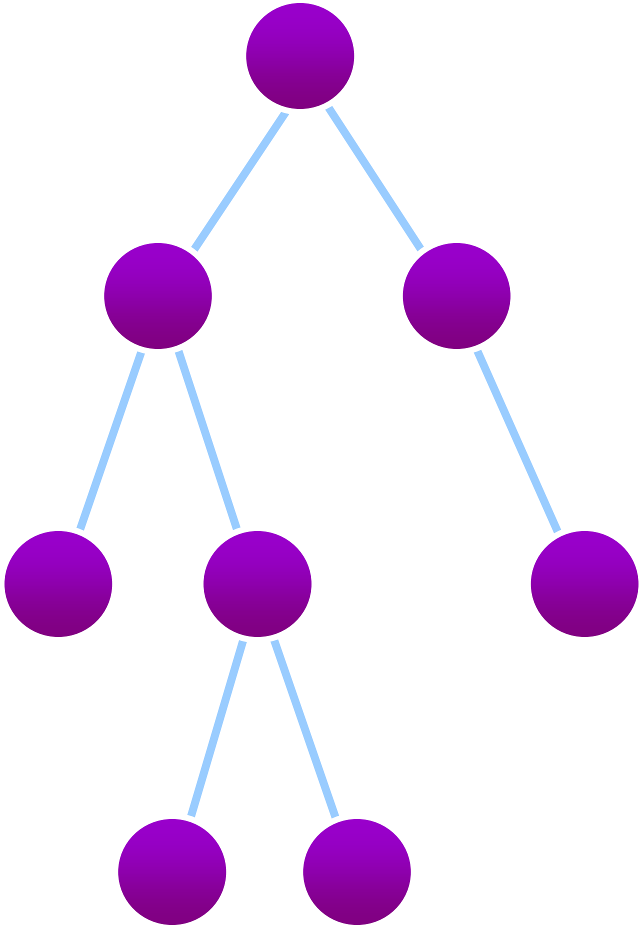
Binary and AVL Trees in C

Jianye Hao

Overview

- Binary tree
  - Degree of tree is 2

```
struct node_s {
  Datatype element;
  struct node_s *leftChild;
  struct node_s *rightChild;
};
typedef struct node_s node;
```



Trees – traversal (Recursion Method)

- Preorder

```
void preorder(node *t) {
    if (t != NULL) {
      printf("%d ", t->element);     /* V */
      preorder(t->leftChild);        /* L */
      preorder(t->rightChild);       /* R */
    }
  }
```

Trees – traversal (Recursion Method)

- Inorder

```
void inorder(node *t) {
  if (t != NULL) {
    inorder(t->leftChild);            /* L */
    printf("%d ", t->element);   /* V */
    inorder(t->rightChild);          /* R */
  }
}
```

Trees – traversal (Recursion Method)

- Postorder

```
void postorder(node *t) {
    if (t != NULL) {
      postorder(t->leftChild);           /* L */
      postorder(t->rightChild);          /* R */
      printf("%d ", t->element);         /* V */
    }
  }
```

Trees - traversal

- Preorder

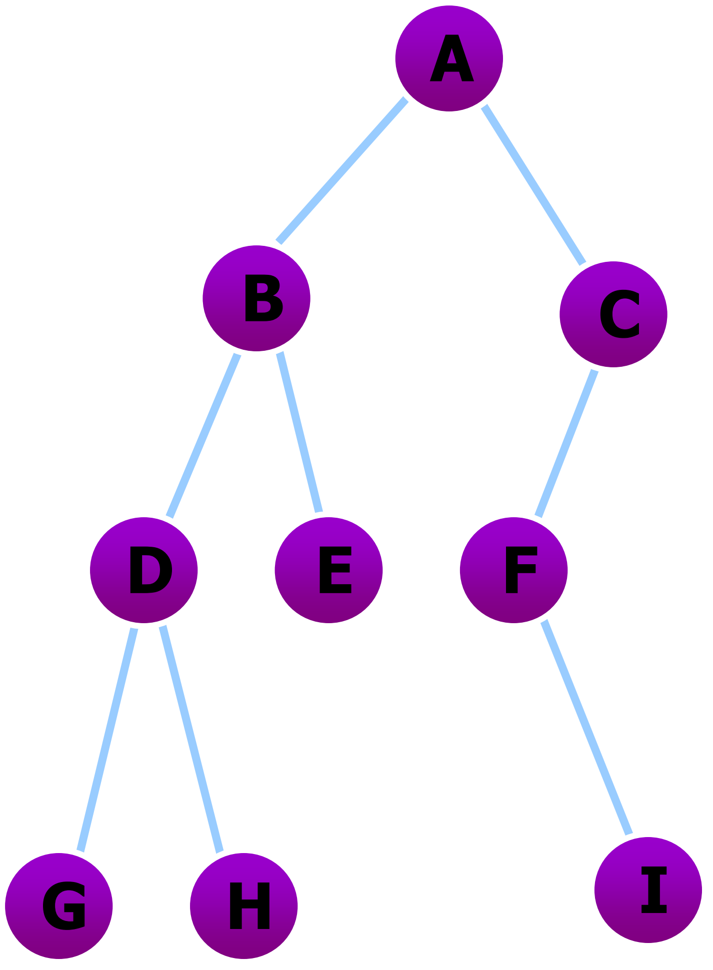
  A B D G H E C F I

- Inorder

  G D H B E A F I C

- Postorder

  G H D E B I F C A

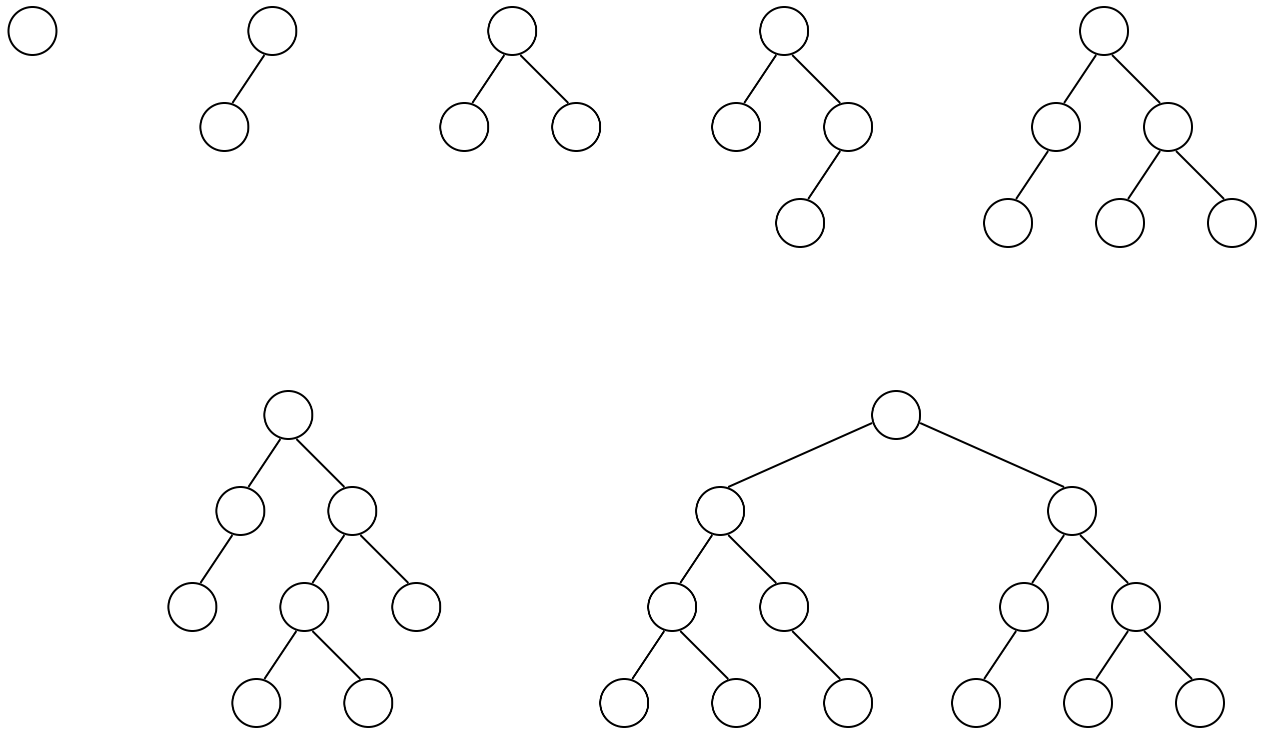


AVL Tree

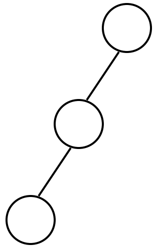
Definition

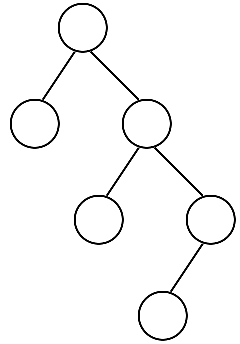
- An AVL tree (or Height-Balanced tree) is a binary search tree such that:
  - The height of the left and right subtrees of the root differ by at most 1.
  - The left and right subtrees of the root are AVL trees.

