CISC 235: Topic 4

Balanced Binary Search Trees

Outline

- Rationale and definitions
- Rotations
- AVL Trees, Red-Black, and AA-Trees
  - Algorithms for searching, insertion, and deletion
  - Analysis of complexity

Balanced Binary Search Trees

Purpose: To achieve a worst-case runtime of O(log n) for searching, inserting and deleting

Three Types We'll Look At :

- AVL Trees
- Red-Black Trees
- AA-Trees

There are many types of balanced BSTs

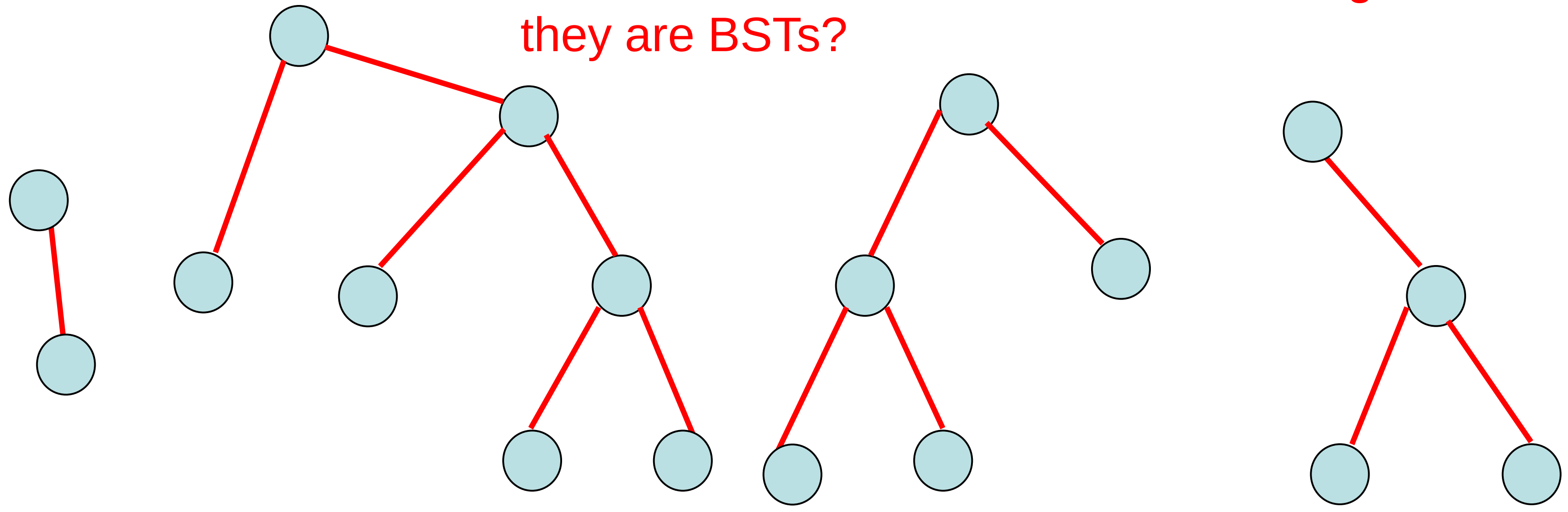
AVL Trees

Invented in 1962 by Russian mathematicians Adelson-Velski and Landis

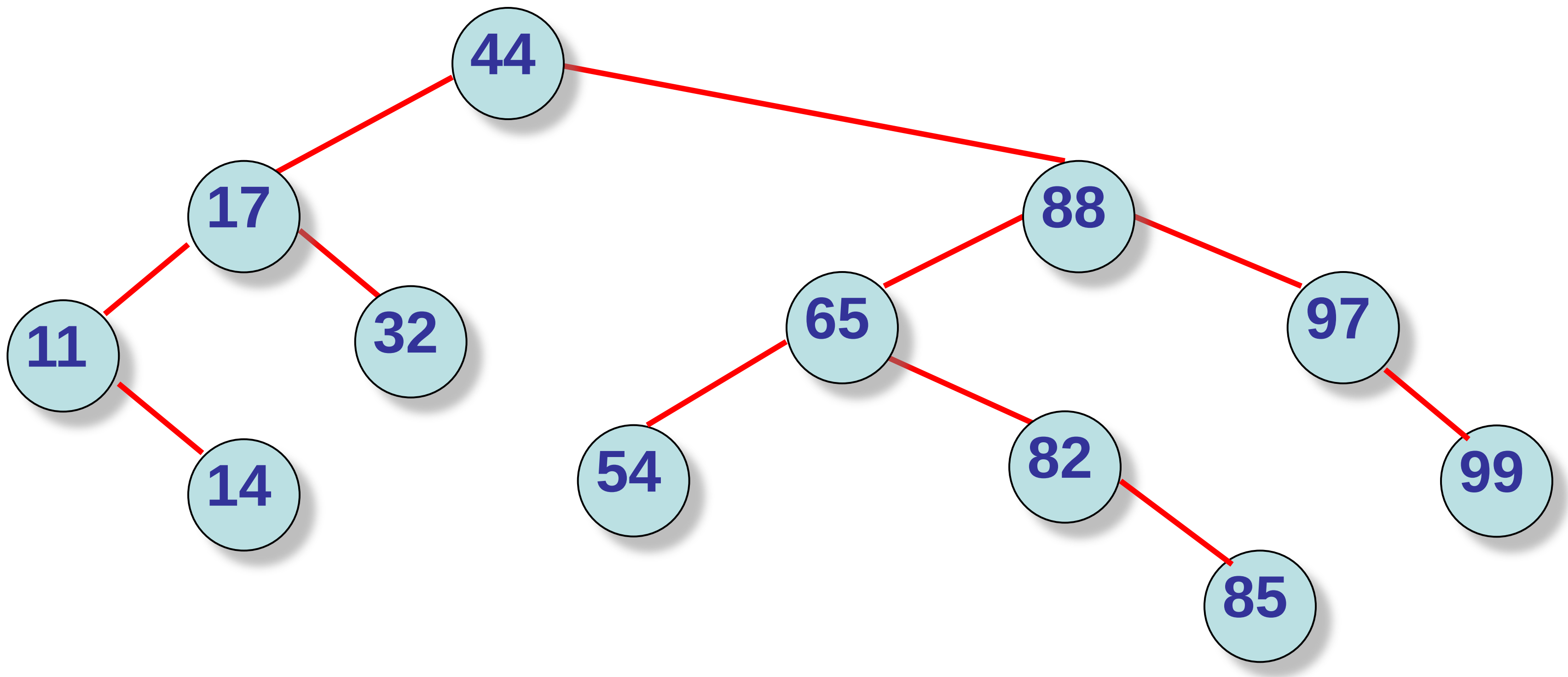
An AVL tree is a binary search tree such that:

- The height of the left and right sub-trees of the root differ by at most 1
- The left and right sub-trees are AVL trees

Which of these are AVL trees, assuming that they are BSTs?



Valid AVL Tree



Note: it is not a requirement that all leaves be on the same or adjacent level

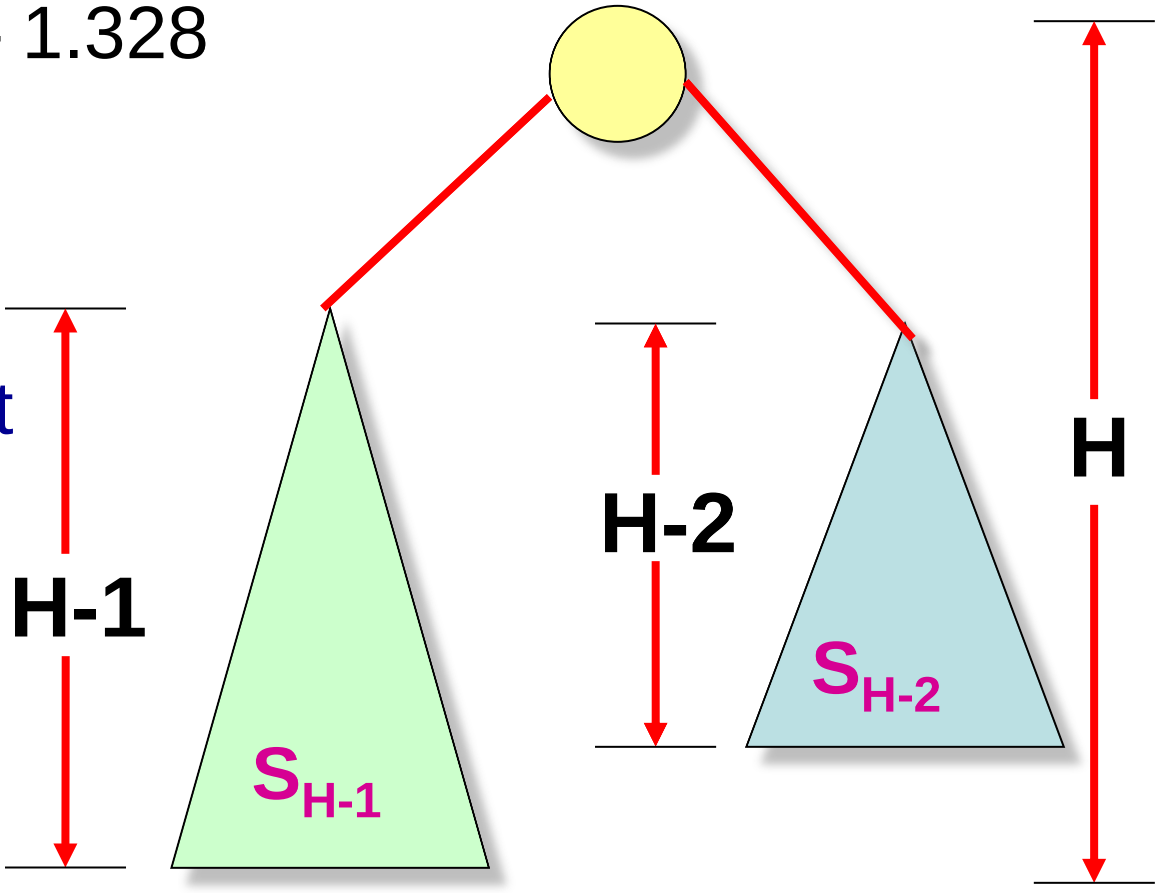
Minimum AVL Tree of Height H

$H < 1.44 \log(N+2) - 1.328$

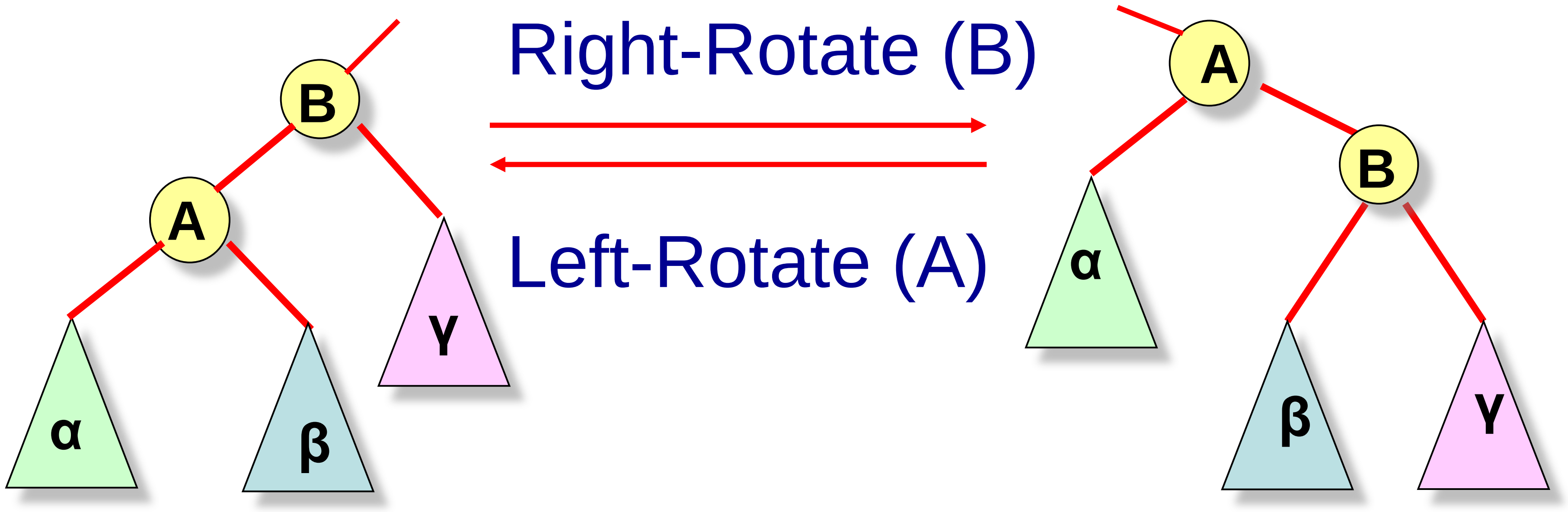
Let S_H be the size of the smallest AVL tree of height H. Then:

$S_0 = 1$, $S_1 = 2$

$S_H = S_{H-1} + S_{H-2} + 1$



Rotations



Rotations maintain the ordering property of BSTs.

