AA TREE

ADVANCED DATA STRUCTURE

21Z306 - ELAVENDHAN

21Z307 - HARRI SASTHAA

21Z344 - SANJAY

21Z371 - PRASUN JHA

21Z372 - NILARGHYA SARKAR

22Z463 – SHASHI PRAKASH

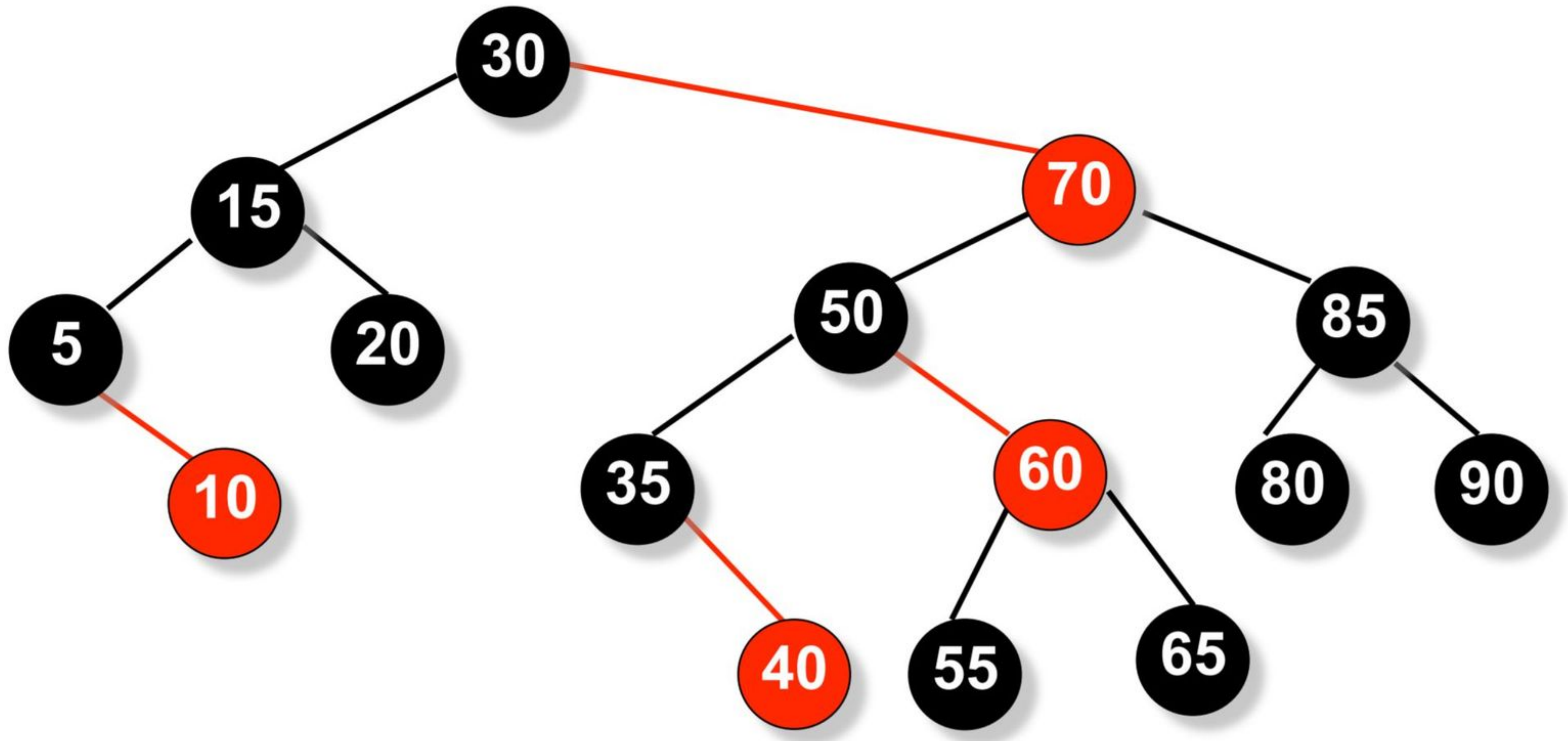
Introduction to AA trees

- AA trees were introduced by Arne Andersson in 1993 and hence the name AA. They are a type of balanced binary search trees. It was developed as a simpler alternative to Red Black trees.

- It eliminates many of the conditions that need to be considered to maintain a red-black tree.

AA Trees

- Unlike in red-black trees, red nodes on an AA tree can only be added as a right sub-child i.e. no red node can be a left sub-child. The tree below is an AA tree.



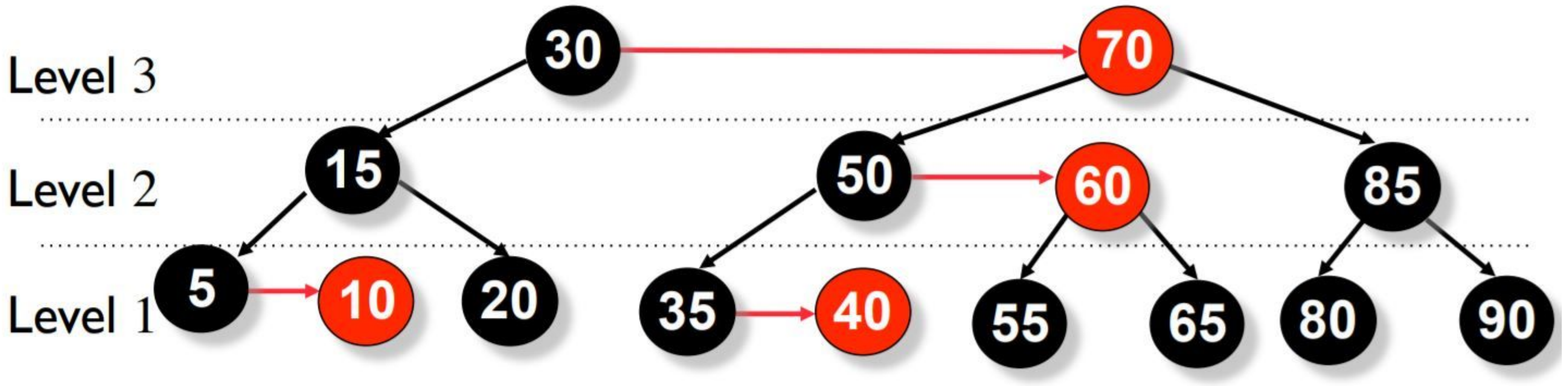
PROPERTIES OF AA TREE

- Every node is colored either red or black.
- The root is black.
- External nodes are black.
- If a node is red, its children must be black.
- All paths from any node to a descendent leaf must contain the same number of black nodes.
- Left children should not be red.

The following five invariants hold for AA trees

- The level of every leaf node is one.
- The level of every left child is exactly one less than that of it parent.
- The level of every right child is equal to or one less than that of its parent.
- The level of every right grandchild is strictly less than that of its grandparent.
- Every node of level greater than one, has two children.