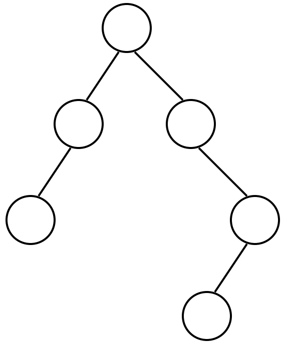
AVL Tree



Non-AVL Tree







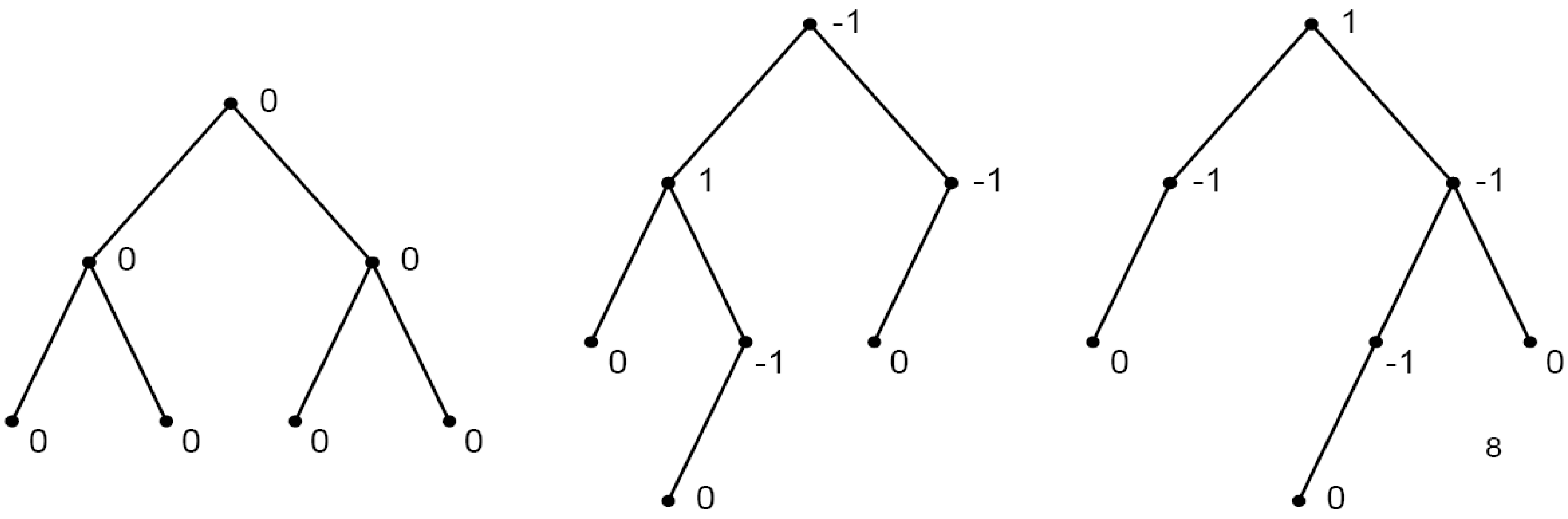
Balance Factor

- To keep track of whether a binary search tree is a AVL tree, we associate with each node a balance factor, which is

  Height(right subtree) – Height(left subtree)

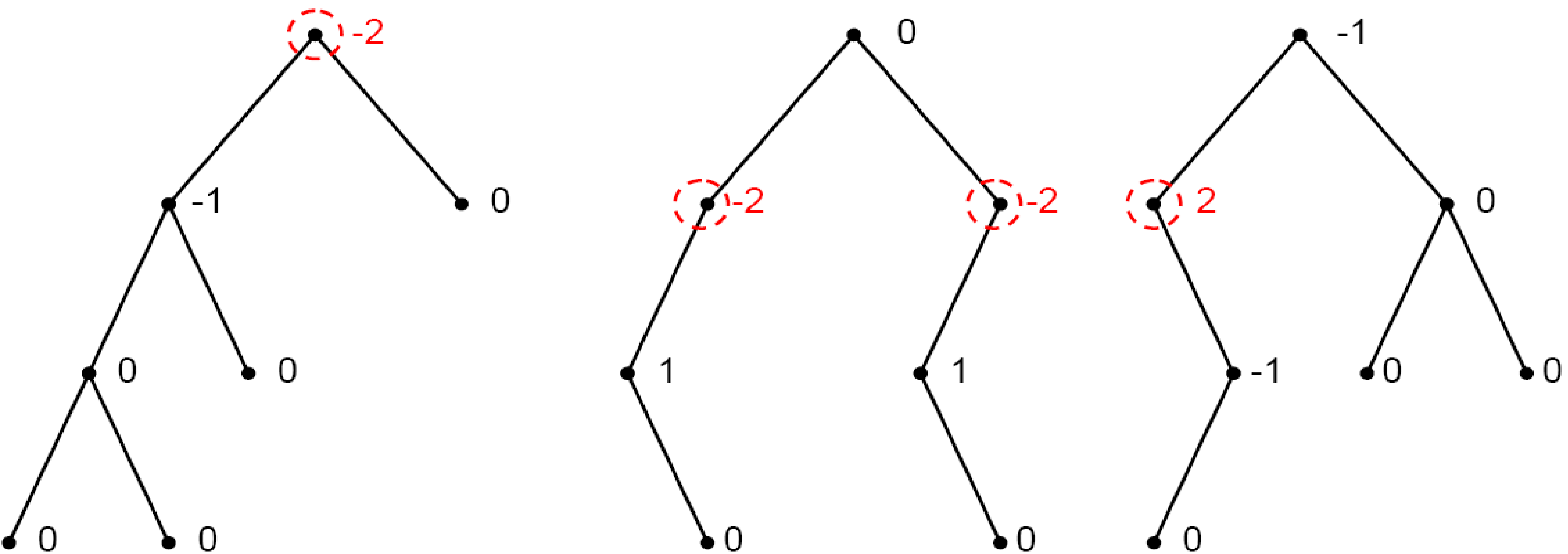
AVL tree

- Height(right subtree) – Height(left subtree)



Non-AVL tree

- Height(right subtree) – Height(left subtree)



AVL tree structure in C

For each node, the difference of height between left and right are no more than 1.

```
struct AVLnode_s {
  Datatype element;
  struct AVLnode *left;
  struct AVLnode *right;
};
typedef struct AVLnode_s AVLnode;
```

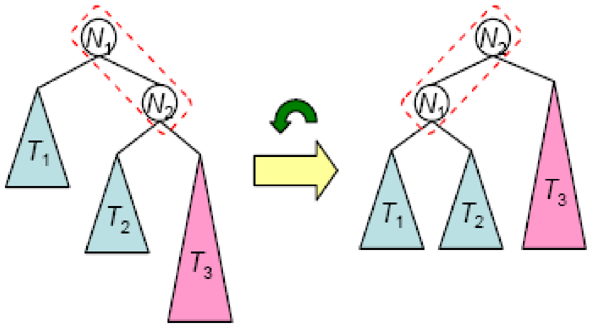
Four Models

- There are four models about the operation of AVL Tree:

  LL RR LR RL

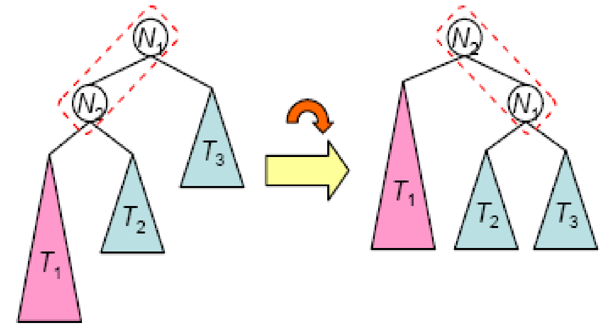
Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



