTS

```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update height
    ELSE
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update height
    ELSE
        Perform Right rotation & update height
}
```

# AVL Tree : Insert

## Insert 25

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

25

Insert 25

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

25

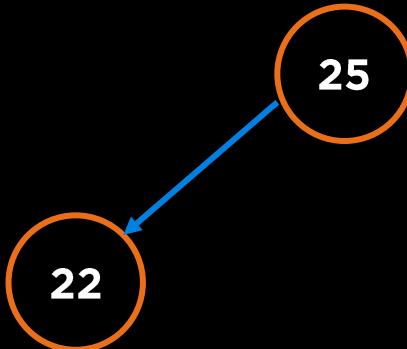
Insert 22

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

Insert 22

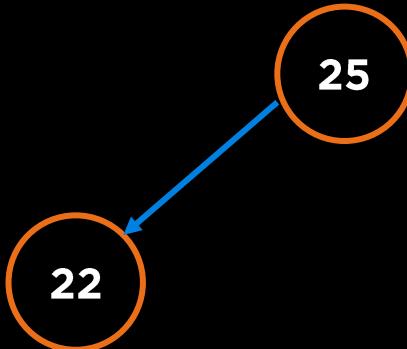


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

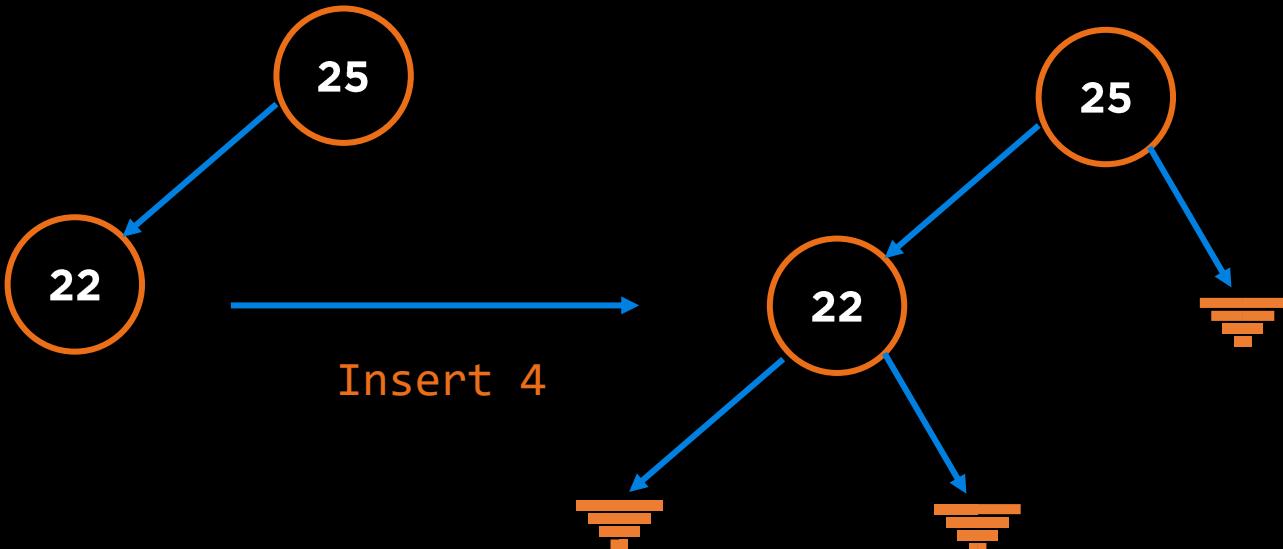
Insert 4



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

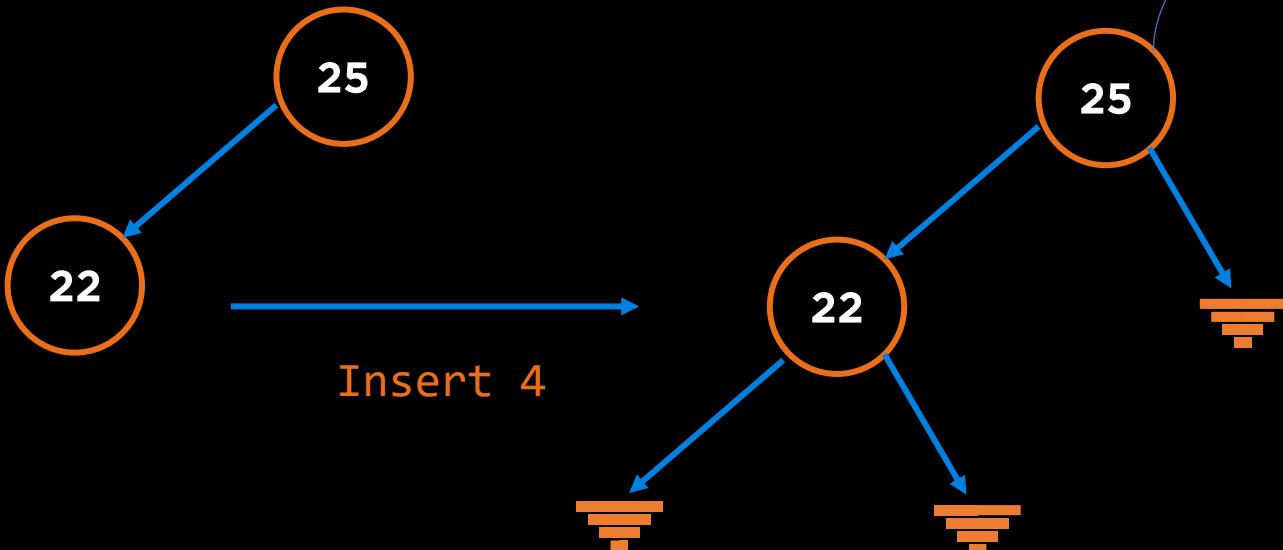
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

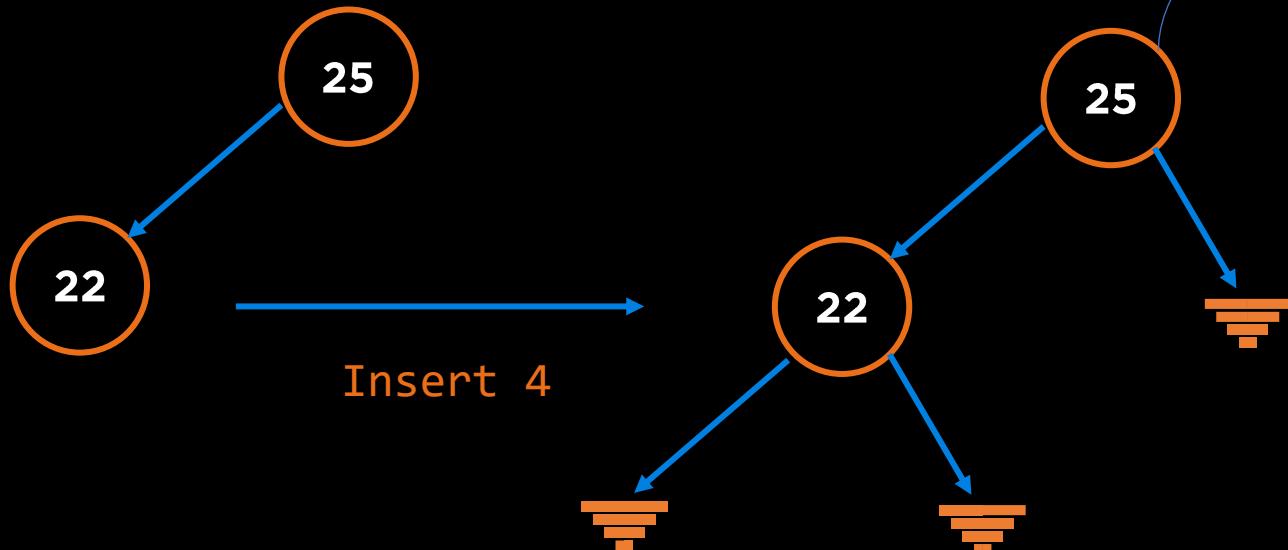
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

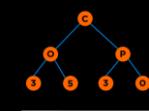
    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

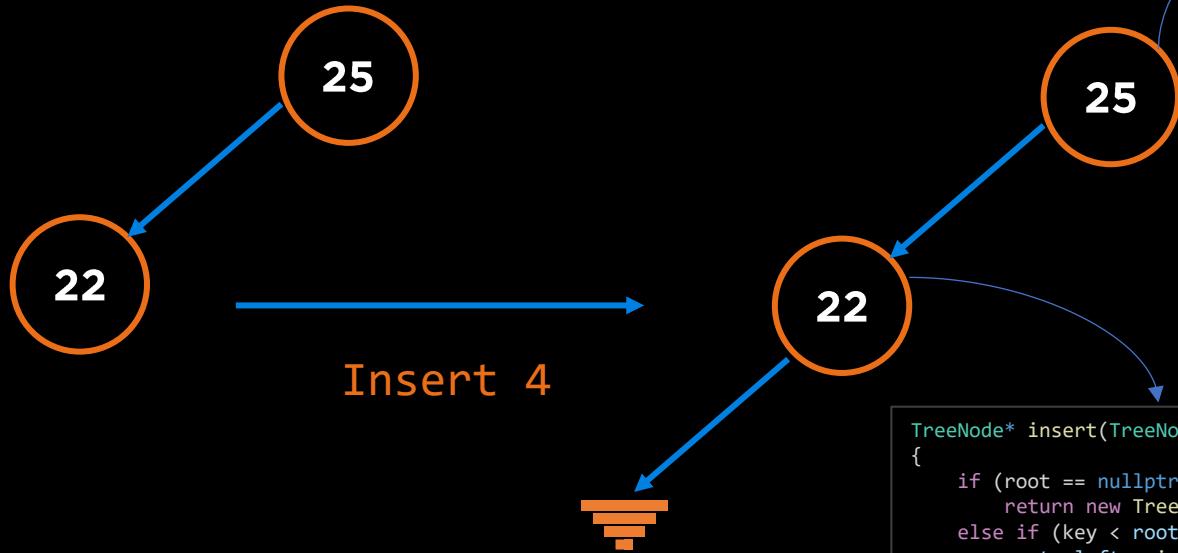


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```



# AVL Tree : Insert



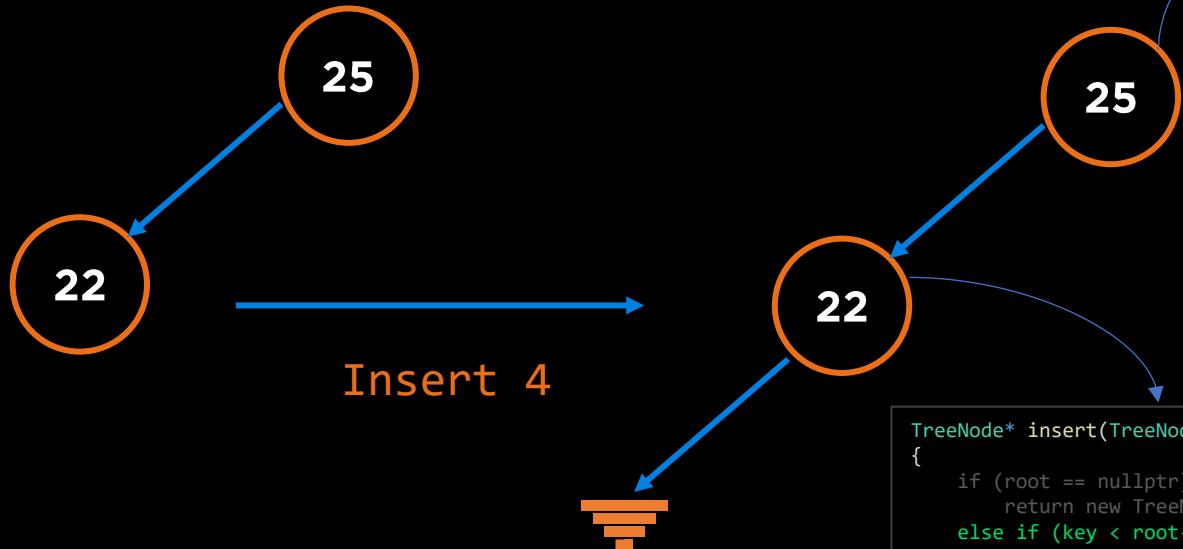
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



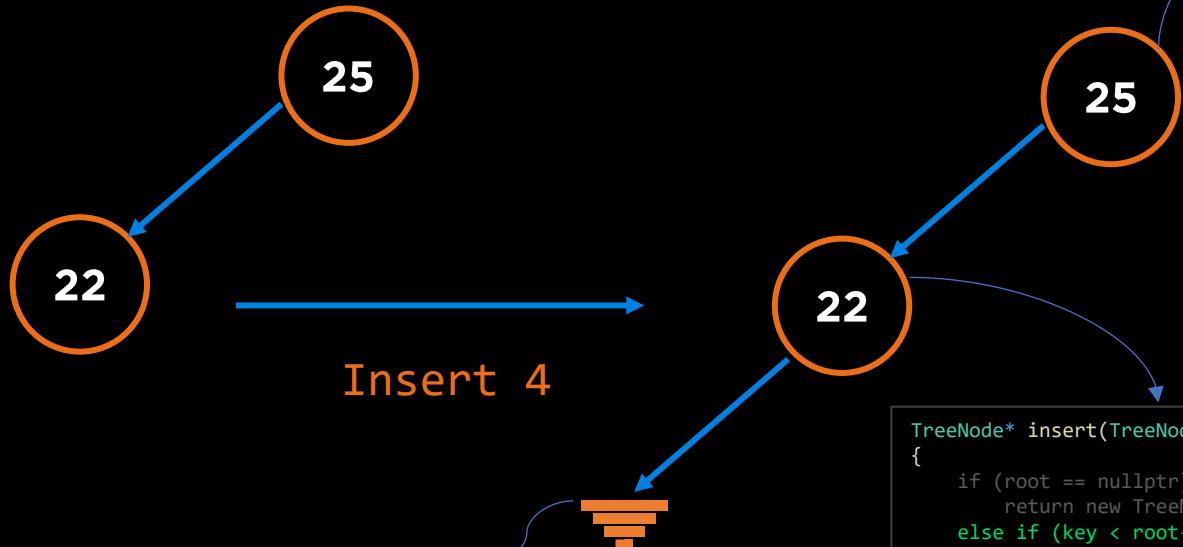
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

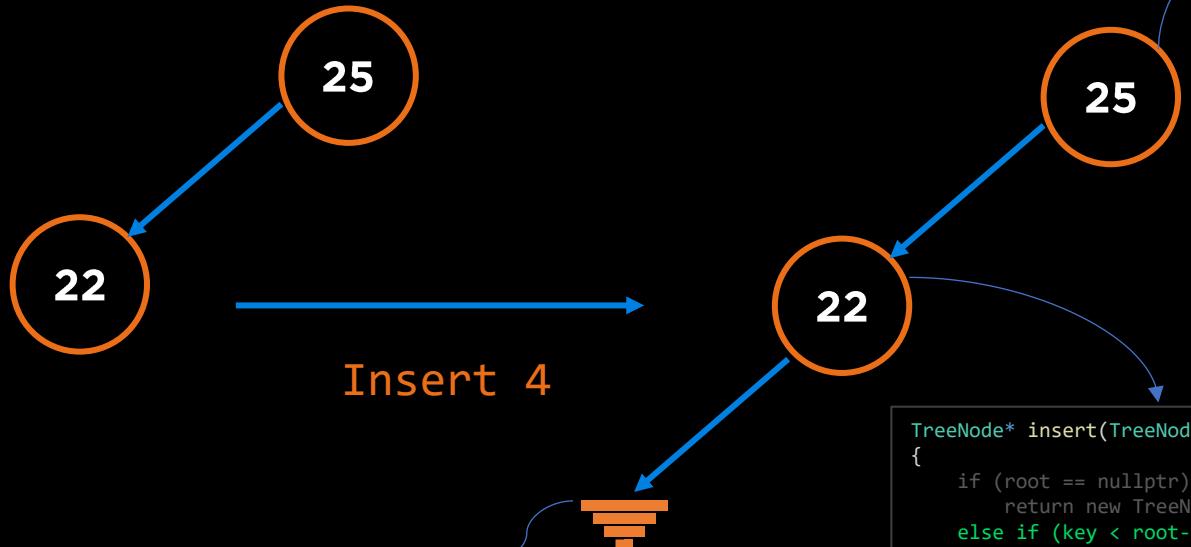
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

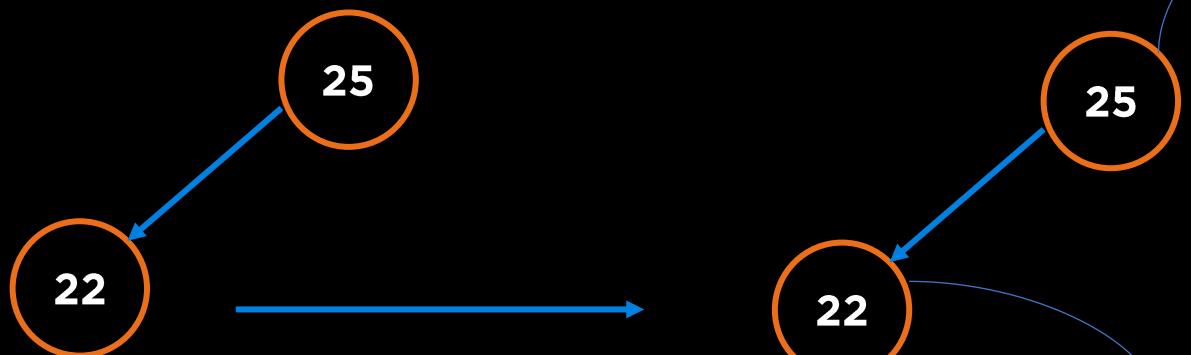
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



Insert 4



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

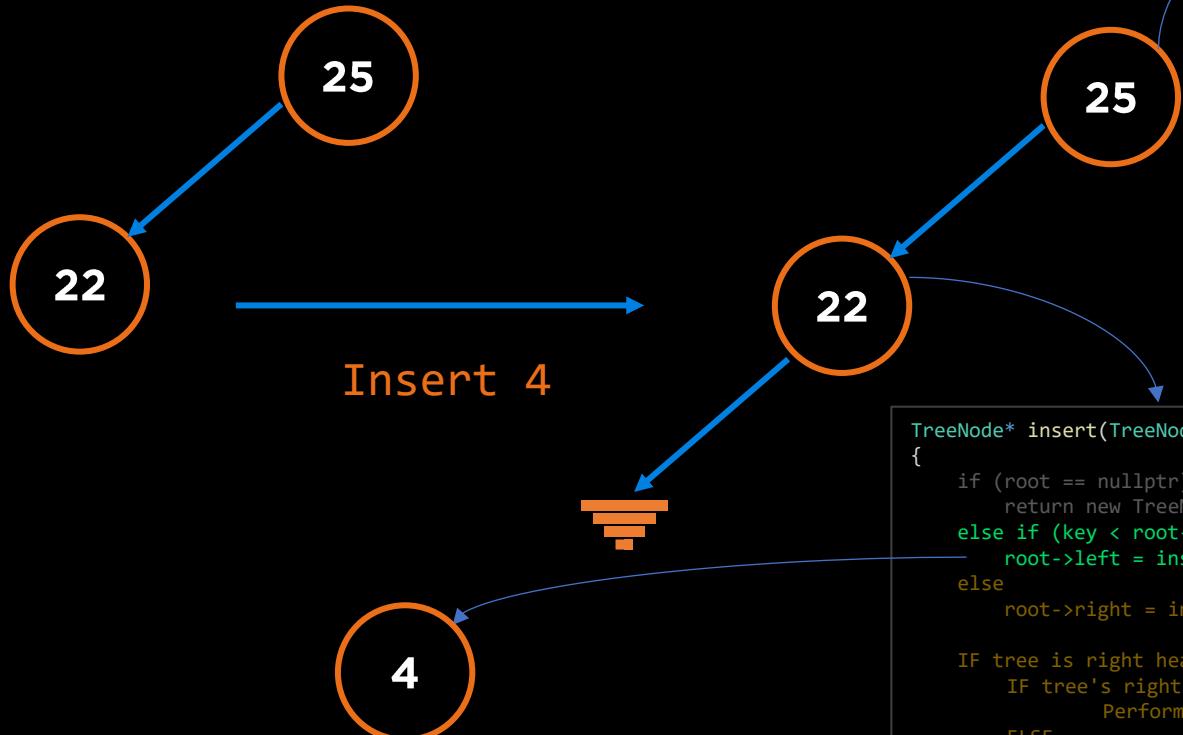
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

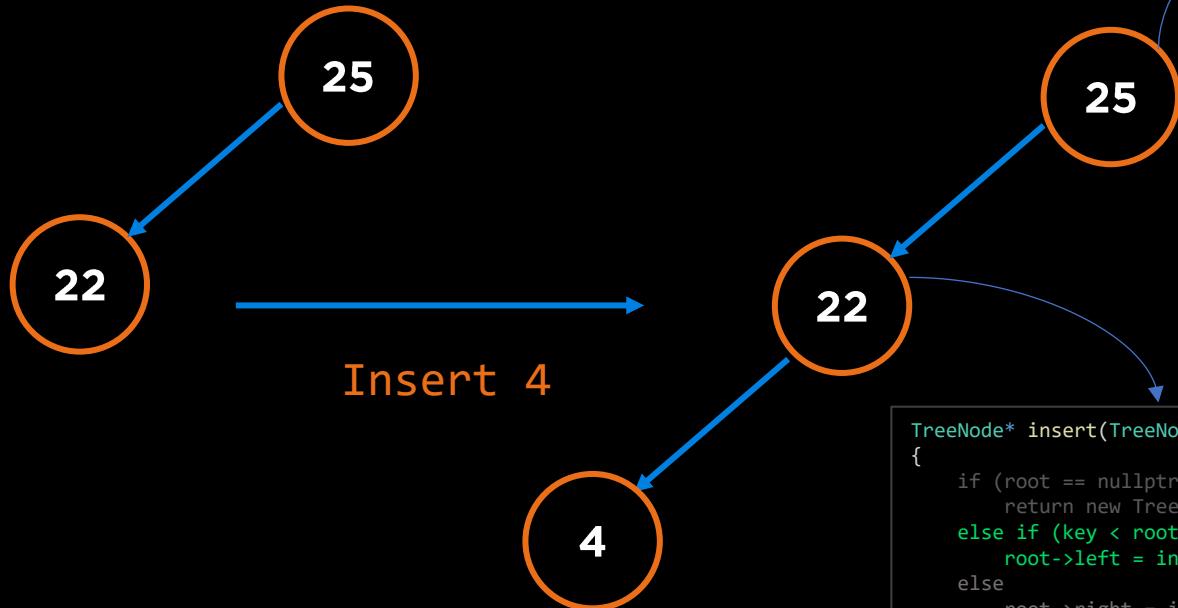
    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



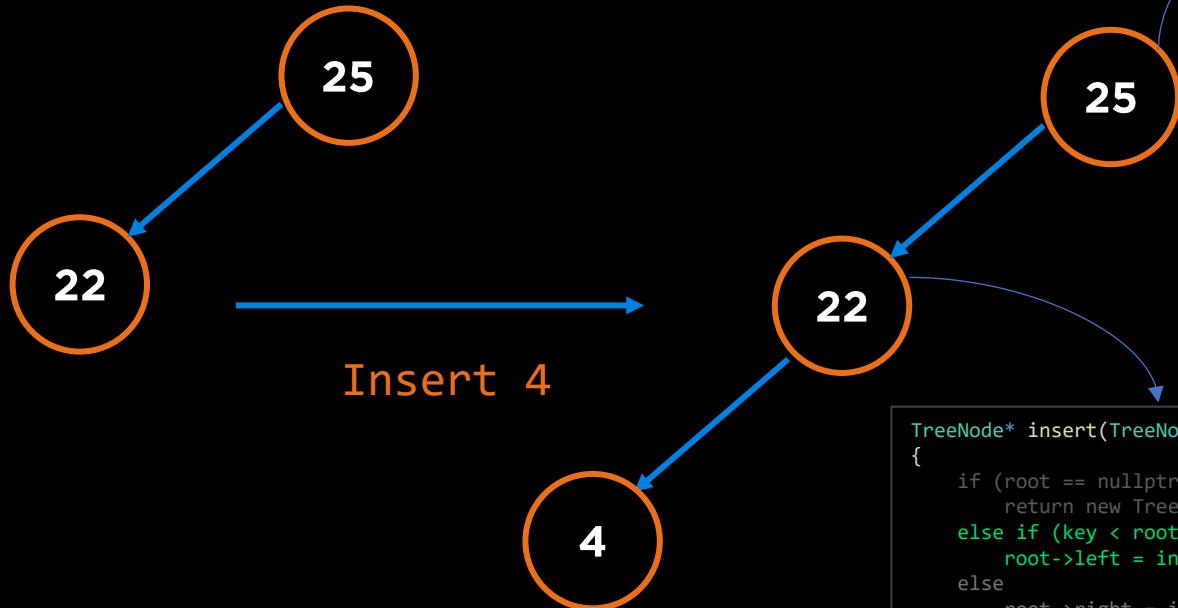
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

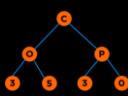


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

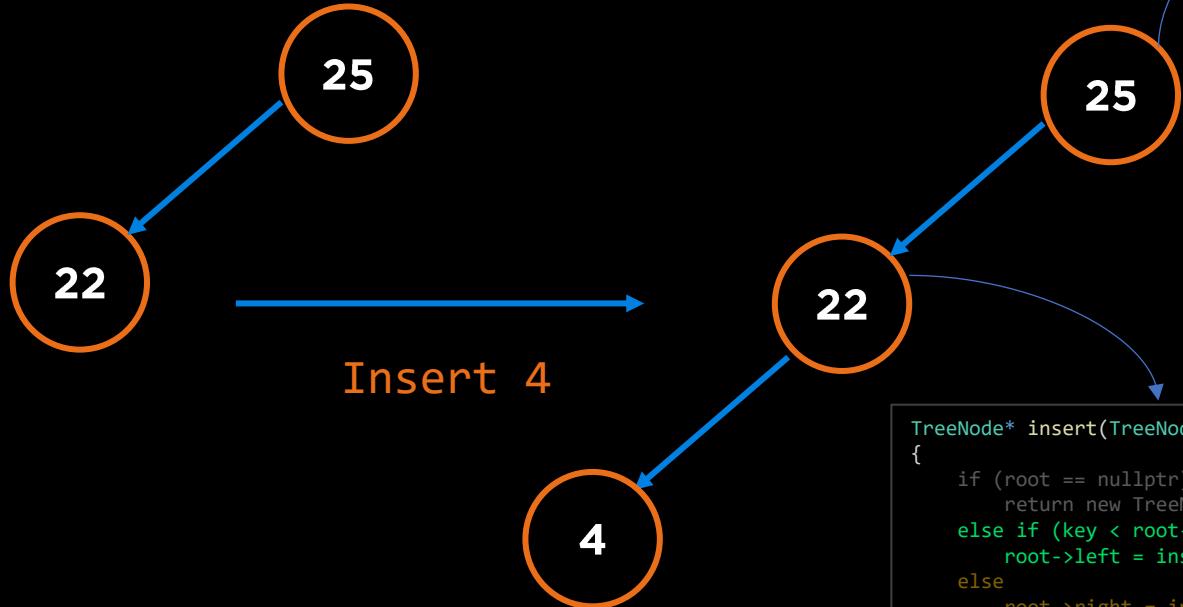
    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```



# AVL Tree : Insert



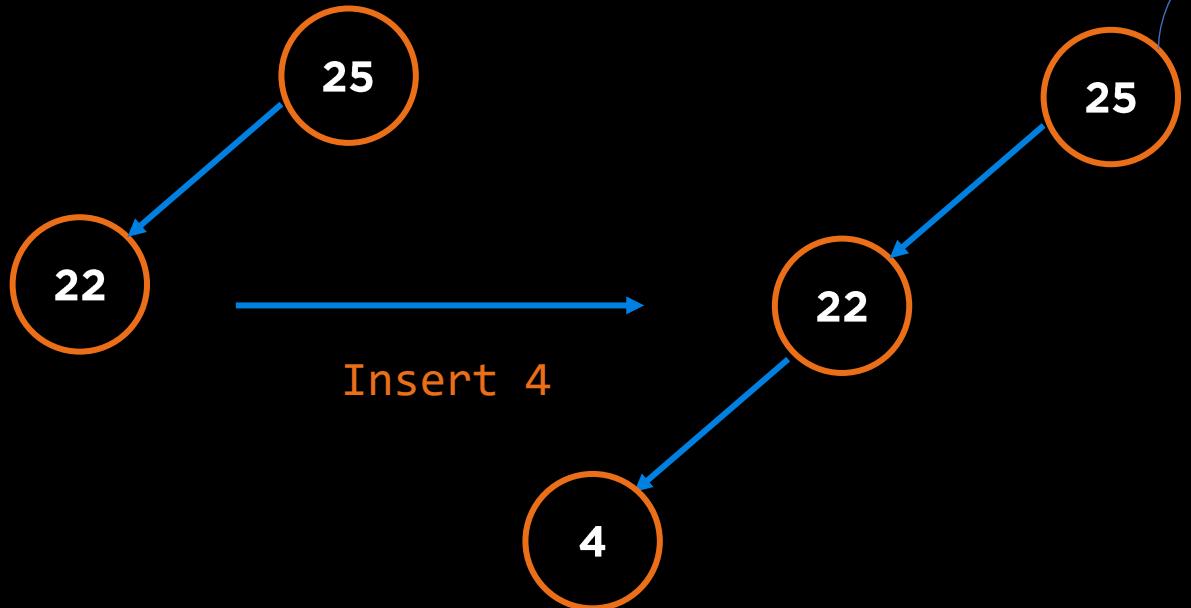
```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

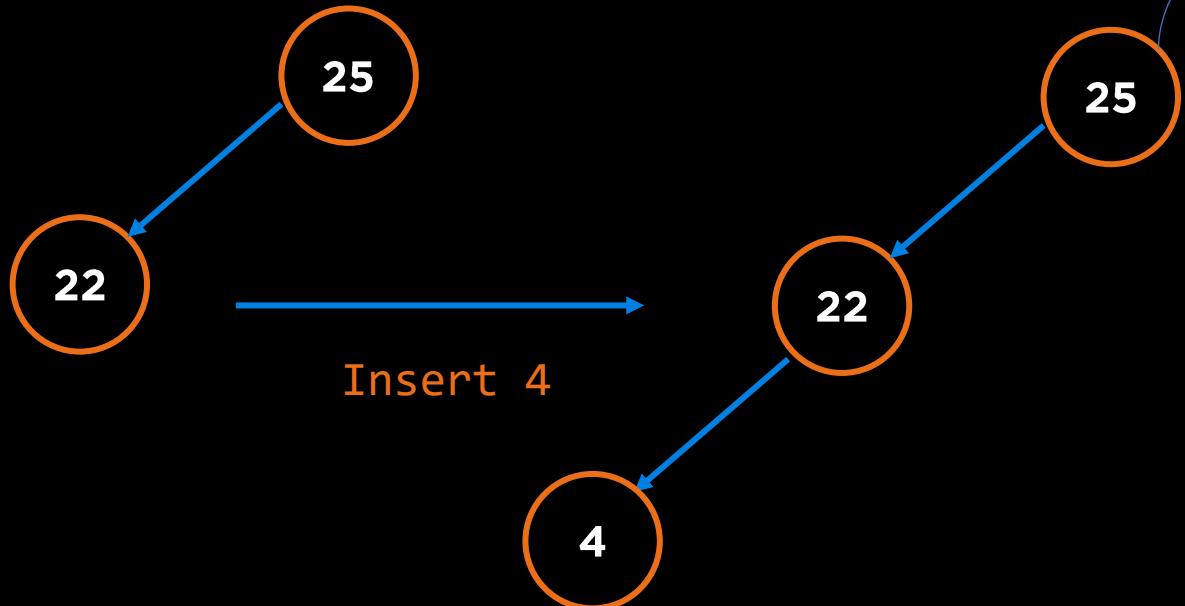
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

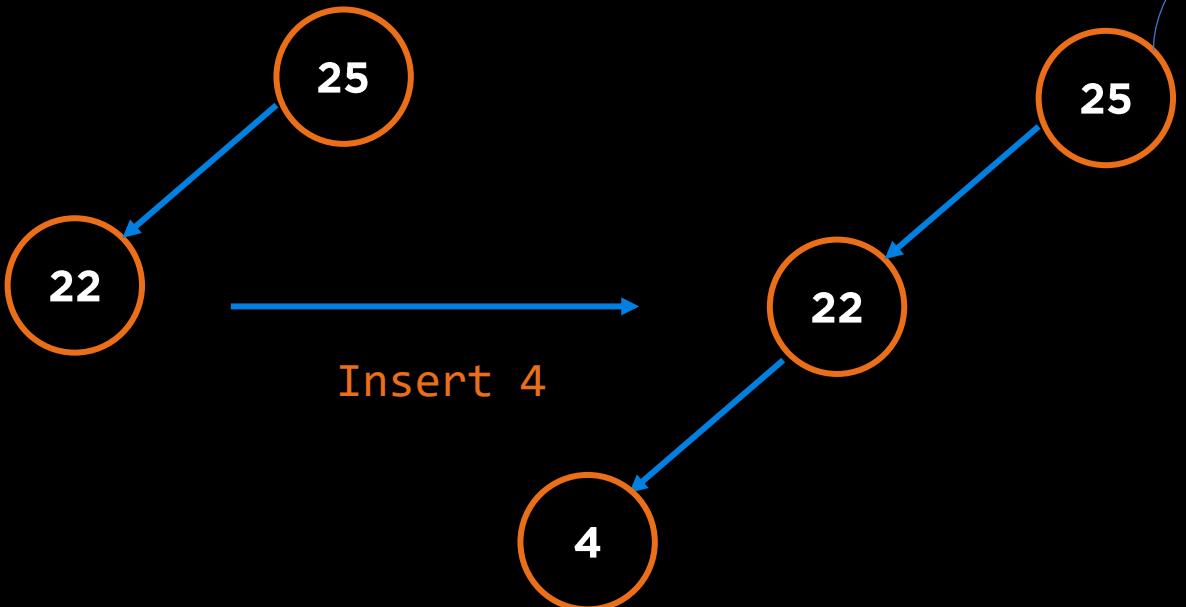
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

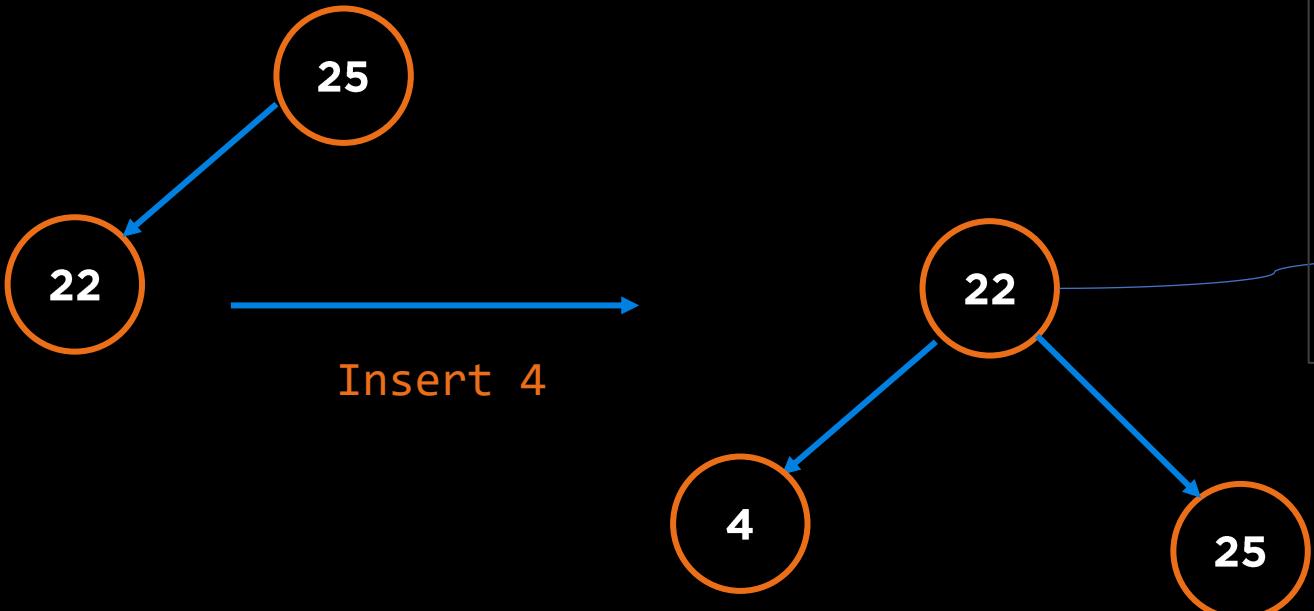
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

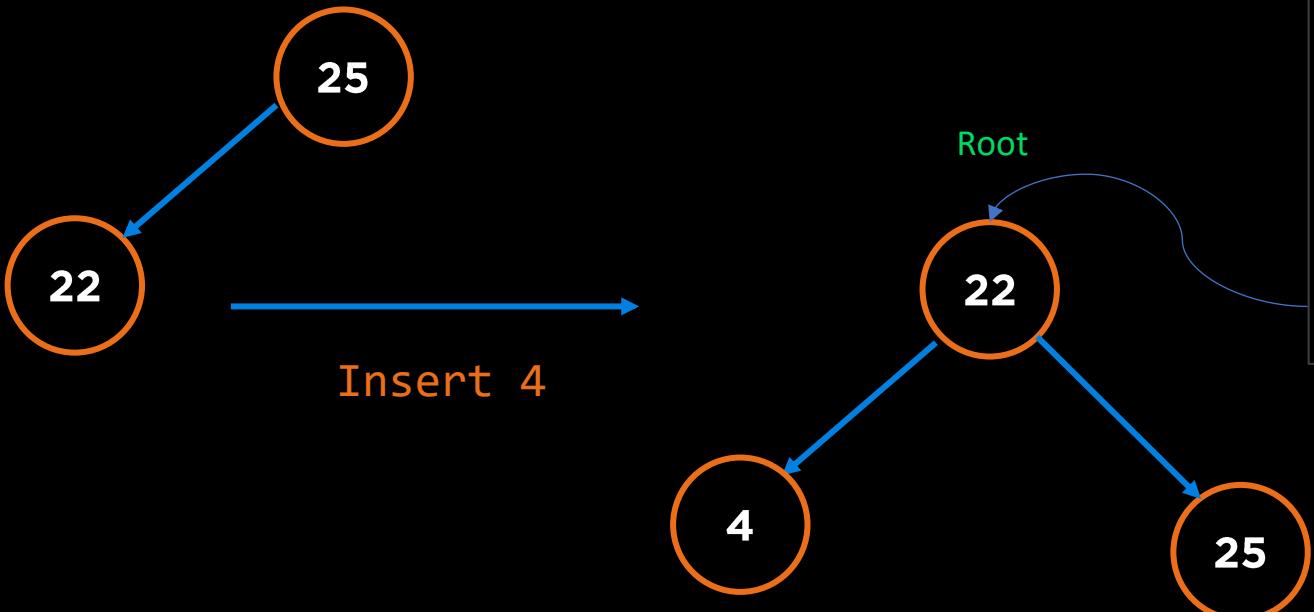
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

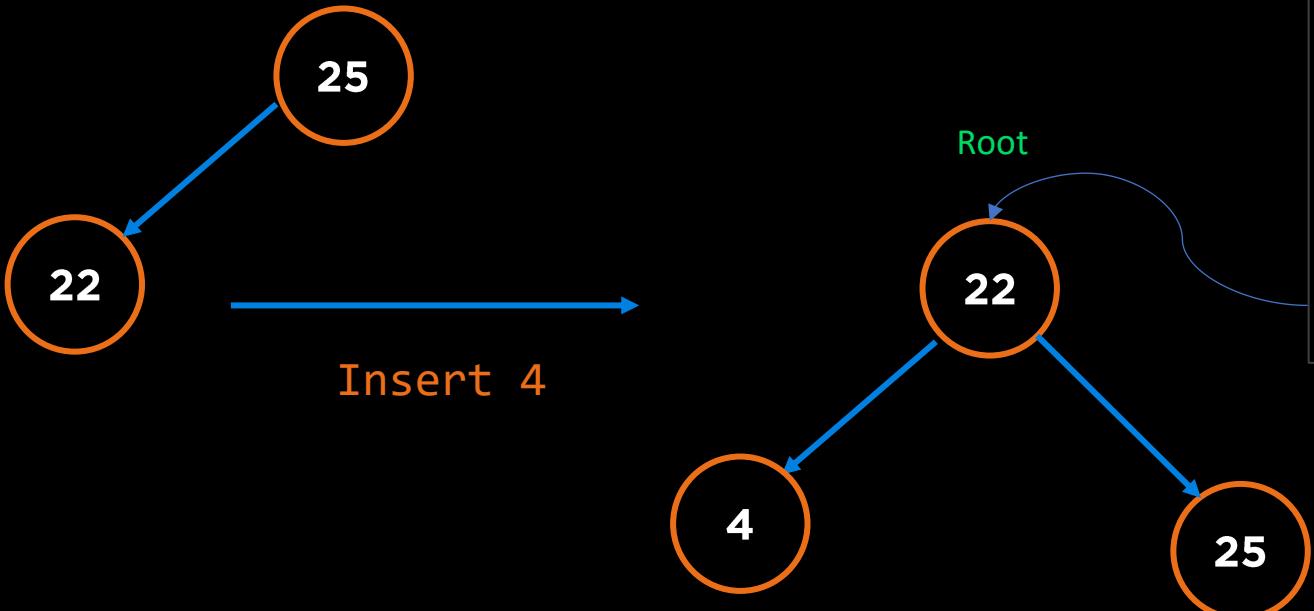
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

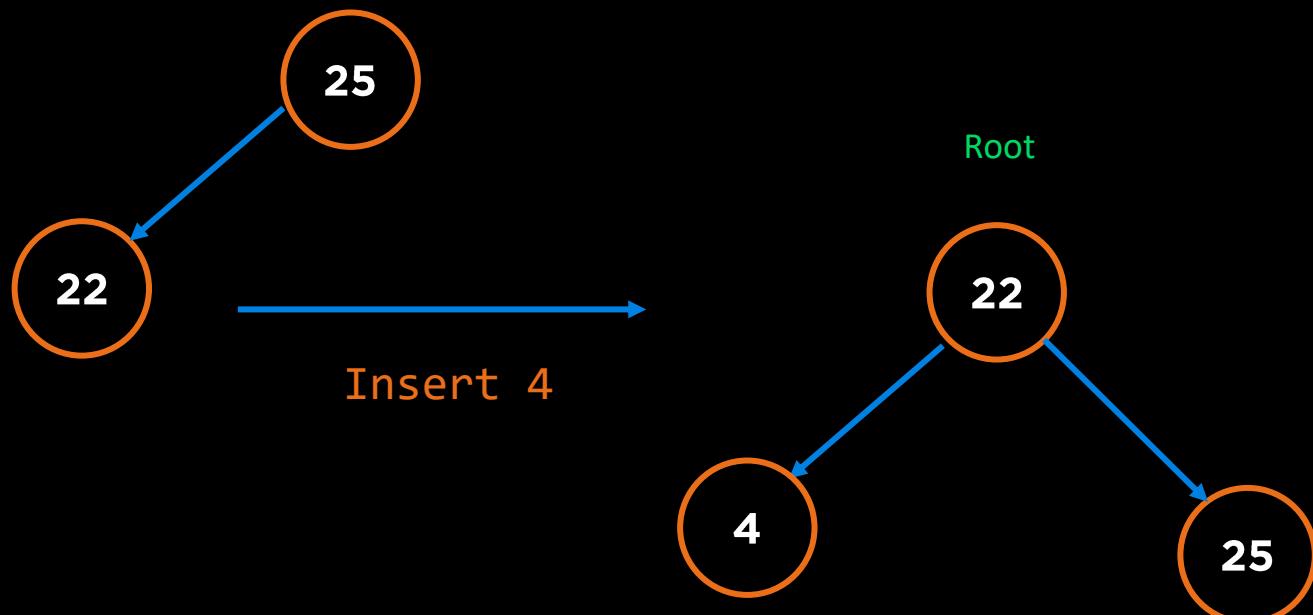
# AVL Tree : Insert



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

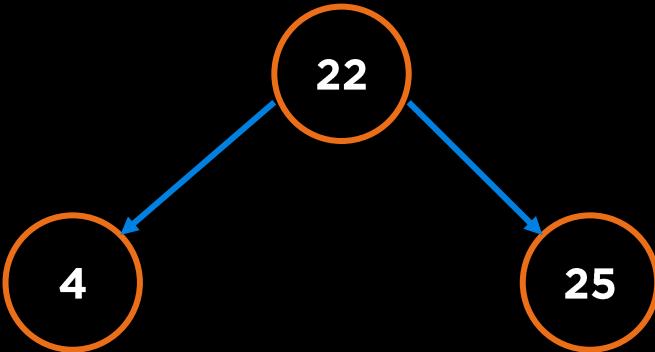
    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



# AVL Tree : Insert

Insert 1

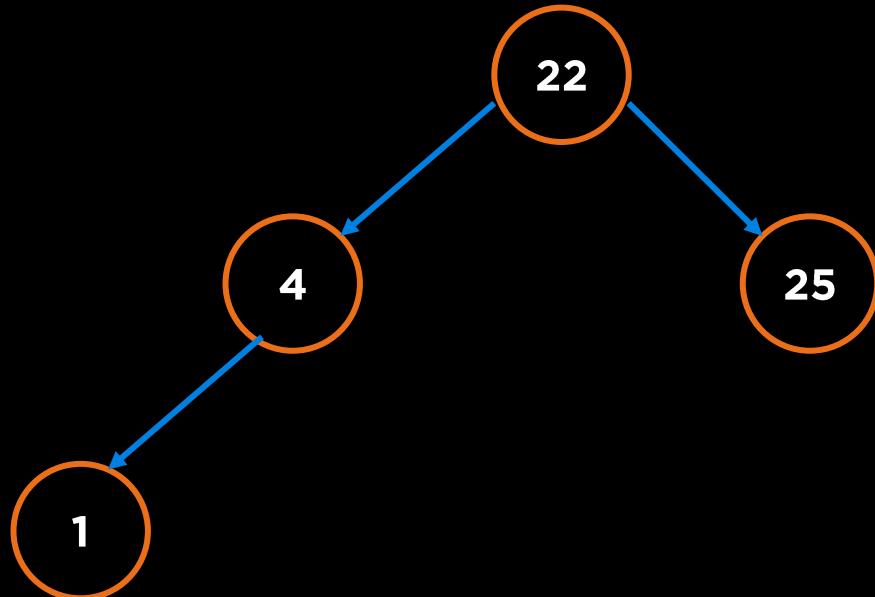


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

Insert 1

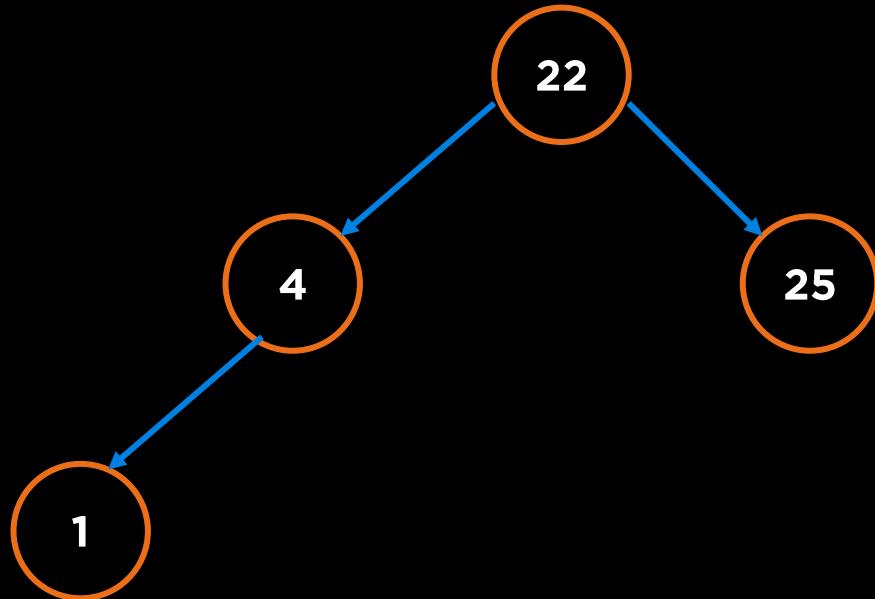


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

Insert 2

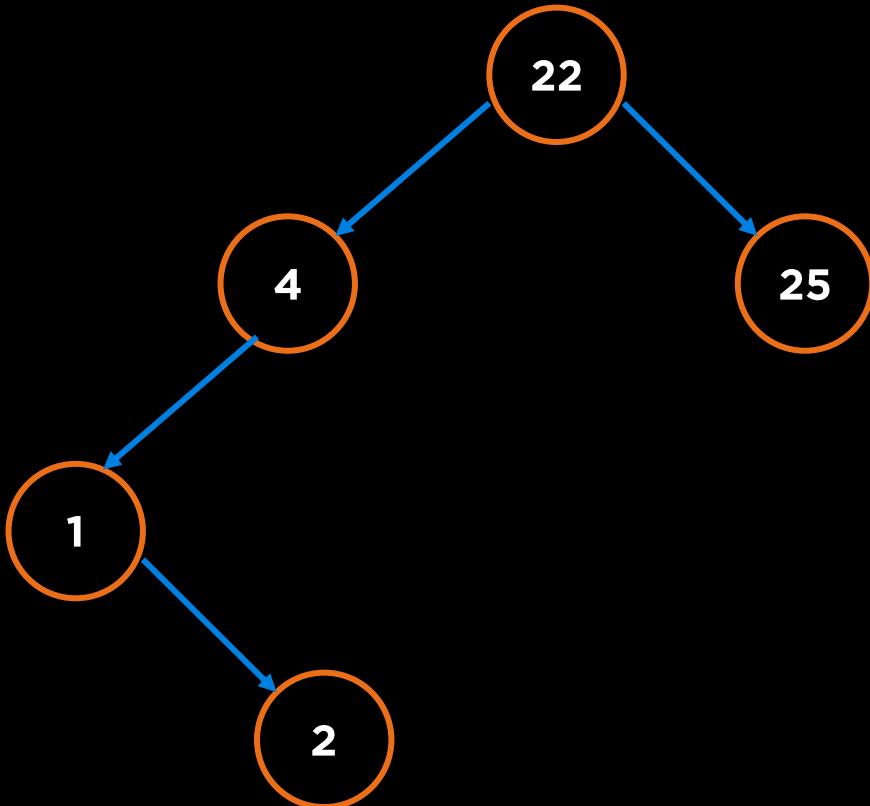