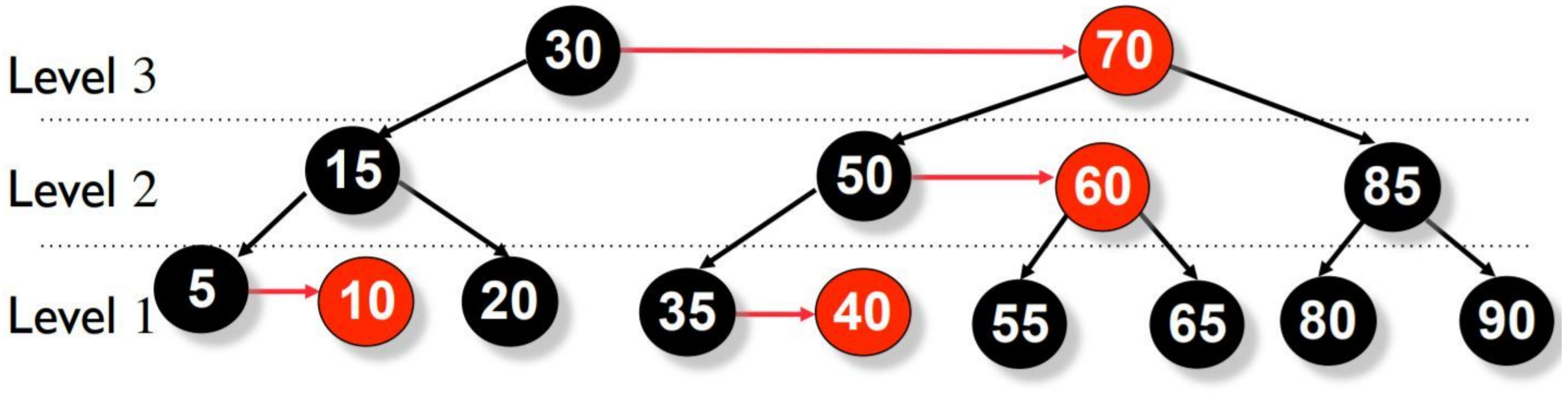
Example



LEVEL

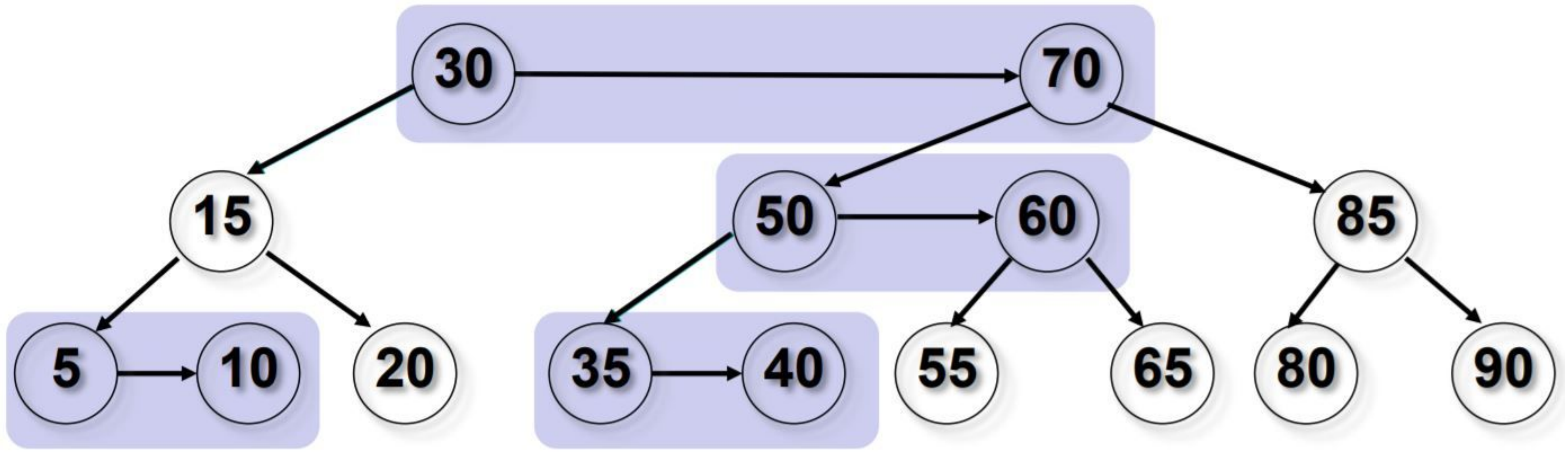
The level of a node in an AA-Tree is:

- 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


Level 3
Level 2
Level 1
30
70
15
50
60
85
5
10
20
35
40
55
65
80
90

Horizontal Link

- A link where the child's level is equal to that of its parent is called a horizontal link.
- Horizontal links are always right links.
- Red nodes are simply nodes that are located at the same level as their parents. For the AA tree shown above, the image below should help you understand which nodes are red and which are black and which are the horizontal links.



Operations on a AA tree

- Search Operation - O(log N)
- Insert Operation - O(log N)
- Delete Operation - O(log N)

