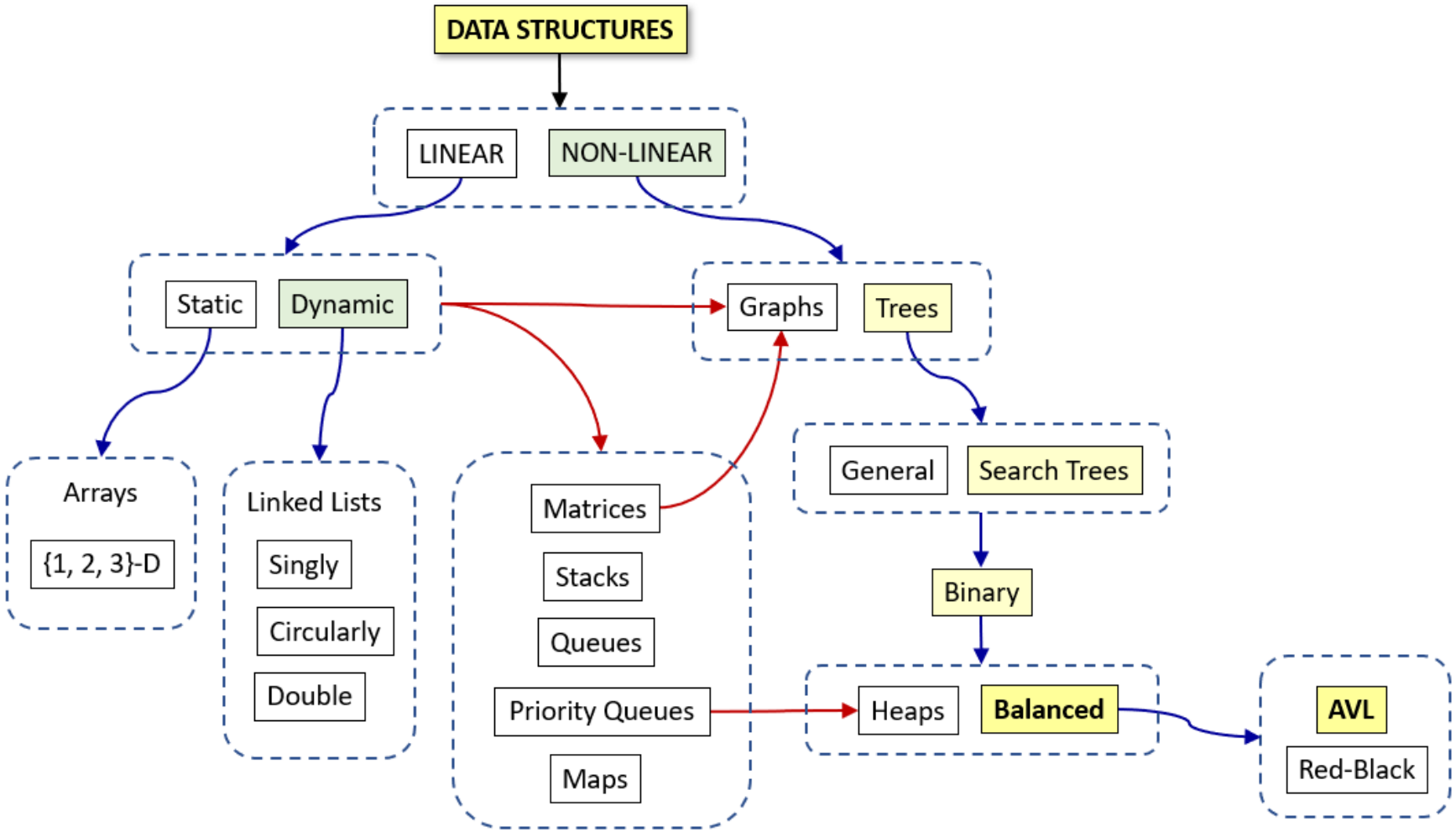
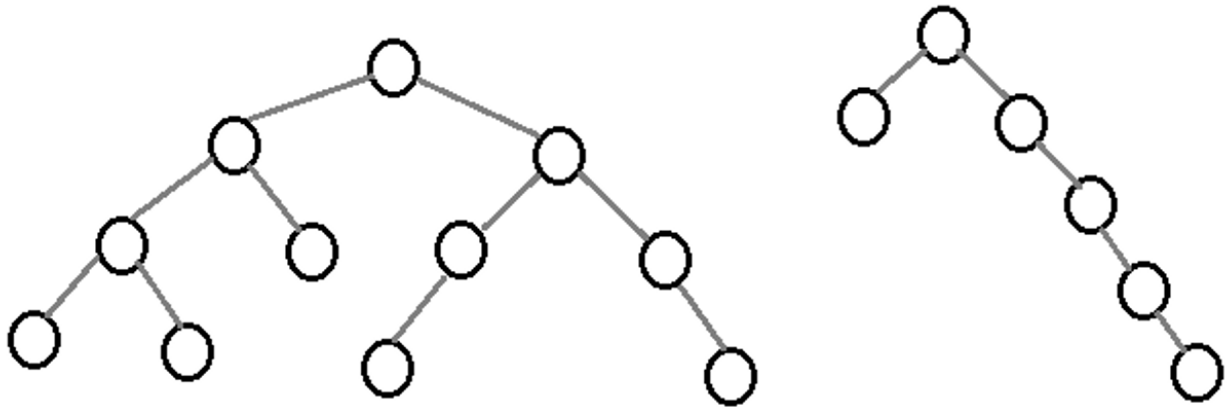
AVL Trees


DATA STRUCTURES
LINEAR
NON-LINEAR
Static
Dynamic
Graphs
Trees
Arrays
{1, 2, 3}-D
Linked Lists
Singly
Circularly
Double
Matrices
Stacks
Queues
Priority Queues
Maps
General
Search Trees
Binary
Heaps
Balanced
AVL
Red-Black

AVL trees

- Binary search trees
  - Great
  - Search, insert,delete = O(log n) time complexity
  - UNLESS they become unbalanced
    then
  - Search, insert,delete = O(n) time complexity



Balanced Binary Tree

Unbalanced Binary Tree

AVL trees

- Binary search trees
  - Great
  - Search, insert,delete = O(log n) time complexity
  - UNLESS they become unbalanced
    then
  - Search, insert,delete = O(n) time complexity

- AVL trees are binary search trees with special insertion and deletion algorithms to keep them well-balanced

- so the time complexity of search insert and delete remain O(log n)

Height of a Tree

- Height of a tree is the length of the longest path from root to some leaf node.
- Height of an empty tree is -1.
- Height of a single node tree is 0.
- Recursive definition:
  - height(t) = 0 if number of nodes = 1

    = -1 if T is empty

    = 1+ max(height(LT), height(RT)) otherwise

AVL Property

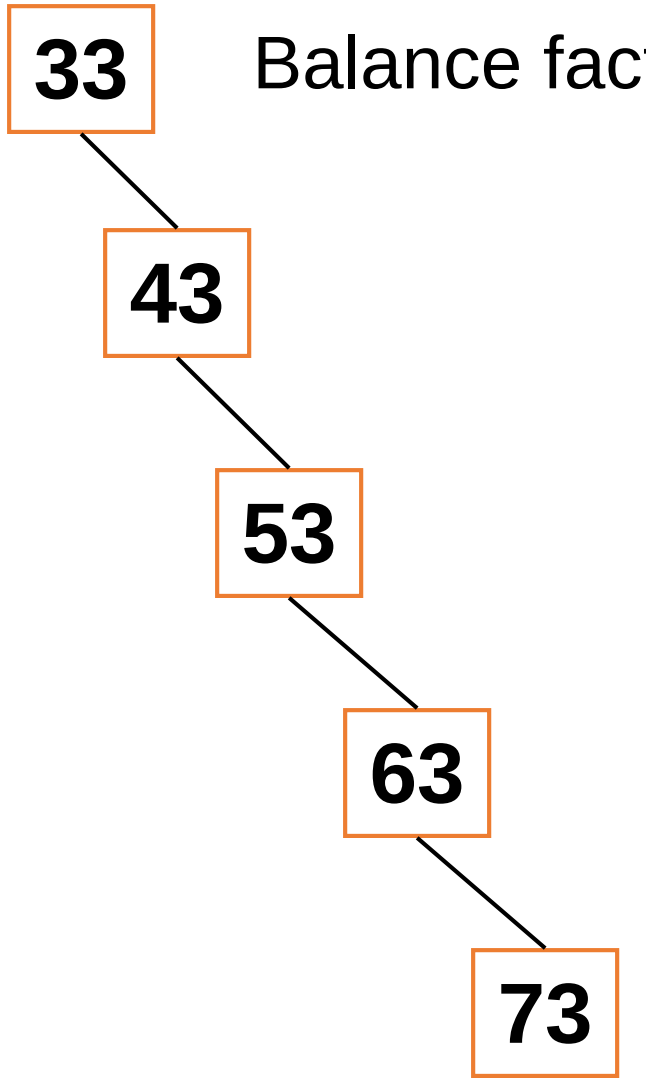
- If N is a node in a binary tree,

  node X has AVL property if

  the heights of the left sub-tree and right sub-tree are equal or if they differ by 1.

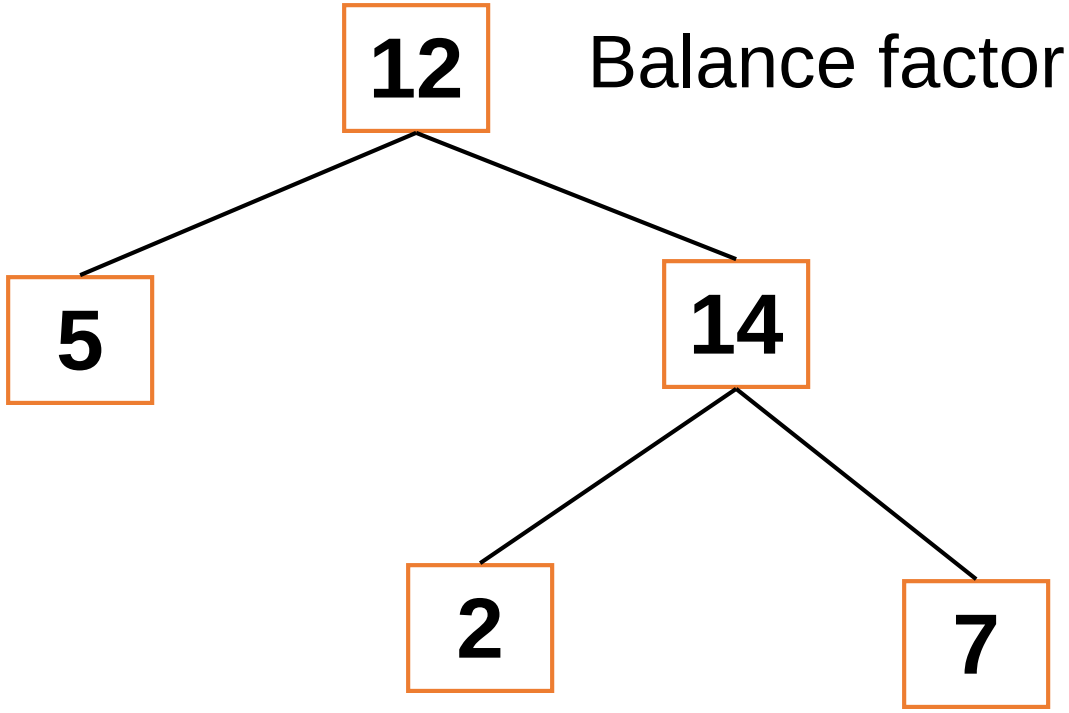
- Let’s look at some examples.

