Advanced BST

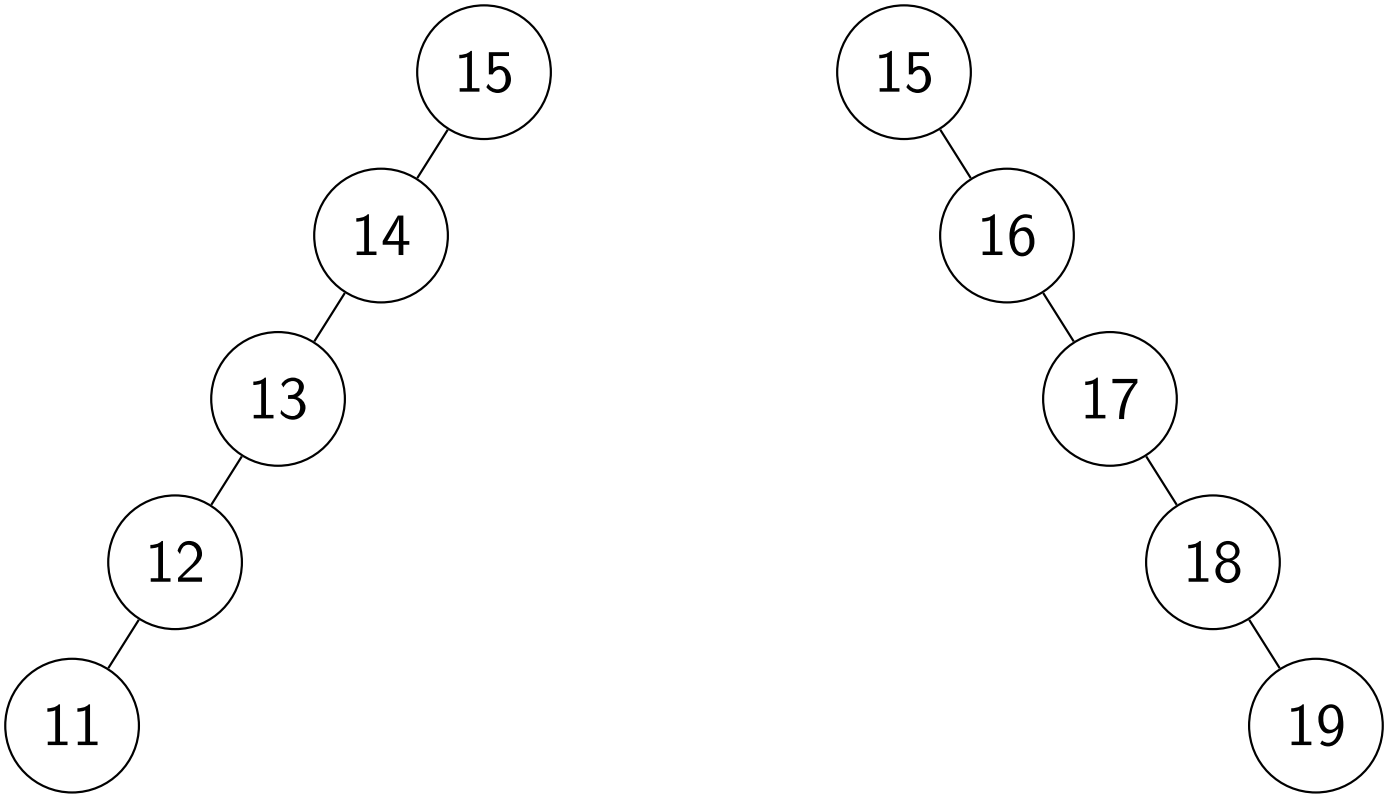
Data Structures and Algorithms

MEng. Duy, Tran Ngoc Bao
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

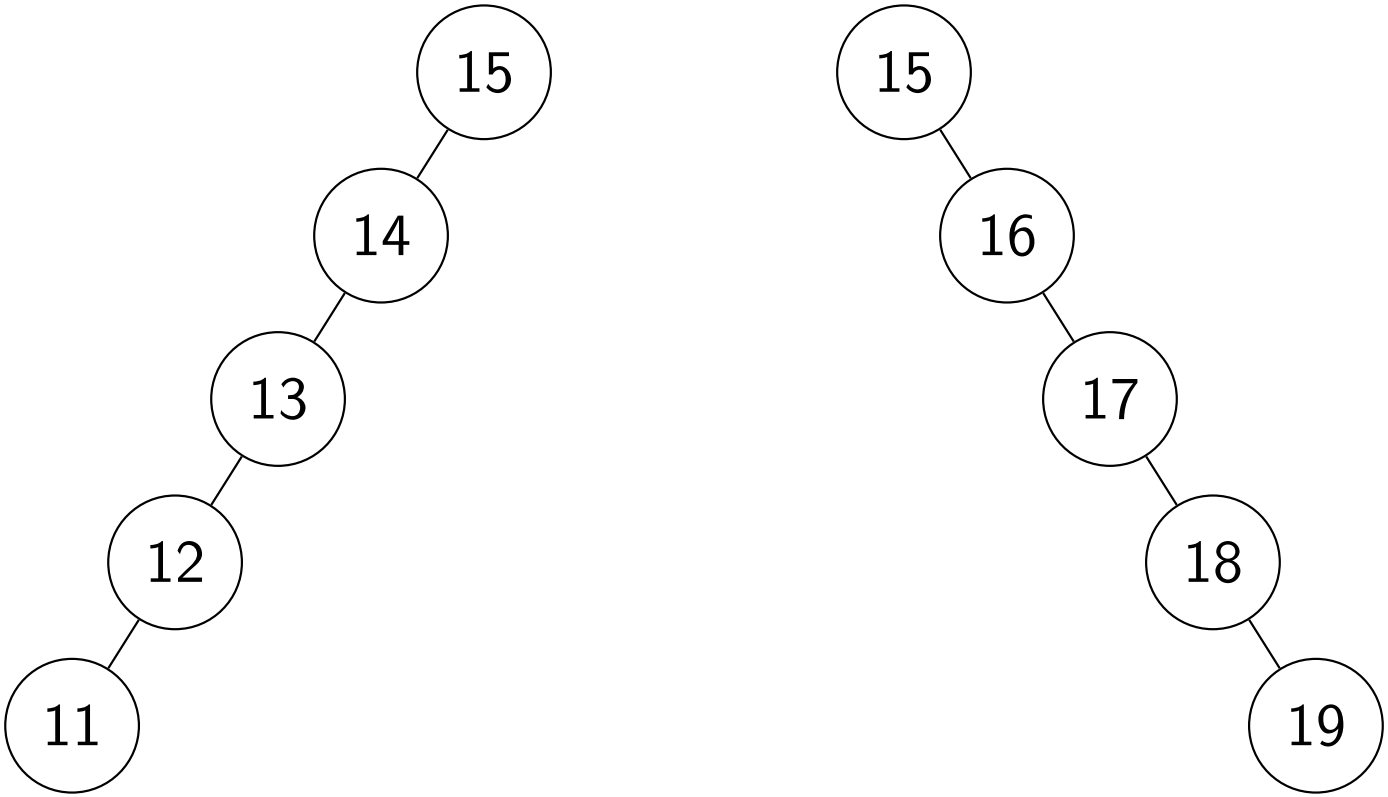
Problem with BST

With ordered input sequences, the BST becomes a singly linked list.



Problem with BST

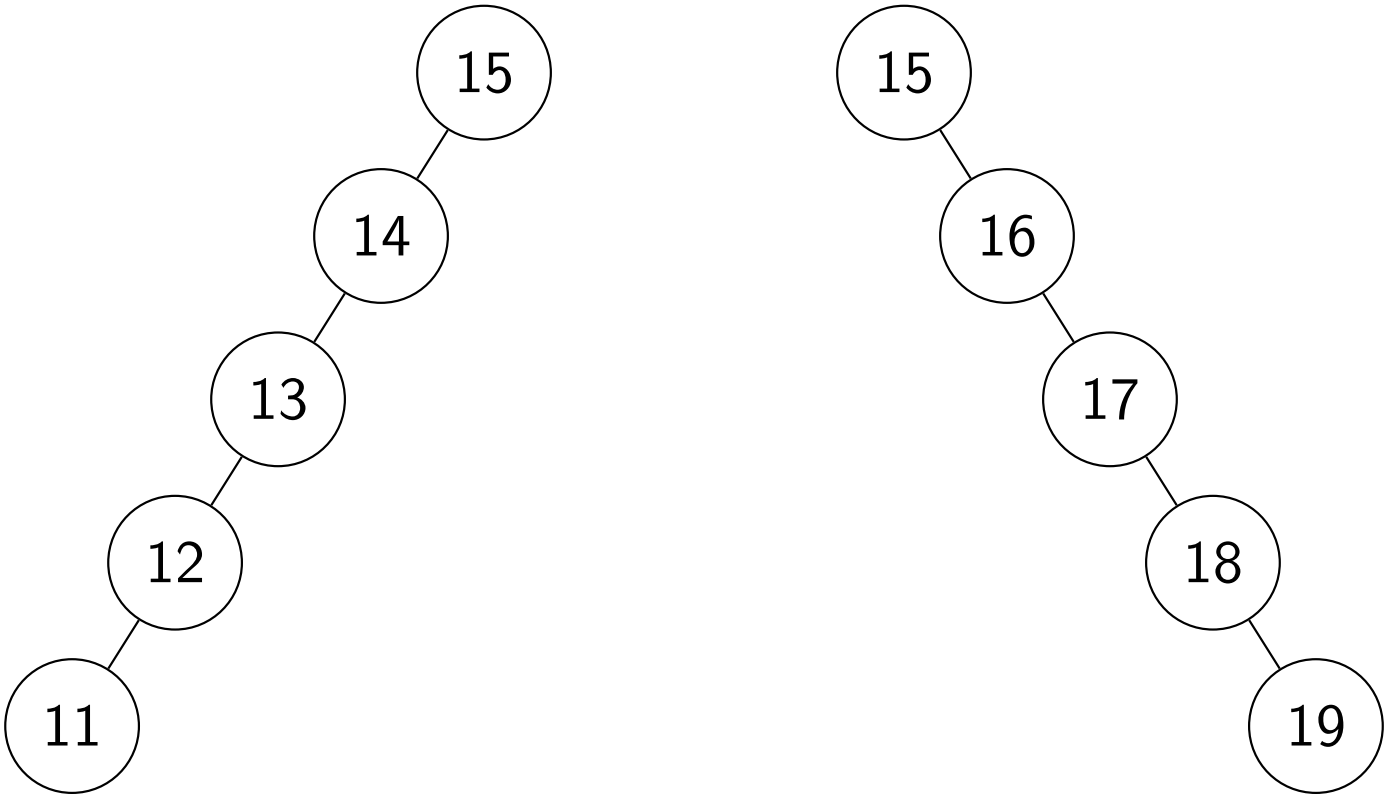
With ordered input sequences, the BST becomes a singly linked list.



All operations: $O(n)$.

Problem with BST

With ordered input sequences, the BST becomes a singly linked list.



Requires another search trees

AVL Tree Concepts

AVL Tree

Definition

AVL Tree is:

- A Binary Search Tree,
- in which the heights of the left and right subtrees of the root differ by at most 1, and
- the left and right subtrees are again AVL trees.

Discovered by G.M.Adel'son-Vel'skii and E.M.Landis in 1962.

AVL Tree

Definition

AVL Tree is:

- A Binary Search Tree,
- in which the heights of the left and right subtrees of the root differ by at most 1, and
- the left and right subtrees are again AVL trees.

Discovered by G.M.Adel'son-Vel'skii and E.M.Landis in 1962.

AVL Tree is a Binary Search Tree that is balanced tree.

AVL Tree

A binary tree is an AVL Tree if

- Each node satisfies BST property: key of the node is greater than the key of each node in its left subtree and is smaller than or equals to the key of each node in its right subtree.
- Each node satisfies balanced tree property: the difference between the heights of the left subtree and right subtree of the node does not exceed one.

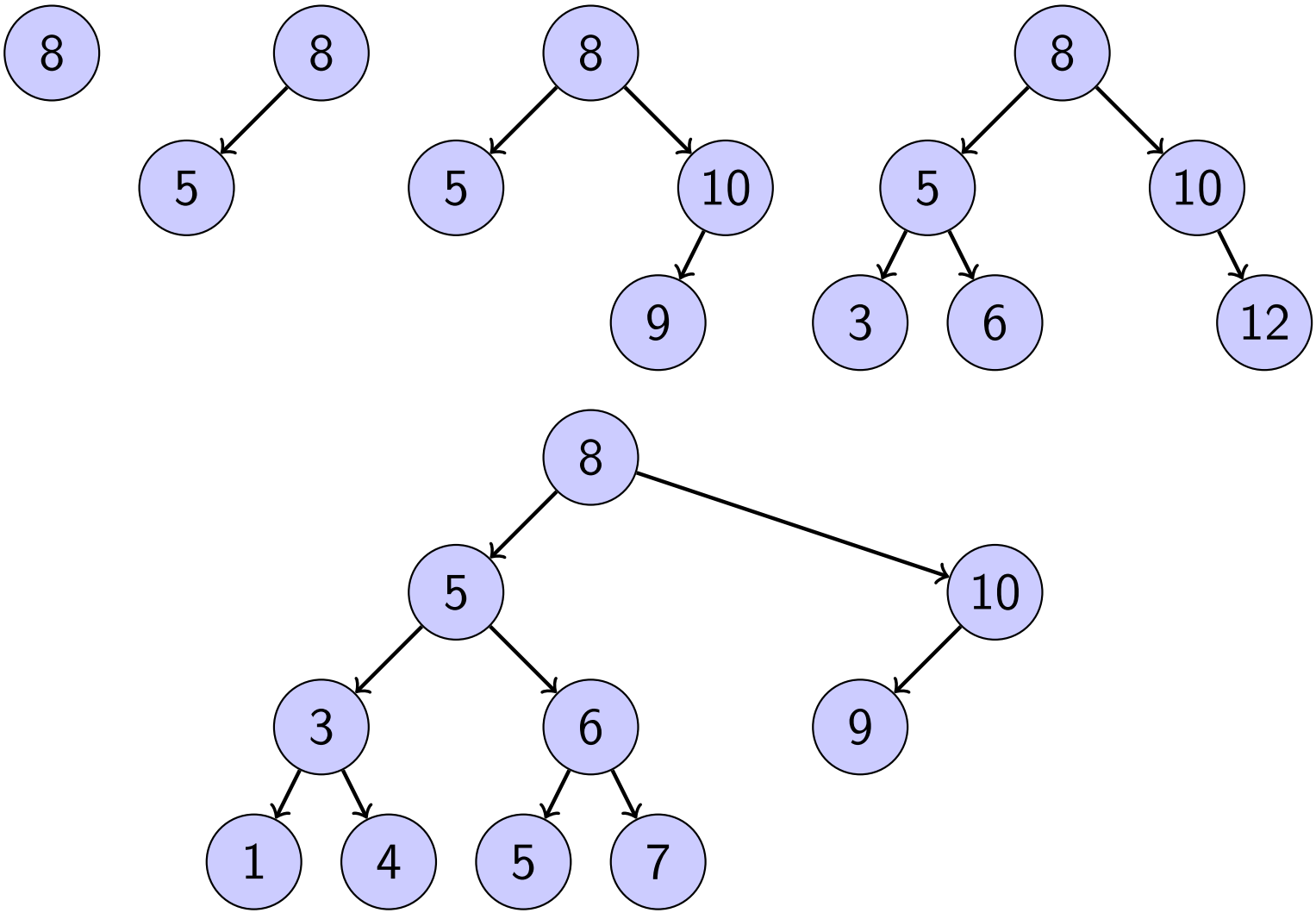
AVL Tree

Balance factor

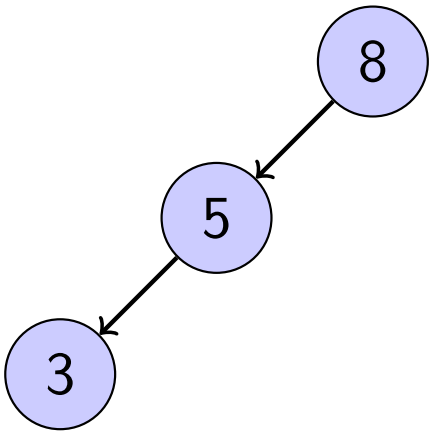
- left_higher (LH): $H_L = H_R + 1$
- equal_height (EH): $H_L = H_R$
- right_higher (RH): $H_R = H_L + 1$

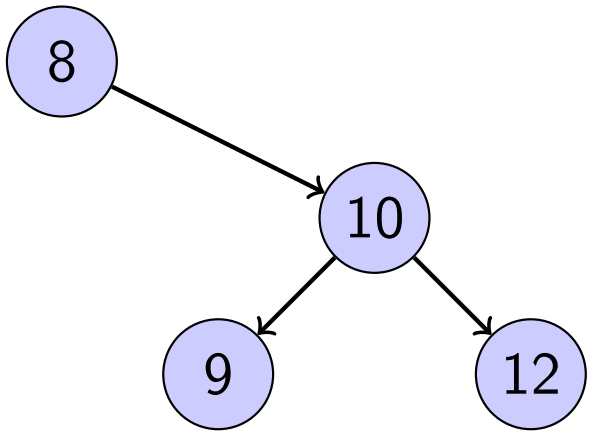
(H_L, H_R: the heights of left and right subtrees)

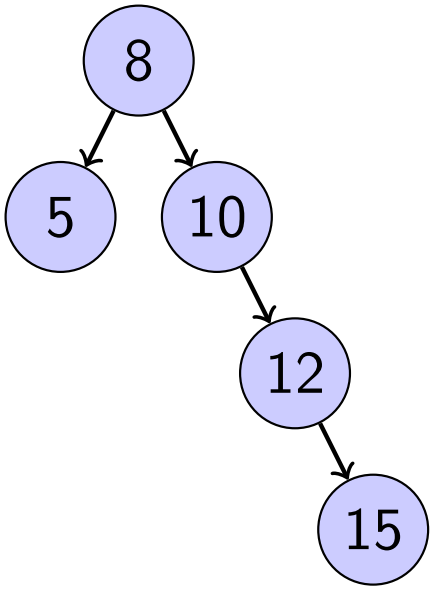
AVL Trees

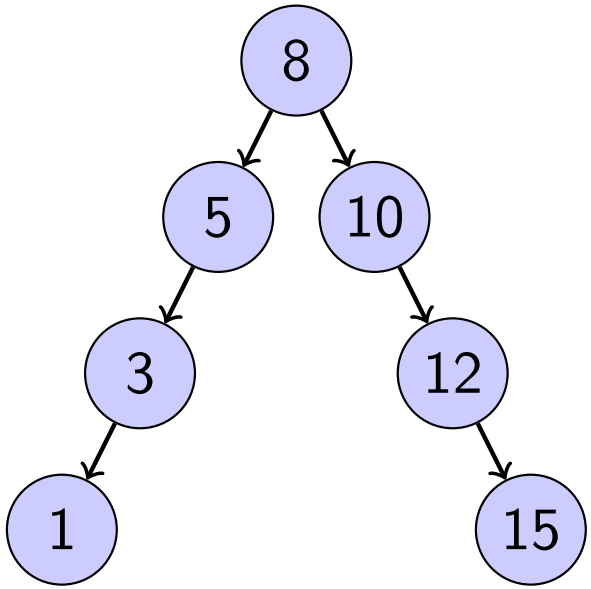


Non-AVL Trees









AVL Balance

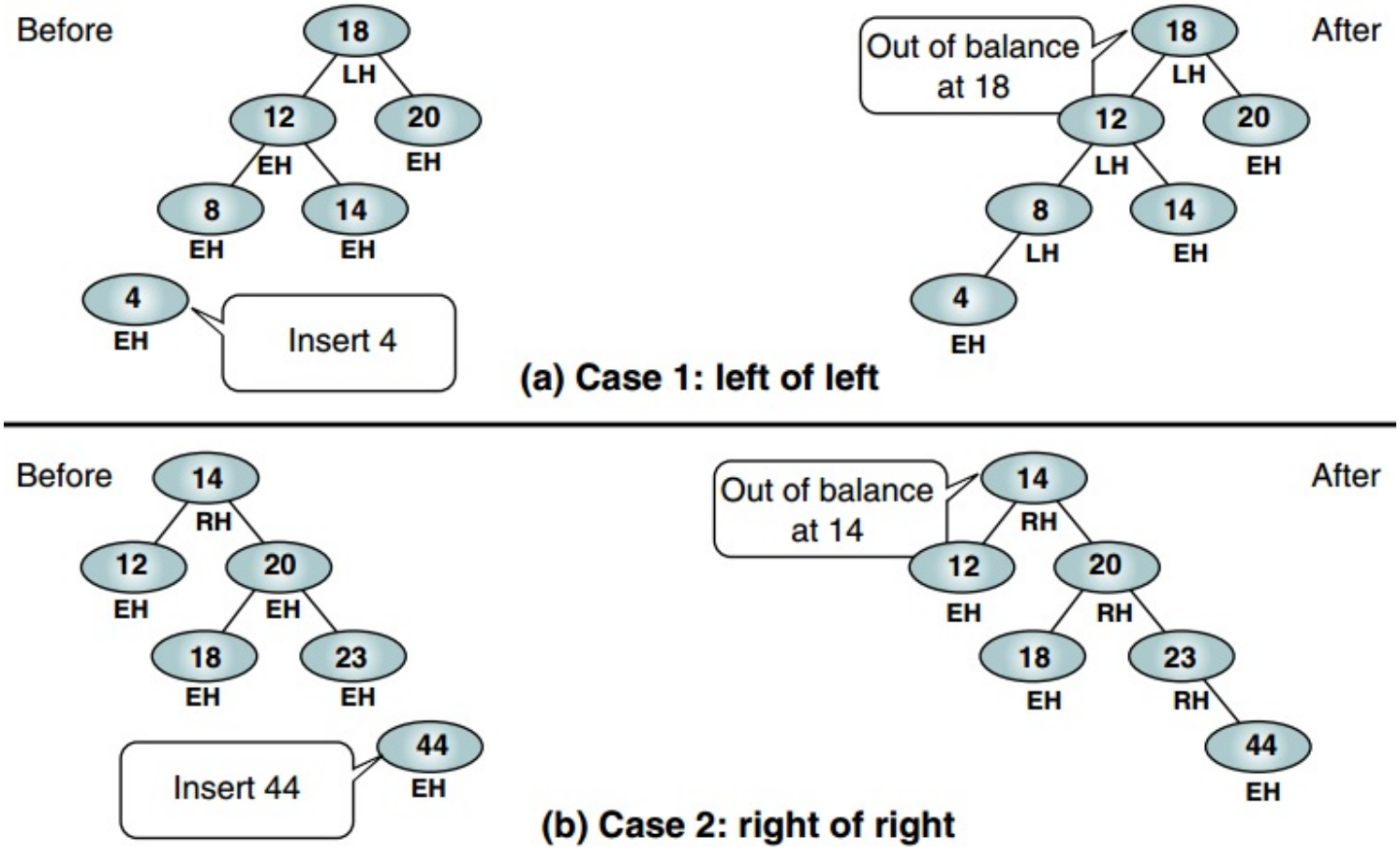
Balancing Trees

- When we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced.
  $\rightarrow$ rebalance the tree.
- Four unbalanced tree cases:
  - left of left: a subtree of a tree that is left high has also become left high;
  - right of right: a subtree of a tree that is right high has also become right high;

