Lecture 6 - Tree (Part 2)

CPE112 - Programming with Data Structures

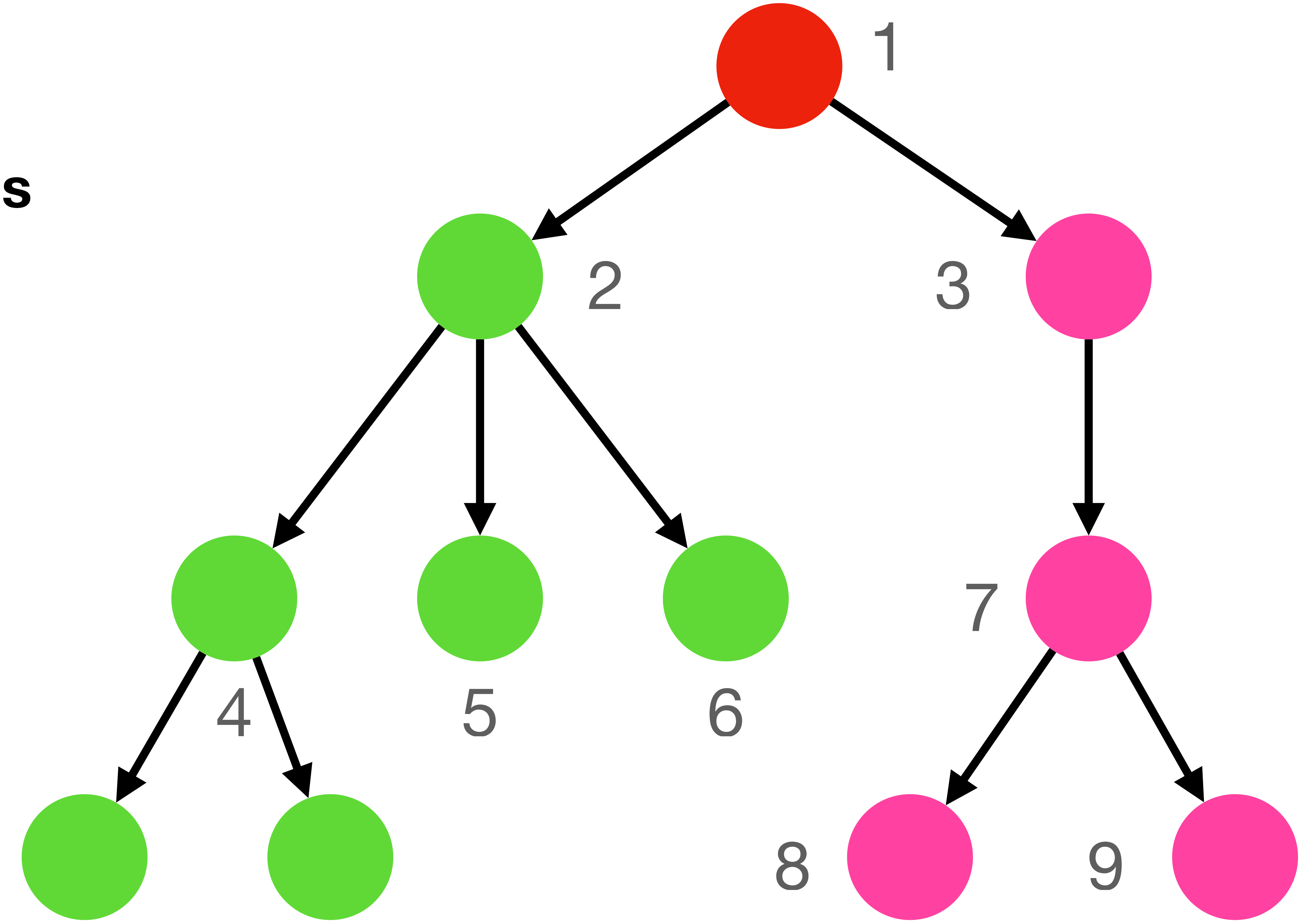
5 March 2025

Dr. Piyanit Wepulanon & Dr. Taweechai Nuntawisuttiwong

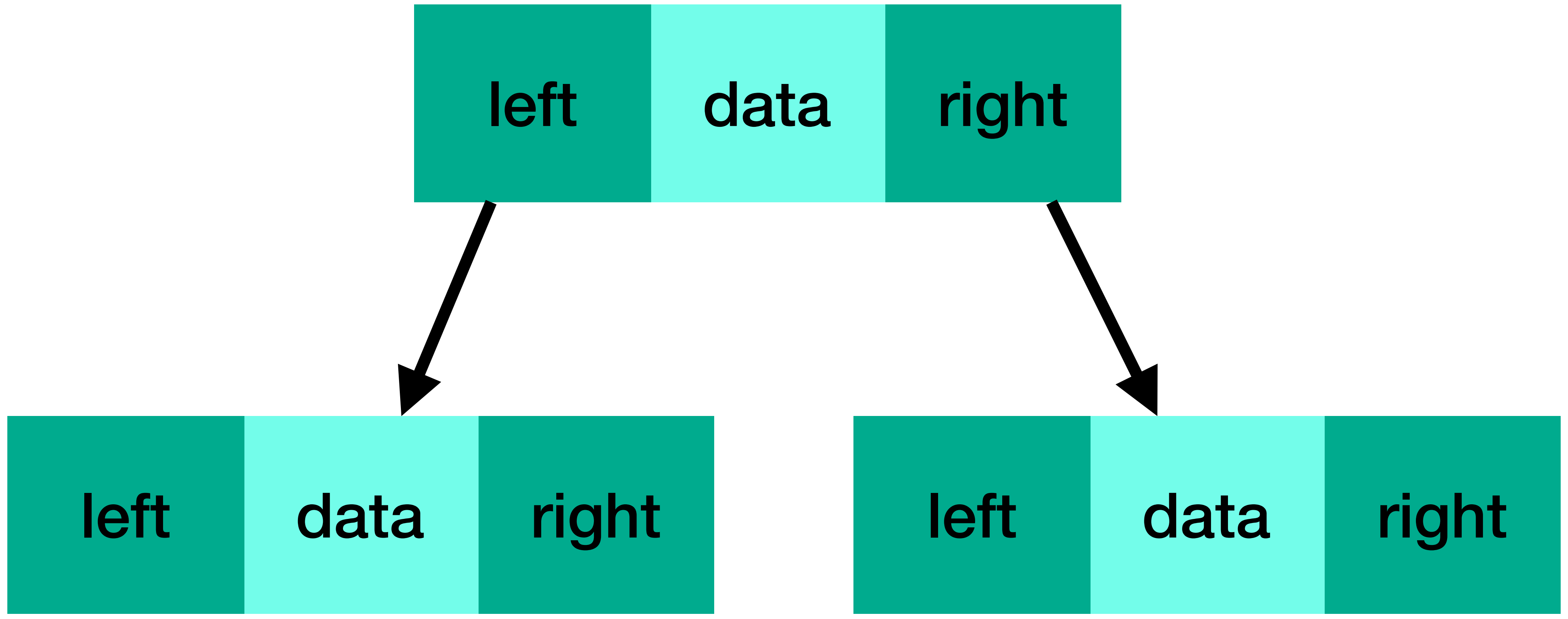
Department of Computer Engineering
KMUTT

Review (1)

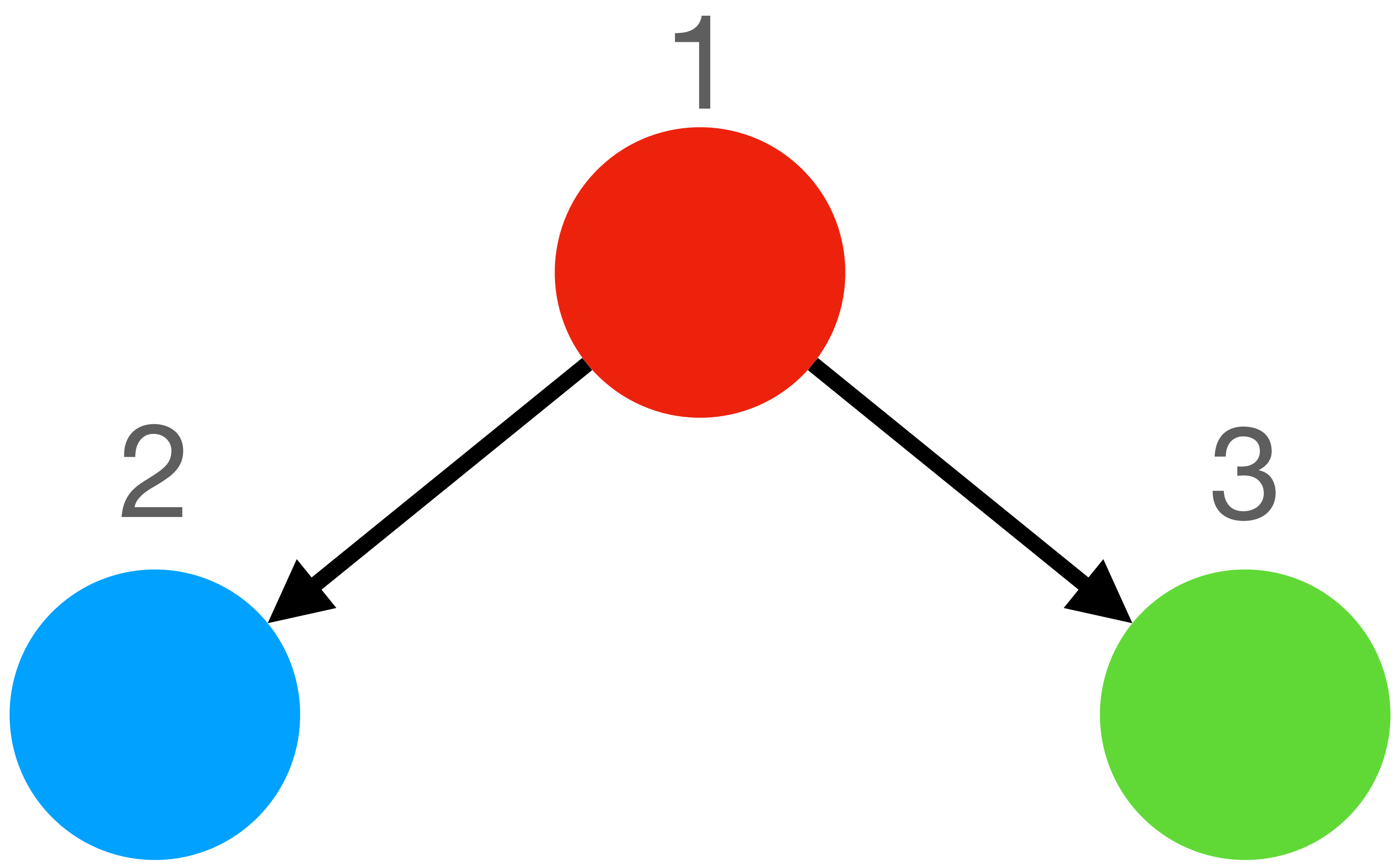
- **Linear vs Hierarchical Data Structures**
- **Tree**
  - Node / Edge / Path
  - Parent / Children / Sibling
  - Ancestor / Descendant
  - Root / Leaf
  - Sub-tree
  - Level / Path Length / Height / Depth



Review (2)



- **Binary Tree**
- **Tree traversal**
  - **Depth-first search:** pre-order, in-order, post-order
  - **Breadth-first search:** Root -> Leaf L -> Leaf R



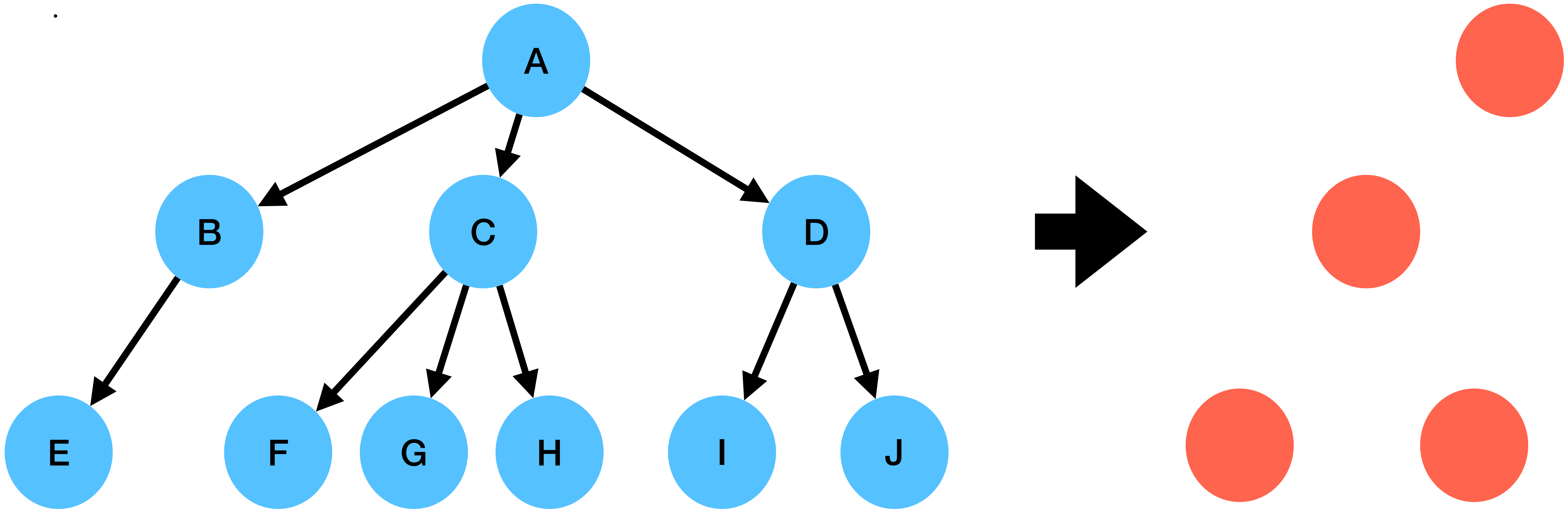
Today’s Goal

- General Tree -> Binary Tree
- Binary Search Tree
- AVL Tree

Creating a Binary Tree from a General Tree

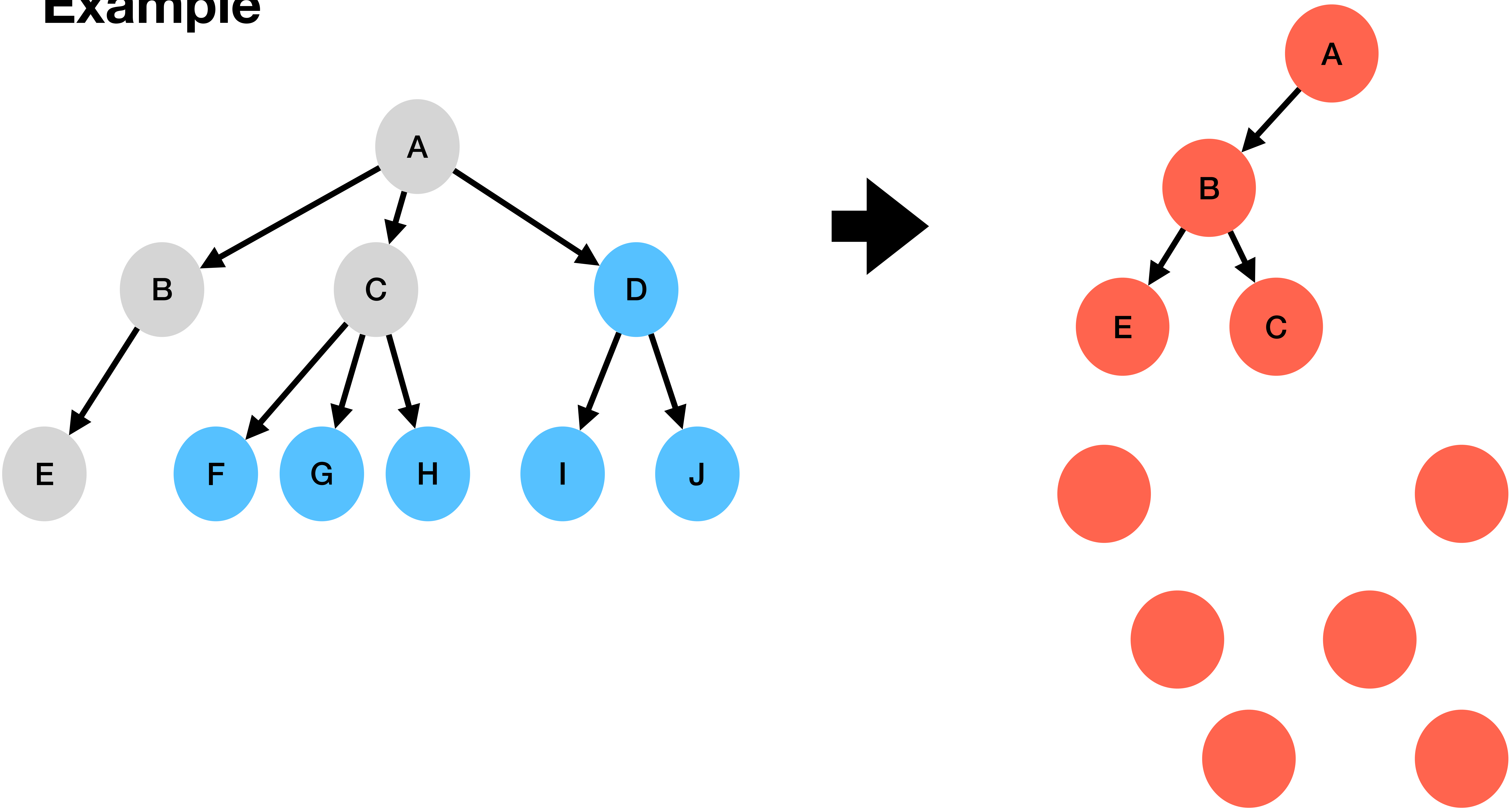
Rules

- **Root** of the BT = **Root** of the GT
- **Left child** of a node in the BT = **Left most child** of the node in the GT
- **Right child** of a node in the BT = **Right sibling** of the node in the GT



