Data Structure and Algorithm

# Binary Search Tree
Balanced Tree

Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

- Red-Black Tree
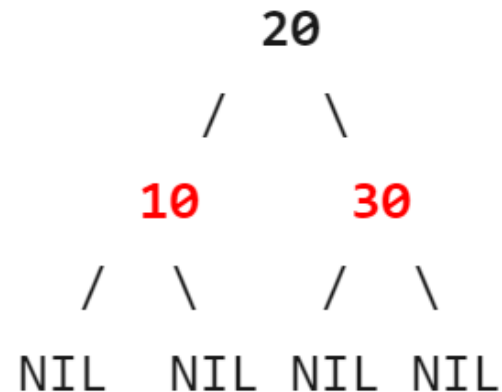- AA Tree

Red-Black Tree

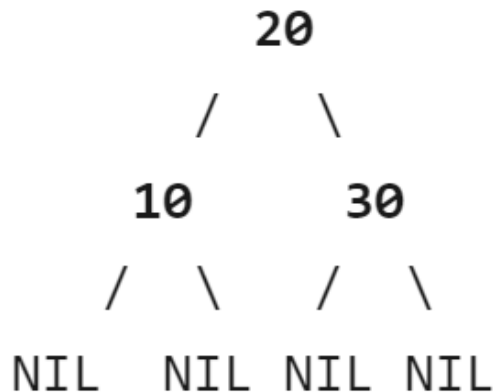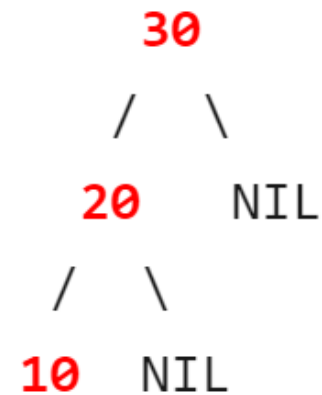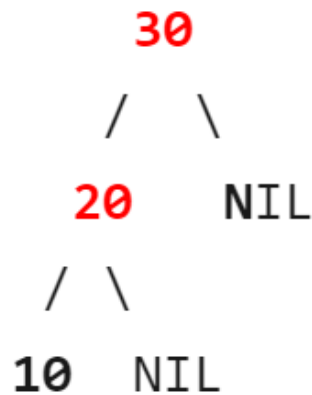- **Red-Black Tree** is a binary search tree which complies with the following rules :

  [1] Every node has a colour either red or black

  [2] The root of tree is always black

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

  [4] Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.
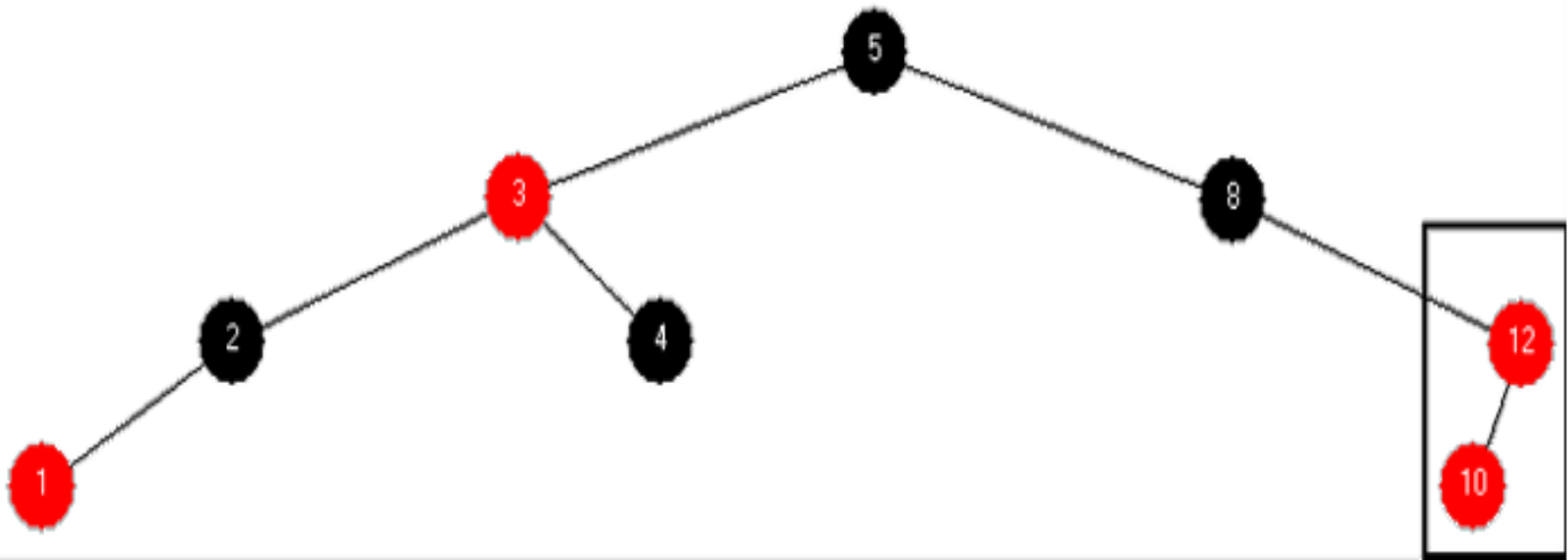
  [5] The NULL node is black.
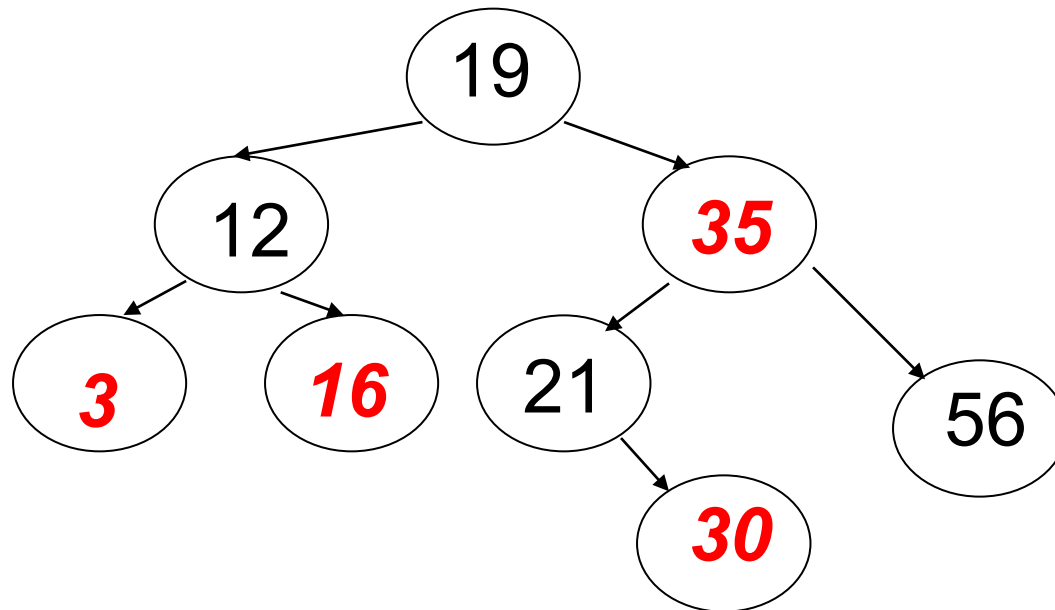
Example

- Which following trees are Red-Black Tree?
  - If not, why?

```
        30                    30                   30
       / \                   /  \                 /  \
     20   NIL              20     NIL            20      NIL
    / \                   / \                   /  \
  10   NIL              10   NIL              10   NIL


        20                                    20
       /   \                                 /    \
     10     30                             10      30
    / \    / \                            / \    / \
  NIL  NIL NIL NIL                      NIL  NIL NIL NIL
```
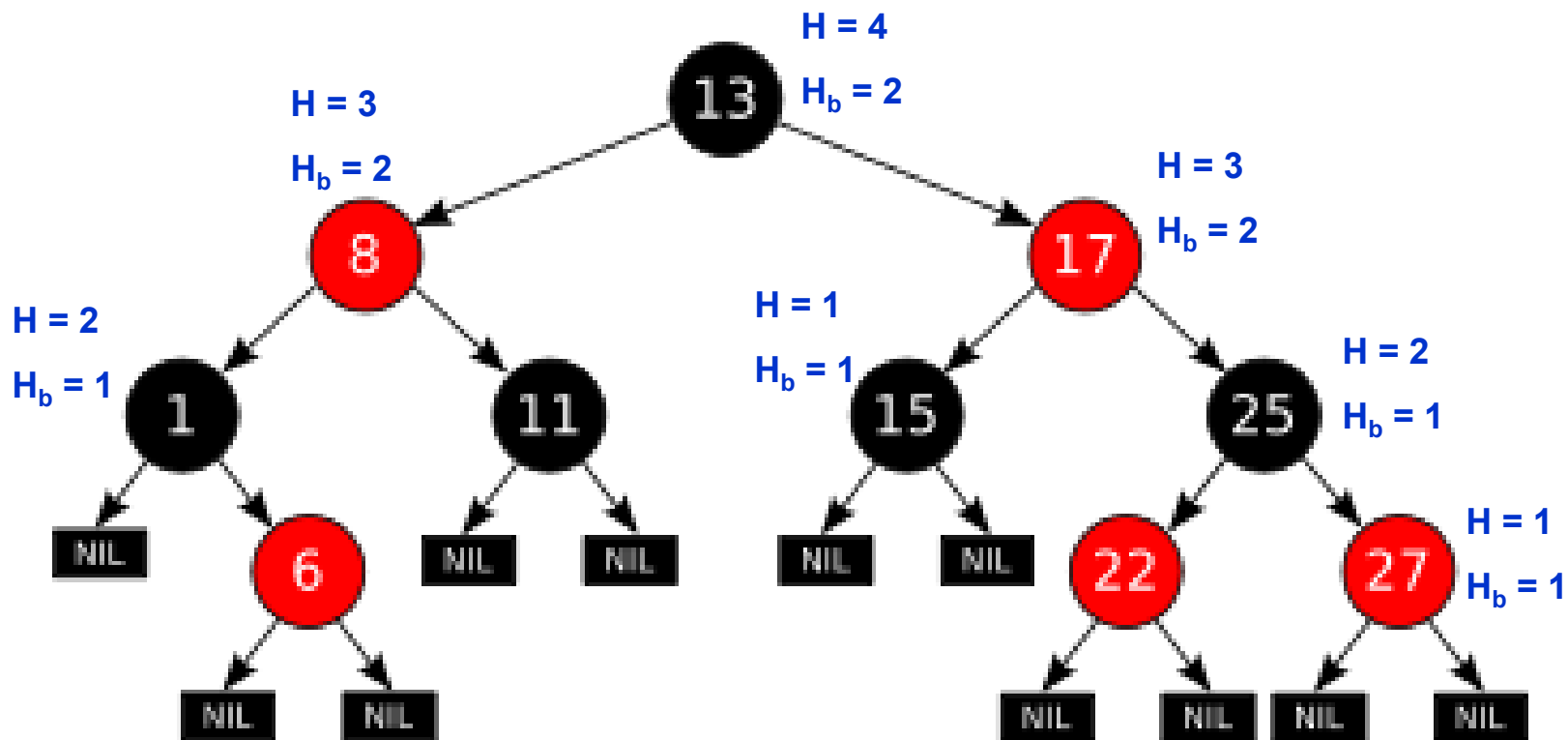
Consequences

- If a Red node has children, it must have 2 children and they must be Black. (Why?)

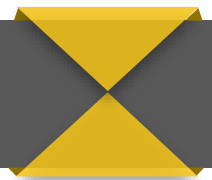- If a Black node has only one child, the child must be a Red leaf. (Tại sao?)

Red-Black True

- Black height – $h_b(x)$: is the number of black nodes on the path from node x to its descendant NULL node (x is not included).

```
typedef enum {BLACK, RED} NodeColor;

typedef int DataType;

typedef struct NodeTag {
    DataType            key;          // Data
    NodeColor           color;
    struct NodeTag    *pLeft;
    struct NodeTag    *pRight;
    struct NodeTag    *pParent;    //for easy traverse
} RBNode;

typedef struct RBNode* RBTREE;
```
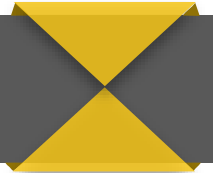
- Insert node:
  - Execute like binary tree search.
  - The new node is inserted at leaf node and assigned a Red color (Why?)
    - Rule 4
  - Check rules:
    - If the parent node is Black → it doesn't matter.
    - If the parent node is Red → violates parent-child are Red → adjust to balance tree

Insertion function

```
RB_Insert_Node(T, z)                     // T: tree; z: new node
    y ← NULL; x ← root[T];               // Travese tree to
    while x ≠ NULL {                     // go to leaf node
        y ← x                            // y: is parent node of x
        if (key[z] < key[x]) x ← left[x];
        else x ← right[x];
    }
    parent[z] ← y;                       // assign parent of z
    if (y == NULL) root[T] ← z;          // if parent is null -> z: root
    else if (key[z] < key[y]) left[y] ← z; //choose branch for z
        else right[y] ← z;
    left[z] ← NULL
    right[z] ← NULL
    color[z] ← RED                       // new node (z) is Red
    RB_Insert_FixUp(T, z)                // check and balance tree
```
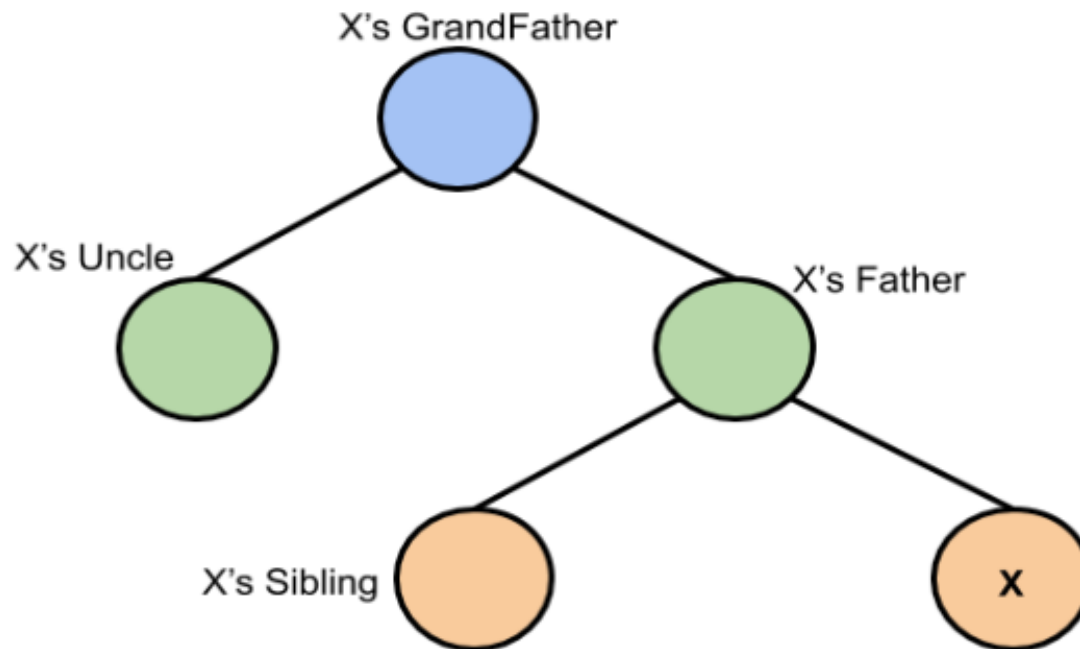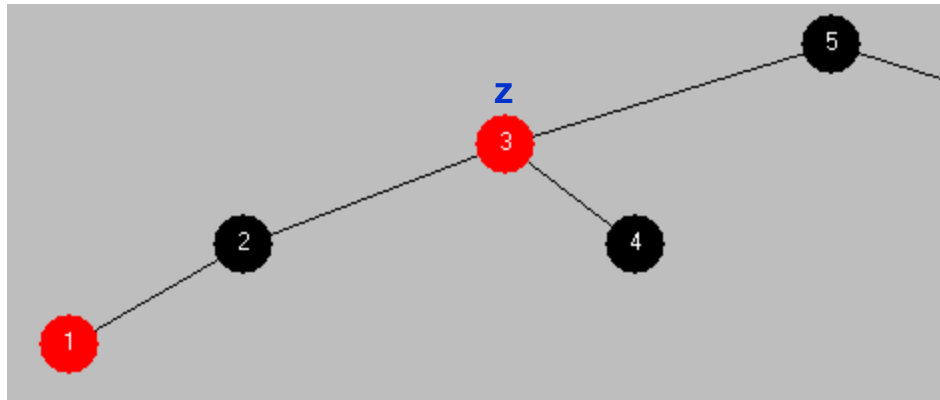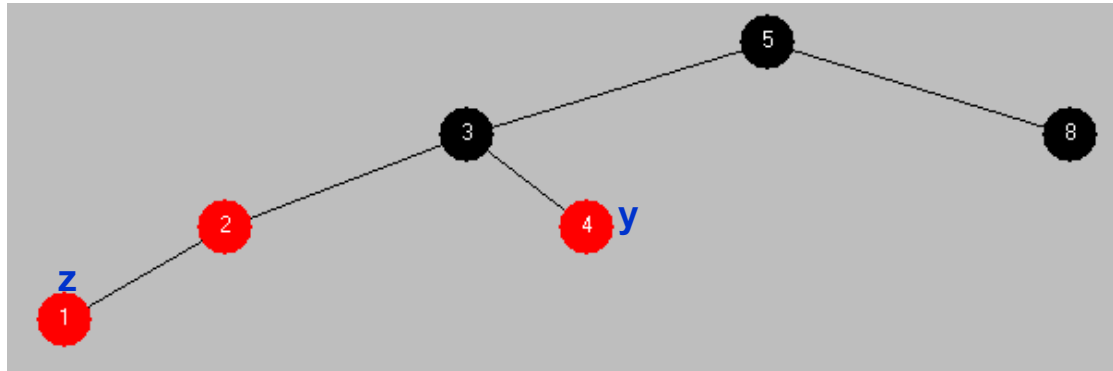
Balance tree

- There are 3 basic steps to do the balancing
  - Recoloring
  - Rotation
    - Left-Rotation
    - Right-Rotation

Recoloring

- Recoloring is the change in colour of the node.
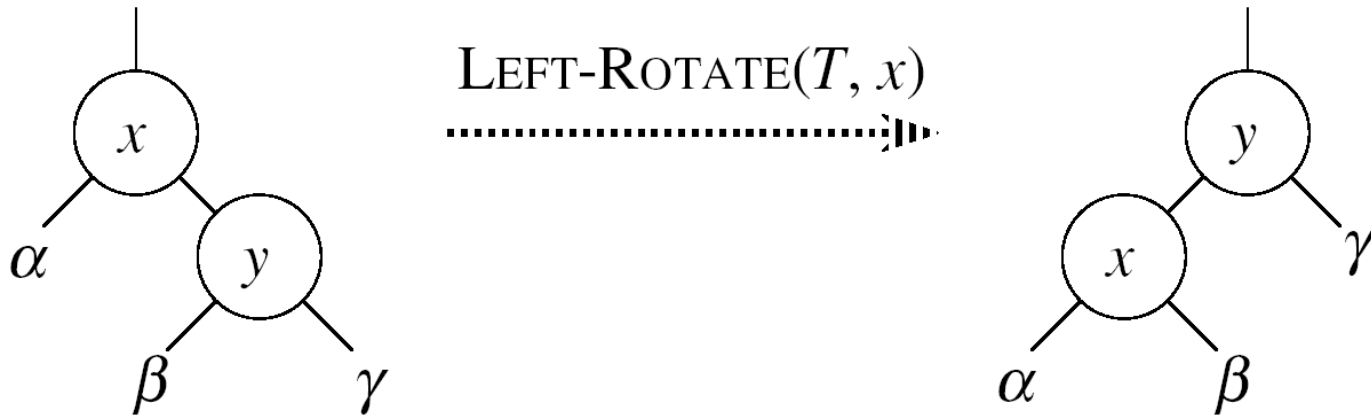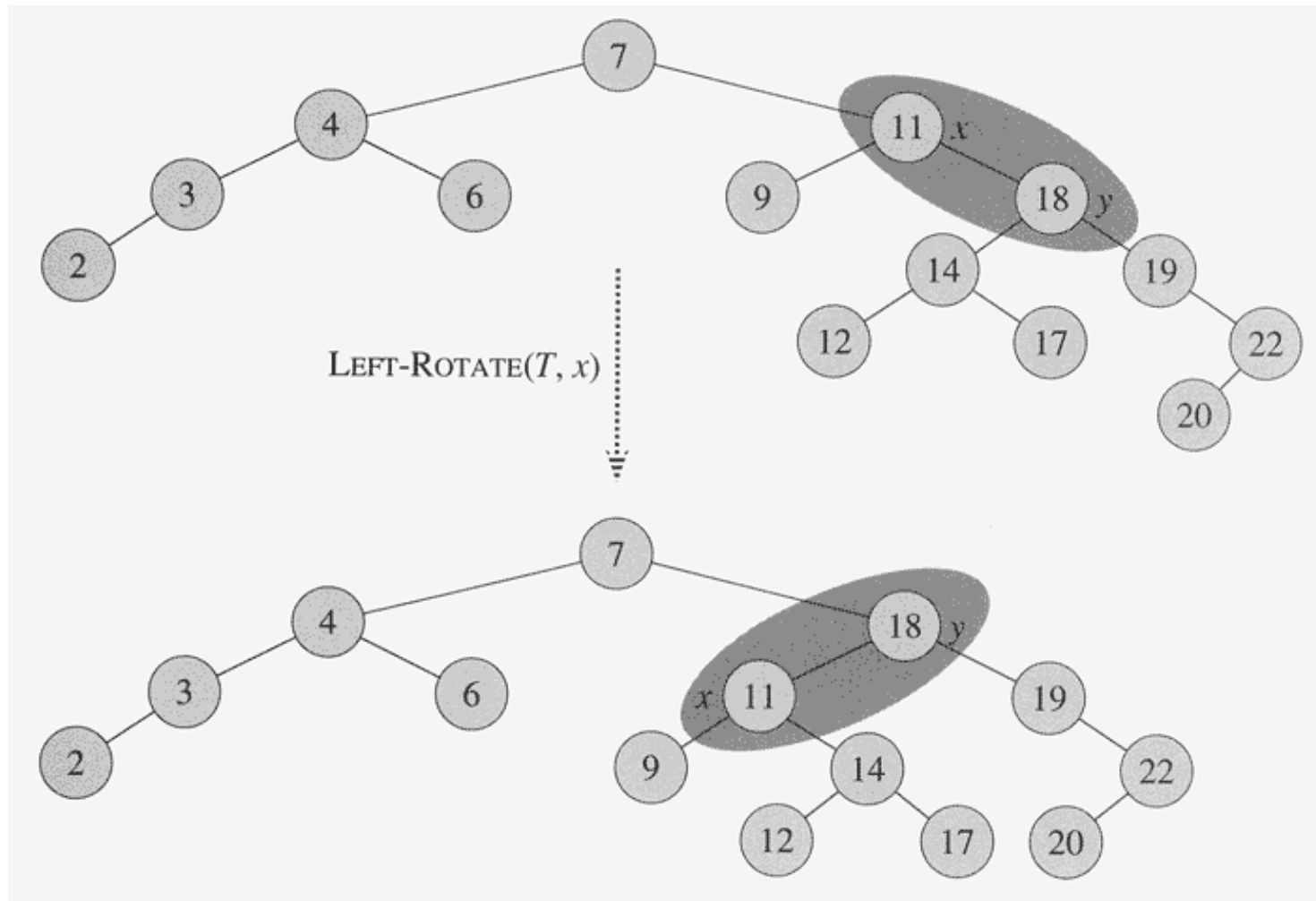 - If it is red then change it to black and vice versa.

