Tutorial 5
Binary and AVL Trees in C

CSCI2100A/ESTR2102 Data Structures (2025 Spring)

Outline

- Quick Review on C Programming (Structures)
- Binary Tree
  - Concept Review
  - Insertion
  - Tree Traversal
  - Deletion
- Binary Search Tree
  - Concept Review
  - Insertion
  - Searching
  - Deletion
- AVL Tree
  - Concept Review
  - Single Rotation
  - Double Rotation

Quick Review on C Programming (Structures)

Structures

- We can define a structure in C like this:

```
struct canteen {
    char name[80];
    int num_employees;
    double profit;
};
```

- To save space, we usually put smaller data types first.
  - In general, `char` < `int` = `float` < `double`
- We can declare a variable of data type `struct canteen` like this:

```
struct canteen cc_can;
```

- We can access a member of `cc_can` like this:

```
cc_can.num_employees = 10;
```

Renaming Structures

- To get rid of the keyword `struct`, we can define an alias to the data type `struct canteen` like this:

```
typedef struct canteen Canteen;
```

- Then, we can declare variables of data type `struct canteen` using either its full name or its alias:

```
struct canteen uc_can;
Canteen na_can;
```

Pointers of Structures

- Sometimes we may want to declare a pointer variable of our own data type. If we use a pointer, we can access the member of the structure with the “->” operator.

```
Canteen cc_can = {
    "Chung Chi Tang Student Canteen",
    10,
    12345.678
};
Canteen *cc_ptr = &cc_can;

printf("%d\n", cc_can.num_employees);         10
printf("%d\n", cc_ptr->num_employees);        10
printf("%d\n", (*cc_ptr).num_employees);      10
```

Dynamic Memory for Structures

- To declare a variable of our own data type dynamically, we can use the `malloc()` function defined in `<stdlib.h>`.

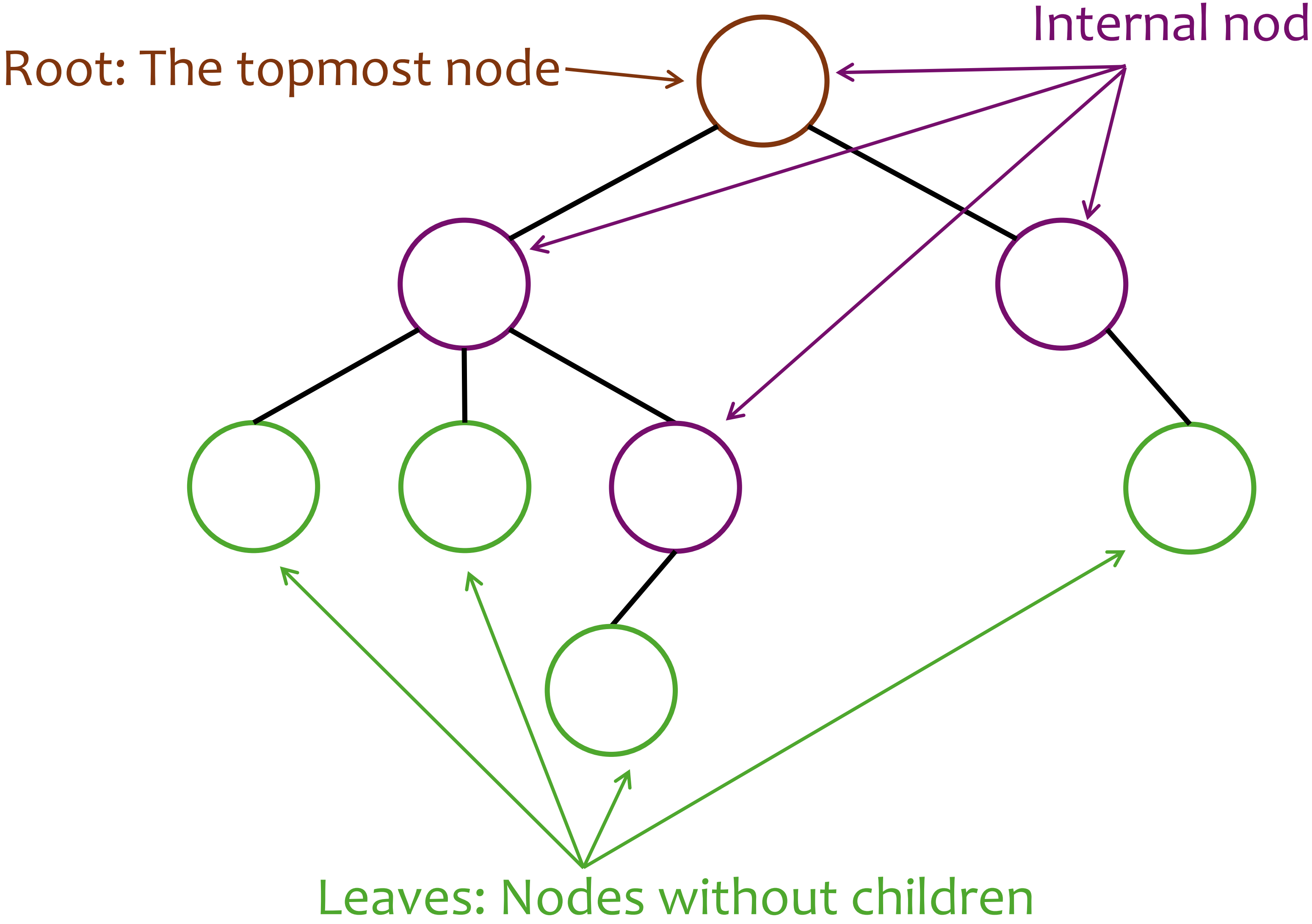
```
Canteen *uc_can = (Canteen *)malloc(sizeof(Canteen));
strcpy(uc_can->name, "Joyful Inn");
printf("%s\n", uc_can->name); // Joyful Inn
free(uc_can);
```

- Remember to `free` the memory after use.

Binary Tree

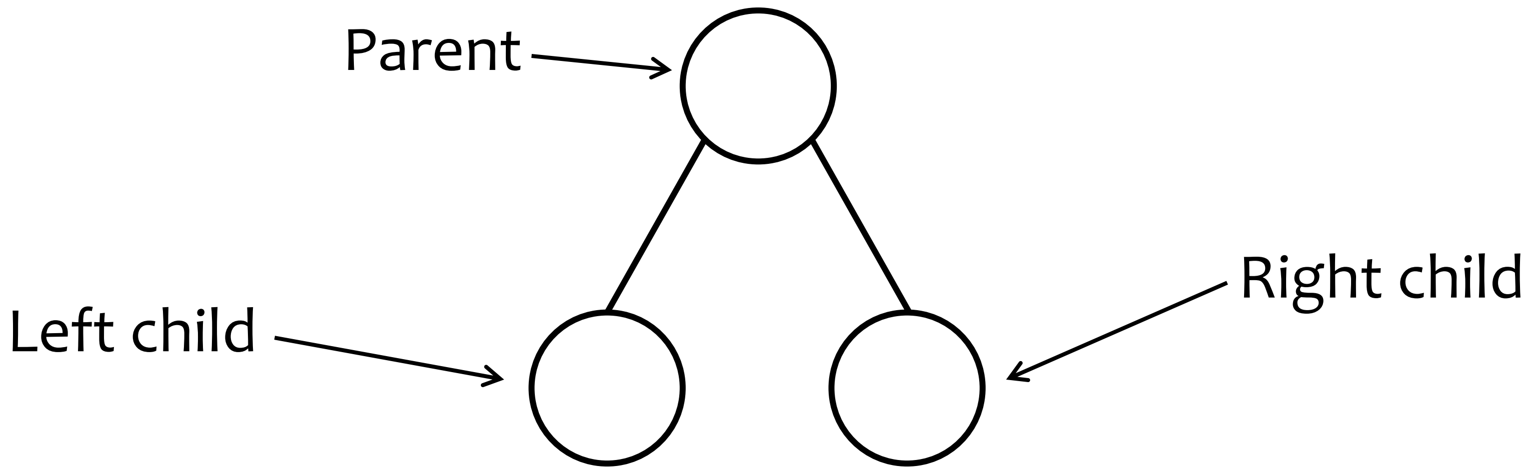
Concept Review

- A tree is a collection of nodes.



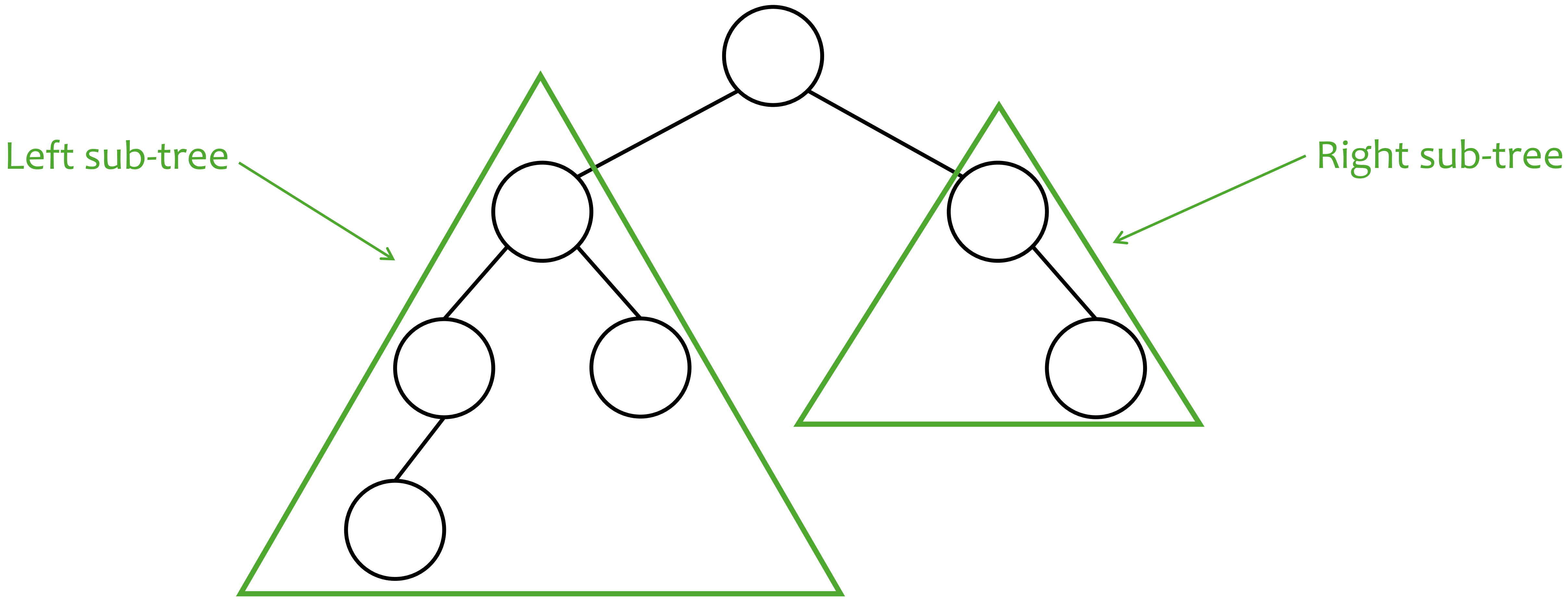
Concept Review

- A binary tree is a tree in which each node has **at most two** children.



Concept Review

- We can treat each child of a root as the root of a sub-tree.



Tree Node

- Based on the recursive structure (children of a root are roots of sub-trees), we can implement a tree node like this:

```
typedef char ElementType; // Assume elements are char’s
typedef struct TreeNode *TreePtr;

struct TreeNode {
    ElementType element;
    TreePtr left;
    TreePtr right;
};
```

Insertion

- To create a tree node, we can use the `malloc()` function.

```
TreePtr create_node(ElementType element) {
    TreePtr new_node = (TreePtr)malloc(
        sizeof(struct TreeNode)
    );
    new_node->element = element;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}
```

Insertion

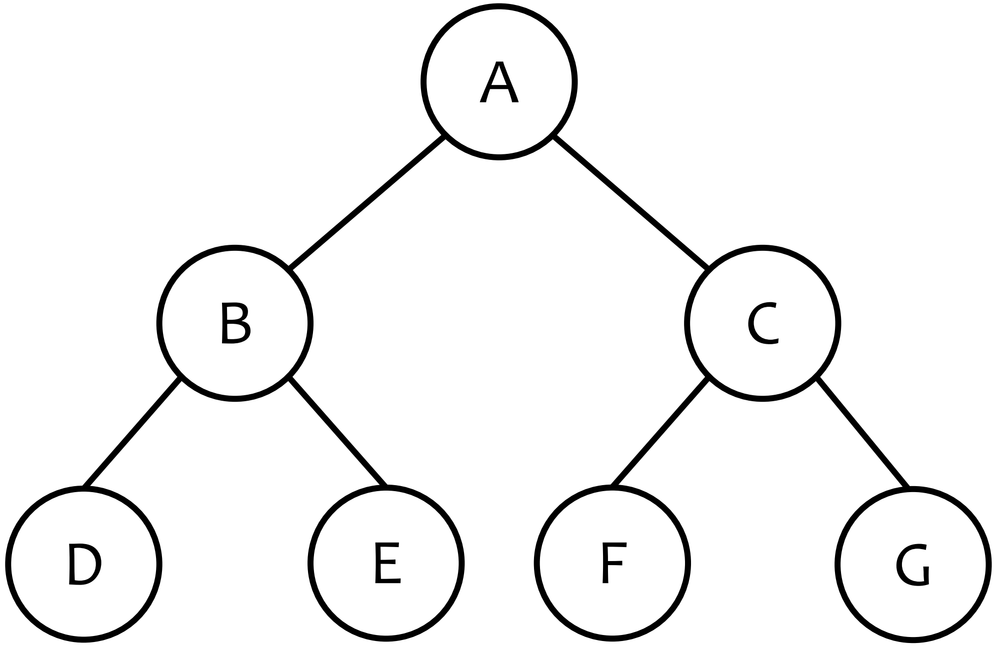
- For insertion, we need to specify the exact location.

```
TreePtr insert_node(ElementType element, TreePtr parent,
int isRight) {
    TreePtr child = create_node(element);
    if (parent != NULL) {
        if (isRight) {
            parent->right = child;
        } else {
            parent->left = child;
        }
    }
    return child;
}
```

Insertion

- Example: Create the following binary tree.