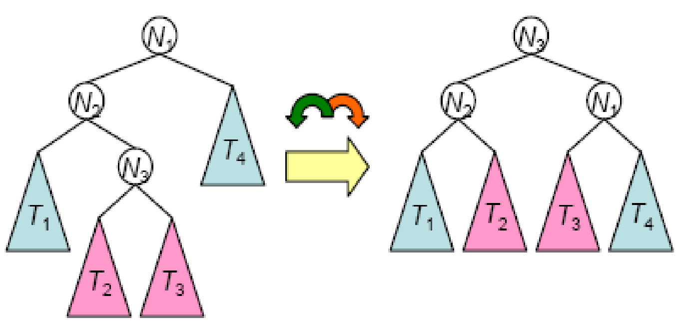
Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



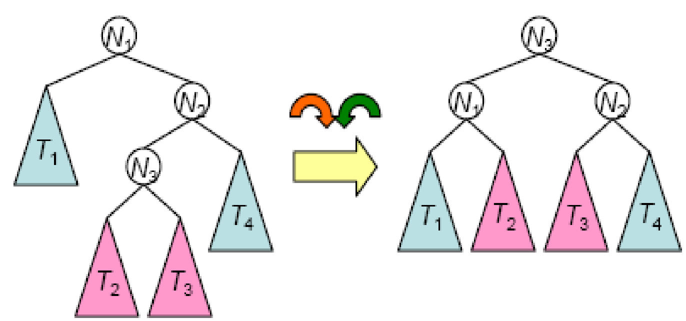
Case 3: insertion to *right* subtree of *left* child