Creating a Binary Tree from a General Tree

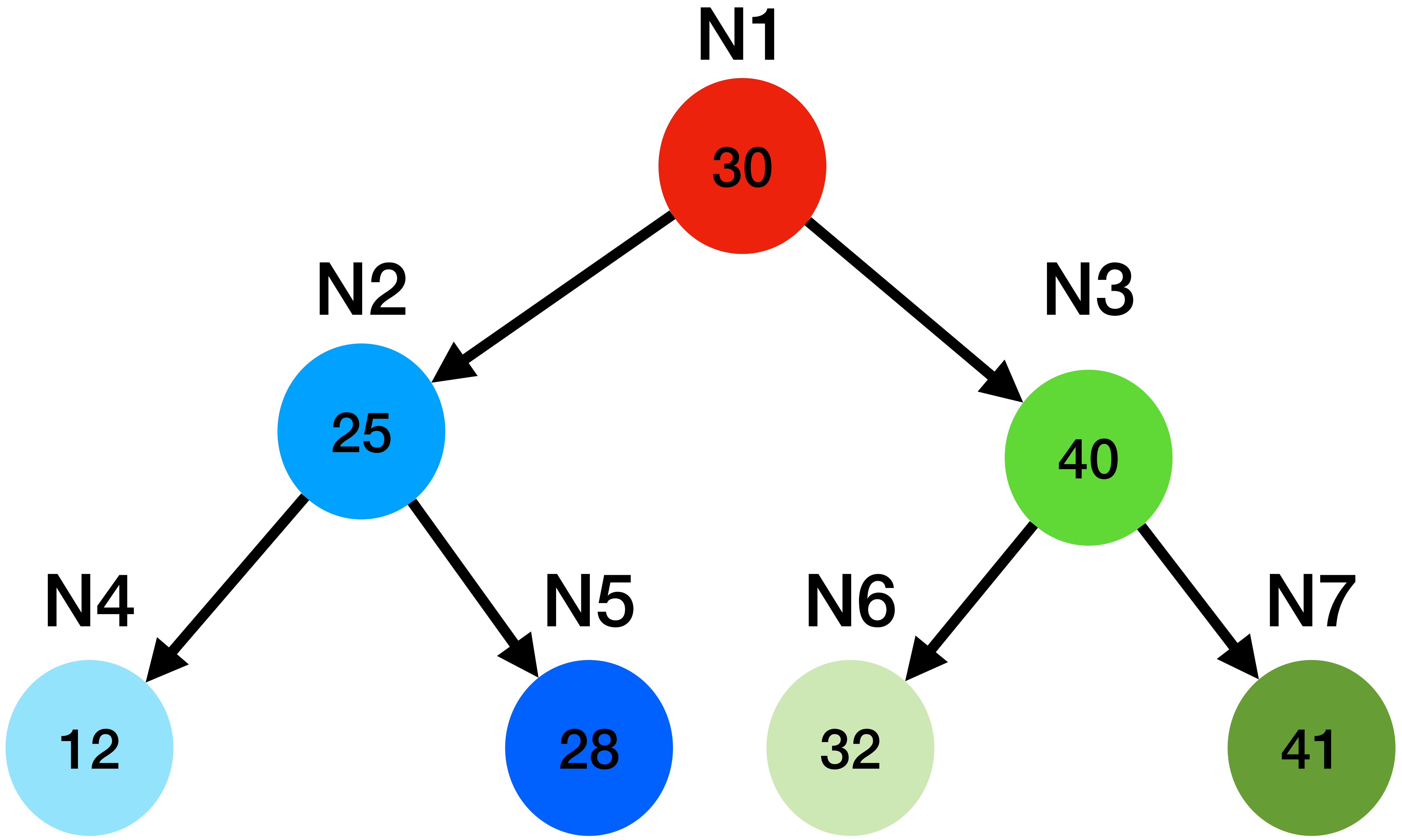
Example



Binary Search Tree

Definition

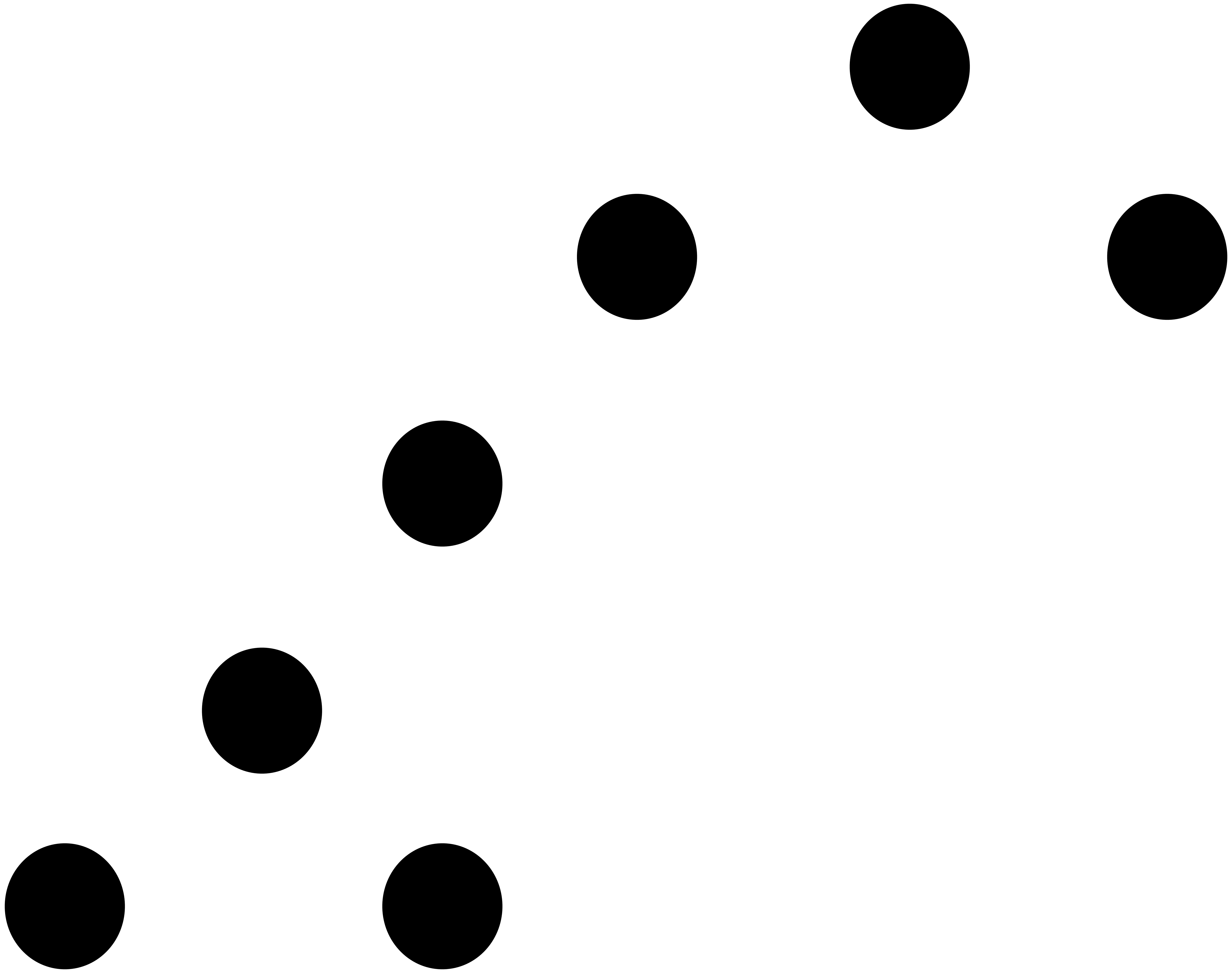
- If T is a binary search tree and N is a node in T, the value of N must be greater than all nodes in the left sub-tree and less than all nodes in the right sub-tree



Binary Search Tree

Create a binary tree

- 40, 32, 41, 30, 25, 12, 28



Binary Search Tree

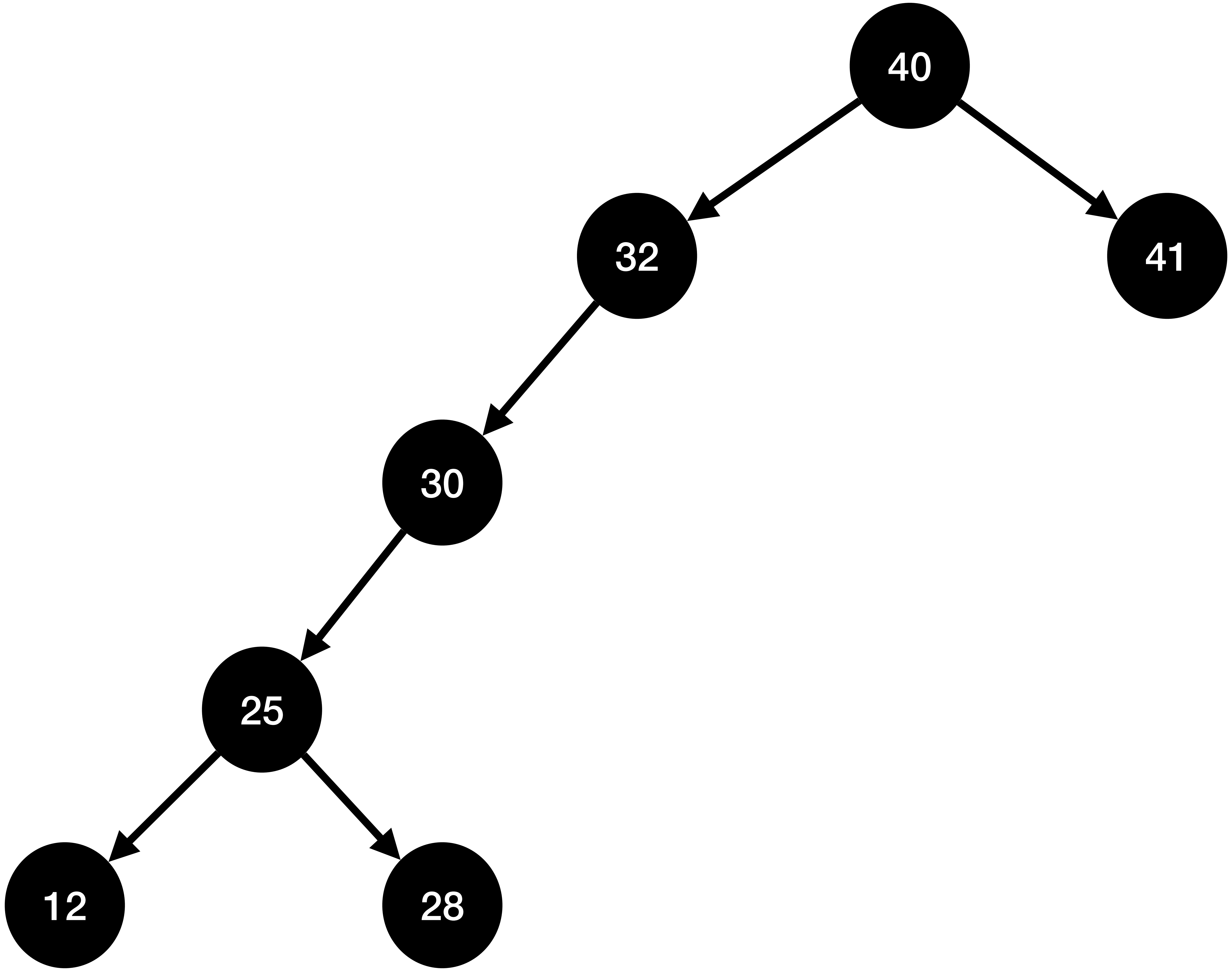
Create a binary tree

```
TREENODE_T* insertNodeToBinaryTree (TREENODE_T* root, TREENODE_T* newNode)
{
   TREENODE_T *p, *previous;
   //CASE 1: Root is null THEN root = newNode

   //CASE 2: Root is not null THEN
   //      (a) find a suitable node for insertion (hint: use a while loop then compare values)
   //      (b) link newNode with the suitable node

   //RETURN Root
}
```

Binary Search Tree

Search a node

- Find 28



Binary Search Tree

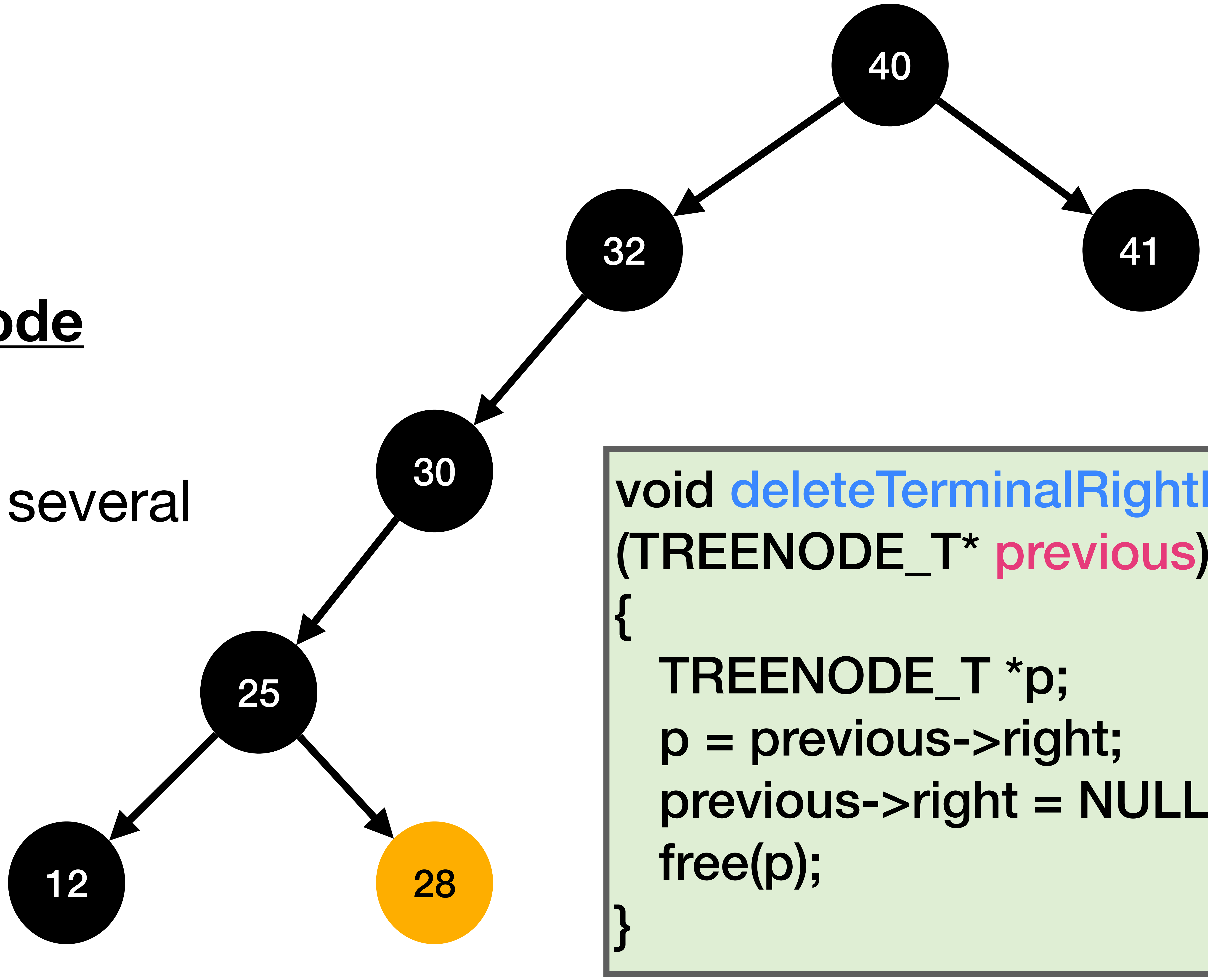
Search a node

```
TREENODE_T* binarySearch (TREENODE_T* root, int key)
{
   TREENODE_T *p, *node;
   p = root; node = NULL;
   do {
         if (p->info == key) node = p;      /*Search found*/
         else if (p->info > key) p = p->left;
         else p = p->right;
   } while ((p != NULL) && (node == NULL));
   return (node);
}
```

Binary Search Tree

Delete a node

- Delete 28
- Need to know the **parent node**
- Delete a node in any tree is complicated since there are several **cases**

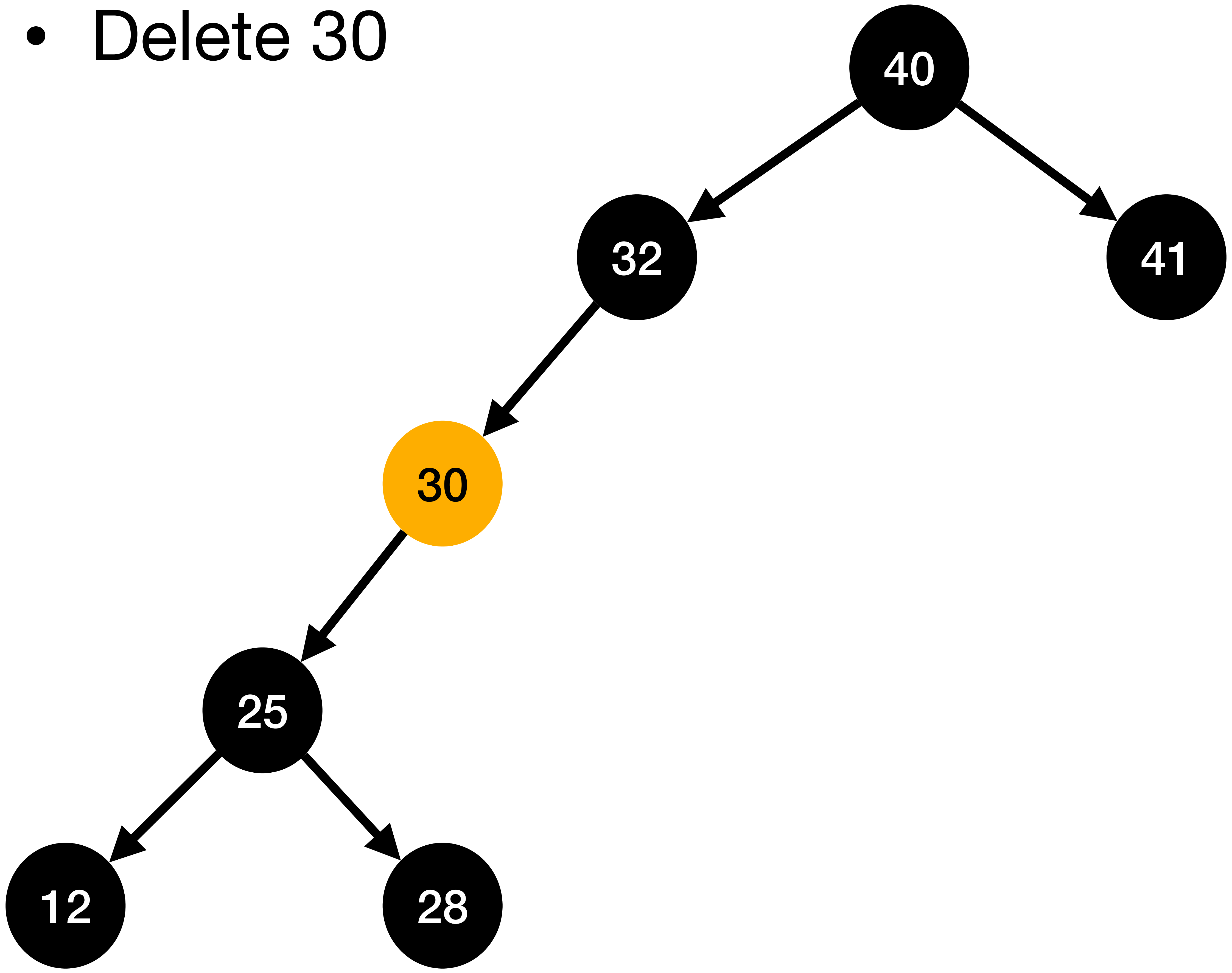


```
void deleteTerminalRightNode
(TREENODE_T* previous)
{
   TREENODE_T *p;
   p = previous->right;
   previous->right = NULL;
   free(p);
}
```