Degenerate BST



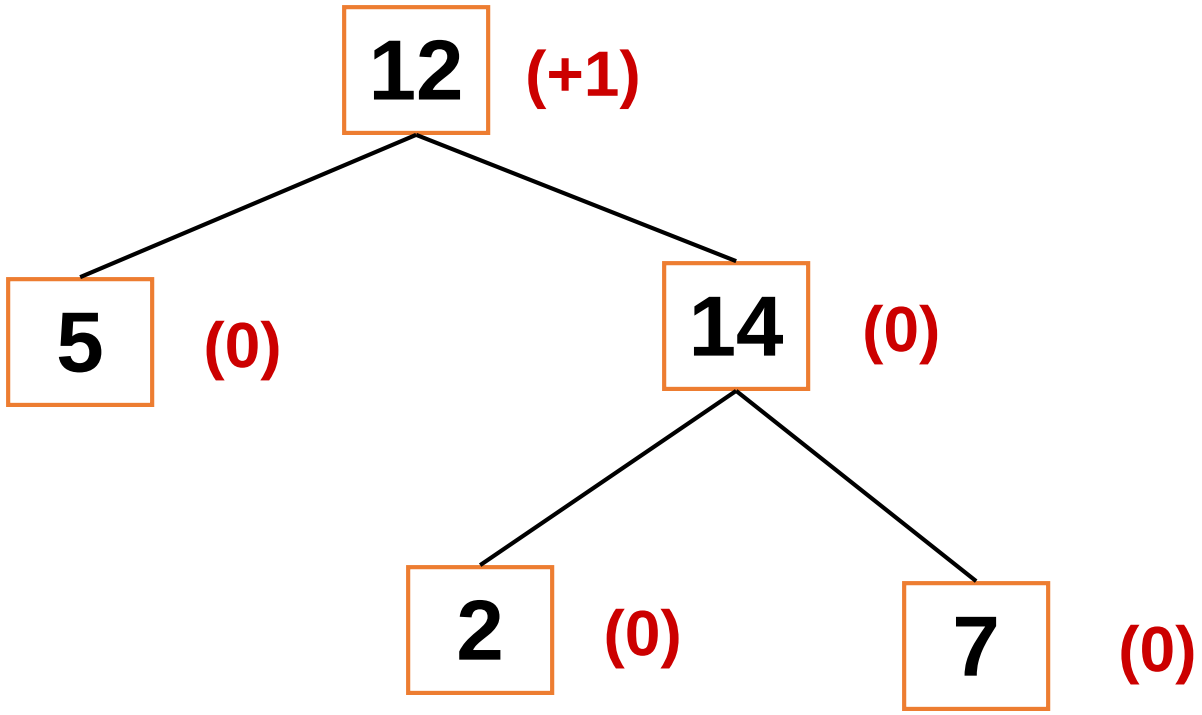
Balance factor = 4 - 0 = 4

Degenerate BST

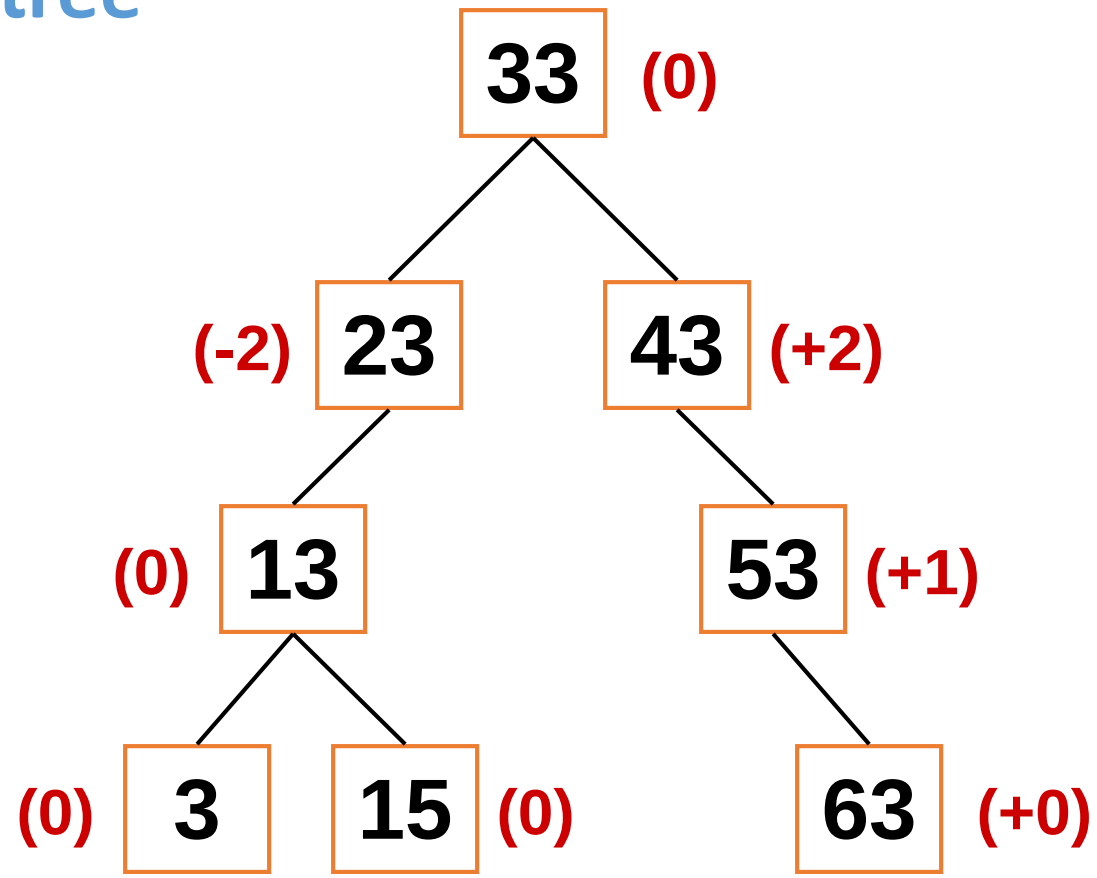


Balance factor = 2 - 1 = 1

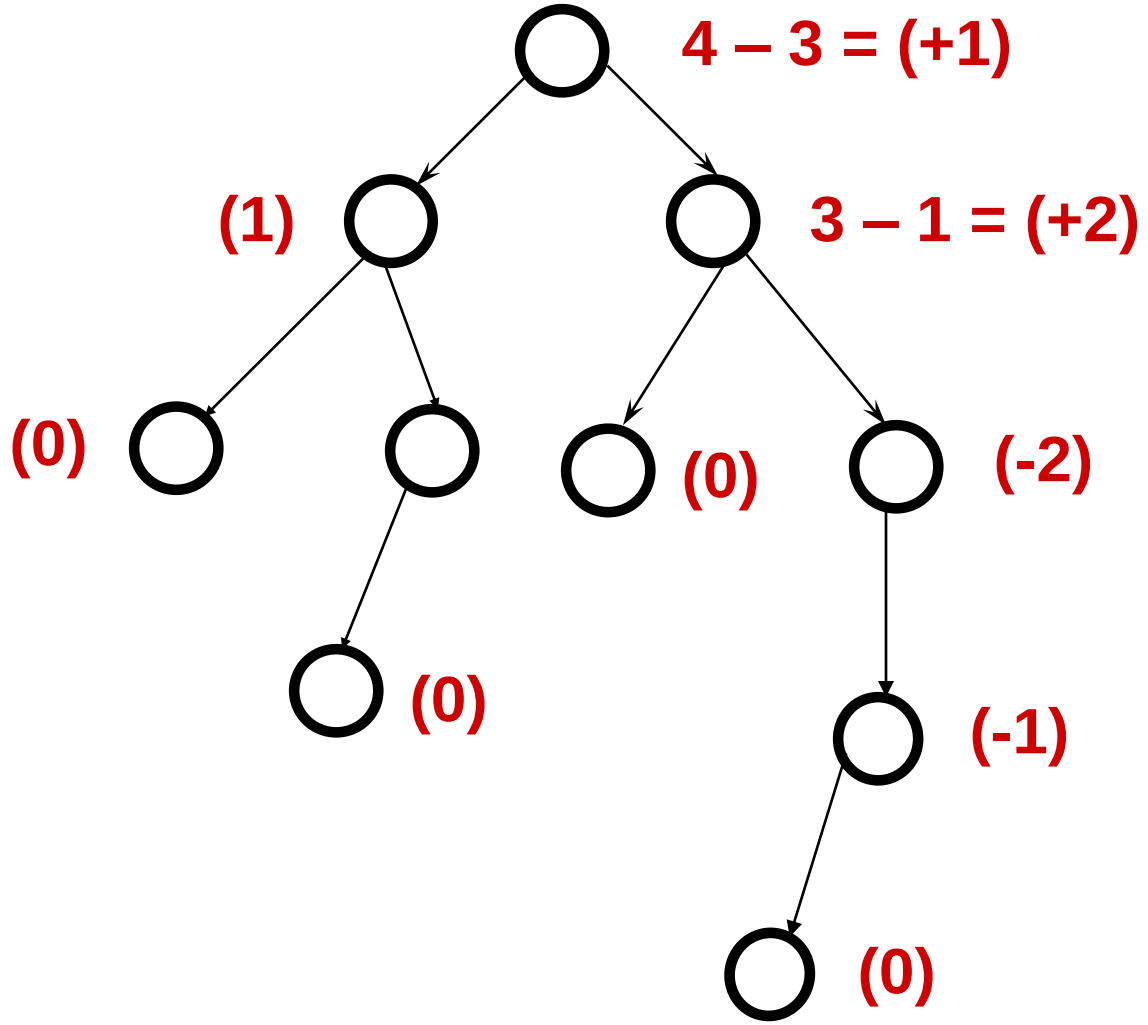
AVL tree



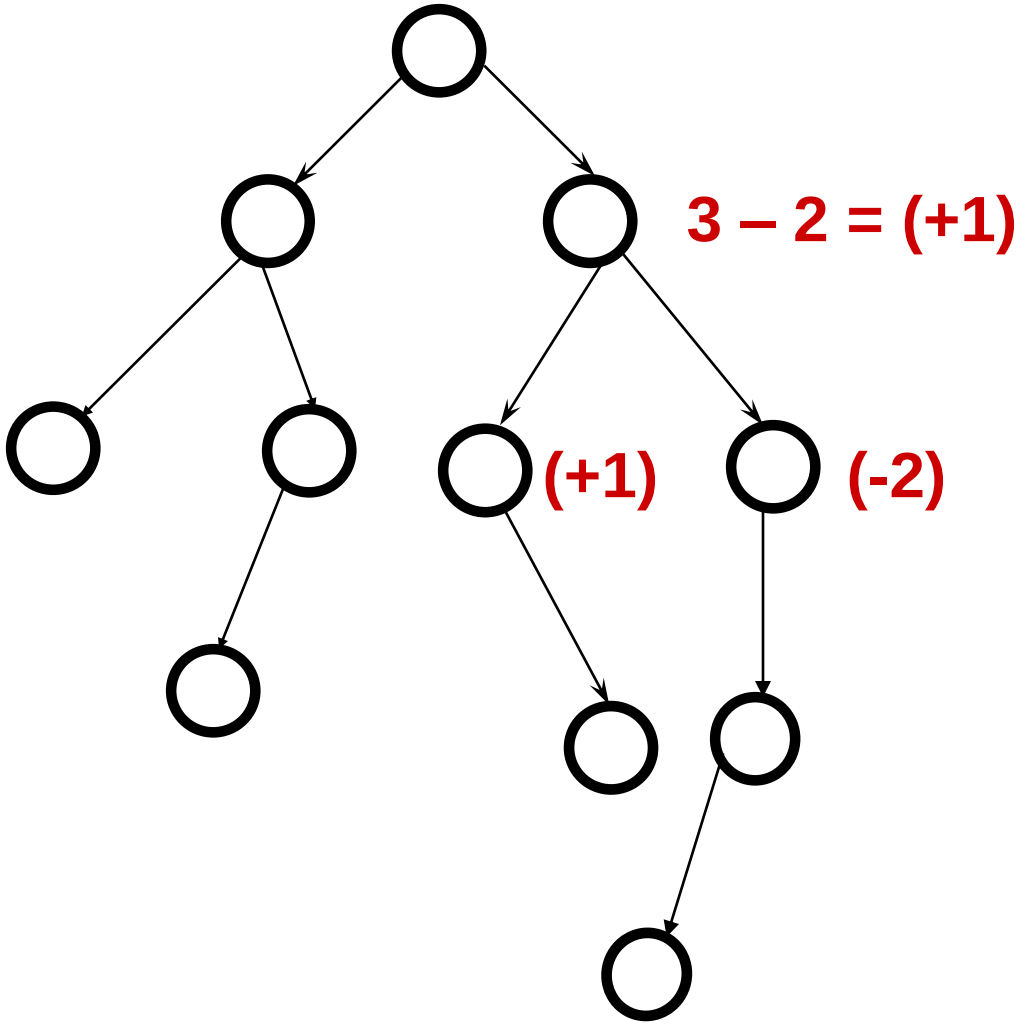
Non-AVL tree



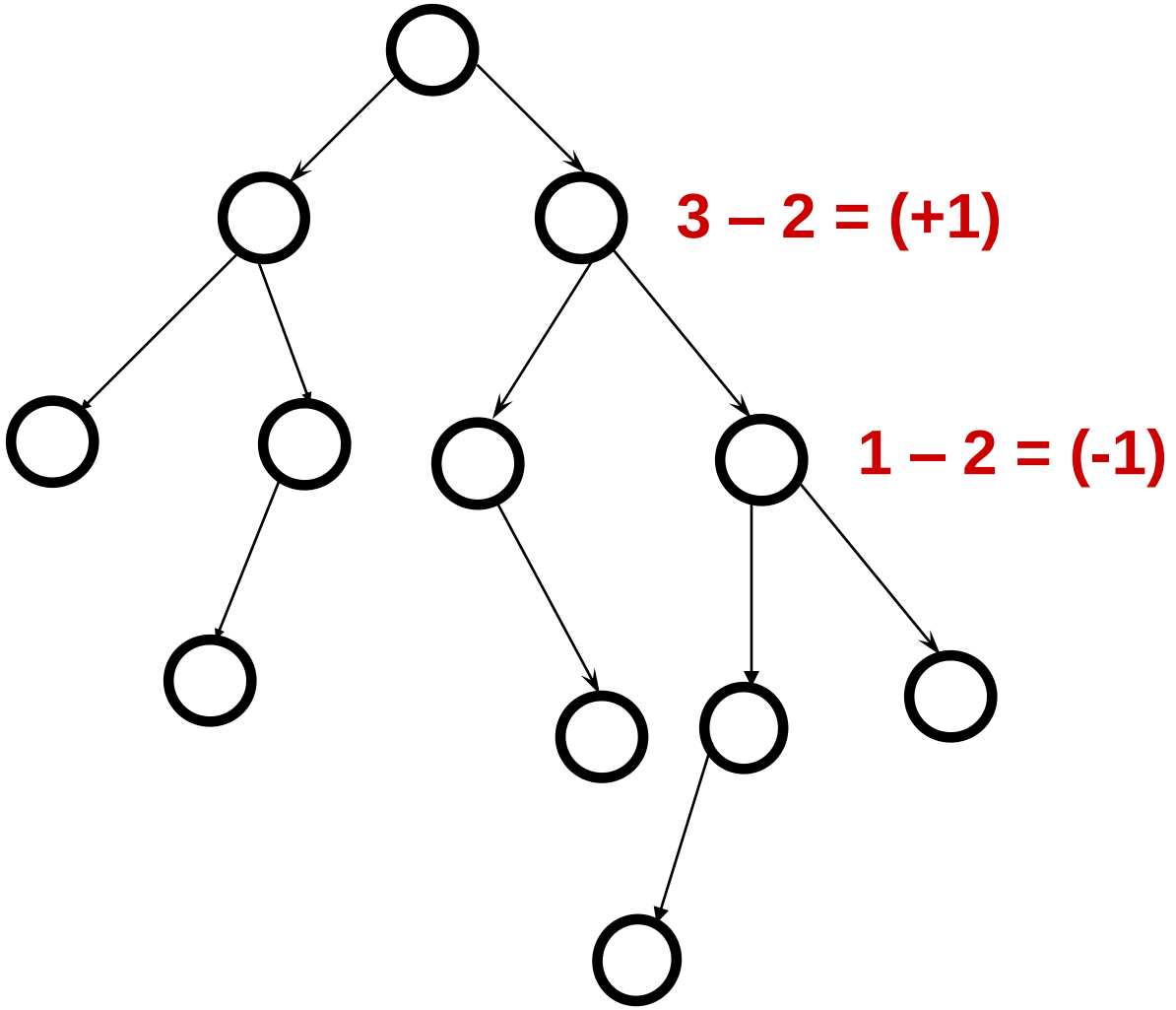
Non-AVL tree



Non-AVL tree



AVL tree

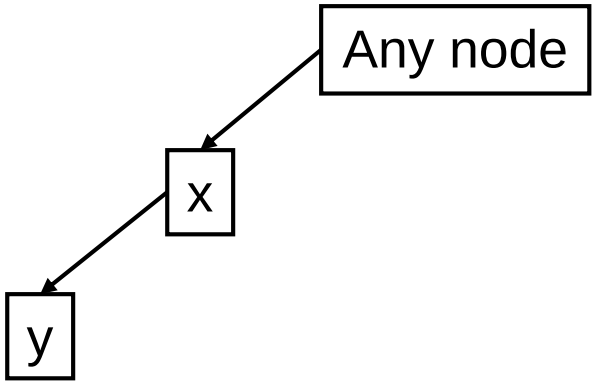


AVL trees

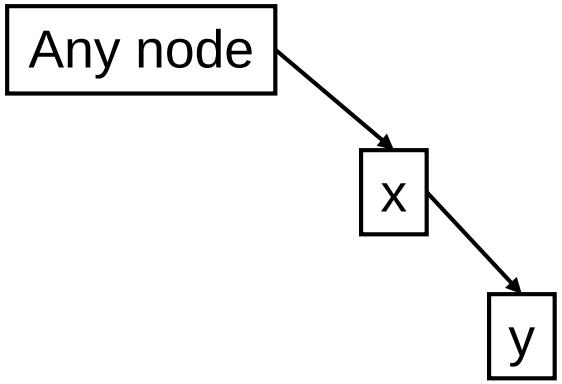
- AVL tree = a binary search tree with the following property: for every node the heights of the left and right subtrees differ at most by one.
- That's all very nice but how do we guarantee it?
- We have to somehow modify the insert and delete functions.
- If, after an insertion or deletion, the property is not satisfied, we "rotate" the tree to make it balanced.

AVL trees

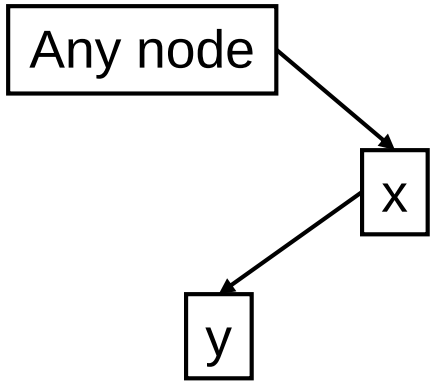
- When can an insertion of a child y at node x cause an imbalance?
  - when both x and y are left children
  - when both x and y are right children
  - when x is a right child and y is a left child
  - when y is a right child and x is a left child



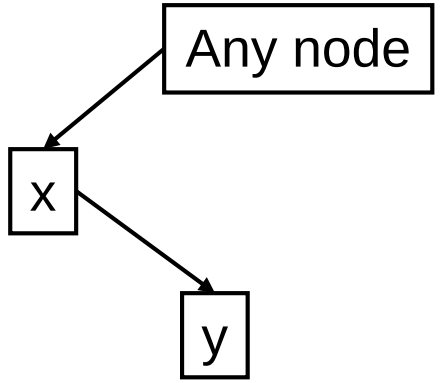
when both x and y are left children



when both x and y are right children



when x is a right child and y is a left child



when y is a right child and x is a left child

AVL trees

- AVL tree = a binary search tree with the following property: for every node the heights of the left and right subtrees differ at most by one.

- That's all very nice but how do we guarantee it?

- We have to somehow modify the insert and delete functions.

- If, after an insertion or deletion, the property is not satisfied, we "rotate" the tree to make it balanced.

https://www.youtube.com/watch?v=mGF61O21W-c

AVL trees

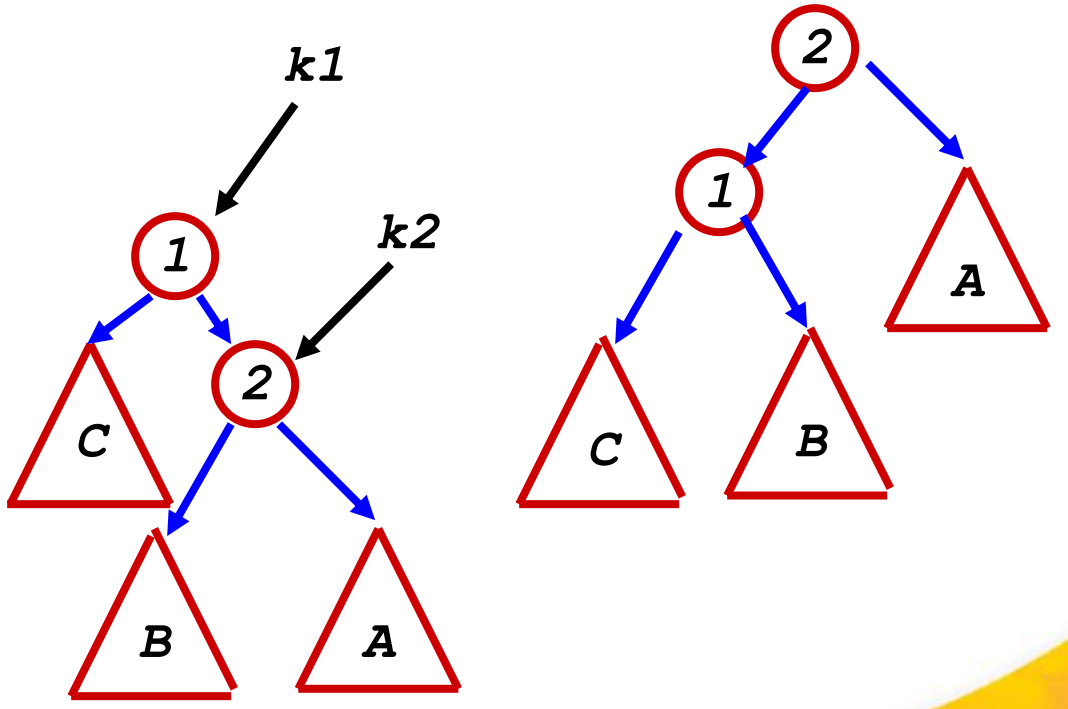
- How/when do we decide whether to rotate?
- Example: insertion
  - step 1: walk down the tree to insert the node in the correct position
  - step 2: walk up the tree checking the property at each node
- we need a helper function to determine the heights of the subtrees of each node.
- we need to be able to determine whether to perform a **single** or a **double** rotation

AVL trees

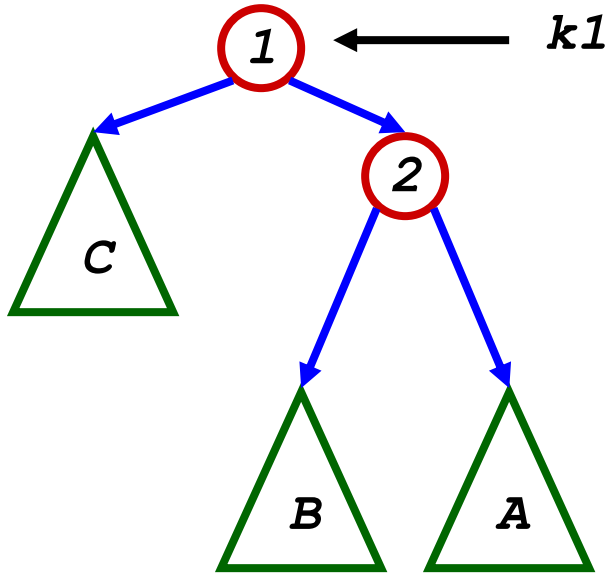
- Insertion
- may not unbalance the tree
- If it does then two possible responses
  - **Single rotation**
  - **Double rotation**

