the right majors, minors &
ucation?
for students' academic and career success
ging for many students - Many students change them
luring college!
e prediction of student success in MMC could
dual students
d their right MMC
chieve their academic goals

Advanced Binary Search Trees

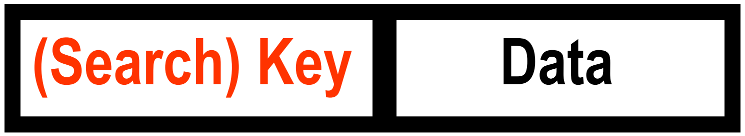
AVL Trees & Splay Trees

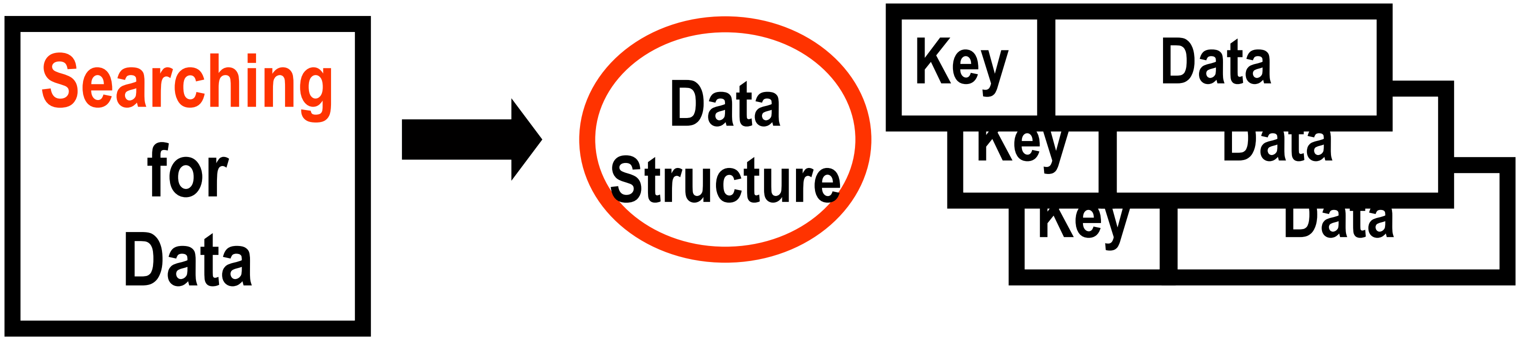
Prof. Young Park

Binary Search Trees (BST)

The Searching Problem

- Fundamental to a variety of computer problems!





A Search Tree?

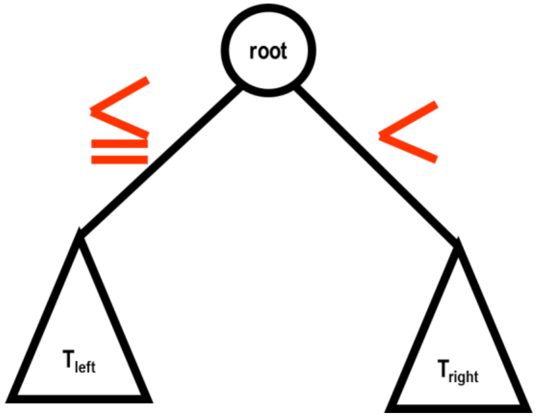
- A tree that maintains its data some sorted order and supports efficient search operations.
  - By constraining the relative positions of the nodes in the tree!
  - **All data items are kept in sorted order!**

A Binary Search Tree?

- **A binary tree + A search tree**
- **An ordered (sorted) binary tree**
- A special kind of binary tree with the ordering condition
  - Between every node and the nodes in its left subtree.
  - Between every node and the nodes in its right subtree.
  - BST Order Property!

The Order Condition of BST

- BST order property - For any node N
  - The key value in every node in N's left subtree is less than or equal to the key value K in N.
  - The key value in every node in N's right subtree is greater than the key value K in N.



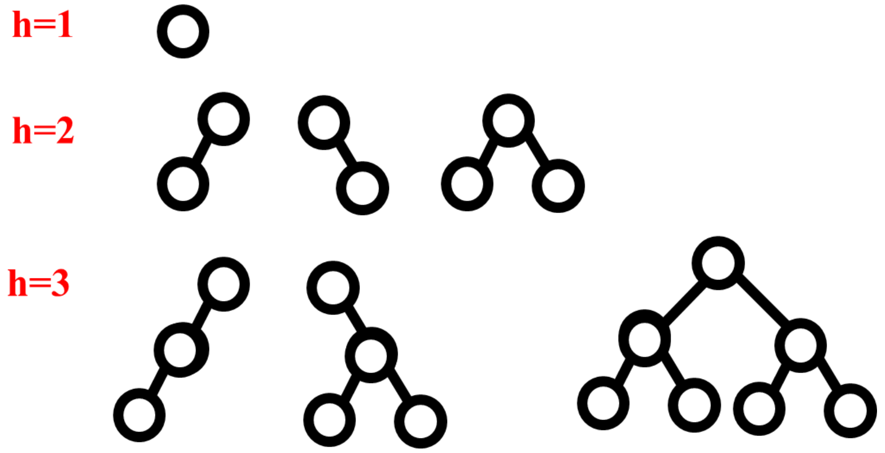
Properties of Binary Search Trees

- What is the minimum number of nodes that a BST of height h can have?

  ➔ **h**

- What is the maximum number of nodes that a BST of height h can have?

  ➔ **$2^h - 1$**



Properties of Binary Search Trees

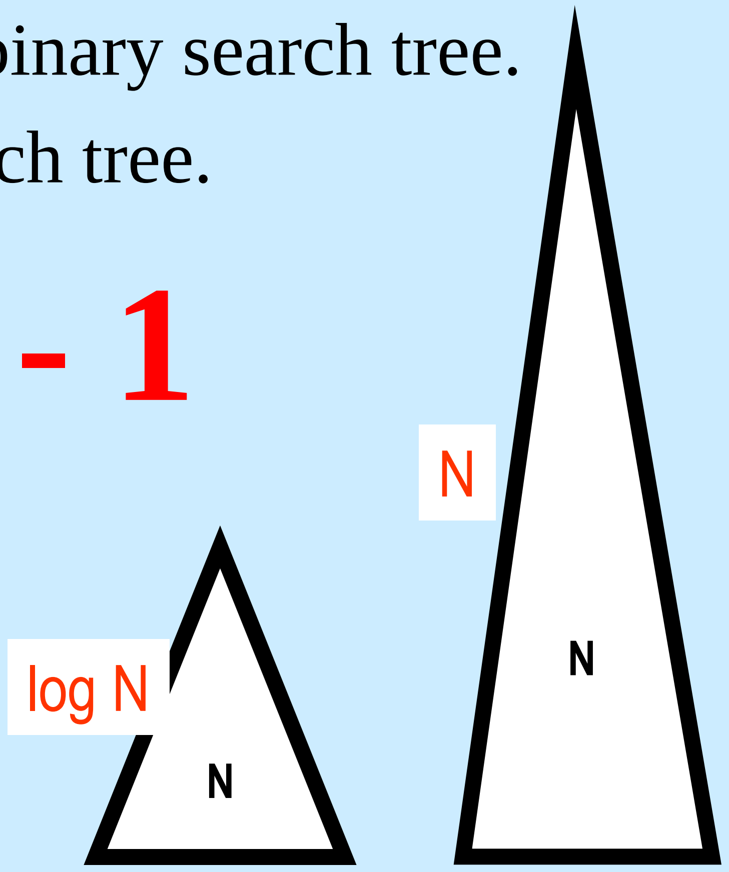
- N= The number of nodes in a binary search tree.
- h = The height of a binary search tree.

➔ $h \leq N \leq 2^h - 1$

➔ $\log N \leq \log (N+1) \leq h \leq N$

➔ Lower Bound: $h = \Omega(\log N)$

➔ Upper Bound: $h = O(N)$



BST Representation

- Pointer-based representation
- Array-based representation

BST Operations

- Insert
- Search/Retrieve
- Delete
- …

Insert Operation

- **Insert (BST, newitem)**
  - If BST == NULL (empty tree) then
    - Create a new node; let BST point to this new node;copy newitem into new node's data portion; set the pointers in the new node to NULL.
  - else if newitem.Key < BST->Key then
    - Insert (BST->LchildPtr, newitem)
  - else
    - Insert (BST->RchildPtr, newitem)
- Worst case
  - O(N)
- Average Case
  - **O(log N)**

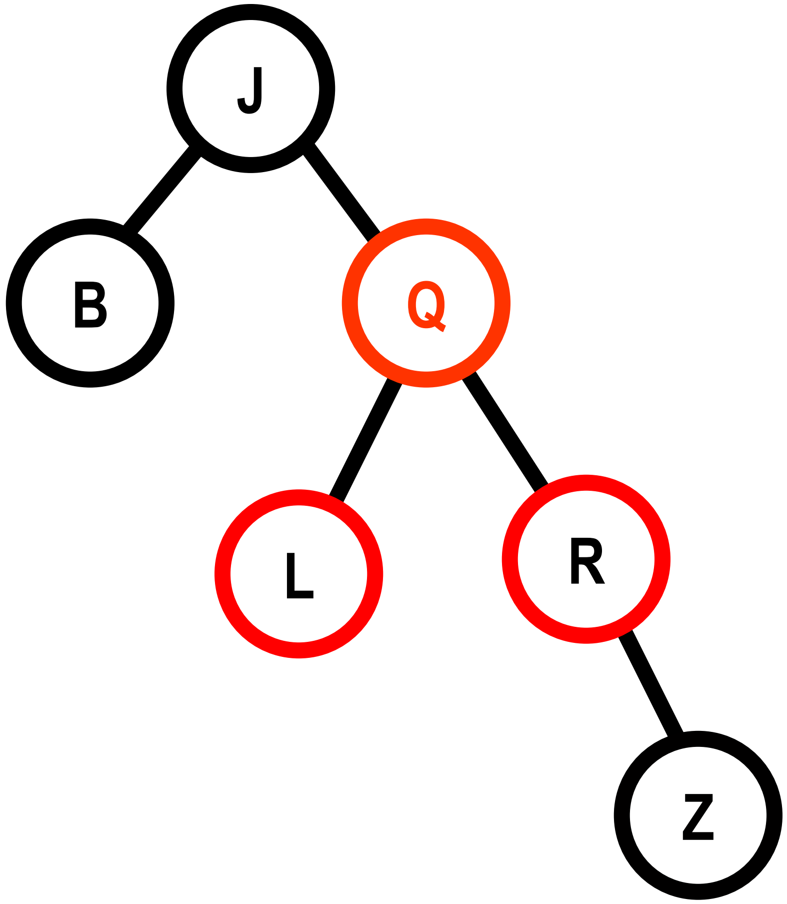
Search/Retrieve Operation

- **Search(BST, SearchKey):**
  - If BST == NULL (empty tree) then
    - Not Found (Unsuccessful search)
  - else if SearchKey == BST->Key then
    - Found (Successful search)
  - else if SearchKey < BST->Key then
    - Search (BST->LchildPtr, SearchKey)
  - else
    - Search (BST->RchildPtr, SearchKey)
- Worst case
  - O(N)
- Average Case
  - **O(log N)**

Delete Operation

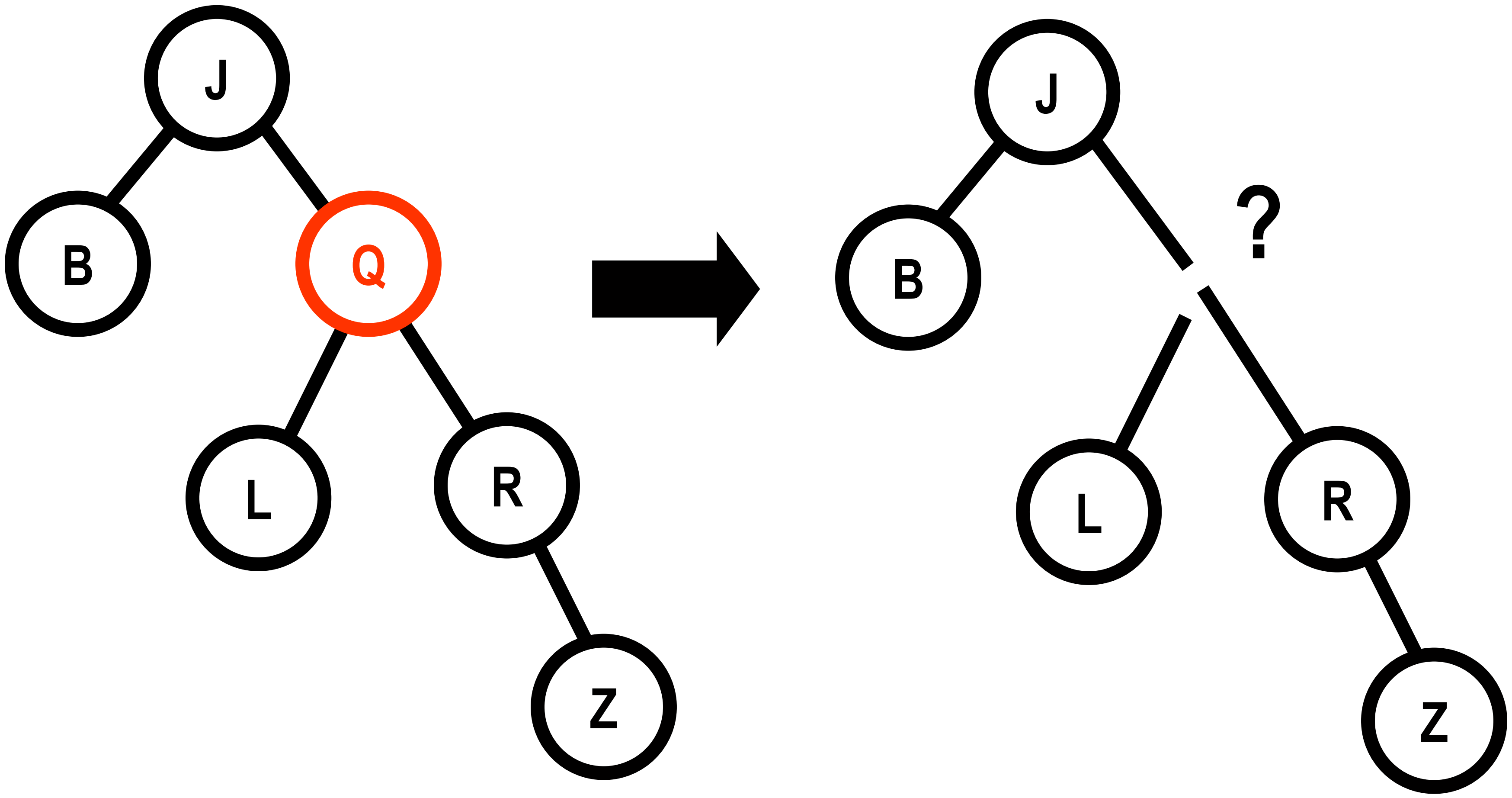
Three cases:

1. Node has no children (leaf node)
2. Node has one child
3. Node has two children



Delete Operation

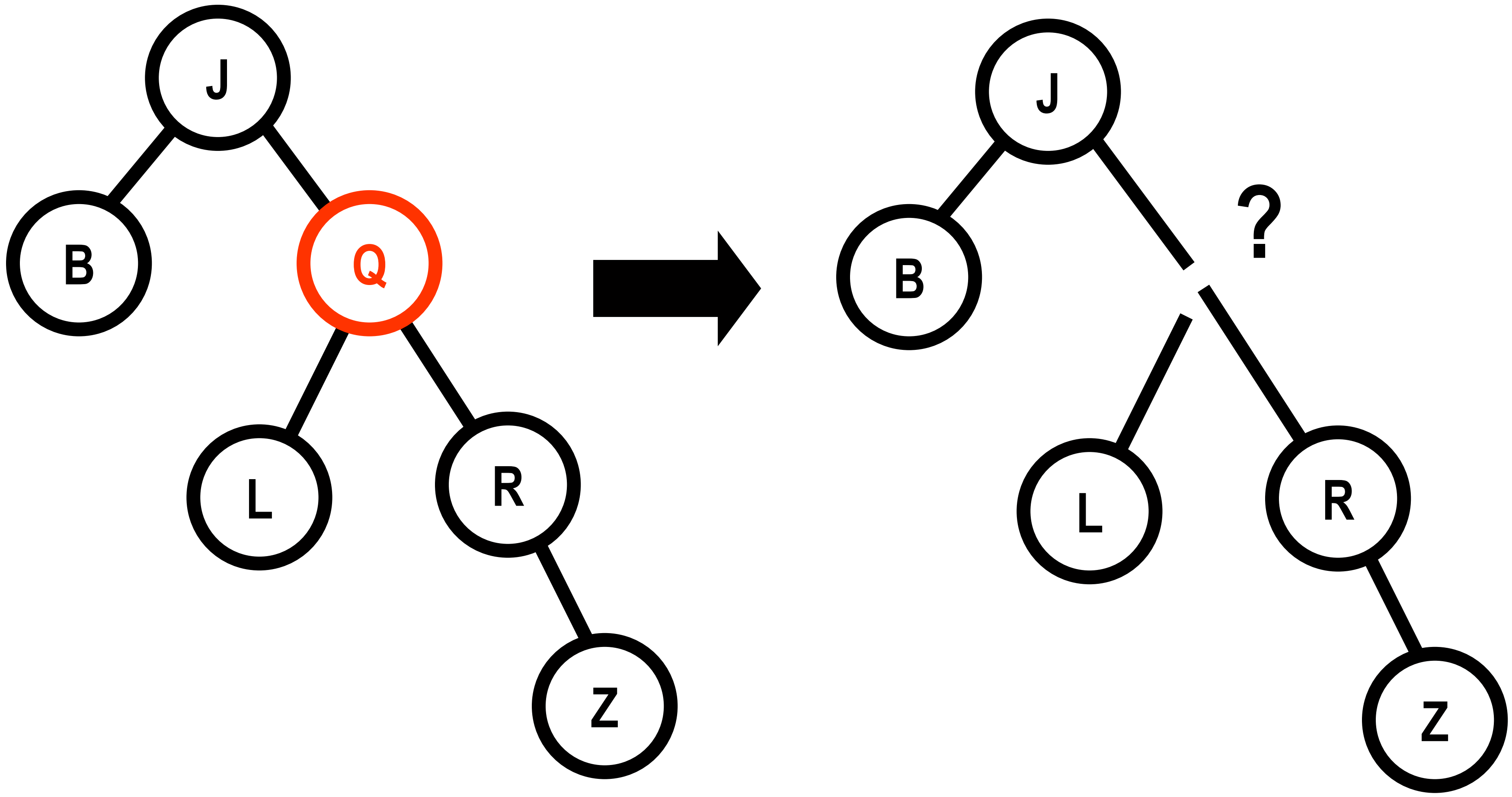
3. Node has two children



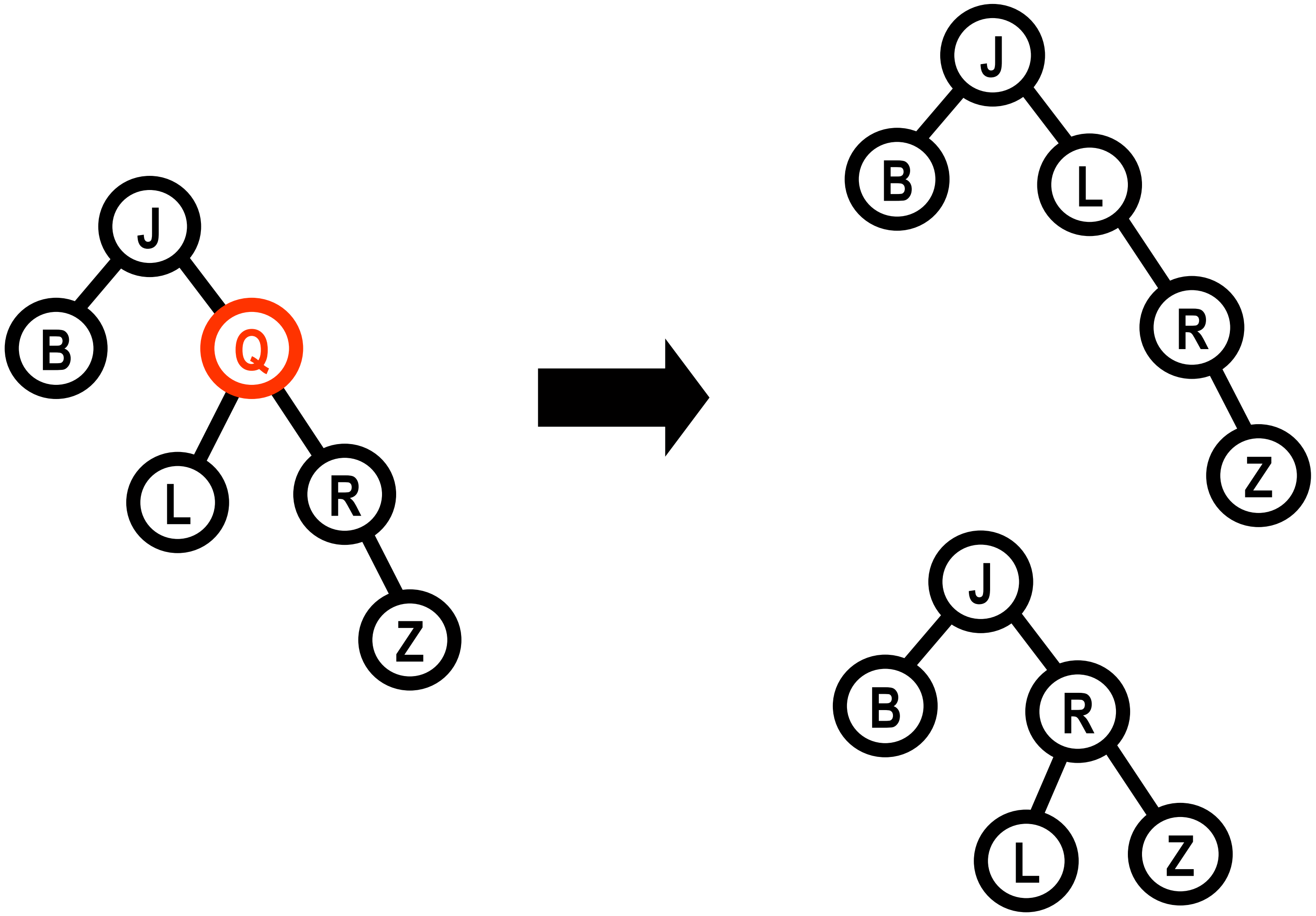
Delete Operation

- **Delete by Merging**

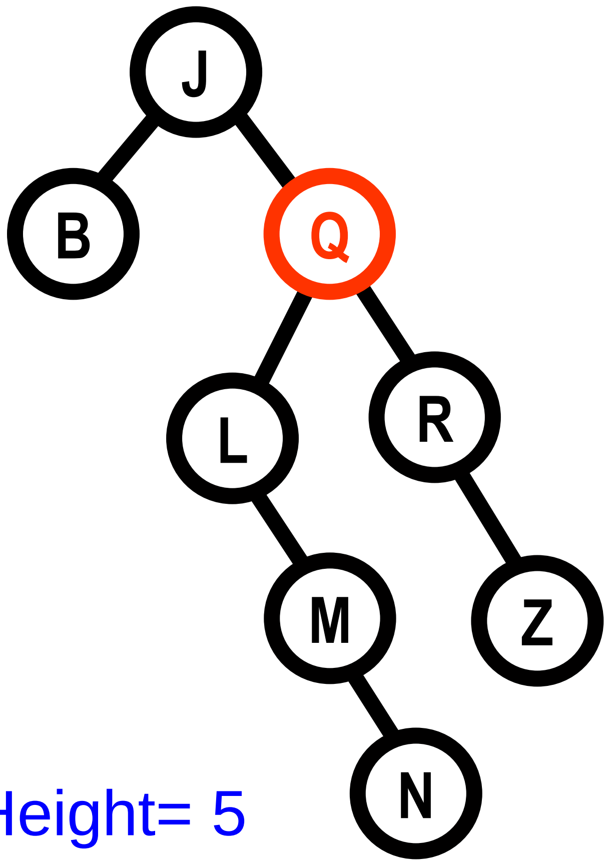
Example: Delete Q by Merging



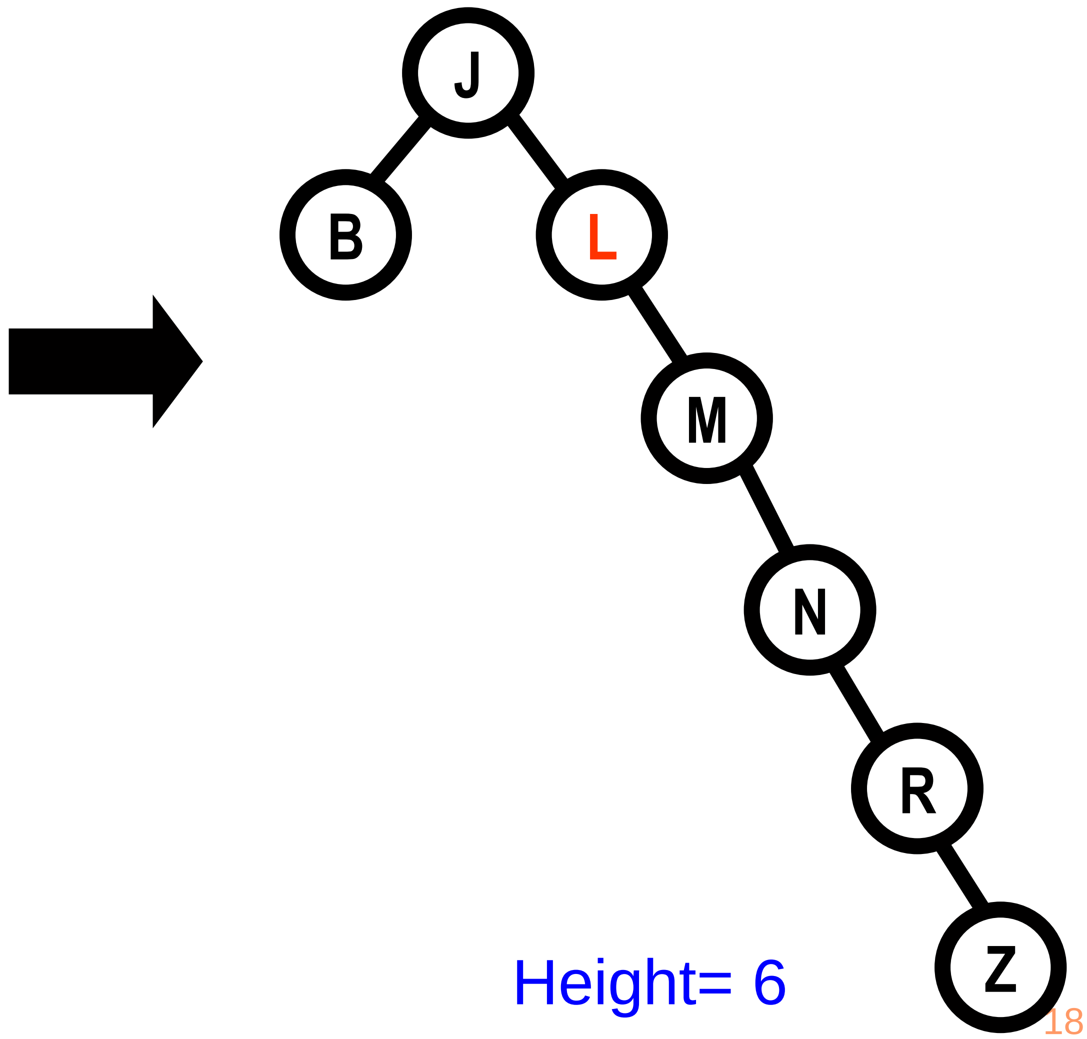
Example: Delete Q by Merging



►QUIZ? Delete Q by Merging



Height= 5



Height= 6

Delete By Merging

- Delete by Merging **can increase the height of BST** even though we delete a data item!