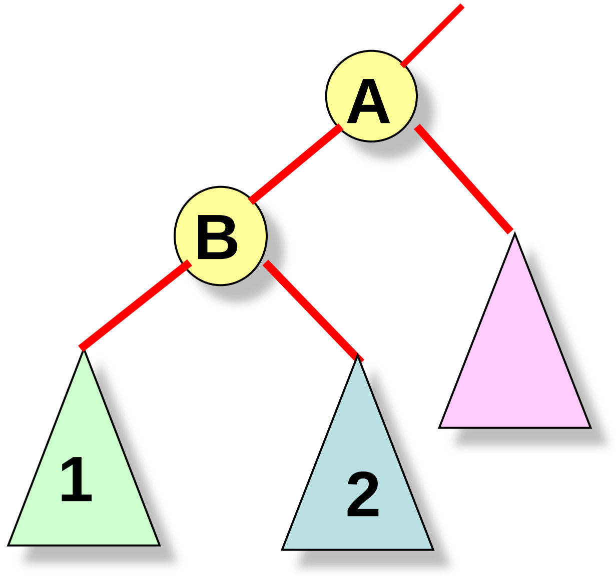
a ϵ α, b ϵ β, c ϵ γ implies a ≤ A ≤ b ≤ B ≤ c

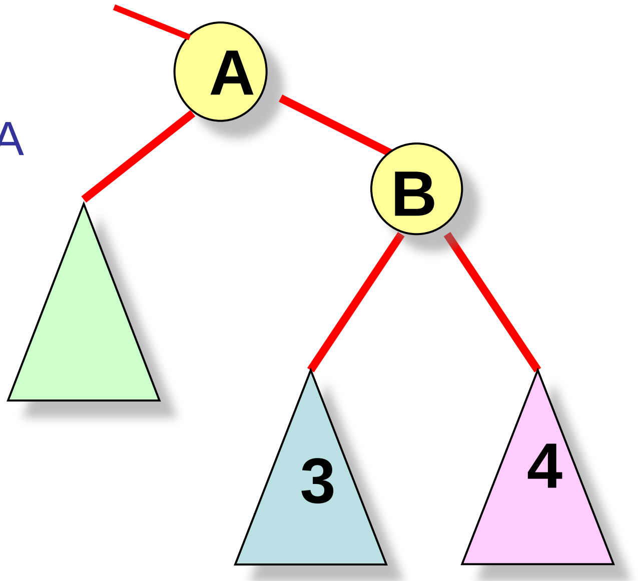
A rotation is an O(1) operation

Insertions: 4 Cases

When inserting into a sub-tree of A, there are 4 cases in which a height violation could occur:

1. Inserting in the left sub-tree of the left child of A
2. Inserting in the right sub-tree of the left child of A
3. Inserting in the left sub-tree of the right child of A
4. Inserting in the right sub-tree of the right child of A



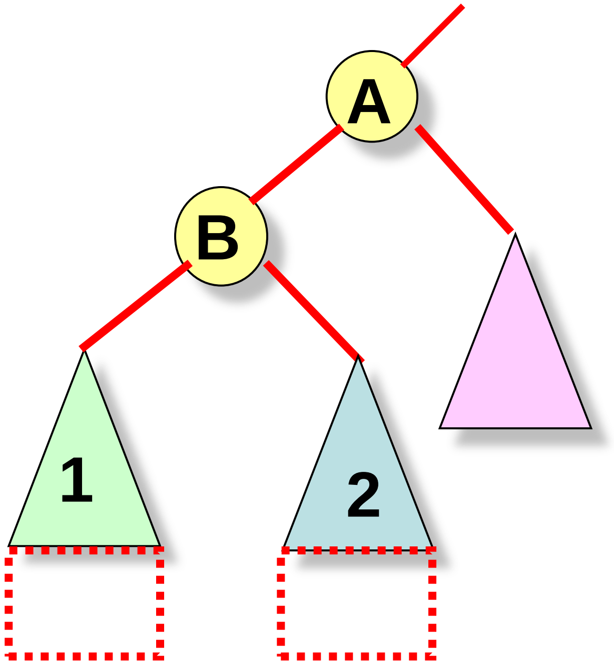


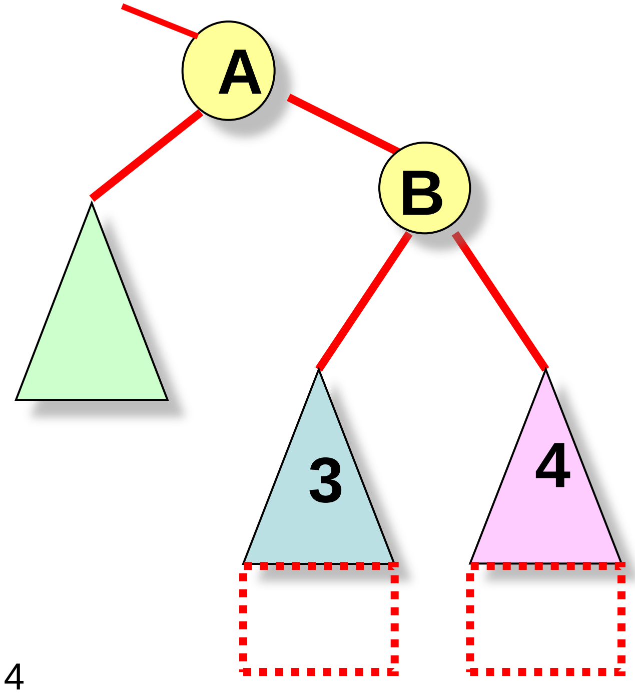
Rotations Required for the 4 Cases

Case 1: Requires a single right rotation to balance

Case 2 and 3: Require double rotations to balance

Case 4: Requires a single left rotation to balance



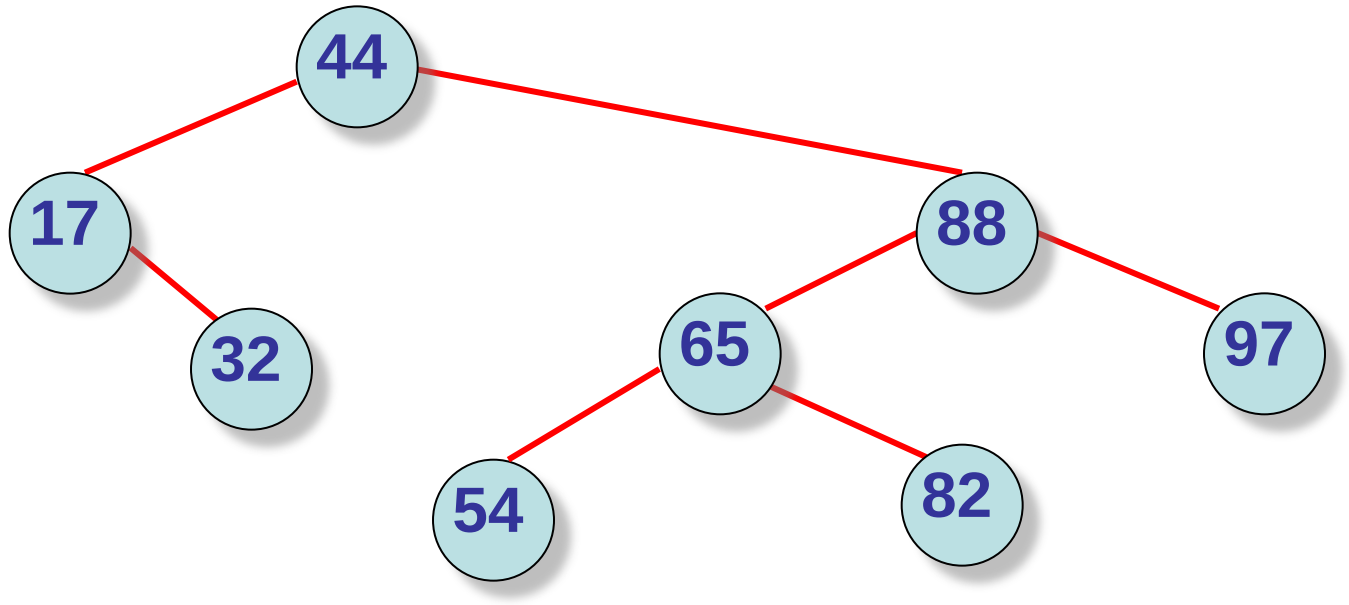


Insertion in an AVL Tree

- First insert node w in AVL tree T as for plain binary search tree
- Then find the first node x going up from w to the root that is unbalanced (if none, are finished)
- Apply appropriate rotation (single or double), which reduces height of sub-tree rooted at x by 1

Since all nodes in T that became unbalanced were on the path of T from w to the root, restoring the sub-tree rooted at x to its original height rebalances the entire tree.

Insertion in an AVL Tree



What insertion value would cause a Case 1 rotation? Case 2? 3? 4?

Deletion in an AVL Tree

- First delete node w in AVL tree T as for plain binary search tree
- Then find the first node x going up from w to the root that is unbalanced (if none, are finished)
- Apply appropriate rotation (single or double), which results either in the sub-tree rooted at x being its original height before the deletion, or in its height being decreased by 1.

Balancing the sub-tree rooted at x may NOT rebalance the entire tree. O(log n) rotations may be required.

Advantages/Disadvantage of AVL Trees

- Advantages
  - O(log n) worst-case searches, insertions and deletions
- Disadvantages
  - Complicated Implementation
    - Must keep balancing info in each node
    - To find node to balance, must go back up in the tree: easy if pointer to parent, otherwise difficult
    - Deletion complicated by numerous potential rotations

Red-Black Trees

- Improvement over AVL Trees:
  - A single top-down pass can be used during insertion and deletion routines

