DTS203TC
Design and Analysis of Algorithms

Lecture 7: Binary Search Trees and Balanced Trees

Dr. Qi Chen

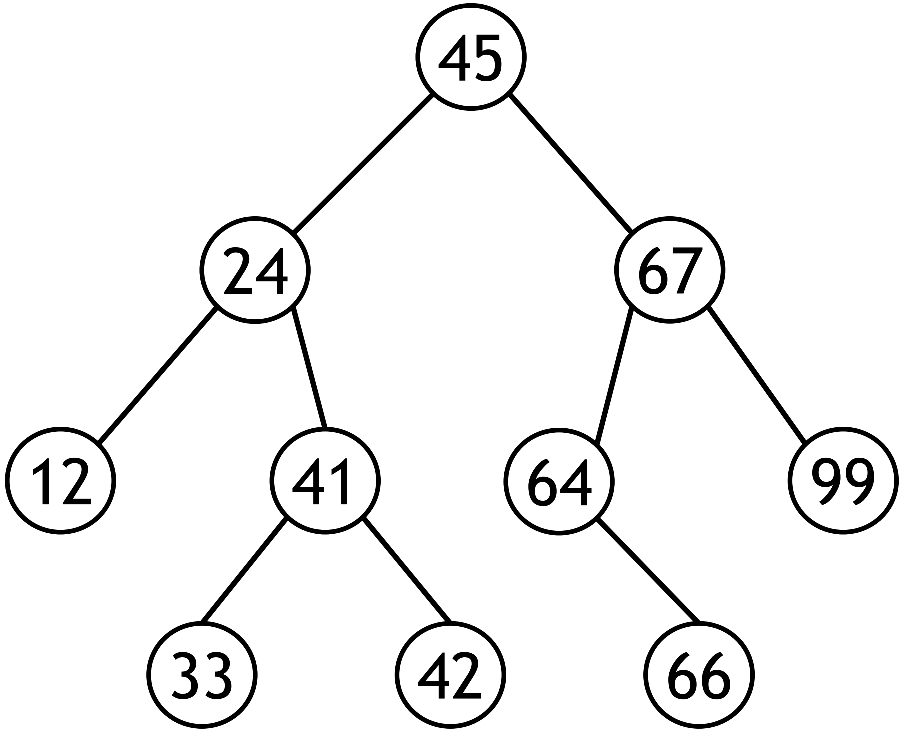
School of AI and Advanced Computing

Learning outcome

- Binary Search Tree
  - Tree Traversal
  - Insert, Search, Delete

- Balanced Trees
  - AVL Tree
  - Red-Black Tree

Binary Search Tree



Binary Search Tree (BST) Definitions

- A binary search tree is a binary tree where each node has a key
- The key in the left child (if exists) of a node is less than (or equal to) the key in the parent
- The key in the right child (if exists) of a node is greater than (or equal to) the key in the parent
- The left& right subtrees of the root are again binary search trees.

Binary Search Tree

- Each Node has
  - Left
  - Right
  - Key

- For each node n, with key k
  - n.left contains only nodes with keys <= k
  - n.right contains only nodes with keys >= k

- O(logn) on average for insert, lookup, and remove.

Binary Tree Traversal

- Inorder
- Preorder
- Postorder

Inorder Traversal: Recuresive

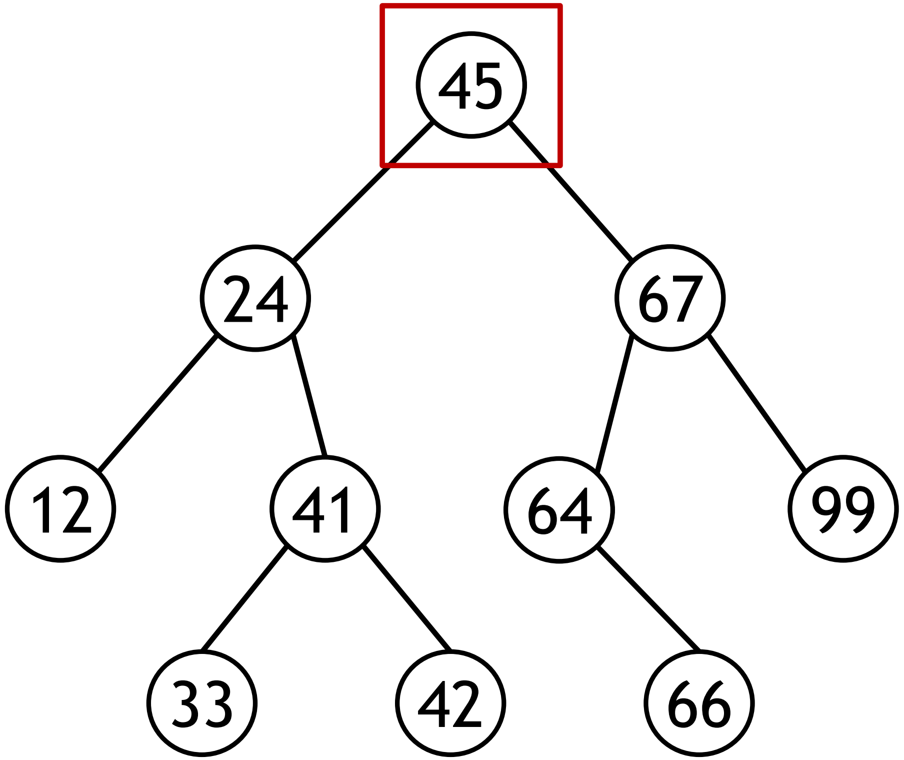
- Traverse the left subtree inorder
- Process (display) the value in the node
- Traverse the right subtree inorder

```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```

Example: Inorder Traversal

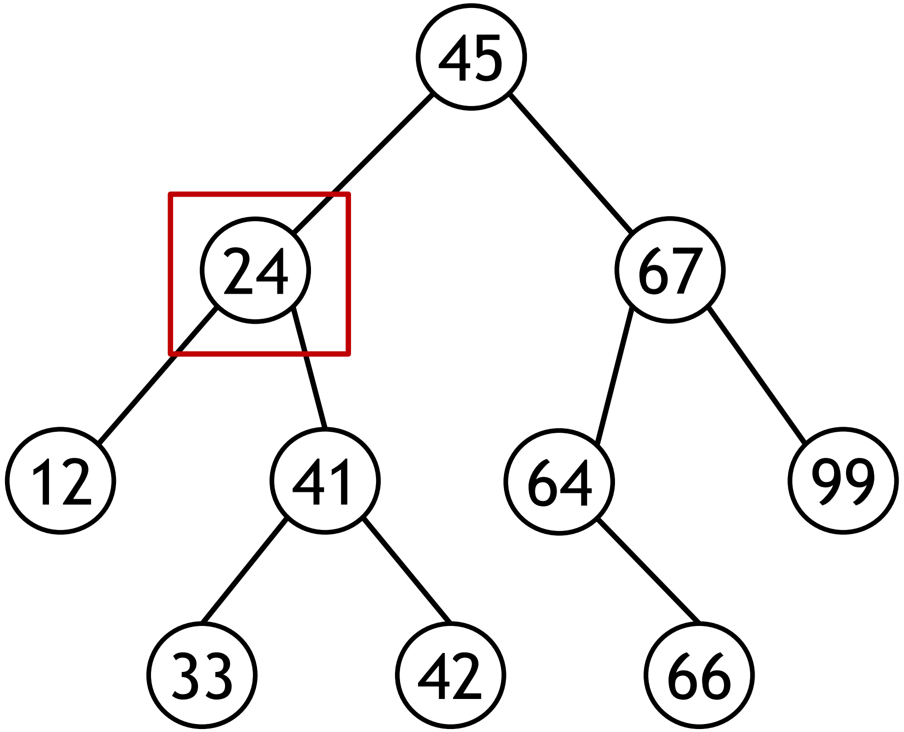
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```

To use inorderTraversal to print all the elements, we call inorderTraversal(root)



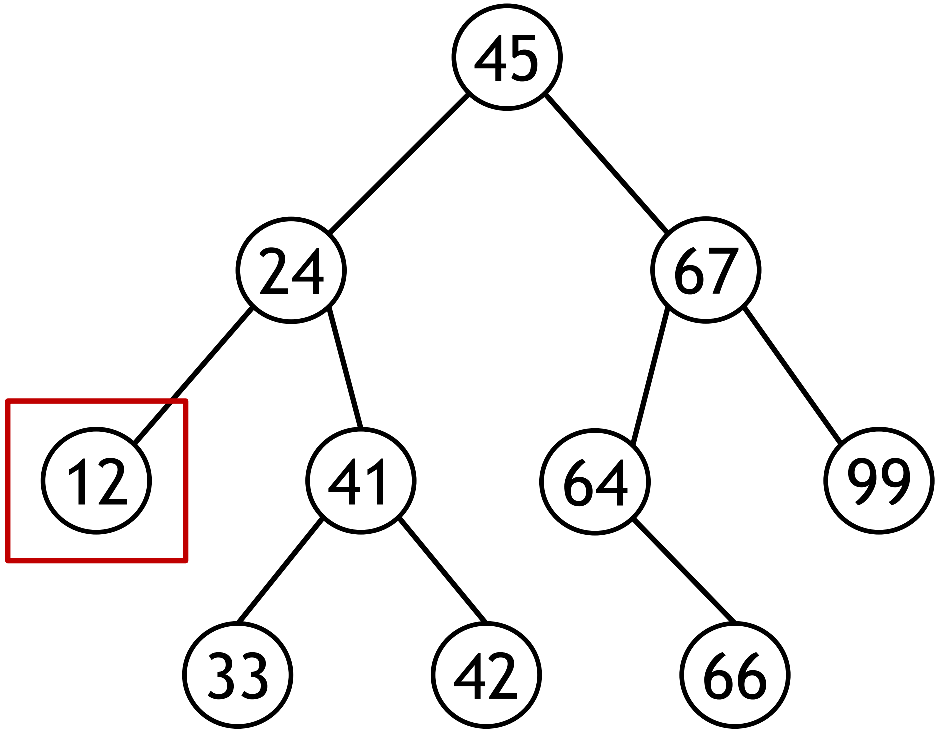
Example: Inorder Traversal

```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



Example: Inorder Traversal

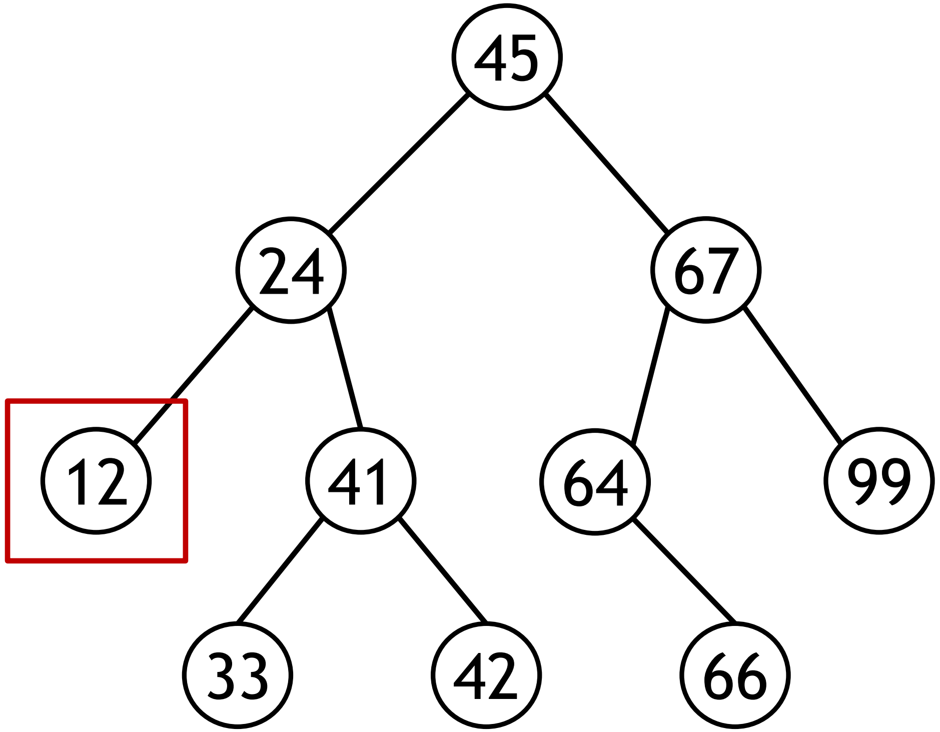
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



Example: Inorder Traversal

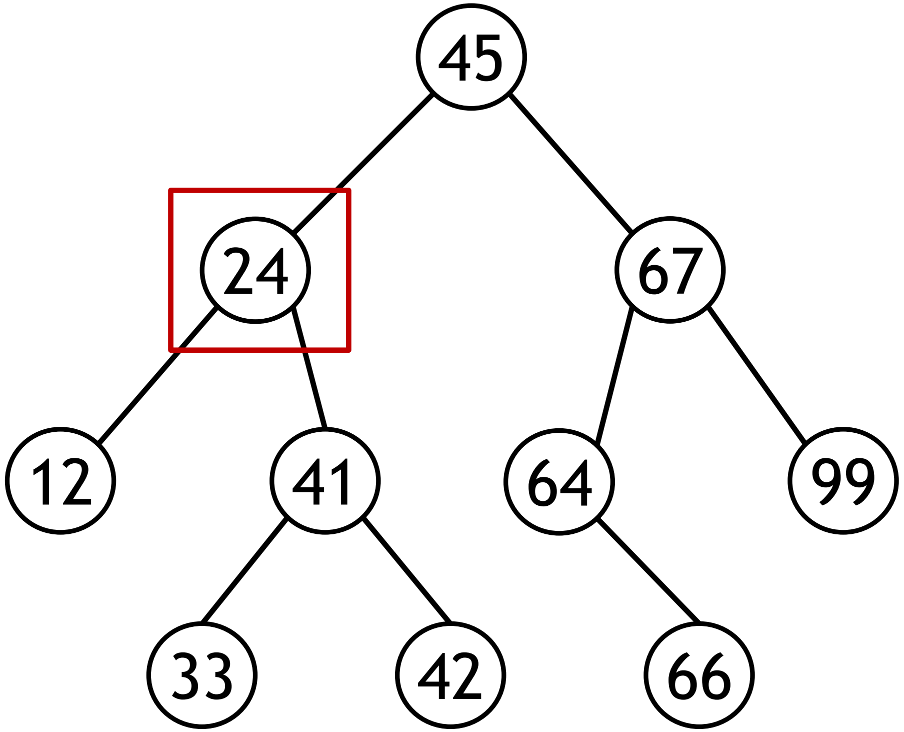
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```

[12,



Example: Inorder Traversal

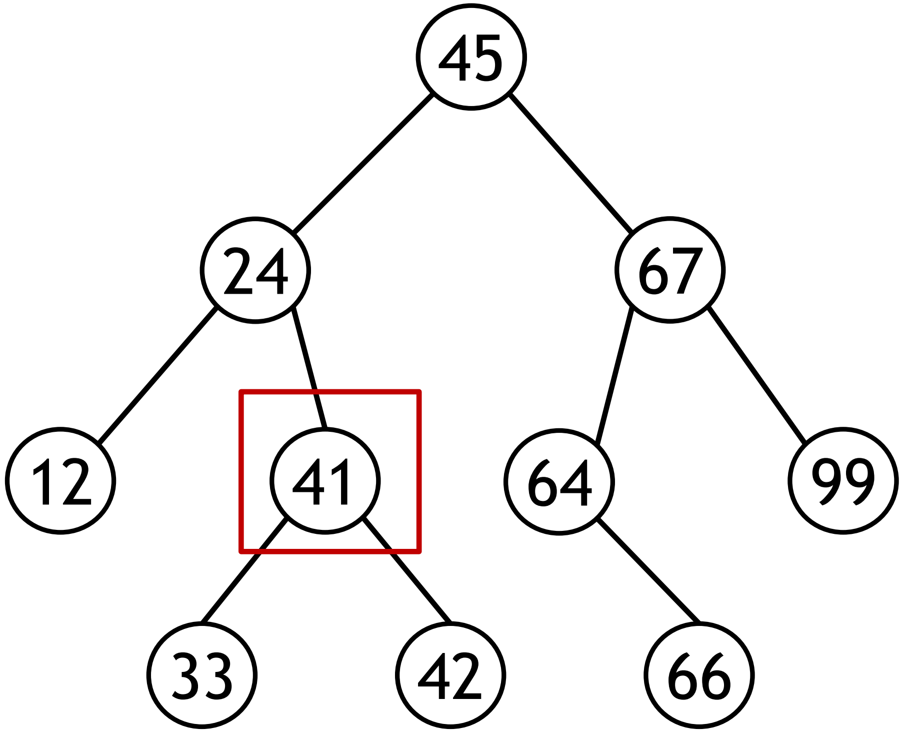
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24

