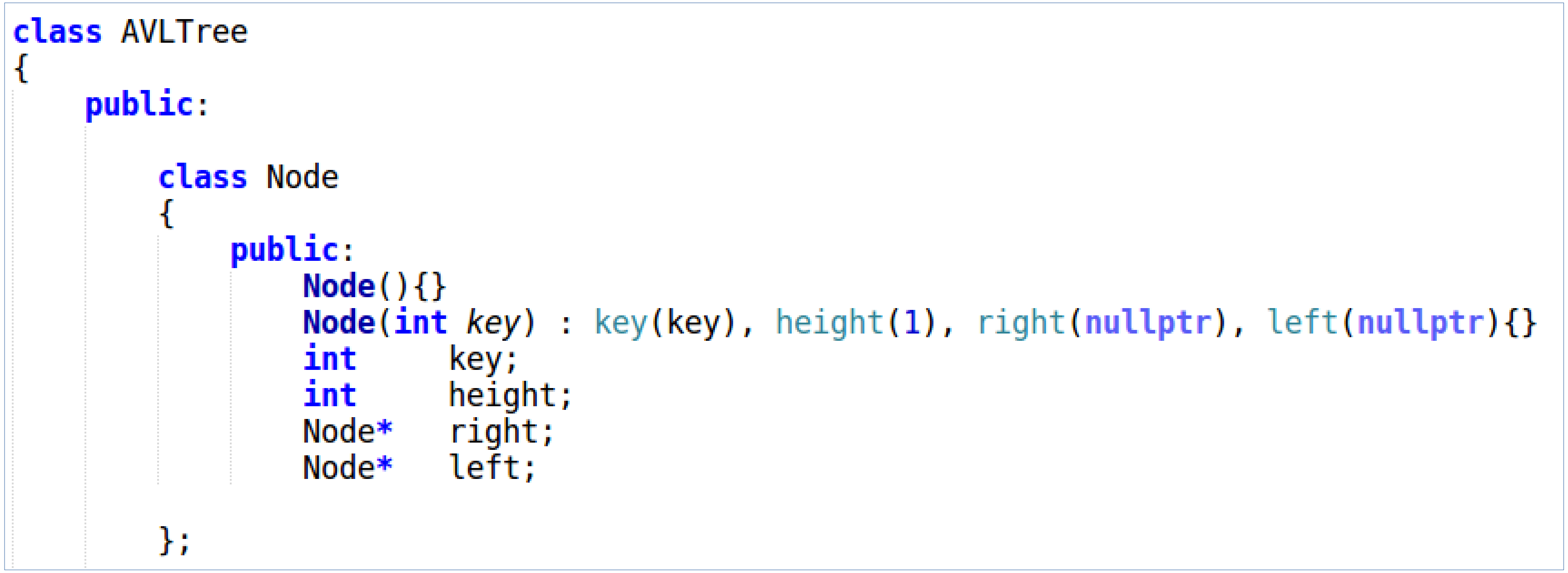
AVL – “self-balancing” binary tree

The heights h of the two child subtrees of any node x differ at most by one.

```
Balance Factor bf(x) = (x→left→height) – (x→right→height)

bf(x) must be -1, 0, or 1
```

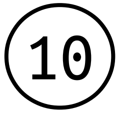
After an insertion/deletion, if the heights of left and right subtrees of node x differ more than one, the tree must automatically rebalance.

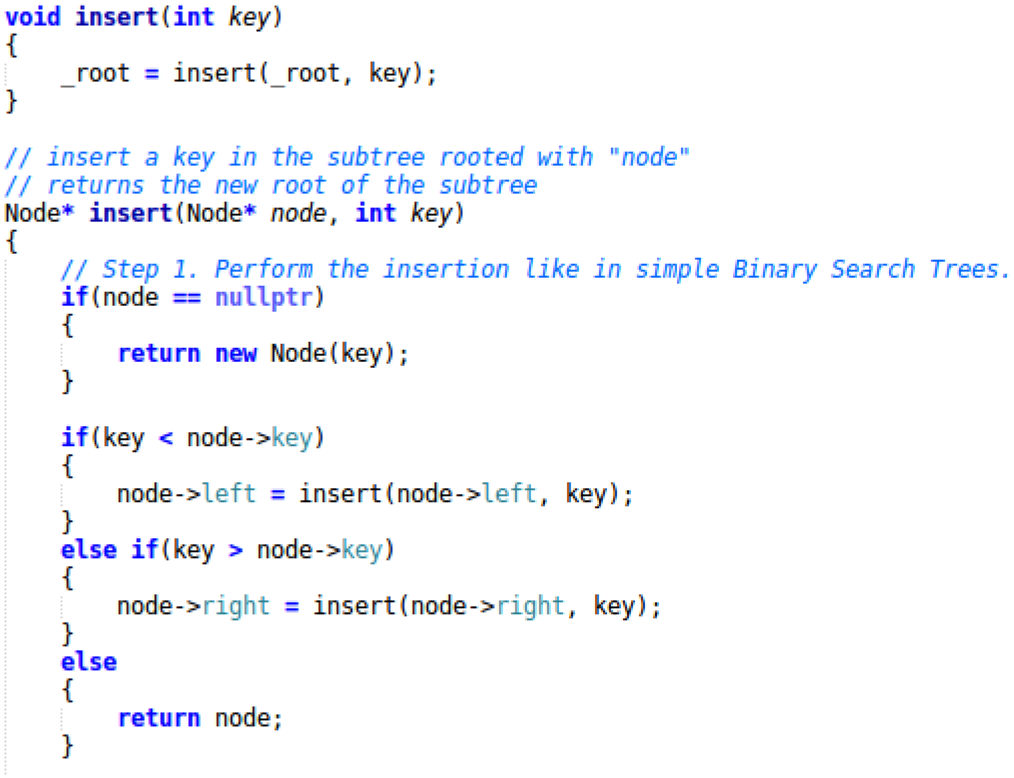


```
class AVLTree
{
    public:

        class Node
        {
            public:
                Node(){}
                Node(int key) : key(key), height(1), right(nullptr), left(nullptr){}
                int     key;
                int     height;
                Node*   right;
                Node*   left;

        };
```

AVL

Insert 10 20 30

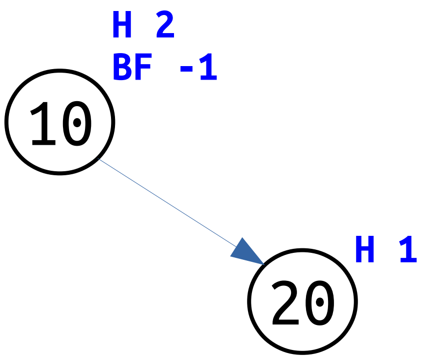


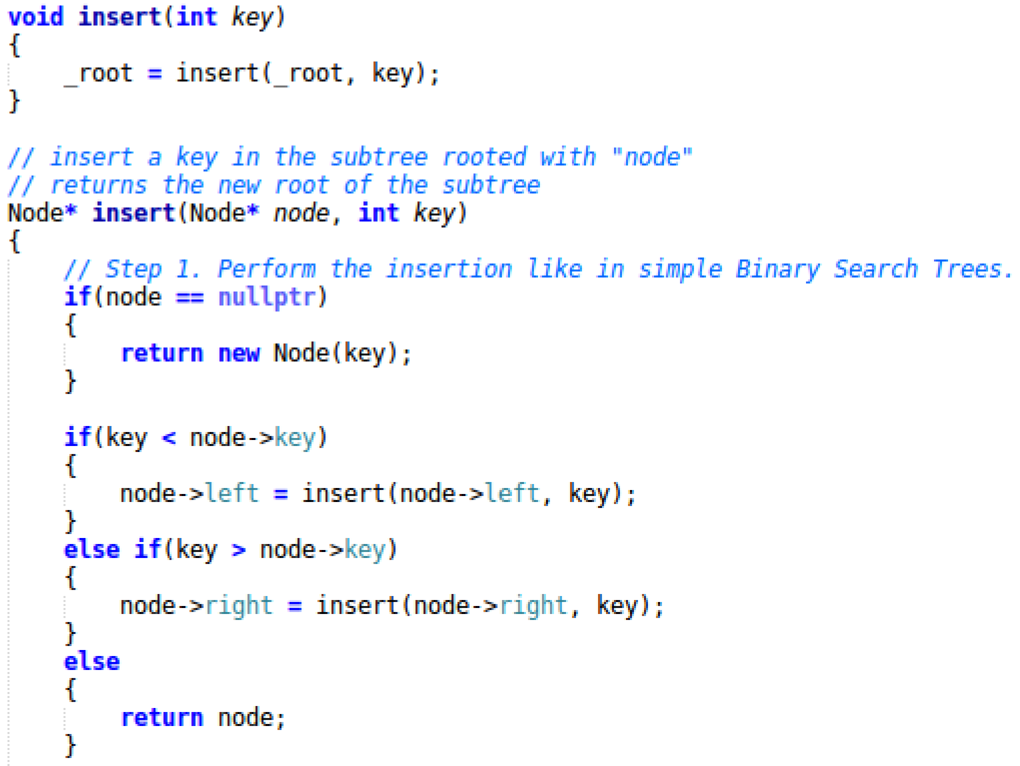


```
void insert(int key)
{
    _root = insert(_root, key);
}

// insert a key in the subtree rooted with "node"
// returns the new root of the subtree
Node* insert(Node* node, int key)
{
    // Step 1. Perform the insertion like in simple Binary Search Trees.
    if(node == nullptr)
    {
        return new Node(key);
    }

    if(key < node->key)
    {
        node->left = insert(node->left, key);
    }
    else if(key > node->key)
    {
        node->right = insert(node->right, key);
    }
    else
    {
        return node;
    }
```

AVL

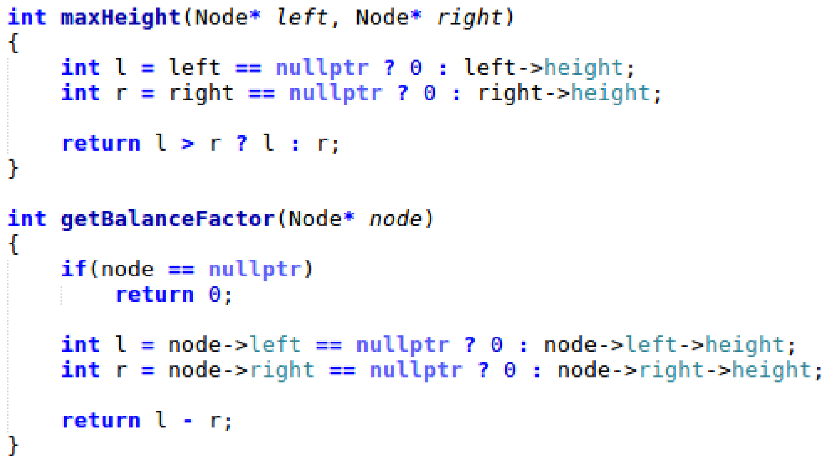
Insert 10 20 30





```
void insert(int key)
{
    _root = insert(_root, key);
}

// insert a key in the subtree rooted with "node"
// returns the new root of the subtree
Node* insert(Node* node, int key)
{
    // Step 1. Perform the insertion like in simple Binary Search Trees.
    if(node == nullptr)
    {
        return new Node(key);
    }

    if(key < node->key)
    {
        node->left = insert(node->left, key);
    }
    else if(key > node->key)
    {
        node->right = insert(node->right, key);
    }
    else
    {
        return node;
    }
```