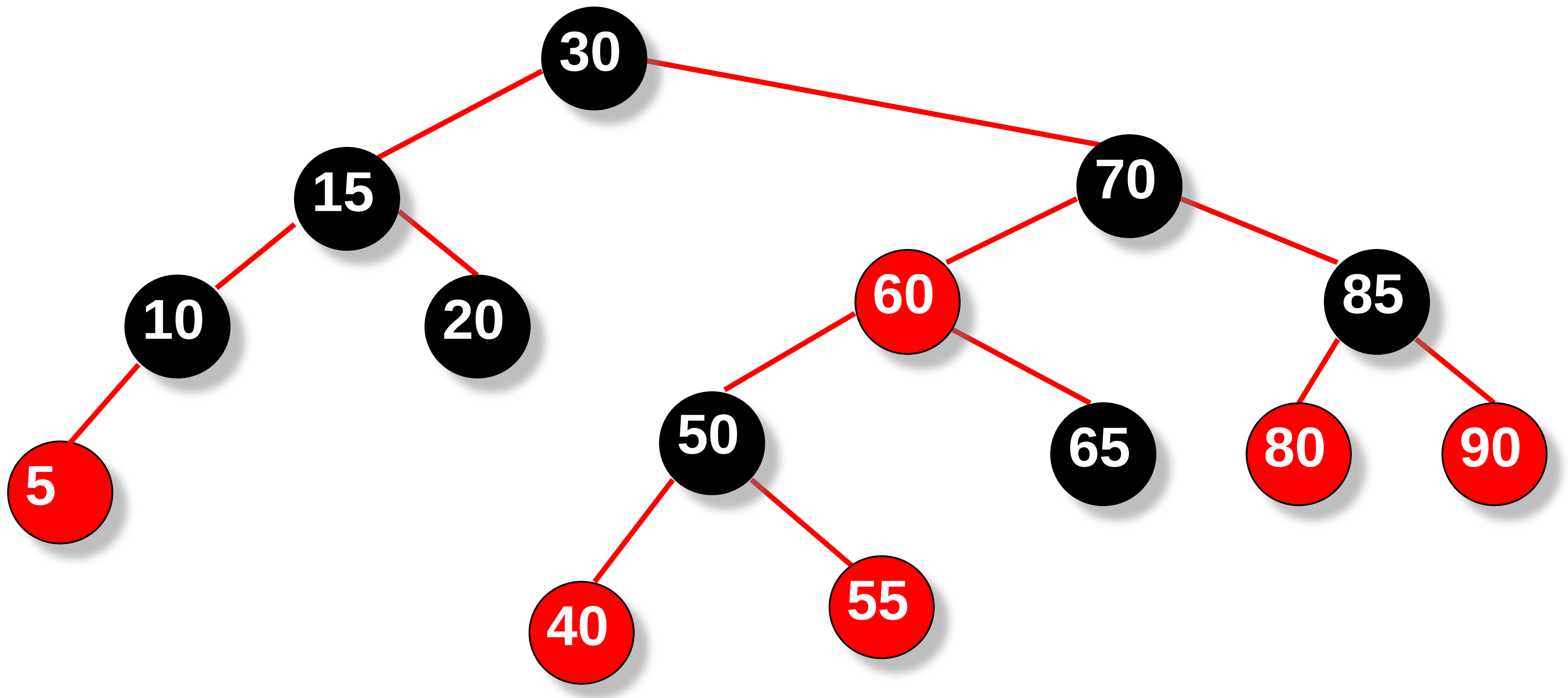
However, the implementation and number of rotation cases is still complex, so we will only look at Red-Black Trees as a step towards considering AA-Trees.

Red-Black Trees

A Red-Black Tree is a binary search tree with the following ordering properties:

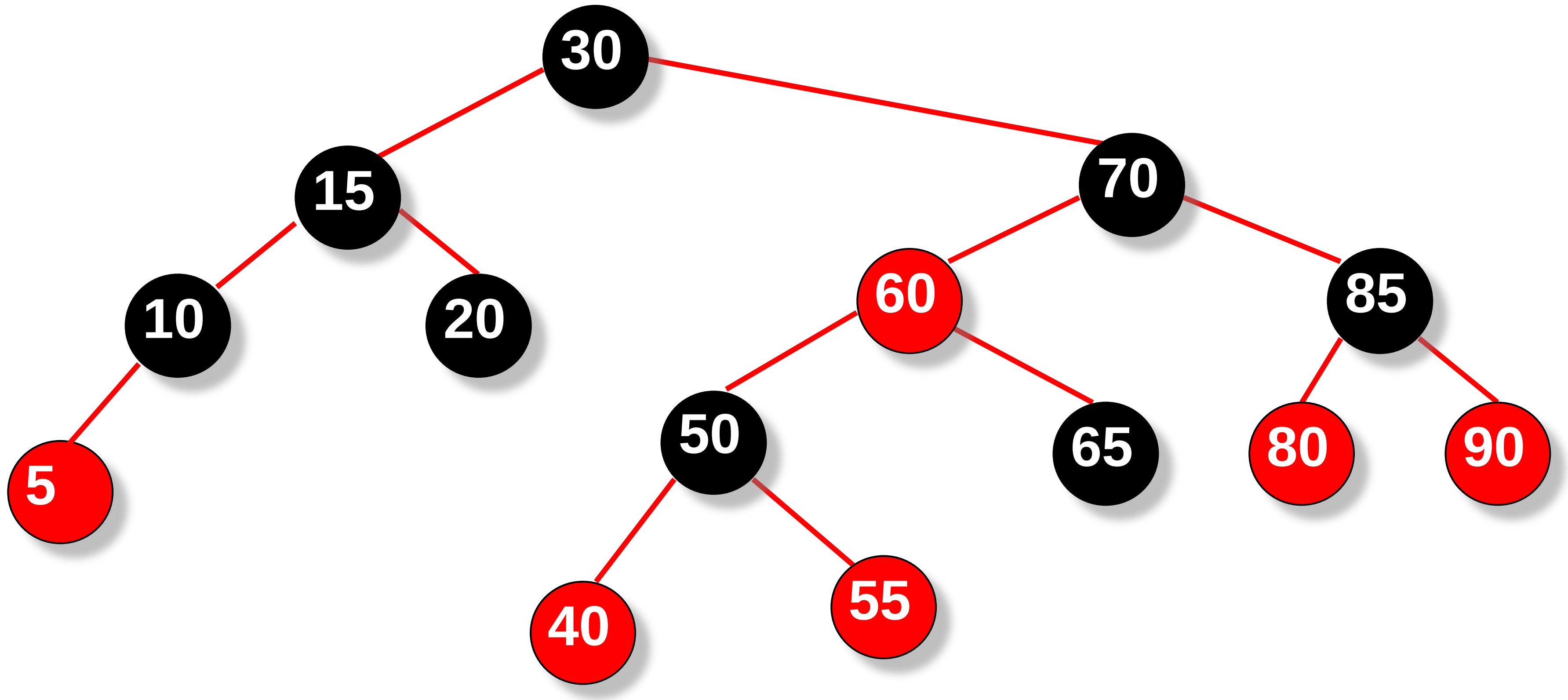
1. Every node is colored either red or black.
2. The root is black
3. If a node is red, its children must be black.
4. All simple paths from any node x to a descendent leaf must contain the same number of black nodes = *black-height(x)*

A Red-Black Tree



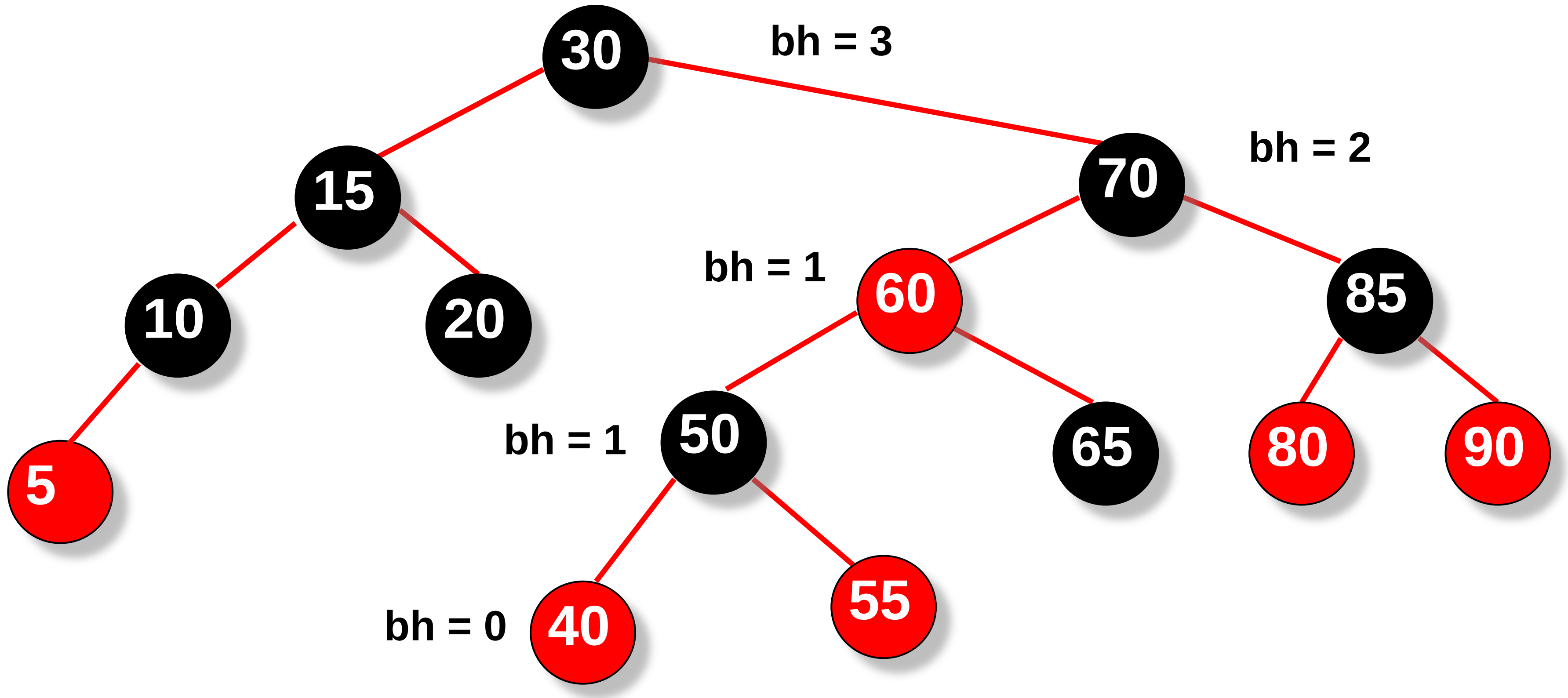
1. Every node is colored either red or black
2. The root is black

A Red-Black Tree



3. If a node is red, its children must be black

A Red-Black Tree



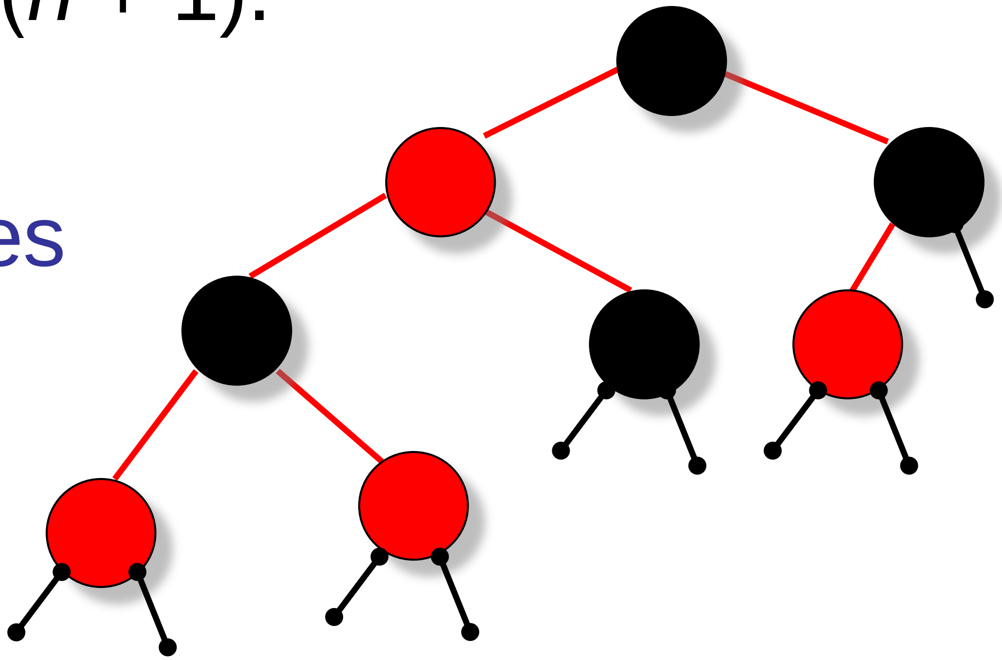
4. All simple paths from any node x to a descendent leaf must contain the same number of black nodes = *black-height(x)*

Height of a Red-Black Tree

Theorem. A red-black tree with n keys has height $h \leq 2 \lg(n + 1)$.

INTUITION:

- Merge red nodes into their black parents.