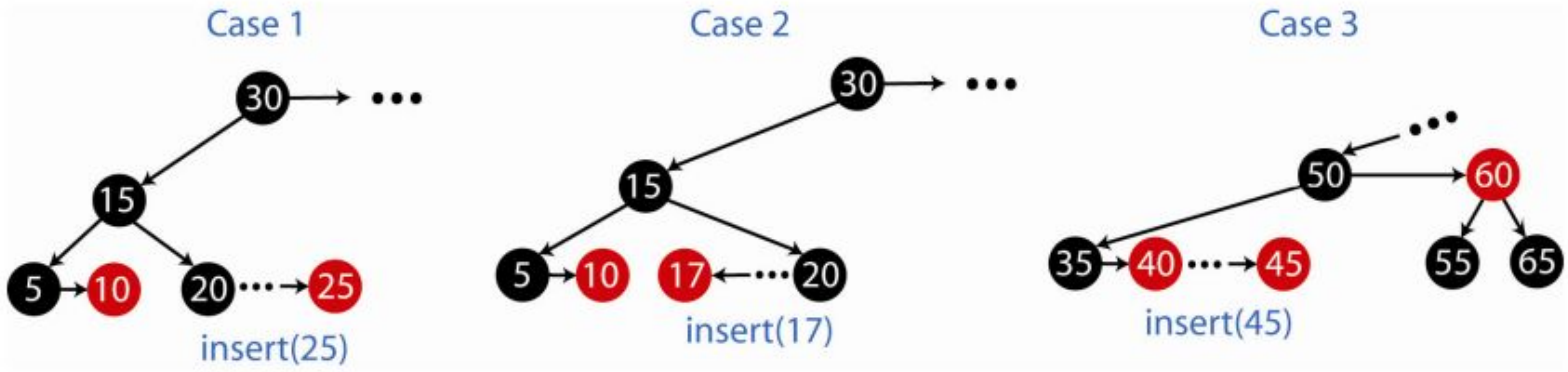
Search Operation

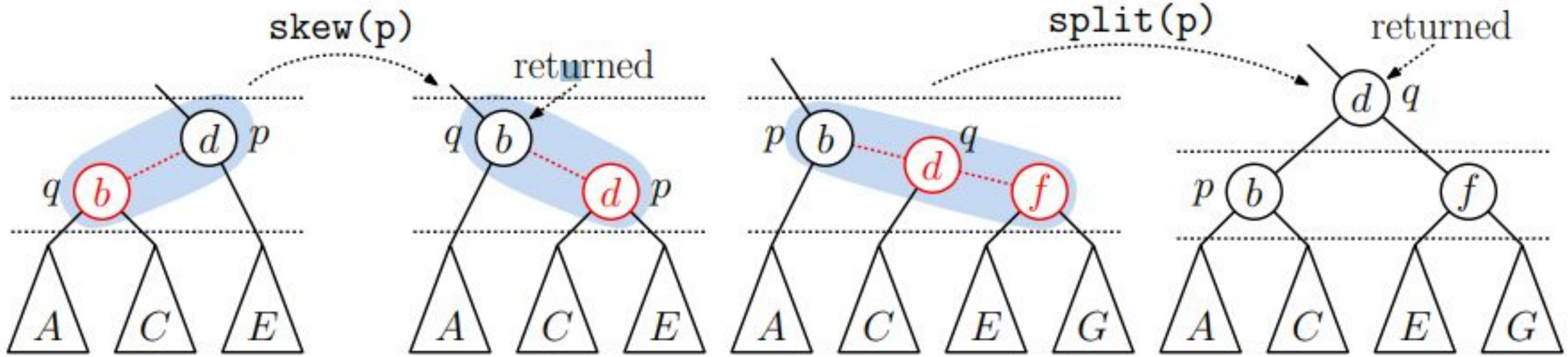
- Since an AA tree is essentially a binary search tree, the search operation is the same as that of a binary serach tree. The fact that it is a balanced binary search tree, just makes the search operation more efficient.

Insertion in AA trees:

Thus, when we insert a node, there are three cases:

- Case 1 – If the node to insert is to the right of its parent, then we insert it at the same level, as a horizontal (right) link. If the grandparent is at a higher level, then we are done.
- Case 2 – If the node to insert is to the left of its parent, then it will be at the same level as its parent, which is a violation. To fix a horizontal left link, we use a procedure called skew.
- Case 3 – If the grandparent is at the same level, then we have 2 consecutive horizontal links (reds), which is a violation. This is fixed by a procedure called split.


Case 1
30
15
5
10
20
25
insert(25)
Case 2
30
15
5
10
17
20
insert(17)
Case 3
50
60
35
40
45
55
65
insert(45)


skew(p)
returned
d
p
q
b
A
C
E
q
b
d
p
split(p)
p
b
d
q
f
G
returned
d
q
p
b
f

Algorithm for skew and split

```
AANode skew(AANode p) {
    if (p.left.level == p.level) {           // red node to our left?
        AANode q = p.left;                   // do a right rotation at p
        p.left = q.right;
        q.right = p;
        return q;                            // return pointer to new upper node
    }
    else return p;                           // else, no change needed
}

AANode split(AANode p) {
    if (p.right.right.level == p.level) {    // right-right red chain?
        AANode q = p.right;                  // do a left rotation at p
        p.right = q.left;
        q.left = p;
        q.level += 1;                        // promote q to next higher level
        return q;                            // return pointer to new upper node
    }
    else return p;                           // else, no change needed
}
```

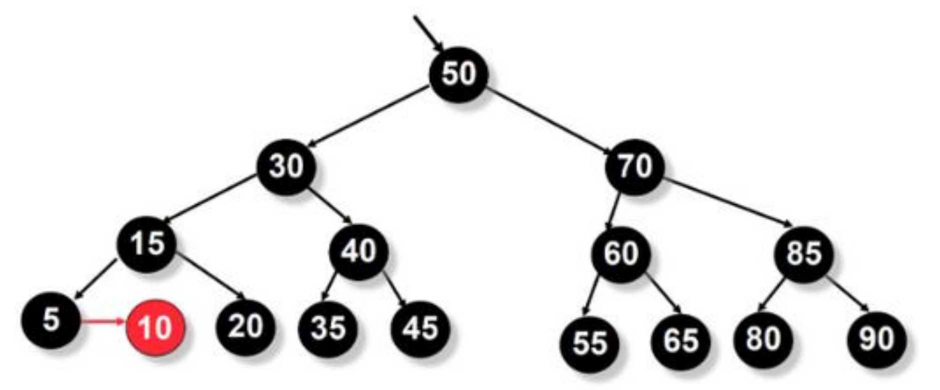
Algorithm for Insertion

```
AANode insert(Key x, Value v, AANode p) {
    if (p == nil)                               // fell out of the tree?
        p = new AANode(x, v, 1, nil, nil);      // ... create a new leaf node here
    else if (x < p.key)                         // x is smaller?
        p.left = insert(x, v, p.left);          // ...insert left
    else if (x > p.key)                         // x is larger?
        p.right = insert(x, v, p.right);        // ...insert right
    else
        throw DuplicateKeyException;            // duplicate key!
    return split(skew(p));                      // restructure and return result
}
```

Deletion in AA trees:

When deleting a node, we encounter the following three cases:

Note: After each replacement, perform the "fixupAfterDelete" operation to ensure tree properties are maintained.



- Case 1 - Delete a Red Leaf Node
- If the node to be deleted is a red leaf, just remove the leaf.
- Case 2 -Delete a Parent with a Single Red Internal Node
- If it is a parent to a single internal node, e.g. 5, it must be black. Replace with its child (must be red) and recolor child black.
- Case 3 -Delete a Node with Two Internal-Node Children

Subcase 3.1: In-Order Successor is a Red Leaf

- If the in-order successor is a red leaf, simply remove the leaf.

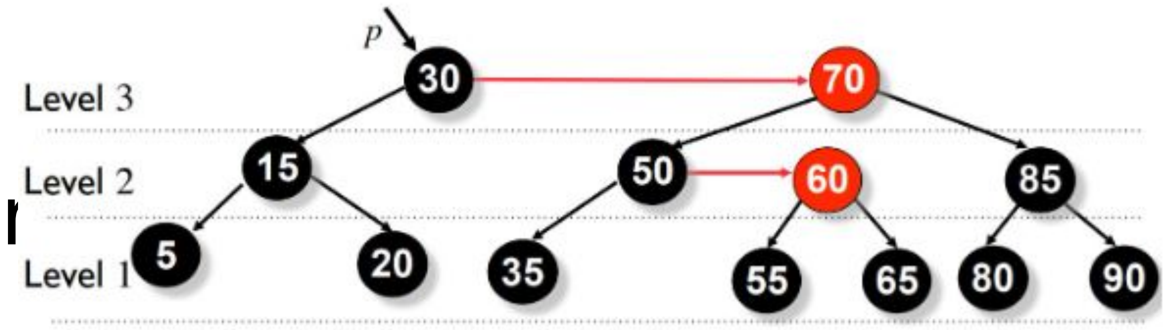
Subcase 3.2: In-Order Successor is a Parent with a Single Red Internal Node

- If the in-order successor is a single-child parent, apply Case 2.