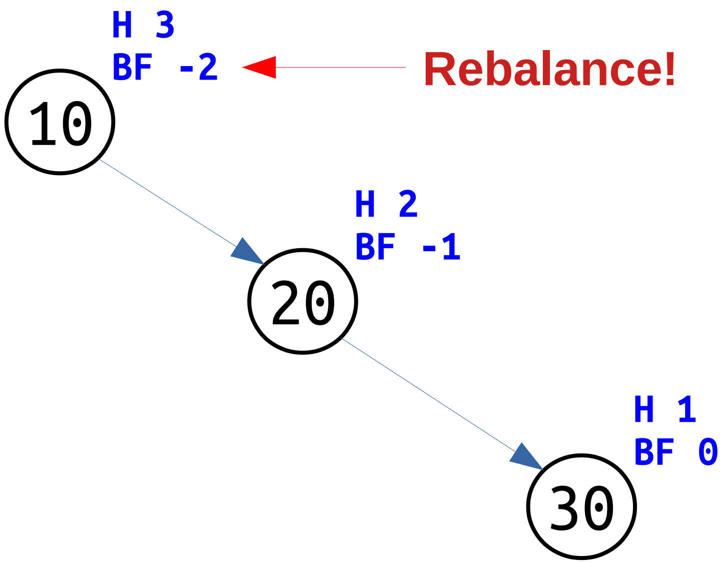
```
// Step 2. Update height of this ancestor
node->height = 1 + maxHeight(node->left,node->right);

// Step 3. Check the balance factor of this ancestor
int bf = getBalanceFactor(node);
```

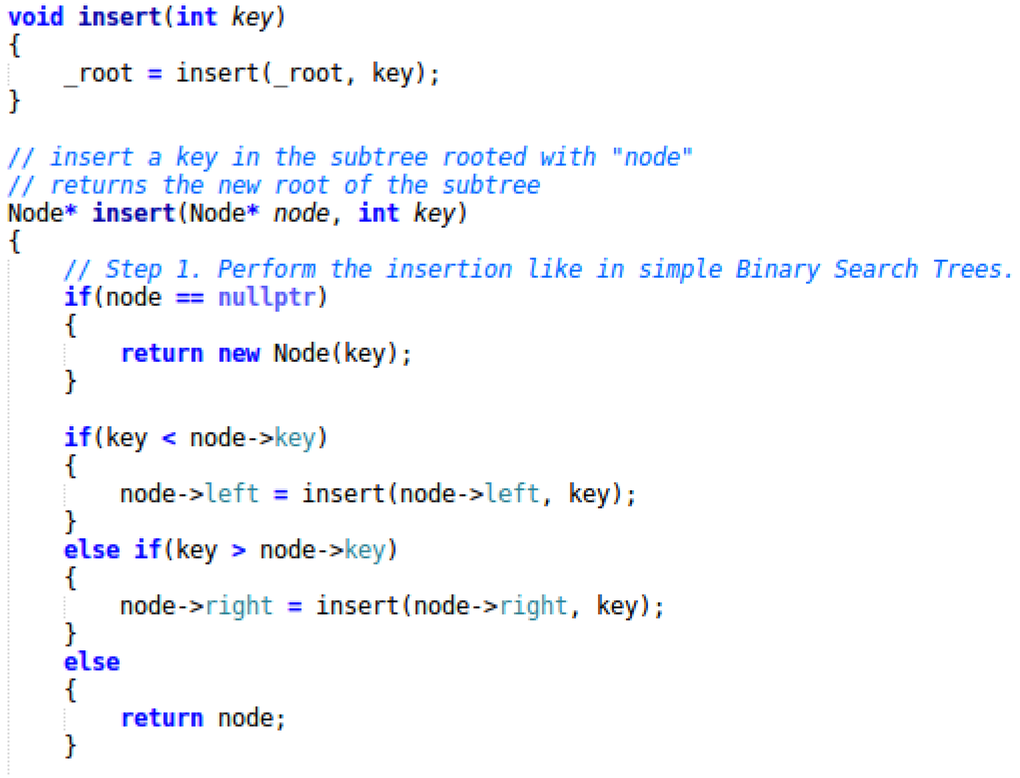


```
int maxHeight(Node* left, Node* right)
{
    int l = left == nullptr ? 0 : left->height;
    int r = right == nullptr ? 0 : right->height;

    return l > r ? l : r;
}

int getBalanceFactor(Node* node)
{
    if(node == nullptr)
        return 0;

    int l = node->left == nullptr ? 0 : node->left->height;
    int r = node->right == nullptr ? 0 : node->right->height;

    return l - r;
}
```

AVL

Insert 10 20 30



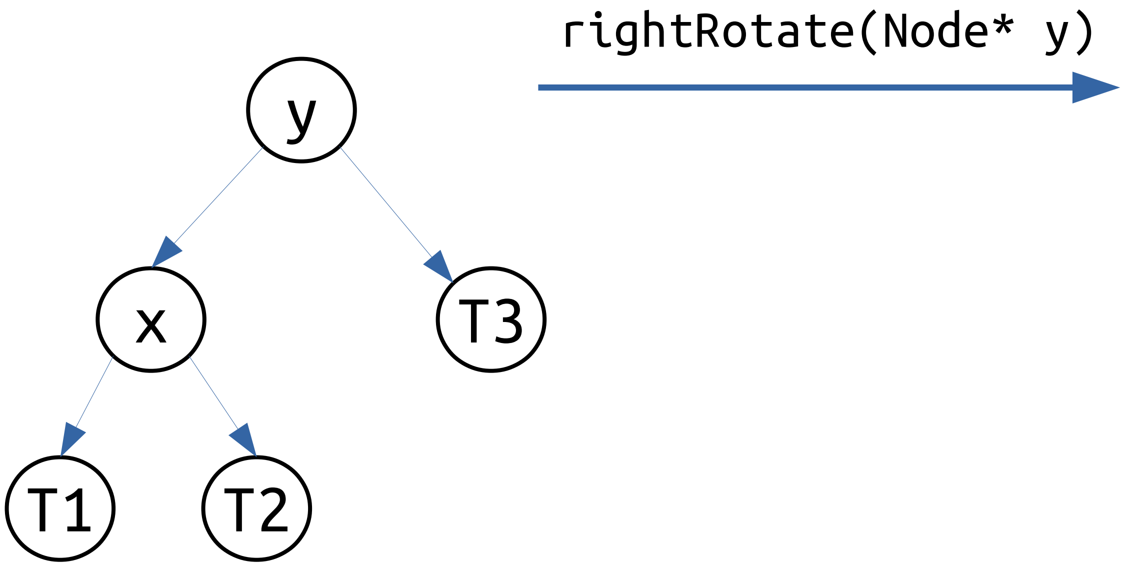


```
void insert(int key)
{
    _root = insert(_root, key);
}

// insert a key in the subtree rooted with "node"
// returns the new root of the subtree
Node* insert(Node* node, int key)
{
    // Step 1. Perform the insertion like in simple Binary Search Trees.
    if(node == nullptr)
    {
        return new Node(key);
    }

    if(key < node->key)
    {
        node->left = insert(node->left, key);
    }
    else if(key > node->key)
    {
        node->right = insert(node->right, key);
    }
    else
    {
        return node;
    }
```

```
// Step 2. Update height of this ancestor
node->height = 1 + maxHeight(node->left,node->right);

// Step 3. Check the balance factor of this ancestor
int bf = getBalanceFactor(node);
```

```
// Step 4. If the subtree is unbalanced, check in which case we are, and balance!
```

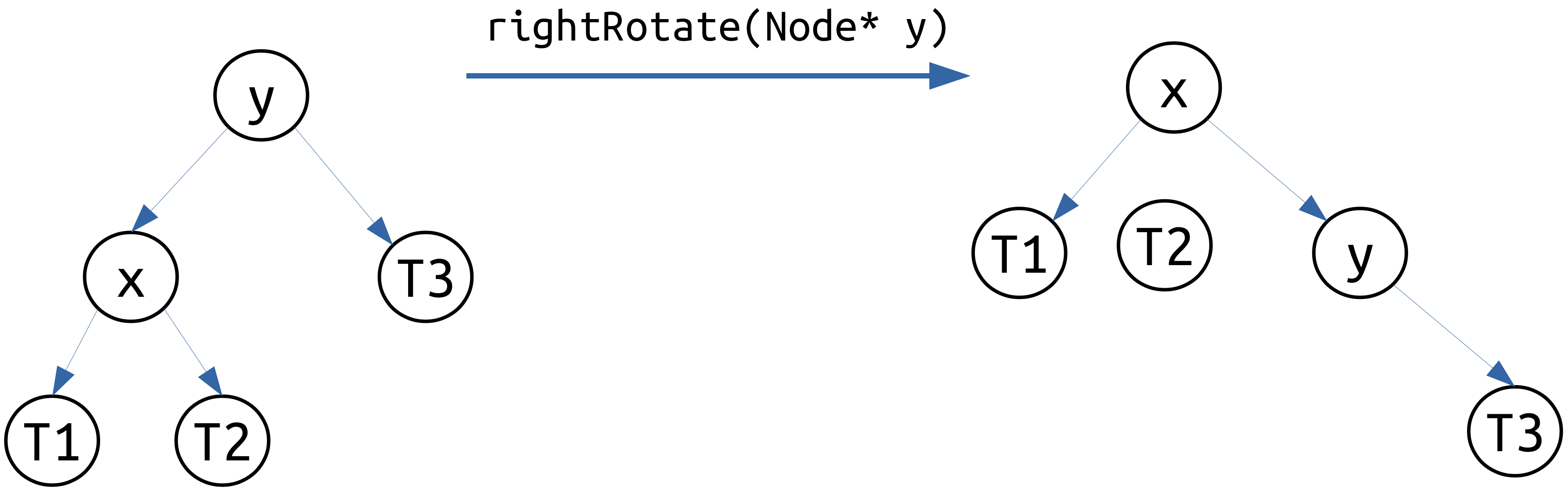
AVL - Rotations



rightRotate(Node* y)