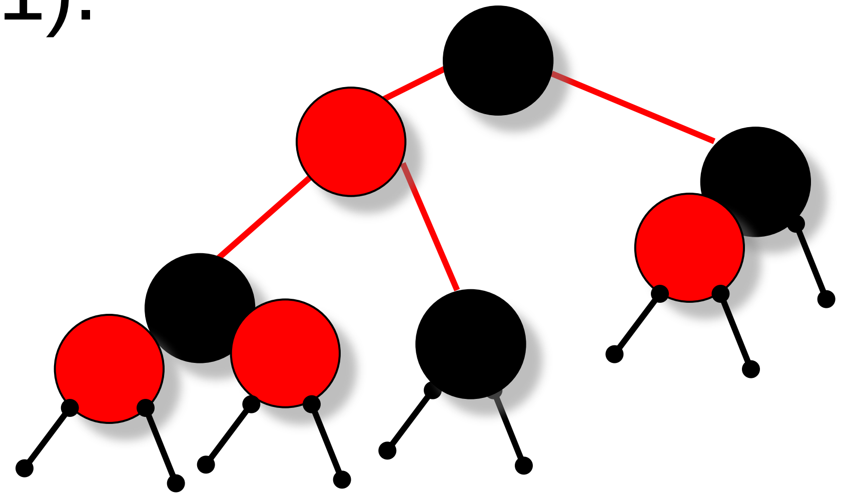


Height of a Red-Black Tree

Theorem. A red-black tree with n keys has height $h \le 2\lg(n + 1)$.

INTUITION:

- Merge red nodes into their black parents.

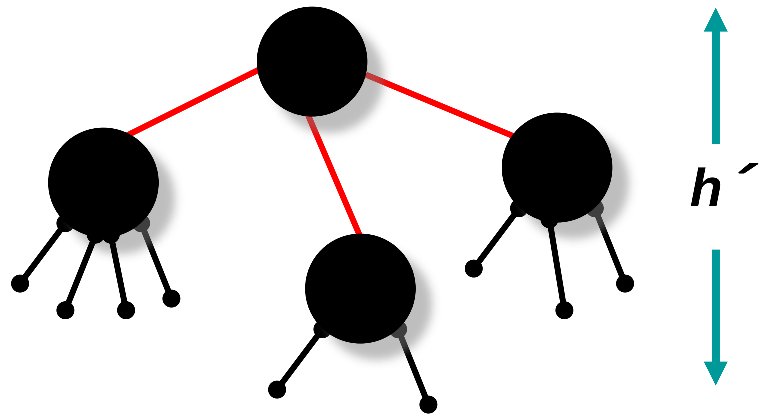


Height of a Red-Black Tree

Theorem. A red-black tree with n keys has height $h \le 2\lg(n + 1)$.

INTUITION:

- Merge red nodes into their black parents.



- This process produces a tree in which each node has 2, 3, or 4 children.

The 2-3-4 tree has uniform depth h' of leaves.

AA-Trees

- Improvement over Red-Black Trees:
  - Fewer rotation cases, so easier to code, especially deletions (eliminates about half of the restructuring cases)

AA-Trees still have O(log n) searches in the worst-case, although they are slightly less efficient.

AA-Tree Ordering Properties

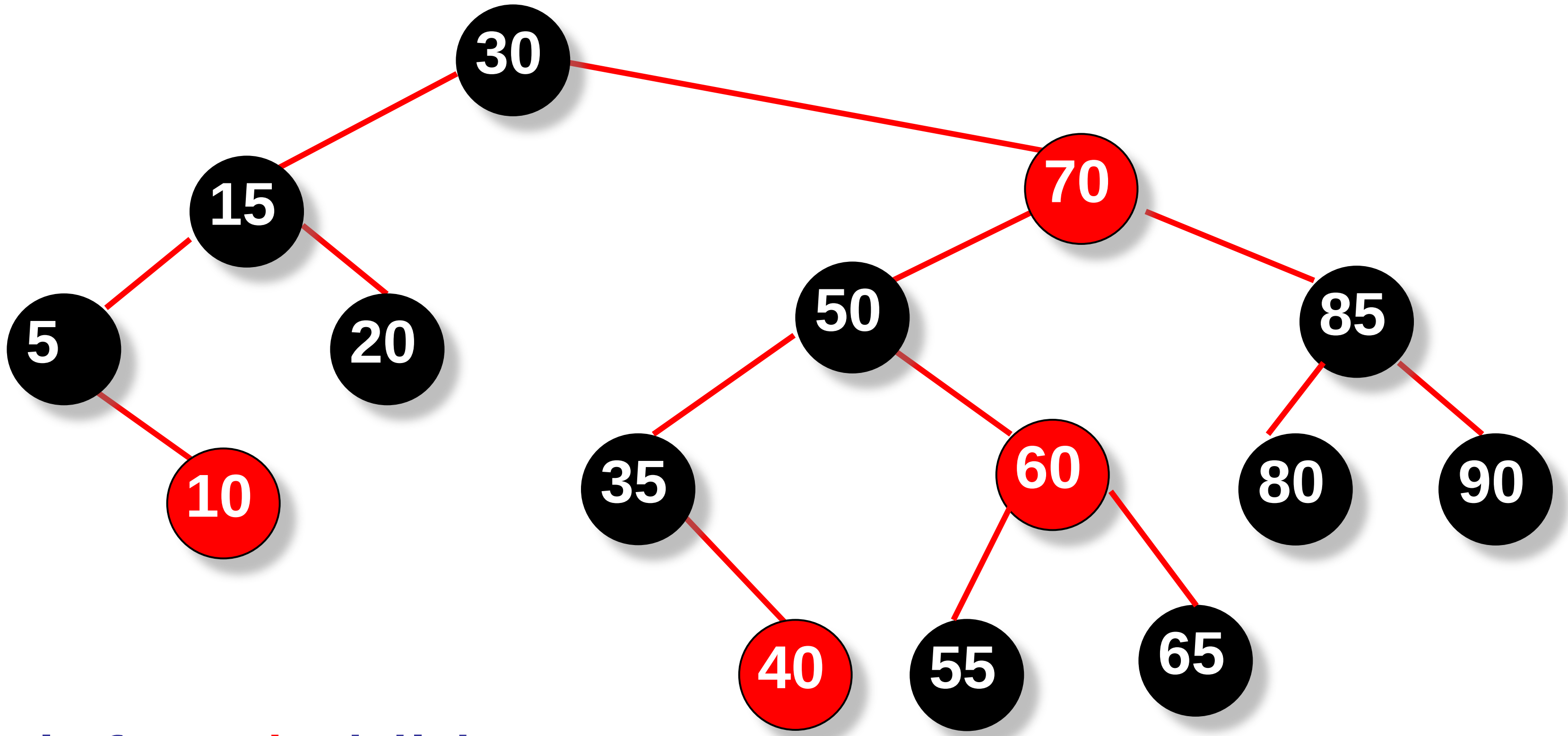
An AA-Tree is a binary search tree with the same ordering properties as a red-black tree:

1. Every node is colored either red or black.
2. The root is black
3. If a node is red, its children must be black.
4. All simple paths from any node x to a descendent leaf must contain the same number of black nodes = *black-height(x)*

PLUS

5. Left children may not be red

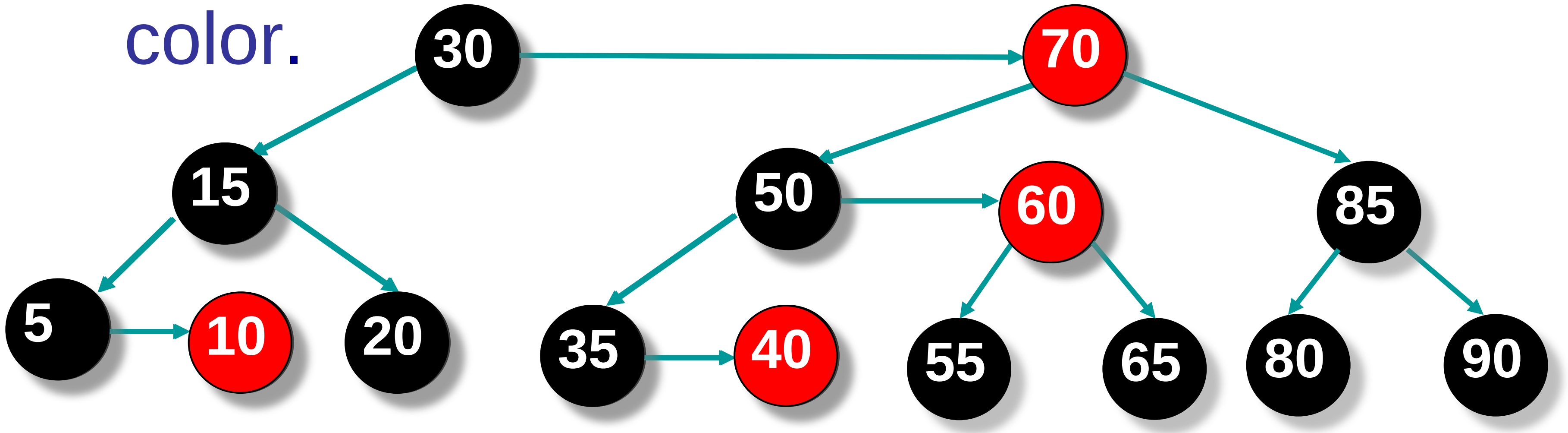
An AA-Tree



No left red children

Representation of Balancing Info

The level of a node is stored instead of its color.

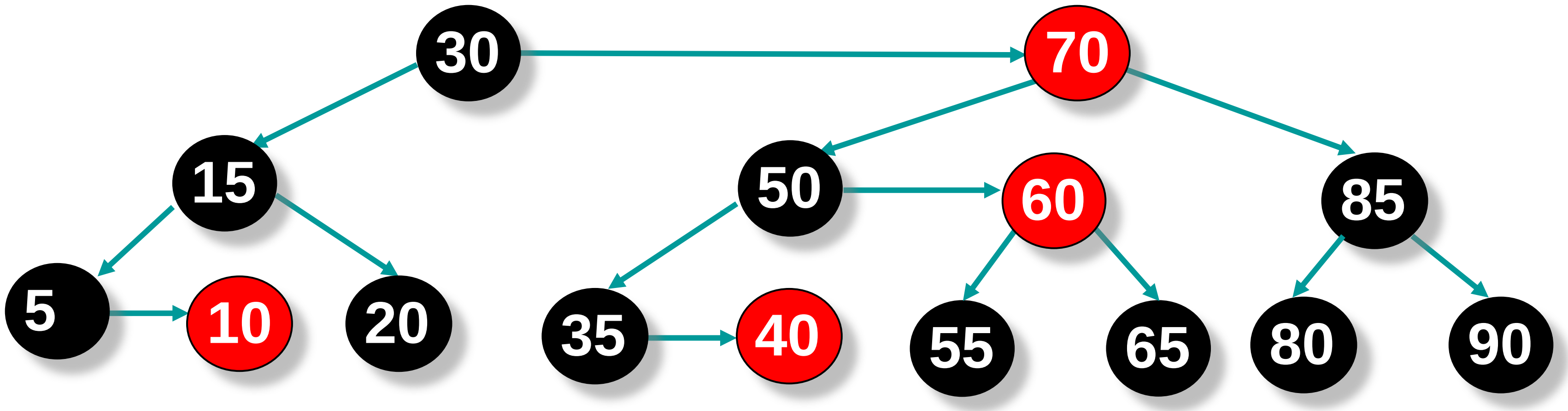


Red children are considered to be at the level of their parent. Note that this is the same tree as that on the previous slide.

Redefinition of “Leaf”

Both the terms leaf and level are redefined:

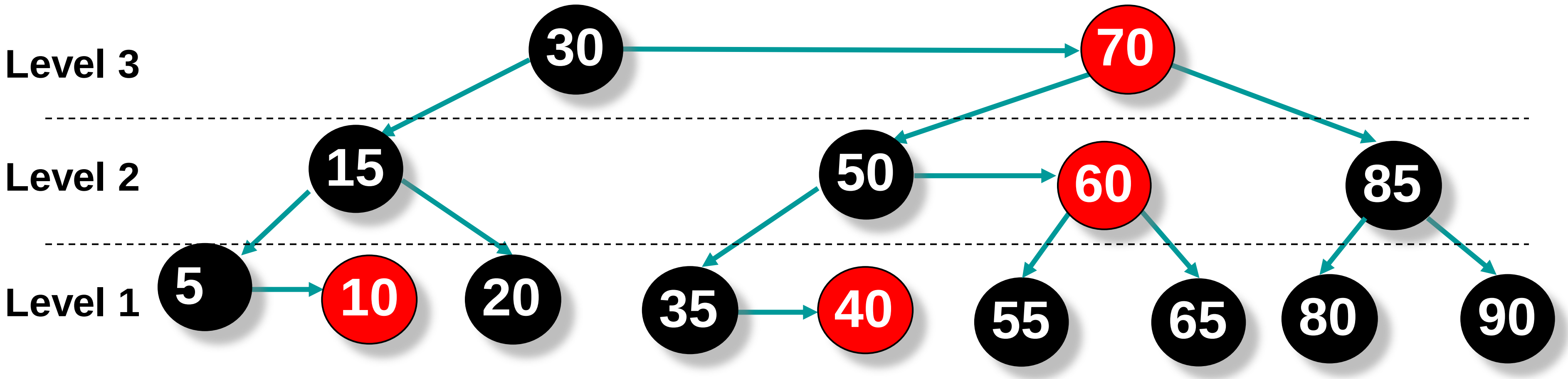
A leaf in an AA-Tree is a node with no black children.