Solution: *Left*-*right* rotation



Case 4: insertion to *left* subtree of *right* child

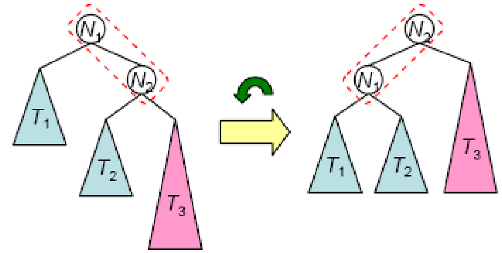
Solution: *Right*-*left* rotation

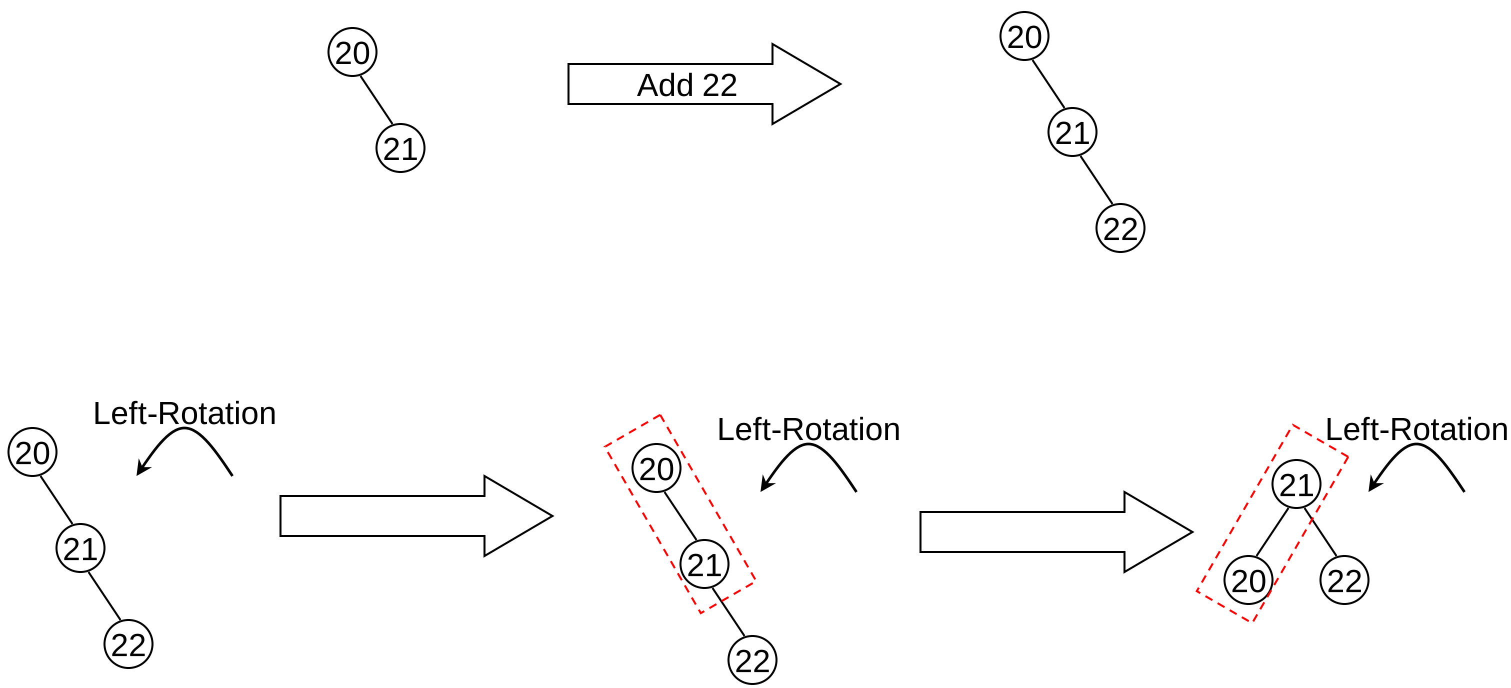


Left-Rotation

Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation

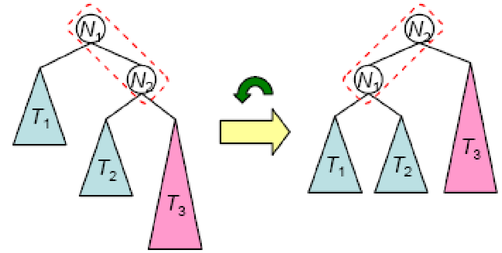


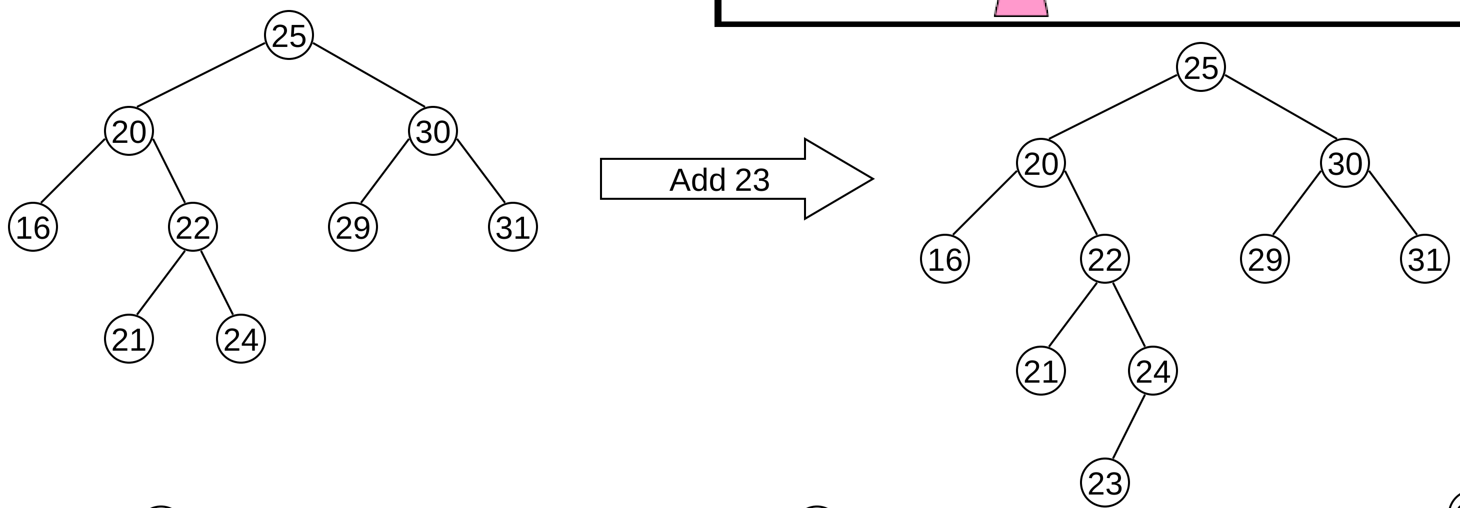


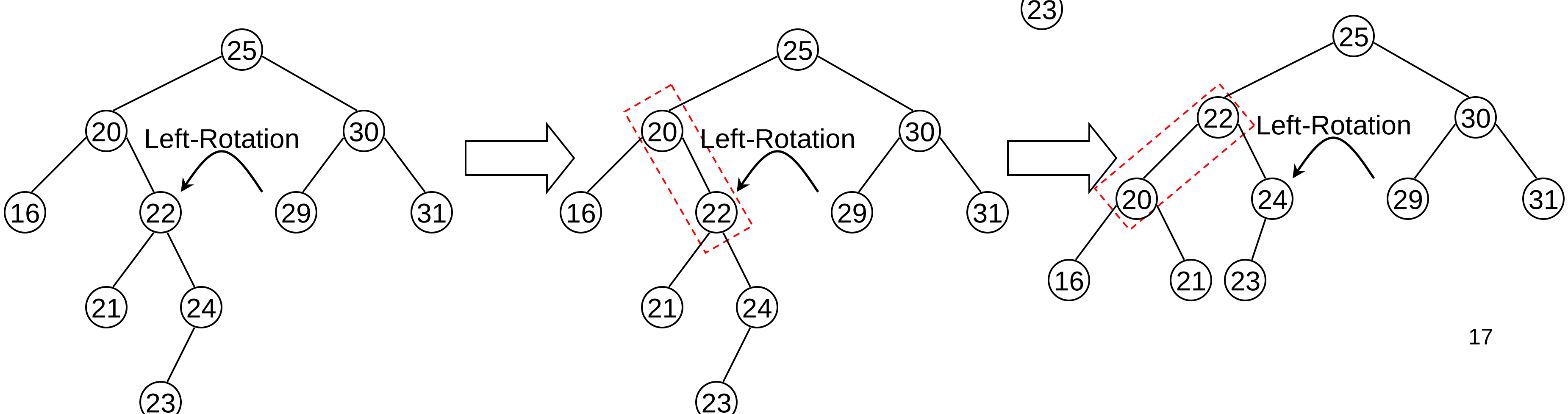
Left-Rotation

Case 1: insertion to *right* subtree of *right* child