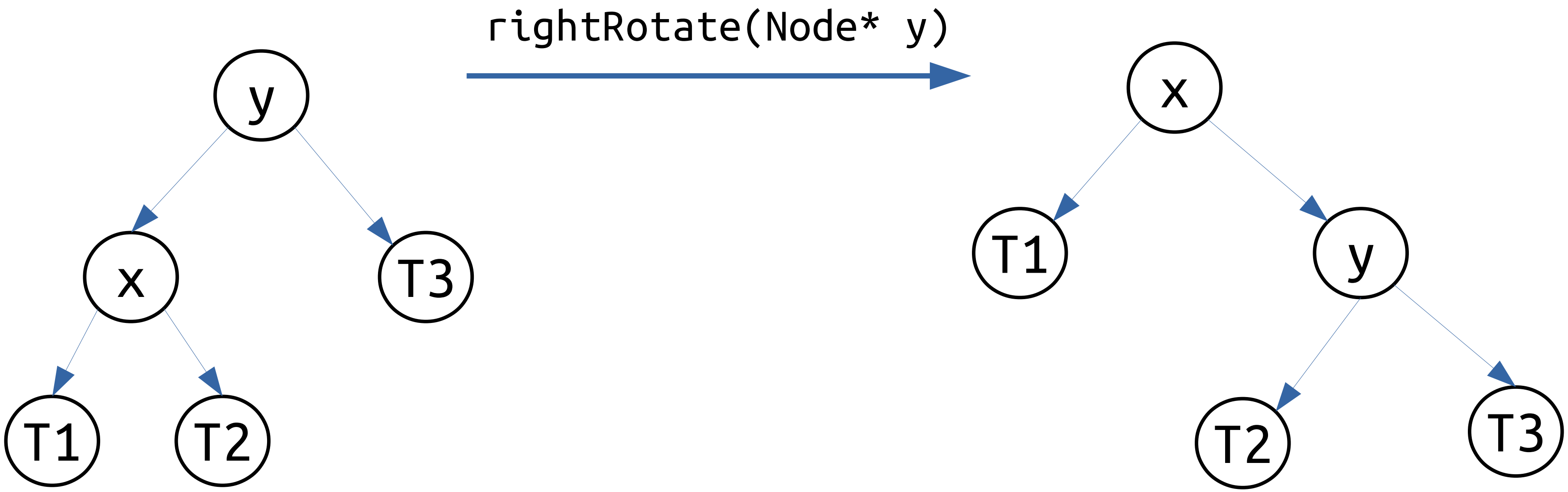
```
Node* rightRotate(Node *y)
//1.  Let x be the left child of y
//2.  Let t2 be the right child of x
--- rotation step ---
//3. The right child of x will be y
//4.  The left child of y will be T2
--- update heights ---
//5. height of y =(max height of his children) + 1
//6. height of x =(max height of x's children) + 1
return x;
```

AVL - Rotations



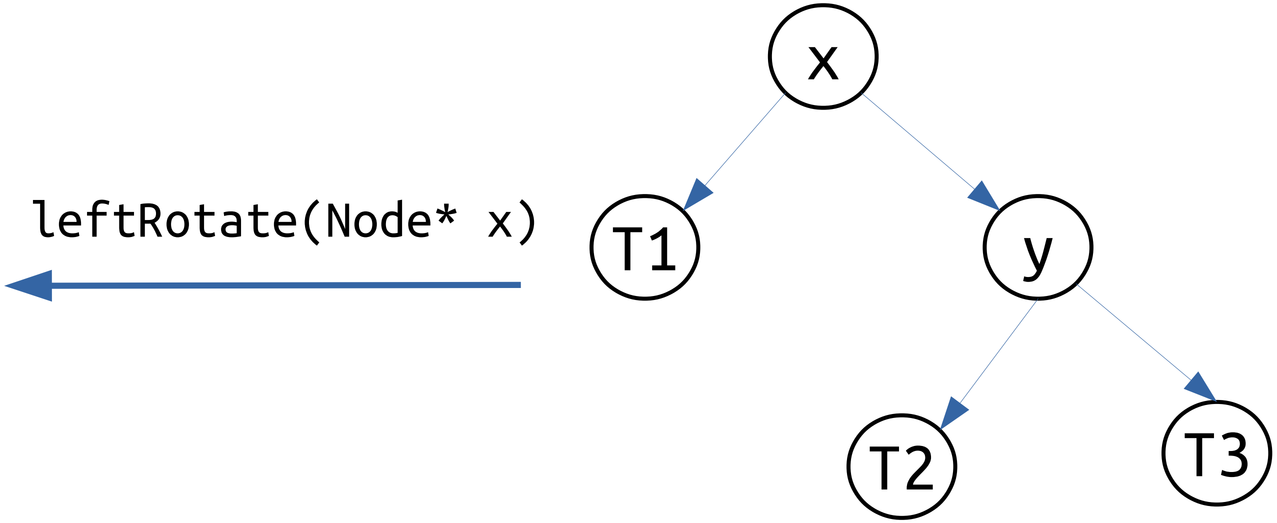
```
Node* rightRotate(Node *y)
//1.  Let x be the left child of y
//2.  Let t2 be the right child of x
--- rotation step ---
//3. The right child of x will be y
//4.  The left child of y will be T2
--- update heights ---
//5. height of y =(max height of his children) + 1
//6. height of x =(max height of x’s children) + 1
return x;
```

AVL - Rotations



```
Node* rightRotate(Node *y)

//1.  Let x be the left child of y

//2.  Let t2 be the right child of x

--- rotation step ---

//3. The right child of x will be y

//4.  The left child of y will be T2

--- update heights ---

//5. height of y =(max height of y’s children) + 1

//6. height of x =(max height of x’s children) + 1

return x;
```

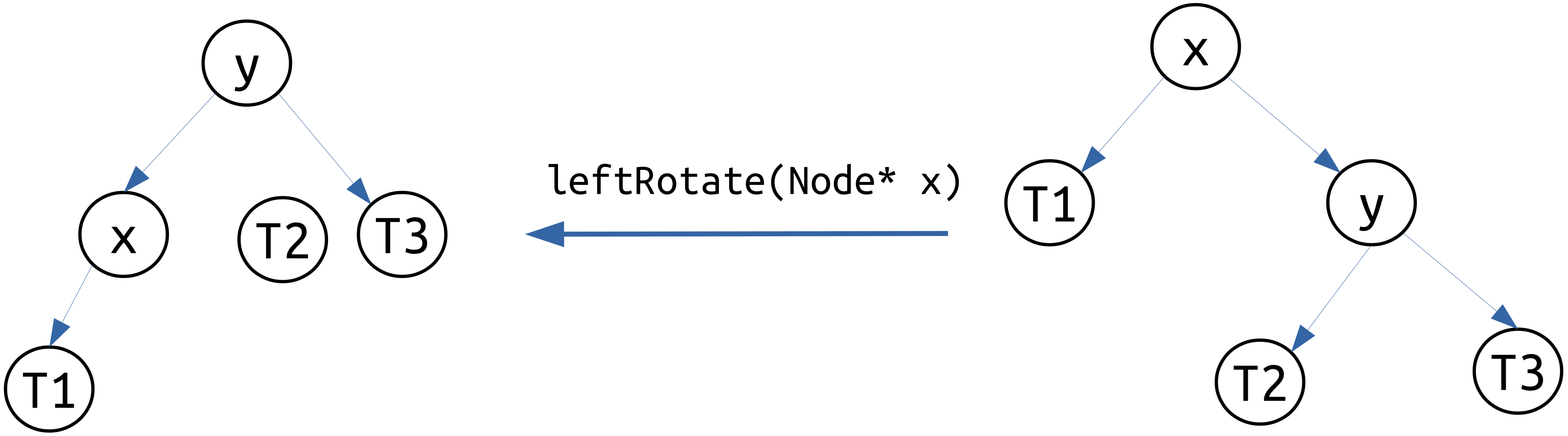
AVL - Rotations



leftRotate(Node* x)

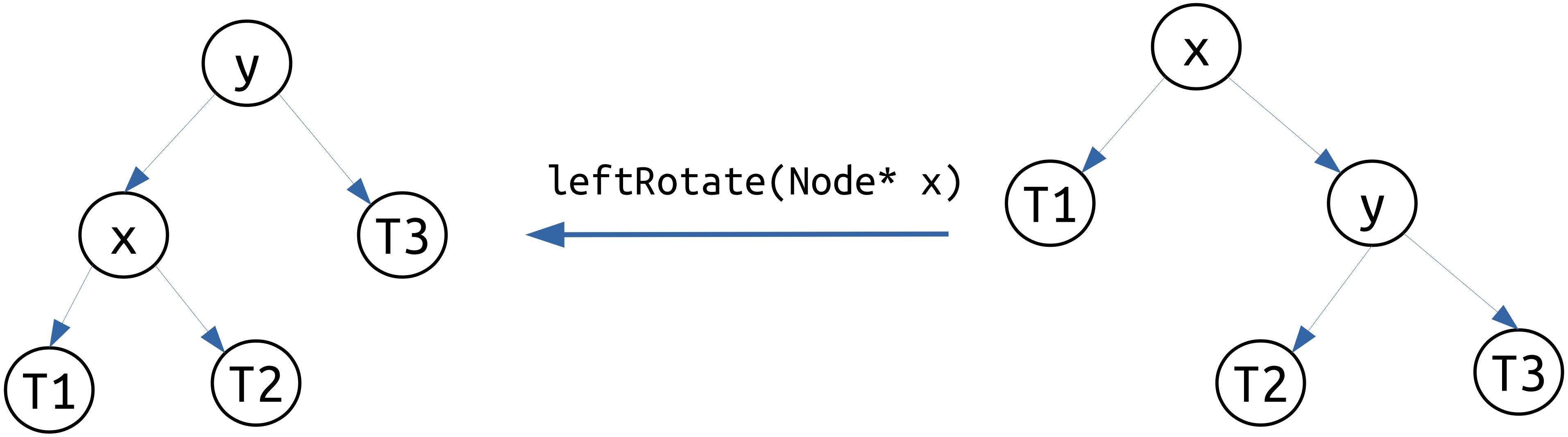
```
Node* leftRotate(Node *x)
//1.  Let y be the right child of x

//2.  Let t2 be the left child of x

--- rotation step ---

//3. The left child of y will be x

//4.  The right child of x will be t2

--- update heights ---

//5. height of x =(max height of x’s children) + 1

//6. height of y =(max height of y’s children) + 1

return y;
```

AVL - Rotations



```
Node* leftRotate(Node *x)

//1.  Let y be the right child of x

//2.  Let t2 be the left child of x

--- rotation step ---

//3. The left child of y will be x

//4.  The right child of x will be t2

--- update heights ---

//5. height of x =(max height of x’s children) + 1

//6. height of y =(max height of y’s children) + 1

return y;
```