Single Rotation Left

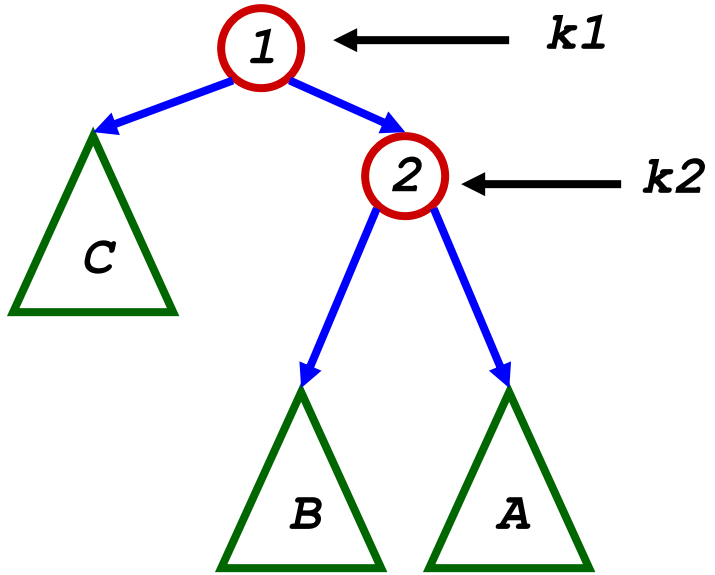
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



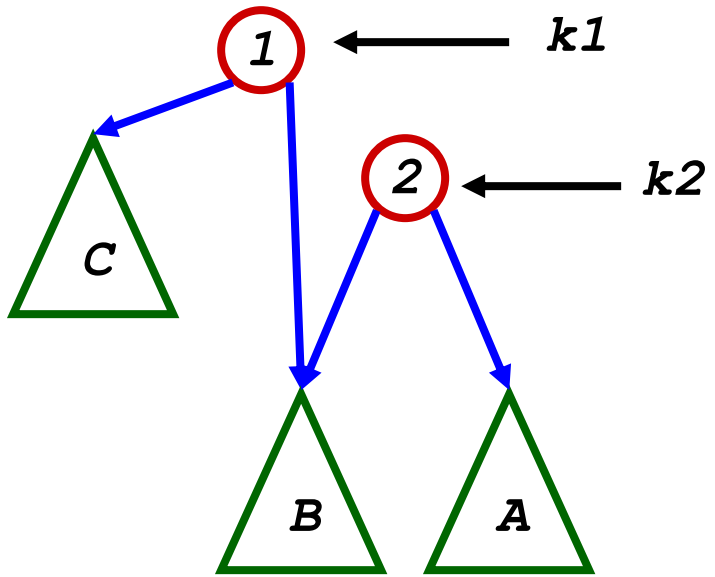
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



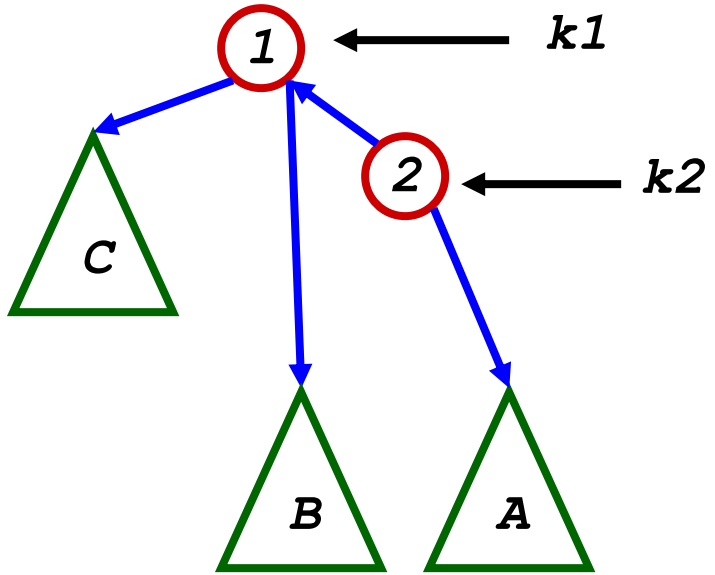
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



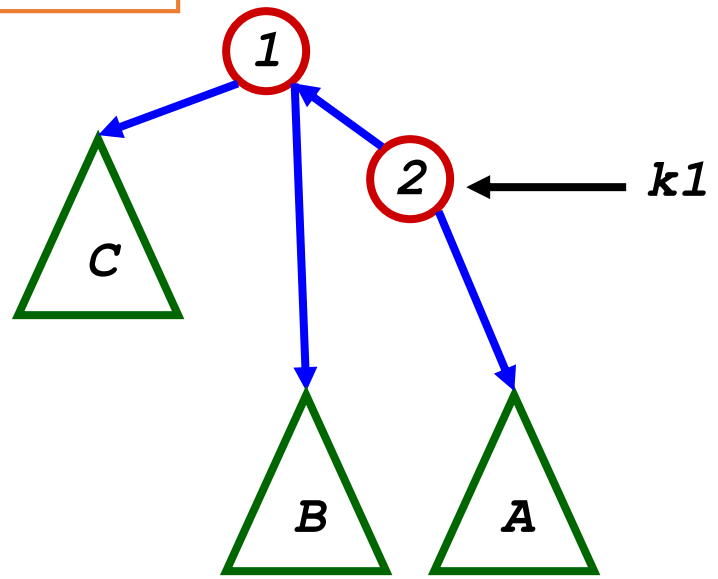
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



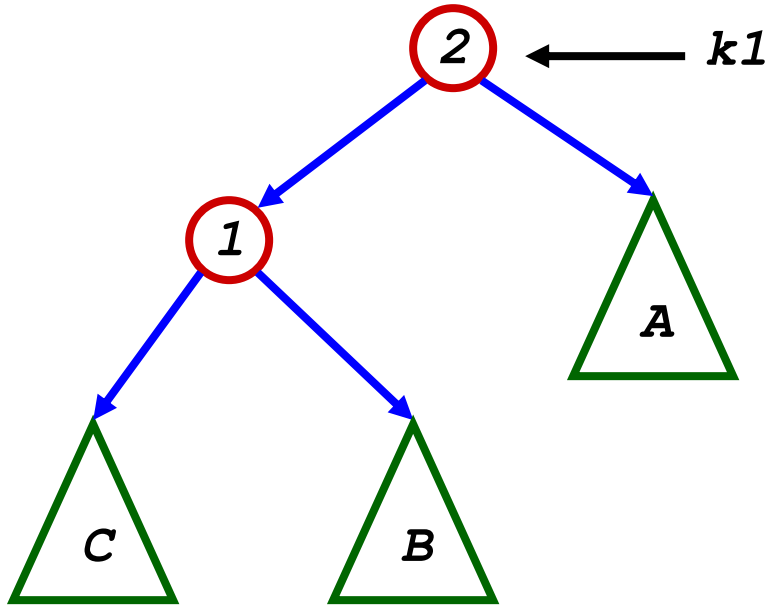
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



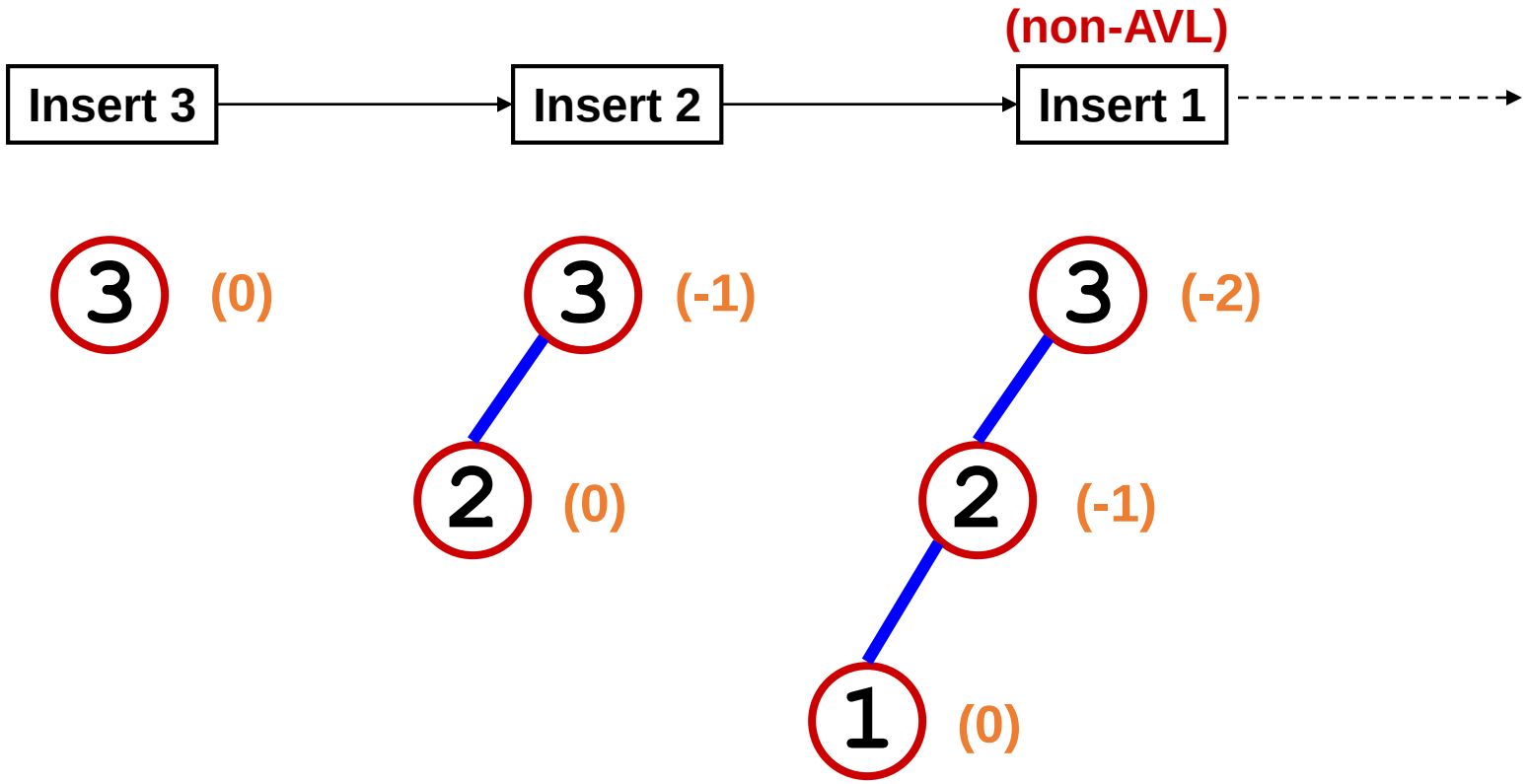
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```