```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert

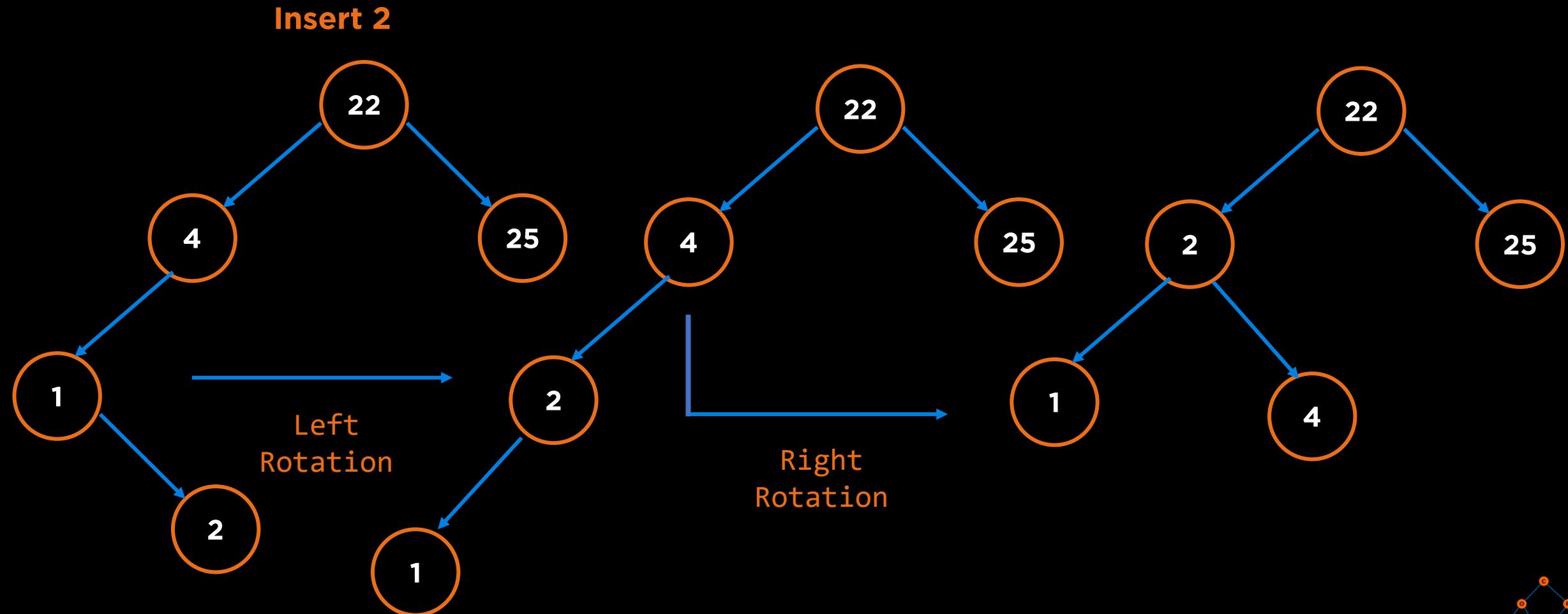
Insert 2



```
TreeNode* insert(TreeNode* root, int key)
{
    if (root == nullptr)
        return new TreeNode(key);
    else if (key < root->val)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    IF tree is right heavy
        IF tree's right subtree is left heavy
            Perform Right Left rotation
        ELSE
            Perform Left rotation
    ELSE IF tree is left heavy
        IF tree's left subtree is right heavy
            Perform Left Right rotation
        ELSE
            Perform Right rotation
    }
    return root;
}
```

# AVL Tree : Insert



# AVL Tree Deletion: After BST Delete

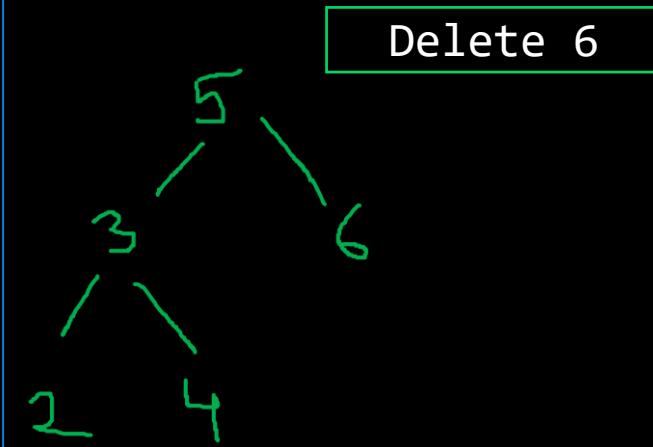
## UPDATE HEIGHTS

```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update height
    ELSE (including equals case)
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update height
    ELSE (including equals case)
        Perform Right rotation & update height
}
```

# AVL Tree Deletion: After BST Delete

## UPDATE HEIGHTS

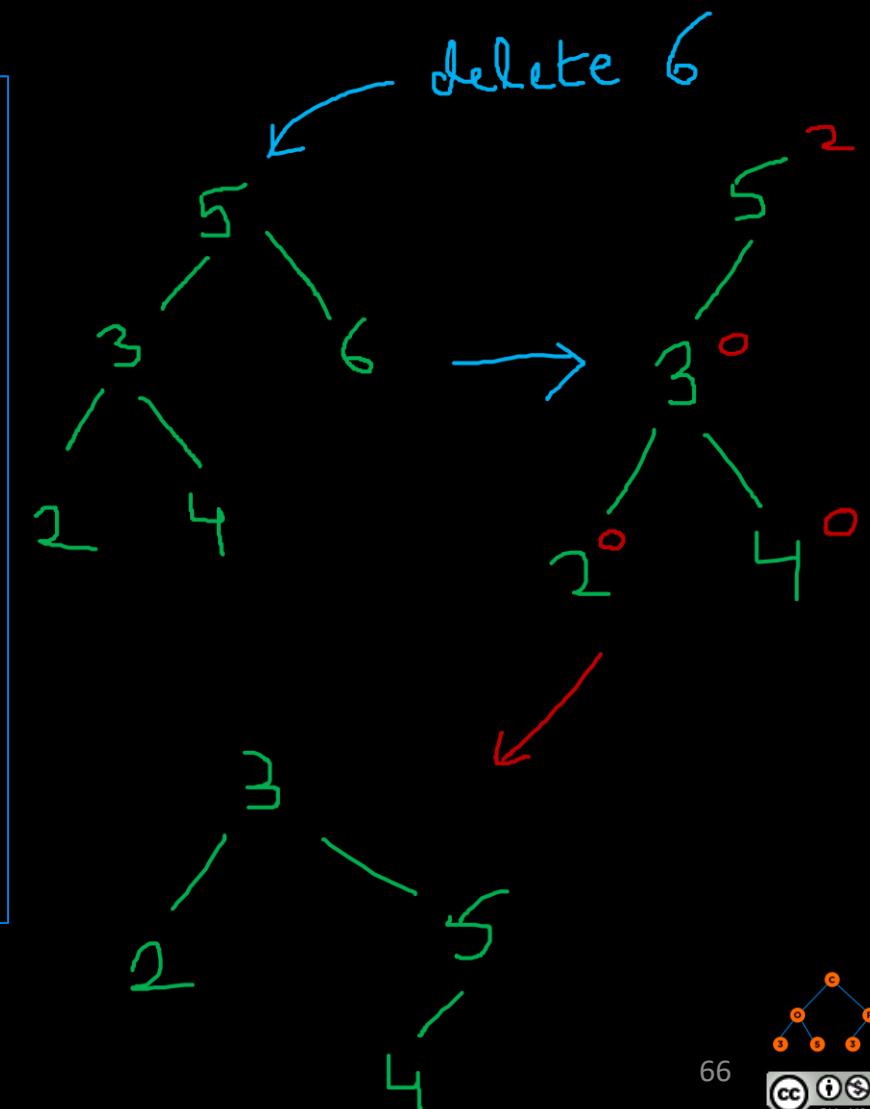
```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update height
    ELSE (including equals case)
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update height
    ELSE (including equals case)
        Perform Right rotation & update height
}
```



# AVL Tree Deletion

## UPDATE HEIGHTS

```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update height
    ELSE (including equals case)
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update height
    ELSE (including equals case)
        Perform Right rotation & update height
}
```



# Question

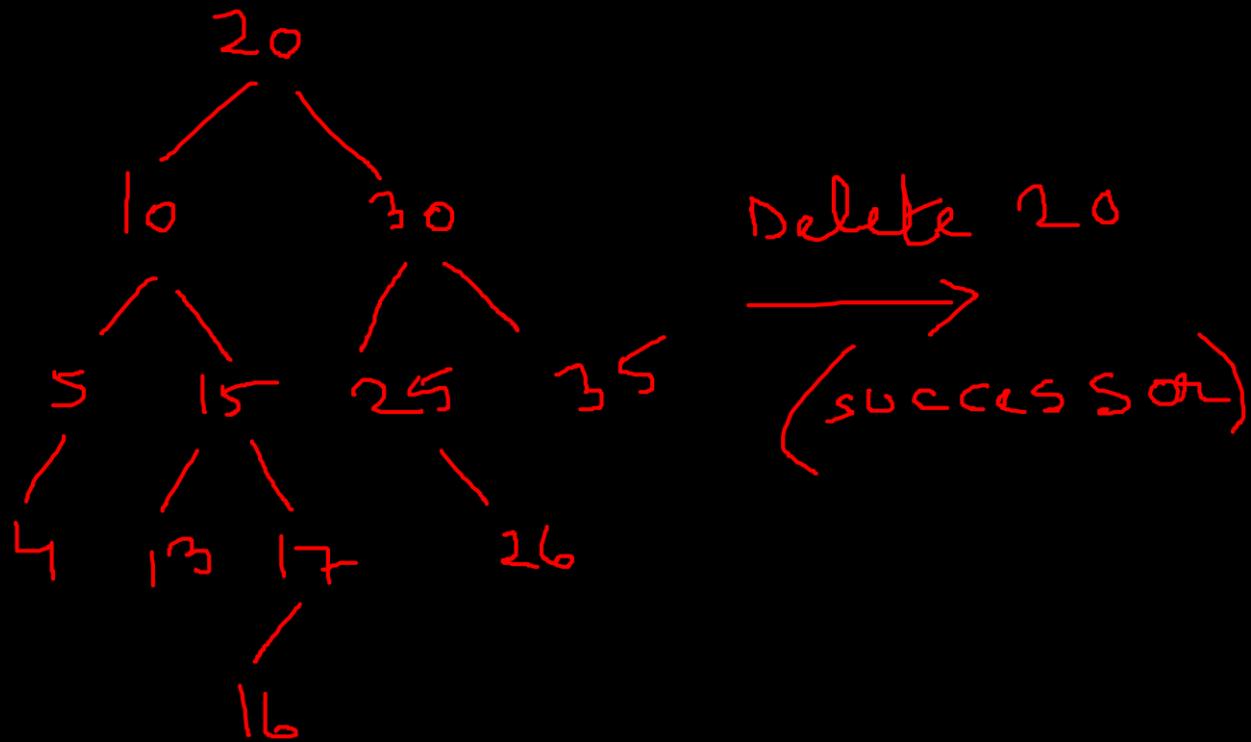
**Apply the following on an AVL Tree:**

- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20

# Question

Apply the following on an AVL Tree:

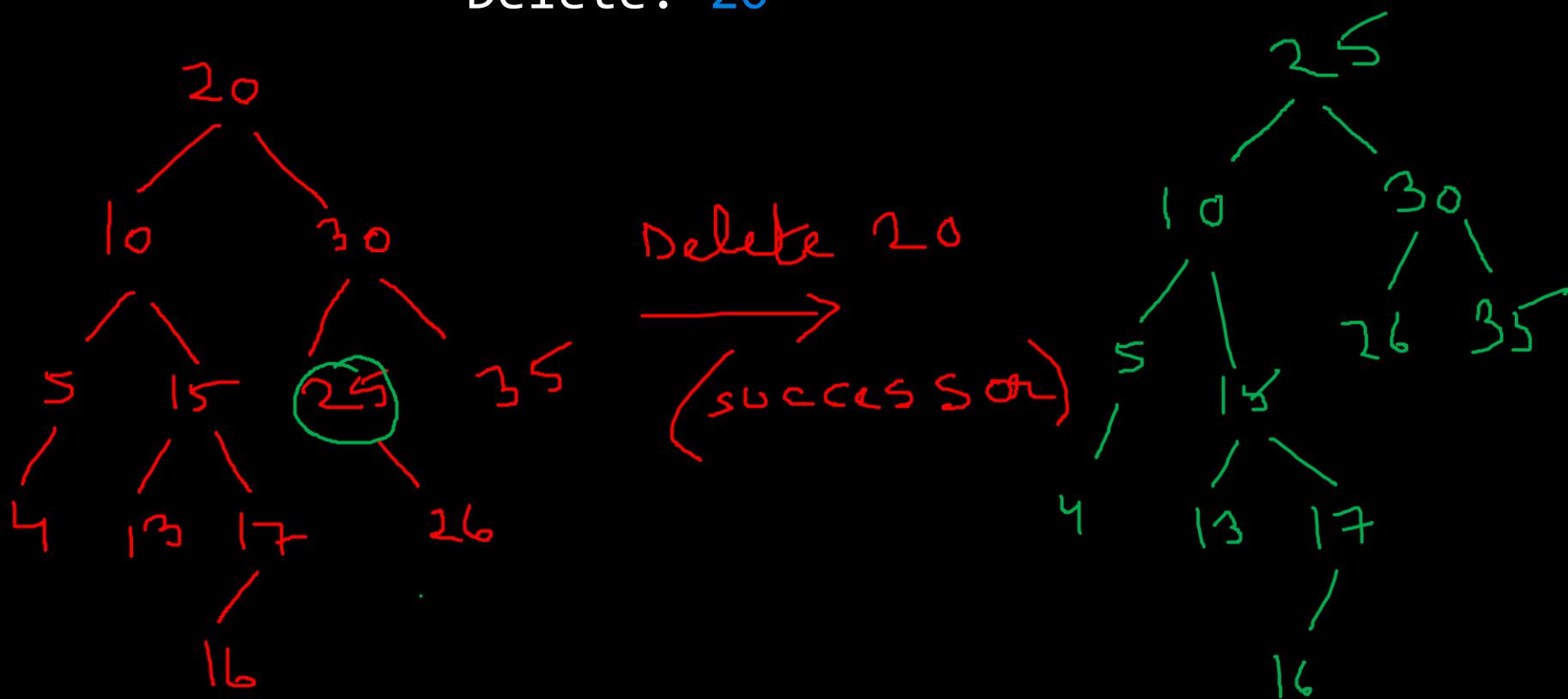
- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20



# Question

Apply the following on an AVL Tree:

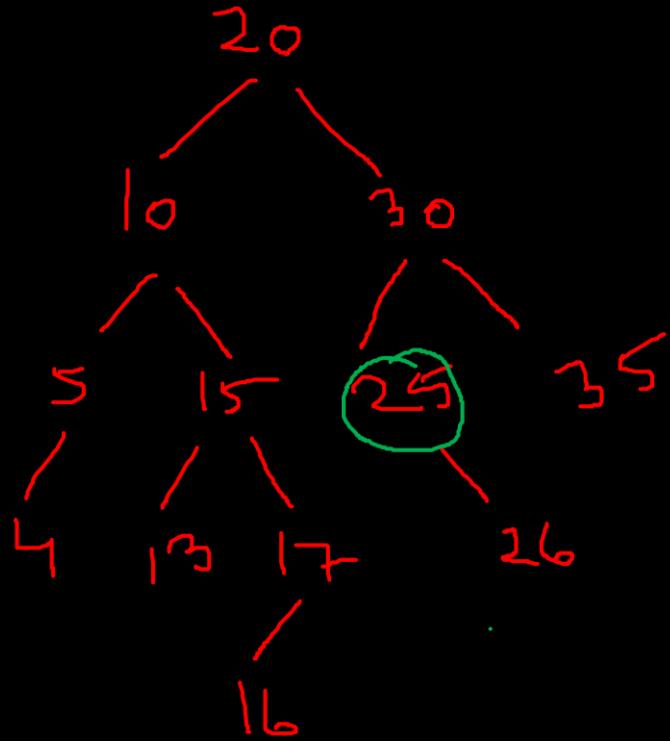
- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20



# Question

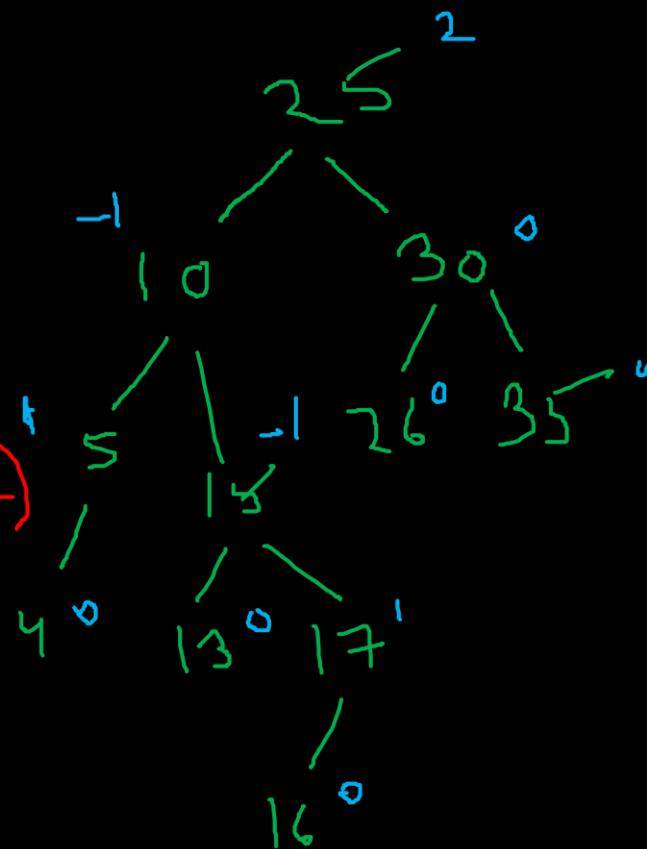
Apply the following on an AVL Tree:

- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20



Delete 20

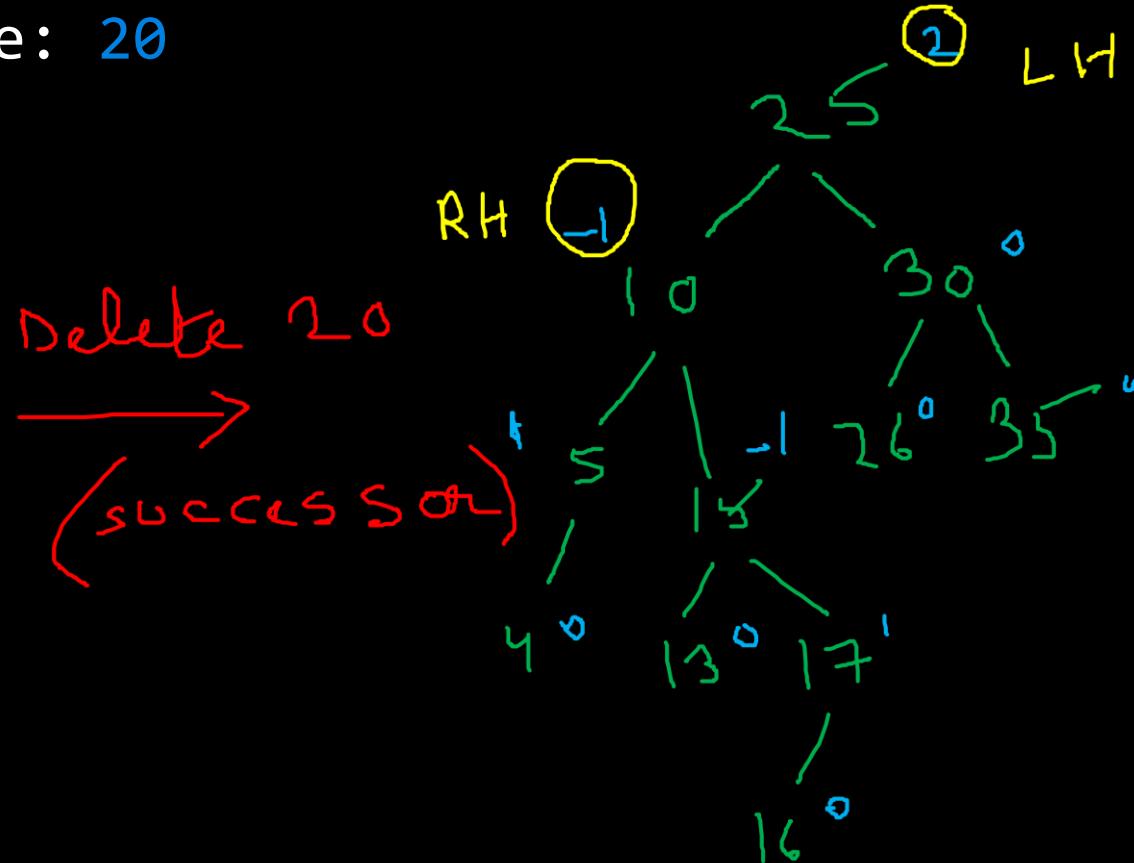
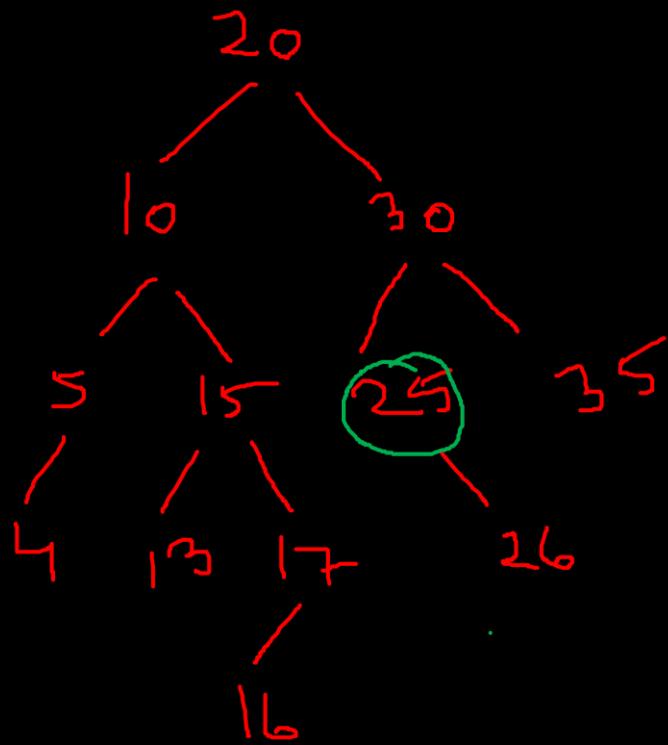
(successor)



# Question

Apply the following on an AVL Tree:

- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20



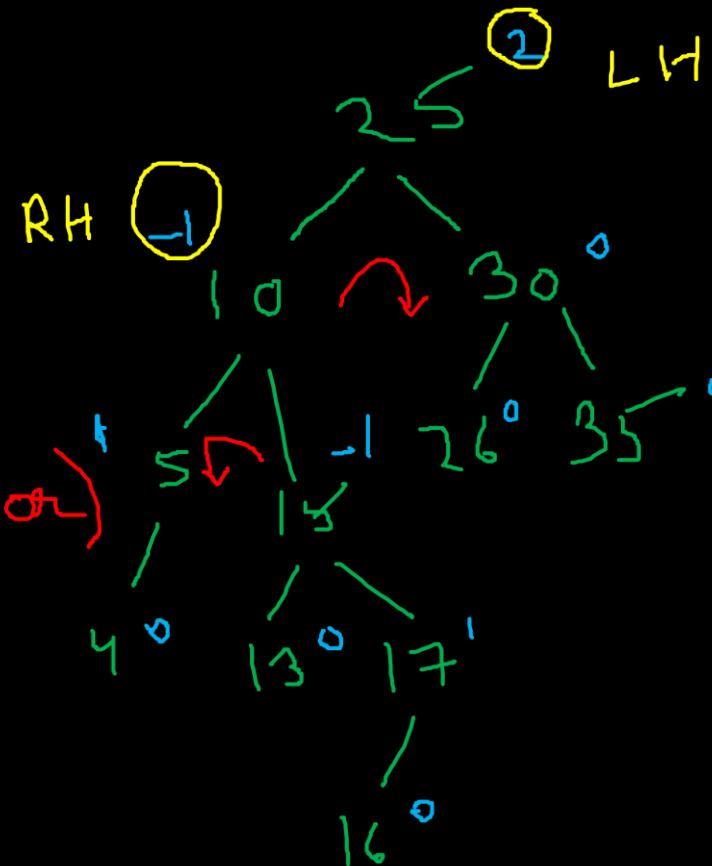
# Question

# Apply the following on an AVL Tree:

- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
  - Delete: 20

Delete 20

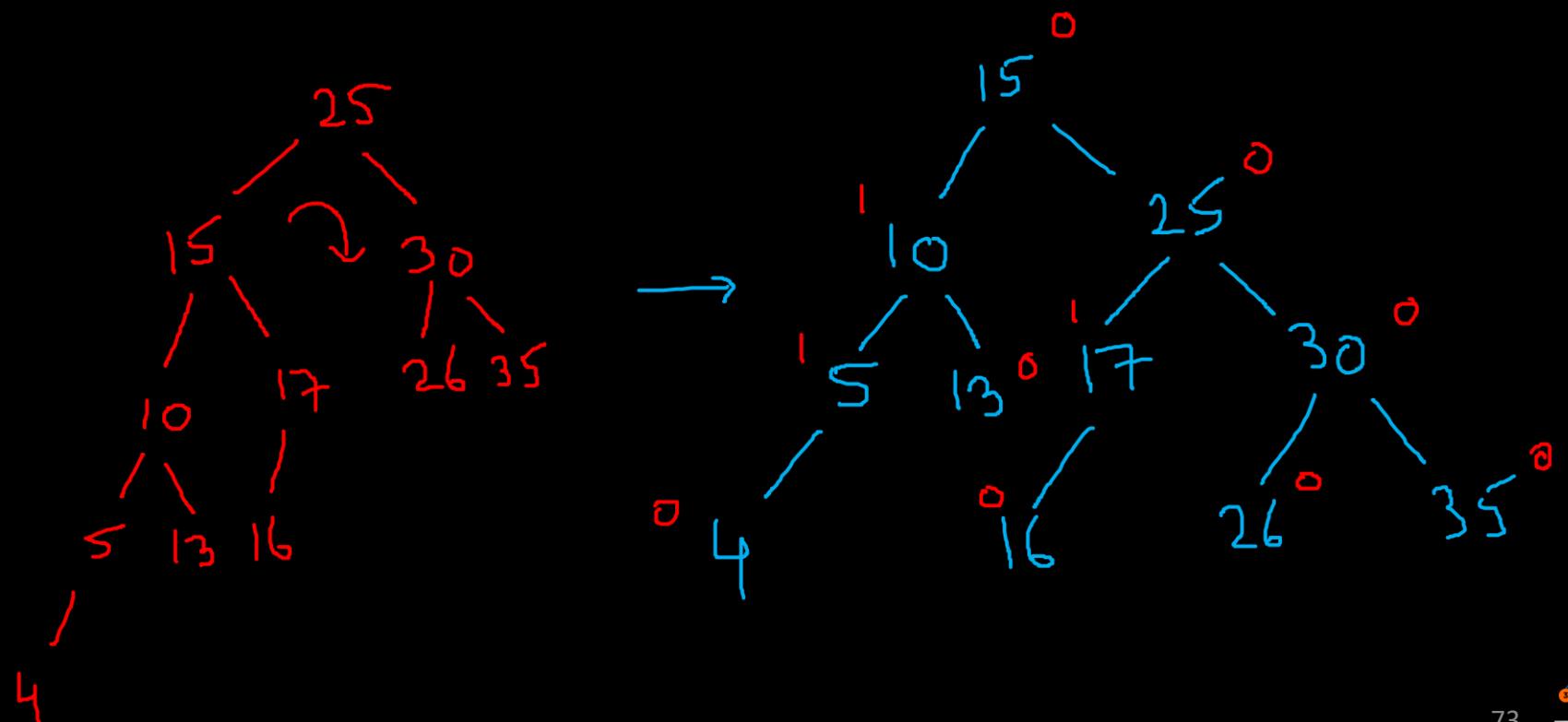
→  
successor



# Question

Apply the following on an AVL Tree:

- Insert: 20, 10, 30, 5, 15, 25, 35, 4, 13, 17, 26, 16
- Delete: 20



# AVL Tree Rotations

Case (Alignment)	Balance Factor		Rotation
	Parent	Child	
Left Left	+2	+1	Right
Right Right	-2	-1	Left
Left Right	+2	-1	Left Right
Right Left	-2	+1	Right Left

```
IF tree is right heavy
{
    IF tree's right subtree is left heavy
        Perform Right Left rotation & update
        height
    ELSE
        Perform Left rotation & update height
}
ELSE IF tree is left heavy
{
    IF tree's left subtree is right heavy
        Perform Left Right rotation & update
        height
    ELSE
        Perform Right rotation & update height
}
```

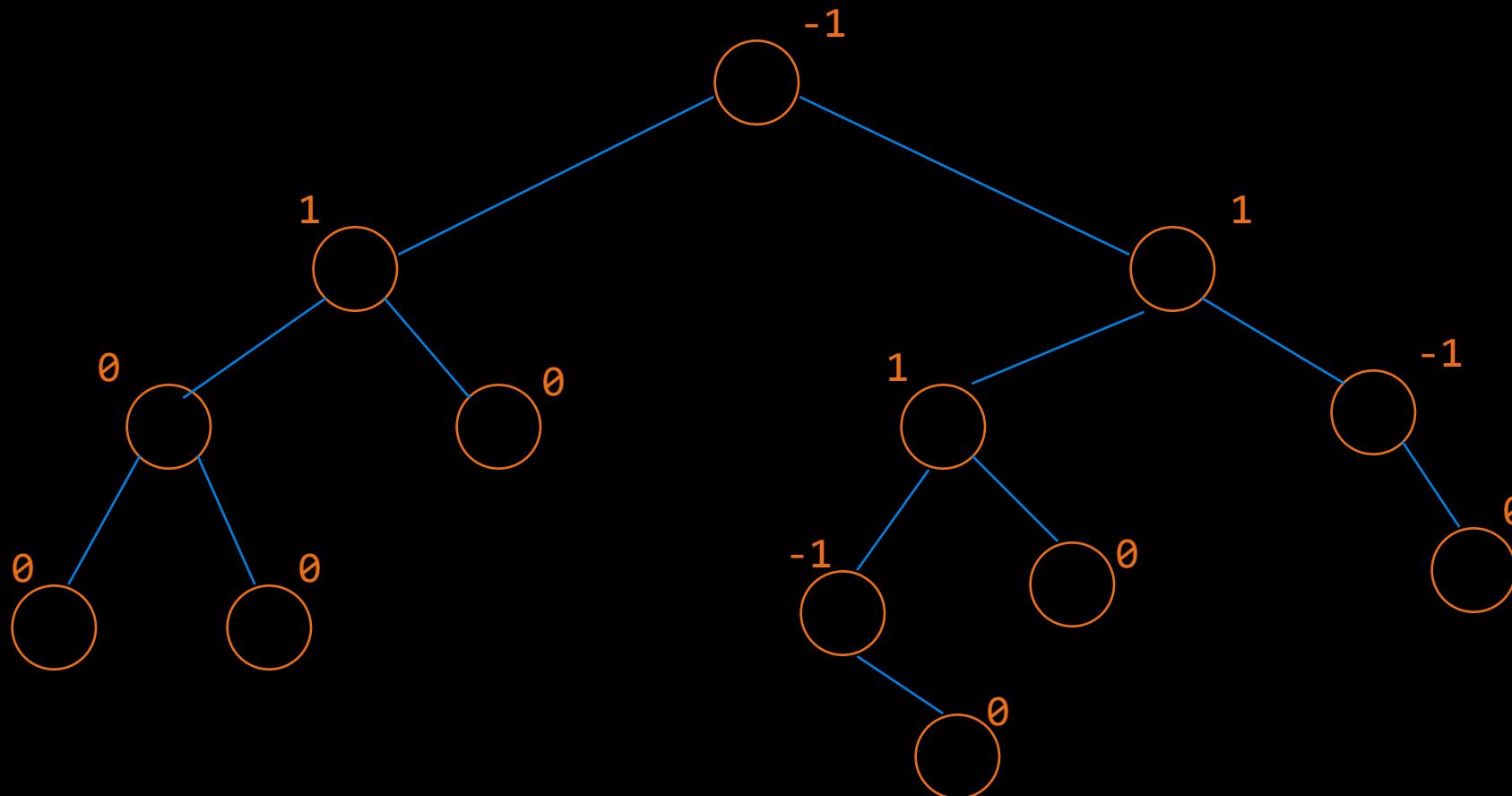
# Mentimeter

**menti.com**  
**84 59 83**



# AVL Tree

Height of an AVL Tree with  $n$  Nodes  $\sim 1.44 \log_2 (n+2)$



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

**Number of Nodes is between  $N_h$  and  $2^h - 1$**

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

**Number of Nodes is between  $N_h$  and  $2^h - 1$**

$$\begin{aligned}N_h &\leq n \leq 2^h - 1 \\n + 1 &\leq 2^h \\\log(n + 1) &\leq \log(2^h) \\\log(n + 1) &\leq \log(2^h) \\\log(n + 1) &\leq h\end{aligned}$$

**Minimum Height of a Tree,  $h \propto O(\log_2(n))$**

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

Number of Nodes is between  $N_h$  and  $2^h - 1$

$$\begin{aligned}N_h &\leq n \leq 2^h - 1 \\n + 1 &\leq 2^h \\\log(n + 1) &\leq \log(2^h) \\\log(n + 1) &\leq \log(2^h) \\\log(n + 1) &\leq h\end{aligned}$$

But we are concerned about the maximum height and not the minimum.