Example: Inorder Traversal

```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```

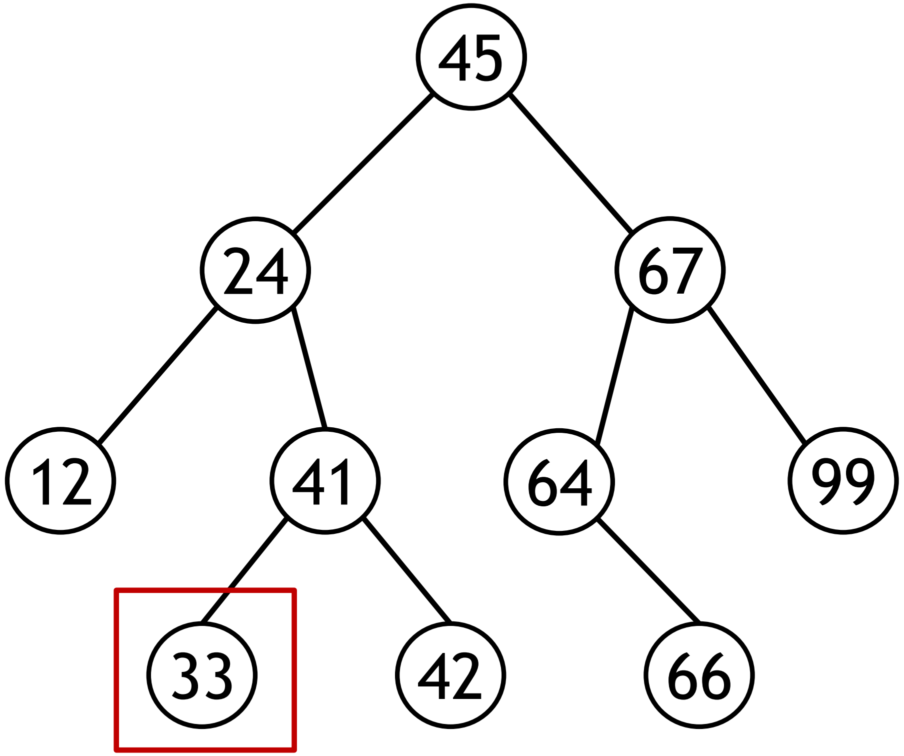


[12, 24

Example: Inorder Traversal

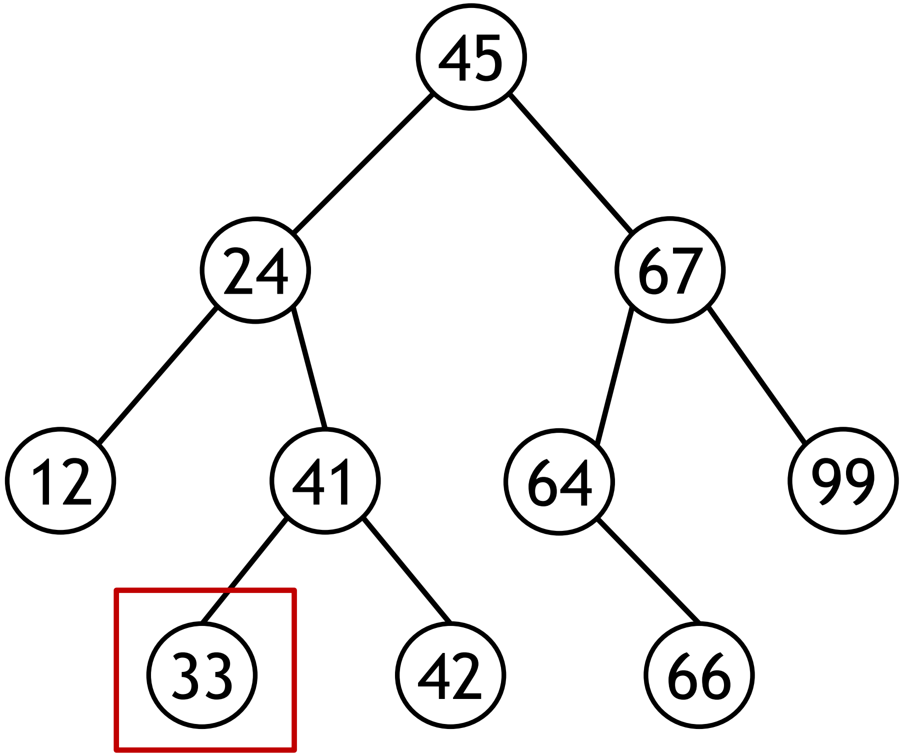
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```

[12, 24



Example: Inorder Traversal

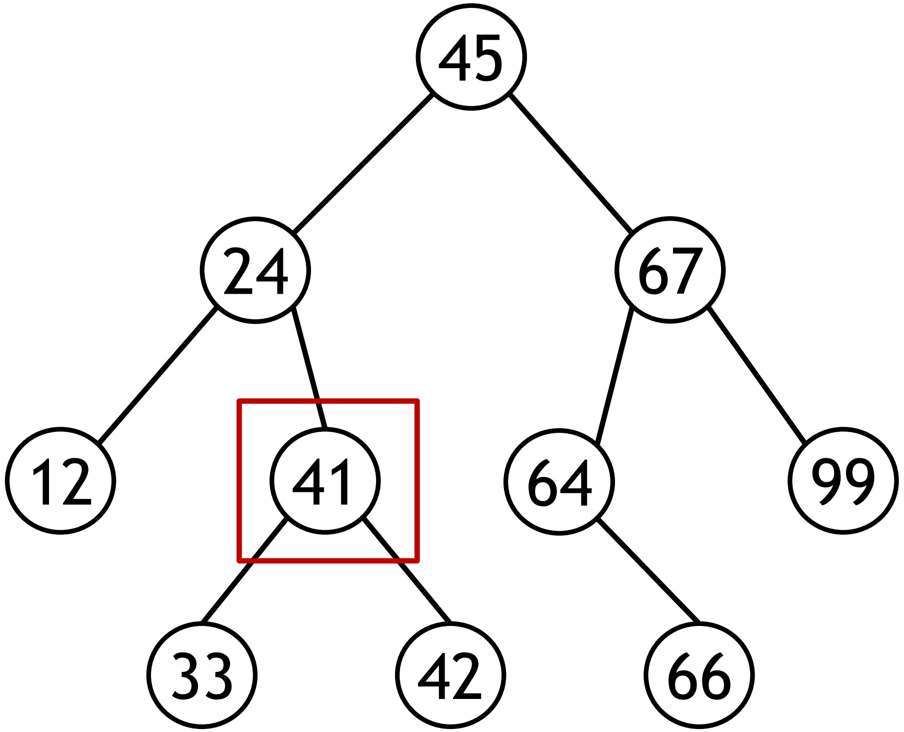
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33

Example: Inorder Traversal

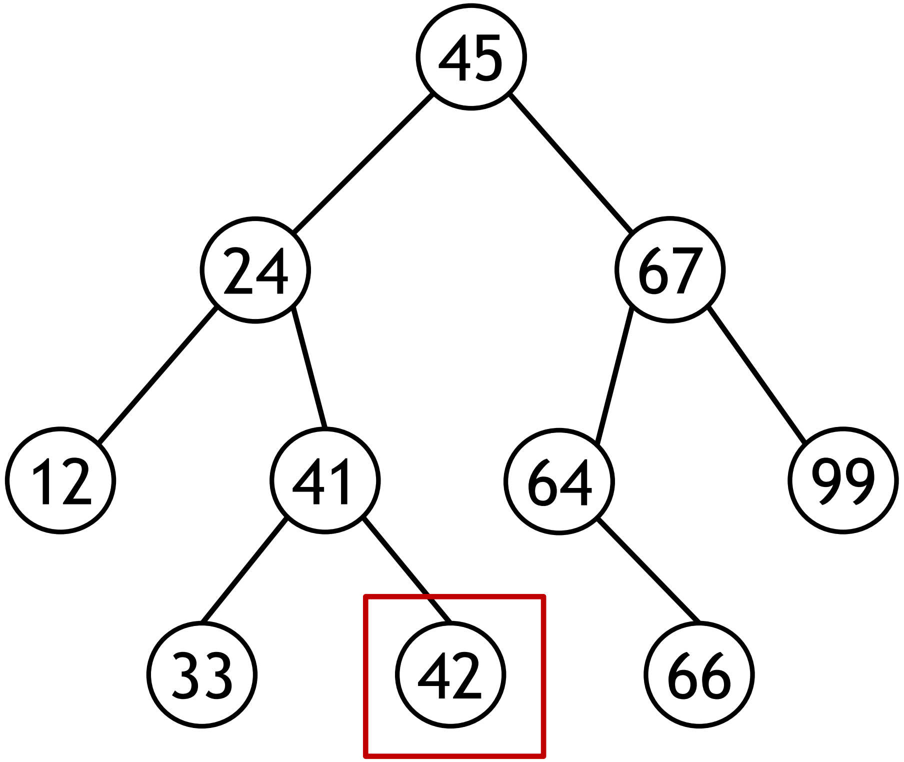
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41

Example: Inorder Traversal

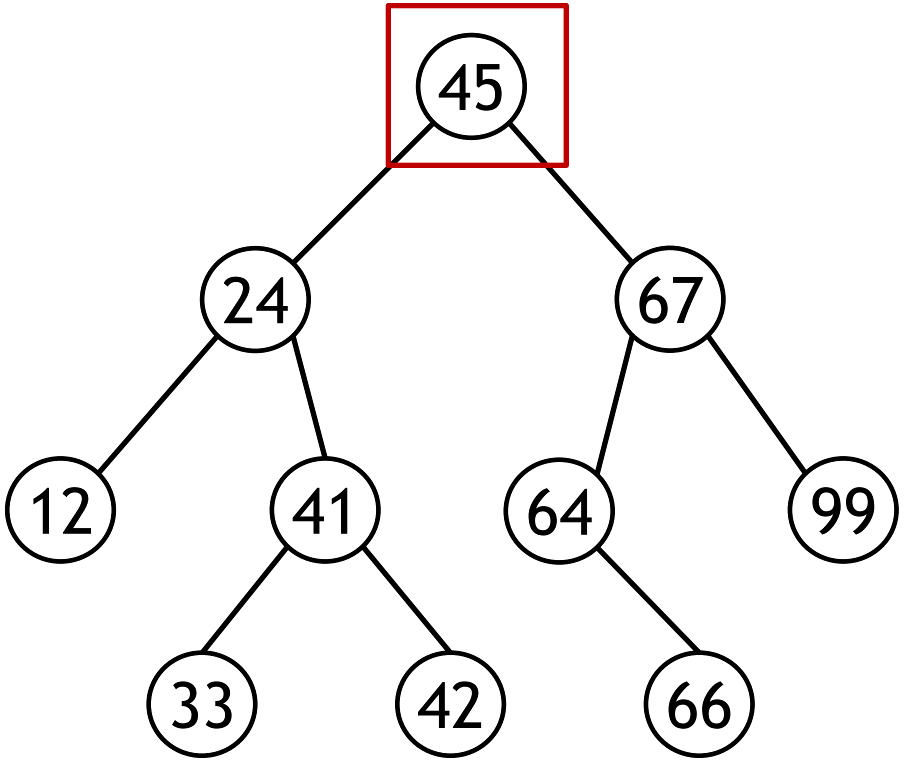
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42

Example: Inorder Traversal

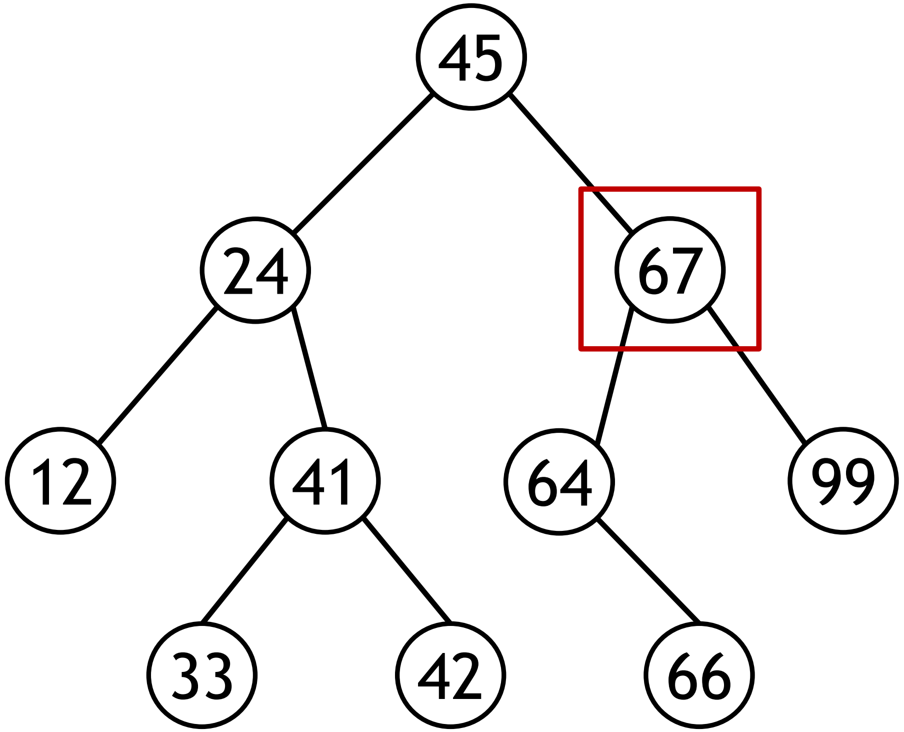
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45

Example: Inorder Traversal

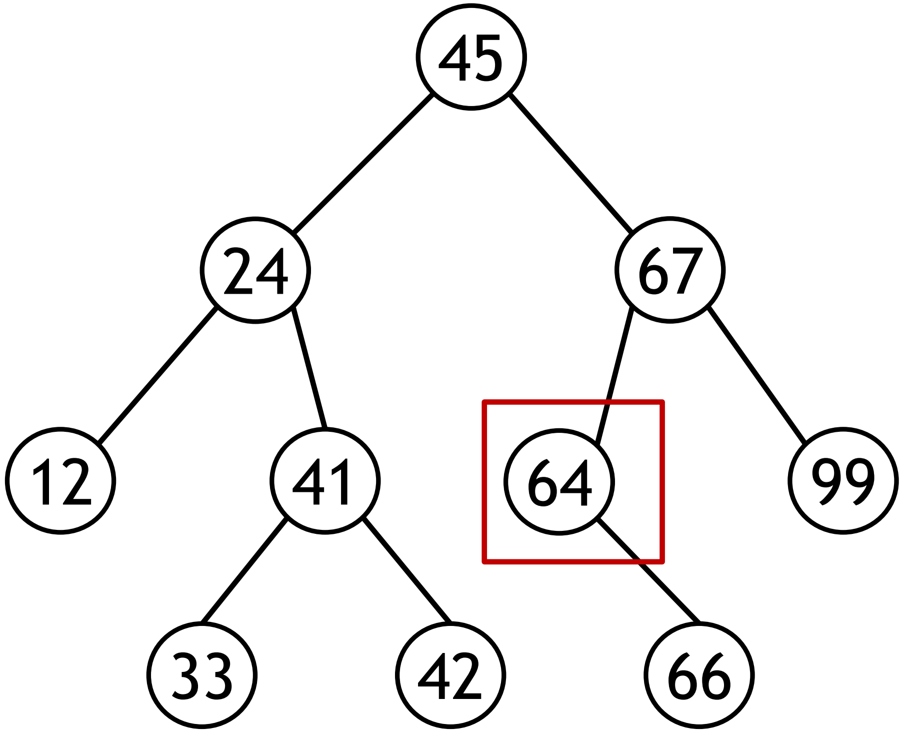
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45

Example: Inorder Traversal

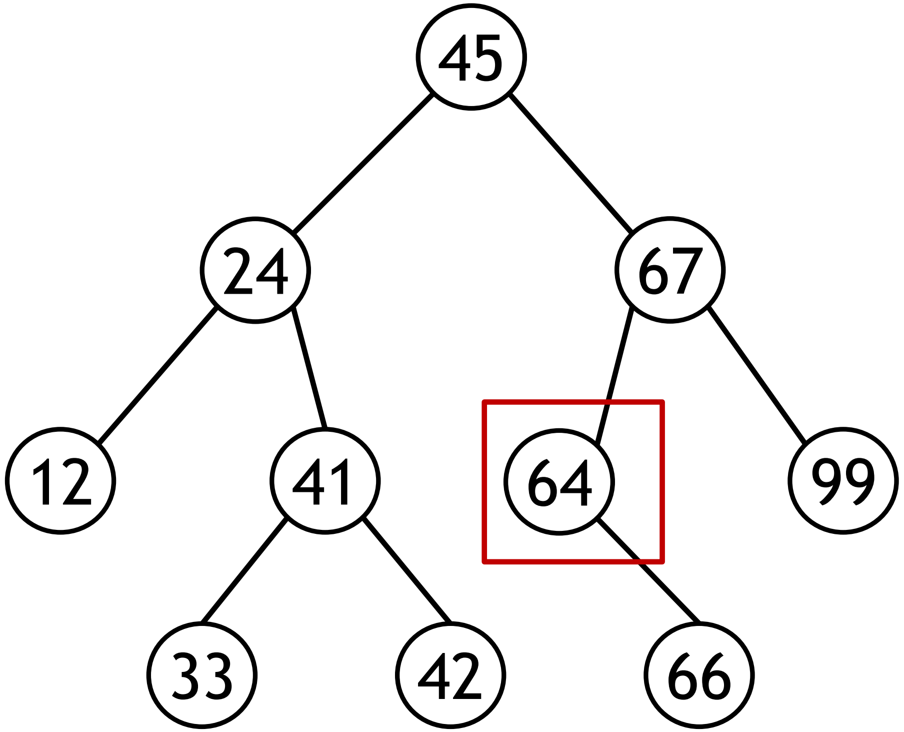
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45

Example: Inorder Traversal

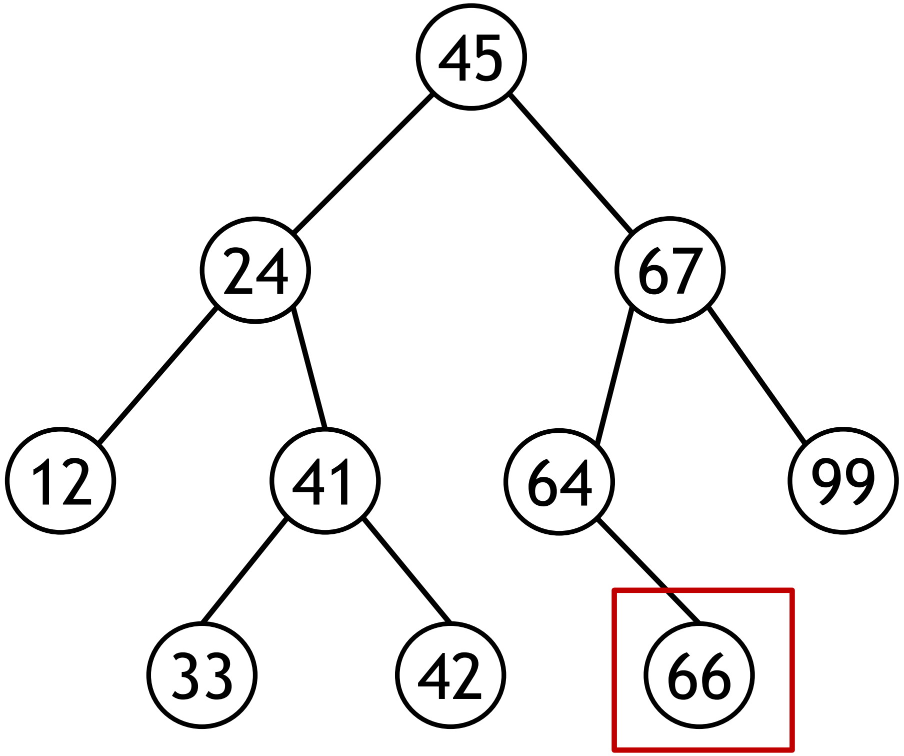
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45, 64

Example: Inorder Traversal

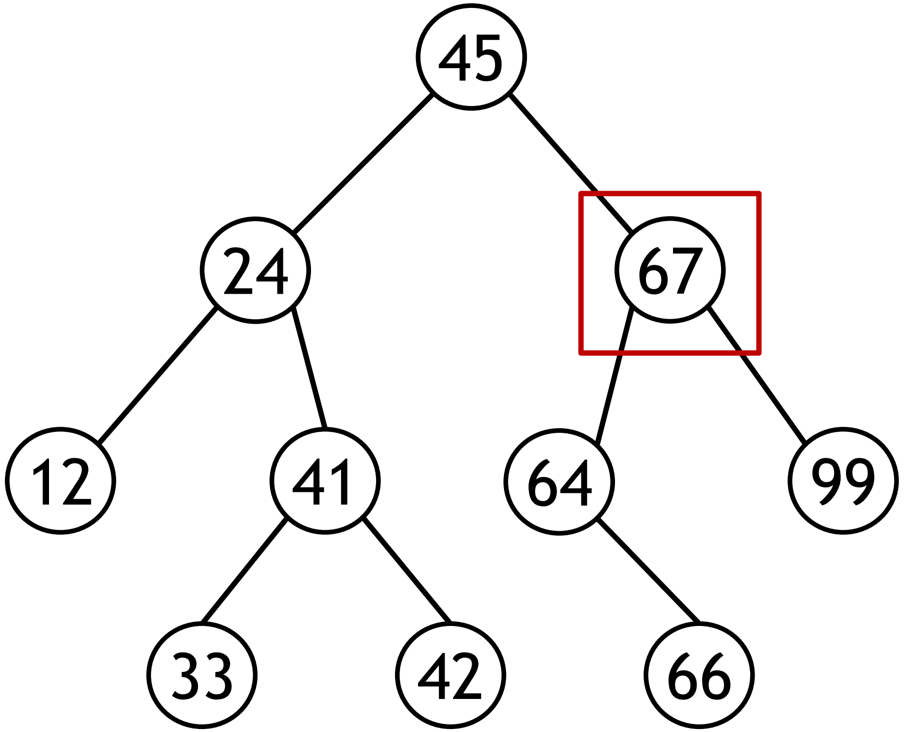
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45, 64, 66

Example: Inorder Traversal

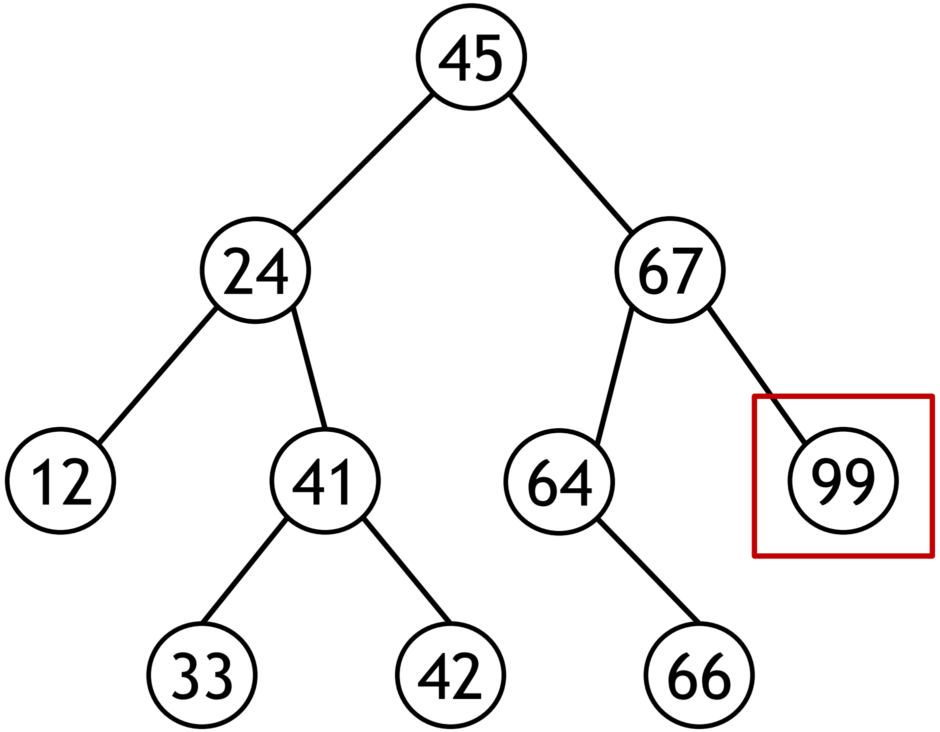
```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45, 64, 66, 67

Example: Inorder Traversal

```
inorderTraversal(x)
If x ≠ NULL
    inorderTraversal(x.left)
    print (x.key)
    inorderTraversal(x.right)
```



[12, 24, 33, 41, 42, 45, 64, 66, 67, 99]

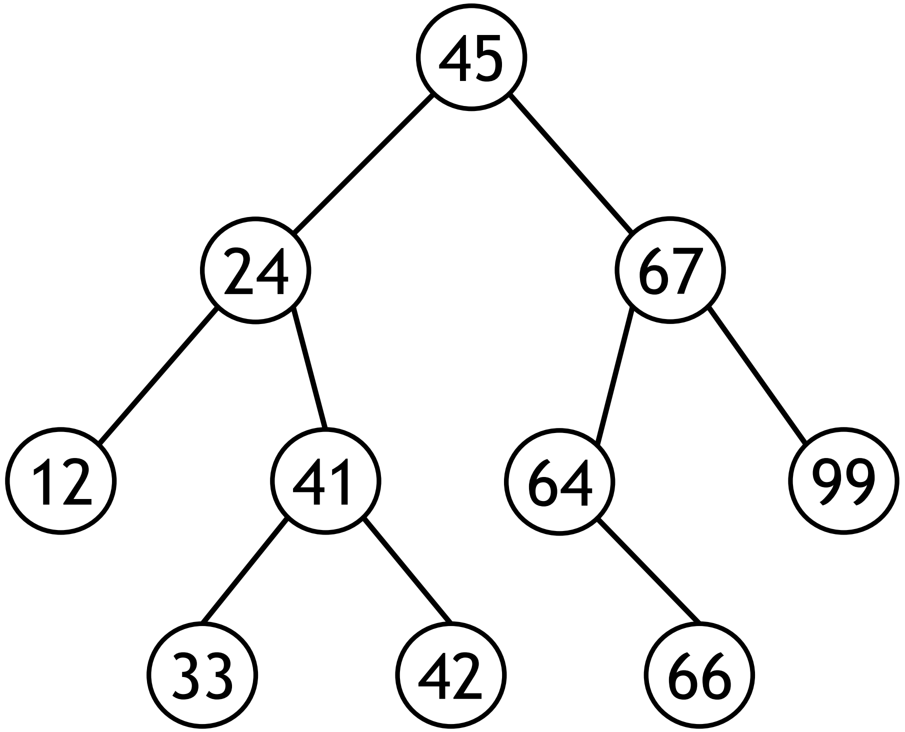
Preorder Traversal: Recuresive

- Process (display) the value in the node
- Traverse the left subtree preorder
- Traverse the right subtree preorder

```
preorderTraversal(x)
If x ≠ NULL
    print (x.key)
    preorderTraversal(x.left)
    preorderTraversal(x.right)
```

Exercise: Preorder Traversal

```
preorderTraversal(x)
If x ≠ NULL
    print (x.key)
    preorderTraversal(x.left)
    preorderTraversal(x.right)
```



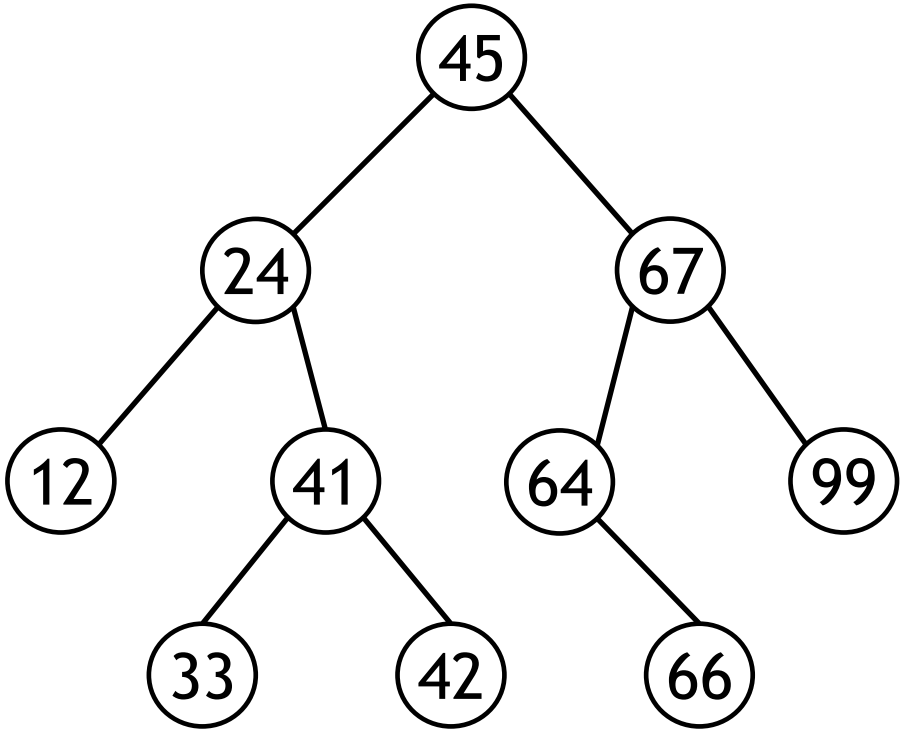
Postorder Traversal: Recuresive

- Traverse the left subtree postorder
- Traverse the right subtree postorder
- Process (display) the value in the node

```
postorderTraversal(x)
If x ≠ NULL
    postorderTraversal(x.left)
    postorderTraversal(x.right)
    print (x.key)
```