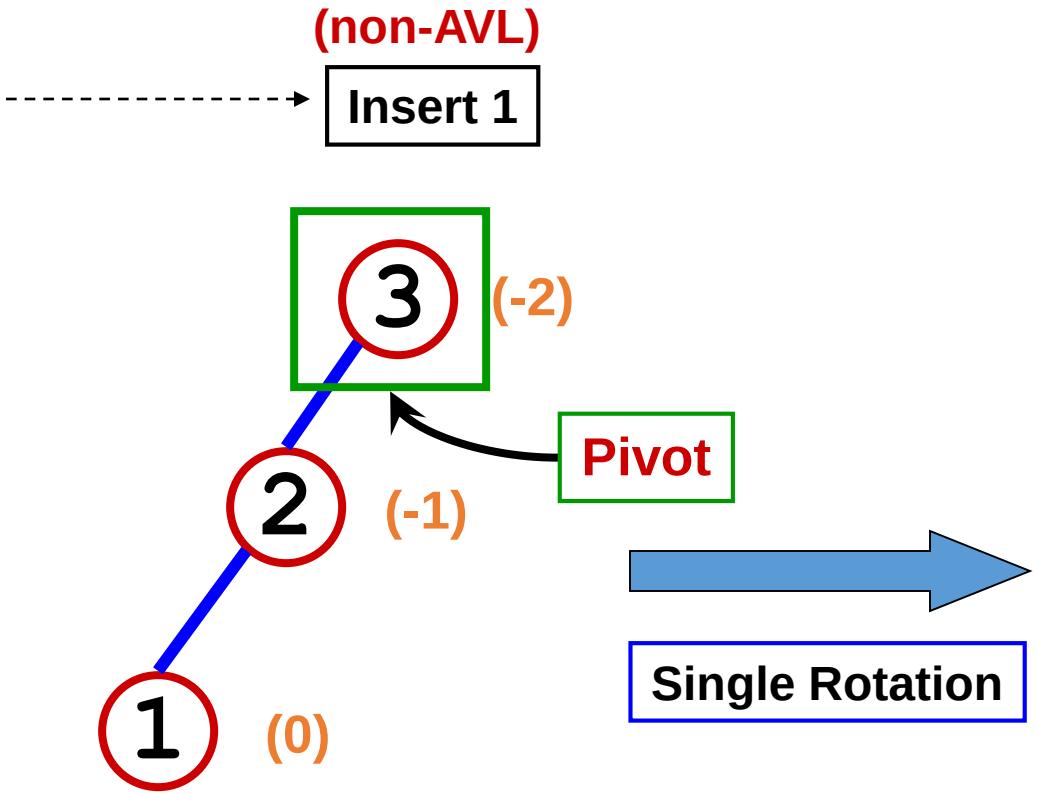


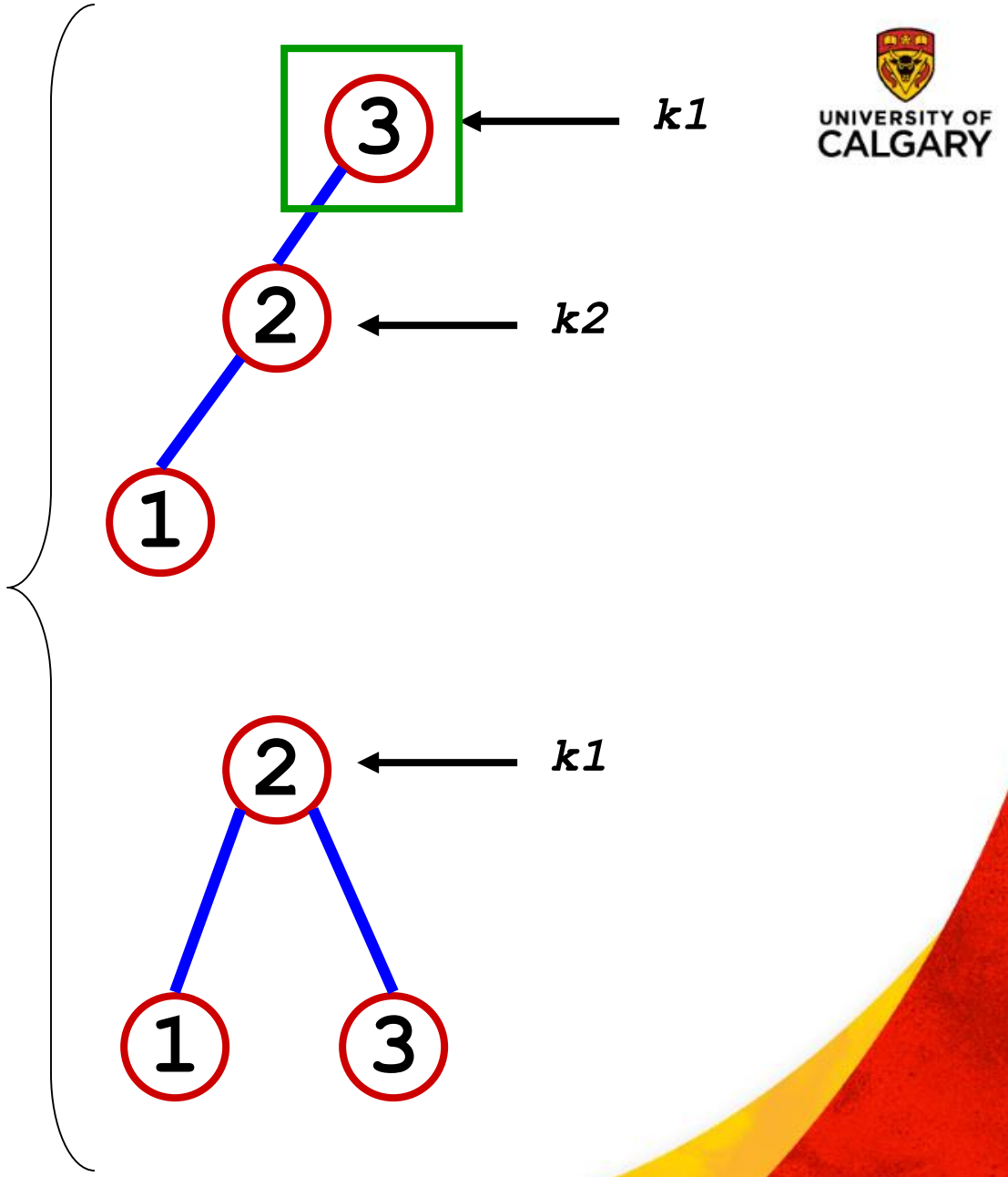
```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```



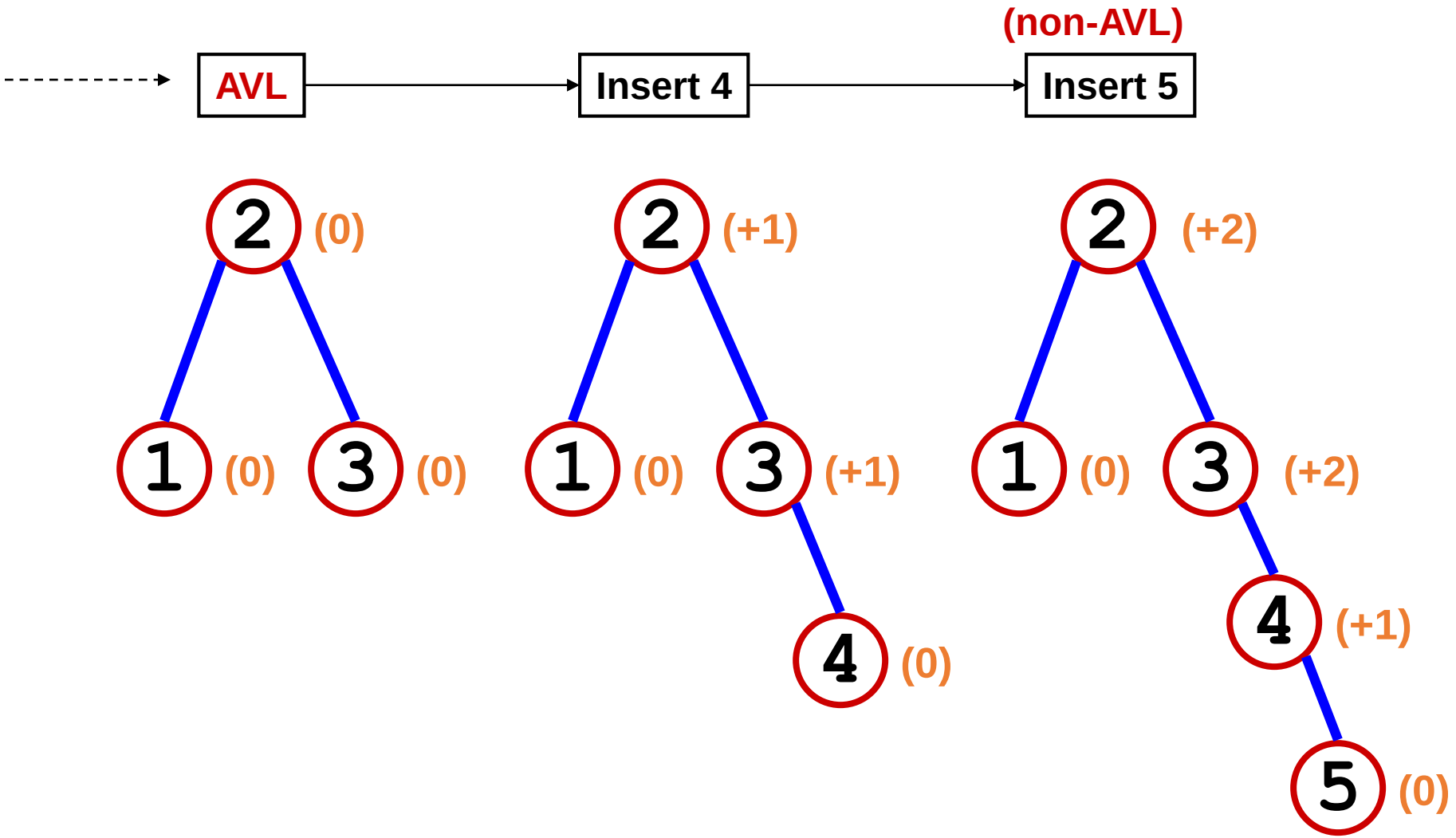
AVL Trees Example

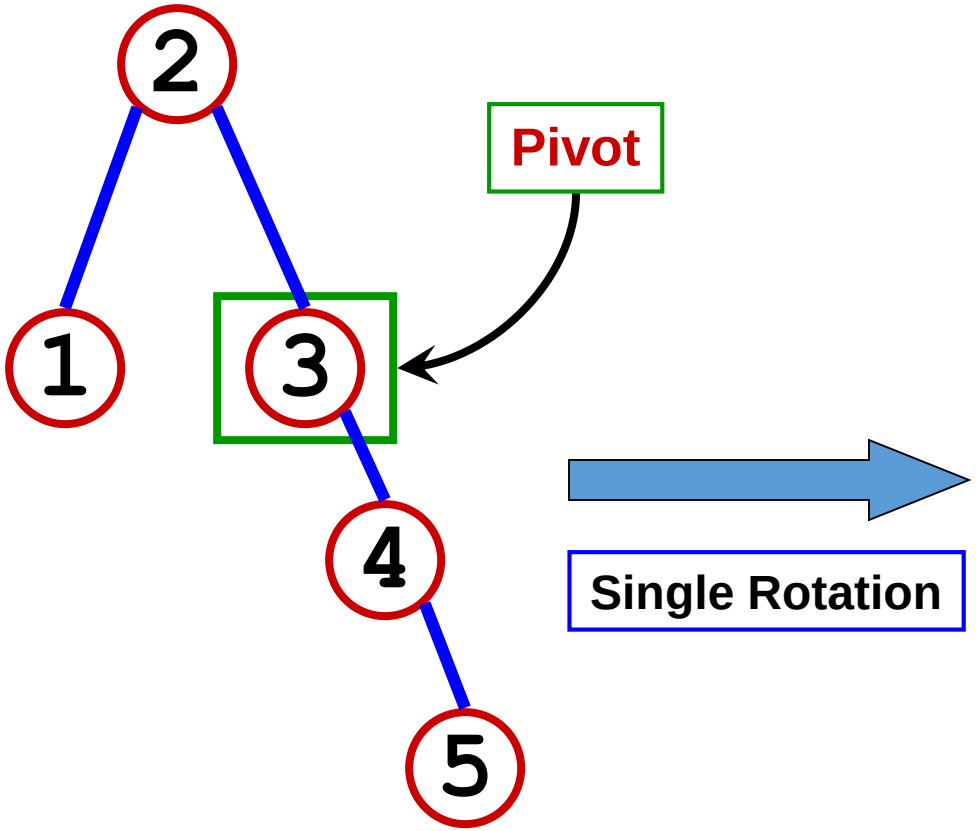


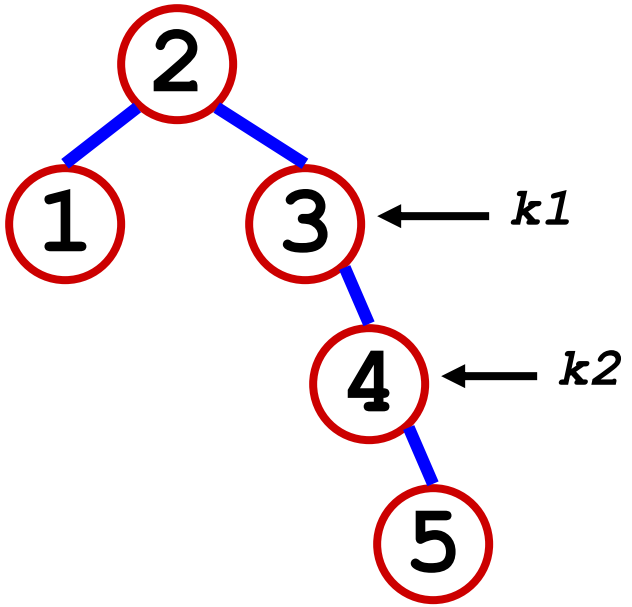


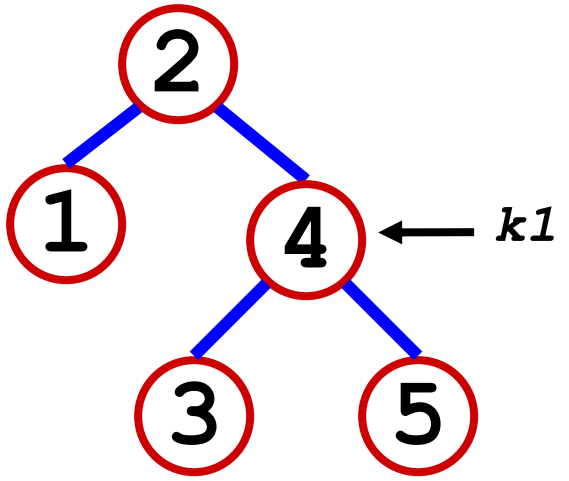



```
void RightRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->left;
        k1->left = k2->right;
        k2->right = k1;
        k1 = k2;
}
```

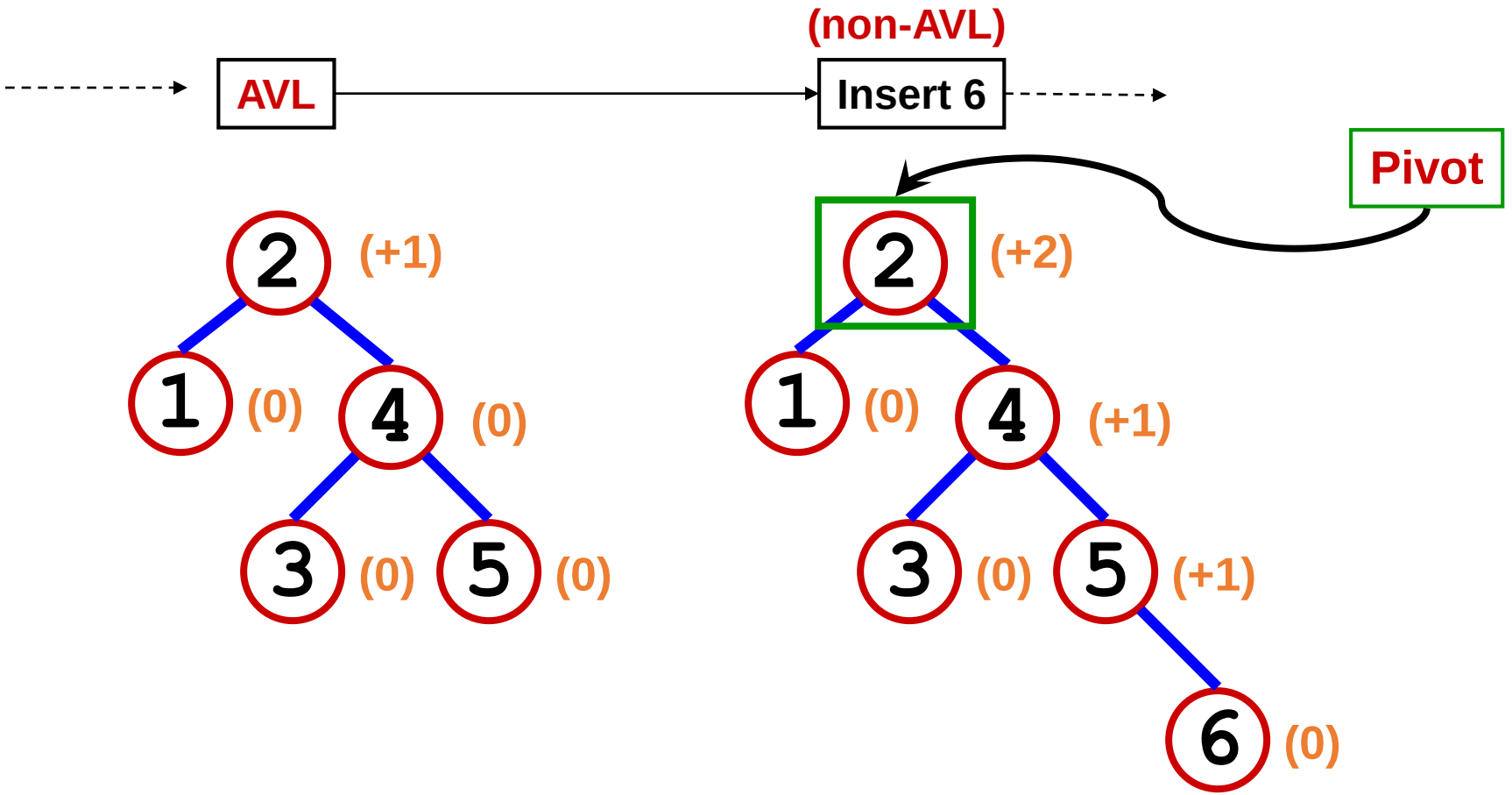

AVL
Insert 4
Insert 5
(non-AVL)
2 (0)
1 (0)
3 (0)
2 (+1)
1 (0)
3 (+1)
4 (0)
2 (+2)
1 (0)
3 (+2)
4 (+1)
5 (0)

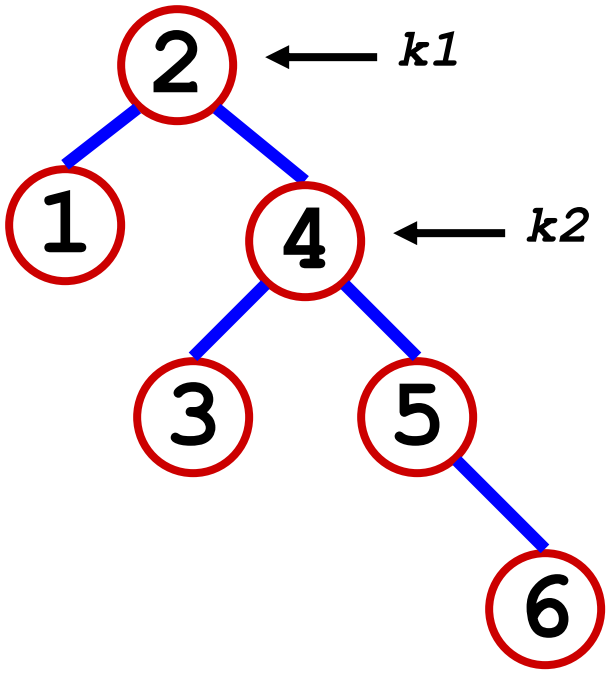




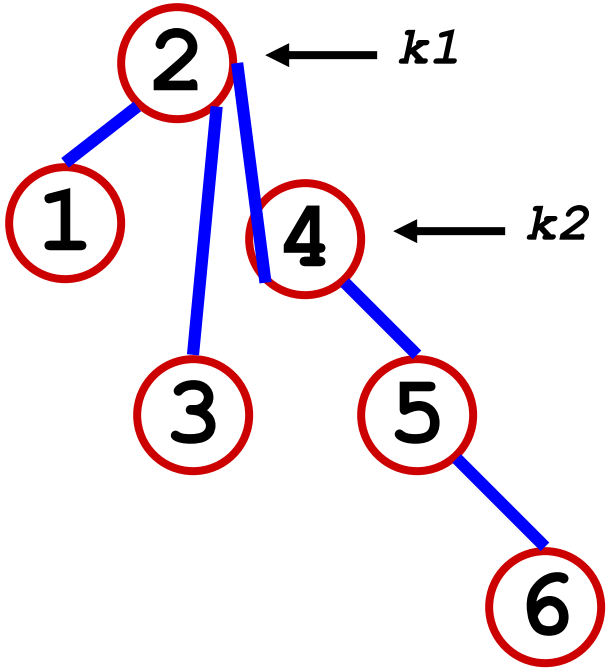


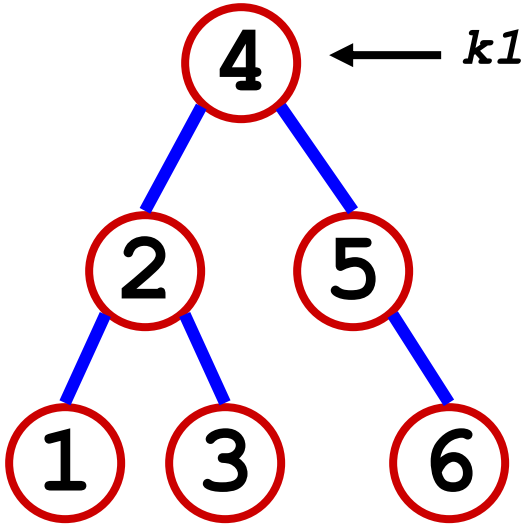

```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```


AVL
(non-AVL)
Insert 6
Pivot
2
(+1)
1
(0)
4
(0)
3
(0)
5
(0)
2
(+2)
1
(0)
4
(+1)
3
(0)
5
(+1)
6
(0)