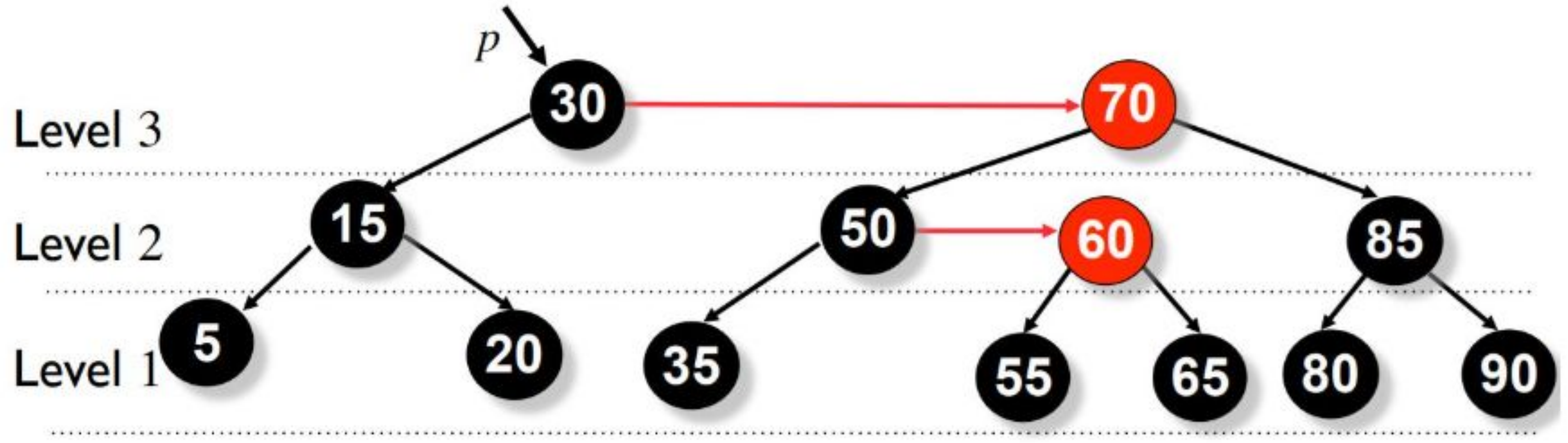
Subcase 3.3: In-Order Successor is a Black Leaf, or the Node to be Deleted is a Black Leaf

Then at each node "p" with two internal-node children, we will do the following steps:

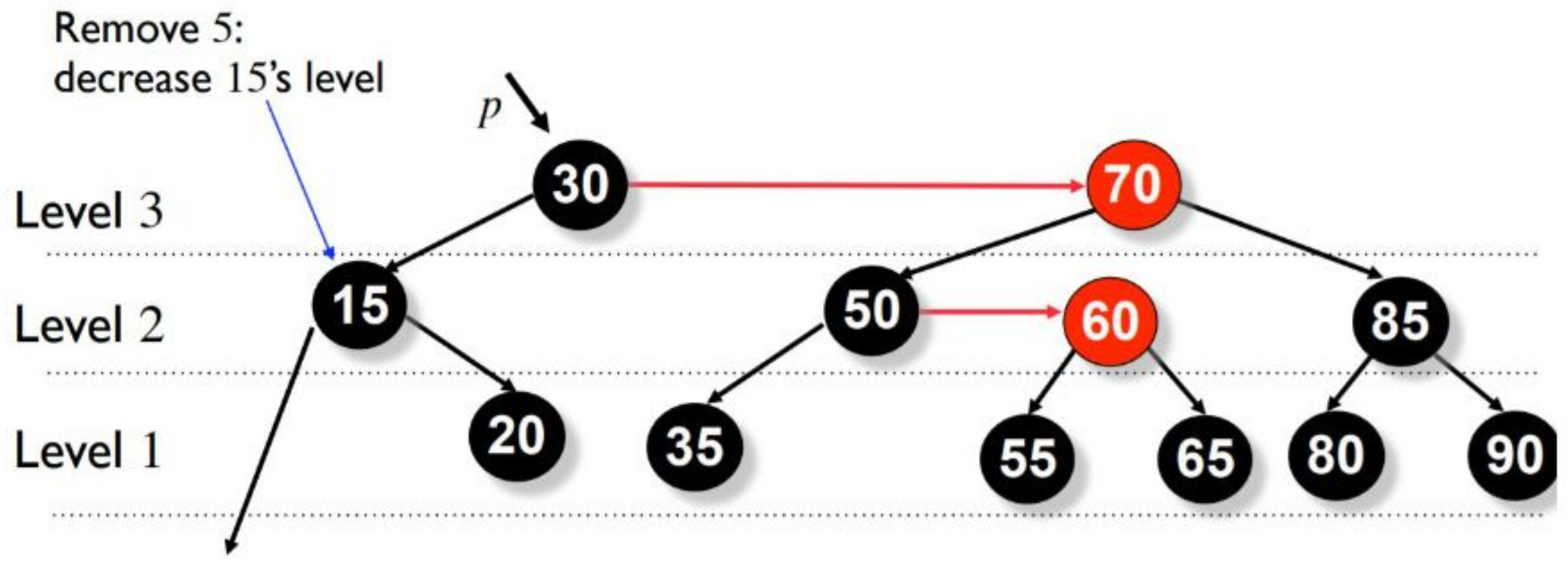
1. If either of "p's" children are two levels below "p," decrease the level of "p" by one.
2. If "p's" right child was a red node, decrease its level too.
3. Perform skew(p), skew(p->right), skew(p->right->right) oper
4. Perform split(p) and split(p->right) operations.