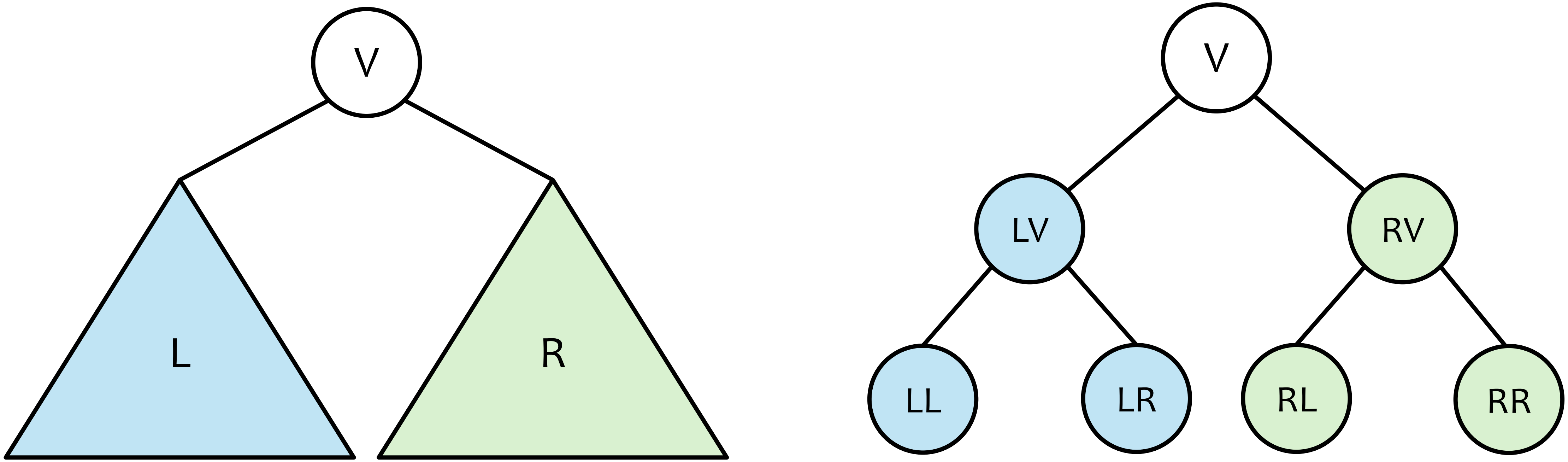
```
TreePtr binary_tree = insert_node('A', NULL, 0);
TreePtr subtree_l = insert_node('B', binary_tree, 0);
TreePtr subtree_r = insert_node('C', binary_tree, 1);
insert_node('D', subtree_l, 0);
insert_node('E', subtree_l, 1);
insert_node('F', subtree_r, 0);
insert_node('G', subtree_r, 1);
```



Tree Traversal

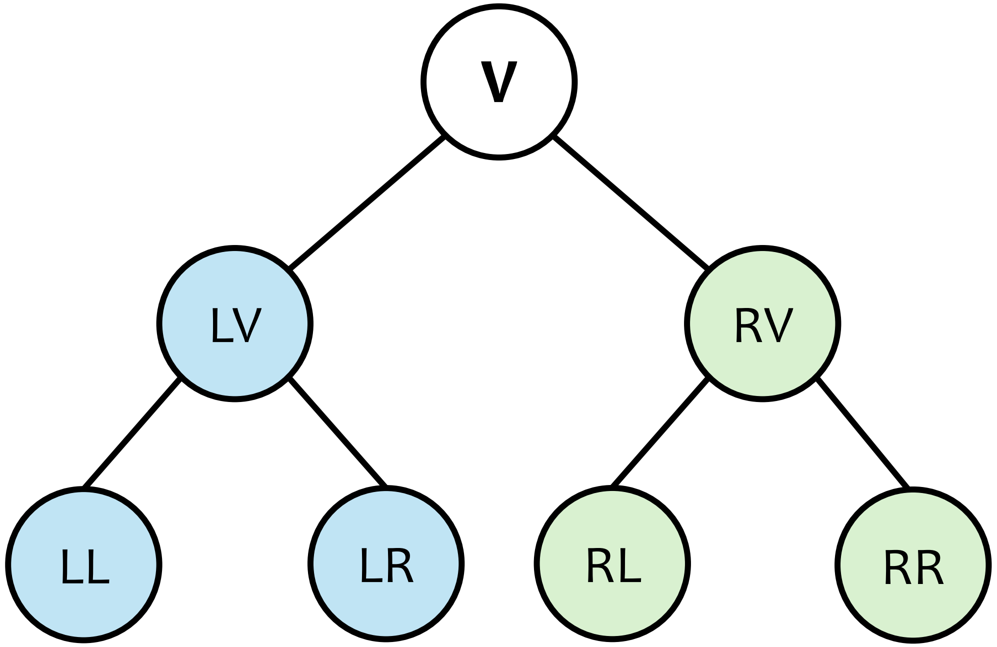
- With the recursive structure of binary trees, we can visit all the nodes in a binary tree by
  - Visiting the root node (V)
  - Visiting all the nodes in the left sub-tree (L)
  - Visiting all the nodes in the right sub-tree (R)

Recursion



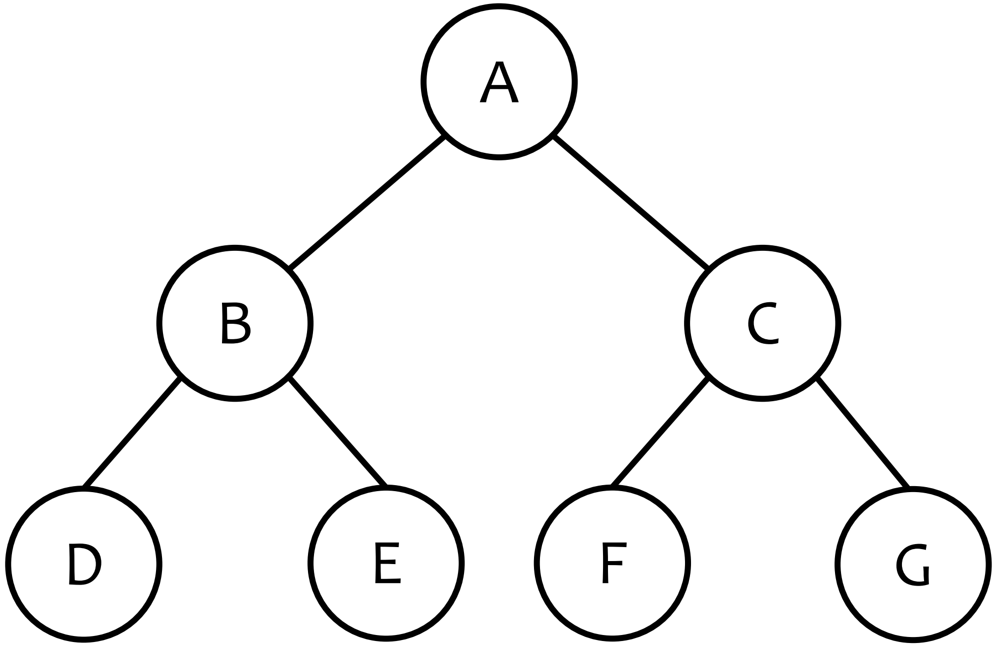
Tree Traversal

- There are 3 main traversal orders for a binary tree
  - Preorder (VLR)
    - **Root** -> Left -> Right
    - **V** -> LV -> LL -> LR -> RV -> RL -> RR
  - Inorder (LVR)
    - Left -> **Root** -> Right
    - LL -> LV -> LR -> **V** -> RL -> RV -> RR
  - Postorder (LRV)
    - Left -> Right -> **Root**
    - LL -> LR -> LV -> RL -> RR -> RV -> **V**



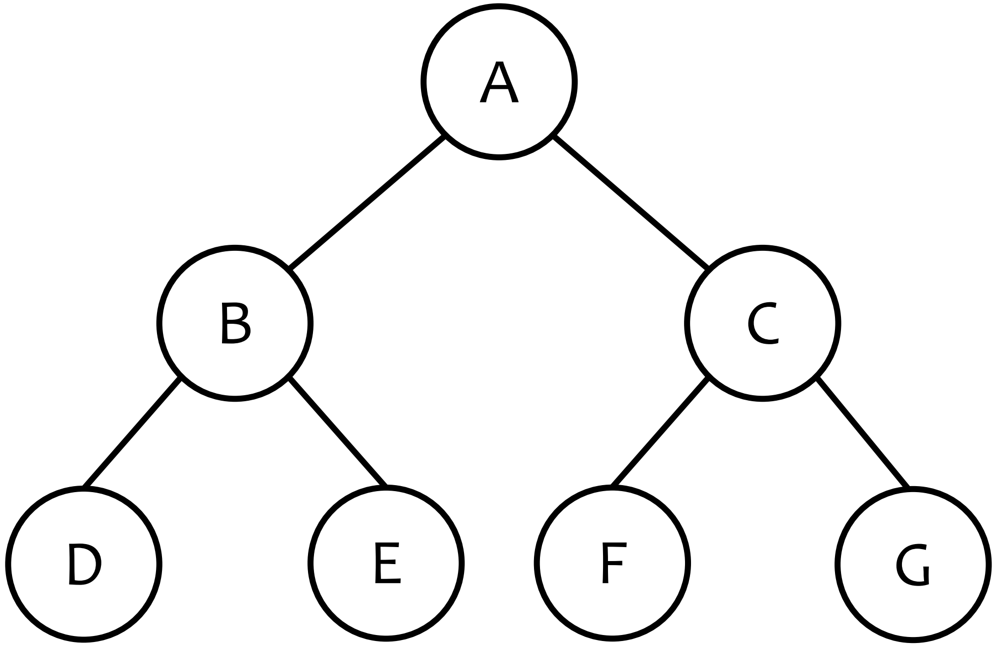
Tree Traversal: Preorder (VLR)

```c
#define PRINT_FORMAT "%c " // Tree node print format

void print_node(TreePtr tree) {
    printf(PRINT_FORMAT, tree->element);
}

void preorder(TreePtr root) {
    if (root != NULL) {
        print_node(root); // V
        preorder(root->left); // L
        preorder(root->right); // R
    }
}

// A B D E C F G
```



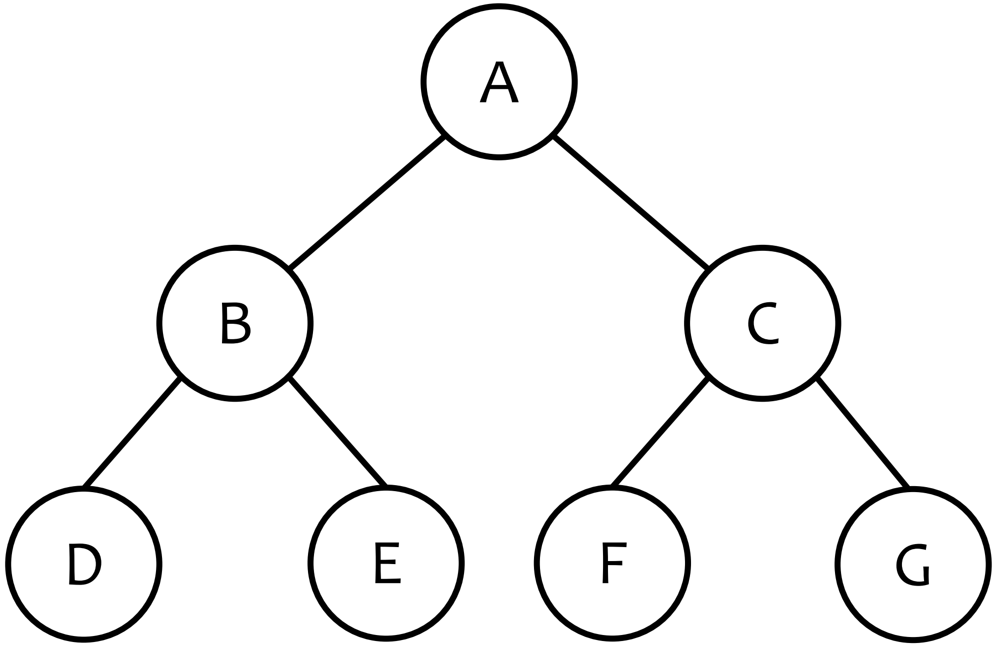
Tree Traversal: Inorder (LVR)

```
void inorder(TreePtr root) {
    if (root != NULL) {
        inorder(root->left); // L
        print_node(root); // V
        inorder(root->right); // R
    }
}

// D B E A F C G
```



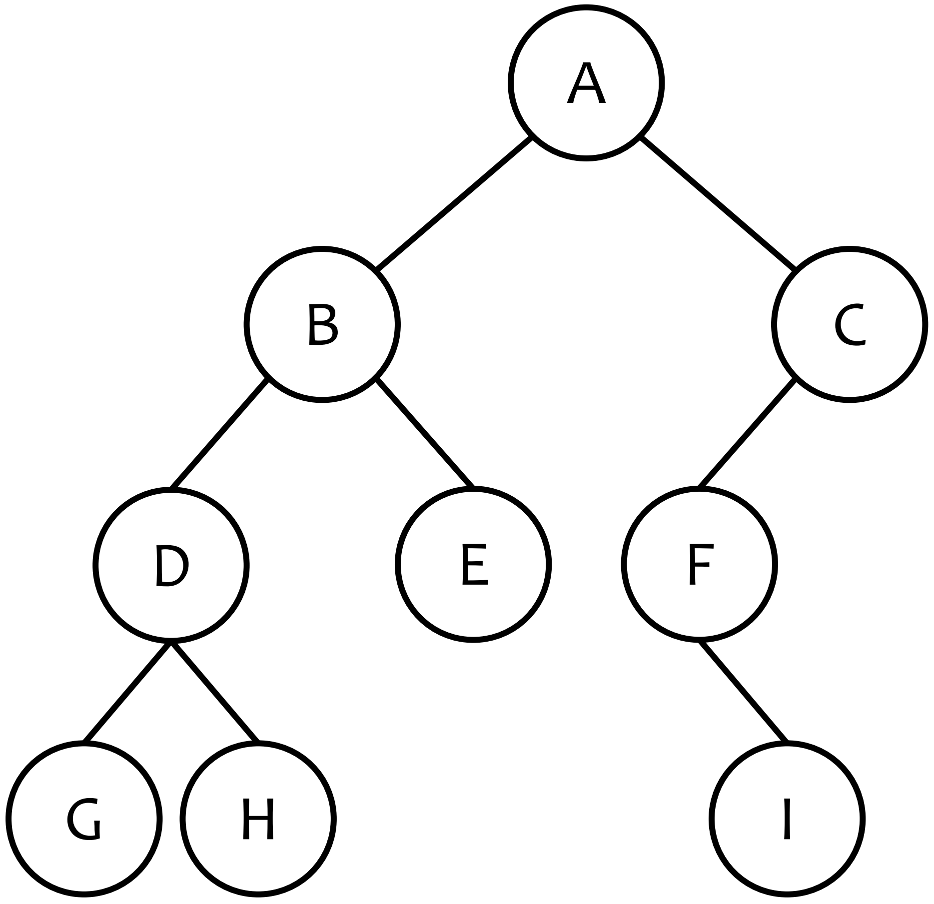
Tree Traversal: Postorder (LRV)

```
void postorder(TreePtr root) {
    if (root != NULL) {
        postorder(root->left); // L
        postorder(root->right); // R
        print_node(root); // V
    }
}

// D E B F G C A
```