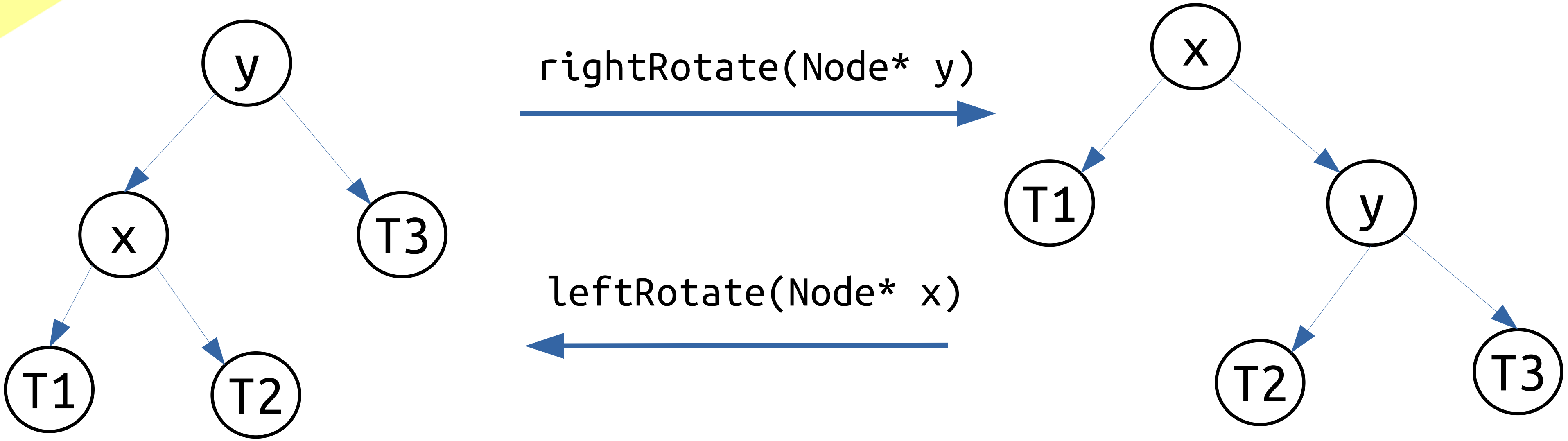
AVL - Rotations



```
Node* leftRotate(Node *x)
//1.  Let y be the right child of x
//2.  Let t2 be the left child of x
--- rotation step ---
//3. The left child of y will be x
//4.  The right child of x will be t2
--- update heights ---
//5. height of x =(max height of x’s children) + 1
//6. height of y =(max height of y’s children) + 1
return y;
```

Important

AVL - Rotations



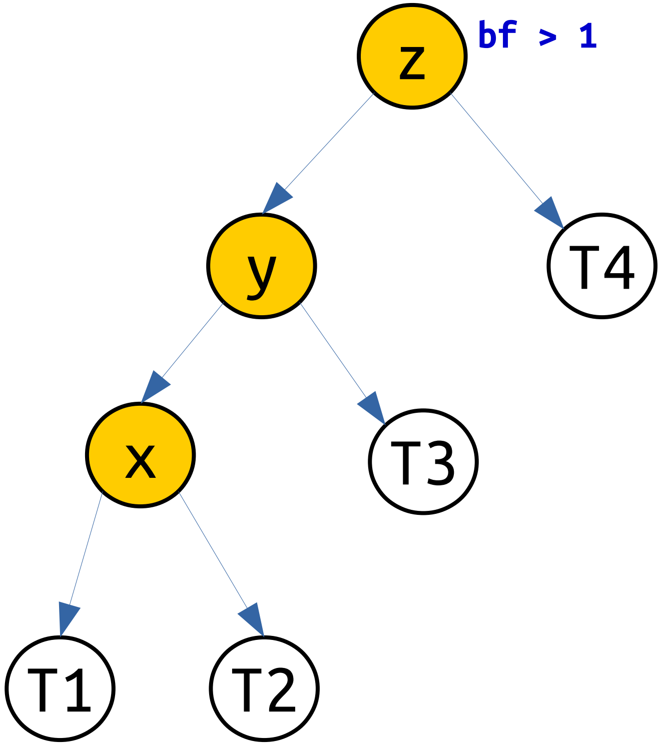
```
Node* rightRotate(Node *y)
//1.  Let x be the left child of y

//2.  Let t2 be the right child of x

--- rotation step ---

//3. The right child of x will be y

//4.  The left child of y will be T2

--- update heights ---

//5. height of y =(max height of y’s children) + 1

//6. height of x =(max height of x’s children) + 1

return x;
```

```
Node* leftRotate(Node *x)
//1.  Let y be the right child of x

//2.  Let t2 be the left child of x

--- rotation step ---

//3. The left child of y will be x

//4.  The right child of x will be t2

--- update heights ---

//5. height of x =(max height of x’s children) + 1

//6. height of y =(max height of y’s children) + 1

return y;
```

AVL - Cases

Case 1. LEFT LEFT

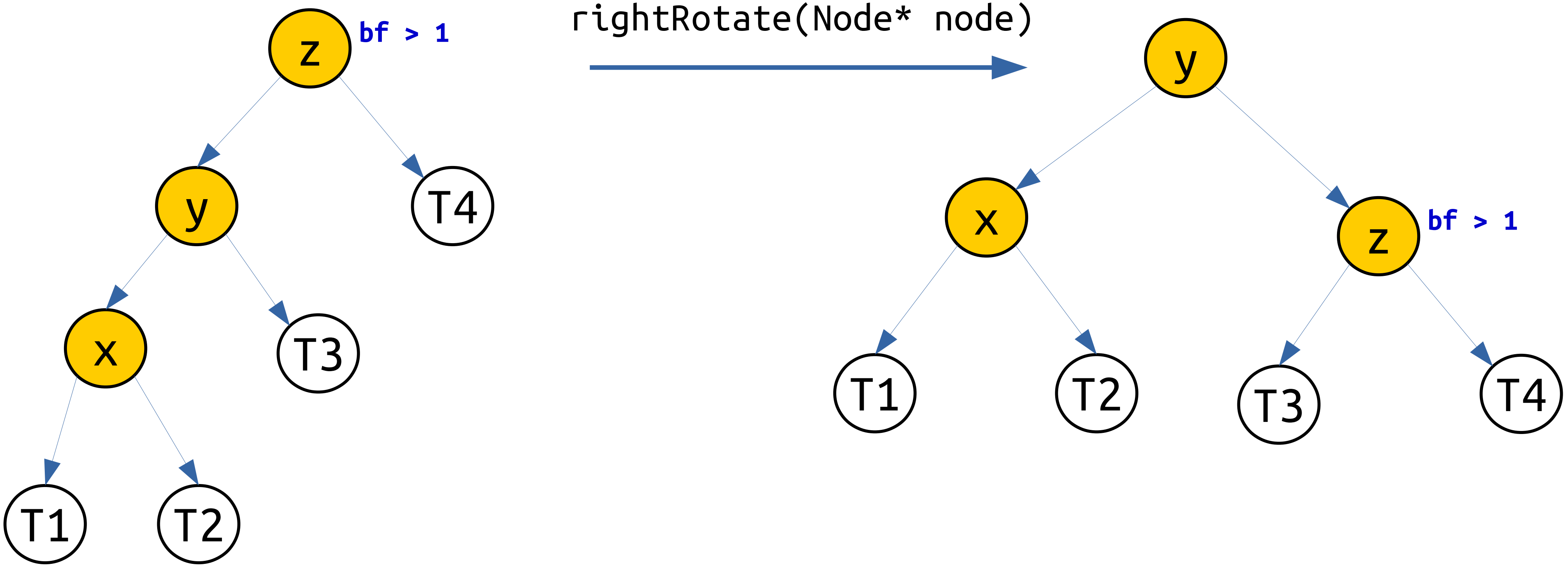
`bf > 1 and key < node->left->key`



AVL - Cases

Case 1. LEFT LEFT

bf > 1 and key < node->left->key



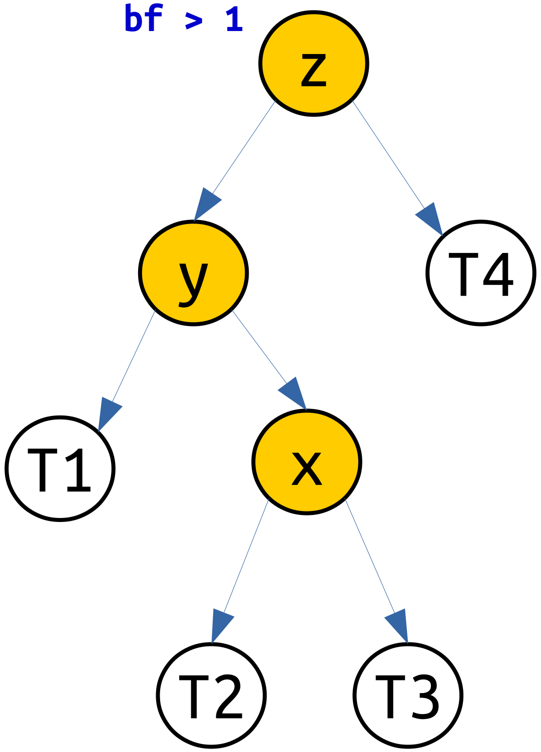
rightRotate(Node* node)

AVL - Cases

Case 2. LEFT RIGHT

bf > 1 and key > node→left->key

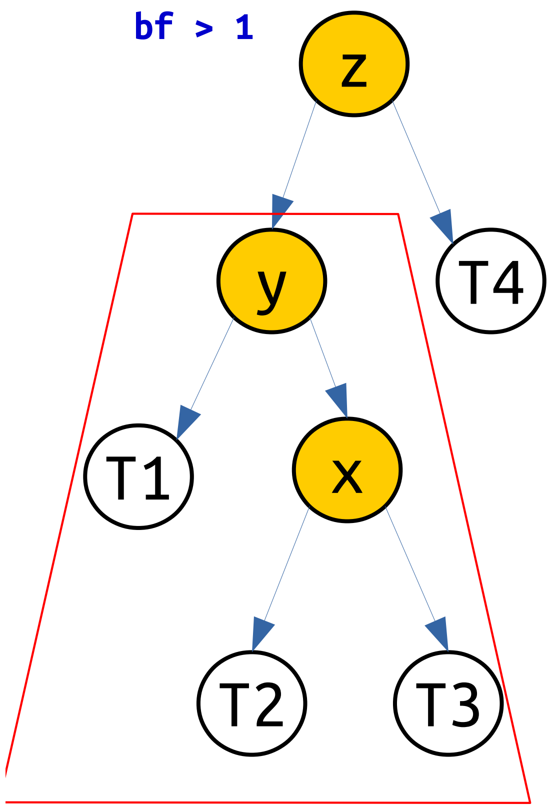
bf > 1



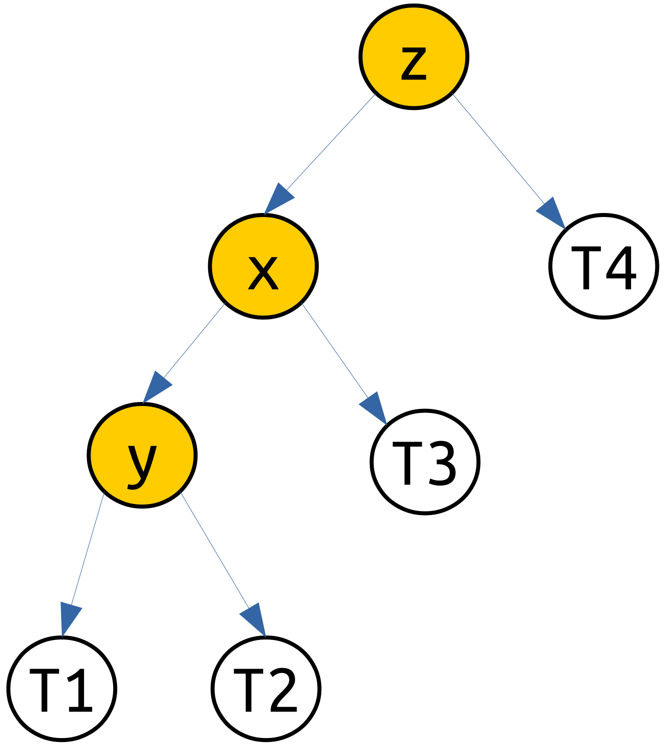
AVL - Cases

Case 2. LEFT RIGHT

bf > 1 and key > node→left->key



leftRotate(node->left)