Binary Search Tree

Delete a node

- Delete 30

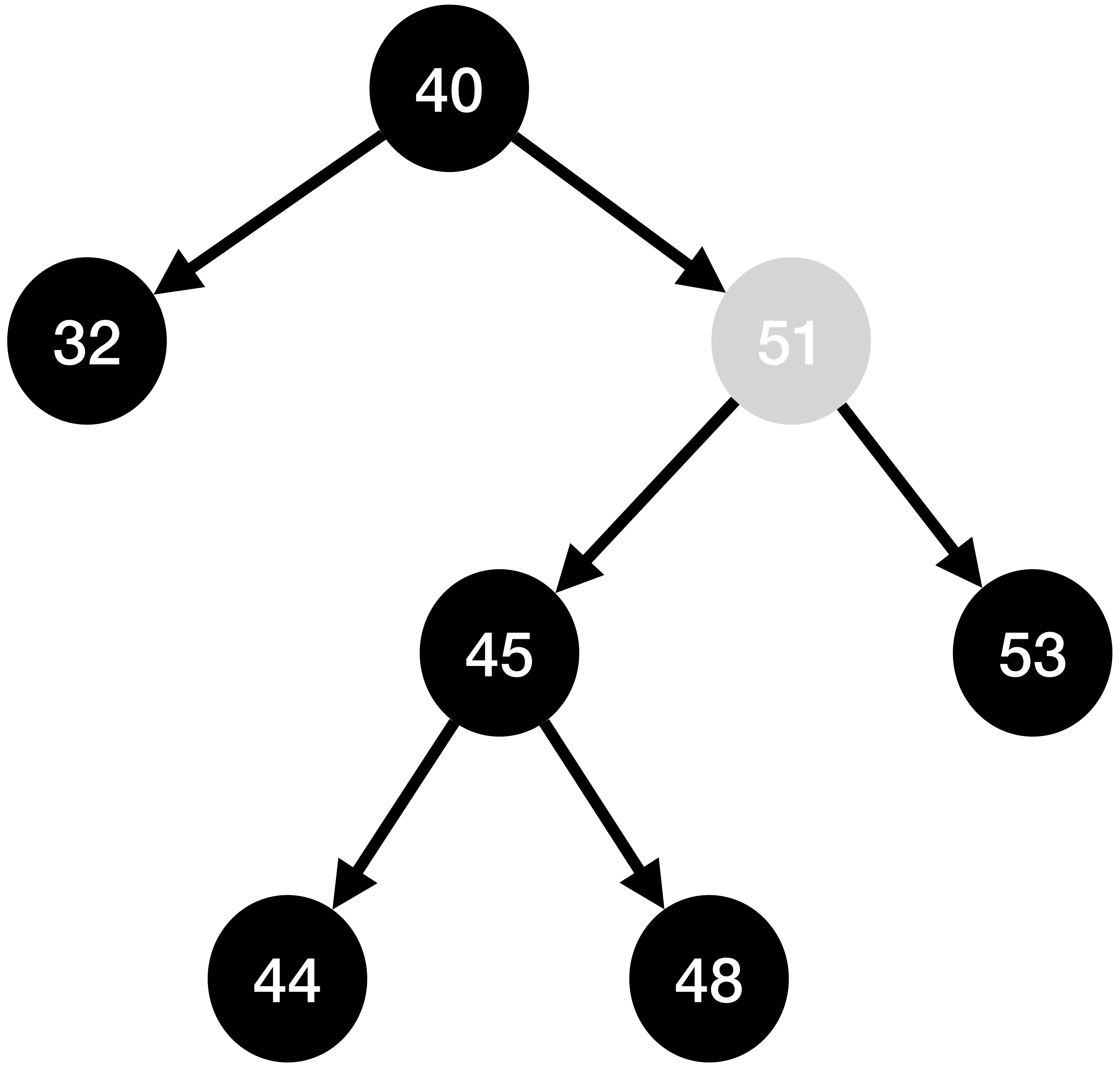


```
void deleteNonTerminalWithOnlyLeftNode
(TREENODE_T* previous)
{
   TREENODE_T *p;
   p = previous->left;
   previous->left = p->left;
   free(p);
}
```

Binary Search Tree

Delete a node

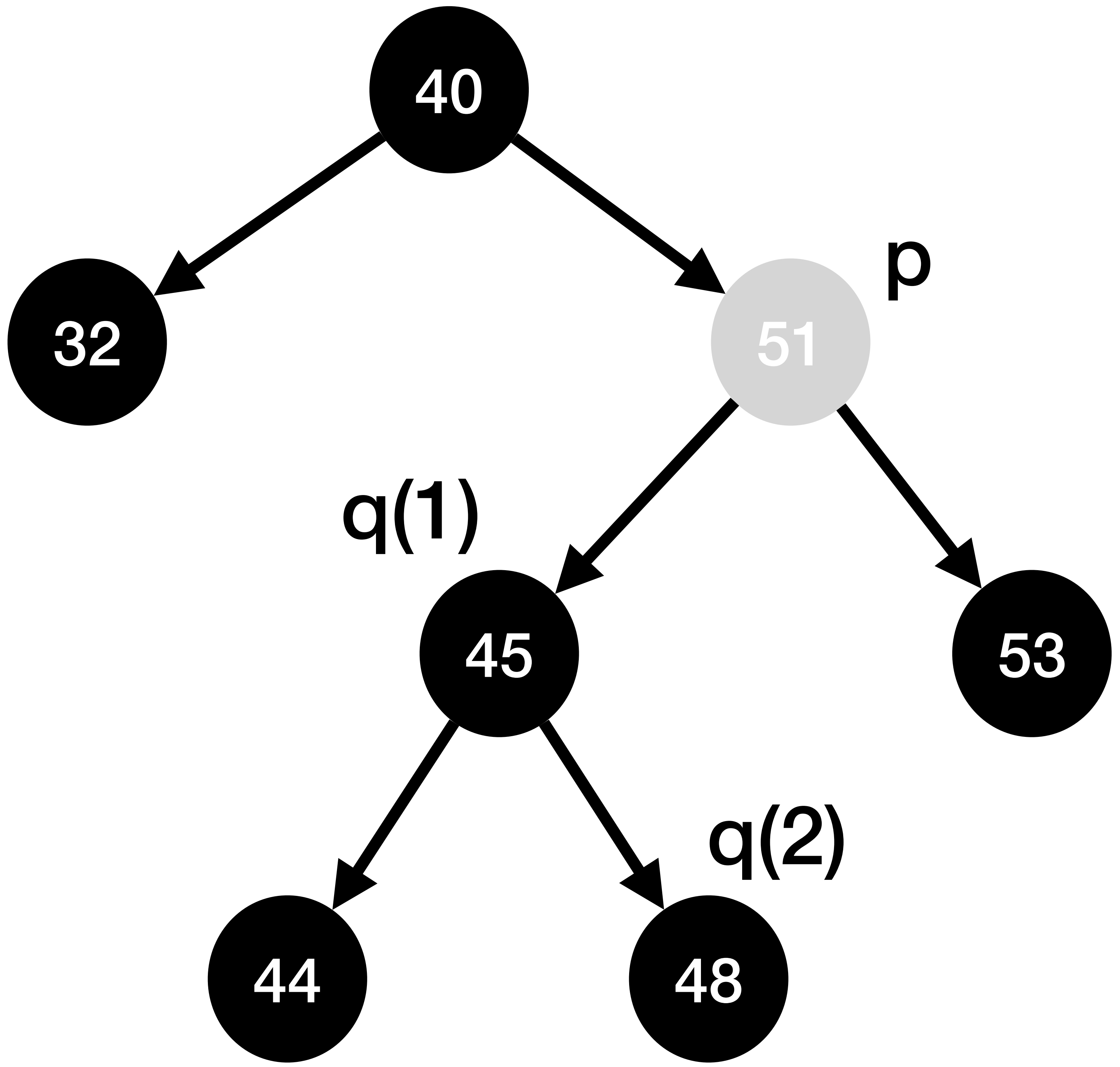
- More case: delete 51

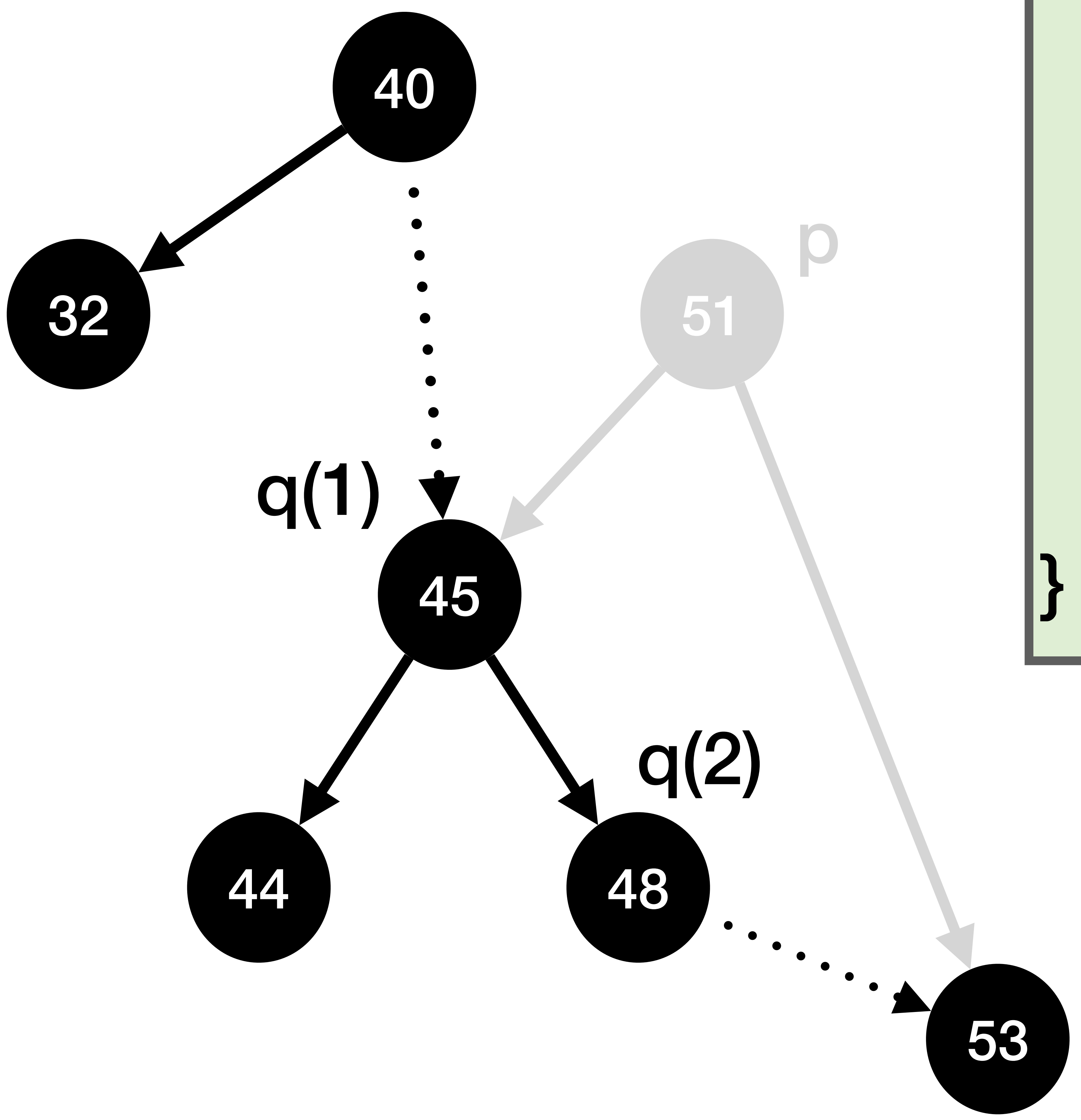


Binary Search Tree

Delete a node

- More case: delete 51





```
void deleteNonTerminalRightNode
(TREENODE_T* previous)
{
   TREENODE_T *p, *q;
   p = previous->right;
   previous->right = q = p->left;
   while (q->right != NULL)
      q = q->right;
   q->right = p->right;
   free(p);
}
```

Binary Search Tree

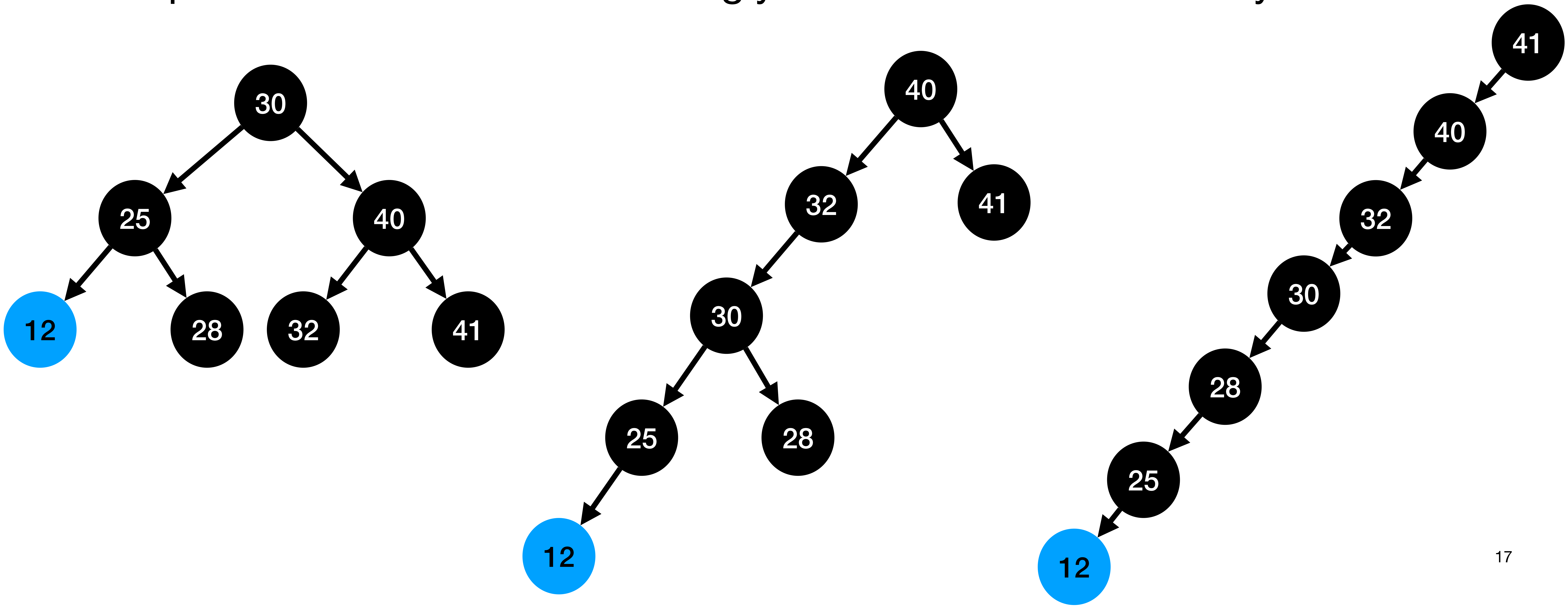
Other operations

- Find the height of a Binary Search Tree
- Find the height of a node
- Find the smallest/largest node
- Delete a Binary Search Tree (delete left sub-tree then right sub-tree)

Binary Search Tree

Balance Trees

- The sequence of information is strongly affected search efficiency.