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

**Number of Nodes is between  $N_h$  and  $2^h - 1$**

$$N_h \leq n$$

# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

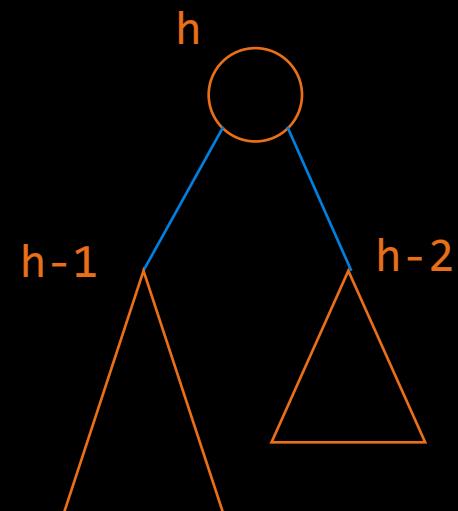
- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

Number of Nodes is between  $N_h$  and  $2^h - 1$

$$N_1 = 1 \quad - (A)$$

$$N_2 = 2 \quad - (B)$$

$$N_h = 1 + N_{h-1} + N_{h-2} \quad - (C)$$



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

Number of Nodes is between  $N_h$  and  $2^h - 1$

$$N_1 = 1 \quad - (A)$$

$$N_2 = 2 \quad - (B)$$

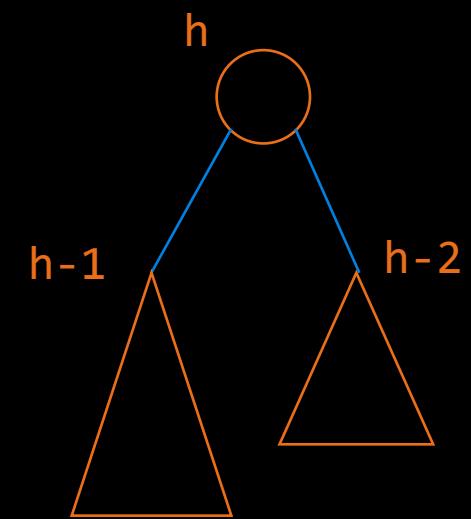
$$N_h = 1 + N_{h-1} + N_{h-2} \quad - (C)$$

Substitute  $h-1$  in the equation C:  $N_{h-1} = 1 + N_{h-2} + N_{h-3}$

Substitute above equation in C:  $N_h = 1 + 1 + N_{h-2} + N_{h-3} + N_{h-2}$

$$N_h = 2 + 2N_{h-2} + N_{h-3}$$

$$N_h \geq 2N_{h-2} \quad - (D)$$



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

$$N_1 = 1 \text{ - (A)}$$

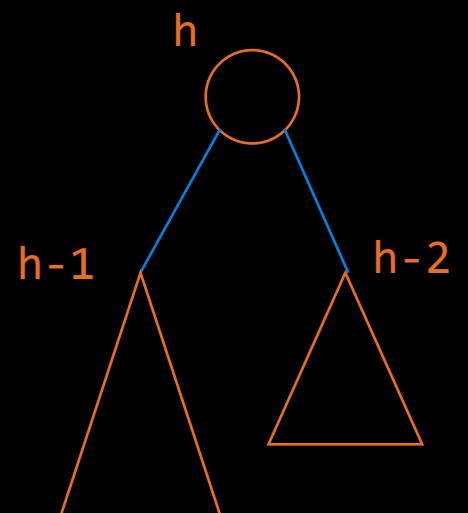
$$N_2 = 2 \text{ - (B)}$$

$$N_h = 1 + N_{h-1} + N_{h-2} \text{ - (C)}$$

$$N_h \geq 2N_{h-2} \text{ - (D)}$$

Replace  $h$  with  $h-2$  in D:  $N_{h-2} \geq 2N_{h-4}$  - (E)

Substitute above equation, E in D:  $N_h \geq 2 \cdot 2 \cdot N_{h-4}$  - (F)



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

$$N_1 = 1 \text{ - (A)}$$

$$N_2 = 2 \text{ - (B)}$$

$$N_h = 1 + N_{h-1} + N_{h-2} \text{ - (C)}$$

$$N_h \geq 2N_{h-2} \text{ - (D)}$$

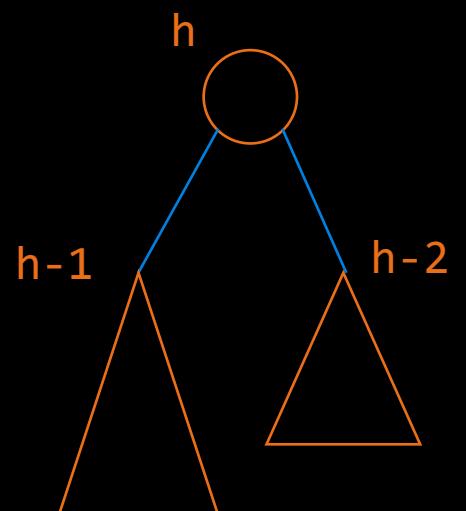
Replace  $h$  with  $h-2$  in D:  $N_{h-2} \geq 2N_{h-4}$  - (E)

Substitute above equation, E in D:  $N_h \geq 2 \cdot 2 \cdot N_{h-4}$  - (F)

Replace  $h$  with  $h-4$  in D:  $N_{h-4} \geq 2N_{h-6}$  - (G)

Substitute above equation, G in F:  $N_h \geq 2 \cdot 2 \cdot 2 \cdot N_{h-6}$

Generalize:  $N_h \geq 2^i \cdot N_{h-2i}$  - (H)



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

$$N_1 = 1 \quad - (A)$$

$$N_2 = 2 \quad - (B)$$

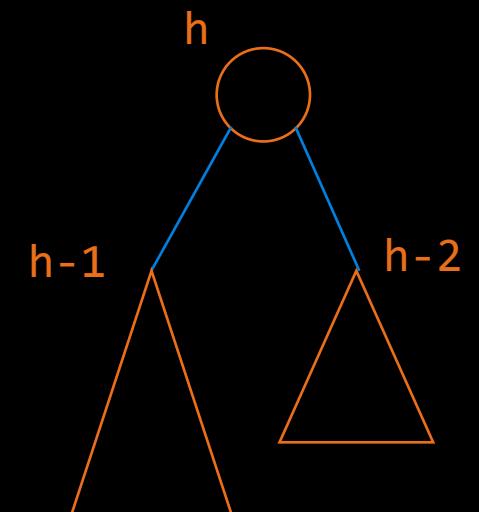
$$N_h \geq 2^i \cdot N_{h-2i} \quad - (H)$$

For,  $i = h/2 - 1$ ,

$$N_h \geq 2^{h/2 - 1} \cdot N_2$$

$$N_h \geq 2^{h/2 - 1} \cdot 2$$

$$N_h \geq 2^{h/2}$$



# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the minimum number of nodes in an AVL Tree of height,  $h$ ?  $\Rightarrow n = N_h$

$$N_1 = 1 \quad - (A)$$

$$N_2 = 2 \quad - (B)$$

$$N_h \geq 2^i \cdot N_{h-2i} \quad - (H)$$

For,  $i = h/2 - 1$ ,

$$N_h \geq 2^{h/2 - 1} \cdot N_2$$

$$N_h \geq 2^{h/2 - 1} \cdot 2$$

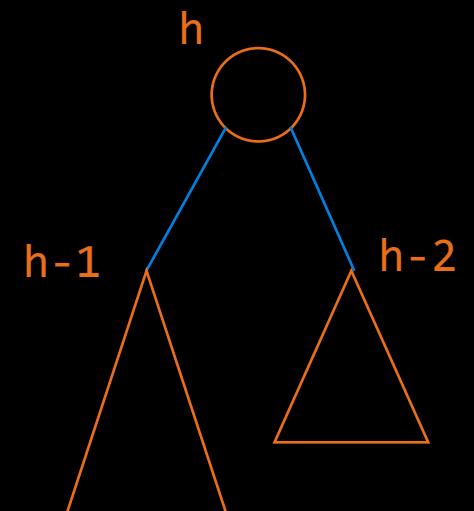
$$N_h \geq 2^{h/2}$$

$$\log_2 N_h \geq \log_2 2^{h/2}$$

$$\log_2 N_h \geq (h/2) \log_2 2$$

$$2 \cdot \log_2 N_h \geq h$$

$$h \leq 2 \cdot \log_2 N_h$$

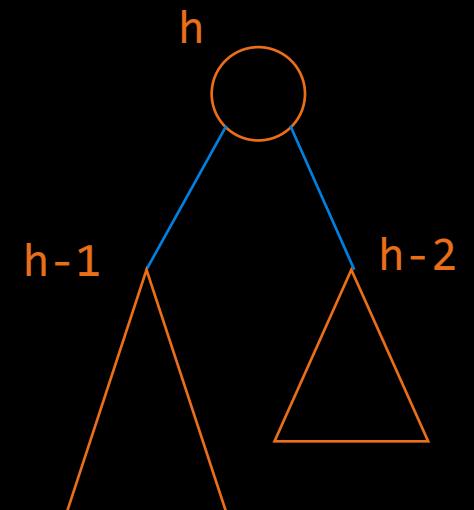


# AVL Tree

Height of an AVL Tree Proof, Questions to ask:

- What are the maximum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = 2^h - 1$
- What are the minimum number of nodes in an AVL Tree of height,  $h$ ? =>  $n = N_h$

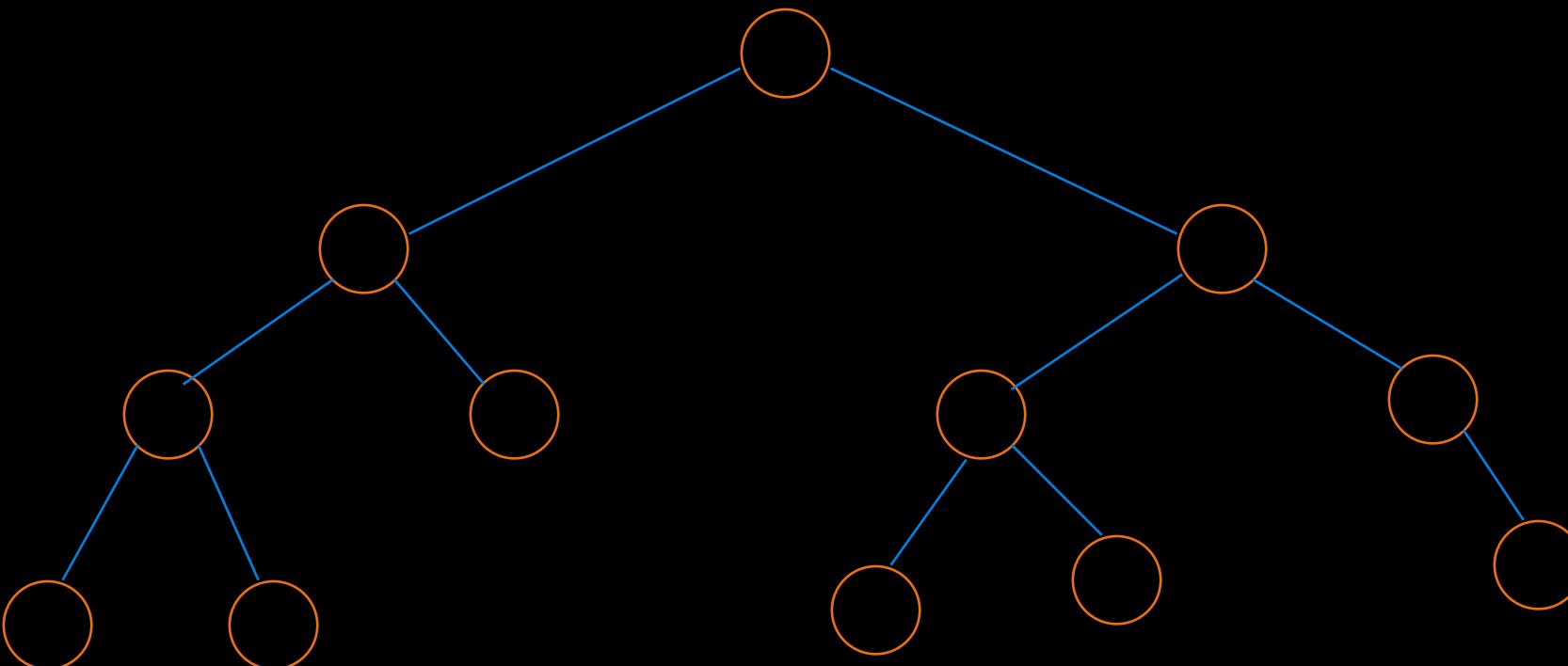
**Maximum Height of a Tree,  $h \propto O(\log_2(n))$**



# AVL Insert, Delete and Search

**Worst Case ~ Height =  $\log n$**

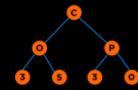
**And Common Operations will be  $O(\log n)$**



# Questions



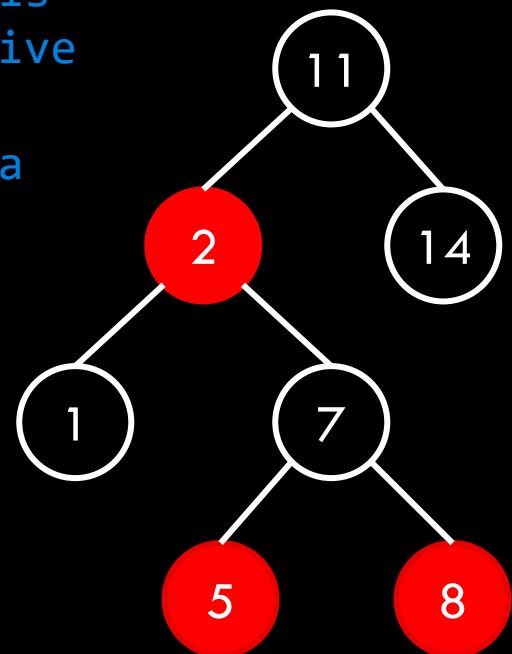
# Red Black Tree



# Red Black Tree Properties

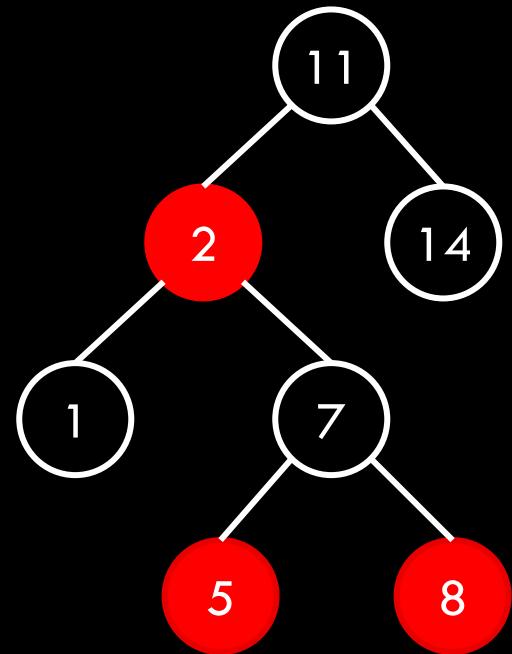
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



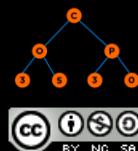
# Red Black Tree General Idea

- Color is stored as a Boolean in the **TreeNode class**
- Fixing the tree invariants by rotation or through color flips



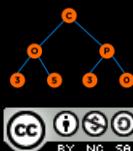
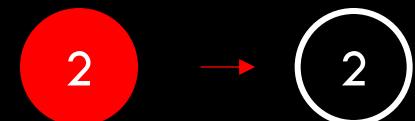
# Red Black Tree Insertion

- **Insert the item into the binary search tree as usual**
- **Color it red**
- **If the tree is empty, color it black and make it a root, we are done**



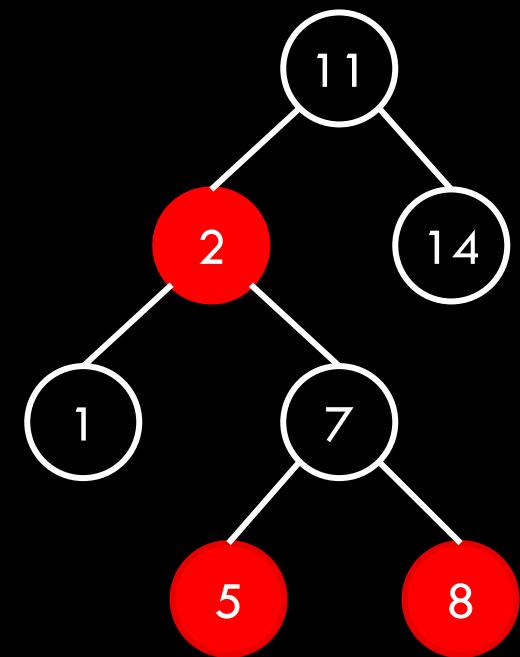
# Red Black Tree Insertion

- **Insert the item into the binary search tree as usual**
- **Color it red**
- **If the tree is empty, color it black and make it a root, we are done**



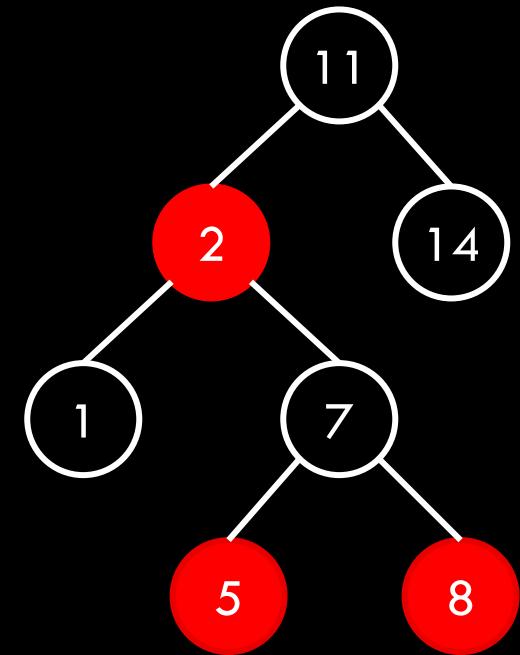
# Red Black Tree Insertion

- **Insert the item into the binary search tree as usual**
- **Color it red**
- **If the parent is black, we are done**



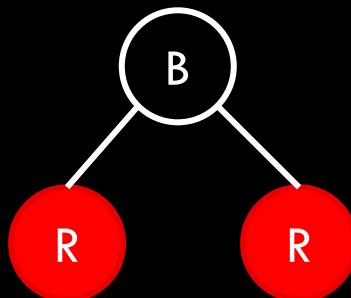
# Red Black Tree Insertion

- **Insert the item into the binary search tree as usual**
- **Color it red**
- **If the parent is red, look at the aunt/uncle or parent's sibling**

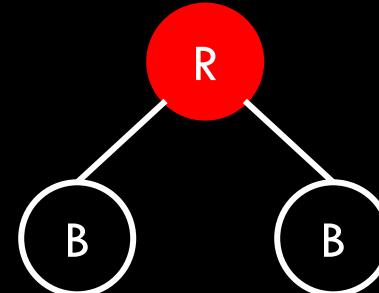


# Red Black Tree Insertion

- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



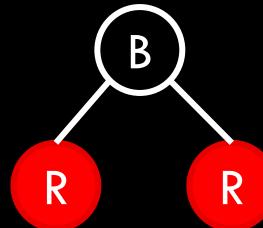
After Color Flip (P, GP)



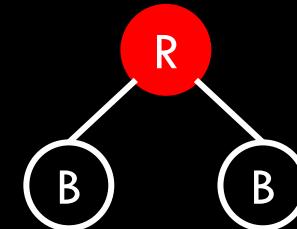
# Example

## Insert 3

- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)



A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

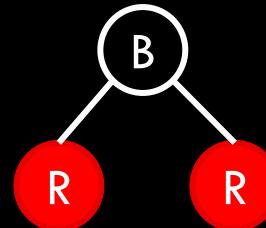


# Example

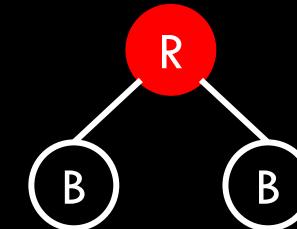
## Insert 3

3

- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

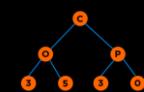


After Color Flip (P, GP)



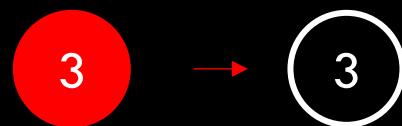
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

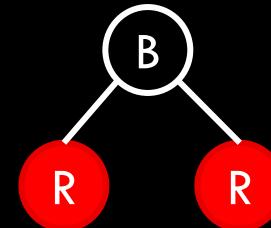


# Example

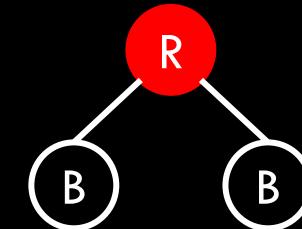
Insert 3



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)



A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

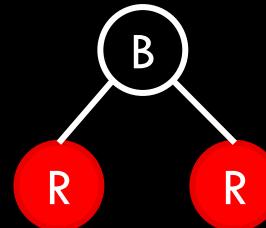


# Example

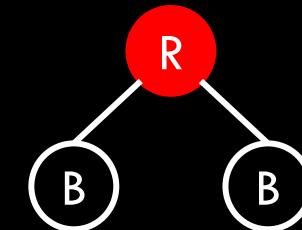
## Insert 1



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)



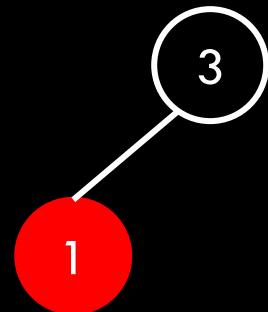
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

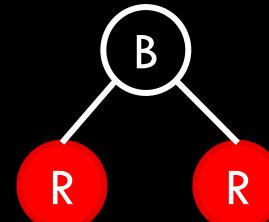


# Example

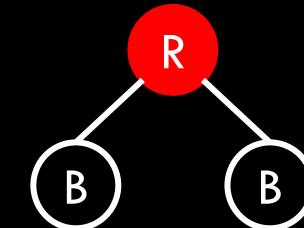
## Insert 1



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

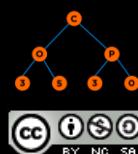


After Color Flip (P, GP)



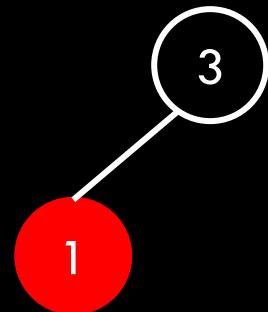
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

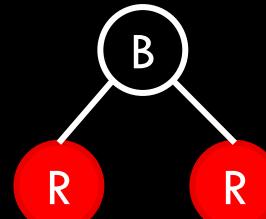


# Example

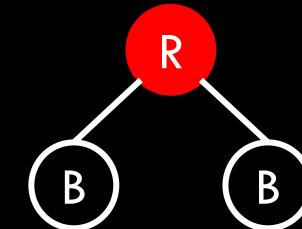
Insert 5



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

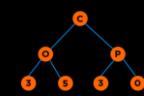


After Color Flip (P, GP)



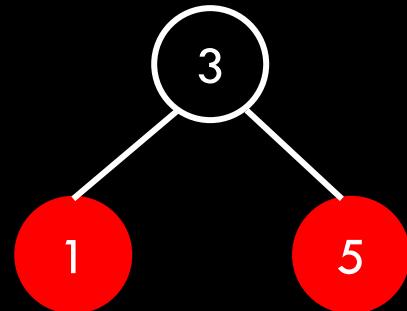
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

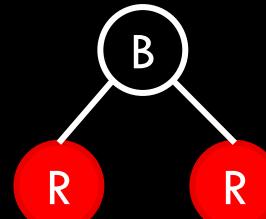


# Example

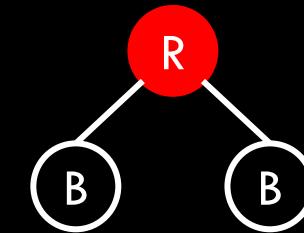
Insert 5



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

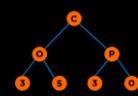


After Color Flip (P, GP)



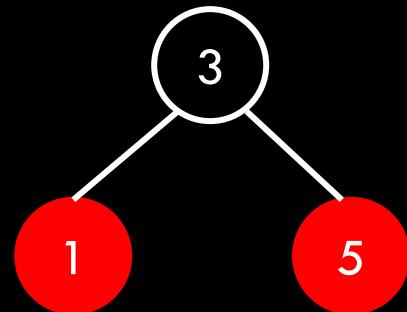
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

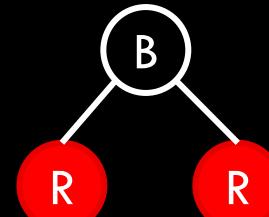


# Example

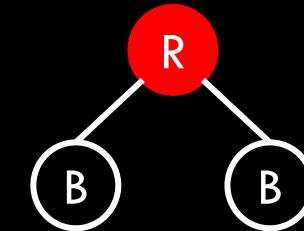
Insert 7



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

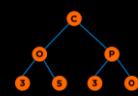


After Color Flip (P, GP)



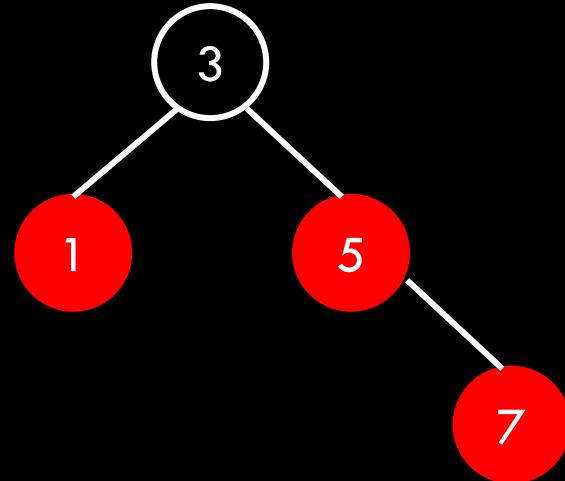
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

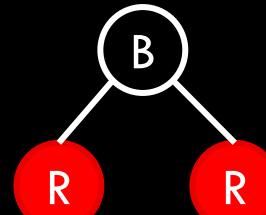


# Example

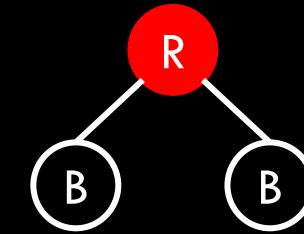
Insert 7



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

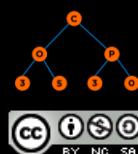


After Color Flip (P, GP)



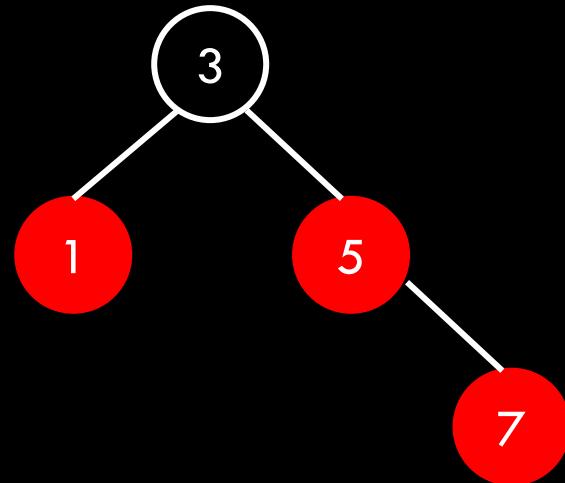
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

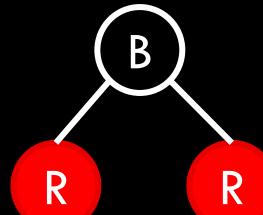


# Example

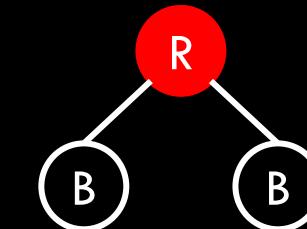
Insert 7



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

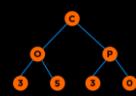


After Color Flip (P, GP)



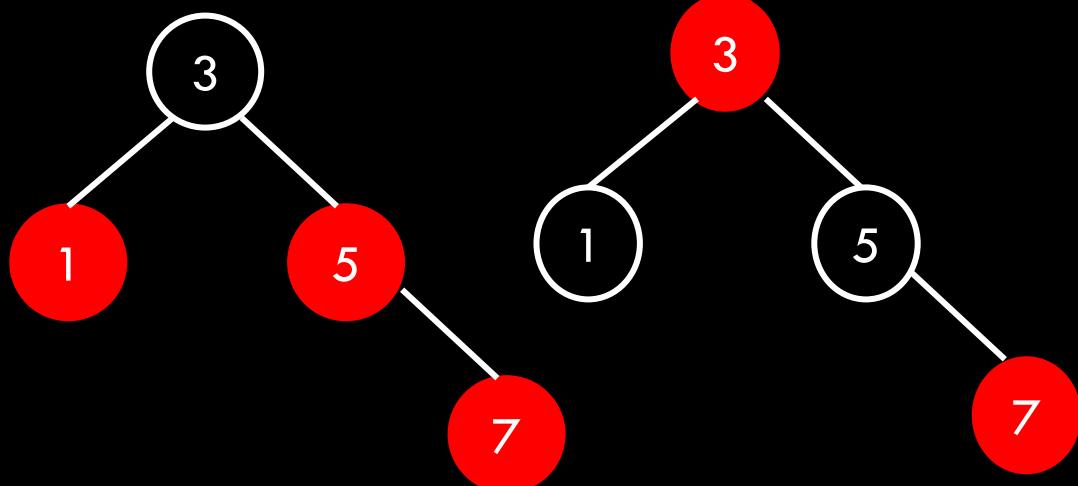
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

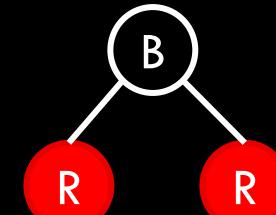


# Example

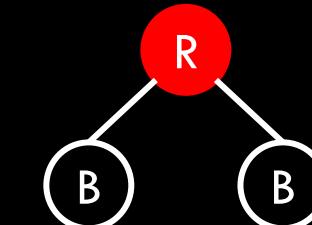
Insert 7



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

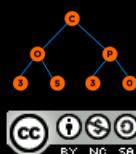


After Color Flip (P, GP)



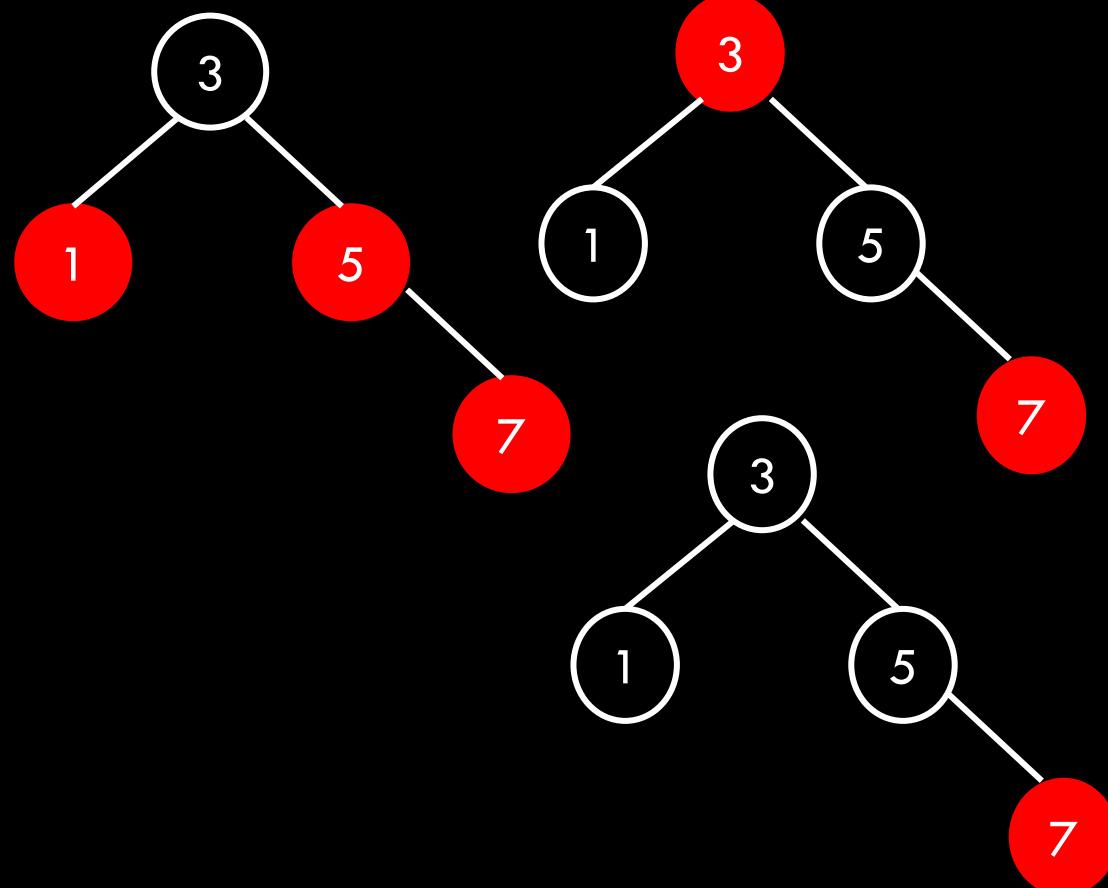
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