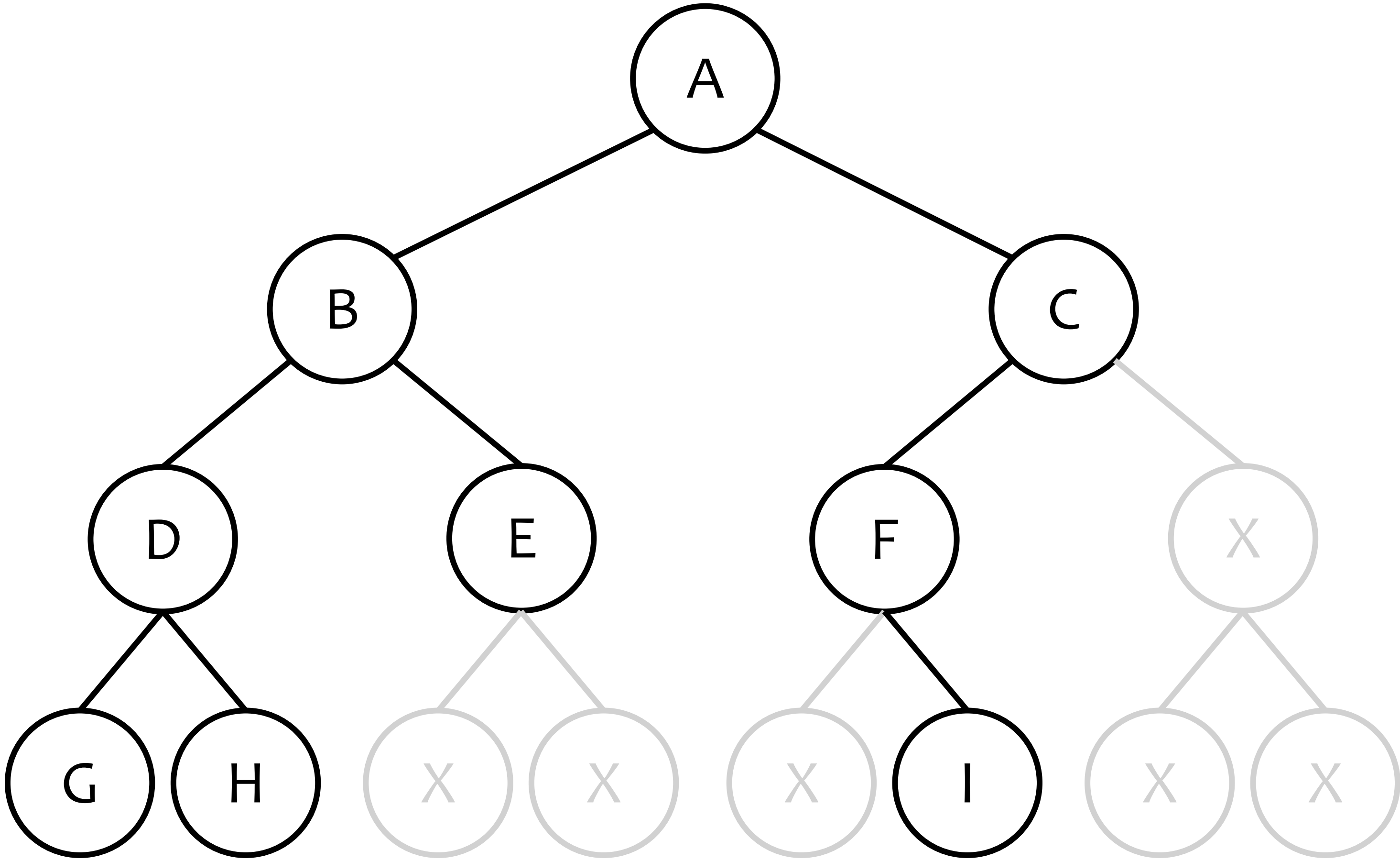
Exercise: Tree Traversal

- Consider the following binary tree. Write down the sequence of the nodes visited when we perform
  - a preorder traversal
  - an inorder traversal
  - a postorder traversal



Exercise: Tree Traversal (Hint)

- “Pad” the tree:



Deletion

- To remove a tree node, we can use the `free()` function.
- Use **postorder** traversal to remove all the descendants before removing the root node.

```
void destroy_node(TreePtr node) {
    if (node != NULL) {
        destroy_node(node->left); // L
        destroy_node(node->right); // R
        free(node); // V
    }
}
```

Deletion

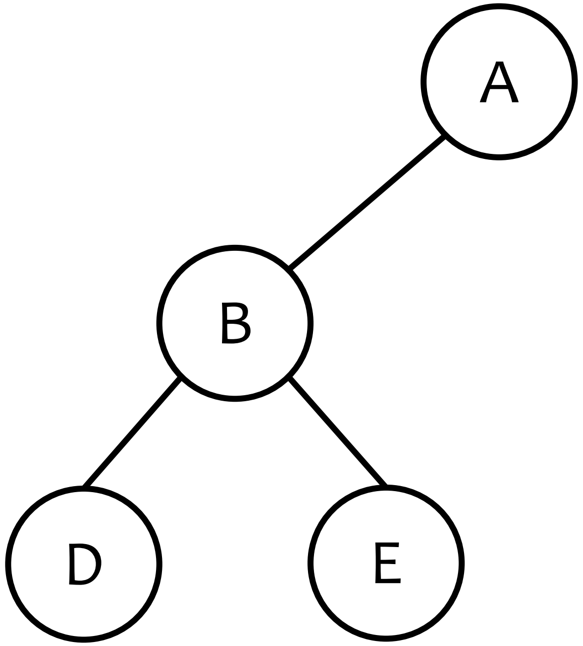
- In this implementation, we delete the left or the right sub-tree.

```
void delete_node(TreePtr parent, int isRight) {
    if (parent->left != NULL || parent->right != NULL) {
        if (isRight) {
        destroy_node(parent->right);
        parent->right = NULL;
        } else {
            destroy_node(parent->left);
            parent->left = NULL;
        }
    } else {
        destroy_node(parent);
        parent = NULL;
    }
}
```

Deletion

- Example: Remove the nodes C, F and G.

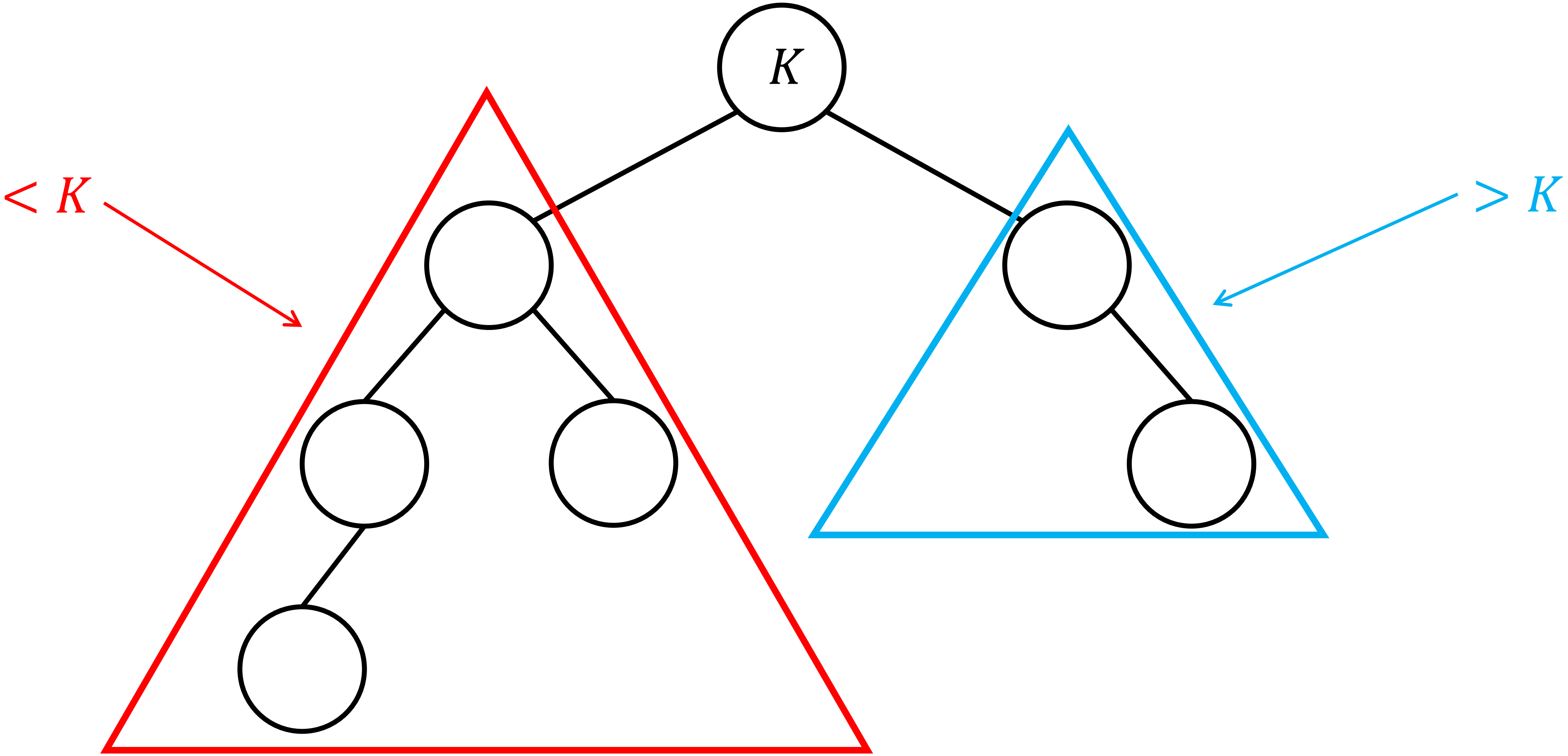
```
delete_node(binary_tree, 1);

preorder(binary_tree);
// A B D E

inorder(binary_tree);
// D B E A

postorder(binary_tree);
// D E B A
```



Binary Search Tree

Concept Review

- A binary search tree is a binary tree in which for every node with key value K:



Insertion

- To insert a tree node, instead of specifying the exact position, we can select a position based on the binary search tree property (L < V < R).

```
TreePtr insert_node(ElementType element, TreePtr tree) {
if (tree == NULL) {
        tree = create_node(element);
    } else if (element < tree->element) {
        tree->left = insert_node(element, tree->left);
    } else if (element > tree->element) {
        tree->right = insert_node(element, tree->right);
    }
    return tree;
}
```

Searching

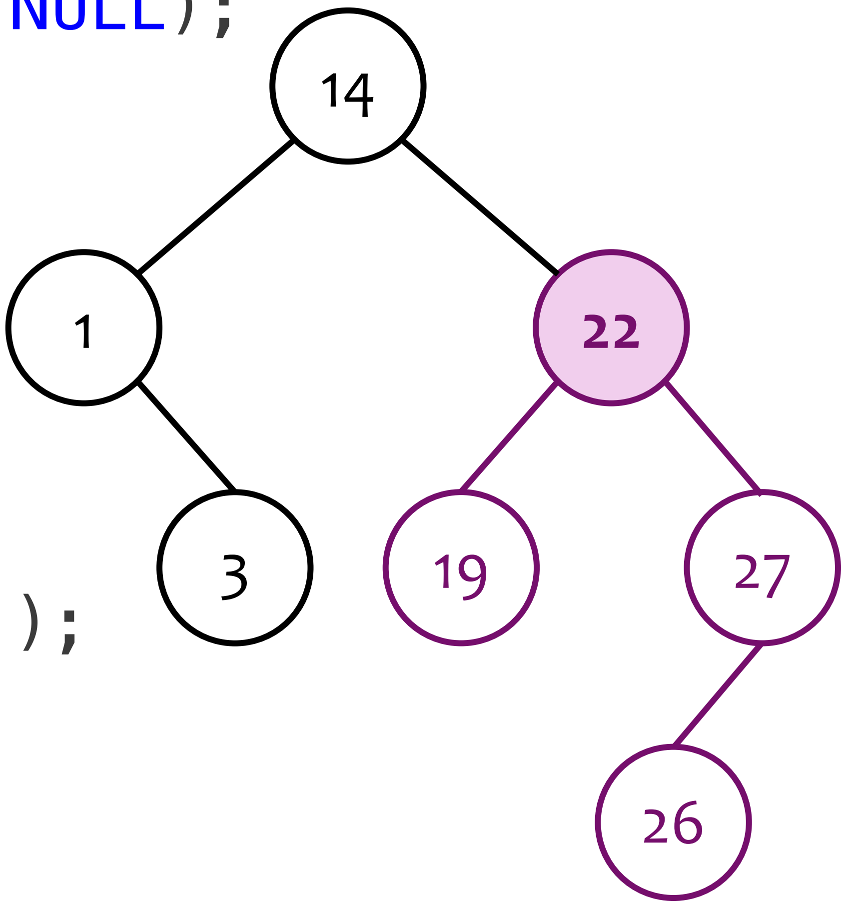
- We can search whether an element exists in the binary search tree using a similar algorithm.

```
TreePtr search_node(ElementType target, TreePtr tree) {
    if (tree != NULL) {
        if (target < tree->element) {
            return search_node(target, tree->left);
        } else if (target > tree->element) {
            return search_node(target, tree->right);
        } else {
            return tree;
        }
    } else {
        return NULL;
    }
}
```

Insertion and Searching

- Example: Create the following binary search tree. Then, find the postorder of the sub-tree with root = 22.

```
TreePtr search_tree = insert_node(14, NULL);
insert_node(1, search_tree);
insert_node(22, search_tree);
insert_node(3, search_tree);
insert_node(19, search_tree);
insert_node(27, search_tree);
insert_node(26, search_tree);

postorder(search_node(22, search_tree));
putchar('\n');
// 19 26 27 22
```



Deletion

- Based on the definition of binary trees, a node can have 0-2 children.

```
TreePtr delete_node(ElementType target, TreePtr tree) {
    TreePtr tmp;
    if (tree != NULL) {
        if (target < tree->element) {
            tree->left = delete_node(target, tree->left);
        } else if (target > tree->element) {
            tree->right = delete_node(target, tree->right);
        } else {
            // Deletion
            if (tree->left != NULL && tree->right != NULL) {
                ... // 2 children
            } else {
                ... // 0-1 child
            }
        }
    }
    return tree;
}
```

Deletion

- To delete a tree node without children, remove it directly.
- To delete a tree node with one child, replace it by its child.

```
tmp = tree; // Node to be deleted
if (tree->left != NULL) {
    tree = tree->left; // Only left child
} else if (tree->right != NULL) {
    tree = tree->right; // Only right child
} else {
    tree = NULL; // No children
}
free(tmp);
```