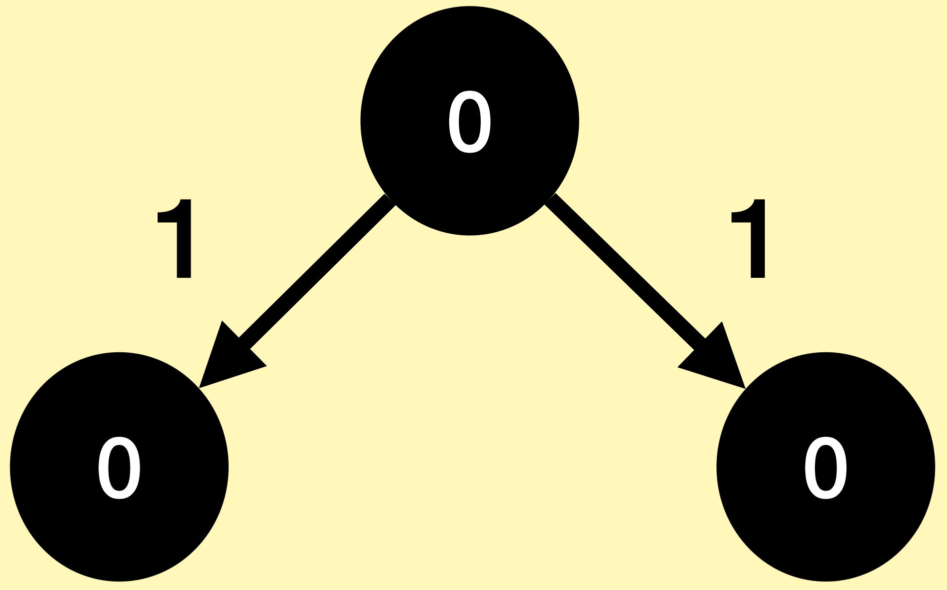


Binary Search Tree

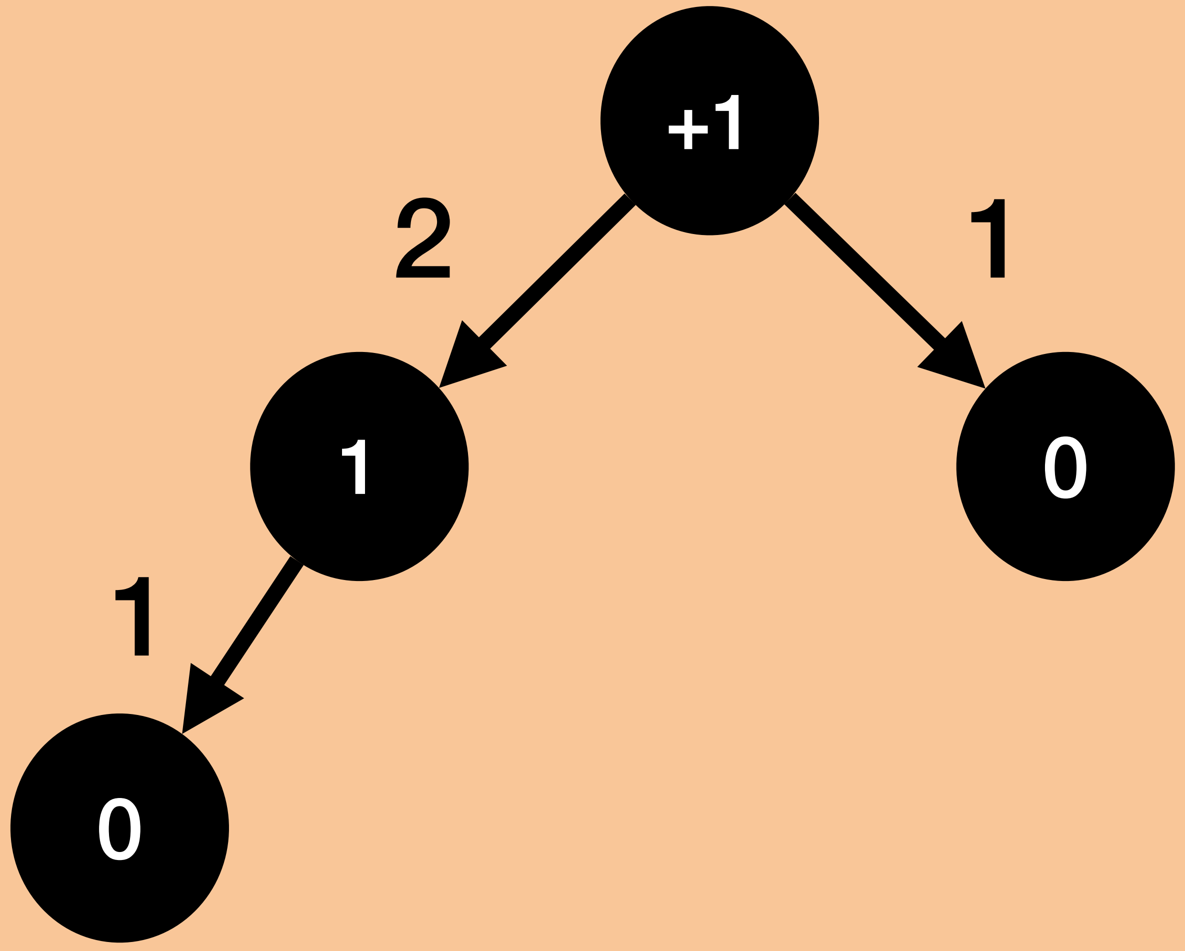
Balance Factor

- Balance factor = Height (left sub-tree) - Height (right sub-tree)
- Note: height is the longest path length to the leaf

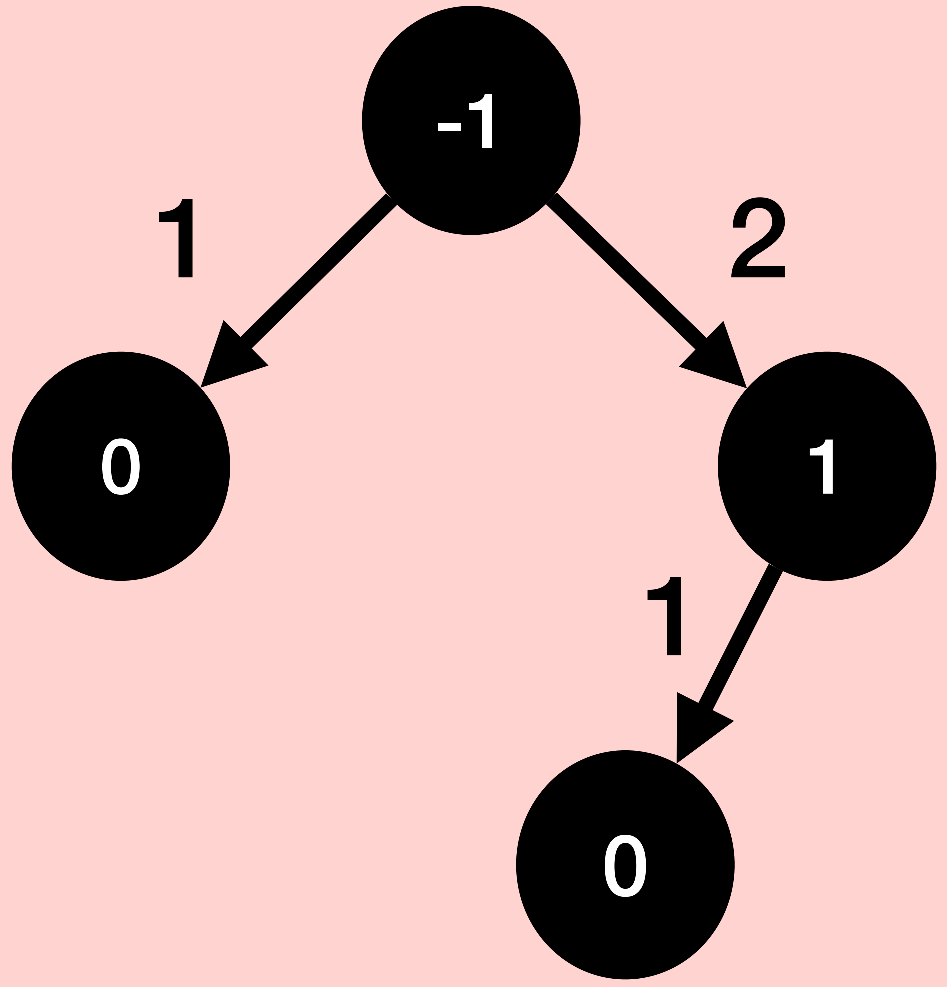
balance factor = 0



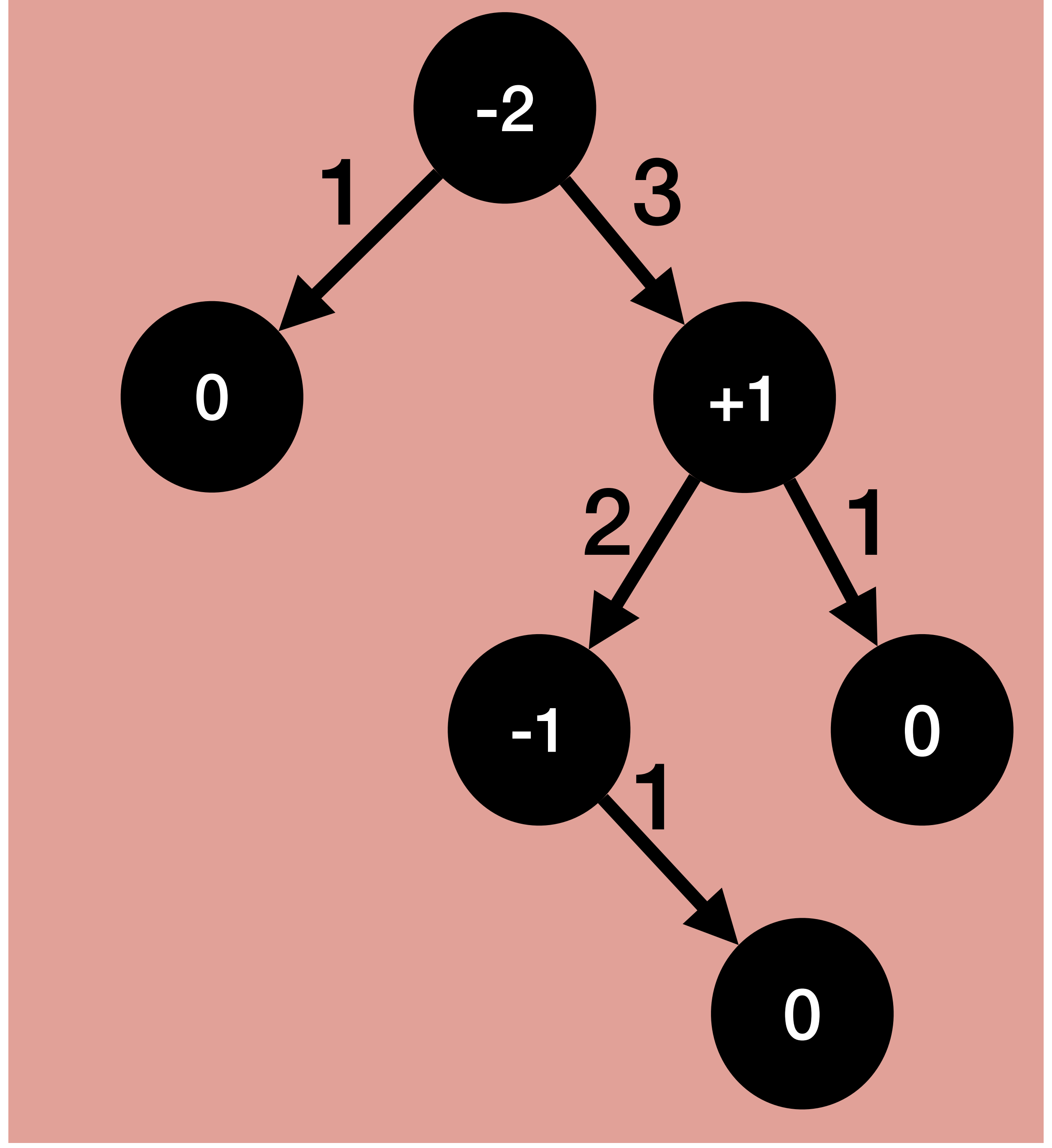
balance factor = 1



balance factor = -1



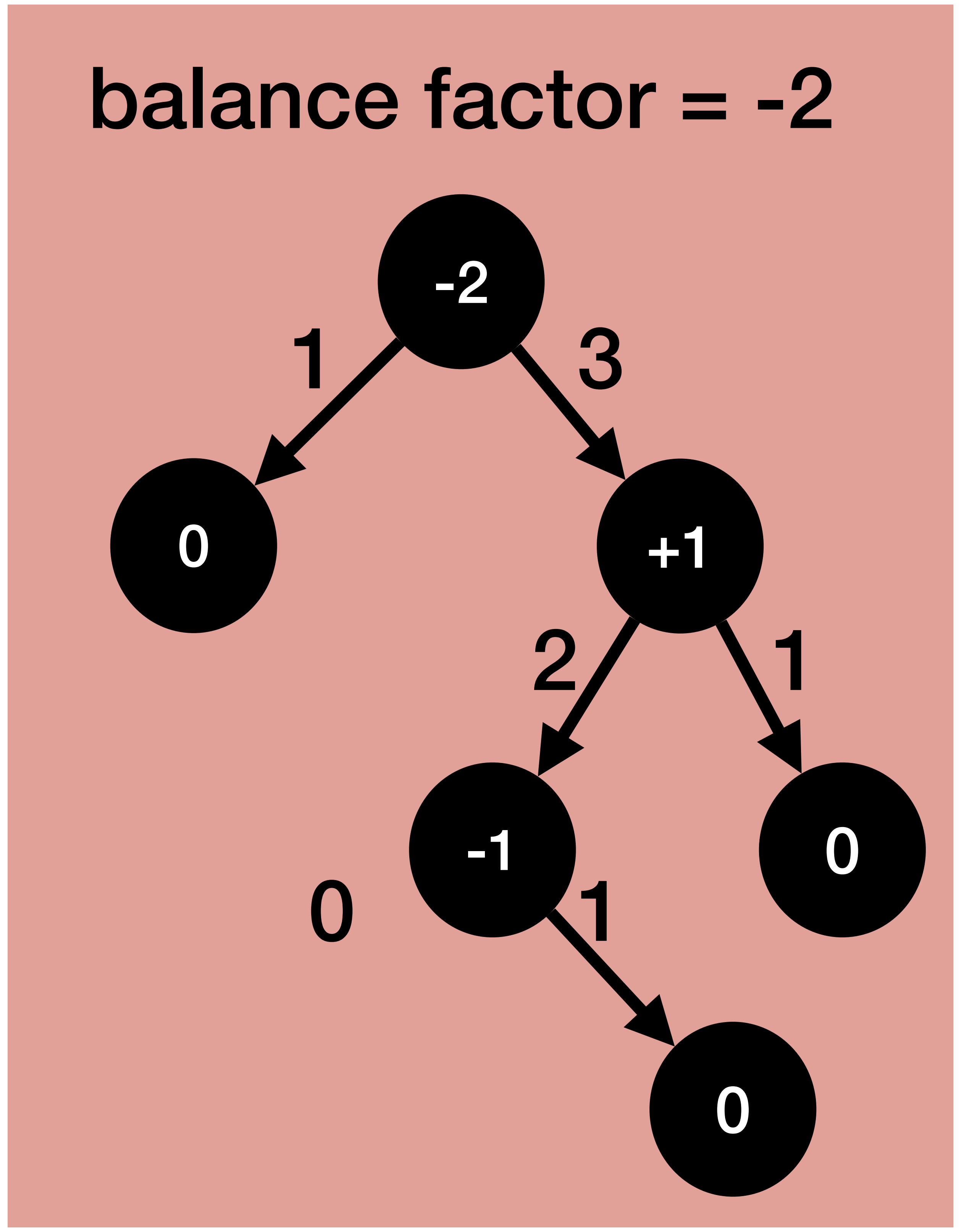
balance factor = -2



AVL Tree

Balanced Binary Search Tree

- A binary search tree is balance if the balance factor of any node is less than **| 1 | (balance factor = -1, 0, or 1)**



balance factor = -2

Q: How to find imbalanced points?
A: Need to know the height and balance of each node

```
typedef struct _treenode
{
   int data                              ;
   struct _treenode* left;
   struct _treenode* right;
} TREENODE_T;
```

AVL Tree

Is a tree balanced?

- Concept: post-order traversal: LT -> RT -> Root

```
void fillHeight (TREENODE_T *node)
{
   int height = 0;
   if (node->left != NULL)  fillHeight(node->left);
   if (node->right != NULL)   fillHeight(node->right);

   if (node->left == NULL && node->right == NULL)   height = 0;   //Case1
   else if (node->left == NULL)   height = node->right->height +1;  //Case2
   else if (node->right == NULL)   height = node->left->height +1;  //Case3
   else if (node->right->height > node->left->height)   height = node->right->height +1; //Case4
   else   height = node->left->height +1;  //Case5

   node->height = height;
}
```

AVL Tree

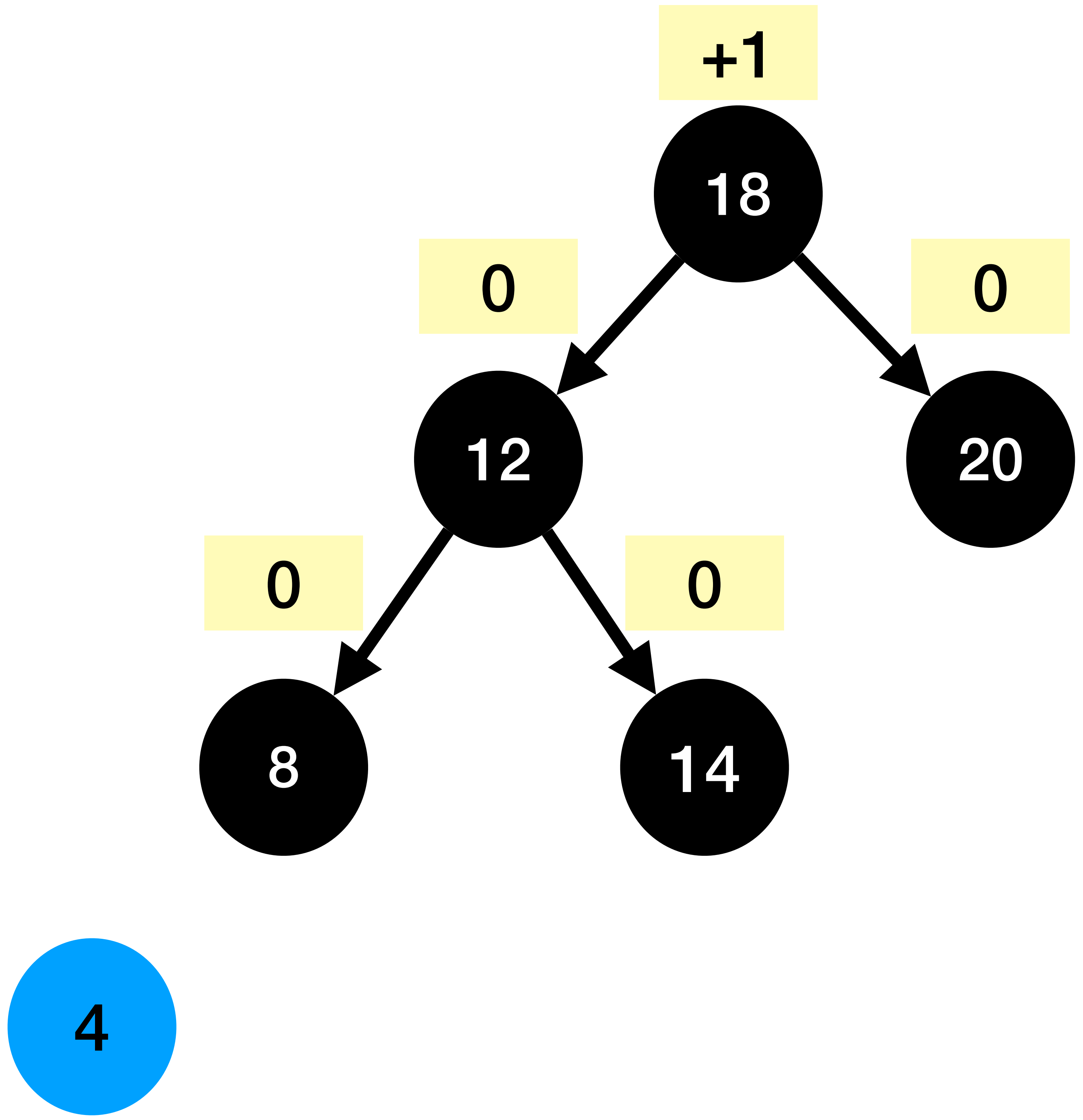
Is a tree balanced?