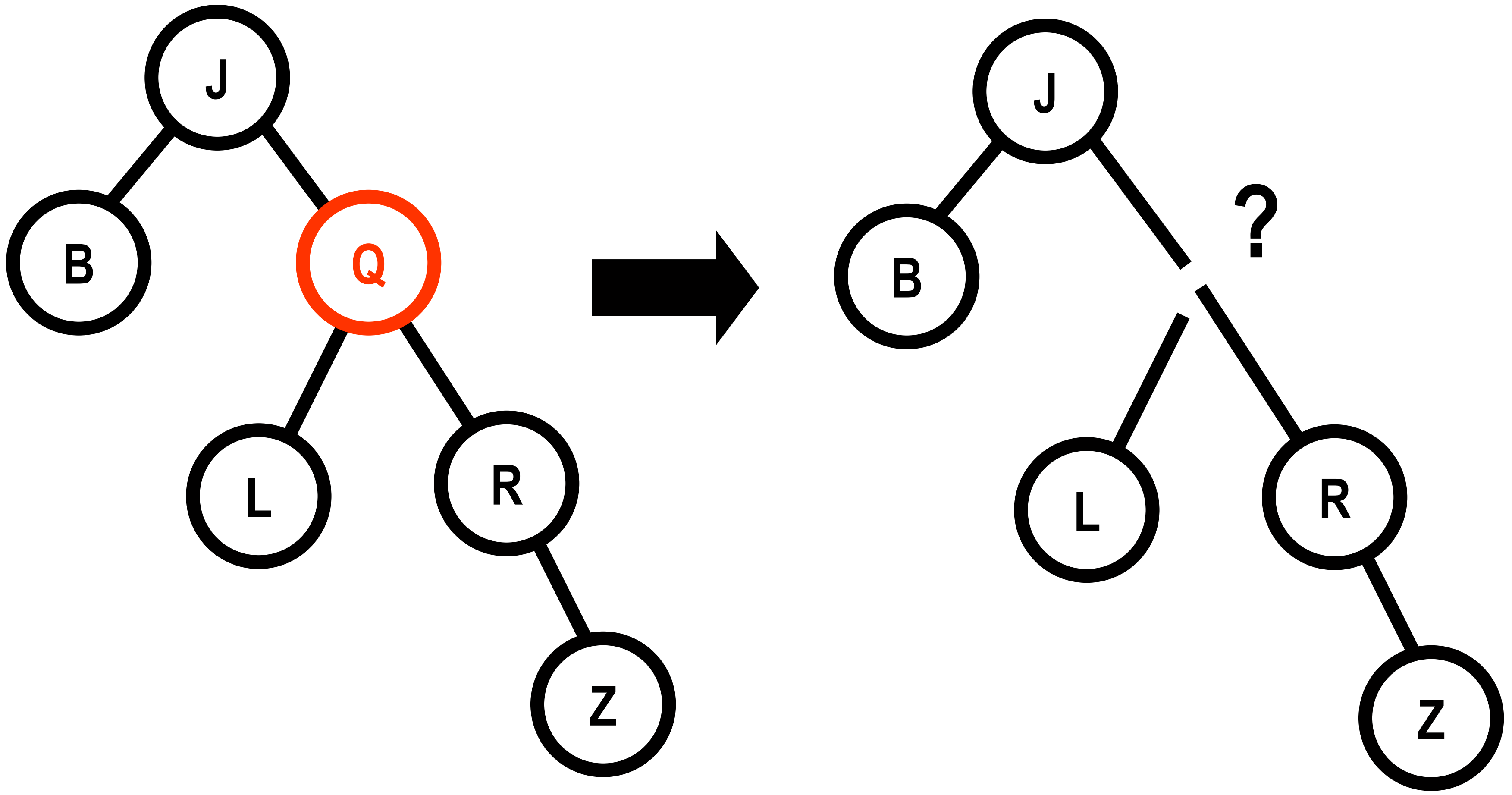
Delete Operation

- **Delete by Copying**
  - By copying IOP (In-Order Predecessor)
  - By copying IOS (In-Order Successor)

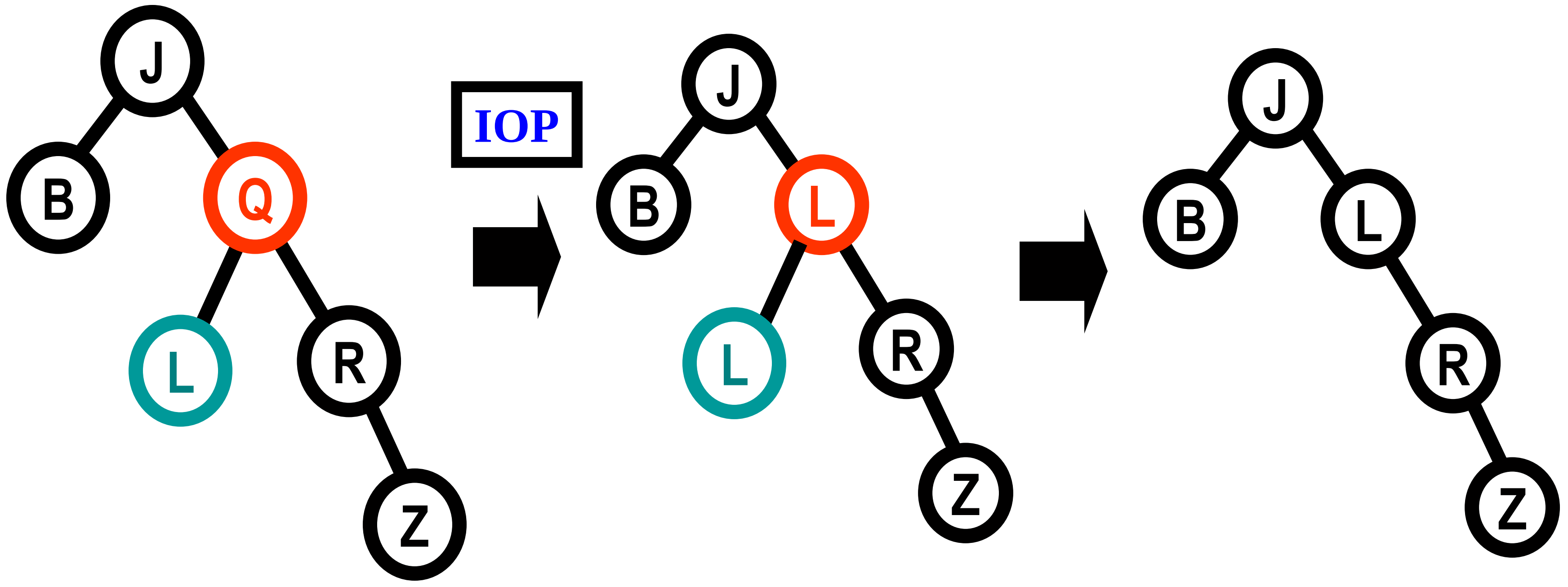
  - Case 3 => Case 1 or Case 2

“Transform-and-Conquer”

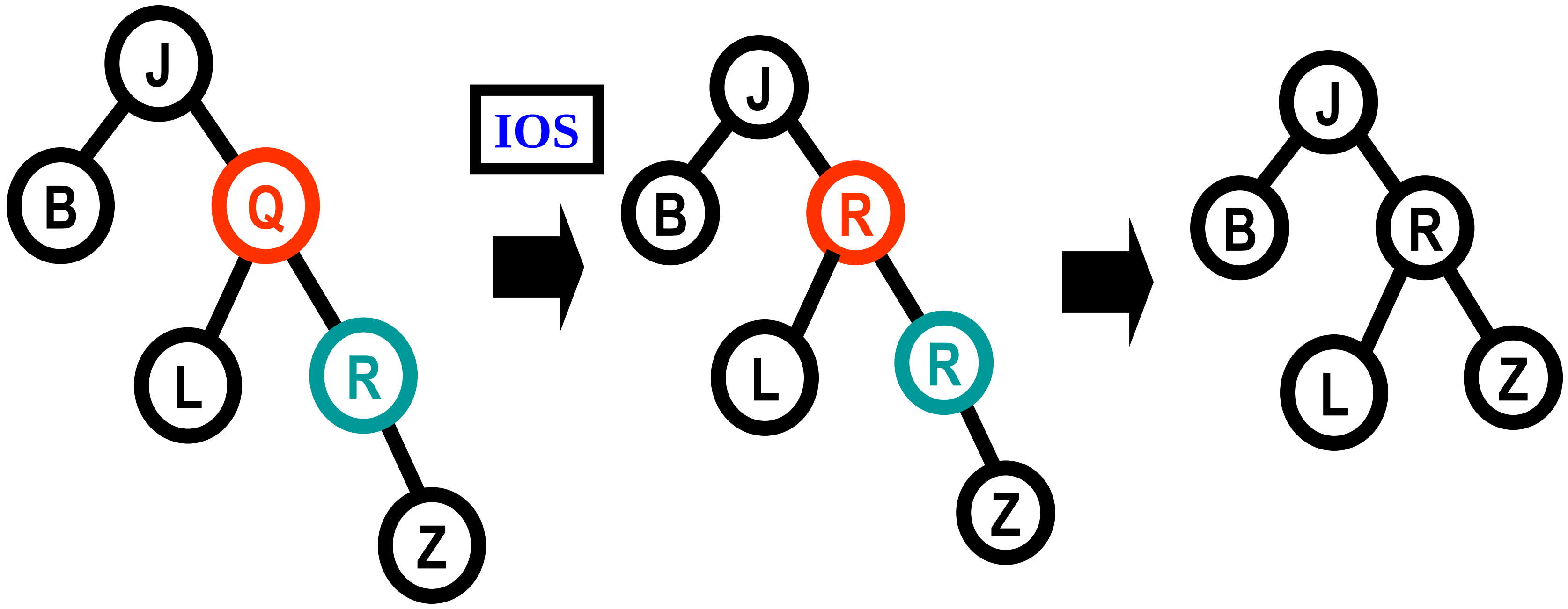
Example: Delete Q



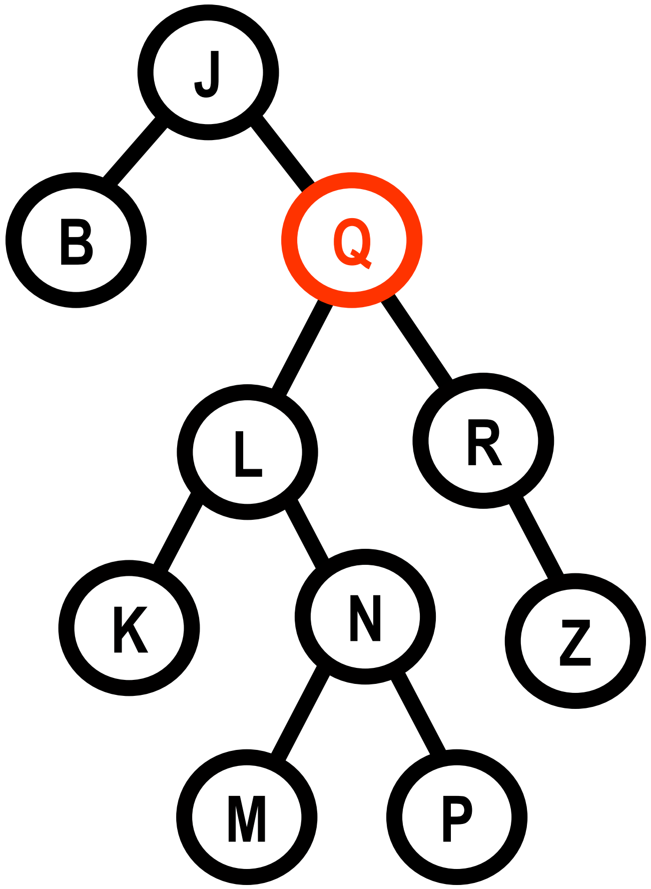
Example: Delete Q



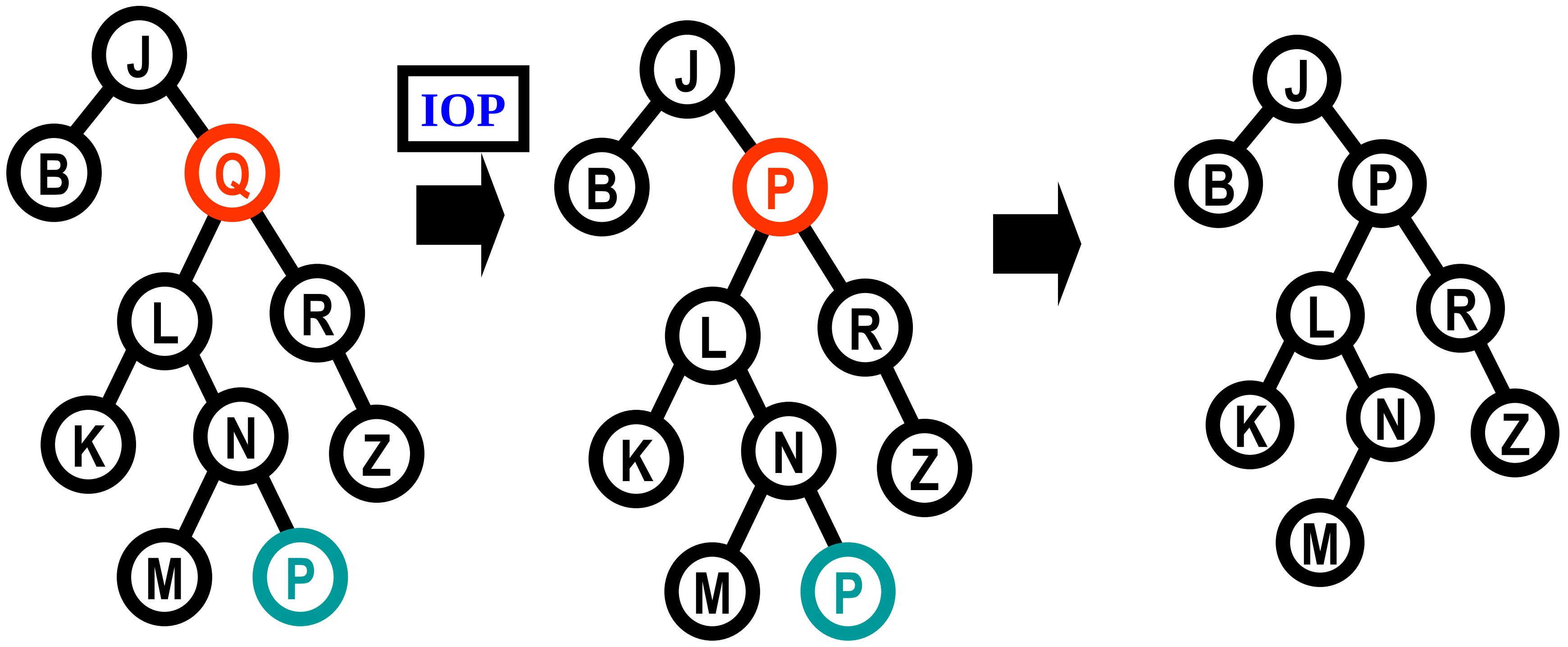
Example: Delete Q



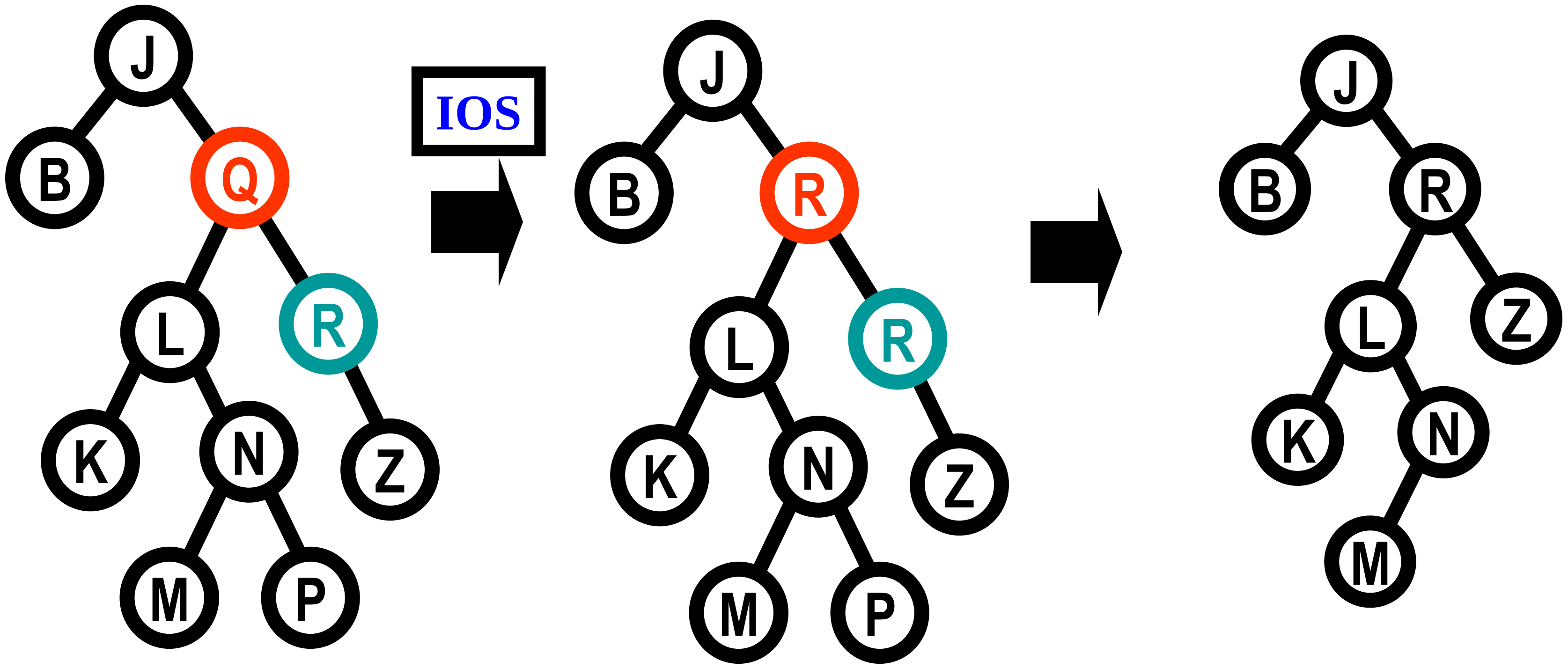
►QUIZ? Delete Q ?



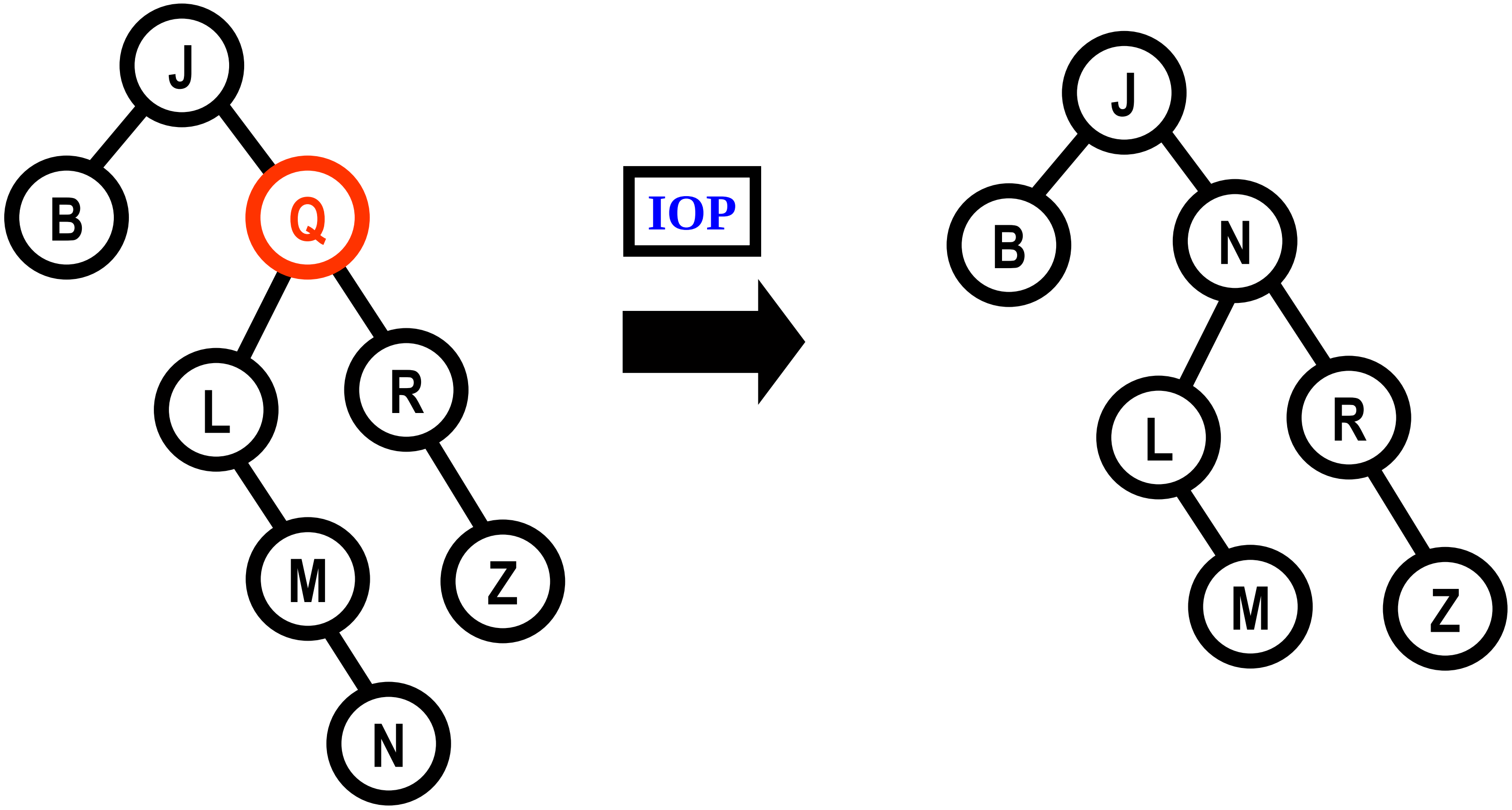
►QUIZ: Delete Q



►QUIZ: Delete Q



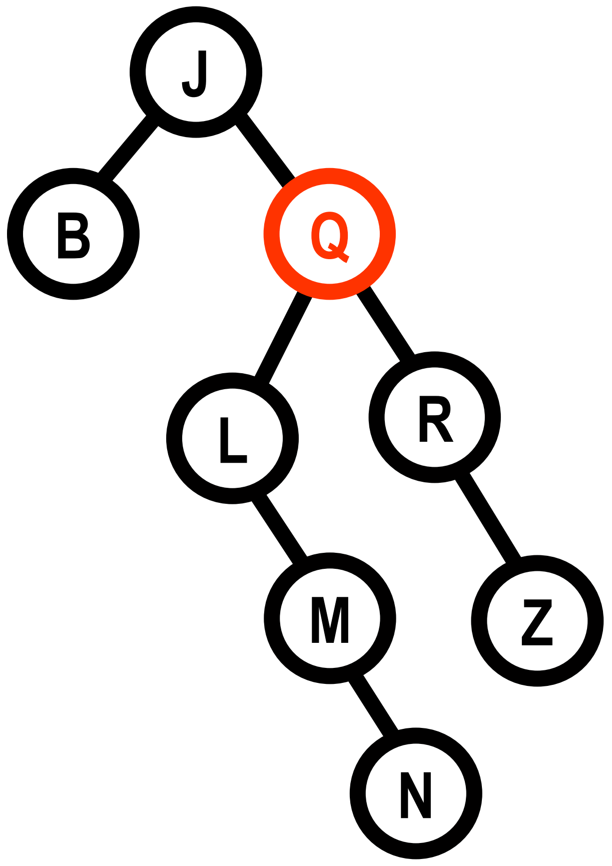
►QUIZ? Delete Q by Copying



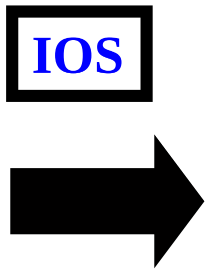
Height= 5

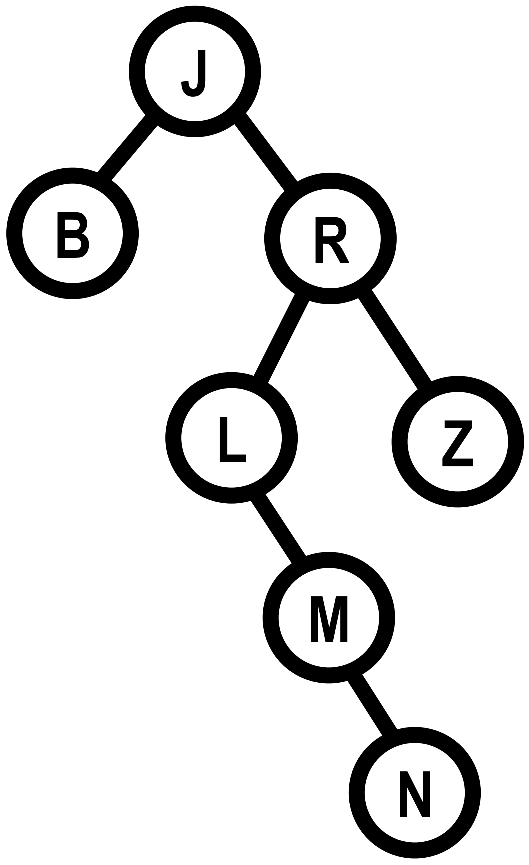
Height= 4

►QUIZ? Delete Q by Copying



Height= 5





Height= 5

Delete By Copying

- Delete by Copying **never increase the height of BST** while we delete a data item!

Delete Operation

- **Delete(BST, SearchKey):**
  - If SearchKey < BST->Key then
    - Delete (BST->LchildPtr, SearchKey)
  - else if SearchKey > BST->Key then
    - Delete (BST->RchildPtr, SearchKey)
  - else (SearchKey == BST->Key)
    - DeleteNode (BST)
      - Three cases:
        1. Node has no children (leaf node)
        2. Node has one child
        3. Node has two children - **Delete by Copying IOP or IOS!**
- Worst case
  - O(N)
- Average Case
  - **O(log N)**

BST Implementation

```
template<class DataType>
class BSTnode
{
public:
   BSTnode();
   BSTnode(DataType D, BSTnode<DataType>* l,
                                    BSTnode<DataType>* r)
          : data(D), LchildPtr(l), RchildPtr(r) { }

private:
   DataType data;
   BSTnode<DataType>* LchildPtr;
   BSTnode<DataType>* RchildPtr;
};
```

BST Implementation

```
template<class DataType>
class BST
{
public:
   BST();
   // search;
   // insert;
   // delete;
   …

private:
BSTnode<DataType>* rootBT;
…

};
```

Binary Search Tree Visualization

- ***Binary Search Tree Visualization***

✓

►QUIZ?

- Compare **Delete-by-copying** with **Delete-by-merging** in BST?

►QUIZ?

- Compare **BST** with **Randomized Skip List?**
  - Insert
  - Search
  - Delete

Advanced Binary Search Trees

Why Advanced Binary Search Trees?

- Why?
  - The **search, insert, delete** operations on **BST**:
    - O(n) time at worst-case
  - **A better search, insert, delete operation?**
    - **O(log N) worst & amortized time** rather than O(N)?

What Advanced Binary Search Trees?

- O(log N) ***worst time*** **for search, insert, delete operations?**
  - ➔ **AVL (search) Trees**
  - ➔ **Red-Black (search) Trees**

- O(log N) ***amortized time*** **for search, insert, delete operations?**
  - ➔ **Self-Adjusting Binary Search Trees!**
  - ➔ **Splay (search) Trees**

What Advanced M-way Search Trees?

- **O(log N) *worst time* for search, insert, delete operations?**
    - ☞ **2-3 (search) Trees**
    - ☞ **2-3-4 (search) Trees**
    - ☞ **B-trees**

Advanced Binary Search Trees

- How?
  - ➔**Height Balanced** Binary Search Trees

A Balanced Tree?

- A tree with some balancing condition.
- A **height balanced** tree
  - ➔ A tree where no leaf is much farther away from the root than any other leaf.
  - ➔ Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

A Balanced Search Tree?

- A **search tree** with some **balancing condition**.
- A **height balanced** search tree

AVL Trees

(Height-Balanced Binary Search Trees)

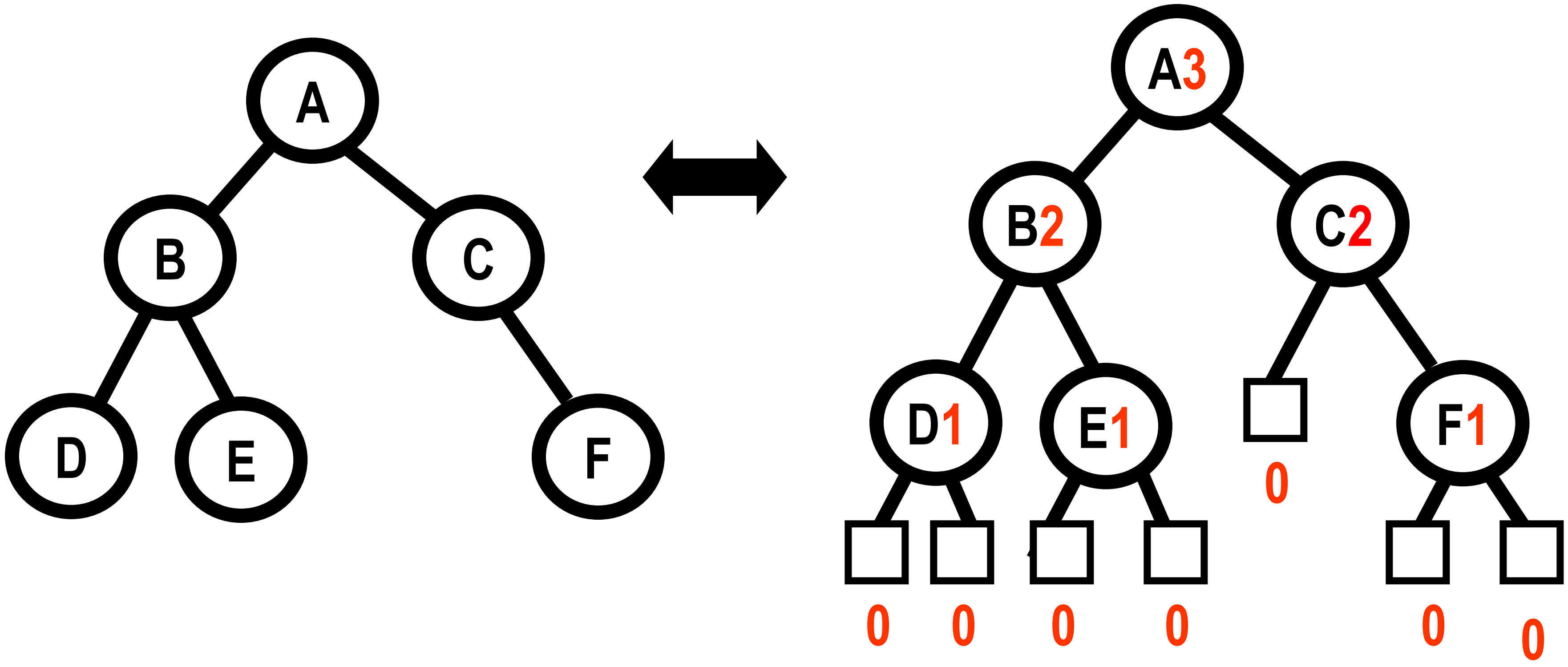
AVL (Search) Tree

- A height-**balanced** **binary search tree**
  - Height is balanced!
  - All data items are kept in sorted order!

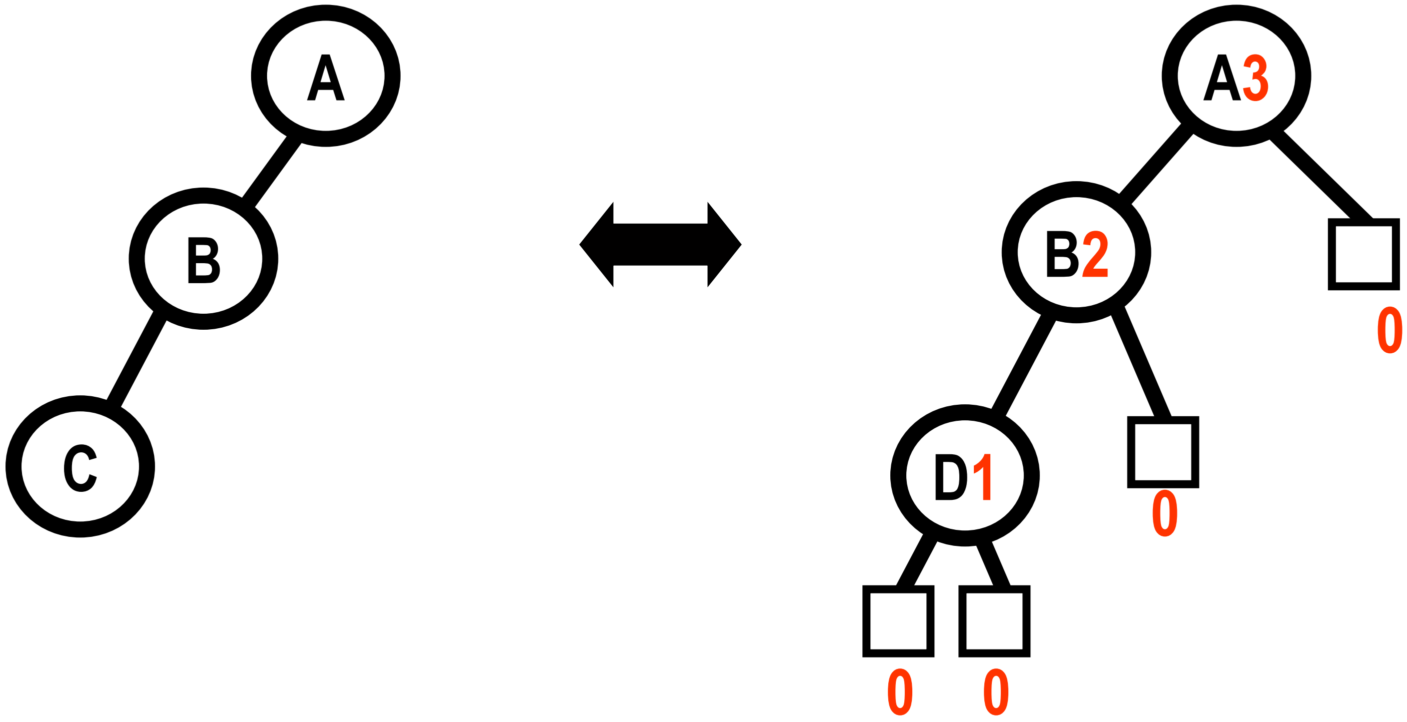
The Height in AVL Tree

- For a node x:
  - ➔**height(x) = The longest path length from the node x to an external node.**
- **height(x)** =
  - ➔**0** if x is an external node
  - ➔**1 + max (height(LeftChild of x), height(RightChild of x))** if x is an internal node

Example: The Height inTree



▶QUIZ? The Height in Tree



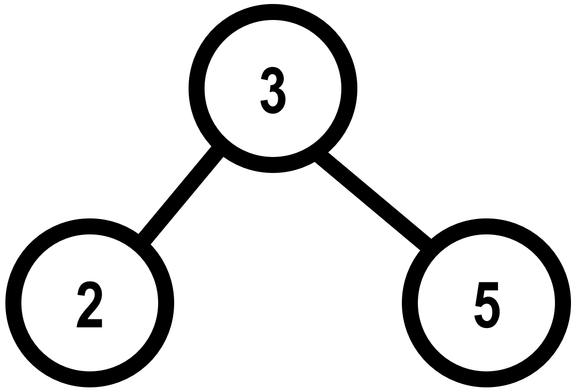
Height vs Shortest Path Length

- height(x) =
  - 0 if x is an external node
  - 1 + max (height(LeftChild of x), height(RightChild of x)) if x is an internal node
- spl(x) = shortest **p**ath **l**ength

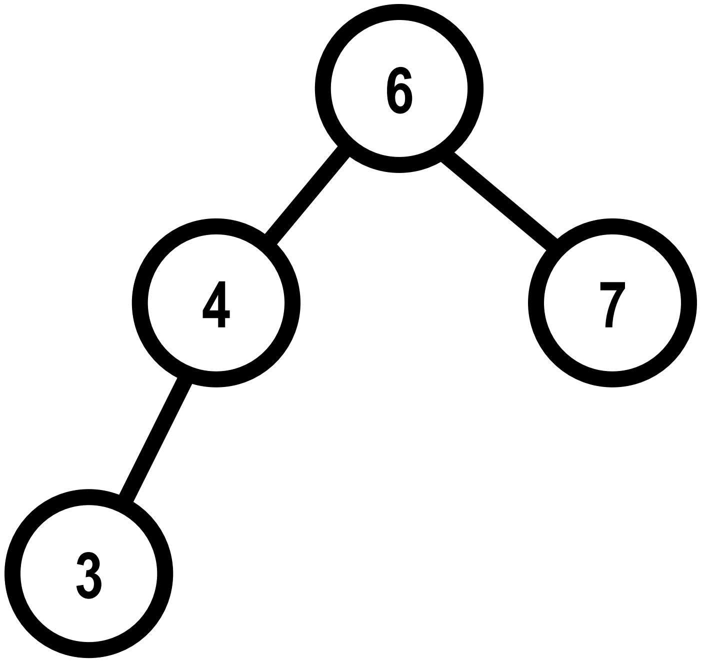
AVL Balancing Condition

- The AVL Balancing Condition:
  - An empty binary tree is AVL balanced.
  - A non-empty binary tree T with T_L and T_R as its left and right subtrees respectively is AVL balanced if
    - (1) Both T_L and T_R are **AVL balanced** &
    - (2) | The **height** of T_L - The **height** of T_R | ≤ **1**.

