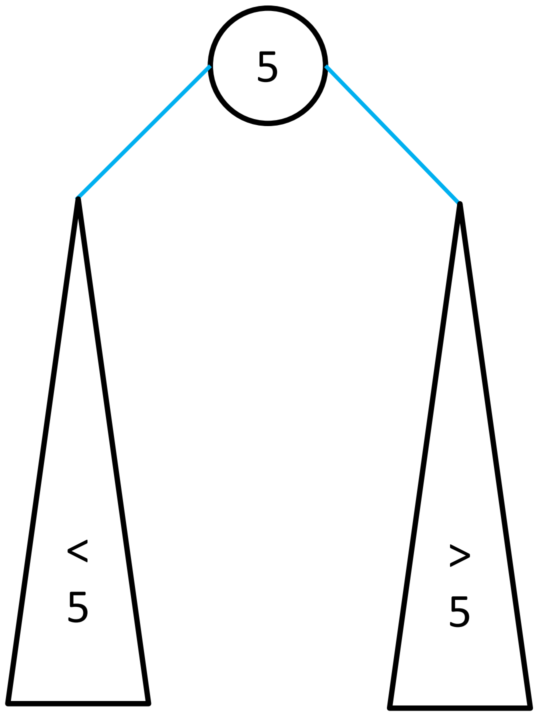
Binary Search Tree

Outline

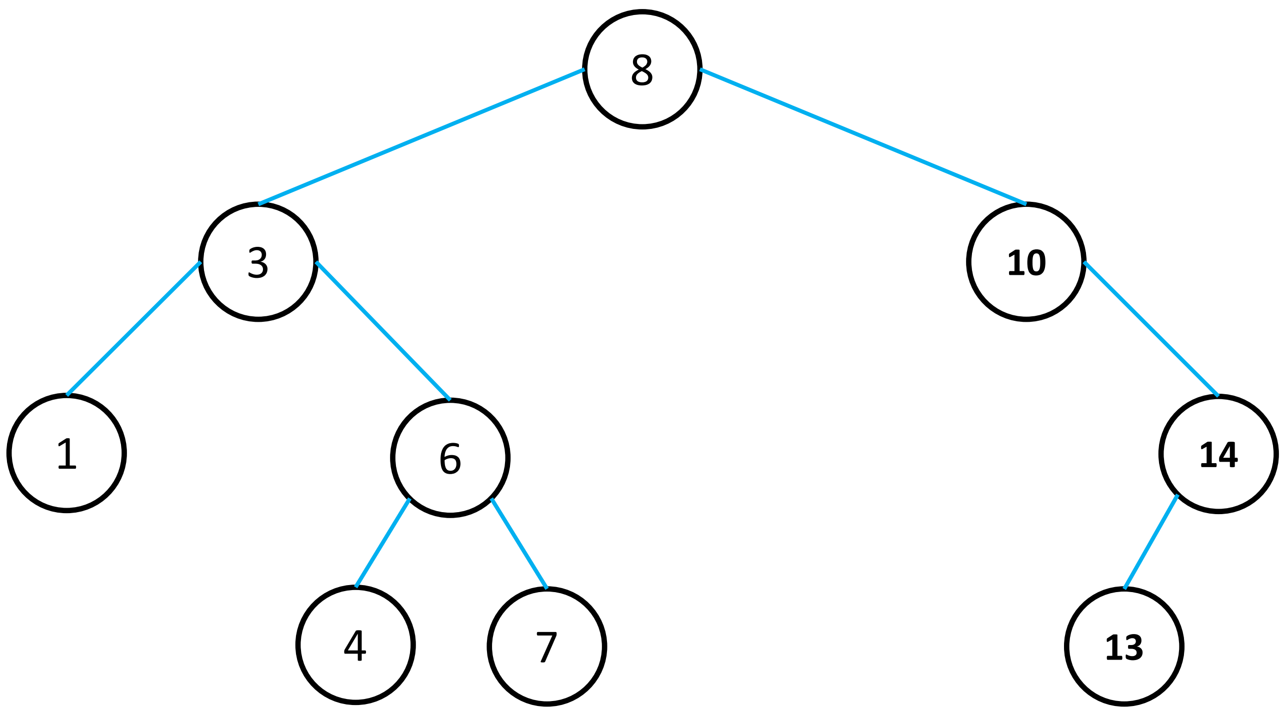
- Binary Search Tree Definition
- Binary Search Tree Operations
  - Finding an item
  - Insertion
  - Deletion
  - Rotation

Binary Search Tree

- A binary search tree (BST) is a binary tree in which
  - The values in the left subtree of a node are always less than the value in the node
  - The values in the right subtree of a node are all greater than the value of the node.
  - The subtrees of a binary search tree must themselves be binary search trees.
- Note that under this definition, a BST never contains duplicate nodes.