Example

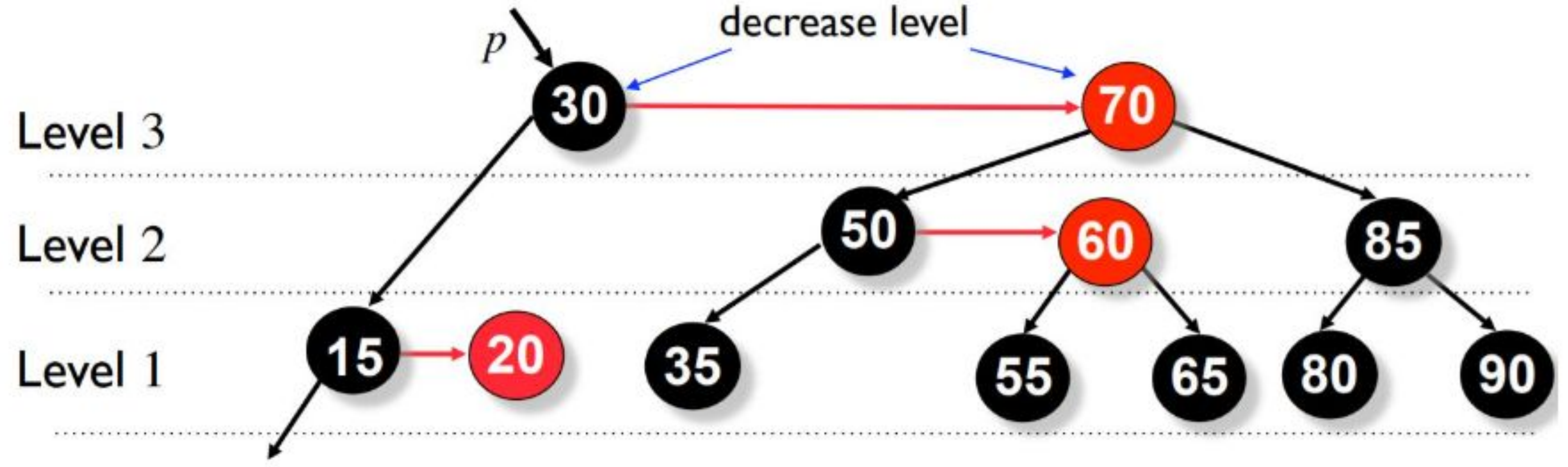


Delete 5

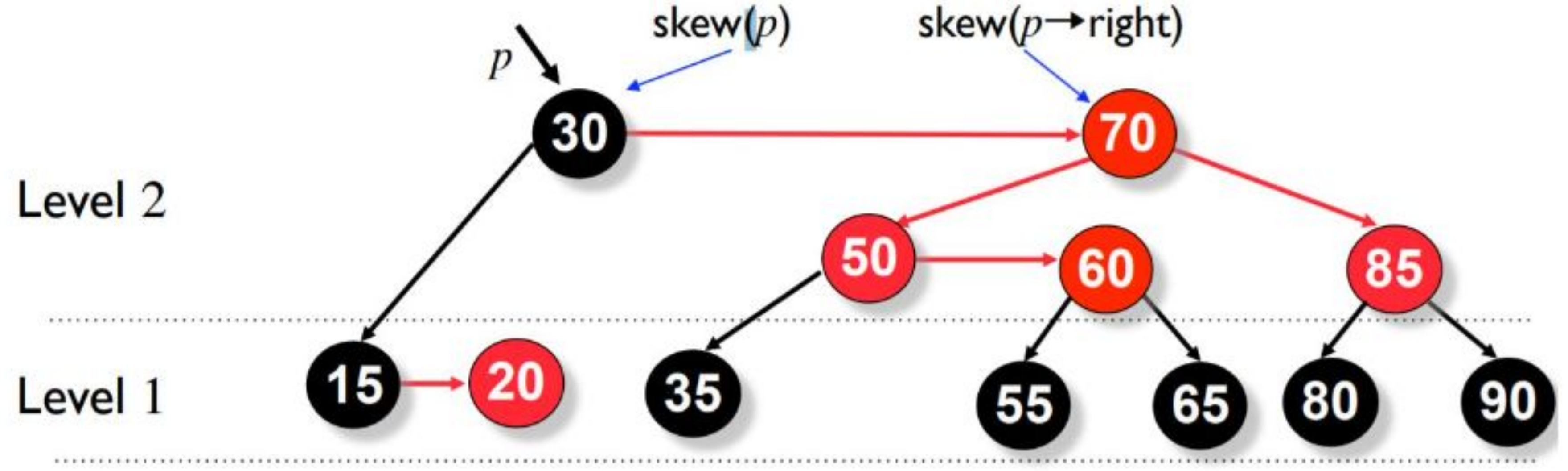


Now we decrease the level.

We follow the rules mentioned above and then decrese the level of **p**. Here **p** is a node with 2 internal children.

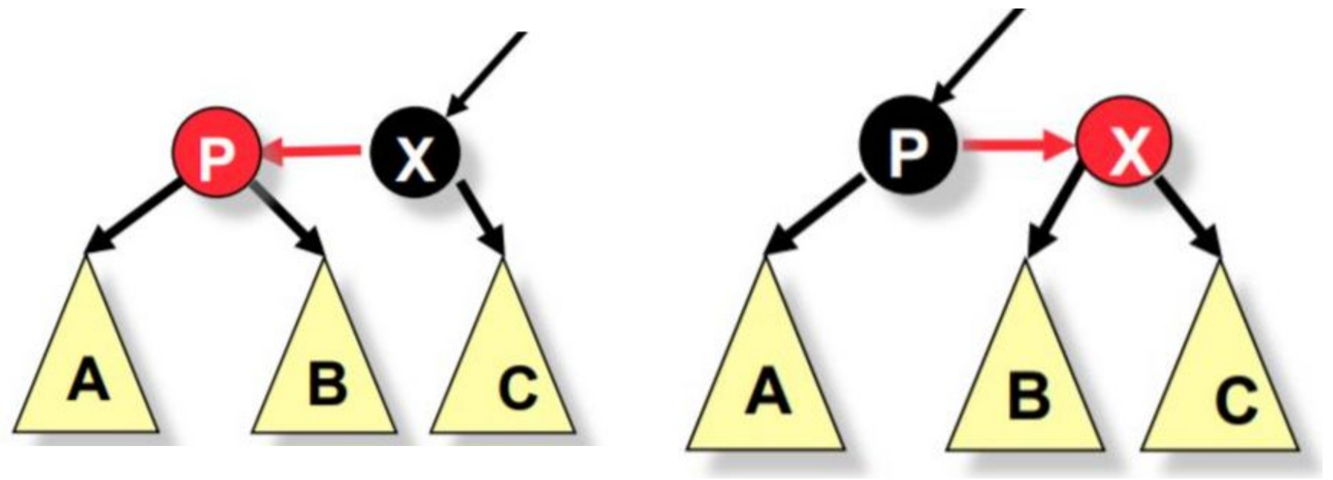


We now do the skew operation on p and p->right.