Recoloring


```
color[parent[z]] ← black
color[y] ← black
color[parent[parent[z]]] ← red
z = parent[parent[z]]
```

- Left-Rotation:

$$\text{LEFT-ROTATE}(T, x)$$

Left-Rotation

$$\text{LEFT-ROTATE}(T, x)$$
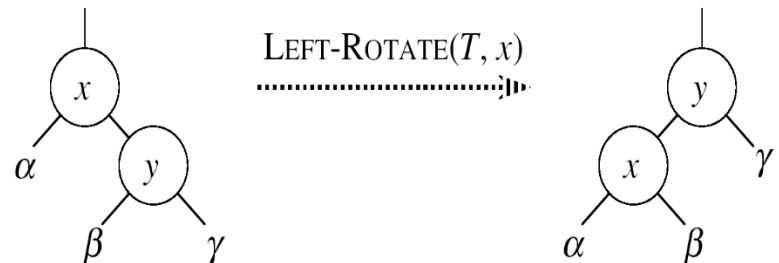
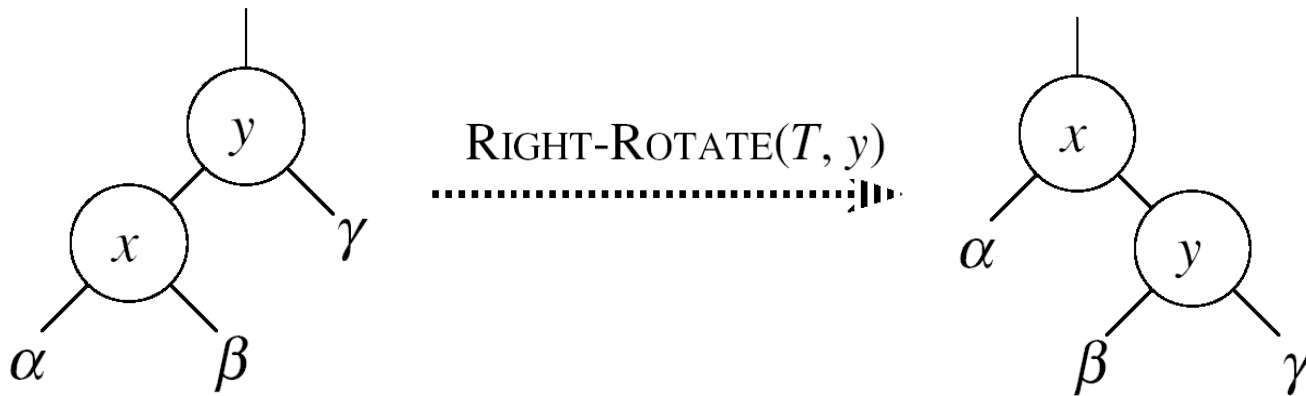Example

Left-Rotation

```
RB_Left_Rotate(T, x)
    y ← right[x];
    right[x] ← left[y];
    if (left[y] ≠ NULL) parent[left[y]] ← x;
    parent[y] ← parent[x];
    if (parent[x] == NULL) root[T] ← y;
    else if (x == left[parent[x]])
                left[parent[x]] ← y;
        else right[parent[x]] ← y;
    left[y] ← x;
    parent[x] ← y;
```
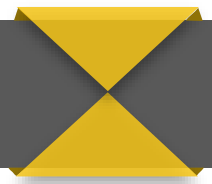
Right-Rotation

- Right-Rotation:

RIGHT-ROTATE(T, y)

- **RB_Right_Rotate(T, x):** similar to the left rotation function

Balance tree

- The algorithms have mainly two cases depending upon the colour of the uncle.
  - If the uncle is red, we do recolour.
  - If the uncle is black, we do rotations and/or recolouring.
- Base on two main cases, we list 6 smaller cases for easy control.
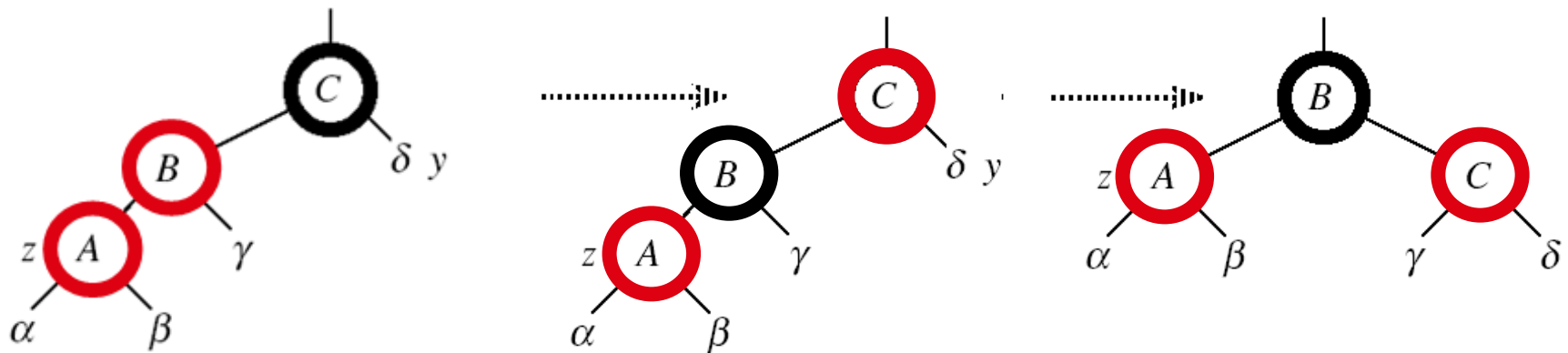
Balance tree

- Case 1: parent is red, uncle is red, it is red → Use recoloring method

If z is a right child
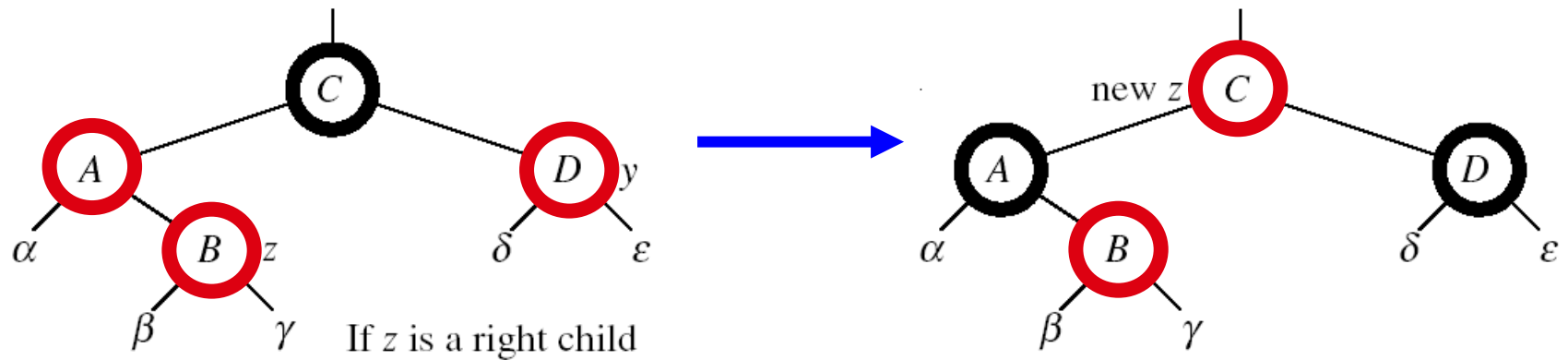
```
color[parent[z]] ← black
color[y] ← black
color[parent[parent[z]]] ← red
z = parent[parent[z]]
```
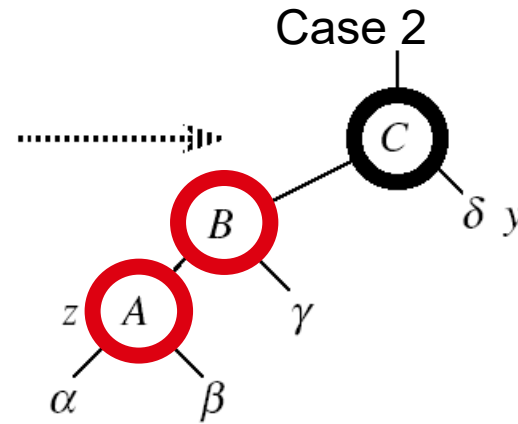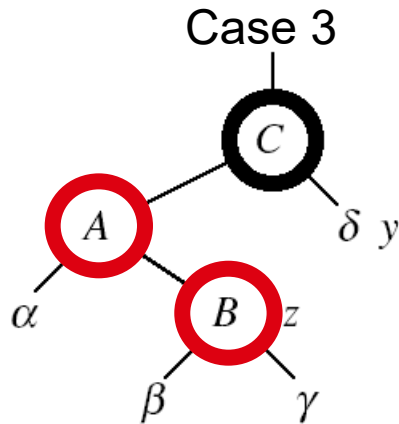
Balance tree

- Case 2: parent is red, uncle is black, it is a left child while parent is left child of grandfather (or otherwise) → Use recoloring and rotation method


```
color[parent[z]] ← black
color[parent[parent[z]]] ← red
RIGHT-ROTATE(T, parent[parent[z]])
```

Balance tree

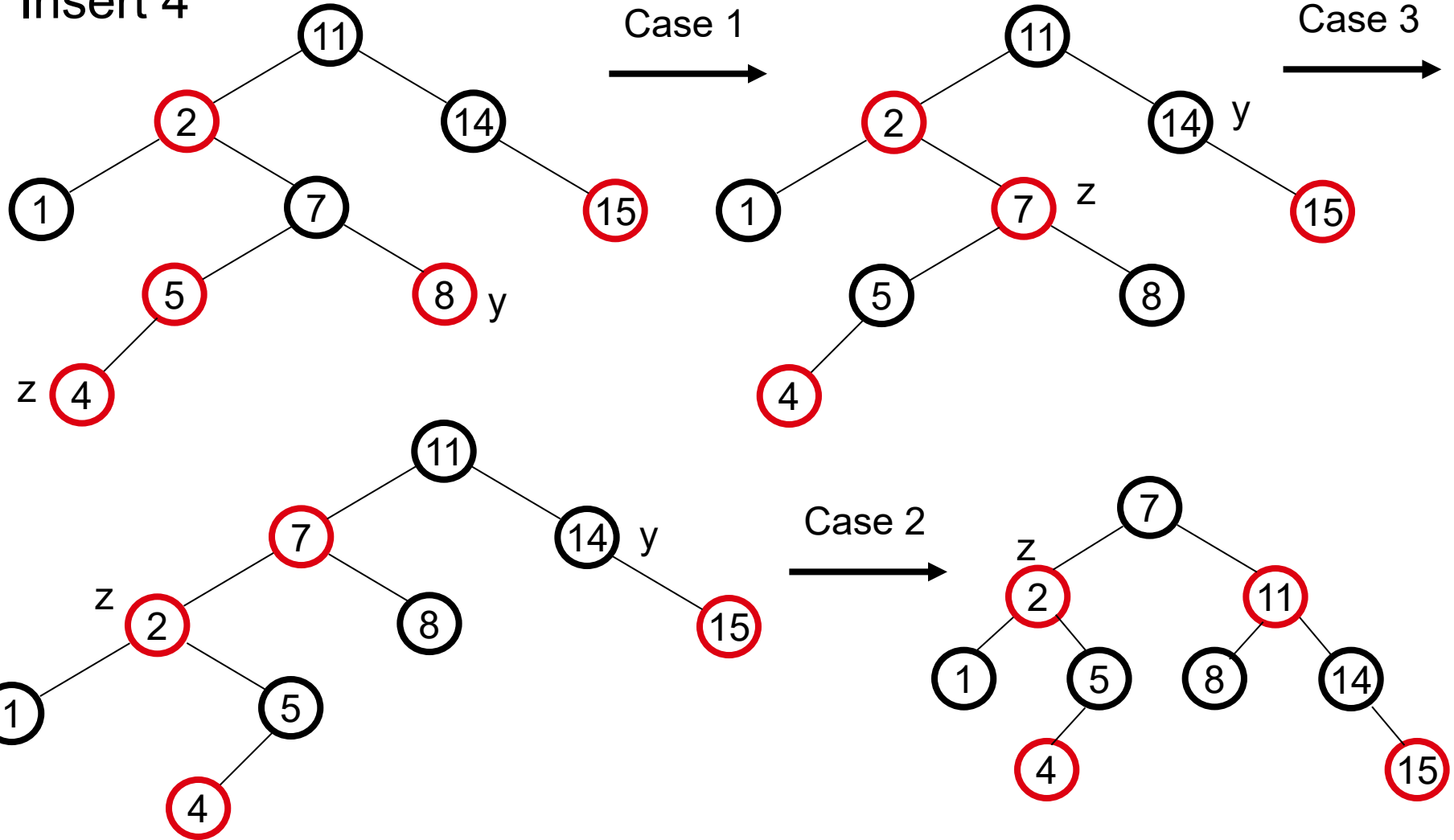- Case 3: parent is red, uncle is black, it is a right child while parent is left child of grandfather (or otherwise) -> rotate to convert to case 2

Case 3

Case 2

```
z ← parent[z]
LEFT-ROTATE(T, z)
"Treat as the case 2"
```

Example

Insert 4

Case 1

Case 3

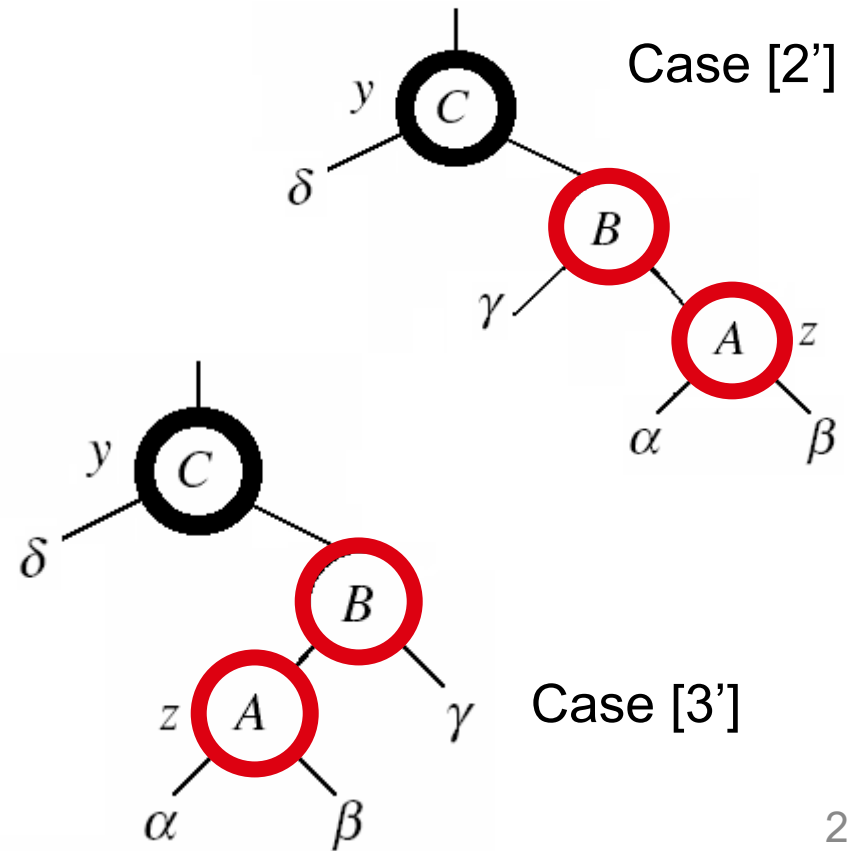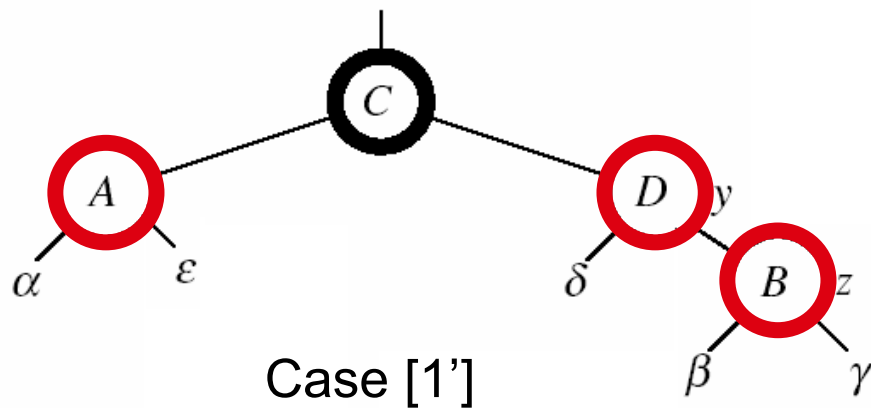Case 2

Balance tree

- Remain cases:
  - 3 similar cases [1'], [2'], [3'] are symmetrical with [1], [2], [3] through the y axis

Case [1']

Case [2']

Case [3']

Balance tree

```
RB_Insert_FixUp(T, z)
    while (parent[z] != NULL && color[parent[z]] == RED)
    {
        // Case [1], [2], [3]
        if (parent[z] == left[parent[parent[z]]]) {
                y ← right[parent[parent[z]]];
                if (color[y] == RED) "Case 1";
                else {
                    if (z == right[parent[z]]) "Case 3";
                     else "Case 2";
                }
        }
        else …// case 1'], [2'], [3']
    }
    color[root[T]] ← BLACK
```

Comments

- Evaluate the Insert node operation :

    – The cost of adding a new element (z): $O(log_2N)$

    – Cost of RB_Insert_FixUp: $O(log_2N)$

    – Total cost: $O(log_2N)$

Delete a node

- Delete a node:
    - Execute like binary tree search.
    - However:
        - For the case of 2 children, only the value is replaced, not the color.
    - After deleting, there are 2 cases to consider :
        - The actually deleted node is the red node
        - The actually deleted node is the black node

26

Delete a node

- Delete a node:
  - If the actually deleted node is red: no violation
    - Every node has a colour either red or black → OK
    - The root of tree is always black → OK
    - The NULL node is black → OK
    - If a node is red, its children must be black → OK because do not create 2 red consecutive nodes
    - Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes → OK because it doesn't change the number of black nodes

Delete a node

- Delete a node:
  - If the actually deleted node is **black**: a violation may occur
    - Every node has a colour either red or black → OK
    - The root of tree is always black → not OK ! Because it can delete root and replace it with a red node
    - The NULL node is black → OK
    - If a node is red, its children must be black → not OK ! Because it is possible to create 2 consecutive red nodes
    - Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes → not OK ! Because it reduces the number of black nodes

Delete a node

- So, before removing the actual deleted node:

    – If the actually deleted node is red, no adjustment is required

    – If the actually deleted node is **black**, it will need to be adjusted to rebalance the tree

Balance

- Delete the black node:
  - Comment: *if the actual black node is deleted, there can be at most 1 child*.
  - We divide it into 2 cases:
    - **Case 1**: the child of the delete black node is in red
    - **Case 2**: the child of the deleted black node is in **black**
      - Note that the null node is black, so in the case of the deleted node there are no children, we still consider that it still has black children.