Binary Search Tree



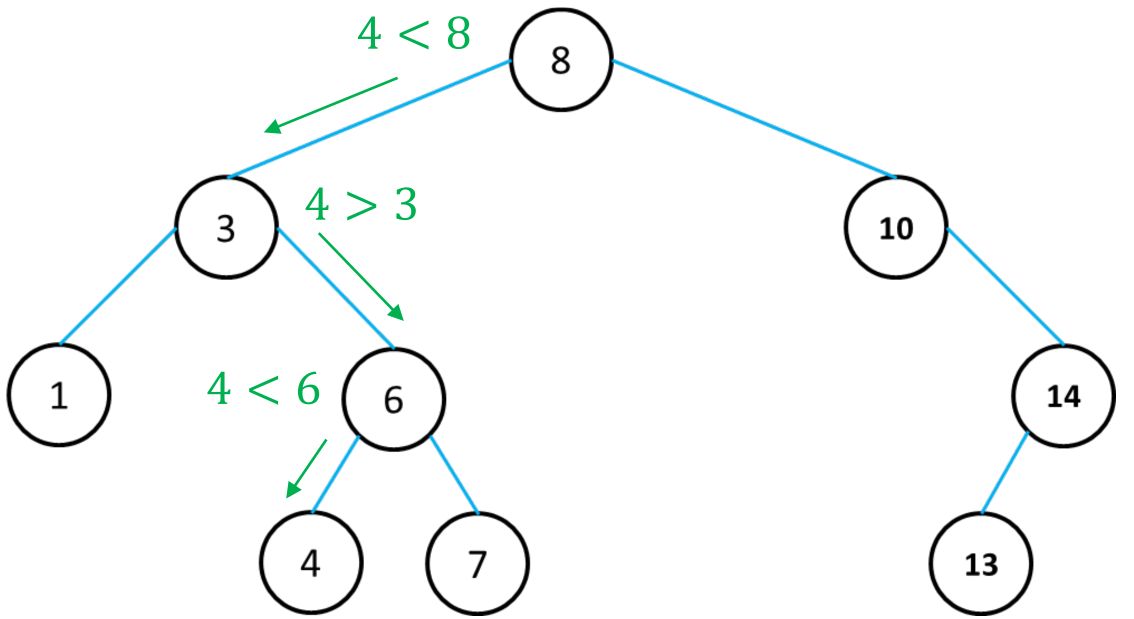
Left < Node < Right

Operations

- Some operations for BSTs:
  - Find an item
  - Insert an item
  - Delete an item
  - Rotation
  - Count the number of nodes
  - Find the height of the tree
  - Traverse(pre-order, in-order, post-order)

Finding an Item in a BST

- How to find 4 to the following BST?



Finding an Item in a BST

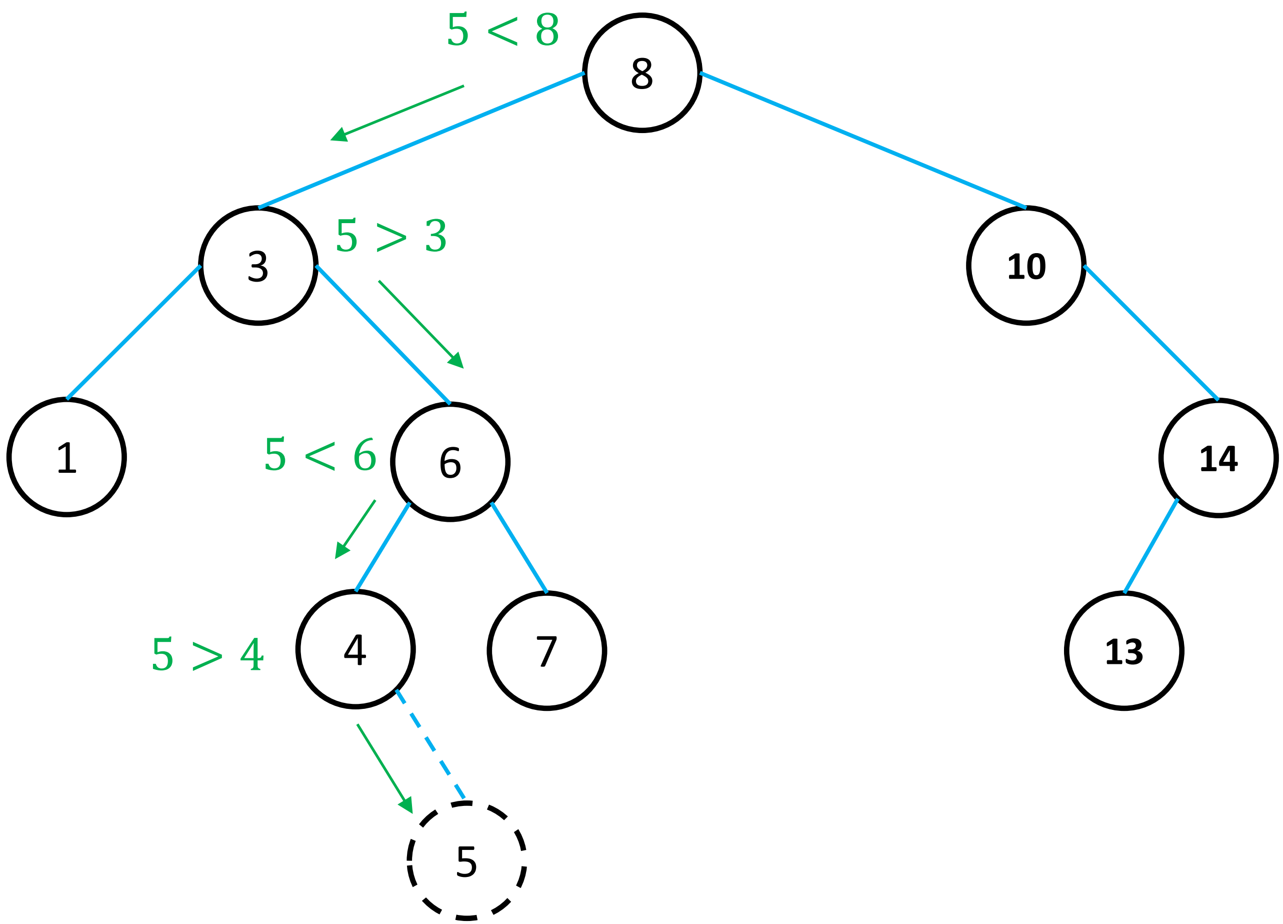
```
bool ItemExists(Tree tree, int Data){
    if (tree == 0)
        return false;
    else if (Data == tree->data)
        return true;
    else if (Data < tree->data)
        return ItemExists(tree->left, Data);
    else
        return ItemExists(tree->right, Data);
}
```