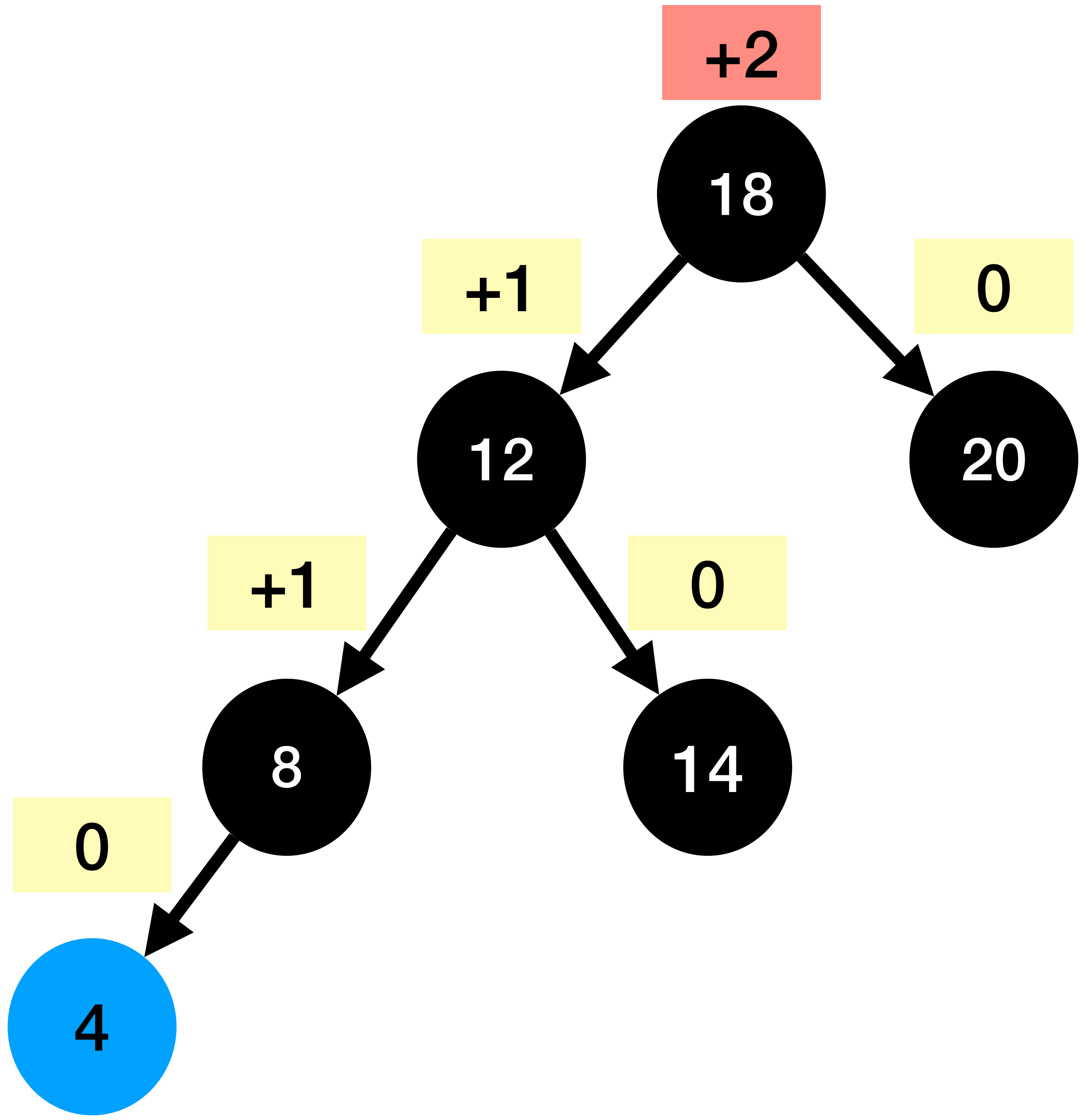
```
void fillBalanceFactor (TREENODE_T *node)
{
   int leftHeight = 0, rightHeight = 0;
   if (node->left != NULL)  fillBalanceFactor(node->left);
   if (node->right != NULL)   fillBalanceFactor(node->right);
   //Get the height of the left sub-tree
   if (node->left == NULL)   …
   else …
   //Get the height of the right sub-tree
   if (node->right == NULL)   …
   else …

   node->balance = …;
}
```

AVL Tree

Lost Balance

- Insert or delete a node

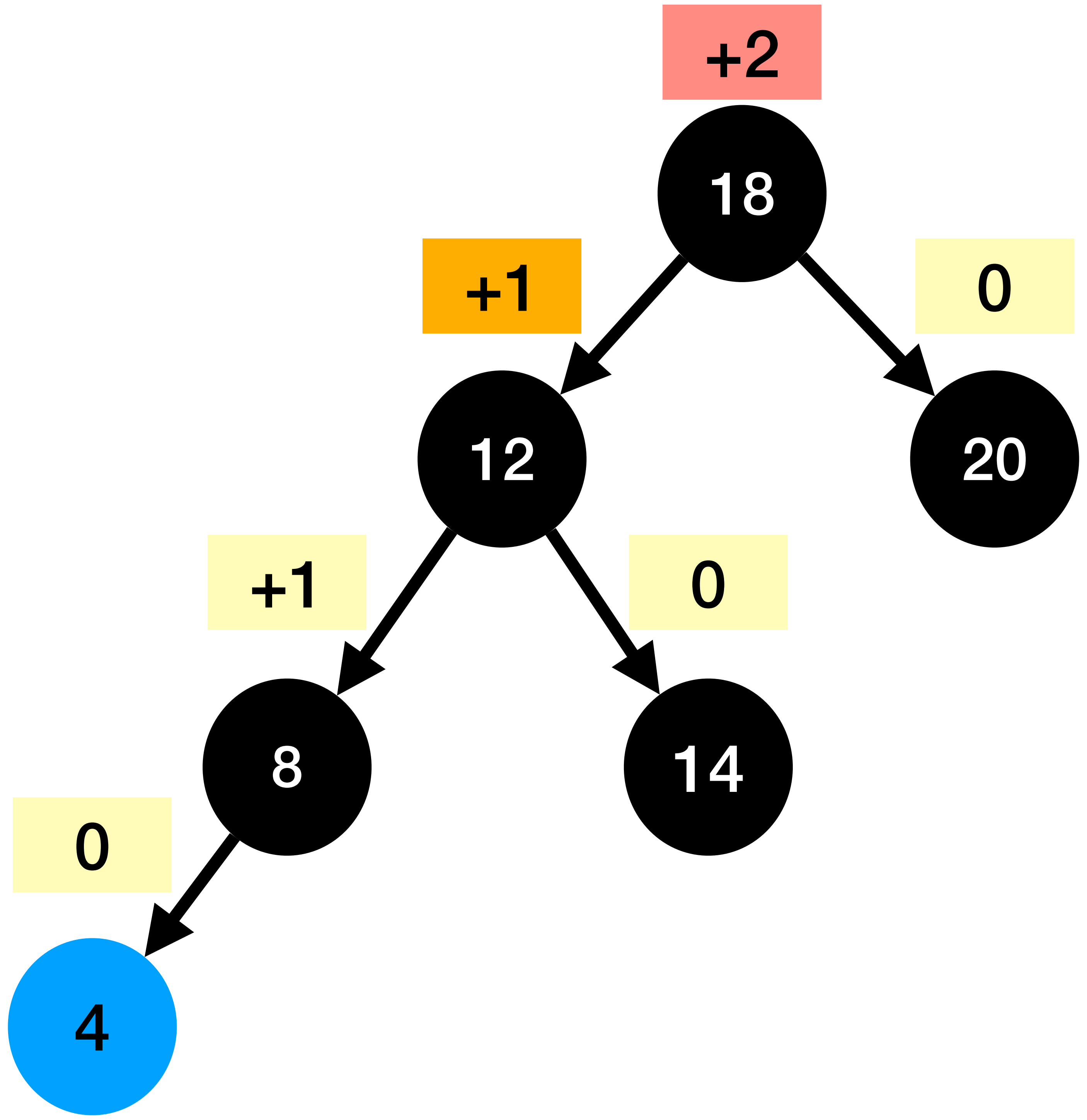




AVL Tree

Rebalancing Rotation

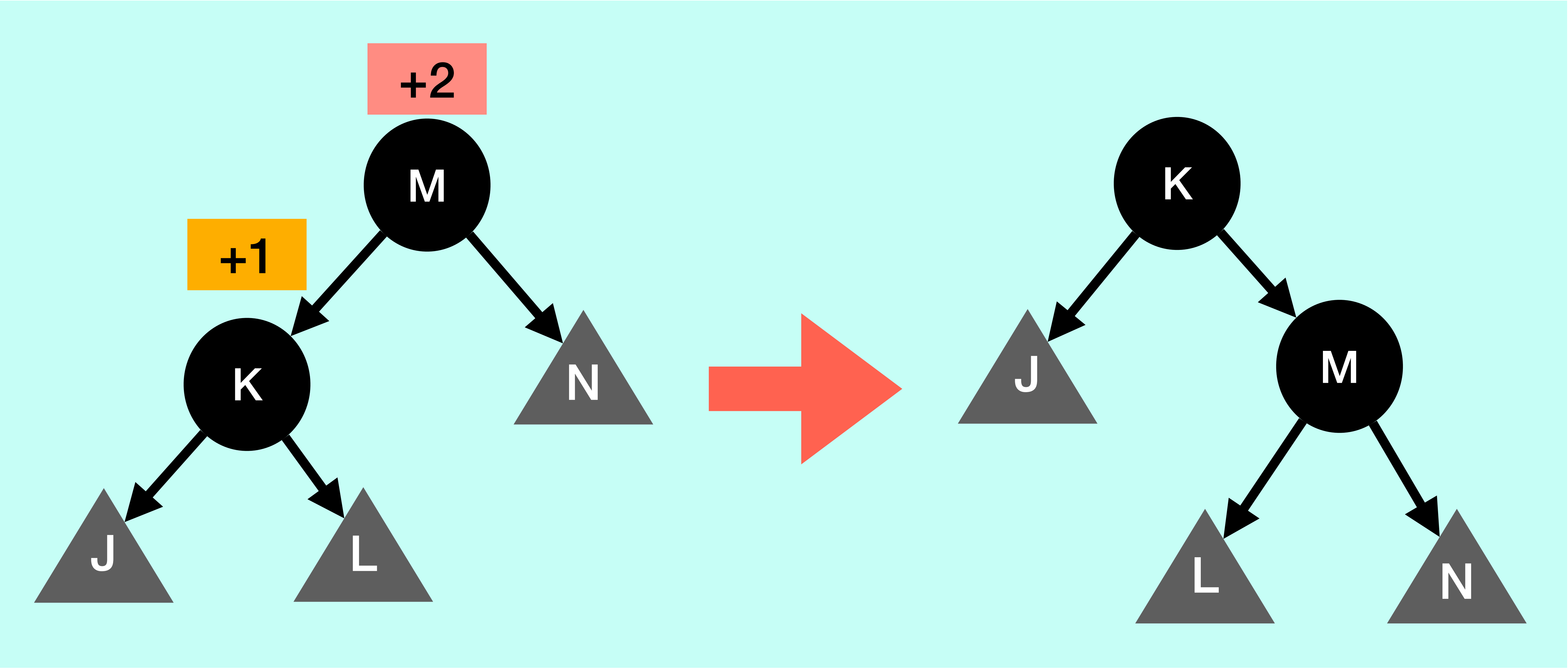
1. ***Left of Left***: The new node is inserted in the left sub-tree of the left sub-tree of the critical node



AVL Tree

Rebalancing Rotation

1. LL Rotation (Rotate Right)

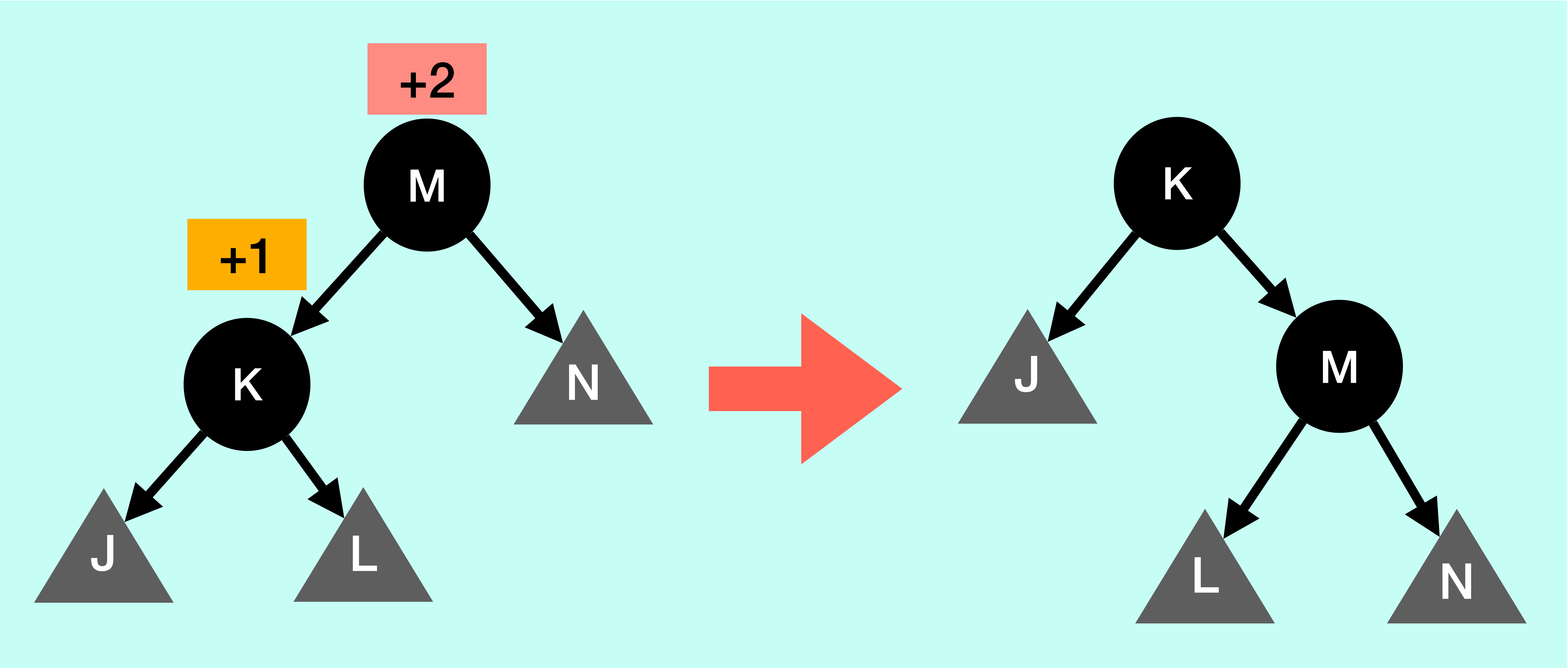


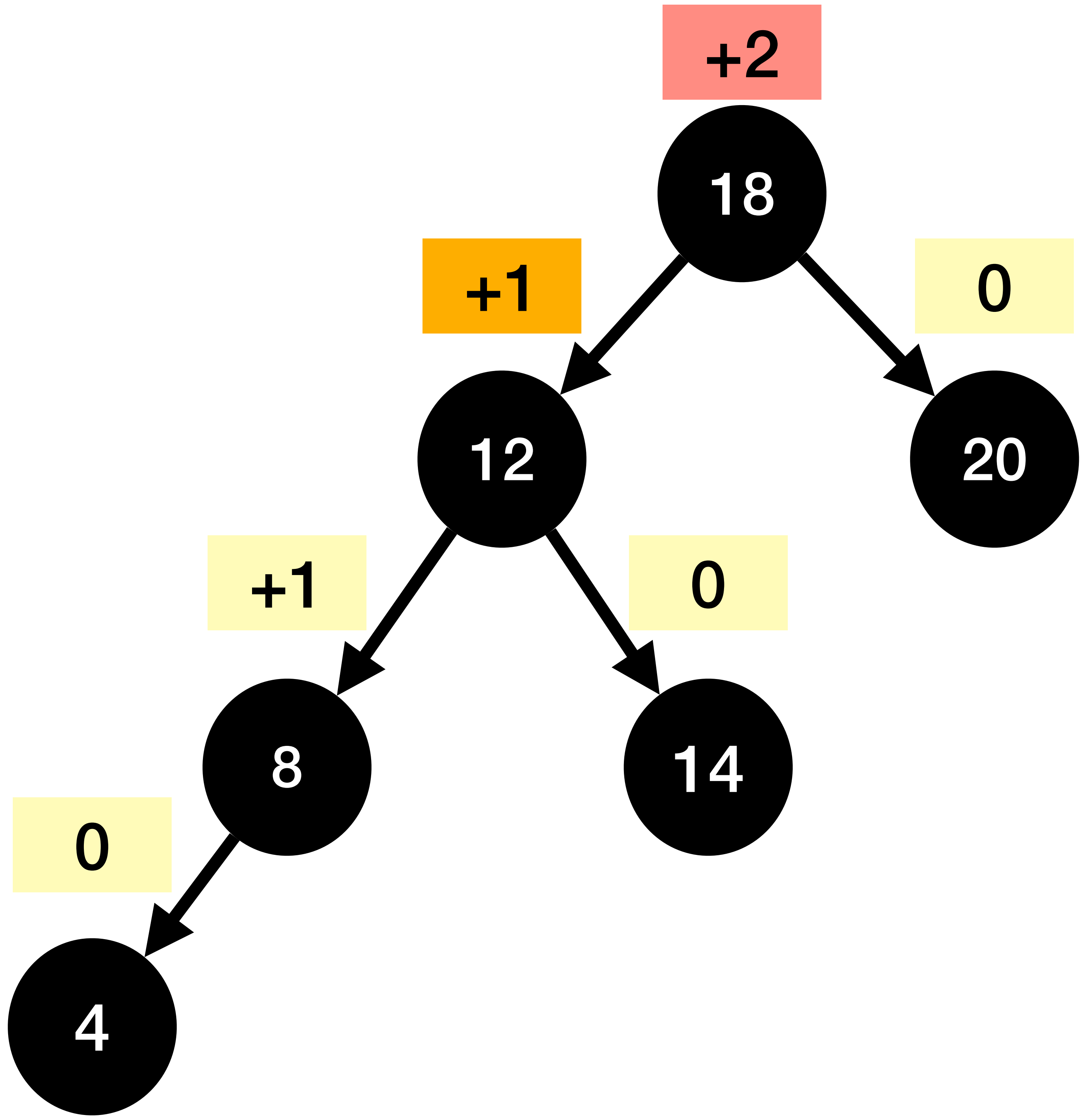
```
TREENODE_T *rotateRight (TREENODE_T *node)
{
   TREENODE_T *temp;
   temp = node->left;
   node->left = temp->right;
   temp->right = node;
   node = temp;
   return(node);
}
```