Deletion

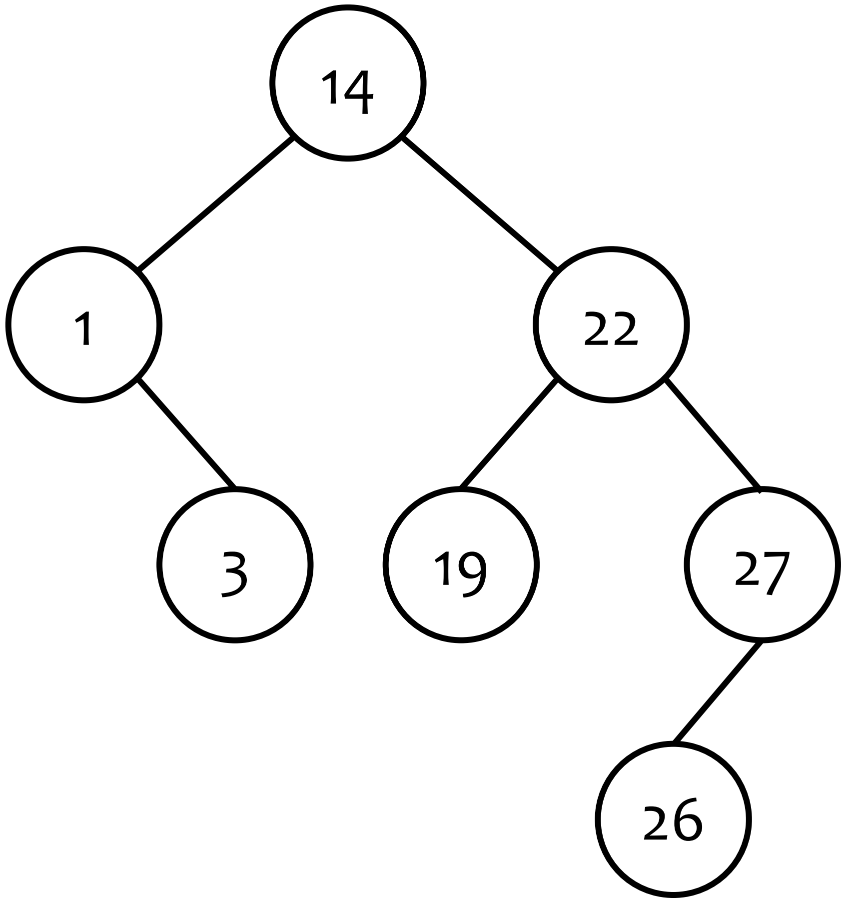
- To delete a tree node with two children, replace it by the node with the **smallest value** in its **right sub-tree** (or the largest value in its left sub-tree).
  - In binary search trees, the **leftmost node** always contains the smallest value.

```
tmp = tree->right; // Right sub-tree
while (tmp->left != NULL) {
    tmp = tmp->left;
} // Find smallest
tree->element = tmp->element;
tree->right = delete_node(tree->element, tree->right);
```

Exercise: Binary Search Tree Operations

- Suppose the variable `search_tree` points to the following binary search tree. What is the output of the following code snippet?

```
delete_node(22, search_tree);
insert_node(22, search_tree);
preorder(search_tree);
putchar('\n');
inorder(search_tree);
putchar('\n');
postorder(search_tree);
putchar('\n');
```

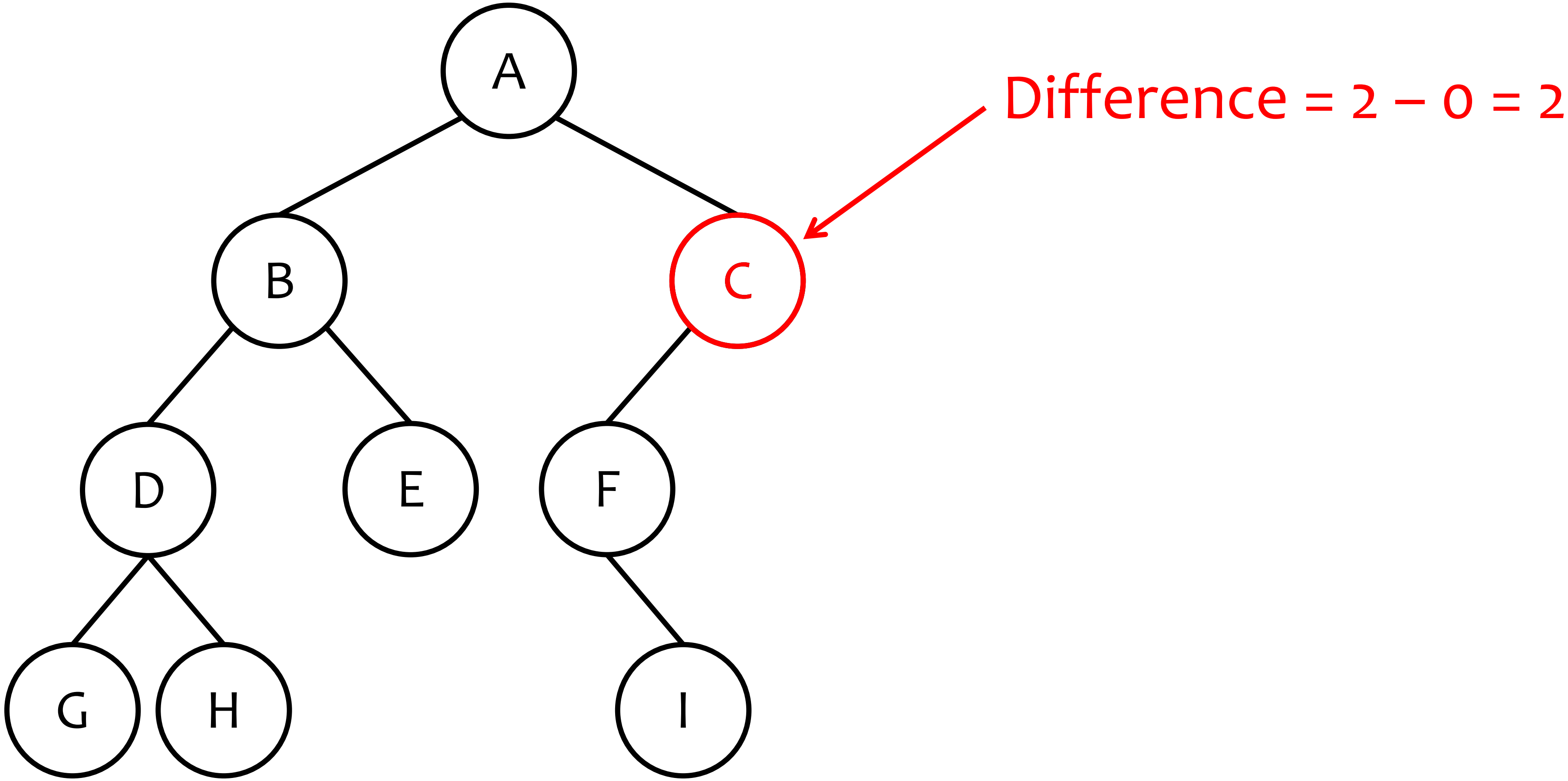


AVL Tree

Concept Review

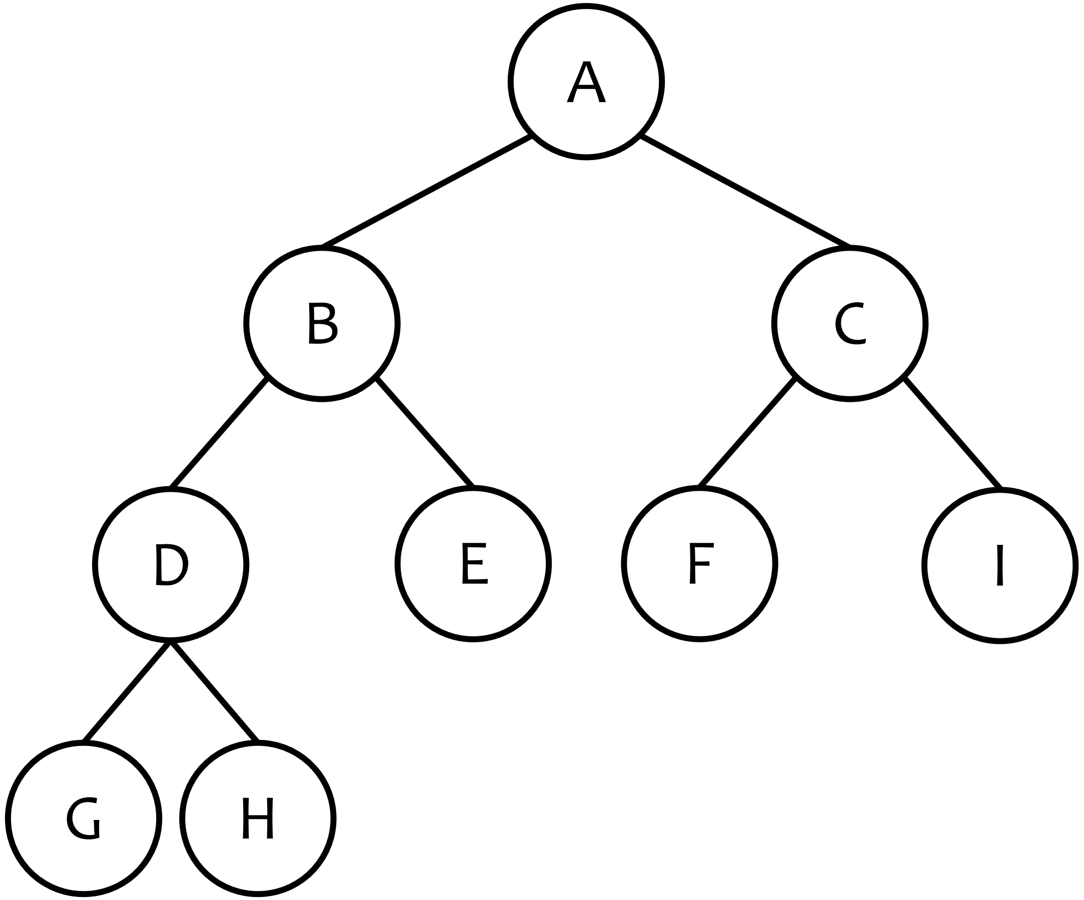
- An AVL tree is a **binary search tree** in which for every node in the tree, the heights of the left and right sub-trees **differ by at most 1**.
- Exercise: Is this an AVL tree?

Answer: No



Concept Review

- An AVL tree is a **binary search tree** in which for every node in the tree, the heights of the left and right sub-trees **differ by at most 1**.
- Example:



Tree Node

- To find the height of a node in a binary (search) tree, we can use the following function.

```
int height(TreePtr tree) {
    if (tree != NULL) {
        int left_height = height(tree->left);
        int right_height = height(tree->right);
        if (left_height > right_height) {
            return left_height + 1;
        } else {
            return right_height + 1;
        }
    } else {
        return -1; // Empty tree
    }
}
```

Tree Node

- The time complexity of the function `height()` is $O(n)$, where n is the number of nodes in the tree, because each node is visited once.
- We sacrifice a little bit of memory to store the height of each node for efficient comparison of heights.

```
typedef struct AVLNode *TreePtr;

struct AVLNode {
    ElementType element;
    int height;
    TreePtr left;
    TreePtr right;
};
```

Tree Node

- We update the function for creating a tree node accordingly.

```
TreePtr create_node(ElementType element) {
    TreePtr new_node = (TreePtr)malloc(
        sizeof(struct AVLNode)
    );
    new_node->element = element;
    new_node->height = 0; // Leaf node
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}
```

Insertion

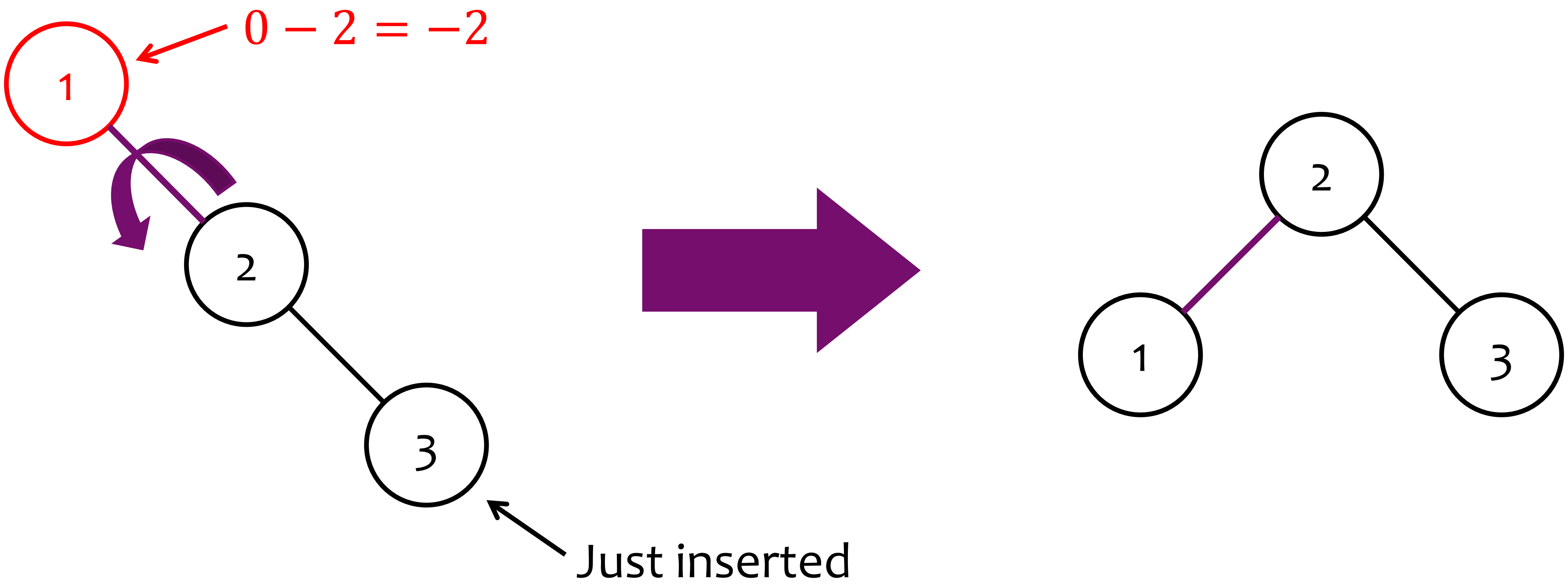
- After insertion of an element, we must check whether the AVL tree property (difference ≤ 1) is preserved. If there is a violation, we need to fix it by re-arranging the nodes.

- 4 possible cases of imbalance
  - **RR** case: Insertion to the **r**ight child's **r**ight sub-tree → Left rotation
  - **LL** case: Insertion to the **l**eft child's **l**eft sub-tree → Right rotation
  - **LR** case: Insertion to the **l**eft child's **r**ight sub-tree → Left-right rotation
  - **RL** case: Insertion to the **r**ight child's **l**eft sub-tree → Right-left rotation

- For single rotation (left or right), 3 sub-trees are involved. For double rotation (left-right or right-left), 4 sub-trees are involved.