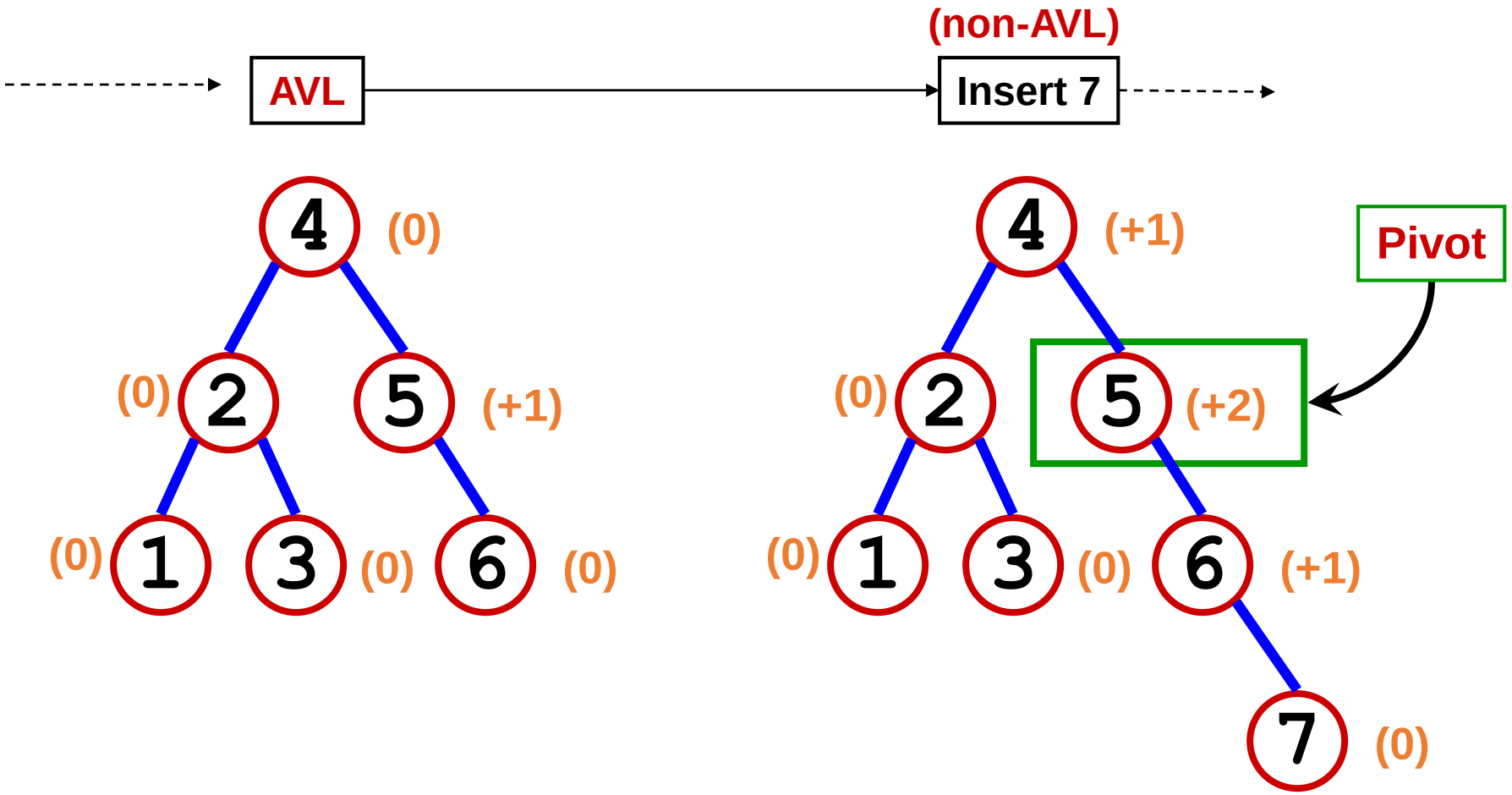


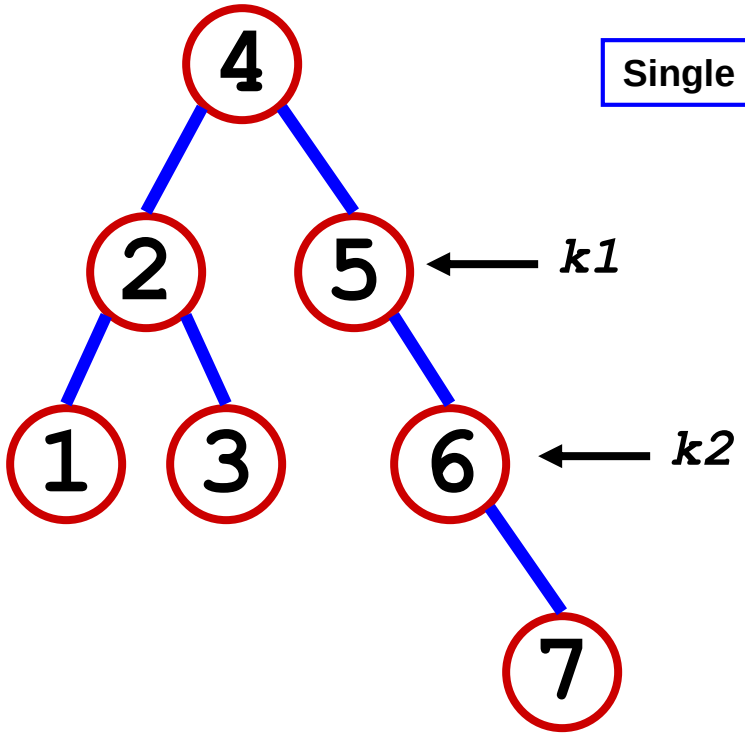
Single Rotation



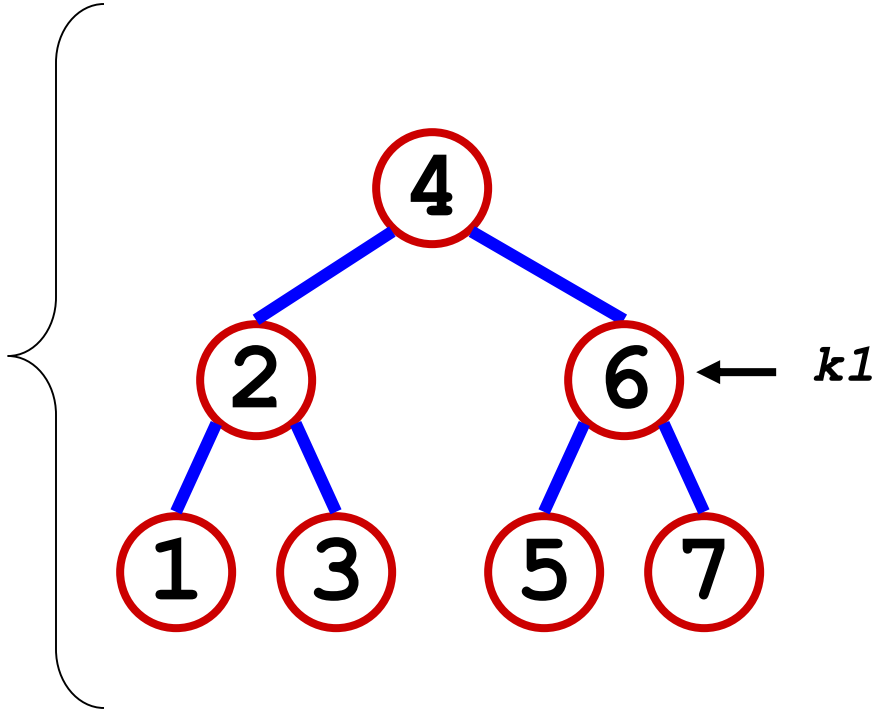



```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```

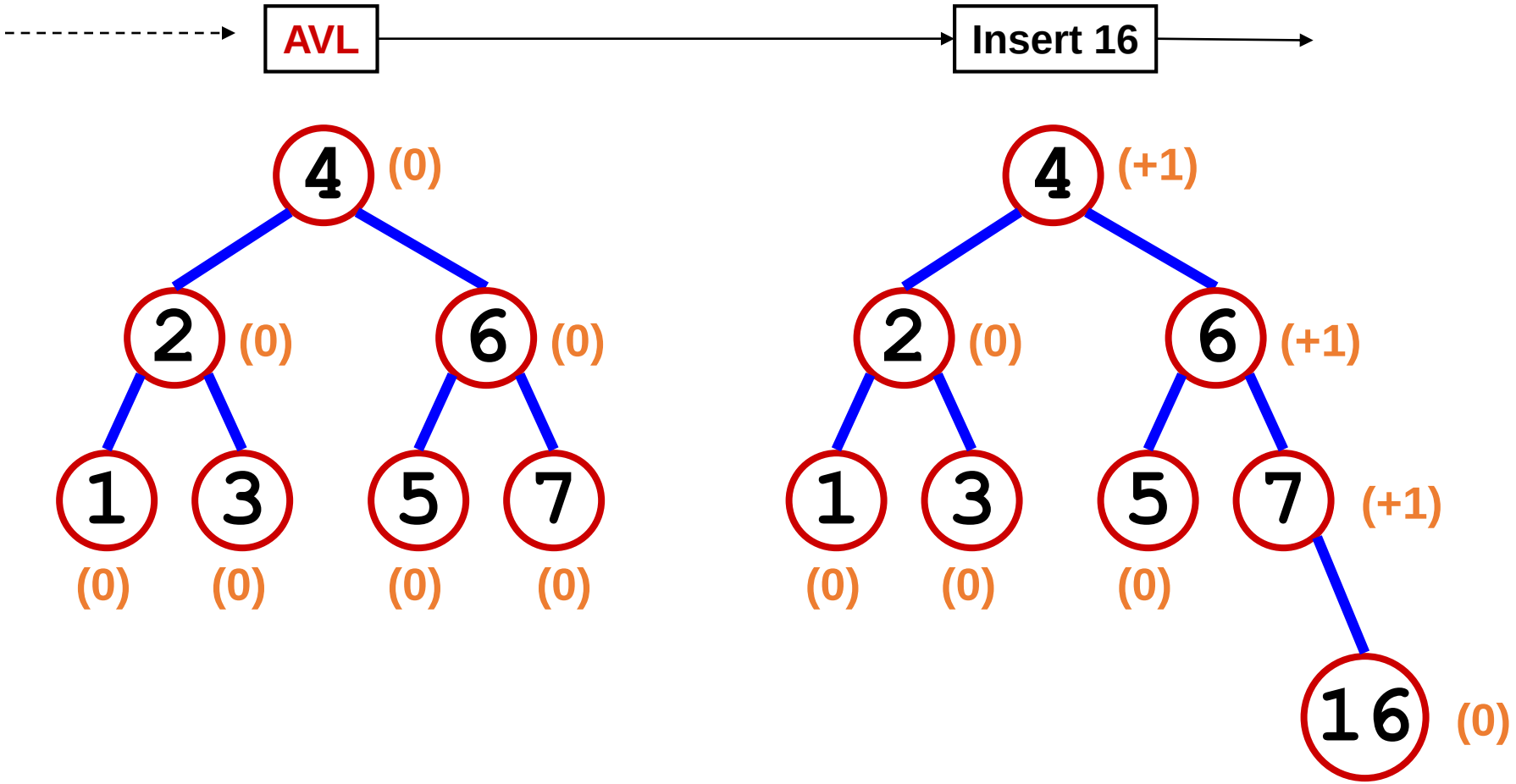

AVL
(non-AVL)
Insert 7
4
(0)
(0)
2
5
(+1)
(0)
1
3
(0)
6
(0)
4
(+1)
Pivot
(0)
2
5
(+2)
(0)
1
3
(0)
6
(+1)
7
(0)

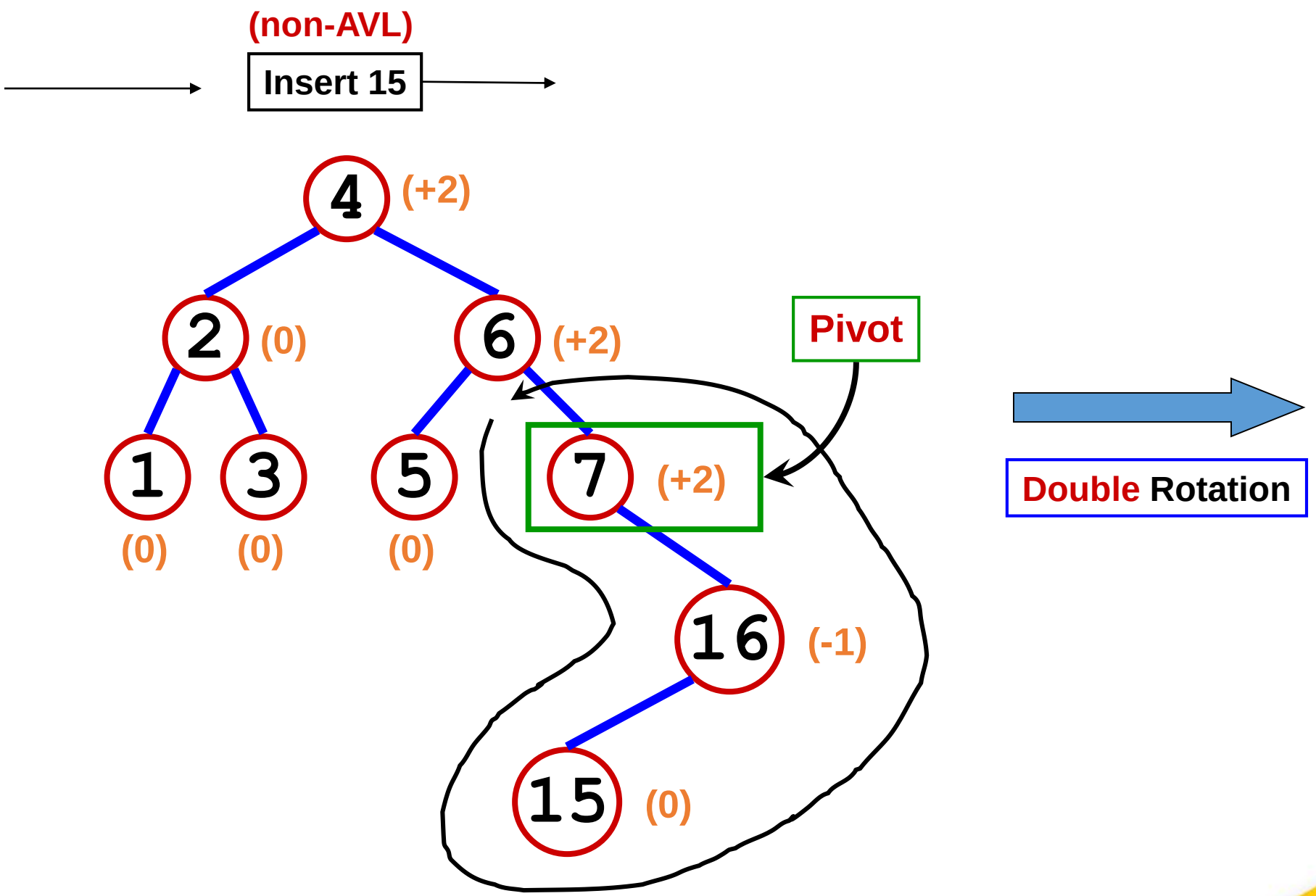


Single Rotation



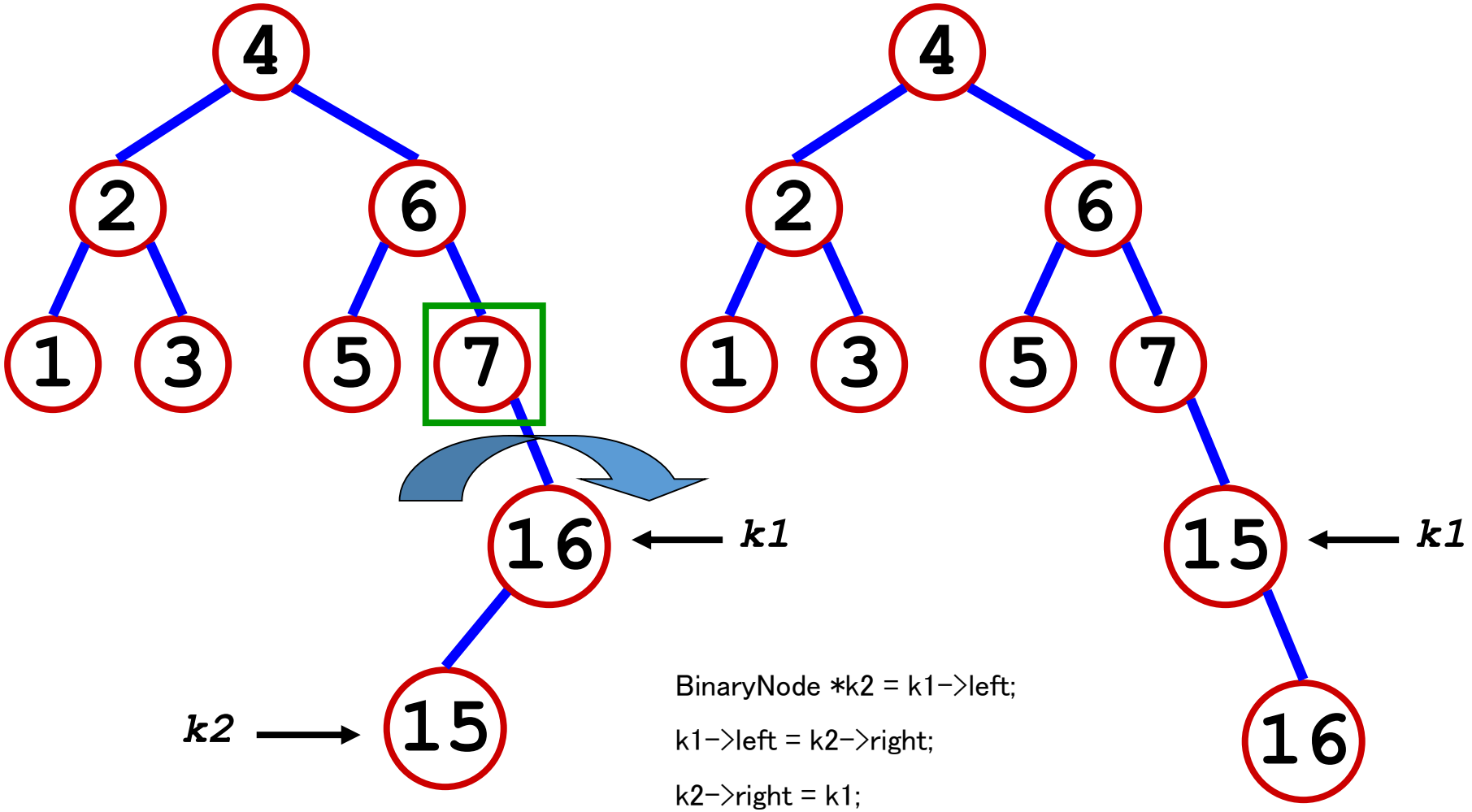