Finding an Item in a BST

- Complexity
  - Best case: $O(1)$
  - Worst case: $O(h)$, where h is the height of the tree.

Insert an Item in a BST

- How to insert 5 into the following BST?

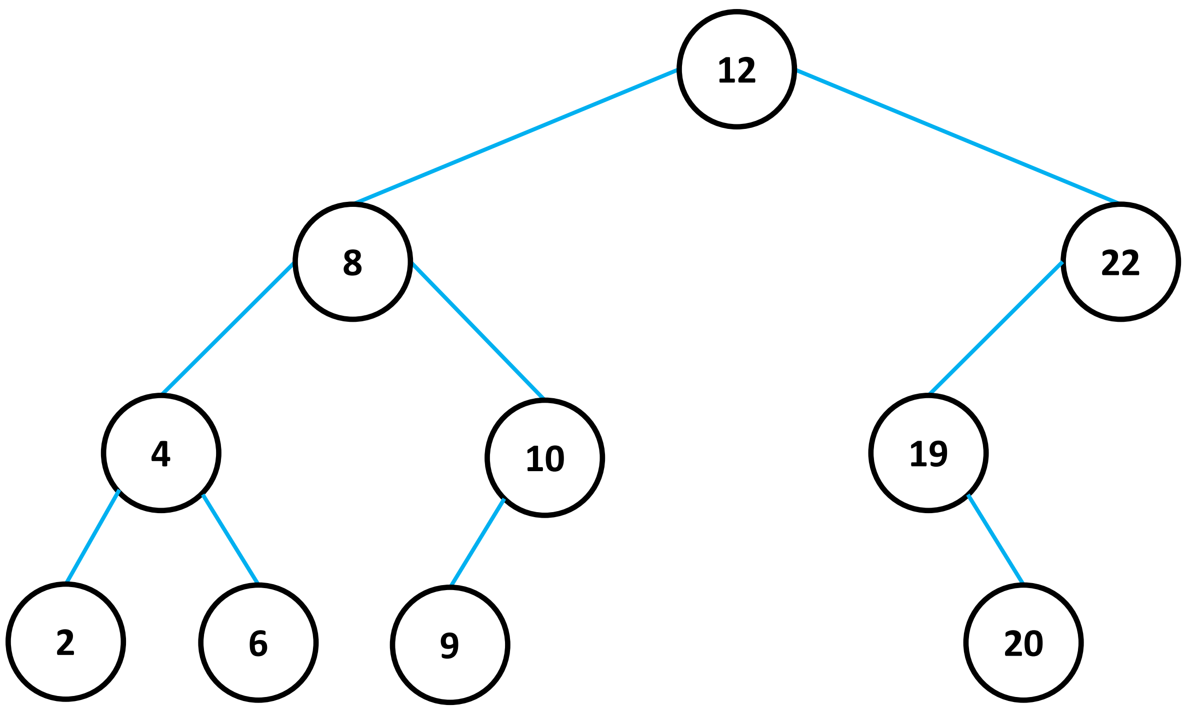


Insert an Item in a BST

```
void InsertItem(Tree &tree, int Data){
    if (tree == 0)
        tree = MakeNode(Data);
    else if (Data < tree->data)
        InsertItem(tree->left, Data);
    else if (Data > tree->data)
        InsertItem(tree->right, Data);
    else  // Data == tree->data
        cout << "Error, duplicate item" << endl;
}
```

Insert an Item in a BST

- Create a tree using these values (in this order):
  - 12, 22, 8, 19, 10, 9, 20, 4, 2, 6