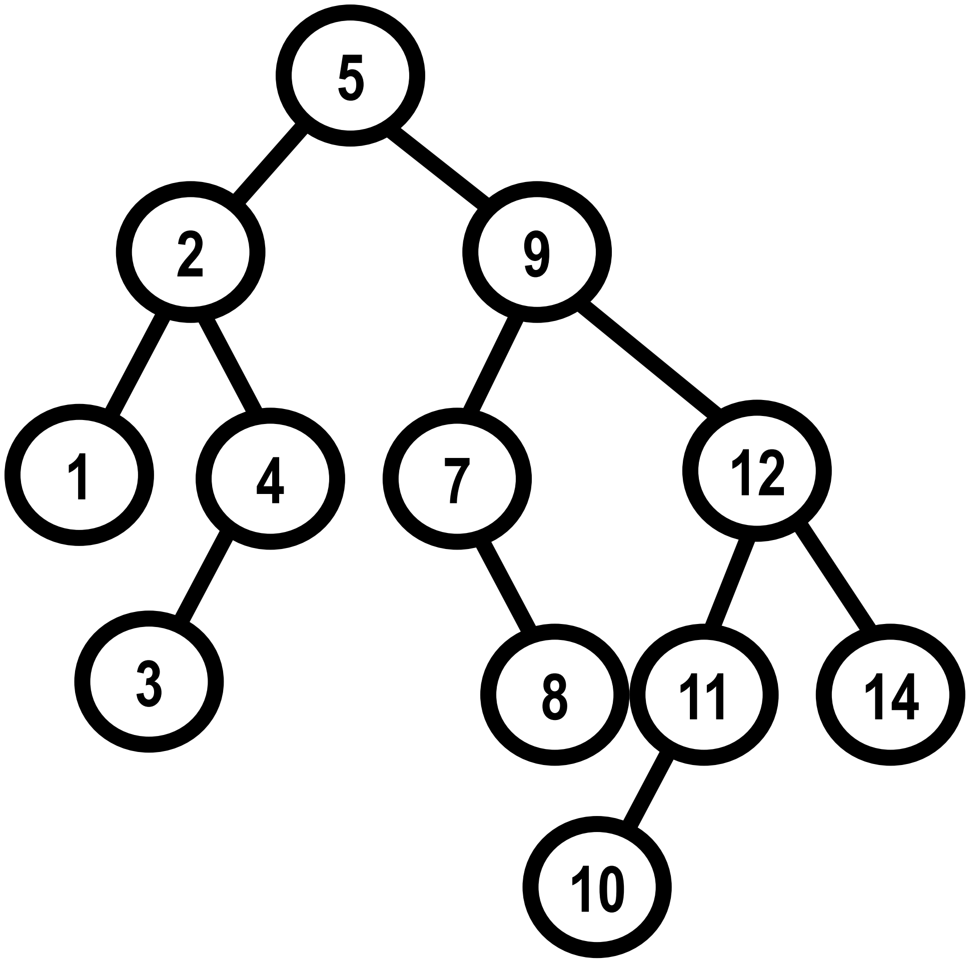
Example: AVL Tree?



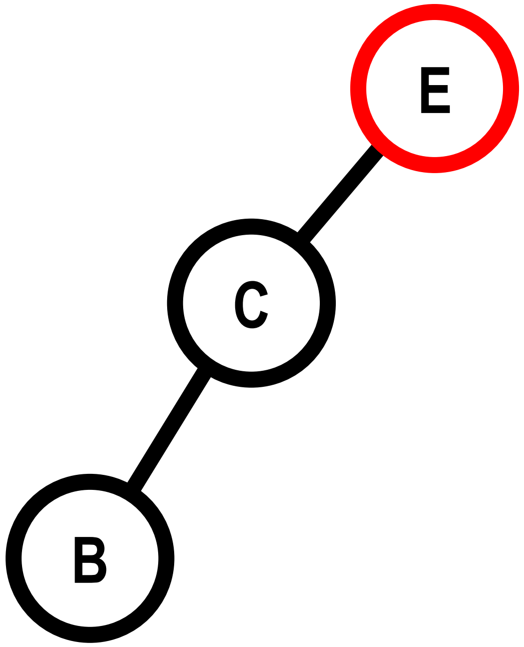
Example: AVL Tree?



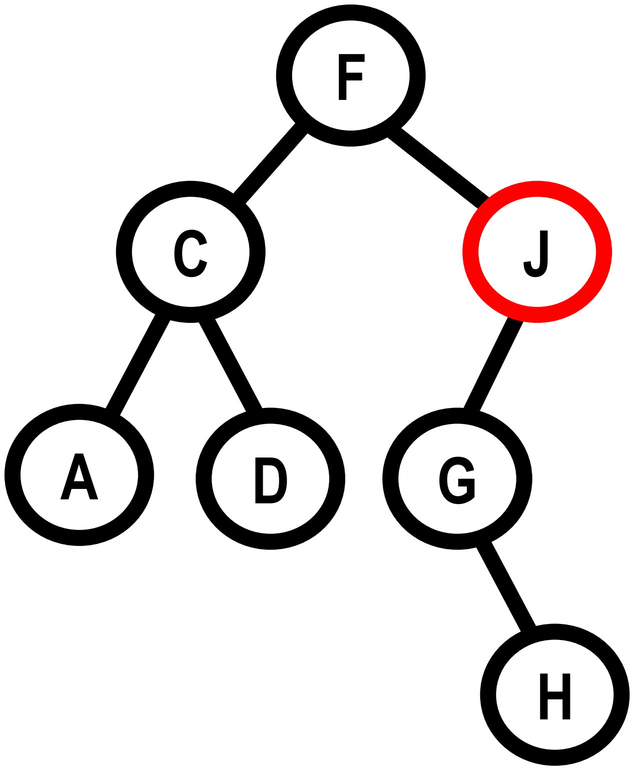
Example: AVL Tree?



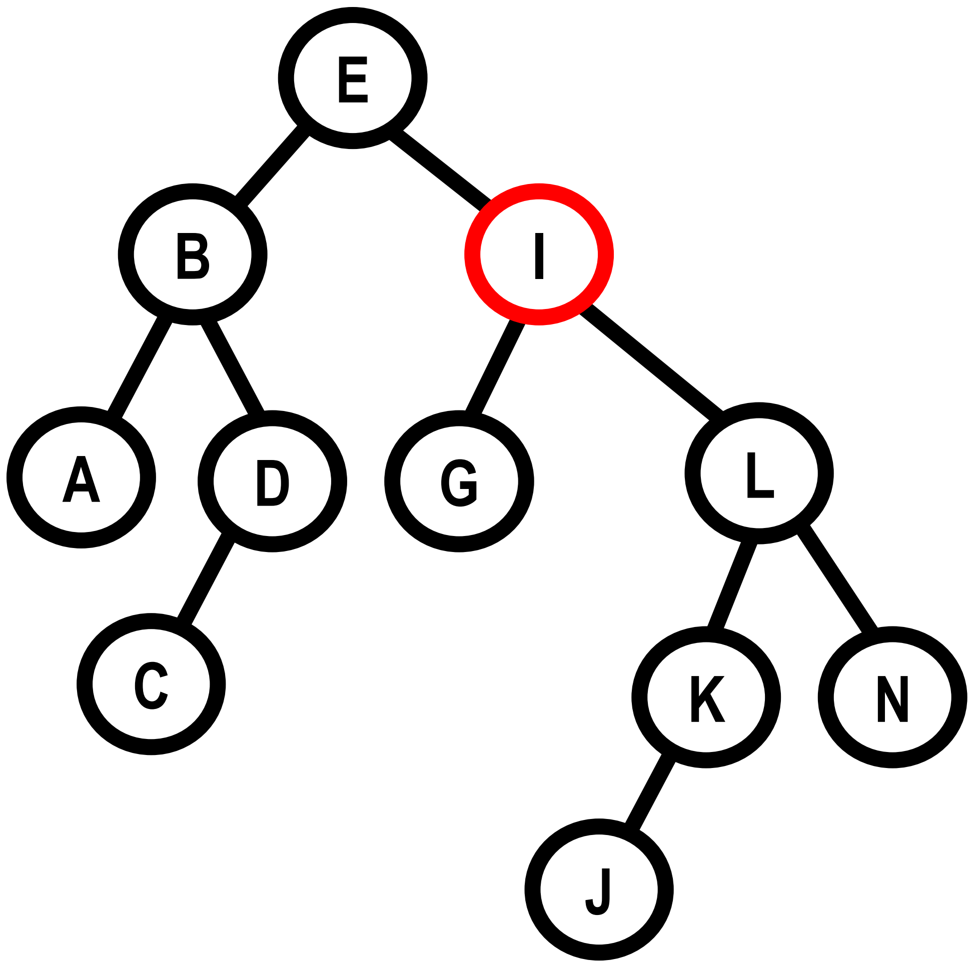
Example: AVL Tree?



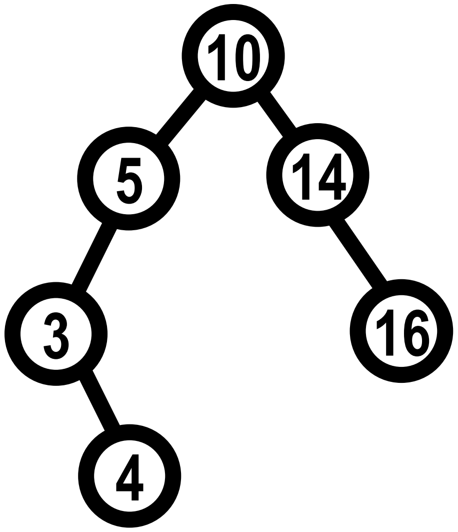
Example: AVL Tree?

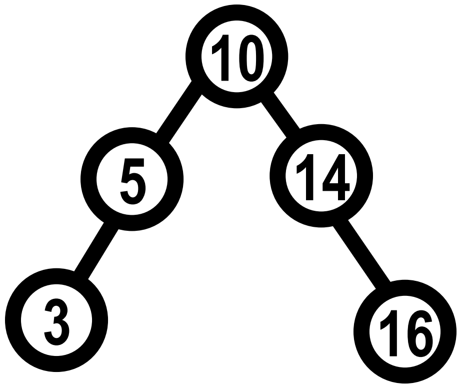


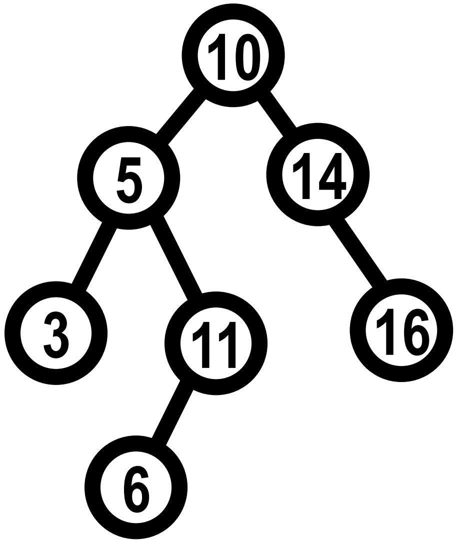
Example: AVL Tree?

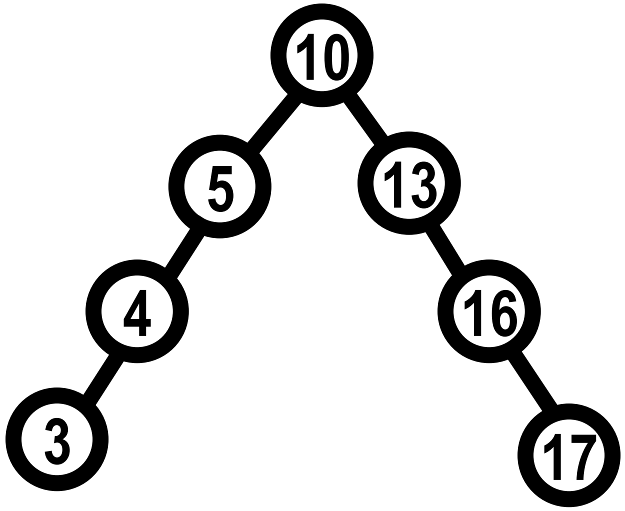


►QUIZ? AVL Tree?









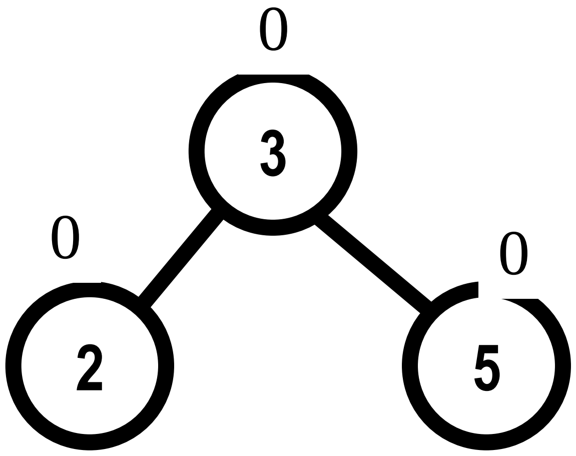
Balance Factor

- **Balance factor** of a node N in a binary tree is
  - **(The height of N_L - The height of N_R)** where N_L and N_R are left and right subtrees of N respectively.
  - BF(N) > 0
    - left subtree too high
  - BF(N) = 0
    - balanced
  - BF(N) < 0
    - right subtree too high

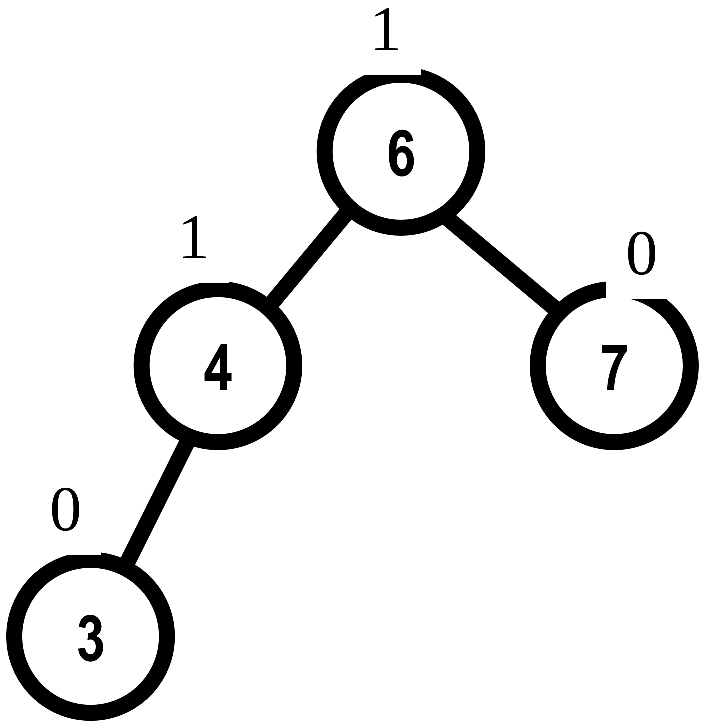
Balance Factors of Nodes in an AVL Tree

- For any node in an AVL tree:
  - BF(N) = 1
  - BF(N) = 0
  - BF(N) = -1

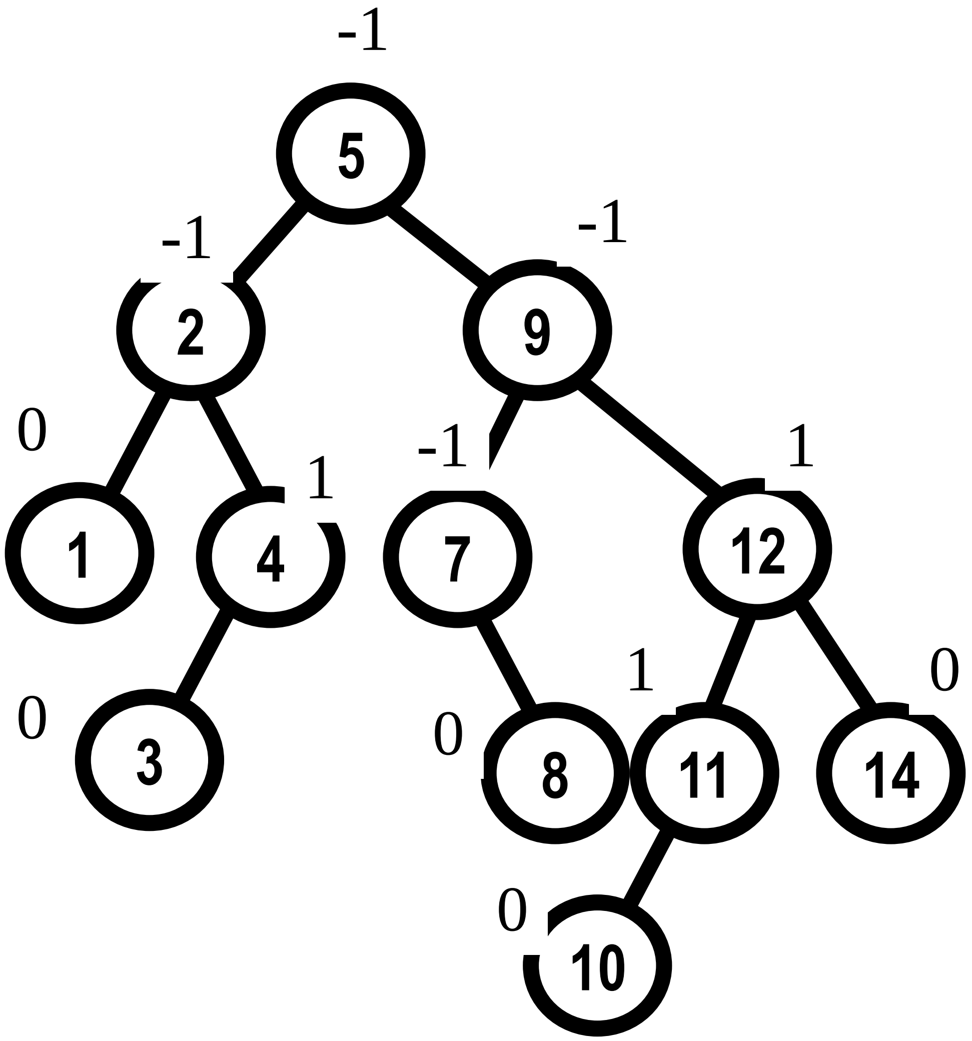
Example: AVL Tree?



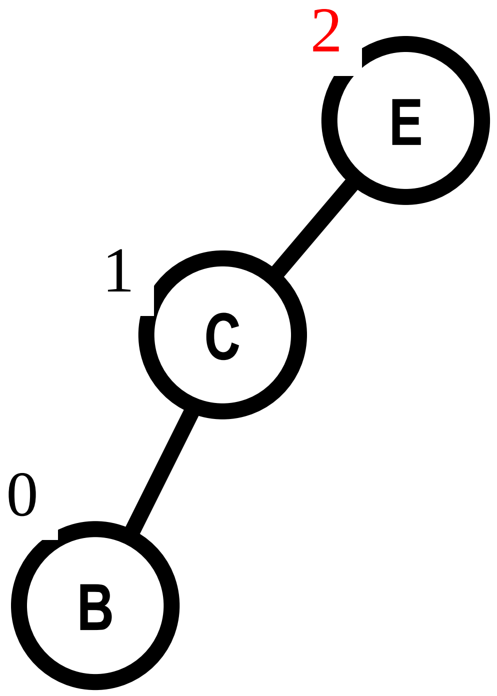
Example: AVL Tree?



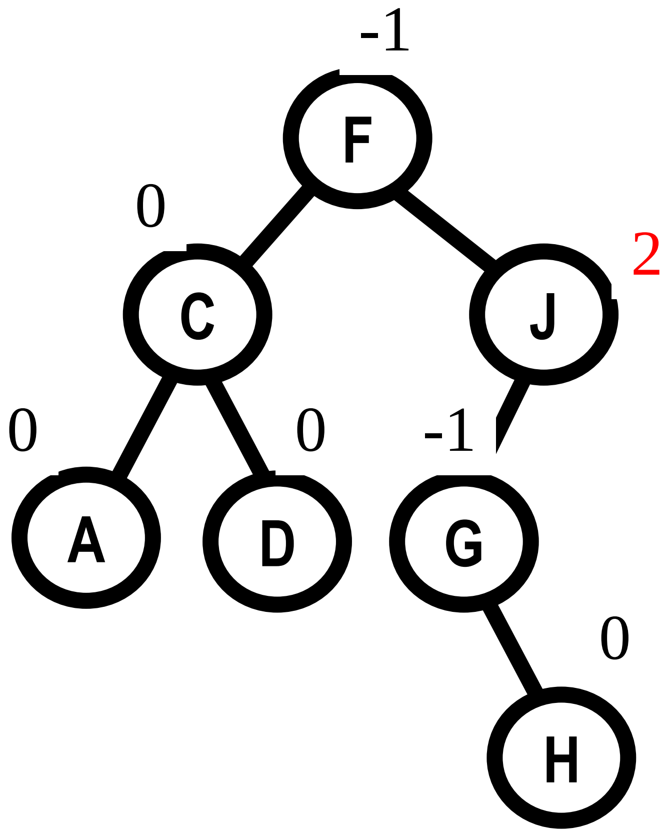
Example: AVL Tree?



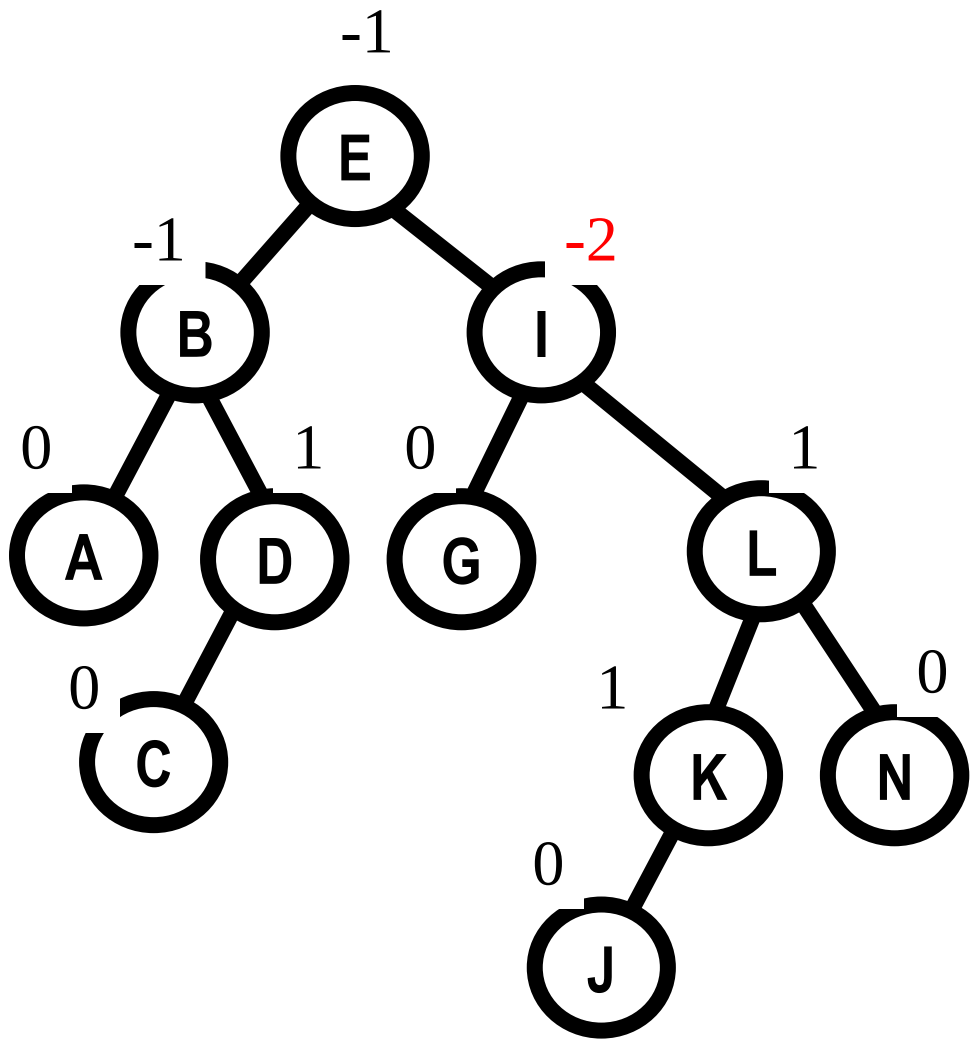
Example: AVL Tree?



Example: AVL Tree?



Example: AVL Tree?



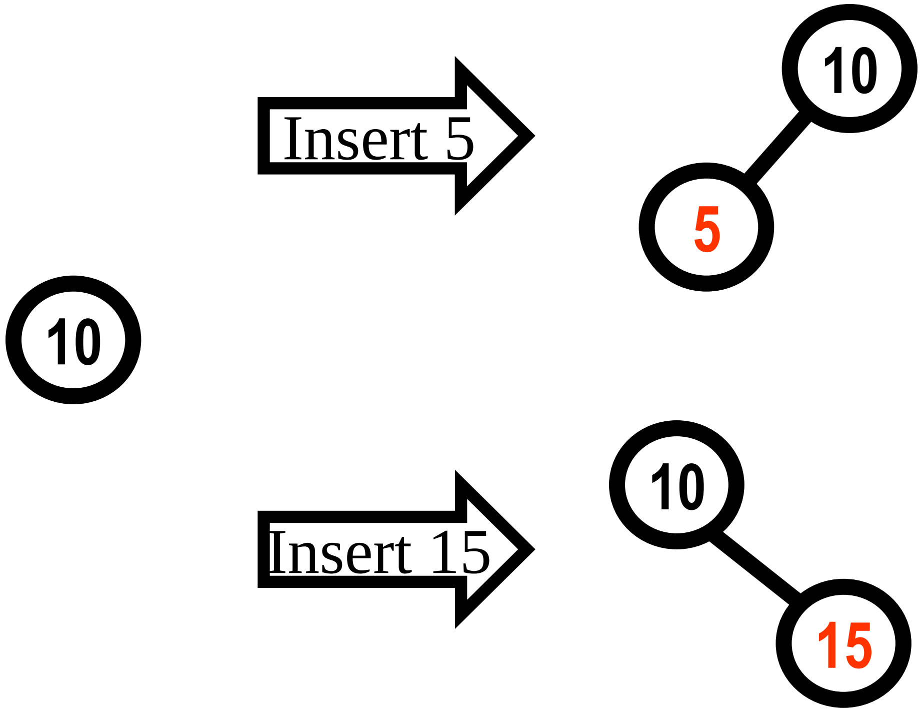
Searching a AVL tree

- Same as with binary search trees.
  - An AVL tree maintains a height close to the minimum.
  - We can search an AVL tree almost as efficiently as a minimum-height binary search tree.

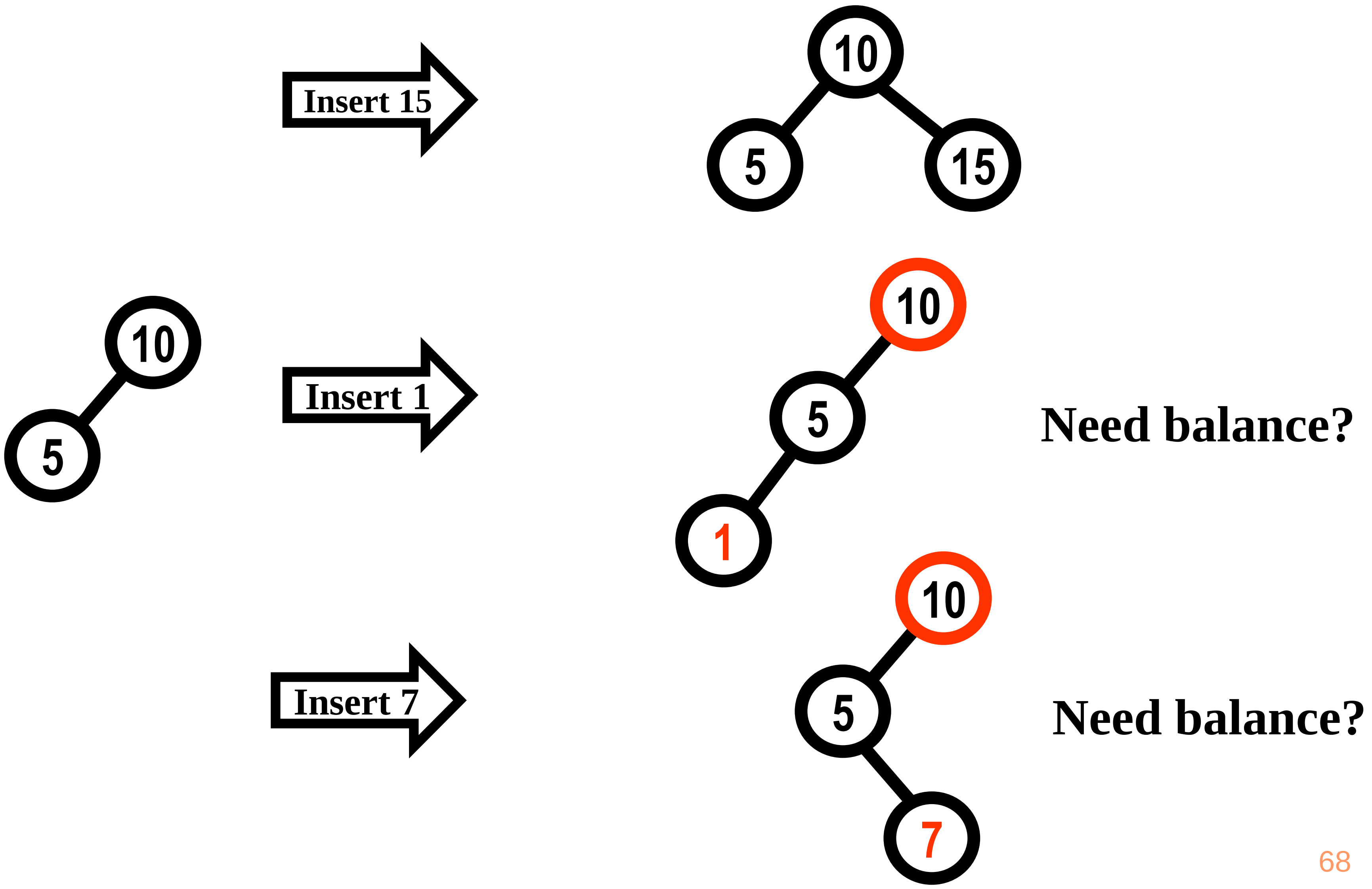
Inserting Items into an AVL tree

- Inserting an item into an AVL tree is a two-part process:
  - ➔The item is inserted using the same method of insertion in binary search trees.
  - ➔**Check** the resulting tree is **AVL balanced**.
    - ☞Check whether any node in the tree has left and right subtrees whose heights differ by more than 1.
  - ➔If not, **balance it**!