```
void LeftRotate( BinaryNode * & k1 )
{
        BinaryNode *k2 = k1->right;
        k1->right = k2->left;
        k2->left = k1;
        k1 = k2;
}
```


AVL
Insert 16
4
(0)
2
(0)
6
(0)
1
3
5
7
(0)
(0)
(0)
(0)
4
(+1)
2
(0)
6
(+1)
1
3
5
7
(+1)
(0)
(0)
(0)
16
(0)


(non-AVL)
Insert 15
4
(+2)
2
(0)
6
(+2)
Pivot
1
3
5
7
(+2)
(0)
(0)
(0)
16
(-1)
15
(0)
Double Rotation

Double Rotation

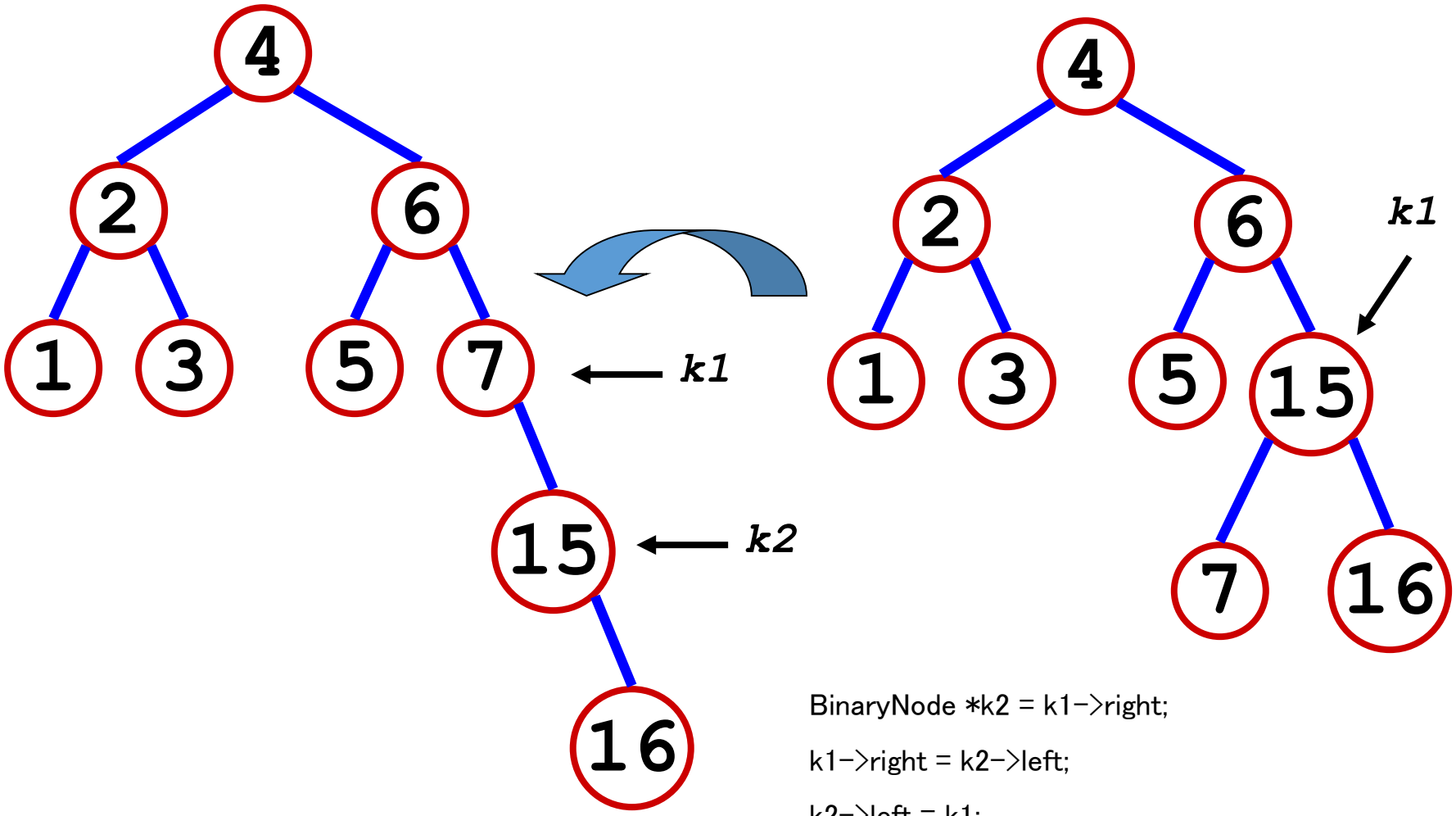
Step 1: Rotate child and grandchild



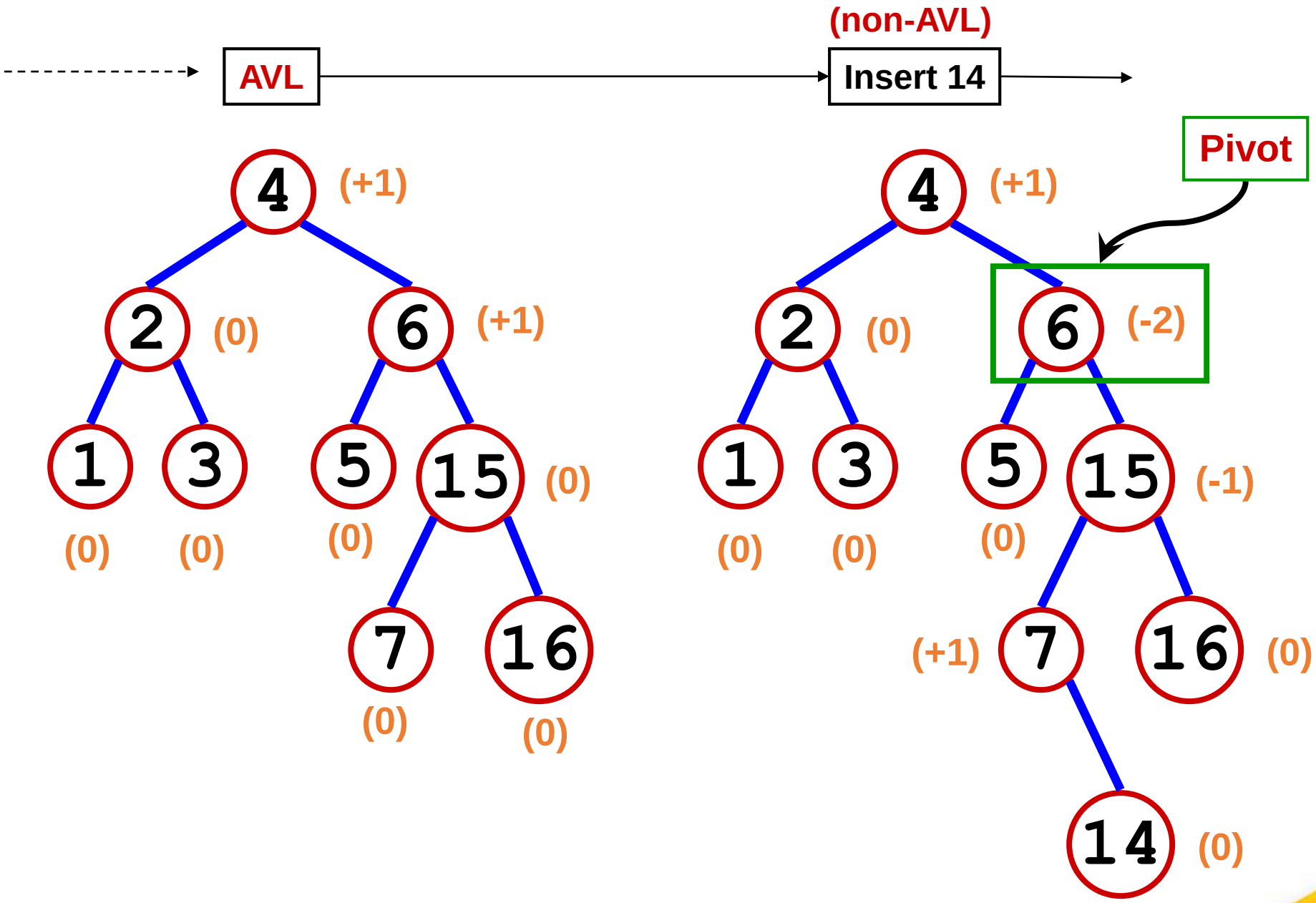

```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

Step 2: Rotate node and new child (AVL)

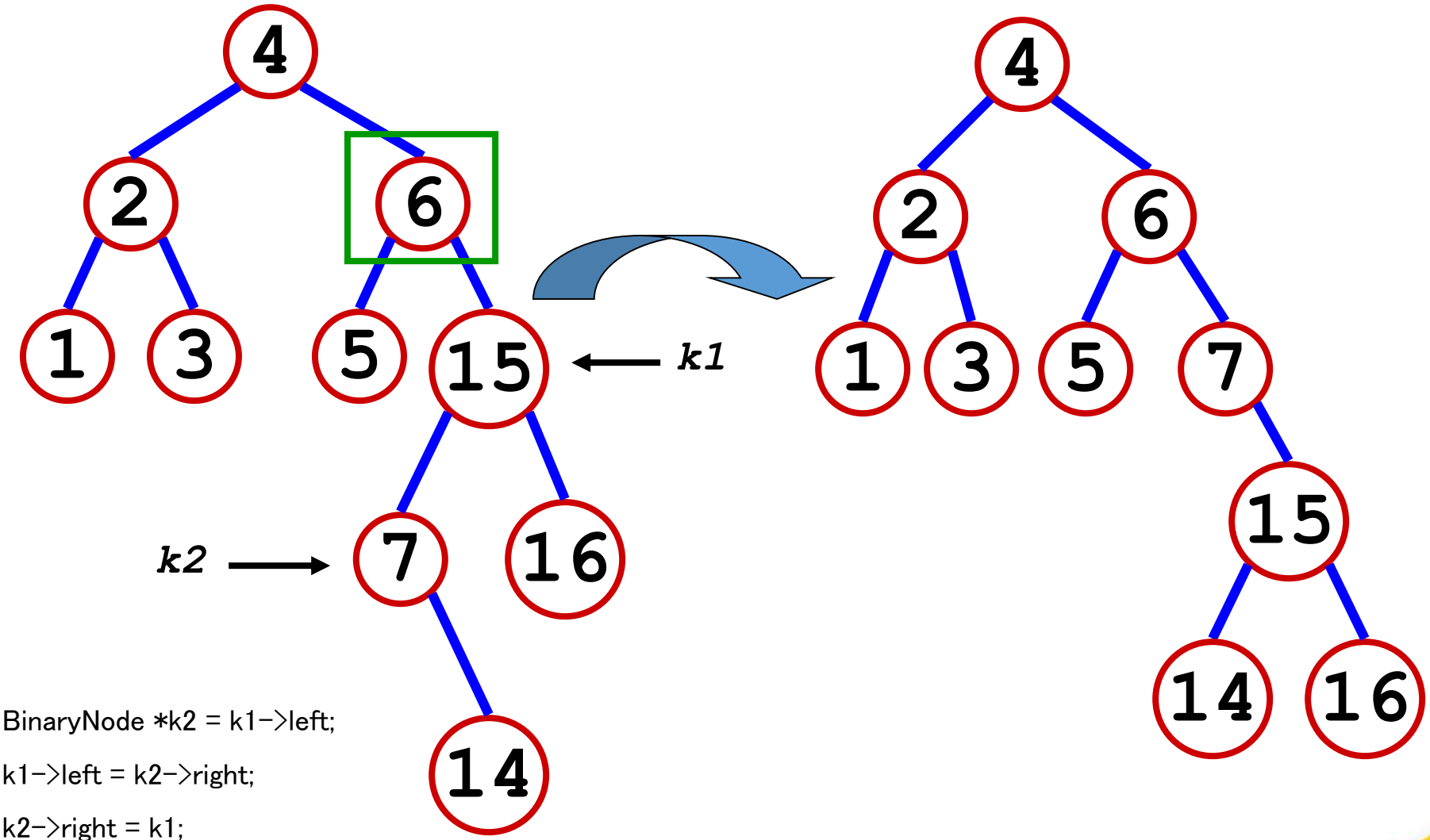



```
BinaryNode *k2 = k1->right;
k1->right = k2->left;
k2->left = k1;
k1 = k2;
```


AVL
Insert 14
(non-AVL)
Pivot
4
(+1)
2
(0)
6
(+1)
1
3
5
15
(0)
7
16
(0)
(0)
(0)
(0)
(0)
(0)
4
(+1)
2
(0)
6
(-2)
1
3
5
15
(-1)
(0)
(0)
(0)
(+1)
7
16
(0)
14
(0)

Double Rotation

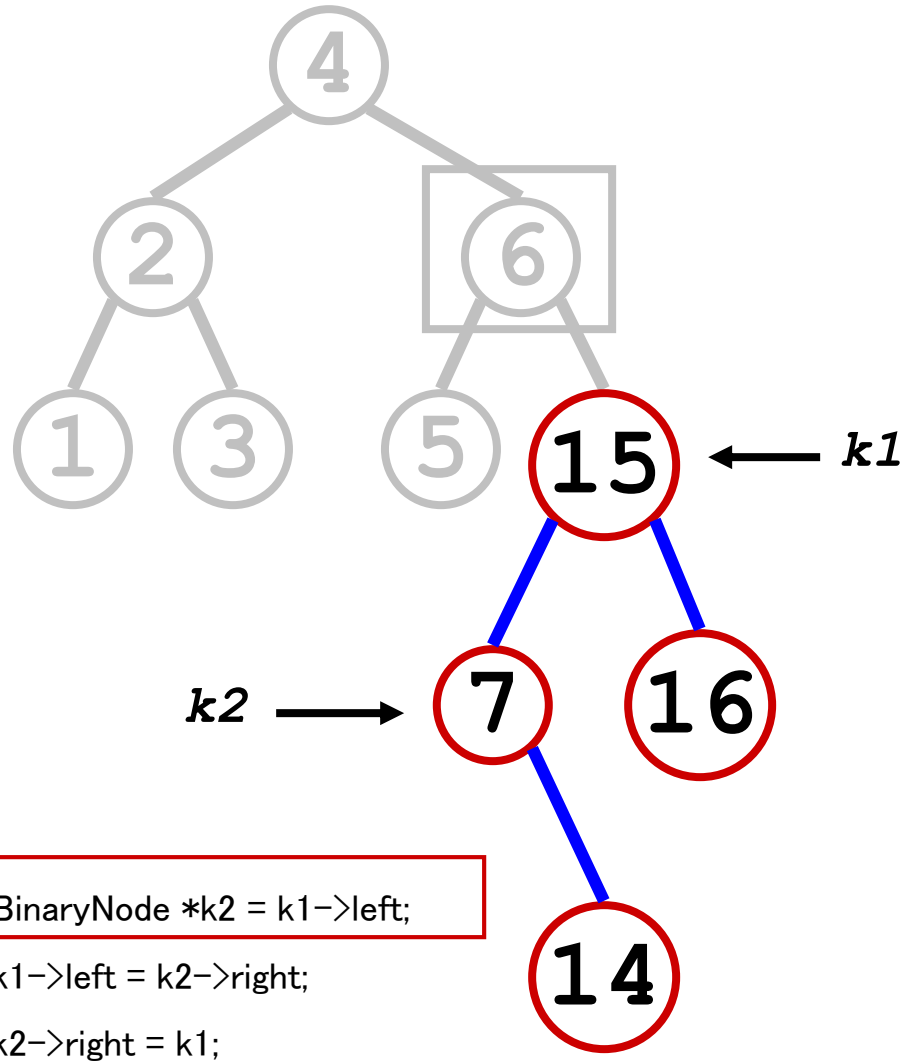
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

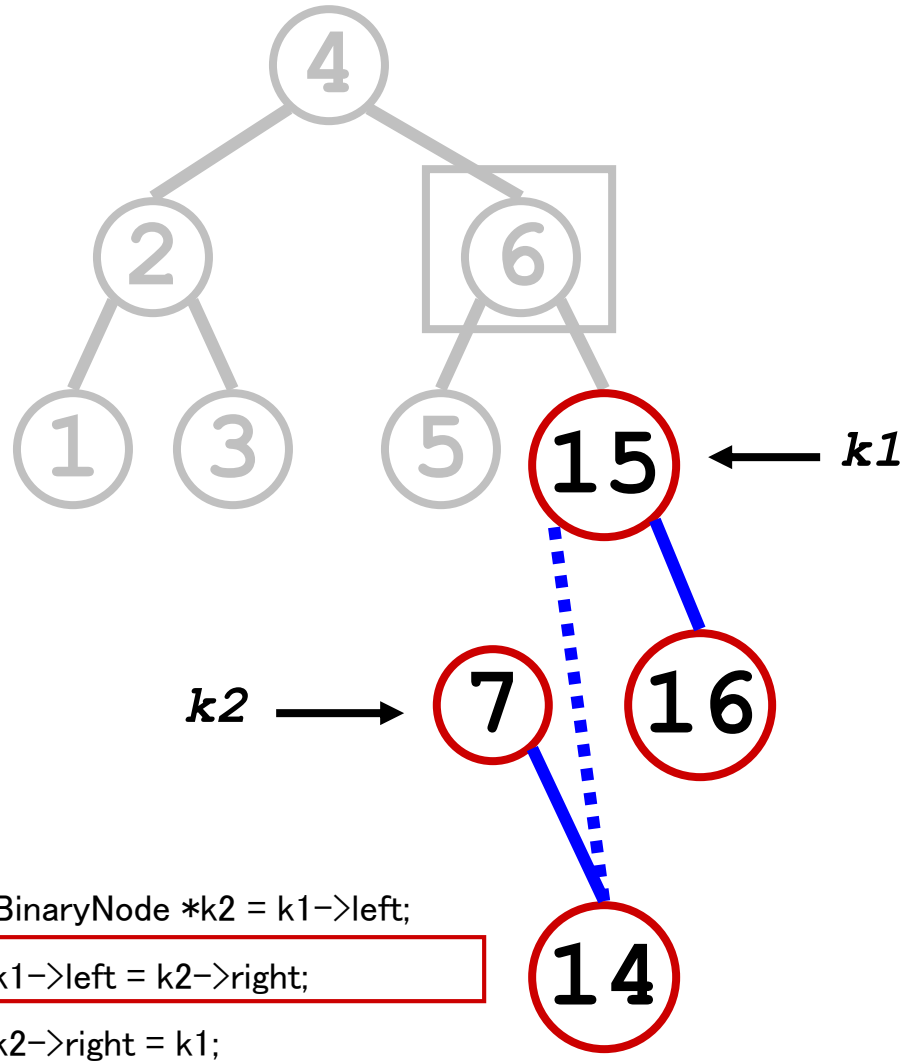
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

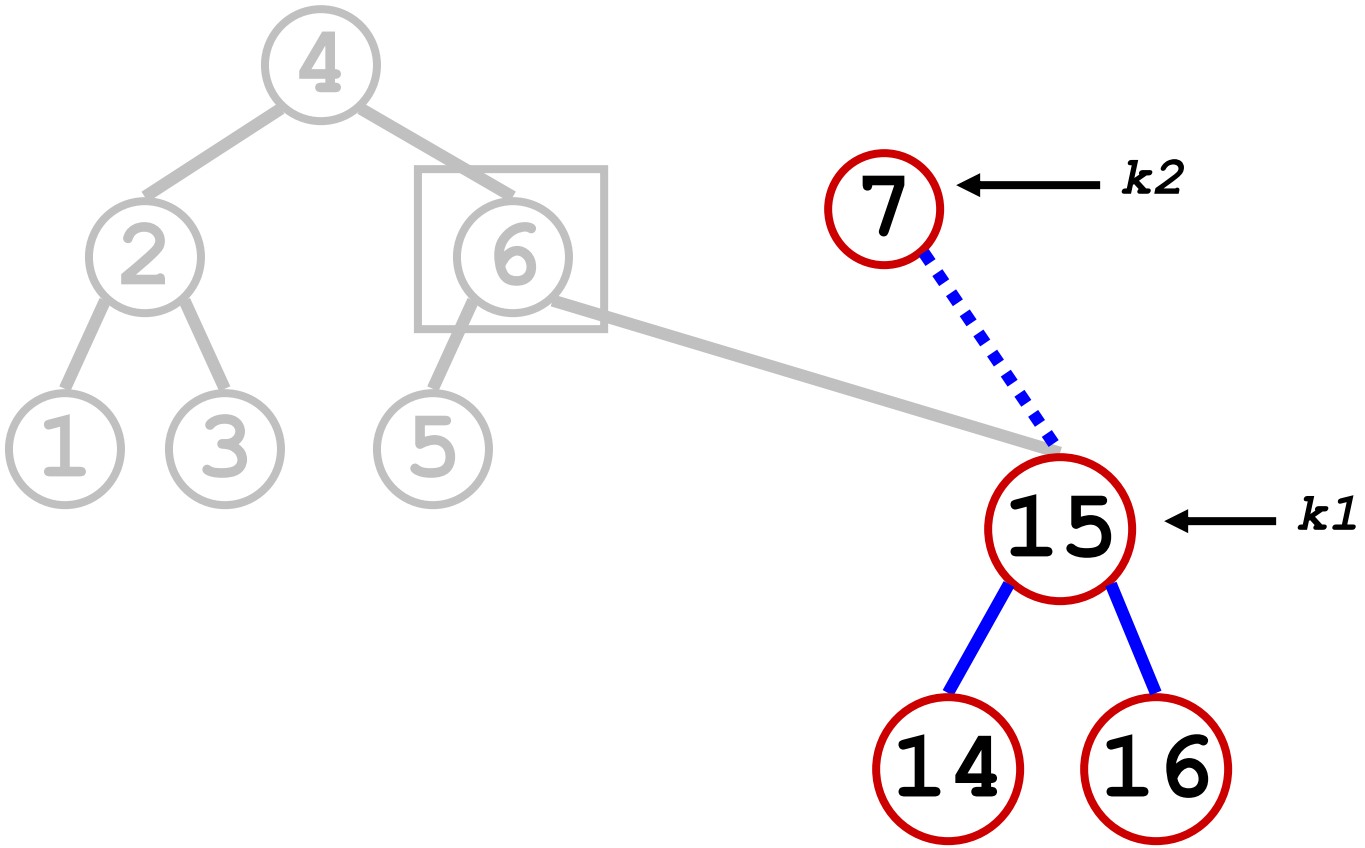
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