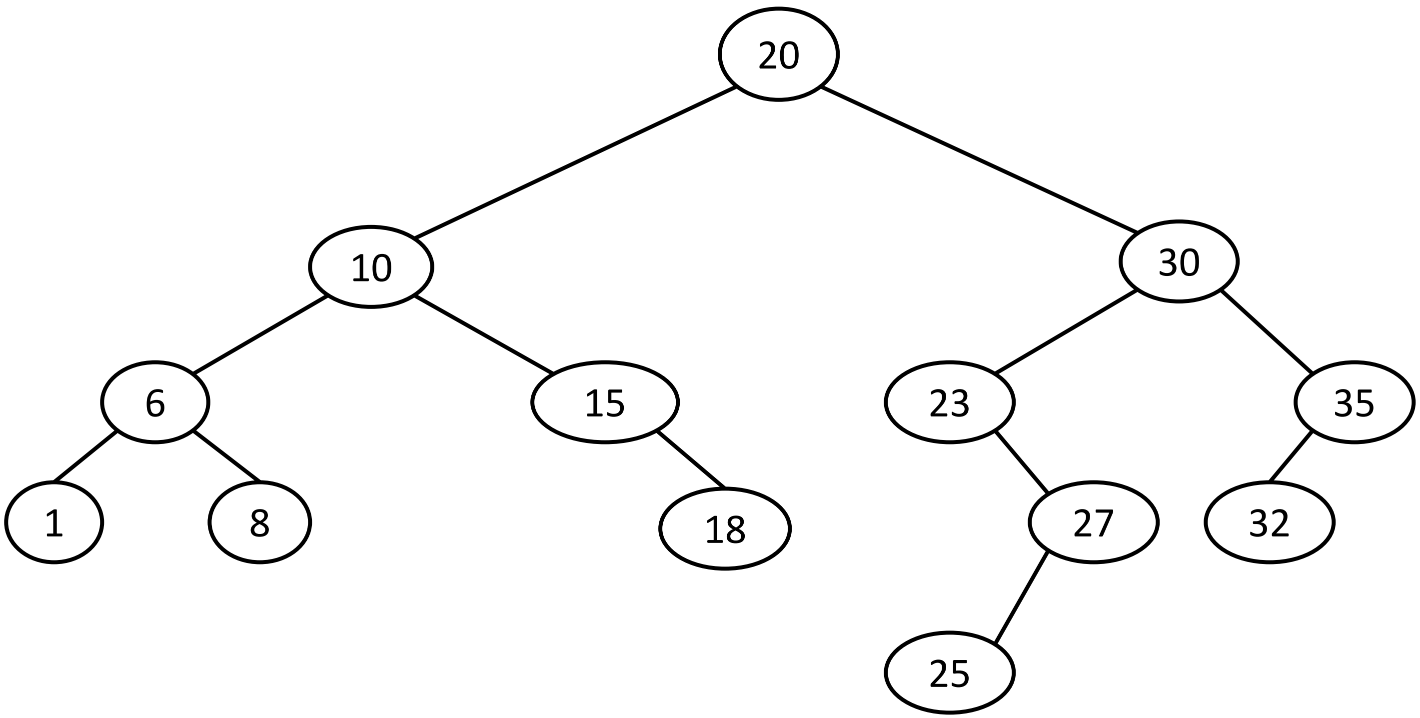


Insert an Item in a BST

- Complexity
  - Best case and worst case: $O(h)$, where h is the of the tree.

Deleting a Node

- The caveat of deleting a node is that, after deletion, the tree must still be a BST.

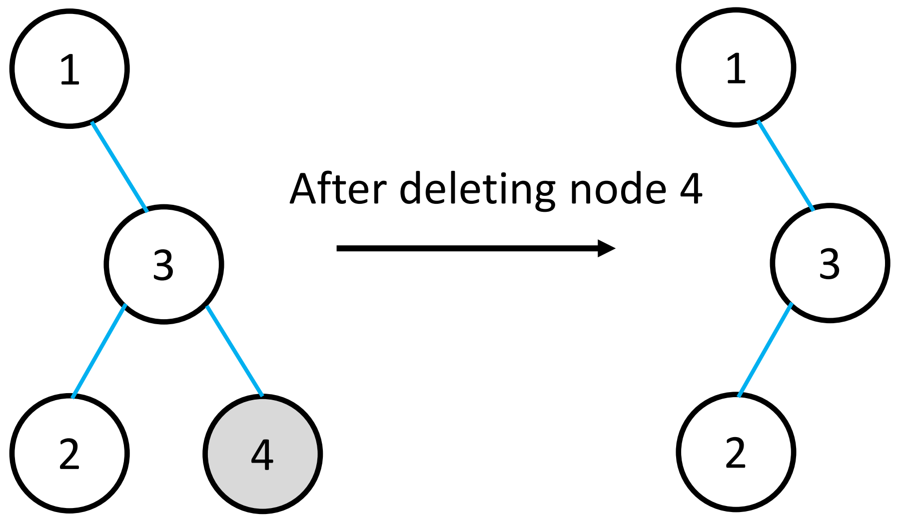


Deleting a Node

- Case 1: The node to be deleted is a leaf node.
- Case 2: The node to be deleted has an empty left child but non-empty right child.
- Case 3: The node to be deleted has an empty right child but non-empty left child.
- Case 4: The node has both children non-empty.