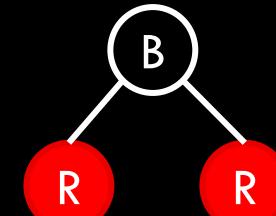


# Example

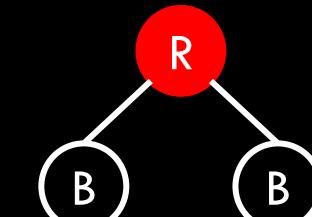
Insert 7



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

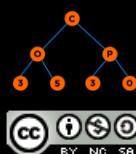


After Color Flip (P, GP)



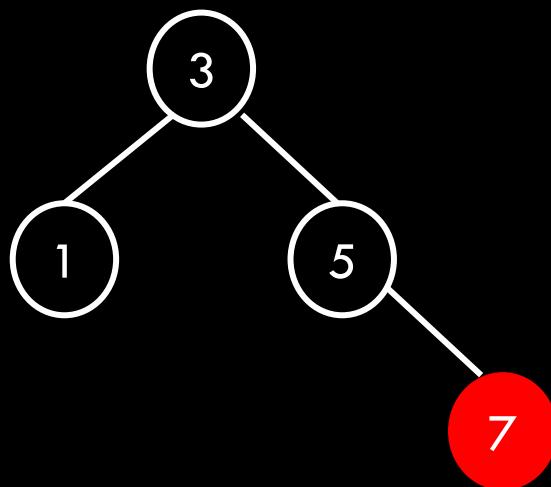
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

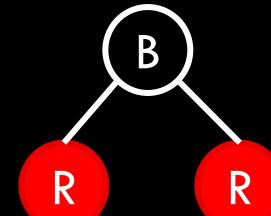


# Example

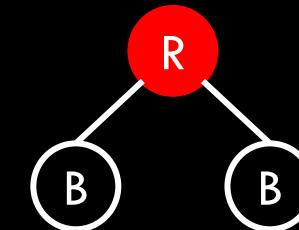
Insert 6



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

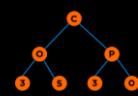


After Color Flip (P, GP)



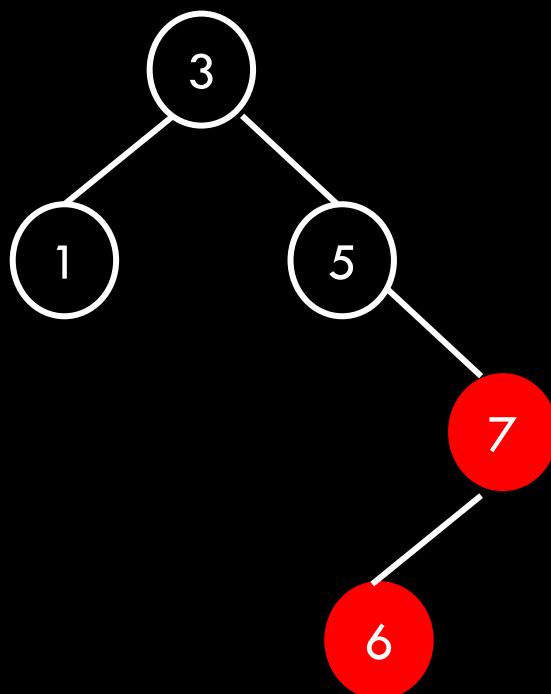
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

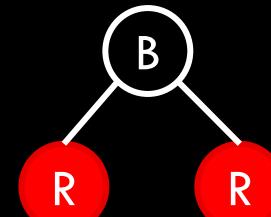


# Example

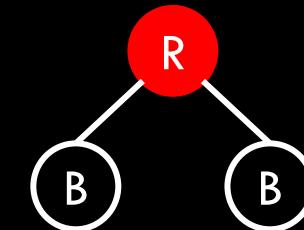
Insert 6



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

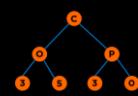


After Color Flip (P, GP)



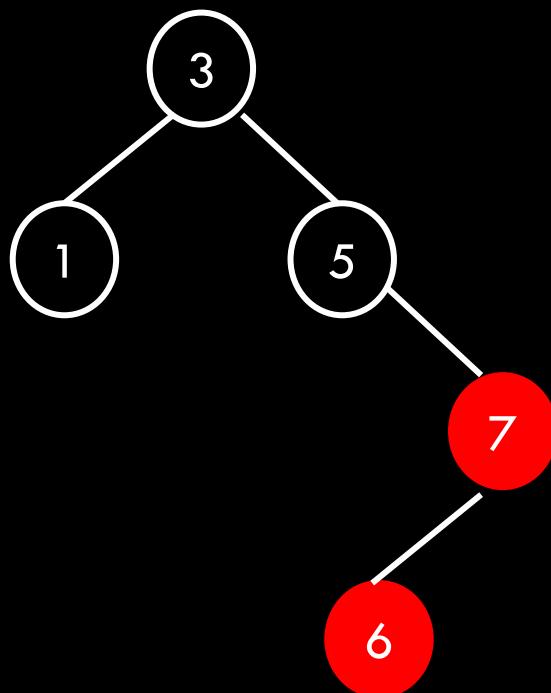
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

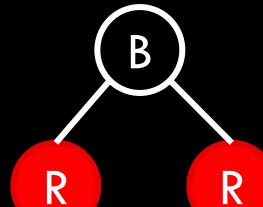
Insert 6



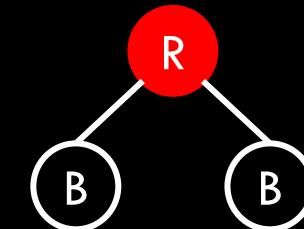
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation



- After Color Flip (P, GP)



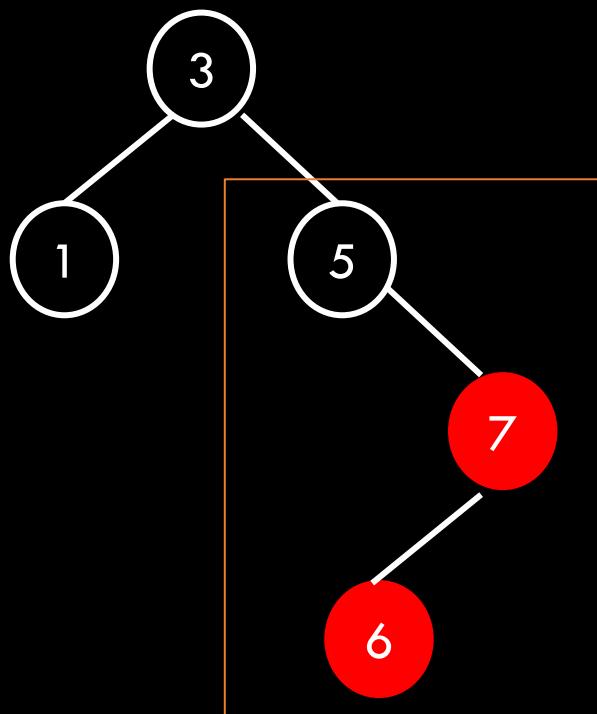
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

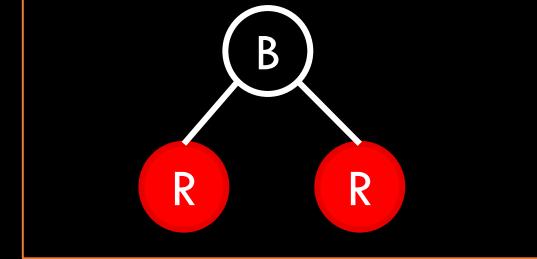
Insert 6



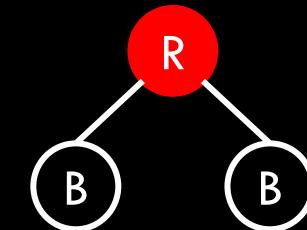
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

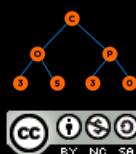


- After Color Flip (P, GP)



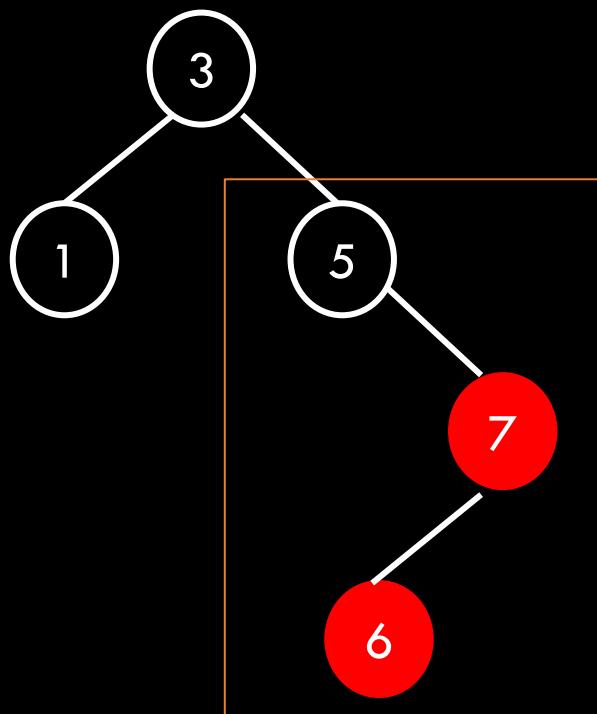
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

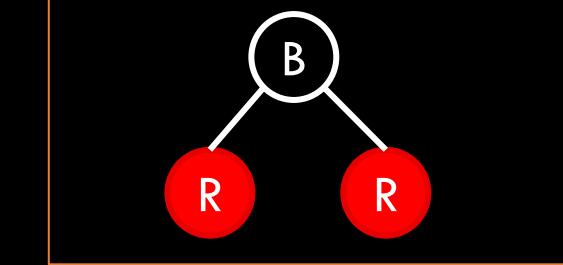
Insert 6



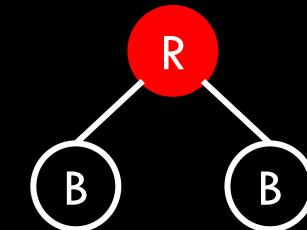
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

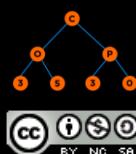


- After Color Flip (P, GP)



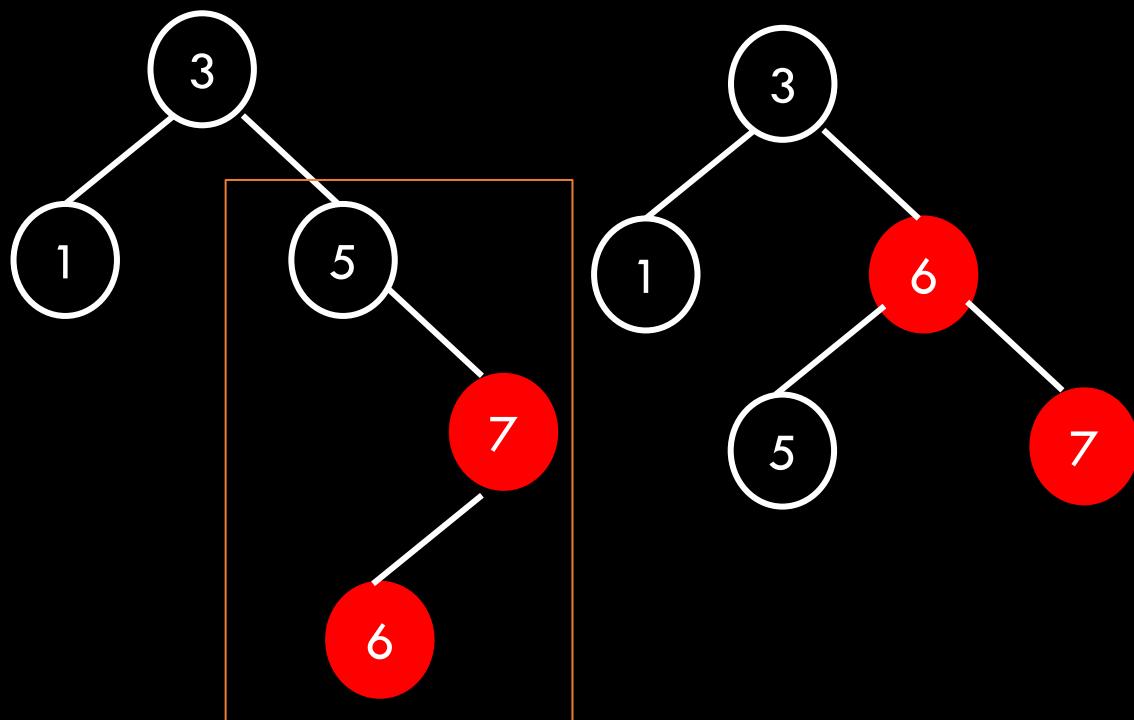
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

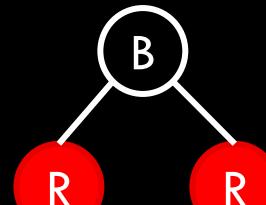
Insert 6



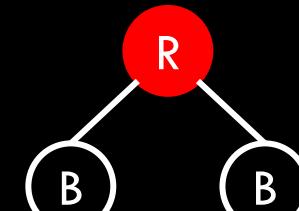
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

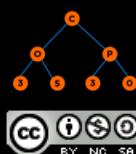


- After Color Flip (P, GP)



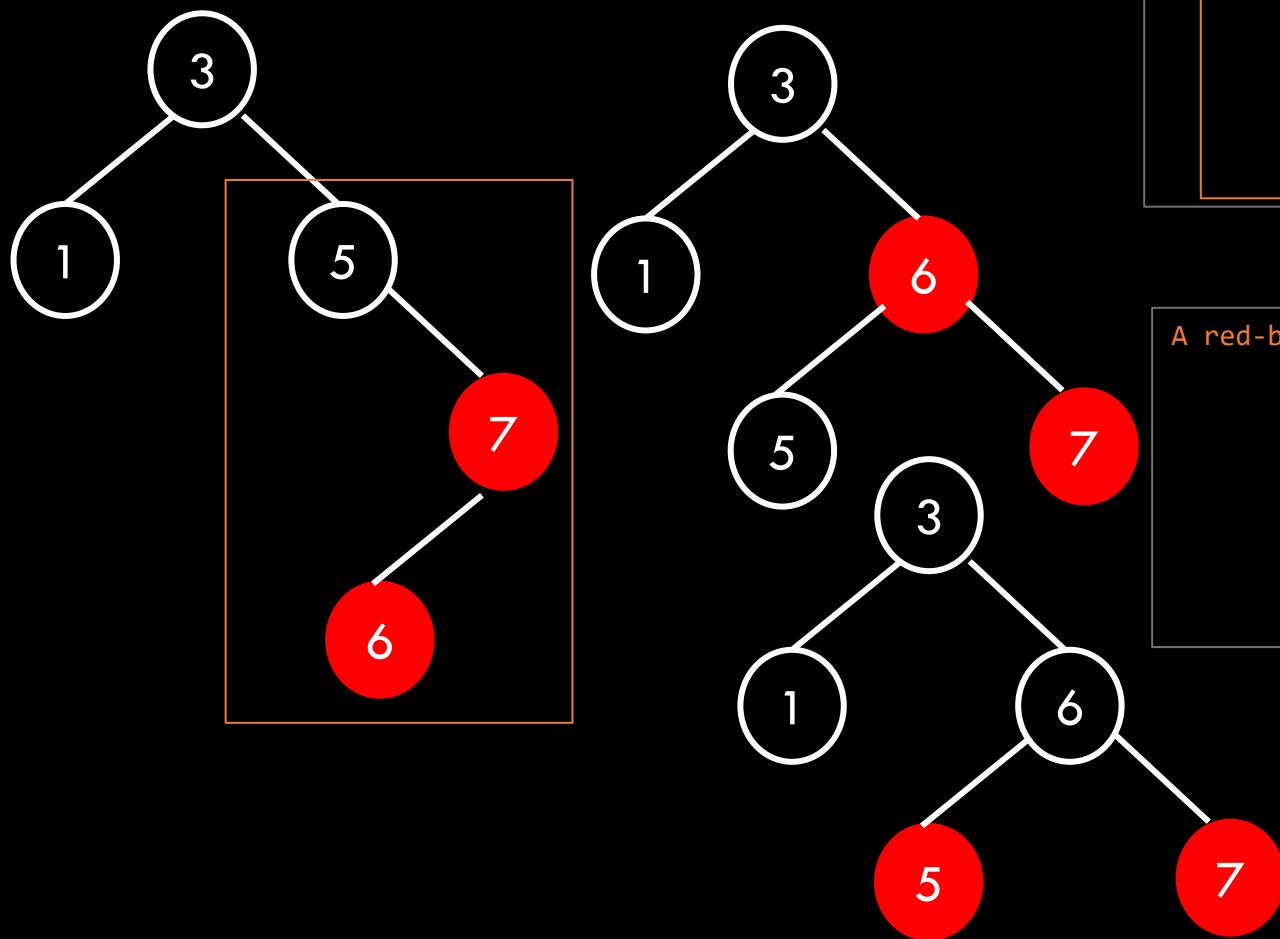
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

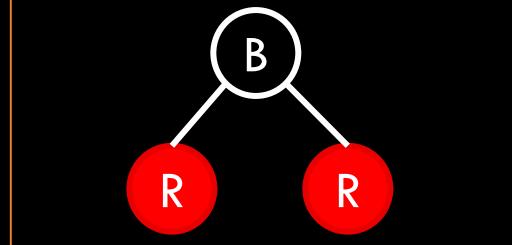
Insert 6



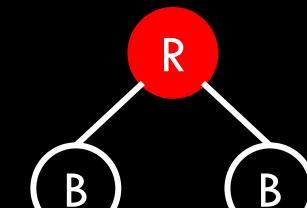
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

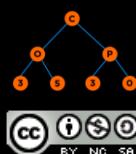


- After Color Flip (P, GP)



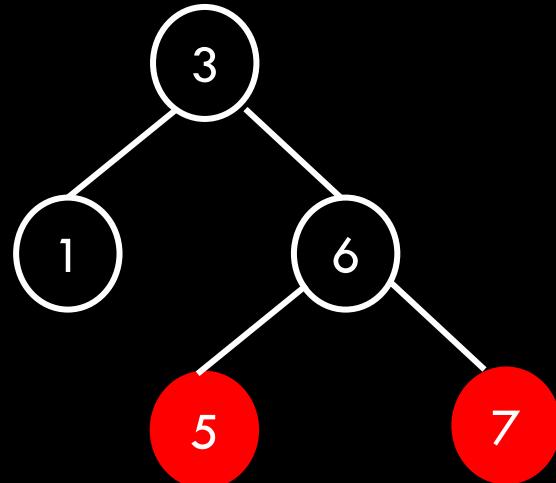
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

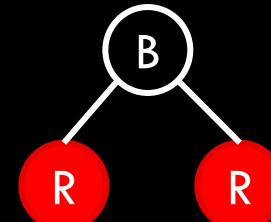


# Example

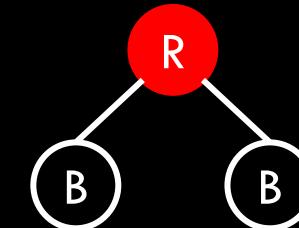
Insert 8



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

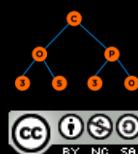


After Color Flip (P, GP)



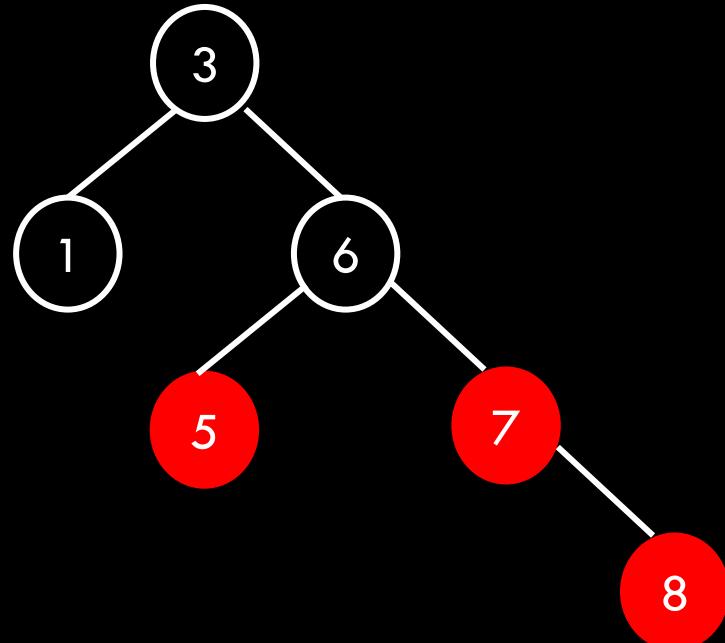
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

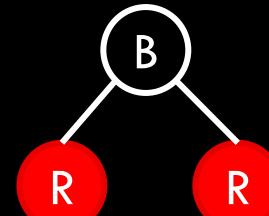


# Example

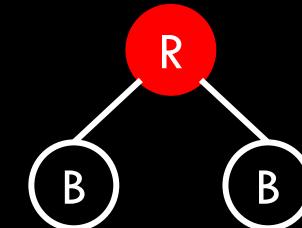
Insert 8



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)



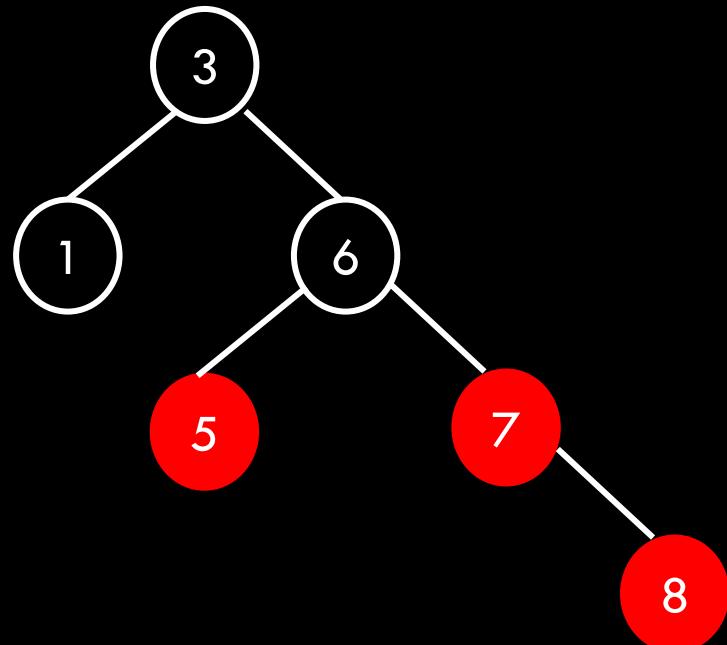
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

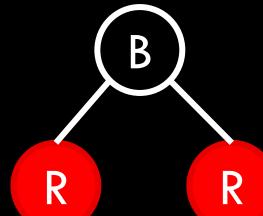


# Example

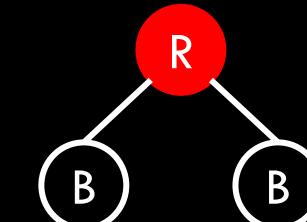
Insert 8



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

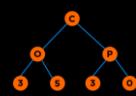


After Color Flip (P, GP)



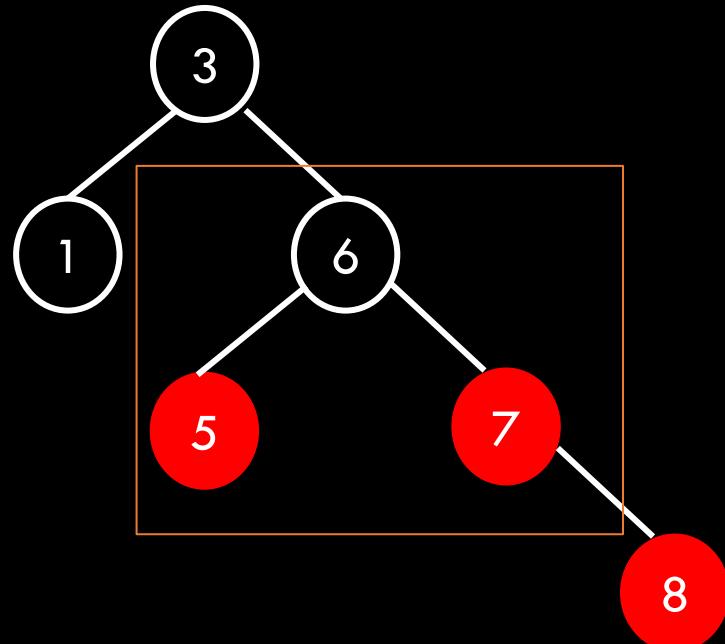
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

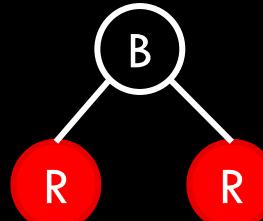


# Example

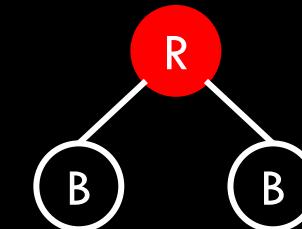
Insert 8



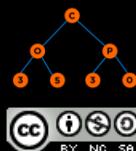
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

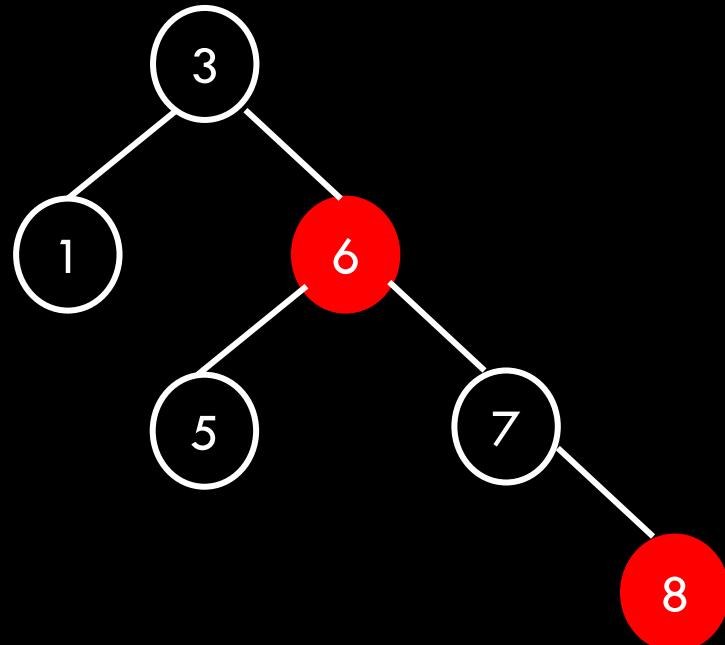


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

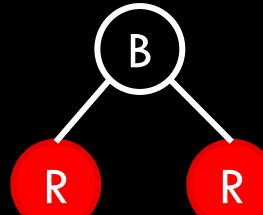


# Example

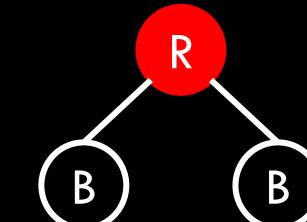
Insert 8



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

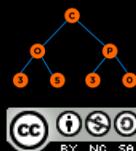


After Color Flip (P, GP)



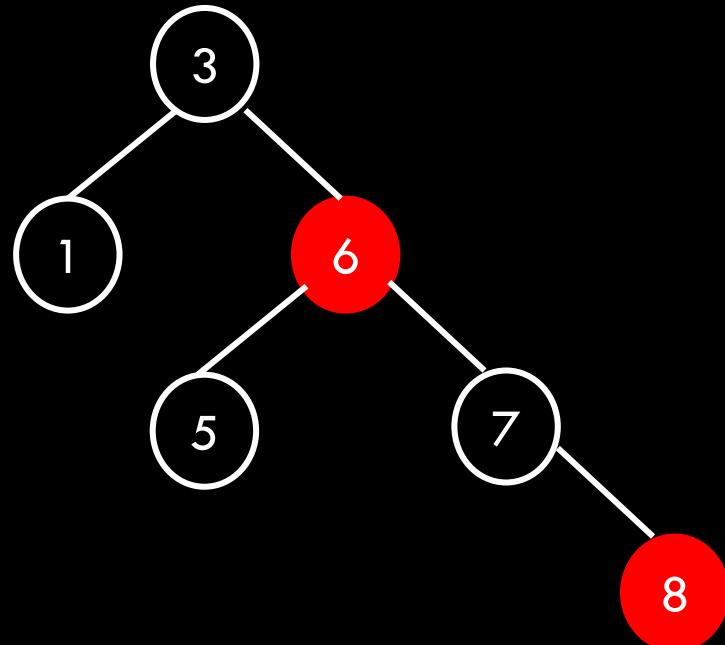
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

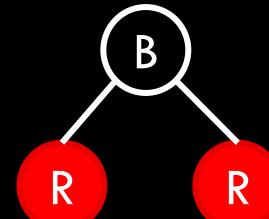


# Example

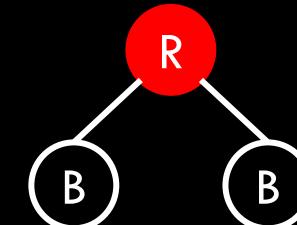
Insert 9



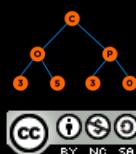
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

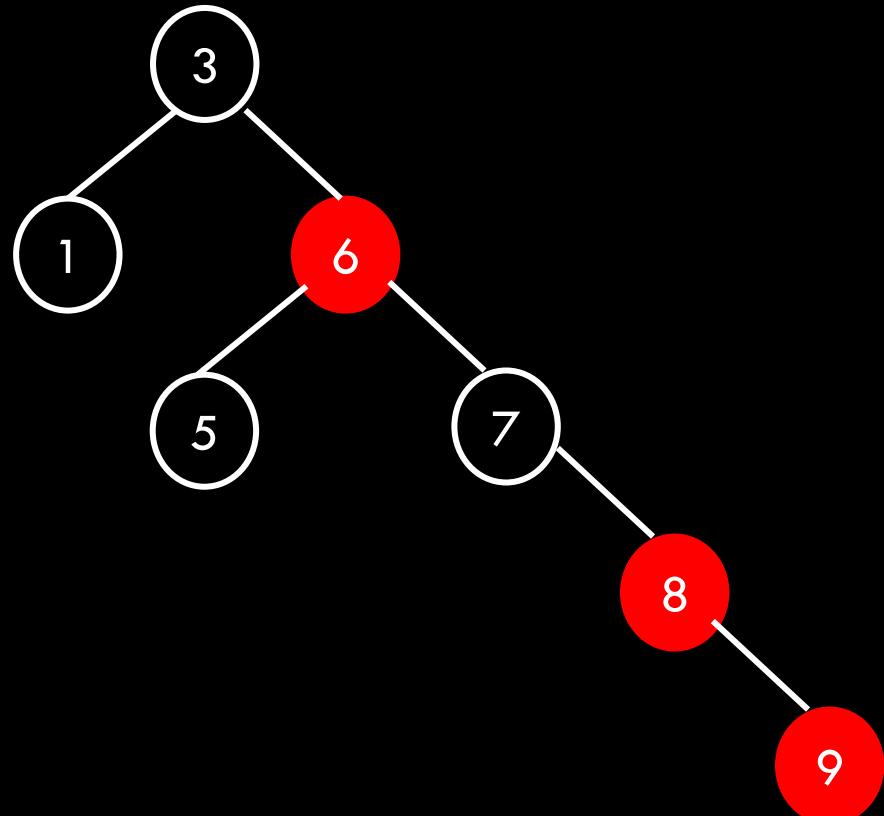


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

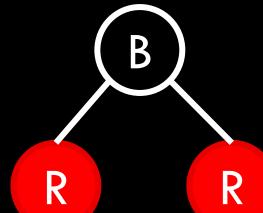


# Example

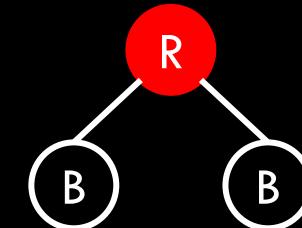
Insert 9



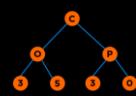
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

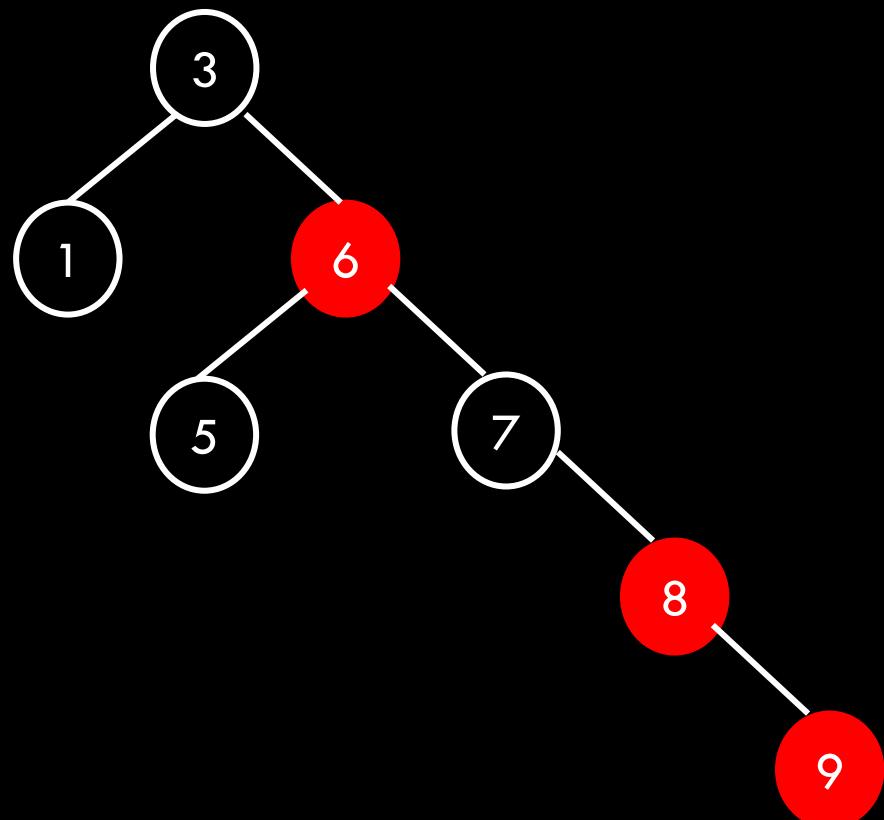


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black



# Example

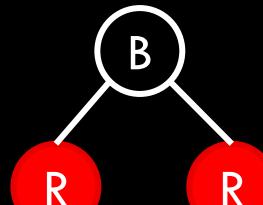
Insert 9



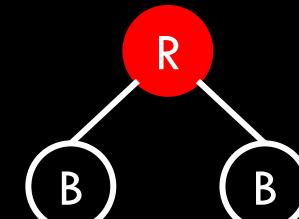
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

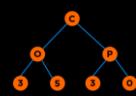


- After Color Flip (P, GP)



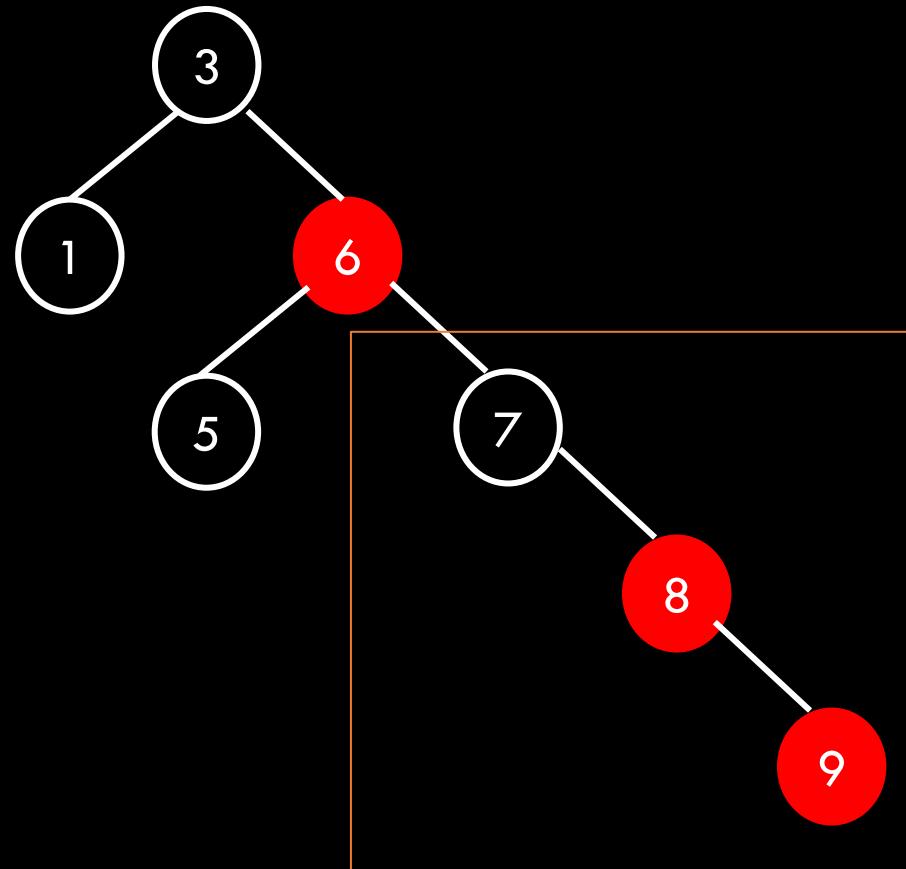
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

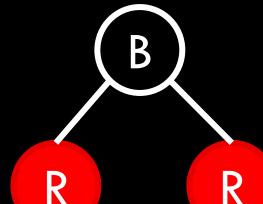
Insert 9



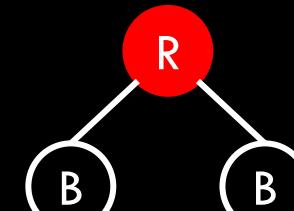
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

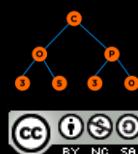


- After Color Flip (P, GP)



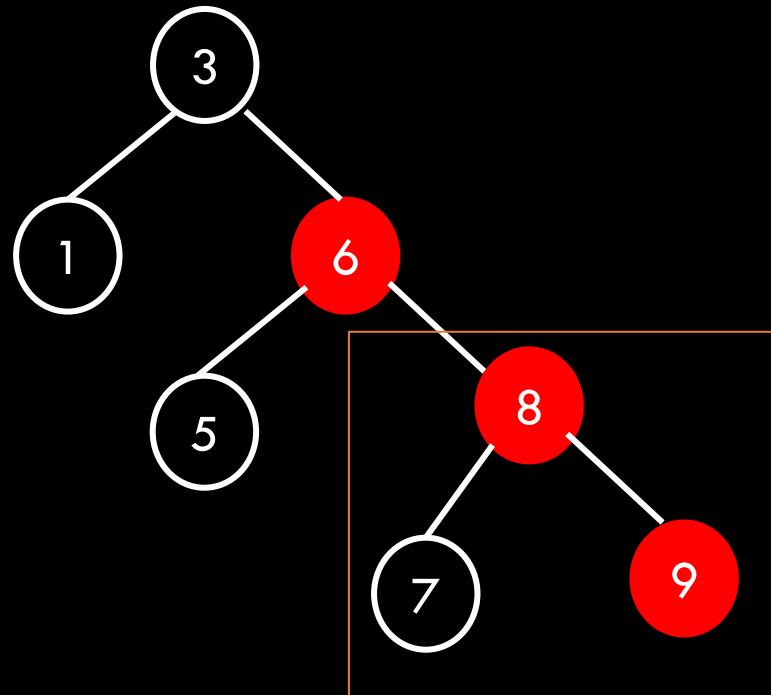
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