Exercise: Postorder Traversal

```
postorderTraversal(x)
If x ≠ NULL
    postorderTraversal(x.left)
    postorderTraversal(x.right)
    print (x.key)
```



Searching

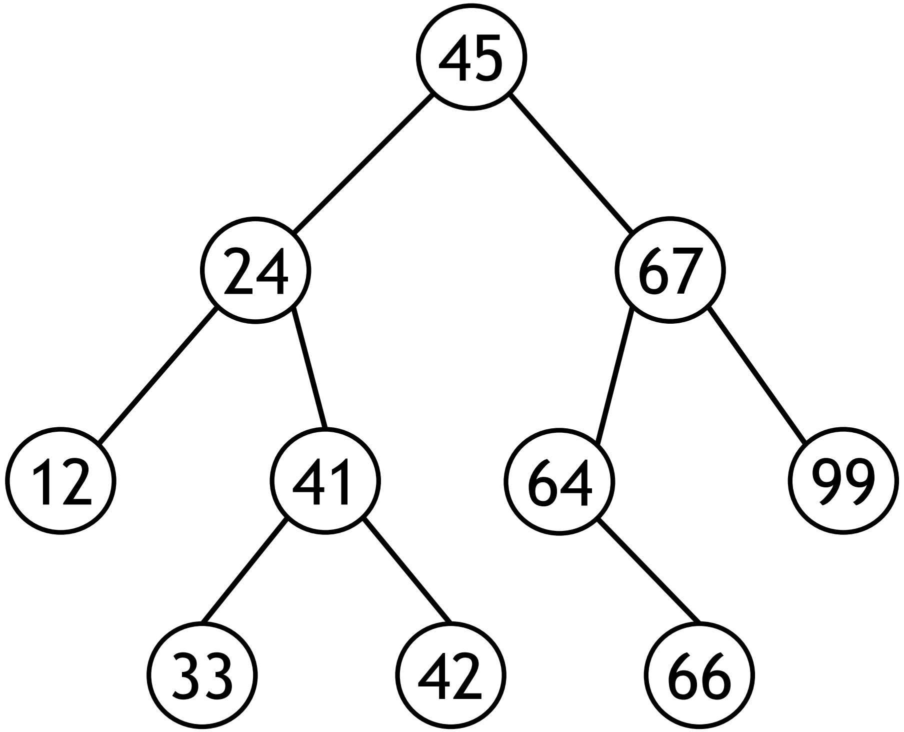
- Given a pointer to the root of the tree and a key, return a pointer to a node with key k if one exists; otherwise, return NULL

```
Search(x,k)
if x == NULL or k == x.key
    return x
if k < x.key
    return Search(x.left,k)
else
    return Search(x.right,k)
```

```
Search(x,k)
while x ≠ NULL and k ≠ x.key
    if k < x.key
        x = x.left
    else
        x = x.right
return x
```

Exercise: Searching

```
Search(x,k)
if x == NULL or k == x.key
    return x
if k < x.key
    return Search(x.left,k)
else
    return Search(x.right,k)
```

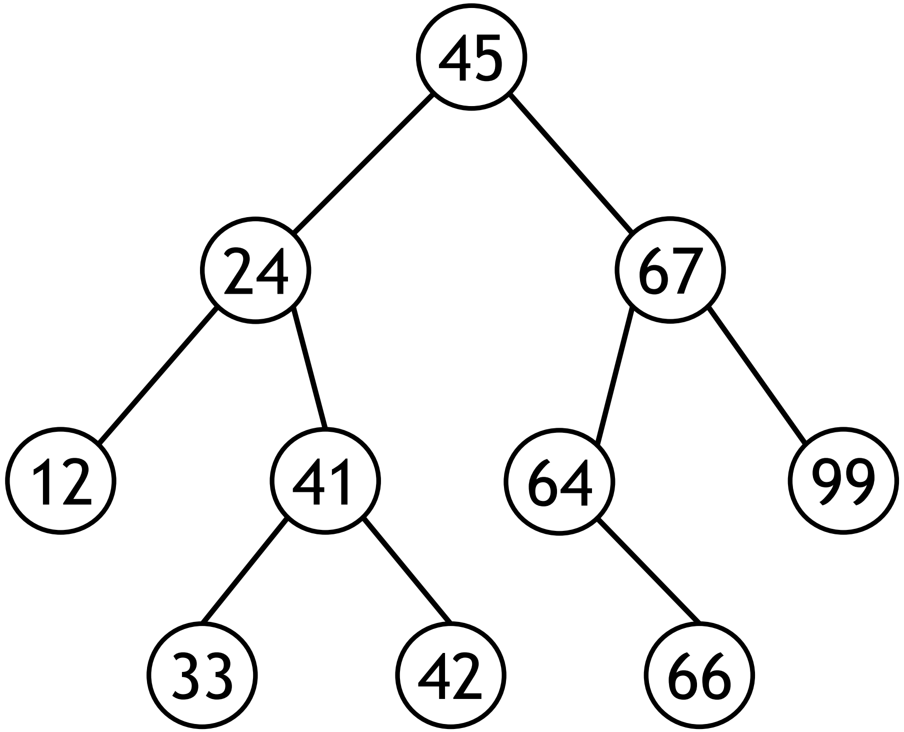


Search(root,33)

Minimum and Maximum

```
Minimum(x)
while x.left ≠ NULL
    x = x.left
return x
```

```
Maximum(x)
while x.right ≠ NULL
    x = x.right
return x
```



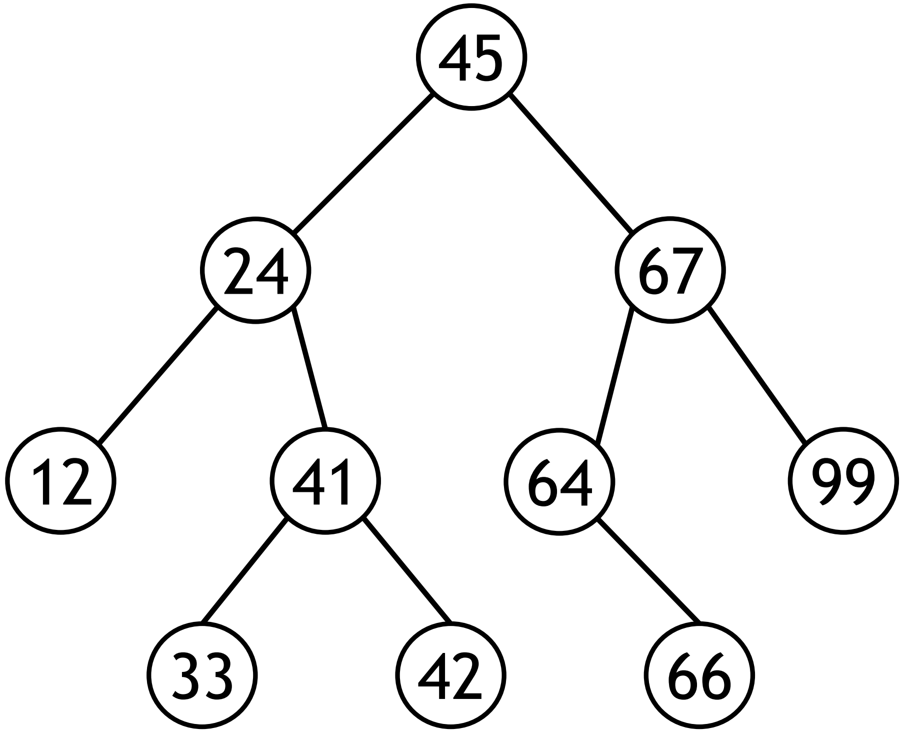
Successor

- If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key.

```
Successor(x)
if x.right ≠ NULL
    return Minimum(x.right)
y=x.parent
while y ≠ NULL and x ==y.right
    x = y
    y = y.parent
return y
```

Successor

```
Successor(x)
if x.right ≠ NULL
    return Minimum(x.right)
y=x.parent
while y ≠ NULL and x ==y.right
    x = y
    y = y.parent
return y
```



Find Successor of the node with:

Key 45

Key 42

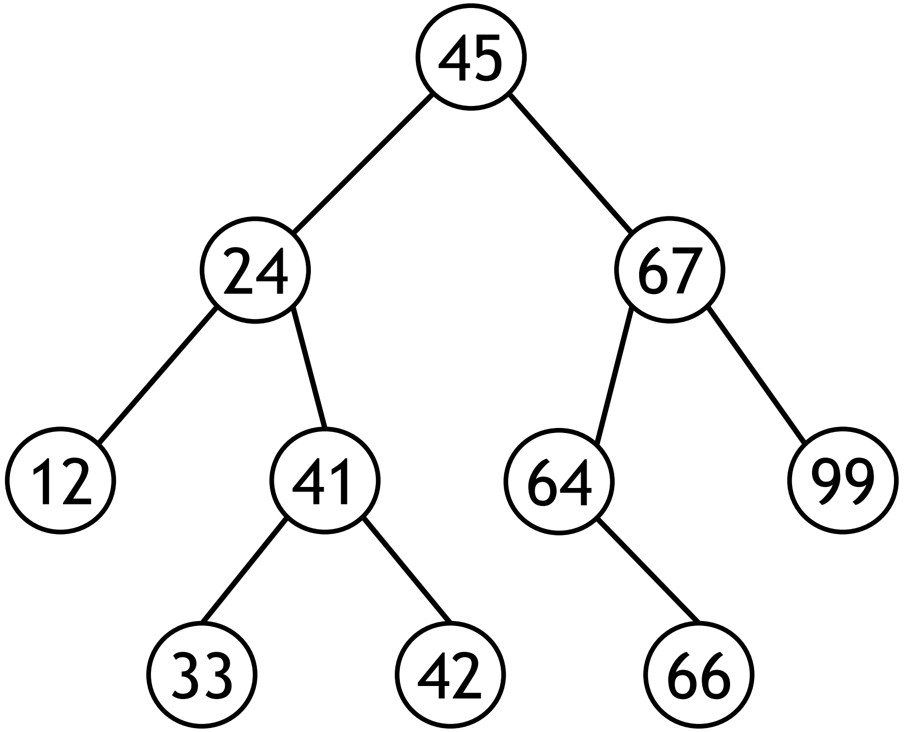
Key 66

Exercise: Predecessor

- The Predecessor of a node x is the node with the largest key smaller than x.key.

Insertion

```
Insert(T,z)
y=NULL
x=T.root
While x ≠ NULL
    y=x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.parent = y
if y == NULL
    T.root = z
elseif z.key < y.key
    y.left = z
else
    y.right = z
```



Insert 65 ?
Insert 32 ?

Deletion

- The overall strategy for deleting a node z from a binary search tree T has three basic cases
  - If z has no children, then we simply remove it by modifying its parent to replace z with NULL as its child.
  - If z has just one child, then elevate that child to take z's position in the tree by modifying z's parent to replace z by z's child.
  - If z has two child, then we find z's successor y and have y take z's position in the tree.

Deletion

- To move subtrees around within the binary search tree, we define a function Transplant to replace one subtree as a child of its parent with another subtree.
- Replace the subtree rooted at node u with the subtree rooted at node v.

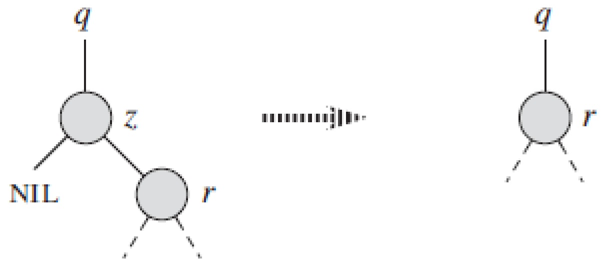
```
Transplant(T,u,v)
if u.parent = NULL
    T.root = v
elseif u==u.parent.left
    u.parent.left = v
else
    u.parent.right = v
if v ≠ NULL
    v.parent = u.parent
```

Deletion

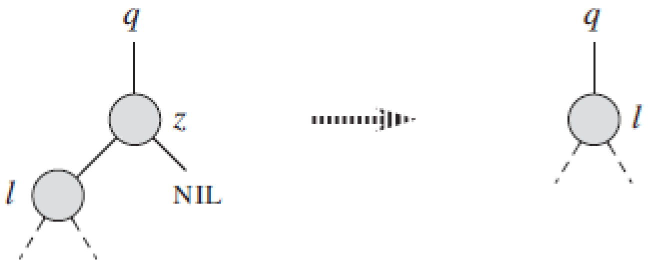
- Delete node z from binary search Tree T:

```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```

(a)

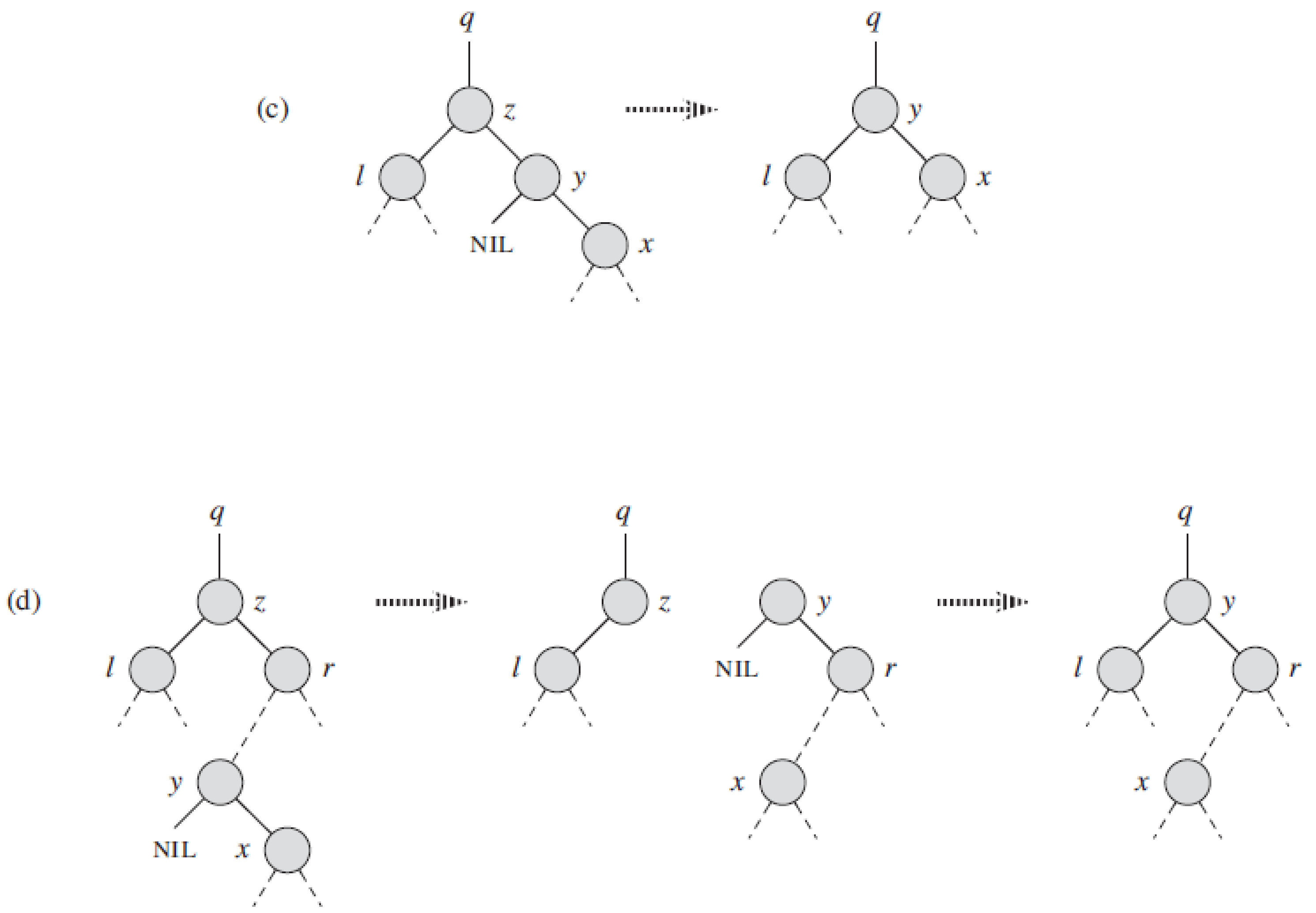


(b)



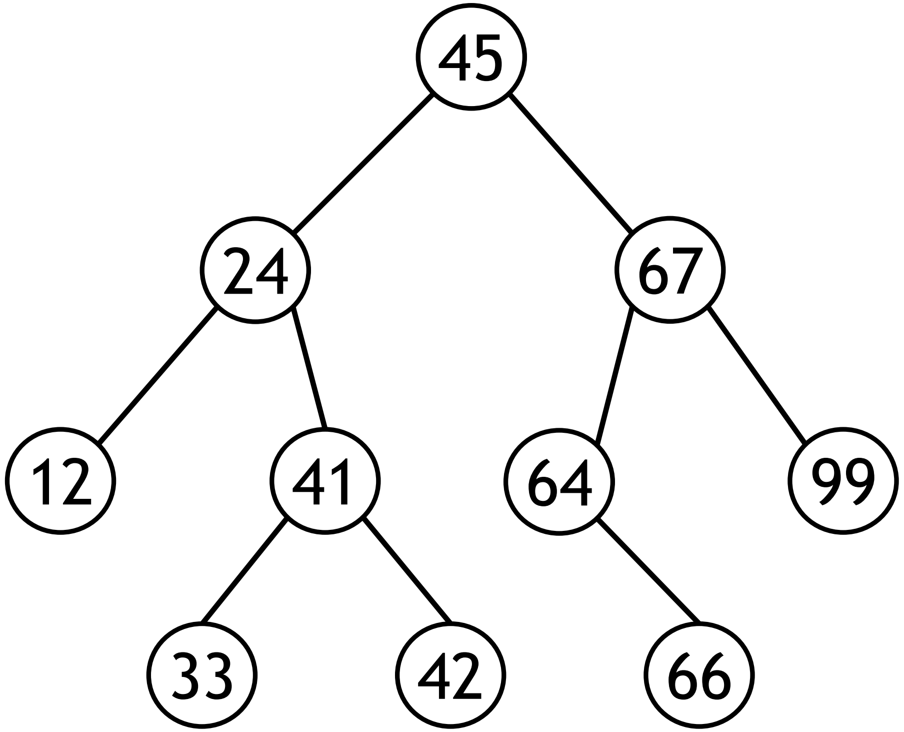
Deletion

```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



Exercise: Deletion

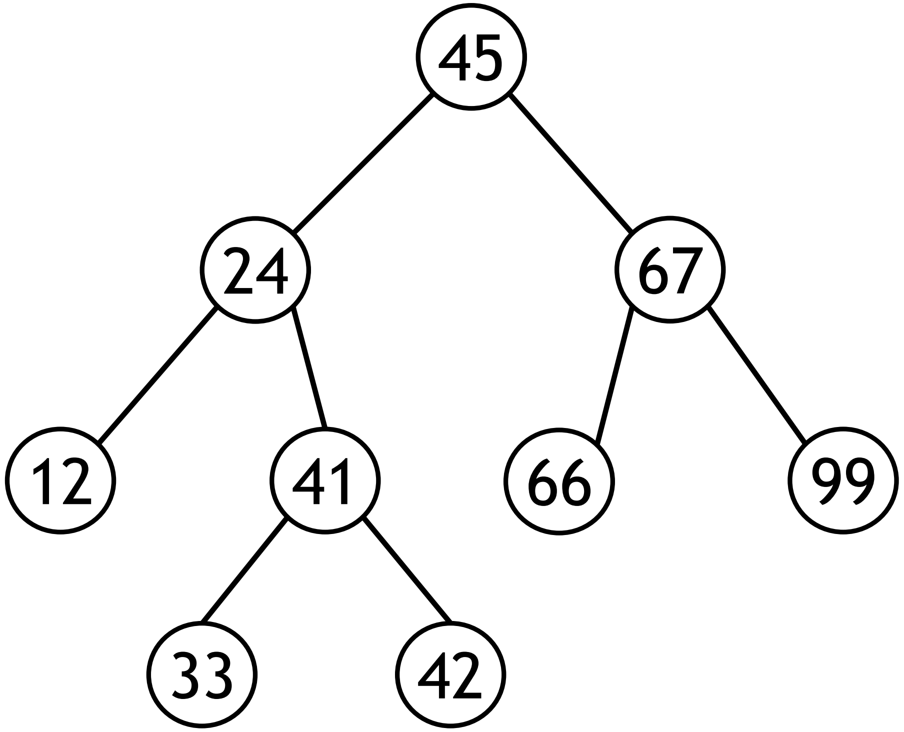
```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



Delete 64 ?

Exercise: Deletion

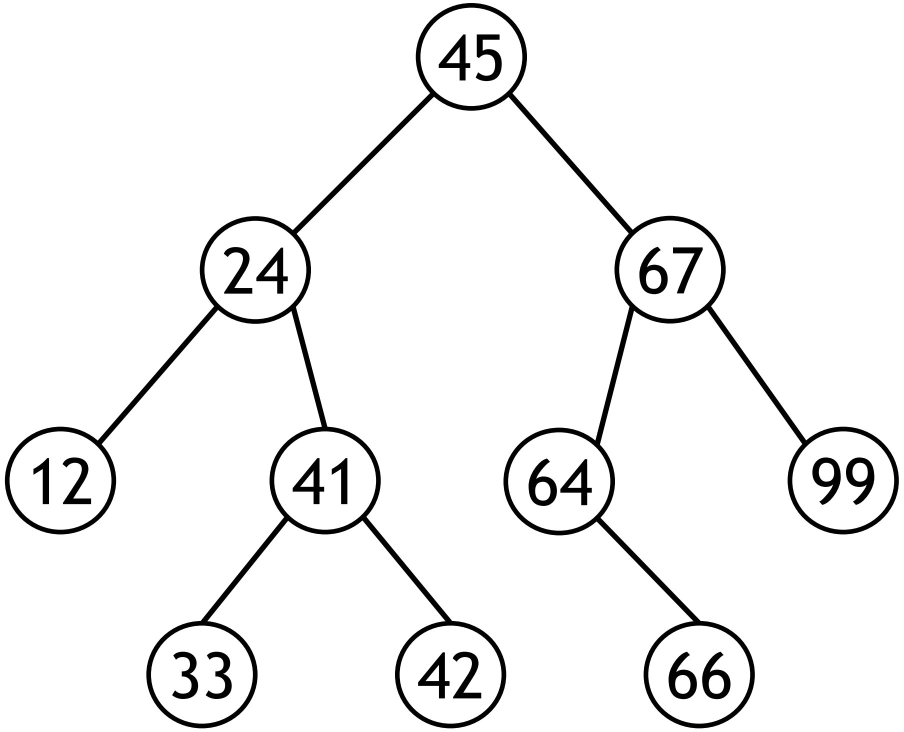
```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