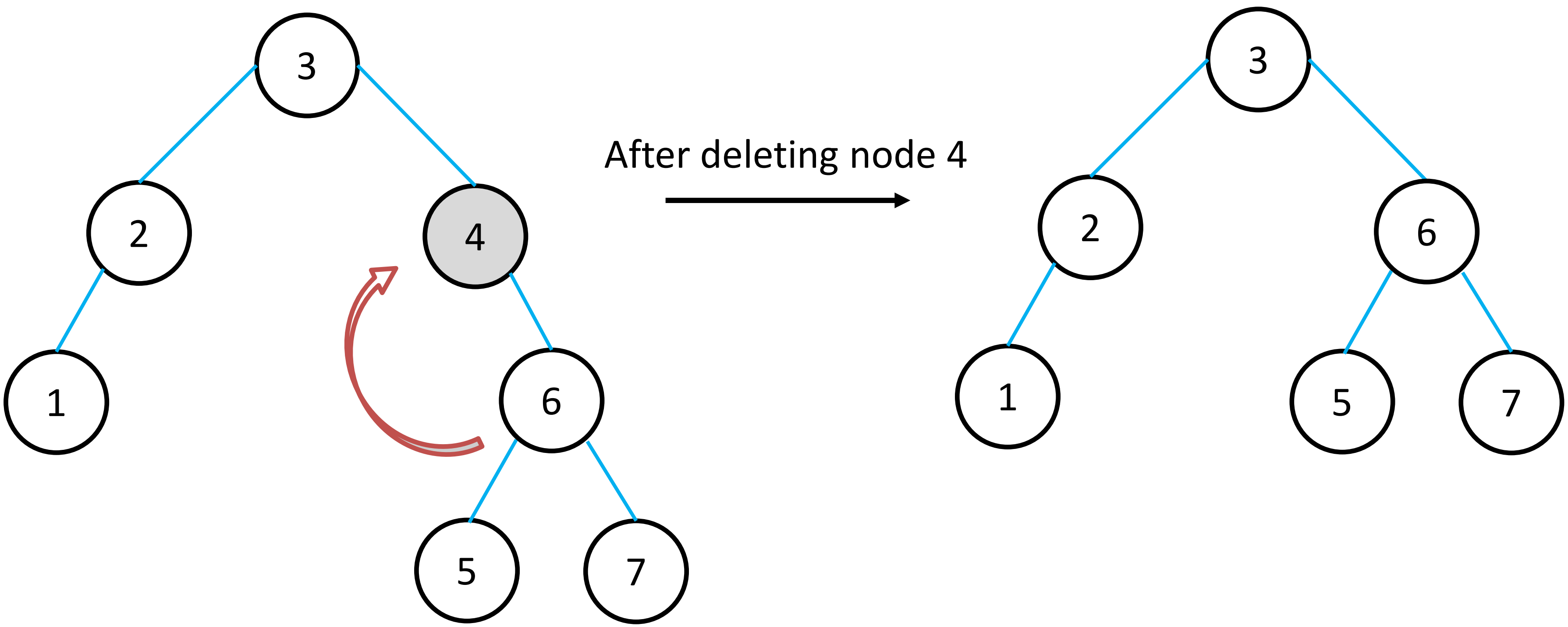
Case 1: Leaf Node

- Set the parent's pointer to this node to NULL.
- Release the memory of the leaf node.



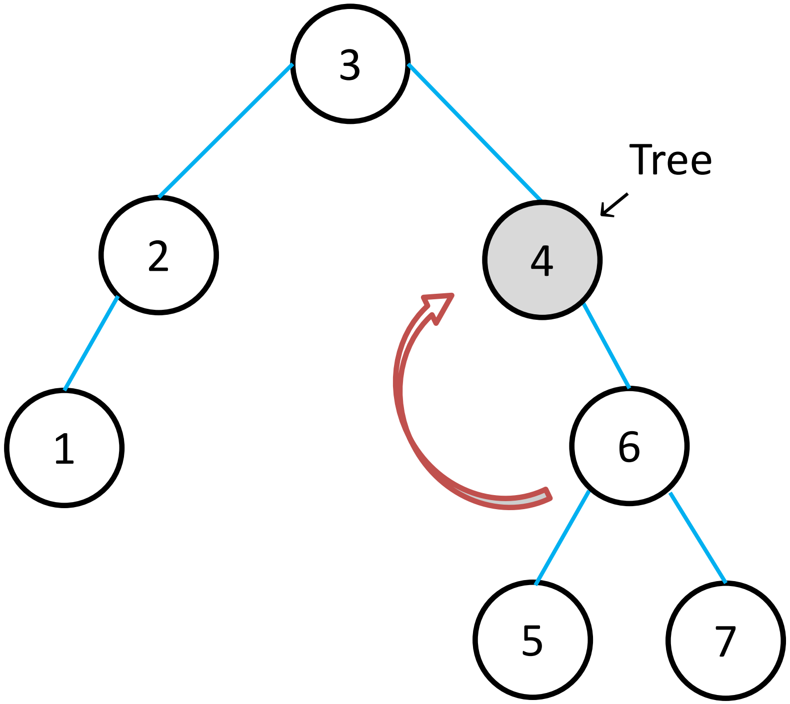
Case 2: Empty Left Child

- Replace the deleted node with its right child.