Balancing Trees

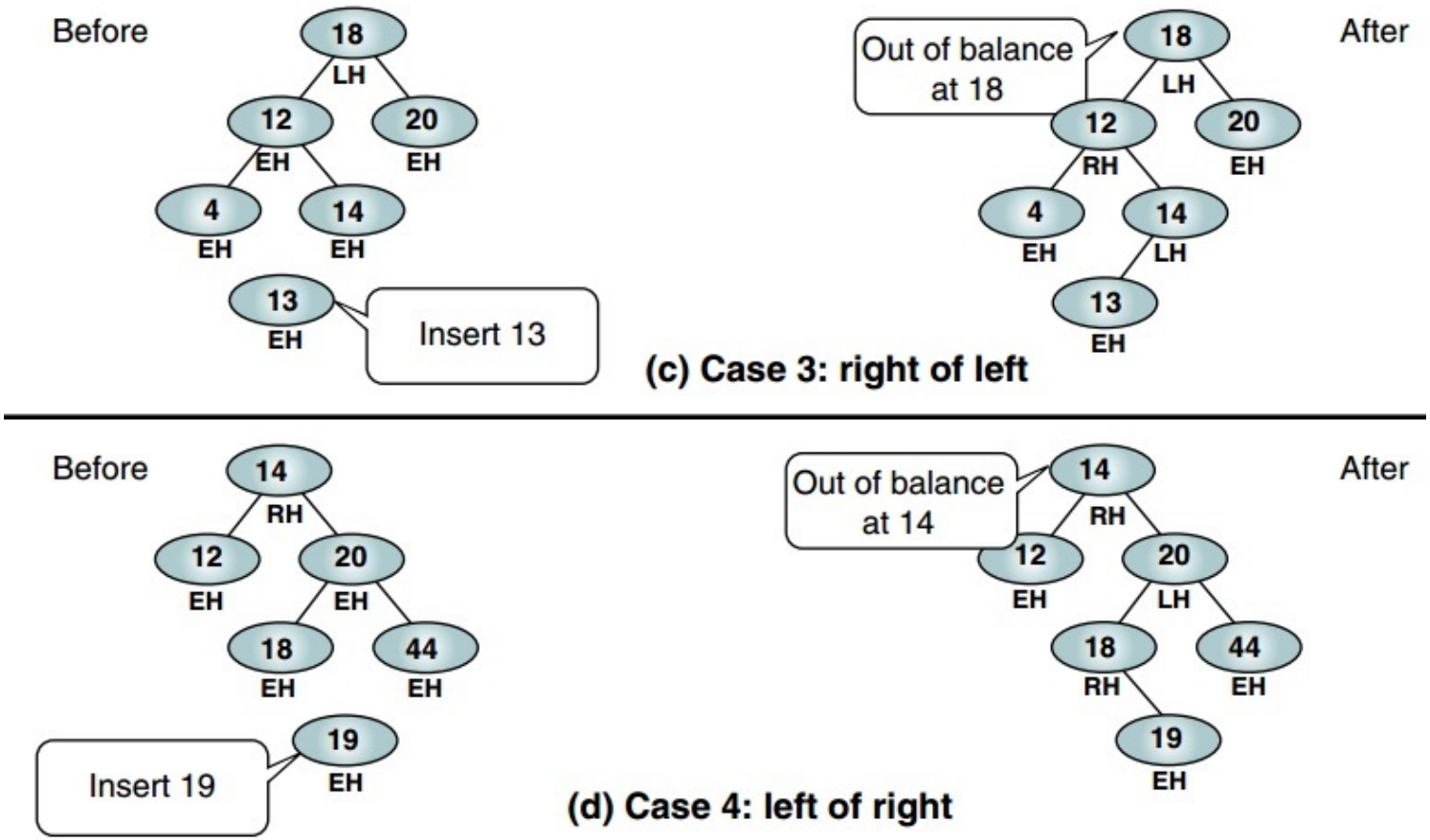
- When we insert a node into a tree or delete a node from a tree, the resulting tree may be unbalanced.
  $\rightarrow$ rebalance the tree.
- Four unbalanced tree cases:
  - right of left: a subtree of a tree that is left high has become right high;
  - left of right: a subtree of a tree that is right high has become left high;

Unbalanced tree cases



(Source: Data Structures - A Pseudocode Approach with C++)

Unbalanced tree cases



(Source: Data Structures - A Pseudocode Approach with C++)

Rotate Right

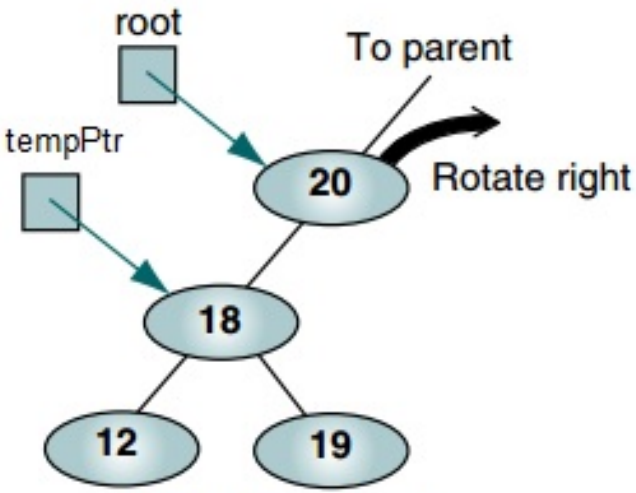
1 **Algorithm** rotateRight(ref root <pointer>)
2 Exchanges pointers to rotate the tree right.

3 **Pre:** `root` is pointer to tree to be rotated
4 **Post:** node rotated and root updated

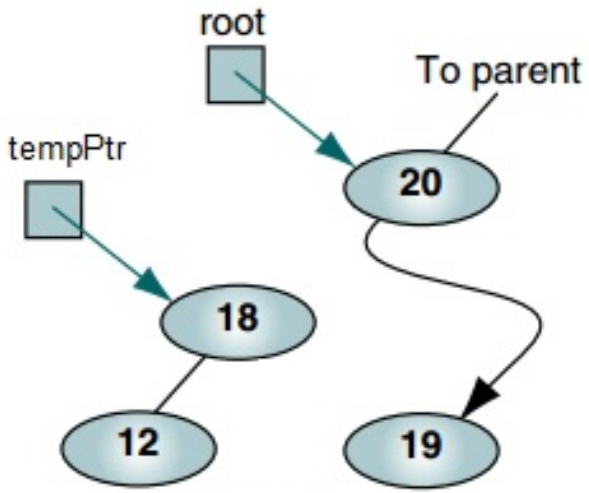
5 tempPtr = root->left
6 root->left = tempPtr->right
7 tempPtr->right = root
8 **Return** tempPtr

9 **End** rotateRight

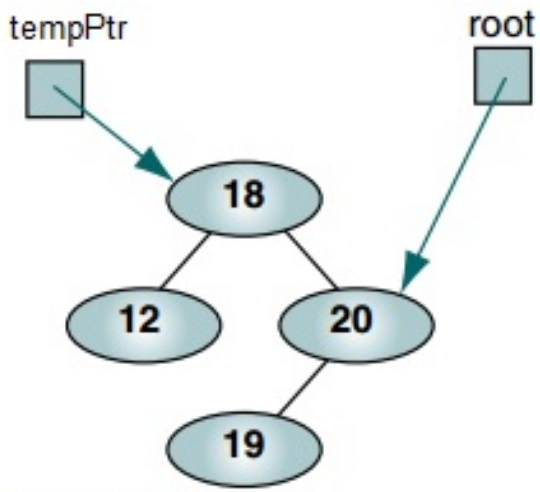
Rotate Right



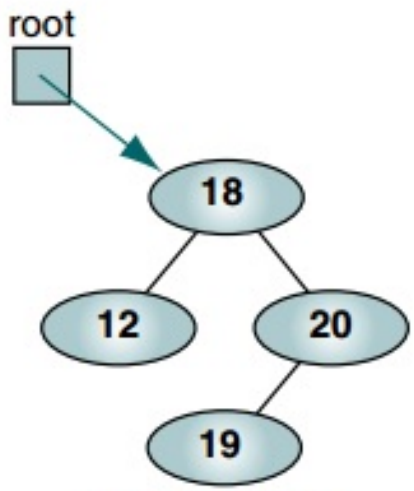
(a) After Step 1



(b) After Step 2



(c) After Step 3



(d) At end

(Source: Data Structures - A Pseudocode Approach with C++)

Rotate Left

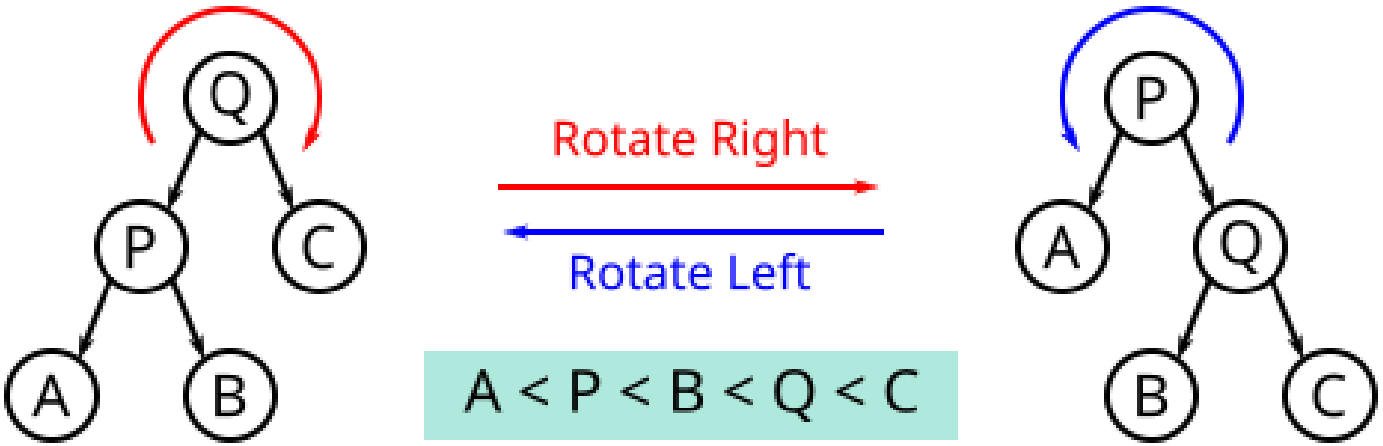
1 **Algorithm** rotateLeft(ref root <pointer>)
2 Exchanges pointers to rotate the tree left.

3 **Pre:** `root` is pointer to tree to be rotated
4 **Post:** node rotated and root updated

5 tempPtr = root->right
6 root->right = tempPtr->left
7 tempPtr->left = root
8 **Return** tempPtr

9 **End** rotateLeft

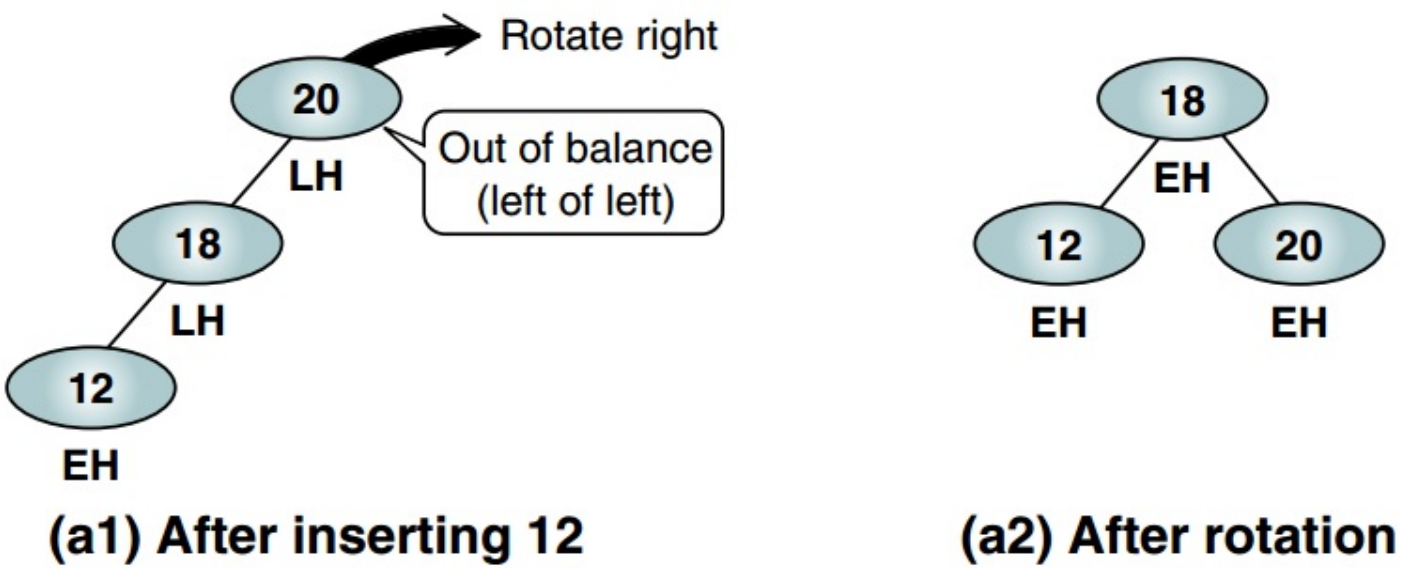
Rotation in tree



Balancing Trees - Case 1: Left of Left

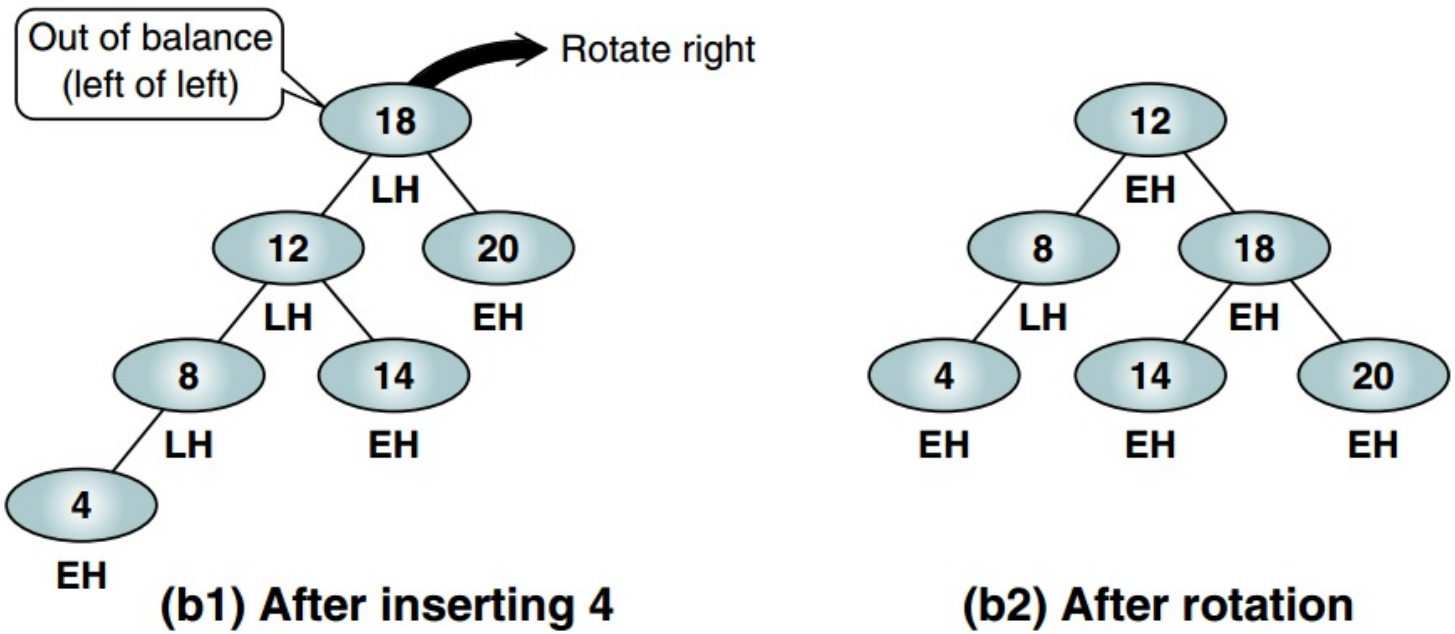
Out of balance condition created by a left high subtree of a left high tree

→ balance the tree by rotating the out of balance node to the right.



(Source: Data Structures - A Pseudocode Approach with C++)

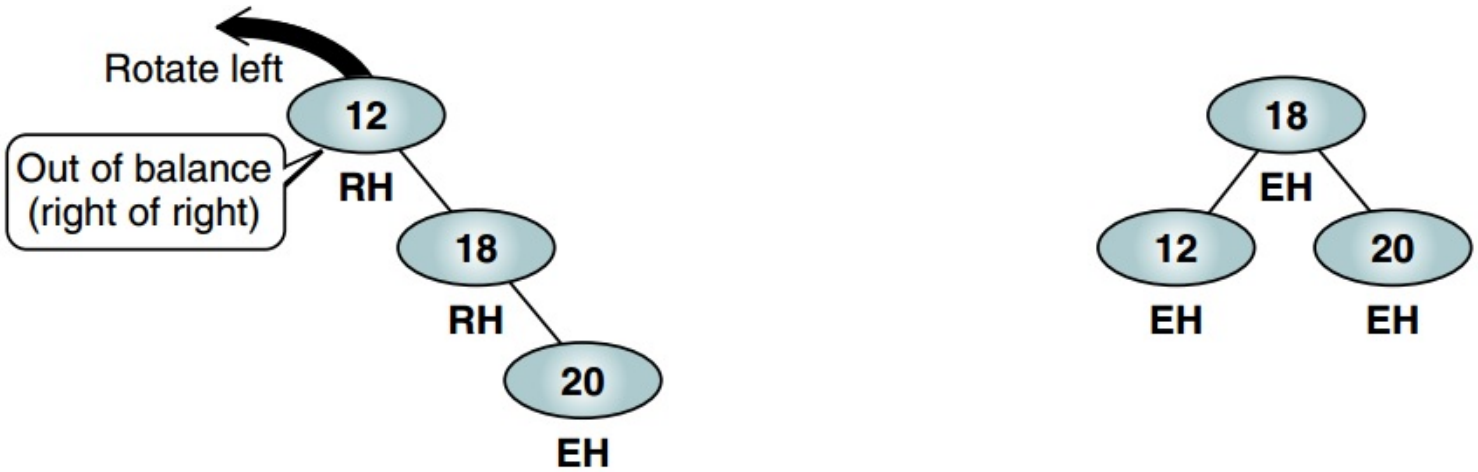
Balancing Trees - Case 1: Left of Left



(Source: Data Structures - A Pseudocode Approach with C++)

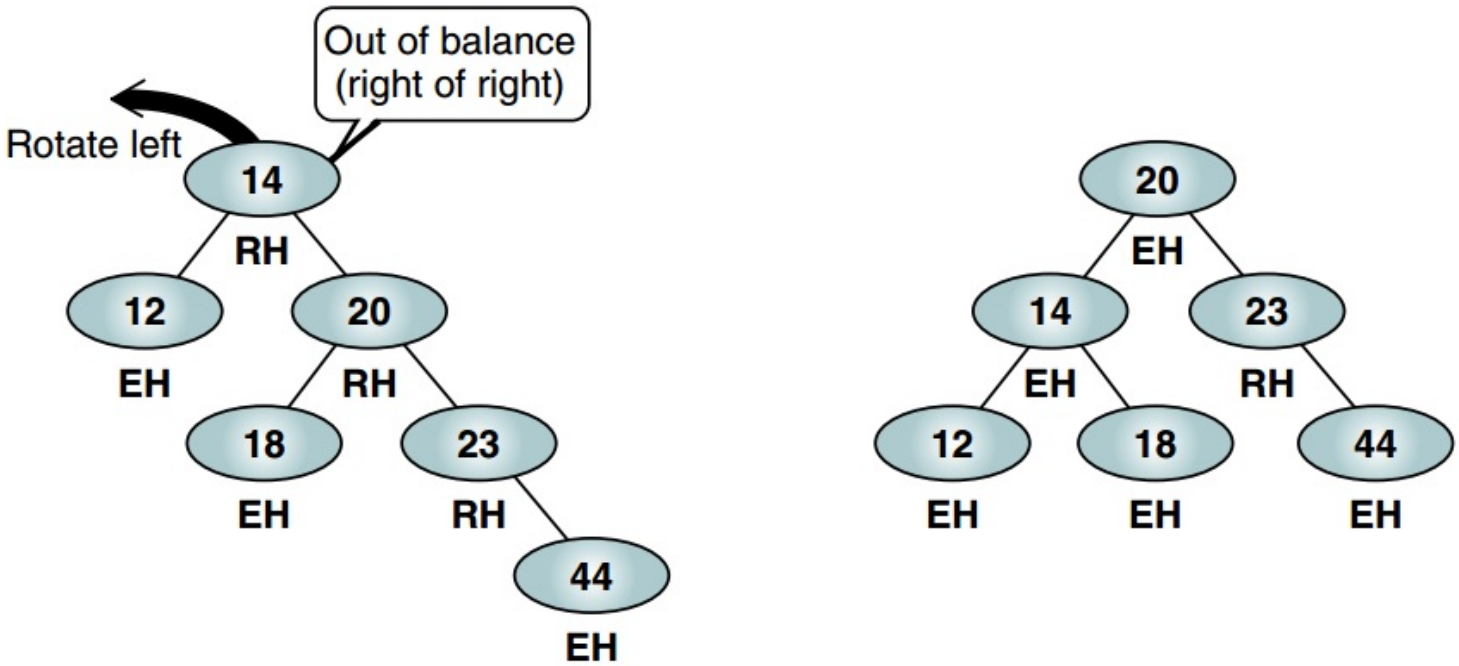
Balancing Trees - Case 2: Right of Right

Out of balance condition created by a right high subtree of a right high tree
→ balance the tree by rotating the out of balance node to the left.