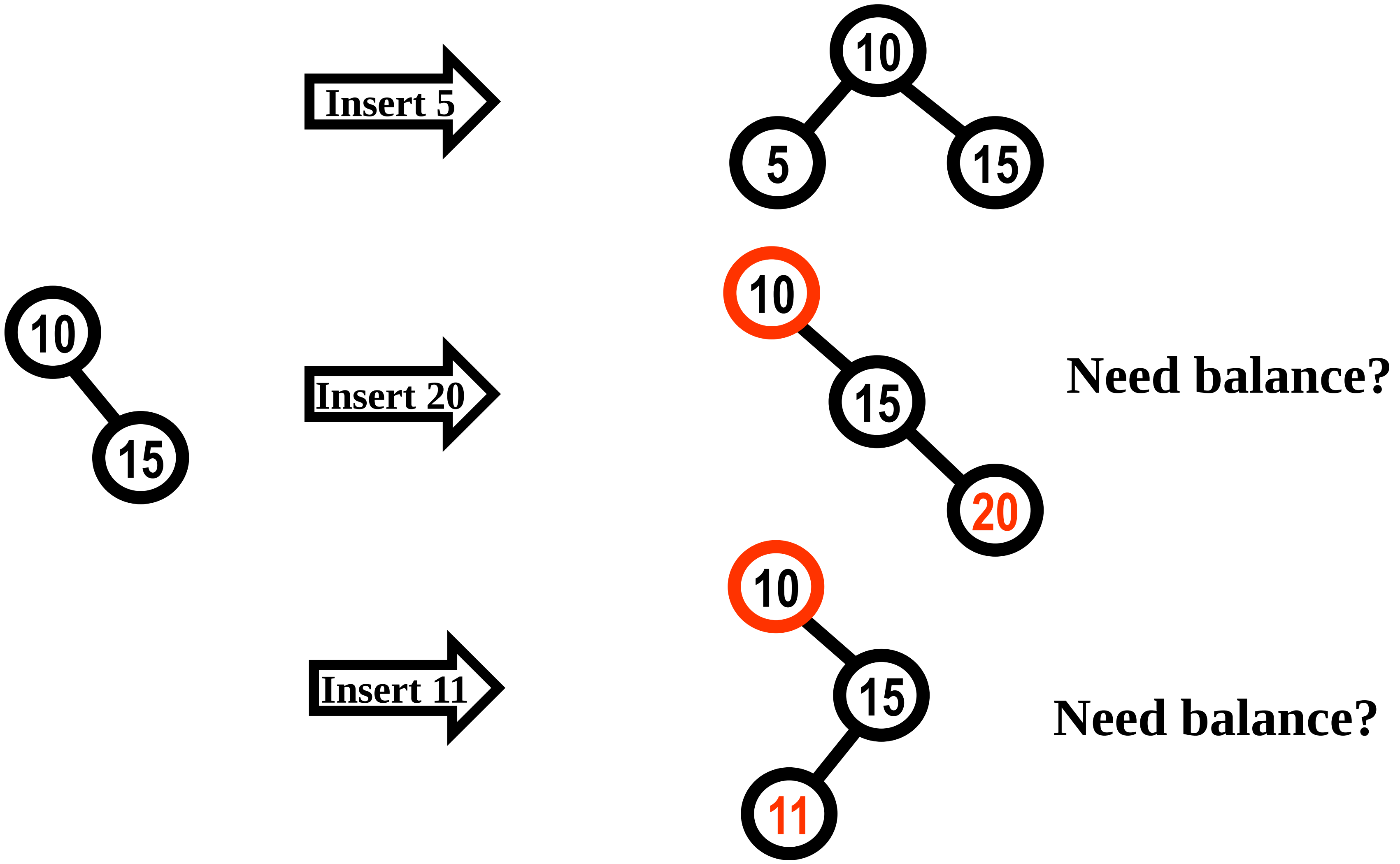
Example: Inserting Items into an AVL tree



Example: Inserting Items into an AVL tree



Example: Inserting Items into an AVL tree



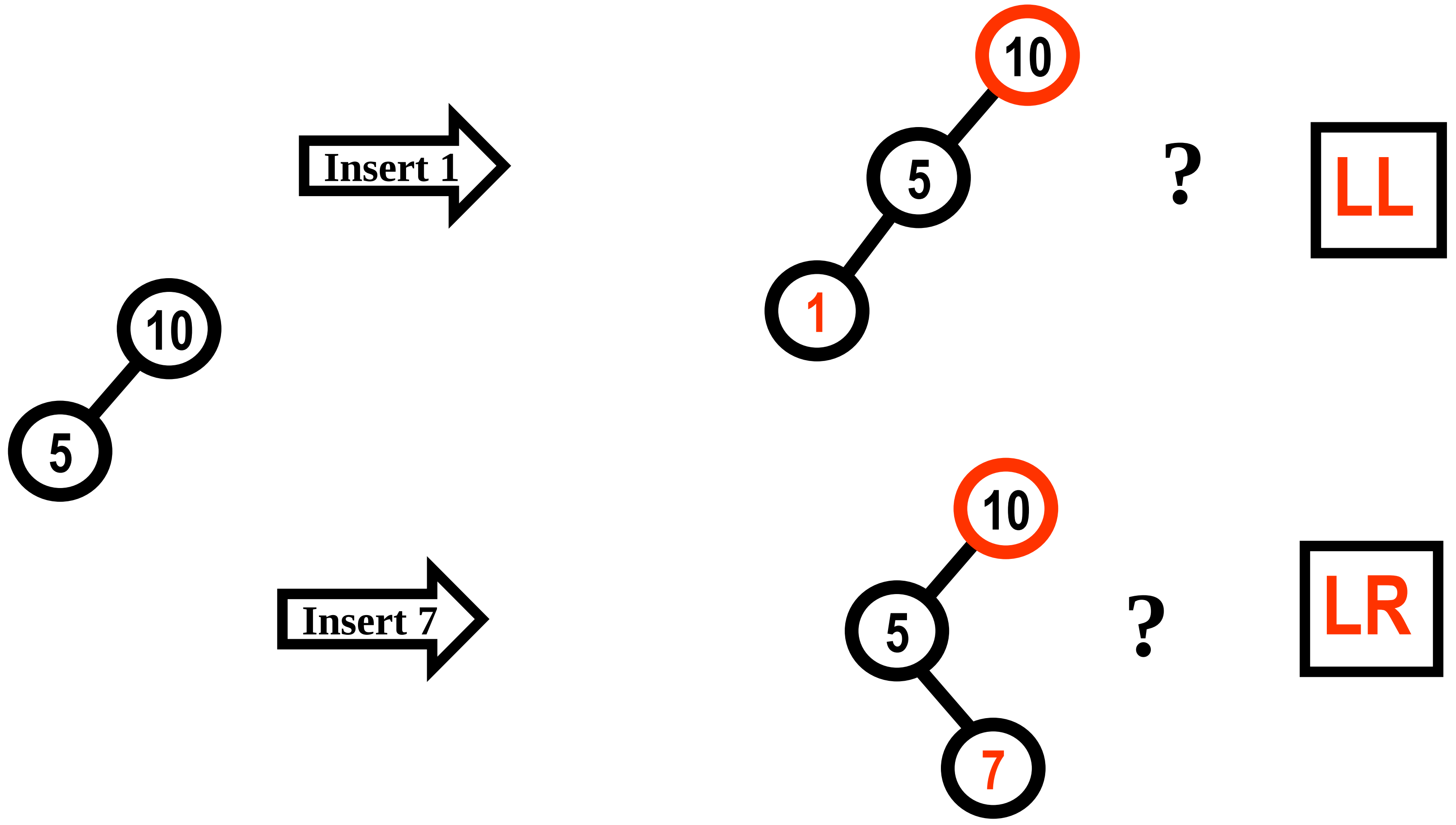
Rebalancing in Insertion

- Insertion may destroy the AVL balancing condition!
  - ➔Need to balance it!

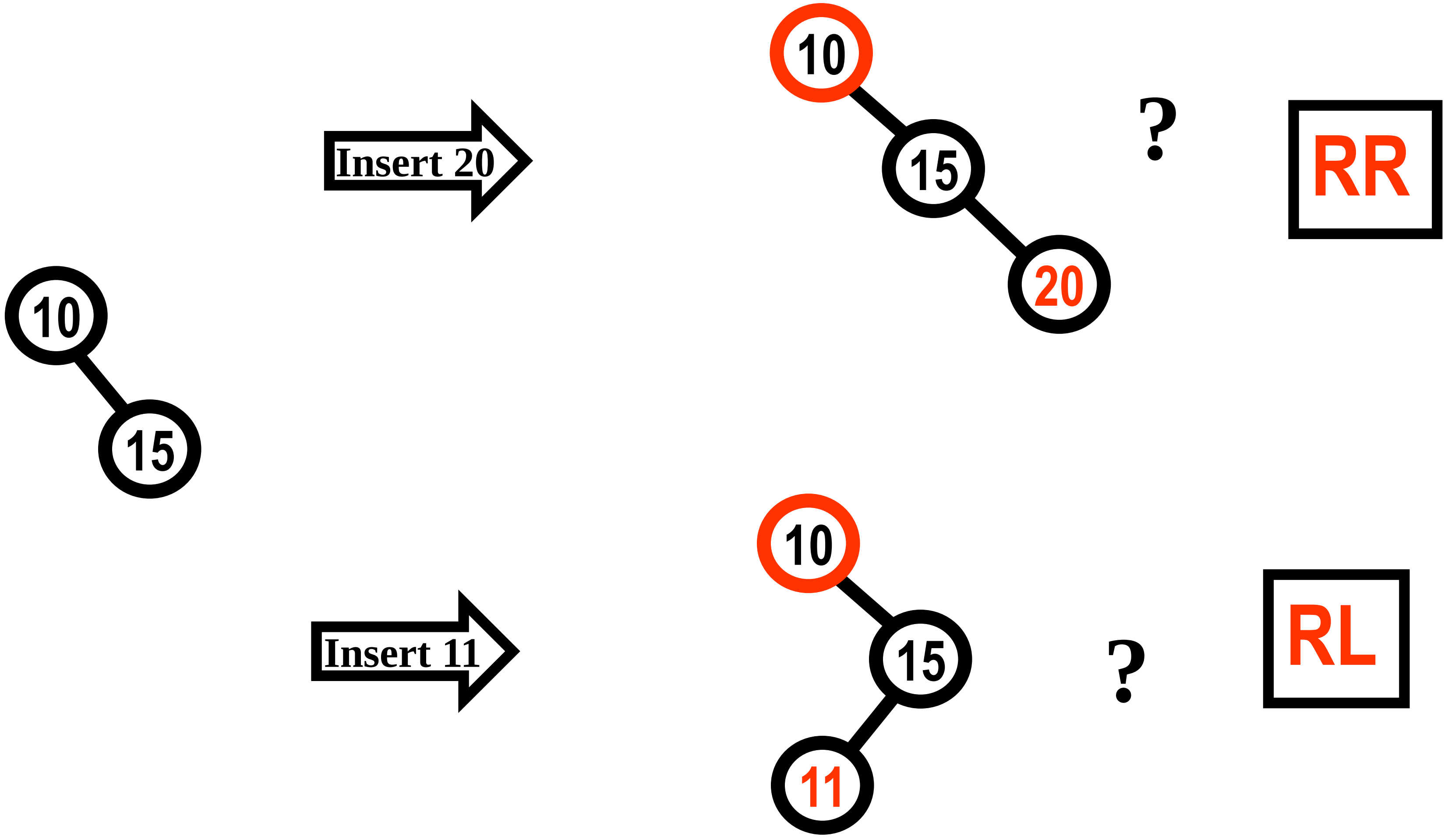
Four Cases for Rebalancing in Insertion

- Characterized by the nearest ancestor A of the inserted node whose balance factor becomes +2 or -2.
  - LL: Inserted in the left subtree of the left subtree of A.
  - RR: Inserted in the right subtree of the right subtree of A.
  - LR: Inserted in the right subtree of the left subtree of A.
  - RL: Inserted in the left subtree of the right subtree of A.

Example: Four Cases



Example: Four Cases



How to Check Four Cases for Rebalancing in Insertion?

- X = A new data
- T = Pointer to a node whose balance factor becomes +2 or -2.
  - If X < T->left->data then
    - LL: Inserted in the left subtree of the left subtree of A.
  - If X > T->right->data then
    - RR: Inserted in the right subtree of the right subtree of A.

How to Check Four Cases for Rebalancing in Insertion?

- X = A new data
- T = Pointer to a node whose balance factor becomes +2 or -2.
  - If X > T->left->data then
    - LR: Inserted in the right subtree of the left subtree of A.
  - If X < T->right->data then
    - RL: Inserted in the left subtree of the right subtree of A.

Re-Balancing AVL trees by Rotations

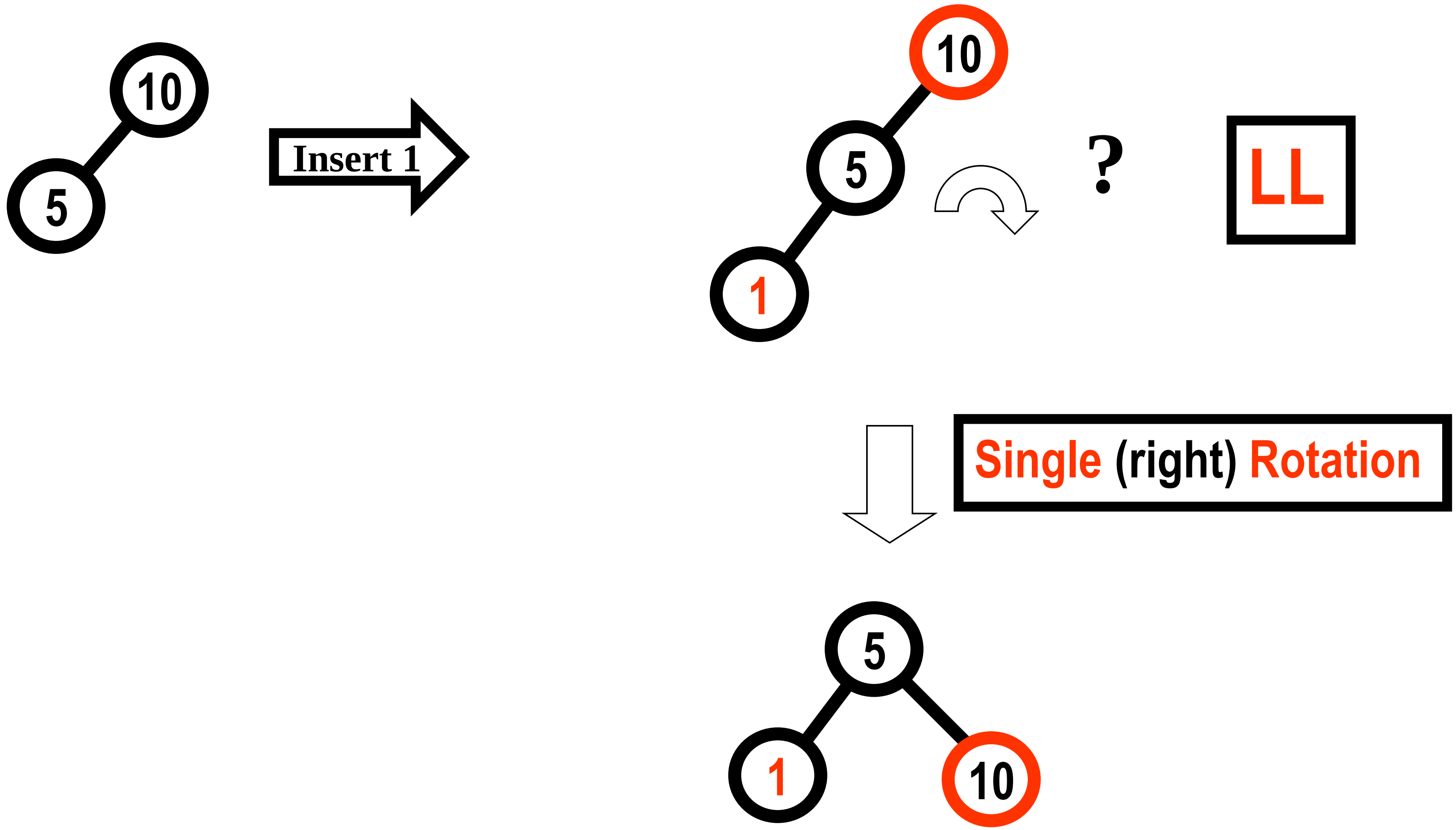
- When an AVL tree becomes unbalanced, bring it back into balance.
- How?
  - By performing an operation called **rotation**.

Idea:
Restore the AVL balance by Rotation!

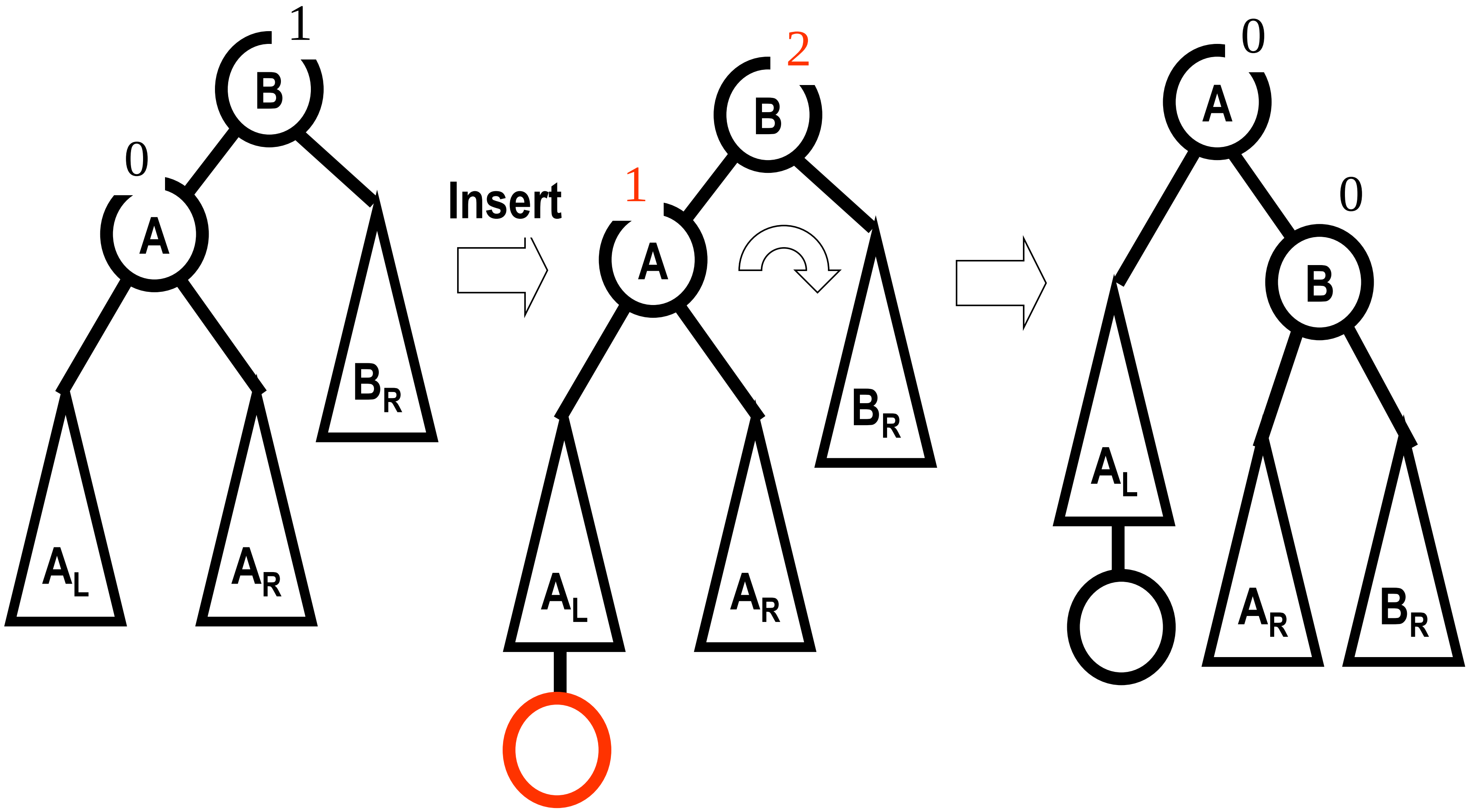
Four Cases for Rebalancing AVL trees

- There are **four cases**:
  - LL
  - RR
  - LR
  - RL
- Each case has its own **rotation**:
  - Case 1: **Single rotation** called LL rotation
  - Case 2: **Single rotation** called RR rotation
  - Case 3: **Double rotation** called LR rotation
  - Case 4: **Double rotation** called RL rotation

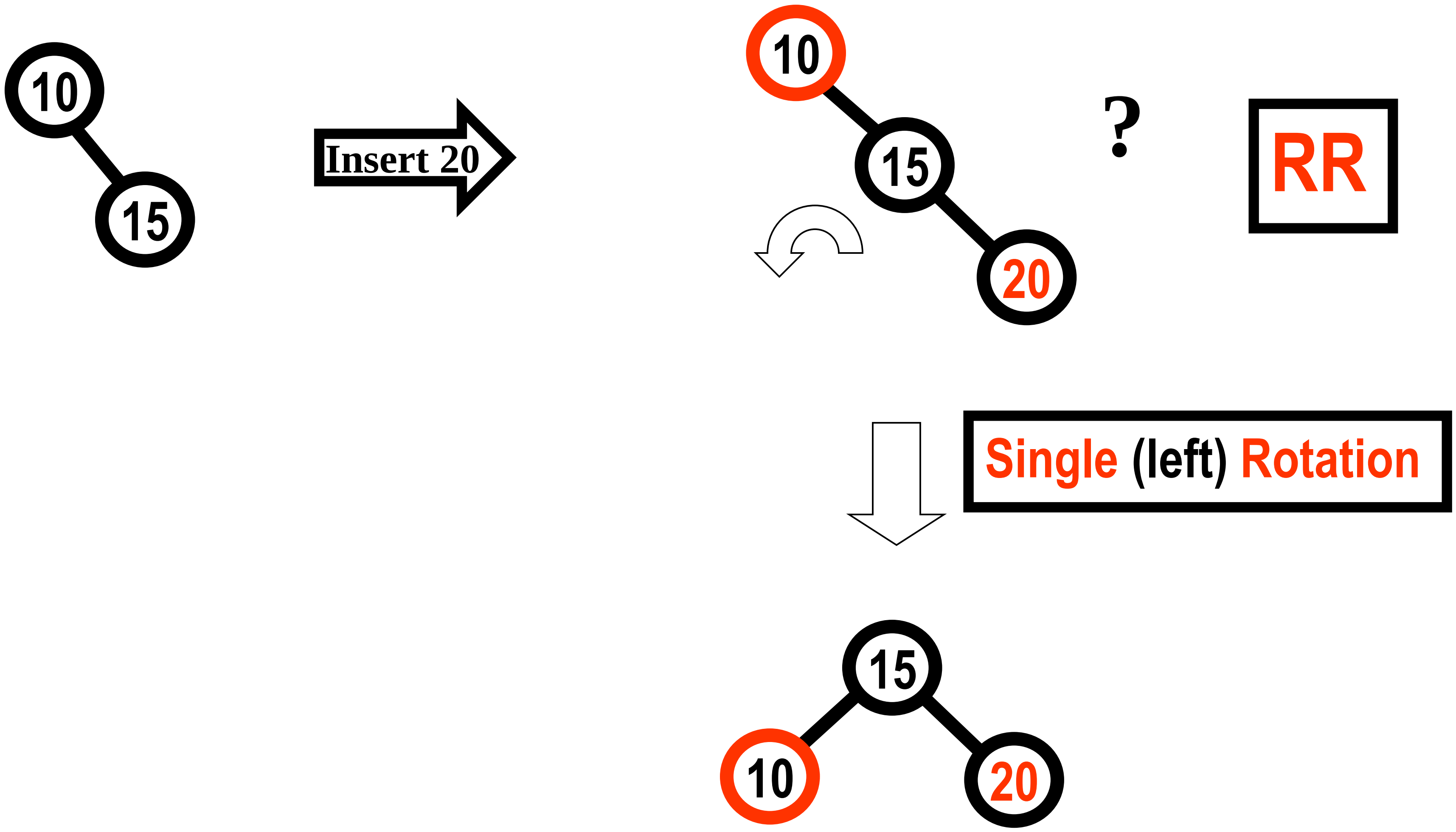
Example: Case 1



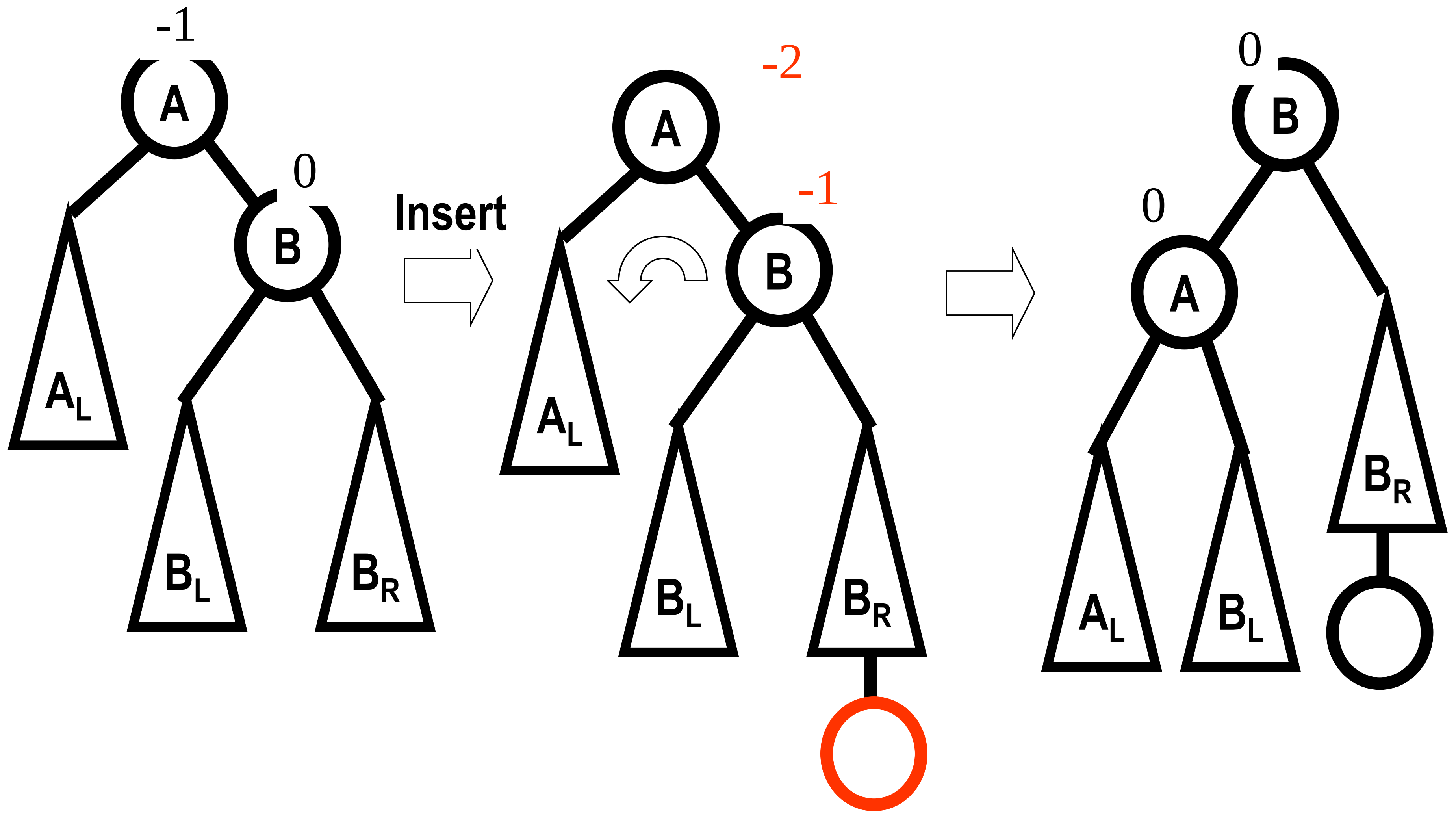
Case 1: Single rotation called LL rotation