Redefinition of “Level”

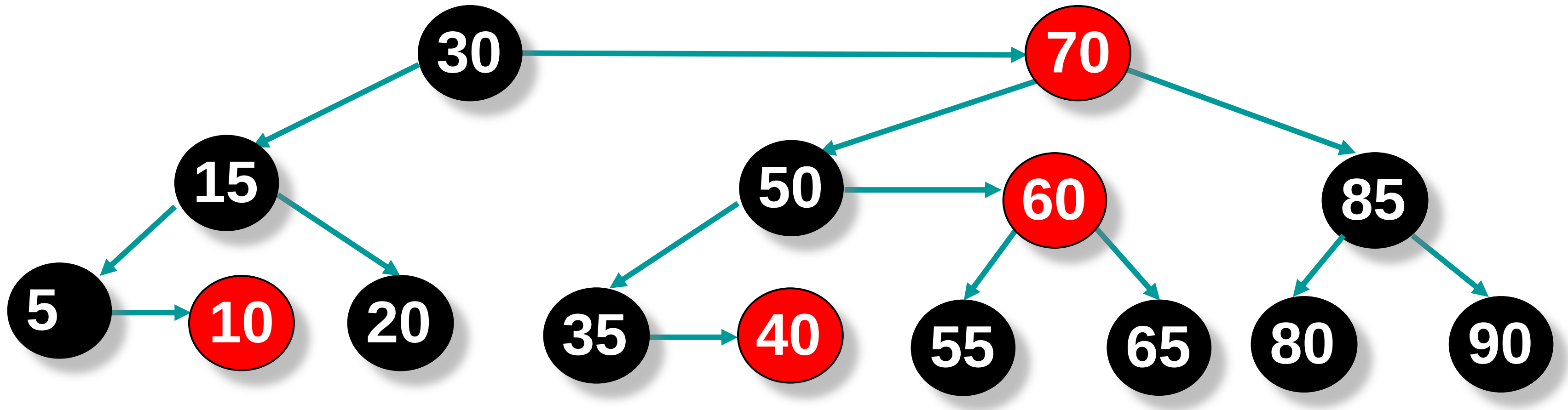
The level of a node in an AA-Tree is:

- Level 1, if the node is a leaf
- The level of its parent, if the node is red
- One less than the level of its parent, if the node is black



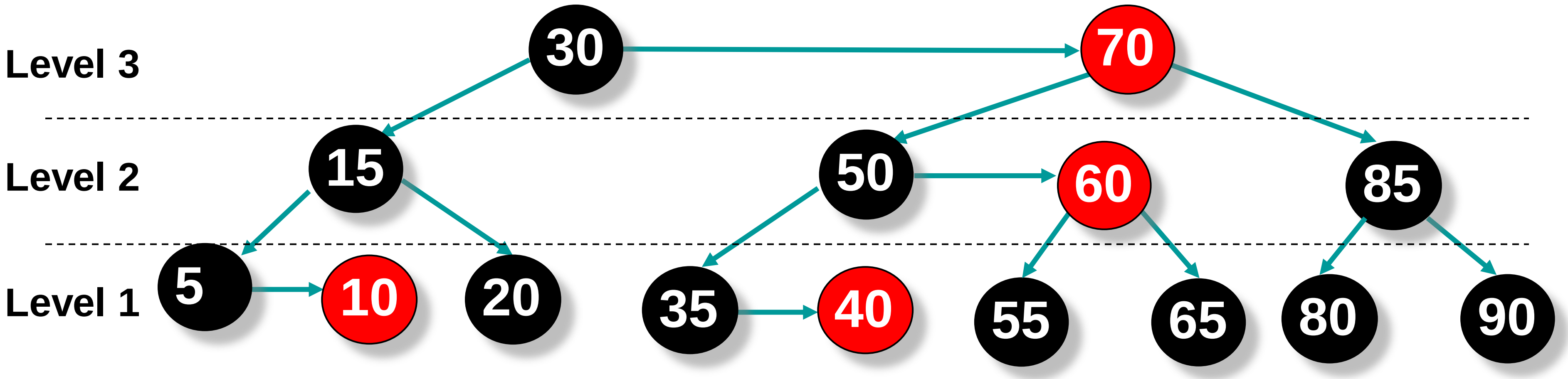
Implications of Ordering Properties

1. **Horizontal links are right links**
   - **because only right children may be red**
2. **There may not be two consecutive horizontal links**
   - **because there cannot be consecutive red nodes**



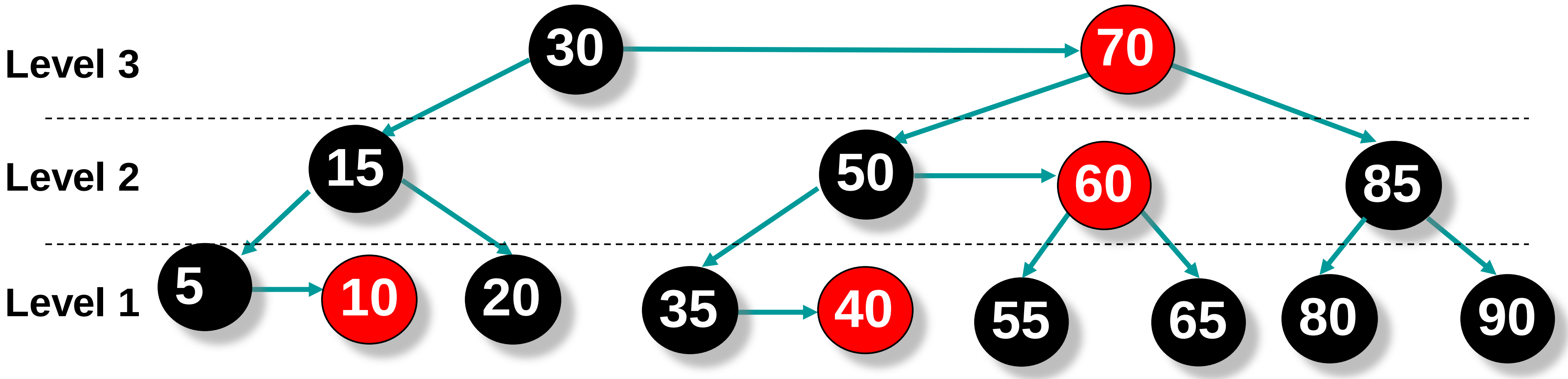
Implications of Ordering Properties

3. **Nodes at level 2 or higher must have two children.**
4. **If a node does not have a right horizontal link, its two children are at the same level.**



Implications of Ordering Properties

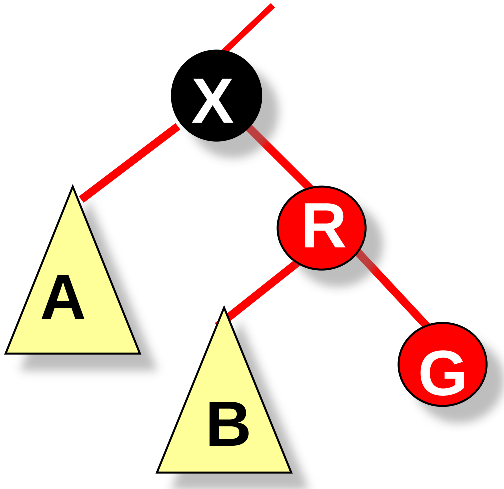
5. **Any simple path from a black node to a leaf contains one black node on each level.**



Adjustments to AA-Trees: Split

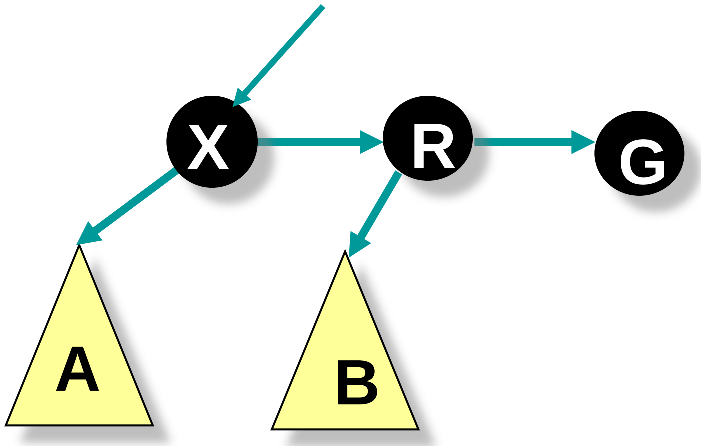
(Color no longer shown for AA-Trees, since only the level is stored)

Problem: With G inserted, there are two reds in a row



Red-Black Tree

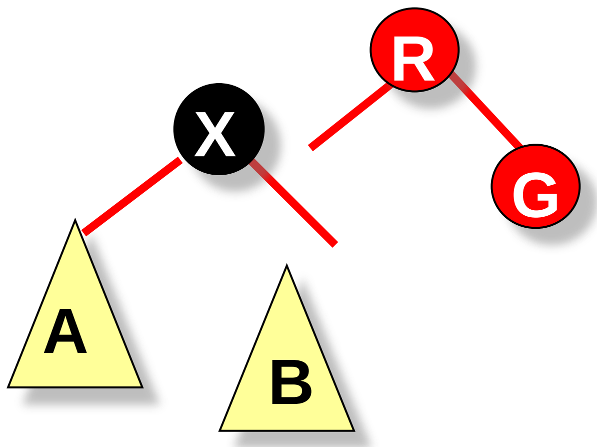
The split procedure is a simple left rotation between X and R