(Source: Data Structures - A Pseudocode Approach with C++)

Balancing Trees - Case 2: Right of Right



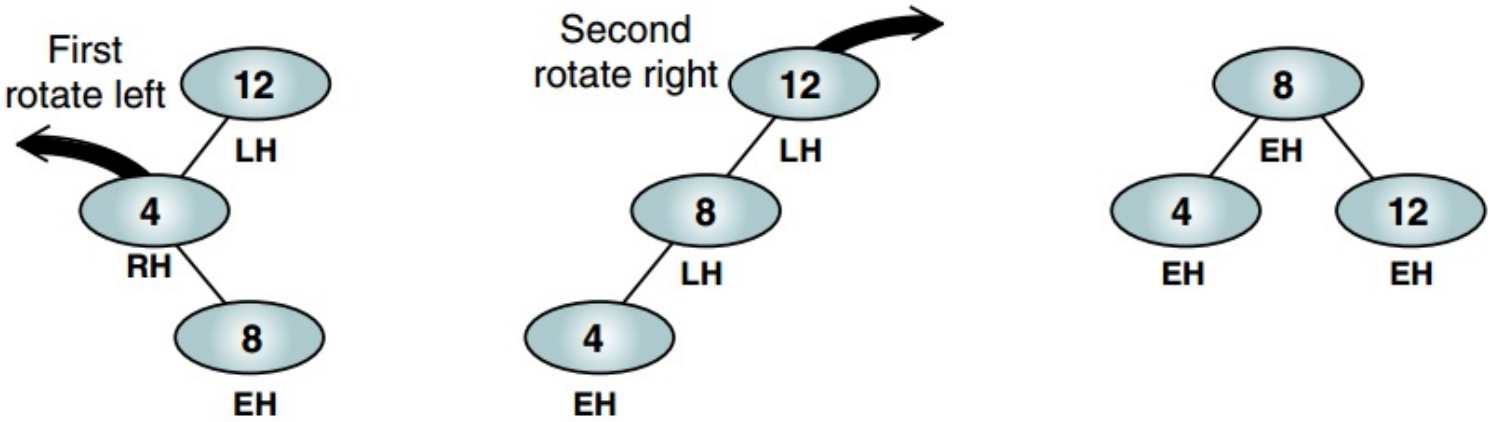
(Source: Data Structures - A Pseudocode Approach with C++)

Balancing Trees - Case 3: Right of Left

Out of balance condition created by a right high subtree of a left high tree

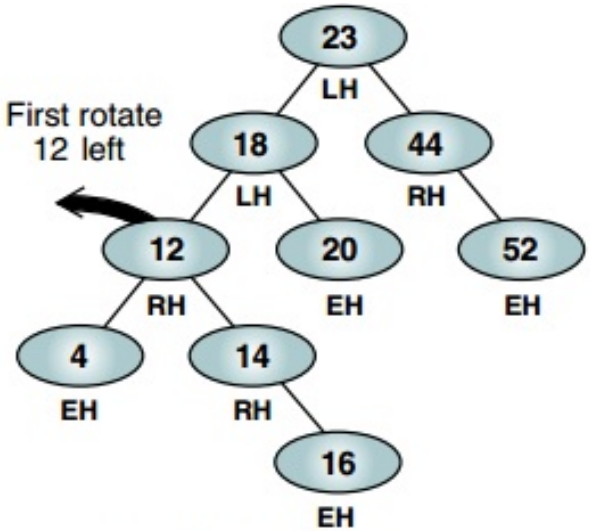
$\rightarrow$ balance the tree by two steps:

1. rotating the left subtree to the left;
2. rotating the root to the right.

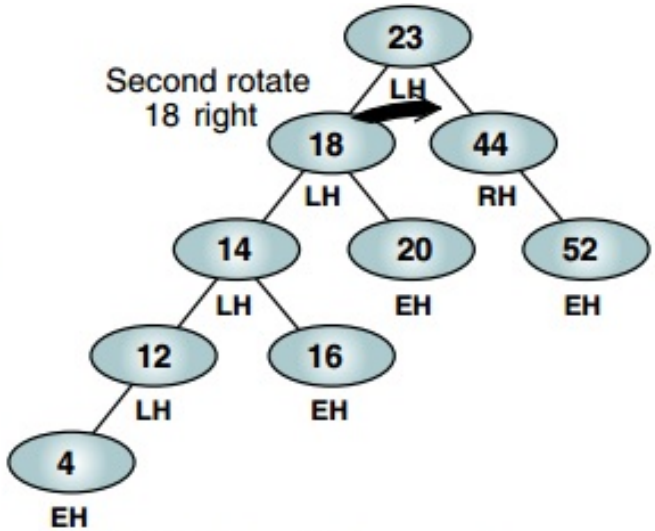


(Source: Data Structures - A Pseudocode Approach with C++)

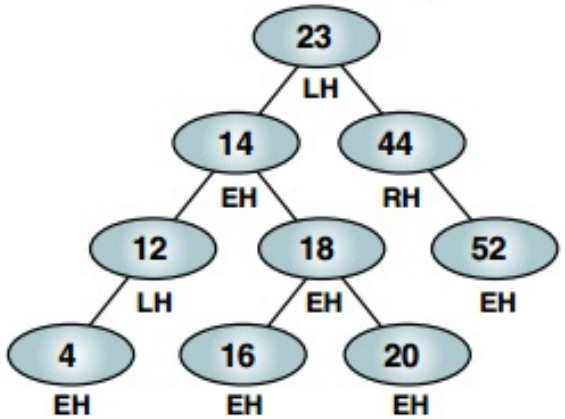
Balancing Trees - Case 3: Right of Left



(b1) Original tree



(b2) After left rotation



(b3) After right rotation

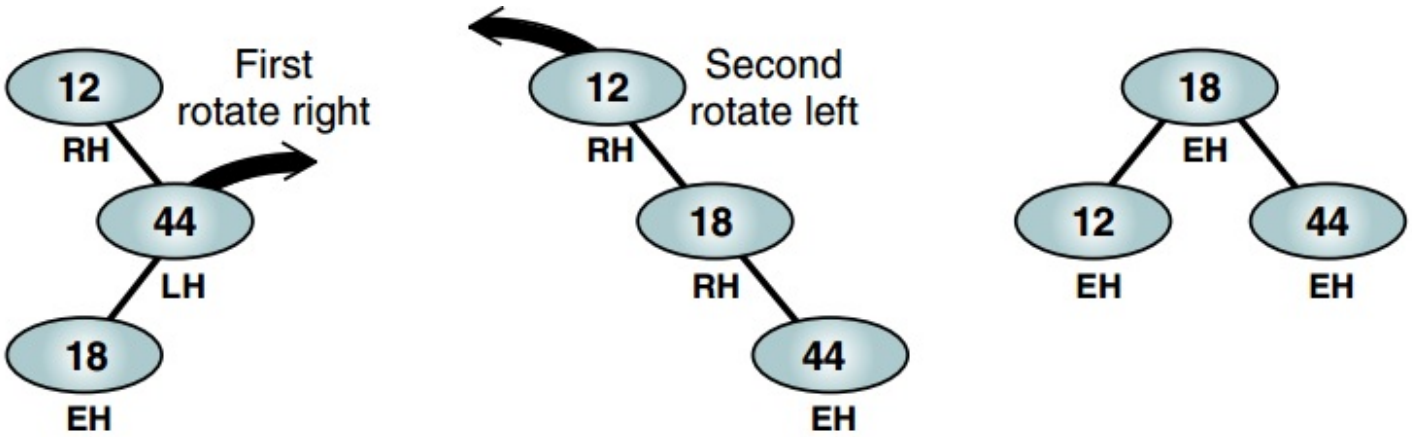
(Source: Data Structures - A Pseudocode Approach with C++)

Balancing Trees - Case 4: Left of Right

Out of balance condition created by a left high subtree of a right high tree

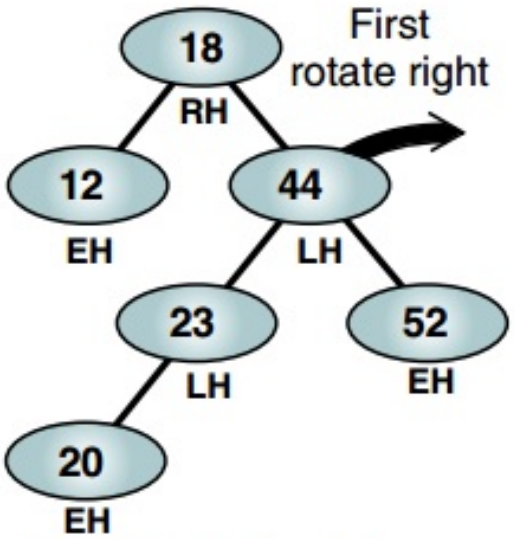
→ balance the tree by two steps:

1. rotating the right subtree to the right;
2. rotating the root to the left.

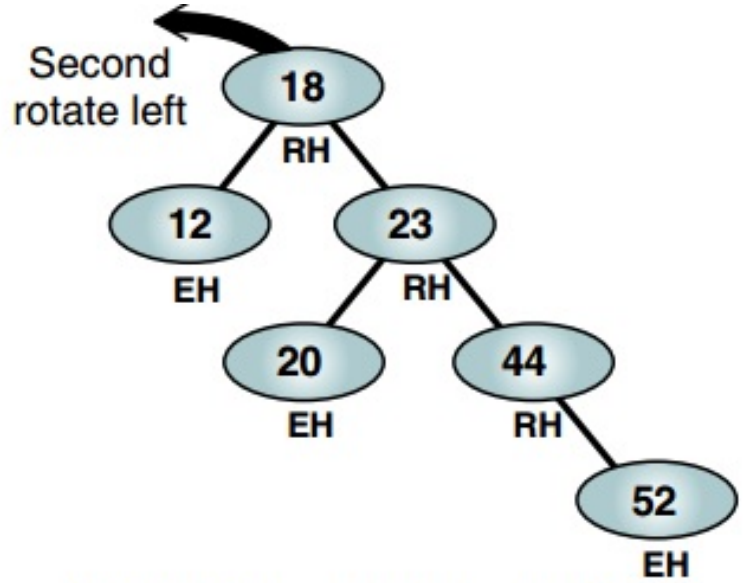


(Source: Data Structures - A Pseudocode Approach with C++)

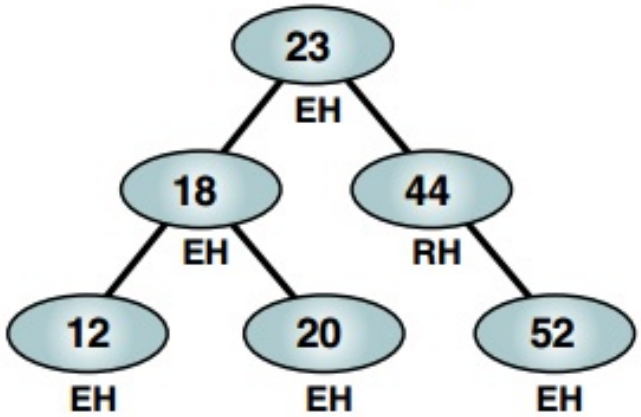
Balancing Trees - Case 4: Left of Right



(b1) Original tree



(b2) After right rotation



(b3) After left rotation

(Source: Data Structures - A Pseudocode Approach with C++)

AVL Tree Operations

AVL Tree Structure

```
node                              avlTree
   data <dataType>                   root <pointer>
   left <pointer>                 end avlTree
   right <pointer>
   balance <balance_factor>
end node
```

```
// General dataTye:
dataType
   key <keyType>
   field1  <...>
   field2  <...>
   ...
   fieldn  <...>
end dataType
```

Note: Array is not suitable for AVL Tree.

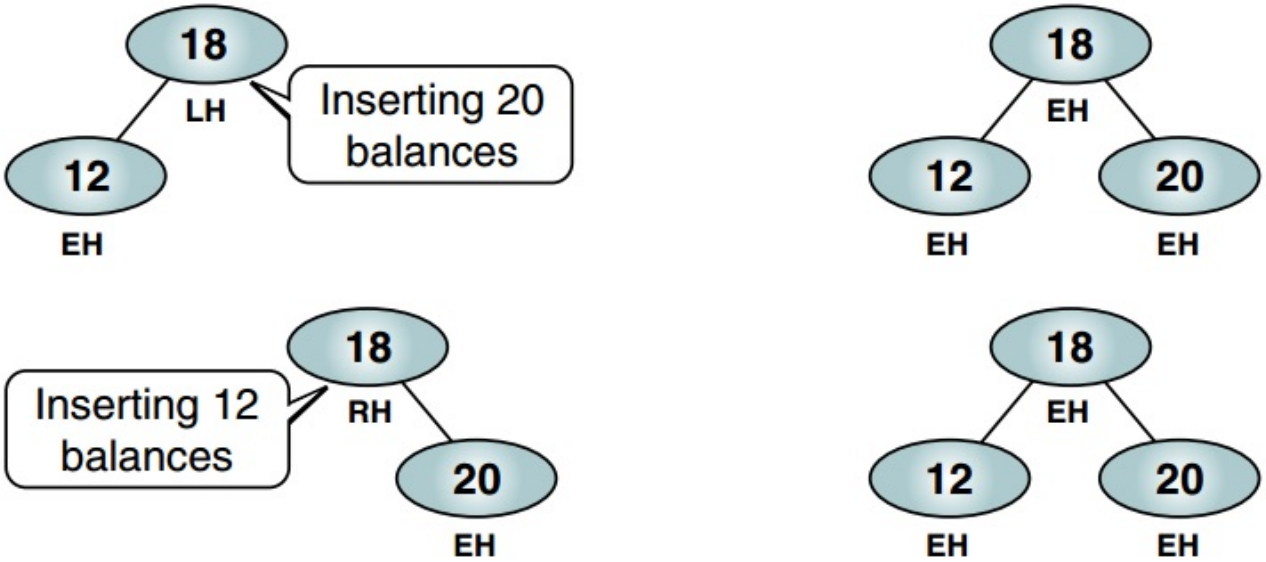
AVL Tree Operations

- Search and retrieval are the same for any binary tree.
- AVL Insert
- AVL Delete

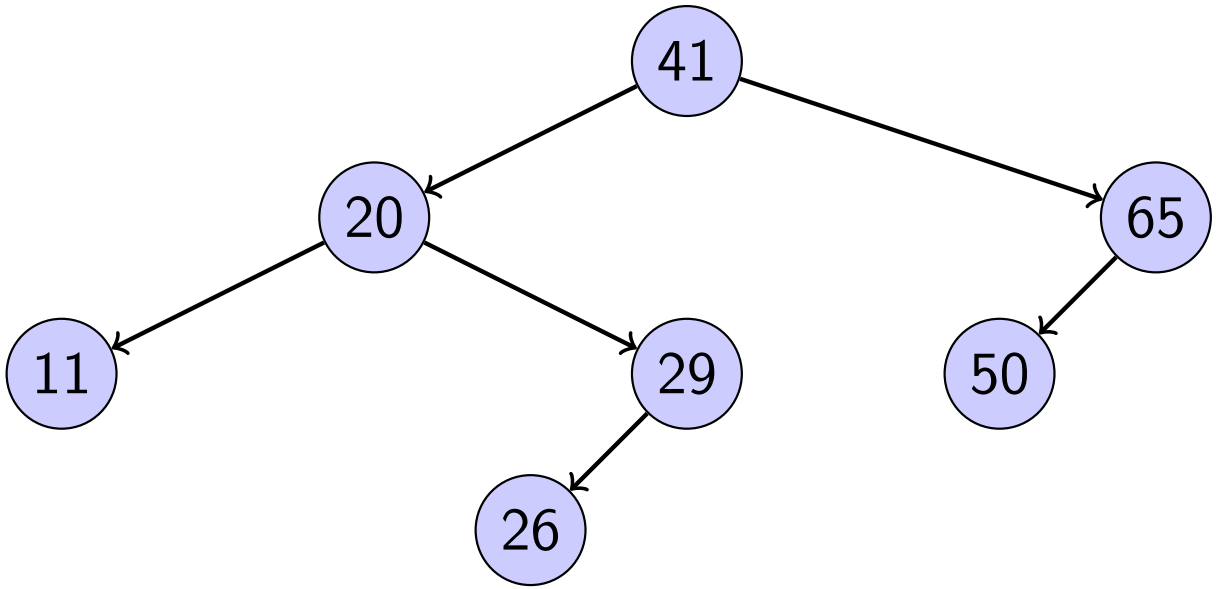
AVL Insert

- Insert can make an out of balance condition.

- Otherwise, some inserts can make an automatic balancing.