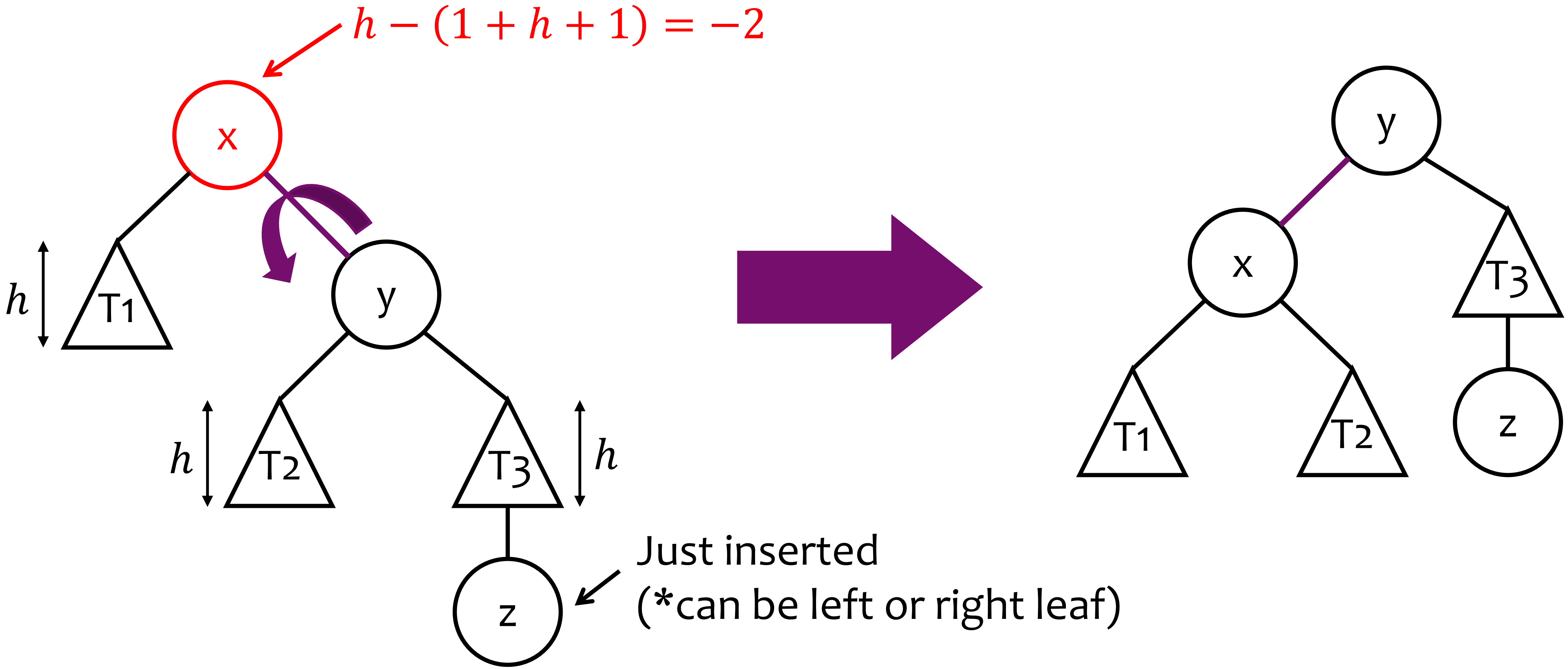
Single Rotation: Left Rotation

- RR case: Insertion to the right sub-tree of the right child



Single Rotation: Left Rotation

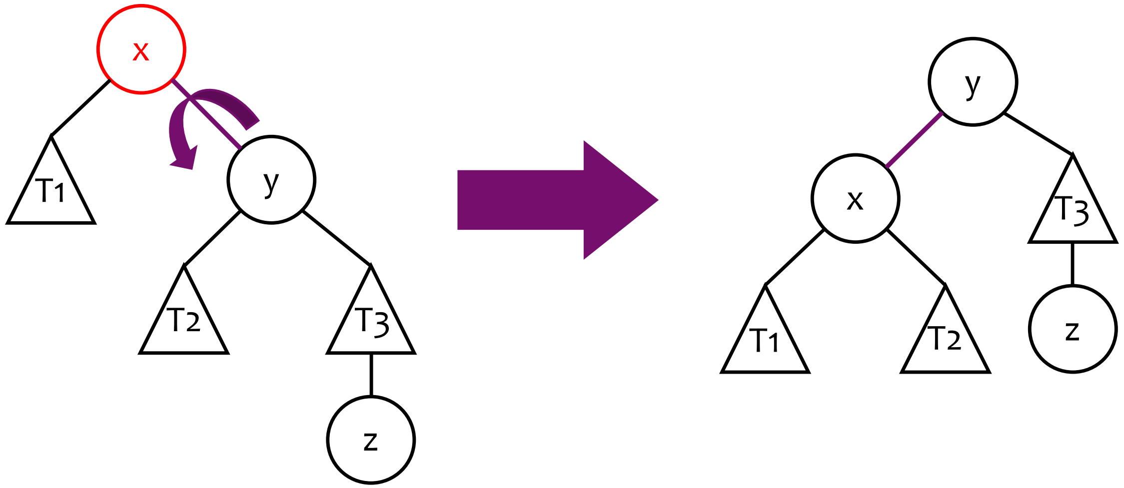
- RR case: Insertion to the right sub-tree of the right child



Single Rotation: Left Rotation

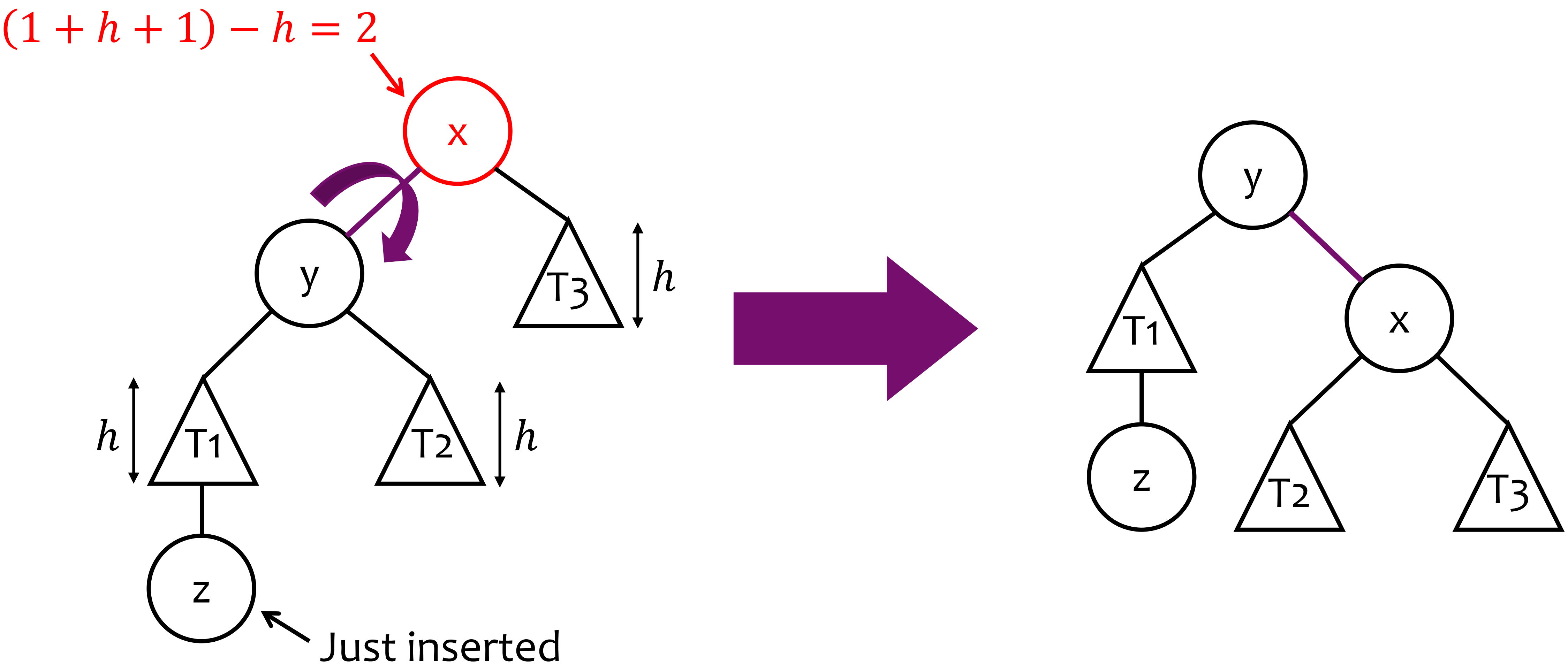
```
TreePtr left_rotation(TreePtr x) {
    // Swap pointers
    TreePtr y = x->right;
    x->right = y->left; // T2
    y->left = x;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(x->height, height(y->right)) + 1;

    return y;
}
```

```
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```



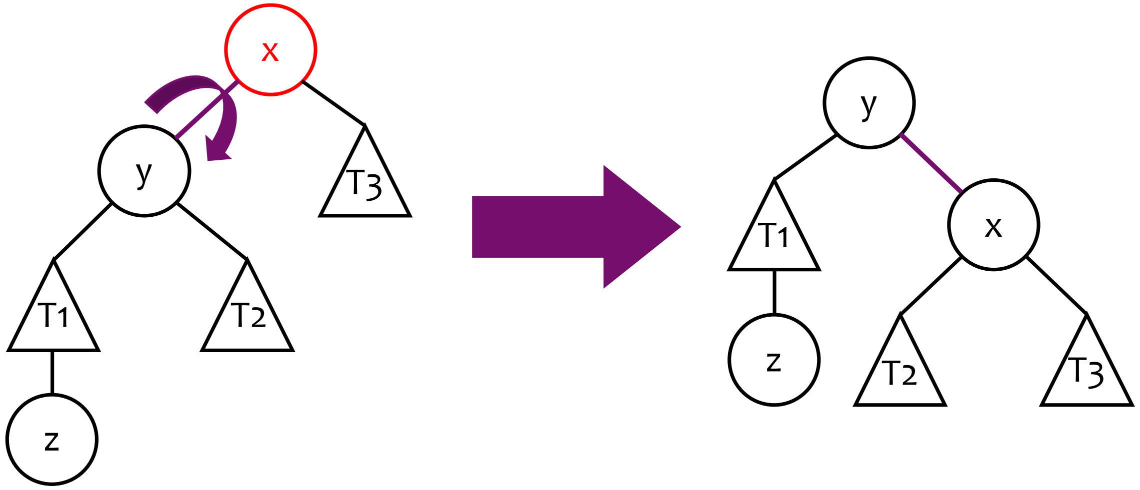
Single Rotation: Right Rotation

- LL case: Insertion to the left sub-tree of the left child



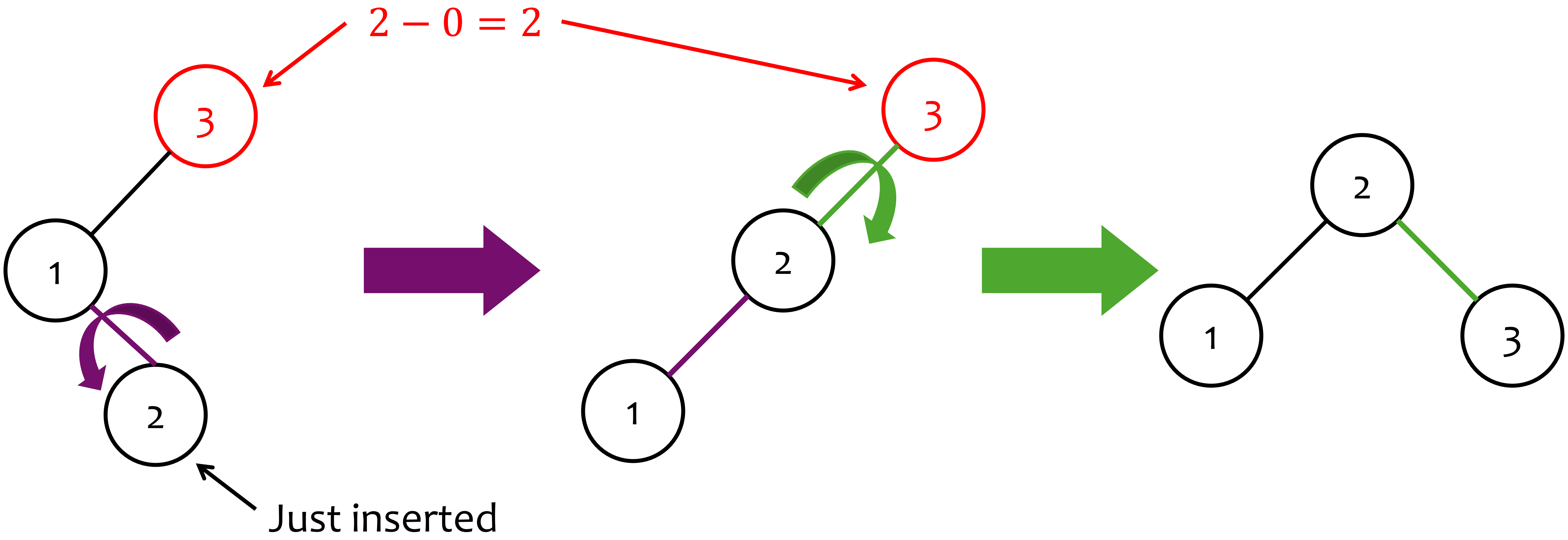
Single Rotation: Right Rotation