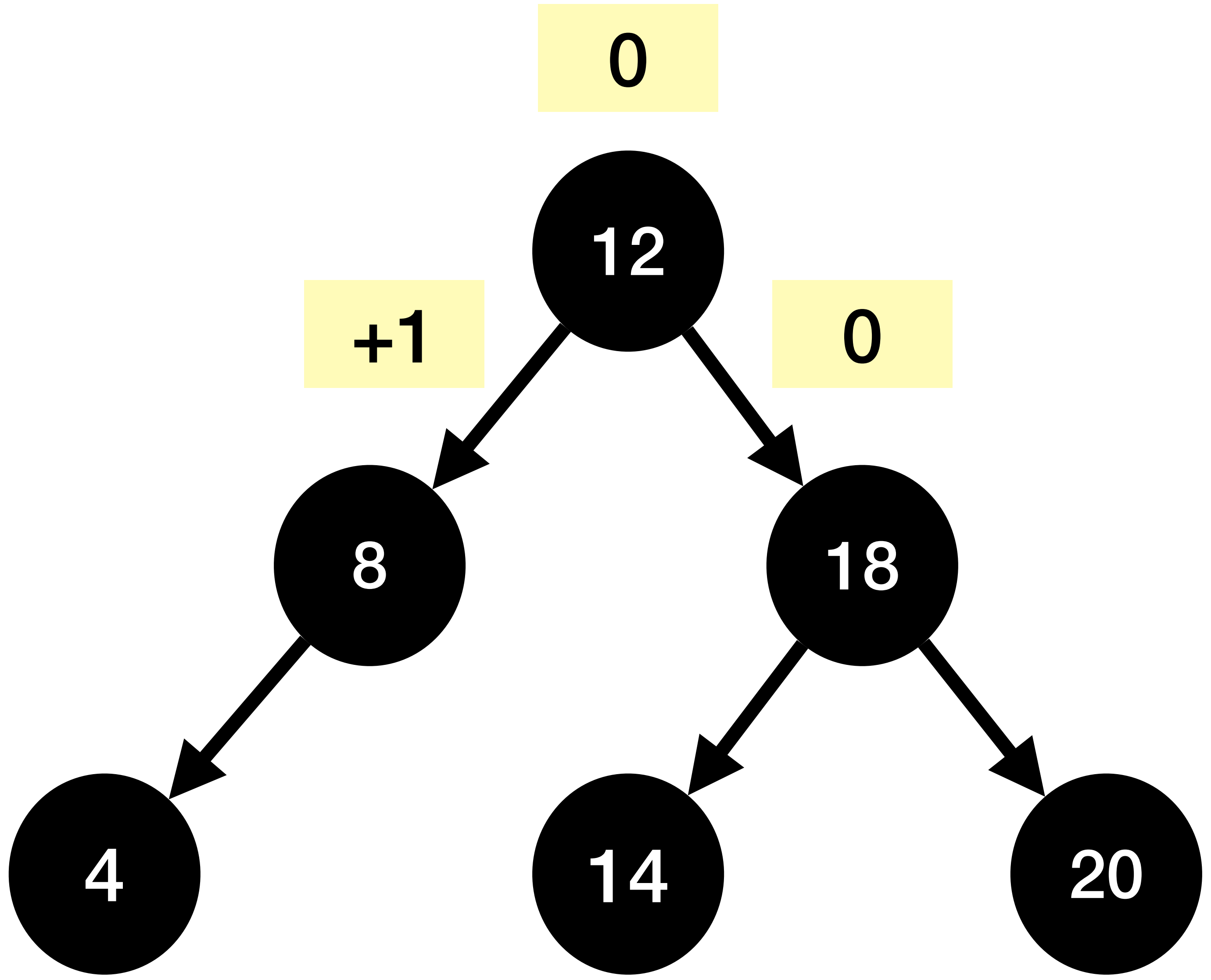
AVL Tree

Rebalancing Rotation

1. LL Rotation (Rotate Right)



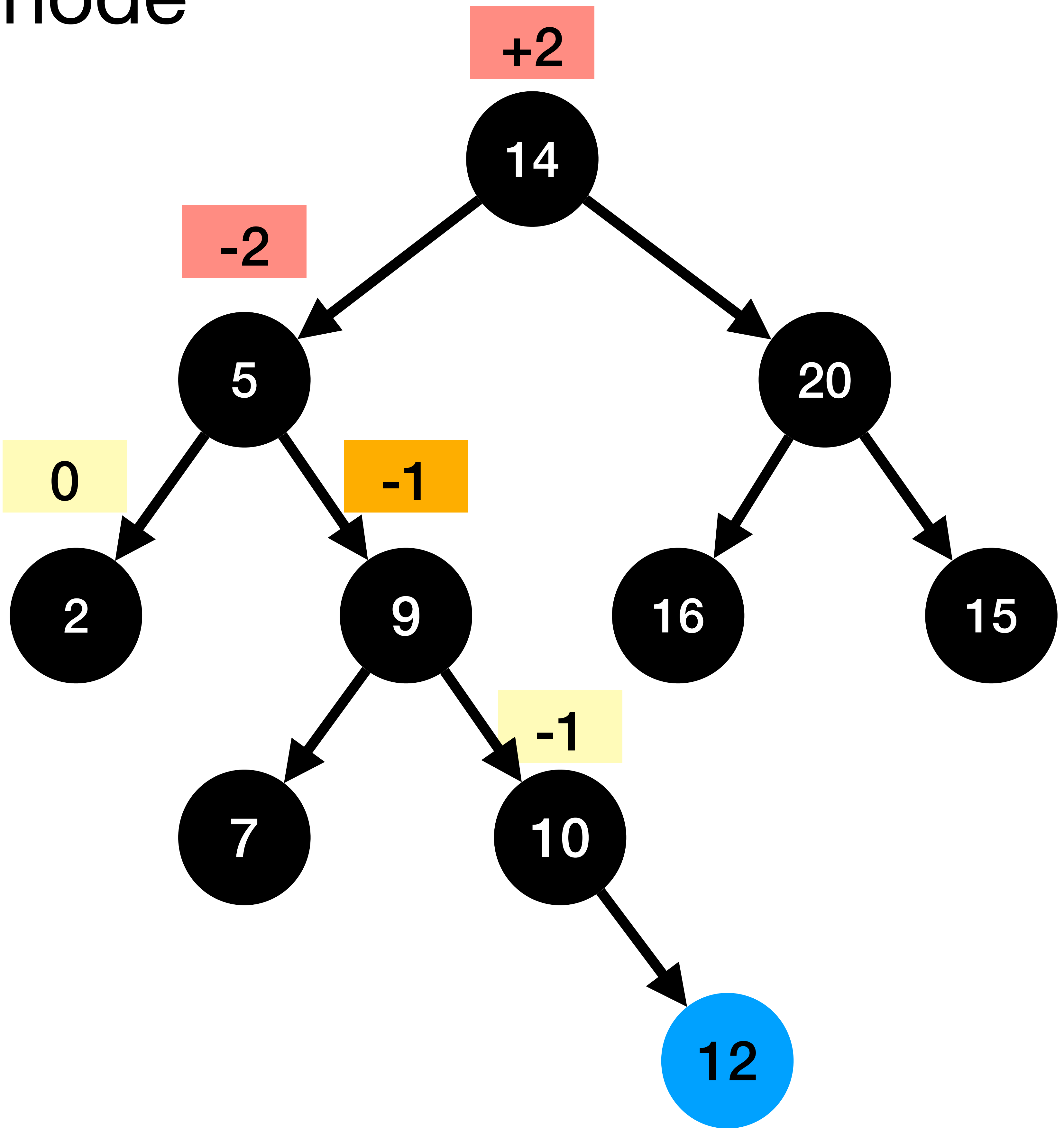




AVL Tree

Rebalancing Rotation

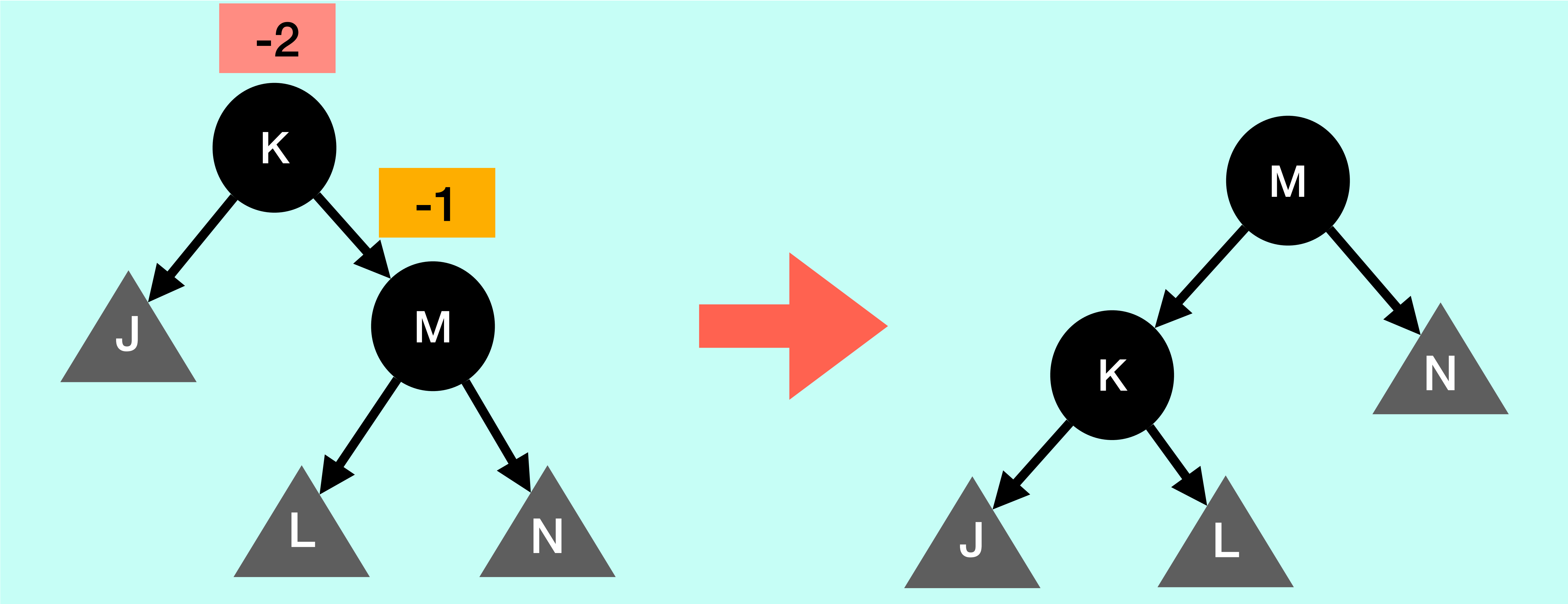
2. ***Right of Right:*** The new node is inserted in the right sub-tree of the right sub-tree of the critical node



AVL Tree

Rebalancing Rotation

2. RR Rotation (Rotate Left)

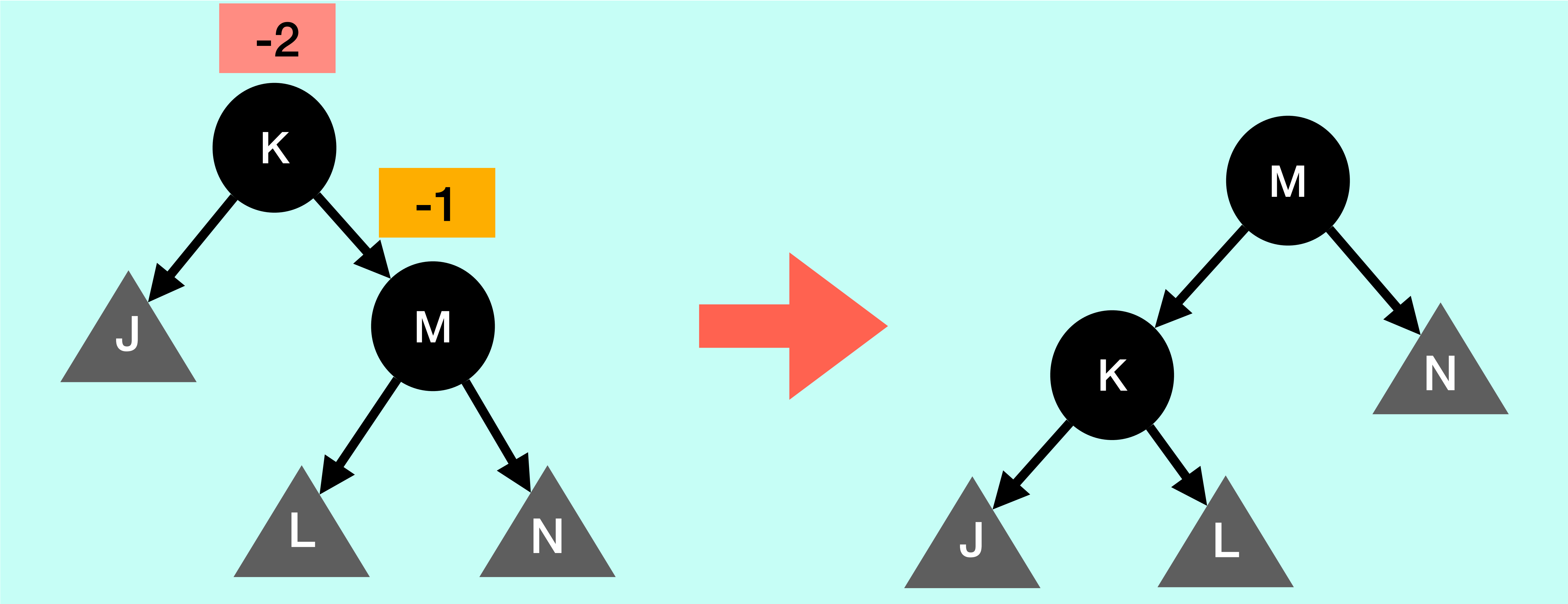


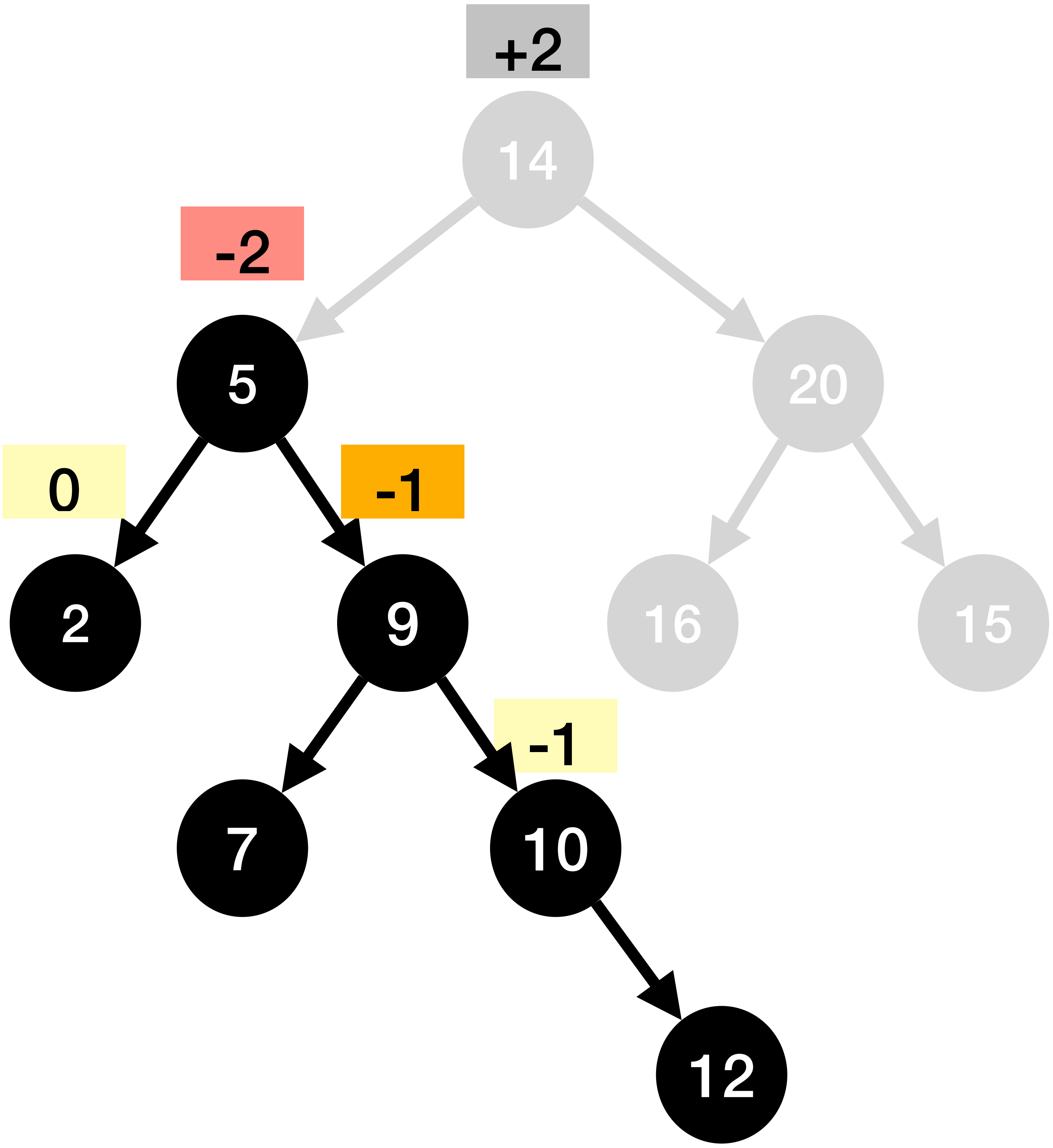
```
TREENODE_T *rotateLeft (TREENODE_T *node)
{
   TREENODE_T *temp;
   temp = node->right;
   node->right = temp->left;
   temp->left = node;
   node = temp;
   return(node);
}
```