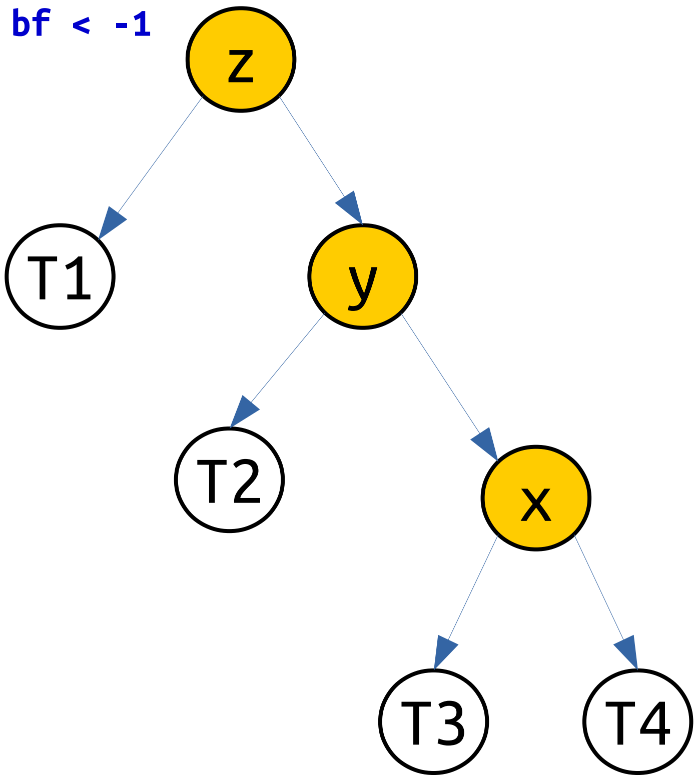
Now, this looks like Case 1. LEFT LEFT

Do right rotation on node z.