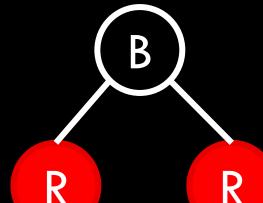
Insert 9



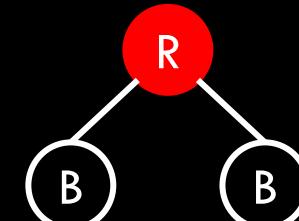
- If the uncle is red, flip colors

If the uncle is black, rotate

- After Rotation

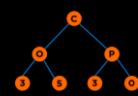


After Color Flip (P, GP)



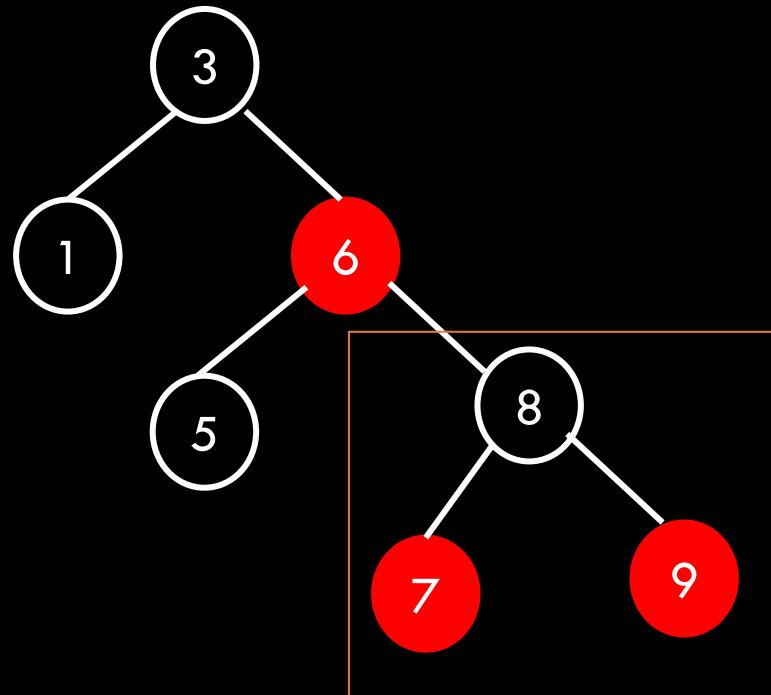
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

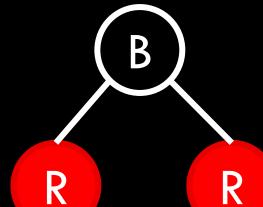
Insert 9



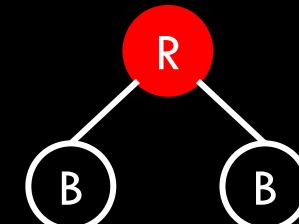
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

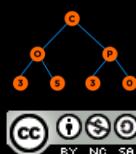


- After Color Flip (P, GP)



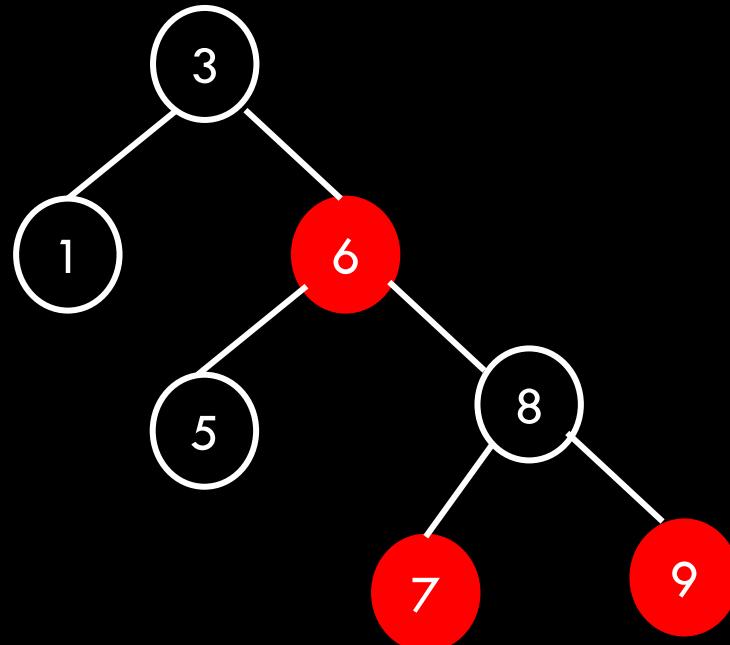
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

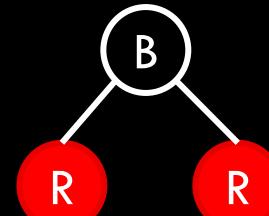


# Example

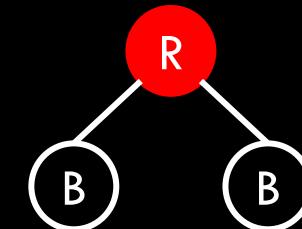
Insert 9



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

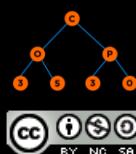


After Color Flip (P, GP)



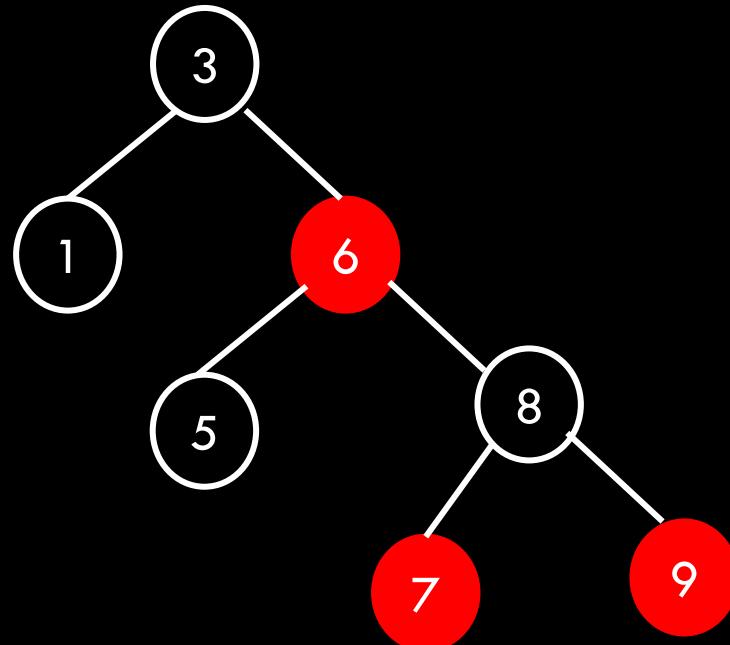
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

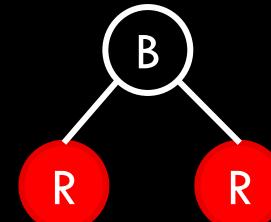


# Example

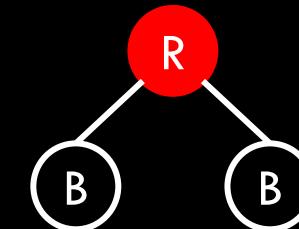
Insert 10



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation

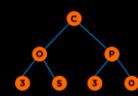


After Color Flip (P, GP)



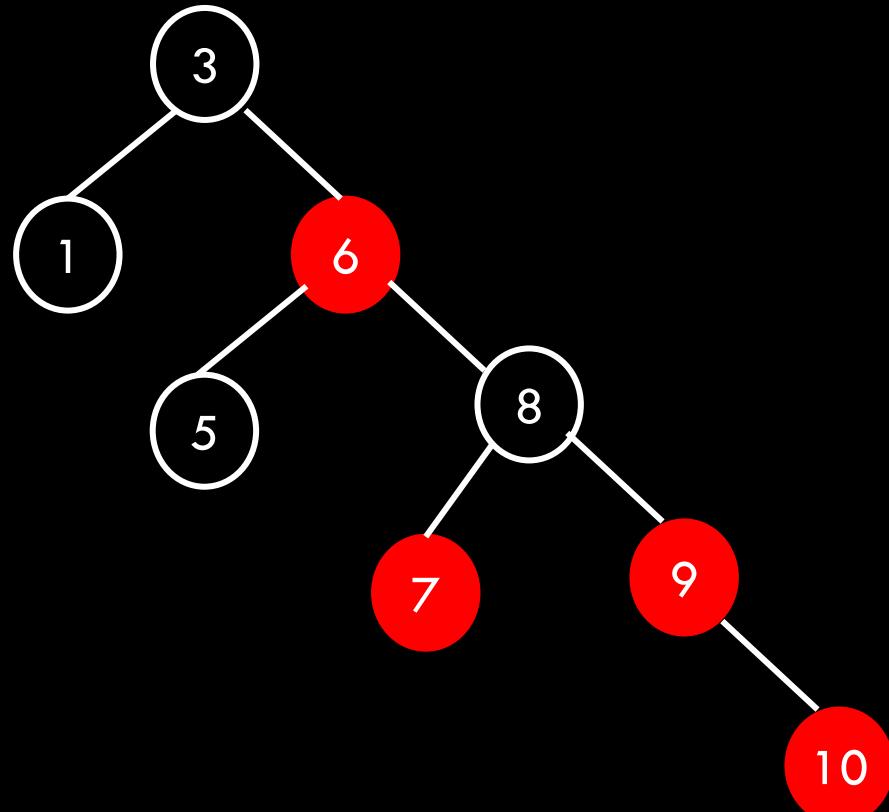
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black

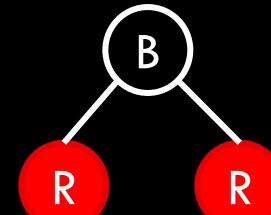


# Example

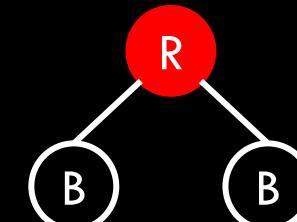
Insert 10



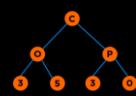
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

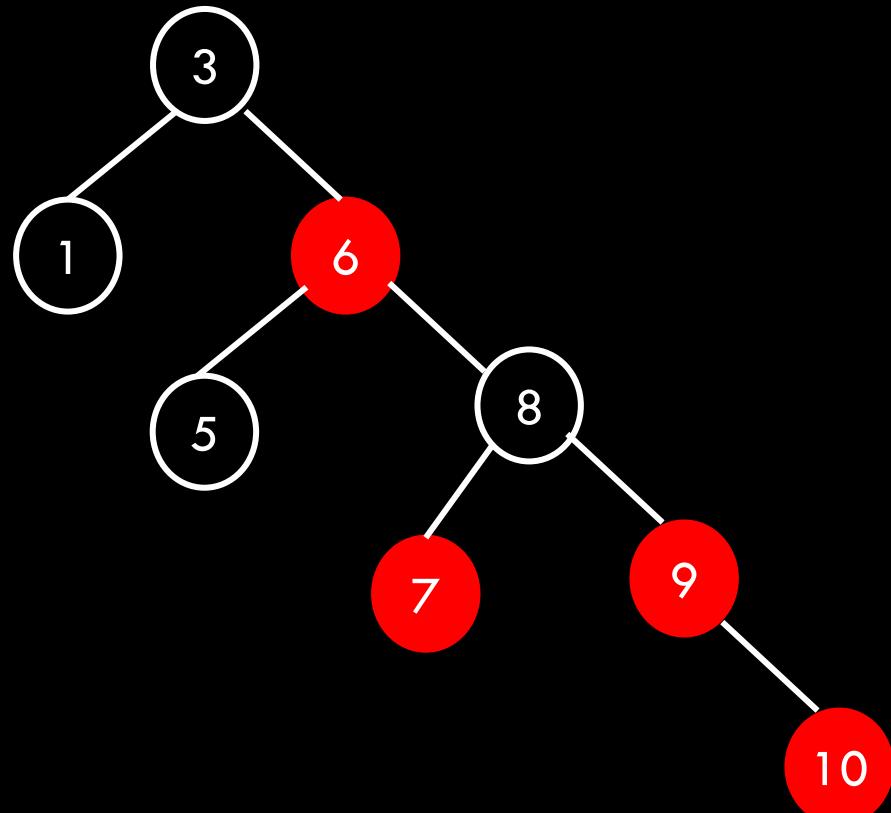


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

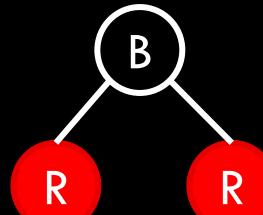


# Example

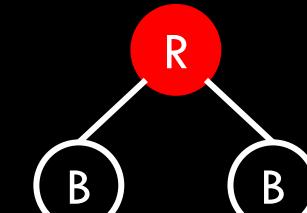
Insert 10



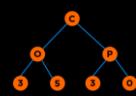
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

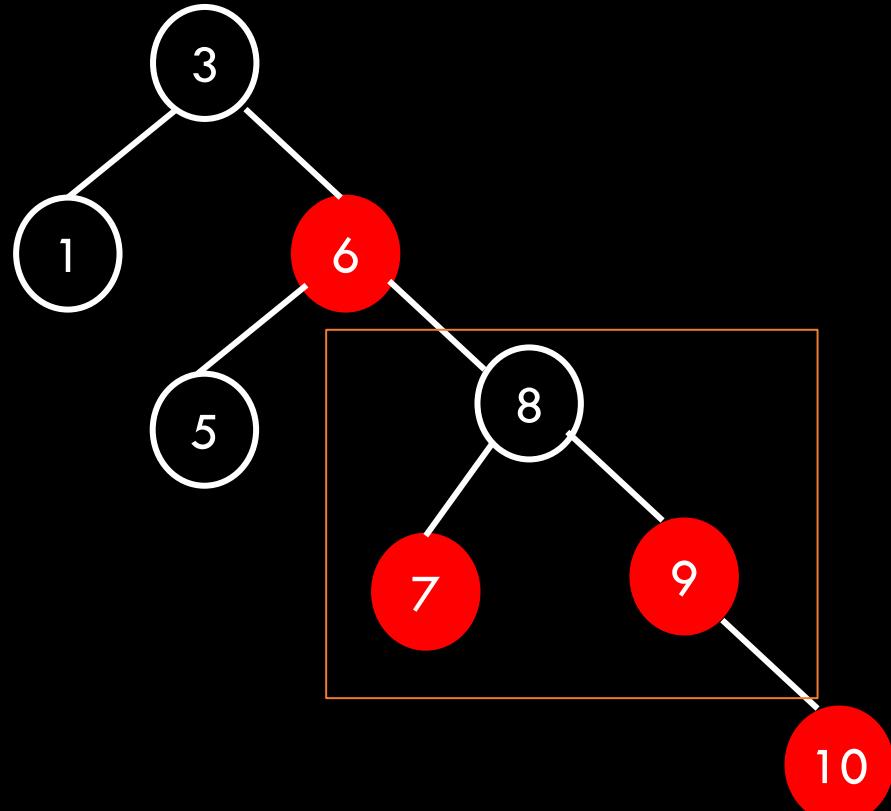


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

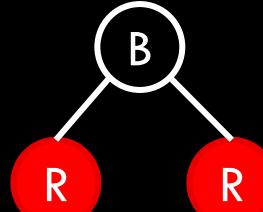


# Example

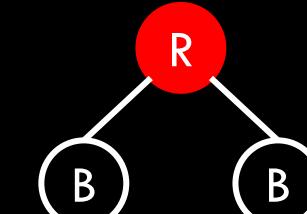
Insert 10



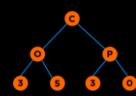
- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

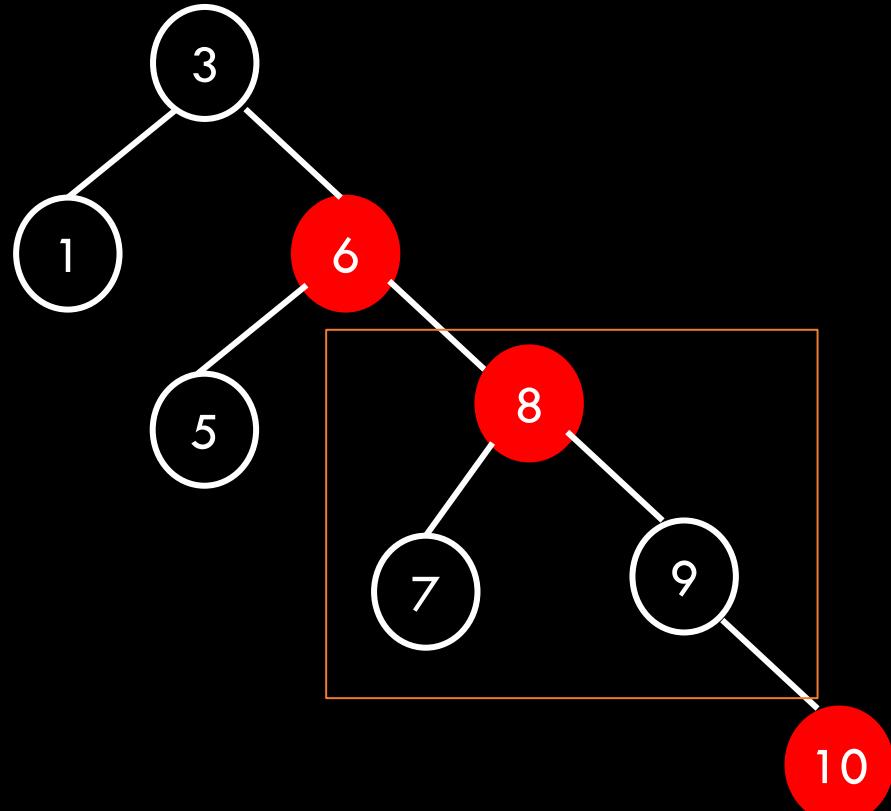


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

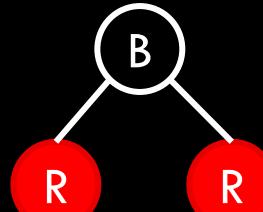


# Example

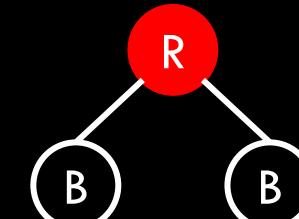
Insert 10



- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



After Color Flip (P, GP)

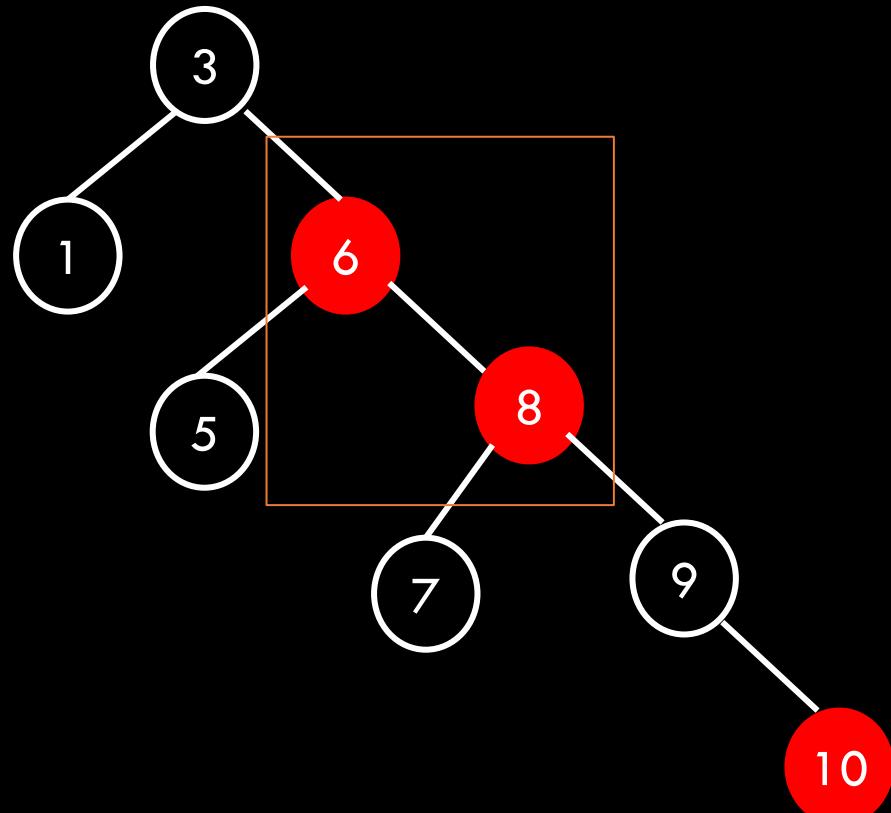


- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black

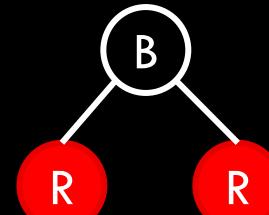


# Example

Insert 10

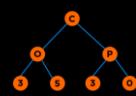


- If the uncle is red, flip colors
- If the uncle is black, rotate
- After Rotation



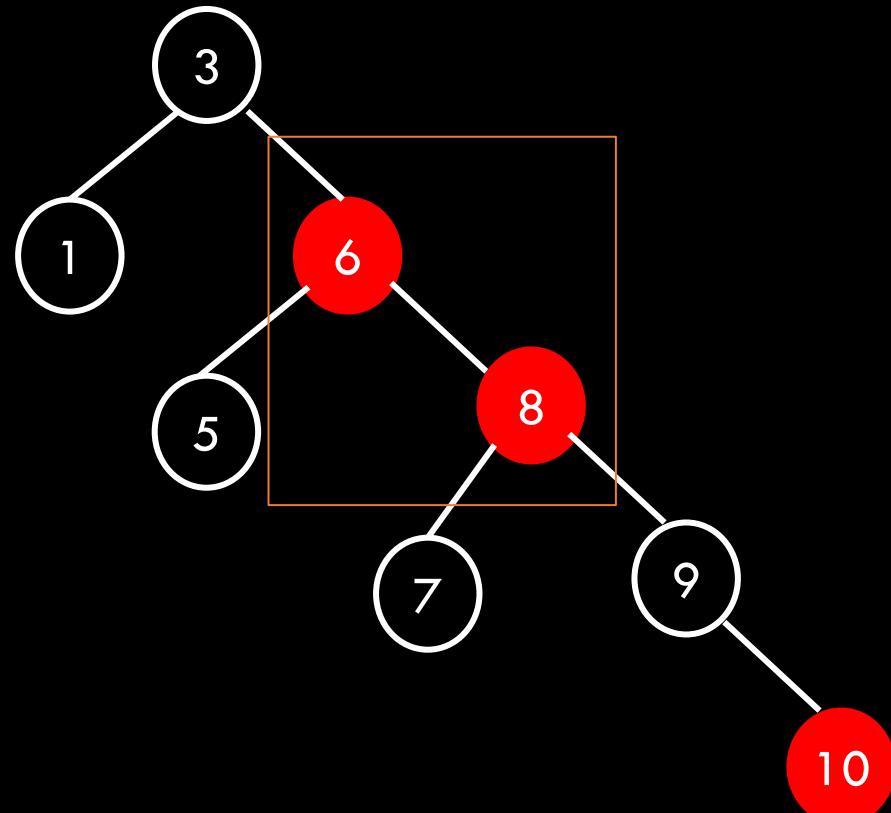
After Color Flip (P, GP)

- A red-black tree maintains the following invariants:
1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
  4. The number of black nodes in any path from the root to a leaf is the same
  5. Null nodes are attached to the leaves and are black



# Example

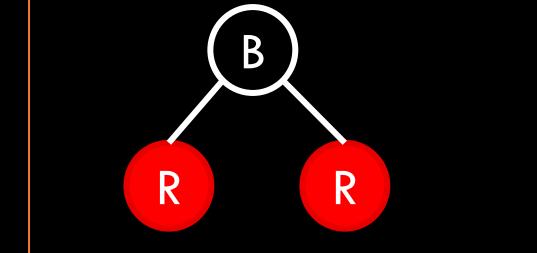
Insert 10



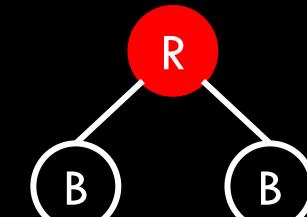
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation



- After Color Flip (P, GP)



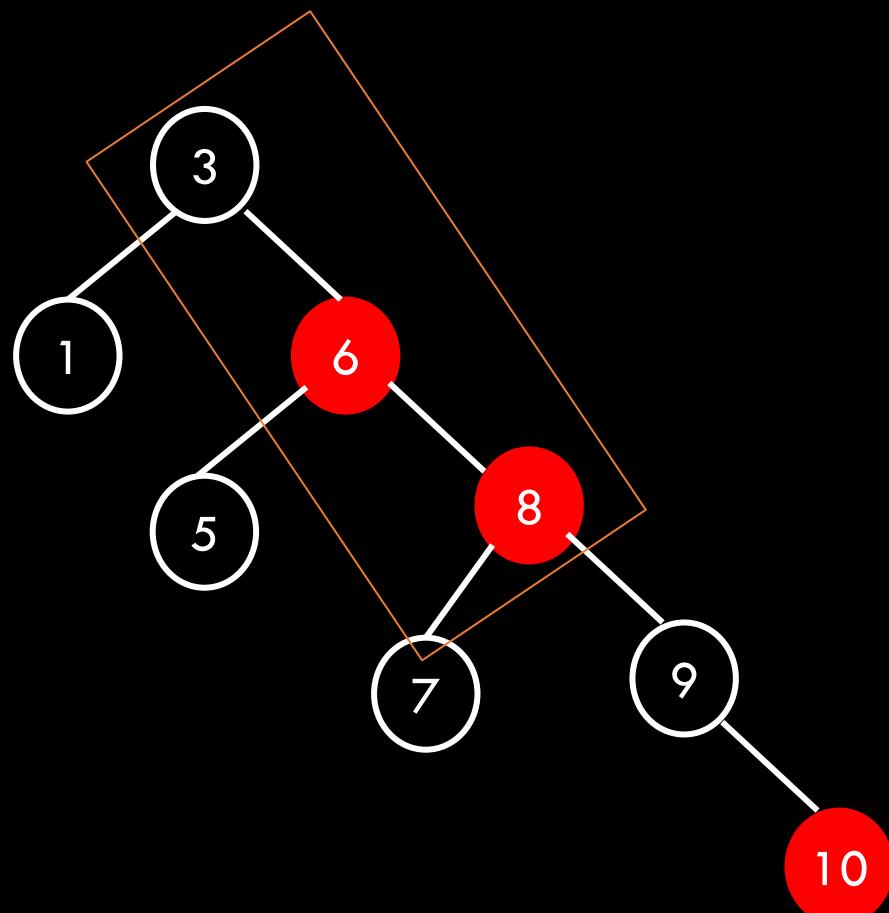
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

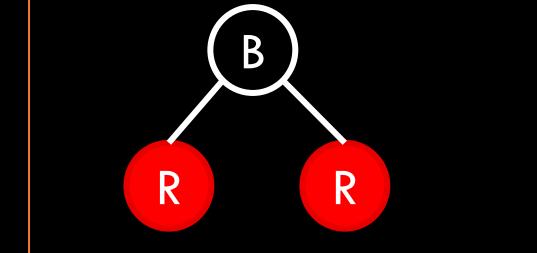
Insert 10



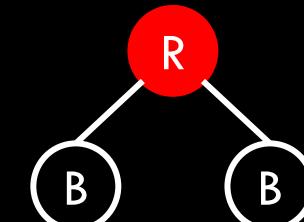
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation



- After Color Flip (P, GP)



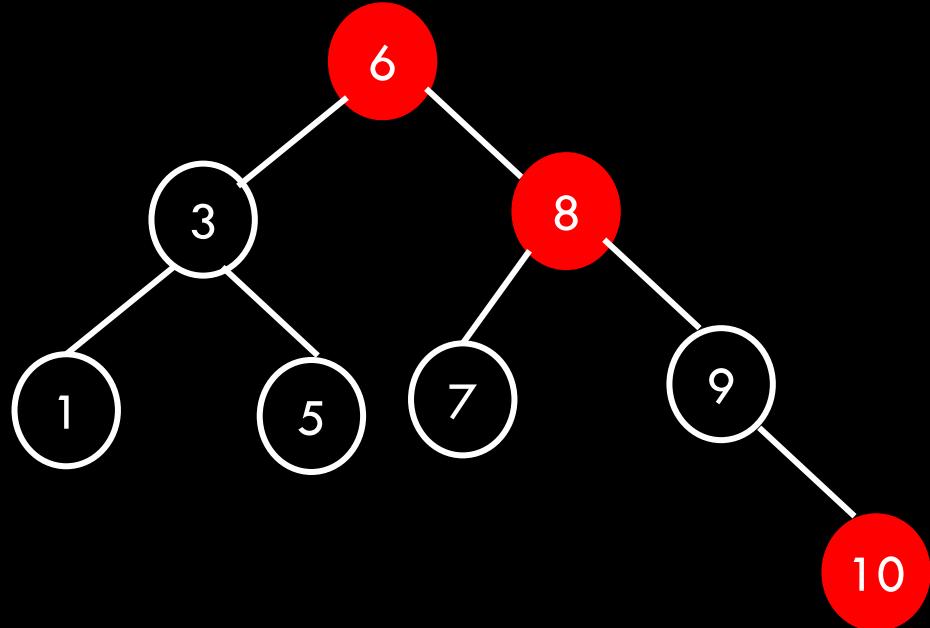
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

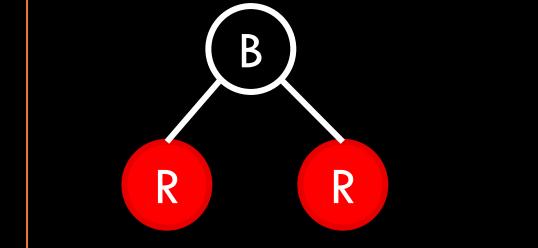
Insert 10



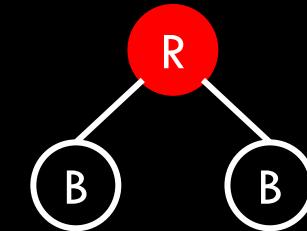
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

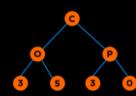


- After Color Flip (P, GP)



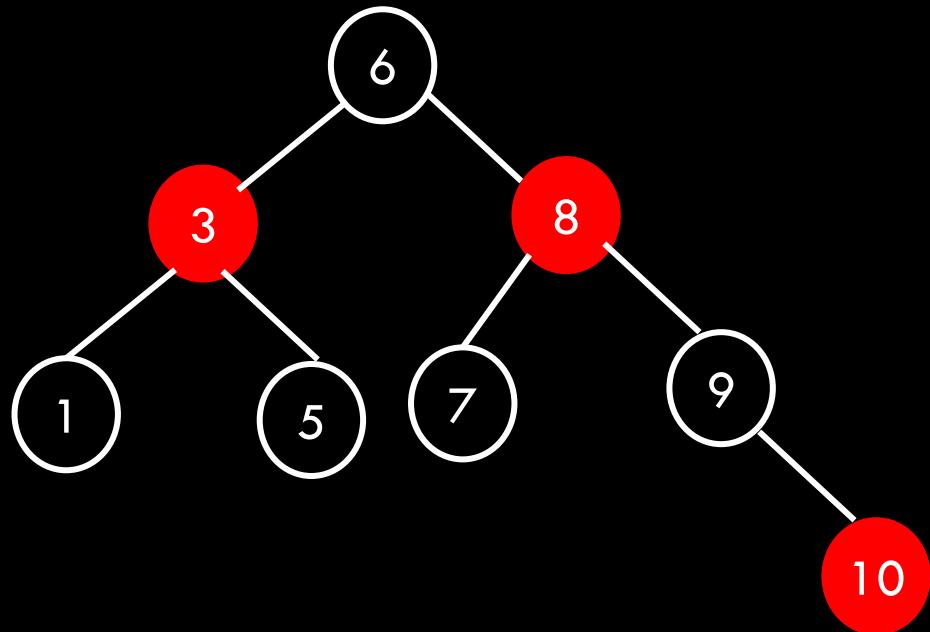
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Example

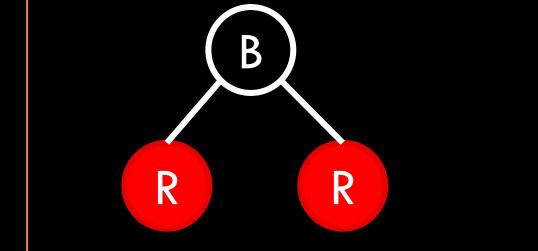
Insert 10



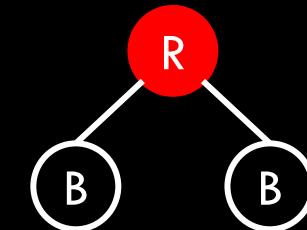
- If the uncle is red, flip colors

- If the uncle is black, rotate

- After Rotation

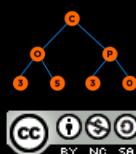


- After Color Flip (P, GP)



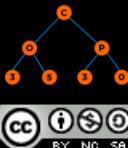
A red-black tree maintains the following invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a null reference is considered to refer to a black node) Or No two consecutive Red nodes
4. The number of black nodes in any path from the root to a leaf is the same
5. Null nodes are attached to the leaves and are black



# Red Black Tree Insertion

```
RBTreeBalance(tree, node)
{
    if (node->parent == null)
    {
        node->color = black
        return
    }
    if (node->parent->color == black)
        return
    parent = node->parent
    grandparent = RBTreeGetGrandparent(node)
    uncle = RBTreeGetUncle(node)
    if (uncle != null && uncle->color == red)
    {
        parent->color = uncle->color = black
        grandparent->color = red
        RBTreeBalance(tree, grandparent)
        return
    }
    if (node == parent->right && parent == grandparent->left)
    {
        RBTreeRotateLeft(tree, parent)
        node = parent
        parent = node->parent
    }
    else if (node == parent->left && parent == grandparent->right)
    {
        RBTreeRotateRight(tree, parent)
        node = parent
        parent = node->parent
    }
    parent->color = black
    grandparent->color = red
    if (node == parent->left)
        RBTreeRotateRight(tree, grandparent)
    else
        RBTreeRotateLeft(tree, grandparent) }
```



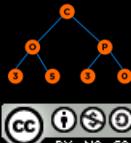
# Red Black Tree Insertion

```
1. Search (top-down) and insert the new item u as in a Binary Search Tree.  
2. Return (bottom-up) and  
2.1 If u is root, make it black and the algorithm ends or  
2.2 if its parent t is black, the algorithm ends  
2.3 If both u and its parent t are red, do one of the following:  
2.3.1. [change colors] If t and its sibling v are red:  
    Color t and v black and their parent p red.  
    Continue the algorithm with p if necessary.  
2.3.2. [rotations] If t is red and v black, perform a rotation.  
    After the rotation, p and its new parent exchange their colors.  
    There are no longer two consecutive red nodes in the tree.
```

## ROTATION:

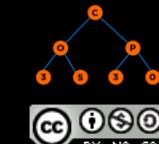
```
1 While the recursion returns, keep track of  
    node p,  
    p's child t and  
    p's grandchild u within the path from inserted node to p.  
2 If rotation is needed in p, do one of the following rotations:  
    if (p.left == t) and (p.left.left == u), single rotation right in p;  
    if (p.right == t) and (p.right.right == u), single rotation left in p;  
    if (p.left == t) and (p.left.right == u), LR-double rotation in p; or  
    if (p.right == t) and (p.right.left == u), RL-double rotation in p.
```

<https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/RedBlack.html>



# Use Case

- Tree Set, Tree Map, Hash Maps are backed up by a Red Black Tree
- C++ STL



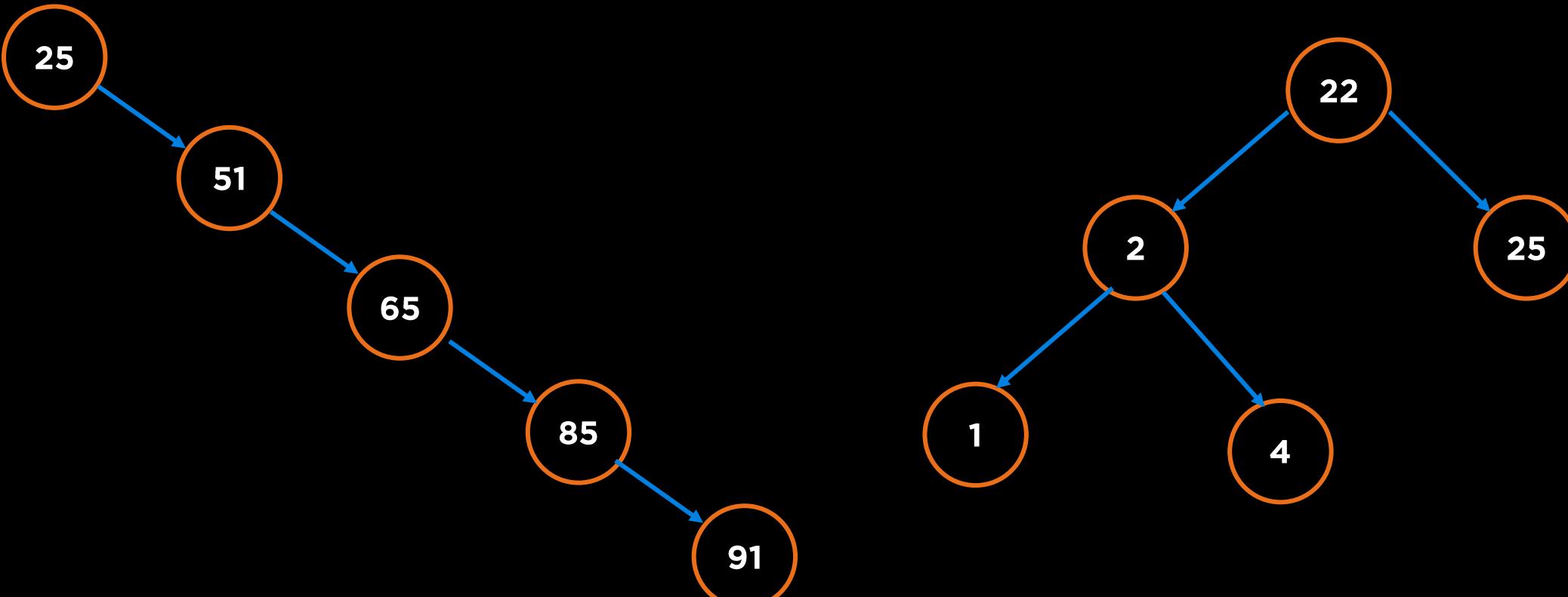
# Performance

	Average	Worst case
▪ <b>Space</b>	$O(n)$	$O(n)$
▪ <b>Search</b>	$O(\log n)$	$O(\log n)$
▪ <b>Insert</b>	$O(\log n)$	$O(\log n)$
▪ <b>Delete</b>	$O(\log n)$	$O(\log n)$