```
TreePtr right_rotation(TreePtr x) {
    // Swap pointers
    TreePtr y = x->left;
    x->left = y->right; // T2
    y->right = x;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), x->height) + 1;

    return y;
}
```



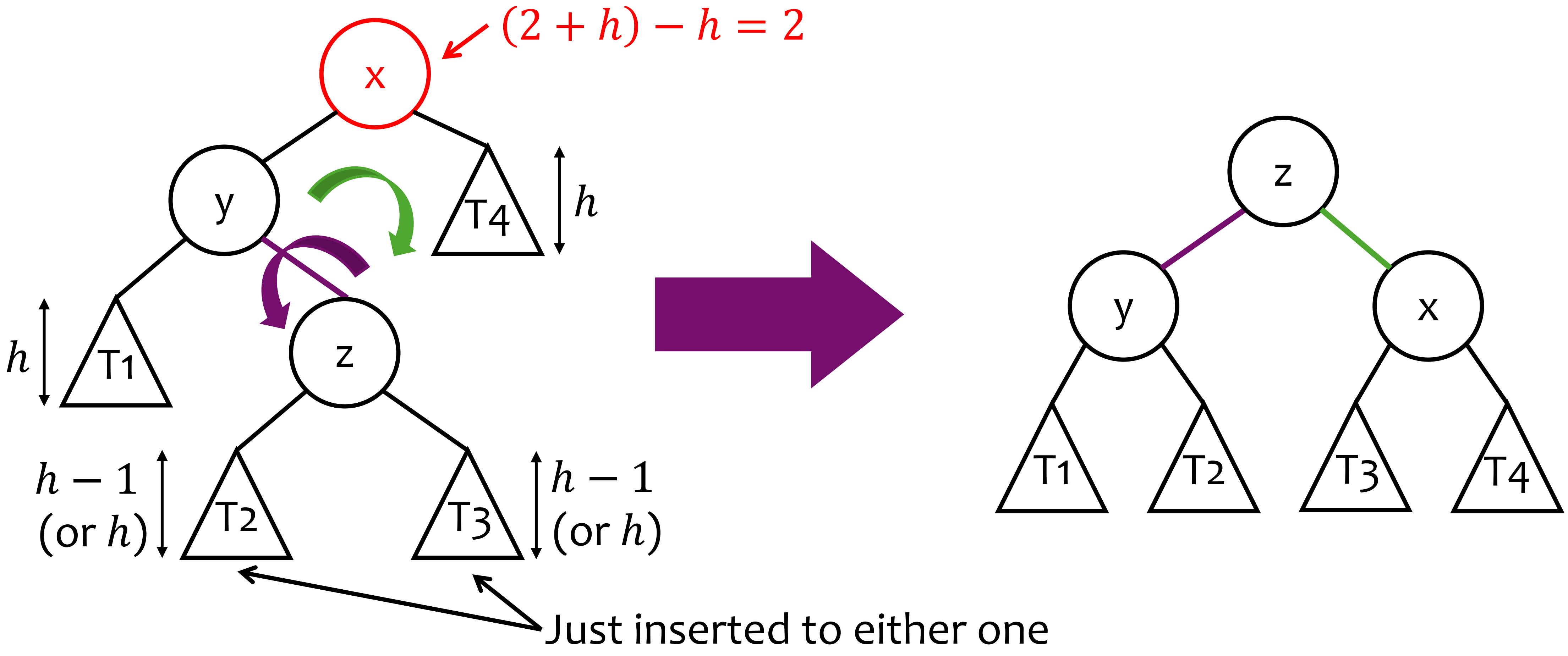
Double Rotation: Left-right Rotation

- LR case: Insertion to the right sub-tree of the left child
  - Step 1: Left rotation of the left child
  - Step 2: Right rotation



Double Rotation: Left-right Rotation

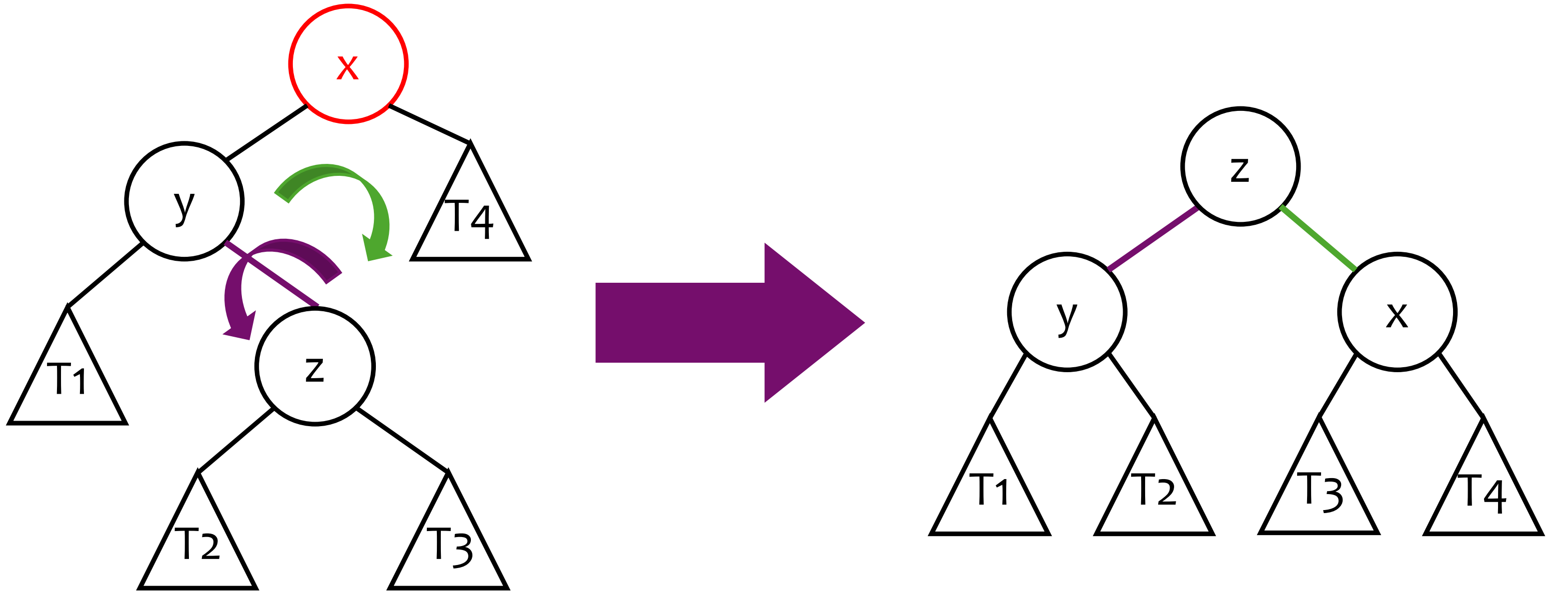
- LR case: Insertion to the right sub-tree of the left child
  - Step 1: Left rotation of the left child
  - Step 2: Right rotation



Double Rotation: Left-right Rotation

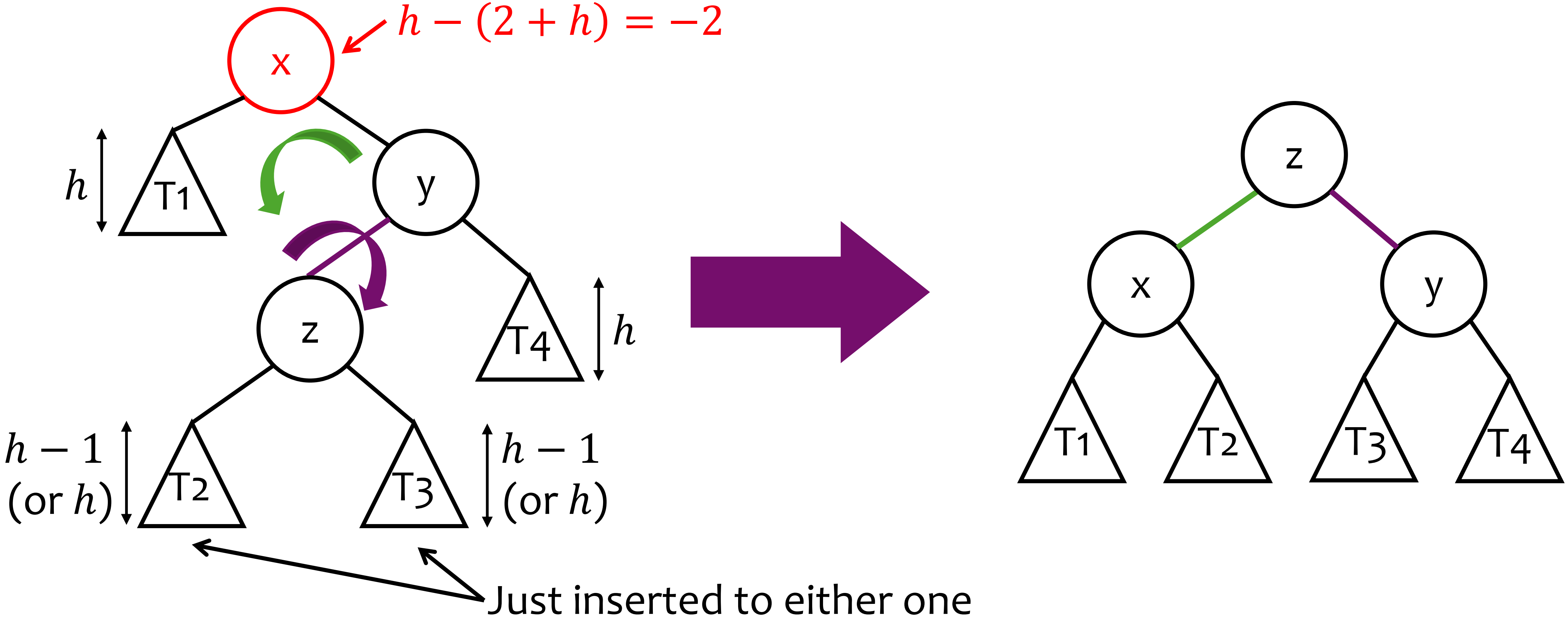
- We can easily implement this by using the functions `left_rotation()` and `right_rotation()`.

```
TreePtr left_right_rotation(TreePtr x) {
    x->left = left_rotation(x->left); // Step 1
    return right_rotation(x); // Step 2
}
```



Double Rotation: Right-left Rotation

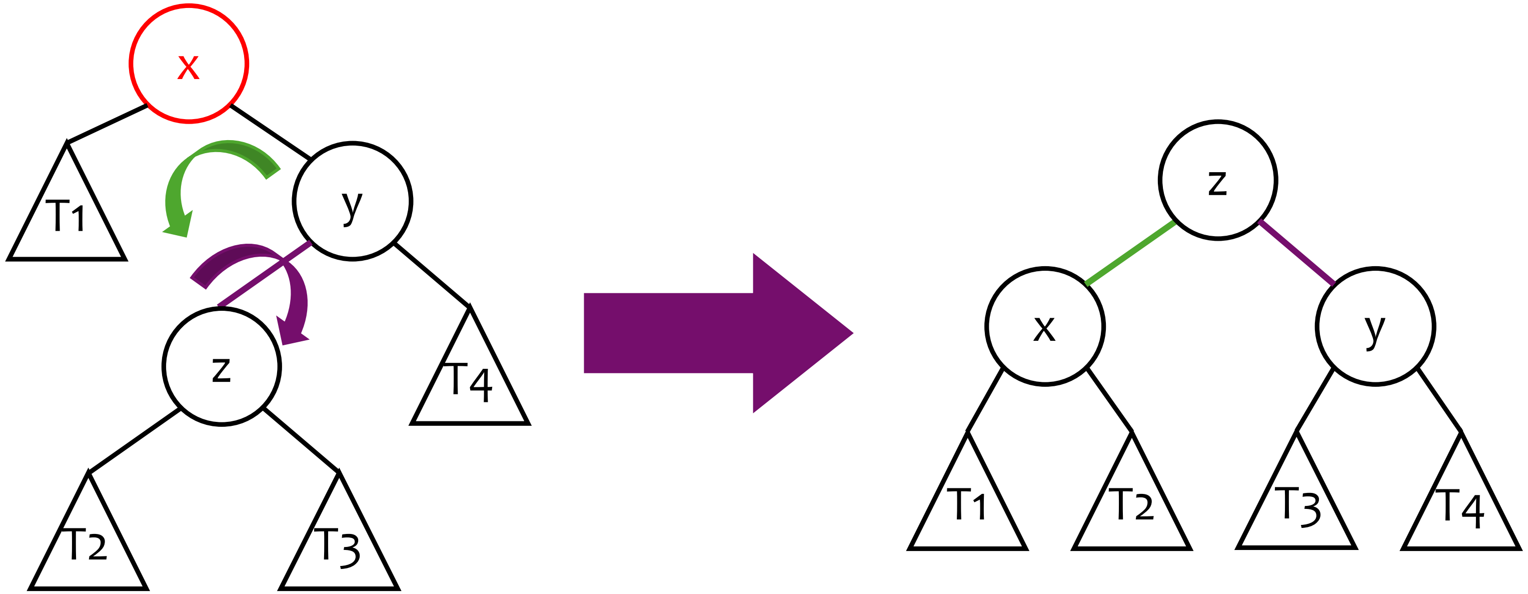
- RL case: Insertion to the left sub-tree of the right child
  - Step 1: Right rotation of the right child
  - Step 2: Left rotation



Double Rotation: Right-left Rotation

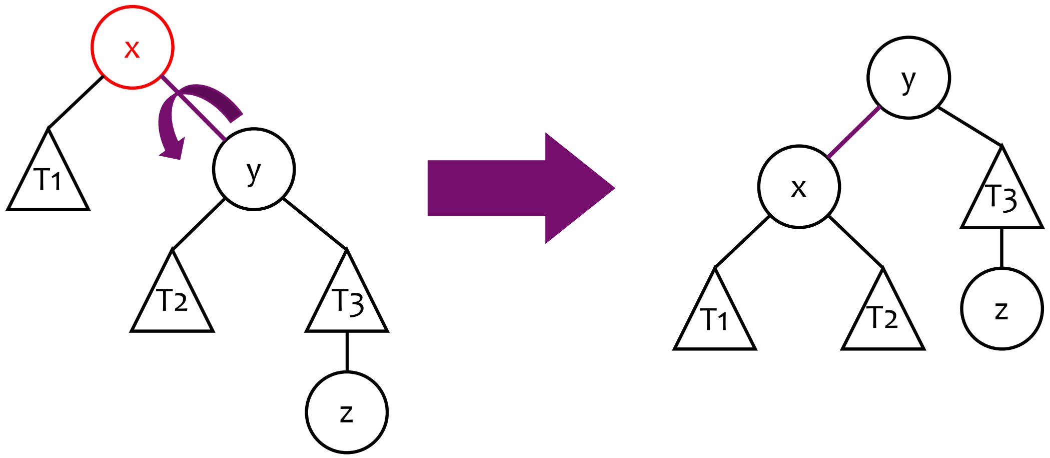
- Similarly, we implement this by using the functions `left_rotation()` and `right_rotation()`.