AVL - Cases

Case 3. RIGHT RIGHT

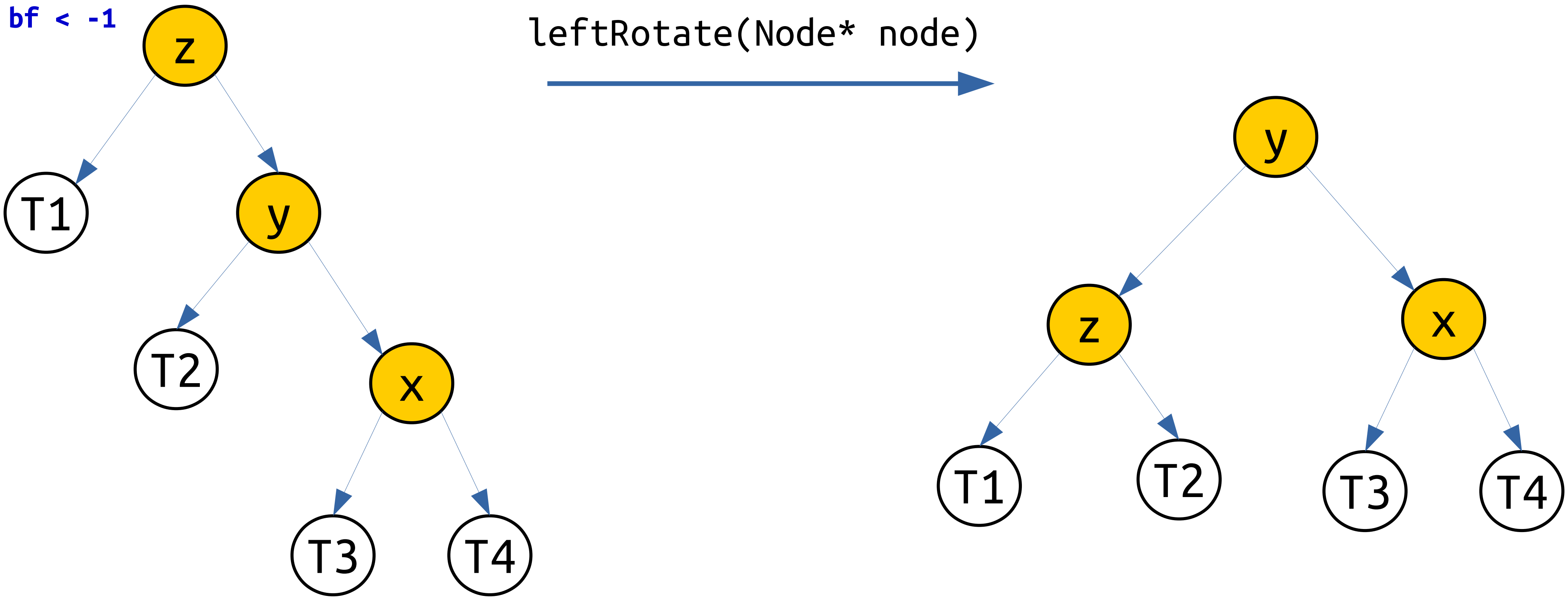
bf < -1 and key > node→right->key



AVL - Cases

Case 3. RIGHT RIGHT

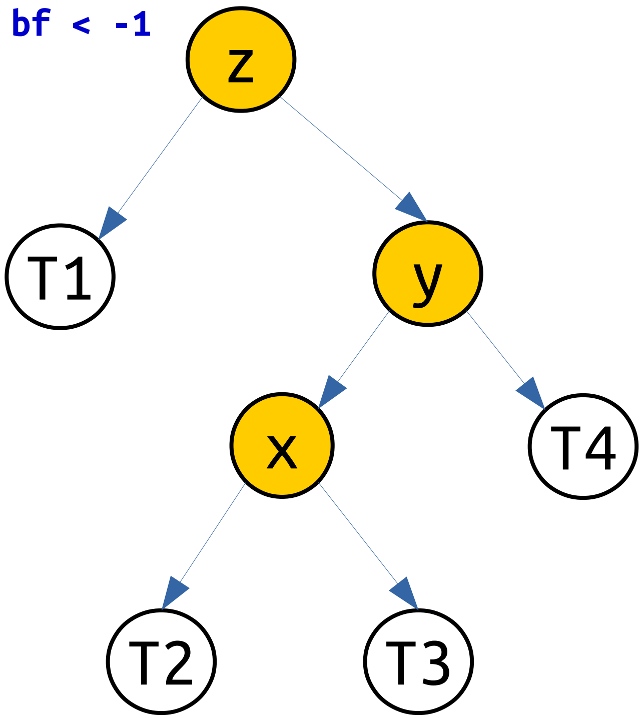
bf < -1 and key > node→right->key



AVL - Cases

Case 4. RIGHT LEFT

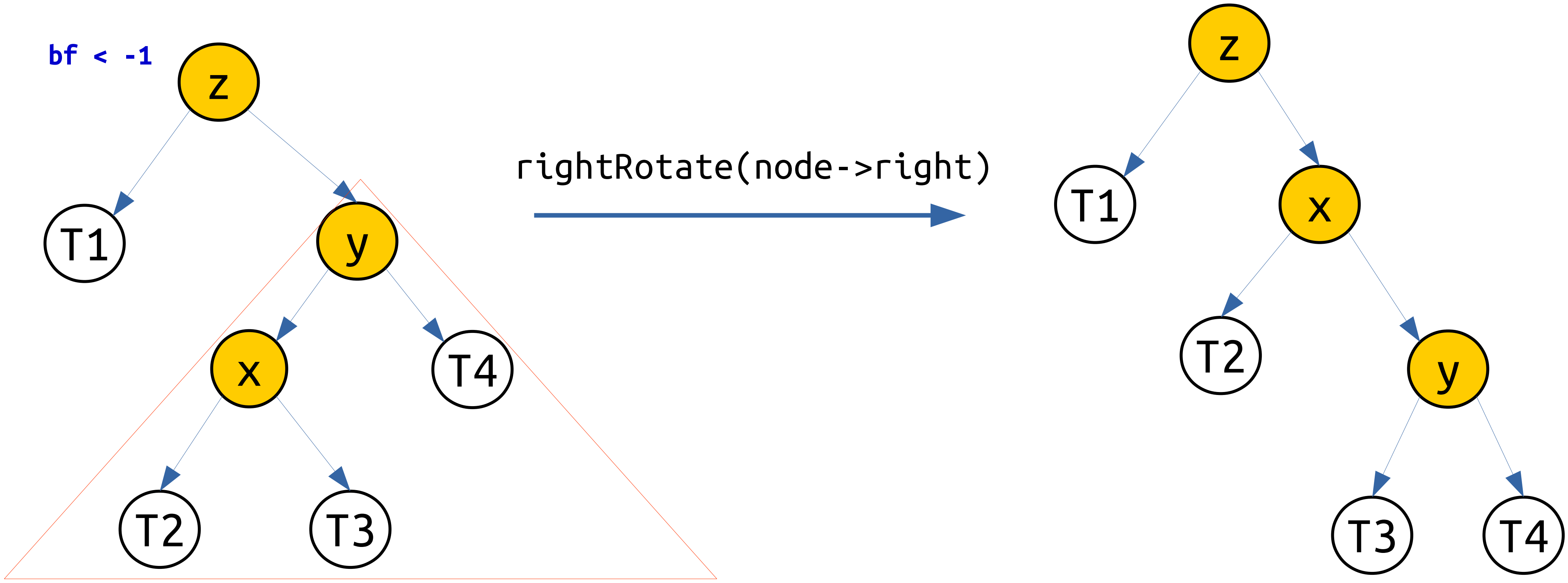
bf < -1 and key < node→right->key



AVL - Cases

Case 4. RIGHT LEFT

bf < -1 and key < node→right->key

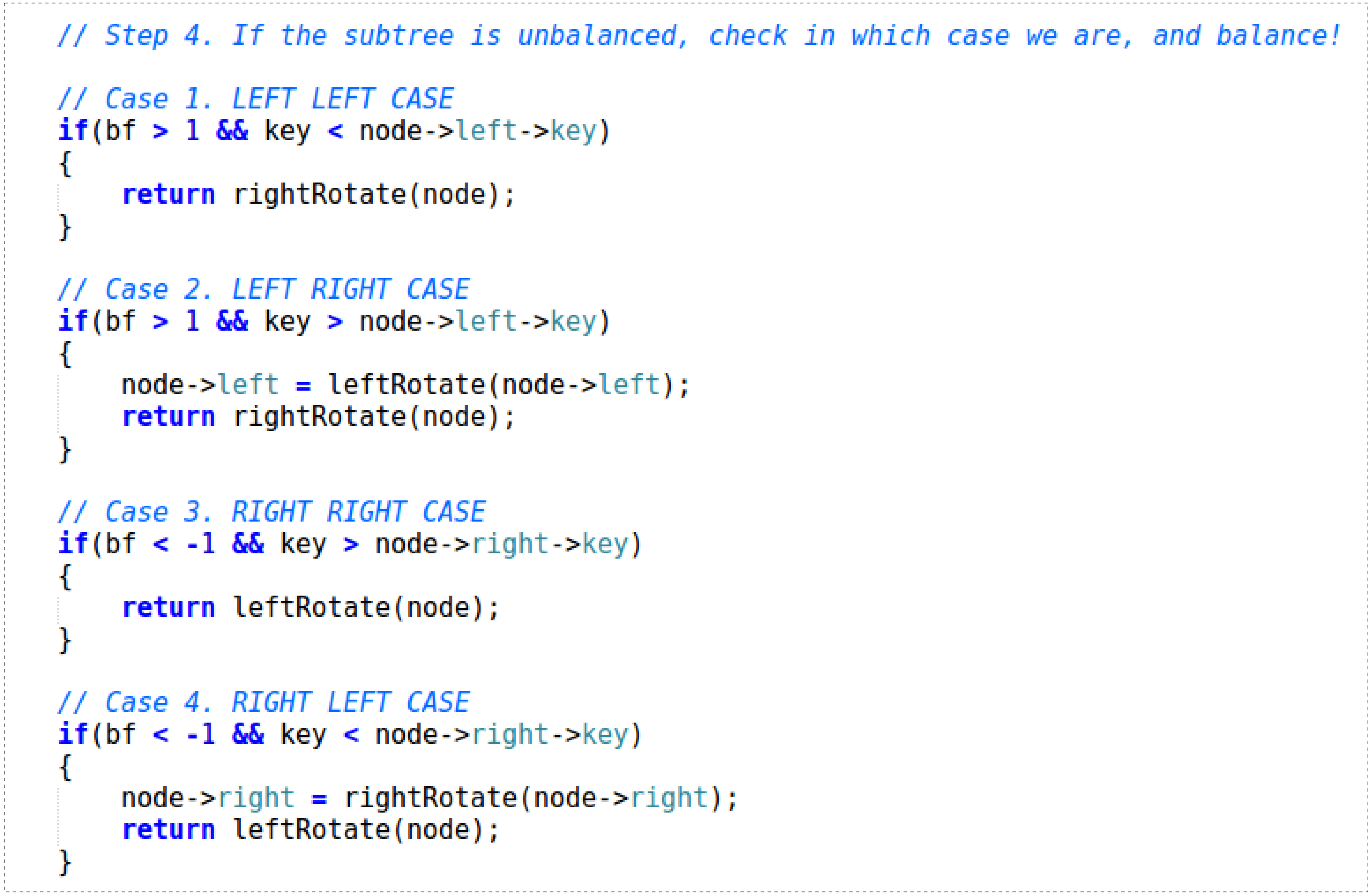


rightRotate(node->right)

Now, this looks like Case 3. RIGHT RIGHT

Do left rotation on node z.