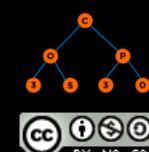
# Splay Tree



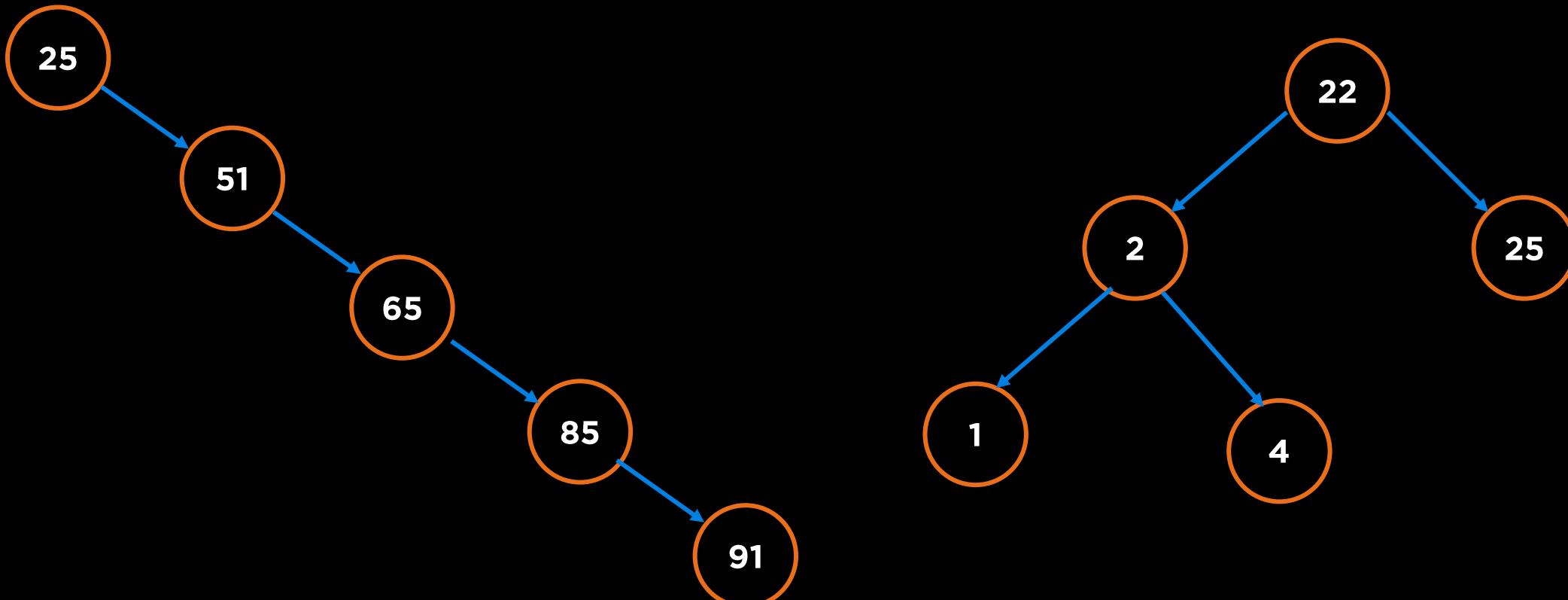
# Searching for Random Inputs



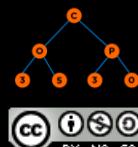
$O(n)$  or  $O(\log n)$  in BST or AVL Tree



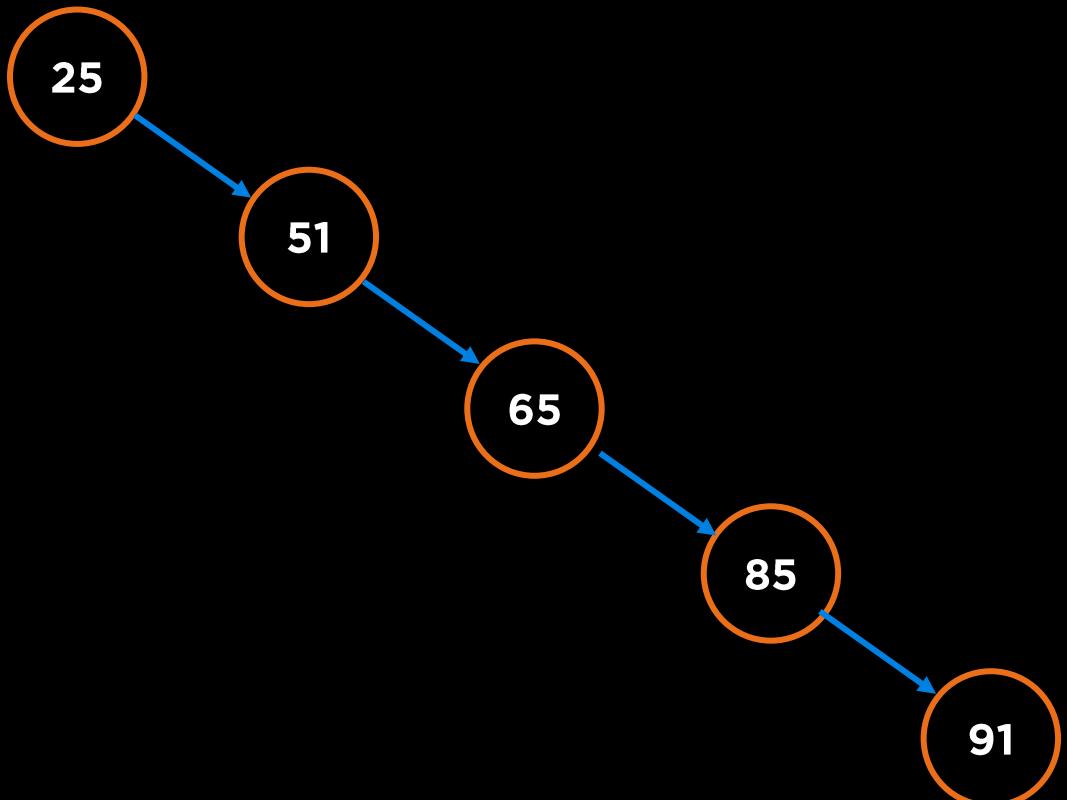
# Searching for Non-Random Inputs



$O(n)$  or  $O(\log n)$  in BST or AVL Tree  
What if the Input is non-Random?



# Enter Splay Tree

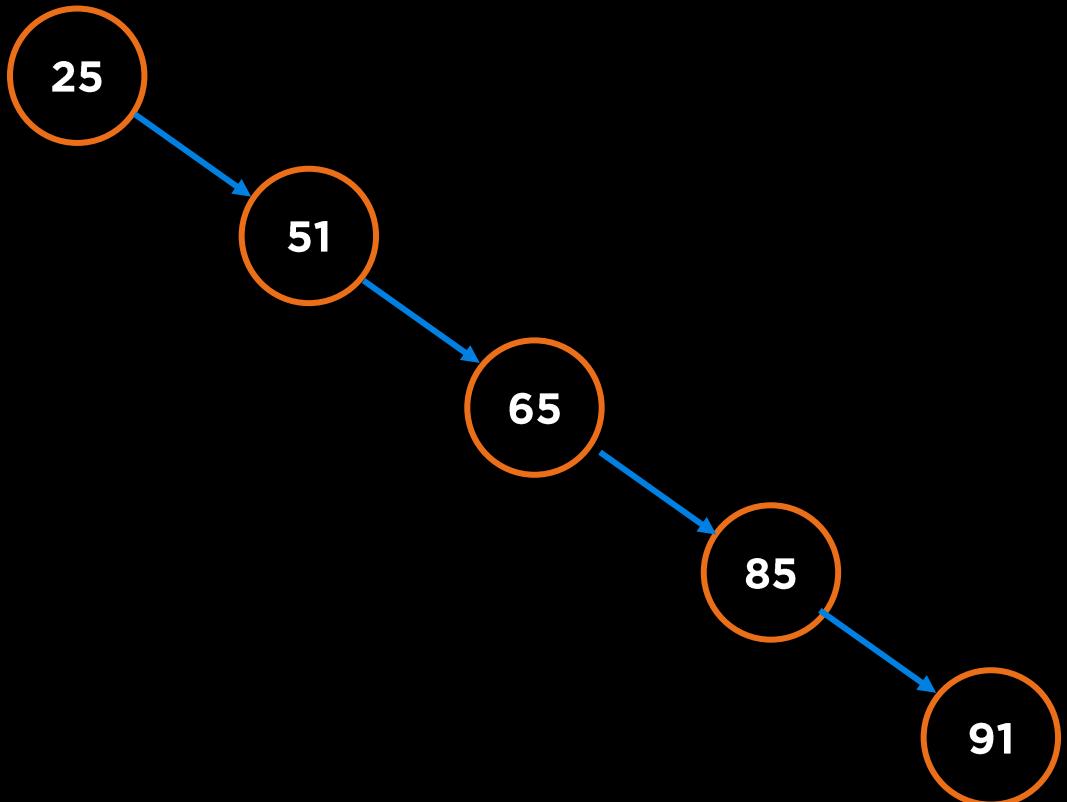


If we are searching 91 again and again,  
bring it closer to root!

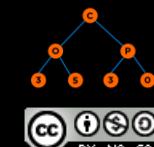
Simple Rotation won't work!

Special rotations involving  
grandparent, parent and child.

# Enter Splay Tree



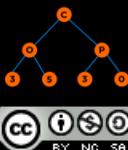
A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again.



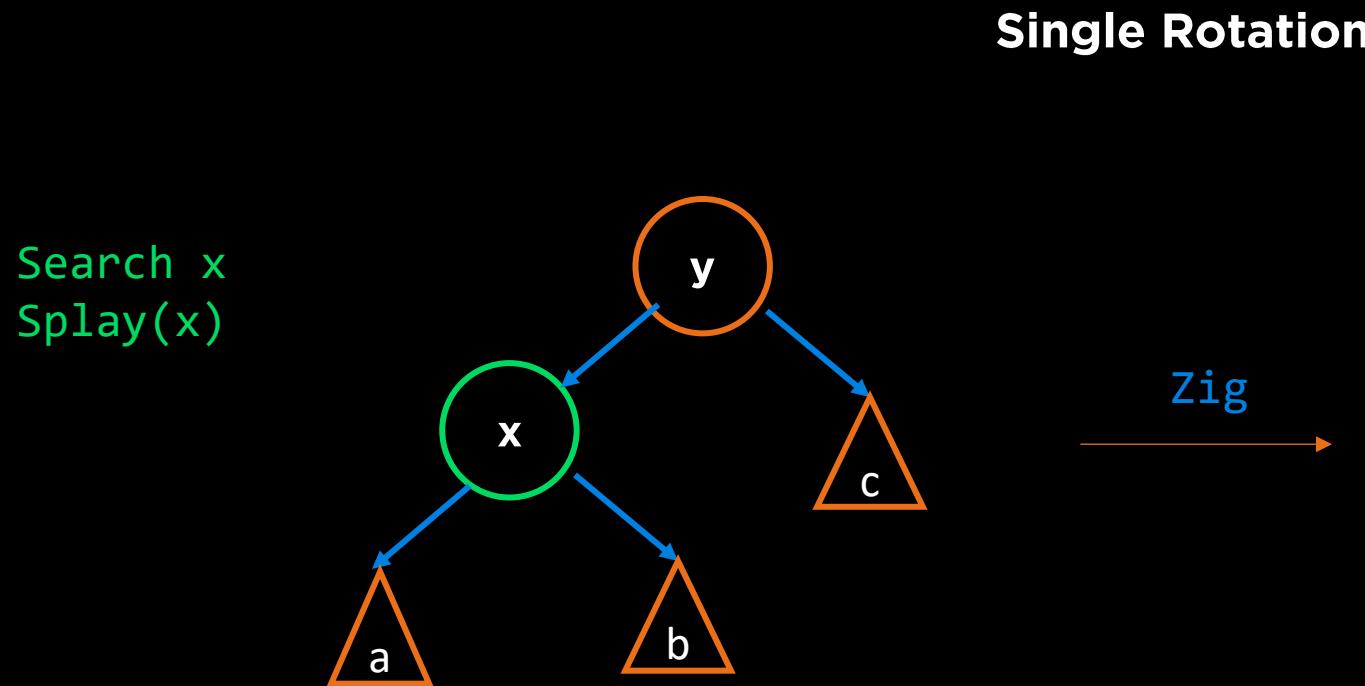
# Splay Tree

- A binary search tree with the additional property that recently accessed elements are quick to access again
- For many sequences of non-random operations, splay trees perform better than other search trees
- The splay tree was invented by Daniel Sleator and Robert Tarjan
- All normal operations on a binary search tree are combined with one basic operation, called **splaying**. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.

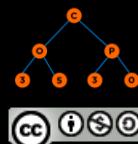
[https://en.wikipedia.org/wiki/Splay\\_tree](https://en.wikipedia.org/wiki/Splay_tree)



# Splay Tree: Zig Rotation



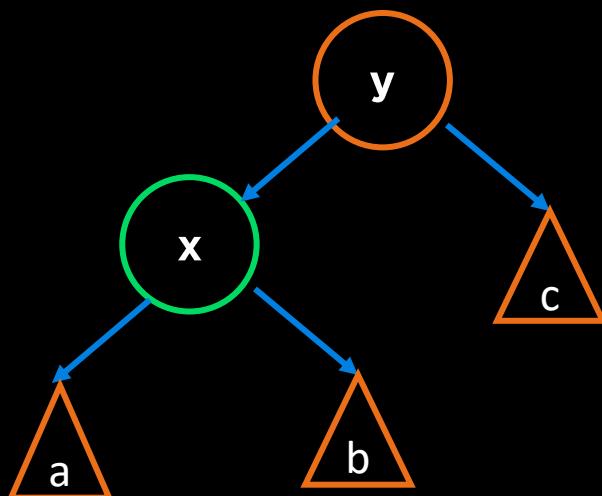
Zig = Right Rotation (Splay(x), x is left of parent and has no grandparent)



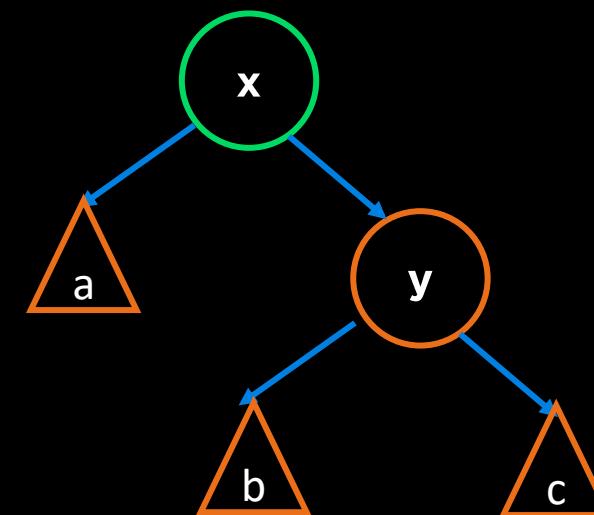
# Splay Tree: Zig Rotation

Single Rotation

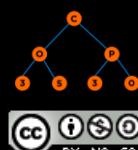
Search x  
Splay(x)



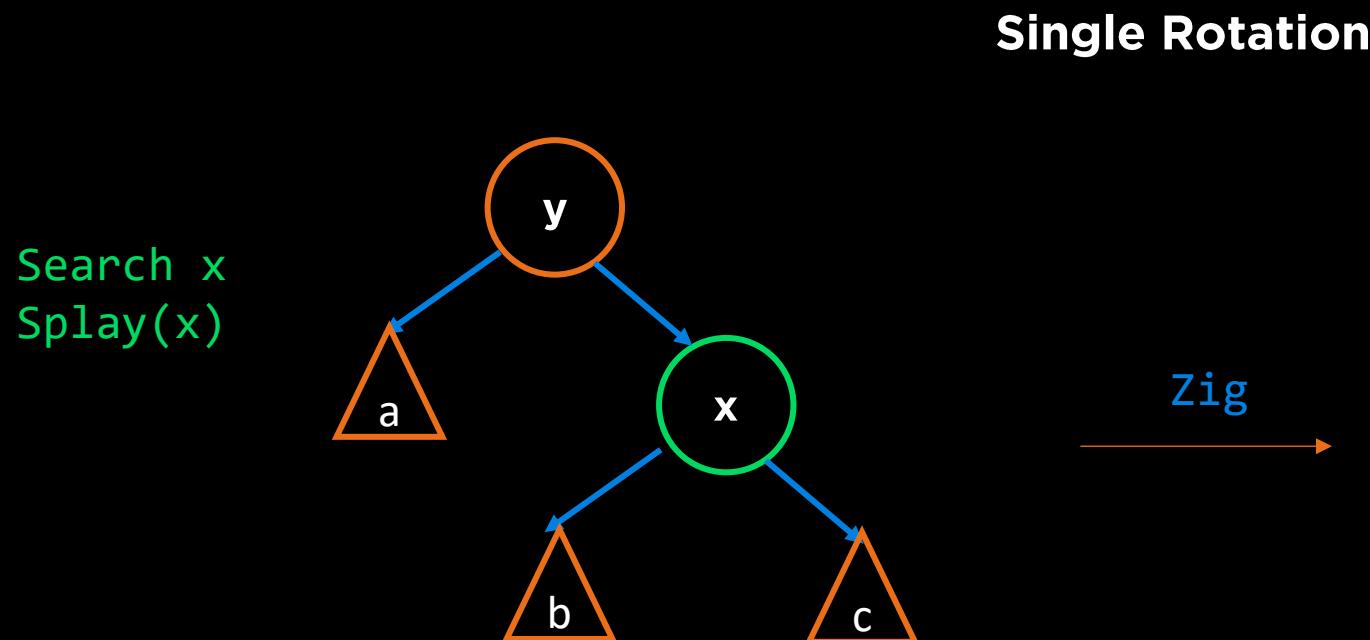
Zig



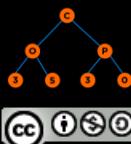
Zig = Right Rotation (Splay(x), x is left of parent and has no grandparent)



# Splay Tree: Zig Rotation

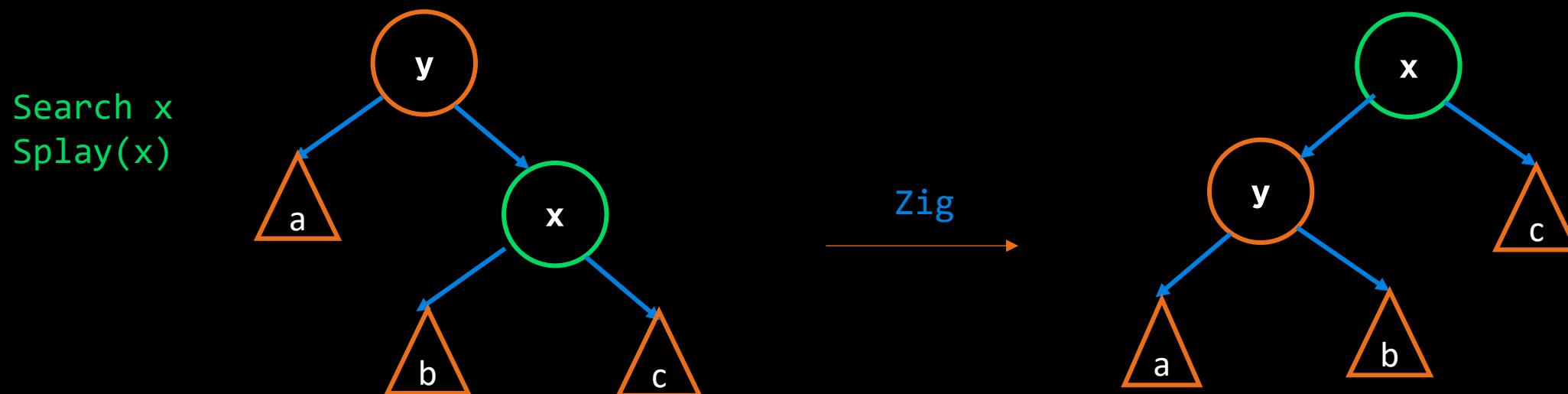


Zig = Left Rotation (Splay(x), x is right of parent and has no grandparent)

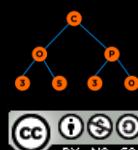


# Splay Tree: Zig Rotation

Single Rotation

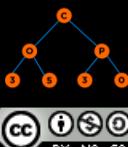
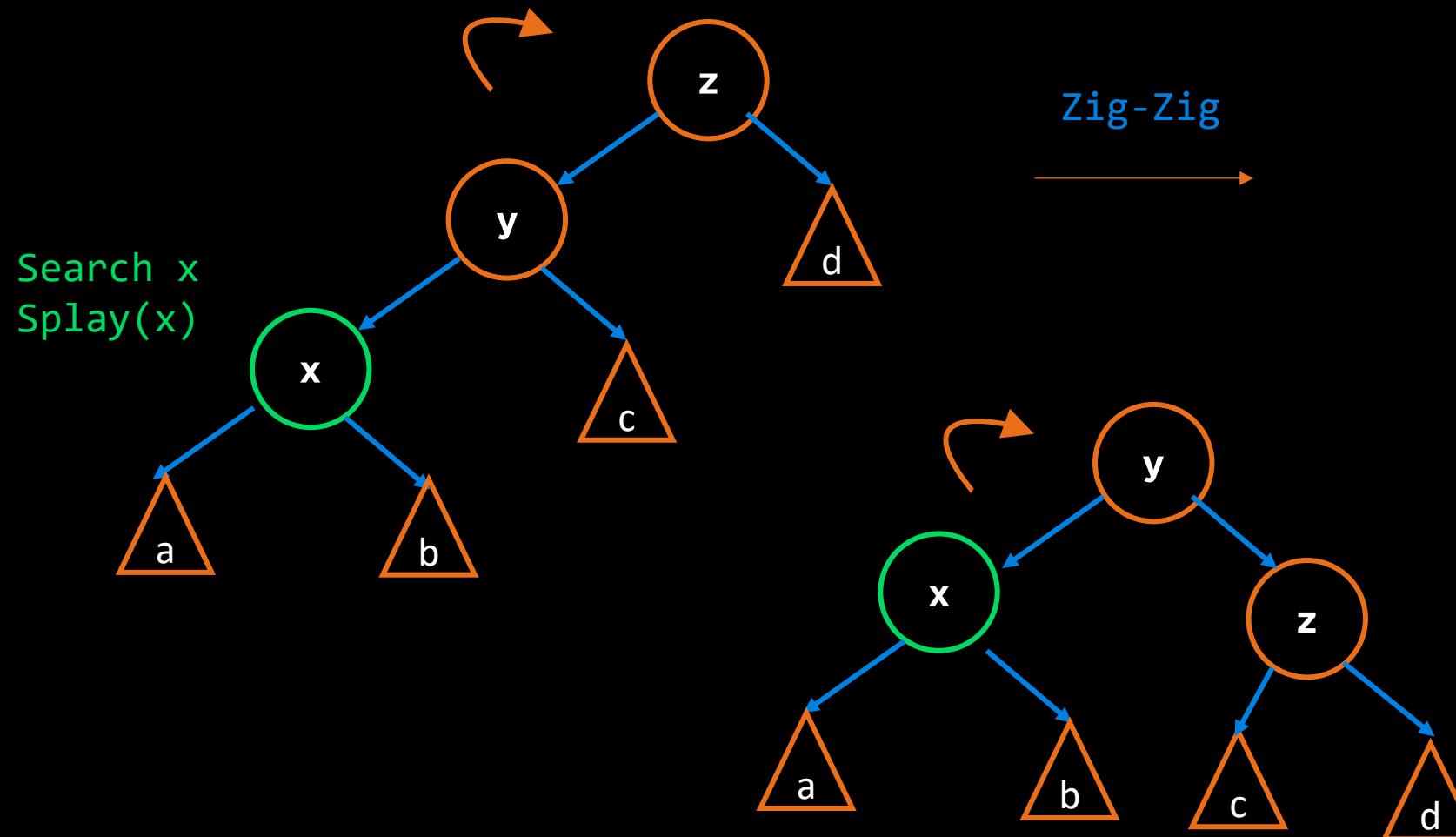


Zig = Left Rotation (Splay( $x$ ),  $x$  is right of parent and has no grandparent)

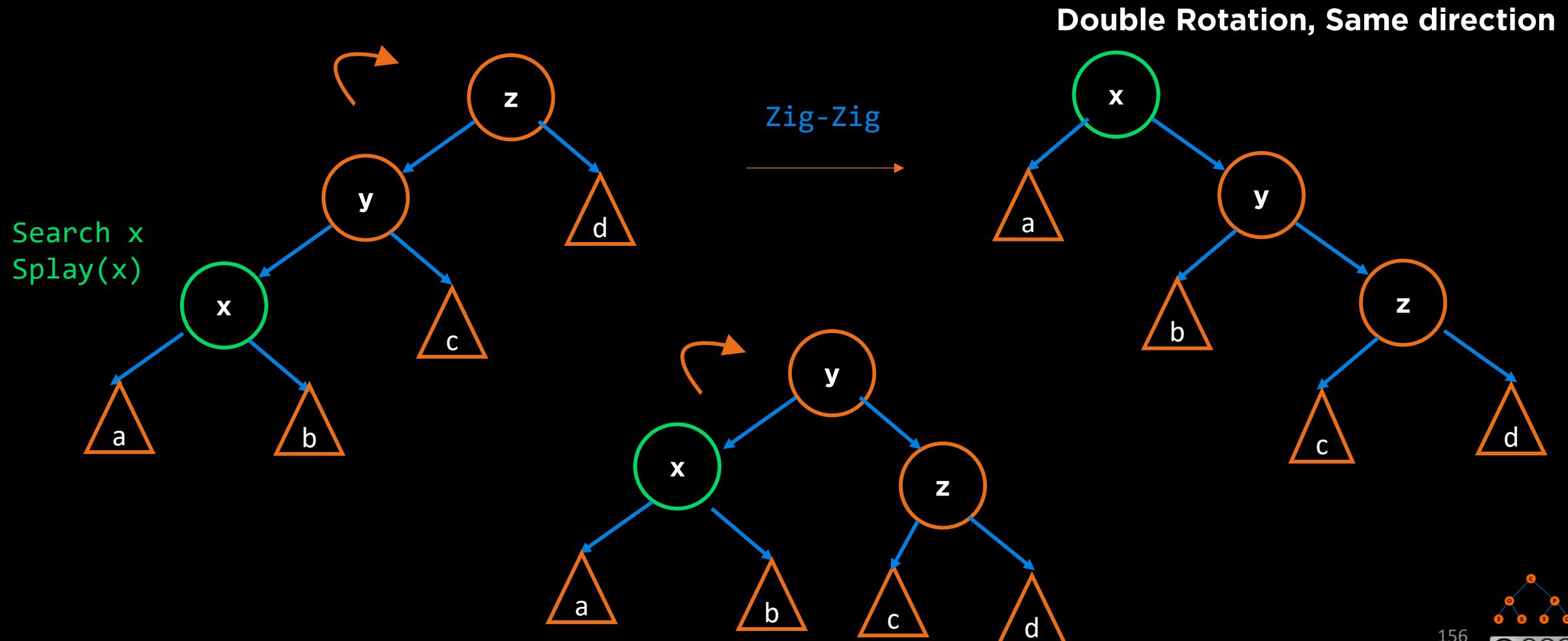


# Splay Tree: Zig Zig Rotation

Double Rotation, Same direction

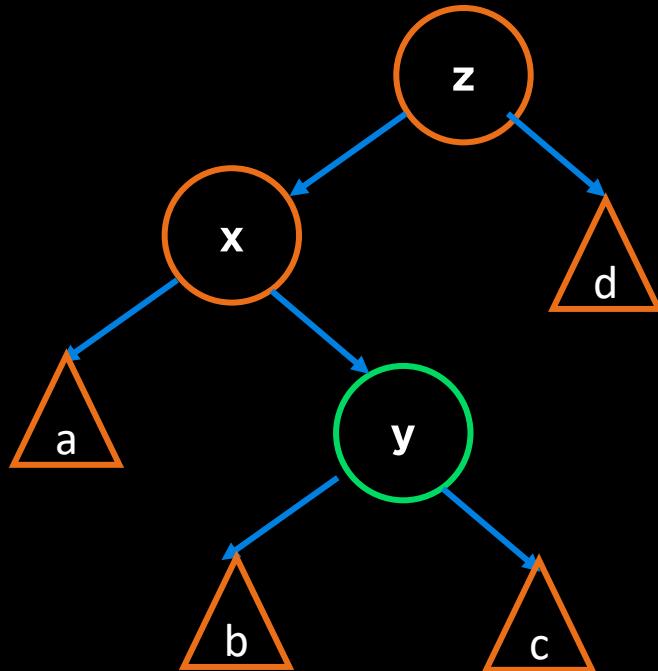


# Splay Tree: Zig Zig Rotation



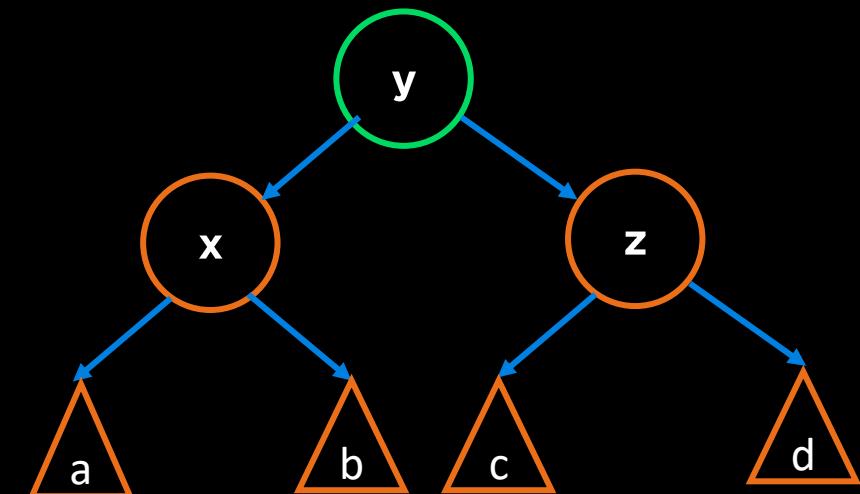
# Splay Tree: Zig Zag Rotation

Search y  
Splay(y)

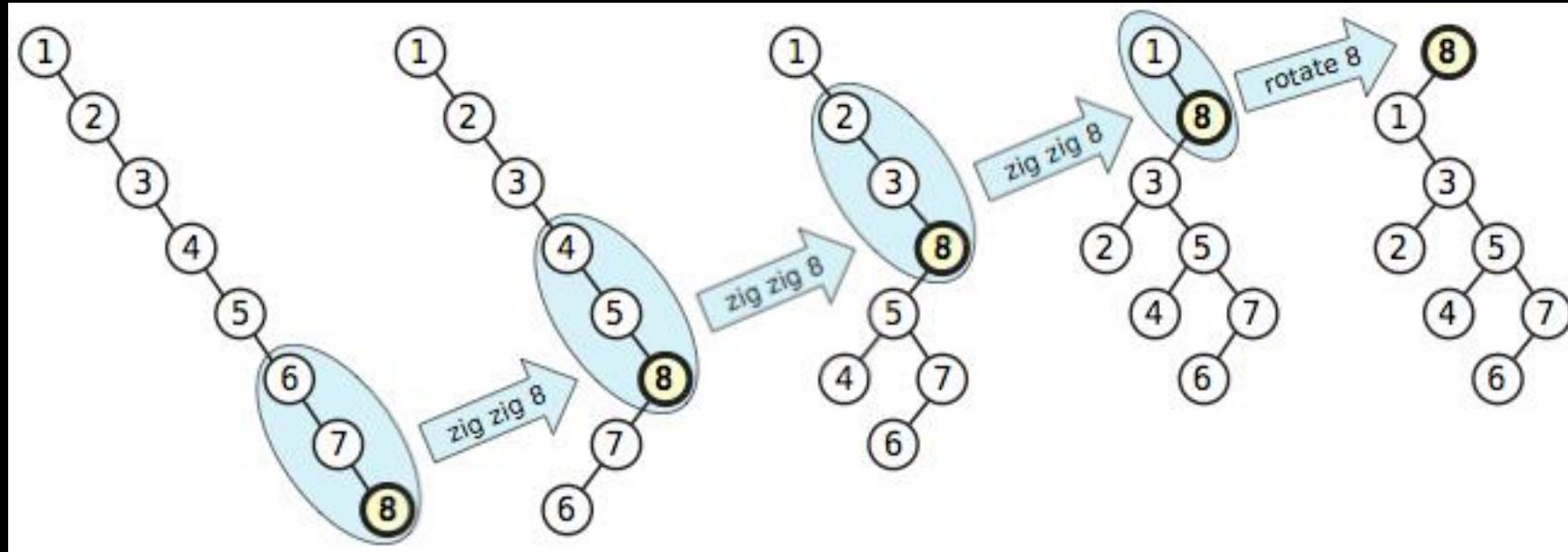


Zig-Zag

Double Rotation, Different direction



# Splay Tree: Basic Idea

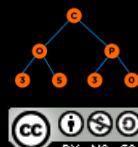


The basic idea of a splay tree is that after a node is accessed, it is pushed to the root via a series of rotations. And it does manage to shorten the tree.

Start at bottom and move up!  $\text{Splay}(N)$  till  $N.\text{Parent} == \text{null}$

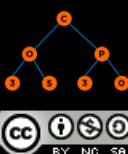
# Splay Tree: Insert/Search

- Same as BST followed by a Splay Operation on the searched node or newly inserted node
- **Splay(N)**
  - ❖ Determine proper case for rotation and apply
    - Zig Zig
    - Zig Zag
    - Zig
  - ❖ If N.Parent != null:
    - Splay(N)



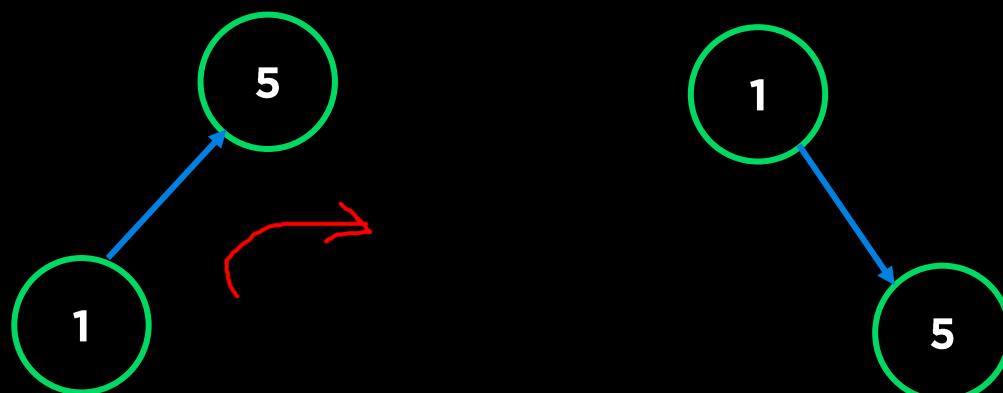
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



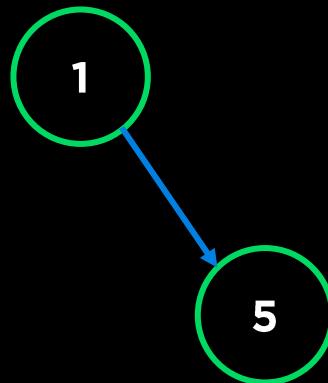
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



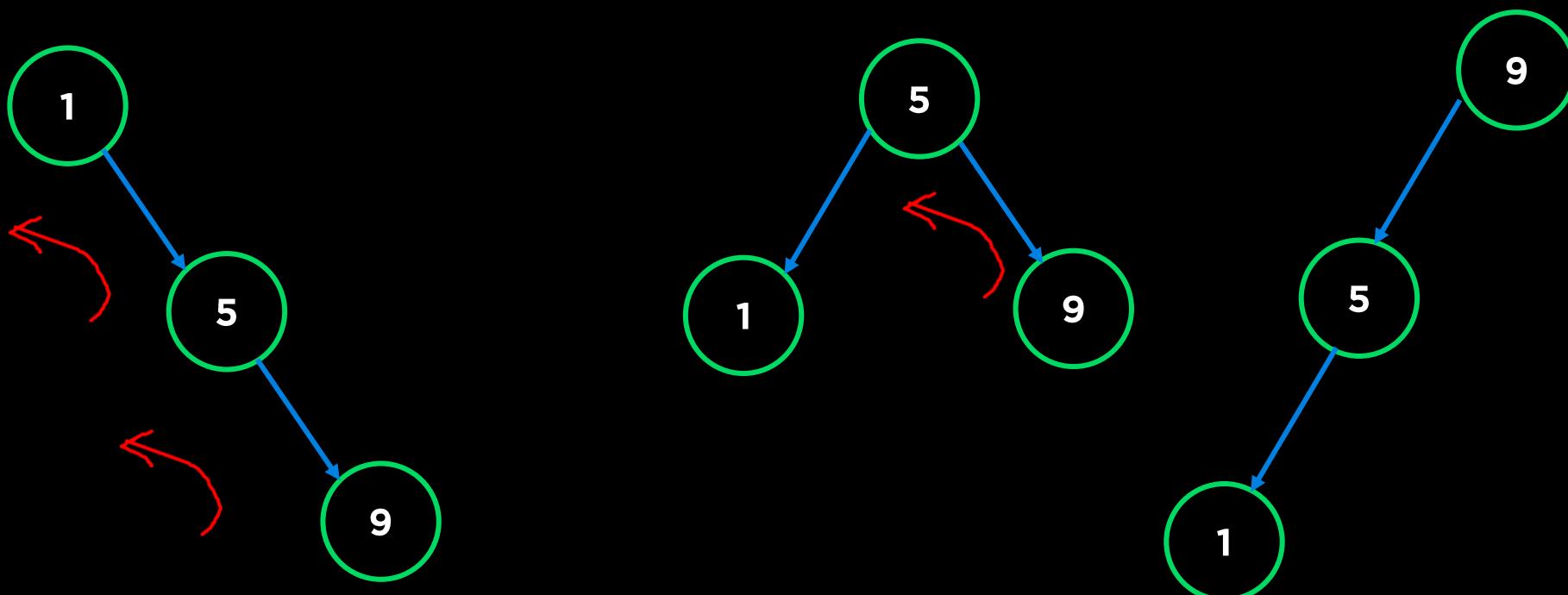
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



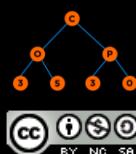
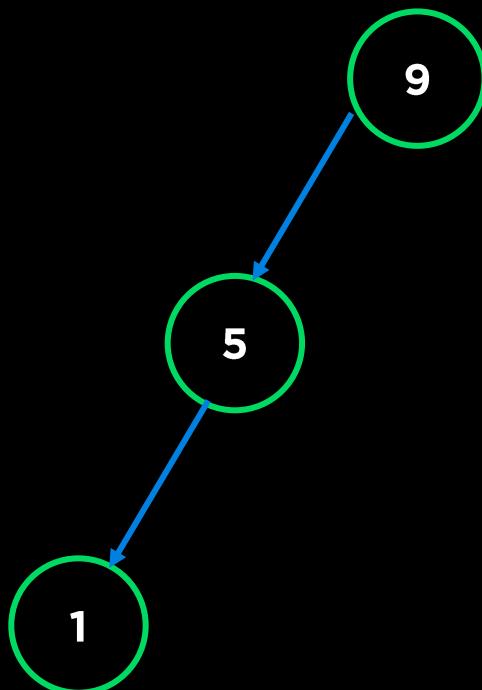
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