Solution: *Left* rotation



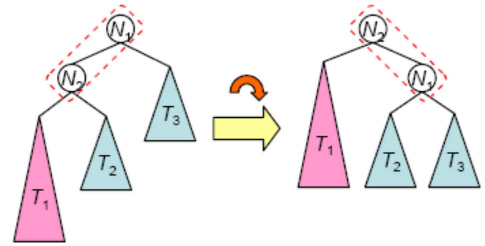


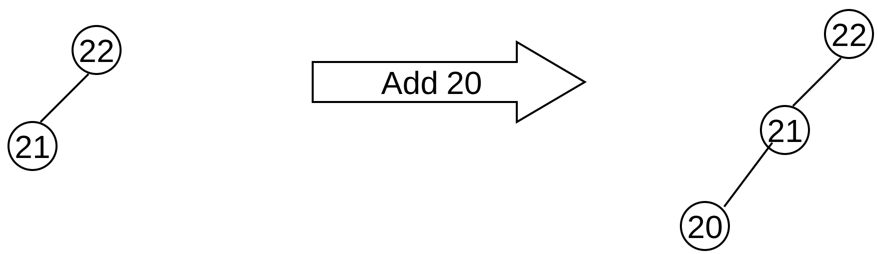


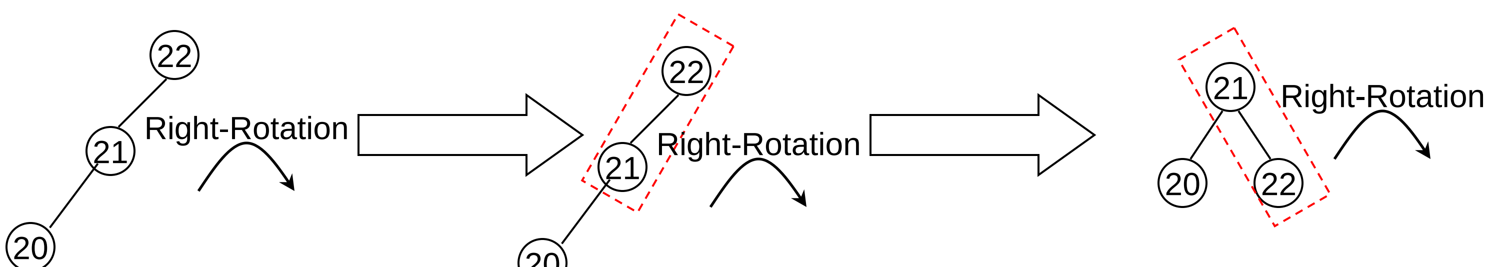
Right-Rotation

Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation



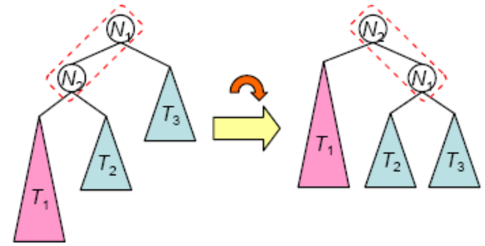


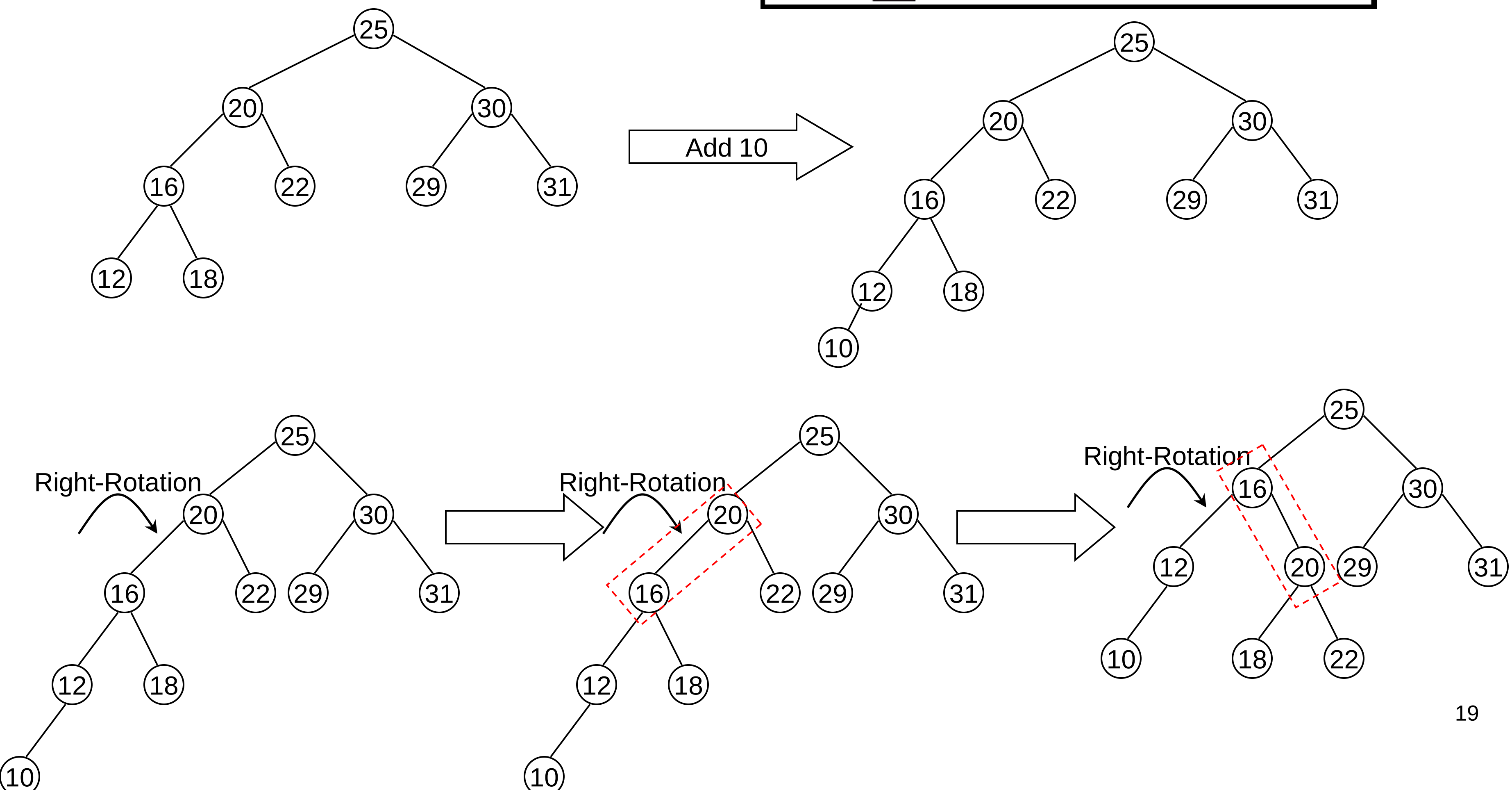


Right-Rotation

Case 2: insertion to *left* subtree of *left* child

Solution: *Right* rotation