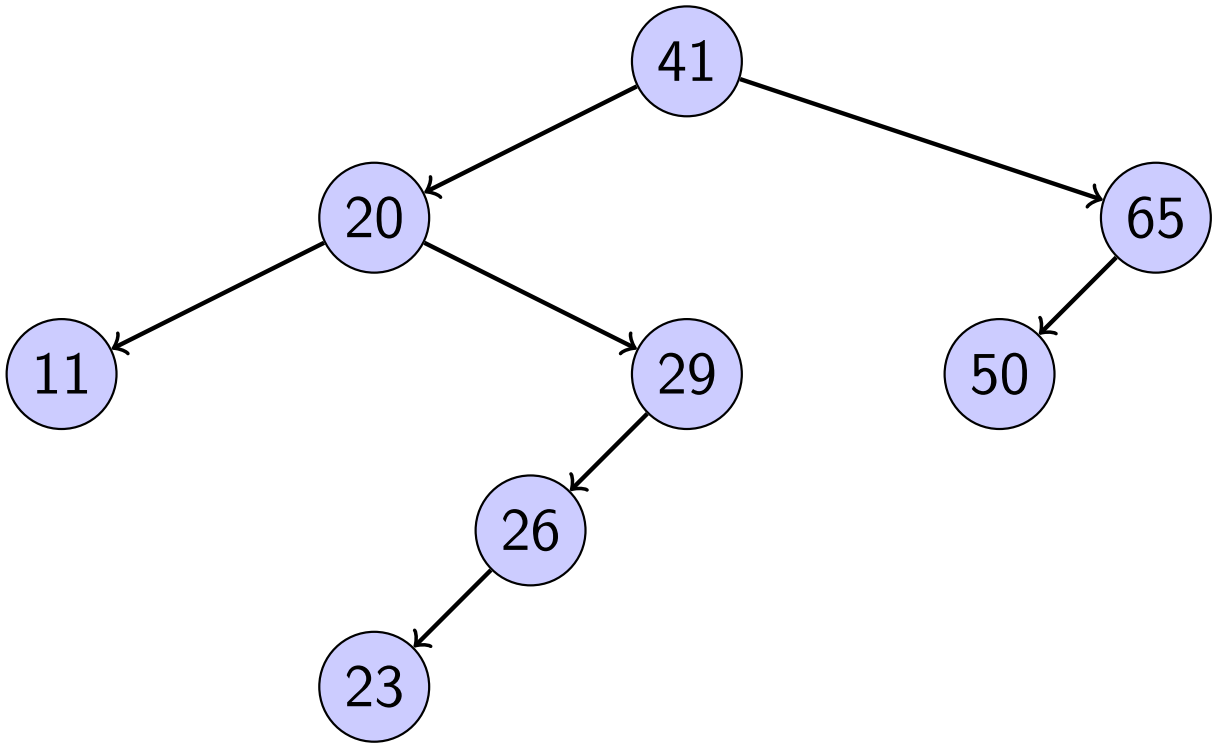


AVL Insertion: Example



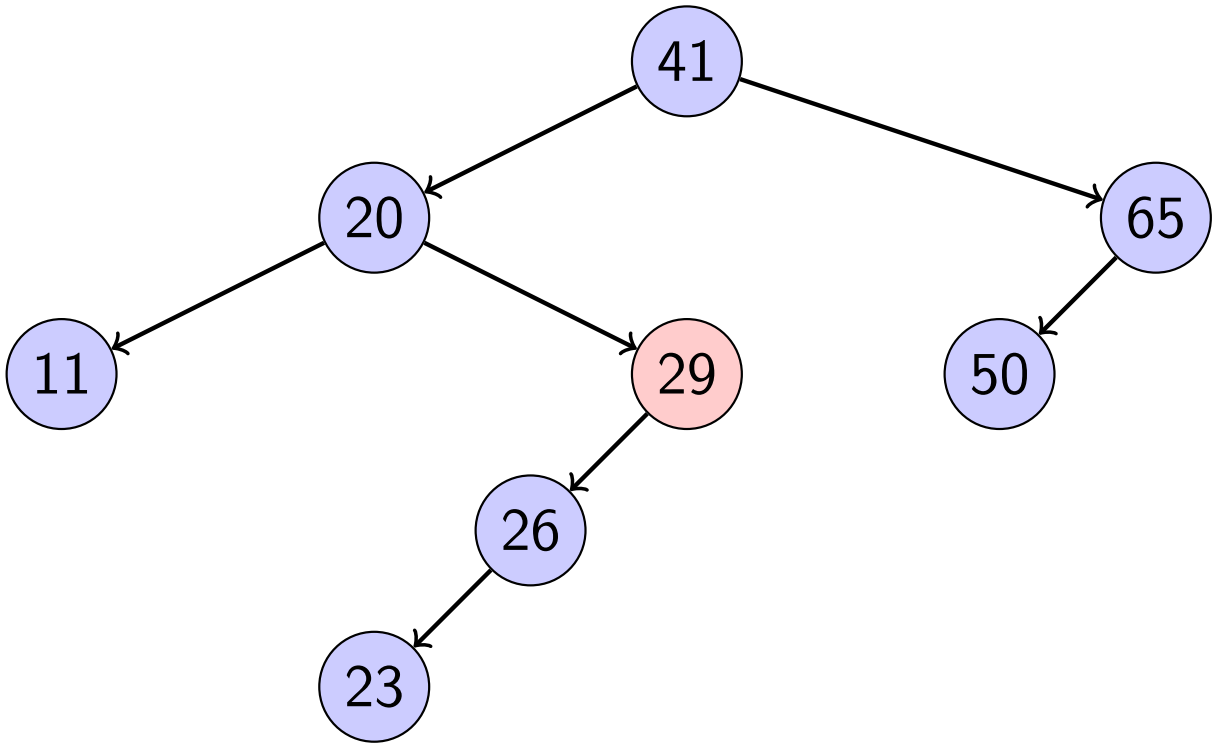
AVL Insertion: Example

Insert **23**



AVL Insertion: Example

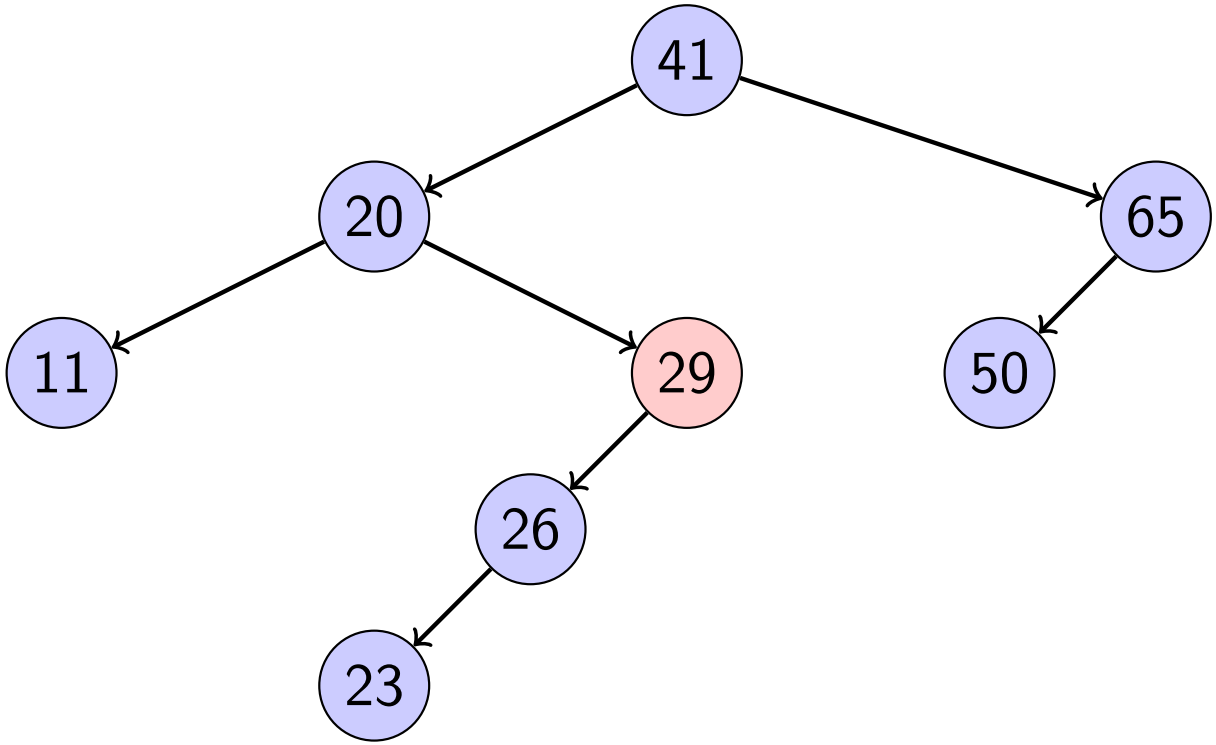
Out of balance: **29**: Left of left



AVL Insertion: Example

Out of balance: **29**: Left of left

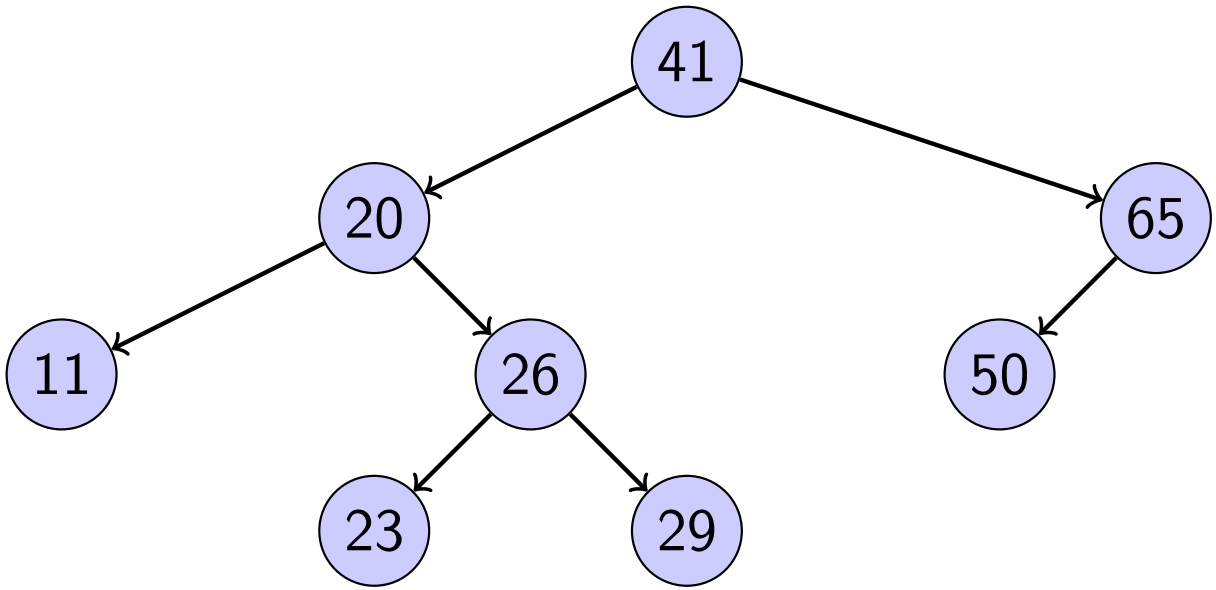
$\Rightarrow$ Rotate right at **29**



AVL Insertion: Example

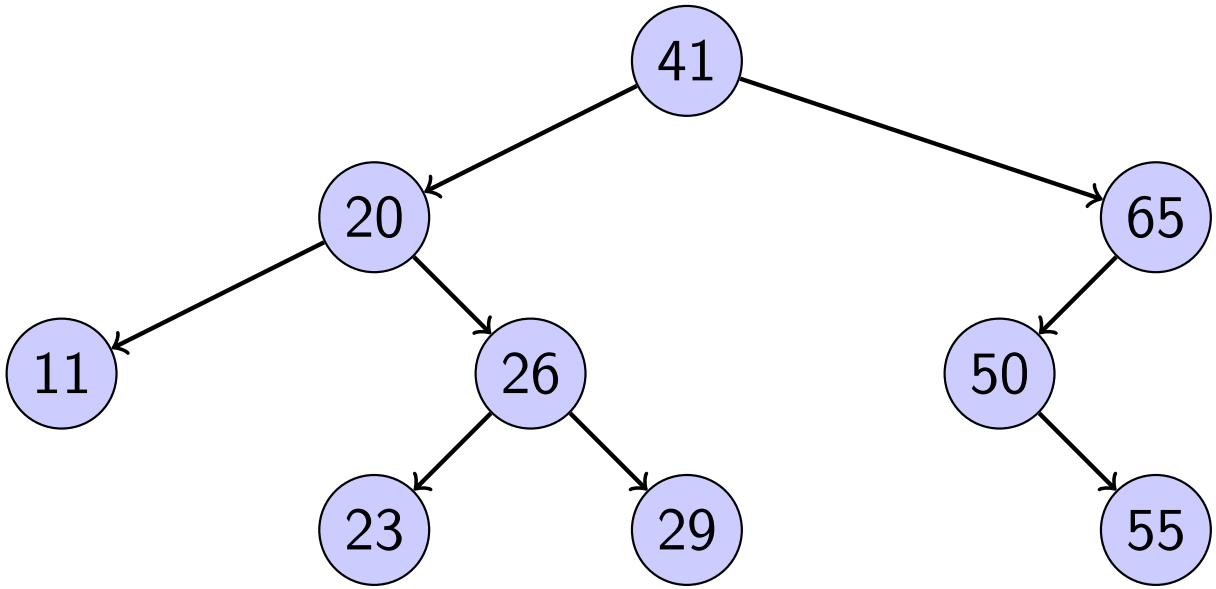
Out of balance: **29**: Left of left

$\Rightarrow$ Rotate right at **29**



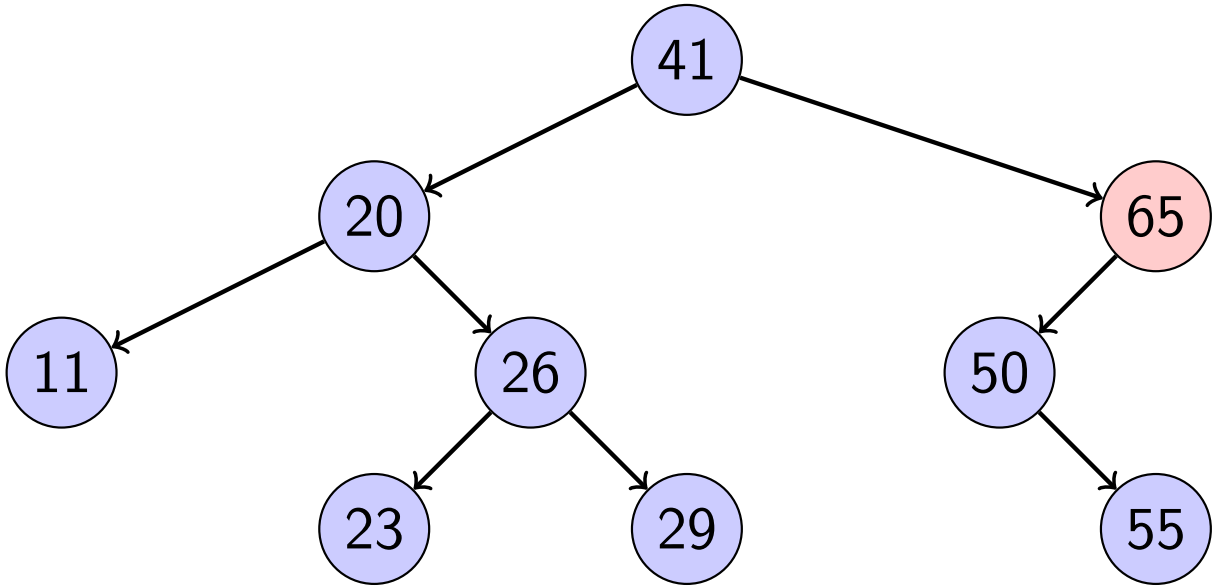
AVL Insertion: Example

Insert **55**



AVL Insertion: Example

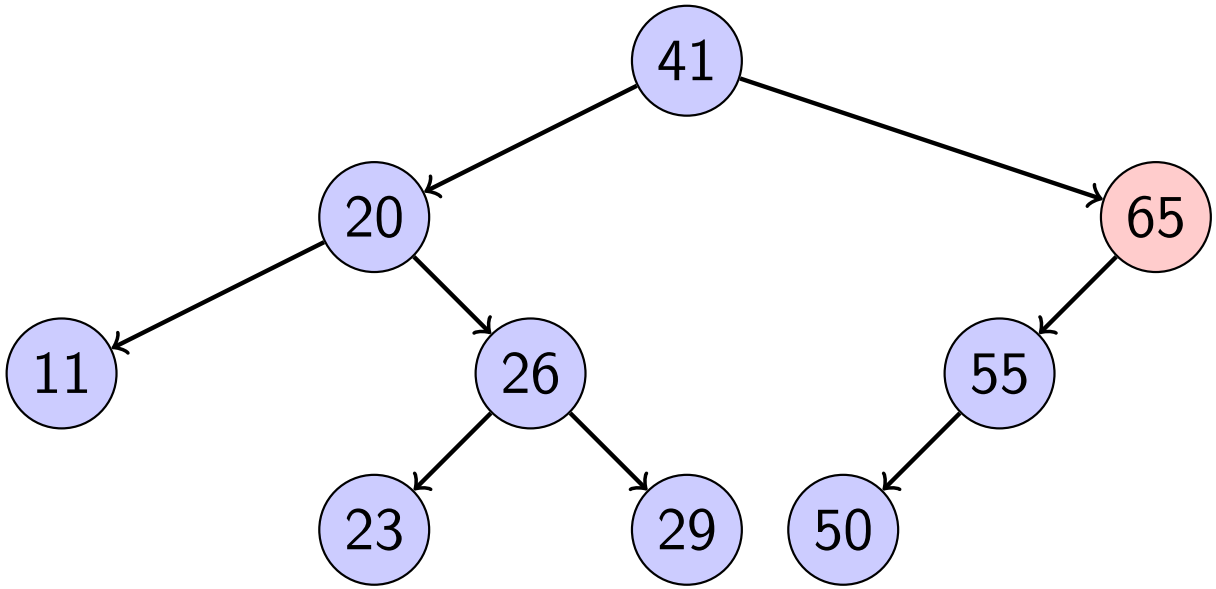
Out of balance: **65**: Left of right



AVL Insertion: Example

Out of balance: **65**: Left of right

$\Rightarrow$ Rotate left at **50**

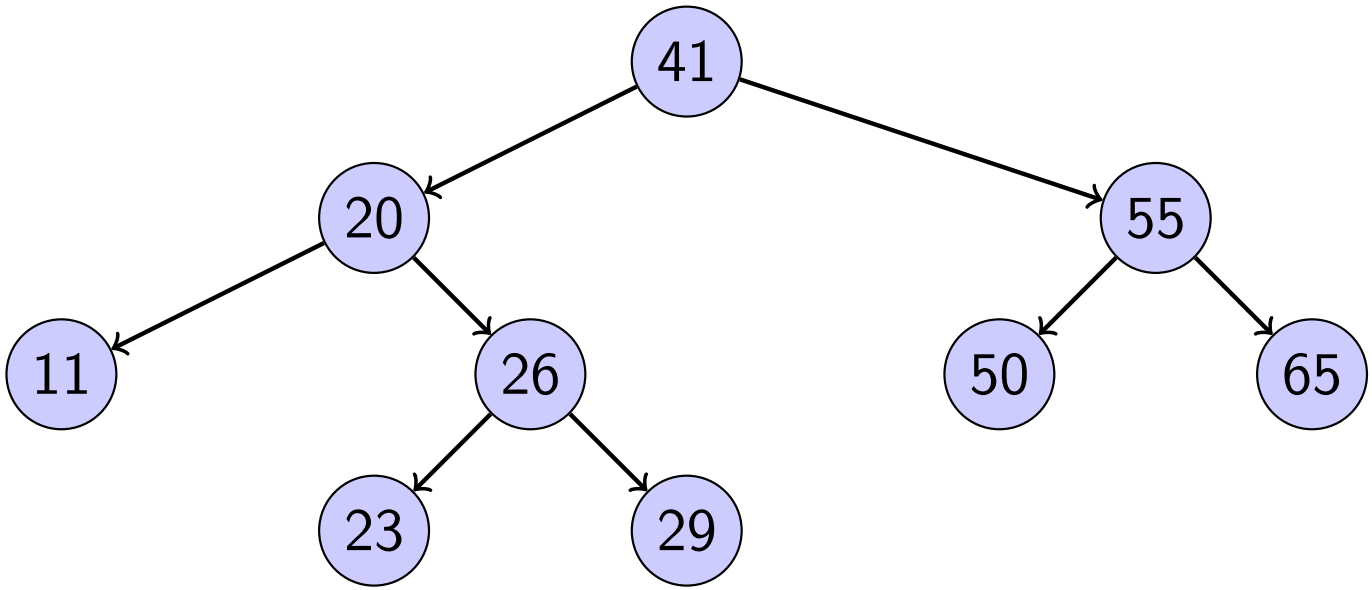


AVL Insertion: Example

Out of balance: **65**: Left of right

$\Rightarrow$ Rotate left at **50**

$\Rightarrow$ Rotate right at **60**



AVL Insert Algorithm

1 **Algorithm** AVLInsert(ref root <pointer>, val newPtr <pointer>, ref taller <boolean>)
2 Using recursion, insert a node into an AVL tree.

3 **Pre:** `root` is a pointer to first node in AVL tree/subtree
4 `newPtr` is a pointer to new node to be inserted
5 **Post:** `taller` is a Boolean: `true` indicating the subtree height has increased, `false` indicating same height
6 **Return** `root` returned recursively up the tree

AVL Insert Algorithm

```
1 // Insert at root
2 if root null then
3 |   root = newPtr
4 |   taller = true
5 |   return root
6 end
```


AVL Insert Algorithm

```
1  if newPtr->data.key < root->data.key
     then
2      root->left = AVLInsert(root->left,
         newPtr, taller)
3      // Left subtree is taller
4      if taller then
5          if root is LH then
6              root = leftBalance(root, taller)
7          else if root is EH then
8              root->balance = LH
9          else
10             root->balance = EH
11             taller = false
12         end
```


AVL Insert Algorithm

```
1  else
2      root->right = AVLInsert(root->right, newPtr,
         taller)
3      // Right subtree is taller
4      if taller then
5          if root is LH then
6              root->balance = EH
7              taller = false
8          else if root is EH then
9              root->balance = RH
10         else
11             root = rightBalance(root, taller)
12         end
13     end
14 end
15 return root
16 End AVLInsert
```


AVL Left Balance Algorithm

1 **Algorithm** leftBalance(ref root <pointer>, ref taller <boolean>)
2 This algorithm is entered when the left subtree is higher than the right subtree.

3 **Pre:** `root` is a pointer to the root of the [sub]tree
4 `taller` is `true`
5 **Post:** `root` has been updated (if necessary)
6 `taller` has been updated

AVL Left Balance Algorithm

```
1 leftTree = root->left

2 // Case 1: Left of left. Single rotation
    right.
3 if leftTree is LH then
4 |   root = rotateRight(root)
5 |   root->balance = EH
6 |   leftTree->balance = EH
7 |   taller = false
```


AVL Left Balance Algorithm

```
1  // Case 2: Right of Left. Double rotation required.
2  else
3      rightTree = leftTree->right
4      if rightTree->balance = LH then
5          root->balance = RH
6          leftTree->balance = EH
7      else if rightTree->balance = EH then
8          leftTree->balance = EH
9      else
10         root->balance = EH
11         leftTree->balance = LH
12     end
13     rightTree->balance = EH
14     root->left = rotateLeft(leftTree)
15     root = rotateRight(root)
16     taller = false
17 end
18 return root
```


AVL Right Balance Algorithm

1 **Algorithm** rightBalance(ref root <pointer>, ref taller <boolean>)
2 This algorithm is entered when the right subtree is higher than the left subtree.

3 **Pre:** `root` is a pointer to the root of the [sub]tree
4 `taller` is `true`
5 **Post:** `root` has been updated (if necessary)
6 `taller` has been updated

AVL Right Balance Algorithm

```
1 rightTree = root->right

2 // Case 1: Right of right. Single rotation
    left.
3 if rightTree is RH then
4 |   root = rotateLeft(root)
5 |   root->balance = EH
6 |   rightTree->balance = EH
7 |   taller = false
```


AVL Right Balance Algorithm

```
1  // Case 2: Left of Right. Double rotation required.
2  else
3      leftTree = rightTree->left
4      if leftTree->balance = RH then
5          root->balance = LH
6          rightTree->balance = EH
7      else if leftTree->balance = EH then
8          rightTree->balance = EH
9      else
10         root->balance = EH
11         rightTree->balance = RH
12     end
13     leftTree->balance = EH
14     root->right = rotateRight(rightTree)
15     root = rotateLeft(root)
16     taller = false
17 end
18 return root
```


AVL Delete Algorithm

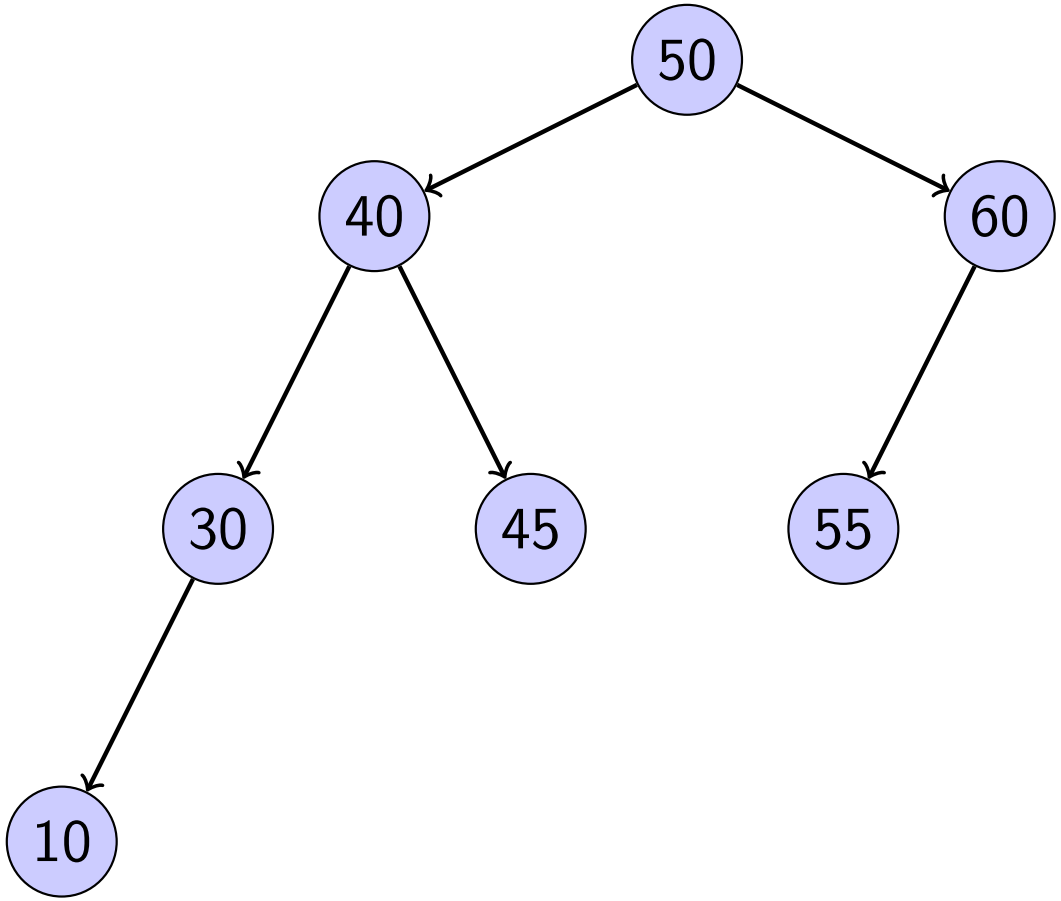
The AVL delete follows the basic logic of the binary search tree delete with the addition of the logic to balance the tree. As with the insert logic, the balancing occurs as we back out of the tree.