After deletion

Exercise: Deletion

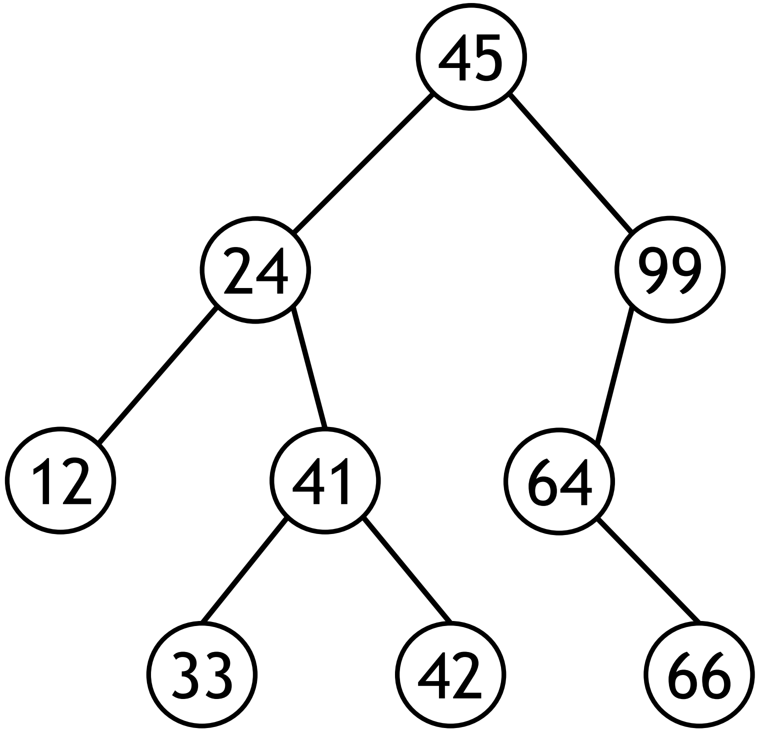
```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



Delete 67 ?

Exercise: Deletion

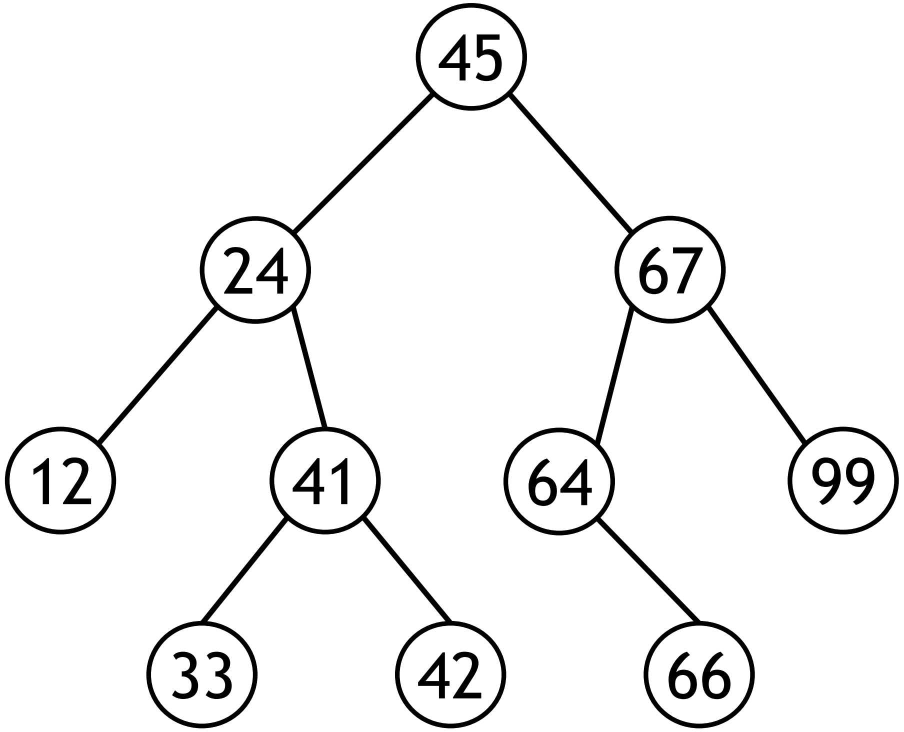
```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



After deletion

Exercise: Deletion

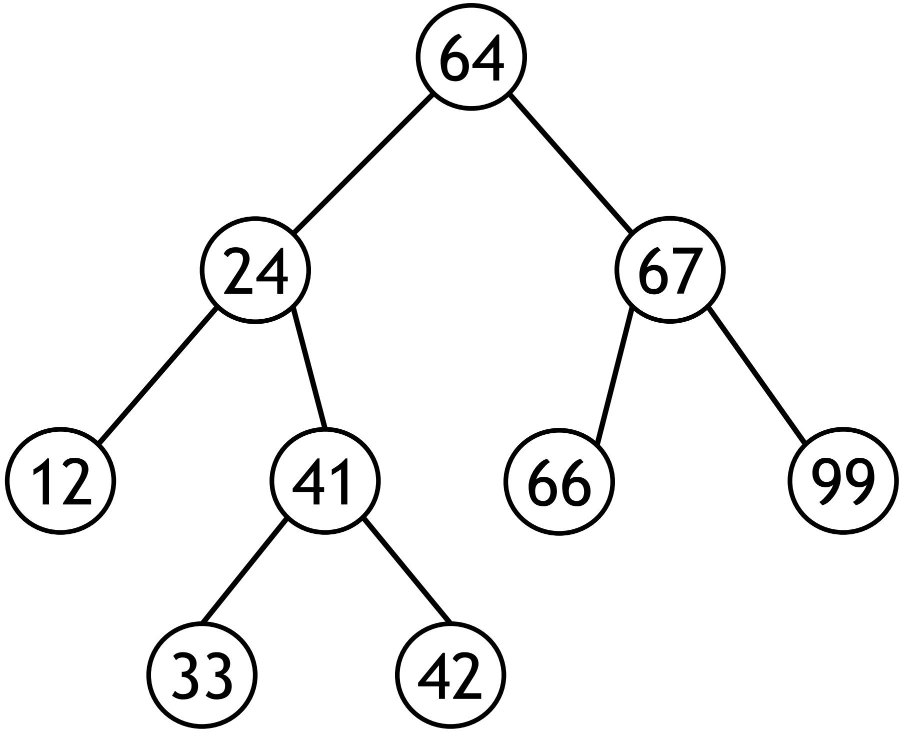
```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



Delete 45?

Exercise: Deletion