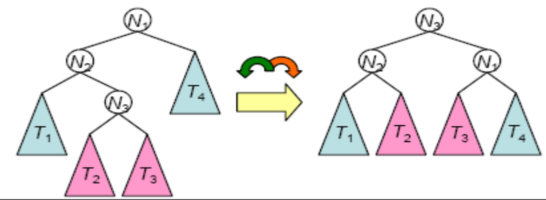


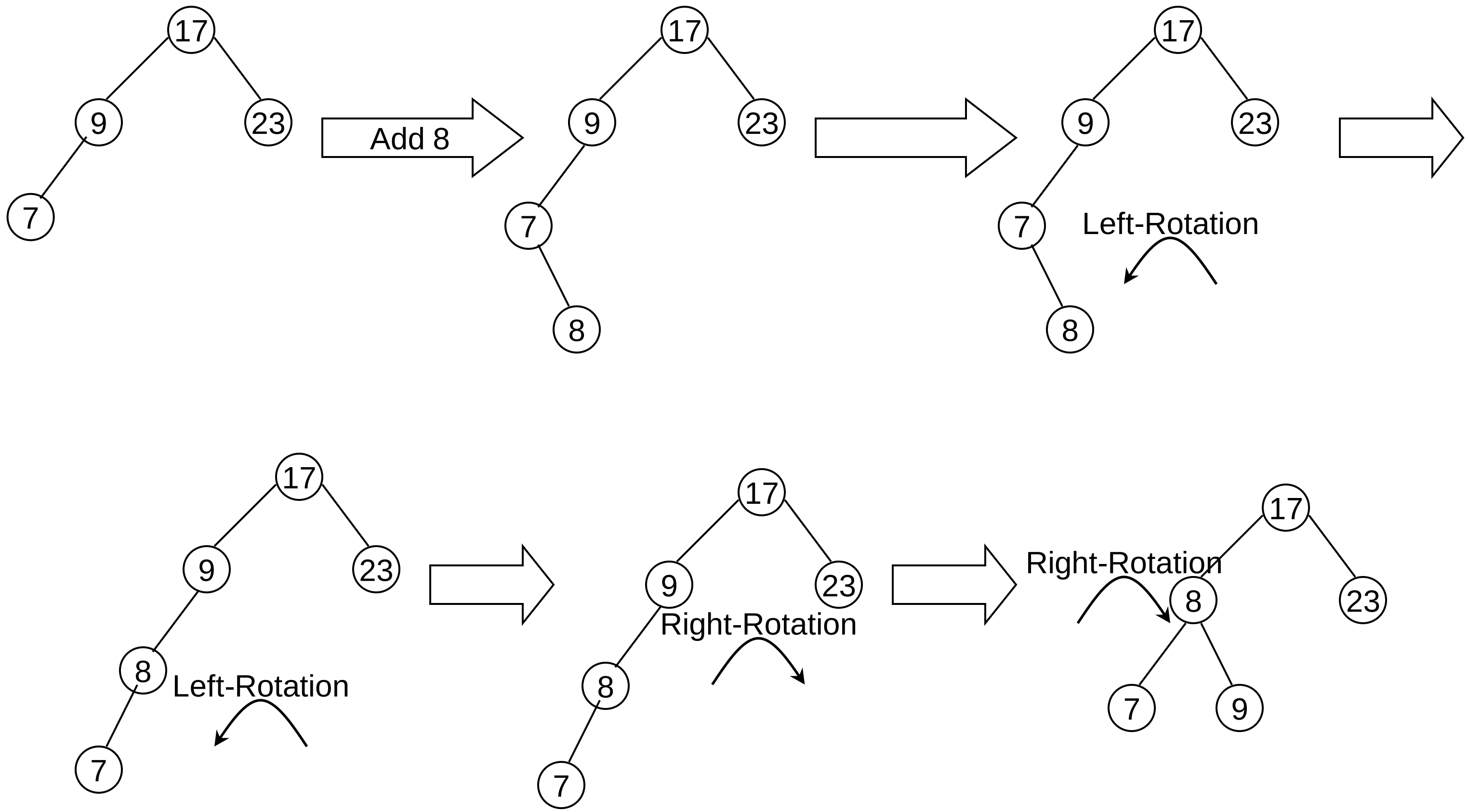


Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

Solution: *Left*-*right* rotation

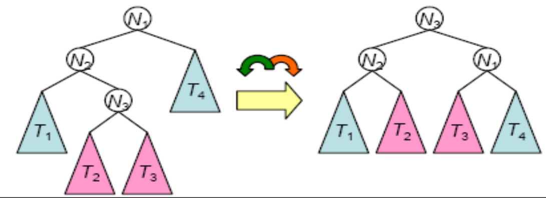


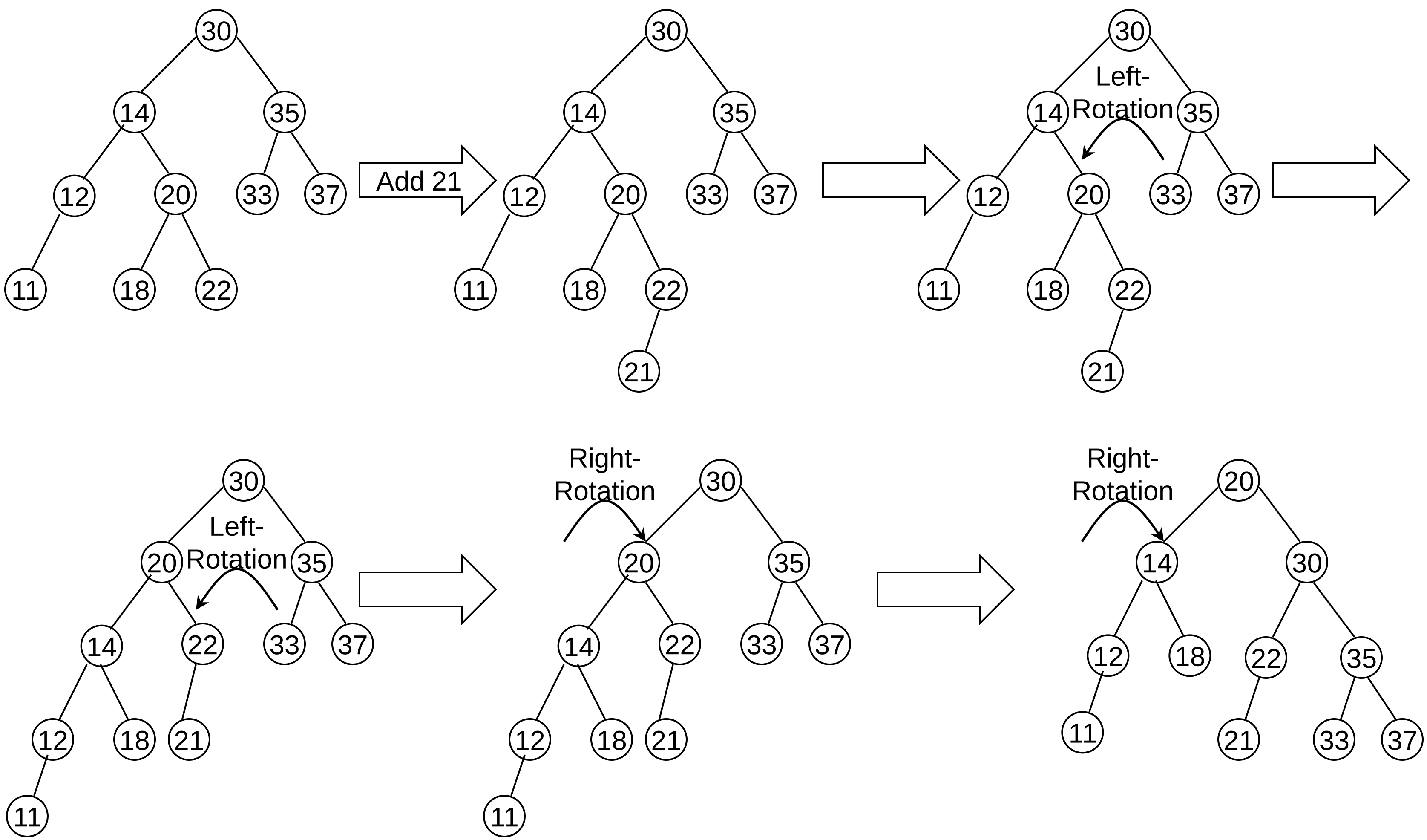


Left-Right Rotation

Case 3: insertion to *right* subtree of *left* child

Solution: *Left-right* rotation

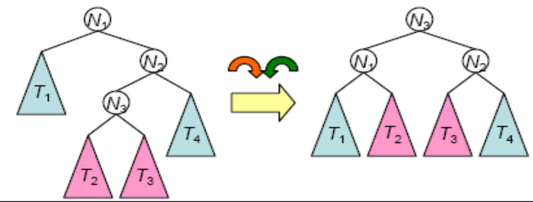




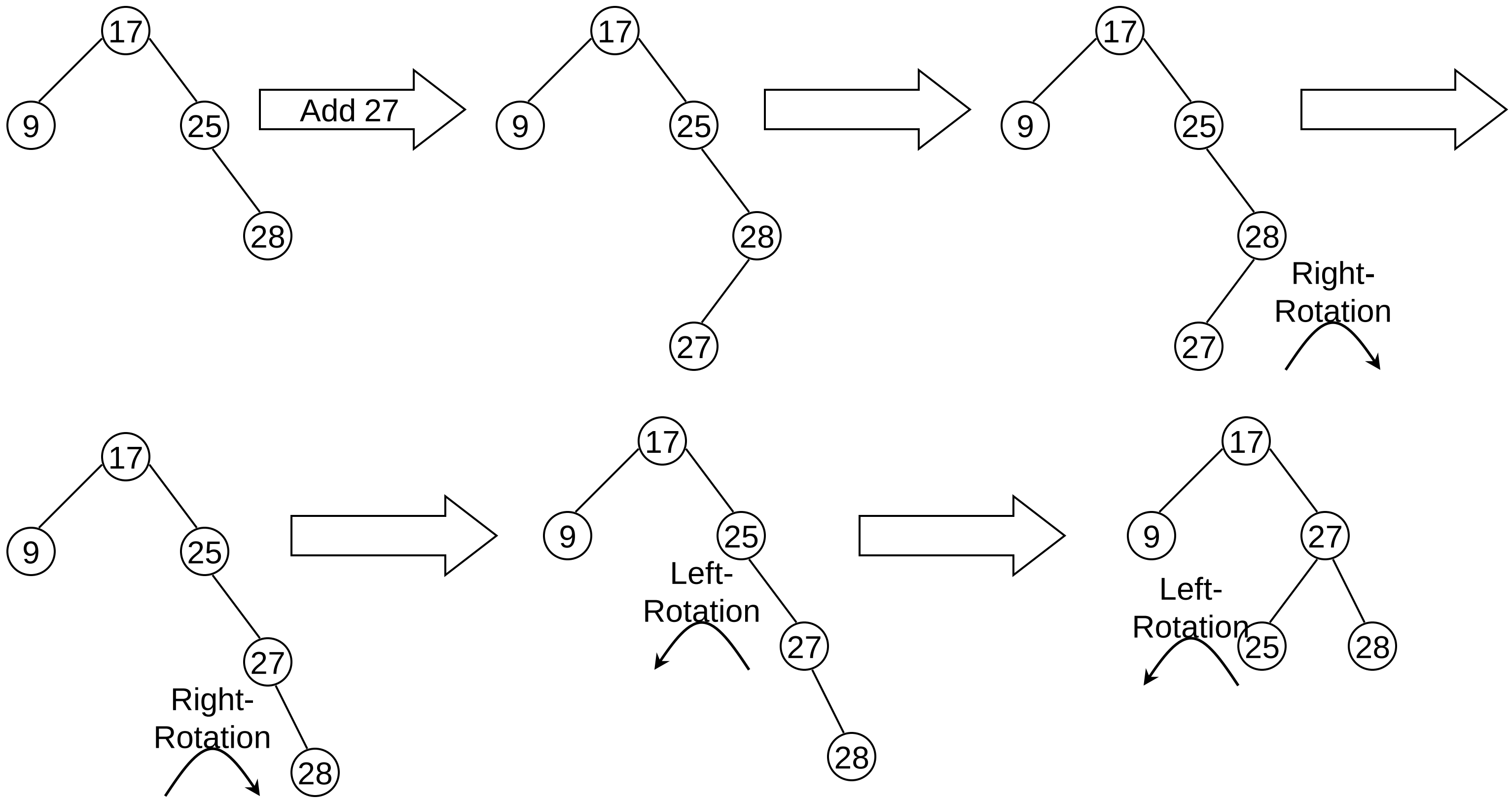
Right-Left Rotation