1 **Algorithm** AVLDelete(ref root <pointer>, val deleteKey <key>, ref shorter <boolean>, ref success <boolean>)
2 This algorithm deletes a node from an AVL tree and rebalances if necessary.

3 **Pre:** `root` is a pointer to the root of the [sub]tree
4 `deleteKey` is the key of node to be deleted
5 **Post:** node deleted if found, tree unchanged if not found
6 `shorter` is `true` if subtree is shorter
7 `success` is `true` if deleted, `false` if not found
8 **Return** pointer to root of (potential) new subtree

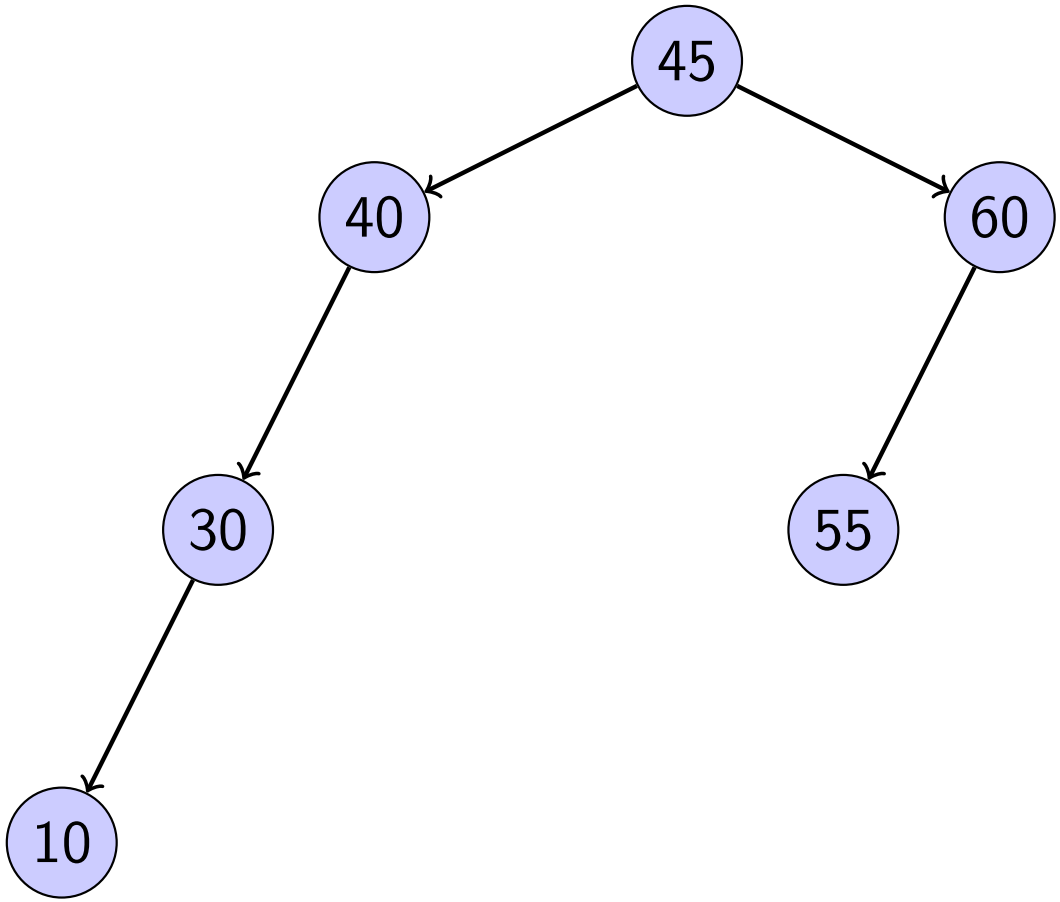
AVL Delete: Example

Delete **50**:



AVL Delete: Example

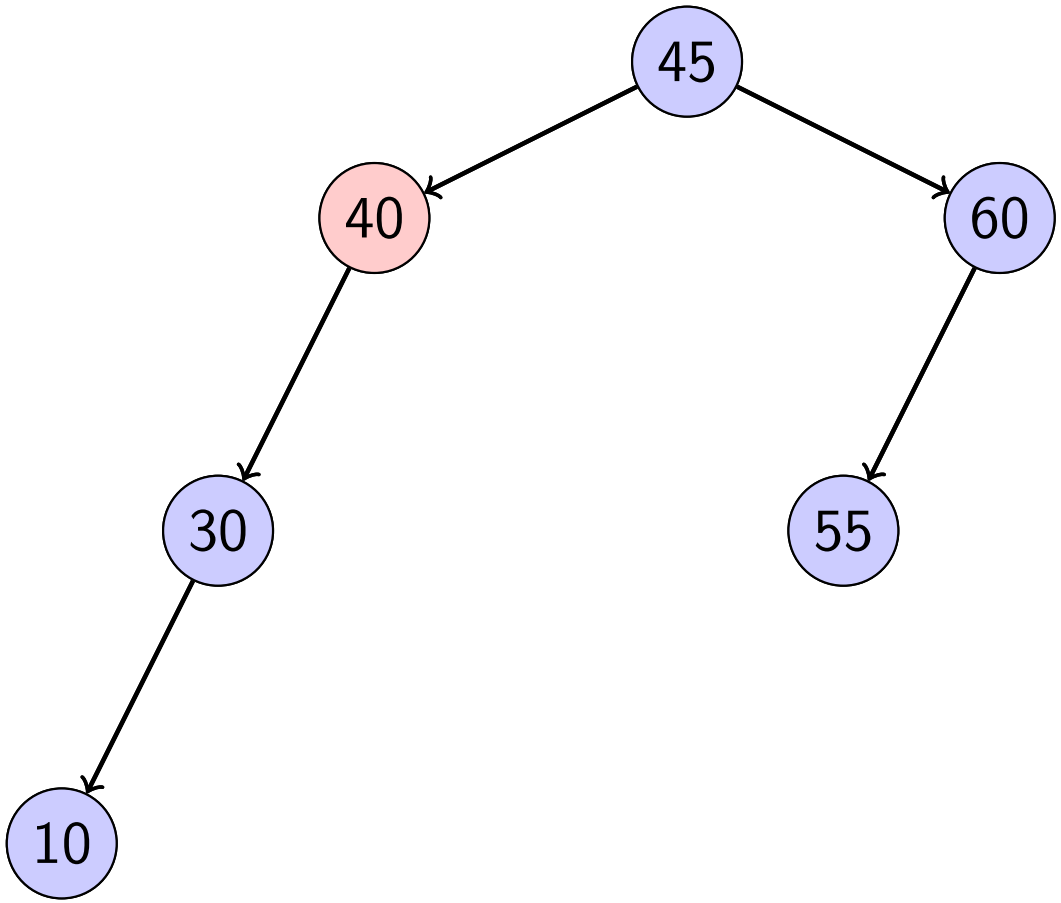
Delete **50**: Use **45** to replace



AVL Delete: Example

Delete **50**: Use **45** to replace

$\Rightarrow$ Out of balance at **40**: Left of left

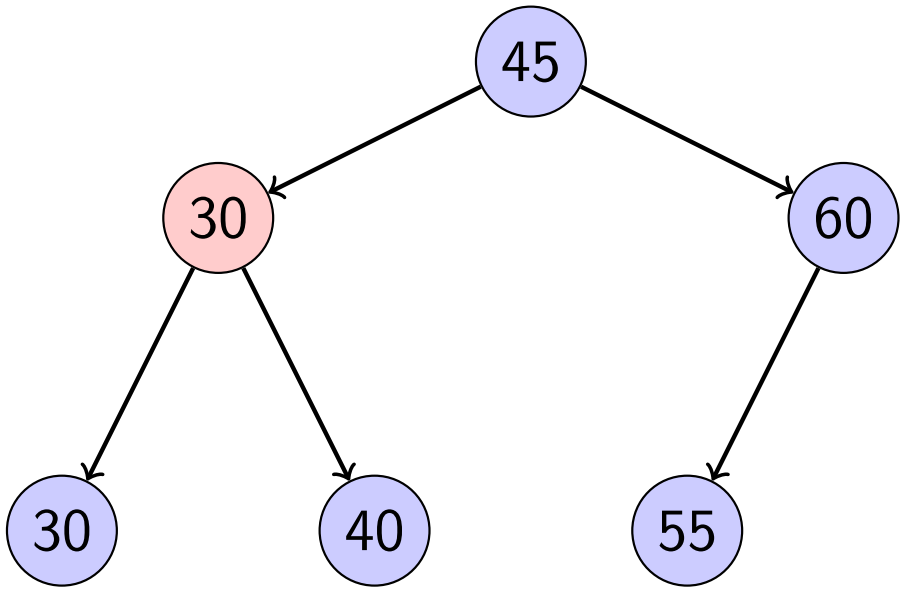


AVL Delete: Example

Delete **50**: Use **45** to replace

$\Rightarrow$ Out of balance at **40**: Left of left

$\Rightarrow$ Rotate right at **40**



AVL Delete Algorithm

```
1  if tree null then
2  |   shorter = false
3  |   success = false
4  |   return null
5  end
6  if deleteKey < root->data.key then
7  |   root->left = AVLDelete(root->left, deleteKey,
   |     shorter, success)
8  |   if shorter then
9  |   |   root = deleteRightBalance(root, shorter)
10 |   end
11 else if deleteKey > root->data.key then
12 |   root->right = AVLDelete(root->right,
   |     deleteKey, shorter, success)
13 |   if shorter then
14 |   |   root = deleteLeftBalance(root, shorter)
15 |   end
```


AVL Delete Algorithm

```
1  // Delete node found – test for leaf node
2  else
3      deleteNode = root
4      if no right subtree then
5          newRoot = root->left
6          success = true
7          shorter = true
8          recycle(deleteNode)
9          return newRoot
10     else if no left subtree then
11         newRoot = root->right
12         success = true
13         shorter = true
14         recycle(deleteNode)
15         return newRoot
```


AVL Delete Algorithm

```
1  else
2      // ... // Delete node has two subtrees
3      else
4          exchPtr = root->left
5          while exchPtr->right not null do
6              exchPtr = exchPtr->right
7          end
8          root->data = exchPtr->data
9          root->left = AVLDelete(root->left,
               exchPtr->data.key, shorter, success)
10         if shorter then
11             root = deleteRightBalance(root,
                   shorter)
12         end
13     end
14 end
15 Return root
16 End AVLDelete
```


Delete Right Balance

```
1  Algorithm deleteRightBalance(ref root <pointer>,
     ref shorter <boolean>)
2  The (sub)tree is shorter after a deletion on the left
     branch. Adjust the balance factors and if necessary
     balance the tree by rotating left.

3  Pre: tree is shorter
4  Post: balance factors updated and balance restored
5  root updated
6  shorter updated

7  if root LH then
8  |    root->balance = EH
9  else if root EH then
10 |    root->balance = RH
11 |    shorter = false
```


Delete Right Balance

```
1  else
2      rightTree = root->right
3      if rightTree LH then
4          leftTree = rightTree->left
5          if leftTree LH then
6              rightTree->balance = RH
7              root->balance = EH
8          else if leftTree EH then
9              root->balance = LH
10             rightTree->balance = EH
11         else
12             root->balance = LH
13             rightTree->balance = EH
14         end
15         leftTree->balance = EH
16         root->right = rotateRight(rightTree)
17         root = rotateLeft(root)
```


Delete Right Balance

```
1  else
2      // ...
3      else
4          if rightTree not EH then
5              root->balance = EH
6              rightTree->balance = EH
7          else
8              root->balance = RH
9              rightTree->balance = LH
10             shorter = false
11         end
12         root = rotateLeft(root)
13     end
14 end
15 return root
16 End deleteRightBalance
```


Delete Left Balance

```
1  Algorithm deleteLeftBalance(ref root <pointer>,
     ref shorter <boolean>)
2  The (sub)tree is shorter after a deletion on the right
     branch. Adjust the balance factors and if necessary
     balance the tree by rotating right.

3  Pre: tree is shorter
4  Post: balance factors updated and balance restored
5  root updated
6  shorter updated

7  if root RH then
8  |    root->balance = EH
9  else if root EH then
10 |    root->balance = LH
11 |    shorter = false
```


Delete Left Balance

```
1  else
2      leftTree = root->left
3      if leftTree RH then
4          rightTree = leftTree->right
5          if rightTree RH then
6              leftTree->balance = LH
7              root->balance = EH
8          else if rightTree EH then
9              root->balance = RH
10             leftTree->balance = EH
11         else
12             root->balance = RH
13             leftTree->balance = EH
14         end
15         rightTree->balance = EH
16         root->left = rotateLeft(leftTree)
17         root = rotateRight(root)
```


Delete Left Balance

```
1  else
2      // ...
3      else
4          if leftTree not EH then
5              root->balance = EH
6              leftTree->balance = EH
7          else
8              root->balance = LH
9              leftTree->balance = RH
10             shorter = false
11         end
12         root = rotateRight(root)
13     end
14 end
15 return root
16 End deleteLeftBalance
```

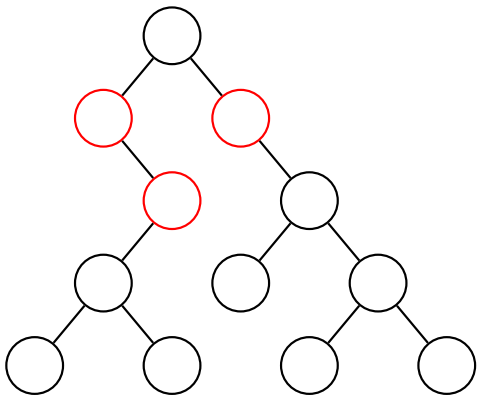

Splay Tree

Non-uniform input sequences

- Search for random elements $O(\log n)$ best possible.

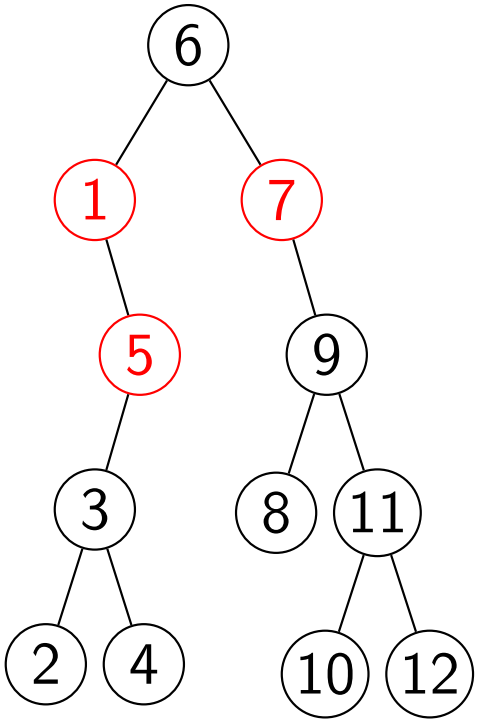
Non-uniform input sequences

- Search for random elements $O(\log n)$ best possible.
- If some items more frequent than others, can do better putting frequent queries near root.



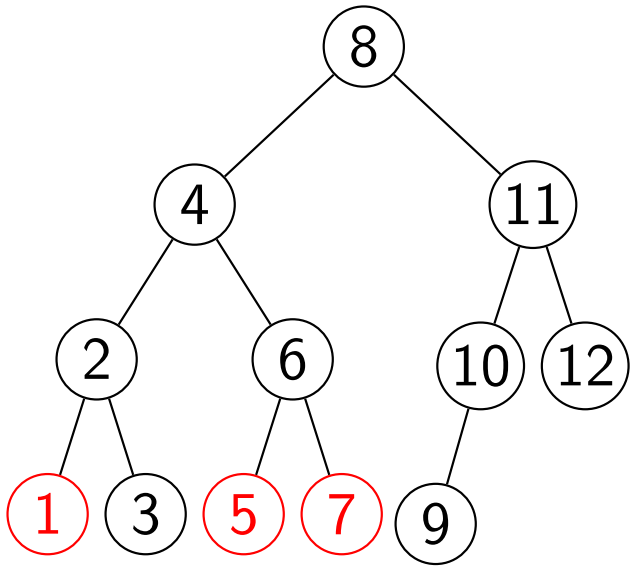
Comparision

UNBALANCED



Total: 0

BALANCED

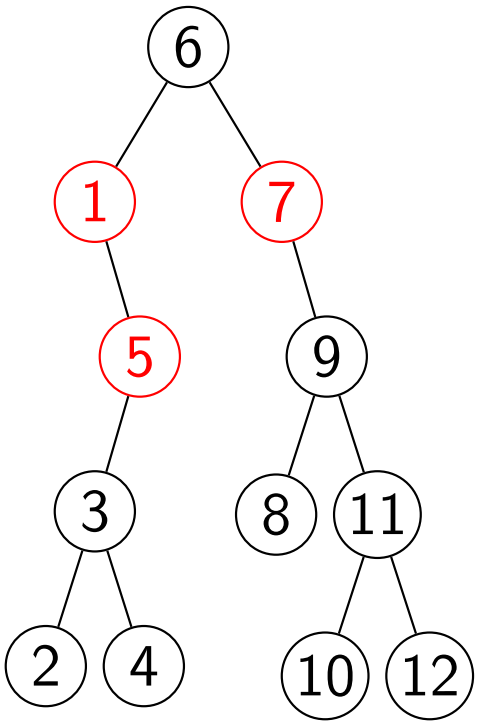


Total: 0

Comparision

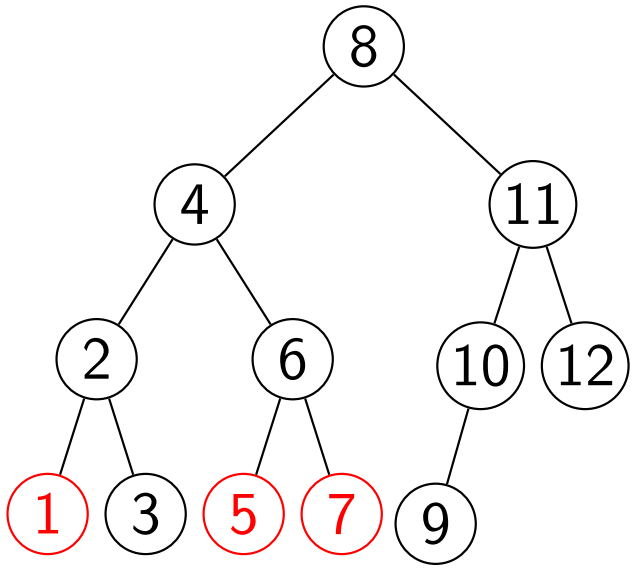
Find 11

UNBALANCED



Total: 4

BALANCED

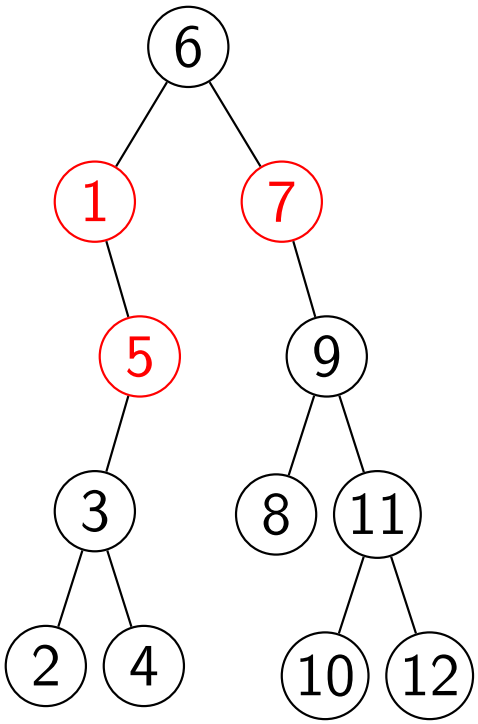


Total: 2

Comparision

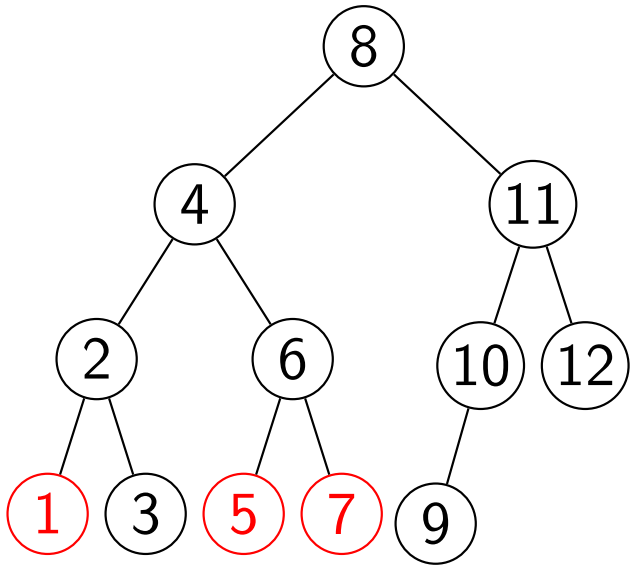
Find 1 (first)