Case 2: Empty Left Child

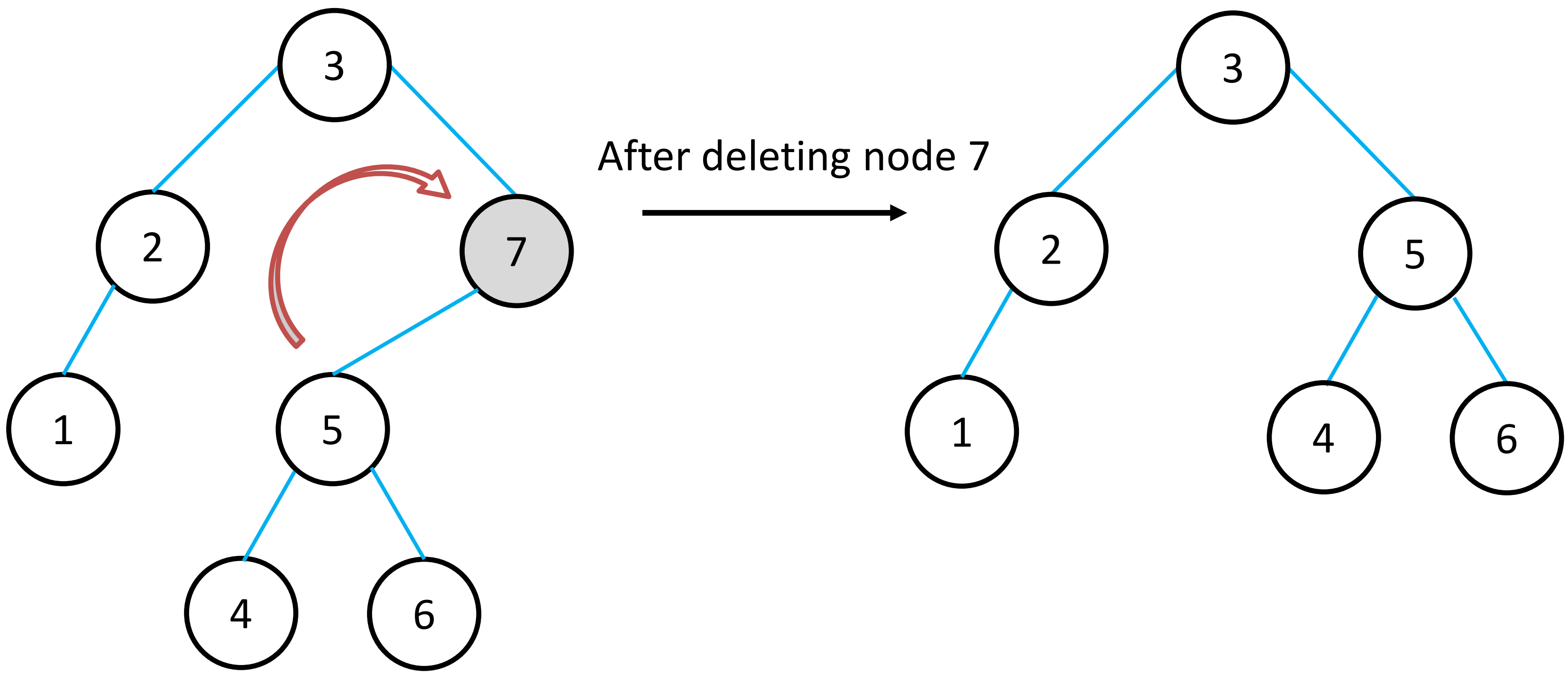
- Replace the deleted node with its right child.




```
if (tree->left == 0){
    Tree temp = tree;
    tree = tree->right;
    FreeNode(temp);
}
```