AA-Tree

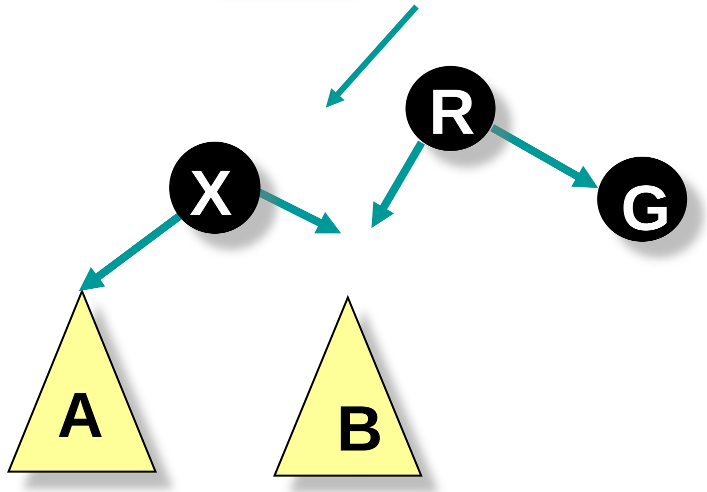
Adjustments to AA-Trees: Split

Problem: With G inserted, there are two reds in a row



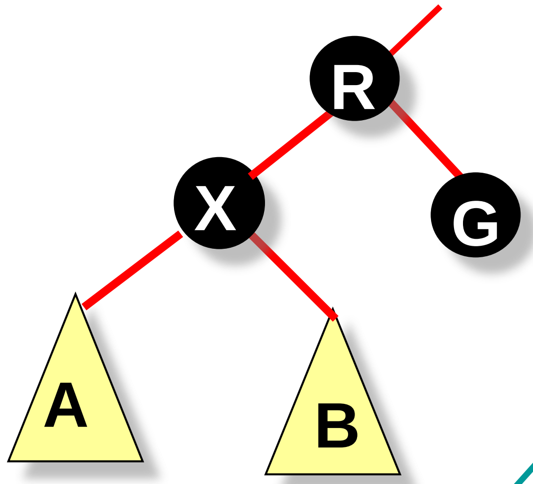
Red-Black Tree

The split procedure is a simple left rotation between X and R

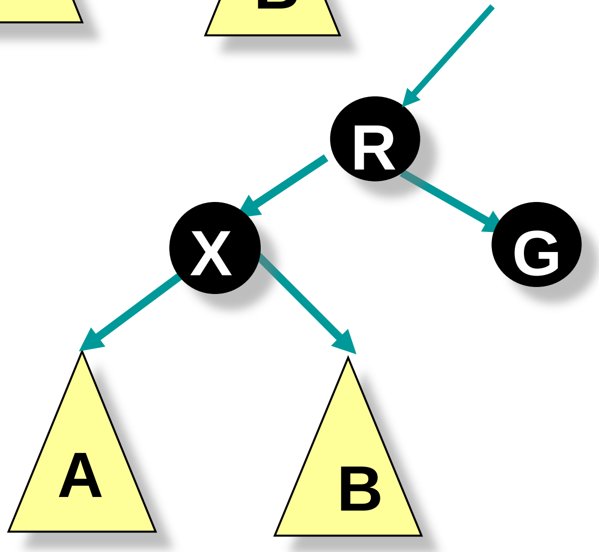


AA-Tree

Adjustments to AA-Trees: Split



Red-Black Tree

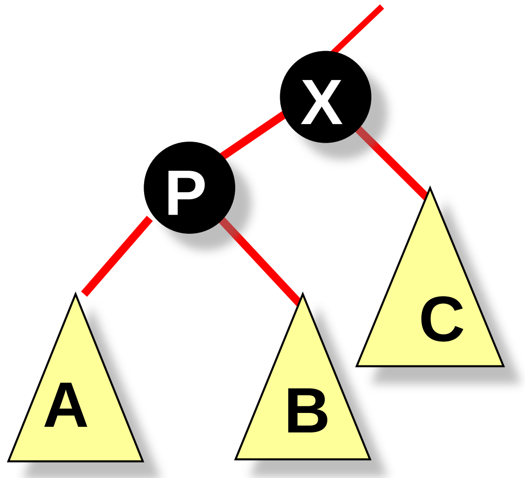


AA-Tree

Note that R's level increases in the AA-Tree

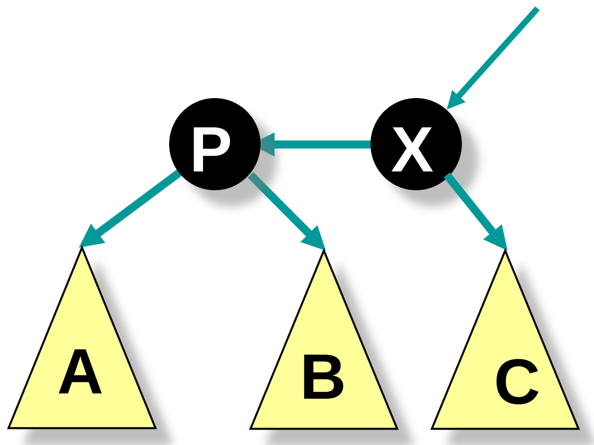
Adjustments to AA-Trees: Skew

Problem:
- Horizontal left link in AA-Tree (too many black nodes on one path)



Red-Black Tree

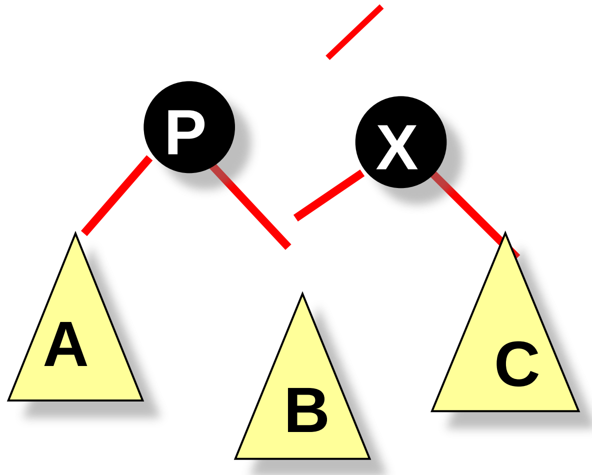
The skew
- procedure is a simple right rotation between X and P



AA-Tree

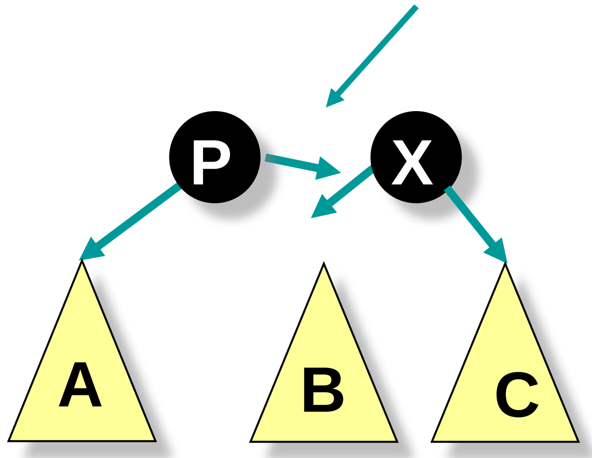
Adjustments to AA-Trees: Skew

Problem:
Horizontal left link in AA-Tree (too many black nodes on one path)



Red-Black Tree

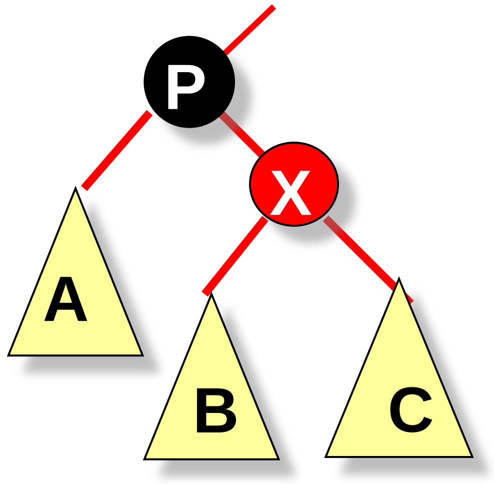
The skew procedure is a simple right rotation between X and P



AA-Tree

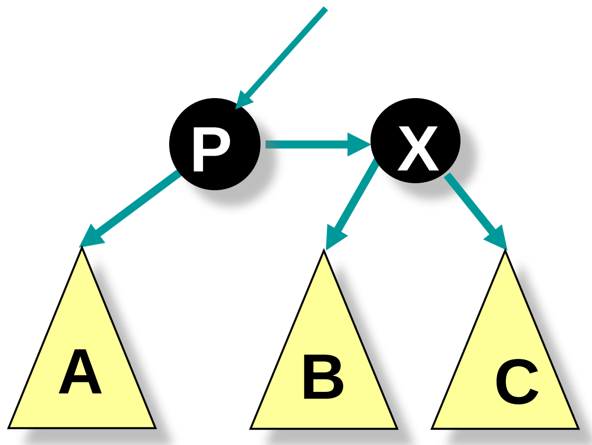
Adjustments to AA-Trees: Skew

Problem:
Horizontal left link in AA-Tree (too many black nodes on one path)