UNBALANCED



Total: 6

BALANCED

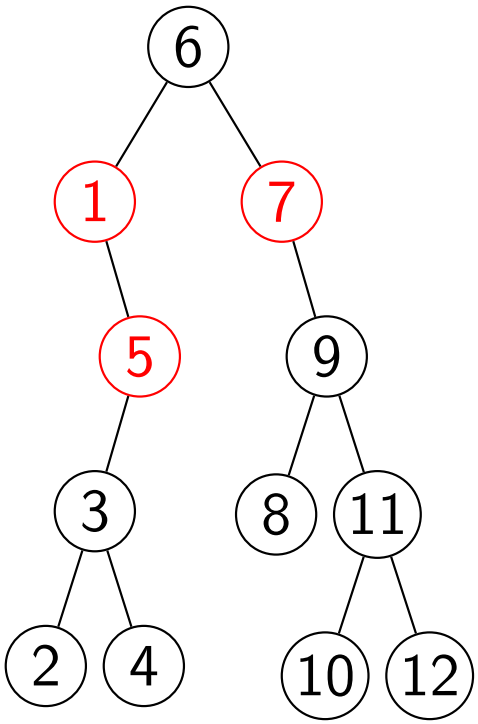


Total: 6

Comparision

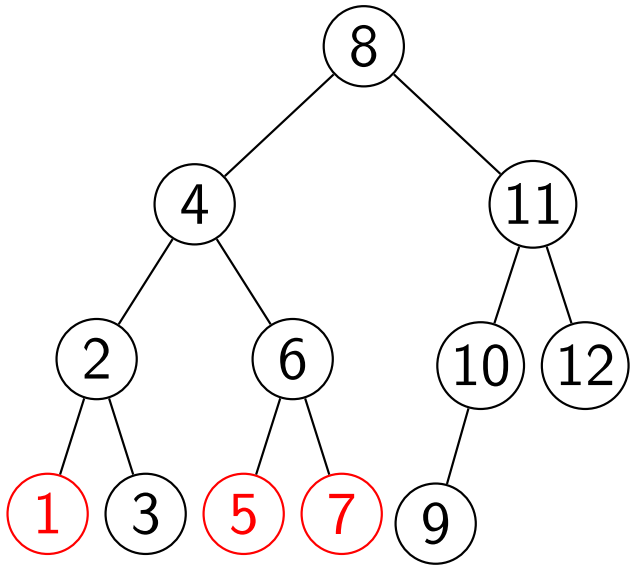
Find 1 (second)

UNBALANCED



Total: 8

BALANCED

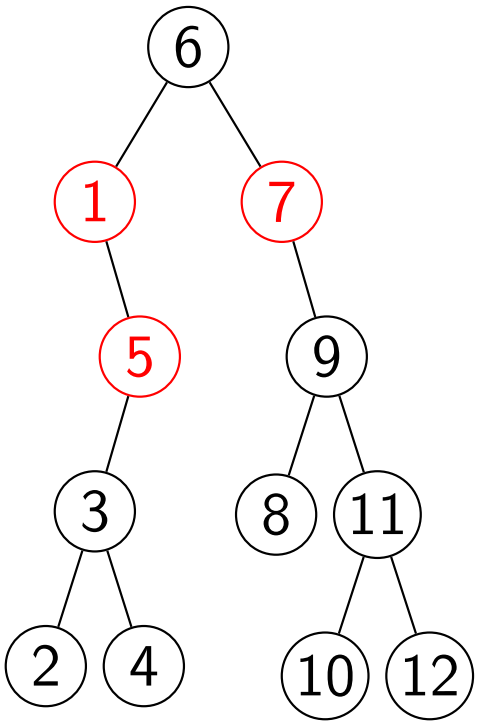


Total: 10

Comparision

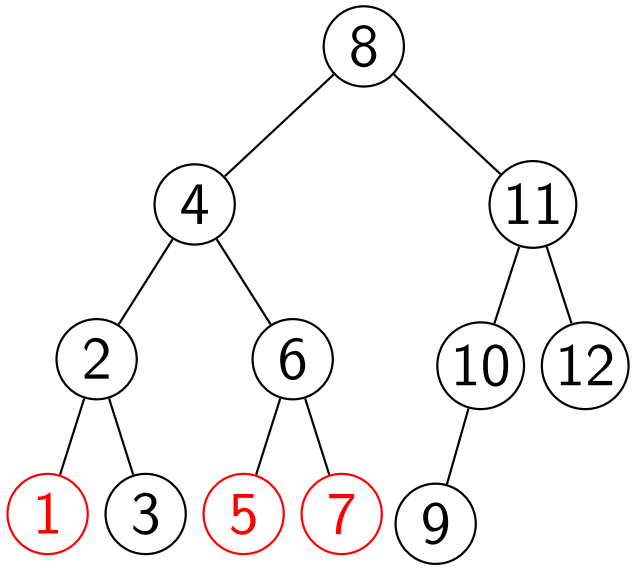
Find 7

UNBALANCED



Total: 10

BALANCED

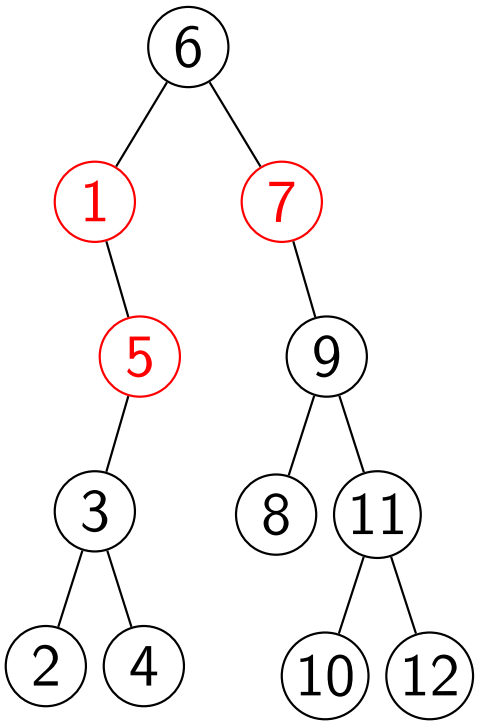


Total: 14

Comparision

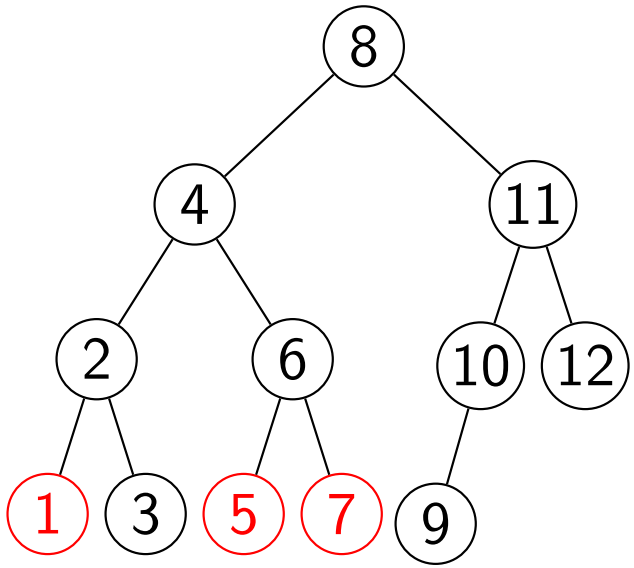
Find 4

UNBALANCED



Total: 15

BALANCED

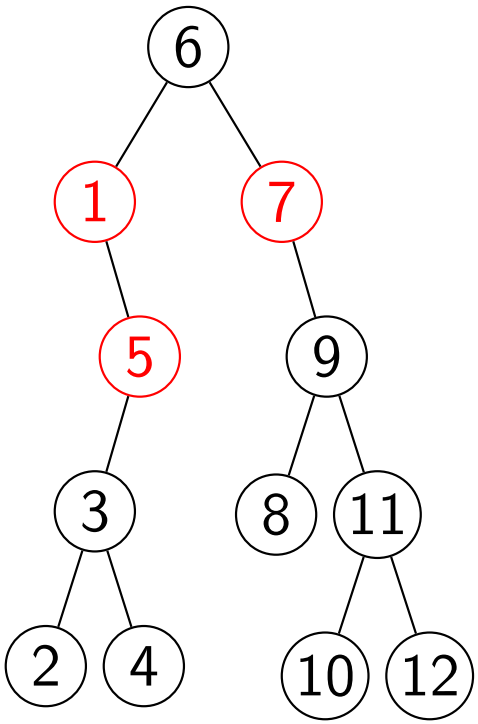


Total: 16

Comparision

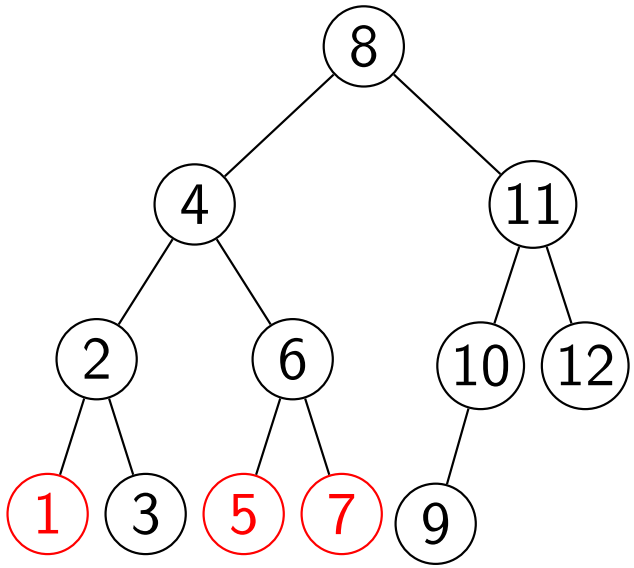
Find 2

UNBALANCED



Total: 20

BALANCED

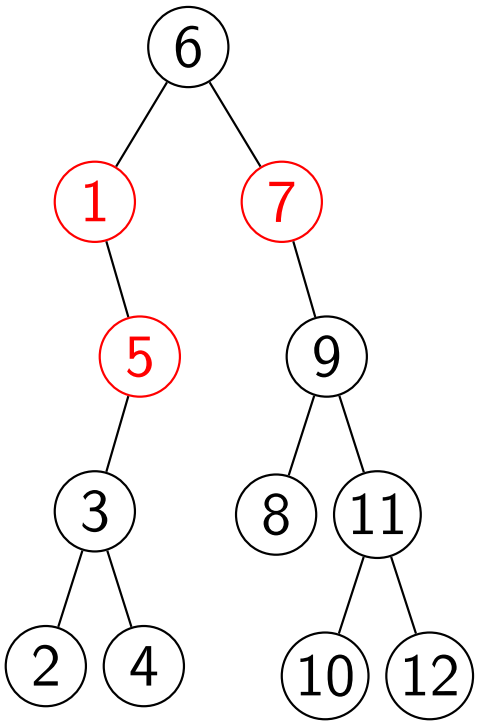


Total: 19

Comparision

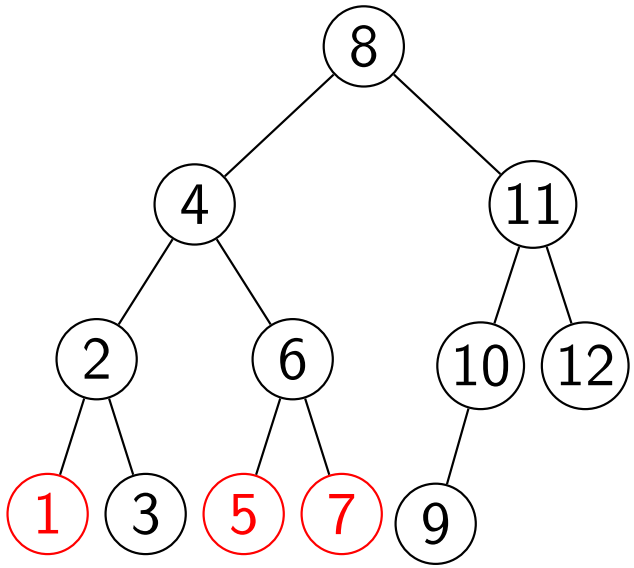
Find 1

UNBALANCED



Total: 22

BALANCED



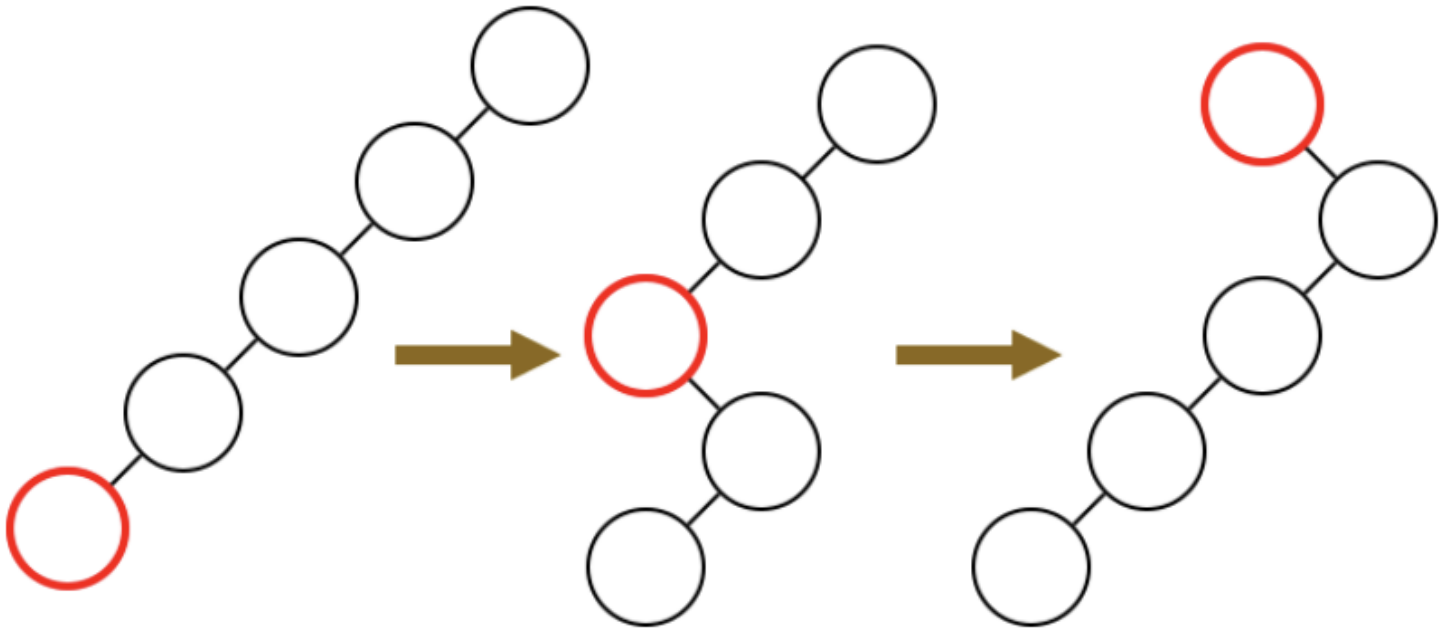
Total: 23

Idea

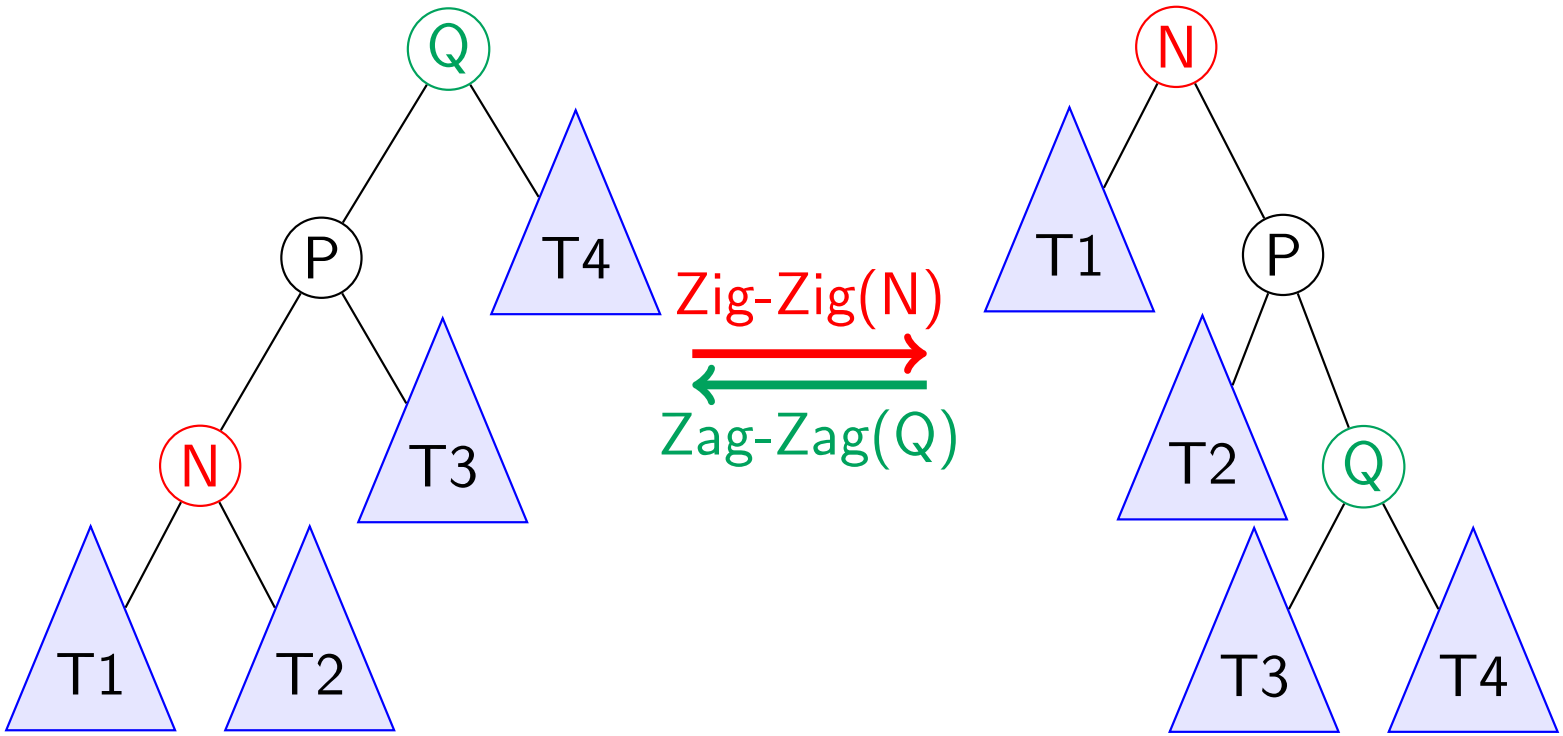
Bring query node to the root.

Idea

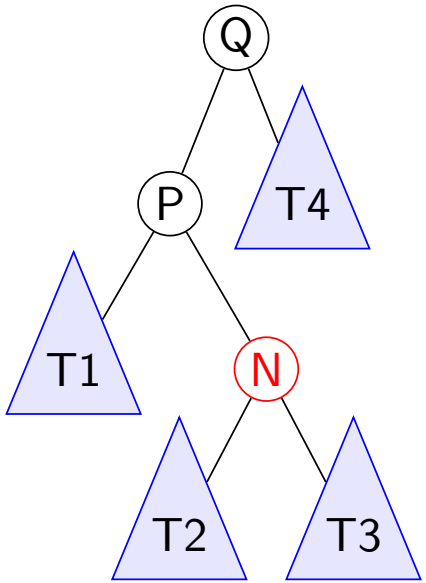
With simple idea: Just rotate to top. Doesn't work.



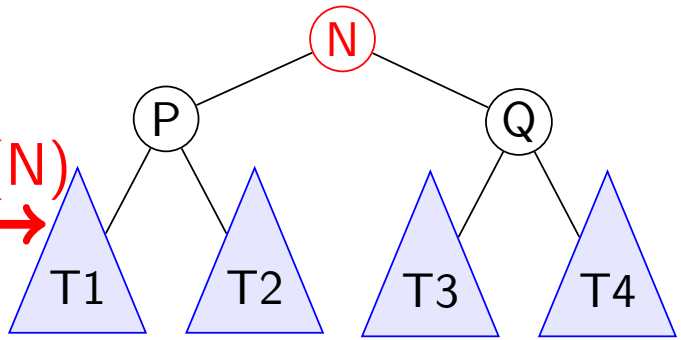
Modification: Zig-Zig/ Zag-Zag

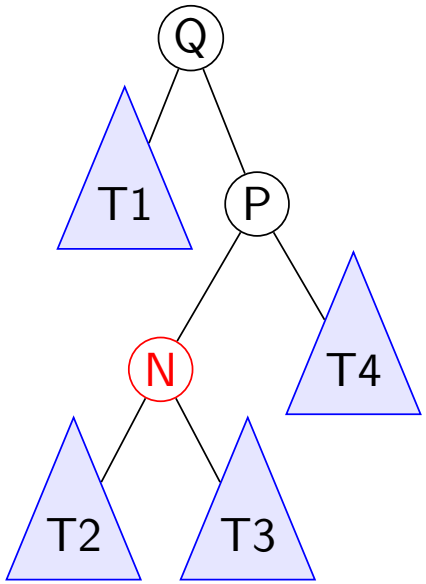


Modification: Zig-Zag

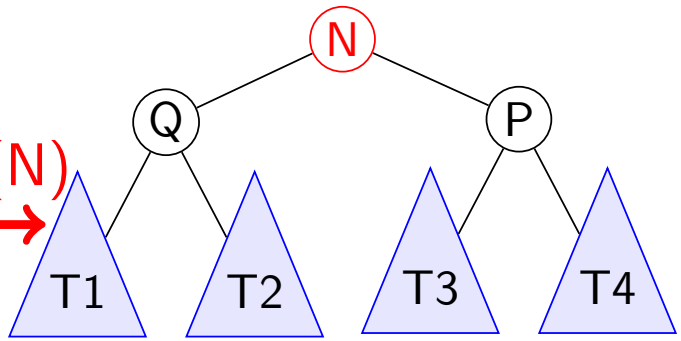


Zig-Zag(N)

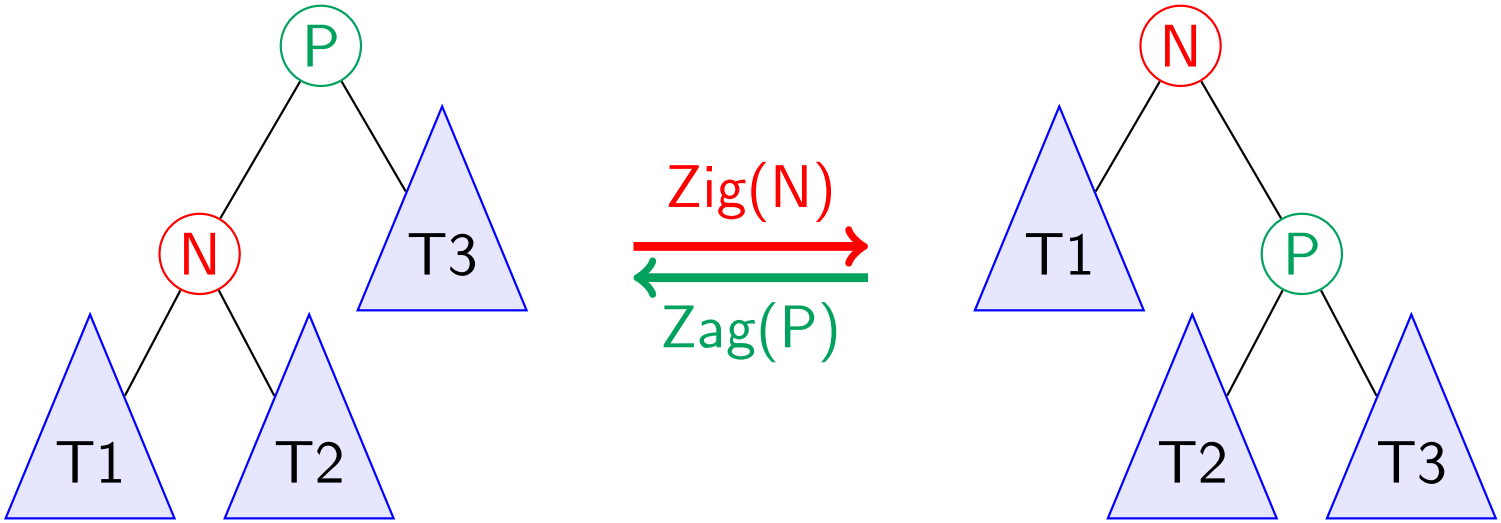




Zag-Zig(N)



Modification: Zig/ Zag

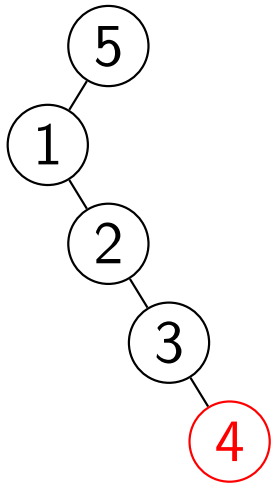


Splay: Pseducode

```
1 Algorithm splay(N)
2 Determine proper case
3 Apply Zig-Zig, Zig-Zag, or Zig as appropriate
4 if N.parent ≠ NULL then
5 |   splay(N)
```

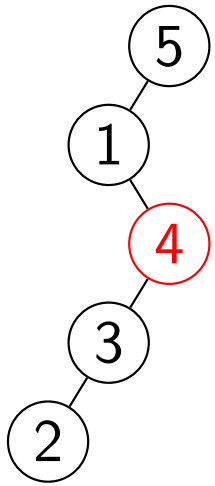

Splay: Problem

Which is the result of splaying the highlighted node?