- Case 1: child c of the deleted black node n is red
  - After deleting node n, we just need to change the color of child node c to black.

Case 1
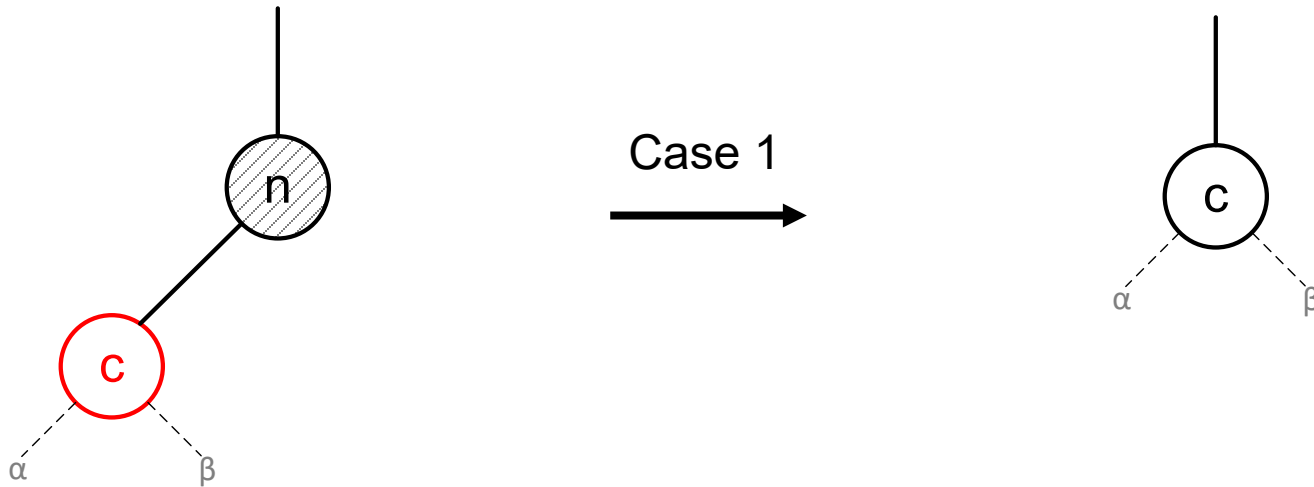
Case 2: black child

- Case 2: child c of the deleted black node n is black.
  - Let s be a sibling node of n, p be the parent node of n.
  - We will in turn perform the following steps:
    - Step 2.1: If n is the root then end. If n has a father then go to the next step.
    - Step 2.2: If s is red, invert parent and sibling colors. Then left rotate tree at p. Go to the next step.
    - Step 2.3: If s is black, the children of s are black and parent of p is black, we change the color s to red. Go back to step 2.1 with new node n is the parent of the current n (correction propagation).
    - Step 2.4: If s is black, the children of s are black but p is red, we reverse the parent and sibling color. Go to **end**.
    - Step 2.5: If s is black, the left child of s is red, the right child of is black, and n is its father's left child, we change color of s with its left child and right rotate at s. At this point, node n will have new siblings. Go to the next step.
    - Step 2.6: If s is black, the right child of s is red and n is its father's left child, we reverse the s and p color, coloring black for the right child of s. Left rotate at p. Go to **end**.
  - Note:
    - *In steps 2.2, 2.5 and 2.6, we will do the same but vice versa if n is the right child of its father.*
    - *Deletion of n, only performed when the steps are completed and n is the actual n need to be deleted, not n of the correction propagation processes.*

```
void replace_node(node* n, node* child) {
    child->parent = n->parent;
    if (n == n->parent->left)
        n->parent->left = child;
    else
        n->parent->right = child;
}

void delete_one_child(node* n)
{
    if (n->color == BLACK) {
        if (child->color == RED)
                child->color = BLACK;
        else
                processCase2(child);
    }

    node* child = is_leaf(n->right) ? n->left : n->right;
    replace_node(n, child);
    delete n;
}
```
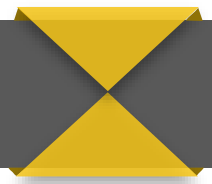
33

Pseudo Code

```
void processCase2(node* n)
{
    doStep2_1(n);
}
```

- ## Case 2.1:
  - – If n has no parent (n is the root): end
    - ▪ If n is the root and there is 1 black, the deletion occurs as usual without losing the tree balance.
    - ▪ This step also stops the correction propagation.
    - ▪ Note: do not delete n if it is an correction propagation node, not the actual node being deleted.

pRoot

n

c

α   β

Case 2.1 →

pRoot   c

α   β

  - – If n has a father, go to step 2.2

Pseudo Code

```c
void doStep2_1(struct node* n)
{
    if (n->parent != NULL)
        doStep2_2(n);
}
```

Case 2 (black child) in detail

- Step 2.2 (n is black, s is red):
  - Reversing father and sibling color.
  - Left rotate at p.
  - Go to step 2.3
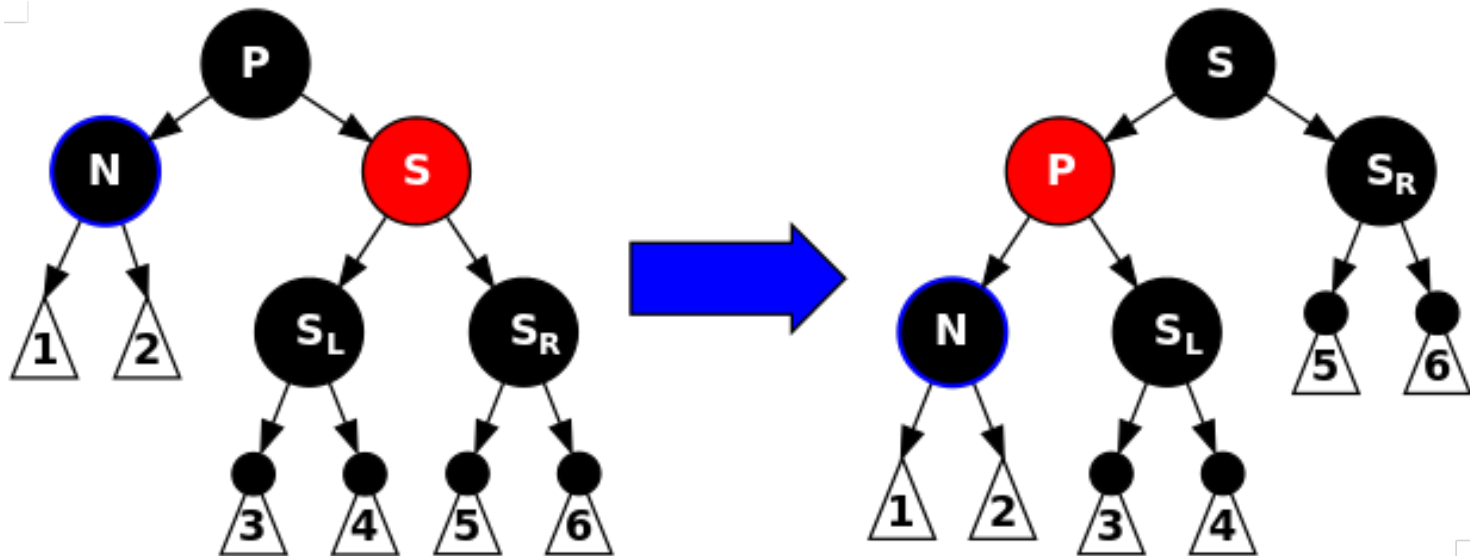
Pseudo Code

```
void doStep2_2(node* n)
{
    node* s = sibling(n);

    if (s->color == RED) {
        n->parent->color = RED;
        s->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    doStep2_3(n);
}
```

Case 2 (black child) in detail

- Bước 2.3 (n is black, s is black, children of s are black, p is black):
  - Change color s to red.
  - Go back to step 2.1 with n is current p (correction propagation)
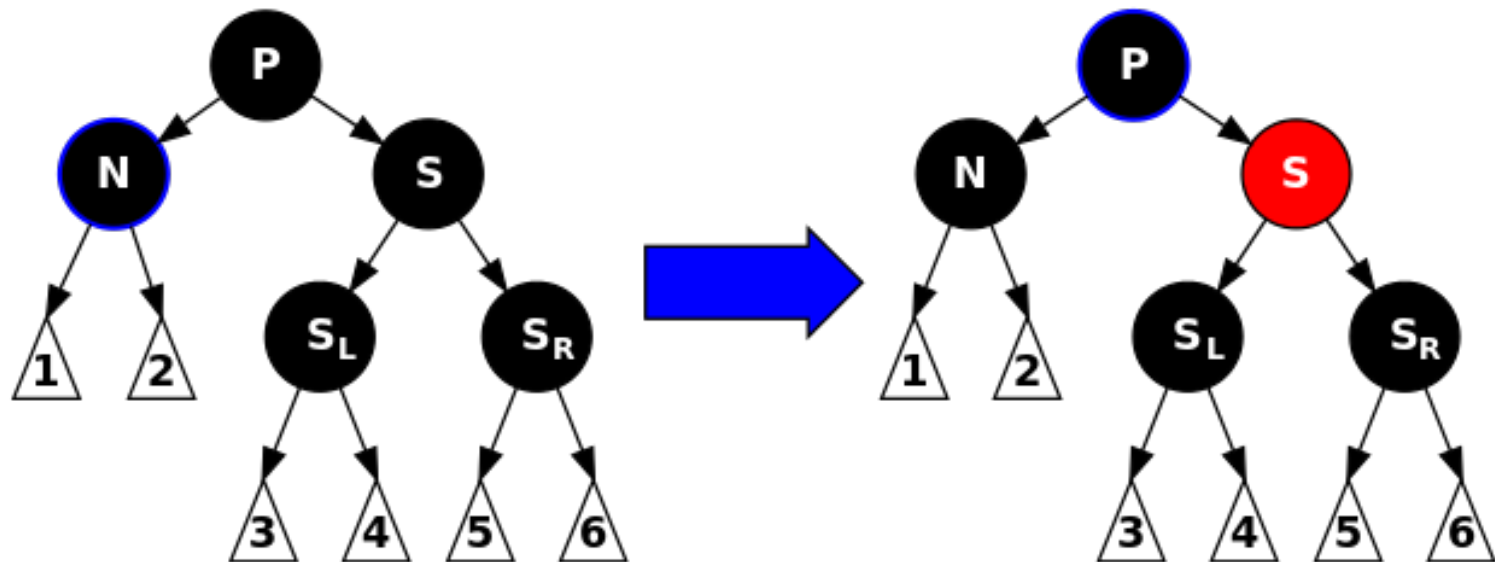
Pseudo Code
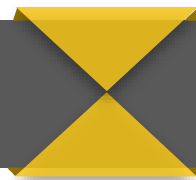
```
void doStep2_3(node* n)
{
    node* s = sibling(n);

    if ((n->parent->color == BLACK) &&
        (s->color == BLACK) &&
        (s->left->color == BLACK) &&
        (s->right->color == BLACK)) {
            s->color = RED;
            doStep2_1(n->parent);
    }
    else
        doStep2_4(n);
}
```

- Step 2.4 (n is black, s is black, children of s is black, p is red):
  - Reversing parent and sibling color.
  - End
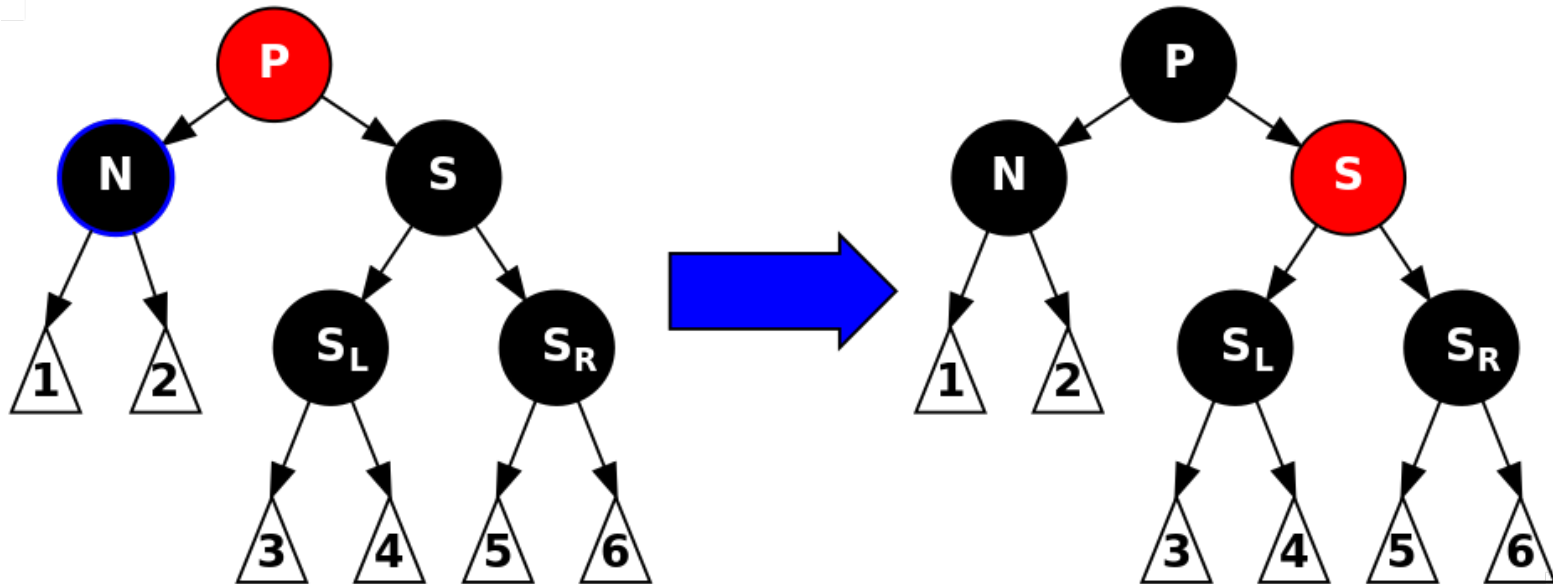
```
void doStep2_4(node* n)
{
    node* s = sibling(n);

    if ((n->parent->color == RED) &&
        (s->color == BLACK) &&
        (s->left->color == BLACK) &&
        (s->right->color == BLACK)) {
            s->color = RED;
            n->parent->color = BLACK;
    }
    else
        doStep2_5(n);
}
```

- Step 2.5 (n is black, s is black, left child of s is red, right child of s is black, n is left child):
  - Reversing s and its left child color.
  - Right rotate at s.
  - Go to step 2.6
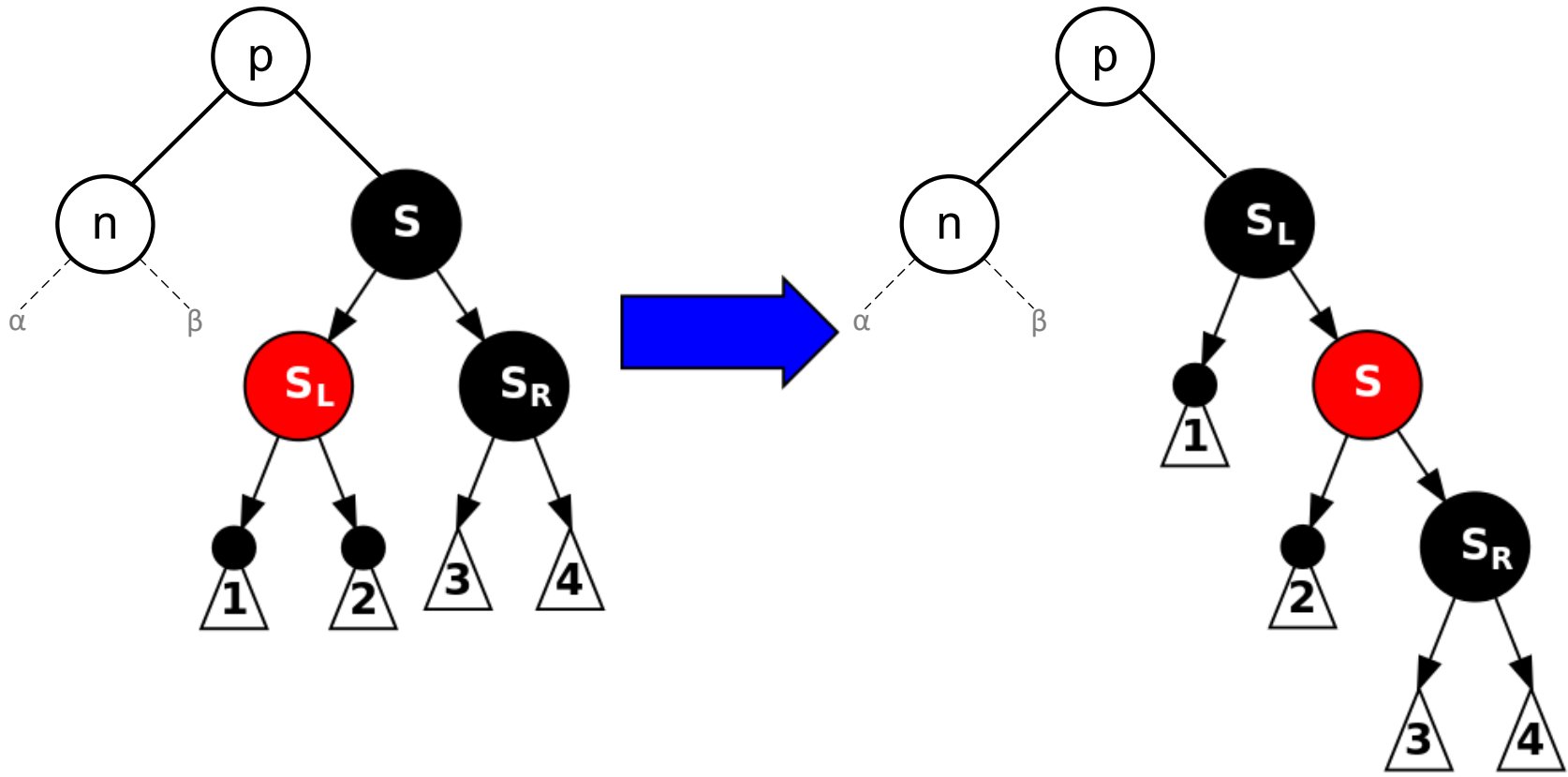
```
void doStep2_5(node* n)
{
    node* s = sibling(n);

    if (s->color == BLACK) {
        if ((n == n->parent->left) &&
            (s->right->color == BLACK) &&
            (s->left->color == RED)) {
            s->color = RED;
            s->left->color = BLACK;
                rotate_right(s);
        }
        else if ((n == n->parent->right) &&
            (s->left->color == BLACK) &&
            (s->right->color == RED)) {
            s->color = RED;
            s->right->color = BLACK;
                rotate_left(s);
        }
    }
    doStep2_6(n);
}
```