```
TreePtr right_left_rotation(TreePtr x) {
    x->right = right_rotation(x->right); // Step 1
    return left_rotation(x); // Step 2
}
```

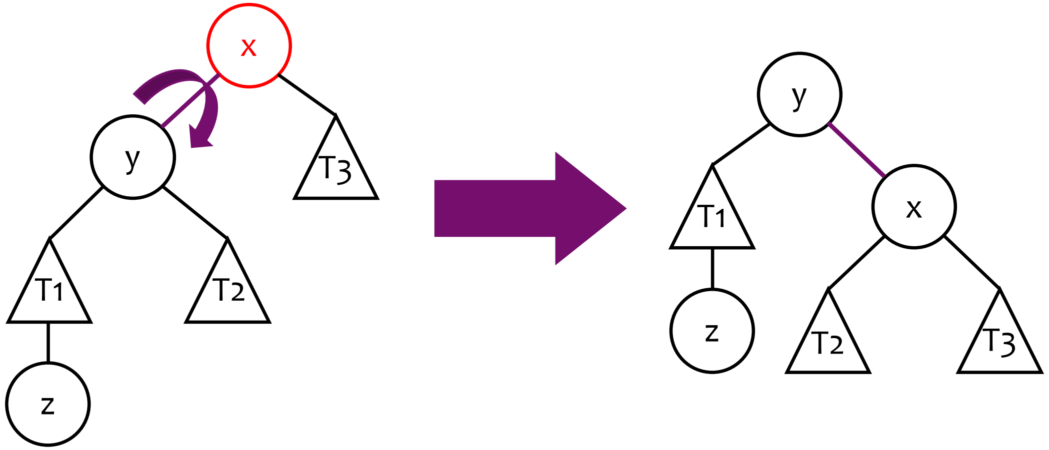


Rotation: A Summary

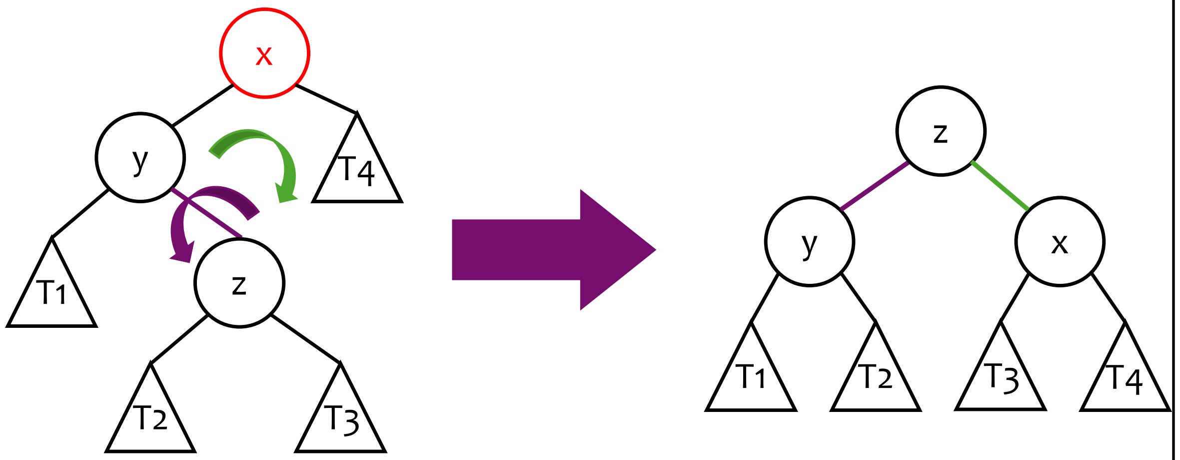
Case 1: Left Rotation (RR)



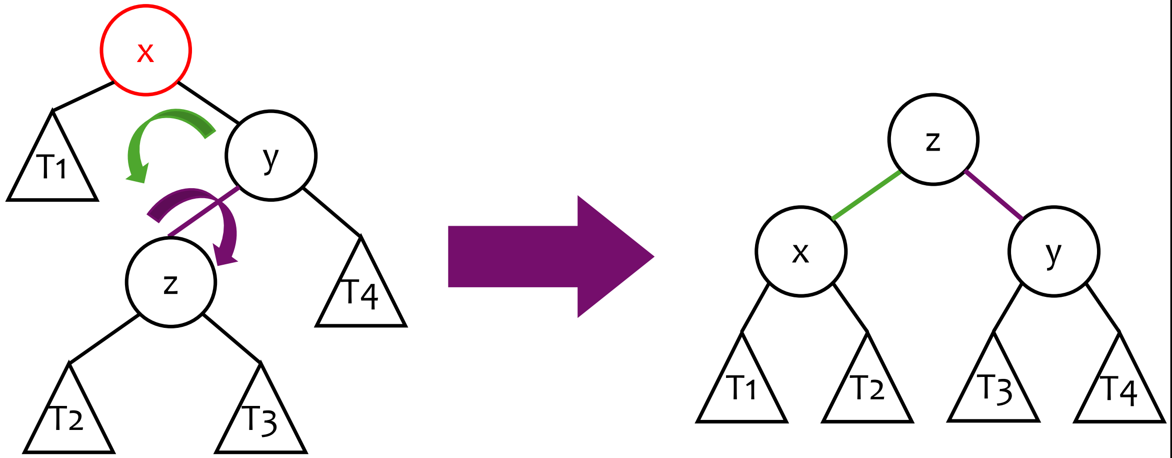
Case 2: Right Rotation (LL)



Case 3: Left-right Rotation (LR)



Case 4: Right-left Rotation (RL)



Insertion

- First, we identify the **lowest** node which violates the AVL tree property.
- Then, we perform the corresponding rotation.
  - If we need to add a node to the left child, it is either case 2 or case 3.

```
tree->left = insert_node(element, tree->left);
if (height(tree->left) - height(tree->right) == 2) {
    if (element < tree->left->element) {
        tree = right_rotation(tree); // Case 2 (LL)
    } else {
        tree = left_right_rotation(tree); // Case 3 (LR)
    }
}
```

Insertion

- First, we identify the **lowest** node which violates the AVL tree property.
- Then, we perform the corresponding rotation.
  - If we need to add a node to the right child, it is either case 1 or case 4.

```
tree->right = insert_node(element, tree->right);
if (height(tree->right) - height(tree->left) == 2) {
    if (element > tree->right->element) {
        tree = left_rotation(tree); // Case 1 (RR)
    } else {
        tree = right_left_rotation(tree); // Case 4 (RL)
    }
}
```

Insertion

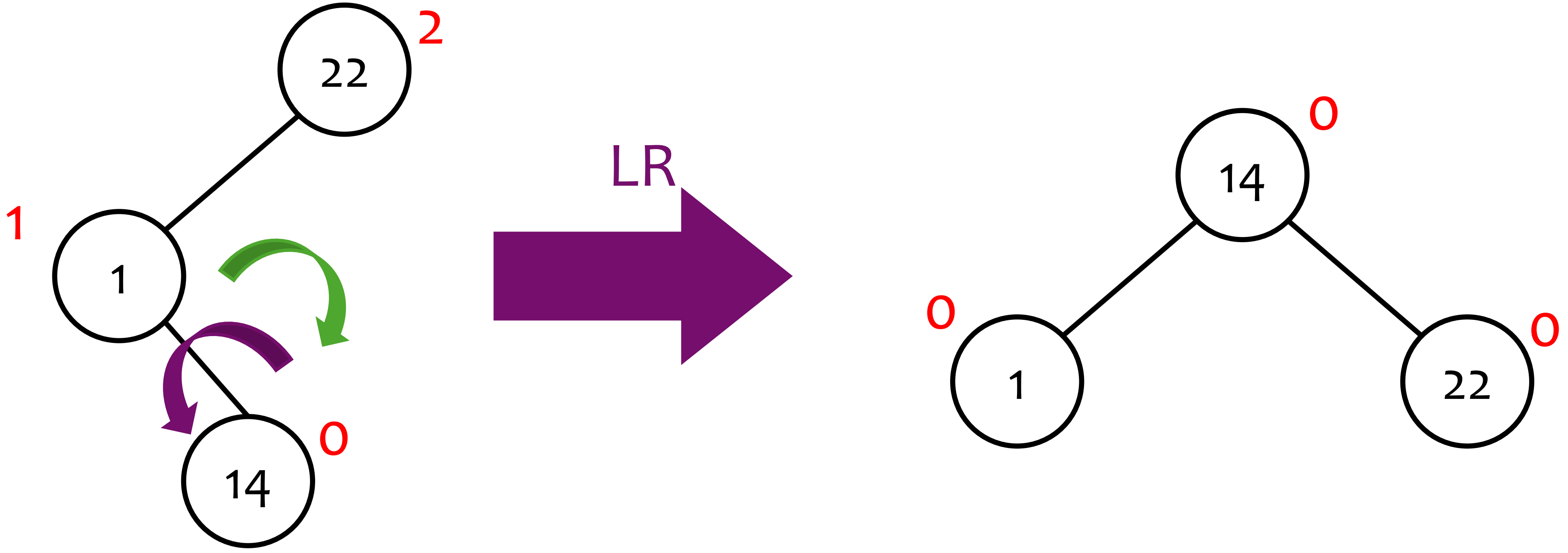
- Finally, we need to **update the height** of the node after insertion.

```
TreePtr insert_node(ElementType element, TreePtr tree) {
    if (tree == NULL) {
        tree = create_node(element);
    } else if (element < tree->element) { ... // P.53
    } else if (element > tree->element) { ... // P.54
    }
    tree->height = max(height(tree->left),
        height(tree->right)) + 1;
    return tree;
}
```

Insertion

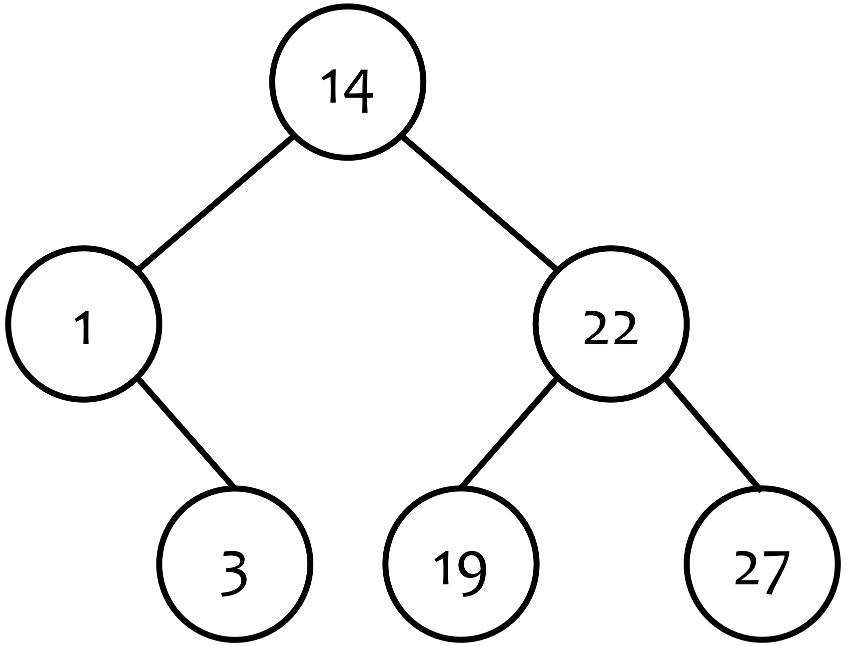
- Example: Create an AVL tree by inserting the following elements in the specified order: 22, 1, 14, 19, 3, 27

```
TreePtr avl_tree = insert_node(22, NULL);
avl_tree = insert_node(1, avl_tree);
avl_tree = insert_node(14, avl_tree);
```



Insertion