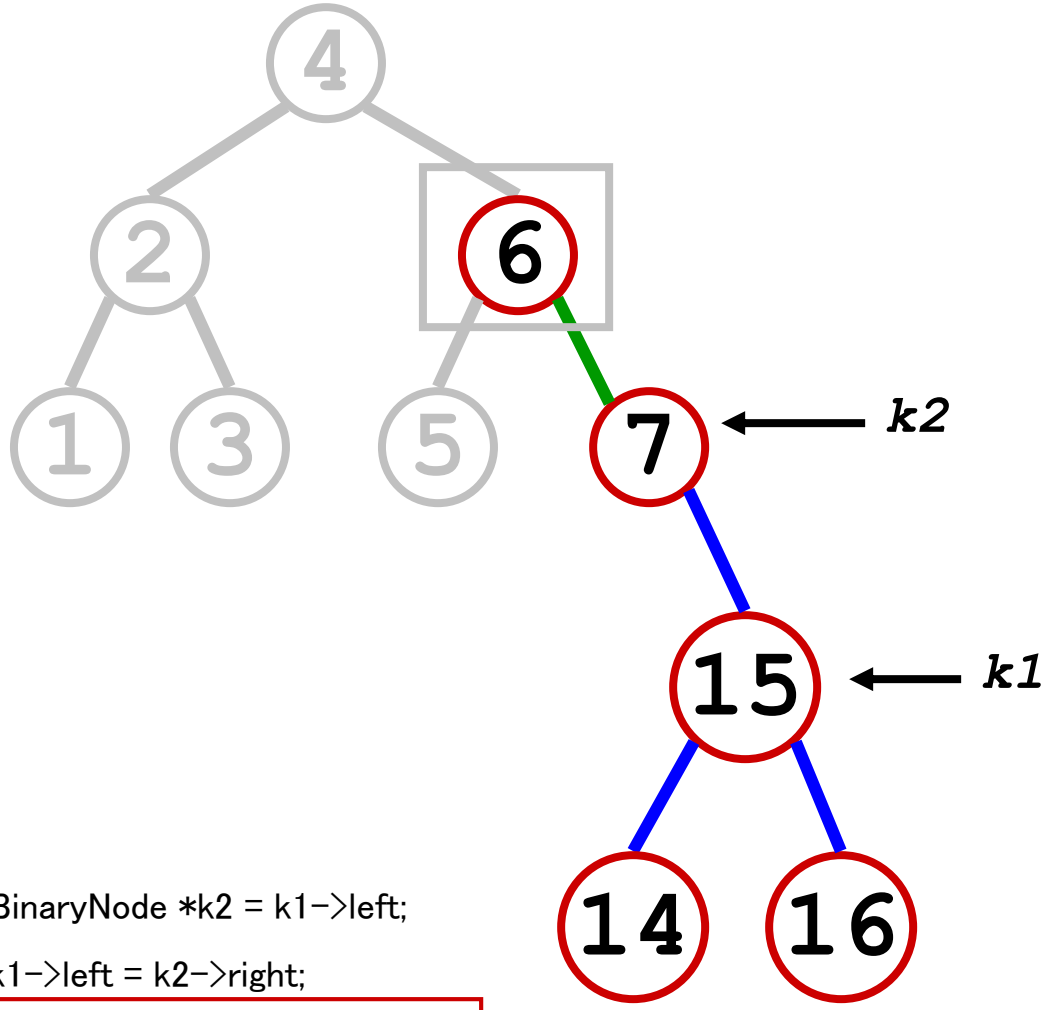
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

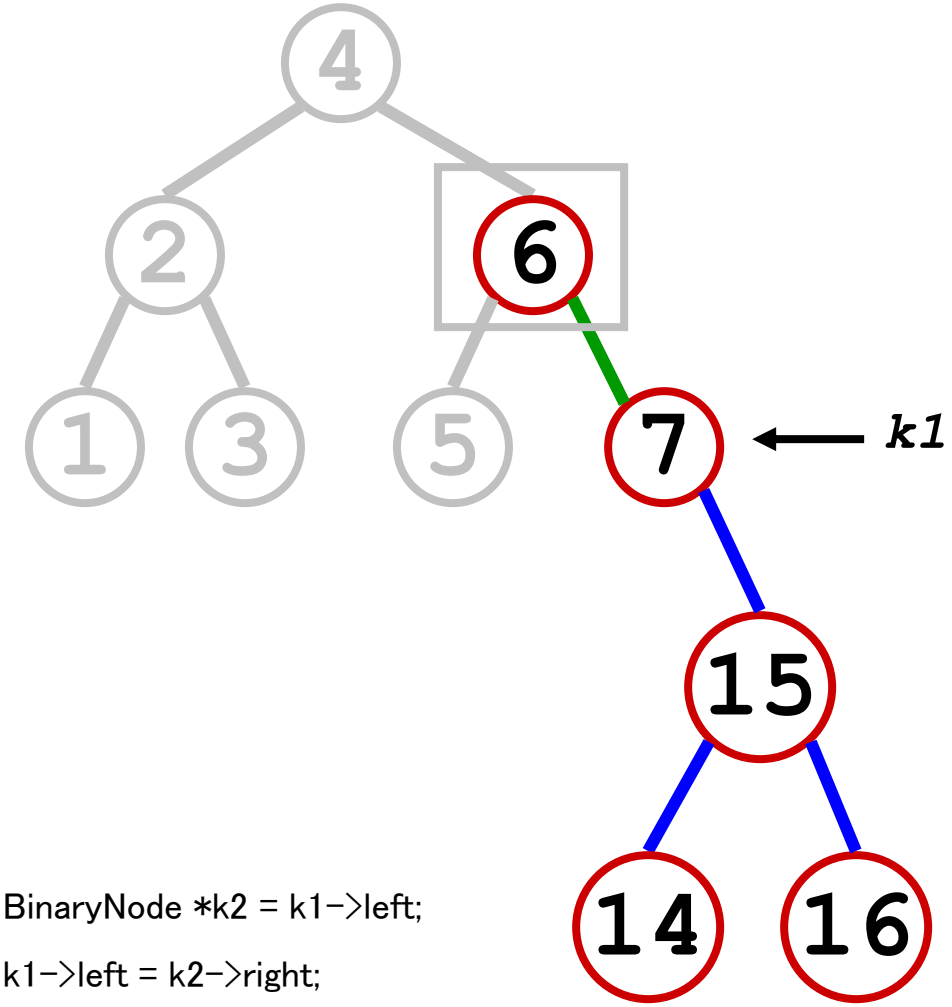
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

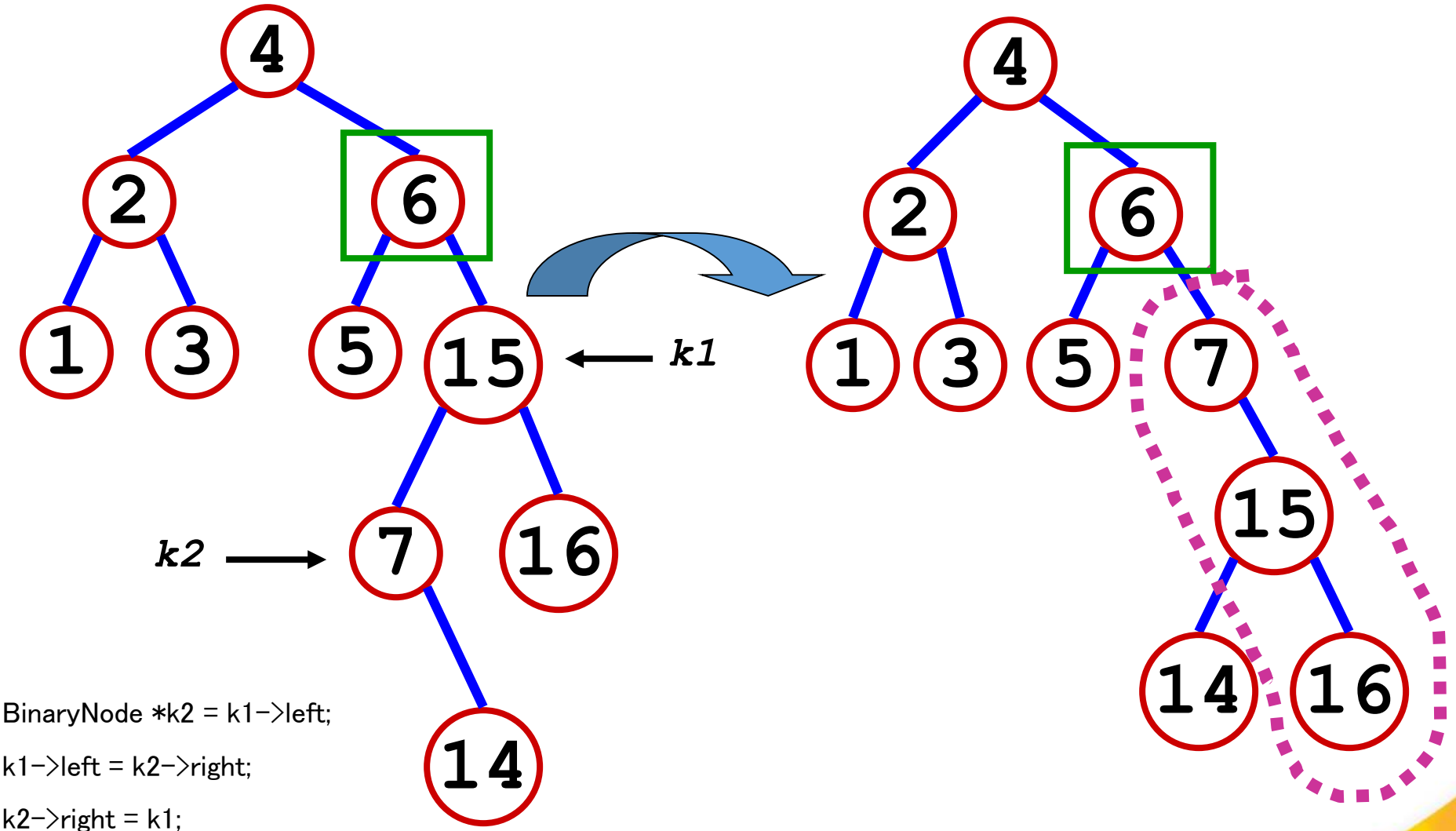
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

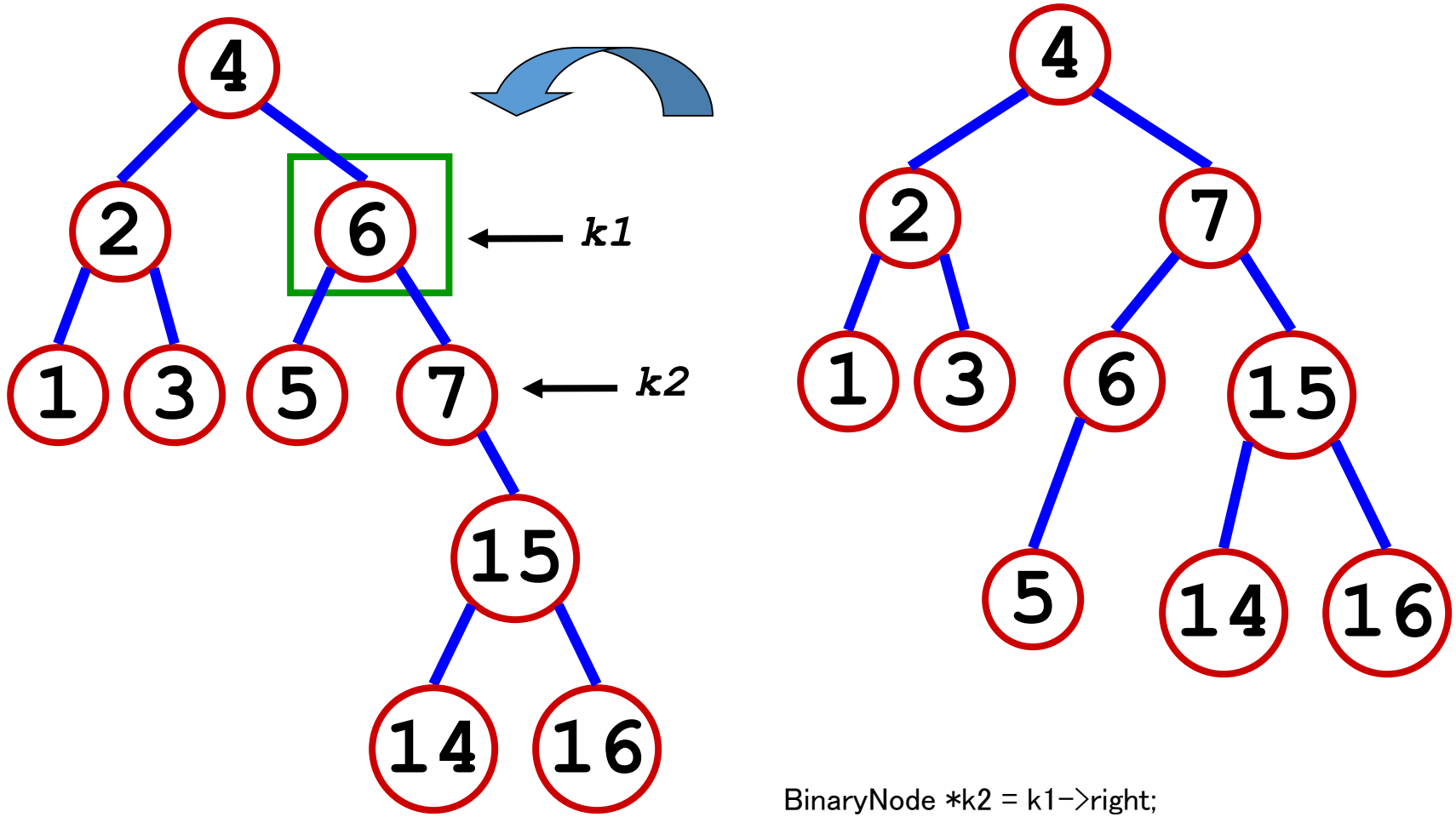
Step 1: Rotate child and grandchild




```
BinaryNode *k2 = k1->left;
k1->left = k2->right;
k2->right = k1;
k1 = k2;
```