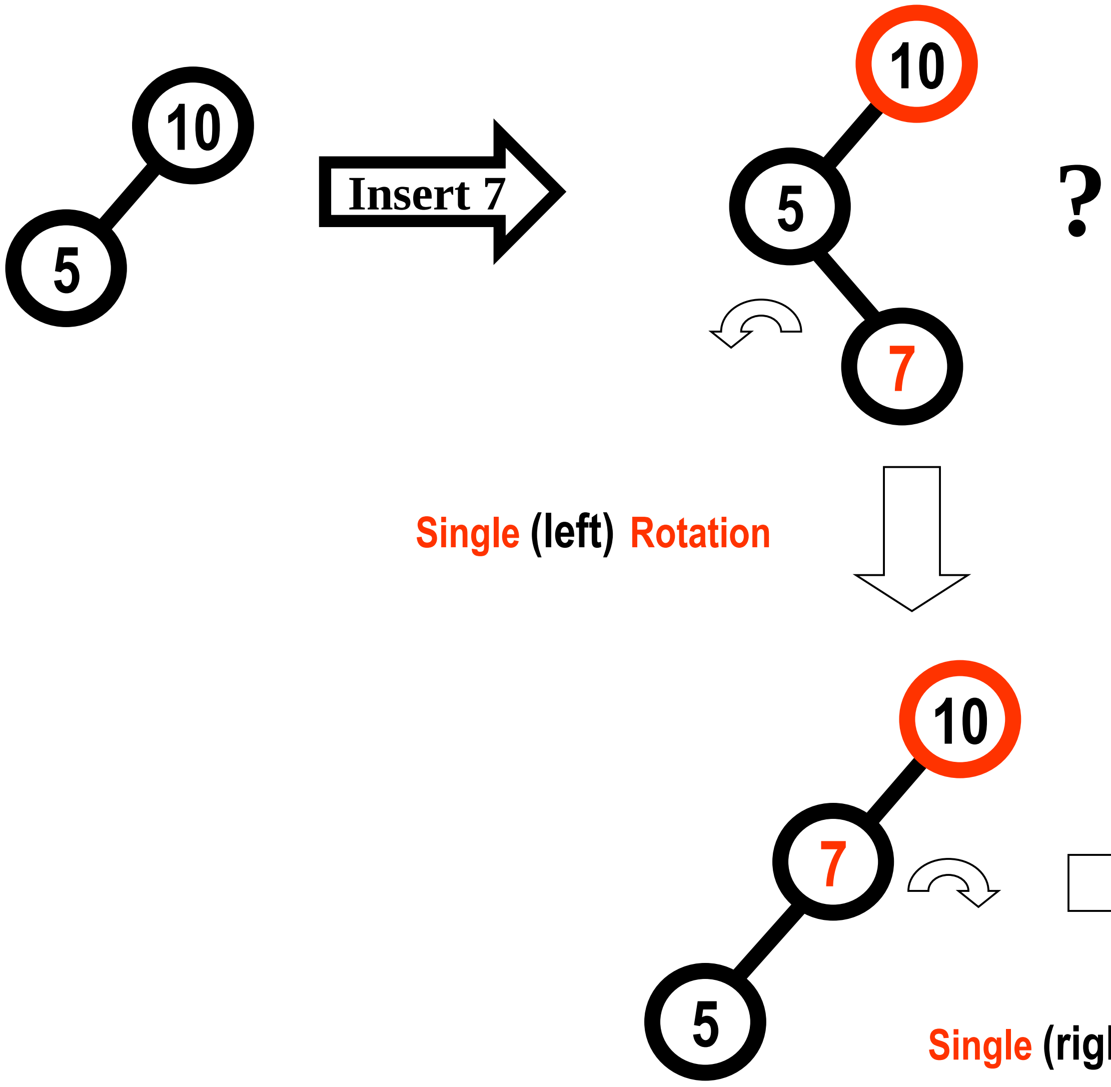
Example: Case 2



Case 2: Single rotation called RR rotation



Example: Case 3



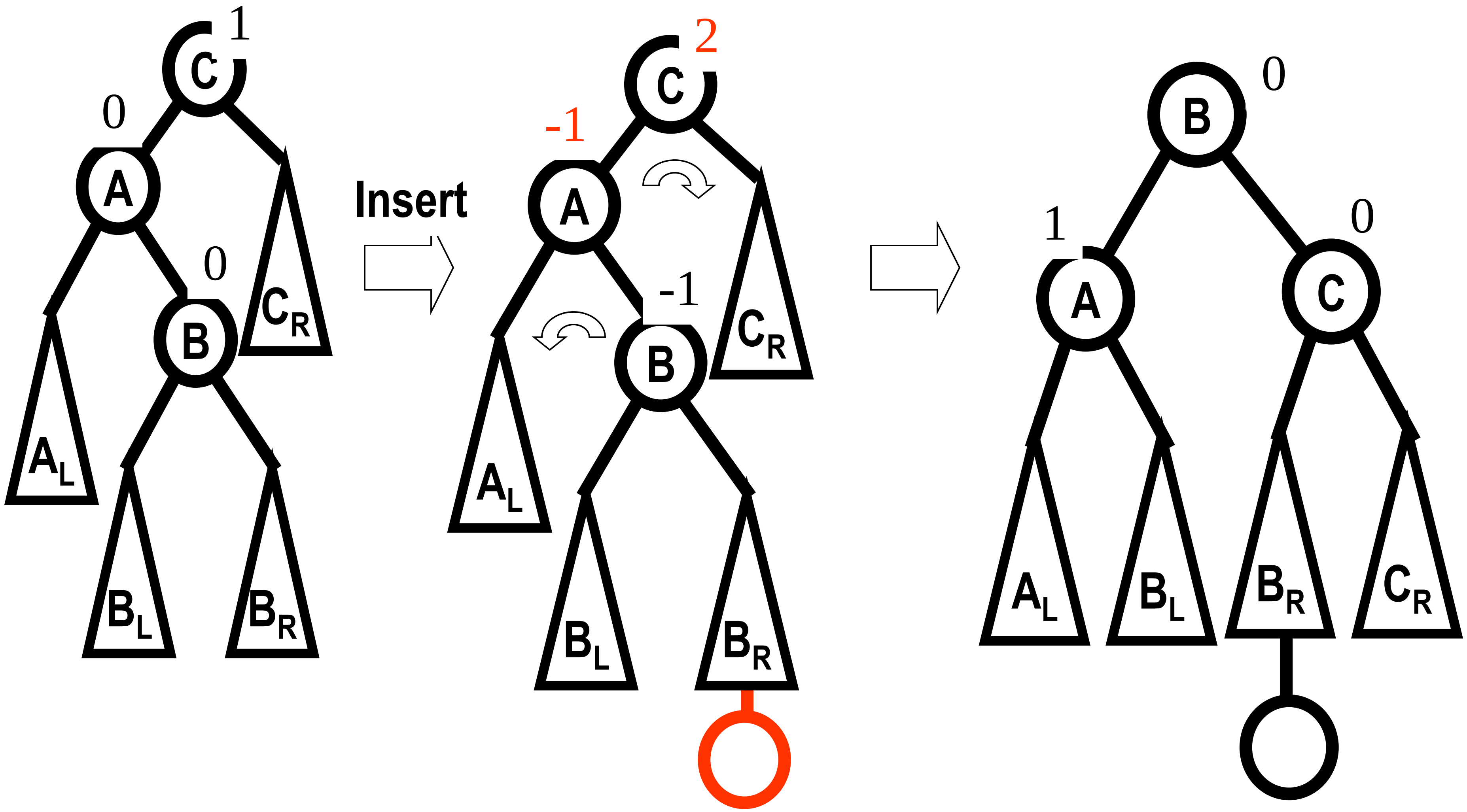
LR

Single (left) Rotation

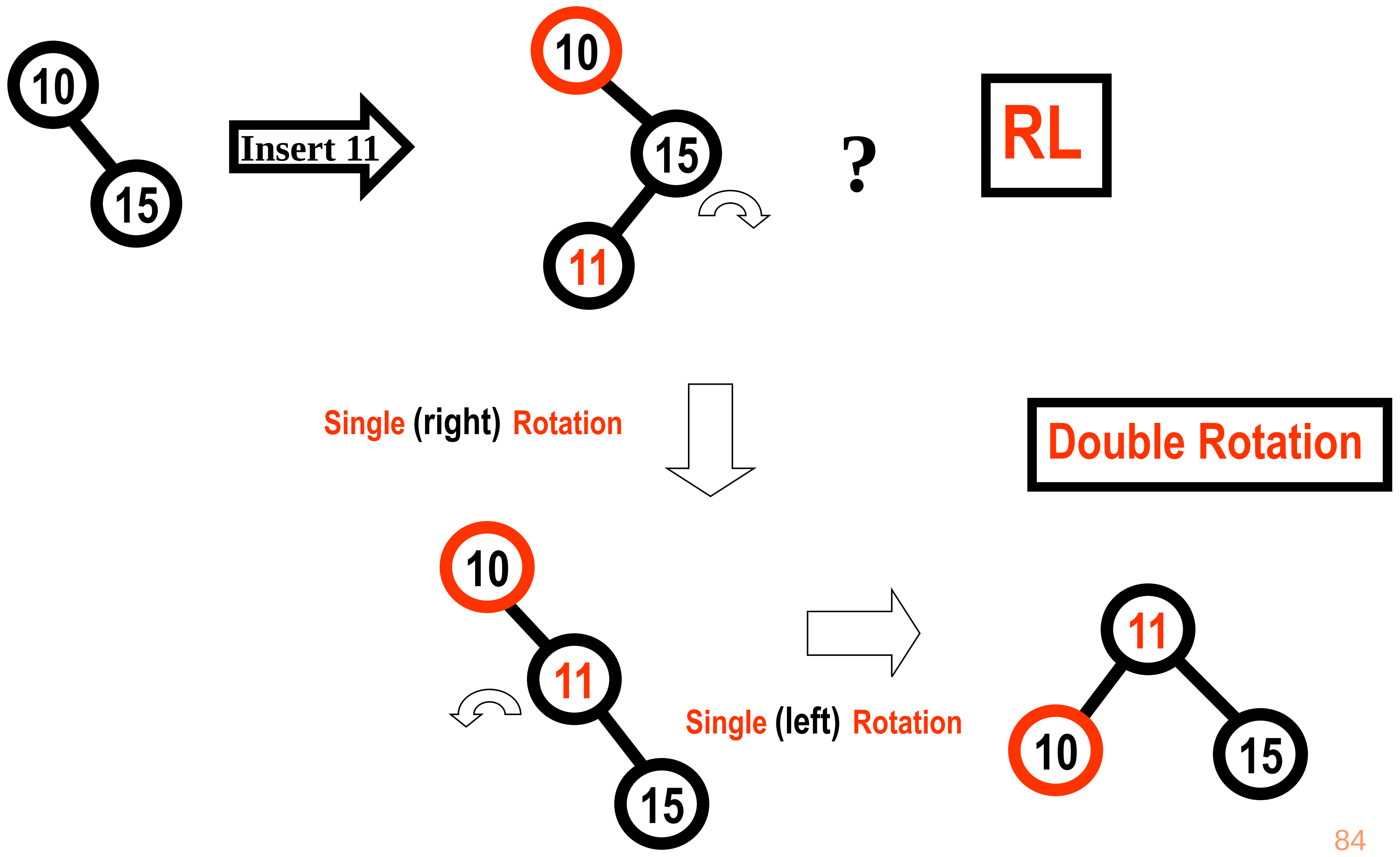
Double Rotation

Single (right) Rotation

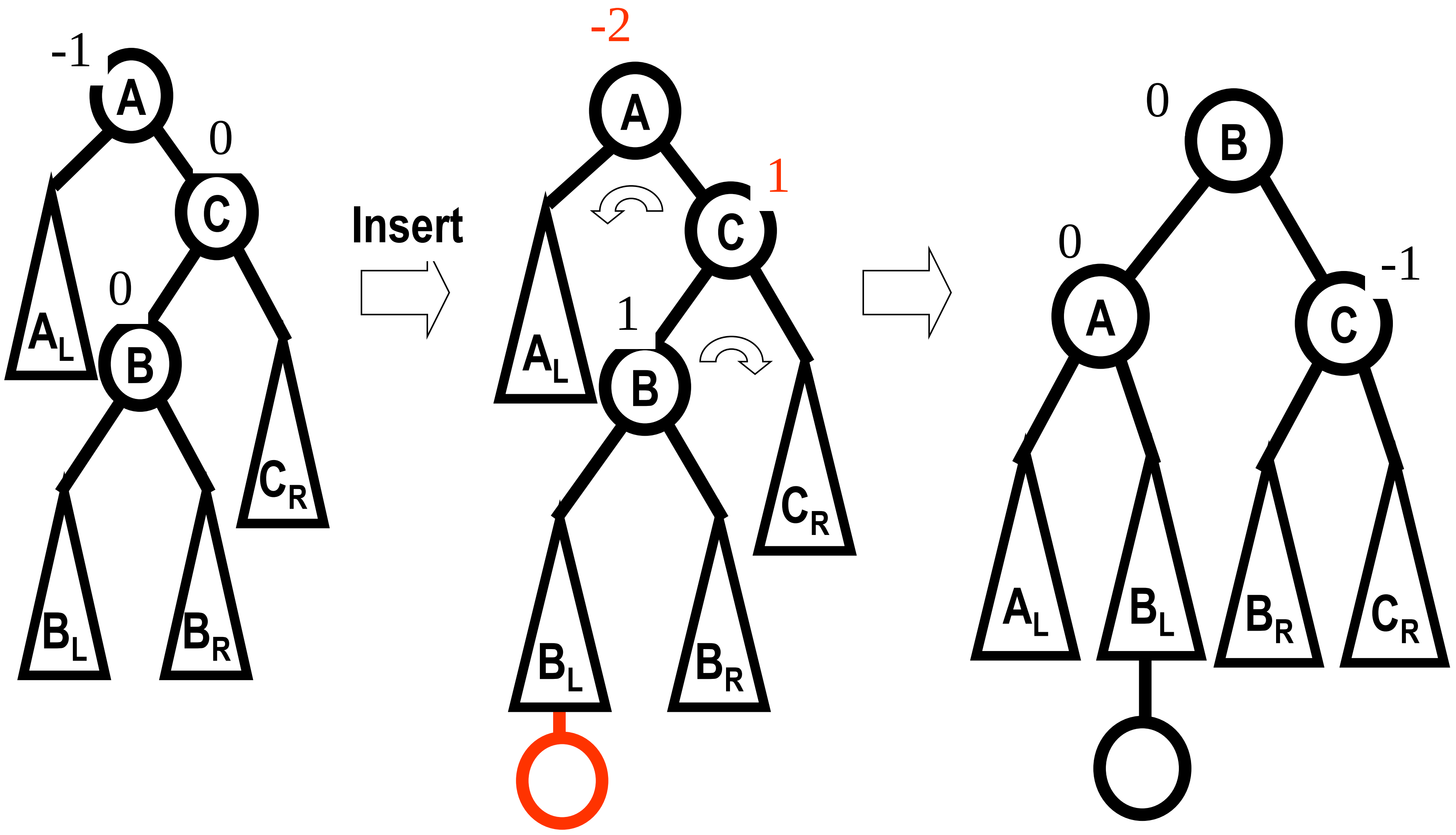
Case 3: Double rotation called LR rotation



Example: Case 4



Case 4: Double rotation called RL rotation



Inserting Items into an AVL tree

- Inserting an item into an AVL tree is a two-part process:
  - The item is inserted using the usual method of insertion in binary search trees.
  - Check the resulting tree is AVL balanced.
    - Check whether any node in the tree has left and right subtrees whose heights differ by more than 1.
  - If not, **balance it!**
    - Case 1: LL - Single rotation called LL rotation
    - Case 2: RR - Single rotation called RR rotation
    - Case 3: LR - Double rotation called LR rotation
    - Case 4: RL - Double rotation called RL rotation

Inserting Item x into an AVL tree whose root is pointed by T

- **Insert(x, T)**
  - If T is an empty tree then
    - …
  - else if (x < T->data) then
    - …
  - else if (x > T->data) then
    - …
  - Update the height of the node pointed by T.

Inserting Item x into an AVL tree whose root is pointed by T

- **Insert(x, T)**
  - If T is an empty tree then
    - Create a new node and make T point to this new node.
  - Update the height of the node pointed by T.

Inserting Item x into an AVL tree whose root is pointed by T

- **Insert(x, T)**
  - else if (x < T->data) then
    - insert(x, T->left);
    - Check the node pointed by T is AVL balanced (the height difference of left and right subtrees is 2);
    - If not balanced, then determine the case and balance;
      - Case 1: **LL** - Single rotation called LL rotation
      - Case 3: **LR** - Double rotation called LR rotation
  - Update the height of the node pointed by T.

Inserting Item x into an AVL tree whose root is pointed by T

- **Insert(x, T)**
  - else if (x > T->data) then
    - insert(x, T->right);
    - Check the node pointed by T is AVL balanced (the height difference of left and right subtrees is 2);
    - If not balanced, then determine the case and balance;
      - Case 2: **RR** - Single rotation called RR rotation
      - Case 4: **RL** - Double rotation called RL rotation.
  - Update the height of the node pointed by T.

Rebalancing AVL trees after Insertion

- After an insertion, only nodes that are on the path from the insertion point to the root might have balance altered.

Rebalancing AVL trees after Insertion

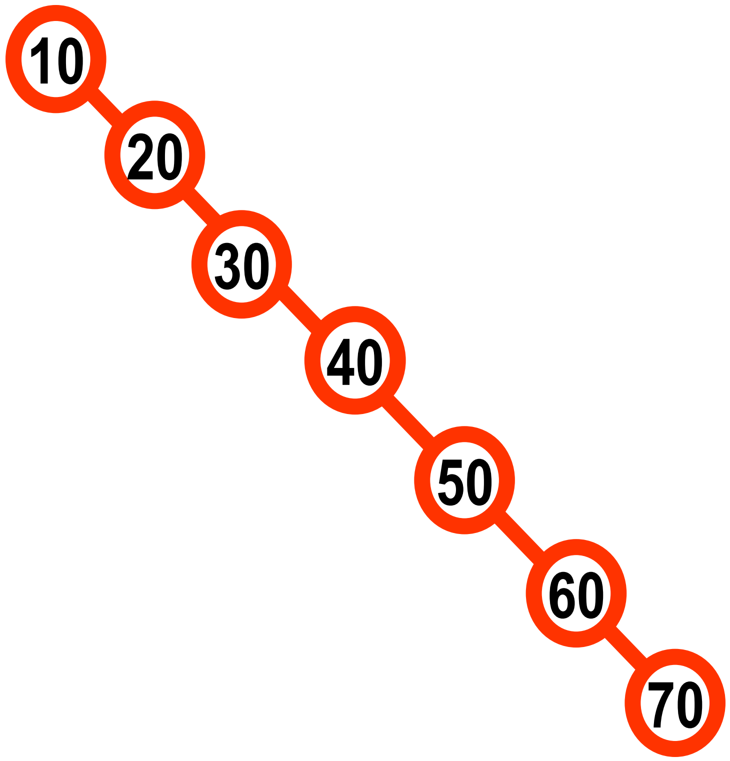
- Follow the path up to the root and check the balancing information.
- Find the first (nearest, deepest) such node.
- Rebalance the tree at that node.

Rebalancing AVL trees after Insertion

- This rebalance guarantees that the entire tree satisfies the AVL balancing condition.
  - ➔**One rotation** (either single or double rotation)!

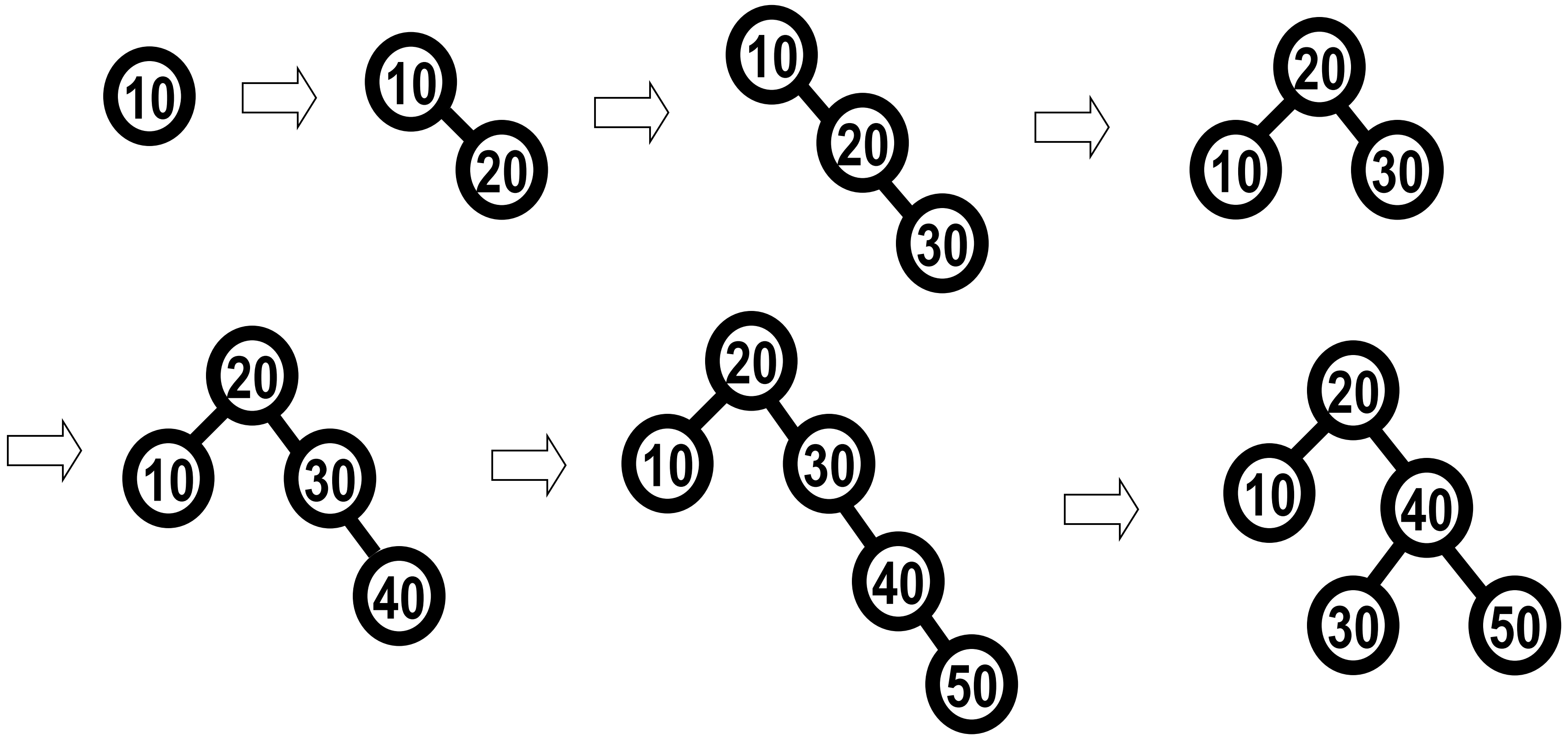
►QUIZ? Binary Search Tree?

Insert 10, 20, 30, 40, 50, 60 and 70



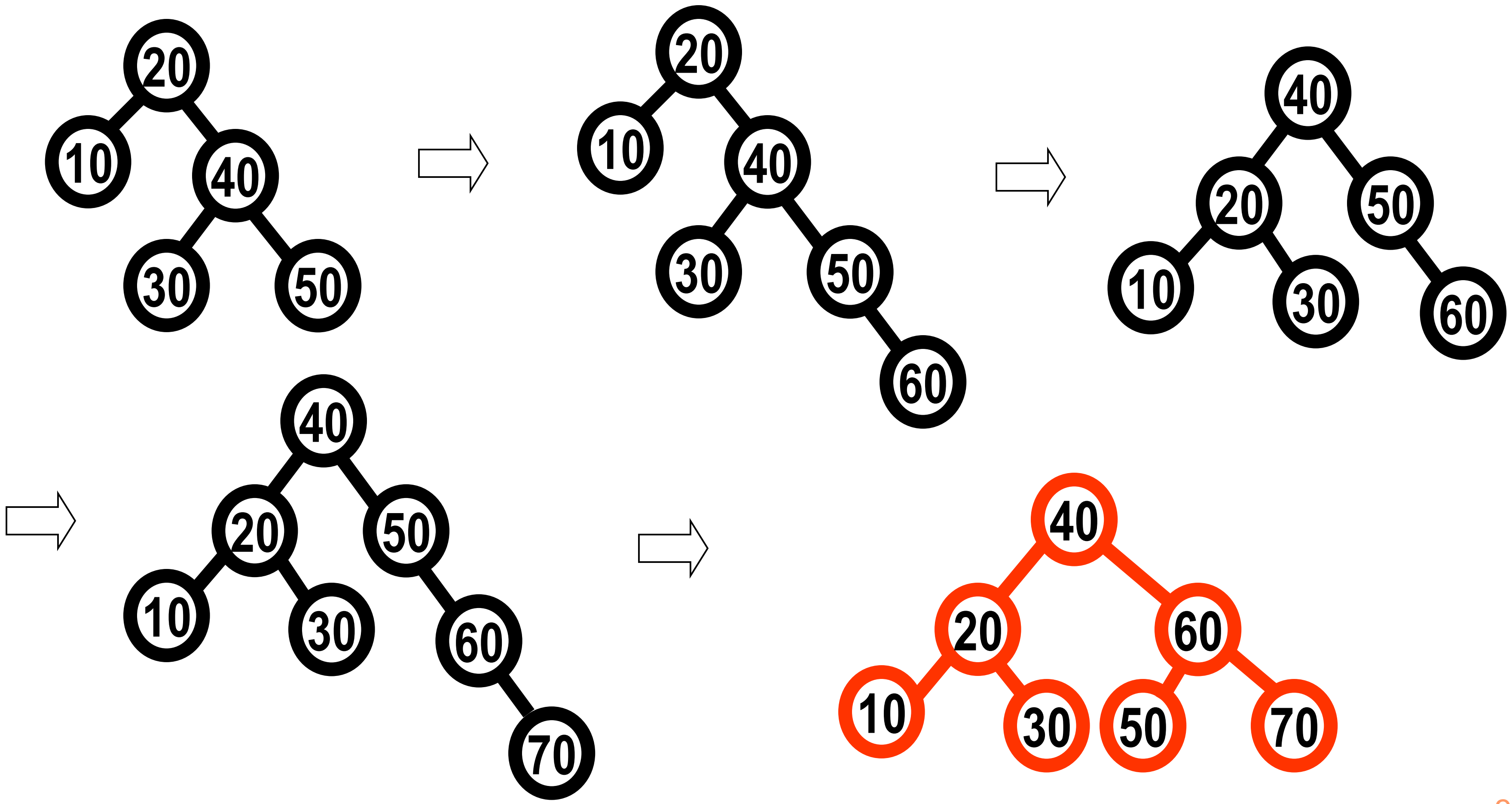
Example: AVL Tree

Insert 10, 20, 30, 40, 50, 60 and 70



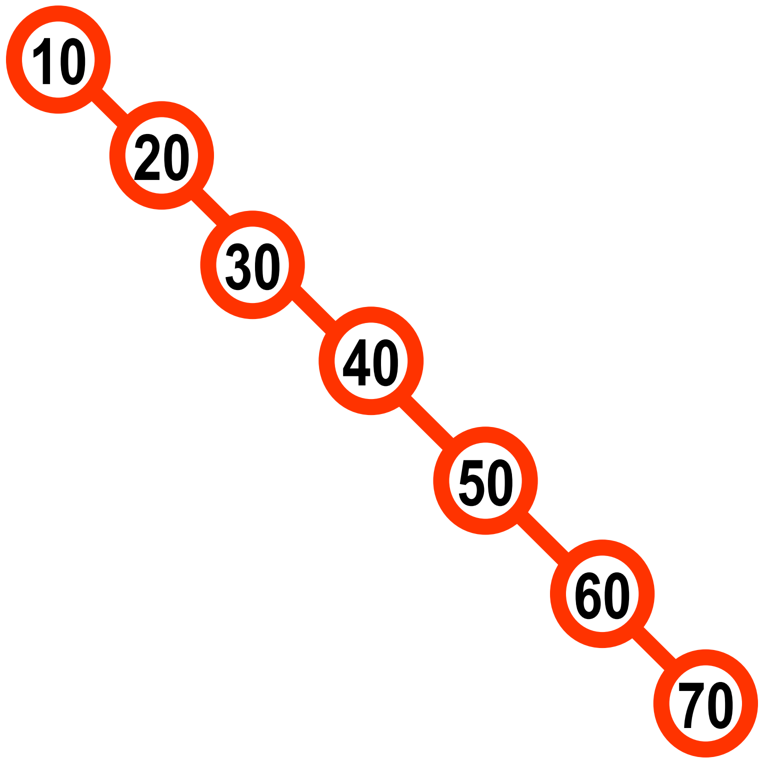
Example: AVL Tree

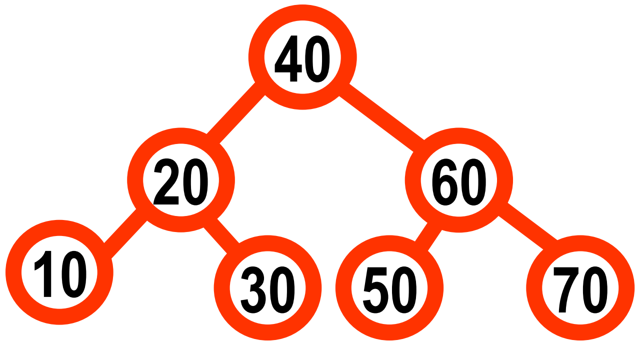
Insert 10, 20, 30, 40, 50, 60 and 70



BST vs AVL Tree

Insert 10, 20, 30, 40, 50, 60 and 70



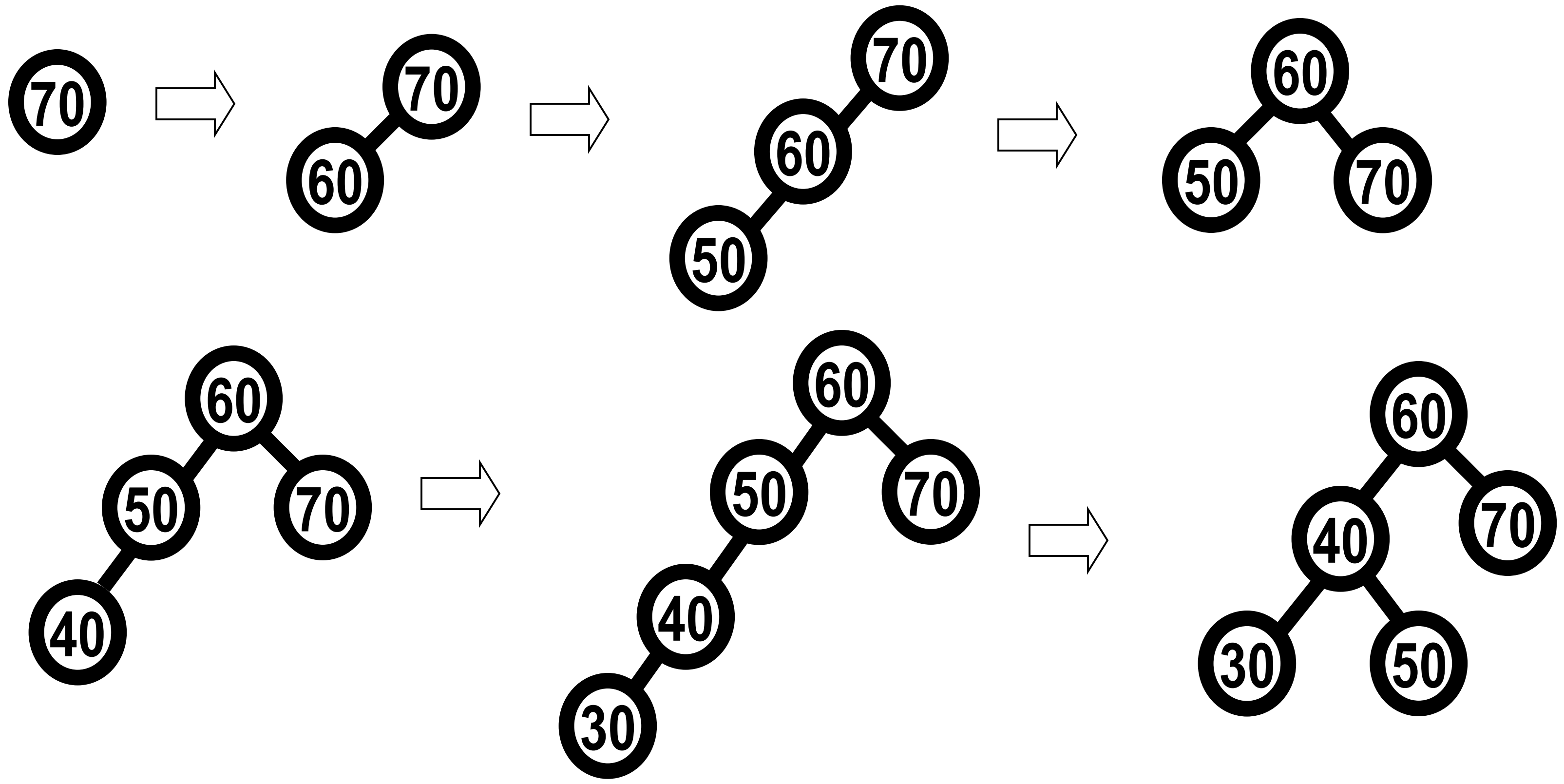


►QUIZ? AVL Tree?

Insert 70, 60, 50, 40, 30, 20 and 10

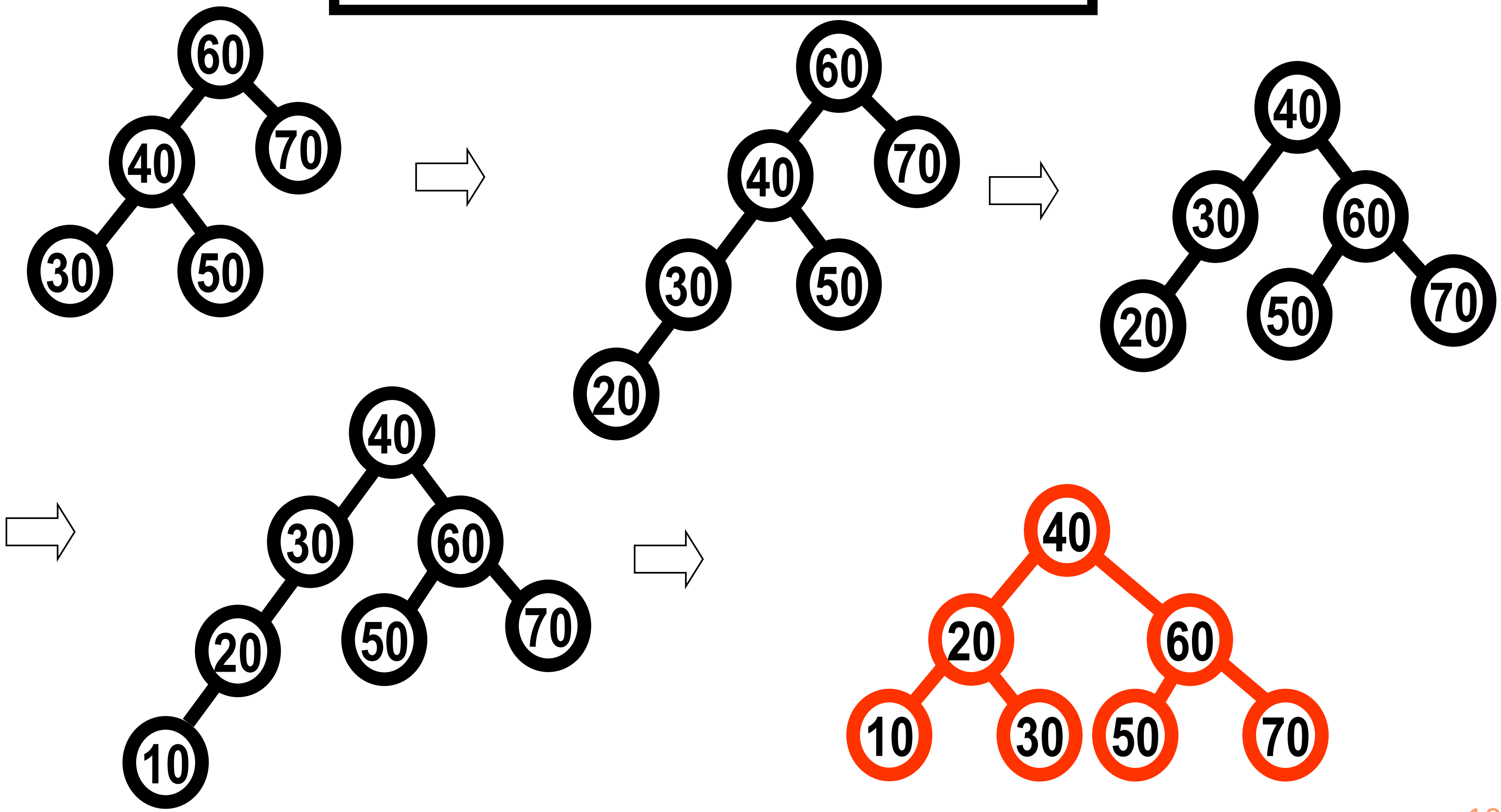
►QUIZ: AVL Tree

Insert 70, 60, 50, 40, 30, 20 and 10



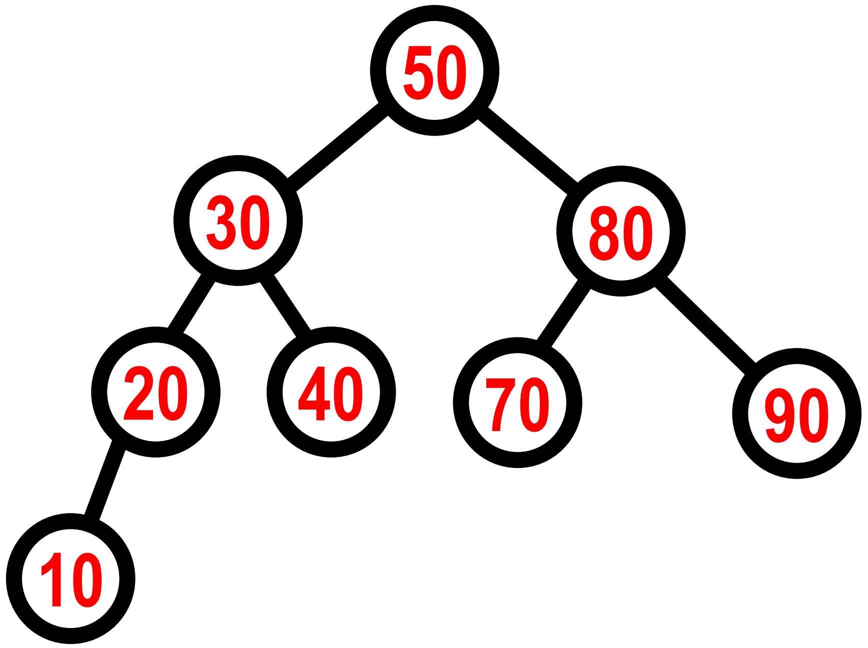
►QUIZ: AVL Tree

Insert 70, 60, 50, 40, 30, 20 and 10



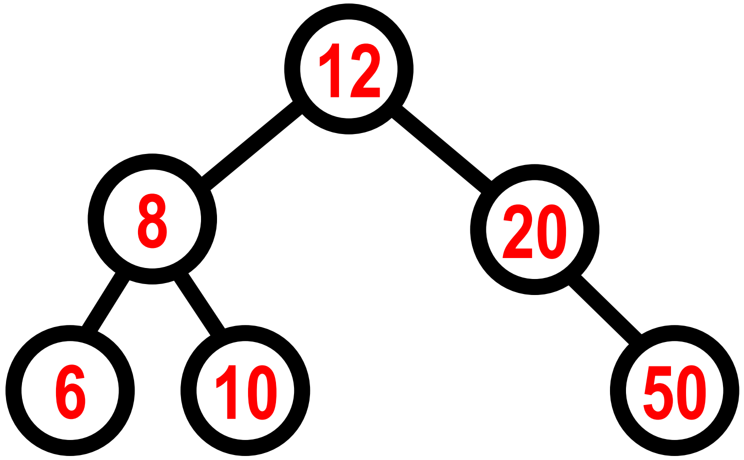
►QUIZ? AVL Tree?