Case 2 (black child) in detail

- Step 2.6 (n is black, s is black, right child of s is red, n is left child):
  - Reversing s and p color
  - Color black for the right child of s
  - Left rotate at p
  - End.
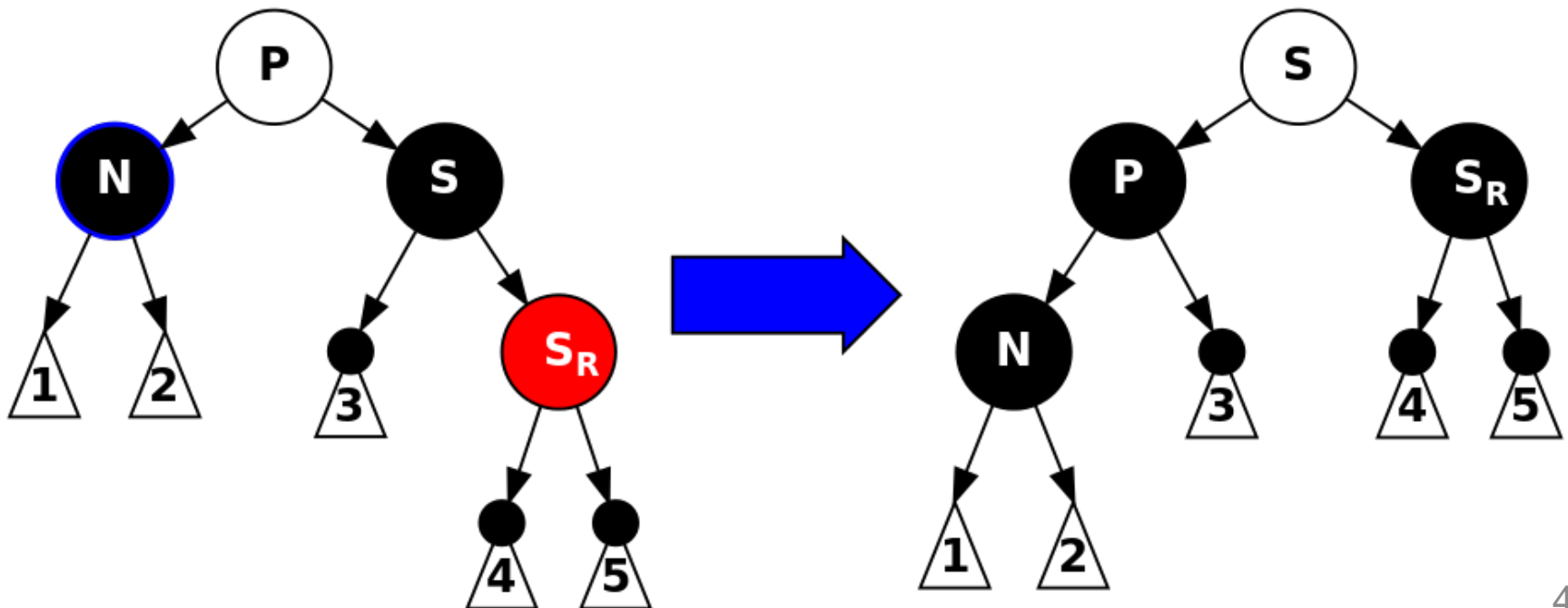
```
void doStep2_6(node* n)
{
    node* s = sibling(n);

    s->color = n->parent->color;
    n->parent->color = BLACK;

    if (n == n->parent->left) {
        s->right->color = BLACK;
        rotate_left(n->parent);
    }
    else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}
```
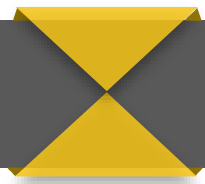
- Comments:
  - Advantages:
    - Search $O(log_2N)$
    - Insert $O(log_2N)$
    - Delete $O(log_2N)$
    - Minimum $O(log_2N)$
    - Maximum $O(log_2N)$
  - Disadvantages:
    - Must store the color attribute and pointer to the parent node
    - More complicated than AVL and AA trees

Outline

- Red-Black Tree
- AA Tree

AA (Arne Andersson) Tree

- AA trees are a variation of the red-black tree
  - Unlike red-black trees, red nodes on an AA tree can only be added as a right child.
  - A red-black tree need to consider seven different shapes to properly balance the tree. An AA tree on the other hand only needs to consider two shapes.
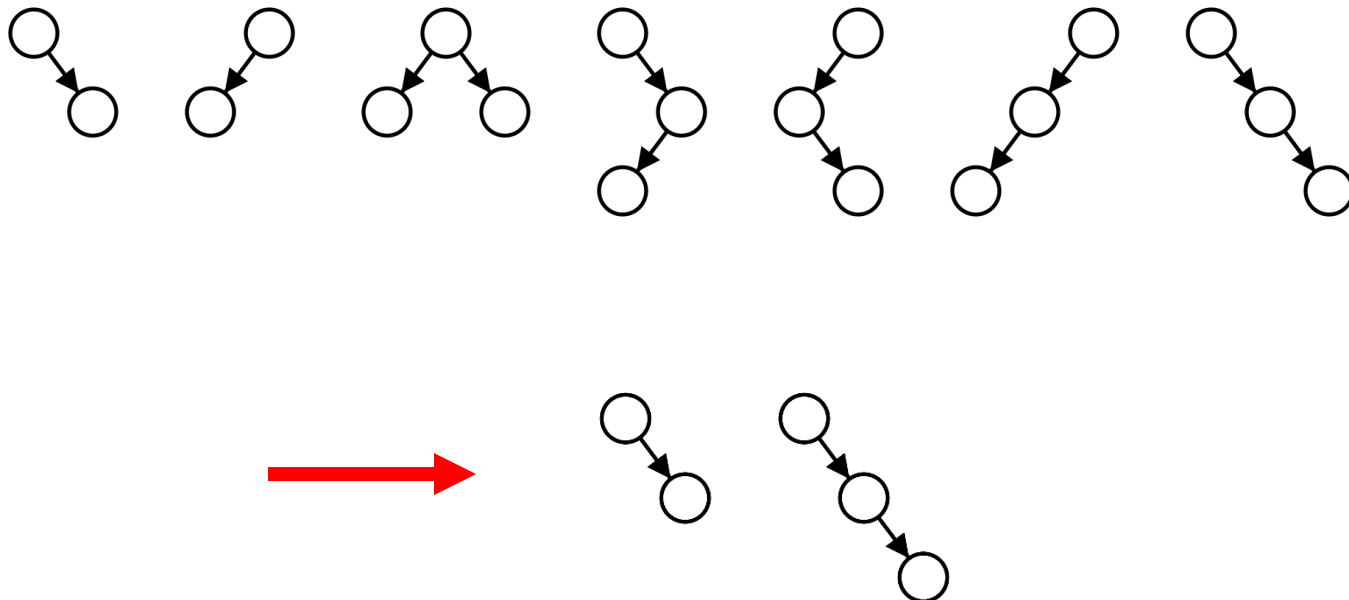
Example
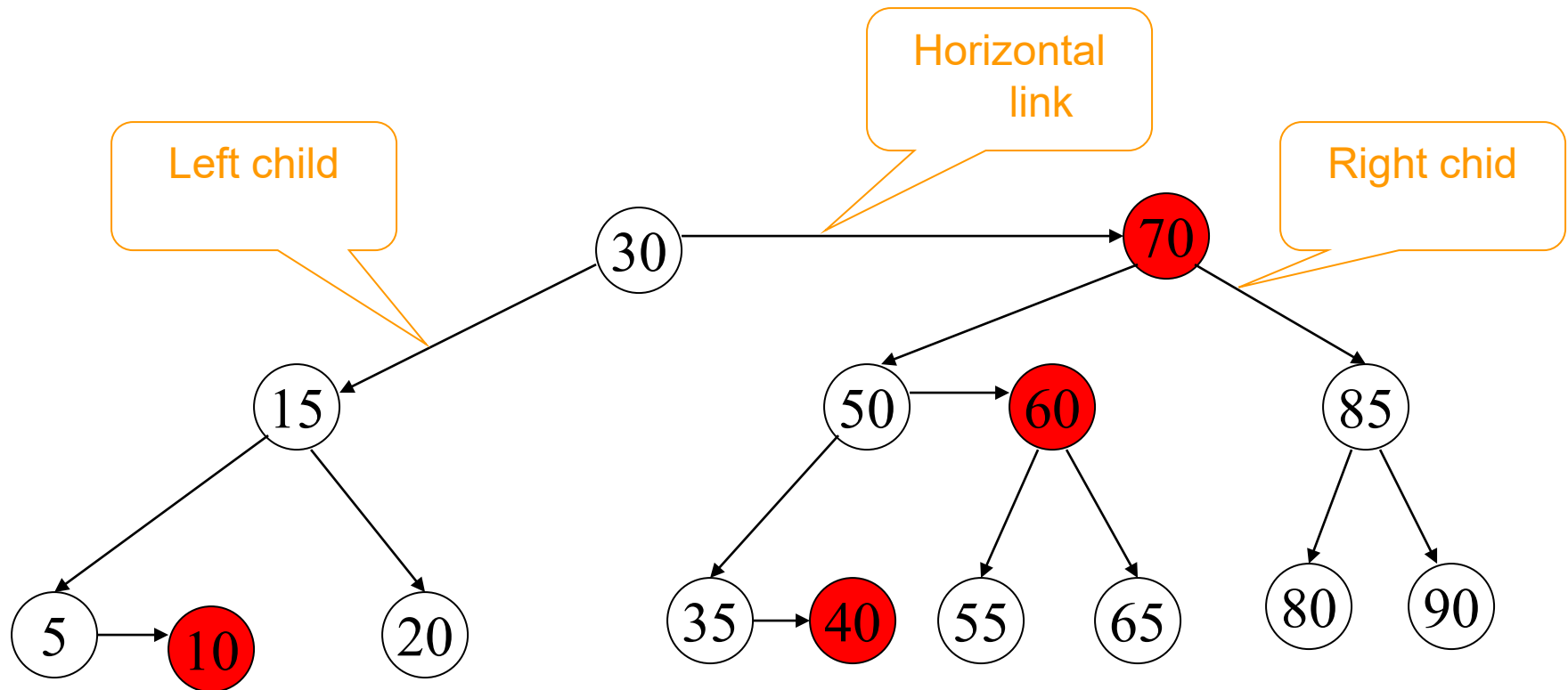
Level of a node

- Level of a node: is the number of left links from that node to NULL
  - The level of the NULL node is 0
  - The level of the leaf node is 1

Level 2

A → B

Level 1

C   D   E
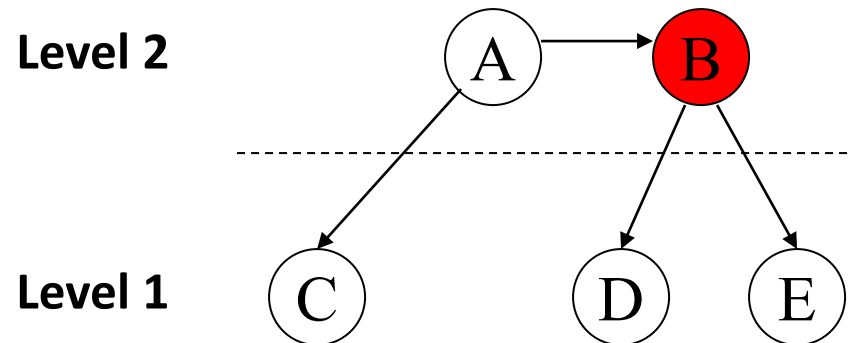
Horizontal link

- Horizontal link: an link between a node and its children at the same level
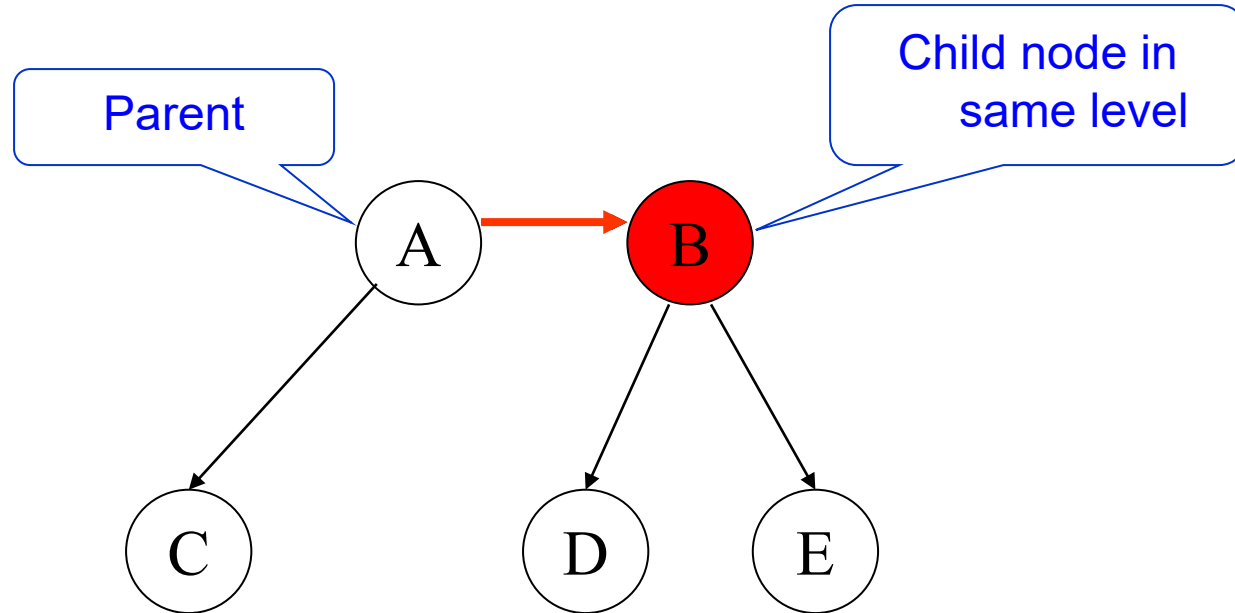
AA Tree

- AA tree is a binary search tree that conforms to the following rules:
  - Horizontal link is always oriented to the right
  - There are no 2 consecutive horizontal links
  - Every node with level greater than 1 will have 2 children
  - If a node has no right horizontal link then its children are at the same level

Example

Node level >
1 →
2 children

Horizontal link is oriented to the right

No 2 consecutive horizontal links

To child nodes are at the same level when the parent node has no horizontal link
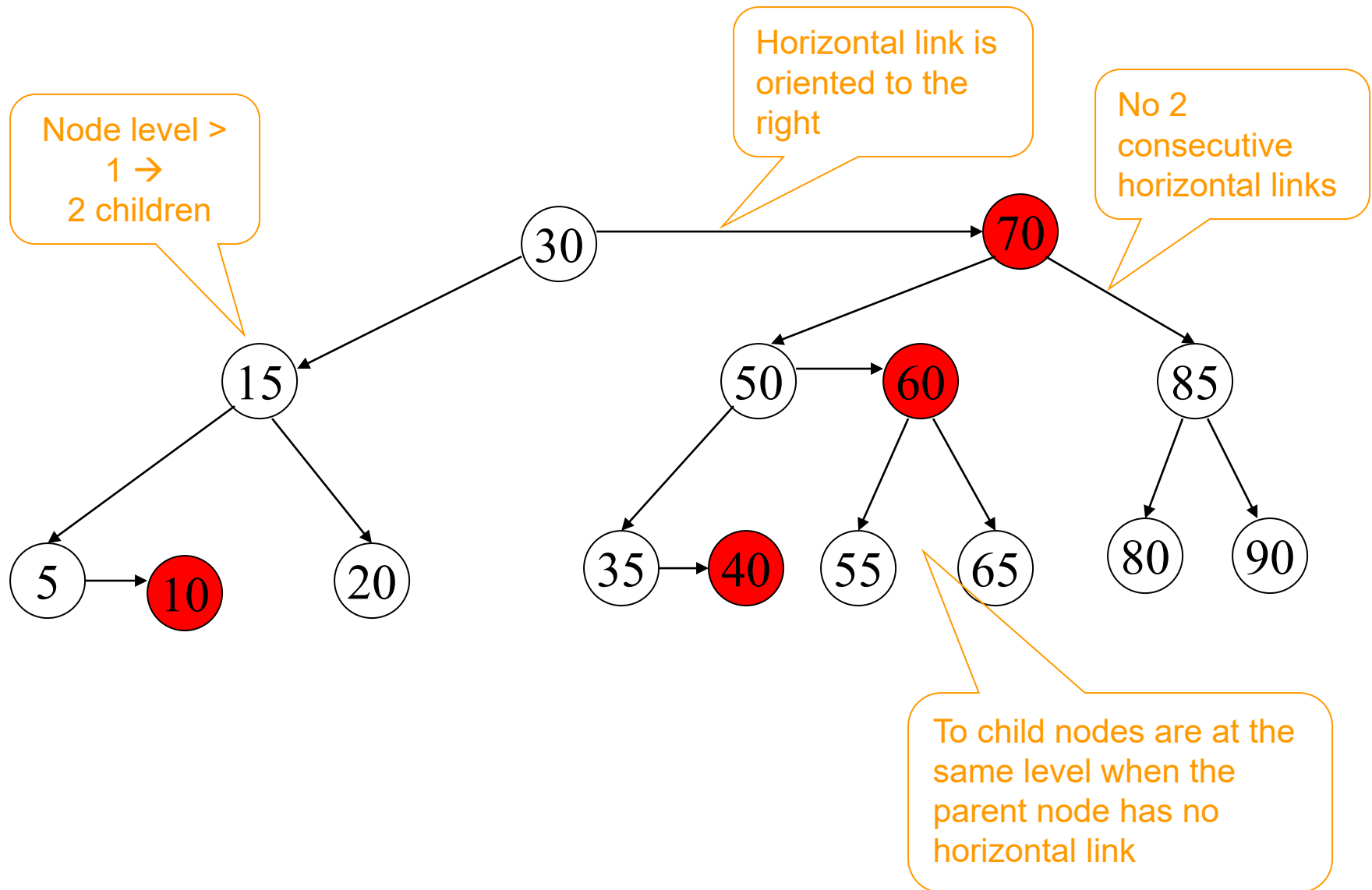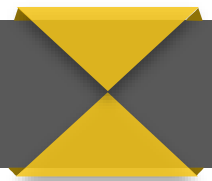
```
typedef int DataType;

typedef struct NodeTag {
    DataType          key;           // Data
    struct NodeTag    *pLeft;
    struct NodeTag    *pRight;
    int               level;      // Left of node
} AANode;


typedef struct AANode* AATREE;
```

Some properties of AA Tree

- Some properties:
  - The level of every leaf node is one.
  - The level of every left child is exactly one less than that of its 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.