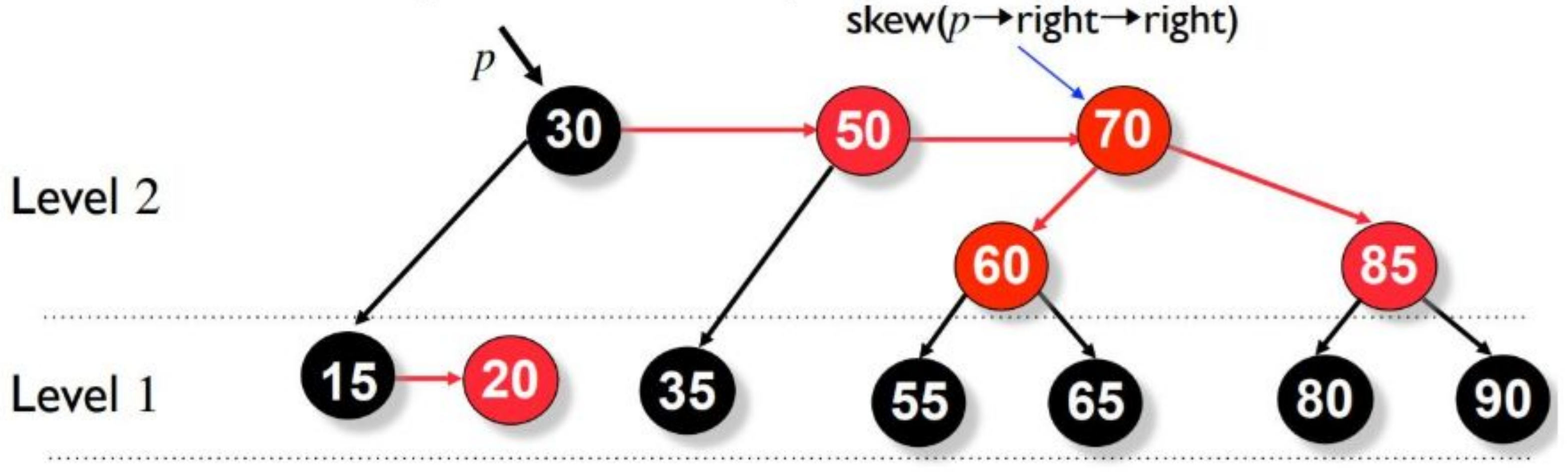


SKEW

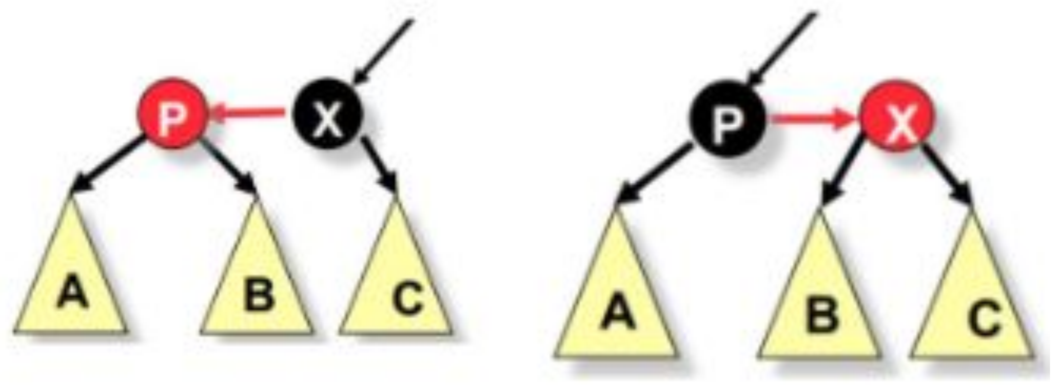
OPERATION



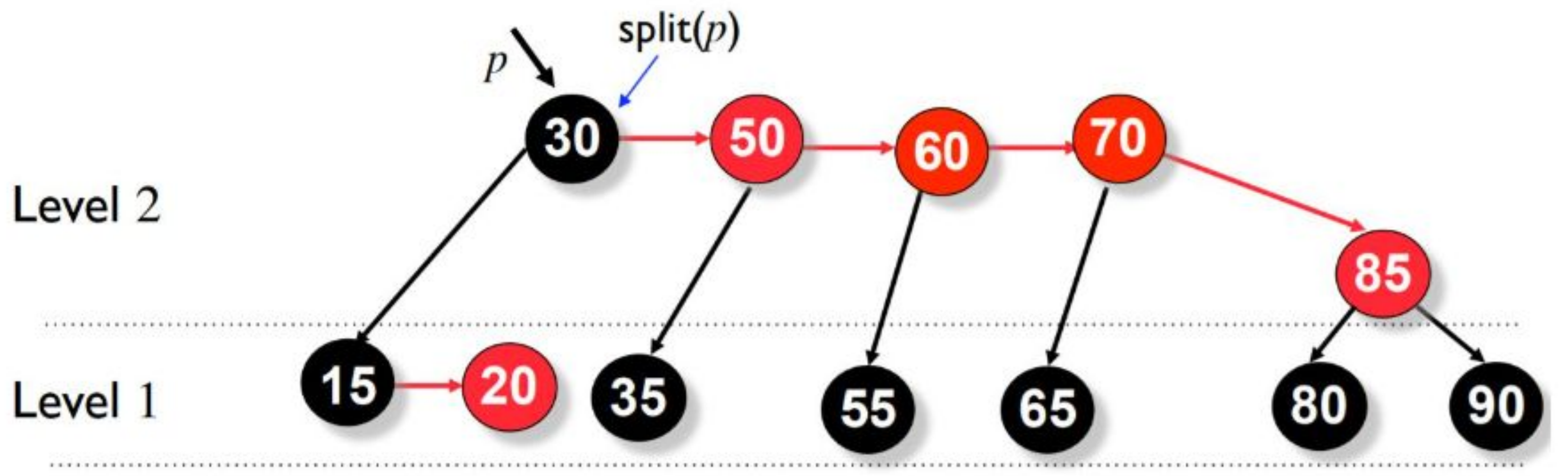
Now we do the skew operation on p->right->right.