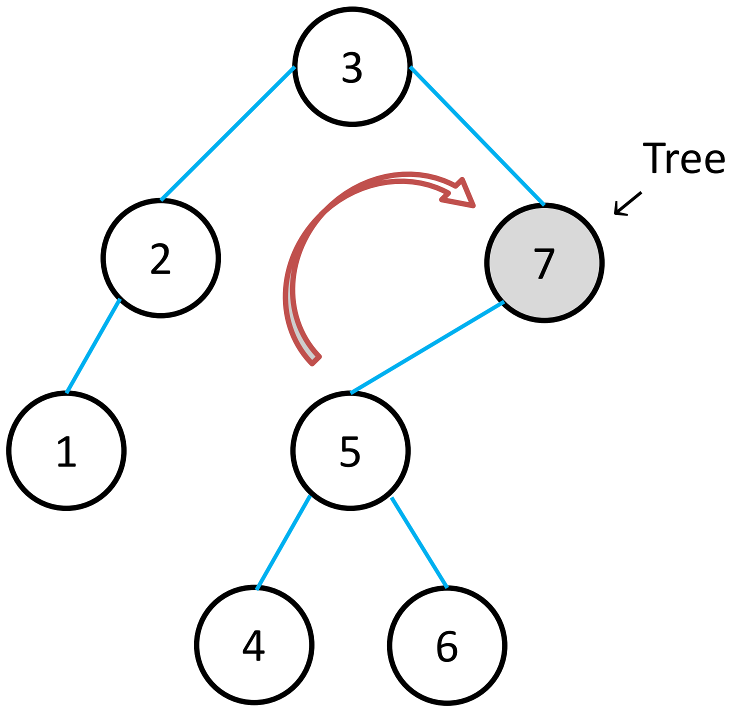
Case 3: Empty Right Child

- Replace the deleted node with its left child.



Case 3: Empty Right Child

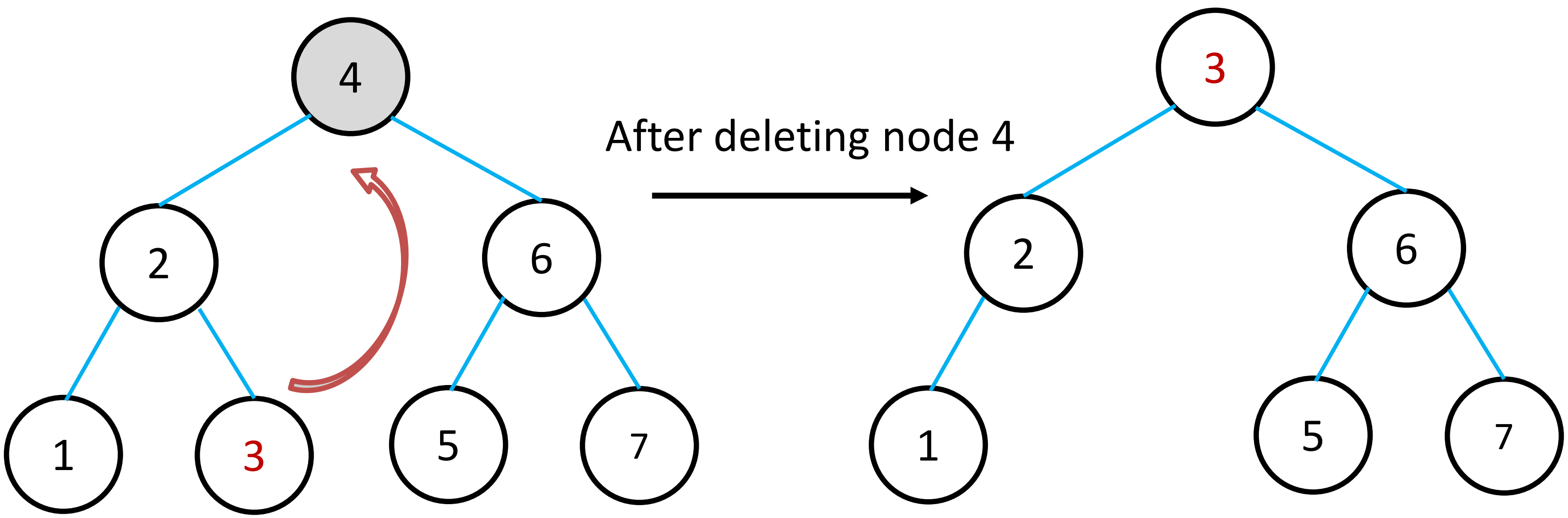
- Replace the deleted node with its left child.




```
if (tree->right == 0){
    Tree temp = tree;
    tree = tree->left;
    FreeNode(temp);
```

Case 4: Non-empty Left and Right Child

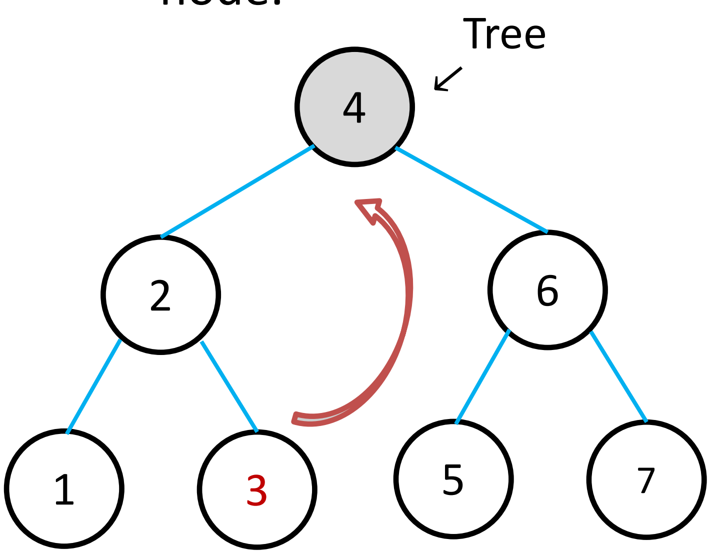
- Replace the data in the deleted node with its predecessor (or successor) under in-order traversal.
- Delete the node that holds the predecessor.