Case 4: insertion to *left* subtree of *right* child

Solution: *Right*-*left* rotation



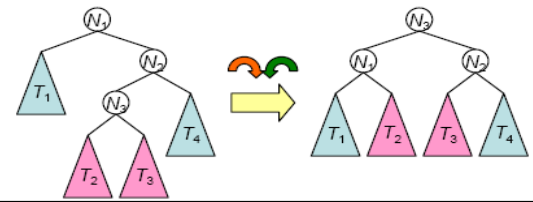
35



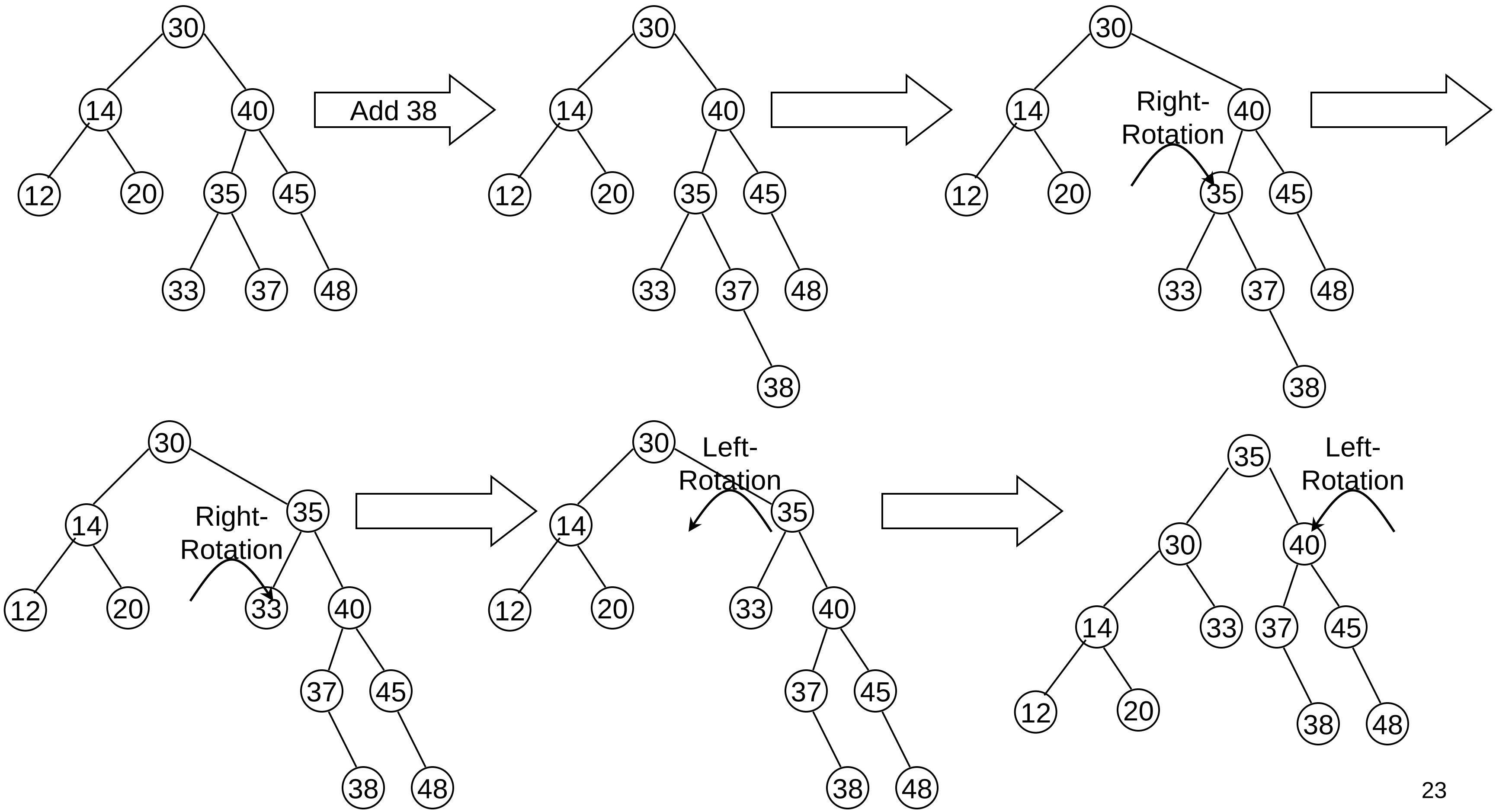
Right-Left Rotation

Case 4: insertion to *left* subtree of *right* child

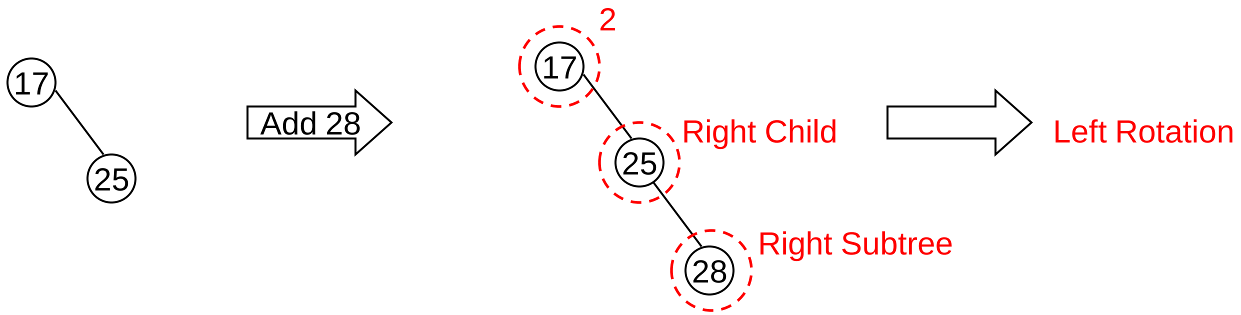
Solution: *Right*-*left* rotation



35

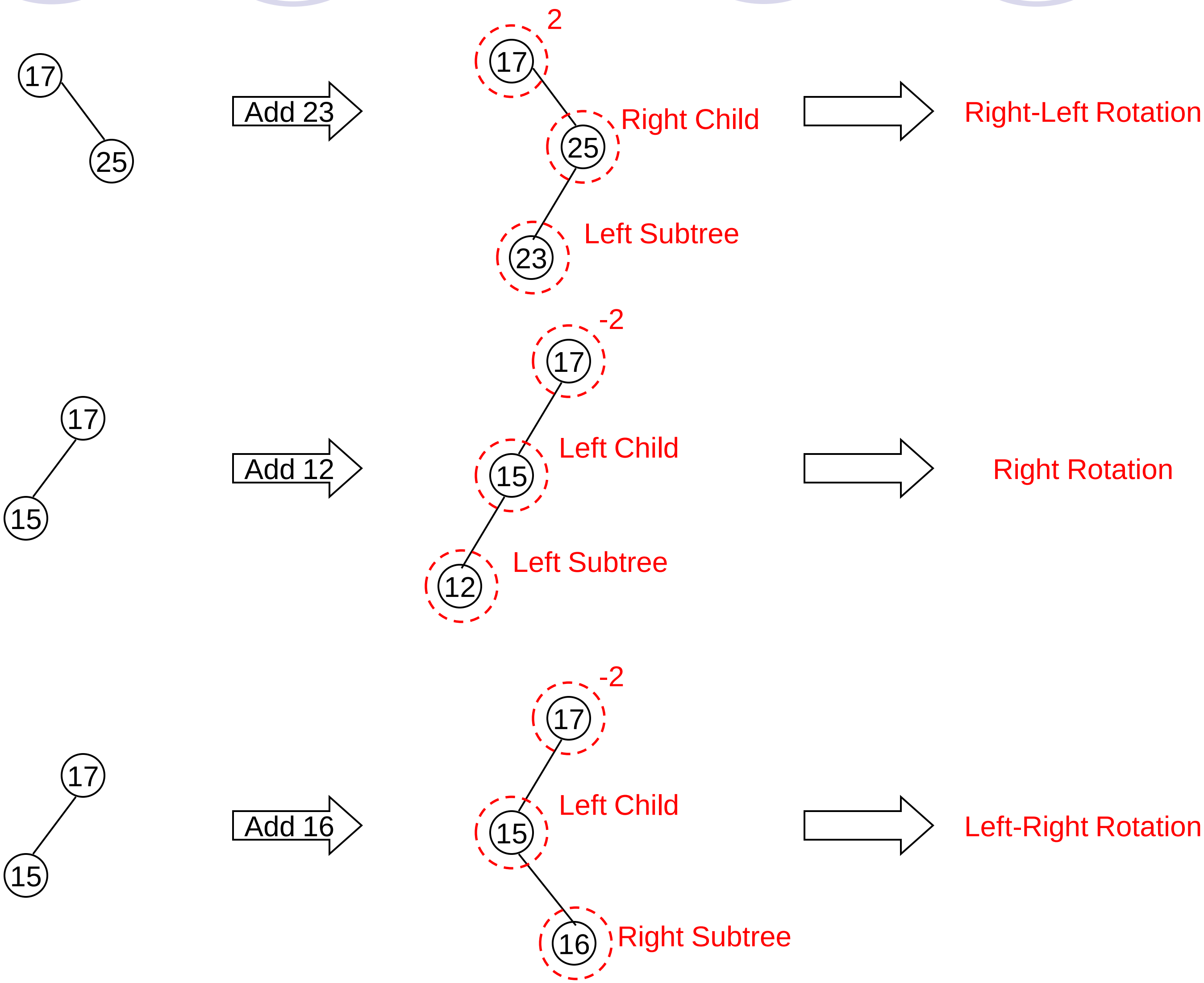


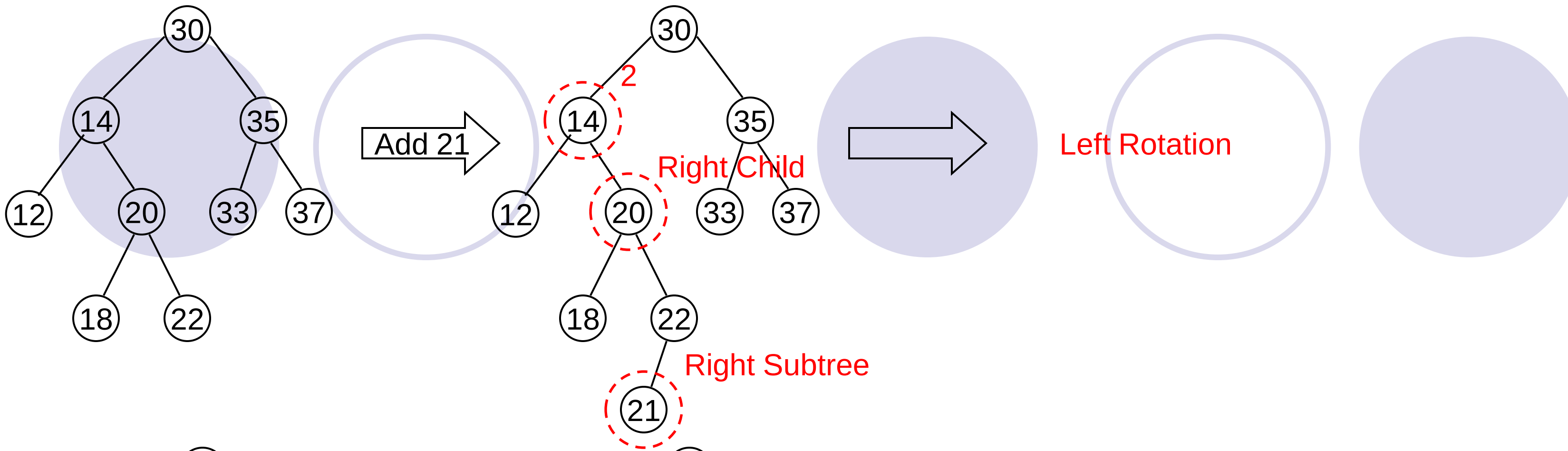
How to identify rotations?