```
avl_tree = insert_node(19, avl_tree);
avl_tree = insert_node(3, avl_tree);
avl_tree = insert_node(27, avl_tree);
```

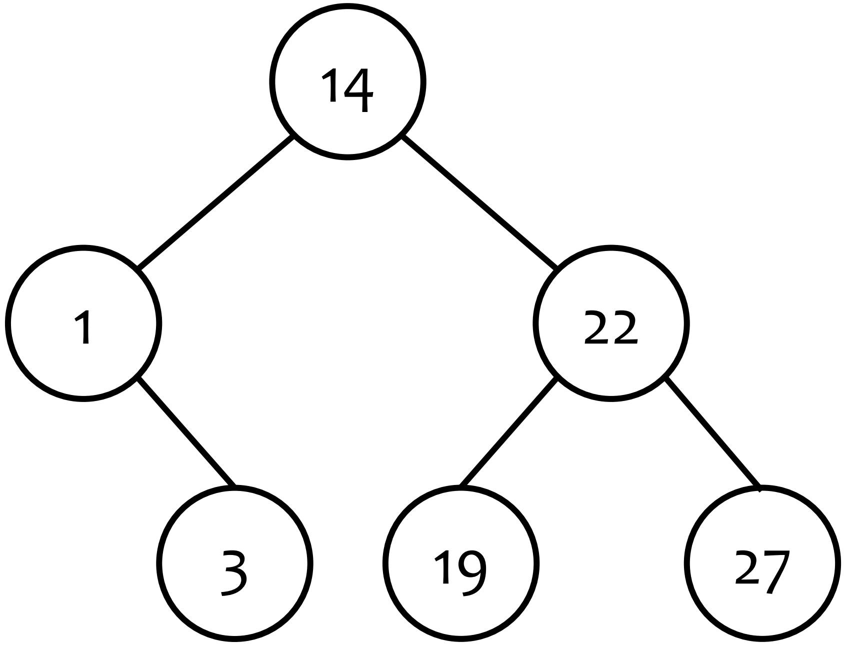


- Unlike the binary search tree, we must assign the output (pointer to the root) to `avl_tree` because the root may change after insertion.

Exercise: AVL Tree Insertion

- Suppose the variable `avl_tree` points to the following AVL tree. What is the output of the following code snippet?

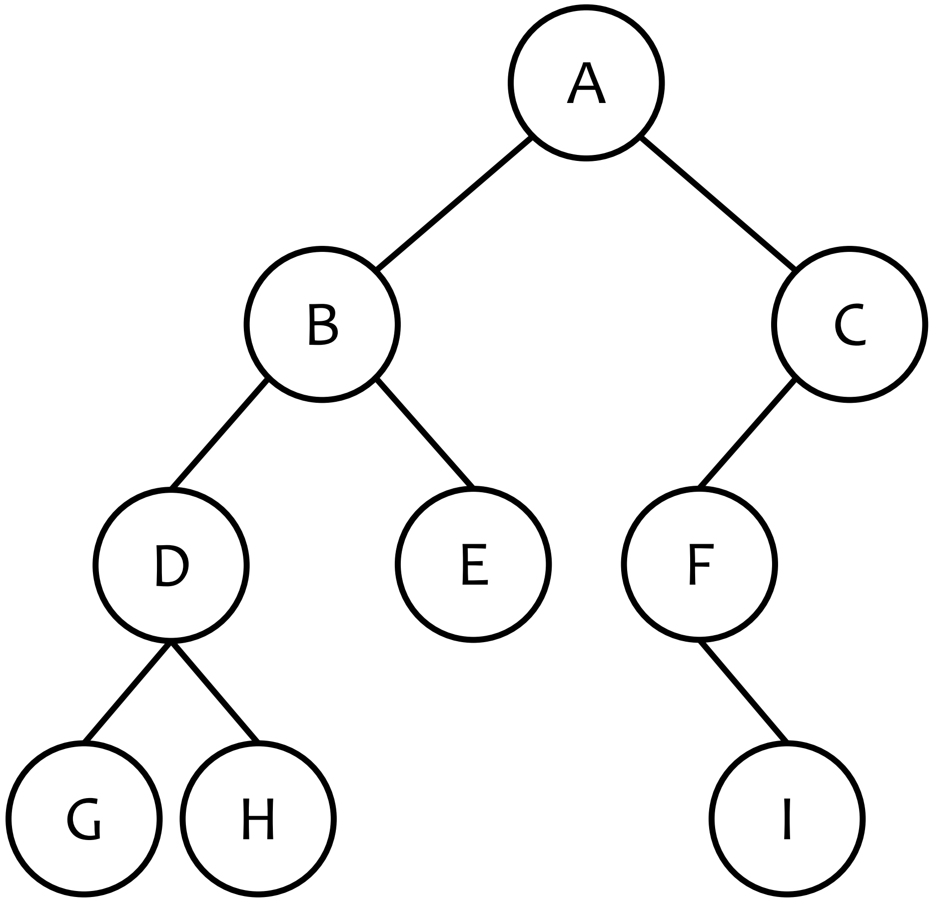
```
avl_tree = insert_node(26, avl_tree);
avl_tree = insert_node(25, avl_tree);
avl_tree = insert_node(5, avl_tree);
preorder(avl_tree);
putchar('\n');
```



Solution to Exercise

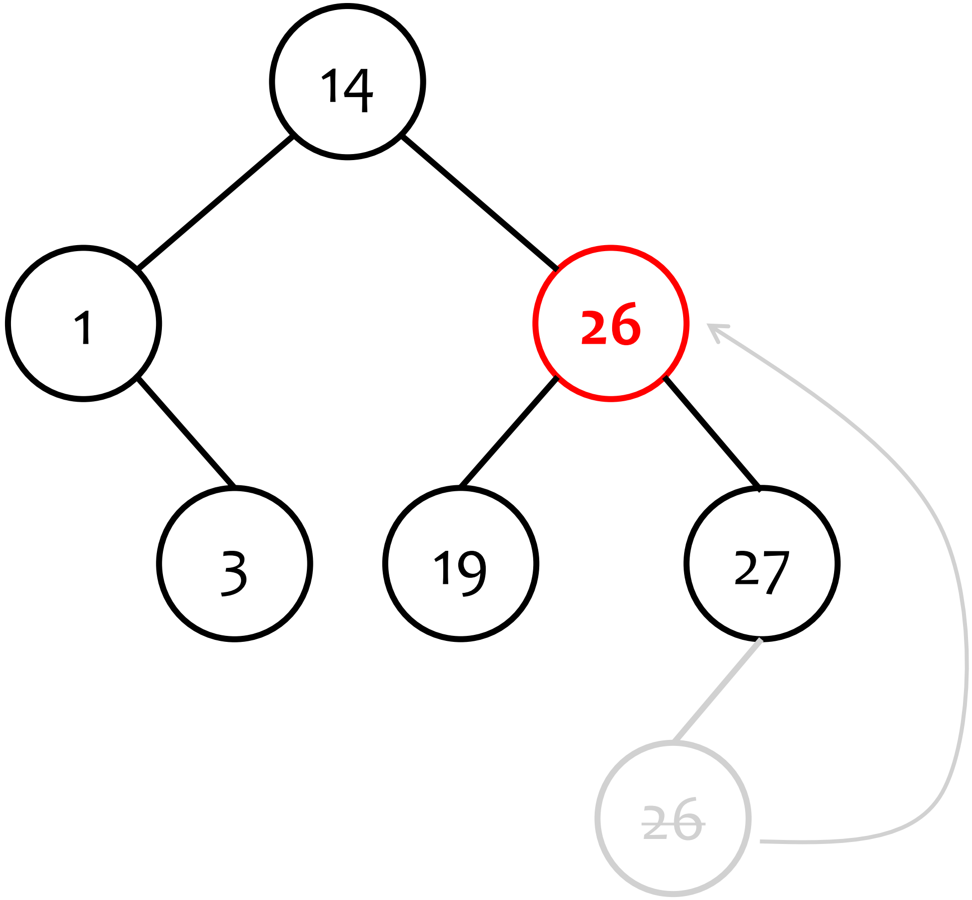
Exercise: Tree Traversal (P.21) [Solution]

- Preorder (VLR)
  - A -> B -> D -> G -> H -> E -> C -> F -> I
- Inorder (LVR)
  - G -> D -> H -> B -> E -> A -> F -> I -> C
- Postorder (LRV)
  - G -> H -> D -> E -> B -> I -> F -> C -> A

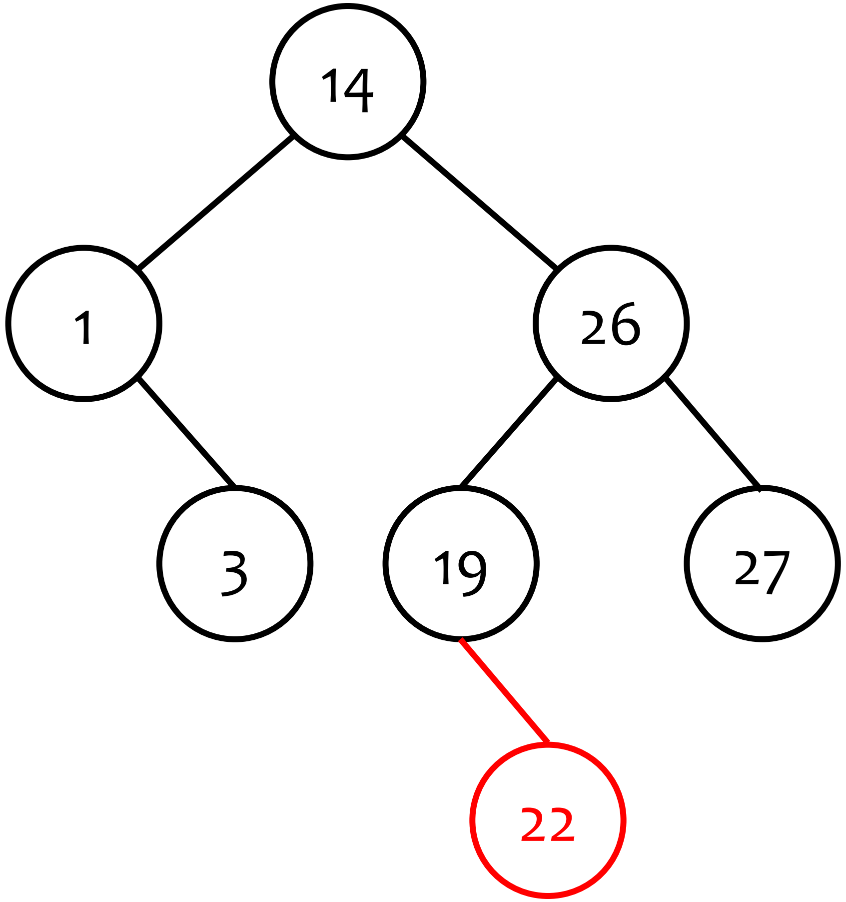


Exercise: Binary Search Tree Operations (P.34) [Solution]

```
delete_node(22, search_tree);
```




```
insert_node(22, search_tree);
```



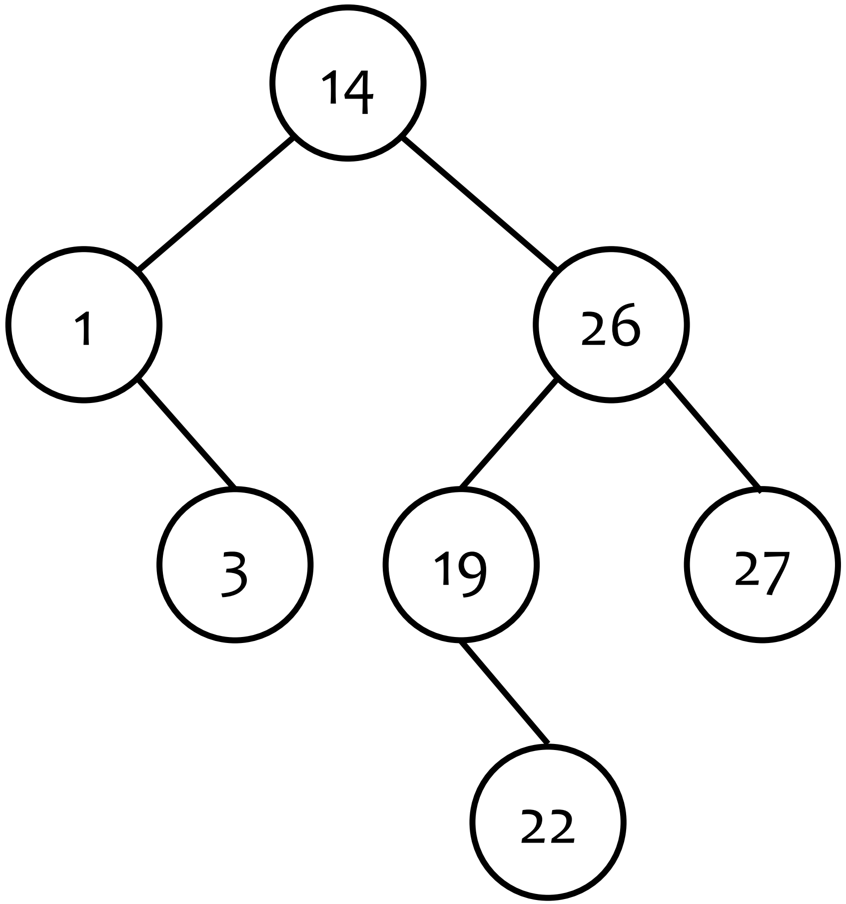
Exercise: Binary Search Tree Operations (P.34) [Solution]

- Output:

```
14  1  3  26  19  22  27
1  3  14  19  22  26  27
3  1  22  19  27  26  14
```

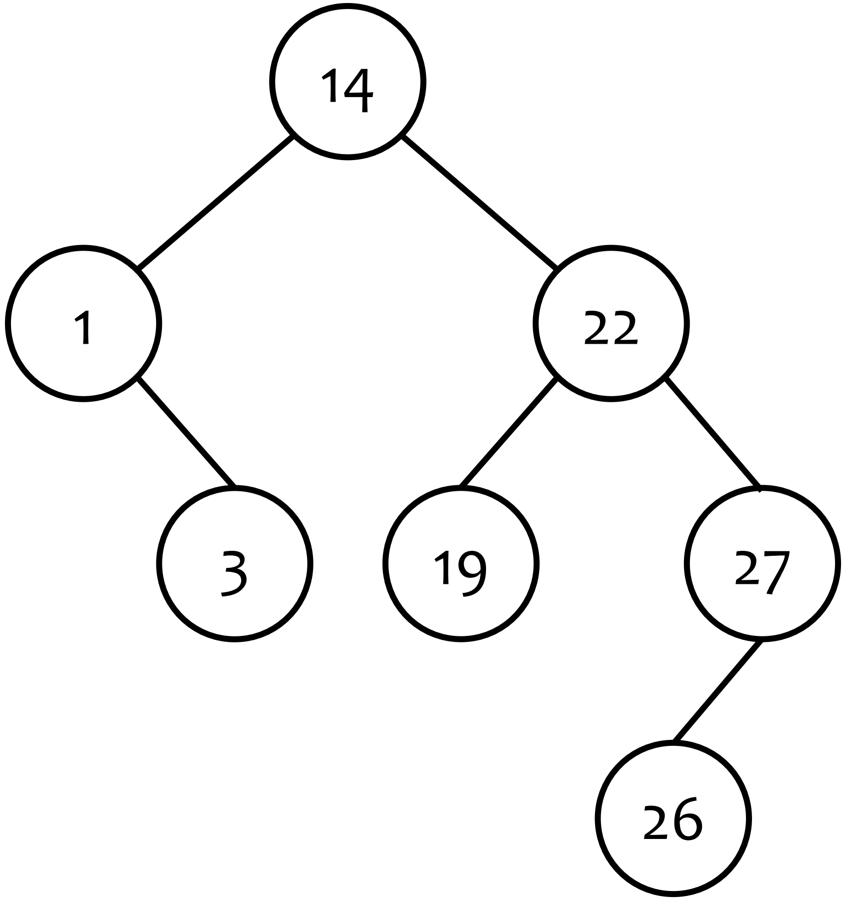
  - To be precise, there is a whitespace character at the end of each line.

- Observation: **Inorder = Ascending order**

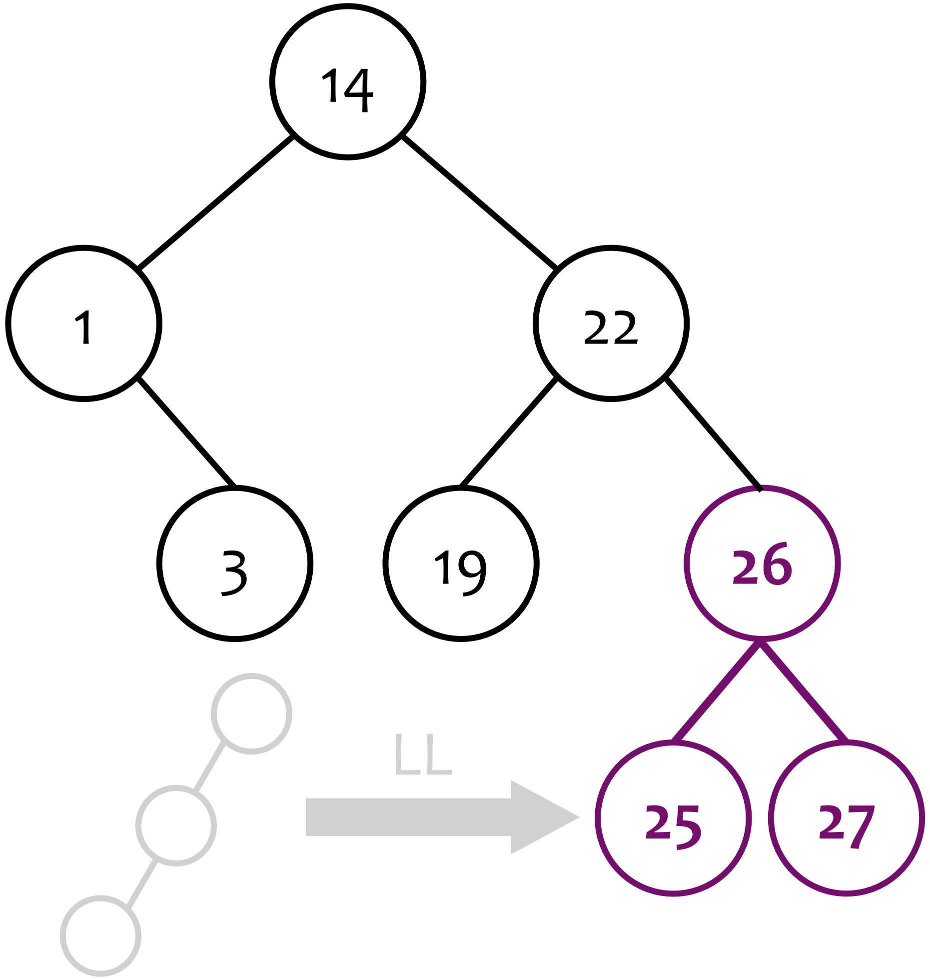


Exercise: AVL Tree Insertion (P.58) [Solution]

```
insert_node(26, avl_tree);
```




```
insert_node(25, avl_tree);
```



Exercise: AVL Tree Insertion (P.58) [Solution]

- Output:

  ```
  14 3 1 5 22 19 26 25 27
  ```

  - To be precise, there is a whitespace character at the end of the line.


```
insert_node(5, avl_tree);
```