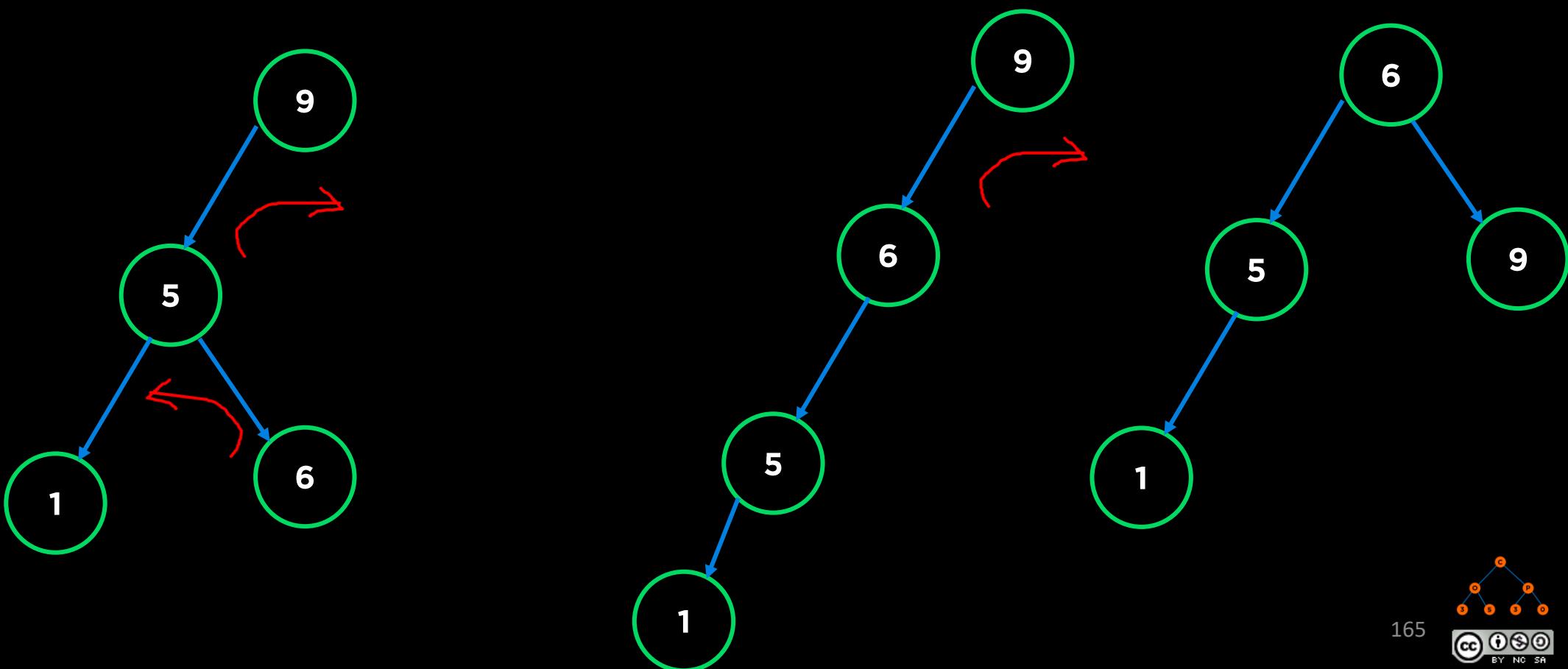
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



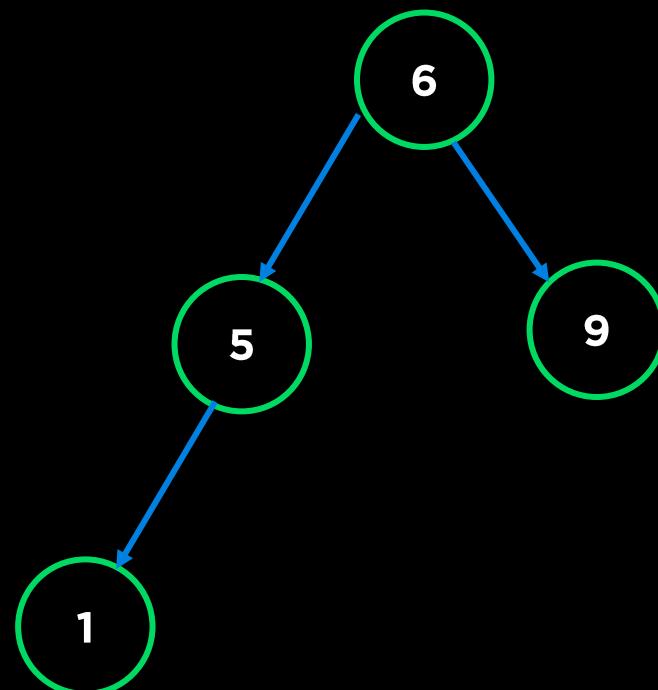
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



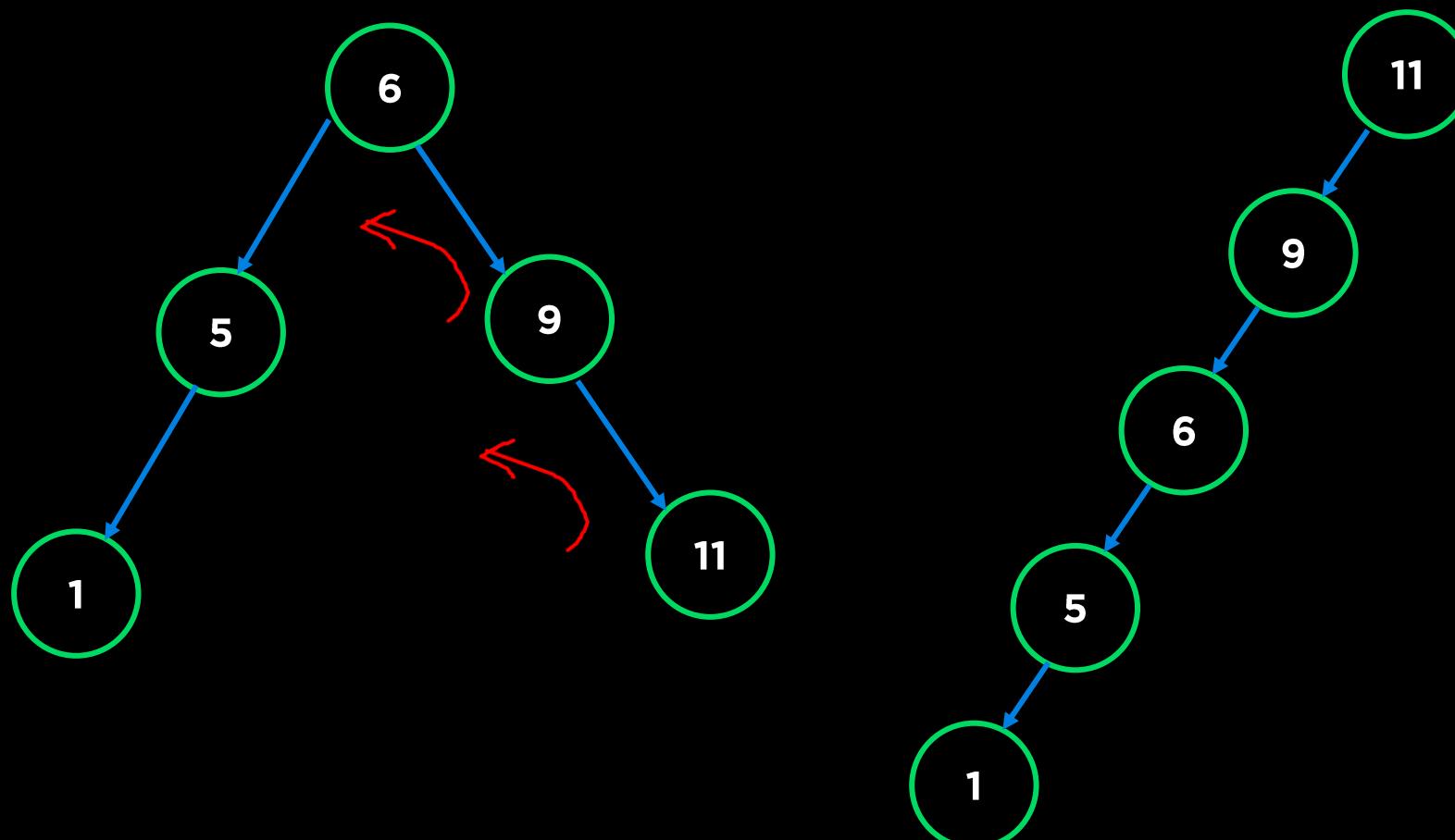
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



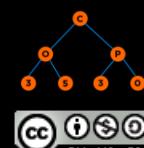
# Splay Tree: Example

Insert the following into a Splay Tree: 5, 1, 9, 6, 11



# Splay Tree: Performance

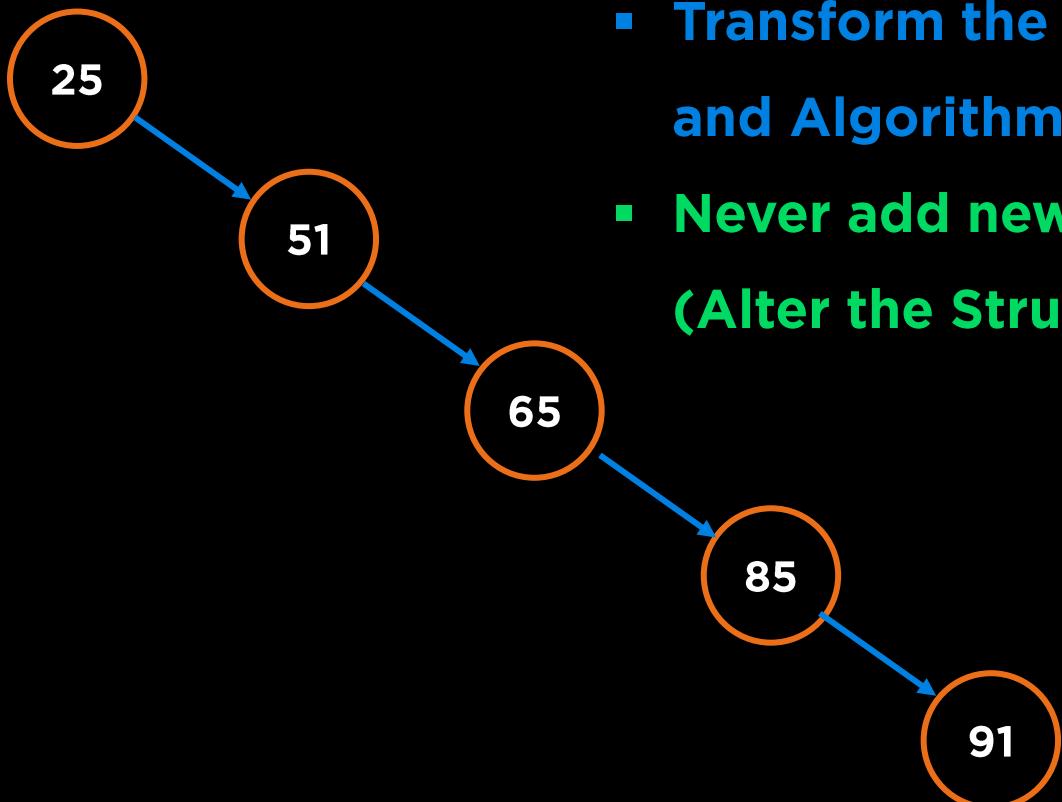
- A splay tree is a data structure that guarantees that  $m$  tree operations will take  $O(m \log n)$  time, where  $n$  is number of nodes
- On average, a tree operation is  $O(\log n)$
- In the worst case, an operation is  $O(n)$ , but subsequent operations are fast
- Implemented in Cache and Garbage Collection



# Resources

- B+ Tree Visualization: <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>
- Splay Tree Visualization: <https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>
- Original Paper, Splay Tree: <https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf>
- <https://stackoverflow.com/questions/7467079/difference-between-avl-trees-and-splay-trees>

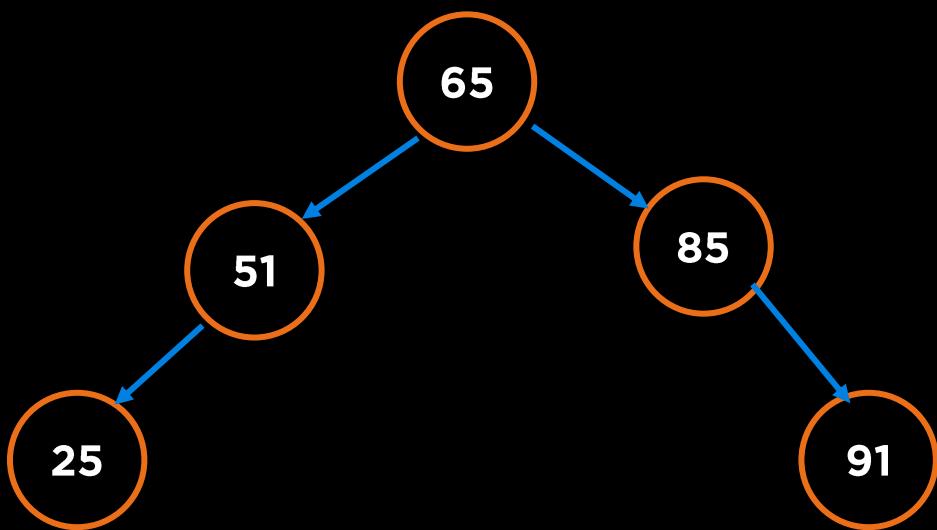
# How do we fix the Worst Case in a BST?



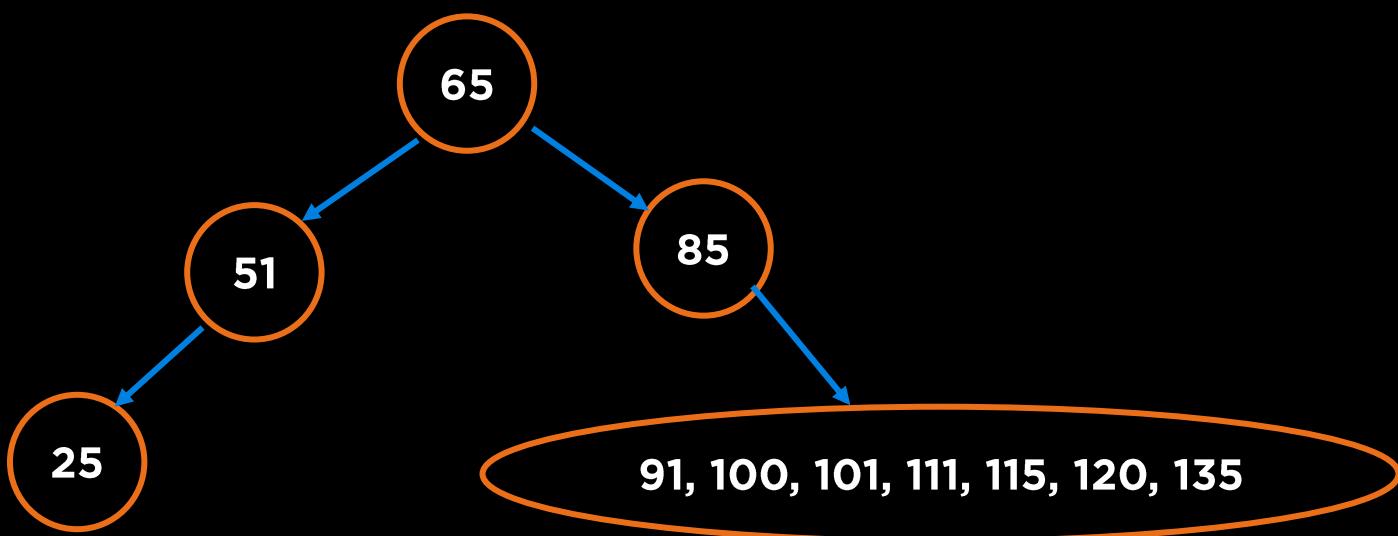
- Transform the “Spindly” Tree to “Bushy Tree” using Tools and Algorithms (Transformation)
- Never add new leaves at the bottom: Increase size of node (Alter the Structure by Design)

# Insertion in Non-Random Order

Insert 100, 101, 111,  
115, 120, 135

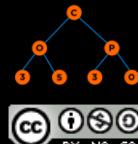
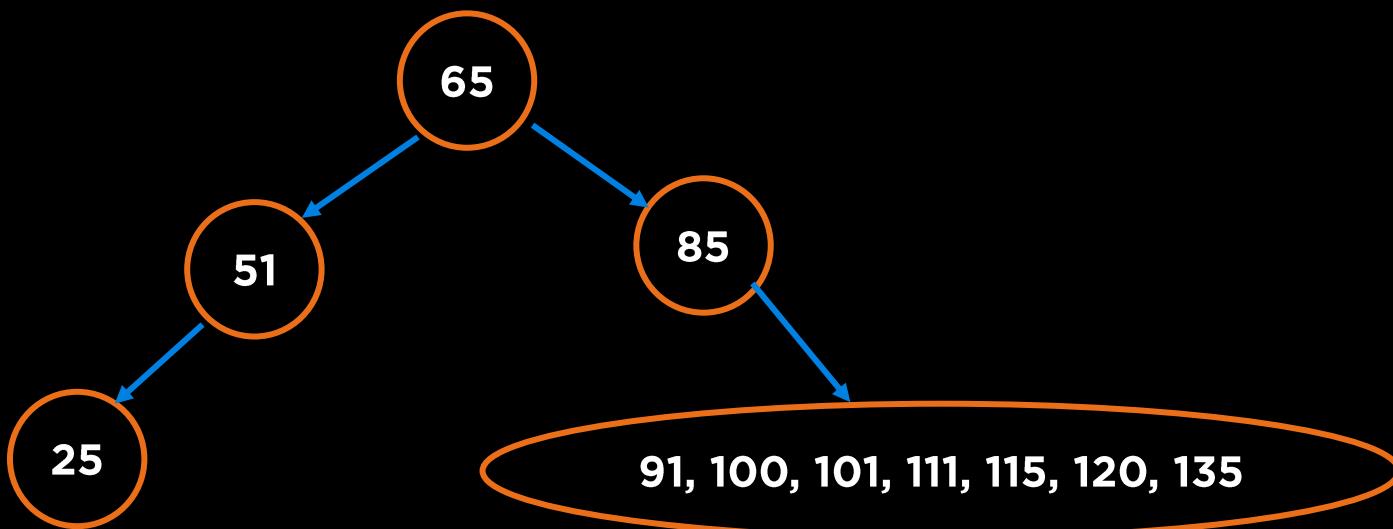


# Insertion in Non-Random Order



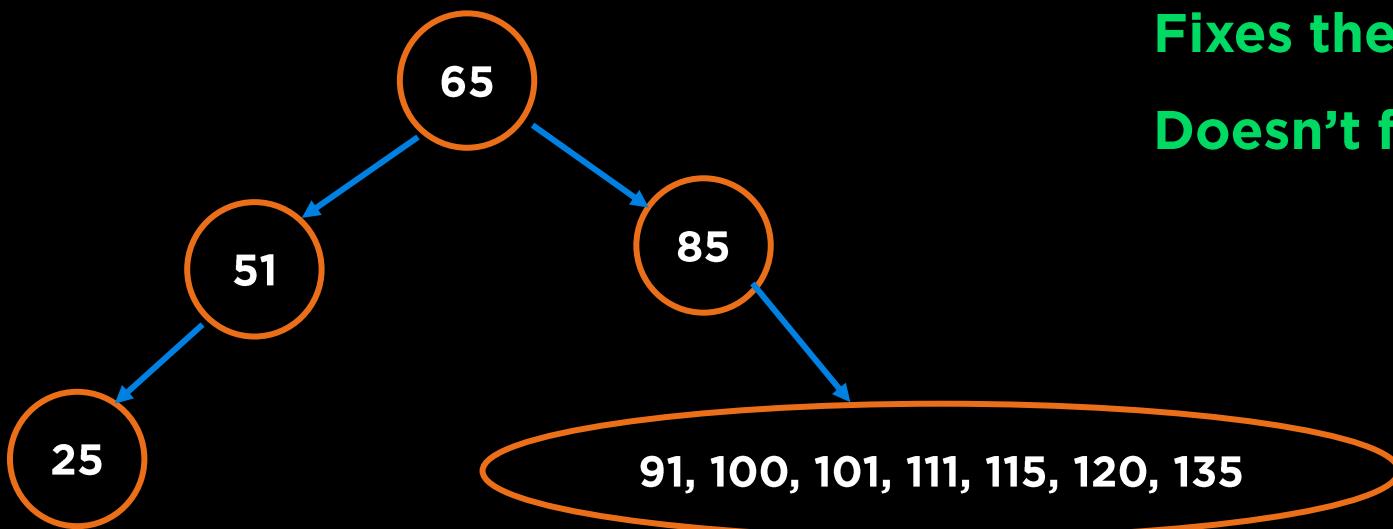
# Insertion in Non-Random Order

- Can lead to overstuffing
- Overstuffed trees have better balanced height
- Consistent BST



# Insertion in Non-Random Order

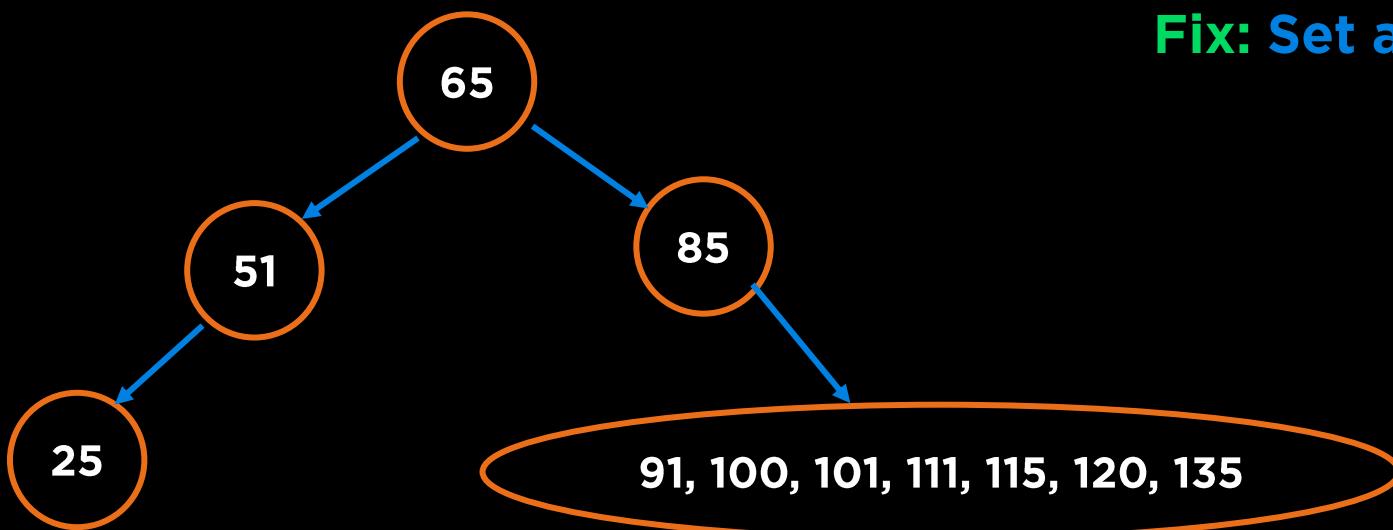
- Can lead to overstuffing
- Overstuffed trees have better balanced height
- Consistent BST



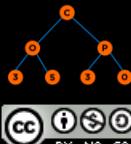
Fixes the height imbalance problem  
Doesn't fix Non-random insertion

# Insertion in Non-Random Order

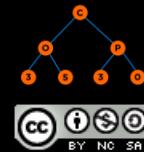
- Can lead to overstuffing
- Overstuffed trees have better balanced height
- Consistent BST



Fix: Set a limit, on the node filling

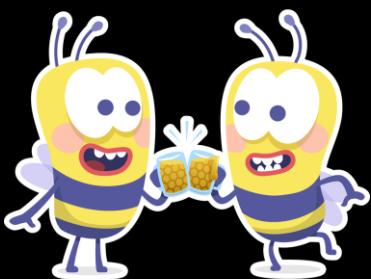
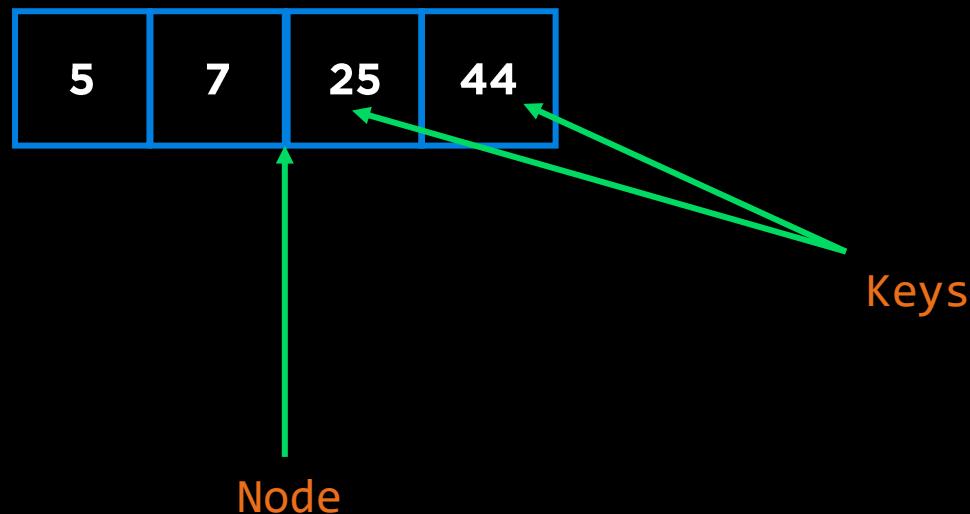


# B Trees



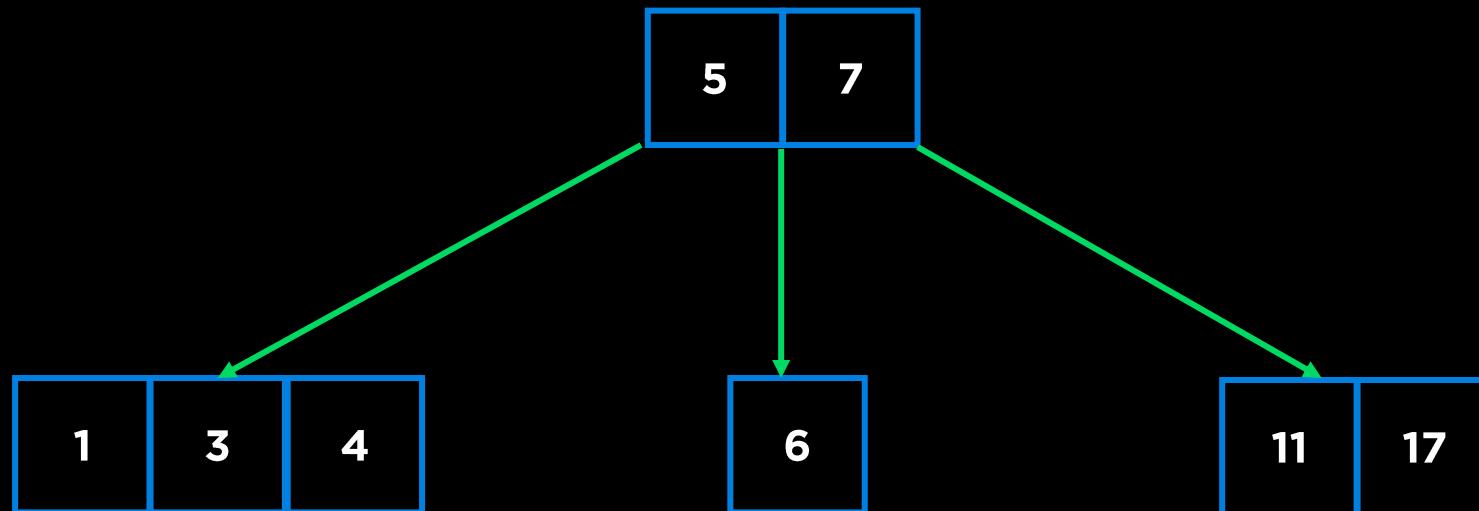
# Property #1

Each Node is a Block Containing Multiple “Keys”



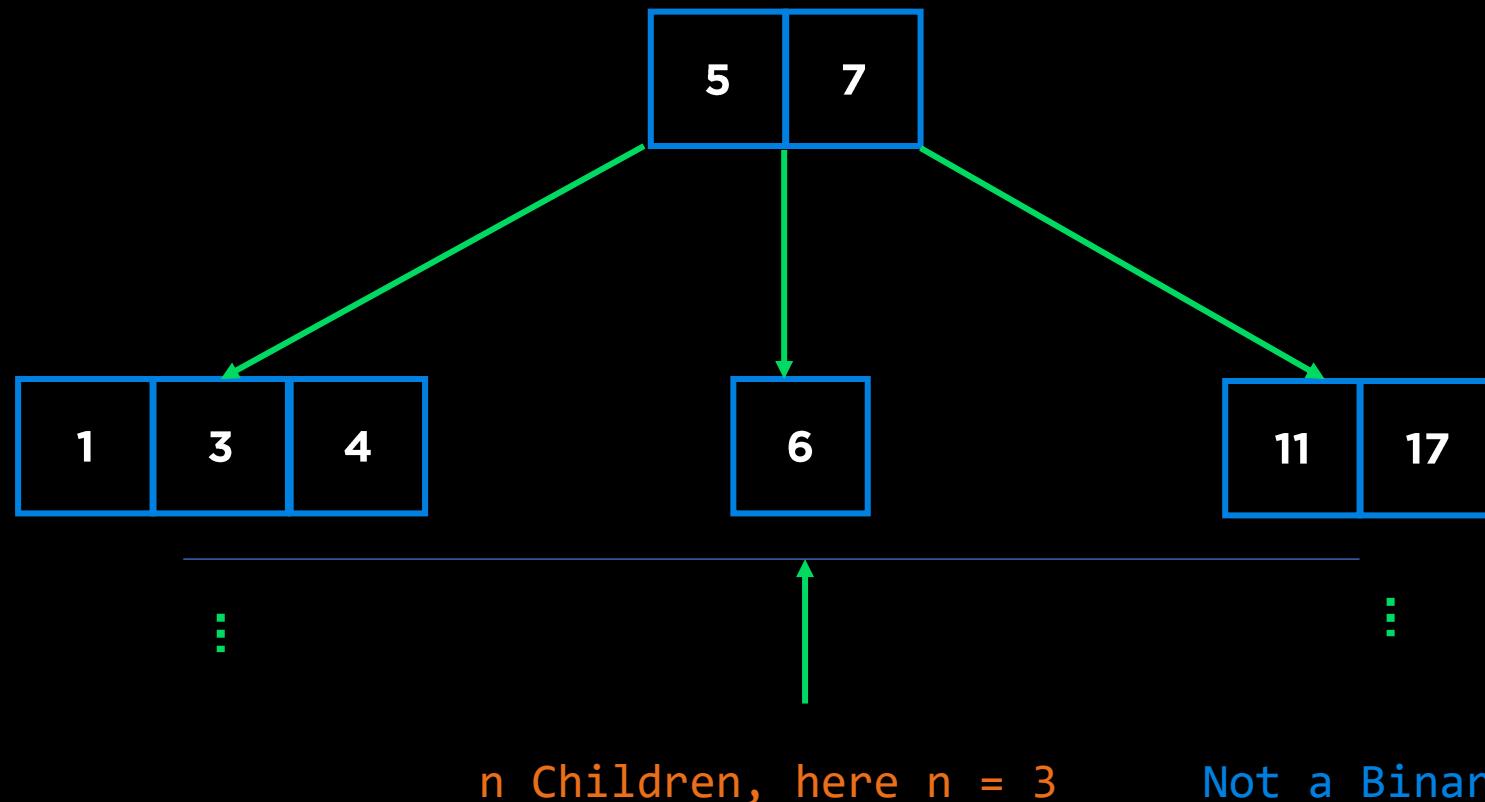
# Property #2

B Trees are n-ary Trees of order n, each node has up to n children. They follow BST property



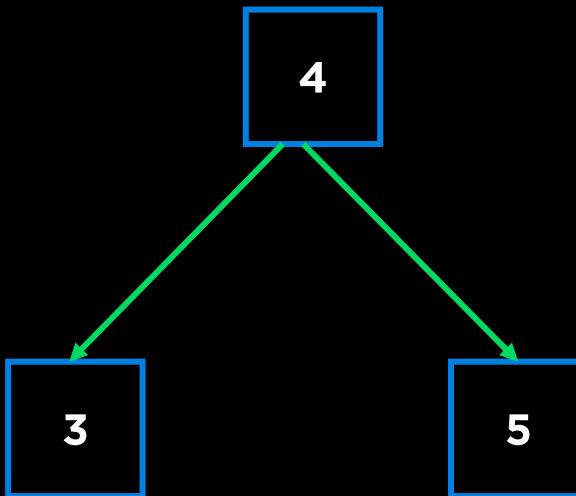
# Property #2

B Trees are n-ary Trees of order n, each node has up to n children. They follow BST property



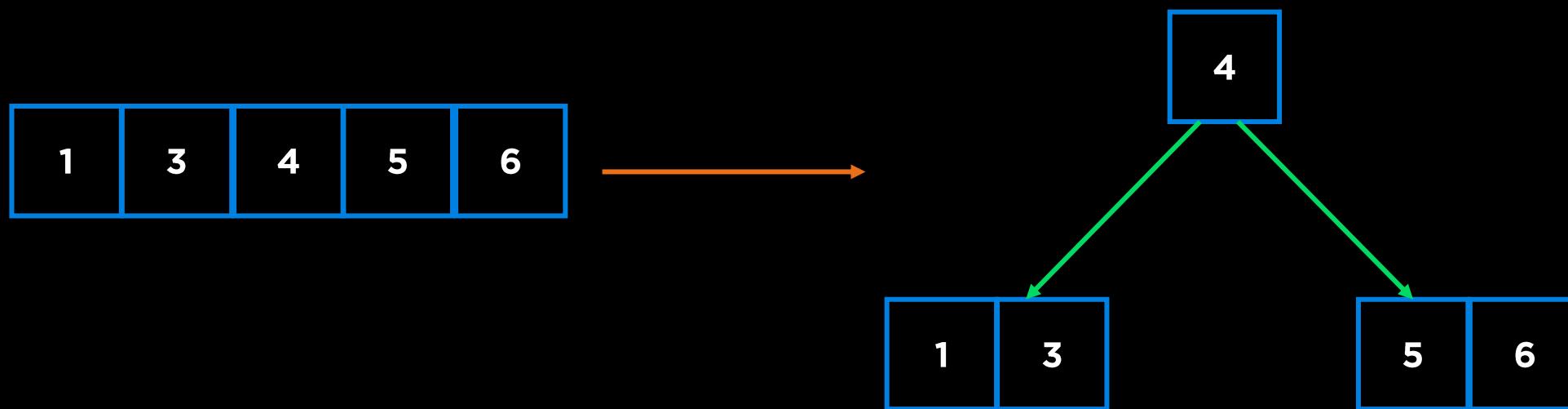
# Property #2

Example B tree: (n=2 is a BST)



# Property #3

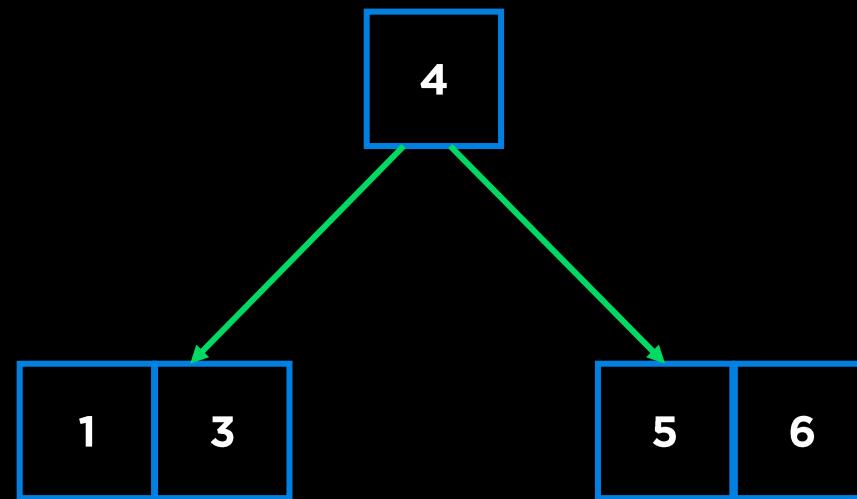
**Tree Building is Bottom-up and Leaves are at same depth**



# Property #4

## Children

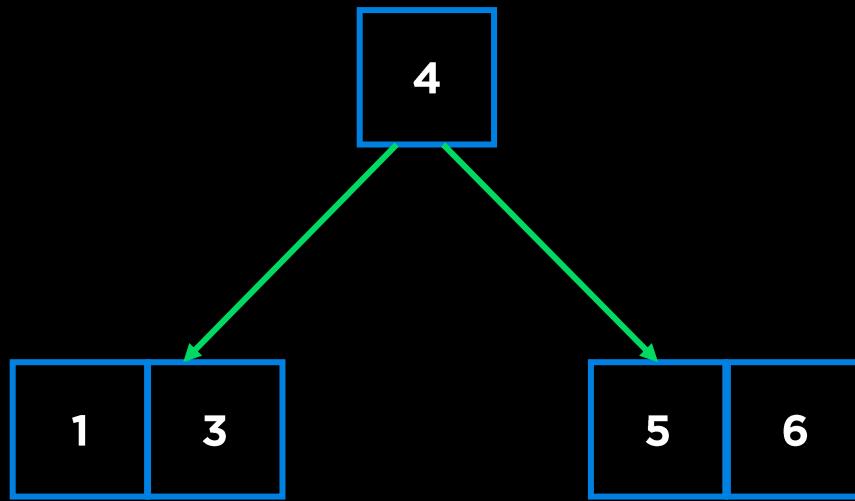
- Root is a leaf or has  $[2, n]$  children
- Non-leaf nodes have  $[\text{ceil}(n/2), n]$  children



# Property #5

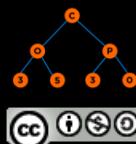
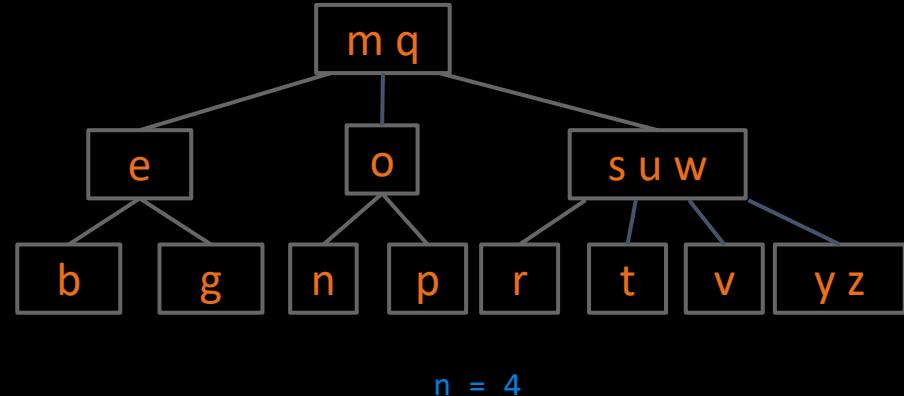
## Keys

- All keys are in Sorted Order
- Leaf nodes store up to n-1 keys.
- A non-leaf node with x children contains x-1 keys.