Insert 50, 80, 90, 40, 20, 70, 30 and 10



►QUIZ? AVL Tree?

Insert 20, 8, 6, 12, 10 and 50



►QUIZ?

- **Inserting** an item into an AVL tree?
  - The item is inserted using the usual method of insertion in binary search trees.
  - Check the resulting tree is AVL balanced.
    - Check whether any node in the tree has left and right subtrees whose heights differ by more than 1.
  - If not, **balance it!**
    - Case 1: LL - Single rotation called LL rotation
    - Case 2: RR - Single rotation called RR rotation
    - Case 3: LR - Double rotation called LR rotation
    - Case 4: RL - Double rotation called RL rotation

Deleting Items from an AVL tree

- Similar to the Insertion operation.
- Deleting an item from an AVL tree is a two-part process:
  - The item is deleted using the usual method of deletion in binary search trees.
  - Check the resulting tree is AVL balanced.
    - Keep track of where a leaf node was (ultimately) removed and fix up the heights above that point.
  - If not, balance it using rotations.

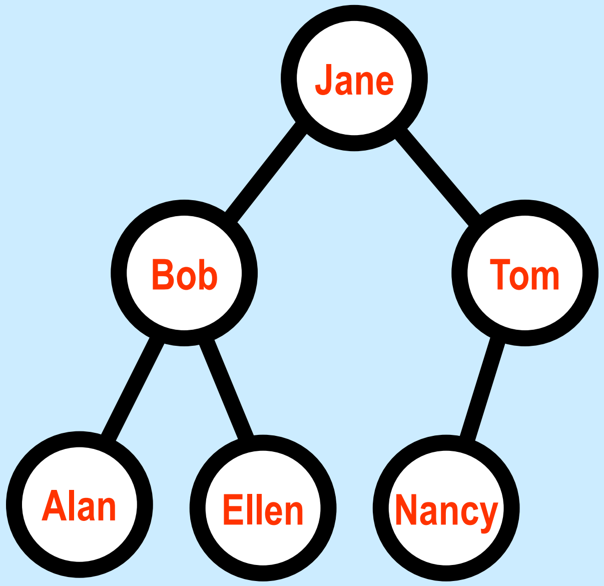
Rebalancing AVL trees after Deletion

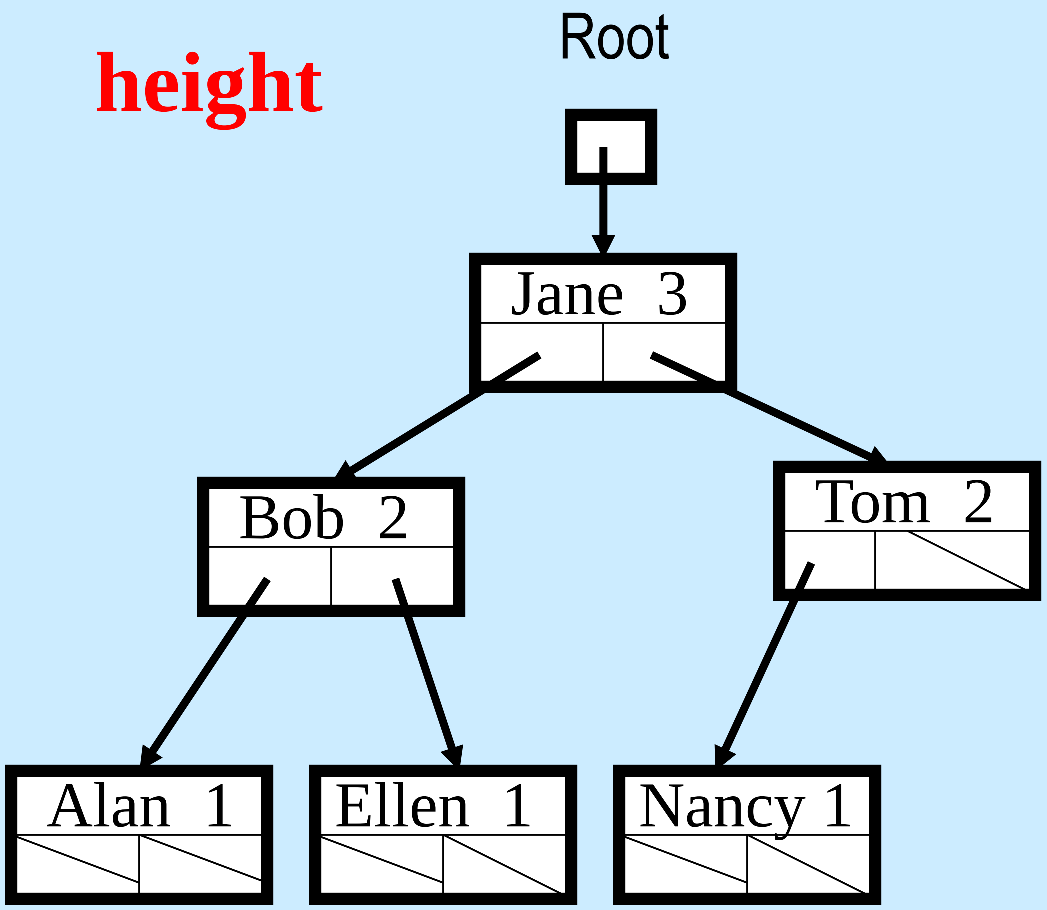
- **O(log N) rotations** (single or double)!

Representation of AVL Trees

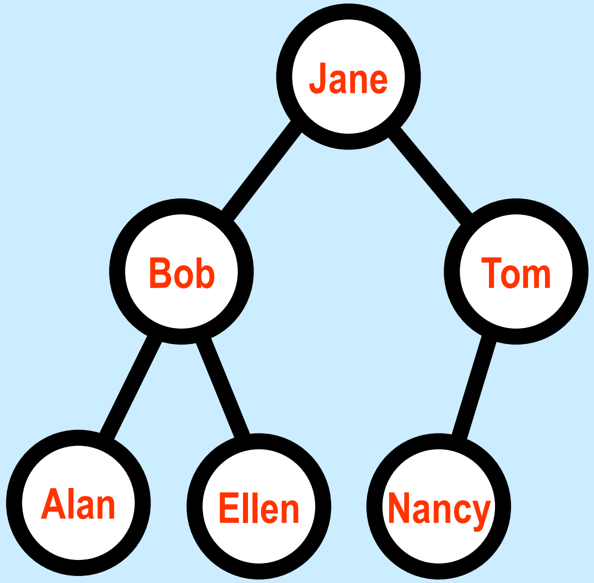
- Similar to representation of binary trees and binary search trees
    - ➔But, add **height** or **bf**.

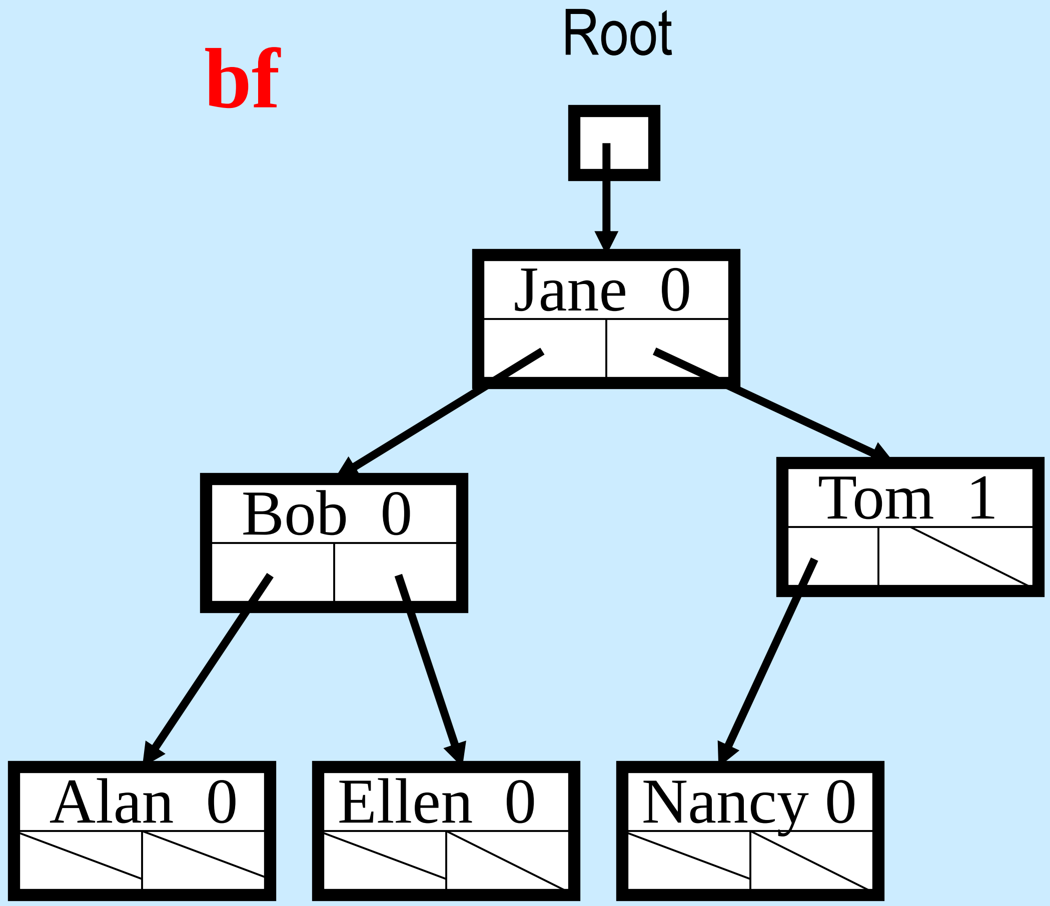
A Pointer-Based Representation of AVL Trees





A Pointer-Based Representation of AVL Trees





Implementation of AVL Trees

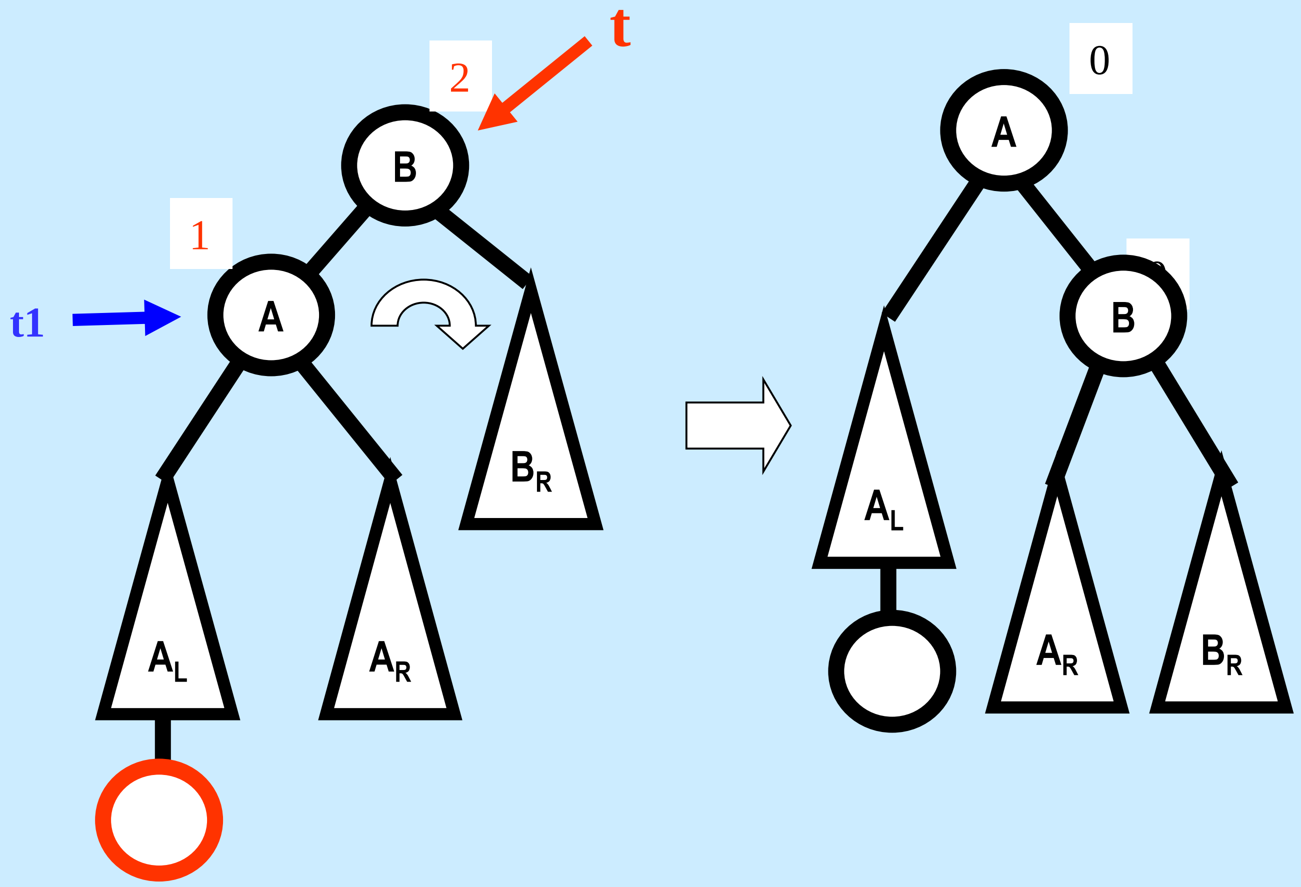
```
template <class DT>
class AvlTree;

template <class DT >
class AvlNode
{
    DT element;
    AvlNode   *left;
    AvlNode   *right;
    int   height;

    AvlNode( const DT & theElement, AvlNode *lt, AvlNode *rt, int h = 1 )
      : element( theElement ), left( lt ), right( rt ), height( h ) { }

    friend class AvlTree<DT>;
};
```

Implementation of AVL Trees

```
template <class DT>
class AvlTree
{
  public:
    AvlTree();
    AvlTree( const AvlTree & rhs );
    ~AvlTree( );
    const AvlTree & operator=( const AvlTree & rhs );

    const DT & search( const DT & x ) const;
    void printAVLTree( ) const;
    void insert( const DT & x );
    void delete( const DT & x );

  private:
    AvlNode<DT> *root;
    …

};
```

Single Rotation (LL)

LL Case:



Single LL Rotation

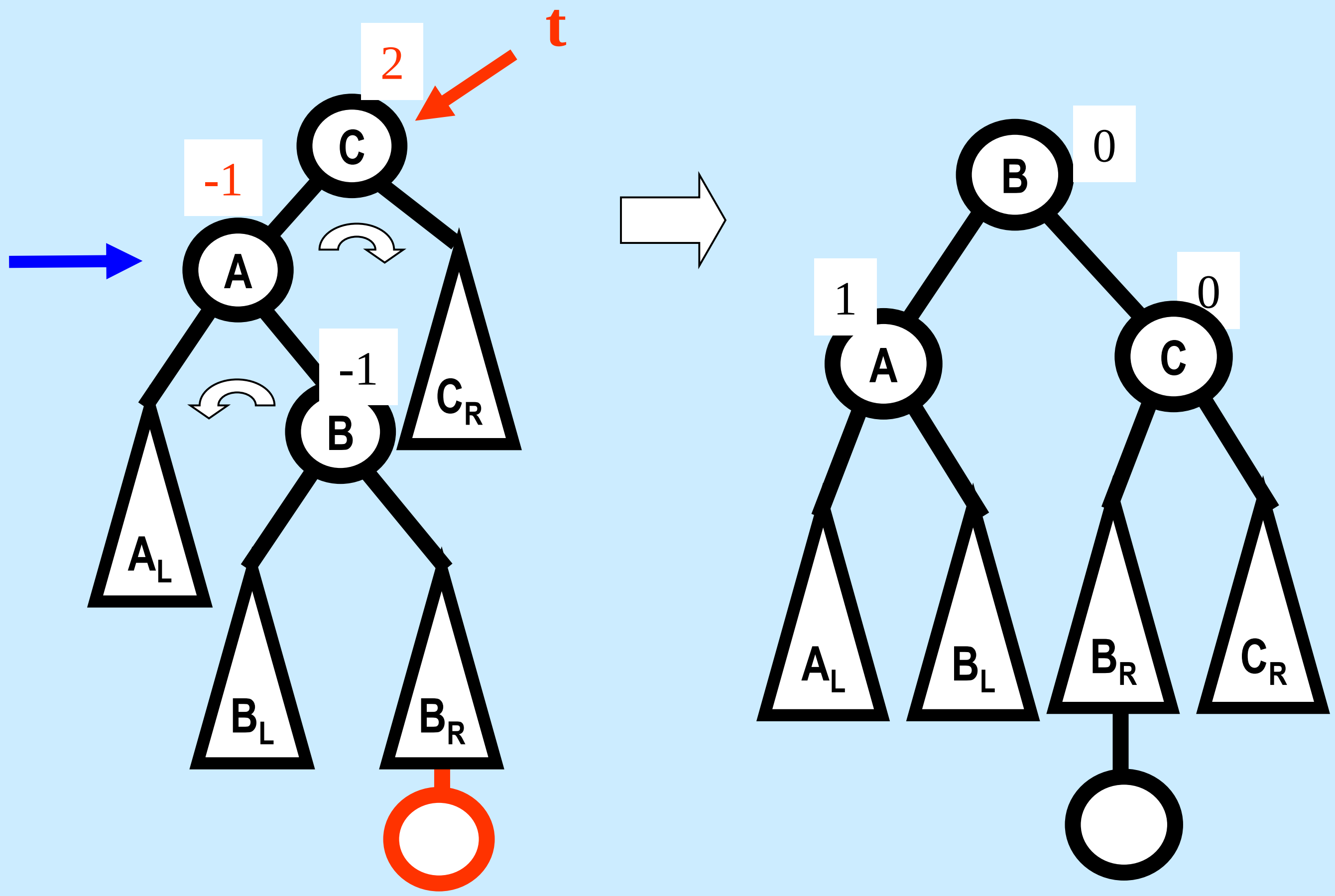
```
template <class DT>
void AvlTree<DT>::
        LLrotateWithLeftChild( AvlNode<DT> * & t ) const
{
    AvlNode<DT> *t1 = t->left;
    // Single Rotate to the right.
    t->left = t1->right;
    t1->right = t;
    // Update height.
    t->height = max( height( t->left ), height( t->right ) ) + 1;
    t1->height = max( height( t1->left ), height( t1->left ) ) + 1;
    // Set new root
    t = t1;
}
```

Single RR Rotation

```
template <class DT>
void AvlTree<DT>::
        RRrotateWithRightChild( AvlNode<DT> * & t ) const
{









}
```

Double Rotation (LR)

LR Case:



Double LR Rotation

```
template <class DT>
void AvlTree<DT>::
LRdoubleWithLeftChild( AvlNode<DT> * &t ) const
{
    // Single Rotate to the left.
    RRrotateWithRightChild( t->left );

    // Single Rotate to the right.
    LLrotateWithLeftChild( t );
}
```

Double RL Rotation

```
template <class DT>
void AvlTree<DT>::
RLdoubleWithRightChild( AvlNode<DT> * &t ) const
{




}
```

Analysis of AVL Tree Operations

- The number of comparisons for a search/retrieval, insertion or deletion is
  - ➔The level (depth) of the element in the AVL tree.
- The maximum number of comparisons for a search/retrieval, insertion or deletion is
  - ➔The height of the AVL tree!

Properties of AVL Trees

- What is the **minimum** number of nodes that an AVL tree of height h can have?
  - ➔ N_h =
    - ☞ 1 if h = 1
    - ☞ 2 if h = 2
    - ☞ $N_{h-1} + N_{h-2} + 1$ if h > 2
    - ☞ 1, 2, 4, 7, 12, 20, 33 ...

Fibonacci Numbers

- F_0 F_1 F_2 … F_n
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...
- F_n =
  - 0 if n = 0
  - 1 if n = 1
  - $F_{n-1} + F_{n-2}$ if n >= 2

Properties of AVL Trees

- What is the **maximum** number of nodes that an AVL tree of height h can have?
  - ➔ **$2^h - 1$**

Properties of AVL Trees

- N= The number of nodes in an AVL tree.
- h = The height of an AVL tree.

➔ $F_{h+2} - 1 \leq N \leq 2^h - 1$

➔ $\log (N+1) \leq h \leq\approx 1.440 \log (N+2) -0.328$

➔ Lower Bound: $h = \Omega (\log N)$

➔ Upper Bound: $h = O (\log N)$

➔ $h = \Theta (\log N)$

Properties of AVL Trees

The height of an AVL tree with n nodes is $\Theta(\log N)$.

AVL Tree Operations -Analysis

- Rotation – Single & Double
  - ➔**O(1)**
- Search, Insert & Delete
  - ➔**O(log N)** worst-case time !!!

►QUIZ?

- **Rotate, DoubleRotate, Insert, Search & Delete** Worst-case time complexity of AVL trees?

AVL Tree History

- Georgy Adelson-Velsky & Evgenii Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian). 146: 263–266.

*** AVL = Adelson-Velsky & Landis**

AVL Tree Visualization

- ***AVL Tree Visualization***

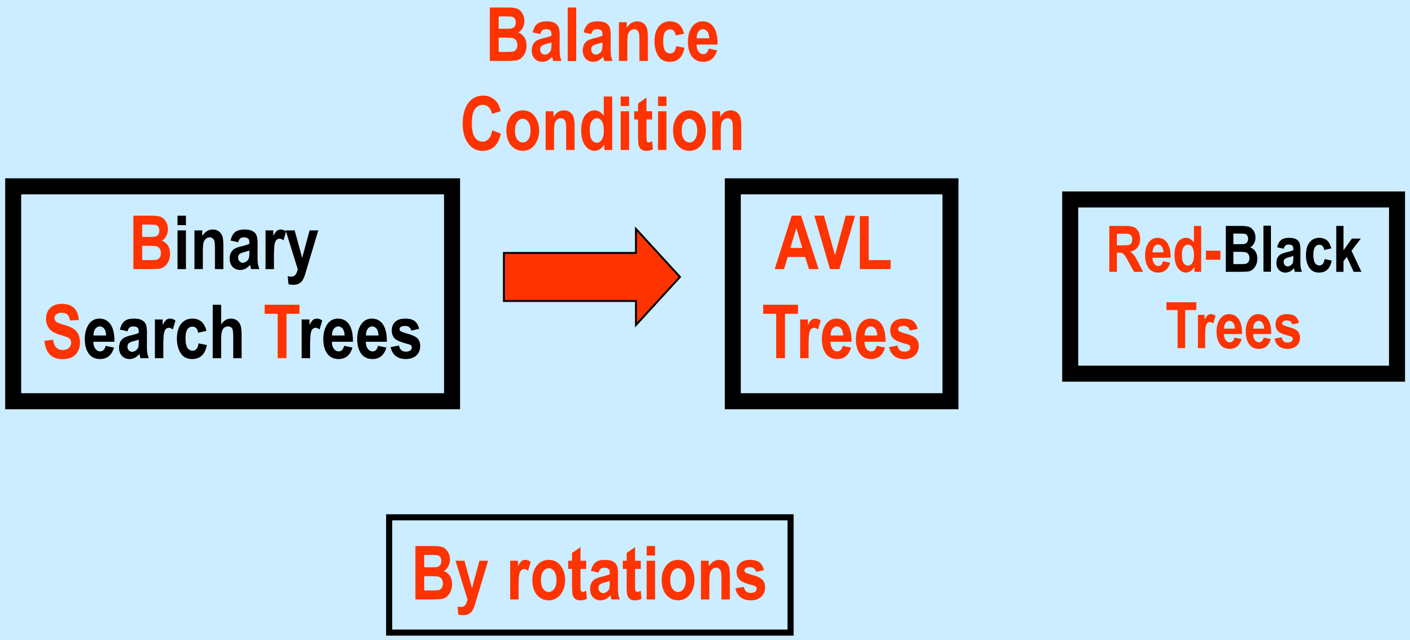
Red-Black Trees

(Height-Balanced Binary Search Trees)

Red Black (Search) Tree

- A **red-black tree** is a **binary search tree** st.
  - Every node is colored either red or black.
  - The root is always black.
  - There are no two adjacent red nodes. If a node is red, then both its children must be black.
  - Every path from root to a NULL node has **the same number of black nodes**.

BSTs vs AVL Trees vs Red-Black Trees



Self-Adjusting Binary Search Trees

Self-Adjusting BSTrees?

- Observation:
  - ➔Not all elements are used with the same frequency!!!
  - ➔Data accessed once, is often soon accessed again!
- Idea?
  - ➔Restructure the tree by moving up the tree those elements that are used more often!!!
  - ➔**Rotation!**

Self-Adjusting Approaches

- To improve the **amortized** **(average time over operations)** **time** !
  - ➔Move one-level up via one single rotation
  - ➔Moving to the root via several single rotations

Splay Trees

(Self-adjusting Binary Search Trees)

Splay (Search) Trees

- **Self-adjusting Binary Search Trees**

- Why?
  - ➔ To improve the **amortized** **(average time over operations)** **time** of **search** !

Splay Trees

- How?
- **Blind adjusting version of AVL trees**
  - Why worry about balances?
  - Just rotate anyway for future operations!
  - **Moving to the root** via **rotations!**
  - Data accessed once, is often soon accessed again!