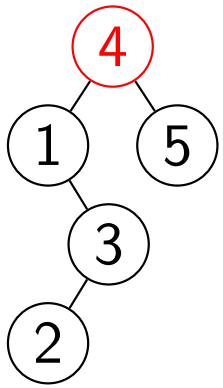
Splay: Problem

Which is the result of splaying the highlighted node?



Splay: Problem

Which is the result of splaying the highlighted node?



Splay Tree: Searching

1 **Algorithm** search(k, R)

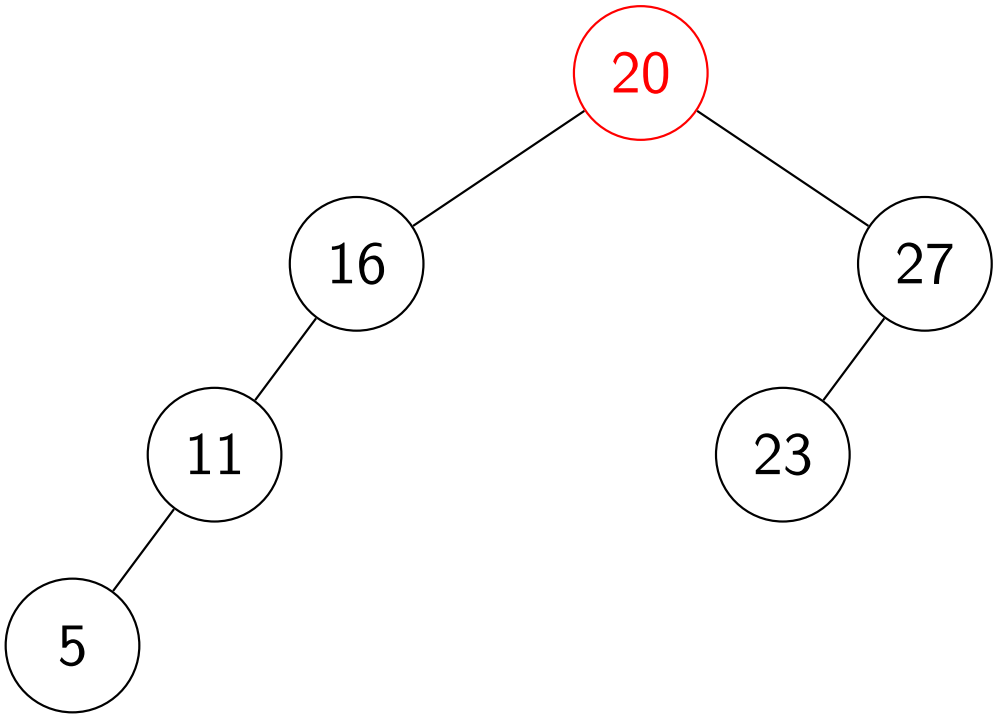
2 $N =$ Find k in the tree R like in BSTs.

3 splay(N)

4 return R

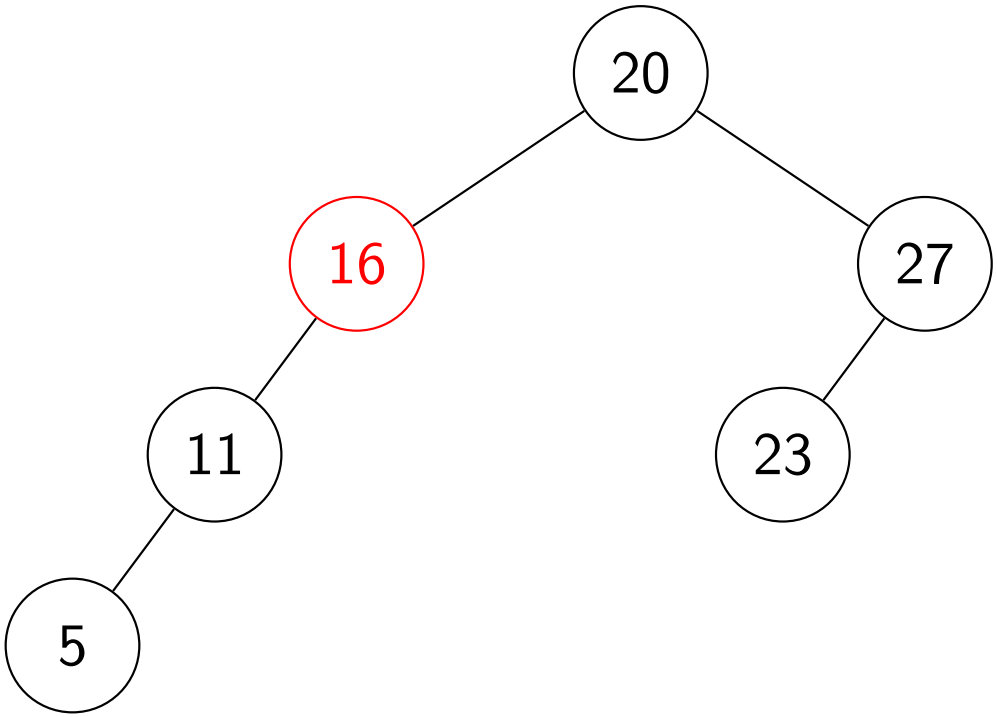
Splay Tree: Searching - Example

Find 5



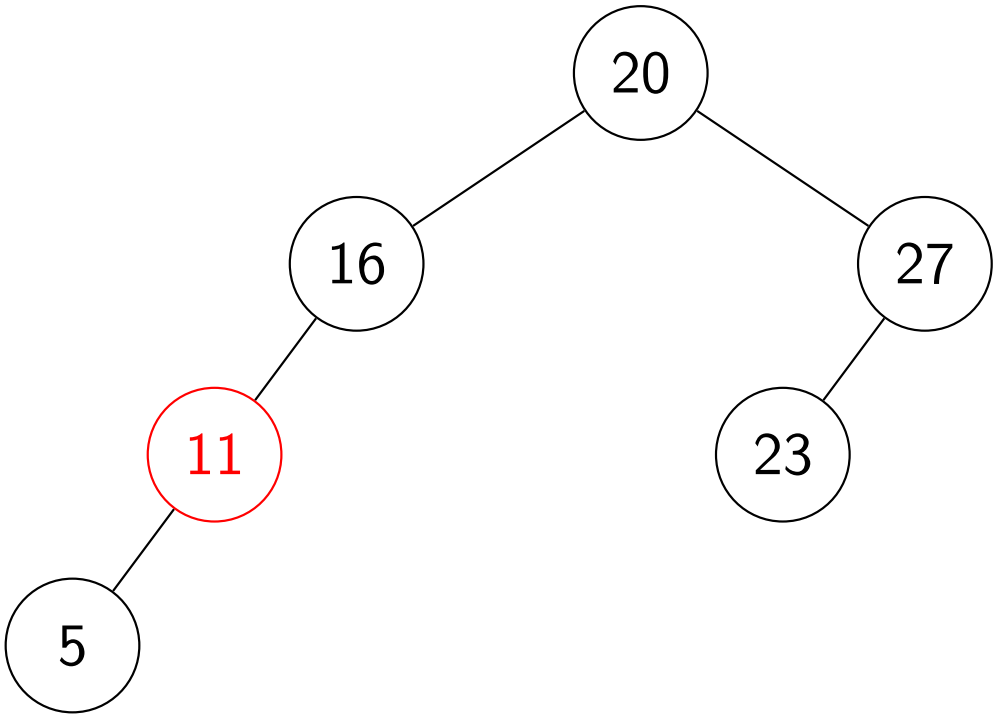
Splay Tree: Searching - Example

Find 5



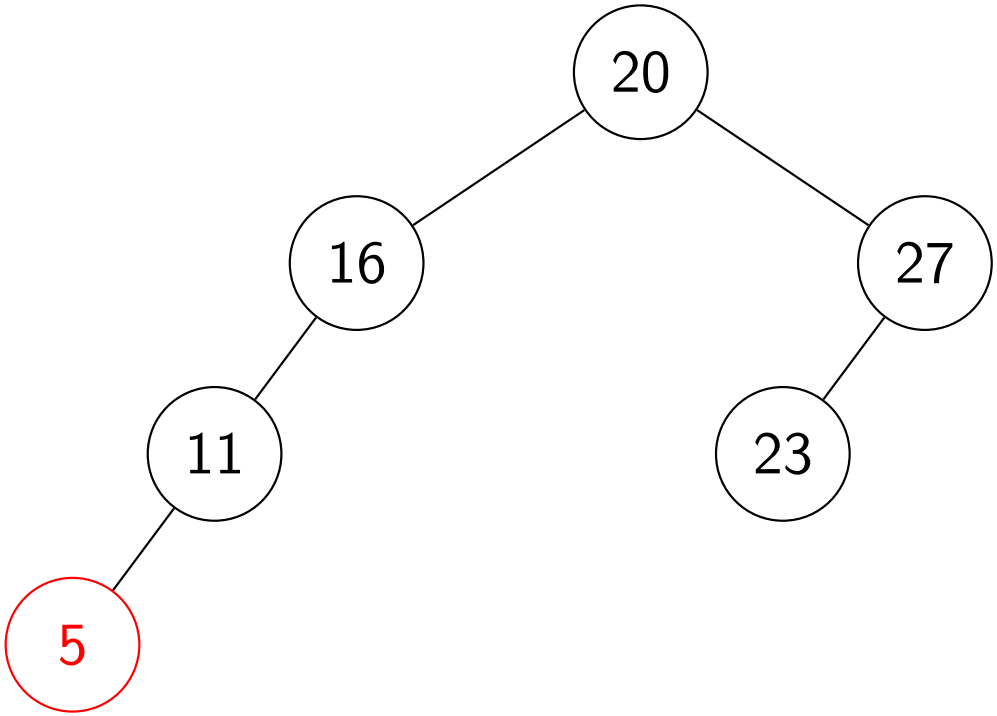
Splay Tree: Searching - Example

Find 5



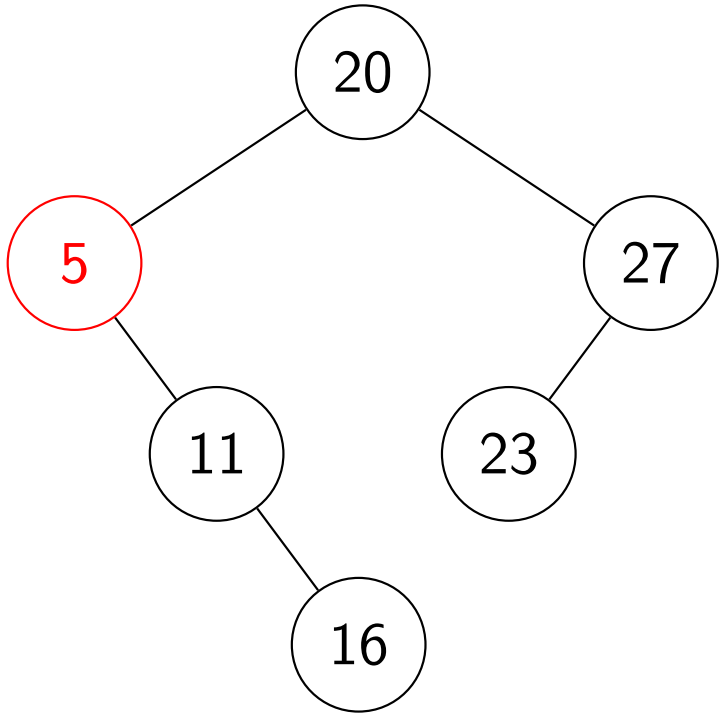
Splay Tree: Searching - Example

Find 5 : **Zig - Zig**



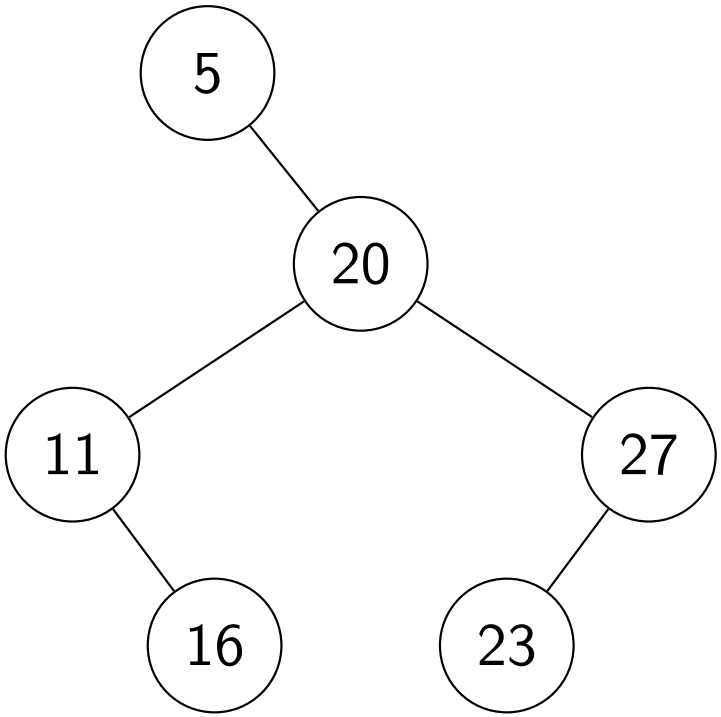
Splay Tree: Searching - Example

Find 5 : **Zig**



Splay Tree: Searching - Example

Find 5



Splay Tree: Insert

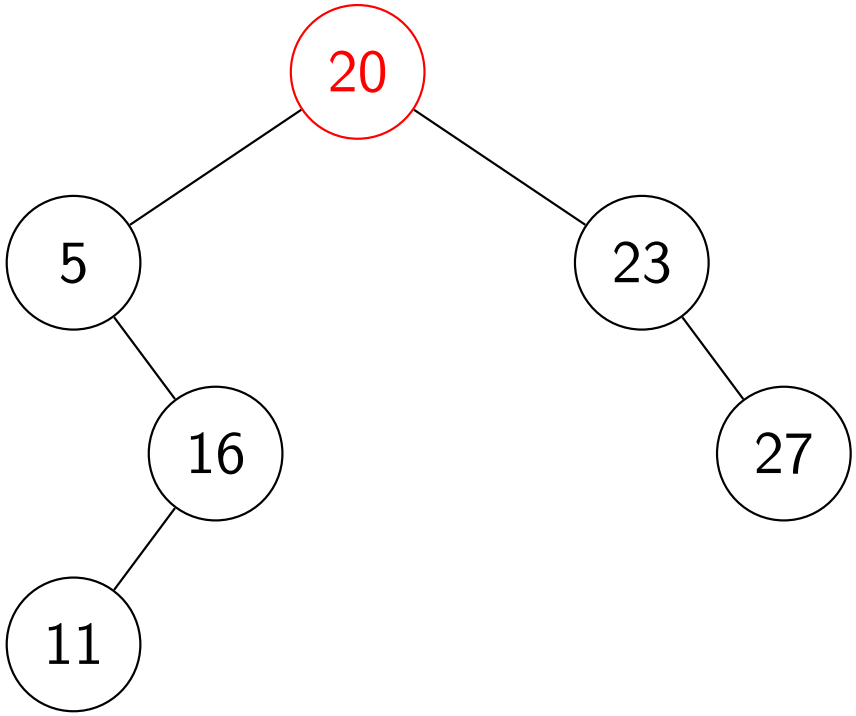
1 **Algorithm** insert(k, R)

2 Insert k into the tree R like in BSTs.

3 find(k, R)