```
Delete(T,z)
If z.left == NULL
    Transplant(T,z,z.right)
elseif z.right == NULL
    Transplant(T,z,z.left)
else
    y = minimum(z.right)
    if y.parent ≠ z
        Transplant(T,y,y.right)
        y.right = z.right
        y.right.parent = y
    Transplant(T,z,y)
    y.left = z.left
    y.left.parent = y
```



After deletion

Implement Binary Search Tree with Python

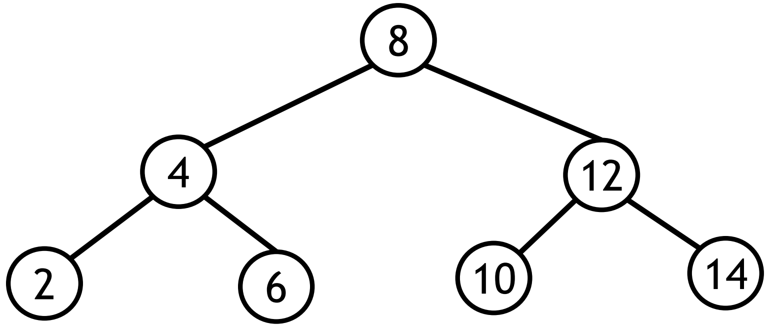
- inorderTraversal
- preorderTraversal
- postorderTraversal
- Minimum
- Maximum
- Search
- Successor
- Insert
- Transplant
- delete

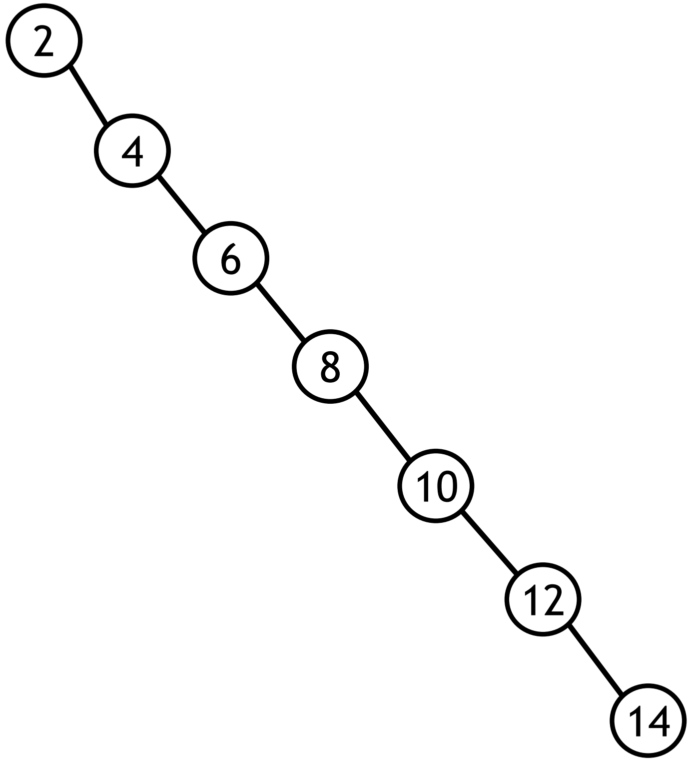
Balanced Trees

Binary Search Tree

- BST operations are O(h), where h is the height of the tree
- The Best case running time of BST operations is O(logn)
- The worst case running time is O(n)
  - What happens when you insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - Unbalanced degenerate tree
  - Problem: lack of "balance"

Balanced and unbalanced BST

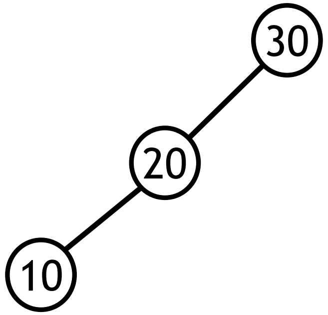




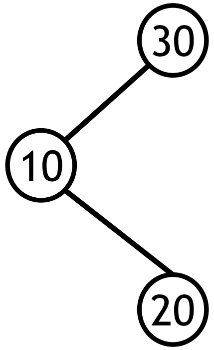
Insert keys with different orders

Keys: 10, 20, 30

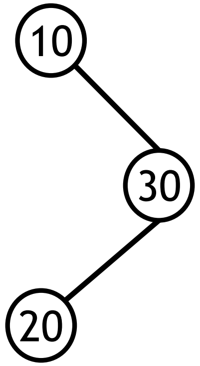
Order: 30, 20, 10



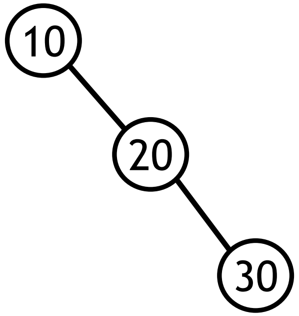
30, 10, 20



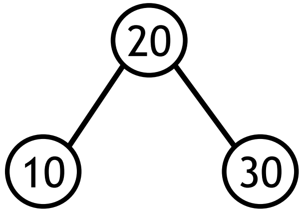
10, 30, 20



10, 20, 30



20, 10, 30
20, 30, 10



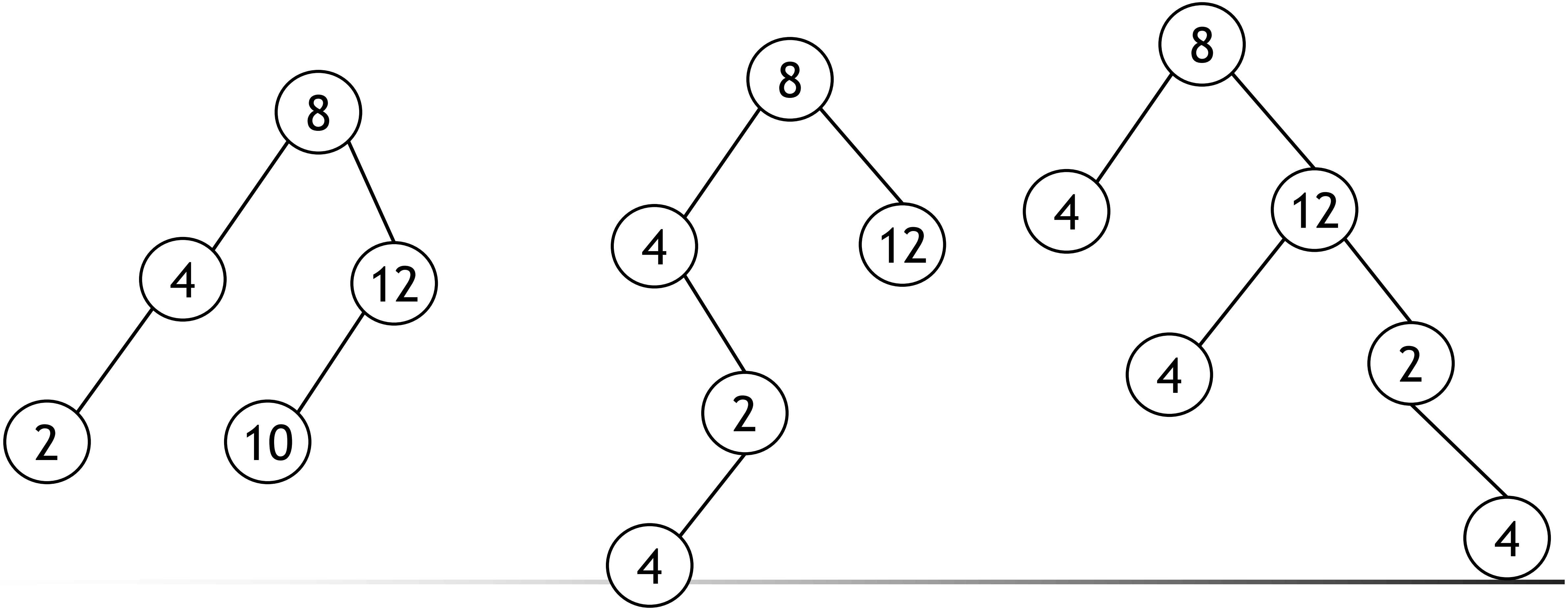
AVL Tree

- AVL tree got its name after its inventor Adelson-Velsky and Landis.
- AVL tree is height-balanced binary search tree

- Balance factor of a node
  - Balance Factor = (Height of Left Subtree - Height of Right Subtree)
  - The self balancing property of an AVL tree is maintained by the balance factor.
- The value of balance factor should always be -1, 0 or +1.

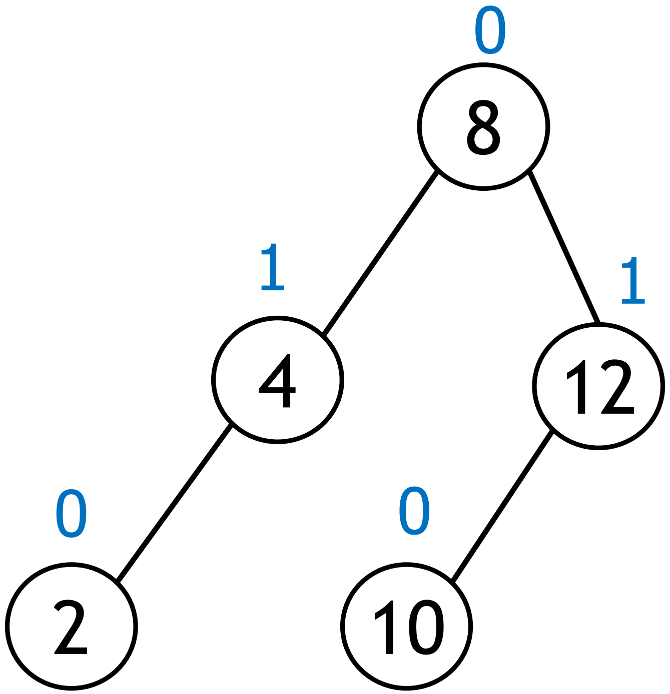