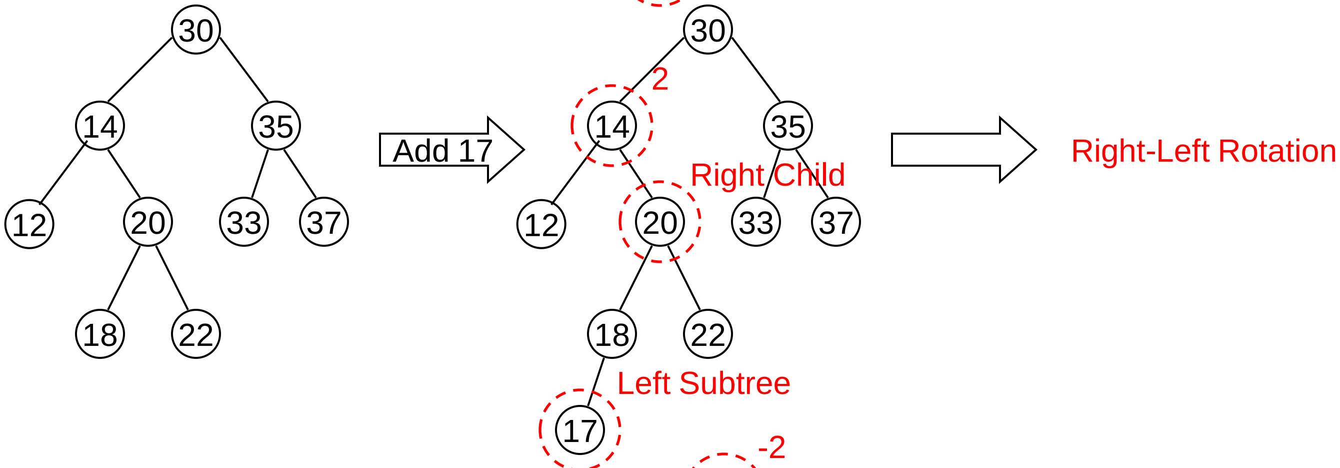


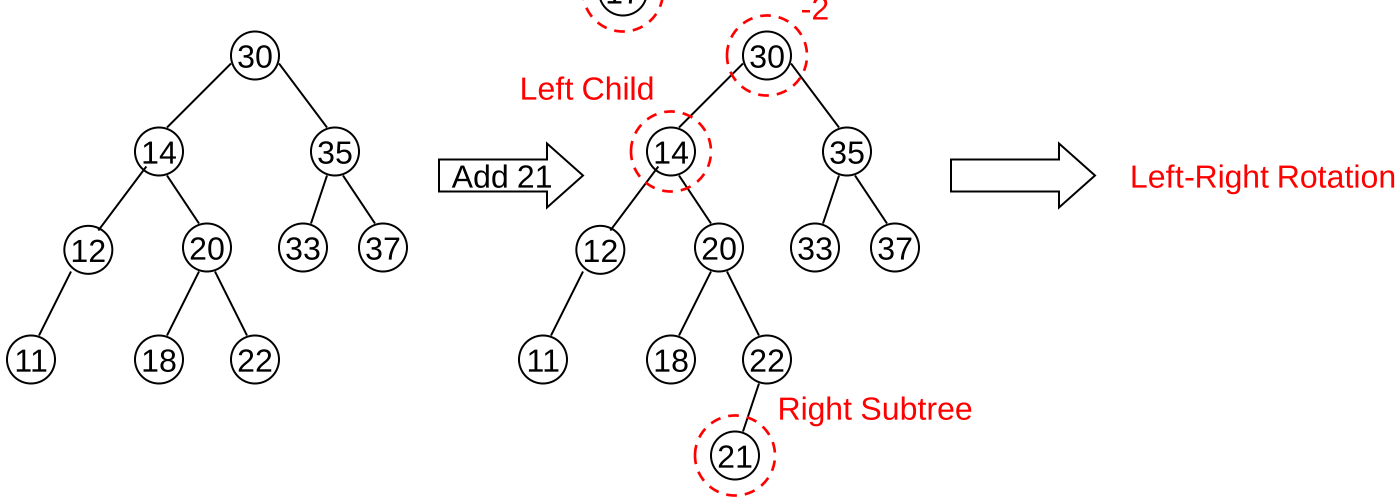
- First find the node that cause the imbalance (balance factor)
- Then find the corresponding child of the imbalanced node (left node or right node)
- Finally find the corresponding subtree of that child (left or right)

How to identify rotations?




Add 21
2
Right Child
Right Subtree
Left Rotation


Add 17
2
Right Child
Left Subtree
Right-Left Rotation


Add 21
-2
Left Child
Right Subtree
Left-Right Rotation

Balancing an AVL tree after an insertion

- Begin at the node containing the item which was just inserted and move back along the access path toward the root.{
  - For each node determine its height and check the balance condition. {
    - If the tree is AVL balanced and no further nodes need be considered.
    - else If the node has become unbalanced, a rotation is needed to balance it.

    }

} we proceed to the next node on the access path.

```
AVLnode *insert(Datatype x, AVLnode *t) {
  if (t == NULL) {
        /* CreateNewNode */
  }
  else if (x < t->element) {
        t->left = insert(x, t->left);
        /* DoLeft */
  }
  else if (x > t->element) {
        t->right = insert(x, t->right);
        /* DoRight */
  }
}
```

AVL tree

- **CreateNewNode**

```
t = malloc(sizeof(struct AVLnode);
t->element = x;
t->left = NULL;
t->right = NULL;
```

AVL tree

- **DoLeft**

```
if (height(t->left) - height(t->right) == 2)
  if (x < t->left->element)
    t = singleRotateWithLeft(t); // LL
  else
    t = doubleRotateWithLeft(t); // LR
```

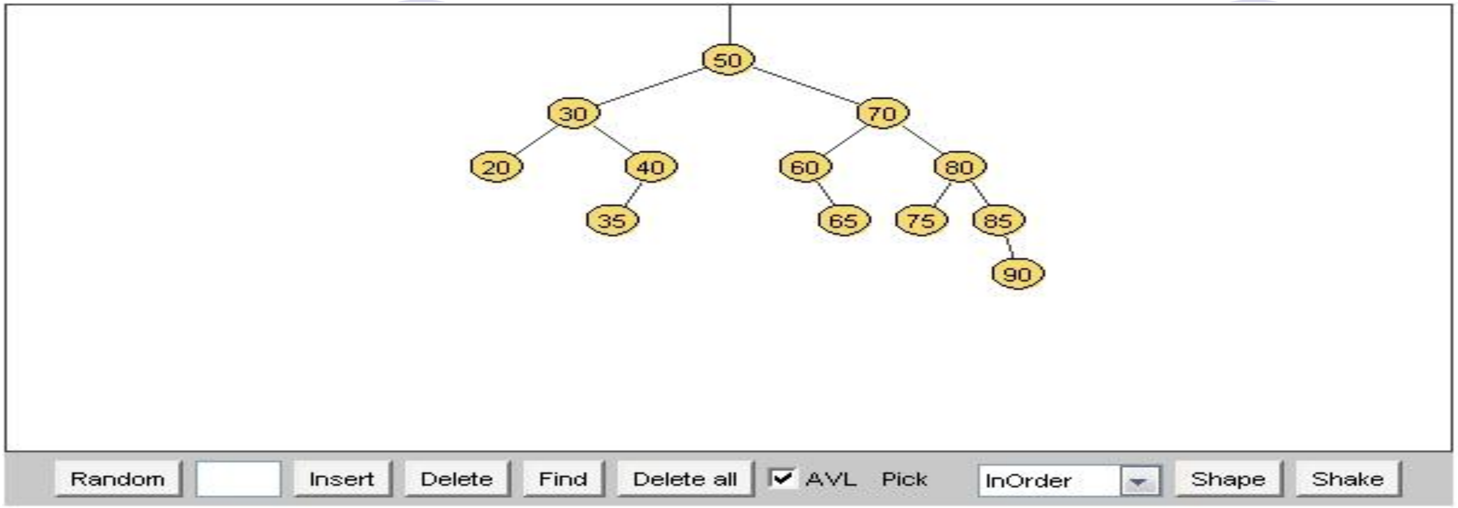
AVL tree

- **DoRight**

```
if (height(t->right) - height(t->left) == 2)
   if (x > t->right->element)
     t = singleRotateWithRight(t); // RR
   else
     t = doubleRotateWithRight(t); // RL
```

Demo

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



- You can insert, delete and locate nodes in the tree using control buttons.
- The data can be entered manually or randomly generated.
- By pressing <Insert> button only, you can quickly build a large tree.

The End

Any Questions?