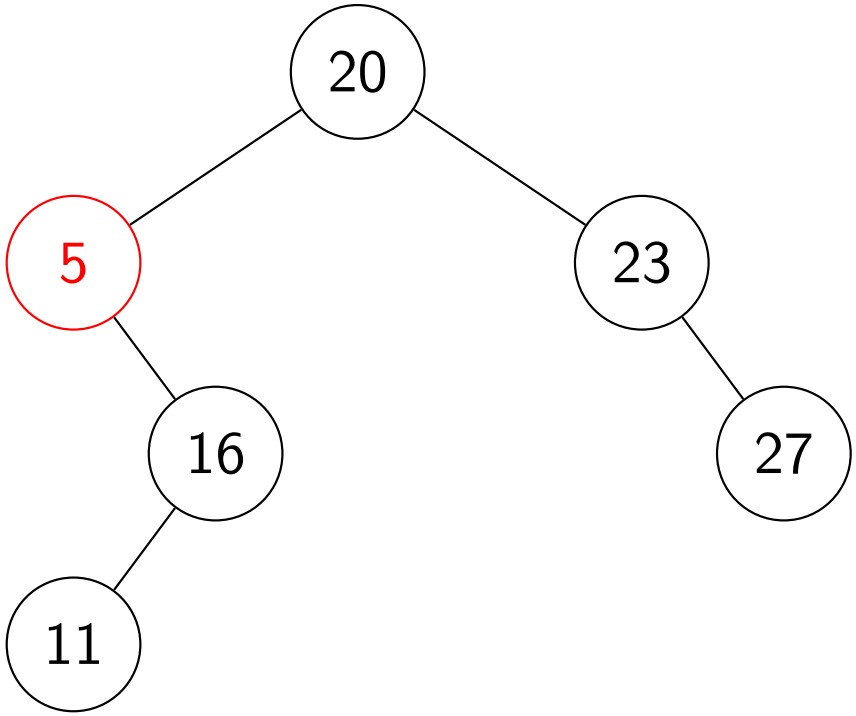
Splay Tree: Insert - Example

Insert 15



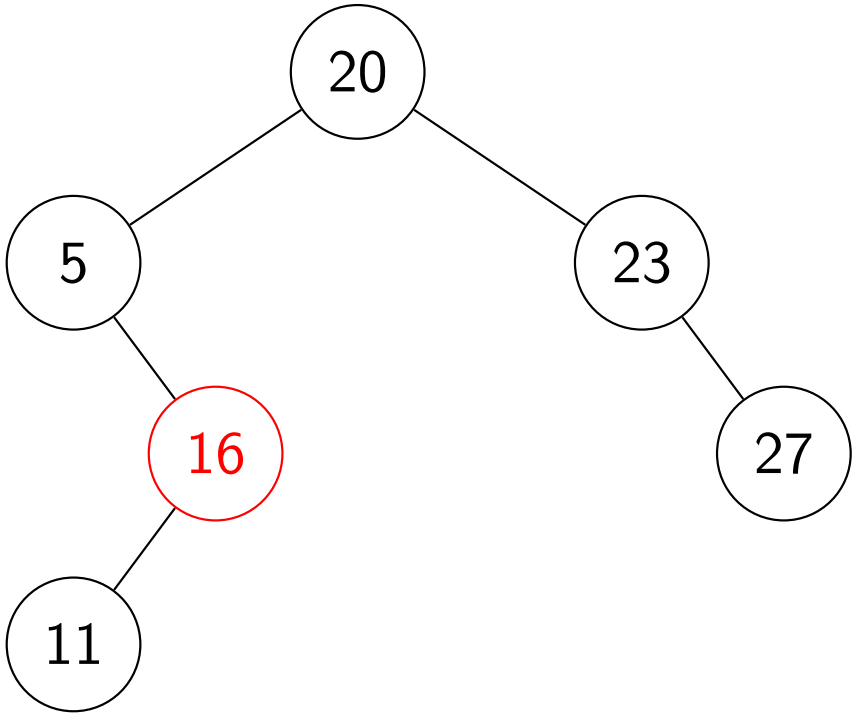
Splay Tree: Insert - Example

Insert 15



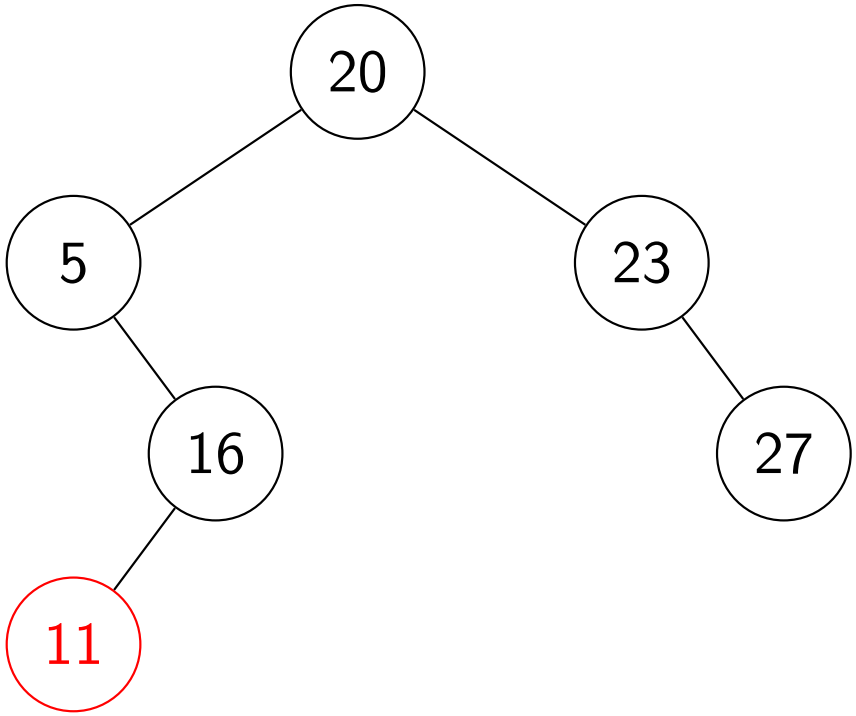
Splay Tree: Insert - Example

Insert 15



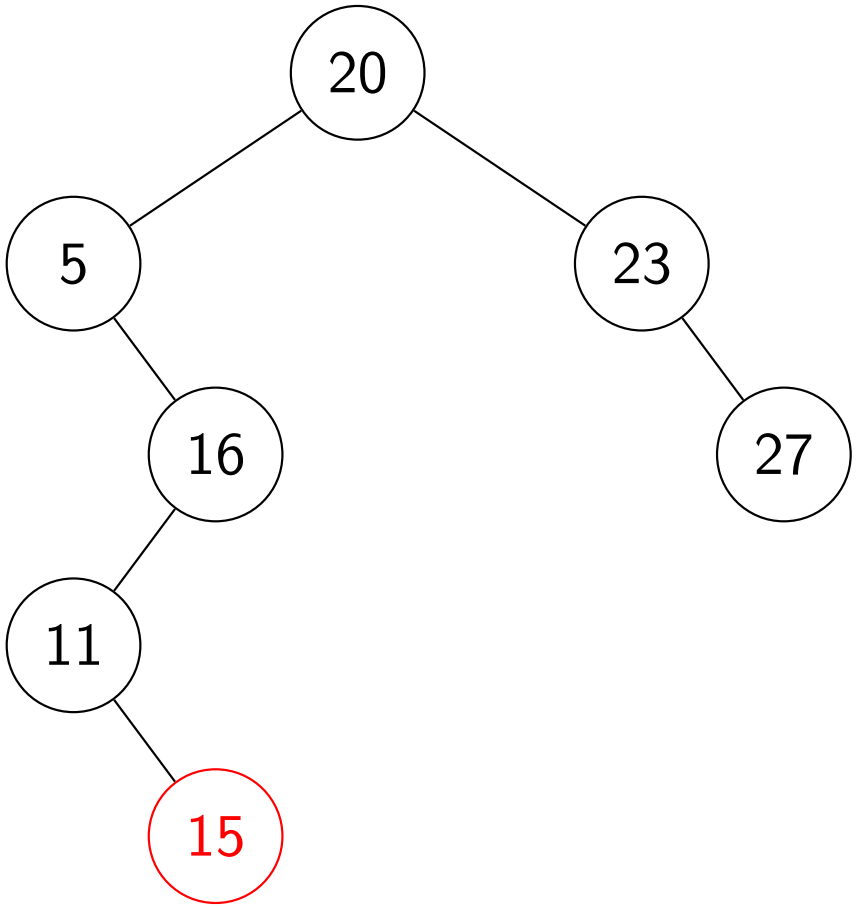
Splay Tree: Insert - Example

Insert 15



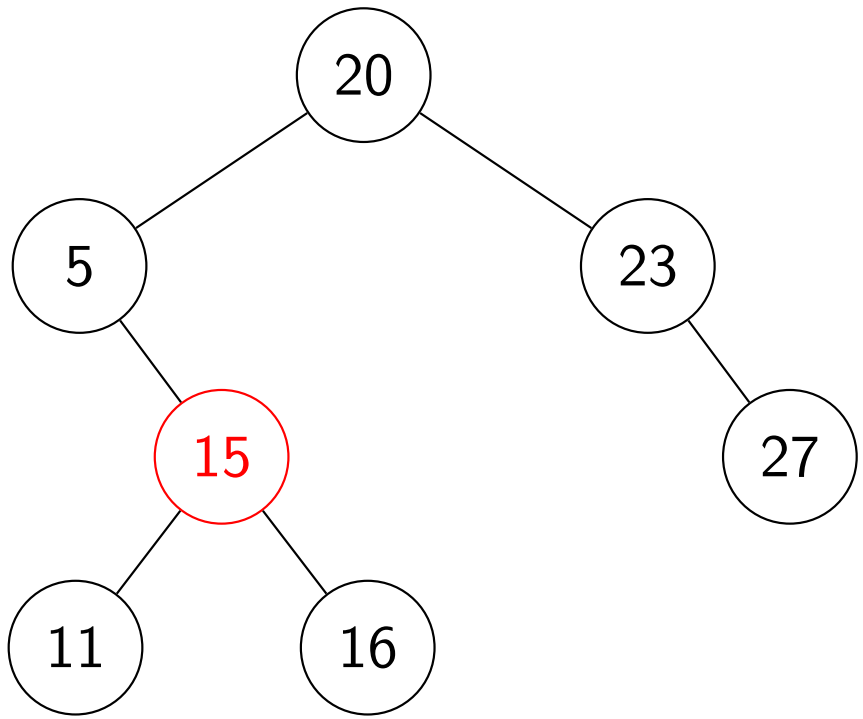
Splay Tree: Insert - Example

Insert 15 : **Zig - Zag**



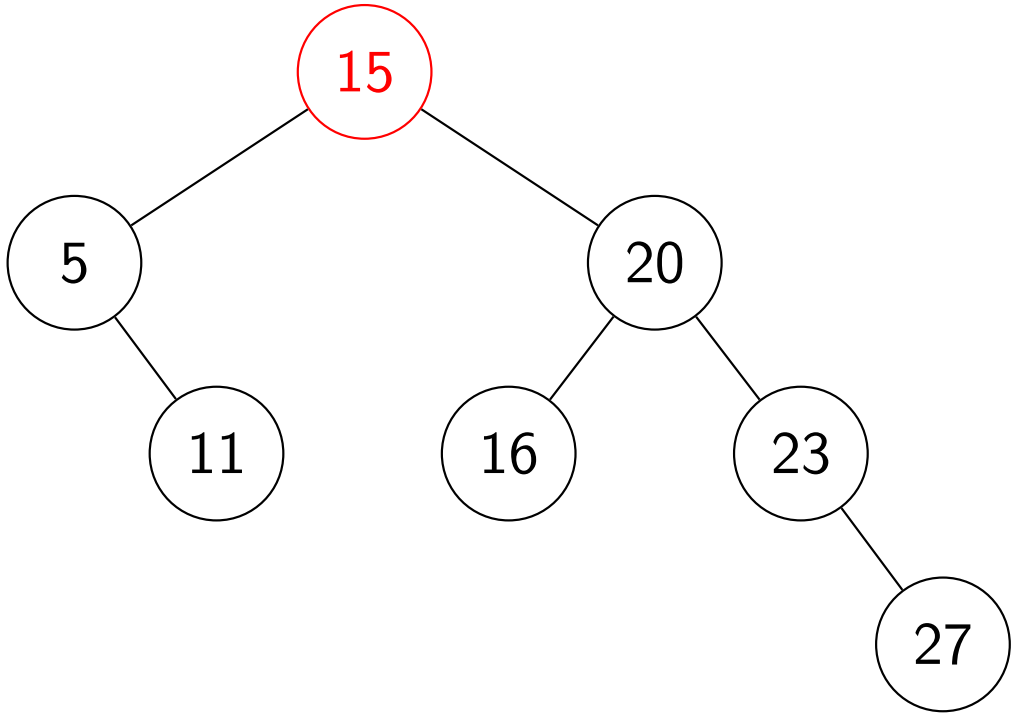
Splay Tree: Insert - Example

Insert 15 : **Zig - Zag**



Splay Tree: Insert - Example

Insert 15

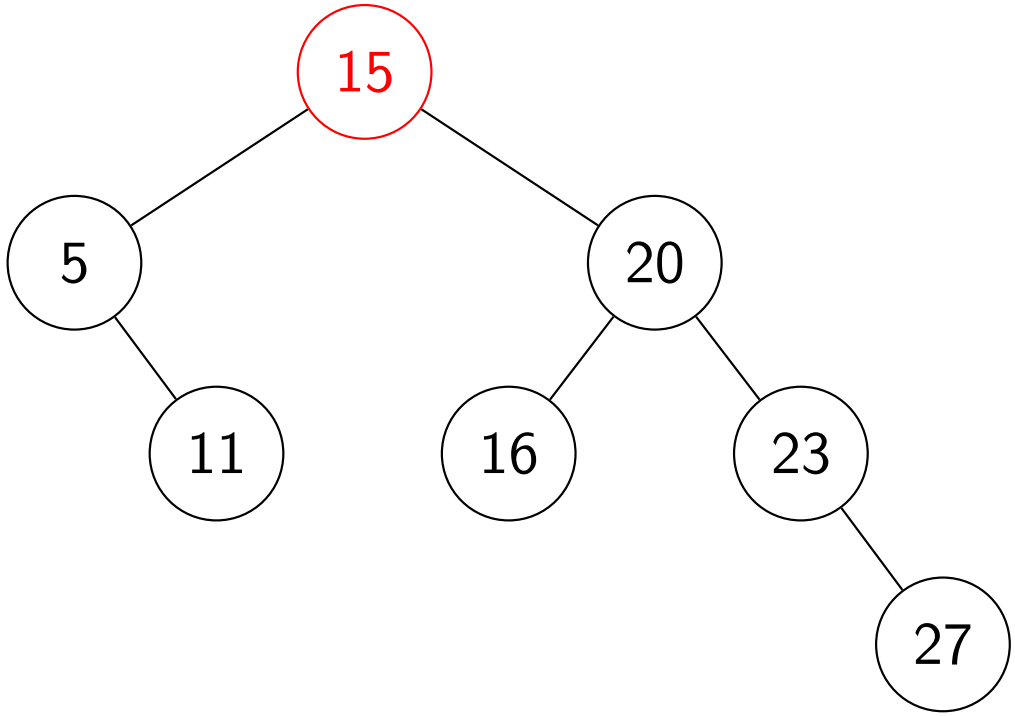


Splay Tree: Deletion

```
1  Algorithm delete(k, R)
2  splay(k, R)
3  N = the largest node in subtree R.left
4  P = R
5  if R.left is empty then
6  |    R = R.right
7  |    recycle(P)
8  |    return
9  splay(N) in subtree R.left
10 R.left.right = R.right
11 R = R.left
12 recycle(P)
```

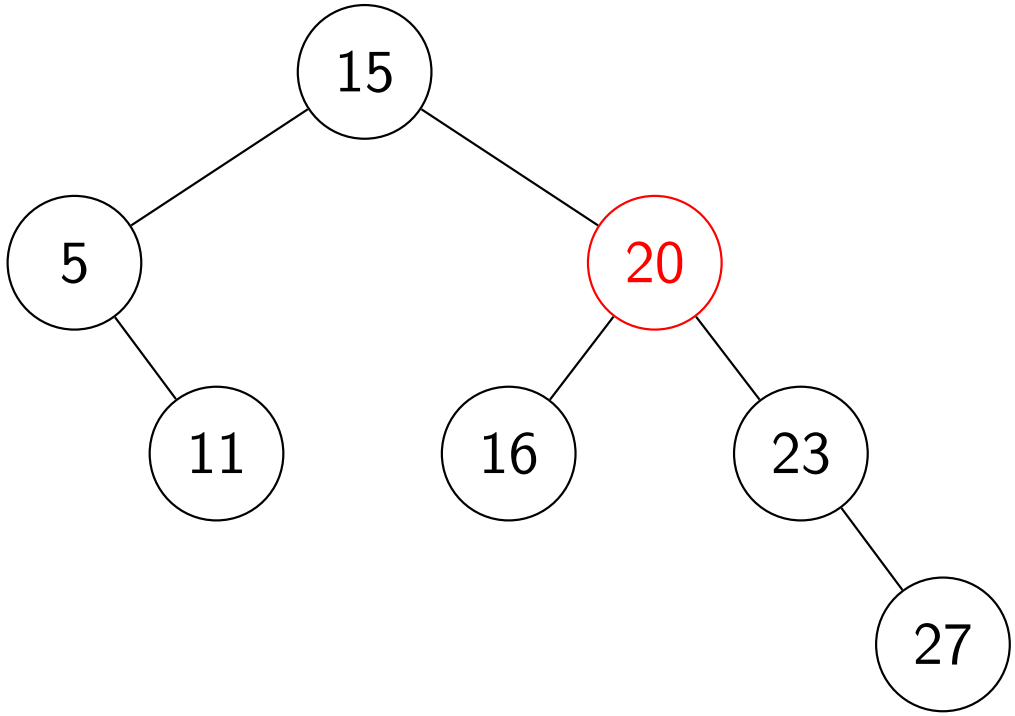

Splay Tree: Deletion - Example

Delete 16



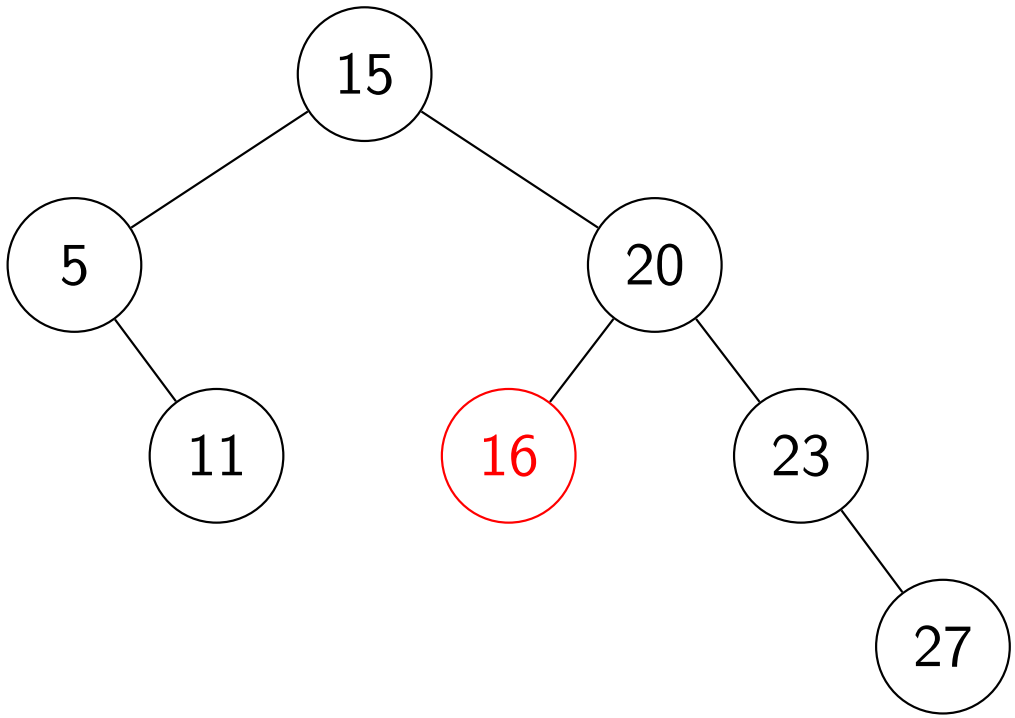
Splay Tree: Deletion - Example

Delete 16



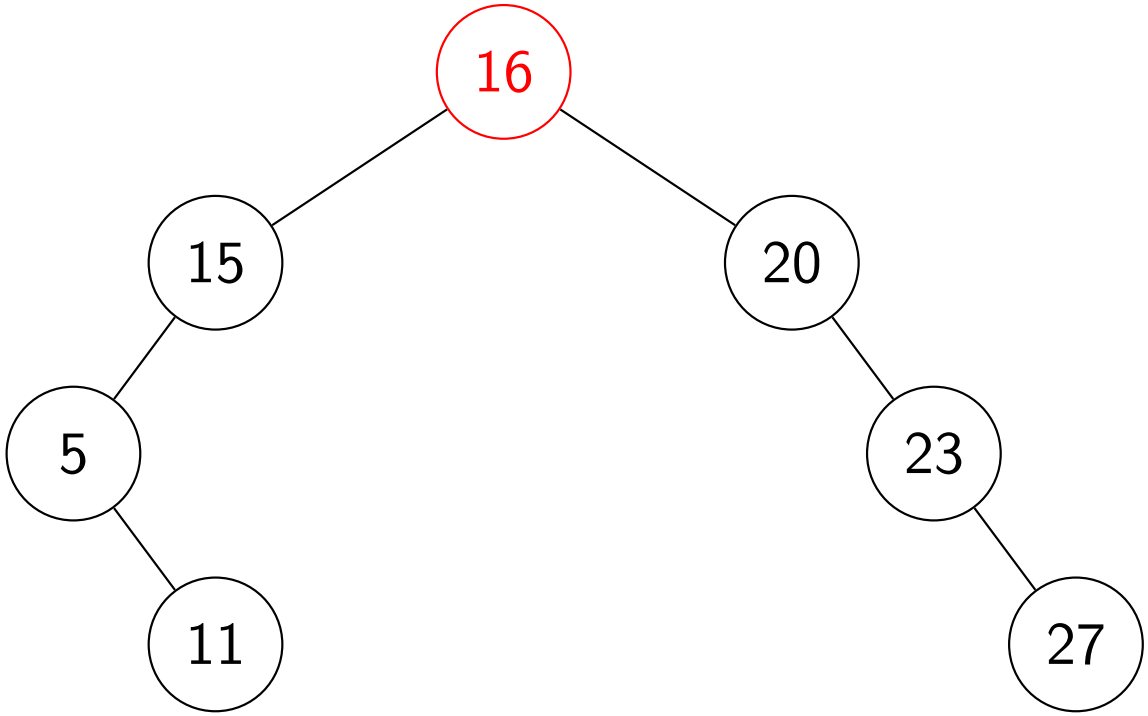
Splay Tree: Deletion - Example

Delete 16 : **Zag - Zig**



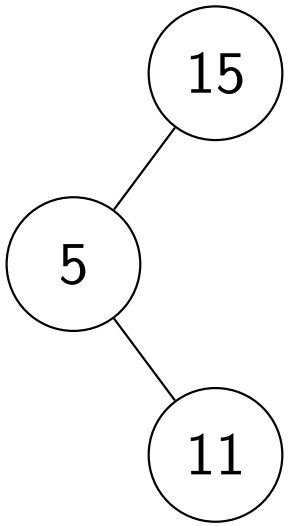
Splay Tree: Deletion - Example

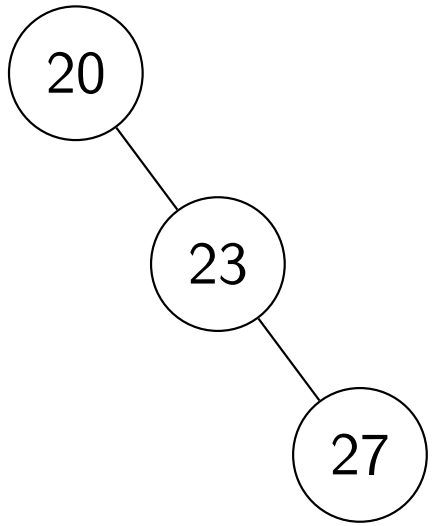
Delete 16



Splay Tree: Deletion - Example

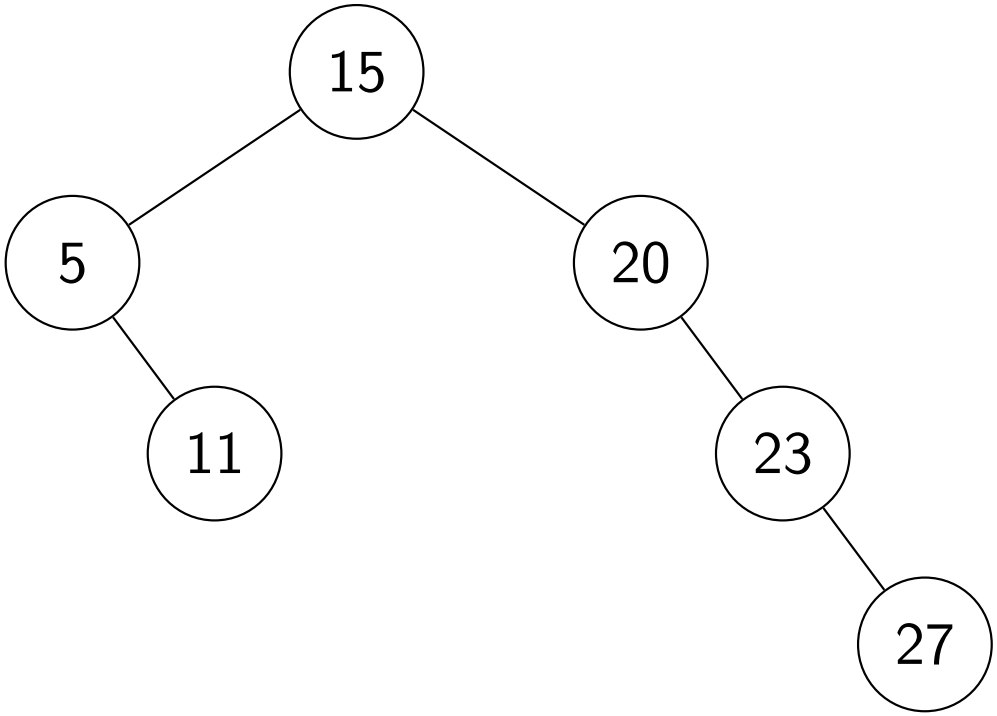
Delete 16





Splay Tree: Deletion - Example

Delete 16 : **Done.**



THANK YOU.