Traverse in AA Tree

- Traverse in Tree: similar with BST

Find "55"
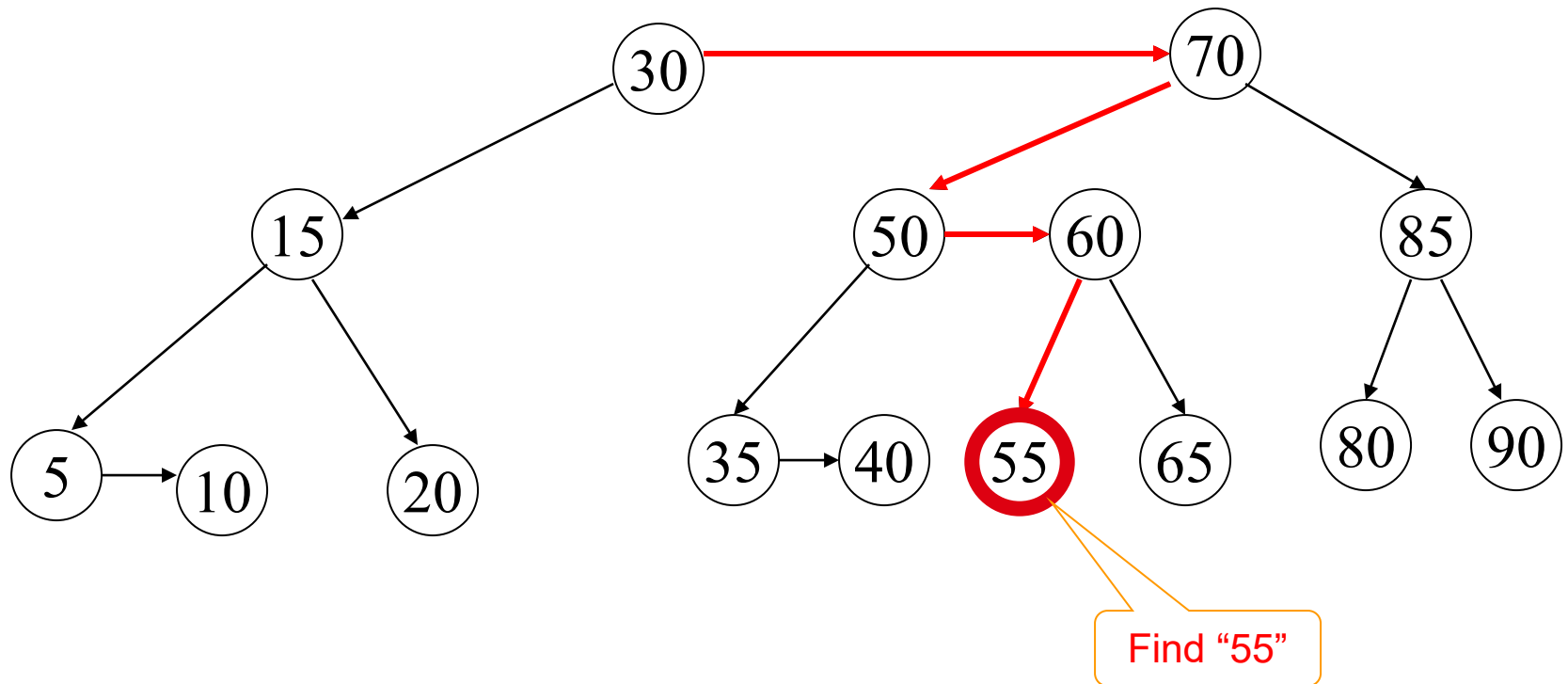
Balance

- Insertions and deletions may transiently cause an AA tree to become unbalanced
  - Only two distinct operations are needed for restoring balance: "skew" and "split".
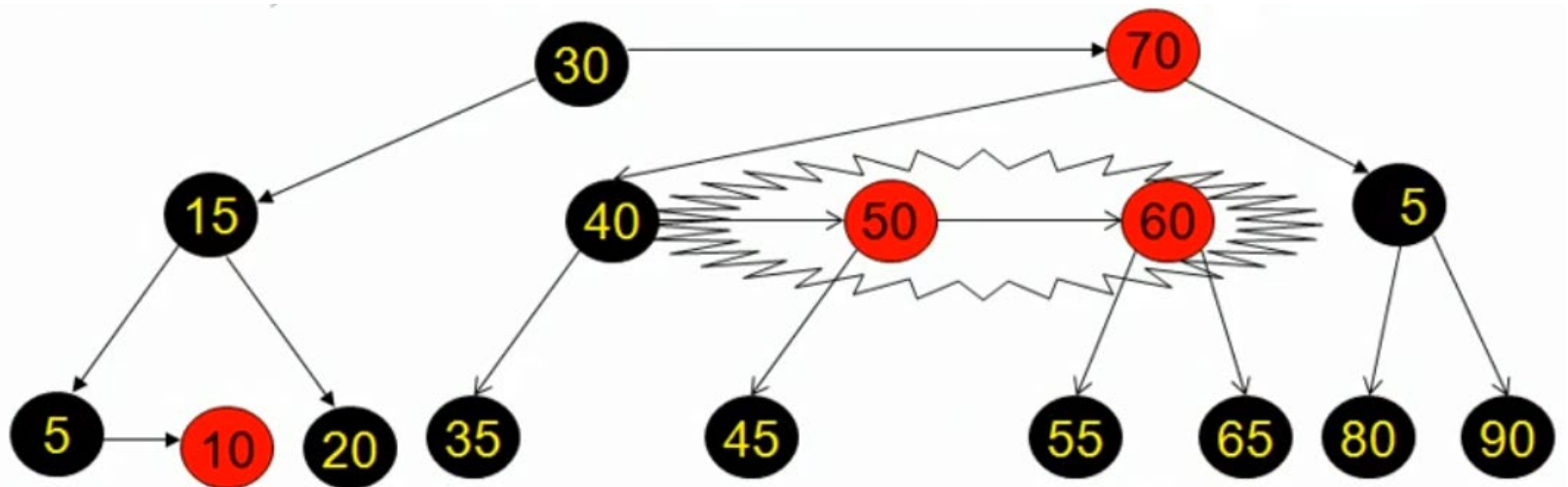
Skew

- Skew is a right rotation to replace a subtree containing a left horizontal link with one containing a right horizontal link instead
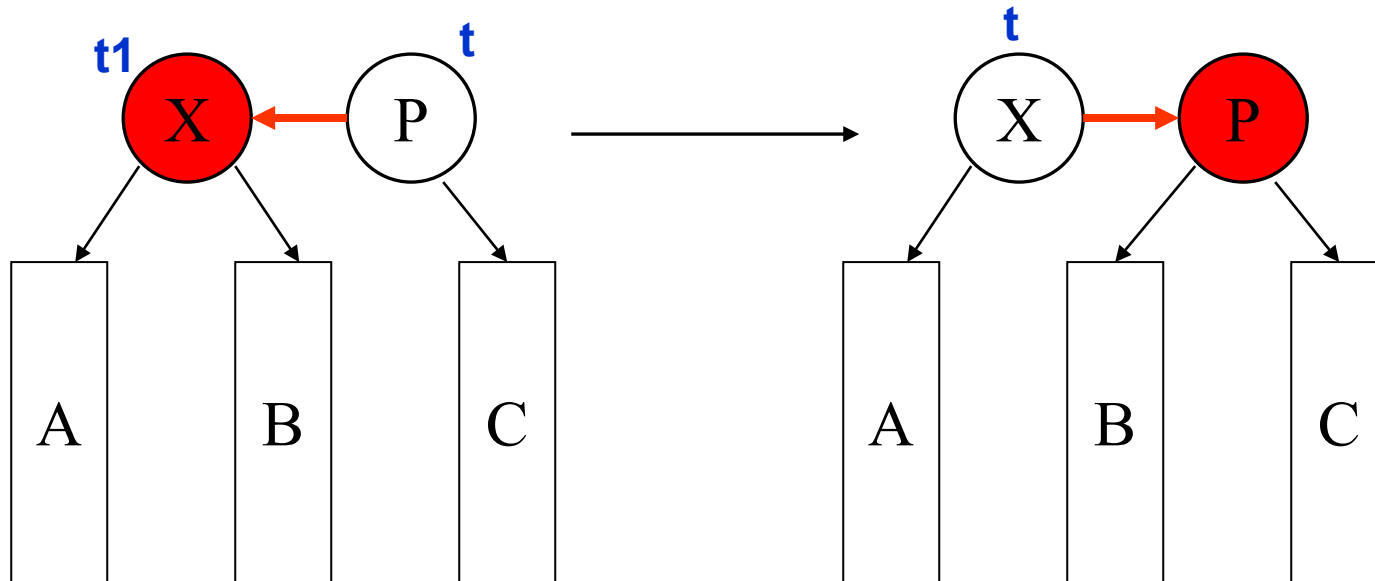
```
AATREE right_rotate(AATREE &t)
{
    AATREE t1;
    t1 = t->pLeft;
    t->pLeft = t1->pRight;
    t1->pRight = t;

    return t1;
}


AATREE Skew(AATREE &t)
{
    if (t->pLeft != NULL)
        if (t->pLeft->level == t->level)
            t = right_rotate(t);
    return t;
}
```

- Skew can cause multiple right-row horizontal links → using Split to adjust
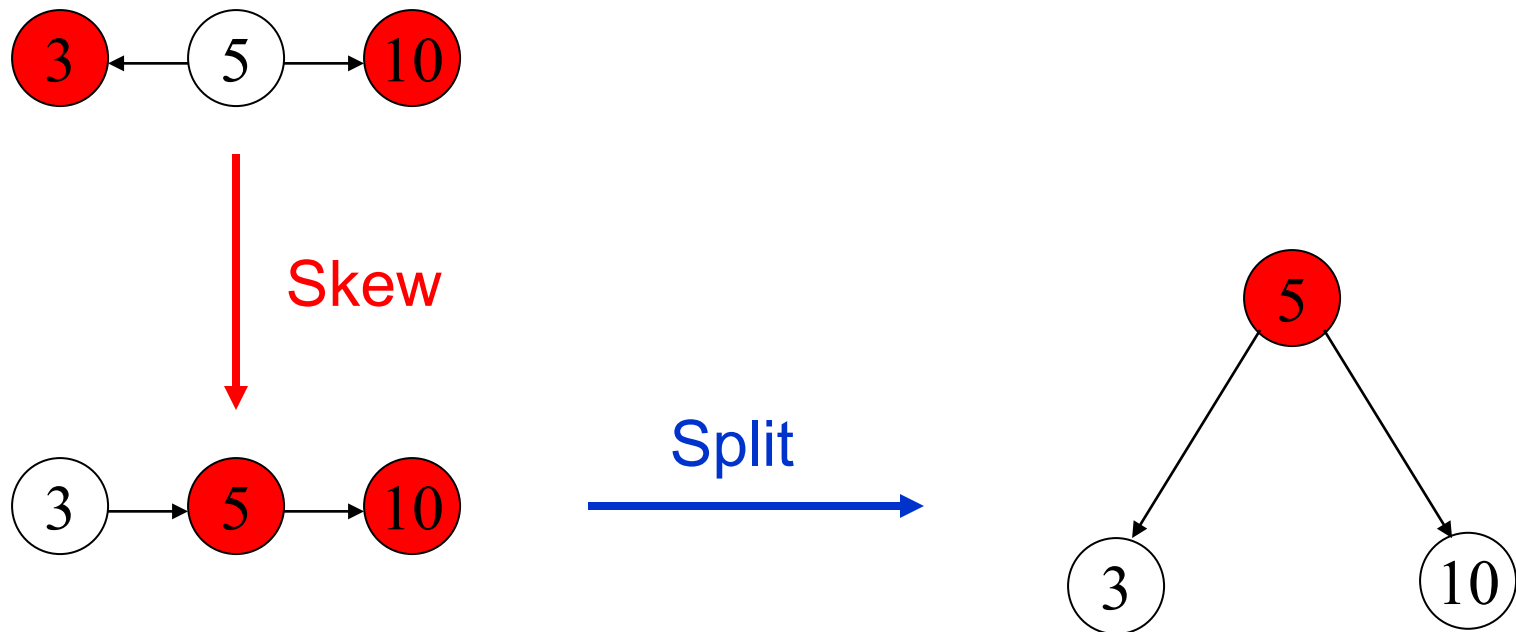
- Split is a left rotation and level increase to replace a subtree containing two or more consecutive right horizontal links with one containing two fewer consecutive right horizontal links
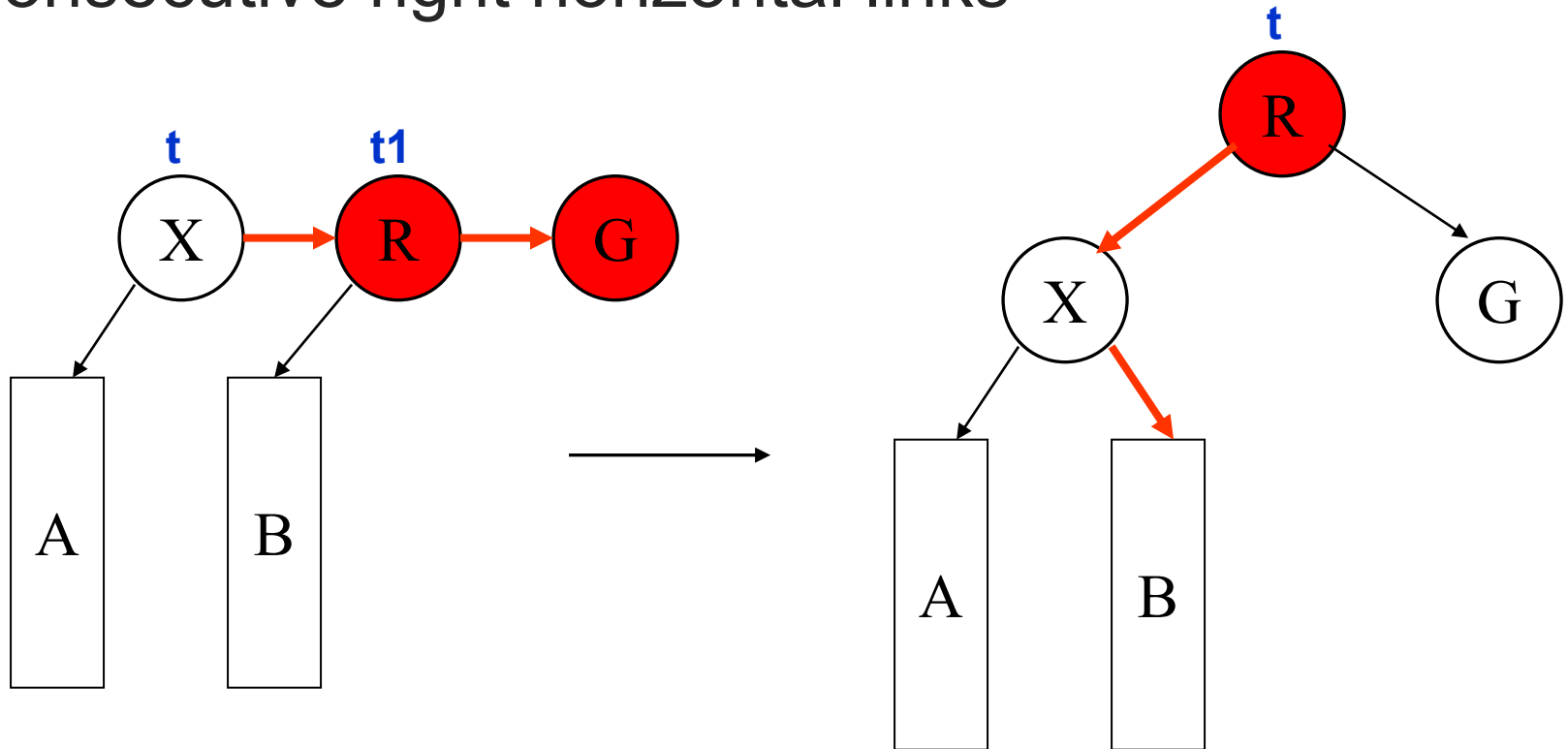
```
AATREE left_rotate(AATREE &t)
{
    AATREE t1;
    t1 = t->pRight;
    t->pRight = t1->pLeft;
    t1->pLeft = t;
    t1->level++;
    return t1;
}
AATREE Split(AATREE &t)
{
    if (t->pRight !=NULL)
       if (t->pRight->pRight != NULL)
         if (t->pRight->pRight->level == t->level)
             t = left_rotate(t);
    return t;
}
```

Insert a new node

- Insert a new node
  - Execute like binary tree search.
    - Insert a node is always done at the node with level = 1.
  - When node added on the left side
    - It will create a left horizontal link → balance with Skew
  - When node added on the right side
    - It will create a right horizontal link
    - If it causes two right horizontal links → balance with Split
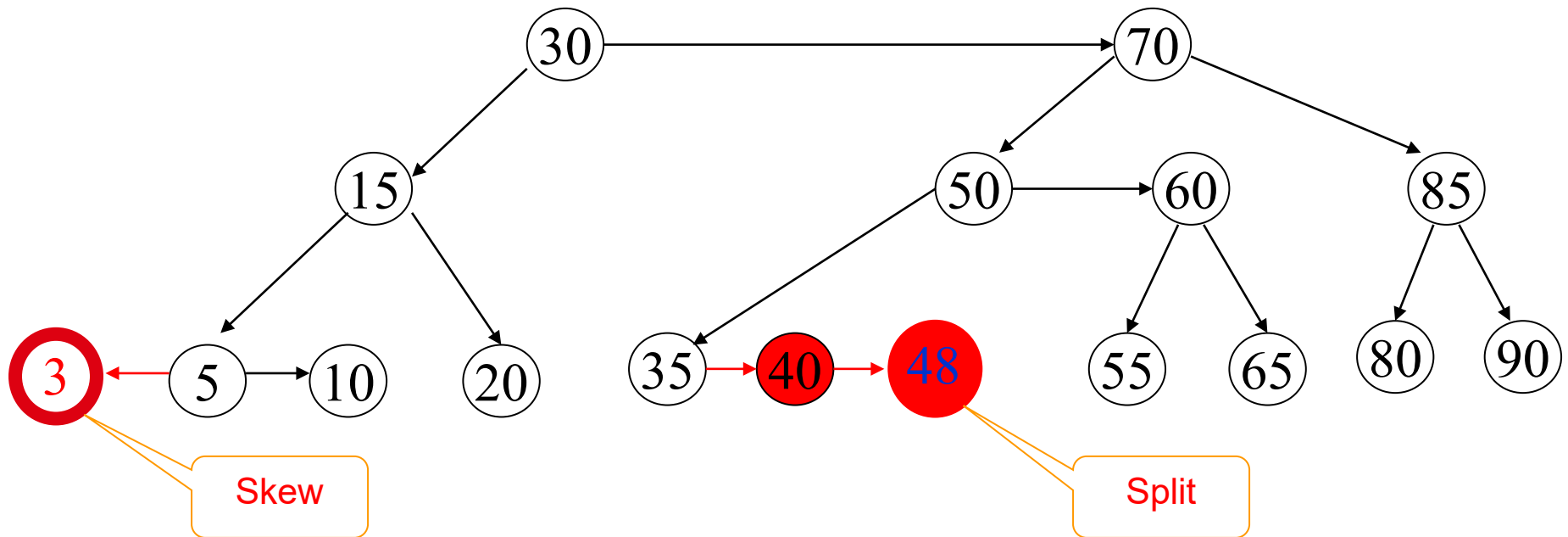
Example

- Insert node 45:
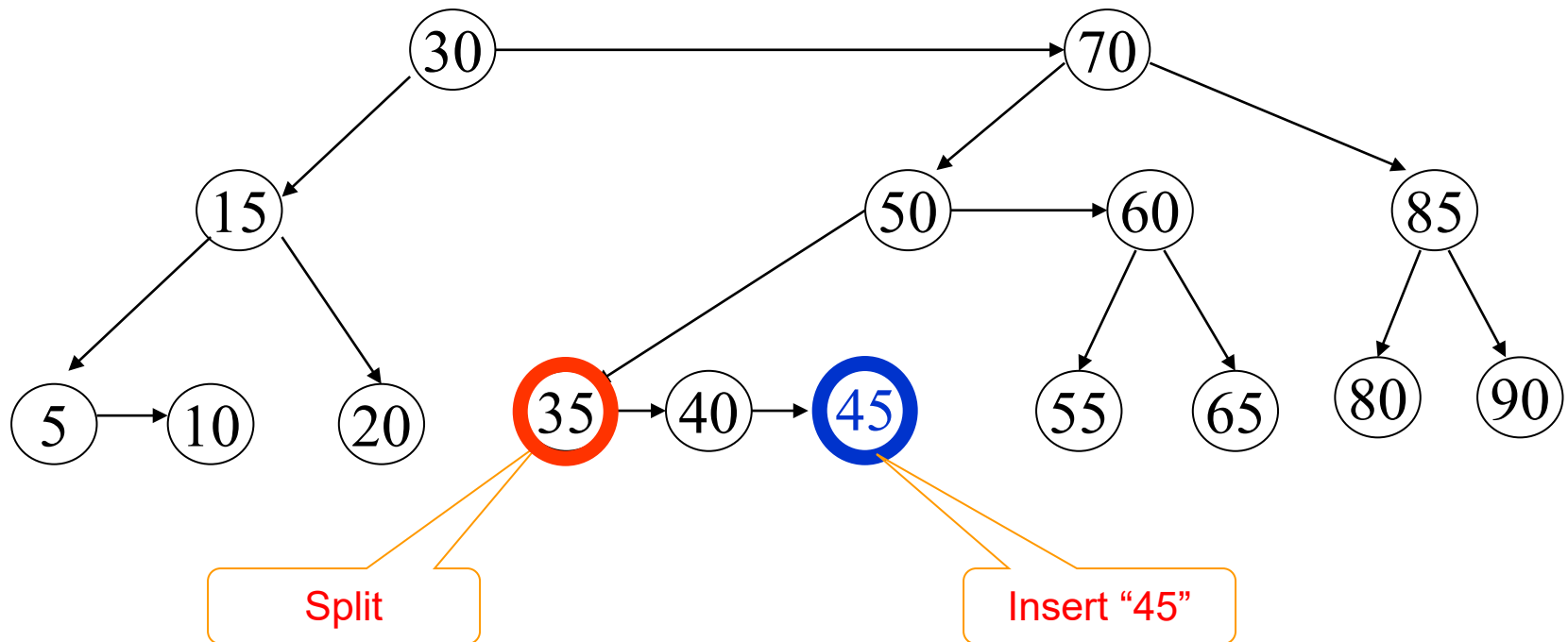
- After split at 35:

Skew

- After skew at 50:

- After split at 40:

Skew

30 → 50 ⇄ 70

15   40   60   85

5 → 10   20   35   45   55   65   80   90
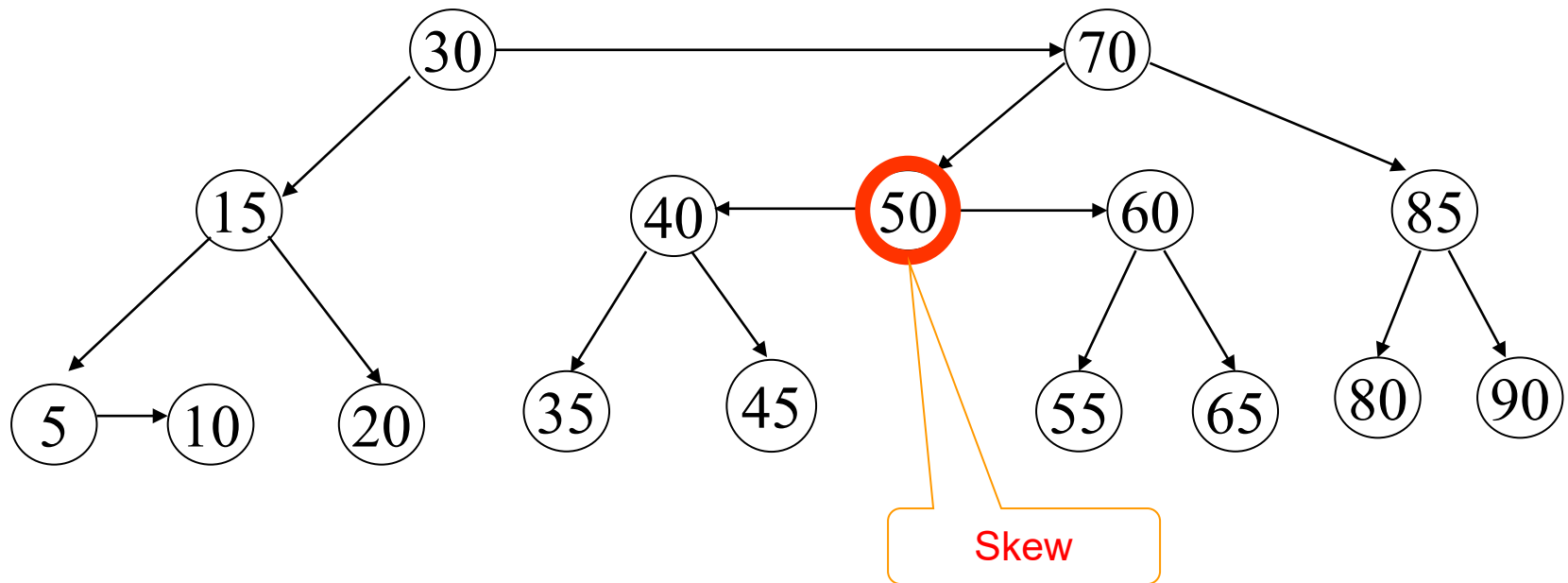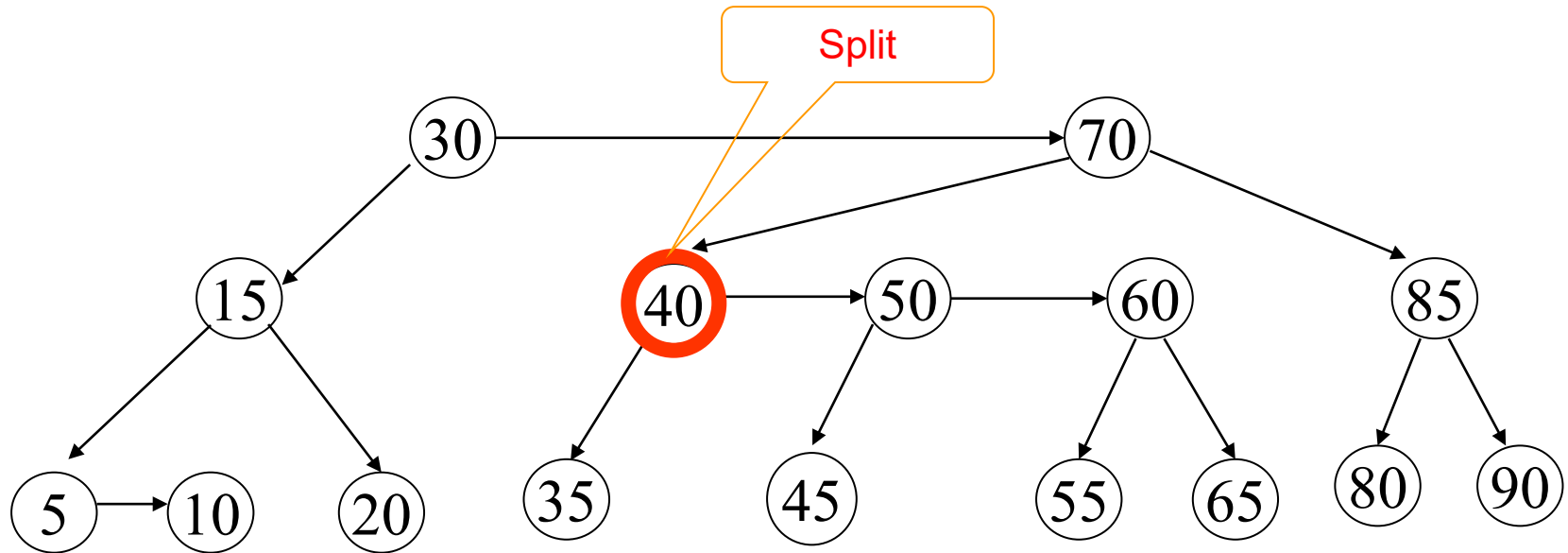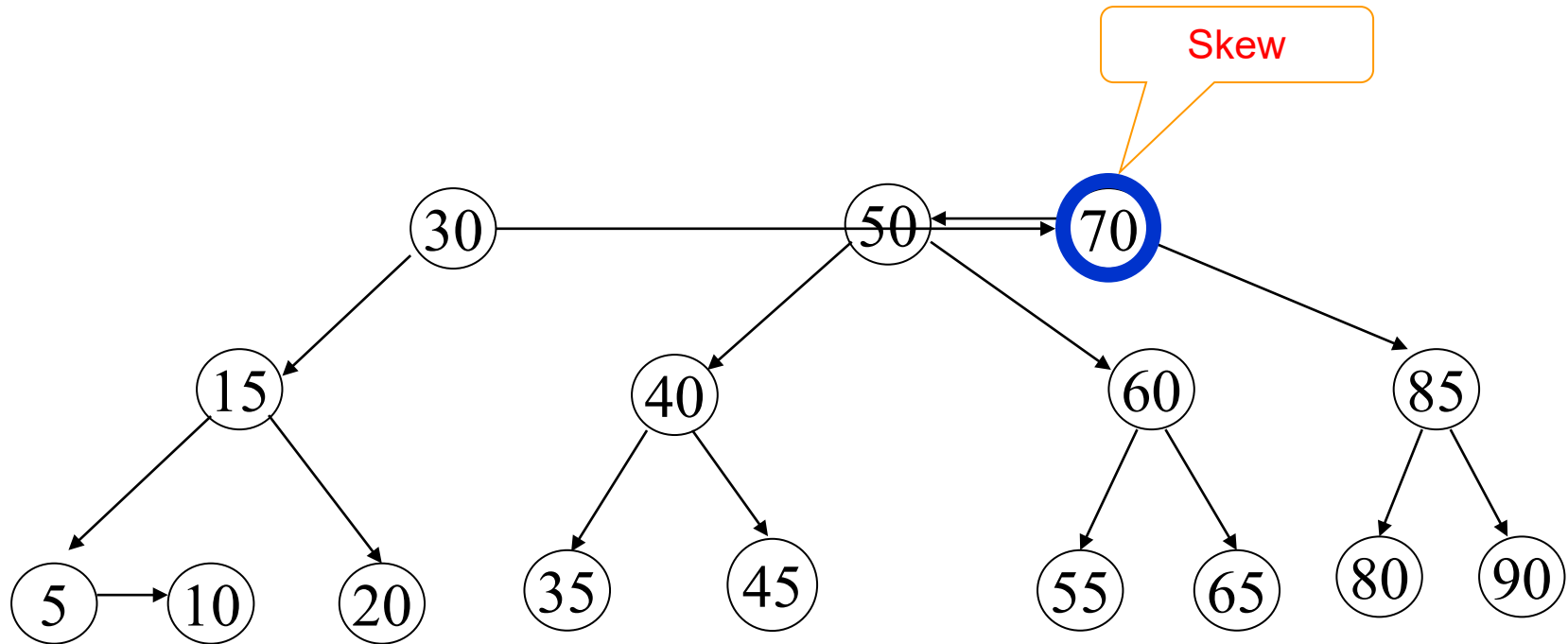
Example

- After skew at 70:
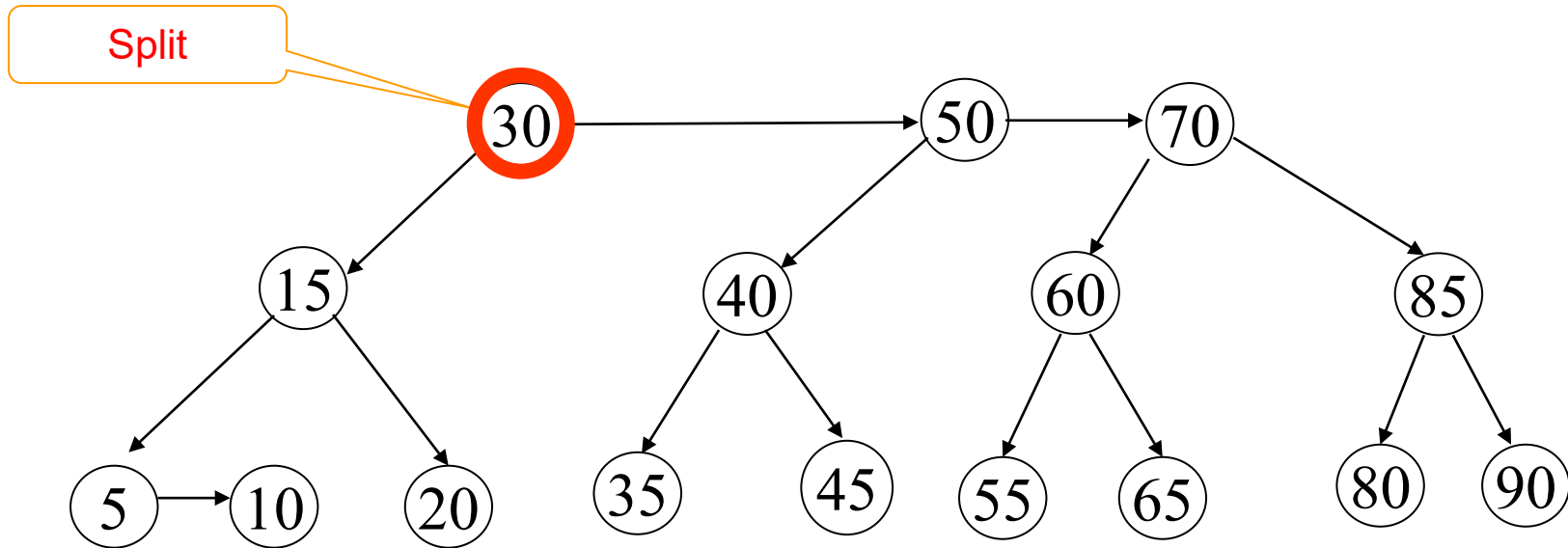
- After split at 30:
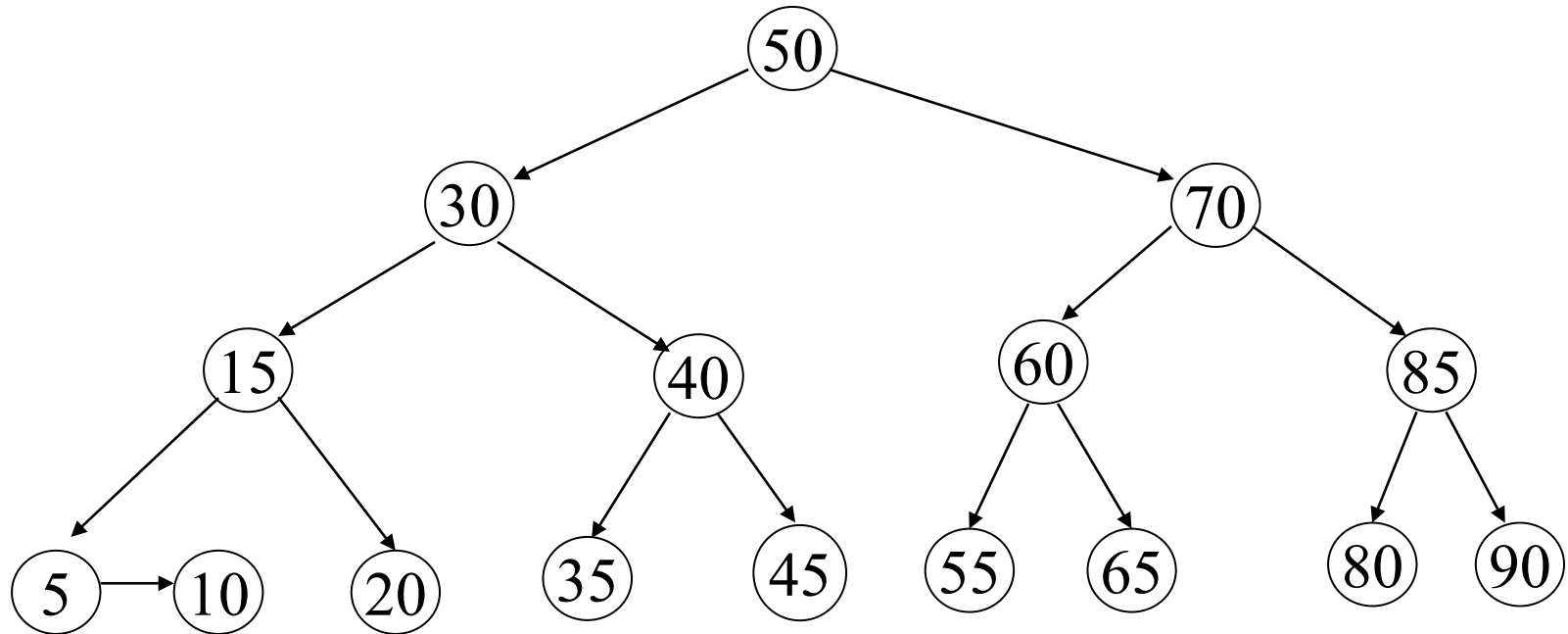
→ STOP !

```
AATREE AA_Insert_Node(DataType x, AATREE t)
{
    if(t == NULL)        {
        t = new AANode;
        t->key = x;
        t->pLeft = t->pRight = NULL;
        t->level = 1;
    }
    else if(x < t->key)
                t->pLeft = AA_Insert_Node(x, t->pLeft);
        else if(x > t->key)
                t->pRight = AA_Insert_Node(x, t->pRight);
        else return t;   // duplicate

    /* Perform skew and then split. The conditionals that
    determine whether or not a rotation will occur or not are
    inside of the procedures, as given above.*/

    t = Skew(t);
    t = Split(t);

    return t;
}
```

- Insert node with 6:

4 → 10

2

8

12

1 3

5 → 7 9

11 13

(6) ??
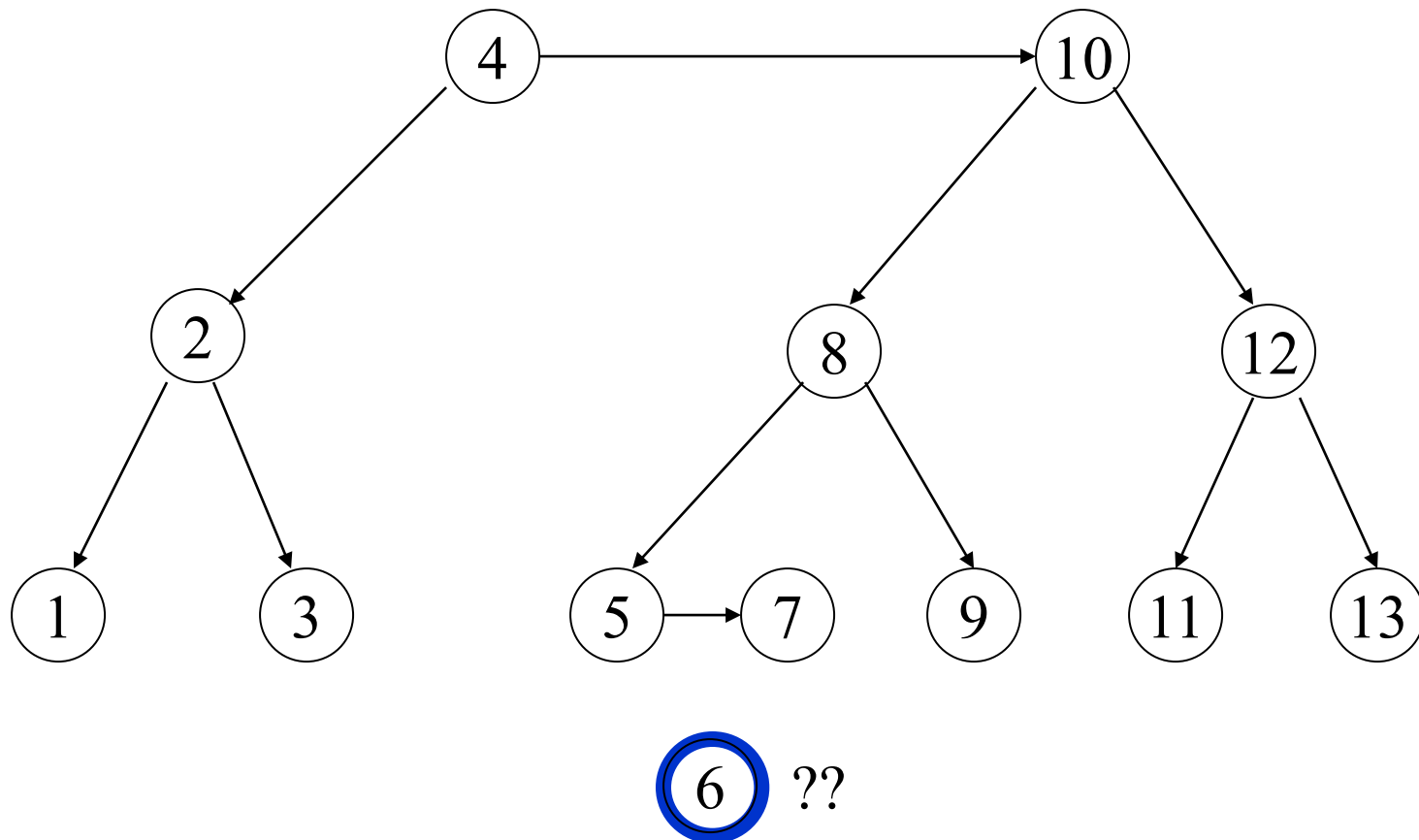
Delete a node

- Delete a node
  - Execute like binary tree search.
    - The actually deleted node is always in level 1.
  - After a removal (balance):
    - The first step is to lower the level of any nodes whose children are two levels below them (who are missing children).
    - Then, the entire level must be skewed and split.
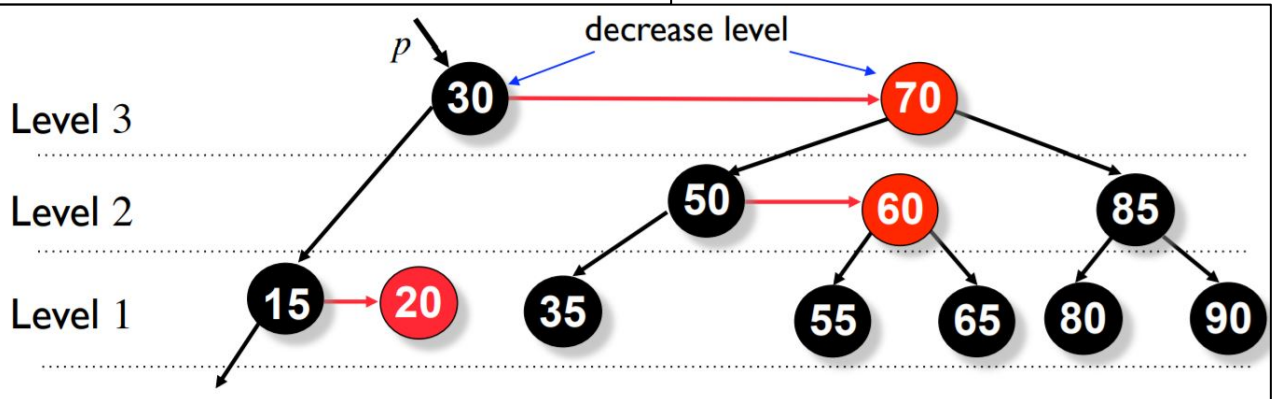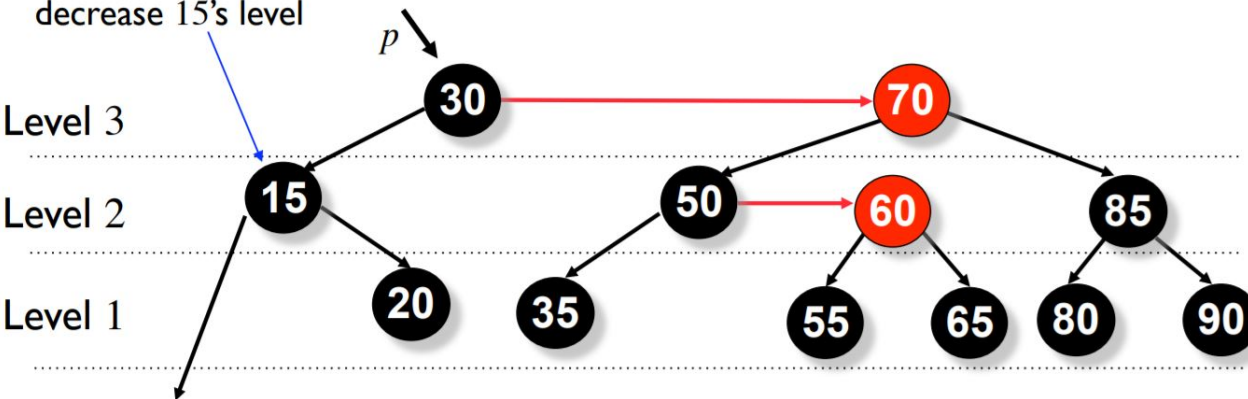
Decrease level

- In the process of deletion, a node can lose one of its children. As a result, we may need to decrease this node's level in the tree.
  - The ideal level of any node is one more than the minimum level of its two children.
  - If we discover that p's current level is higher than this ideal value, we set it to its proper value.
  - If p's right child is a red node (that is, p.right.level == p.level prior to the adjustment), then the level of p.right needs to be decreased as well.