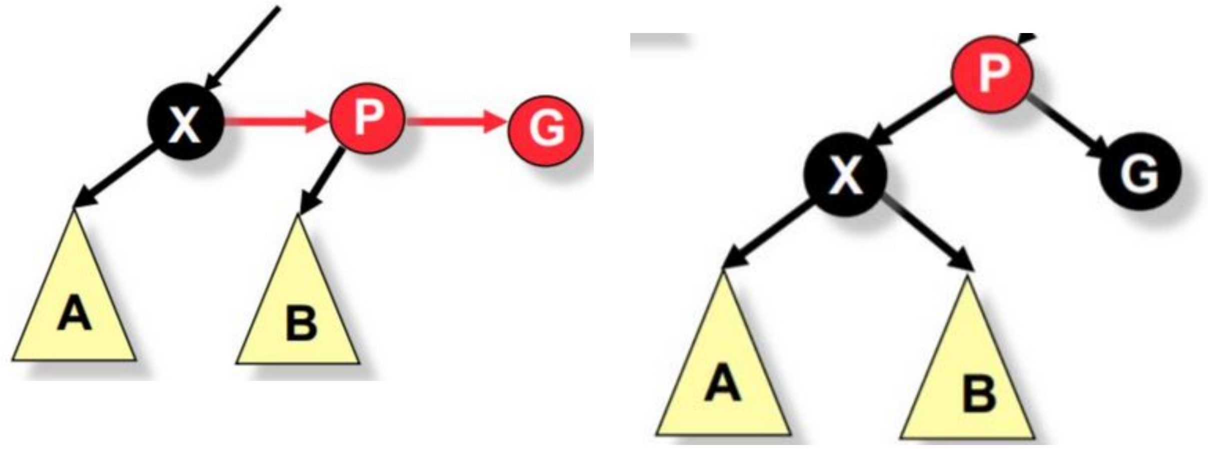
SKEW OPERATION



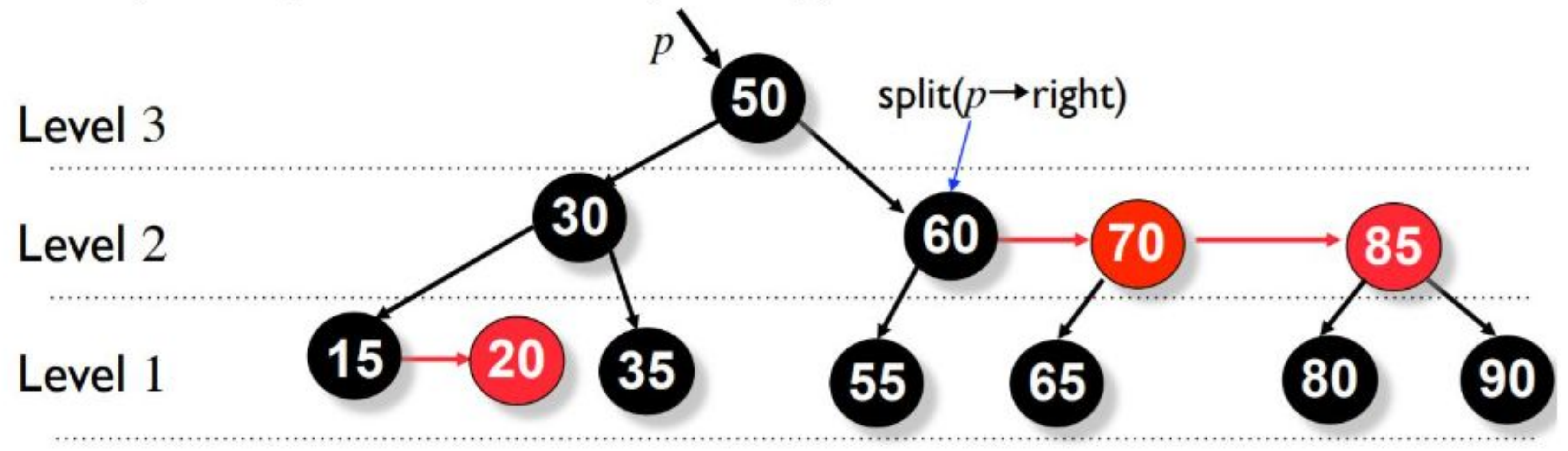
Now we do split operation on p.



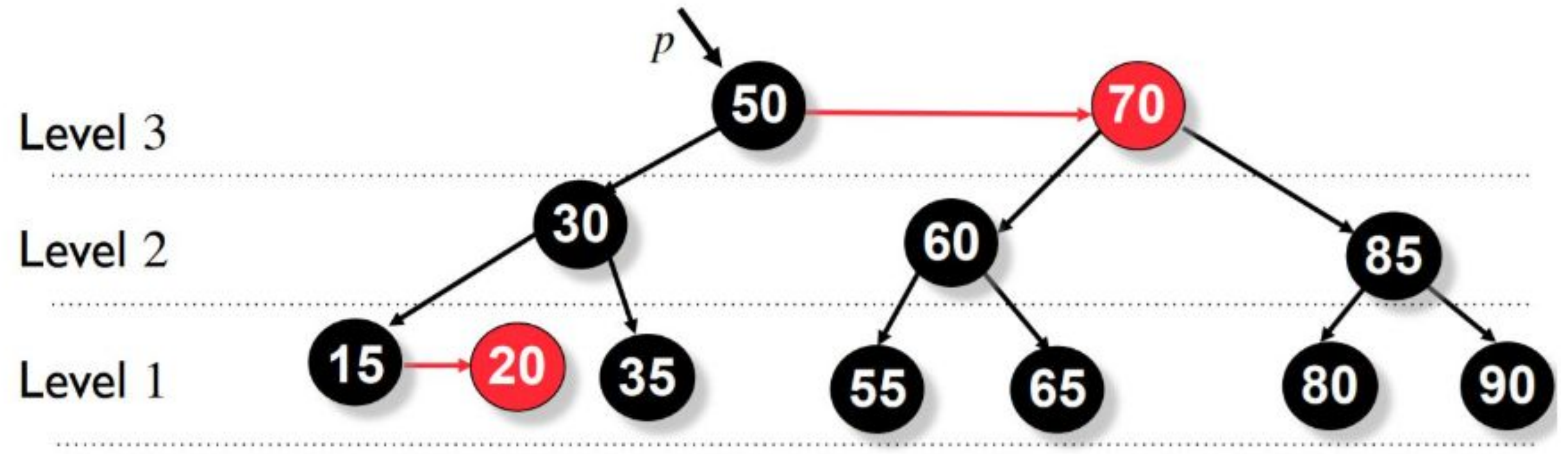
SPLIT OPEATION



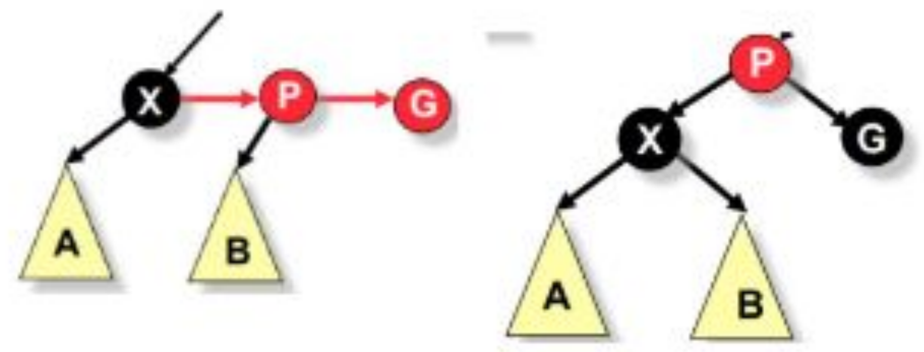
Now we do split operation on p->right.