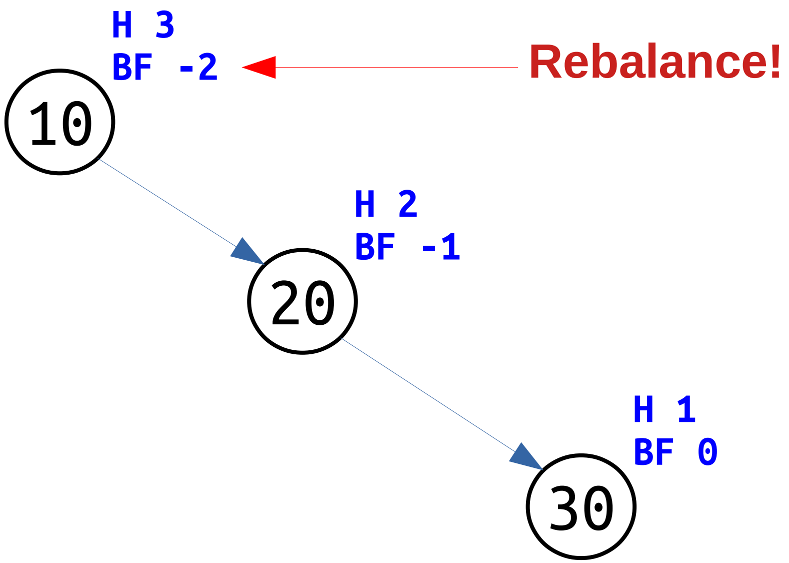
AVL - Cases



```
// Step 4. If the subtree is unbalanced, check in which case we are, and balance!

// Case 1. LEFT LEFT CASE
if(bf > 1 && key < node->left->key)
{
    return rightRotate(node);
}

// Case 2. LEFT RIGHT CASE
if(bf > 1 && key > node->left->key)
{
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Case 3. RIGHT RIGHT CASE
if(bf < -1 && key > node->right->key)
{
    return leftRotate(node);
}

// Case 4. RIGHT LEFT CASE
if(bf < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

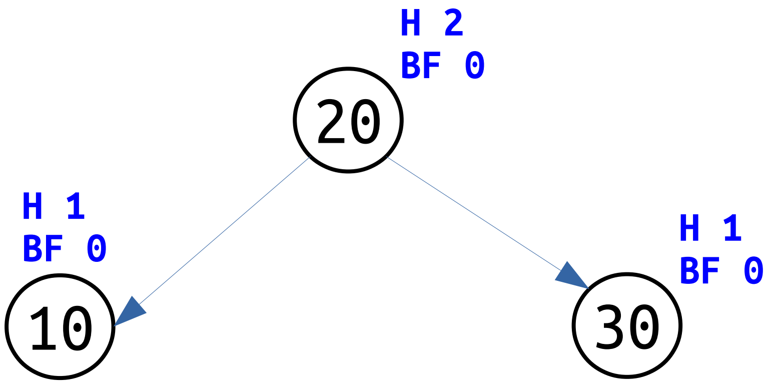