Decrease level

- Delete node 5
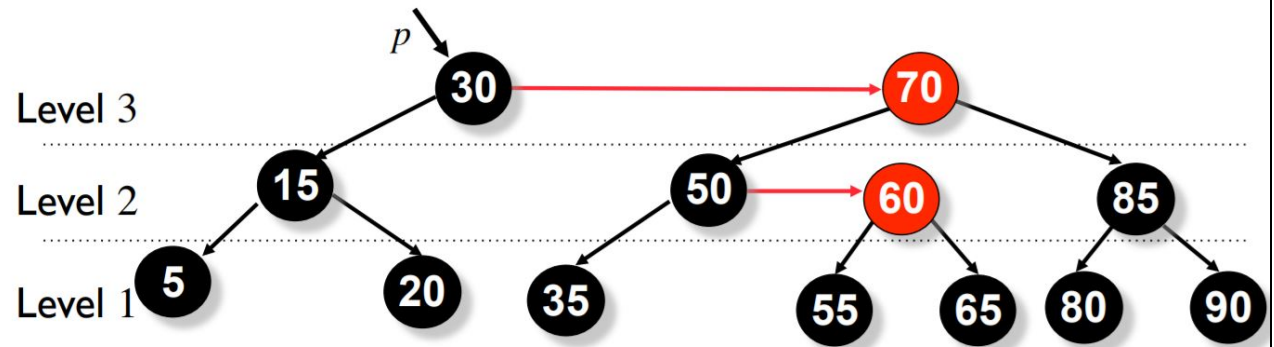
```
AANode AA_Update_Level(AANode p)
{
    int idealLevel = 1 + min(p.left.level, p.right.level);
    if (p.level > idealLevel) { // p's level is too high?
        p.level = idealLevel; // decrease its level
        if (p.right.level > idealLevel) // p's right child red?
                p.right.level = idealLevel; //fix its level as well
    }
    return p;
}
```
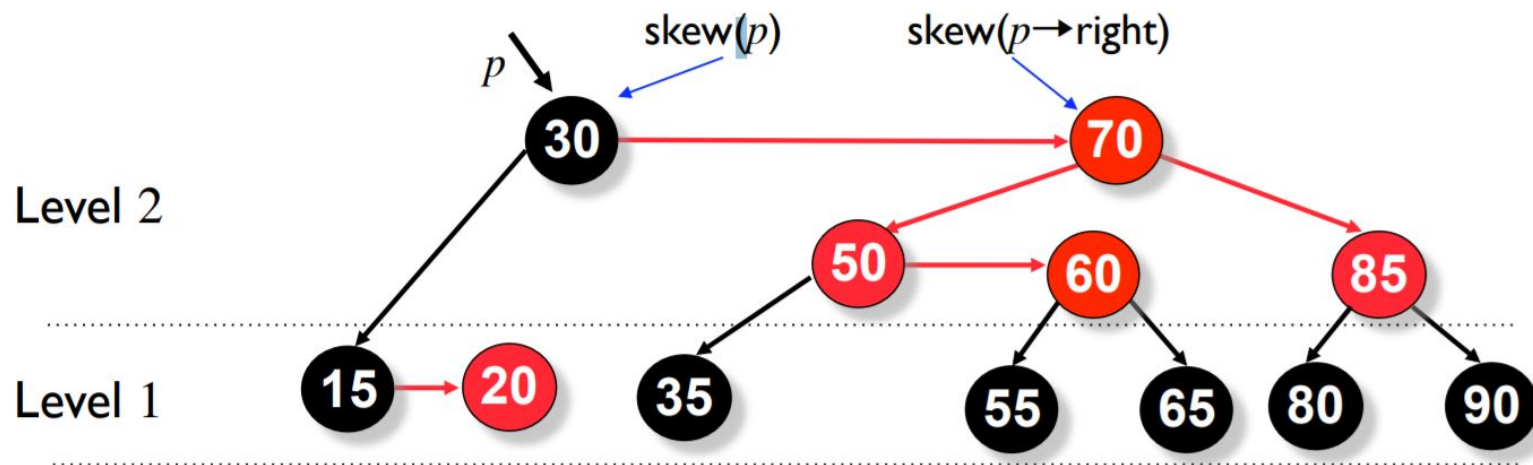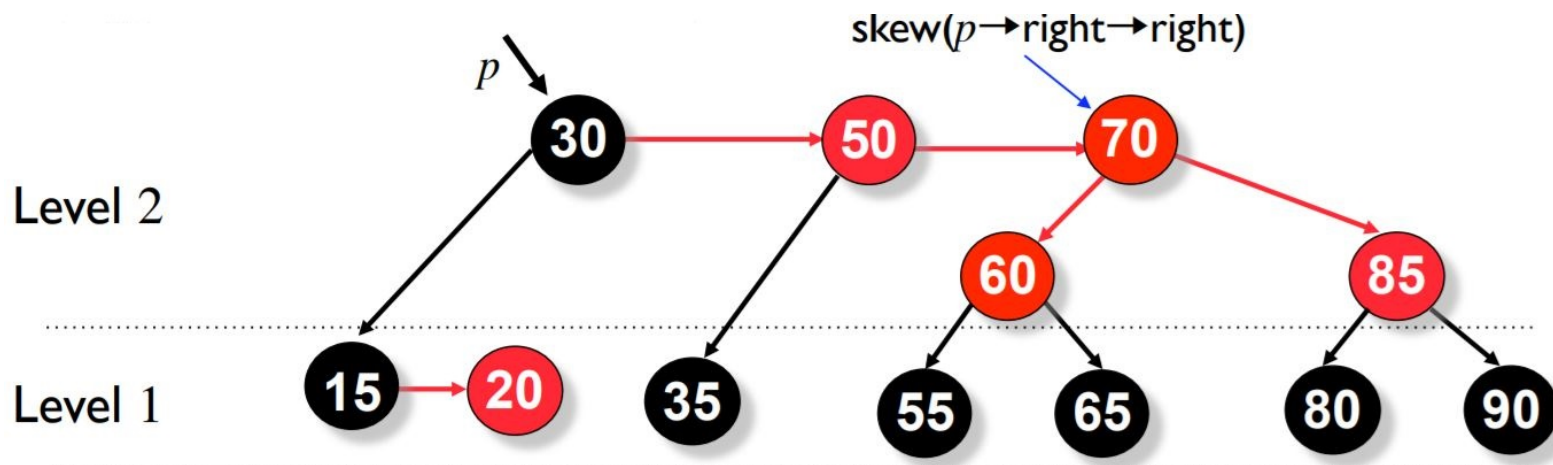
Skew and Split

- After update level of any node:
  - Perform skew operations for p, p.right, and p.right.right.
  - Next, perform two splits, one at p, and the other to its right-right grandchild, which becomes its right child after the first split.

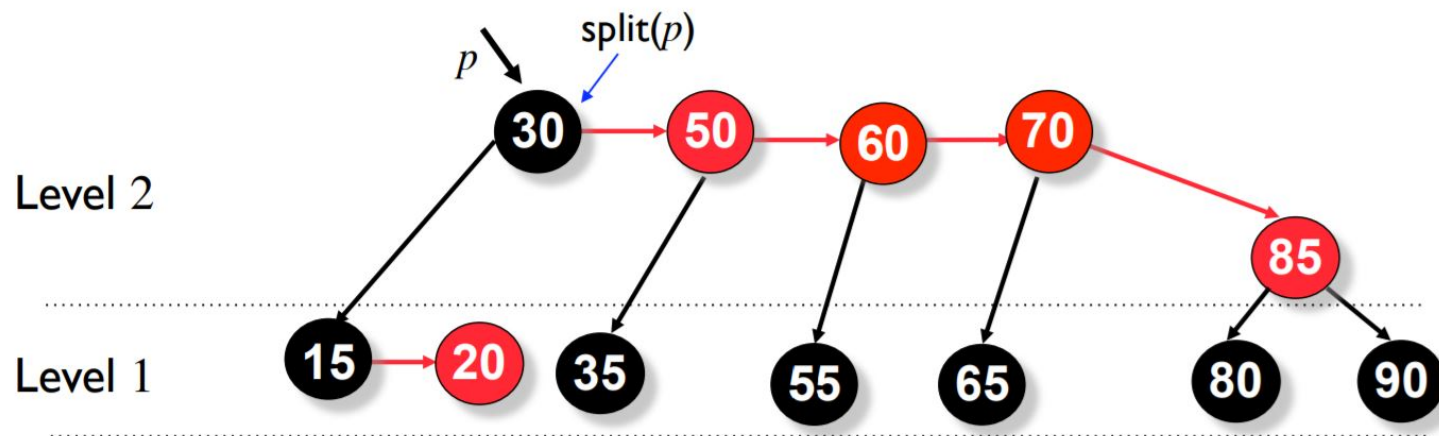# Skew and Split

- Do the skew operation on p and p->right

- Do the skew operation on p->right->right

Skew and Split
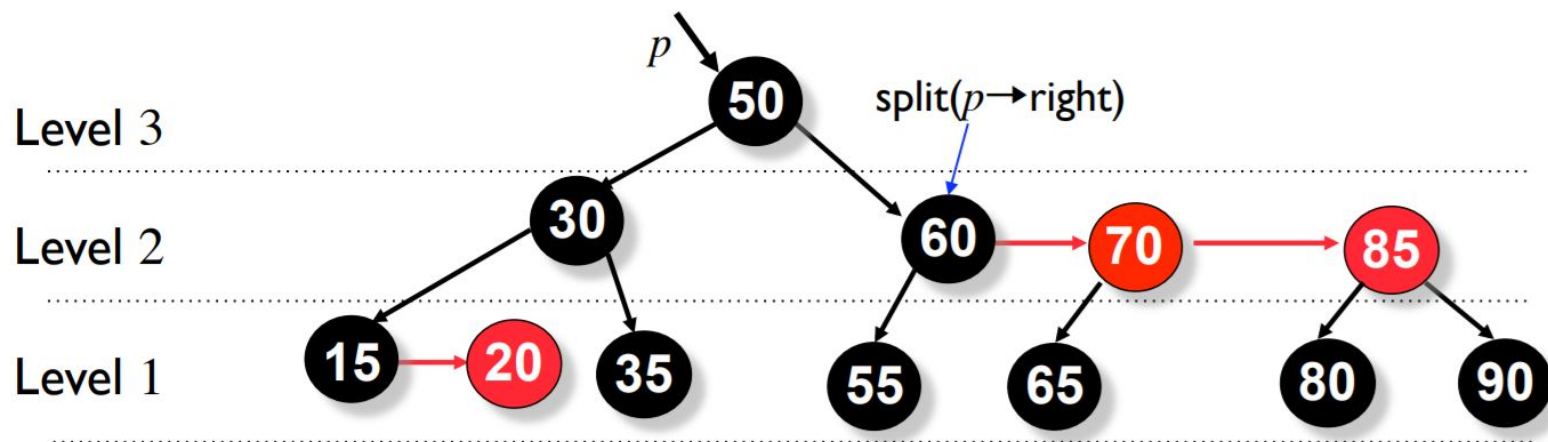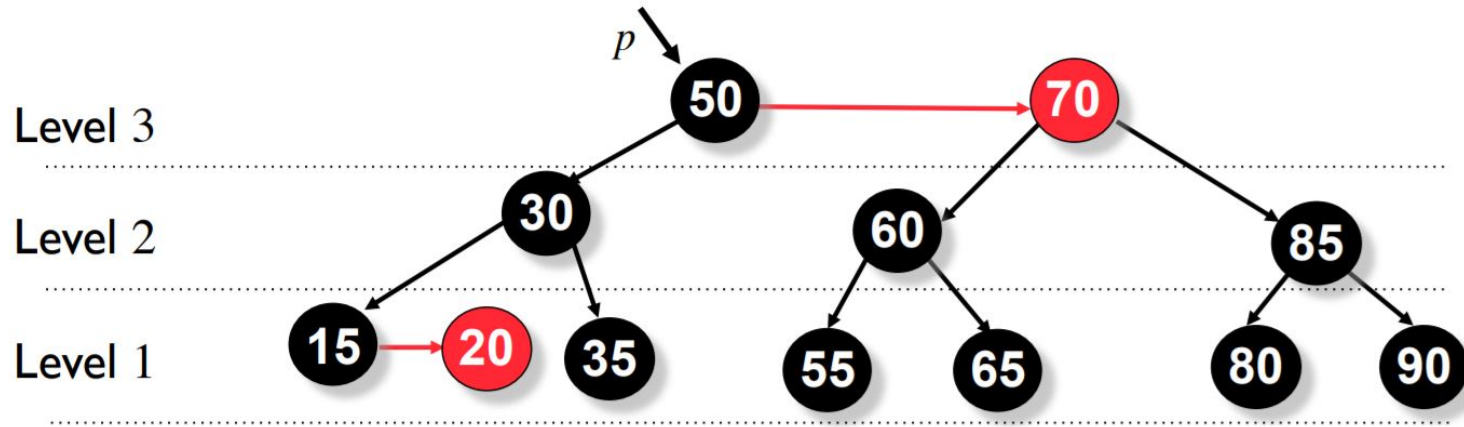
- Do split operation on p

- Do split operation on p->right
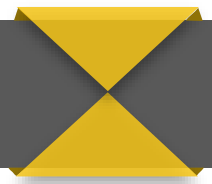
- After final split

```
AANode AA_Balance_Tree(AANode p)
{
   p = AA_Update_Level(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;
}
```
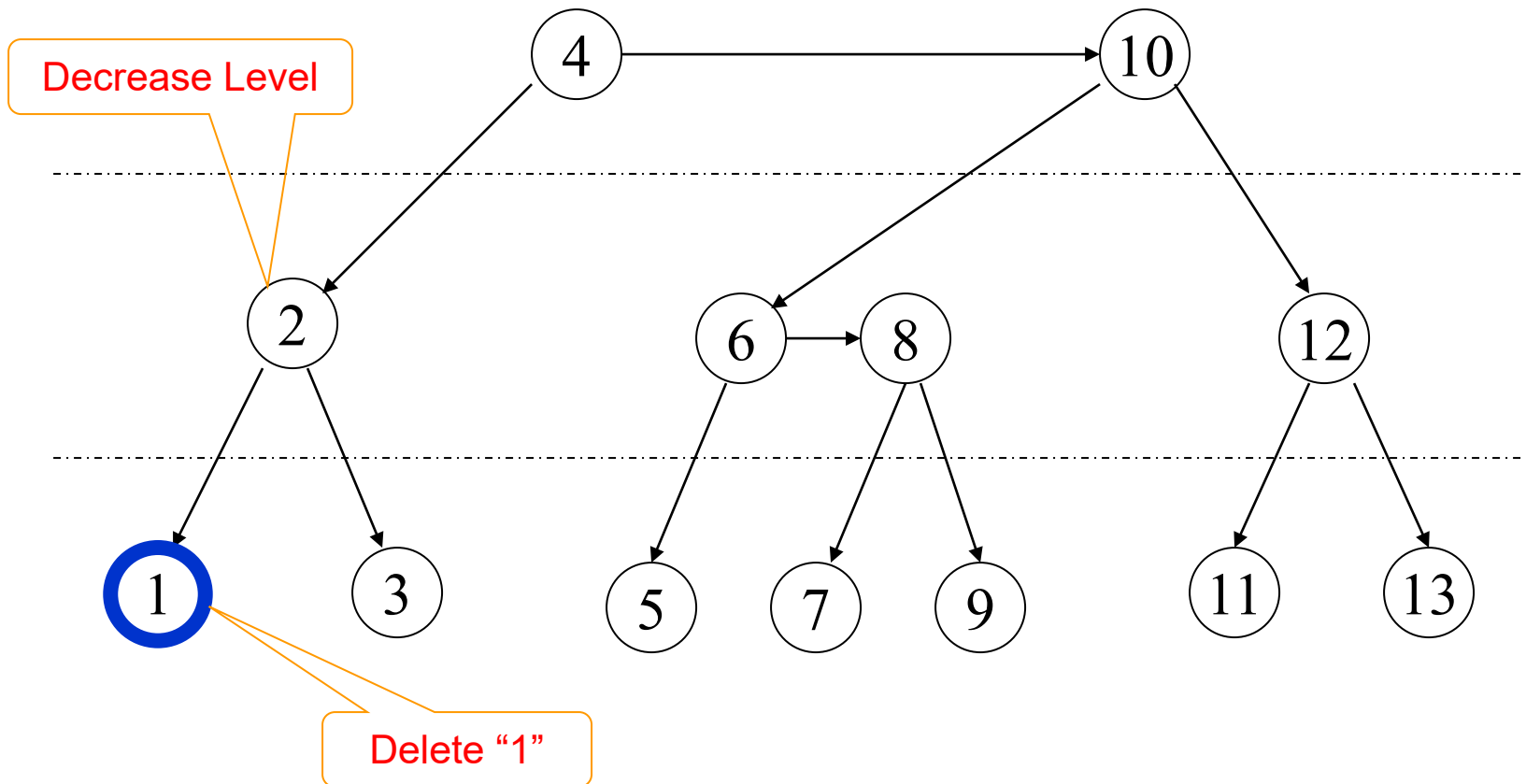
```
AANode delete(Key x, AANode p) {
    if (p == nil) // fell out of tree?
        throw KeyNotFoundException; // ...error - no such key
    else {
        if (x < p.key) // look in left subtree
                p.left = delete(x, p.left);
        else if (x > p.key) // look in right subtree
                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?
                        // get replacement from right
                        AANode r = inorderSuccessor(p);
                        // copy replacement contents here
                        p.copyContentsFrom(r);
                        // delete replacement
                        p.right = delete(r.key, p.right);
                }
                else { // no right child?
                        // get replacement from left
                        AANode r = inorderPredecessor(p);
                        p.copyContentsFrom(r); // copy replacement contents
                        p.left = delete(r.key, p.left); // delete replacement
                }
        }
        return AA_Balance_Tree(p); // fix structure after deletion
    }
}
```
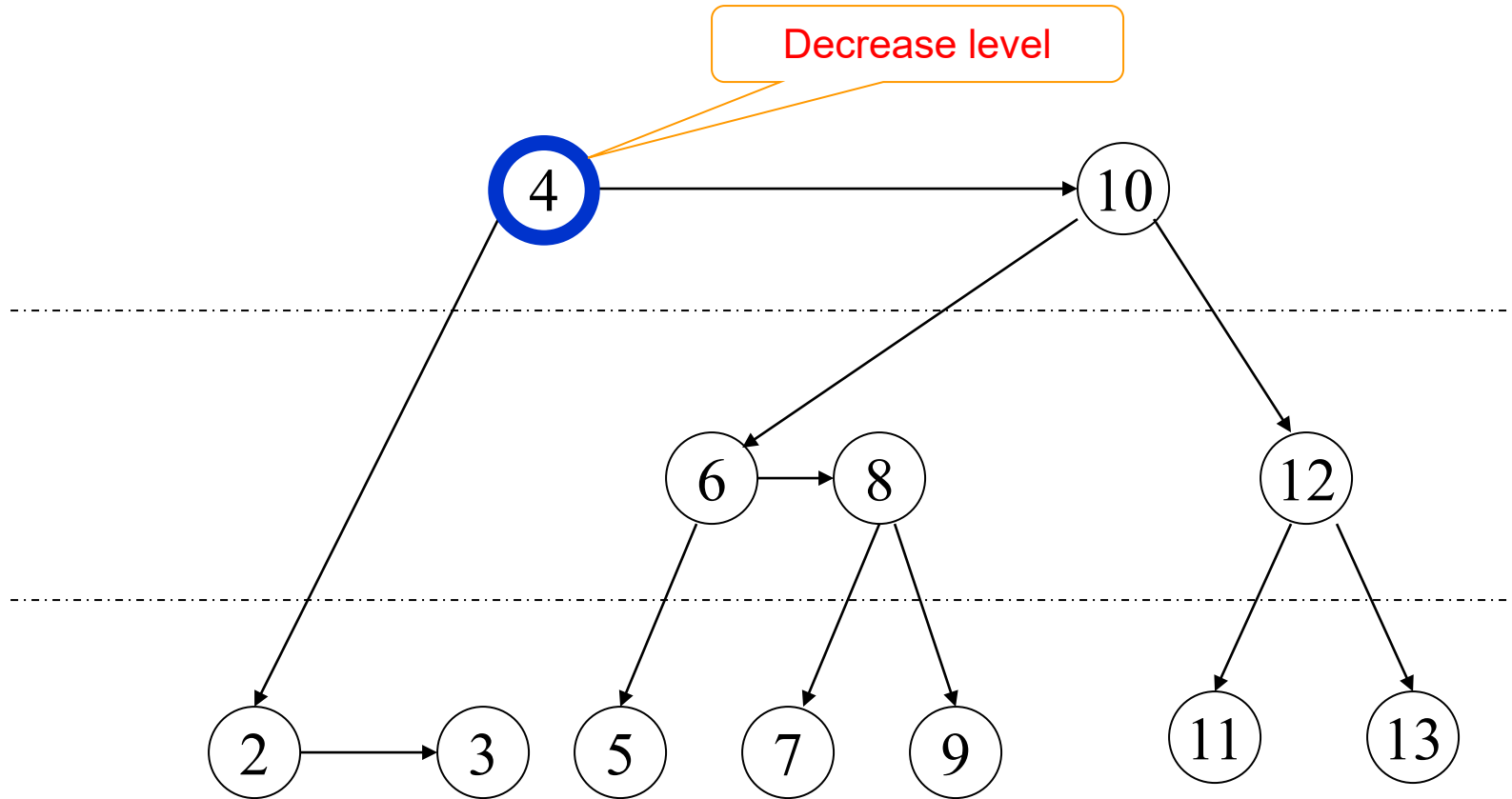
83