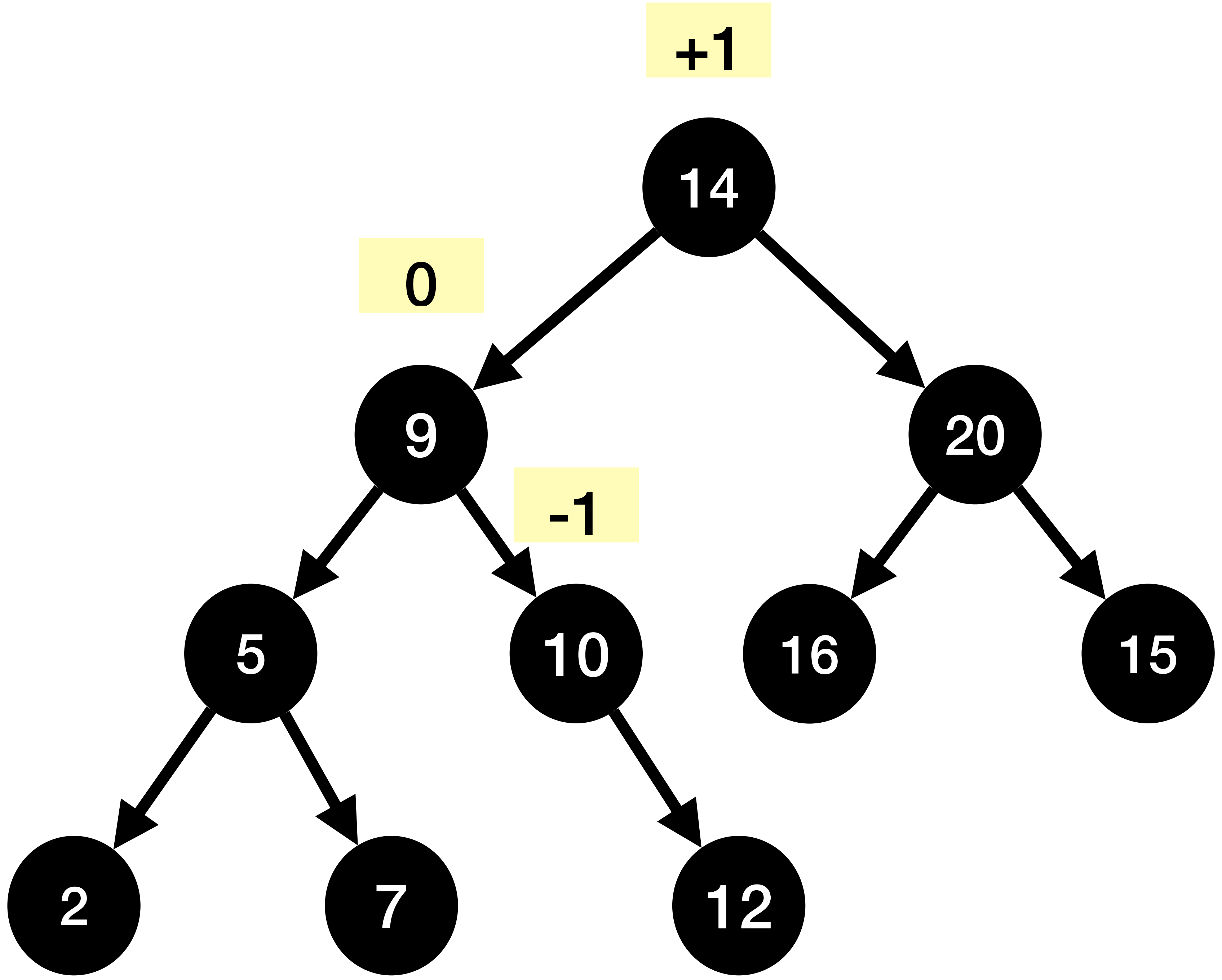
AVL Tree

Rebalancing Rotation

2. RR Rotation (Rotate Left)



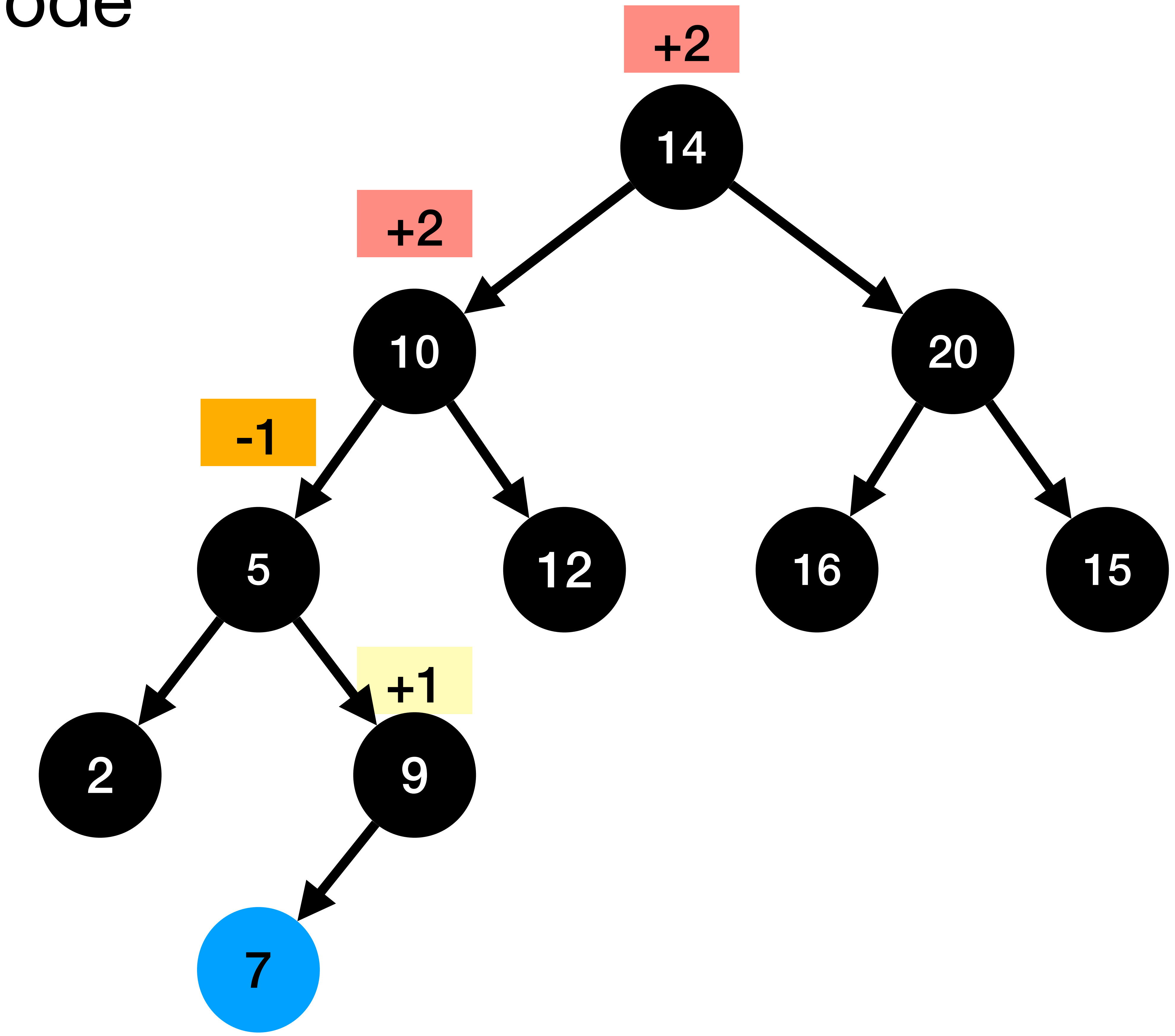




AVL Tree

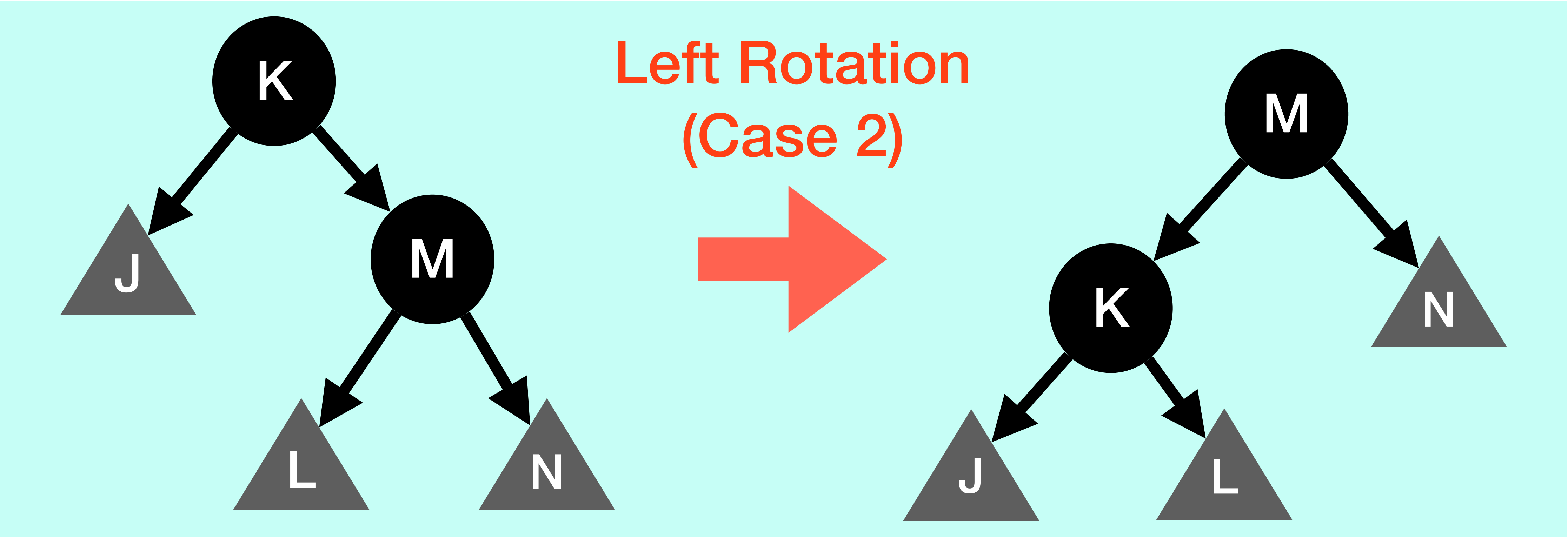
Rebalancing Rotation

3. ***Right of Left***: The new node is inserted in the right sub-tree of the left sub-tree of the critical node



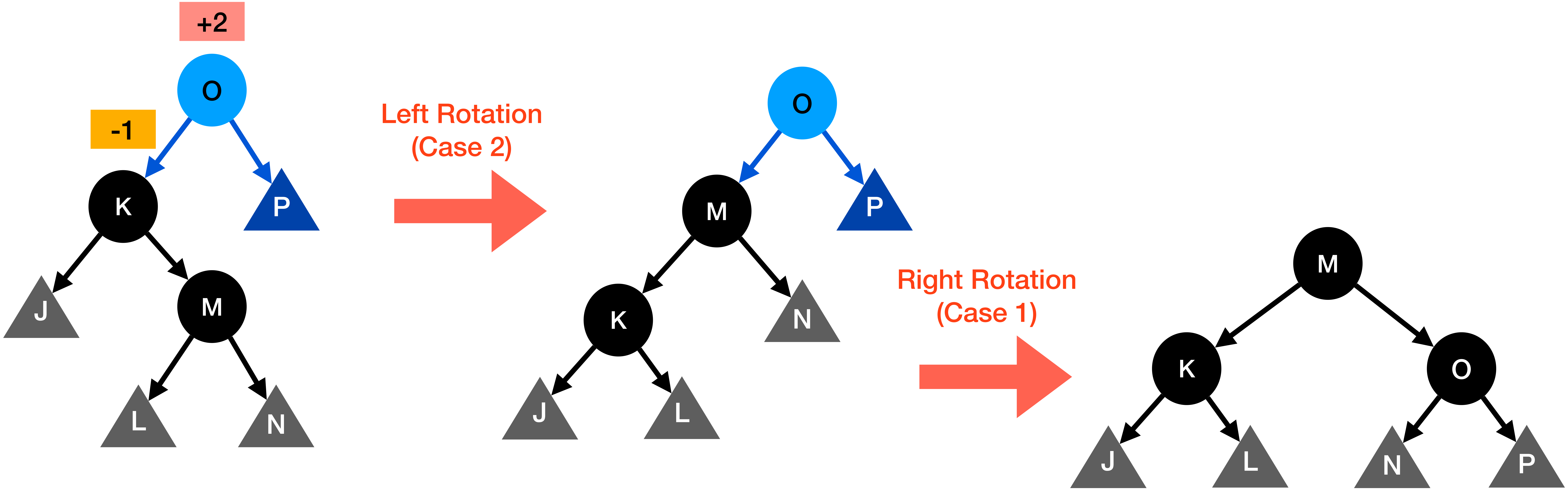
AVL Tree

Rebalancing Rotation



Left Rotation
(Case 2)

3. LR Rotation (Rotate Left to Right)



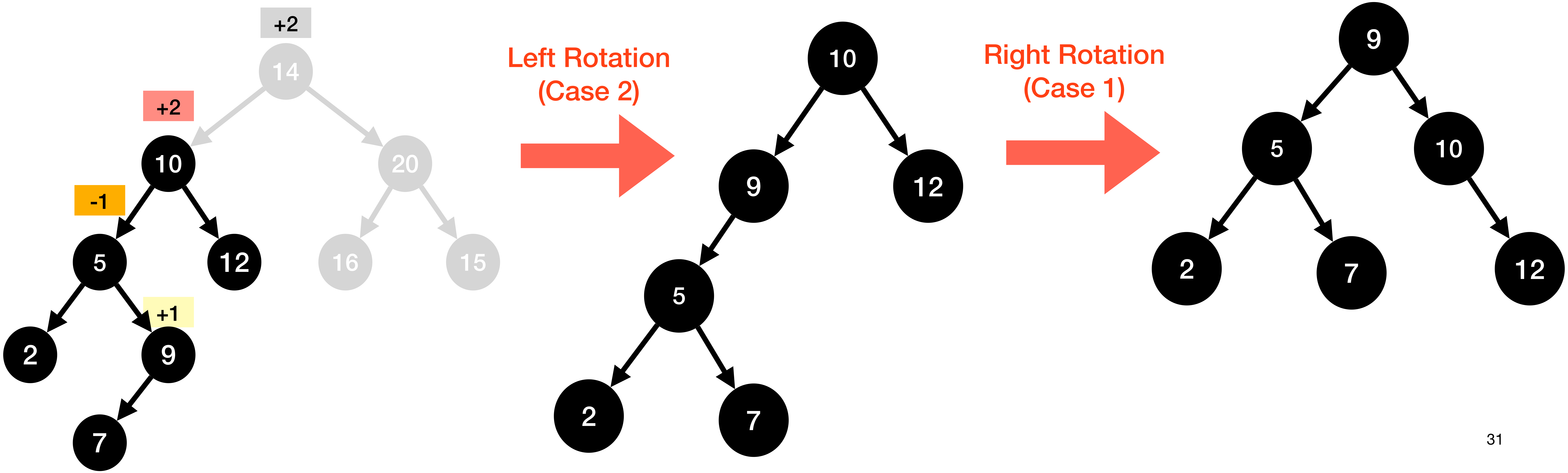
Left Rotation
(Case 2)

Right Rotation
(Case 1)

AVL Tree

Rebalancing Rotation

3. LR Rotation (Rotate Left to Right)



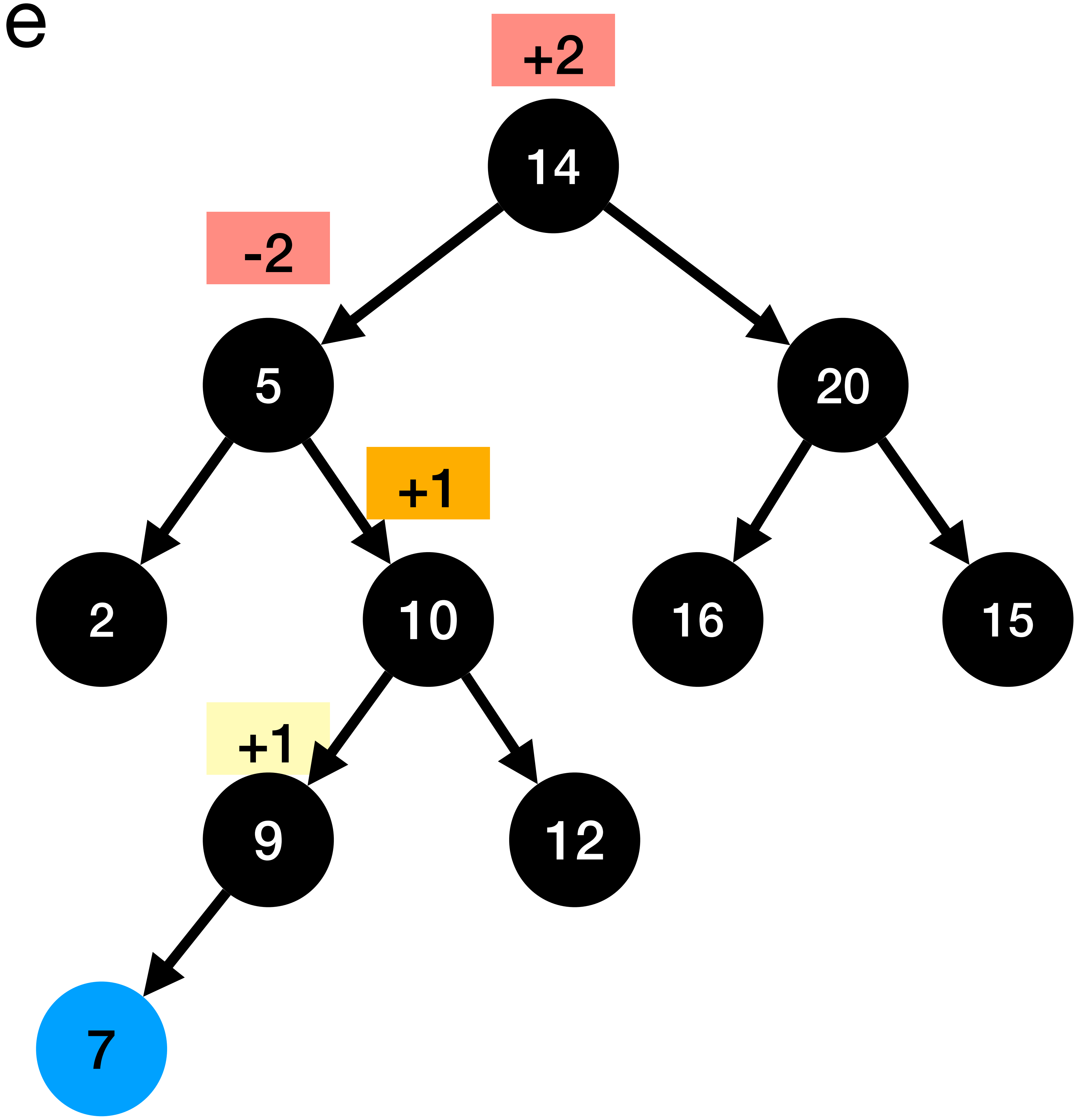
Left Rotation (Case 2)

Right Rotation (Case 1)

AVL Tree

Rebalancing Rotation

4. ***Left of Right***: The new node is inserted in the left sub-tree of the right sub-tree of the critical node



AVL Tree

Rebalancing Rotation

4. RL Rotation (Rotate Right to Left)

Right Rotation (Case 1)