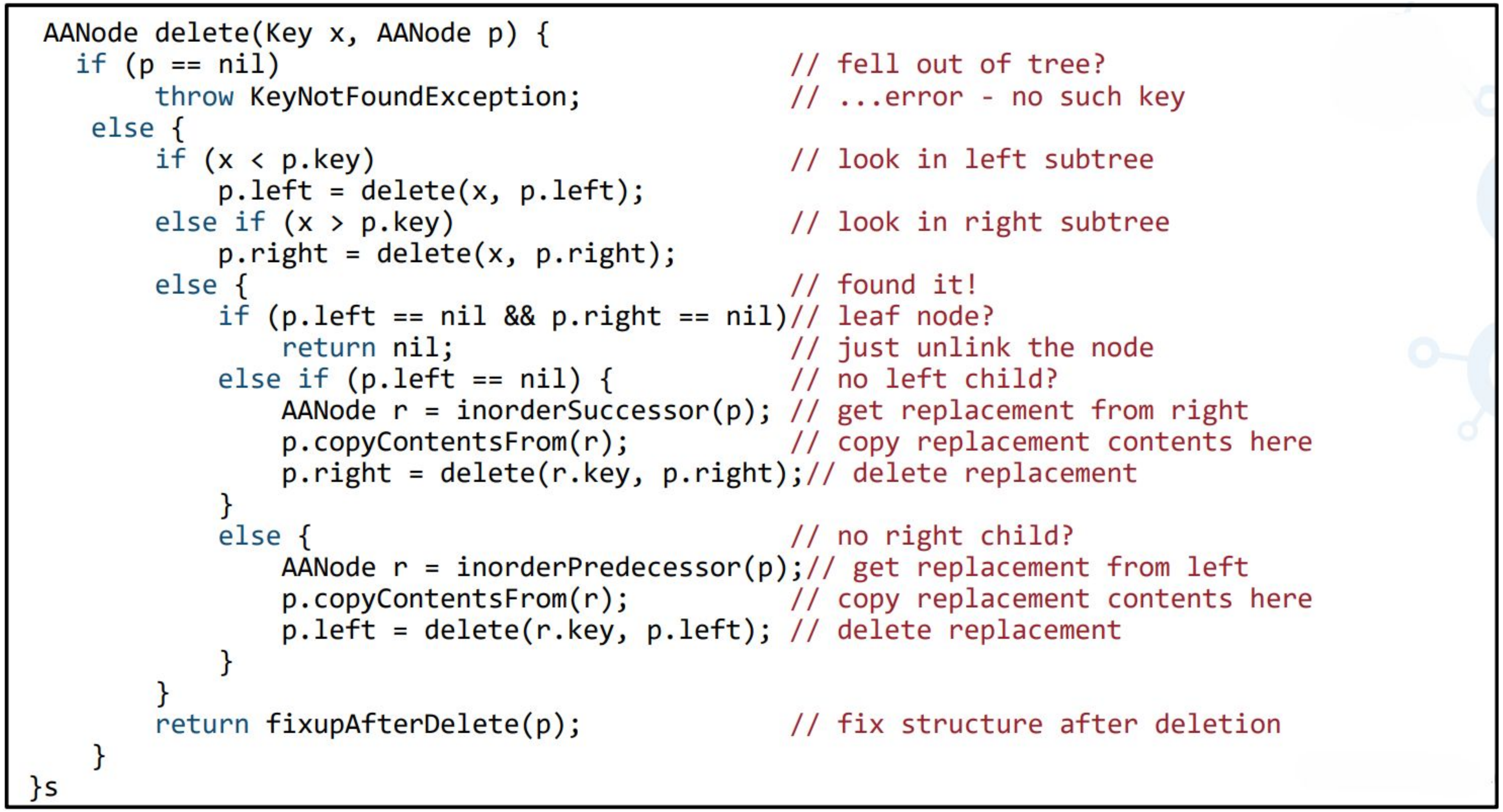
As you can see below, this tree is finally balanced and satisfies the properties of an AA tree.



SPLIT OPEATION



Algorithm Deletion



```
 AANode delete(Key x, AANode p) {
   if (p == nil)                                // fell out of tree?
        throw KeyNotFoundException;             // ...error - no such key
    else {
        if (x < p.key)                          // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.key)                     // look in right subtree
            p.right = delete(x, p.right);
        else {                                  // found it!
            if (p.left == nil && p.right == nil)// leaf node?
                return nil;                     // just unlink the node
            else if (p.left == nil) {           // no left child?
                AANode r = inorderSuccessor(p); // get replacement from right
                p.copyContentsFrom(r);          // copy replacement contents here
                p.right = delete(r.key, p.right);// delete replacement
            }
            else {                              // no right child?
                AANode r = inorderPredecessor(p);// get replacement from left
                p.copyContentsFrom(r);          // copy replacement contents here
                p.left = delete(r.key, p.left); // delete replacement
            }
        }
        return fixupAfterDelete(p);             // fix structure after deletion
    }
}s
```

Algorithm for updateLevel and fixupAfterDelete

```
AANode updateLevel(AANode p) {                          // update p's level
    int idealLevel = 1 + min(p.left.level,  p.right.level);
    if (p.level > idealLevel) {                         // p's level is too high?
        p.level = idealLevel;                           // decrease its level
        if (p.right.level > idealLevel)                 // p's right child red?
            p.right.level = idealLevel;                 // ...fix its level as well
    }
    return p;
}
```

```
AANode fixupAfterDelete(AANode p) {
    p = updateLevel(p);                                 // update p's level
    p = skew(p);                                        // skew p
    p.right = skew(p.right);                            // ...and p's right child
    p.right.right = skew(p.right.right);                // ...and p's right-right grandchild
    p = split(p);                                       // split p
    p.right = split(p.right);                           // ...and p's (new) right child
    return p;
}
```

Advantages of AA tree

- Efficient Balancing: AA trees are designed to be easy to implement while maintaining a balanced structure. They use a simple set of rules to keep the tree balanced, ensuring that the height remains relatively shallow. This balance results in efficient search, insertion, and deletion operations with a time complexity of O(log N).
- Simplicity of Implementation: Compared to some other self-balancing trees like AVL trees or Red-Black trees, AA trees have simpler rules and require less code complexity. This simplicity can make them easier to implement and debug.

- Memory Efficiency: AA trees often require less memory overhead than some other self-balancing trees, which can be beneficial when dealing with memory-constrained environments.
- Low Maintenance Overhead: AA trees are designed to maintain balance with relatively low overhead. This means that the balancing operations, such as rotations and level adjustments, are straightforward and do not significantly impact the performance of tree operations.

Applications of AA tree:

1. Dynamic Sets and Dictionaries: AA trees are used for maintaining dynamic sets of elements, where elements can be inserted, deleted, and searched for efficiently. They provide logarithmic time complexity for these operations.

2. Ordered Data Storage: AA trees can be used as ordered data structures to store elements in sorted order. This makes them useful for applications where you need to retrieve elements in a specific order, such as maintaining a sorted list of items.

3. Symbol Tables: Symbol tables, which are key-value stores, can be implemented using AA trees. The keys are stored in sorted order, and the tree structure allows for efficient key lookups.

4. File Systems: AA trees can be used in file systems to manage directory structures and maintain a hierarchical organization of files and folders. They help with efficient file lookups, additions, and deletions.

5. Cache Management: In memory management and caching, AA trees can be used to organize and manage cached data efficiently. When a cache has a size limit, an AA tree can be used to maintain frequently accessed items in an ordered structure.

6. Optimization Algorithms: AA trees are used in some optimization algorithms and data structures, such as the Dijkstra algorithm for finding shortest paths in a graph, to efficiently maintain priority queues.

THANK YOU