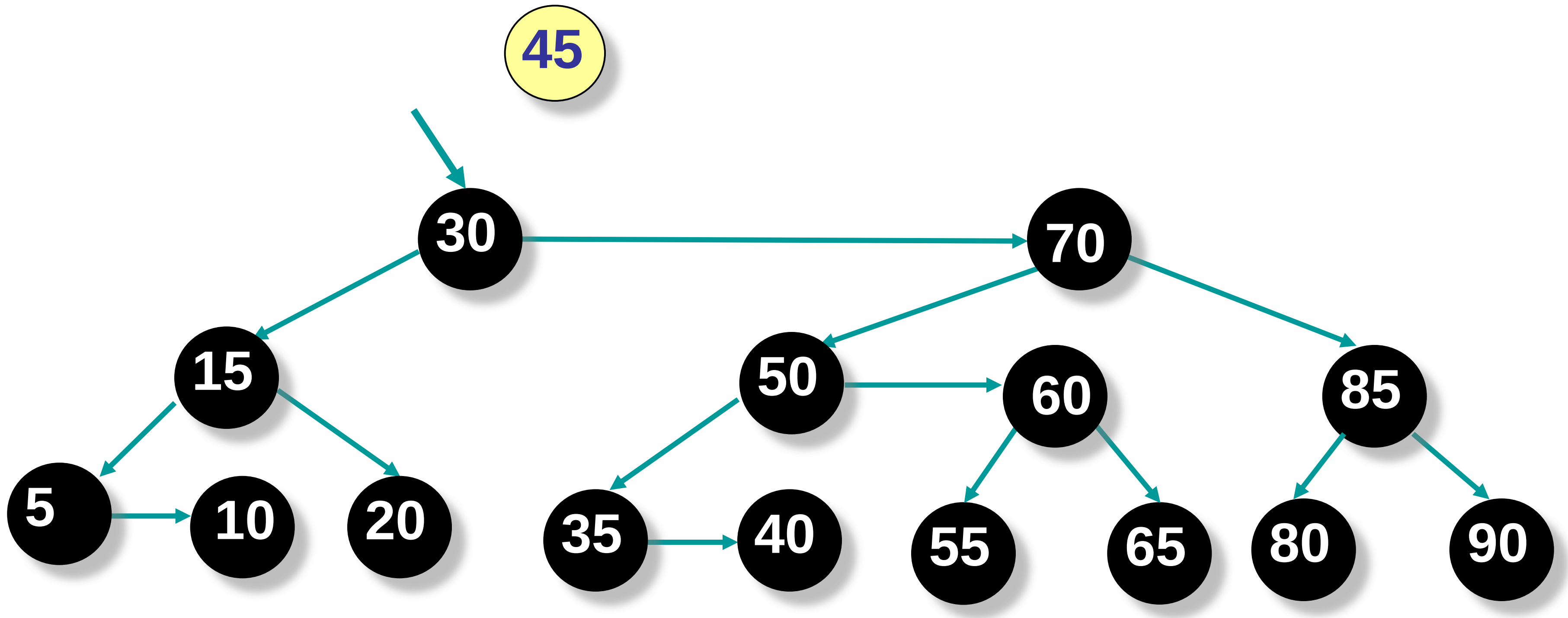
Red-Black Tree

The skew procedure is a simple right rotation between X and P



AA-Tree

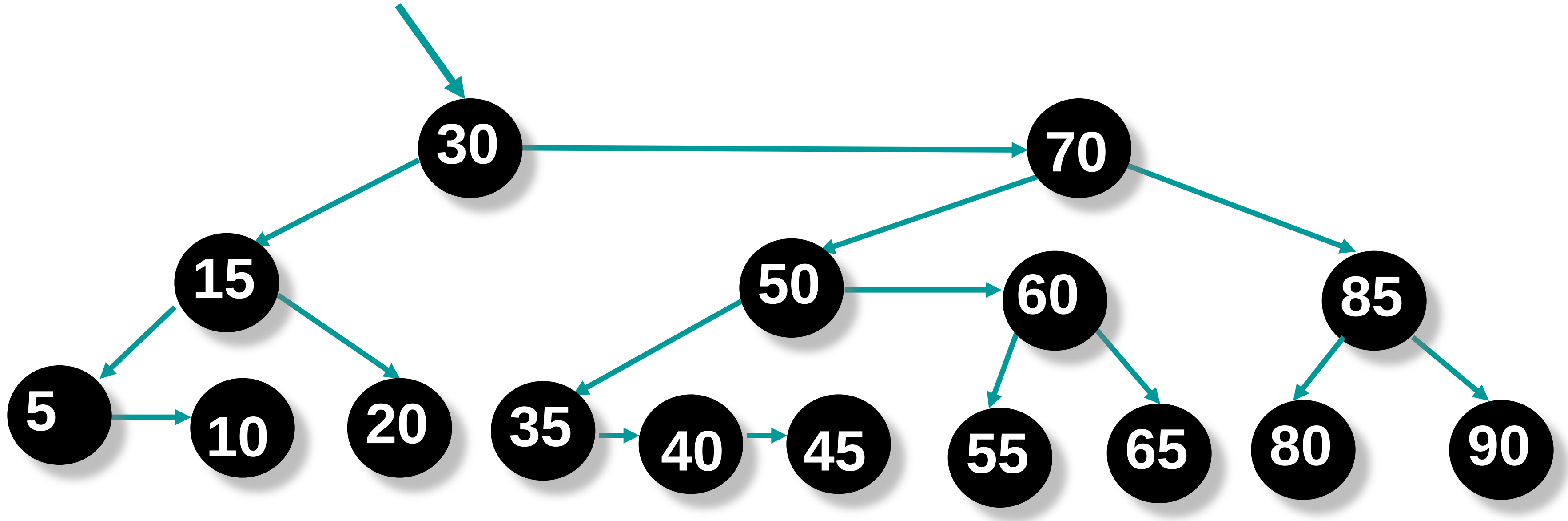
Example: Insert 45



First, insert as for simple binary search tree

Example: Insert 45

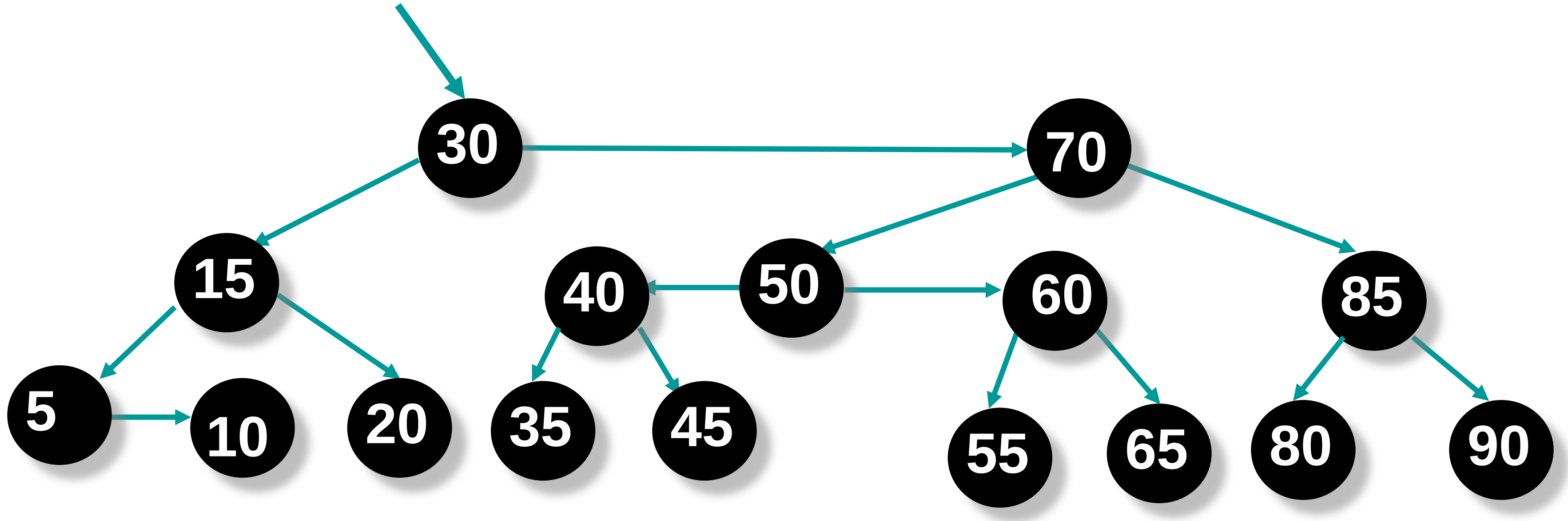
After insert to right of 40:



Problem: Consecutive horizontal links starting at 35, so need split

Example: Insert 45

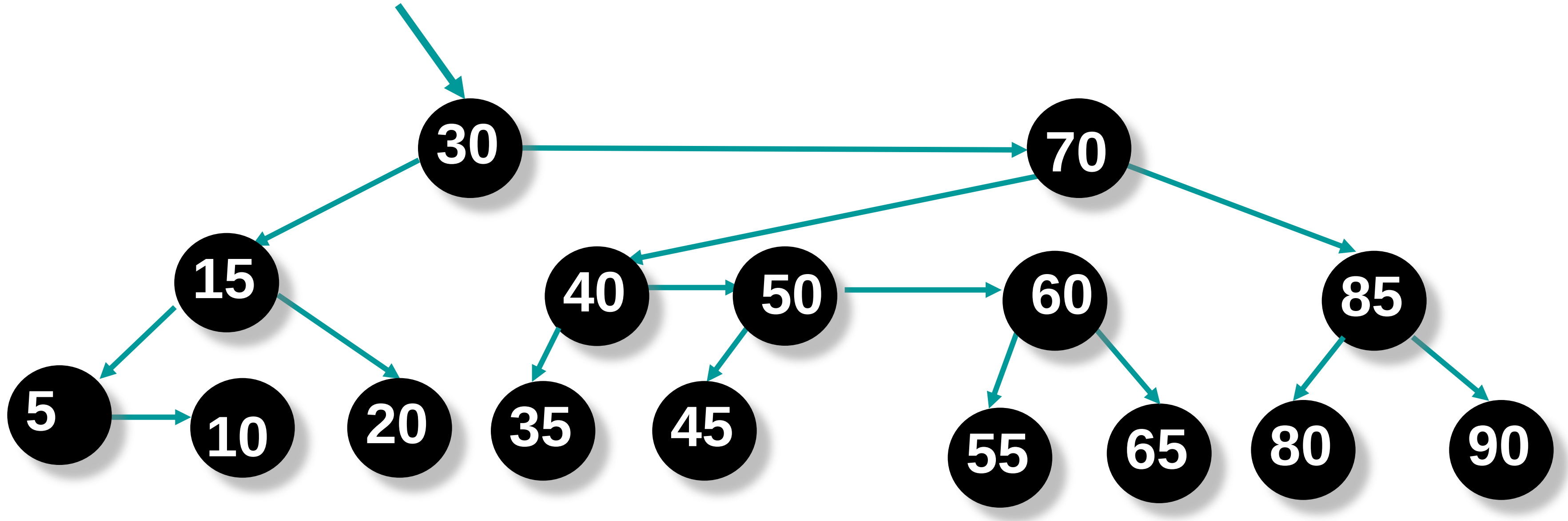
After split at 35:



Problem: Left horizontal link at 50 is introduced, so need skew

Example: Insert 45

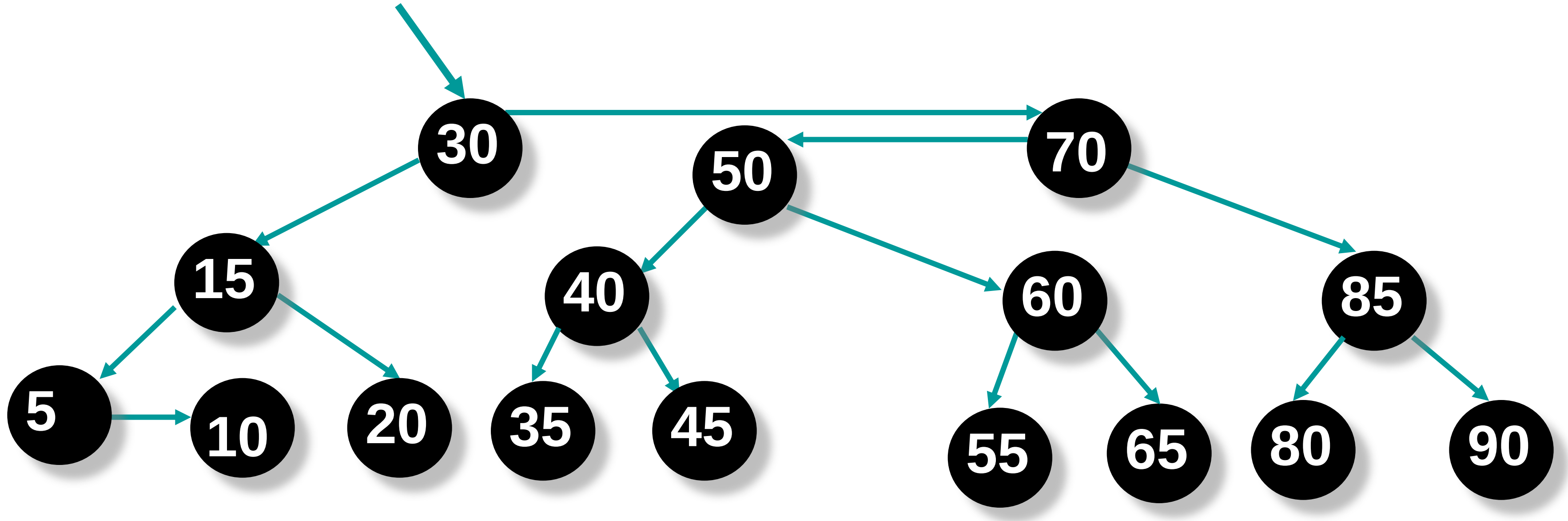
After skew at 50:



Problem: Consecutive horizontal links starting at 40, so need split

Example: Insert 45

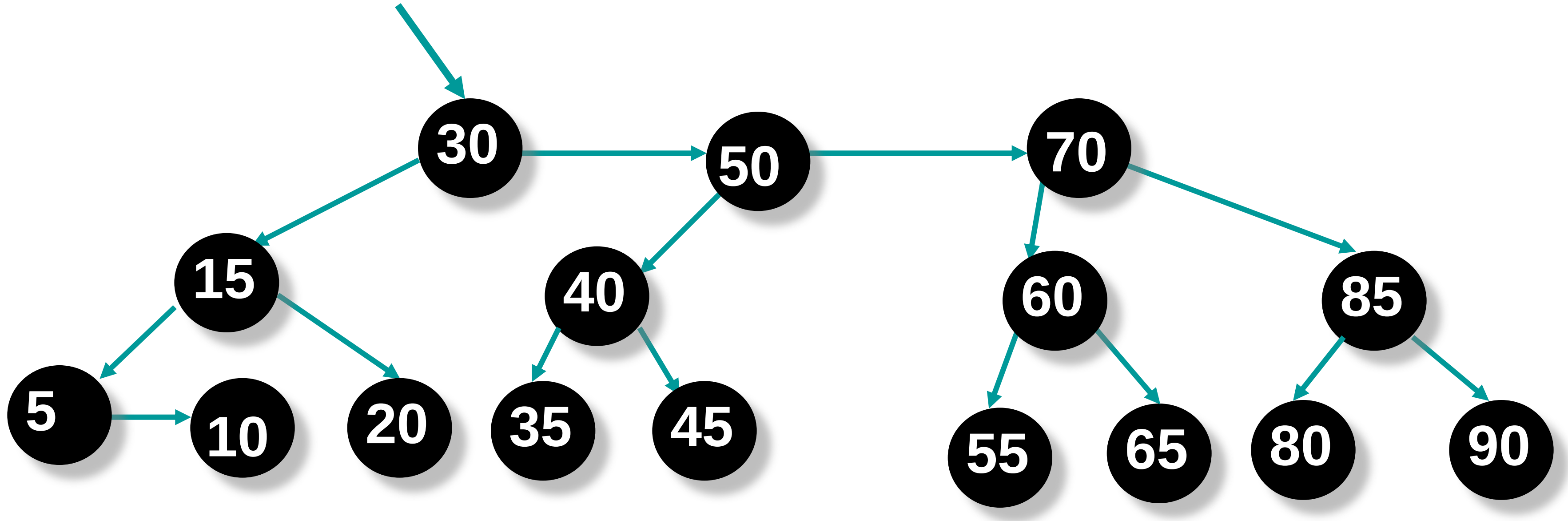
After split at 40:



Problem: Left horizontal link at 70 is introduced (50 is now on same level as 70), so need skew