Search for Items in a Splay Tree

- **Search** similar to BST.
- Percolate the node to the root with rotations.
  - ➔ “**splay a node to the root**”

  - ➔ If the search is successful, then the node that is found is splayed and becomes the new root.
  - ➔ Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

Inserting Items into a Splay Tree

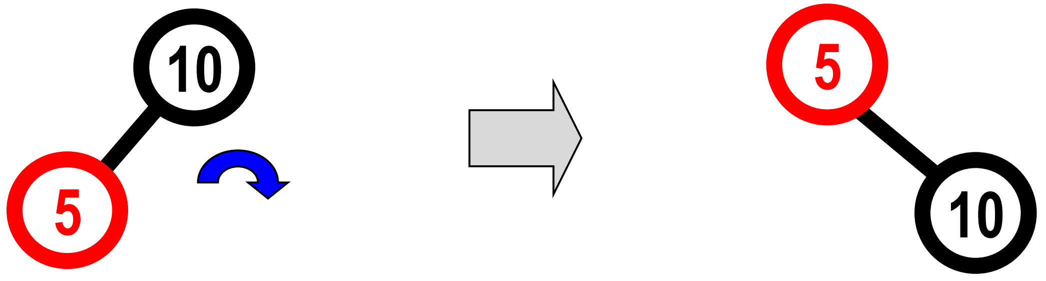
- **Insert** a node.
- Percolate the node to the root with rotations.
  - ➔"**splay a node to the root**"

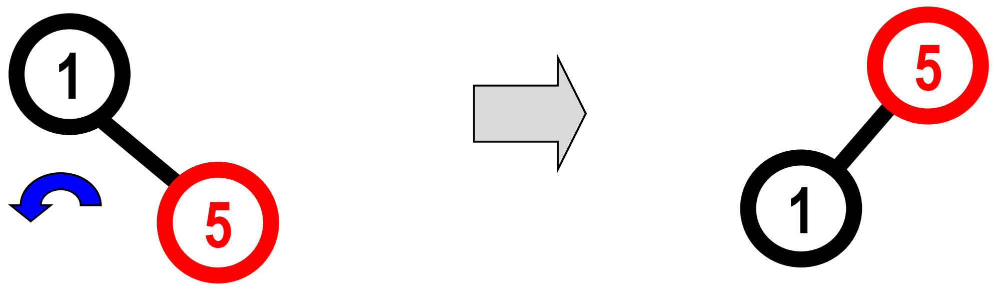
Splaying the Node to the Root

- Percolate the node to the root with rotations – **Splaying!**
- Two cases:
  1. The node has **no grandparent.**
  2. The node has **a grandparent.**

Splaying the Node to the Root

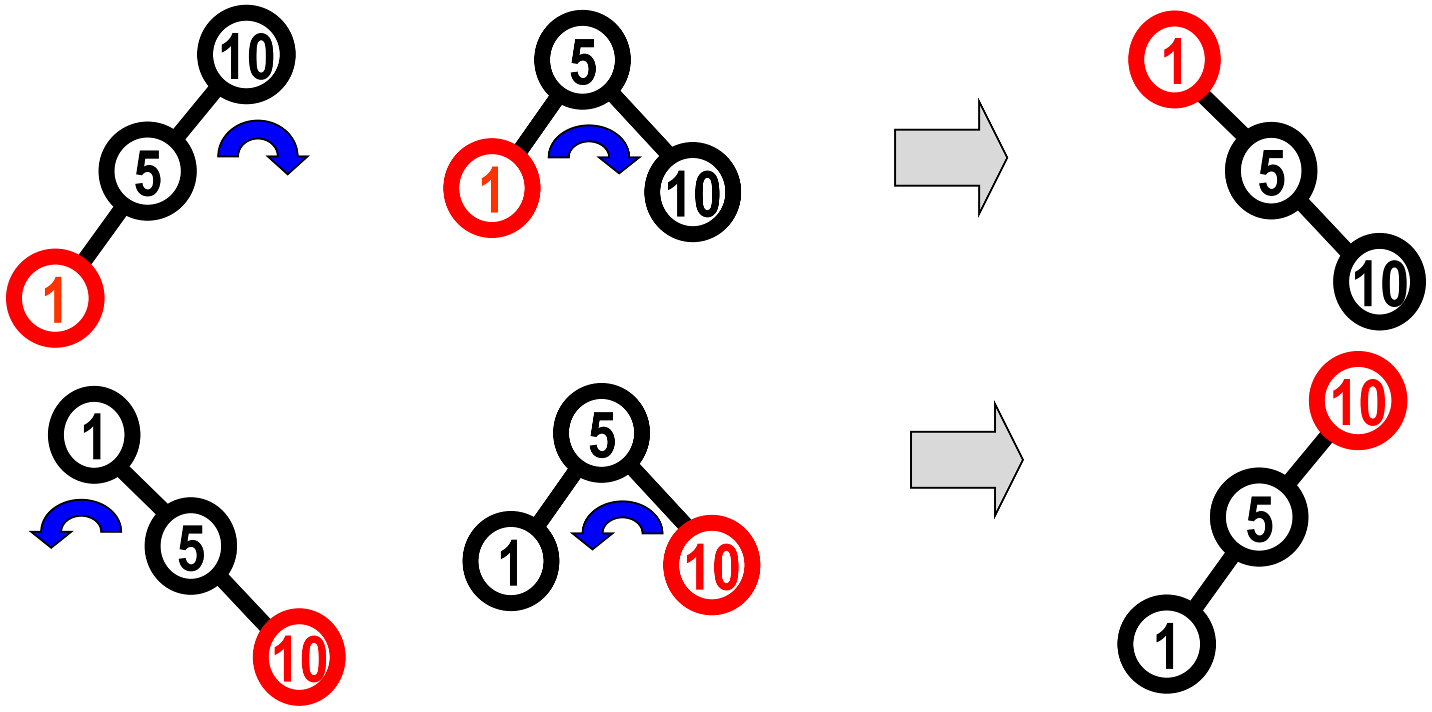
1. Cases: (The node has **no grandparent**)
   - **Zig (L):** Rotate to the right
   - **Zag (R):** Rotate to the left





Splaying the Node to the Root

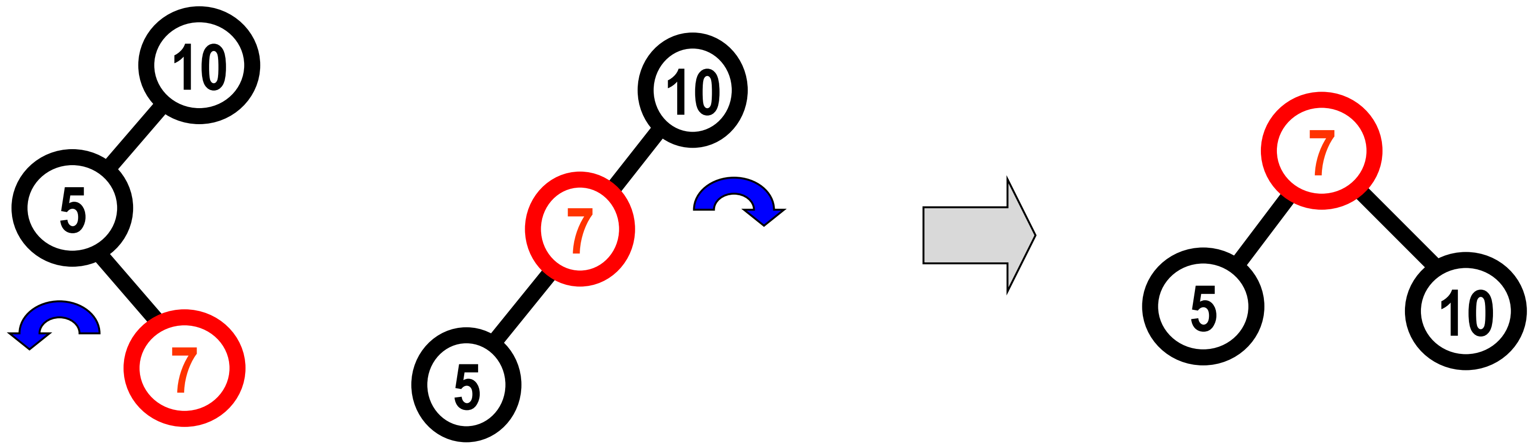
2. Cases: (The node has **a grandparent**)
   - **Zig-Zig (LL):** Rotate to the right + Rotate to the right
   - **Zag-Zag (RR):** Rotate to the left + Rotate to the left



Splaying the Node to the Root

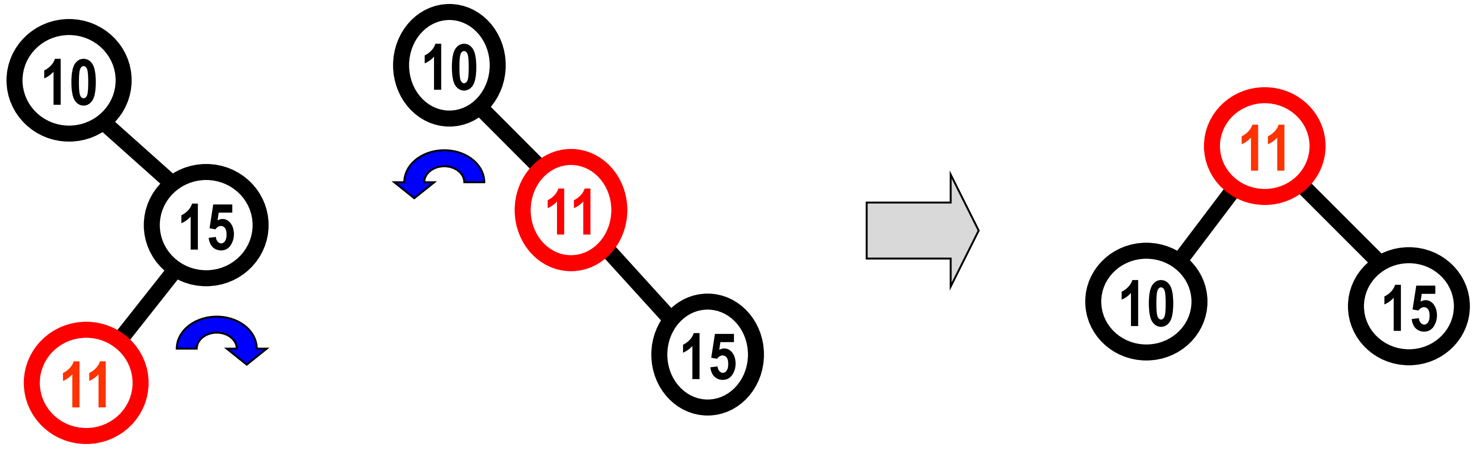
2. Cases: (The node has **a grandparent**)

➔ **Zig-Zag (LR):** Rotate to the left + Rotate to the right



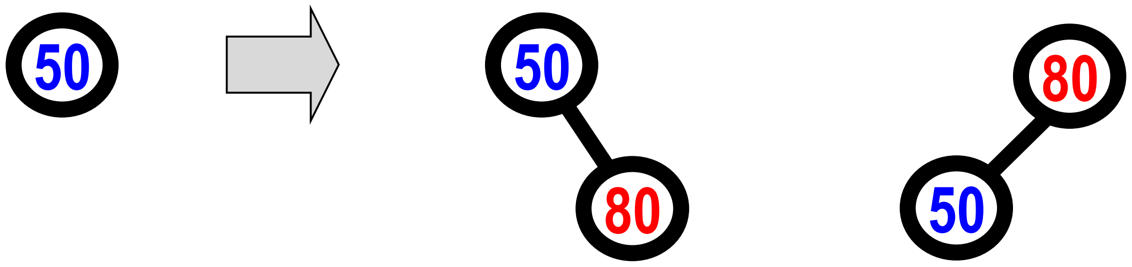
Splaying the Node to the Root

2. Cases: (The node has **a grandparent**)
   - ➔ **Zag-Zig (RL):** Rotate to the right + Rotate to the left



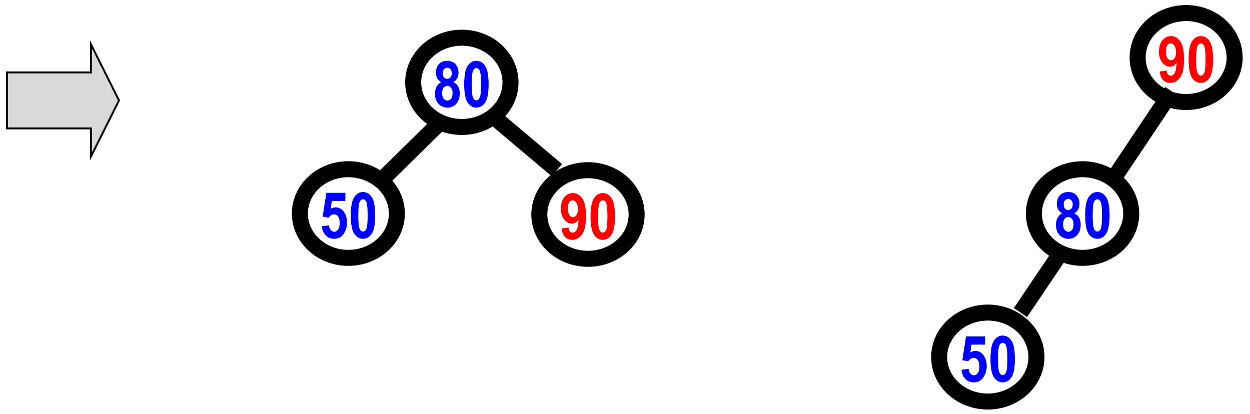
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



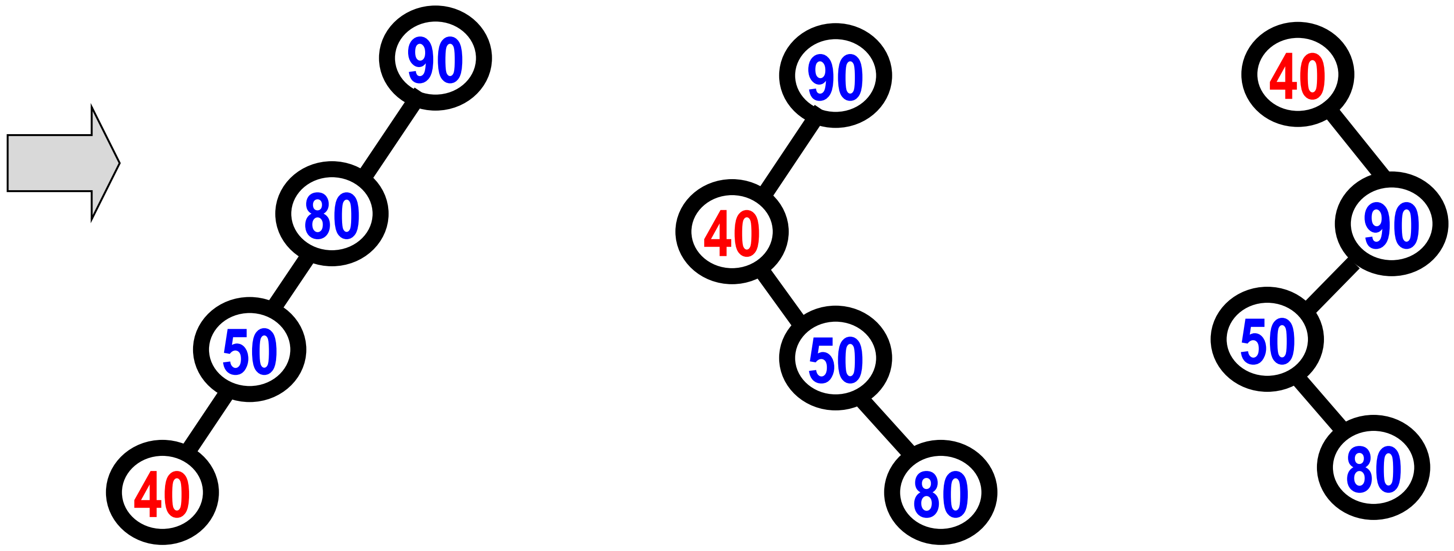
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



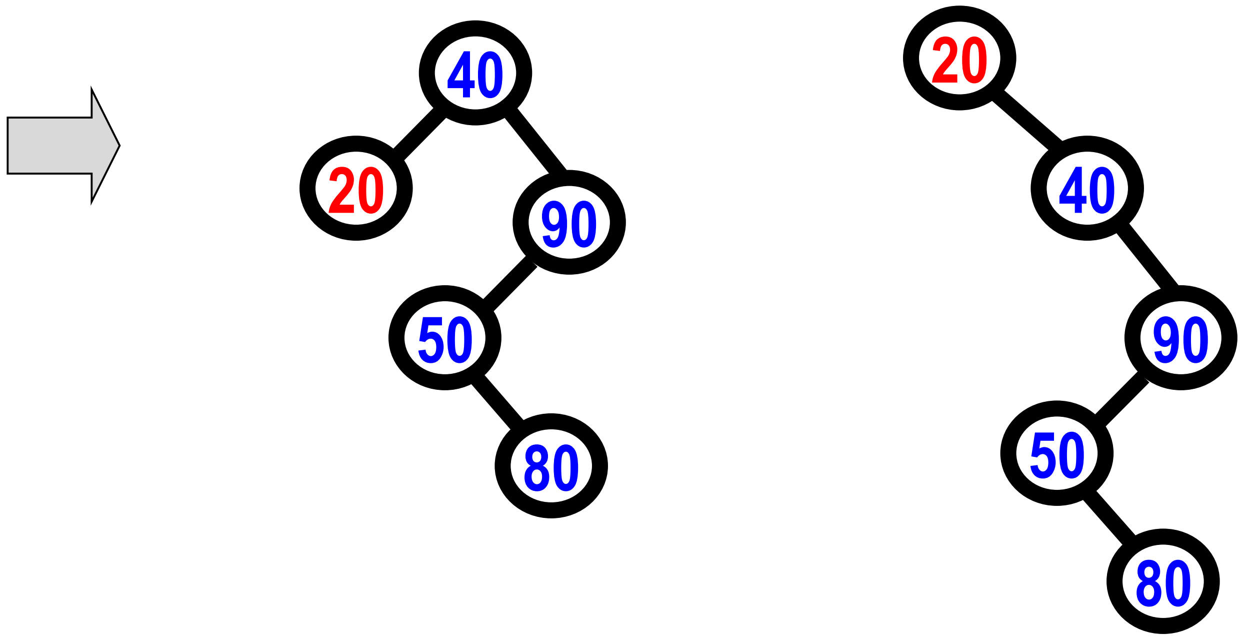
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



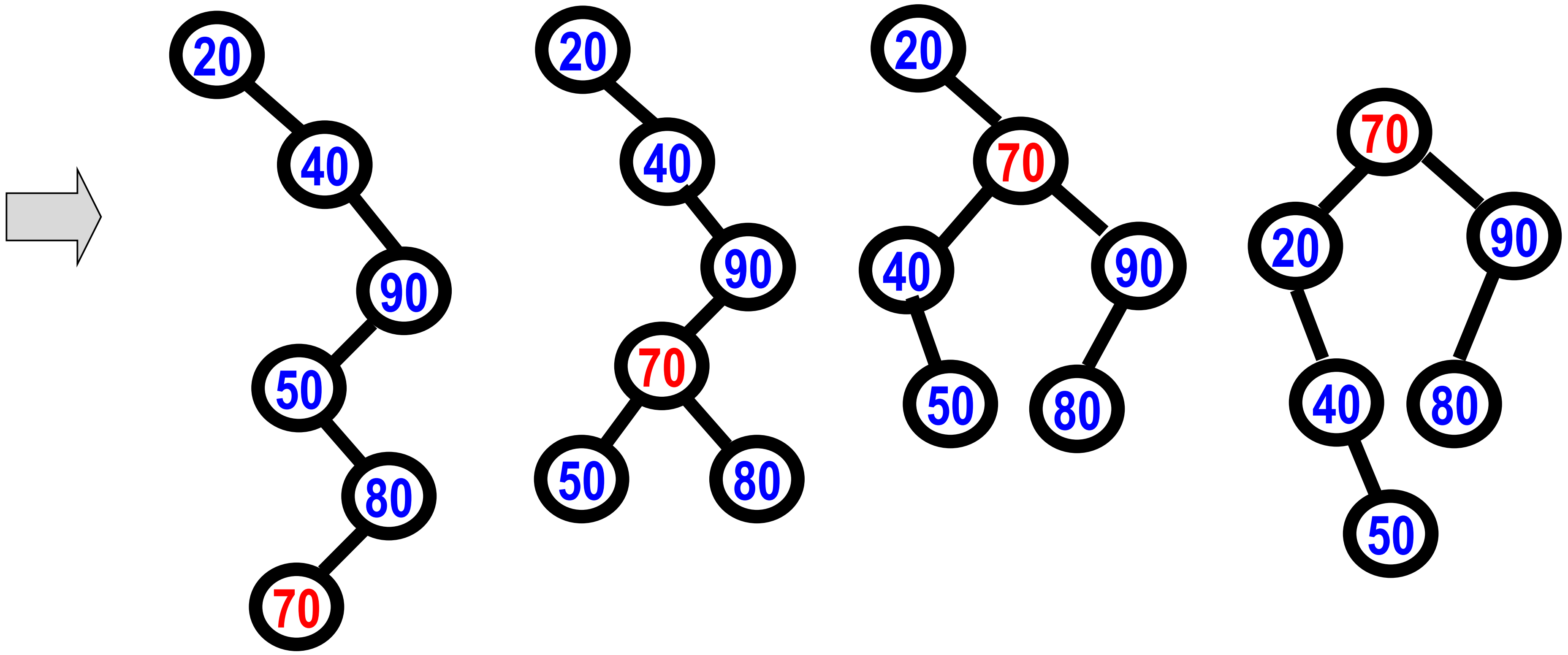
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



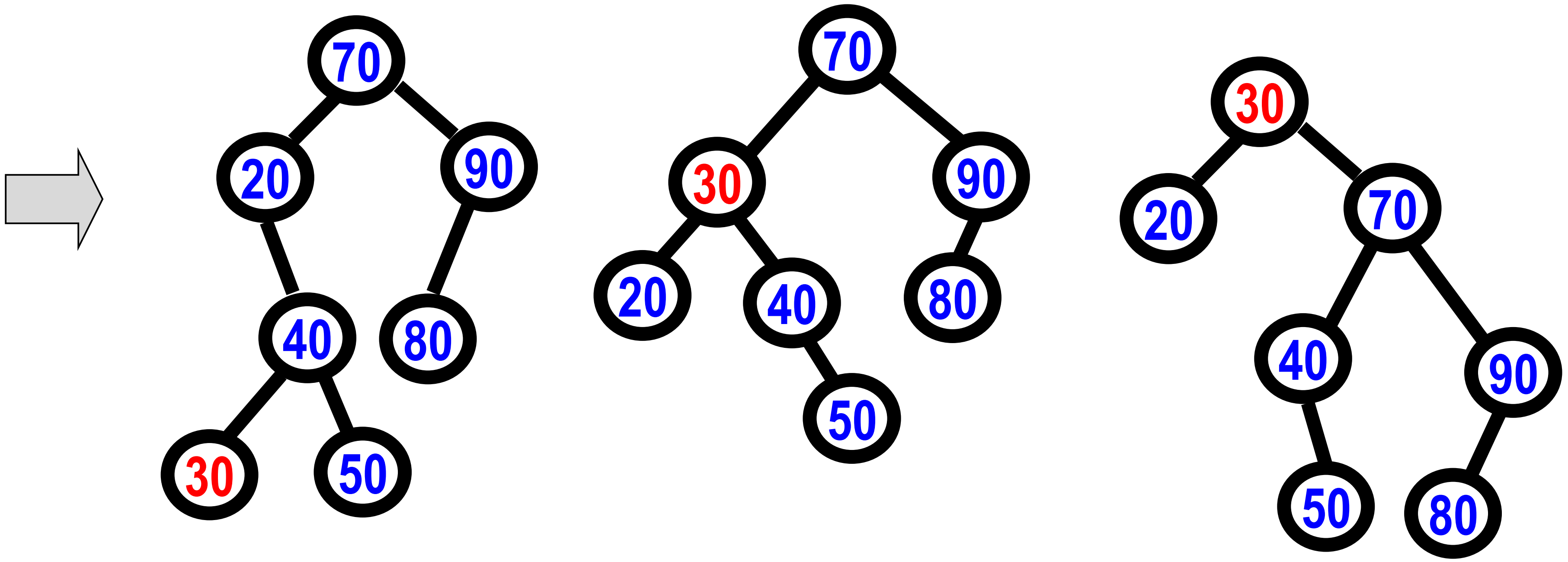
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



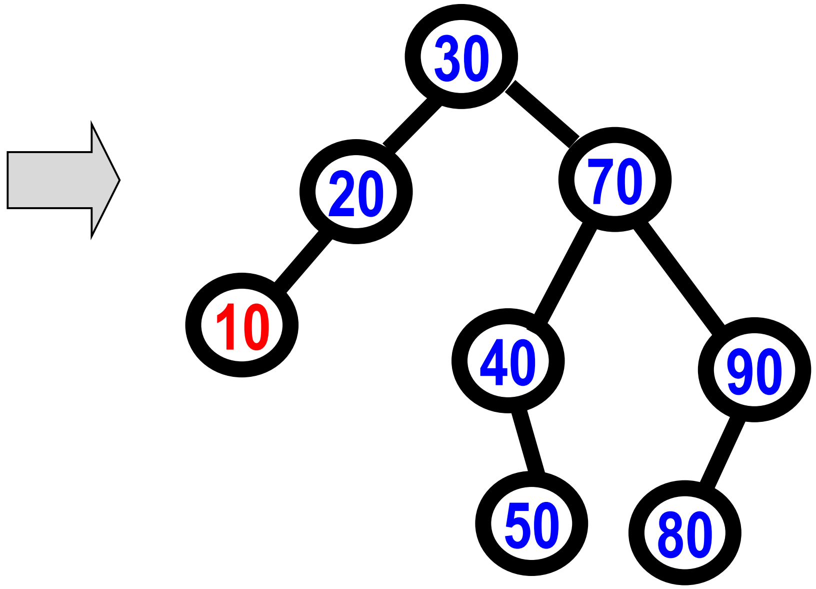
Example: Splay Tree

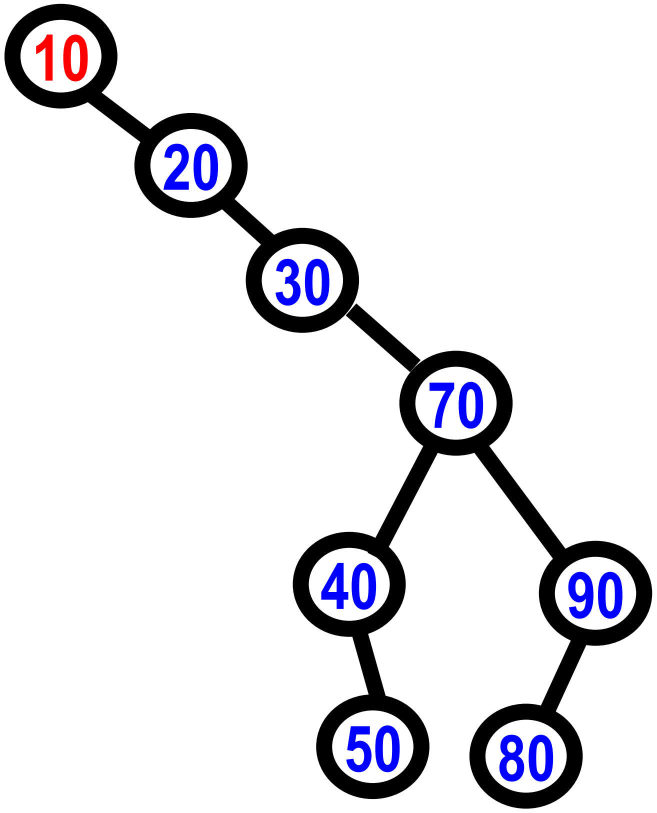
Insert 50, 80, 90, 40, 20, 70, 30 and 10



Example: Splay Tree

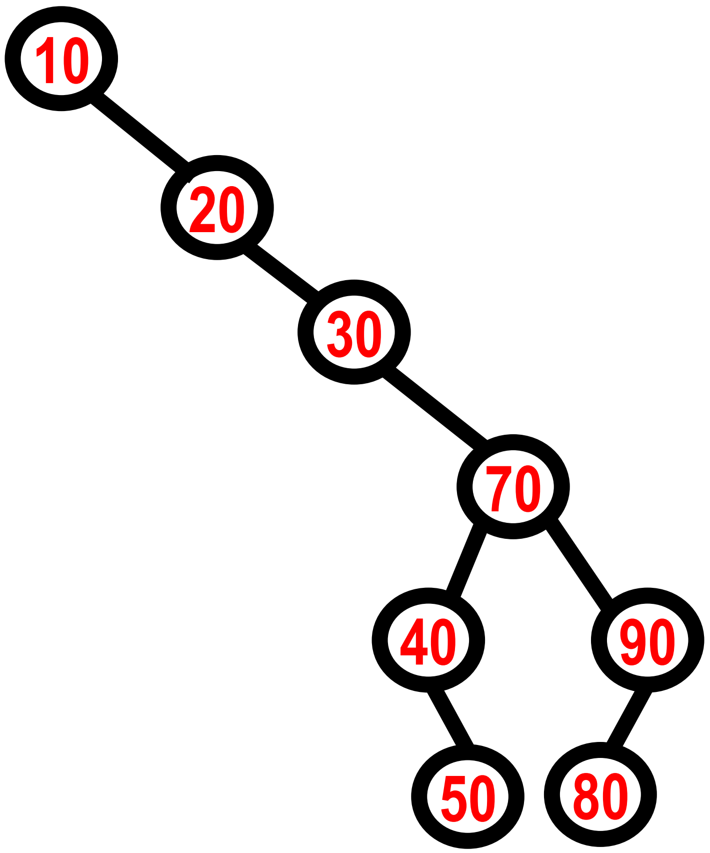
Insert 50, 80, 90, 40, 20, 70, 30 and 10





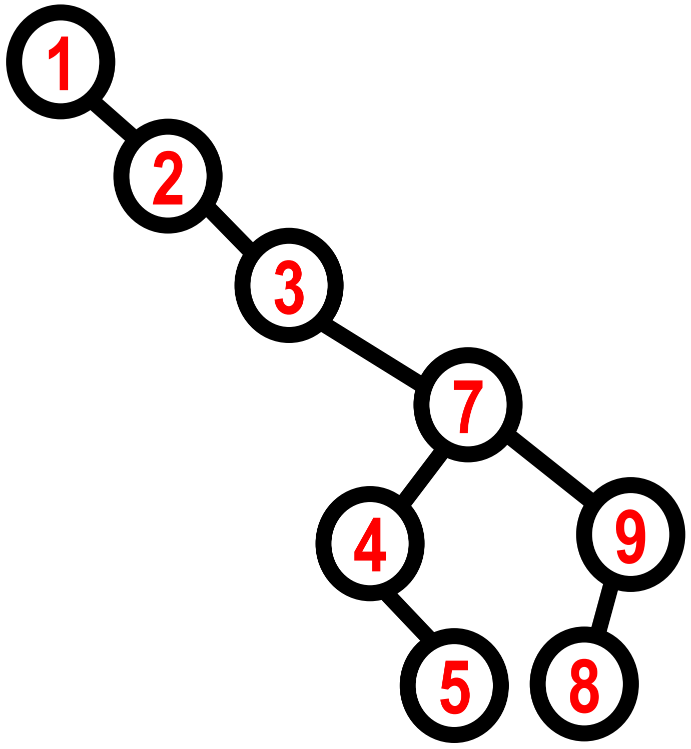
Example: Splay Tree

Insert 50, 80, 90, 40, 20, 70, 30 and 10



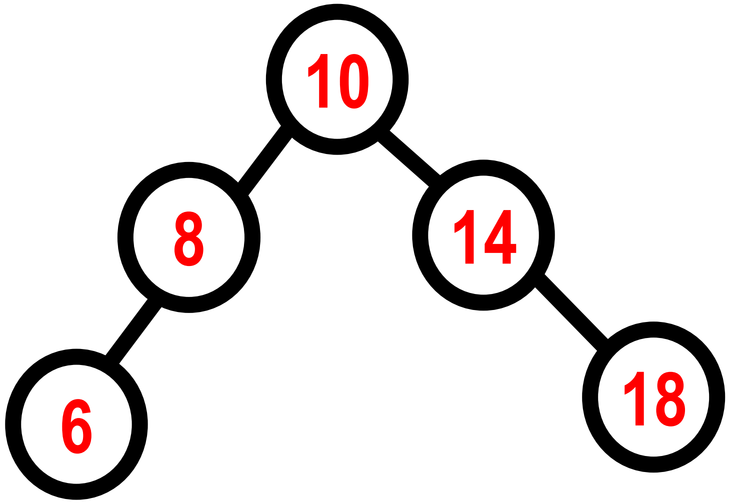
►QUIZ? SplayTree?

Insert 5, 8, 9, 4, 2, 7, 3 and 1



►QUIZ? SplayTree?

Insert 8, 18, 6, 14 and 10



►QUIZ?

- Inserting Items into a Splay Tree?
  - ➔**Insert** a node.
  - ➔Percolate the node to the root with rotations.
    - ☞“**splay a node to the root**”

Splay Trees

- Worst case time for an operation
  - ➔**O(n)**
- **Amortized time** per operation
  - ➔**O(log n)**

Splay Tree History

- Sleator, Daniel D. & Tarjan, Robert E. (1985). "Self-Adjusting Binary Search Trees". Journal of the ACM. 32 (3): 652–686.

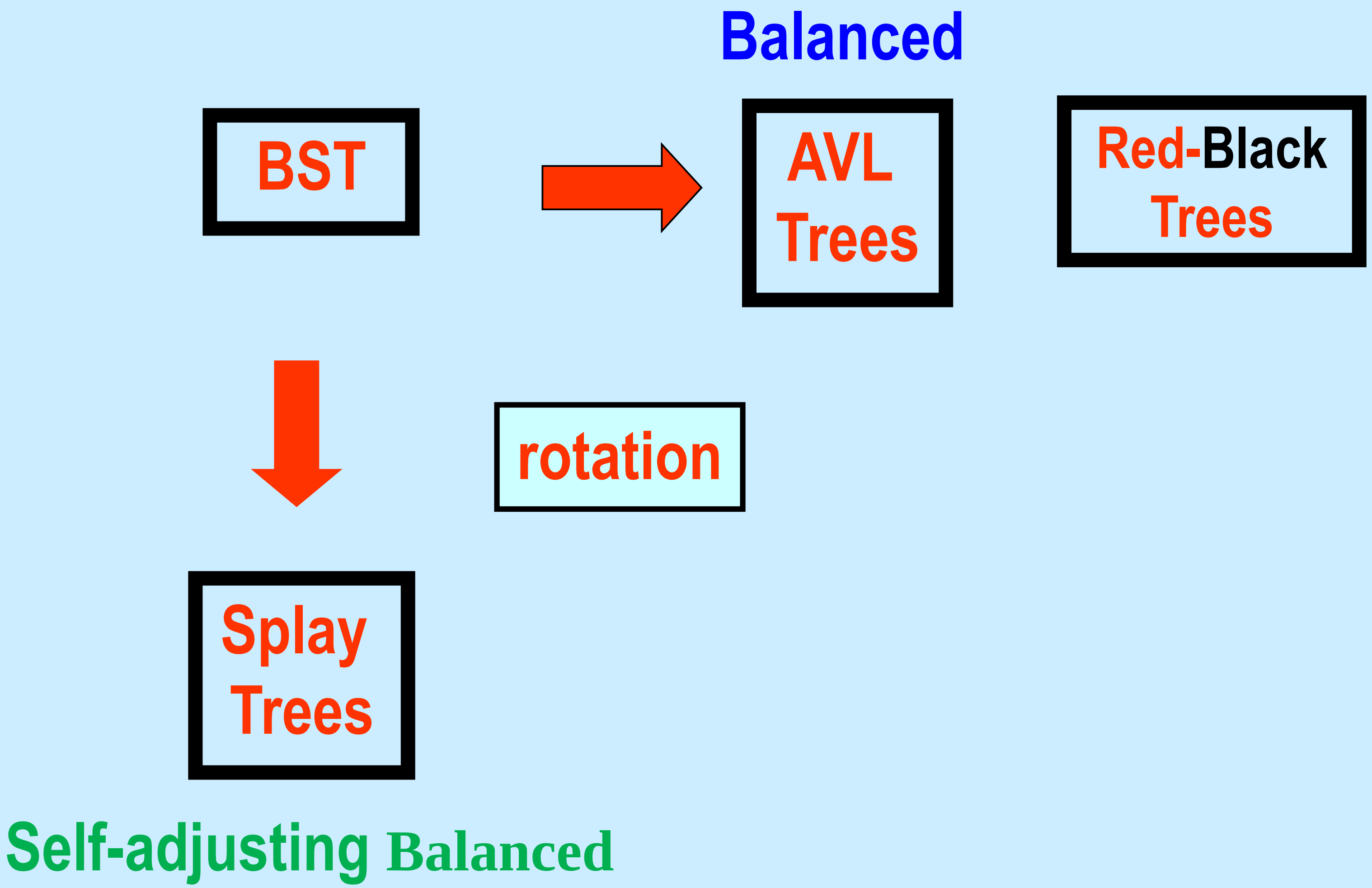
Splay Tree Visualization

- ***Splay Tree Visualization***

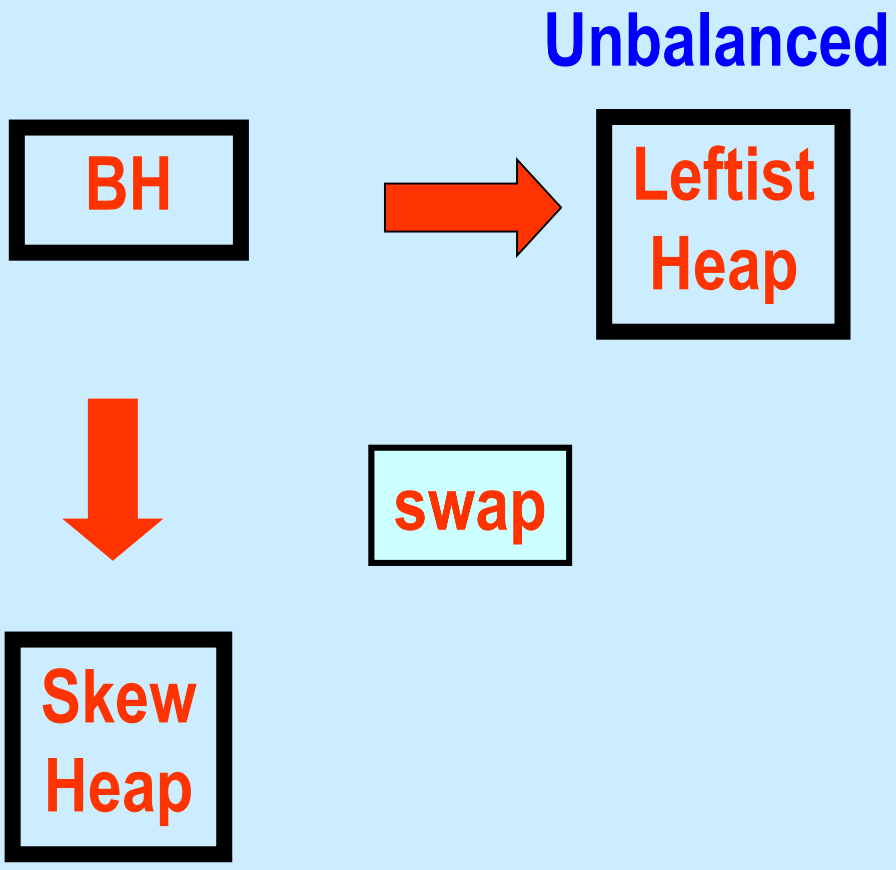
►QUIZ? AVL Trees vs Splay Trees?