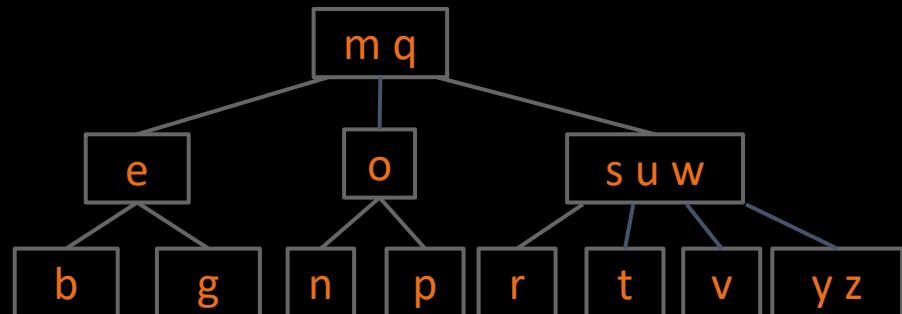
# Properties Summary

- **Each Node is a Block Containing Multiple “Keys”**
- **B Trees are n-ary Trees or have an order “n”**
- **Children**
  - **Root is a leaf or has [2, n] children**
  - **Non-leaf nodes have [ $\text{ceil}(n/2)$ , n] children**
- **Keys**
  - **All keys are in Sorted Order**
  - **Leaf nodes store up to n-1 keys.**
  - **A non-leaf node with x children contains x-1 keys.**
- **All leaves are at same depth, so the tree is always balanced**
- **Data items are stored in leaves and non-leaf nodes in a B Tree. In a B+ Tree, data is stored in only leaves**
- **When a node is full (has more than n-1 keys) Splitting occurs**

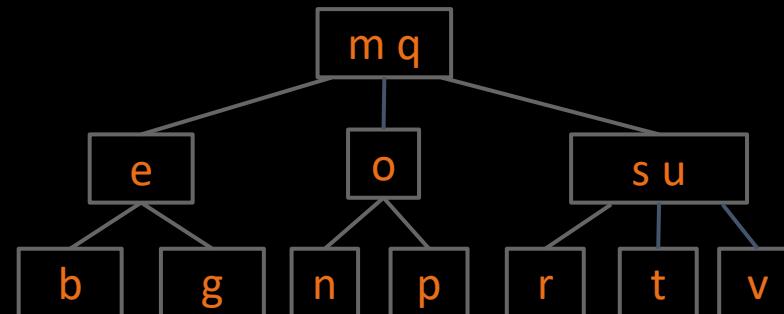


# Examples

- B-trees of order  $n=4$  are also called a 2-3-4 tree or a 2-4 tree.
  - “2-3-4” refers to the number of children that a node can have, e.g. a 2-3-4 tree node may have 2, 3, or 4 children.
- B-trees of order  $n=3$  are also called a 2-3 tree.
- Keys can be very large in case of file systems



2-3-4 a.k.a. 2-4 Tree:  
Max 3 items per node.  
Max 4 non-null children per node.  
 $n=4$



2-3 Tree:  
Max 2 items per node.  
Max 3 non-null children per node.  
 $n=3$

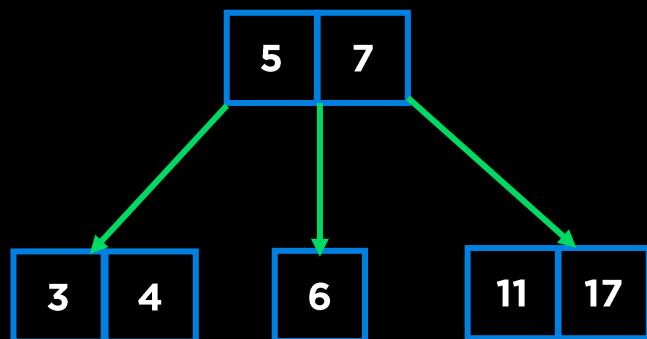
# B Tree Insertion

2-3 Tree:

Max 2 items per node.

Max 3 non-null children per node.

Insert 1



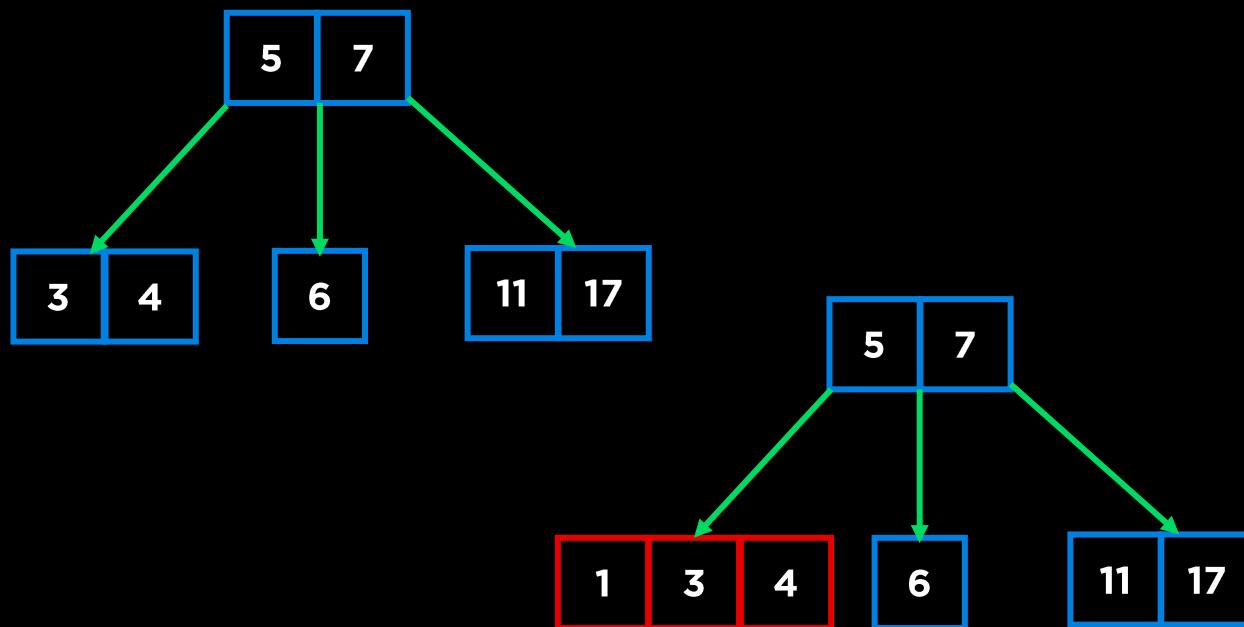
# B Tree

2-3 Tree:

Max 2 items per node.

Max 3 non-null children per node.

Insert 1



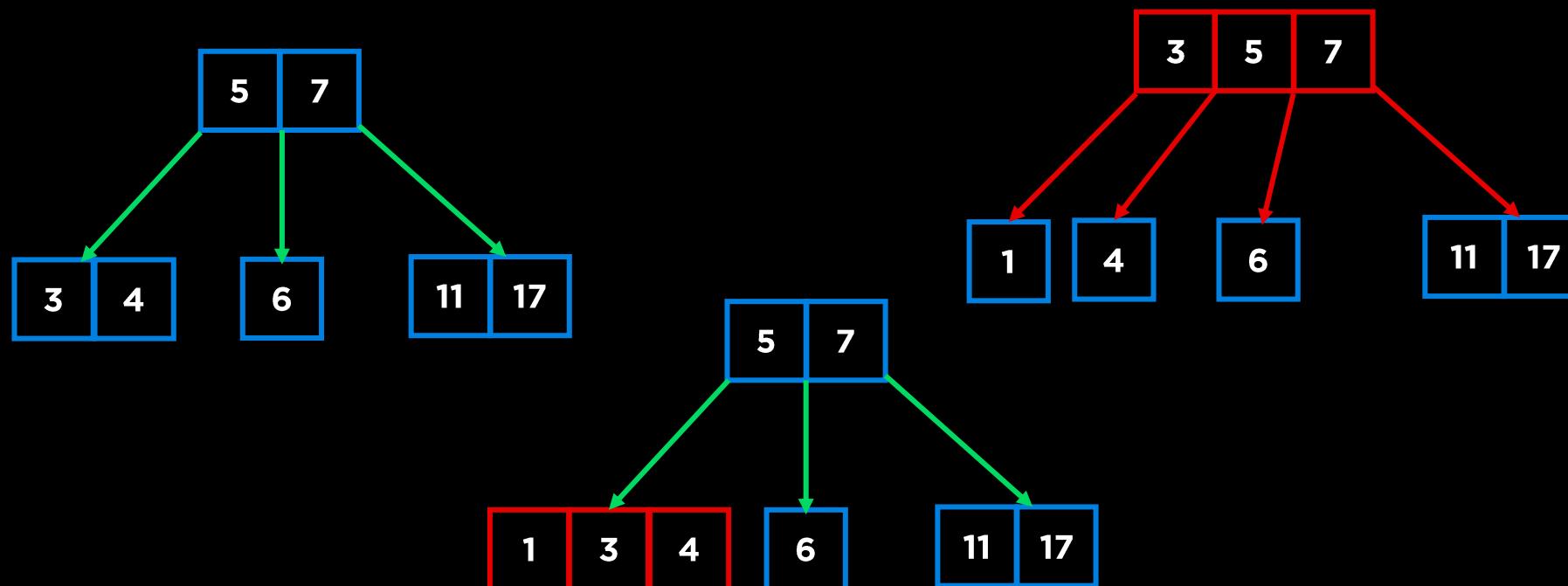
# B Tree

2-3 Tree:

Max 2 items per node.

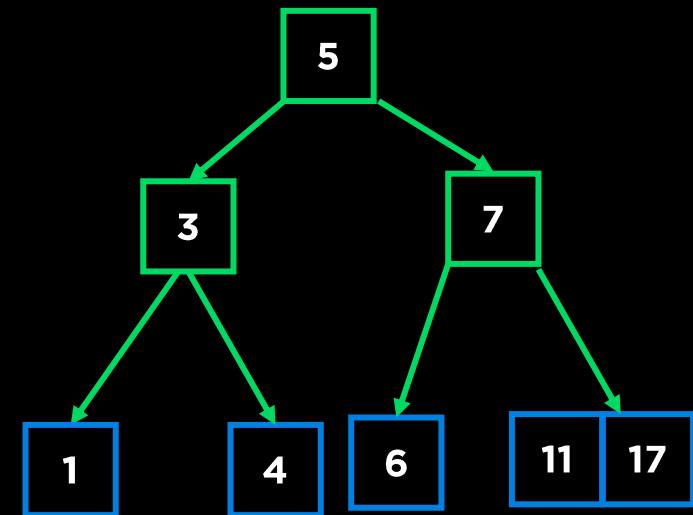
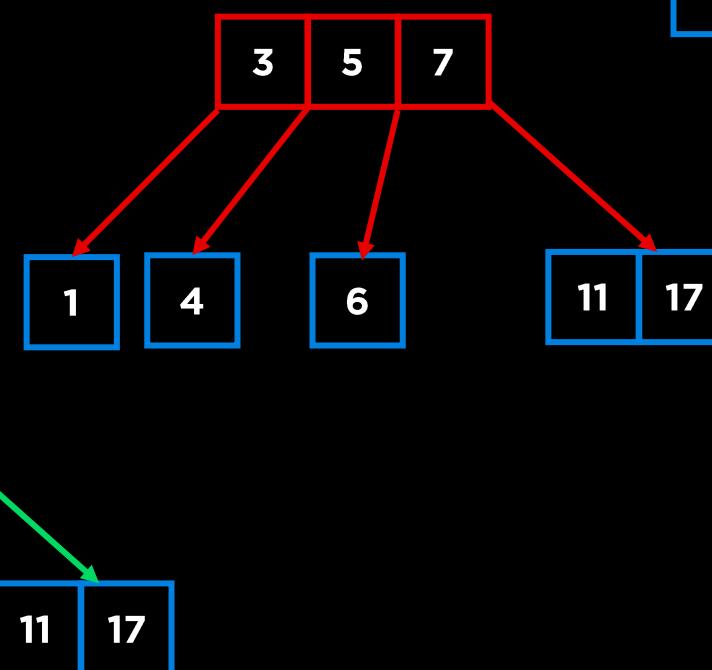
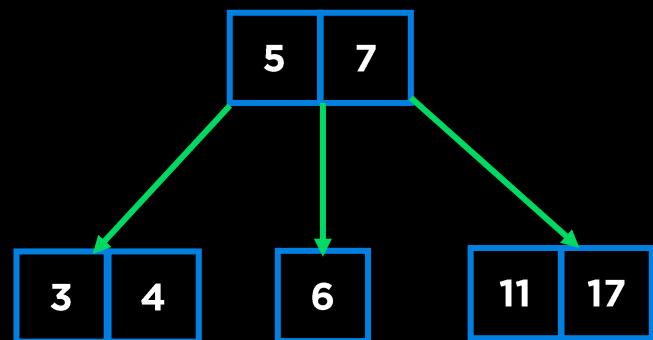
Max 3 non-null children per node.

Insert 1



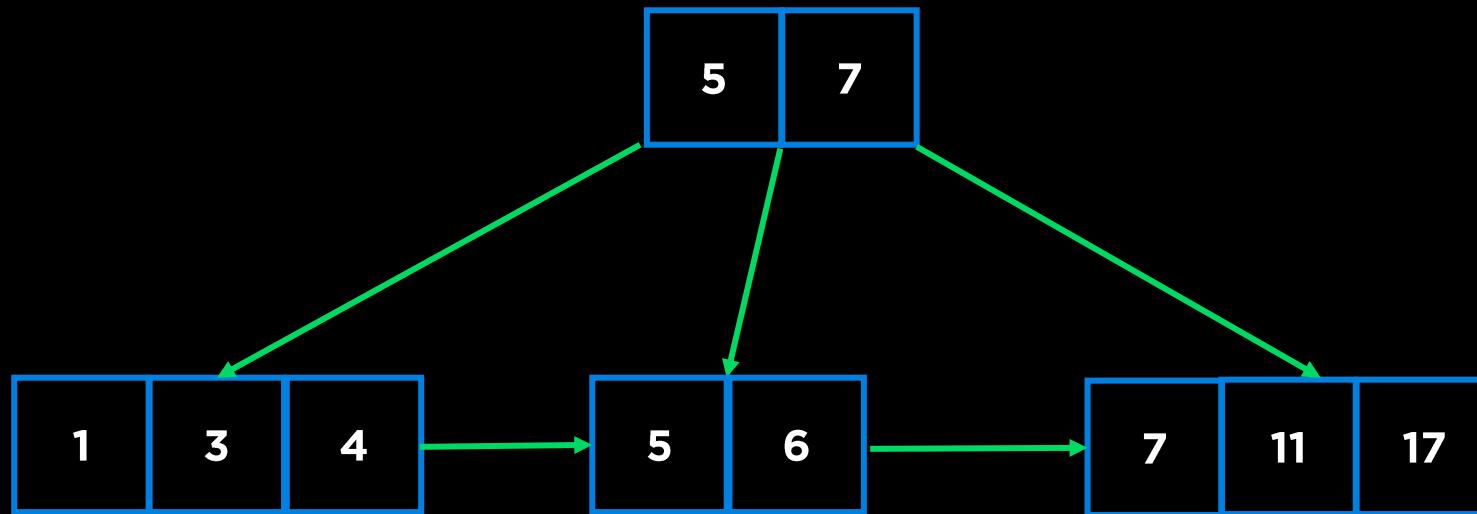
# B Tree

2-3 Tree:  
Max 2 items per node.  
Max 3 non-null children per node.



Height is still perfectly balanced!

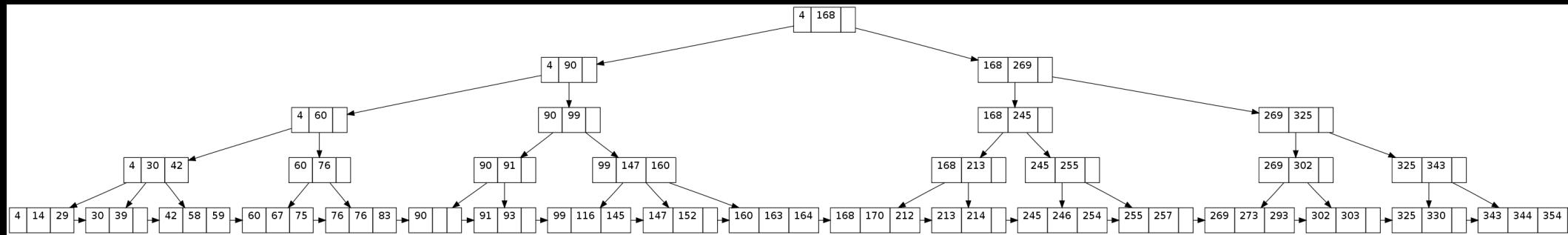
# B+ Tree



B Tree + All data is stored in the leaves +  
The leaves have pointers to the other leaves forming a linked list for faster traversal

[https://en.wikipedia.org/wiki/B%2B\\_tree](https://en.wikipedia.org/wiki/B%2B_tree)

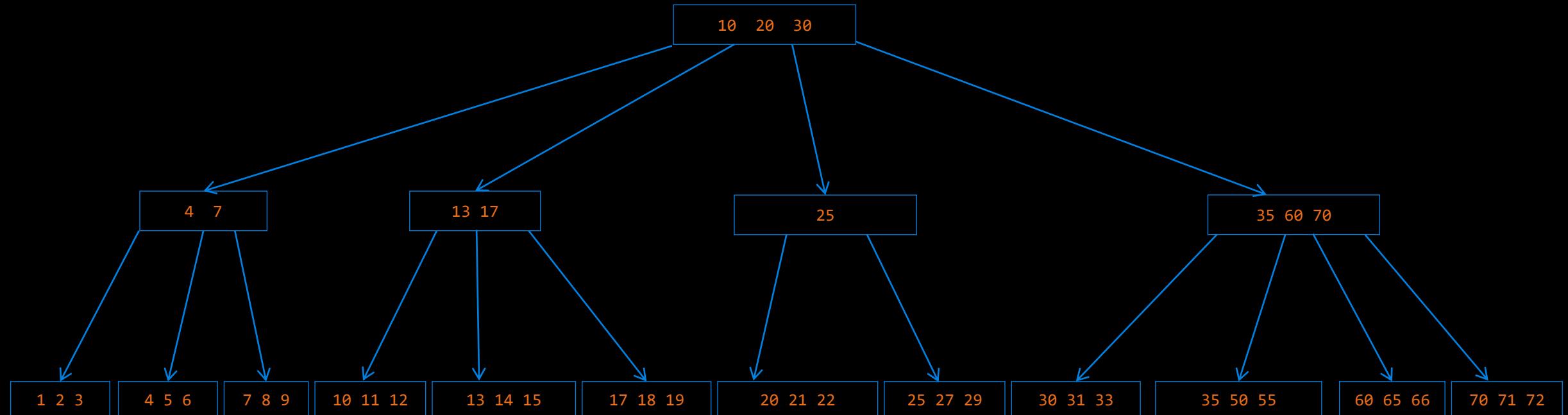
# B+ Tree



$N = 4$

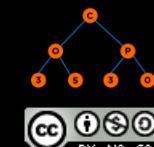
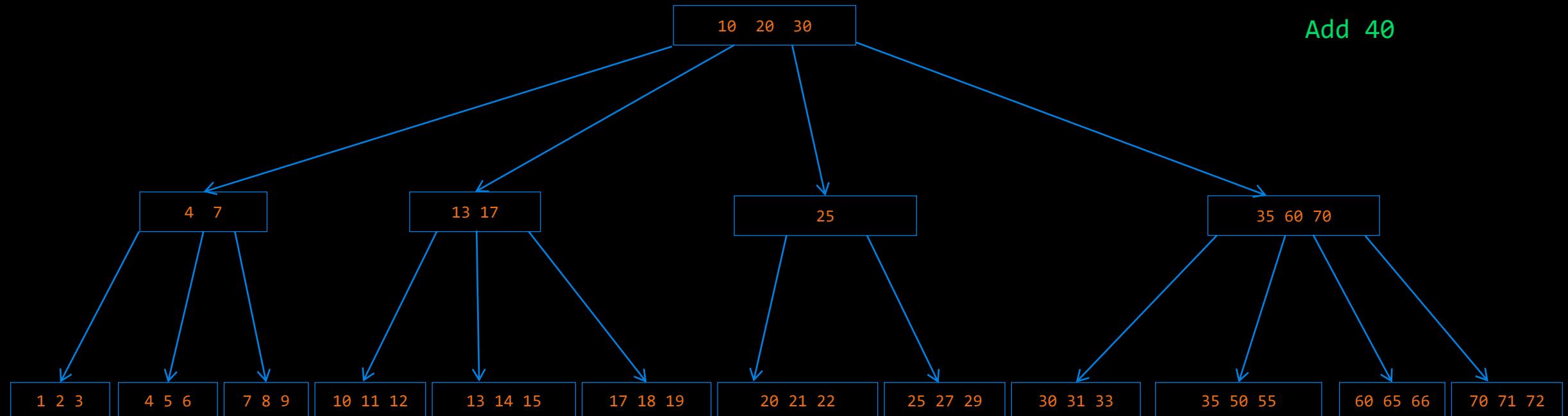
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



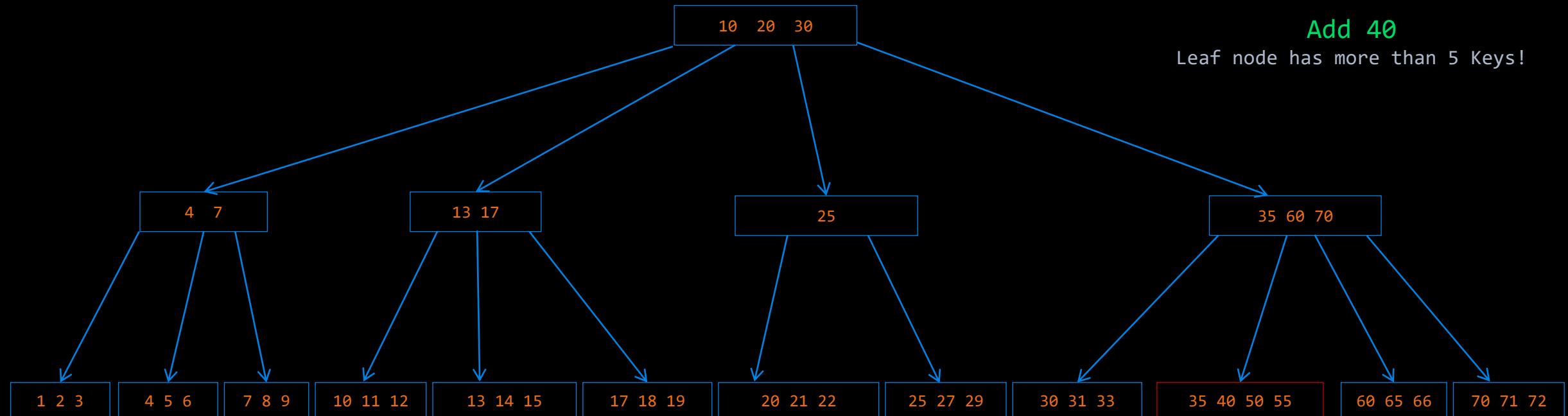
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



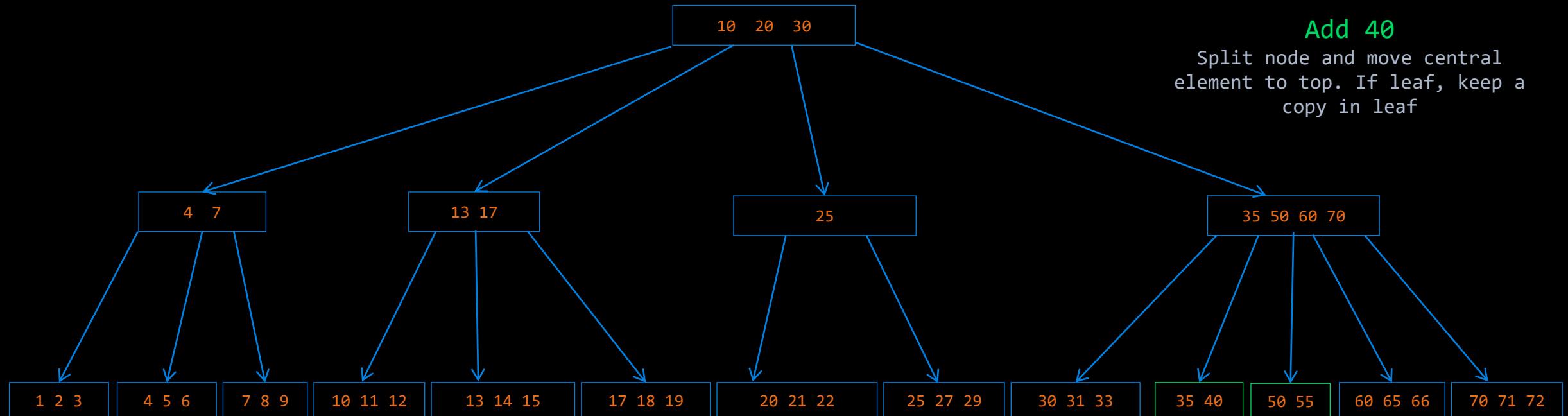
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



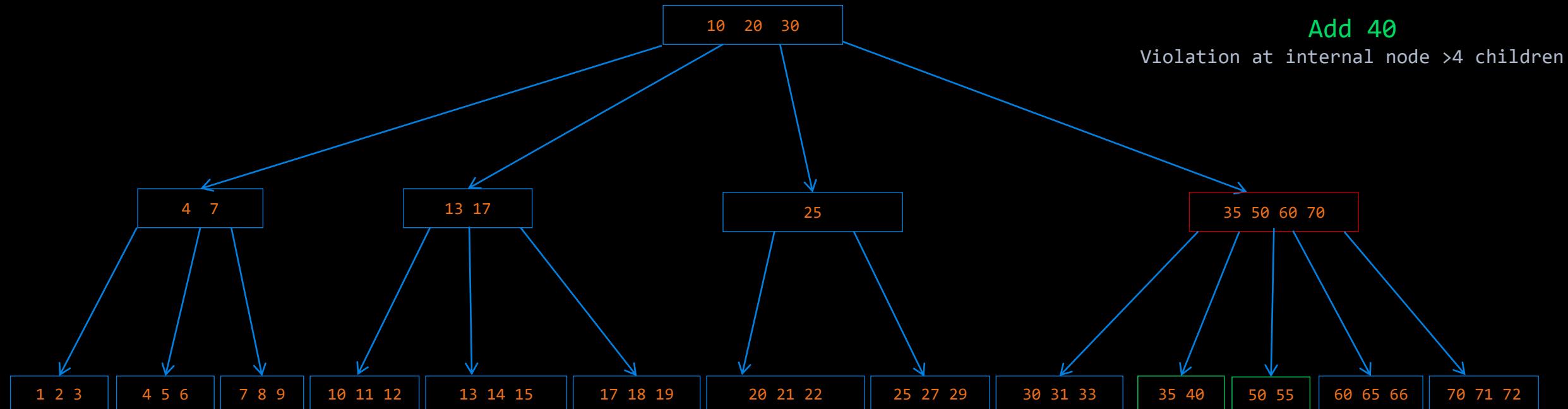
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



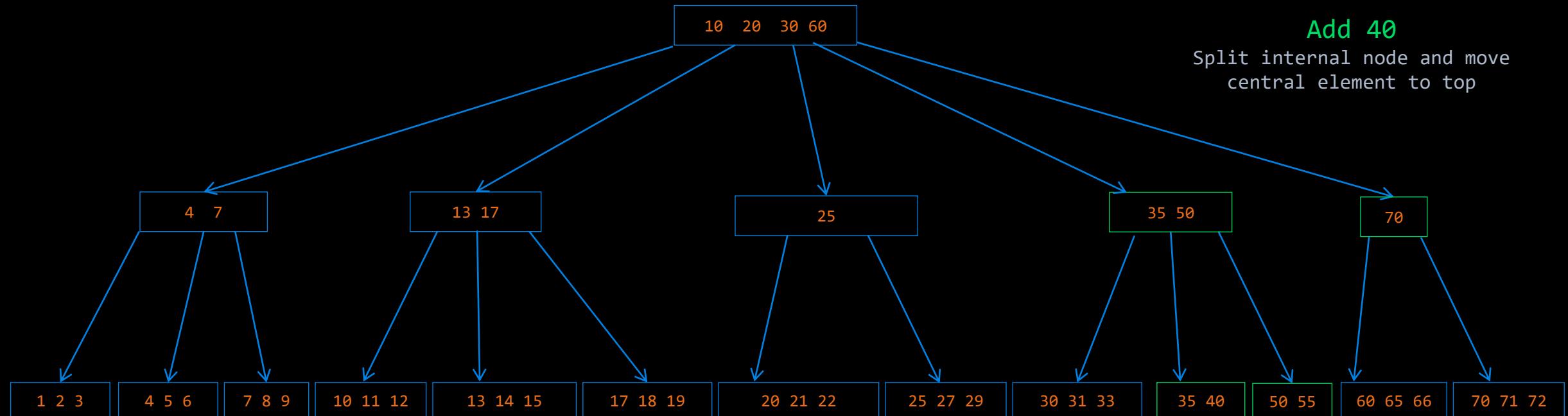
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



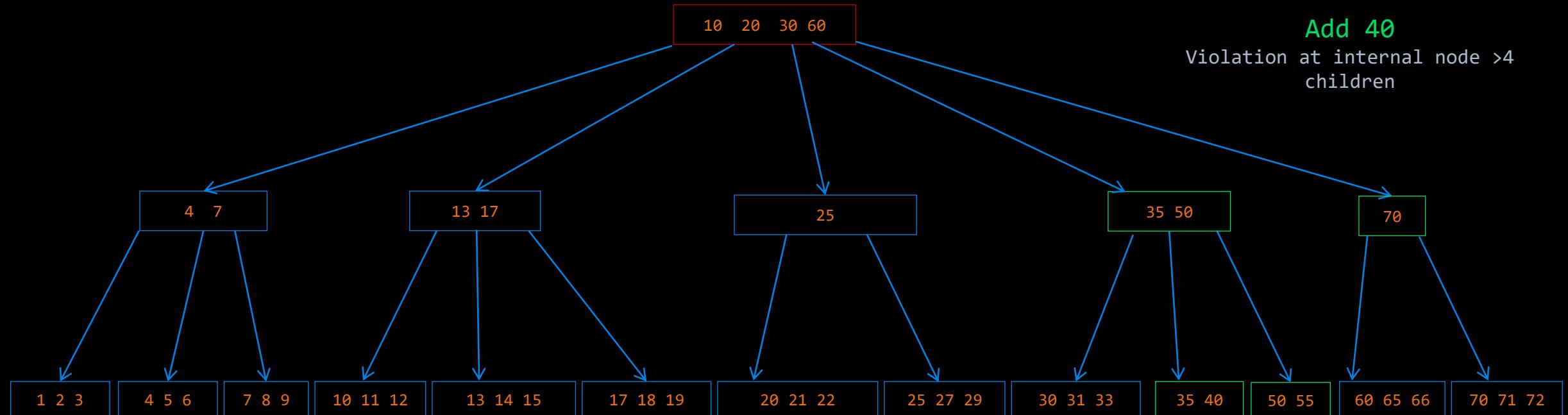
# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),

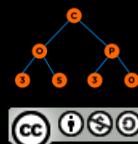
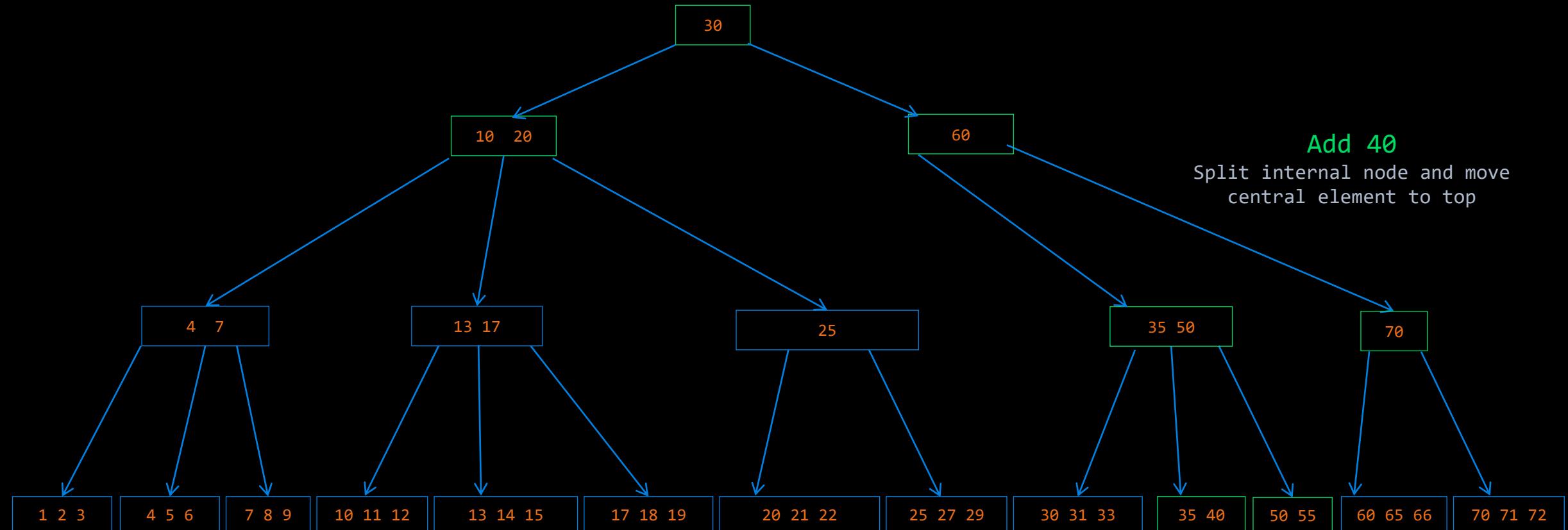


# B+ Tree Insertion

$N = 4$  (at most 4 children and 3 keys),



# B+ Tree Insertion



# B+ Tree

A completely full B+ Tree with N=3 and height = 2 (has level 0, 1, 2) has how many unique values?

# B+ Tree

A completely full B+ Tree with N=3 and height = 2 (has level 0, 1, 2) has how many unique values?

Level 0 - 2 values

Level 1 - 3 nodes, each with 2 values = 6 values

Level 2 -  $3 \times 3 = 9$  nodes, each with 2 values = 18 values

18 values as we only count last level which is unique

# B Tree

A completely full B Tree with N=3 and height = 2 (has level 0, 1, 2)  
has how many unique values?

# B Tree

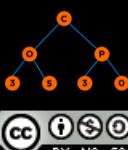
A completely full B Tree with N=3 and height = 2 (has level 0, 1, 2)  
has how many unique values?

Level 0 - 2 values

Level 1 - 3 nodes, each with 2 values = 6 values

Level 2 -  $3 \times 3 = 9$  nodes, each with 2 values = 18 values

26 values as we only count last level which is unique



# Use Case

- Hard Drives, Databases, Filesystems
- Indexing Example
  - Tracks, Sectors and Blocks

# Performance

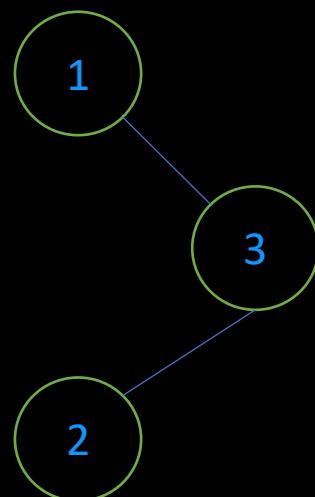
- Height: Between  $\sim \log_{l+1}(n)$  and  $\sim \log_2(n)$
- Largest possible height is all non-leaf nodes have  $l/2$  items
- Smallest possible height is all nodes have  $l$  items
- Overall height is therefore  $O(\log n)$
- Search Time:  $O(hl) \sim O(l \log(n))$ , where
  - $h$  is height of tree
  - $n$  is maximum number of children
  - $l$  is maximum number of keys
- Search Time:  $O(\log(n))$ , as  $l$  is a constant

# Tries

# Mentimeter

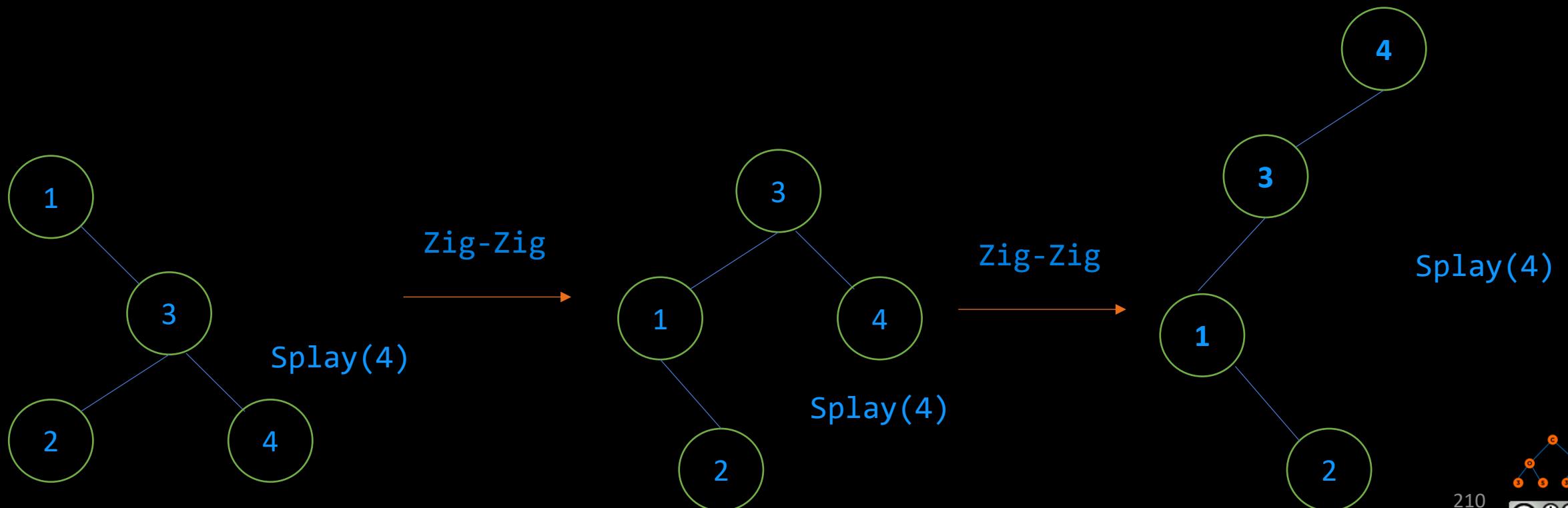
# Mentimeter

What will be the value of root node if we insert 4 into this Splay Tree?



# Mentimeter

What will be the value of root node if we insert 4 into this Splay Tree?



# Tips for Proposal (3a): Recommended

- All members of the teams should bring ideas before picking one. You can do a vote on which idea you like the most in your group and then propose it. Everyone should contribute!
- Take ideas from Graduate Level Courses or from other domains
- Treat this as a high-stakes personal project (more accountability)
- Take risks, long term bets, scope the project well rather than playing safe
- Have clear guidelines on who is doing what? Possible roles:
  - Lead the overall design of the code including interfaces
  - Implementing Algorithm/DS 1
  - Implementing Algorithm/DS 2
  - Testing
  - Design of User Interface
  - Creating a Video
  - Creating the Documentation
  - Analyzing Time Complexity