- Delete node 1:

Decrease Level

4 → 10

2

6 → 8

12

1

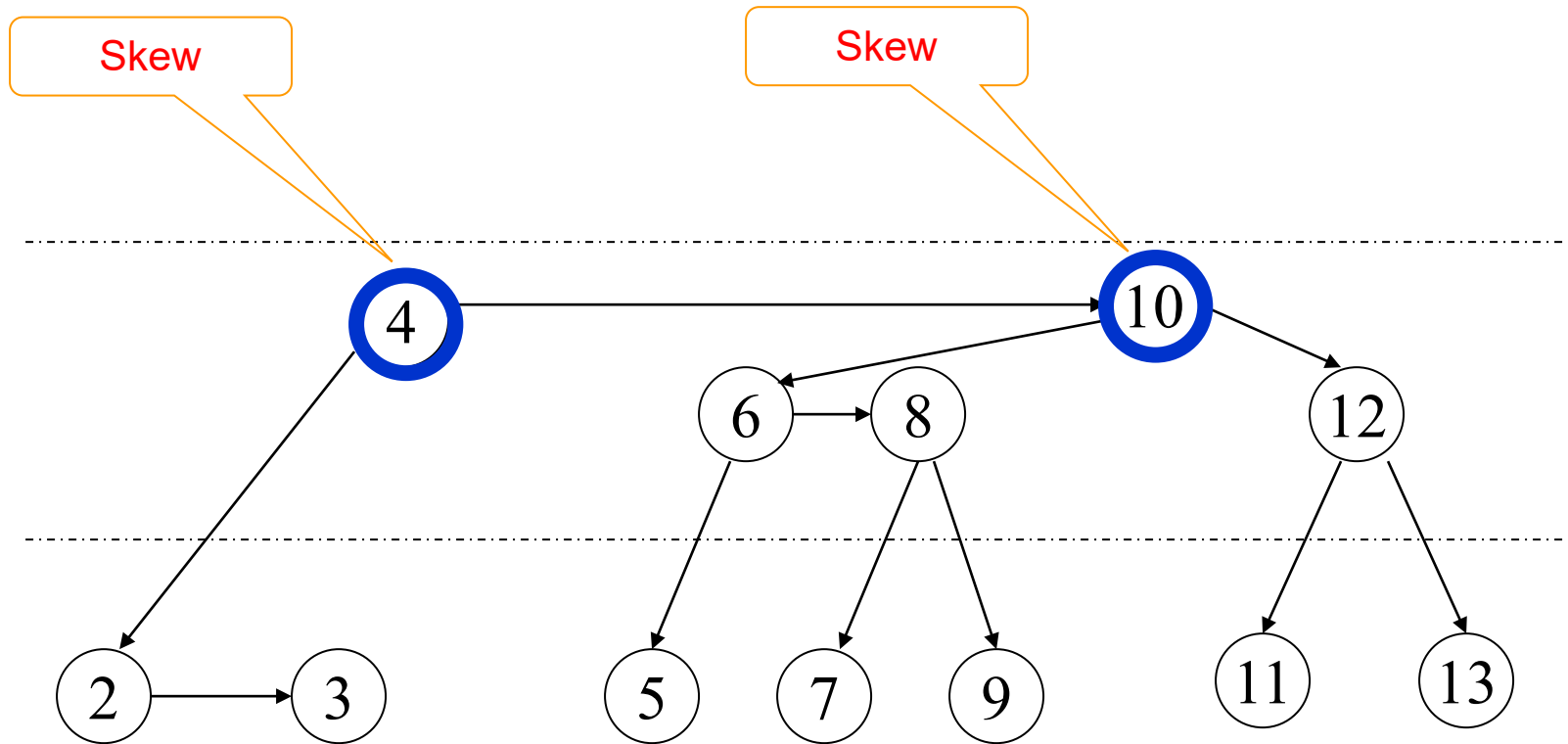3

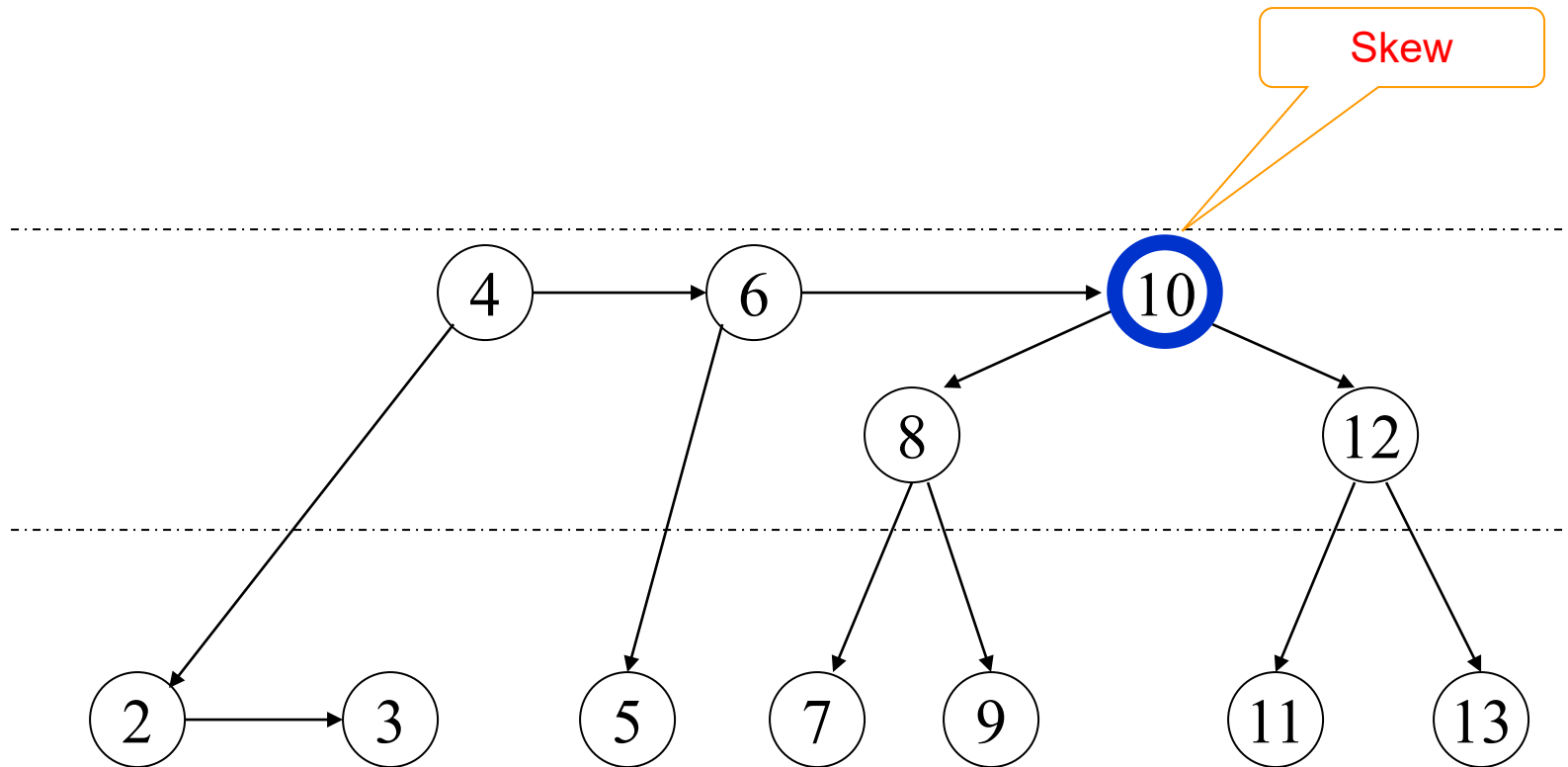5

7

9

11

13

Delete "1"

Example

- After decrease level of 2:

Example

- After decrease level of 4 and 10, skew at node 4 and 10:

Example

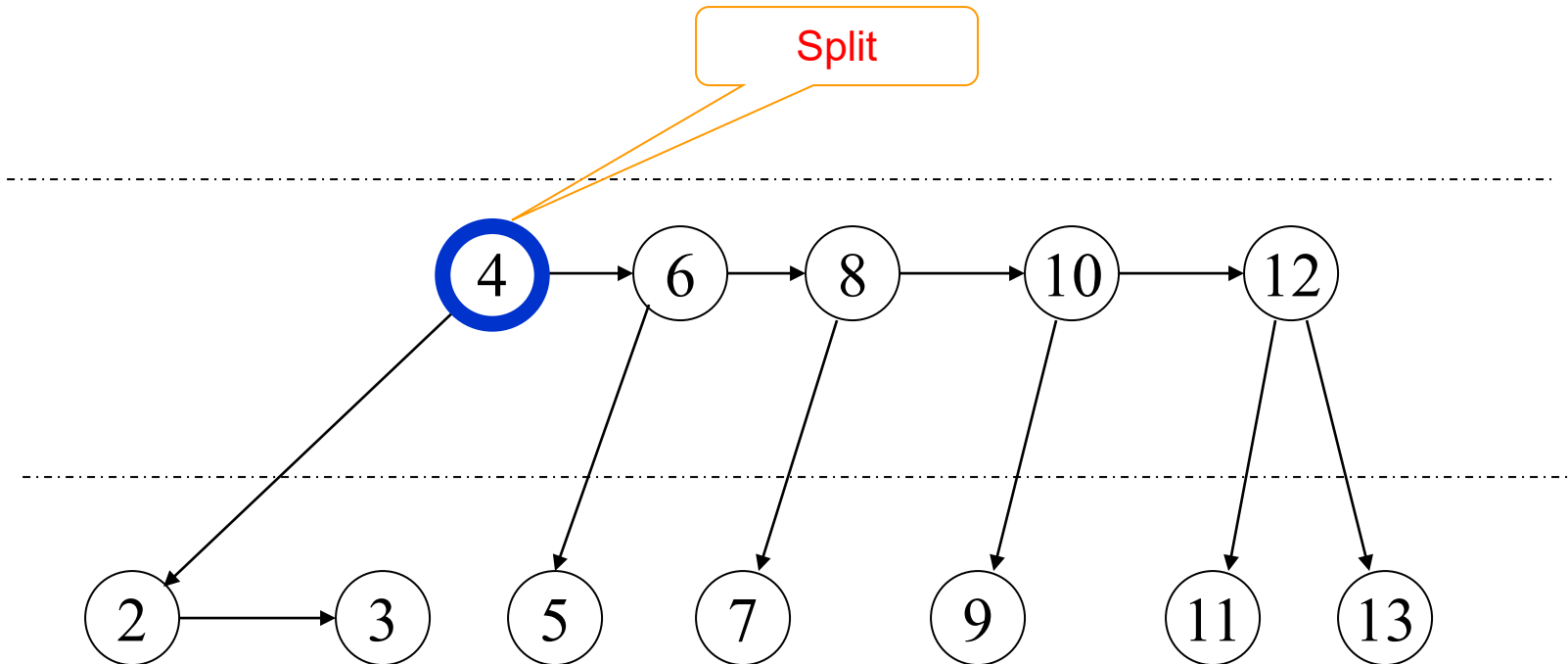- After skew at node 4 and 10, skew at node p→right→right (node 10):

Example

- After skew at node p→right→right (node 10):

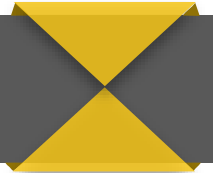# Example

- After split at node 4, split at node 8 (p→right):
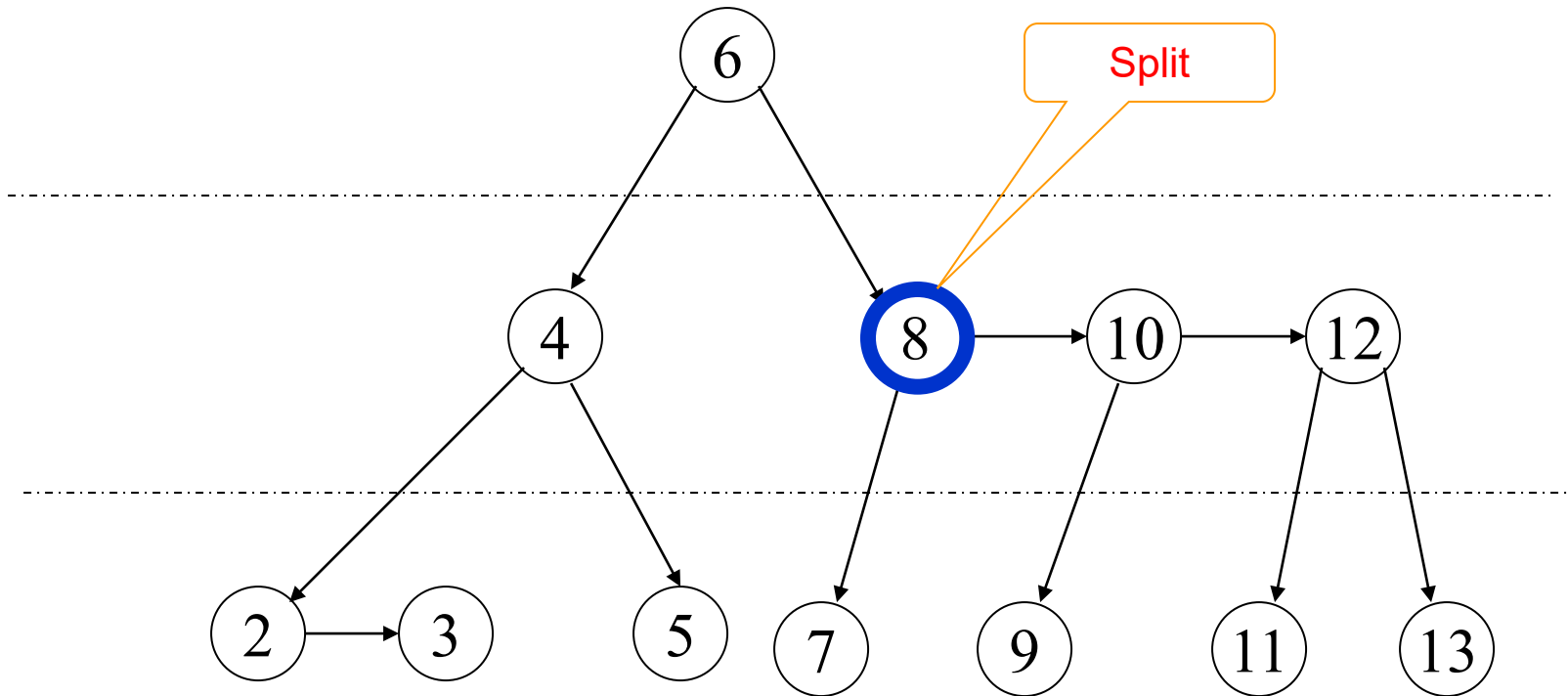
Example

- After split at node 8 (p→right):
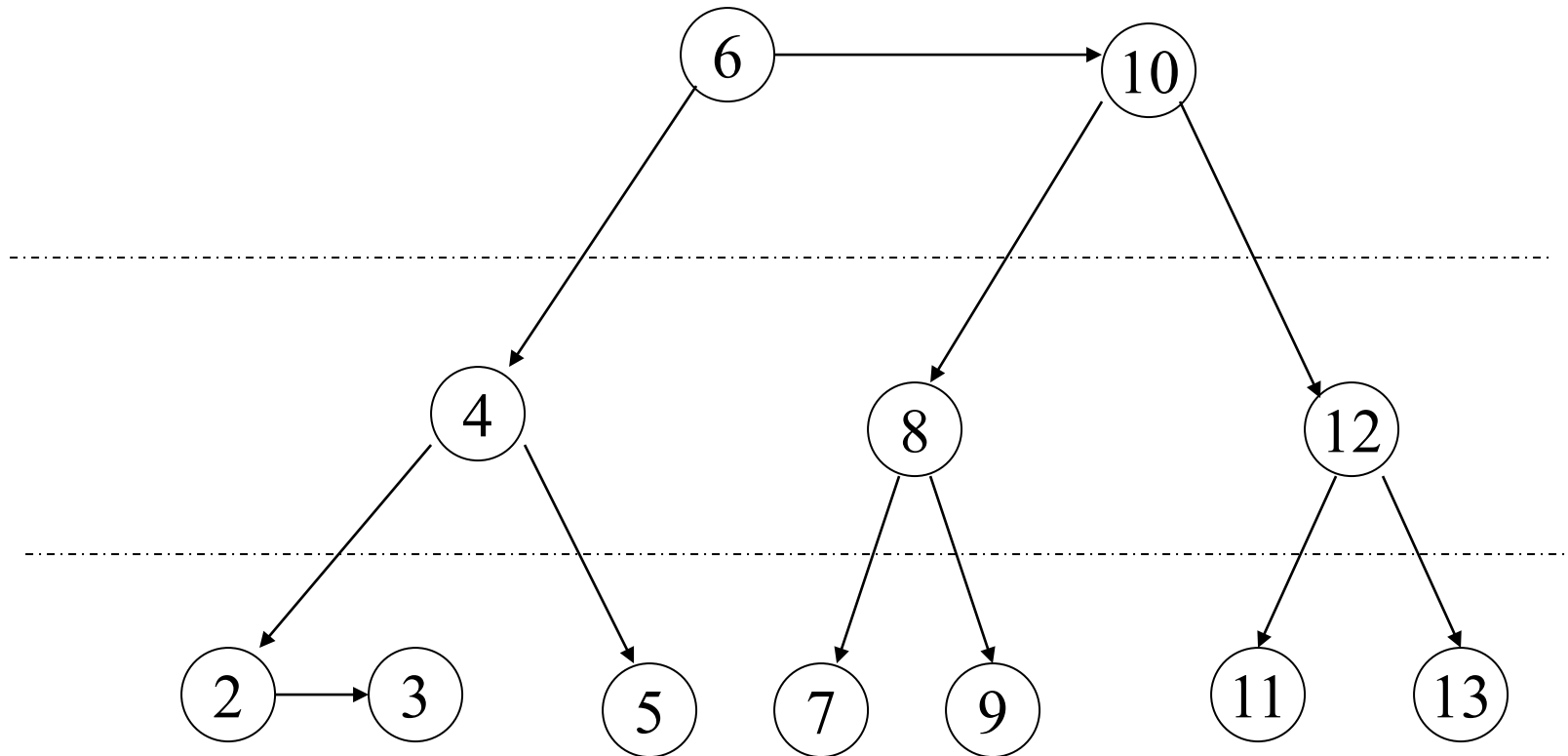
→ STOP !

AA Tree

- Comments:
  - Complexity $O(log_2N)$
  - No need to save pointer to parent node (pParent)
  - Simpler than the Red-Black tree

- Create AA-Tree by inserting the following values in order:

10, 85, 15, 70,20, 60, 30, 50, 65, 80,90, 40, 5, 55, 35, 2, 45, 17, 99

- After building AA Tree from previous data, delete values in tree in order:

30, 90, 5, 2, 85, 17, 55, 20, 65

The End.