- Compare **AVL** vs **Splay**?

BSTs, AVL Trees & Splay Trees

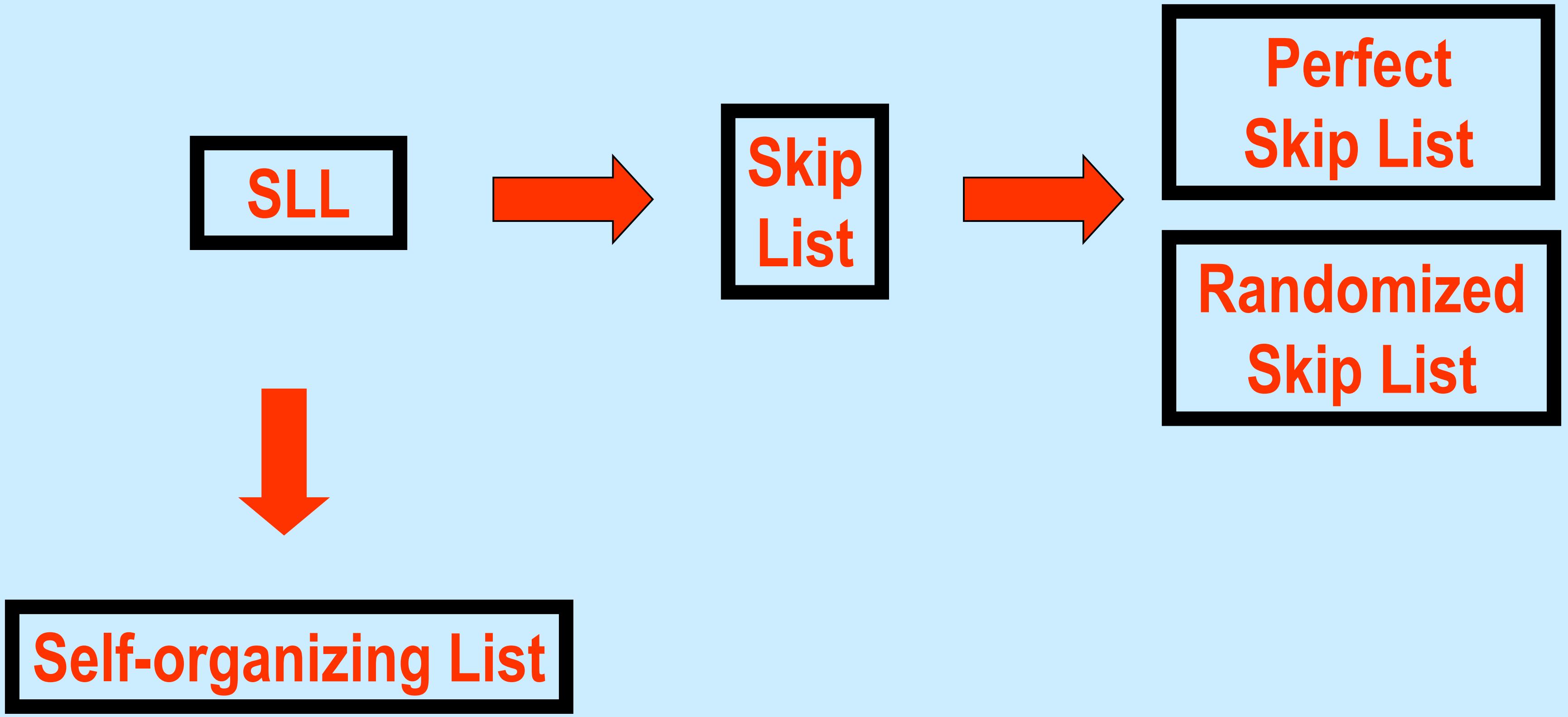


Binary Heaps, Leftist Heaps & Skew Heaps



Self-adjusting Unbalanced

SLLs, Skip Lists & Self-organizing Lists



Self-adjusting

►QUIZ? Splay Trees vs Skew Heaps?

- BSTs
- AVL Trees
- Splay Trees

- Binary Heaps
- Leftist Heaps
- Skew Heaps

►QUIZ? Splay Trees vs Skew Heaps vs Self-Organizing Lists?

- Binary Search Trees
- AVL Trees
- Splay Trees

- Binary Heaps
- Leftist Heaps
- Skew Heaps

- Singly-Linked Lists
- Skip Lists
- Self-Organizing Lists

Homework Assignment

►Homework Assignment?

- Draw the **AVL tree** that results when you insert items with the keys 4, 10, 3, 6, 5 and 25 in that order into an initially empty AVL tree.
- Draw the **splay tree** that results when you insert items with the keys 4, 9, 3, 7, 5 and 6 in that order into an initially empty splay tree.

►Homework Assignment?

- ***Advanced Binary Search Tree*: <u>The AVL Search Tree</u>**
- Design and implement the AVL Search Tree (AvlST) ADT.
- You should include ***insert*, *search*, *showAvlST*** and ***showBF***.
  - ➔ The operation ***showAvlST*** prints the keys in the AVL search tree as rotated counterclockwise 90 degrees from its conventional orientation using a "reverse" inorder tree traversal.
  - ➔ The operation ***showBF*** prints the balance factors in the AVL search tree as rotated counterclockwise 90 degrees from its conventional orientation using a "reverse" inorder tree traversal.

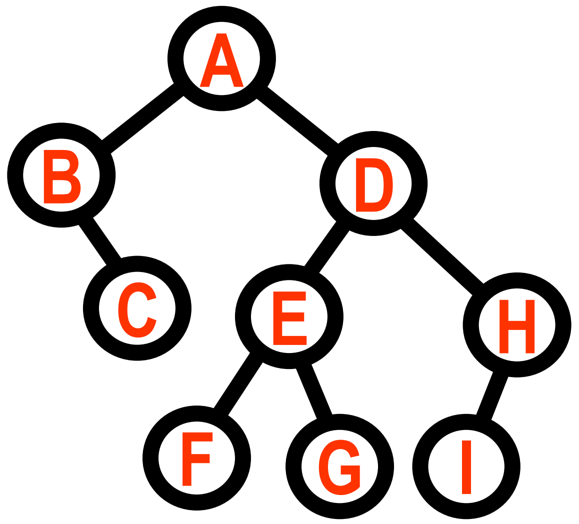
Test Cases

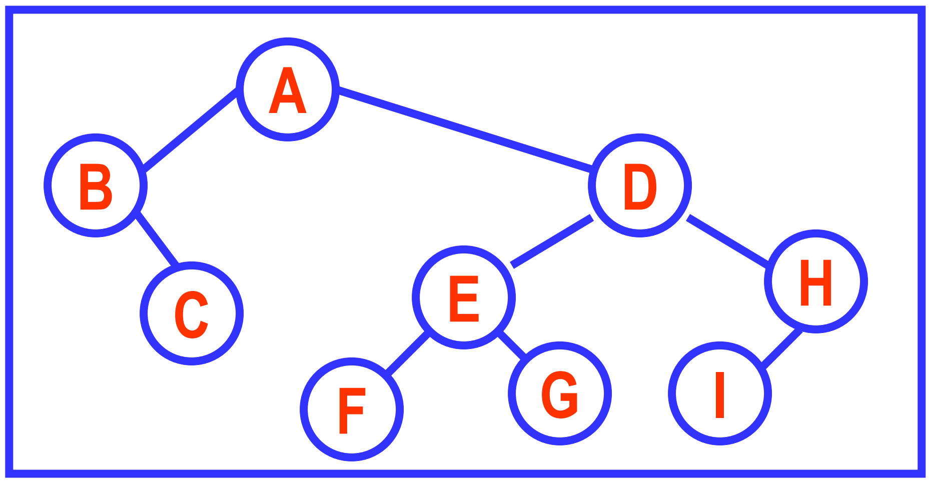
- **Insert 4, 10, 3, 6, 5 and 25 into an empty AVL tree.**
- **Insert 10, 20, 30, 40, 50, 60 & 70 into an empty AVL tree.**

Printing AVL Search Trees?

- **AVL Search Trees & Splay Search Trees** are special Binary Trees!

Printing Binary Trees?



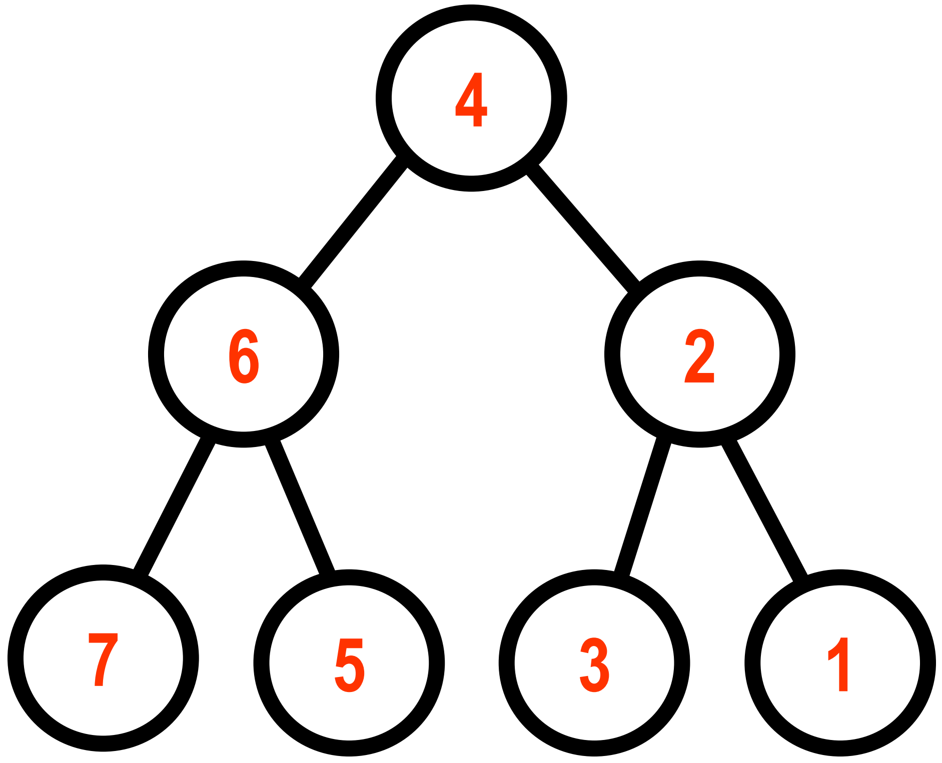


- Idea?
  - Backward In-order Traversal & Print

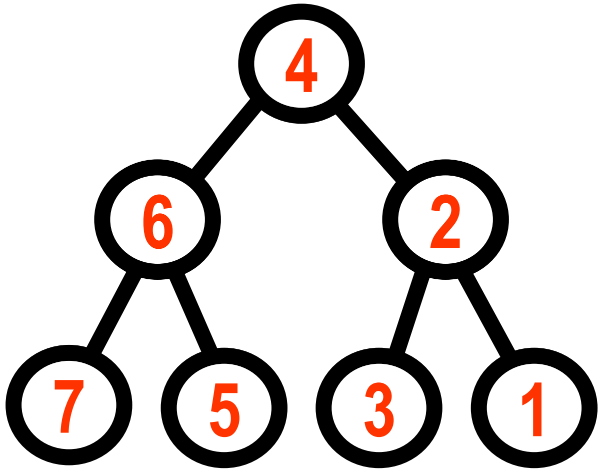
Backward In-order Traversal of Binary Trees

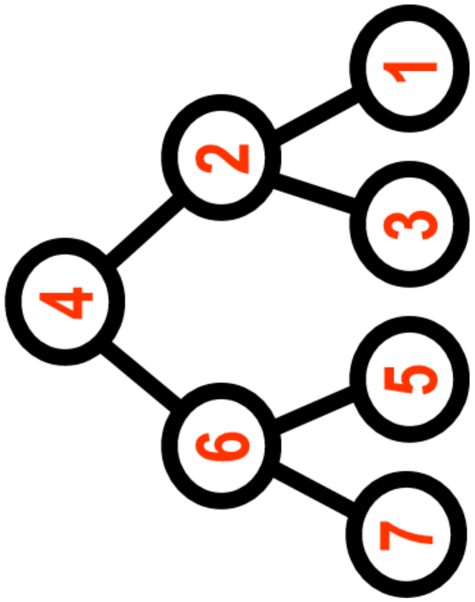
- Process right-child subtree, then a node, then left-child subtree.
- If the tree is not empty then
  - ➔Backward Inorder traverse the right subtree recursively.
  - ➔Visit the root & process!
  - ➔Backward Inorder traverse the left subtree recursively.

Backward In-order Traversal - Processing Order

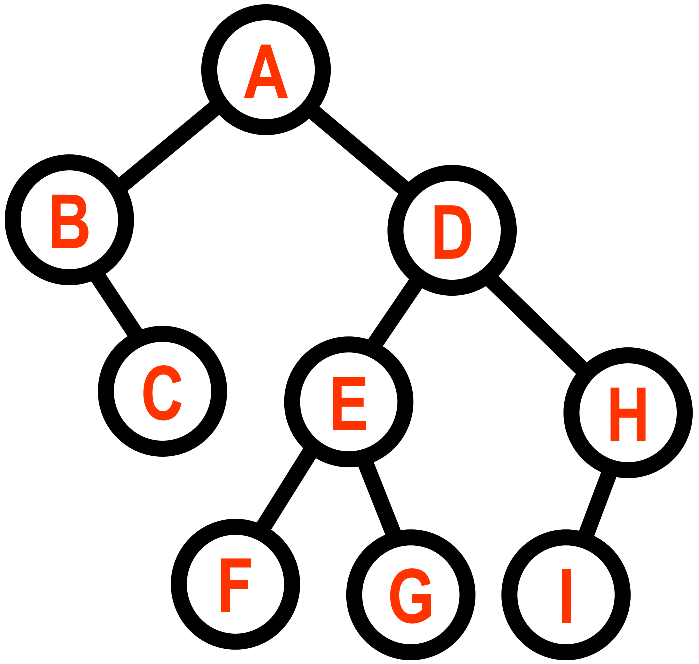


Backward In-order Traversal - Processing Order



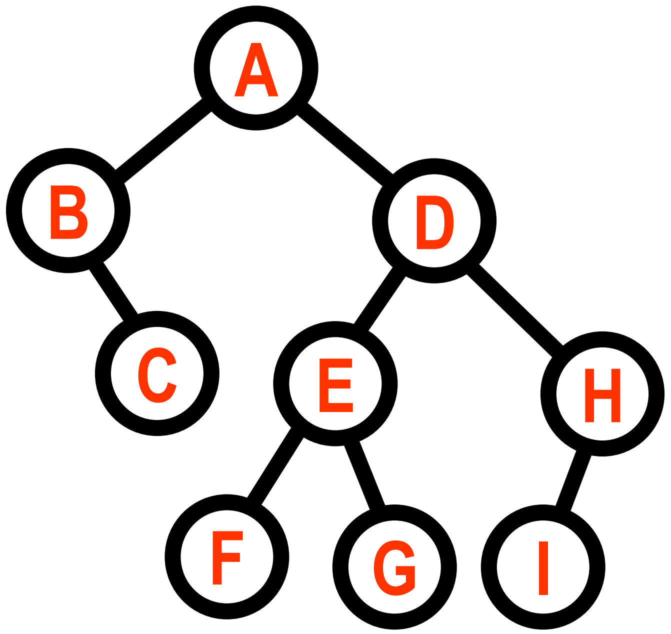


Example: Backward Inorder Traversal & Printing



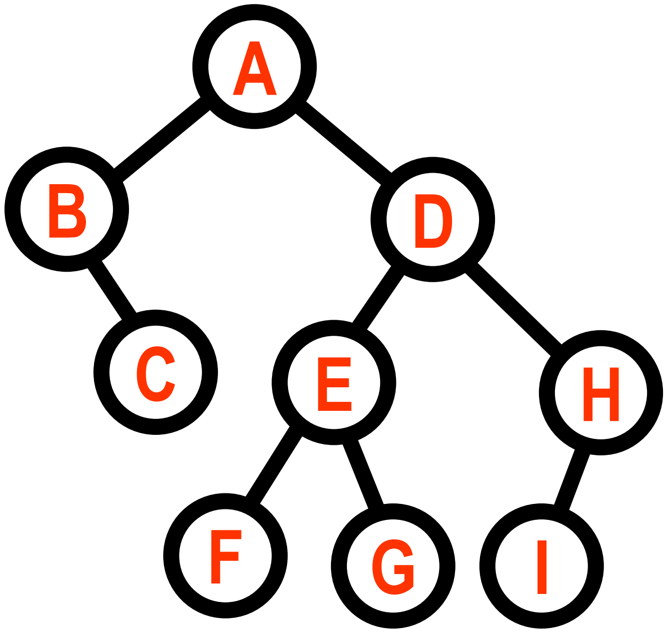
H I D G E F A C B

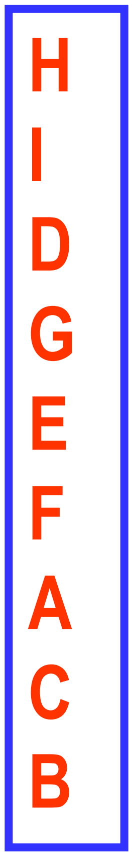
Example: Backward Inorder Traversal & Printing

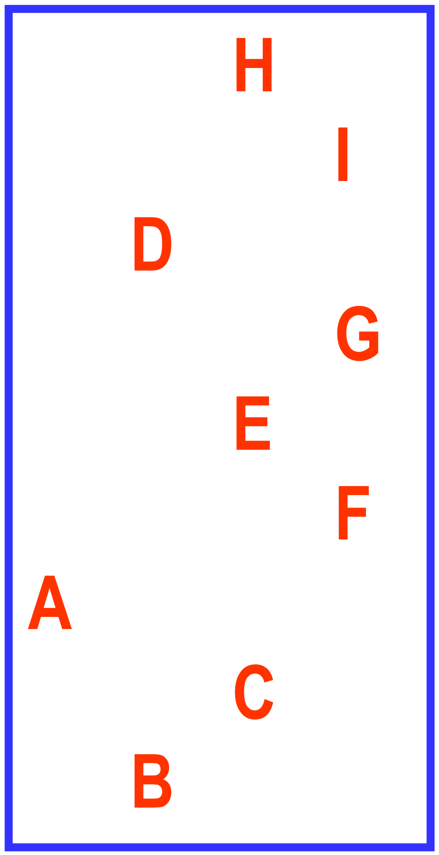


H
I
D
G
E
F
A
C
B

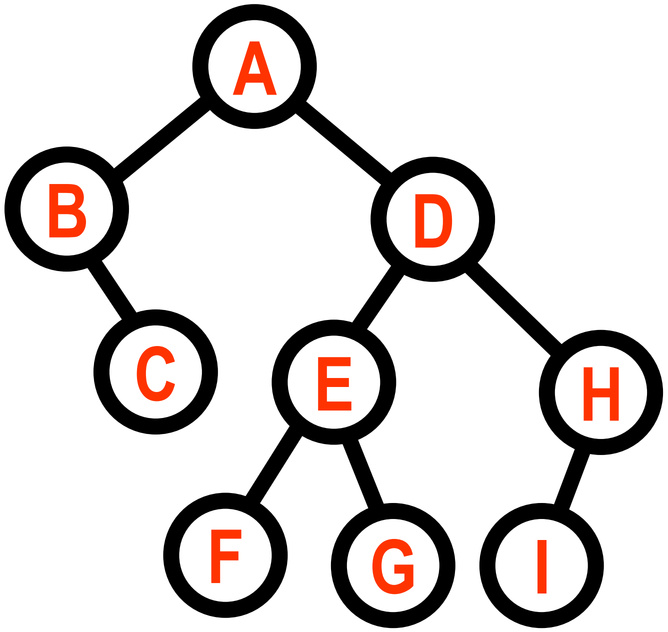
Example: Backward Inorder Traversal & Printing







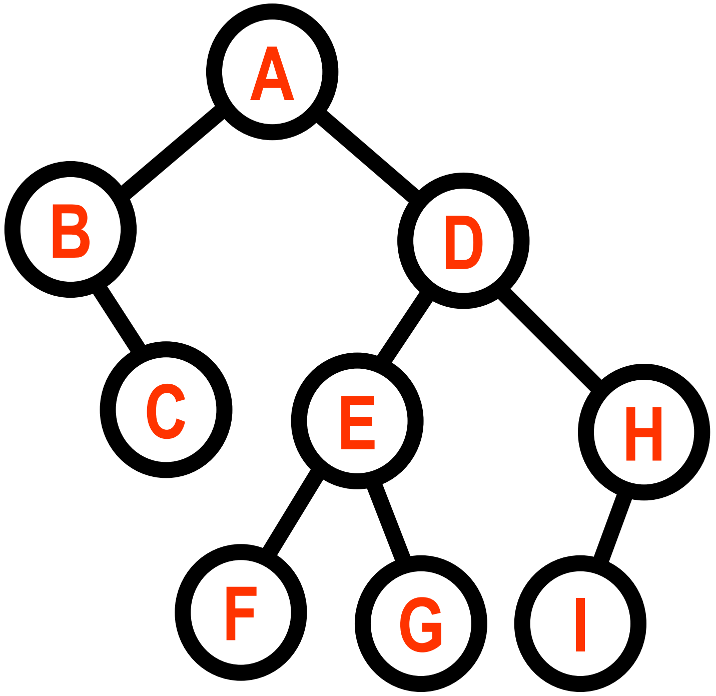
Example: Backward Inorder Traversal & Printing

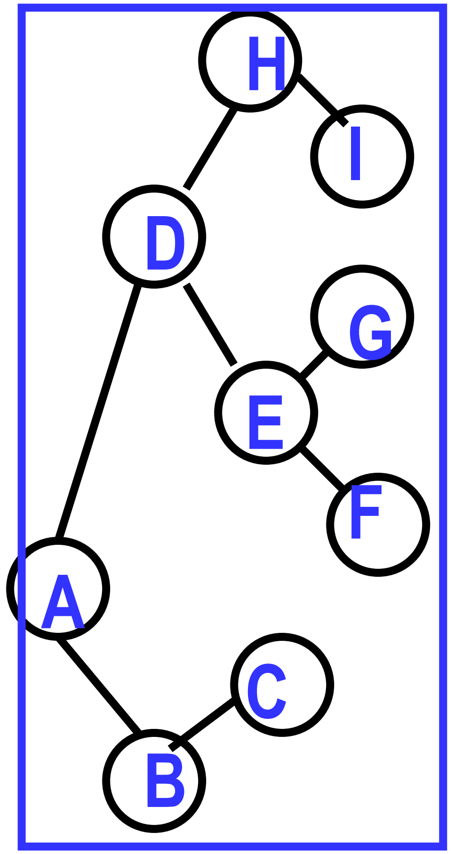


```
H
I
D
G
E
F
A
C
B
```

```
    H \
      I
  D <
      G
    E <
      F
A <
    C
  B /
```

Example: Backward Inorder traversal & Printing





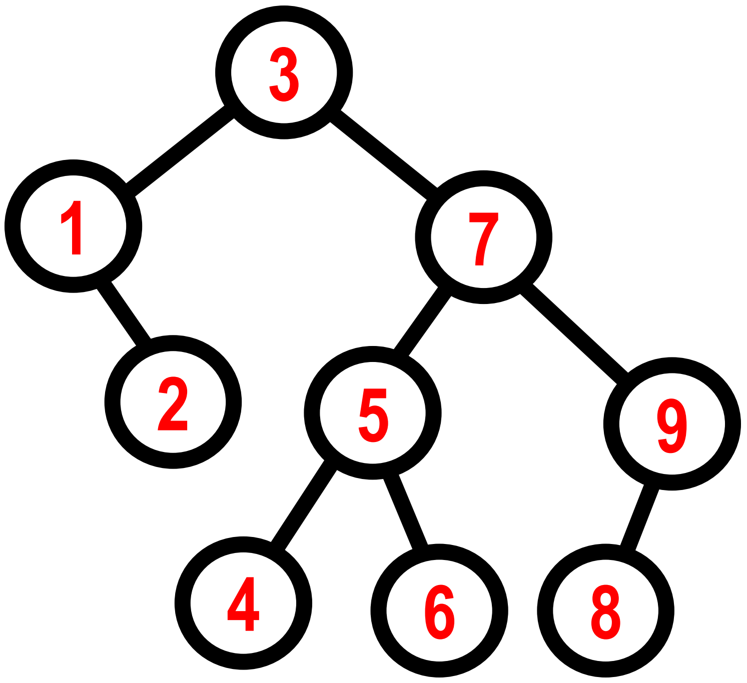
PrintBTree

```
void printBTree() const

// Outputs the keys in a binary tree.
// The tree is output rotated counterclockwise 90 degrees
// using a "reverse" inorder traversal.

{
    if ( root == 0 )
       cout << "Empty tree" << endl;
    else
    {
       cout << endl;
       printBTreeHelper(root,1);
       cout << endl;
    }
}
```

PrintBTreeHelper

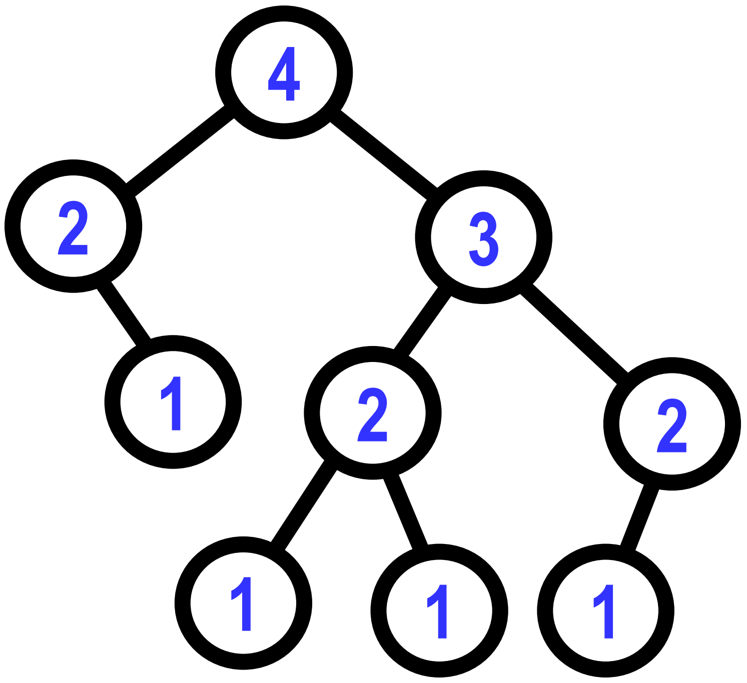
```
void printBTreeHelper( BTreeNode *p, int level ) const

// Recursive helper for printBTree.
// Outputs the subtree whose root node is pointed to by p.
// level is the level of this node within the tree.
{
     int j;
     if ( p != 0 )
     {
        printBTreeHelper(p->right,level+1);          // Output right subtree
        for ( j = 0 ; j < level ; j++ ) cout << "\t";
        cout << " " << p->Key;    // Output key
        if ( ( p->left != 0 ) && ( p->right != 0 ) ) cout << "<";
        else if ( p->right != 0 ) cout << "/";
        else if ( p->left != 0 ) cout << "\\";
        cout << endl;
        printBTreeHelper(p->left,level+1);          // Output left subtree
    }
}
```

►QUIZ? Printing AVL Search Tree Keys?



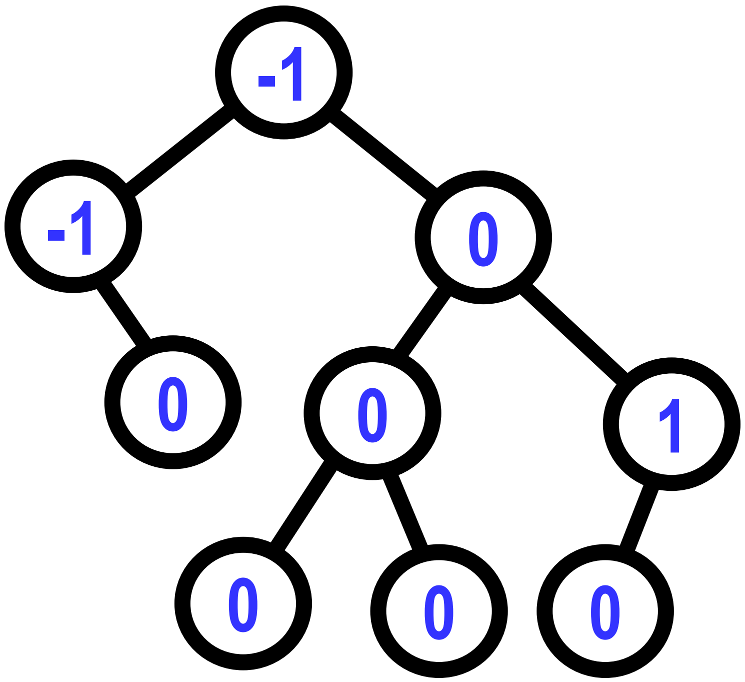
```
        9 \
            8
    7 <
            6
        5 <
            4
3 <
        2
    1 /
```

►QUIZ? Printing AVL Search Tree Heights?



```
        2 \
            1
    3 <
            1
        2 <
            1
4 <
        1
    2 /
```

►QUIZ? Printing AVL Search Tree BFs?



```
        1 \
            0
    0 <
            0
        0 <
            0
-1 <
        0
    -1 /
```

the right majors, minors & concentr
ucation?
for students' academic and career success
ging for many students - *Many students change their*
during college!

e prediction of student success in MMC could
dual students
d their right MMC
chieve their academic goals

Y. PARK • DEPT OF CS&IS, BRADLEY UNIVERSITY

1/2

END

Prof. Young Park