AVL Tree

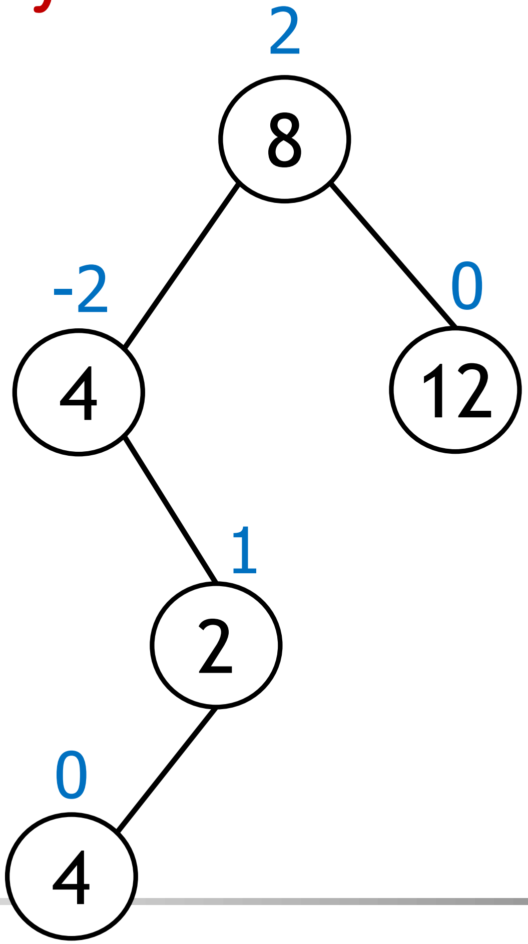
- Balance factor = height of left subtree – height of right subtree
- Balance factor = {-1, 0, 1}

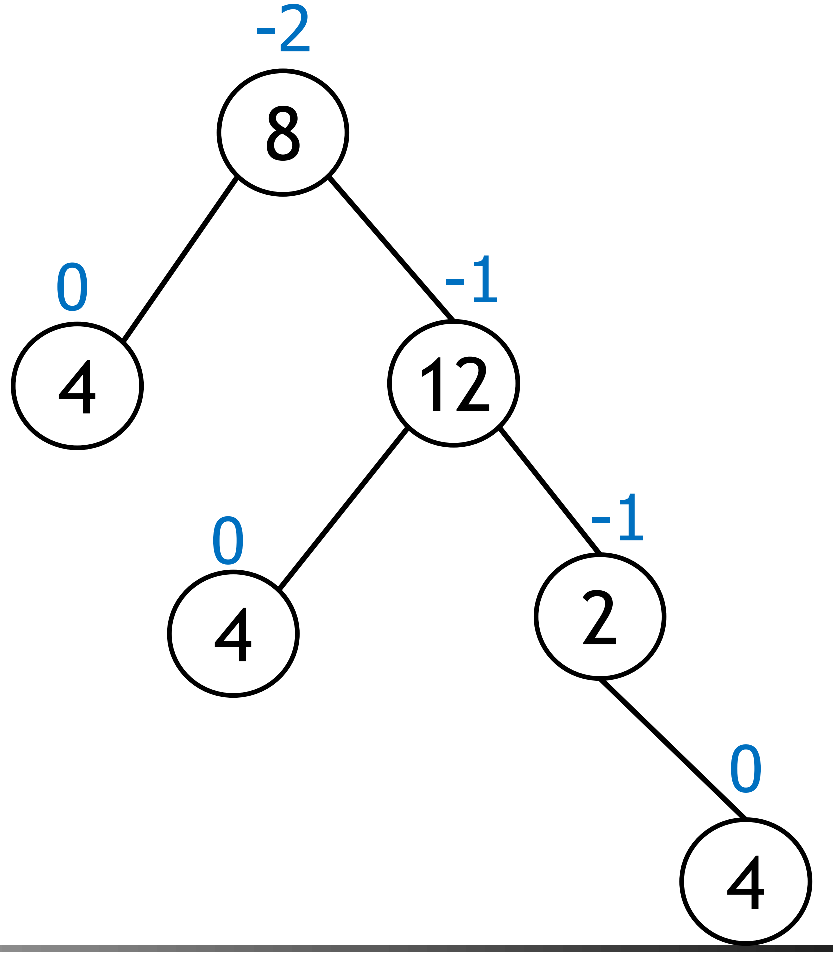


AVL Tree

- Balance factor = height of left subtree – height of right subtree
- Balance factor = {-1, 0, 1}





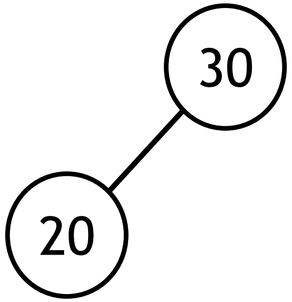


Operations on an AVL tree

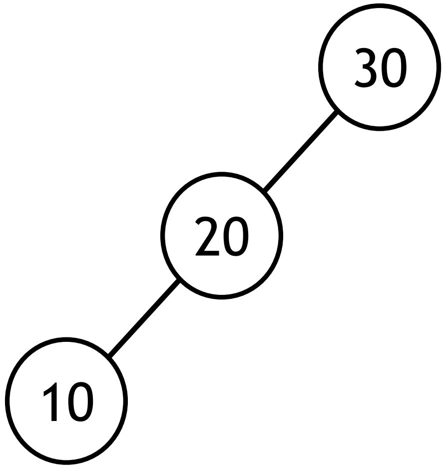
- Rotation
- Insertion
- Deletion

Single Rotation

Initially

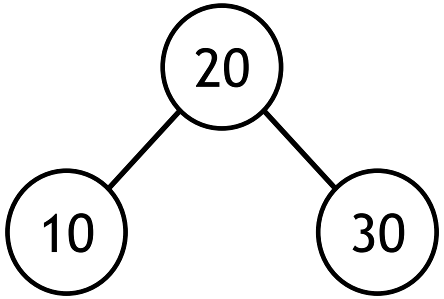


Insert 10



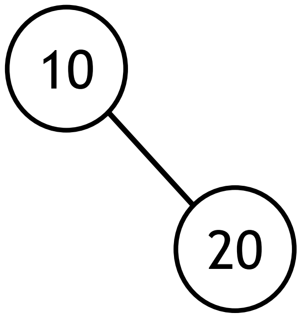
imbalance

After Right Rotation

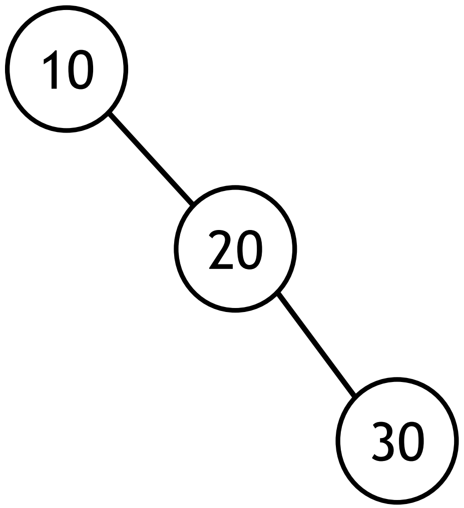


Single Rotation

Initially

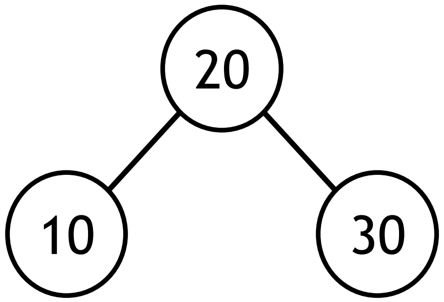


Insert 30

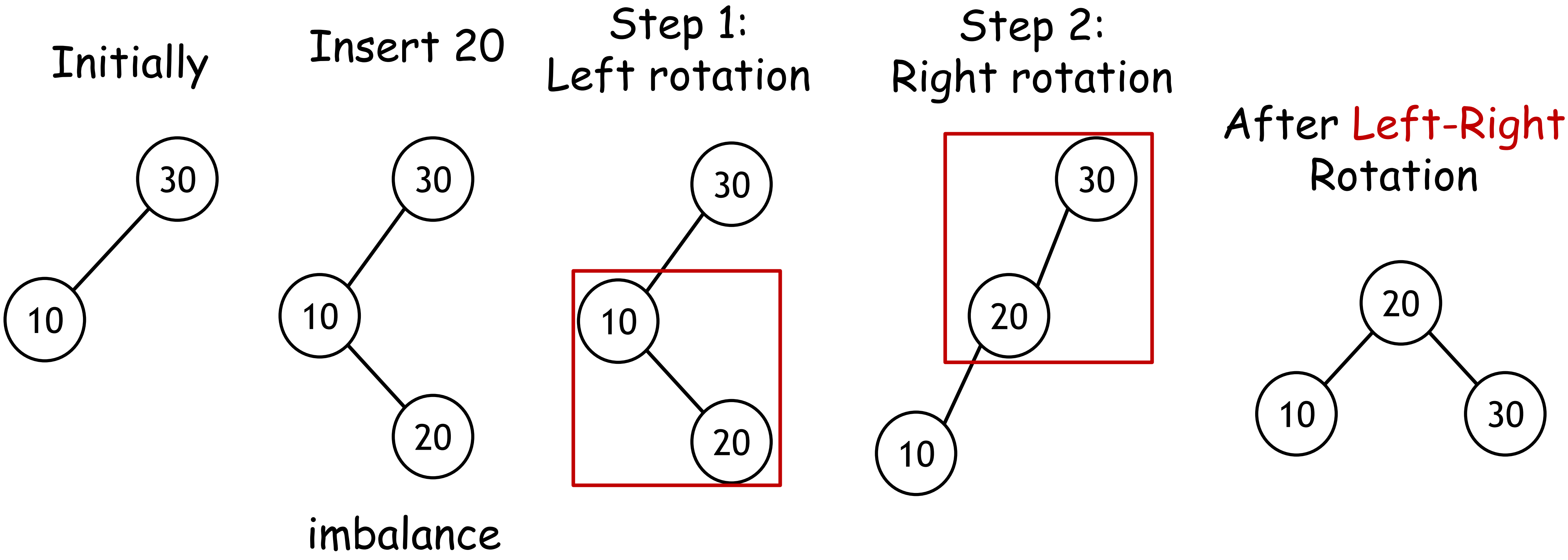


imbalance

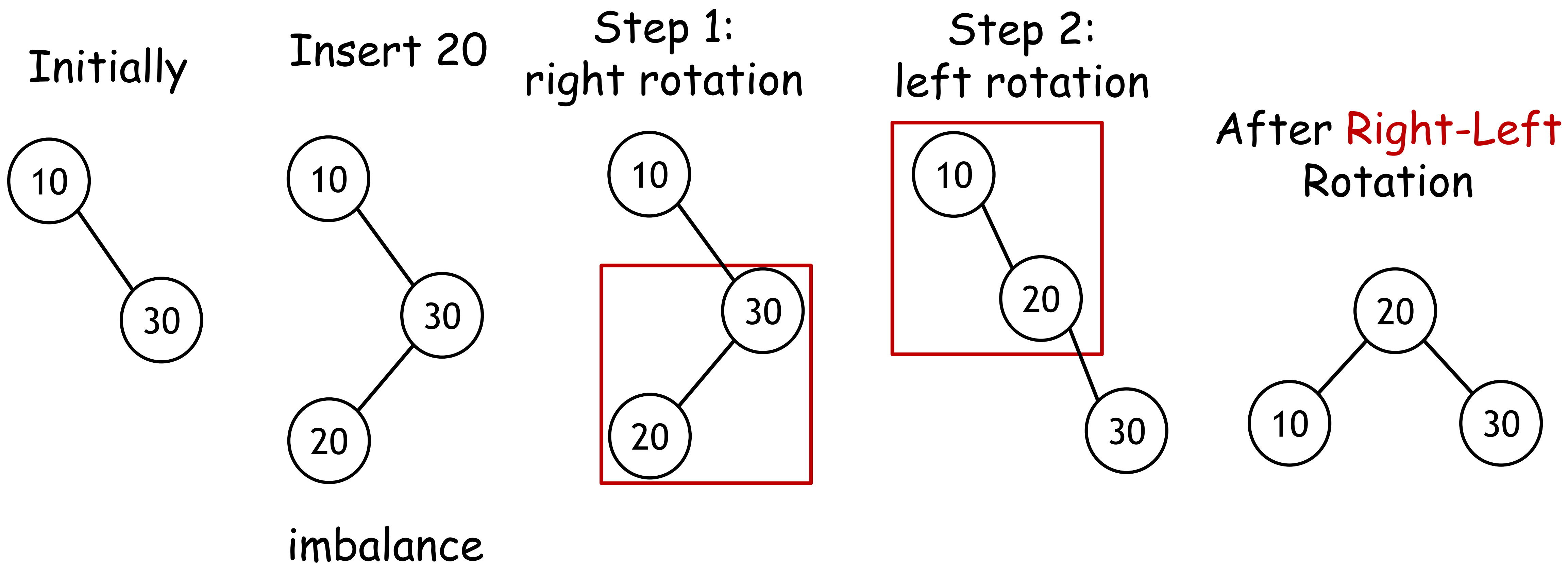
After Left Rotation



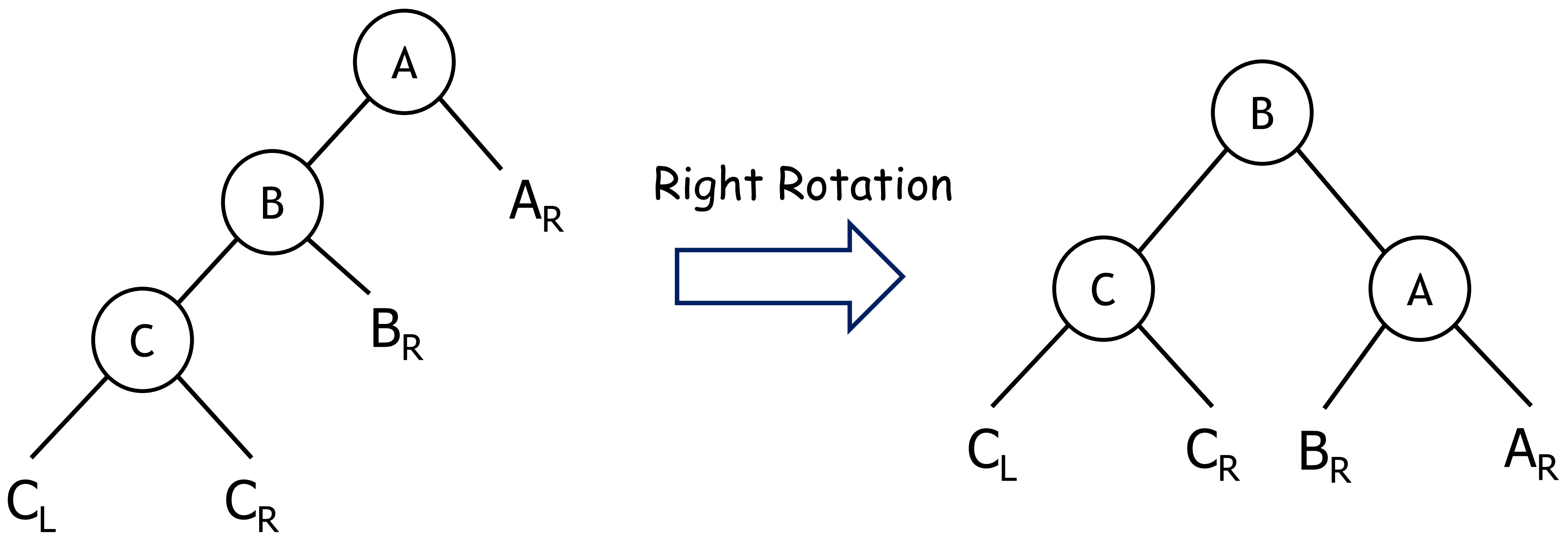
Double Rotation



Double Rotation



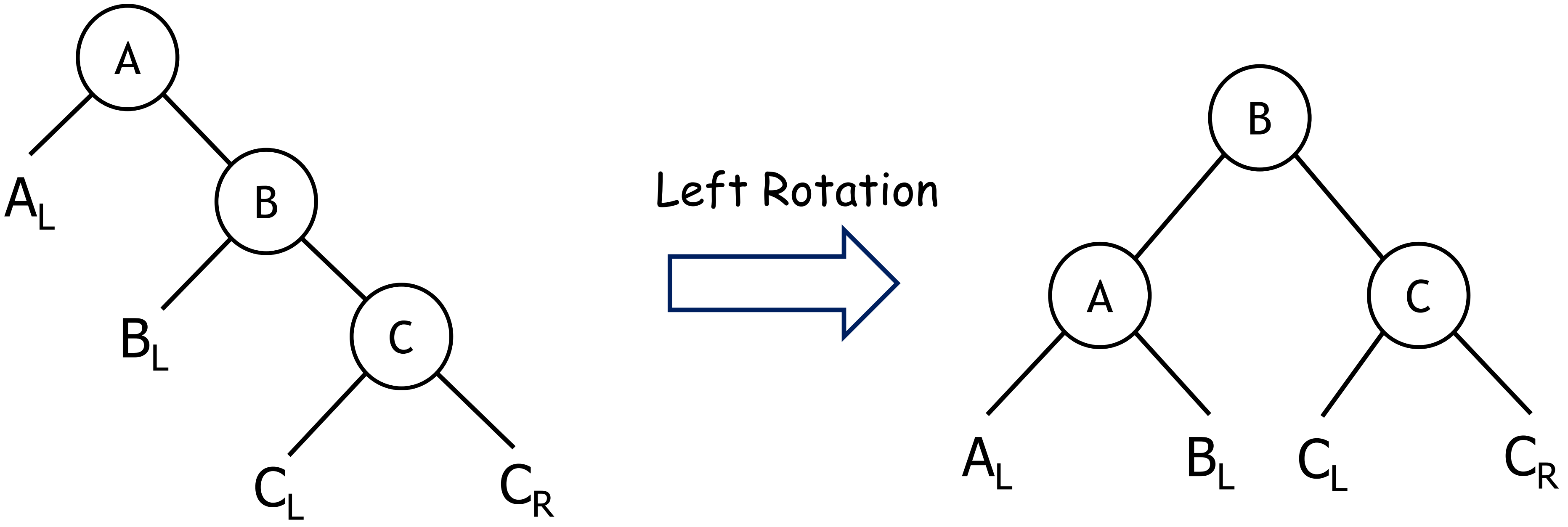
Rotations



Maintain BST property:

- C_L,C_R,A_R are still the subtrees of their original parent
- For all values v in B_R : $B < B_R < A$, so B_R become the new left subtree of A

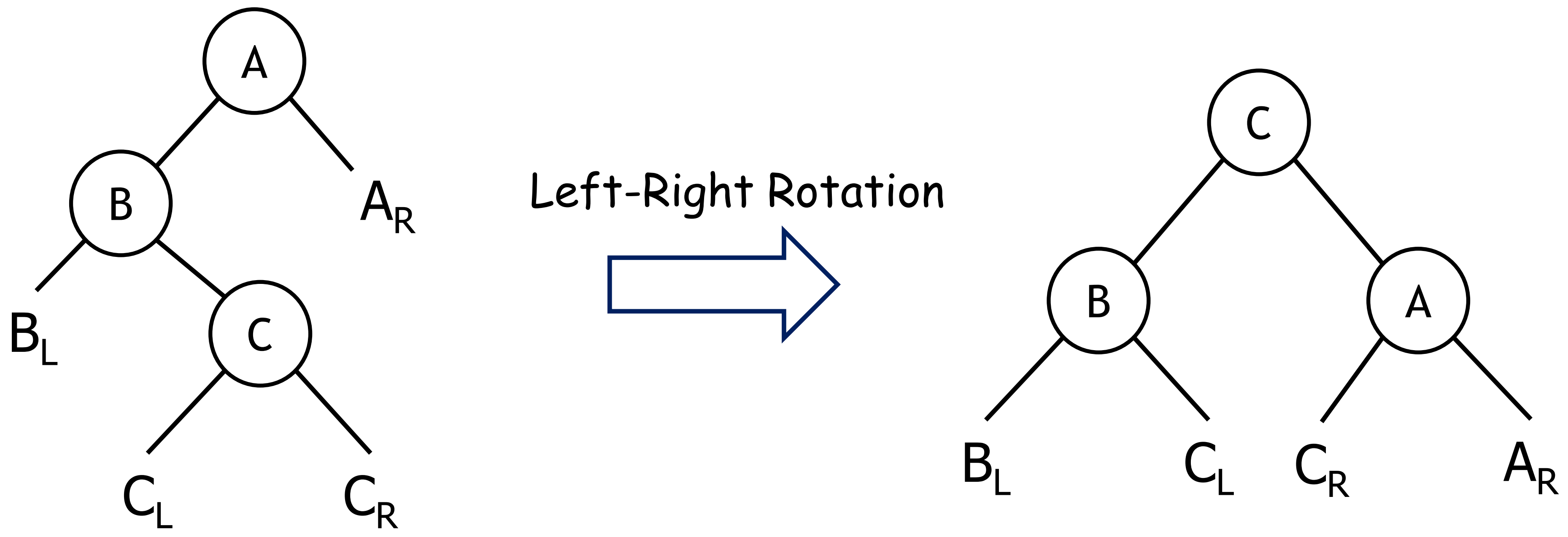
Rotations



Maintain BST property:

- C_L,C_R,A_L are still the subtrees of their original parent
- For all values v in B_L : $A < B_L < B$, so B_L become the new right subtree of A

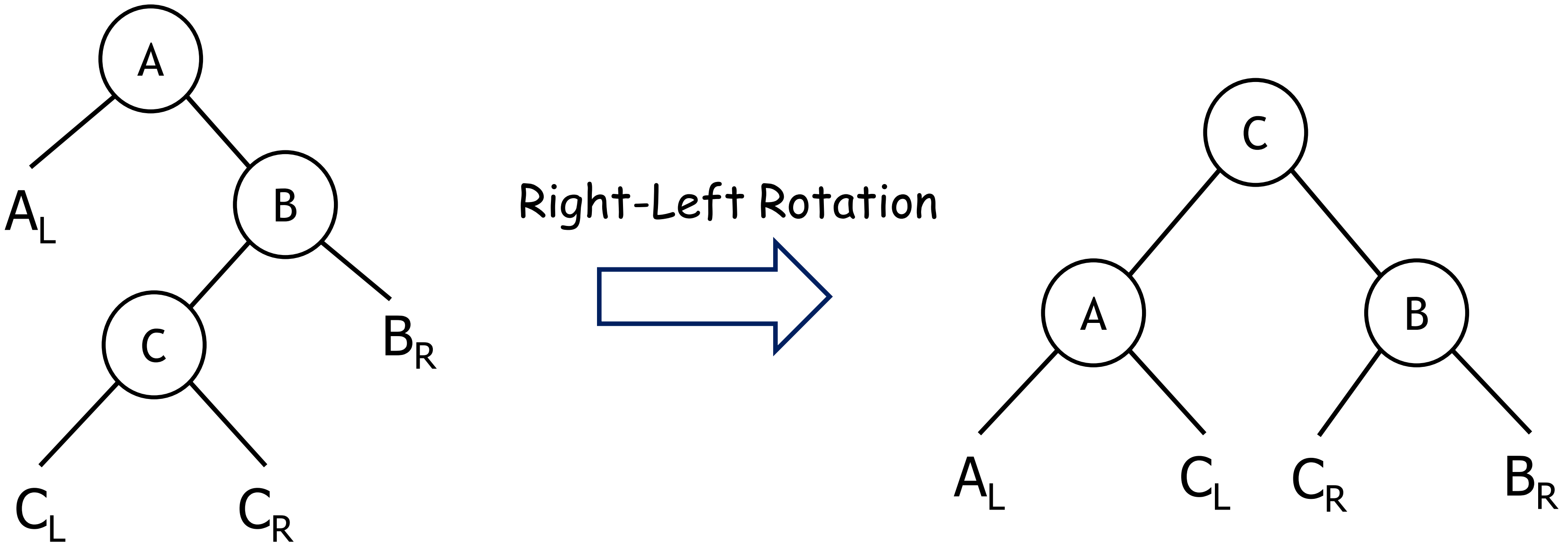
Rotations



Maintain BST property:

- B_L,A_R are still the subtrees of their original parent
- For all values v_1 in C_L : $B < C_L < C$, so C_L become the new right subtree of B
- For all values v_2 in C_R : $C < C_R < A$, so C_R become the new left subtree of A

Rotations

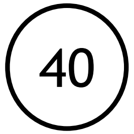


Maintain BST property:

- A_L,B_R are still the subtrees of their original parent
- For all values v_1 in C_L : $A < C_L < C$, so C_L become the new right subtree of A
- For all values v_2 in C_R : $C < C_R < B$, so C_R become the new left subtree of B

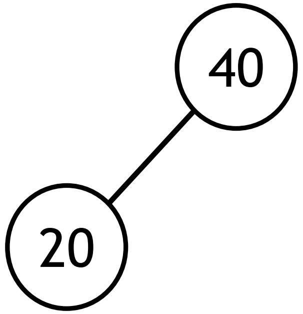
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



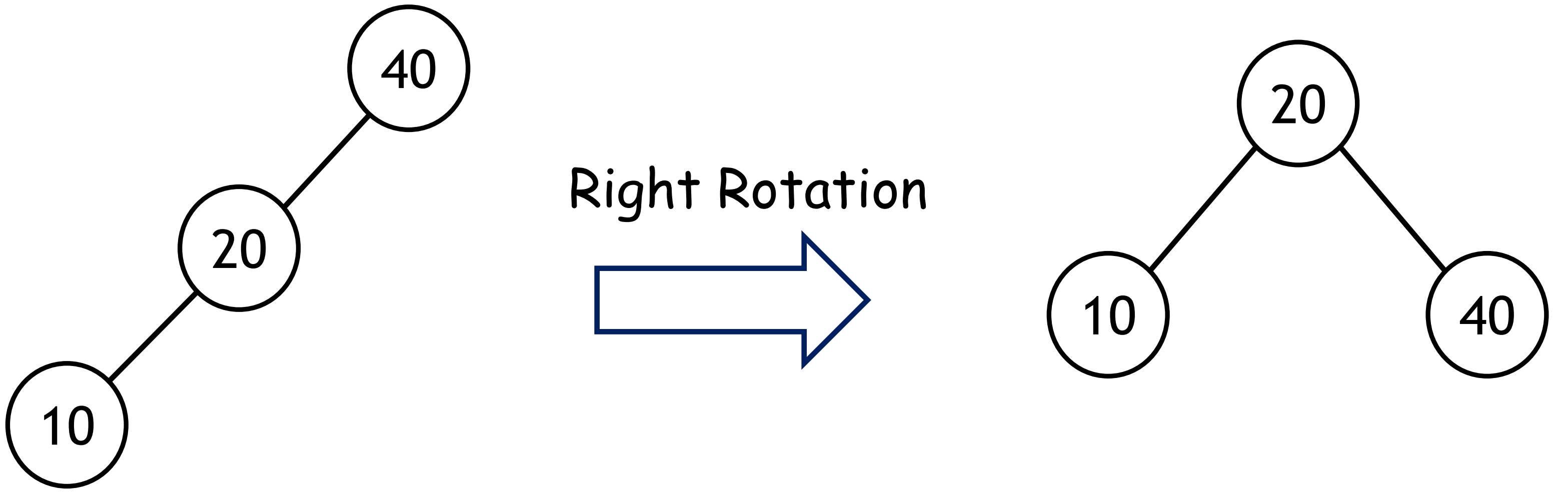
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



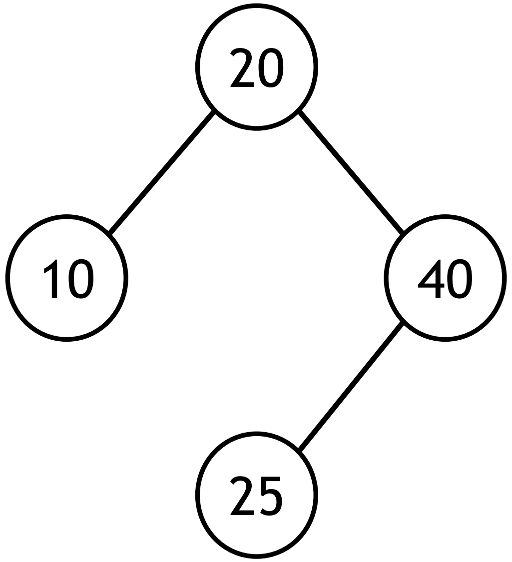
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



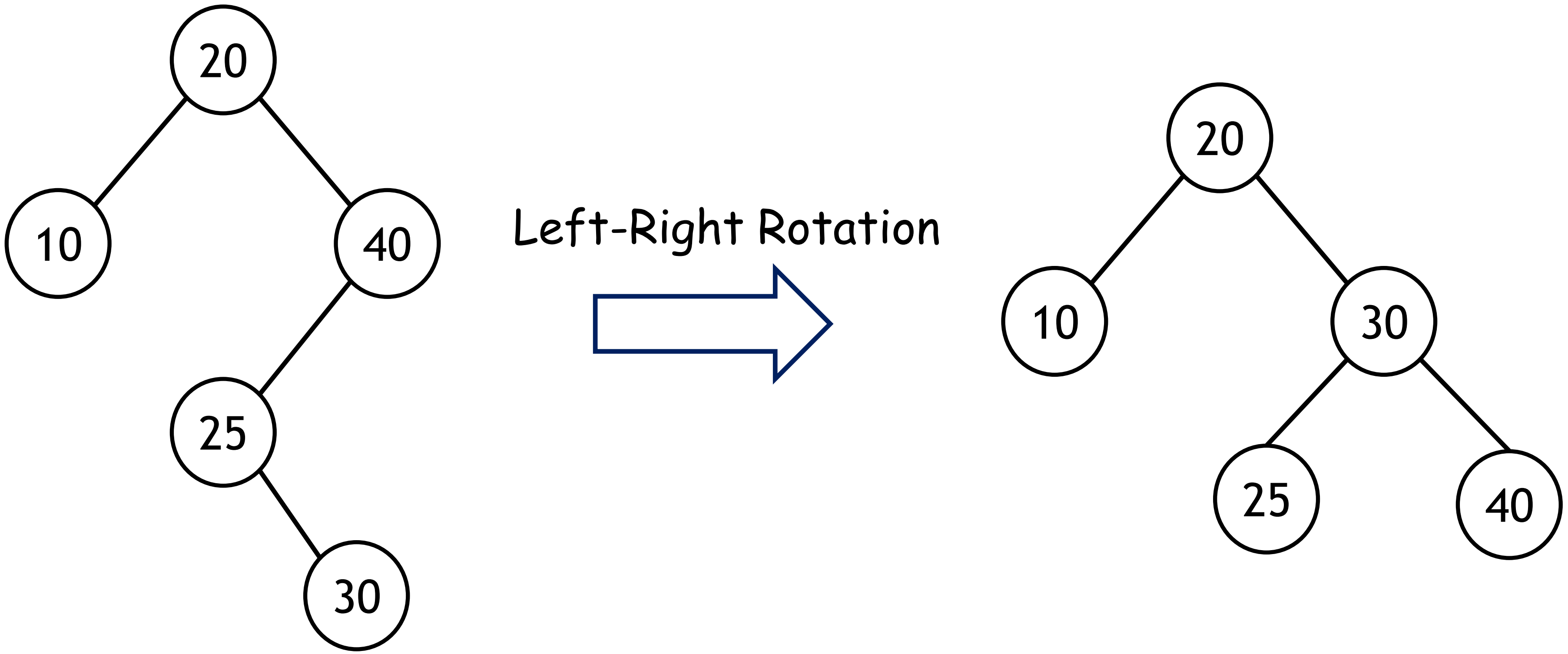
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



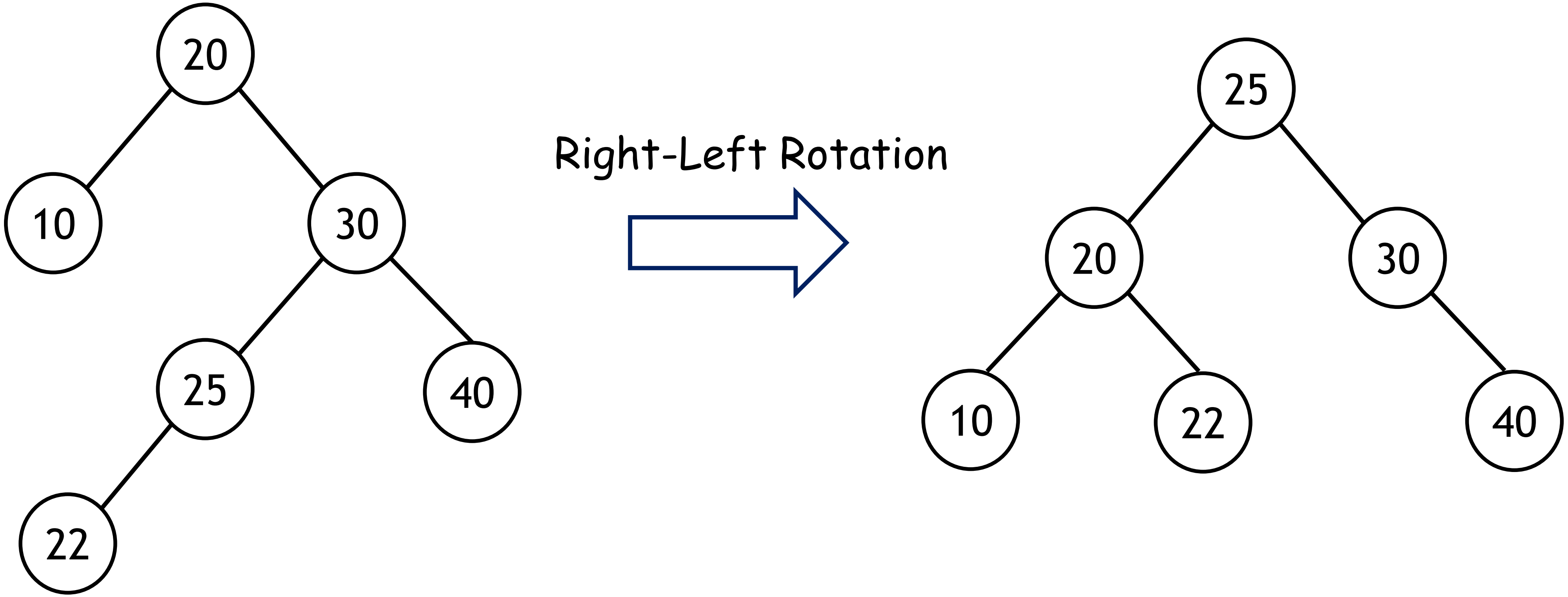
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



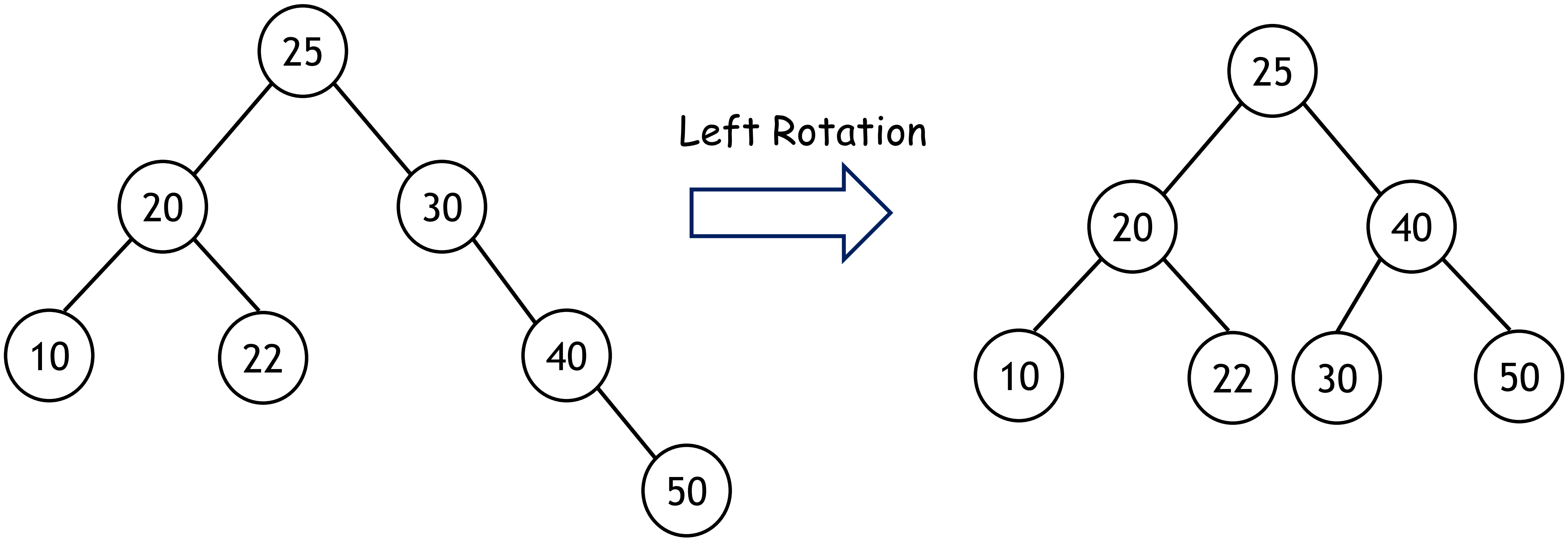
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



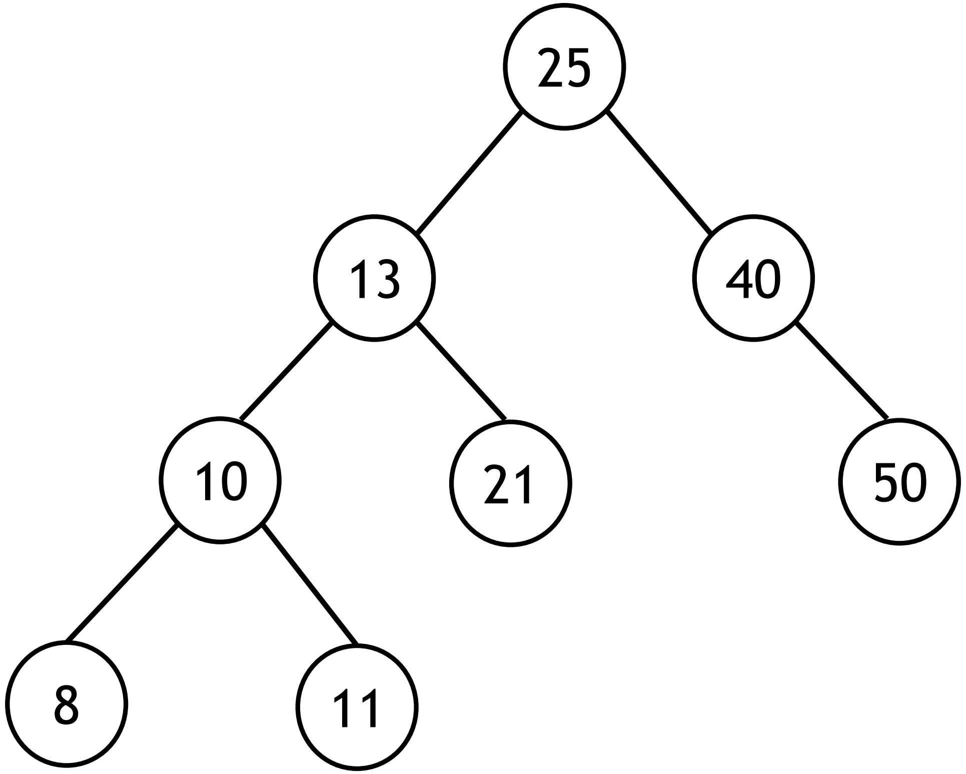
AVL Tree Rotation Example

- Insert: 40, 20, 10, 25, 30, 22, 50



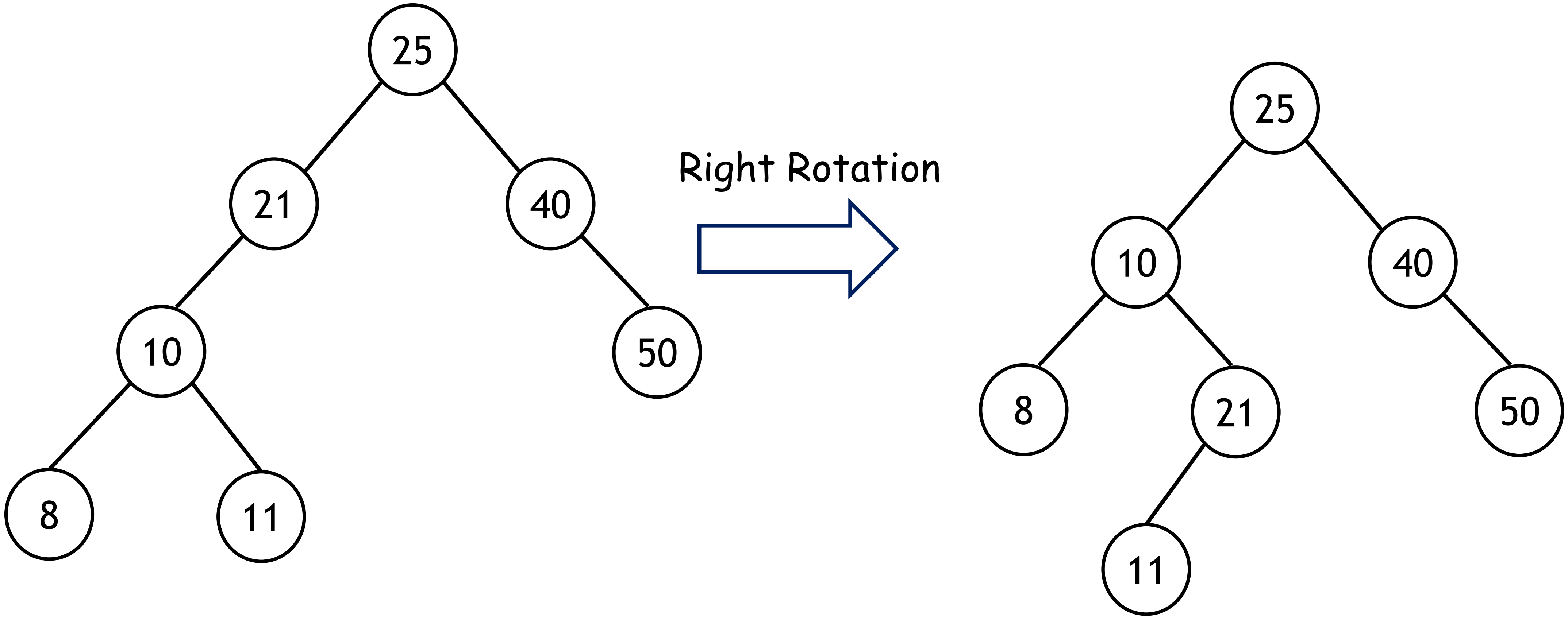
AVL Tree Rotation Example

- Delete: 13



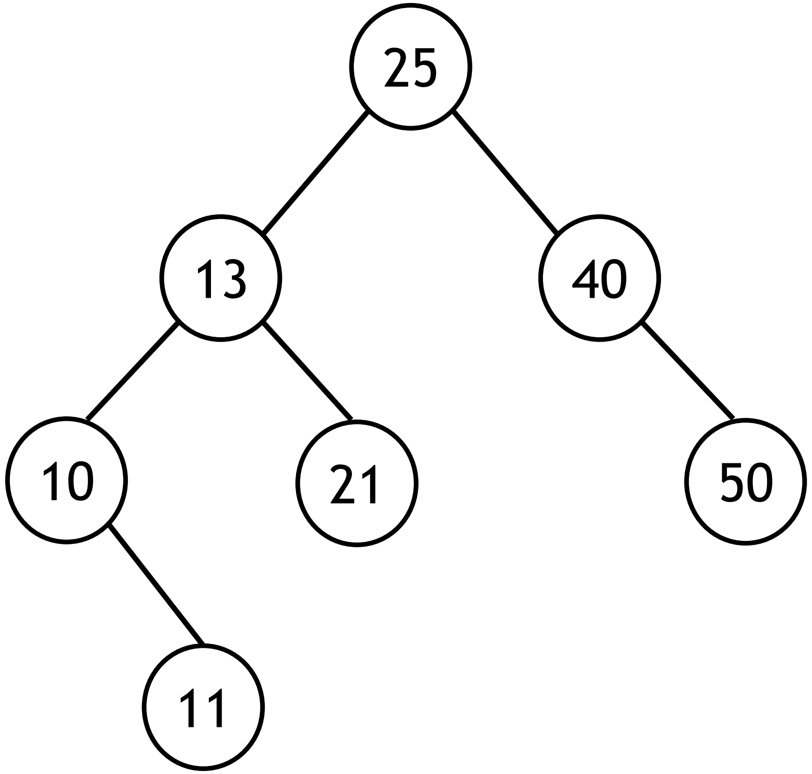
AVL Tree Rotation Example

- Delete: 13



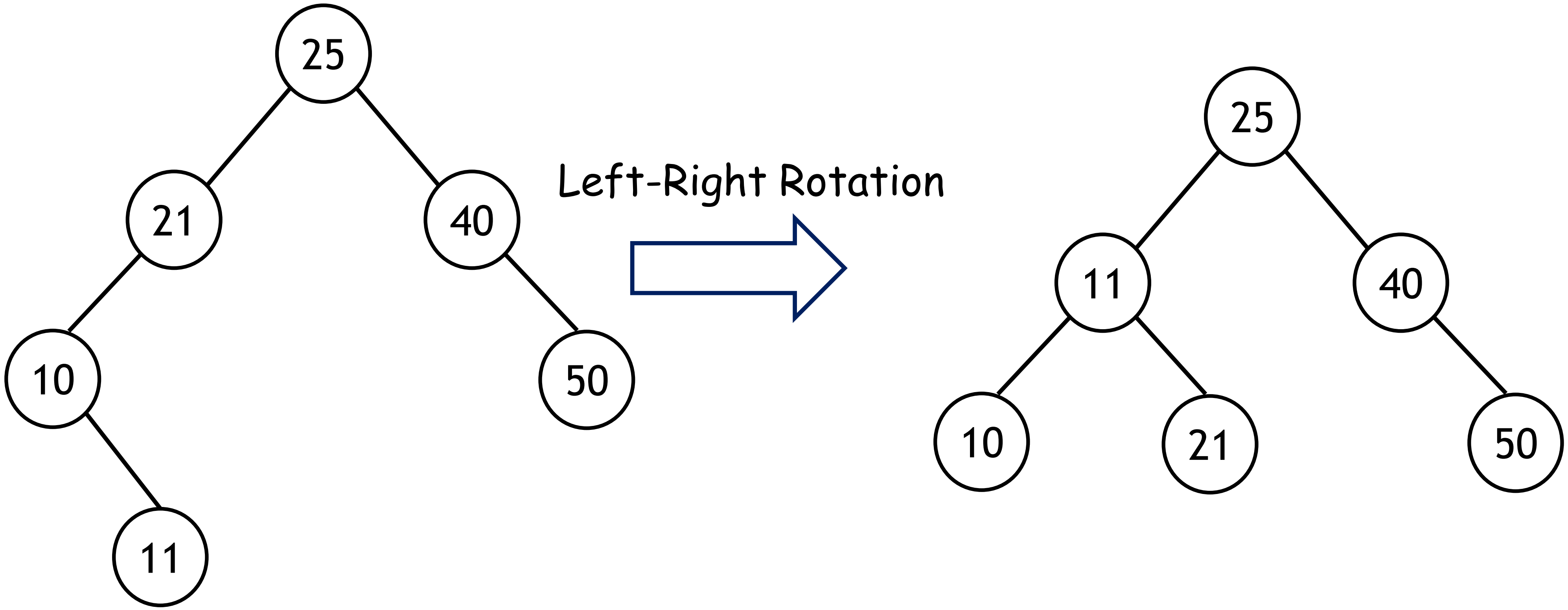
AVL Tree Rotation Example

- Delete: 13



AVL Tree Rotation Example

- Delete: 13



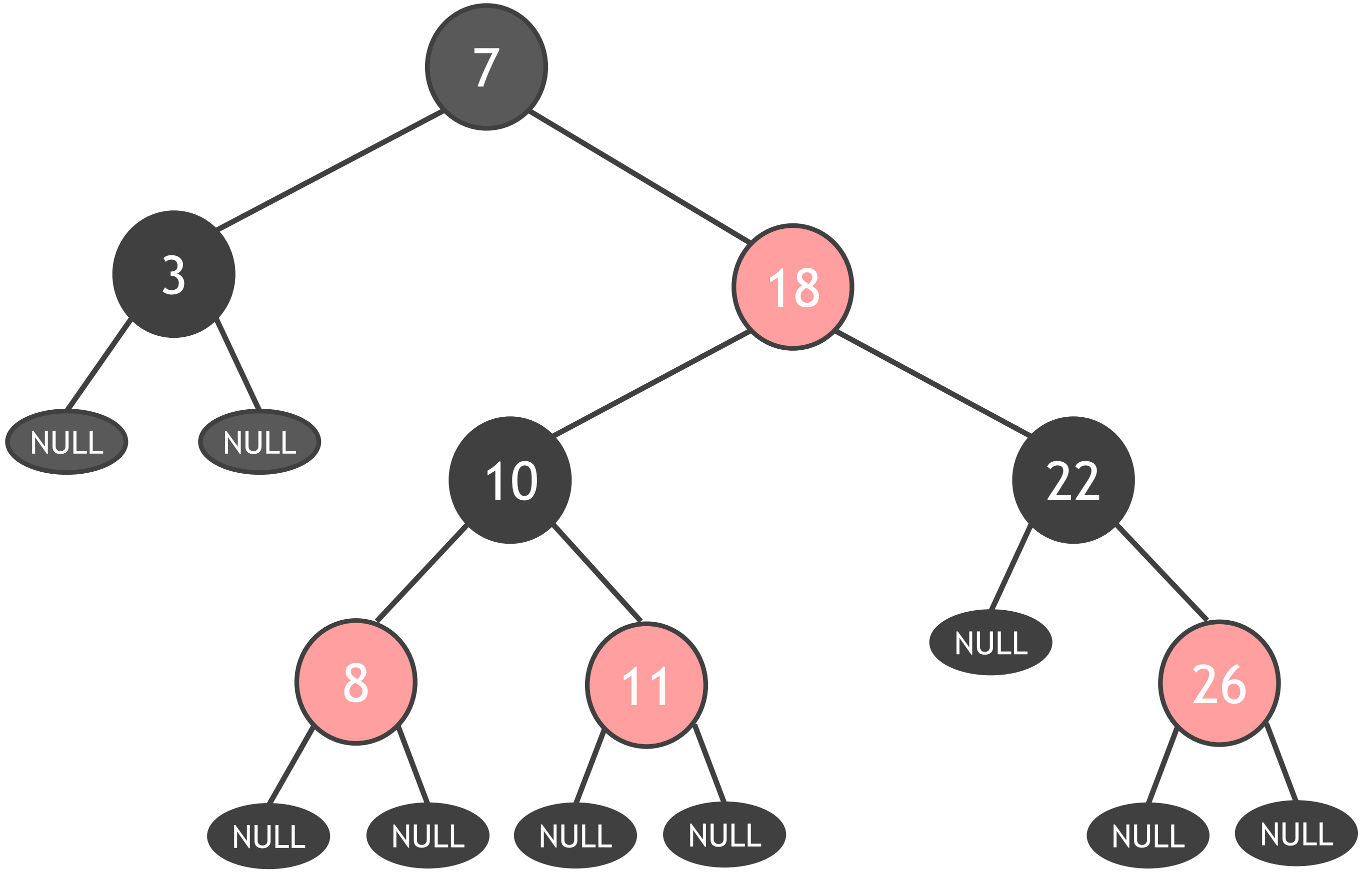
Red-Black Tree

- This data structure requires an extra one-bit color field in each node.

- Red-Black properties:
  - 1. Every node is either red or black.
  - 2. The root is black.
  - 3. The leaves (NULL's) are black.
  - 4. If a node is red, then both its children are black.
  - 5. All simple paths from any node x, excluding x, to a descendant leaf have the same number of black nodes.

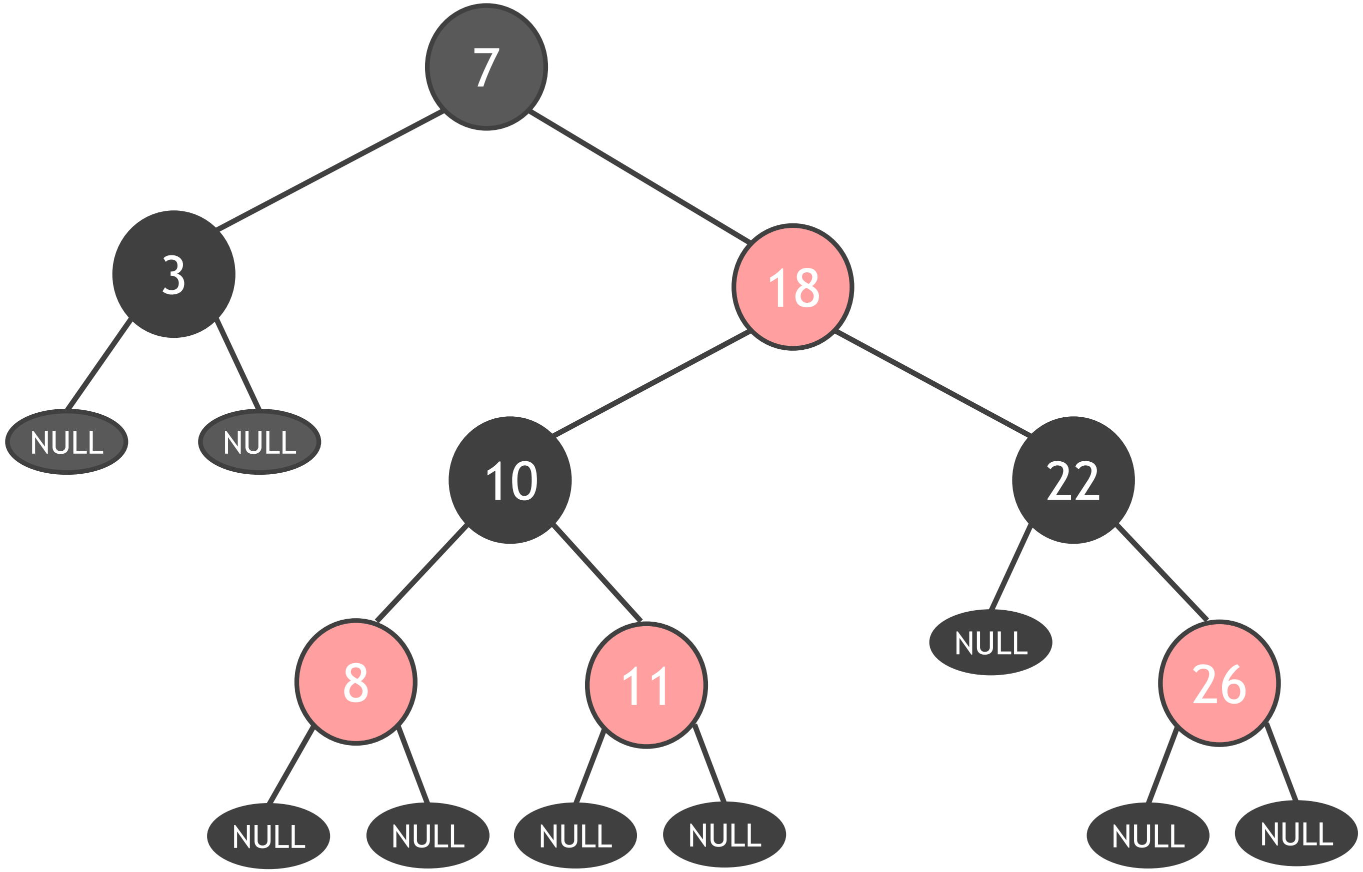
Example of a red-black tree

1. Every node is either red or black.