Example: Insert 45

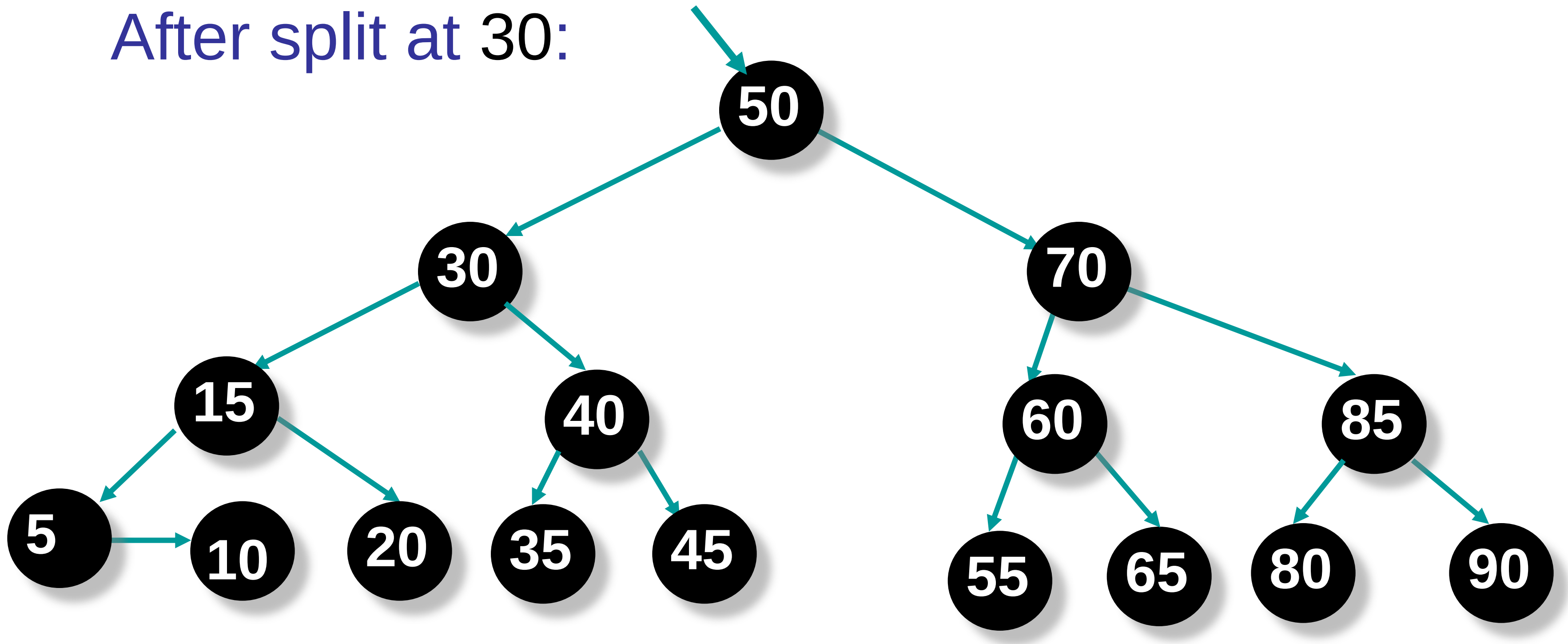
After skew at 70:



Problem: Consecutive horizontal links starting at 30, so need split

Example: Insert 45

After split at 30:



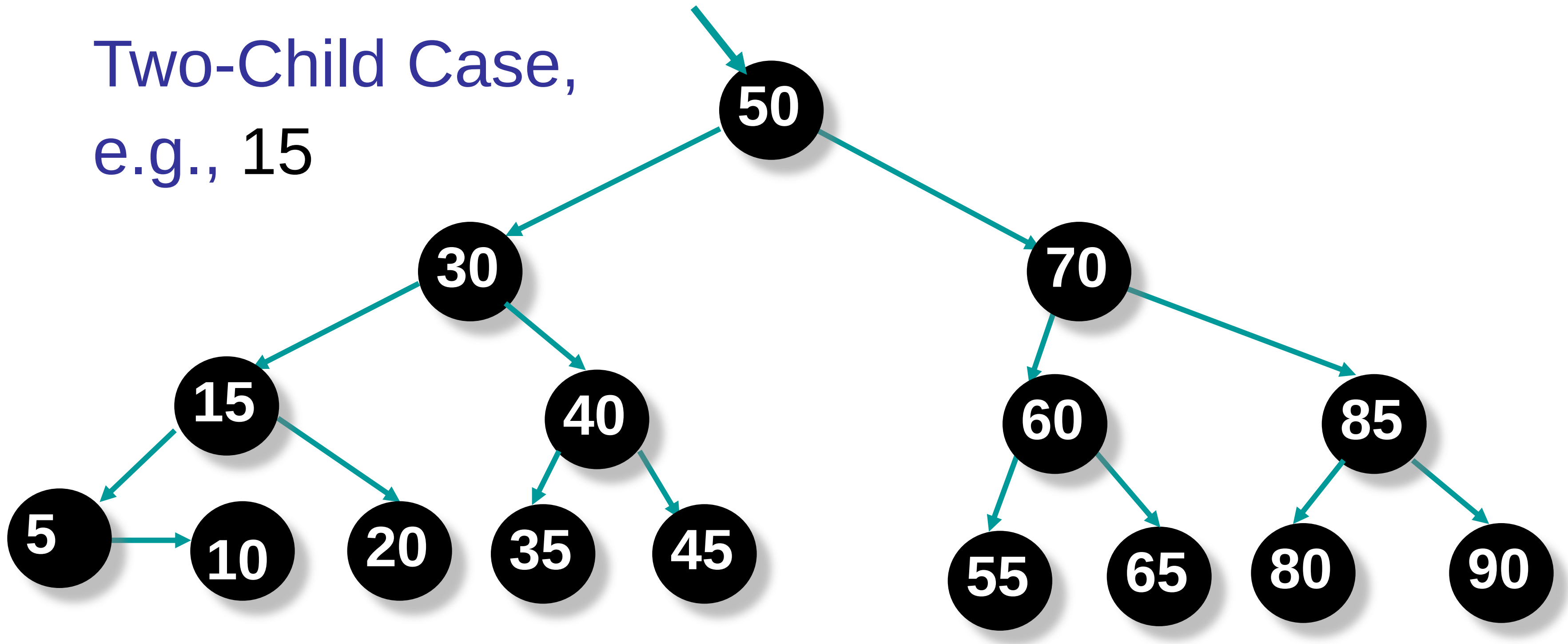
Insertion is complete (finally!)

AA-Tree Insertion Algorithm

```
// Inserts node y into AA-Tree rooted at node x
// Only for tree nodes with no pointer to parent
AAInsert ( x, y )
        if ( x = NIL )  // have found where to insert y
                then x ← y
        else if key[ y ] < key[ x ]
                then AAInsert( left[ x ], y )
        else if key[ y ] > key[ x ]
                then AATInsert( right[ x ], y )
        else
                y is a duplicate; handle duplicate case
        skew ( x )    // Do skew and split at each level
        split ( x )
```

Deletion

Two-Child Case,
e.g., 15

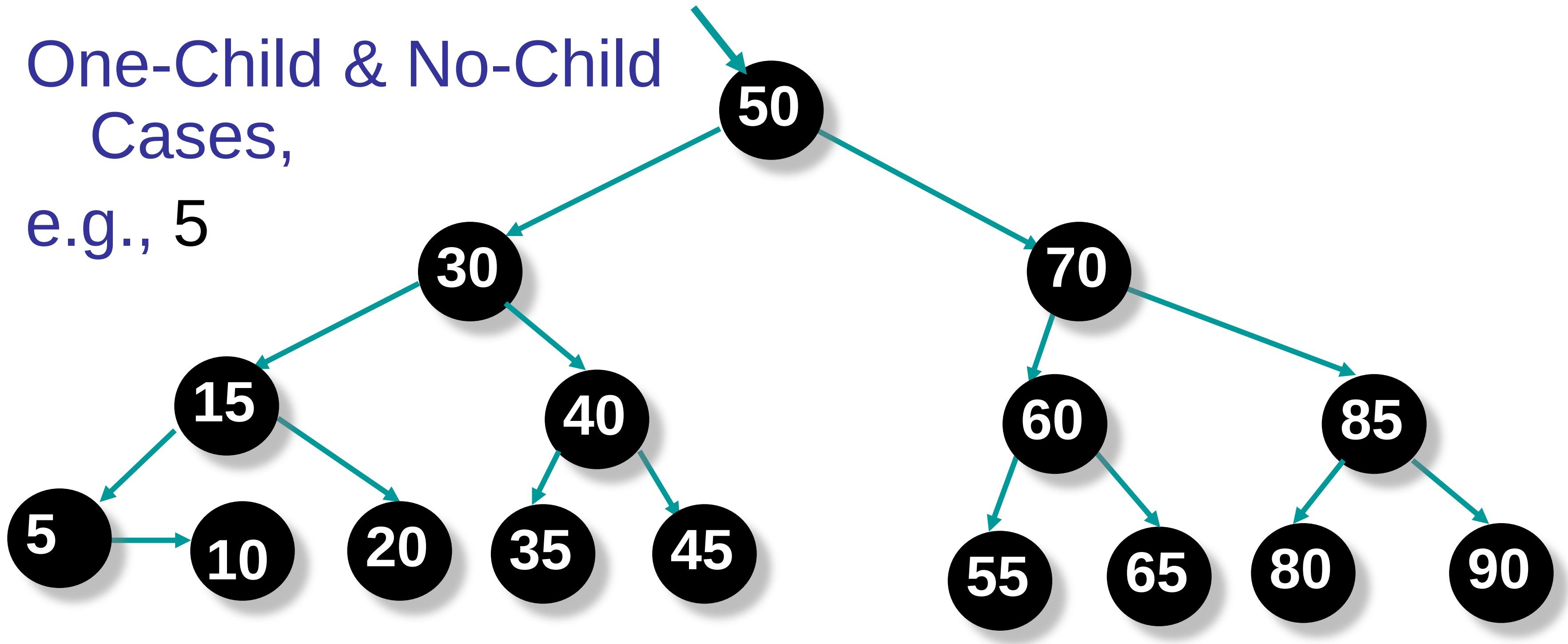


Same as for simple BST: replace with smallest right child or largest left child and recursively call delete

Deletion

One-Child & No-Child Cases,

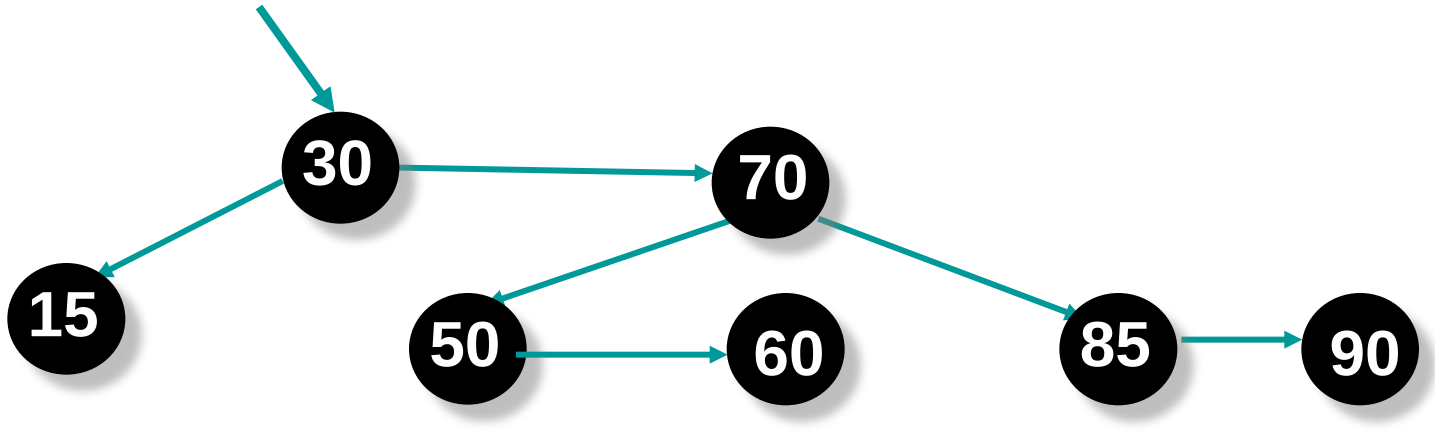
e.g., 5



Note that these are all at level one, so everything boils down to deleting a level one node

Deletion at Level 1

In the worst case, deleting one leaf node, e.g., 15, could cause six nodes to all become at one level, introducing horizontal left links.



However, it turns out that all cases can be handled by three calls to skew, followed by two calls to split (implementation can be found in various texts if you need it someday).

BST Applets

http://people.ksp.sk/~kuko/bak/index.html

http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

http://www.cis.ksu.edu/~howell/viewer/viewer.html