AVL

Insert 30



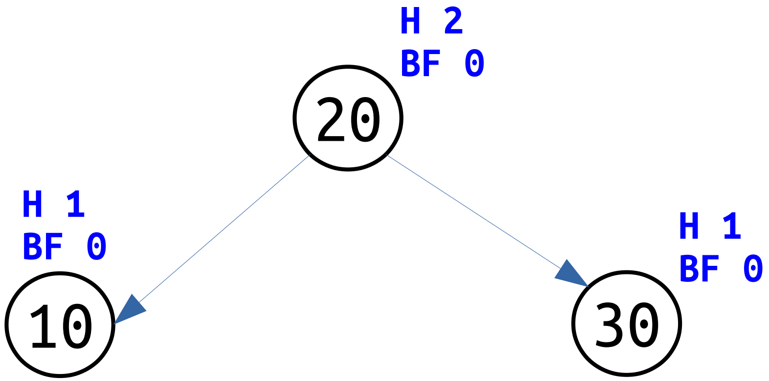
AVL

Insert 30



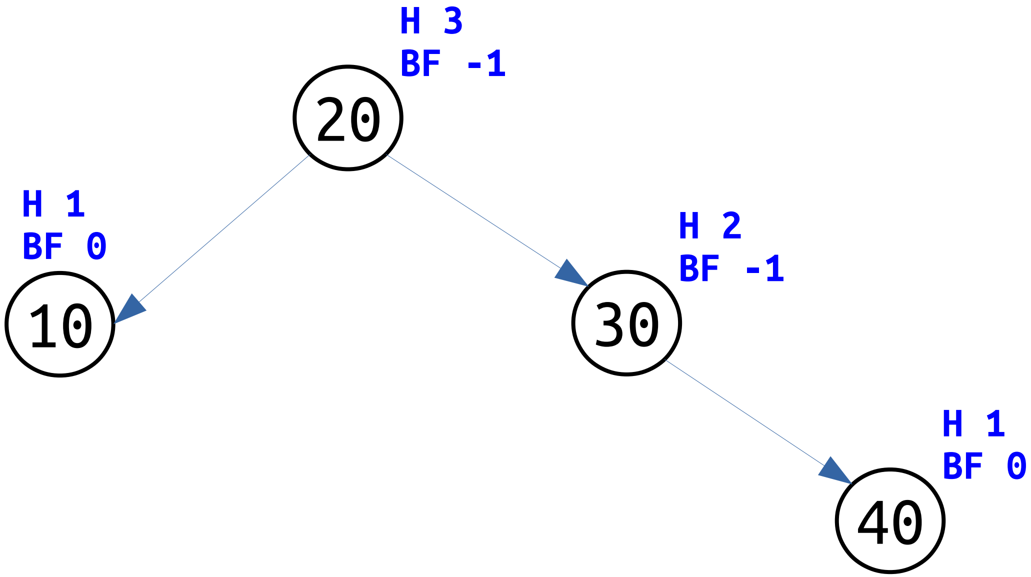
AVL

Insert 40



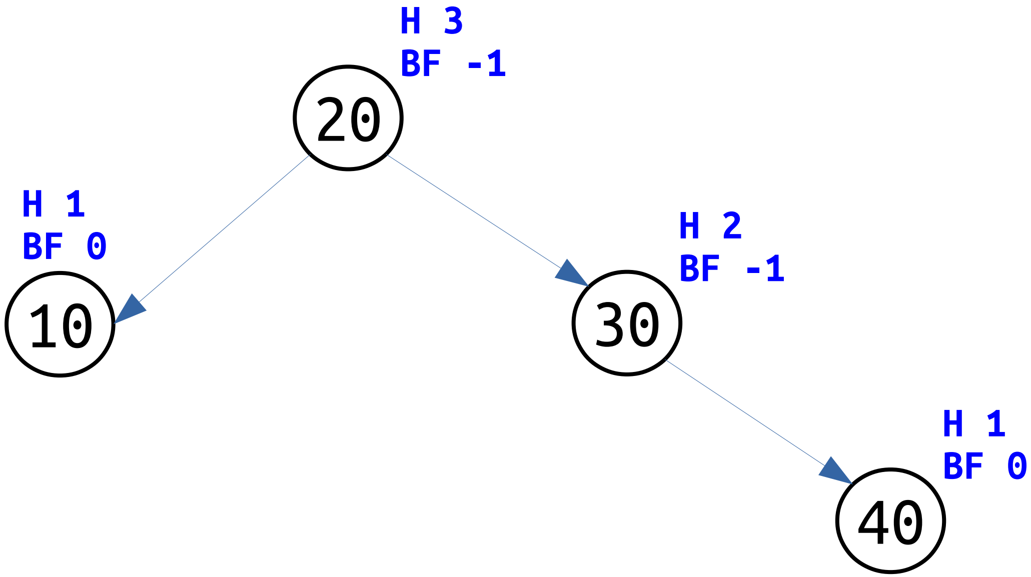
AVL

Insert 40



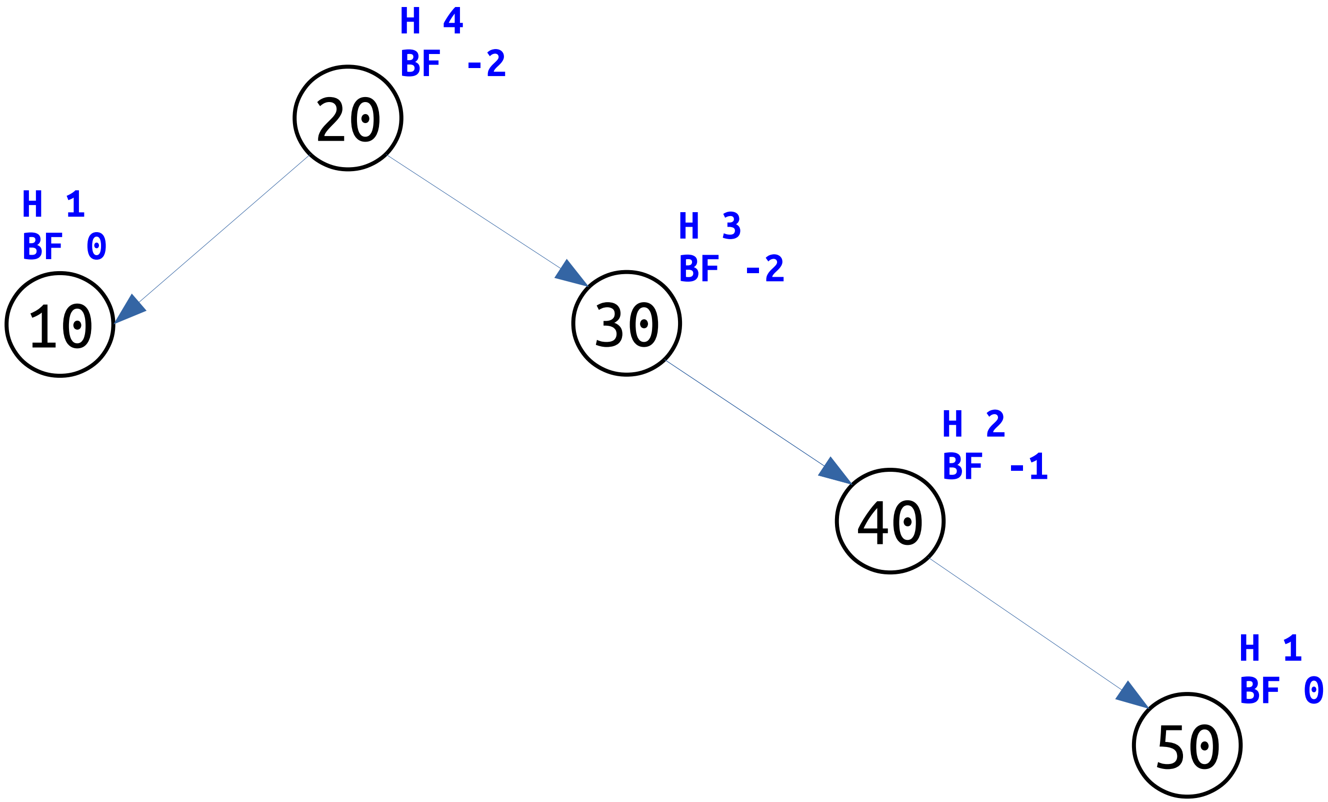
AVL

Insert 50



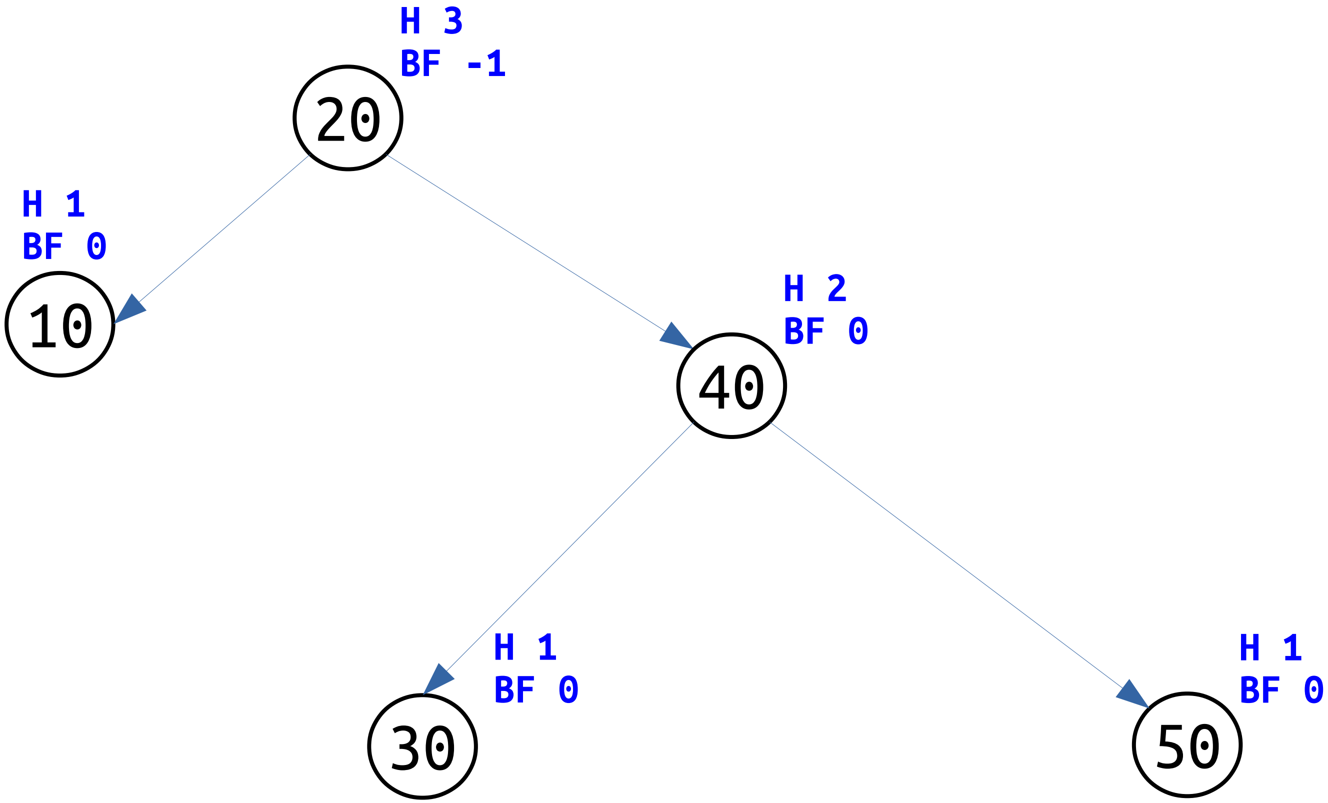
AVL

Insert 50



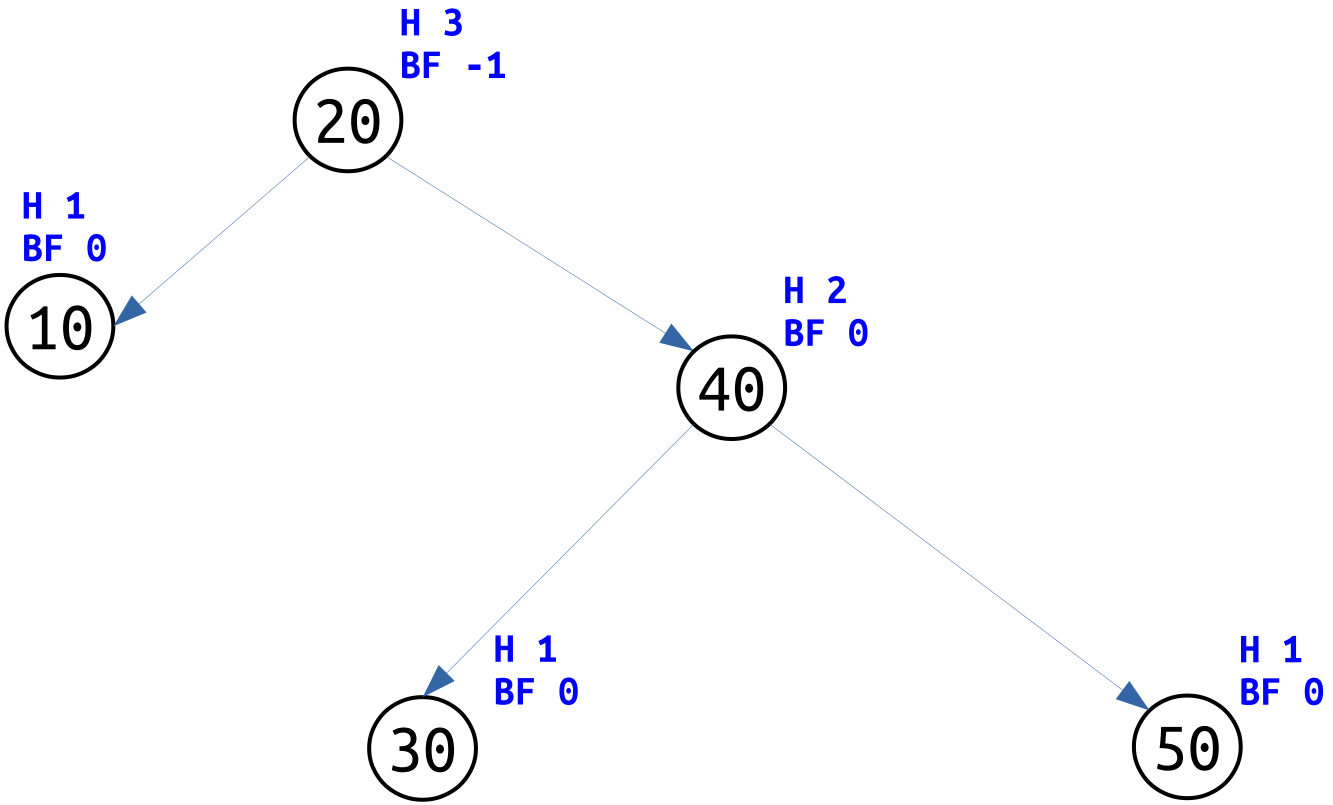
AVL

Insert 50



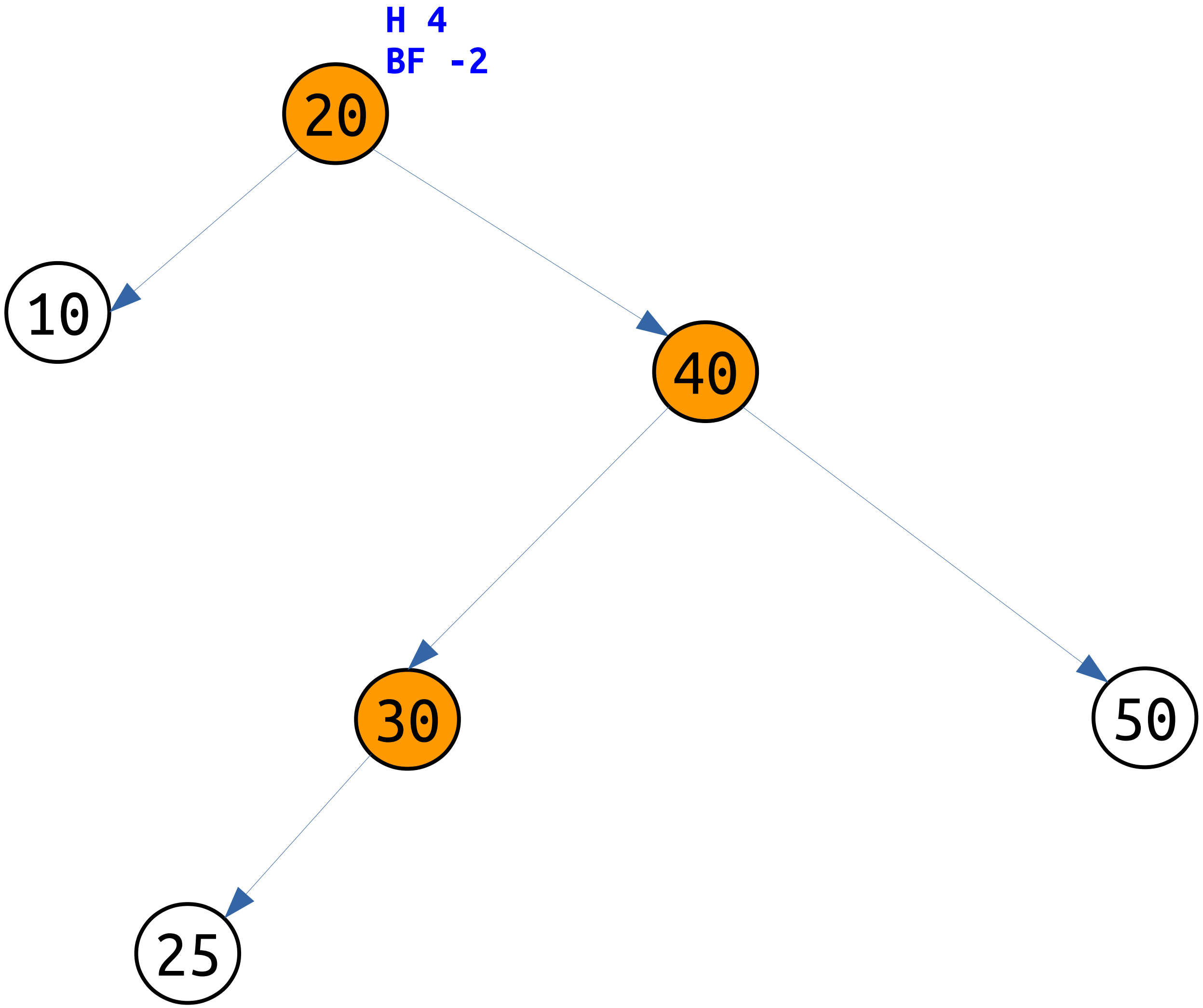
AVL

Insert 25



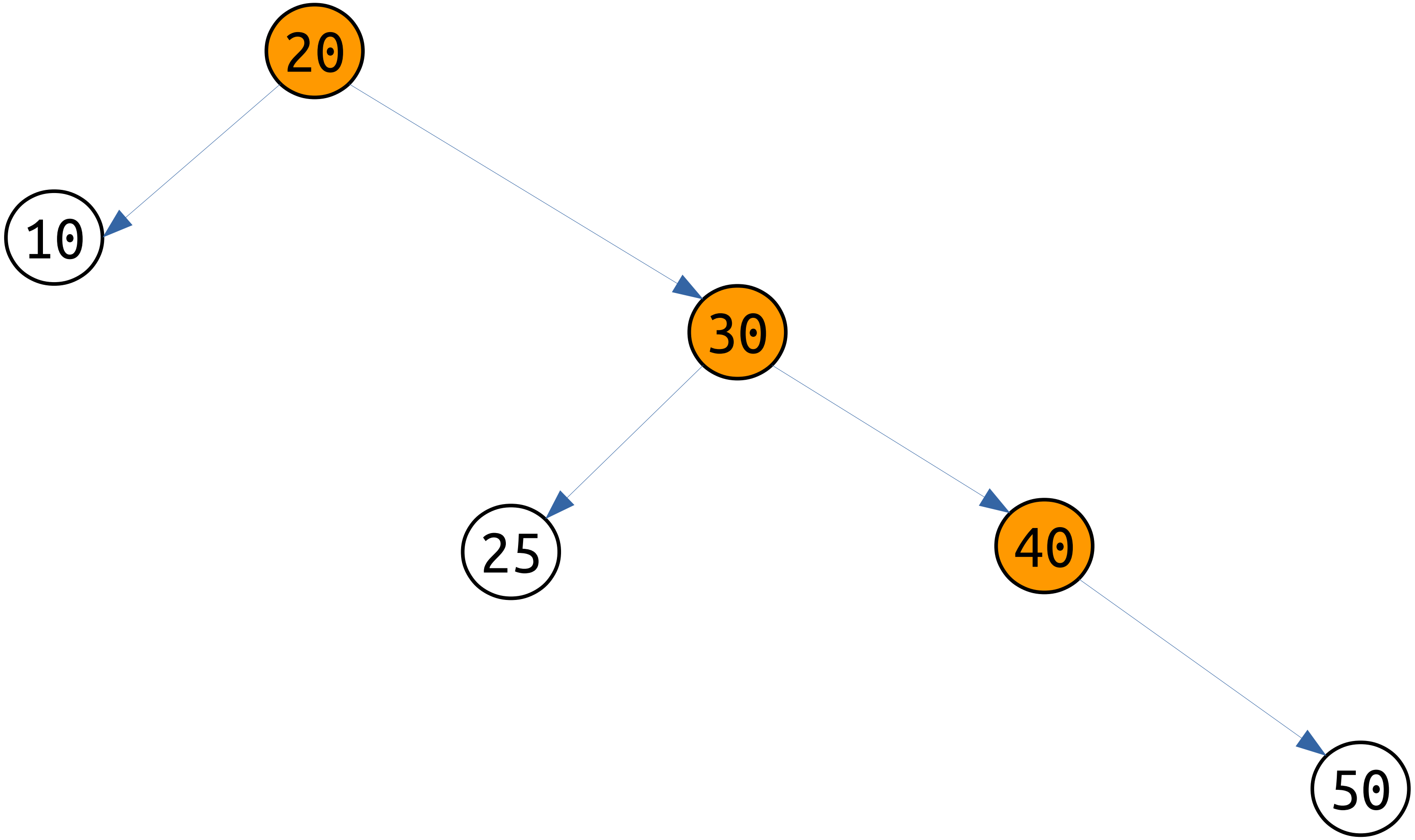
AVL

Insert 25



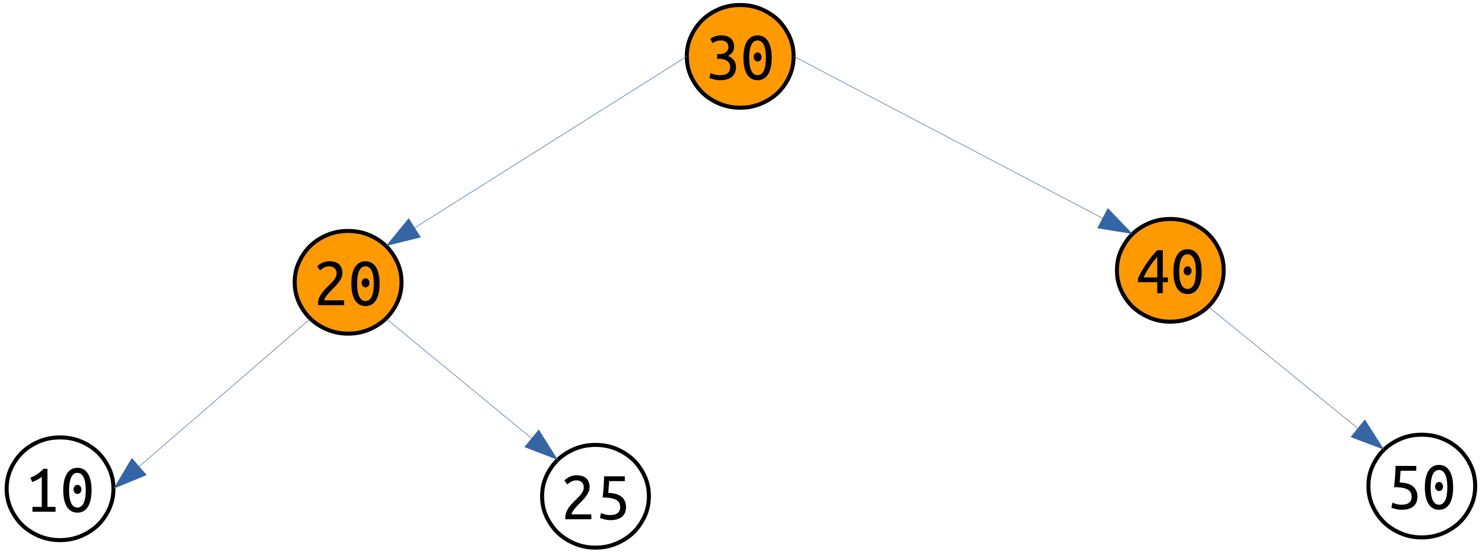
AVL

Insert 25



AVL

Insert 25


30
20
40
10
25
50