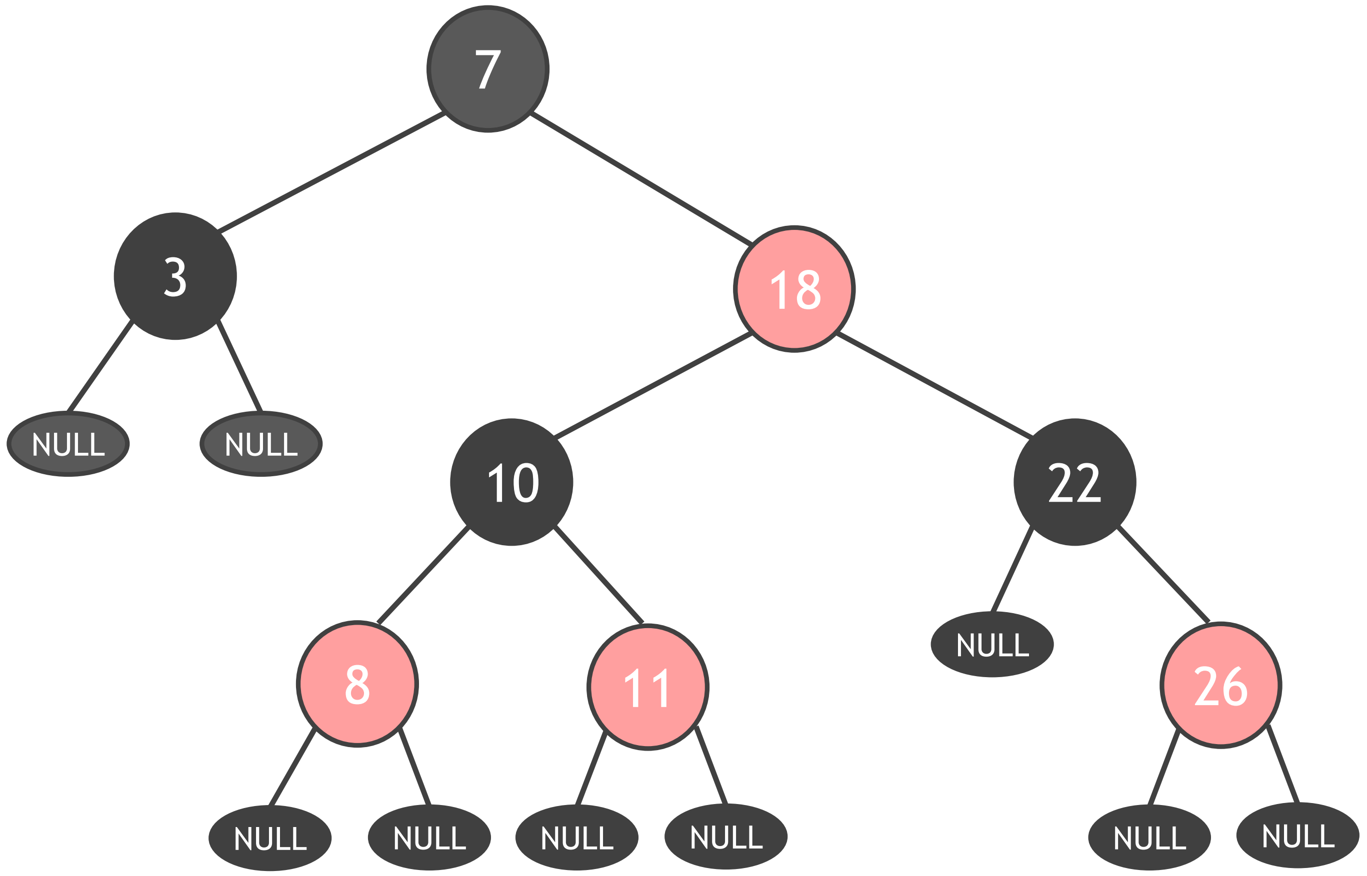
Example of a red-black tree

2. The root is black.



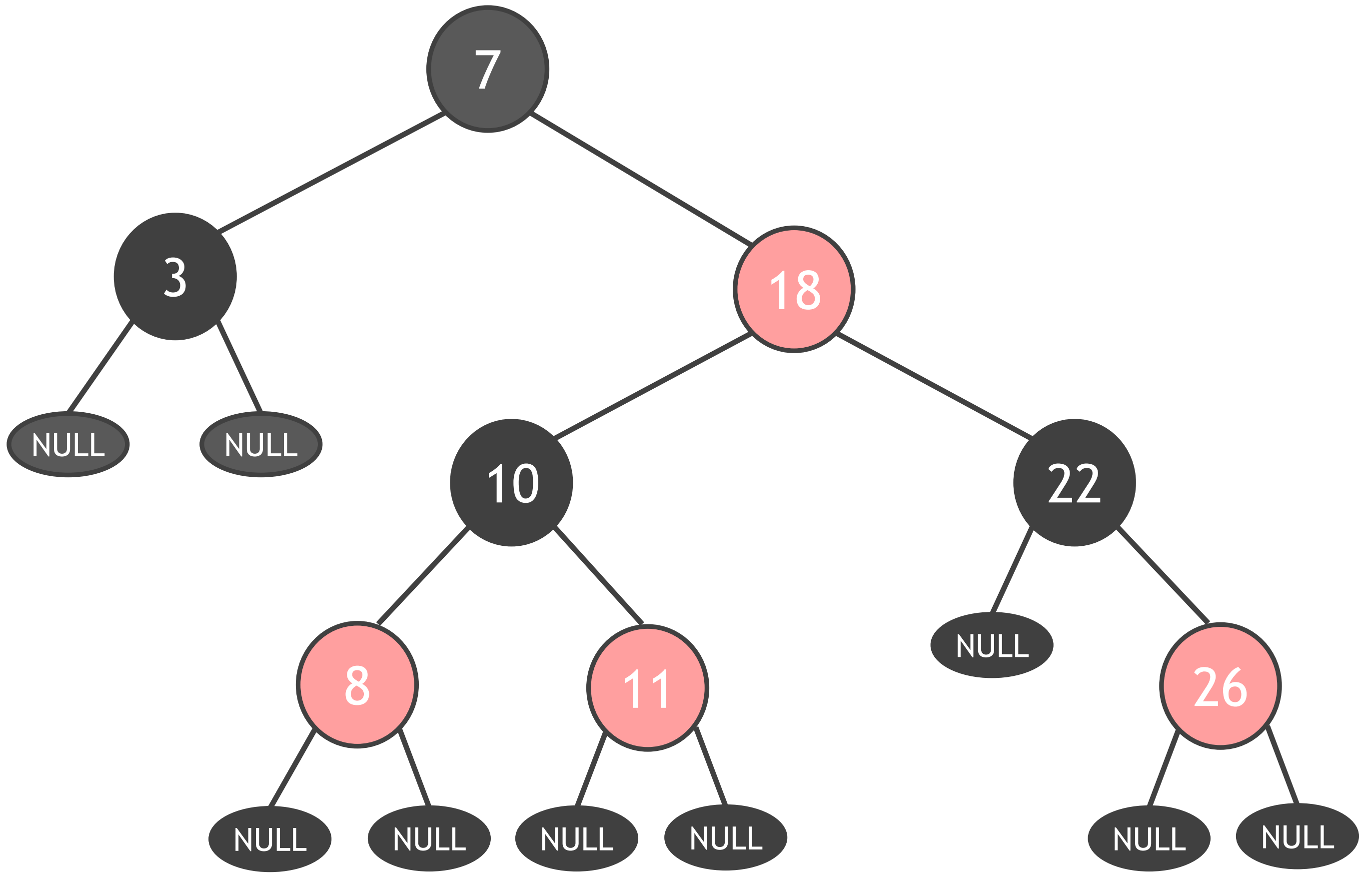
Example of a red-black tree

3. The leaves (NULL's) are black.



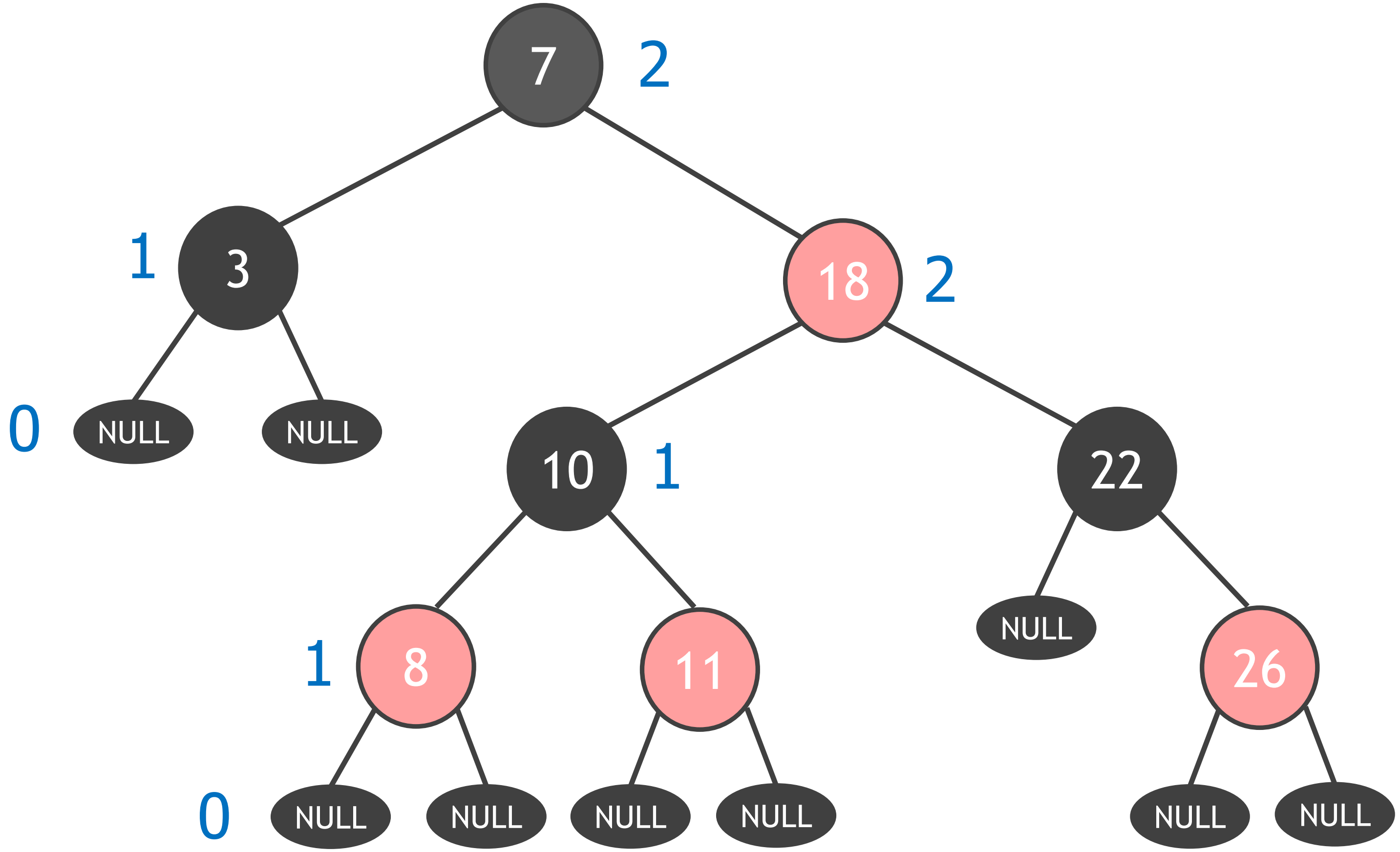
Example of a red-black tree

4. If a node is red, then both its children are black.



Example of a red-black tree

5. All simple paths from any node x, excluding x, to a descendant leaf have the same number of black nodes.



Height of a red-black tree

- A red-black tree with n keys has height h:

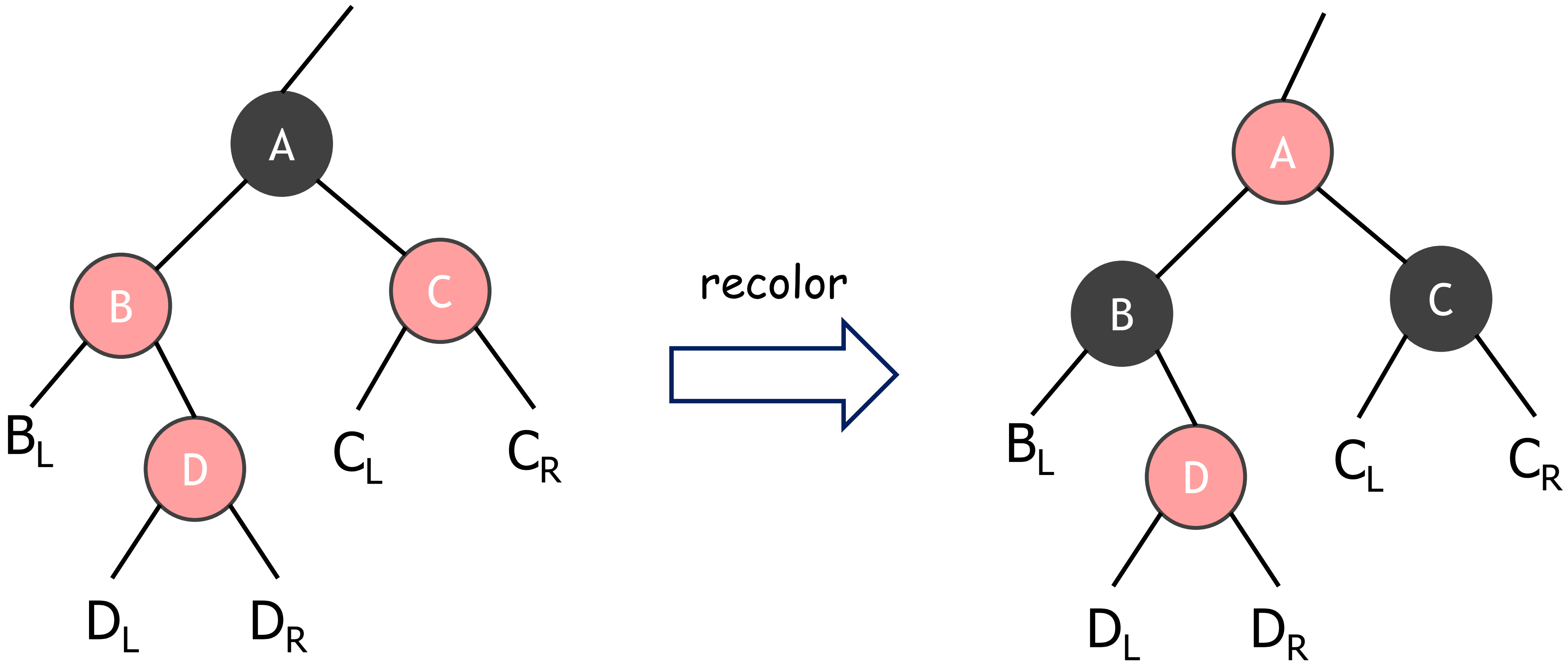
$$h \le 2\log(n+1)$$

Proof?

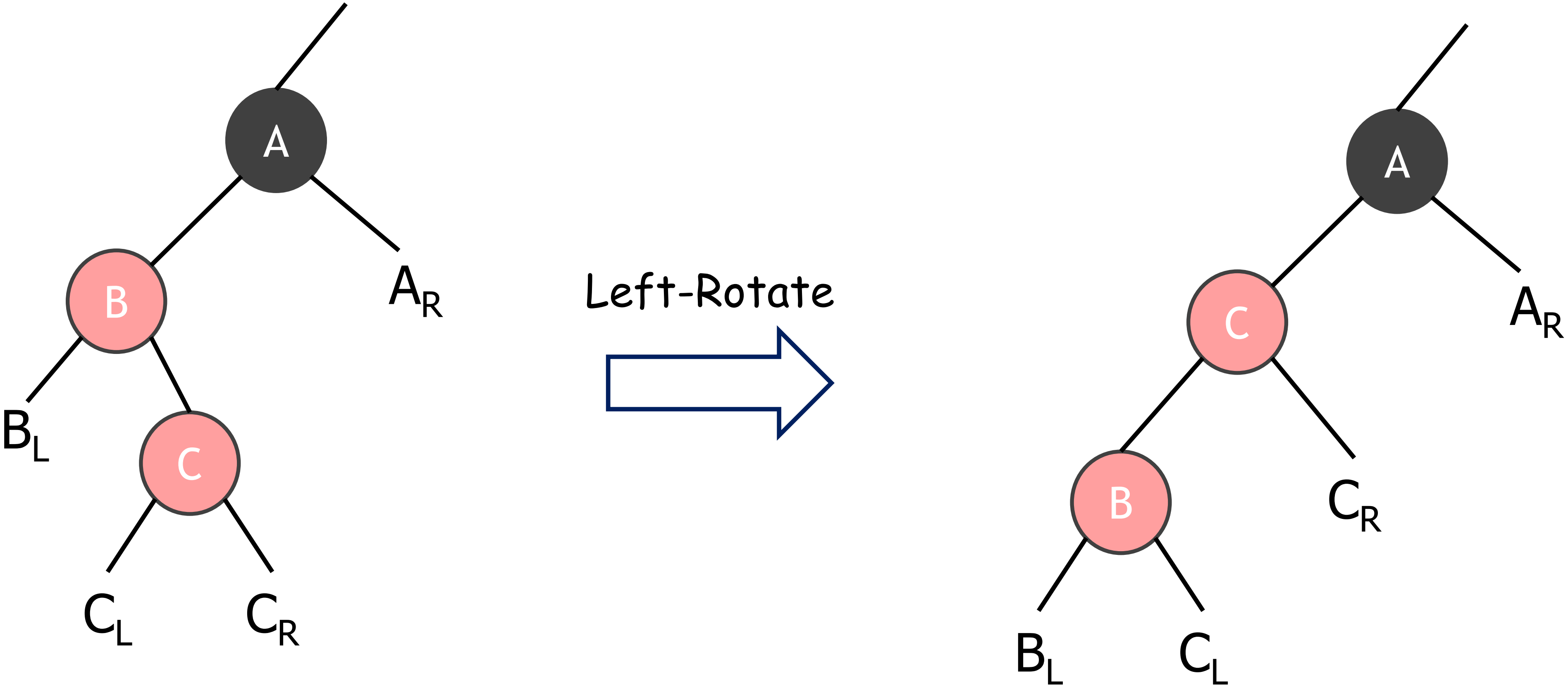
Red-Black Tree operations

- The Insert and Delete operations cause modifications to the red-black tree:
  - The operation itself,
  - Color change
  - Rotations

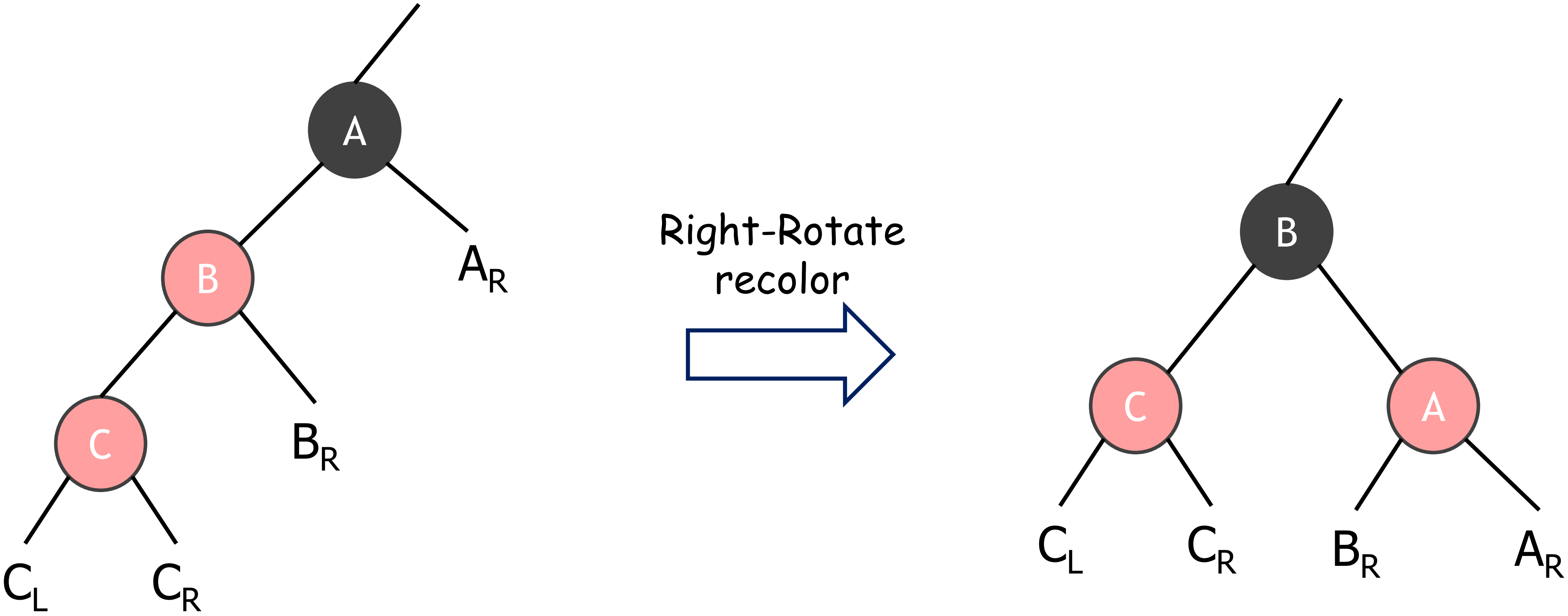
Case 1



Case 2

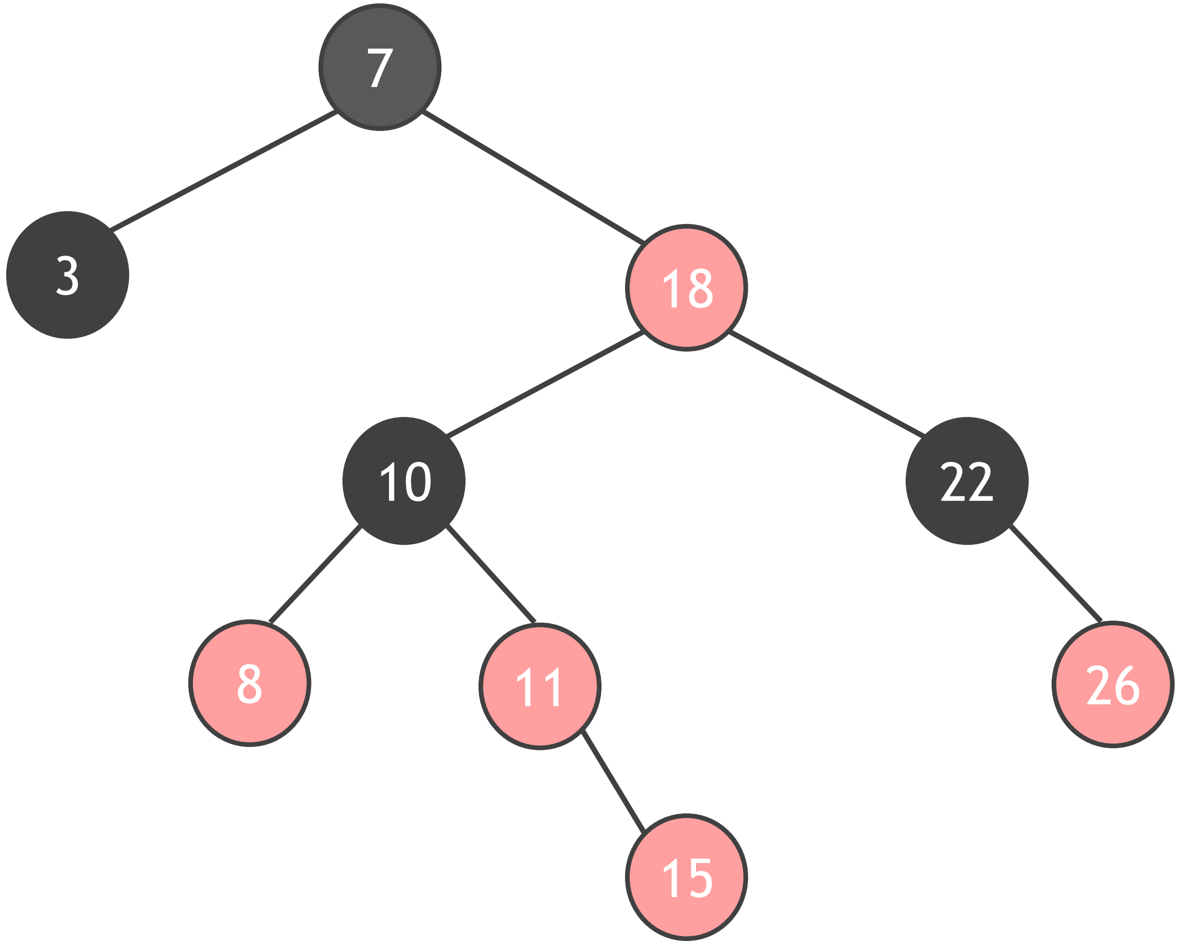


Case 3



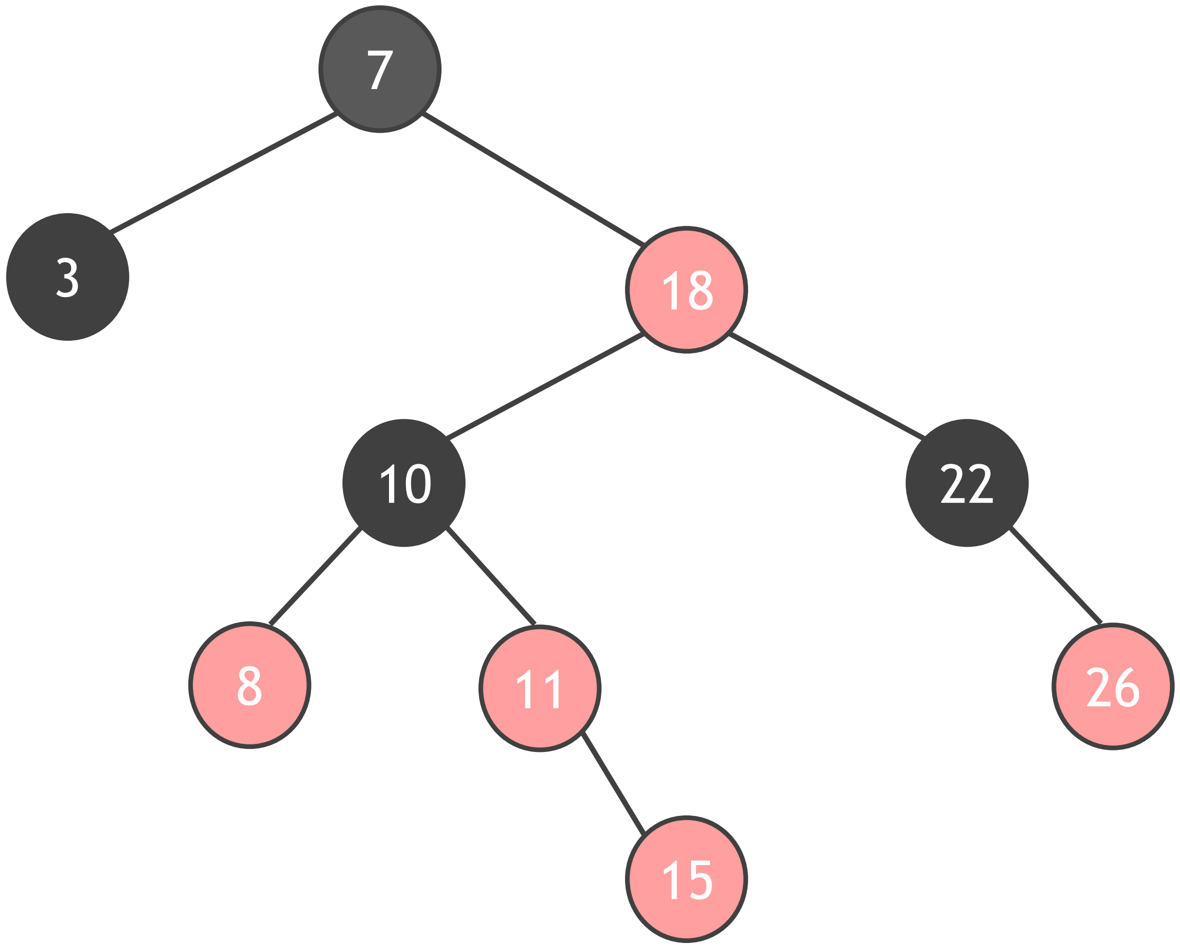
Example: Insert into a red-black tree

- Insert $x = 15$



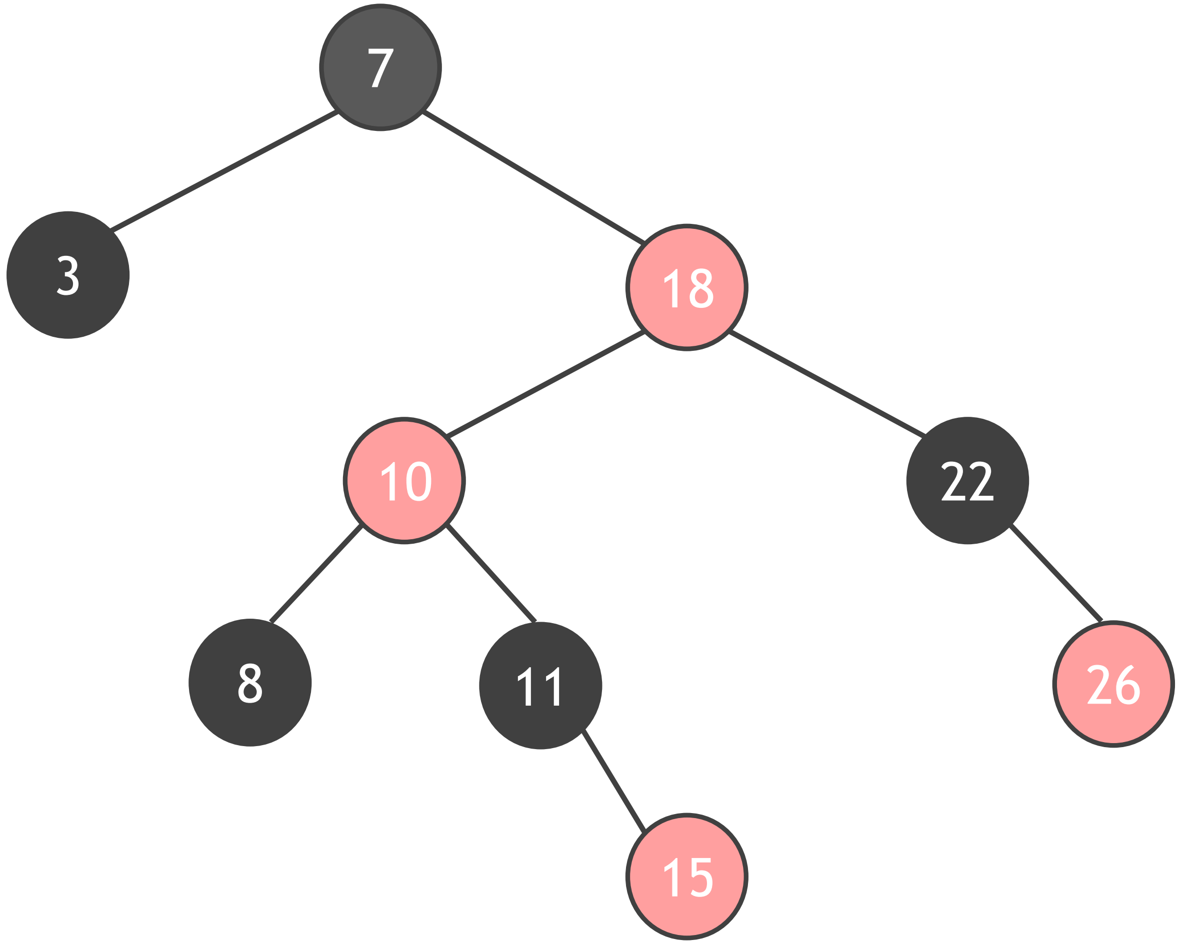
Example: Insert into a red-black tree

- Insert x = 15
- Case 1': recolor



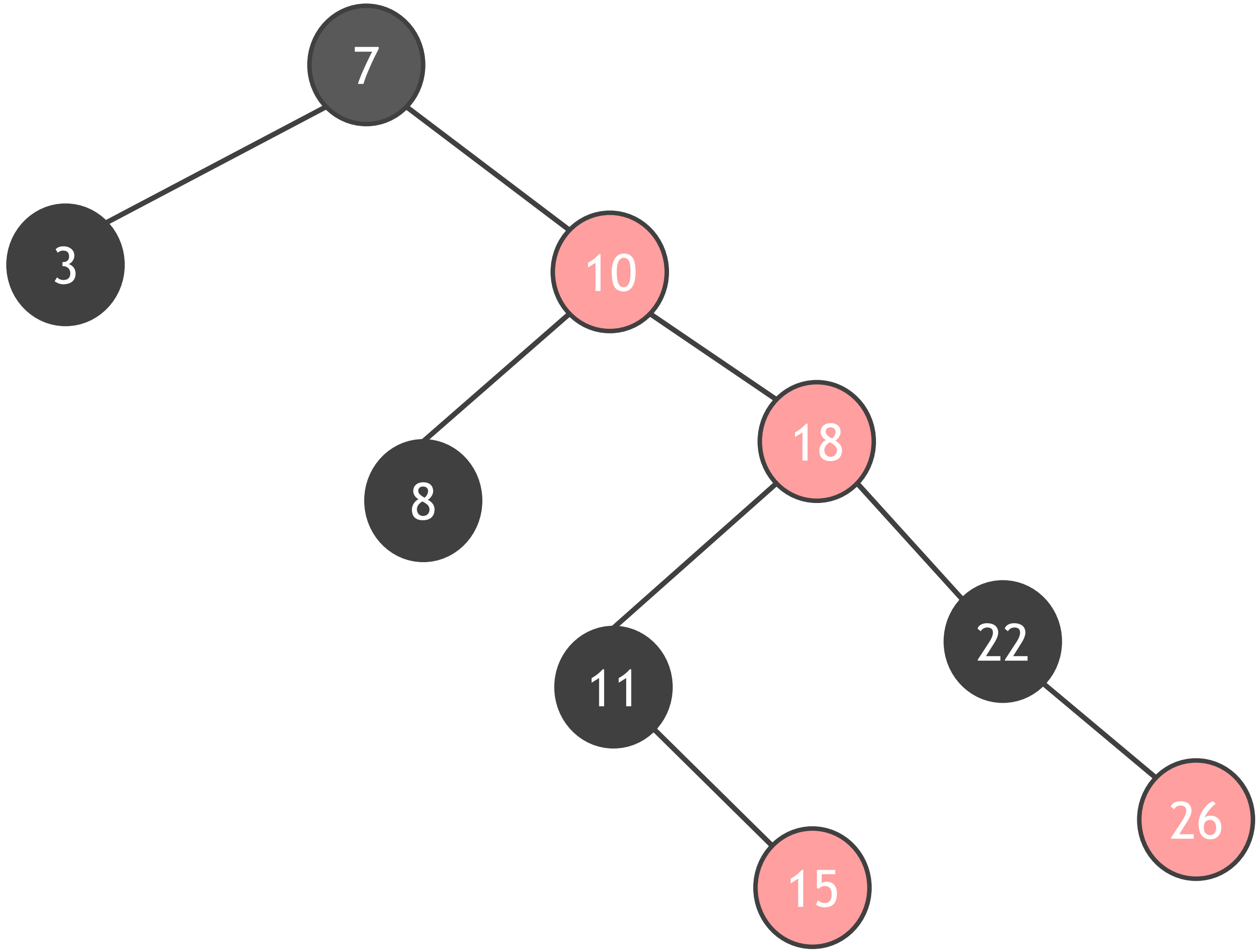
Example: Insert into a red-black tree

- Insert x = 15
- Case 1′: recolor
- Case 2′: Right-rotation



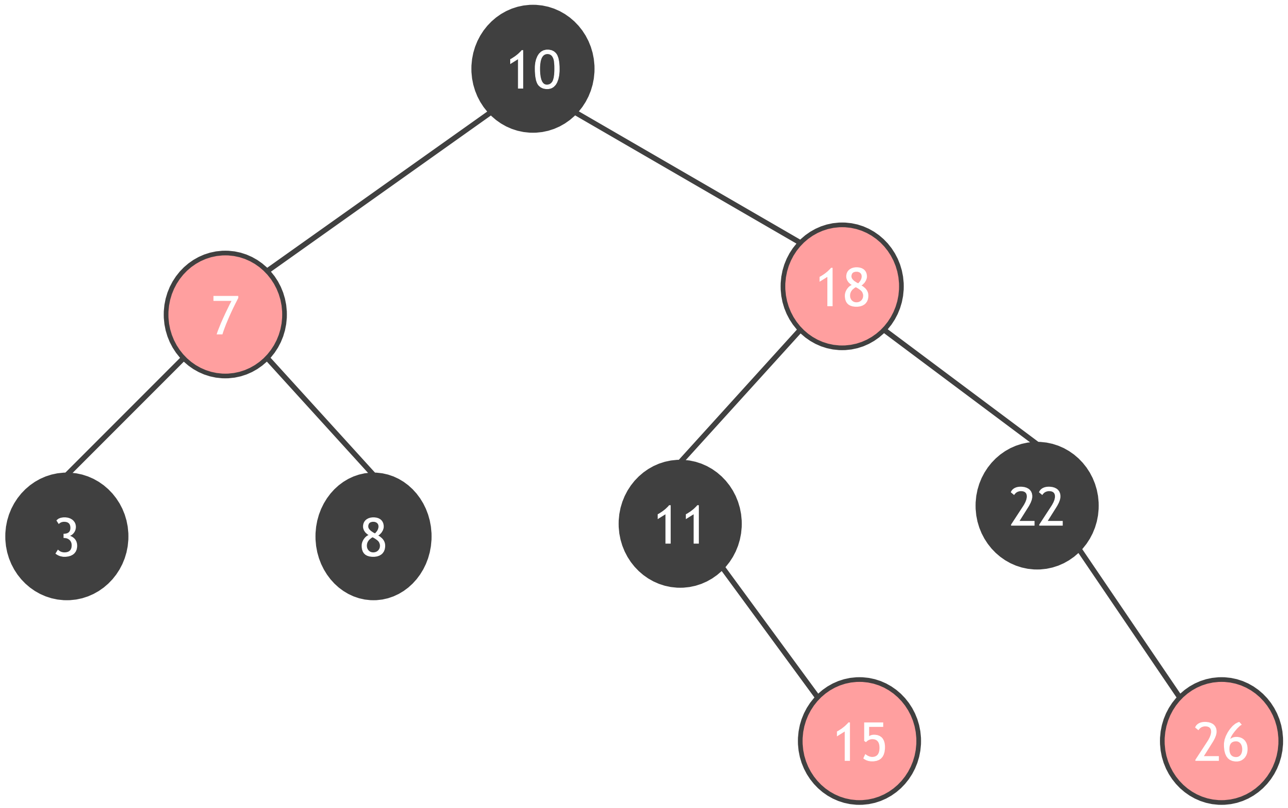
Example: Insert into a red-black tree

- Insert x = 15
- Case 1′: recolor
- Case 2′: Right-rotation
- Case 3′: Left-rotation and recolor



Example: Insert into a red-black tree

- Insert x = 15
- Case 1′: recolor
- Case 2′: Right-rotation
- Case 3′: Left-rotation and recolor
- Done!



Learning outcome

- Binary Search Tree
  - Tree Traversal
  - Insert, Search, Delete

- Balanced Trees
  - AVL Tree
  - Red-Black Tree