Double Rotation

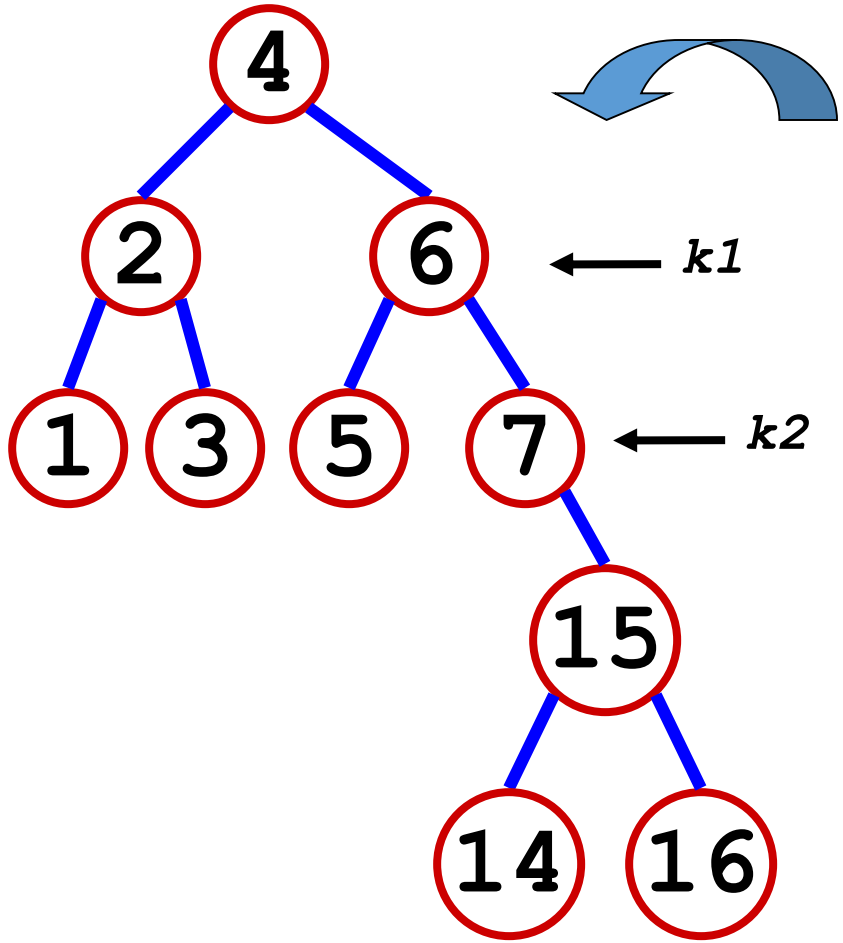
Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;
k1->right = k2->left;
k2->left = k1;
k1 = k2;
```

Double Rotation

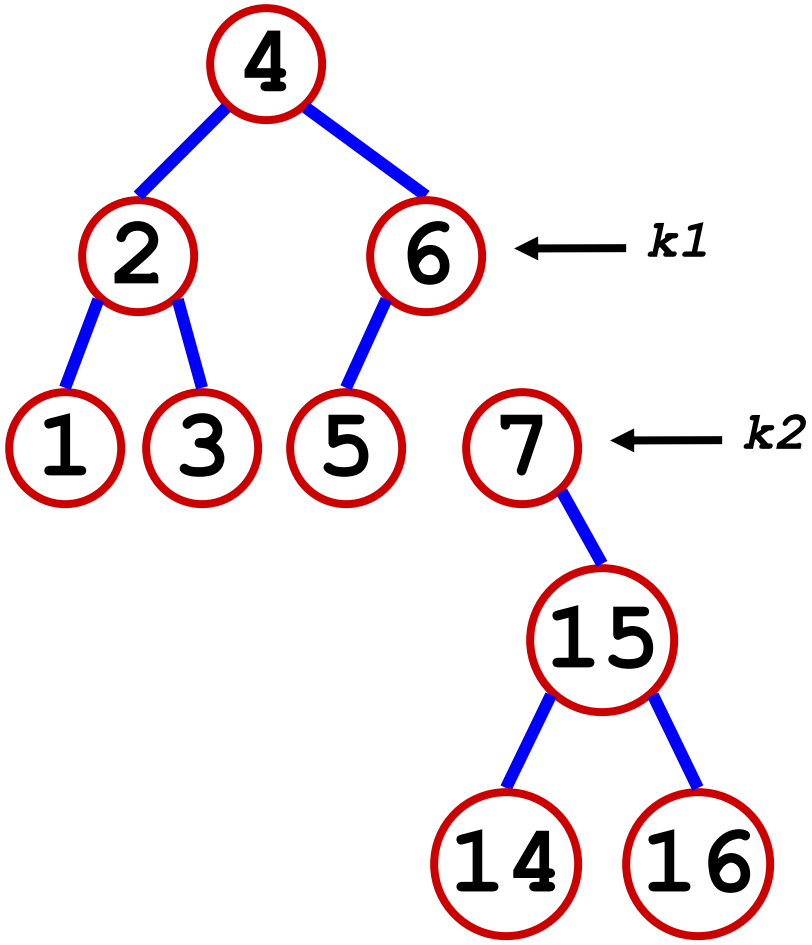
Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;
k1->right = k2->left;
k2->left = k1;
k1 = k2;
```

Double Rotation

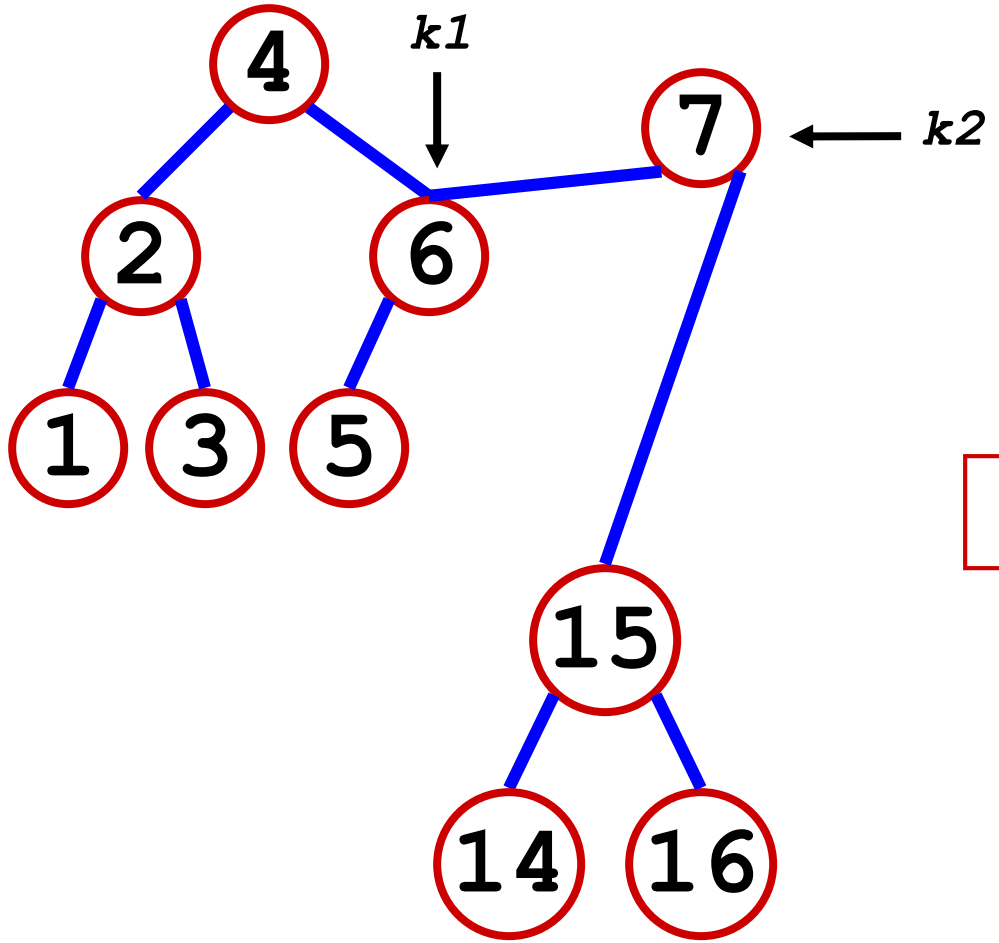
Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;

k1->right = k2->left;

k2->left = k1;

k1 = k2;
```

Double Rotation

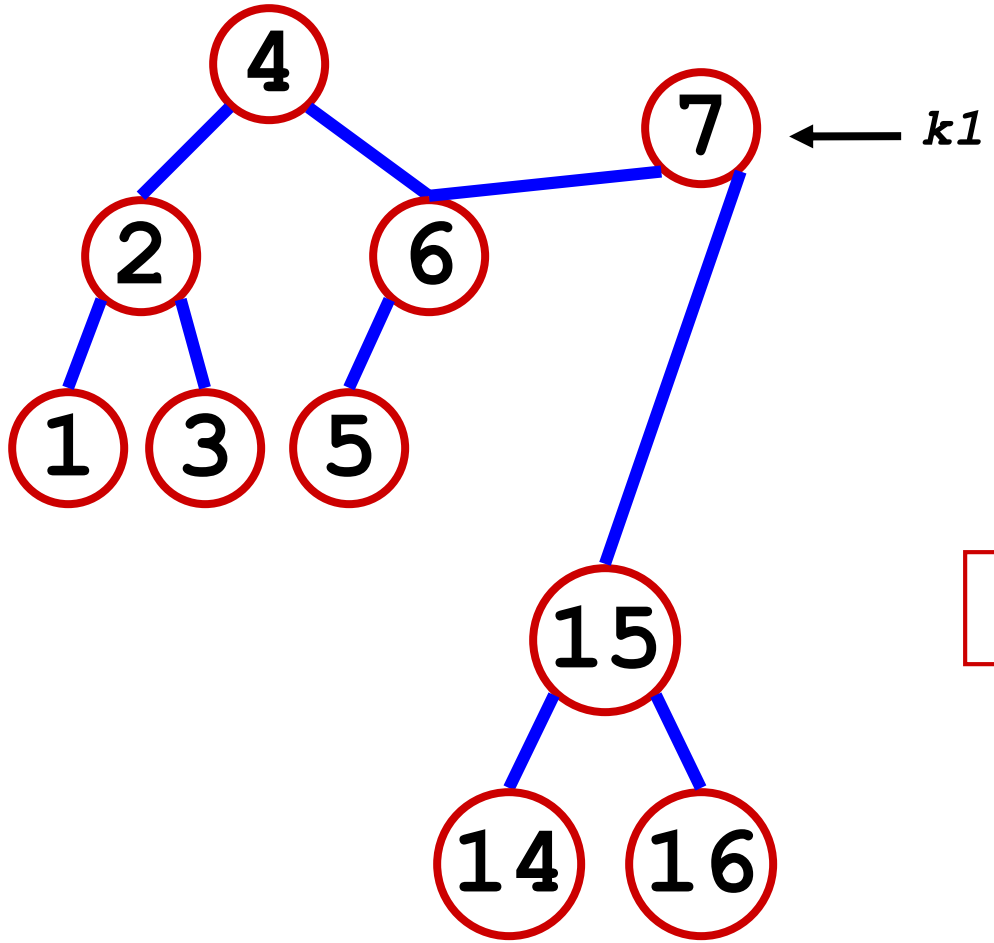
Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;
k1->right = k2->left;
k2->left = k1;
k1 = k2;
```

Double Rotation

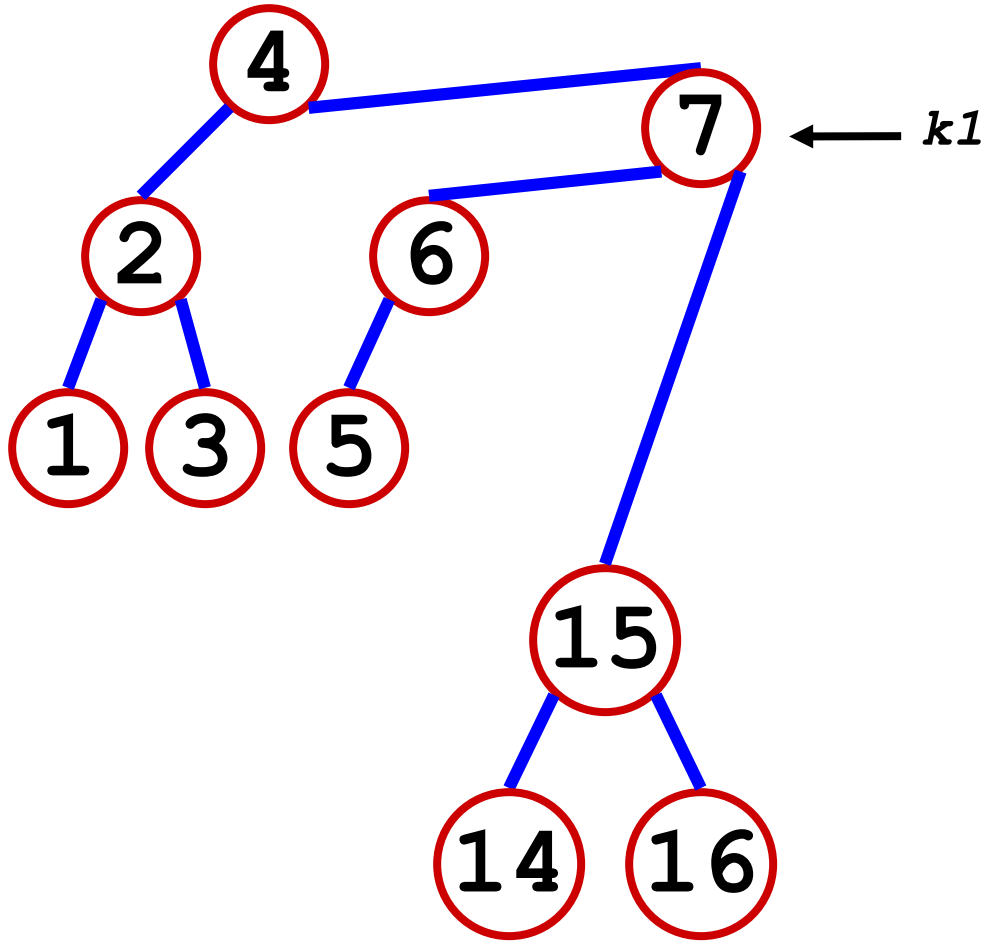
Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;

k1->right = k2->left;

k2->left = k1;

k1 = k2;
```

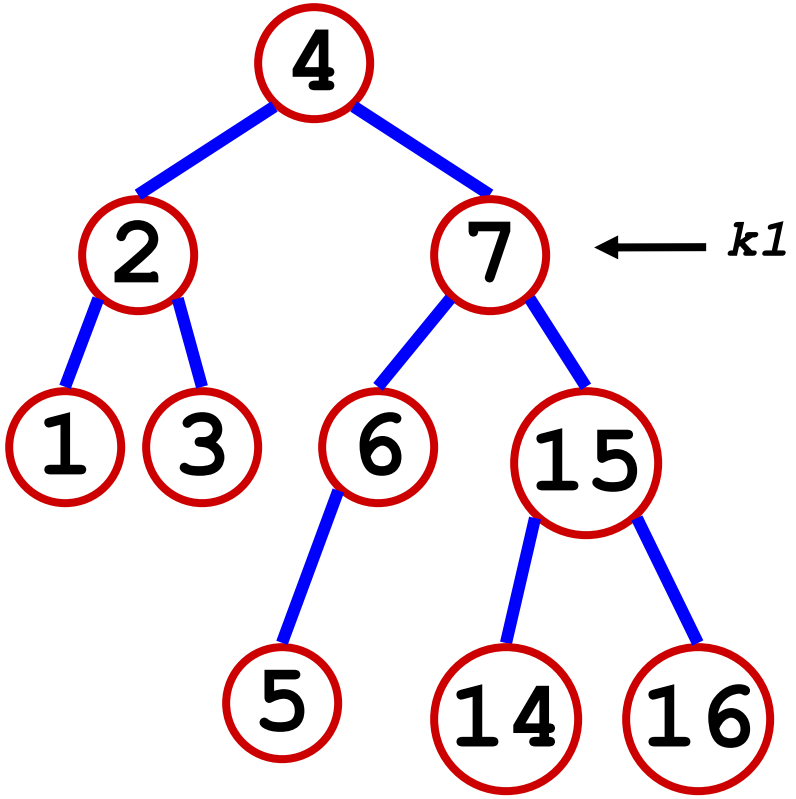
Double Rotation

Step 2: Rotate node and new child (AVL)



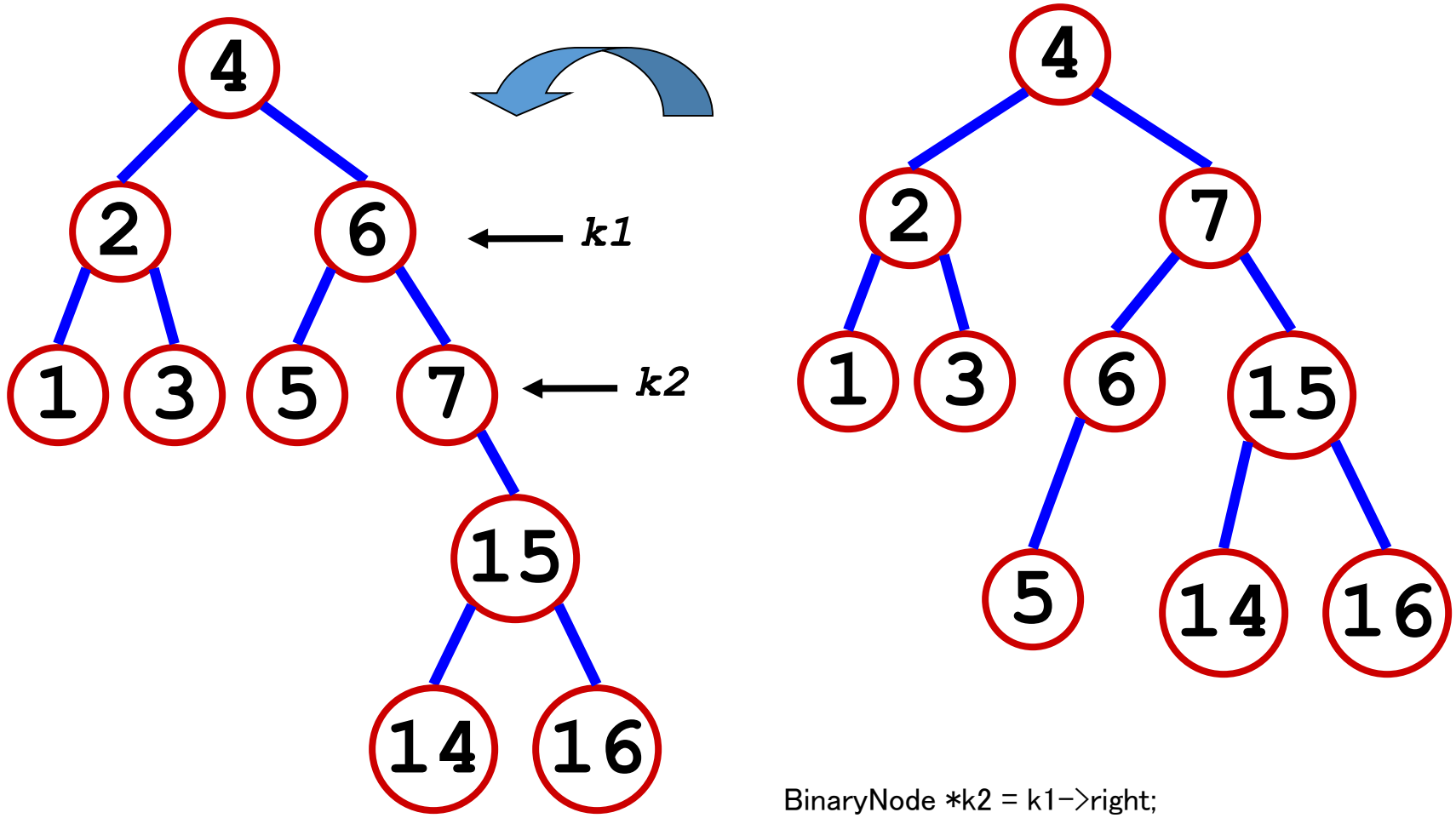
Double Rotation

Step 2: Rotate node and new child (AVL)



Double Rotation

Step 2: Rotate node and new child (AVL)




```
BinaryNode *k2 = k1->right;
k1->right = k2->left;
k2->left = k1;
k1 = k2;
```