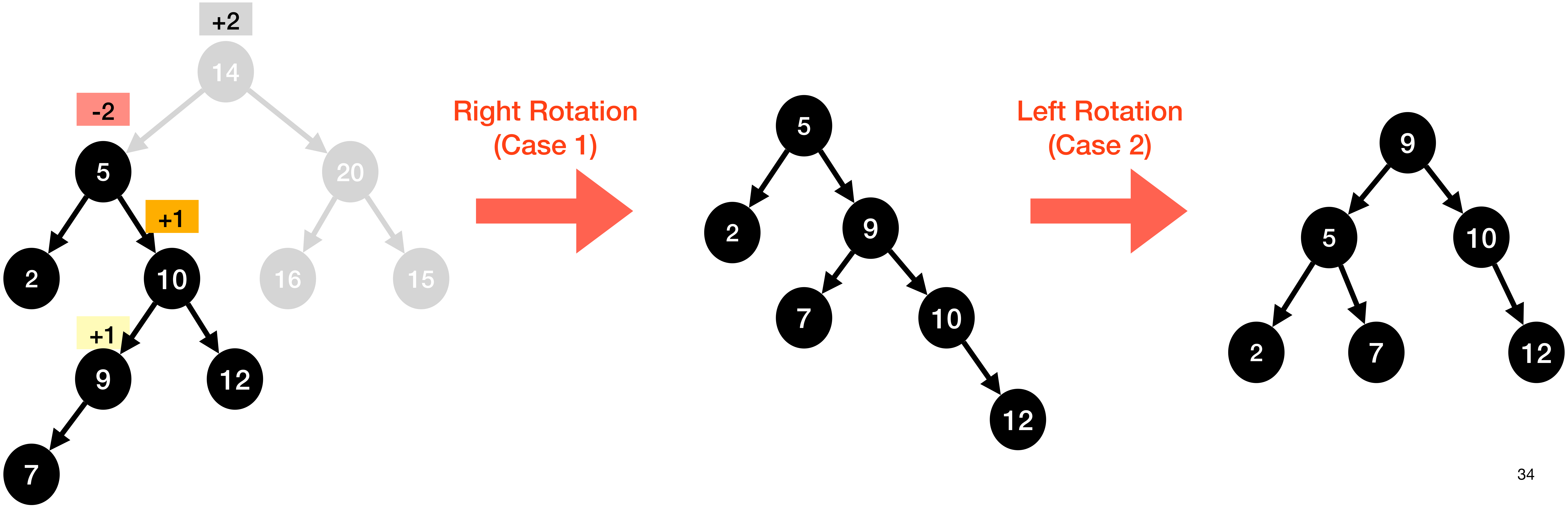
Left Rotation (Case 2)

-2
K
+1
J
O
M
P
L
N
K
J
M
L
O
N
P
M
K
O
J
L
N
P

AVL Tree

Rebalancing Rotation

4. RL Rotation (Rotate Right to Left)



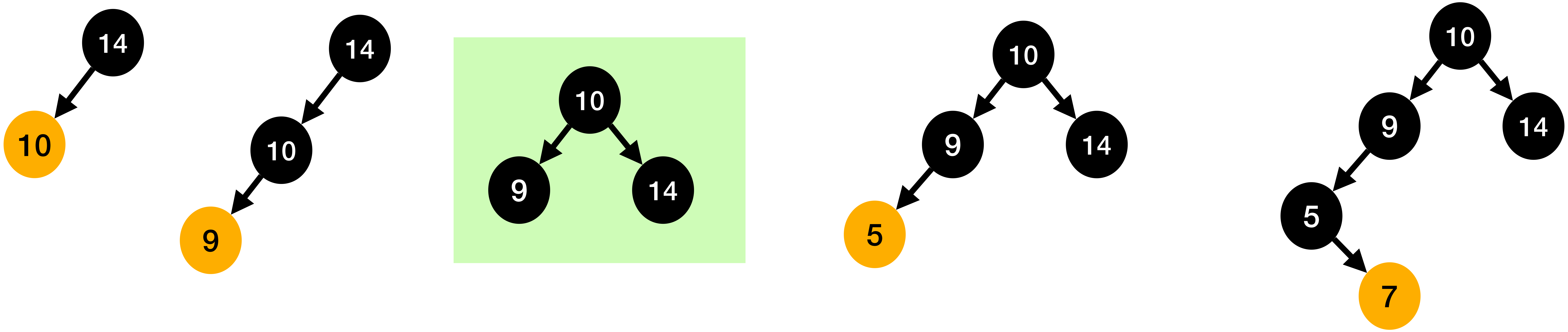
Right Rotation (Case 1)

Left Rotation (Case 2)

AVL Tree

Create an AVL Tree

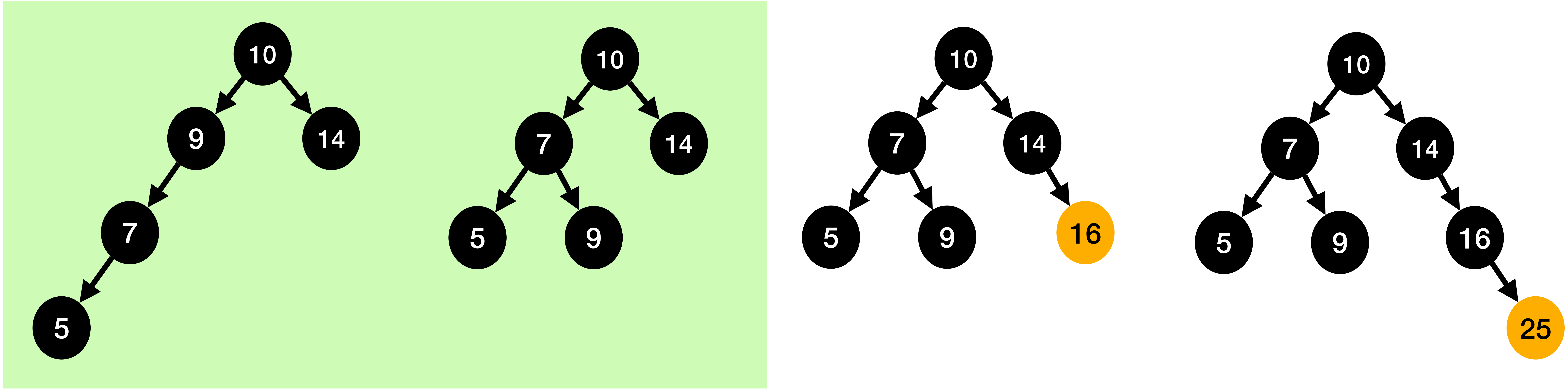
- 14, 10, 9, 5, 7, 16, 25



AVL Tree

Create an AVL Tree

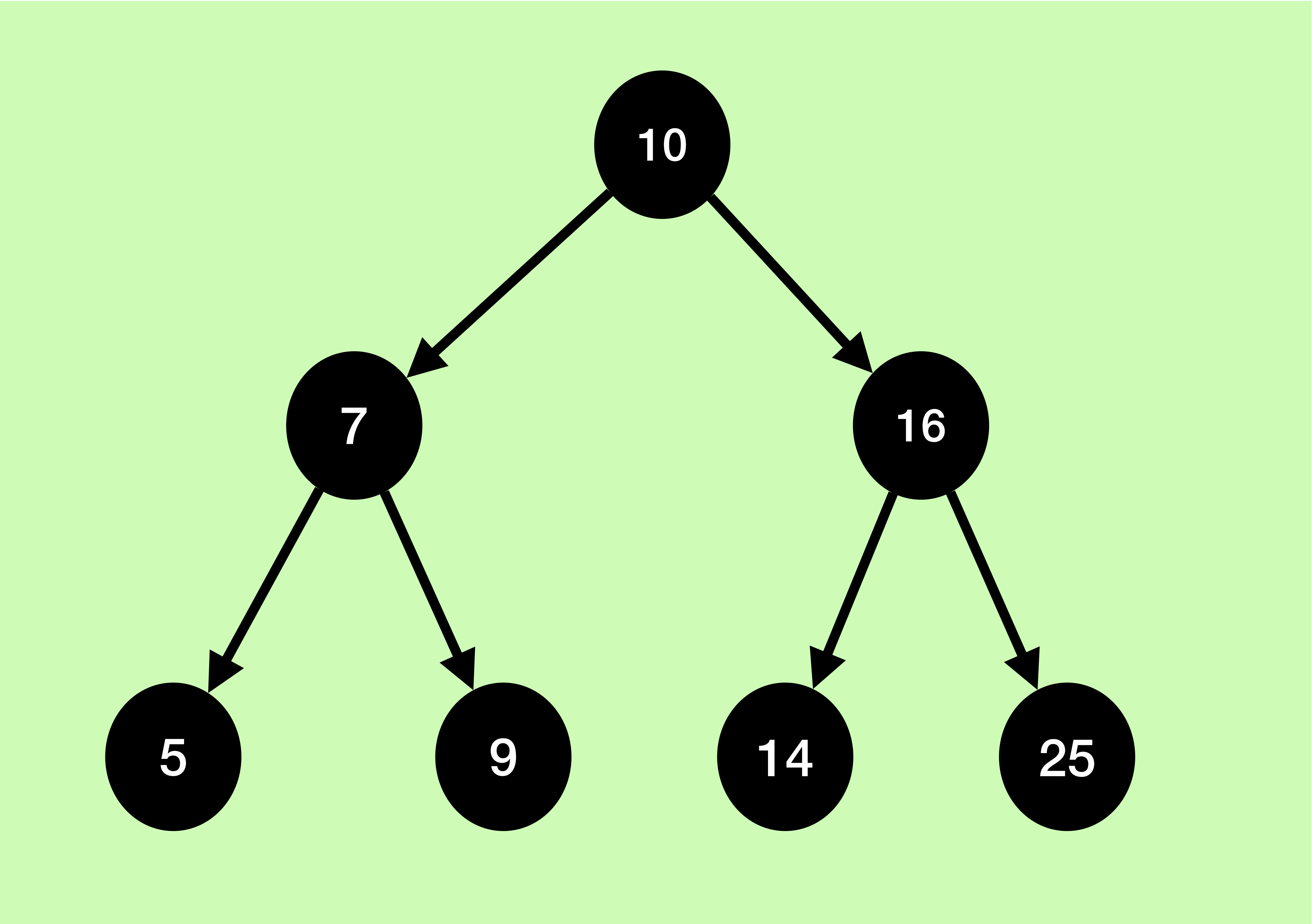
- 14, 10, 9, 5, 7, 16, 25



AVL Tree

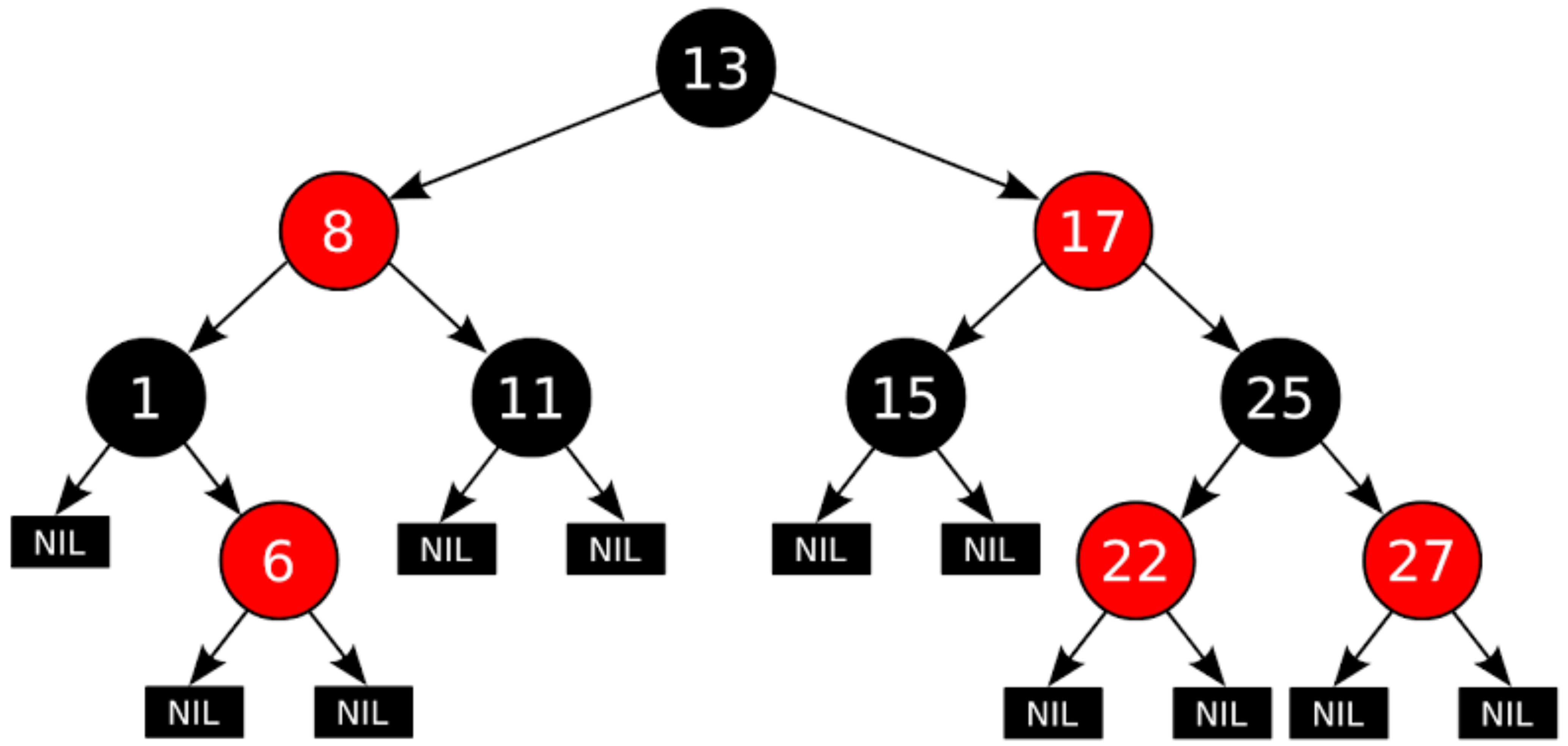
Create an AVL Tree

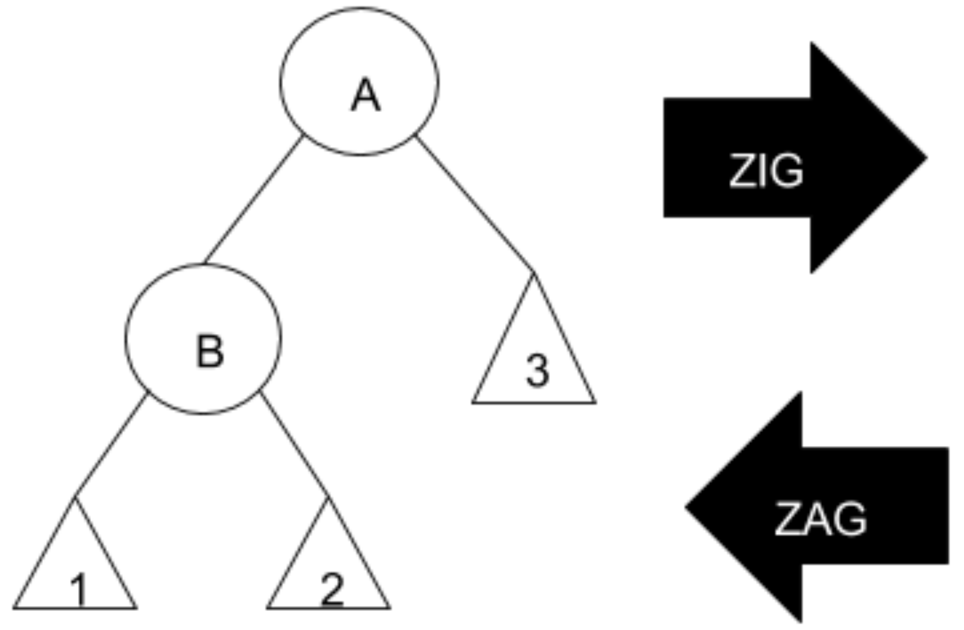
- 14, 10, 9, 5, 7, 16, 25

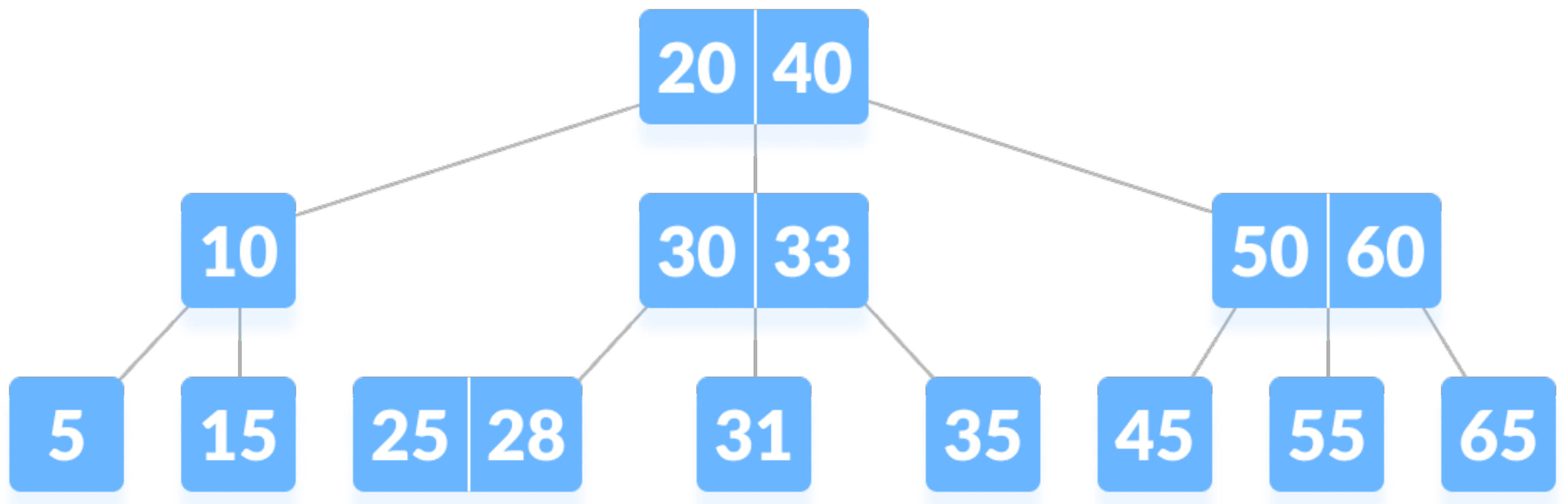


More trees

- Red-black tree
- Splay tree
- B-Tree
- Etc.







Wrap up

- Binary Tree from General Tree
- Binary Search Tree
  - Operations: Create, Search, Delete
- AVL Tree
  - Balance Factor
  - Rotation cases: LL, RR, LR, RL