In-order Traversal order: 1 2 3 4 5 6 7

Case 4: Non-empty Left and Right Child

- Why predecessor?
  - Recall for binary search tree, left < node < right.
  - An in-order traversal (left, then node, then right) will always produce a sequence of values in increasing numerical order.
  - Therefore, the predecessor is the maximum value in left subtree of the node.



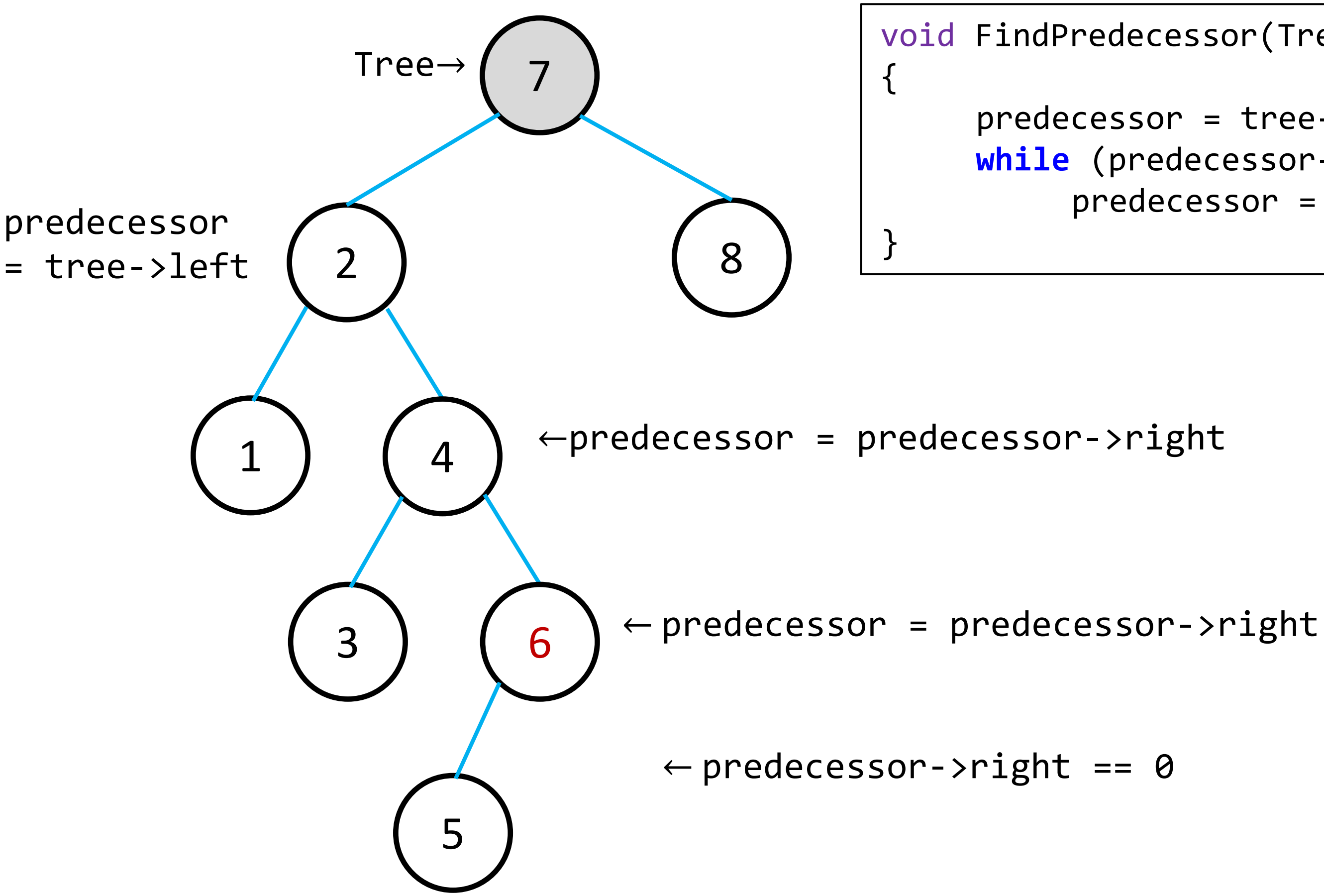
In-order Traversal order: 1 2 3 4 5 6 7

```
else{
    Tree pred = 0;
    FindPredecessor(tree, pred);
    tree->data = pred->data;
    DeleteItem(tree->left, tree->data);
}
```

- We are replacing the data in the node, not the node itself.
- The predecessor is still in the tree, so we still need to delete it.

Find the Predecessor

- Note that the predecessor is the rightmost node in the left subtree.




```
void FindPredecessor(Tree tree, Tree &predecessor)
{
	predecessor = tree->left;
	while (predecessor->right != 0)
		predecessor = predecessor->right;
}
```

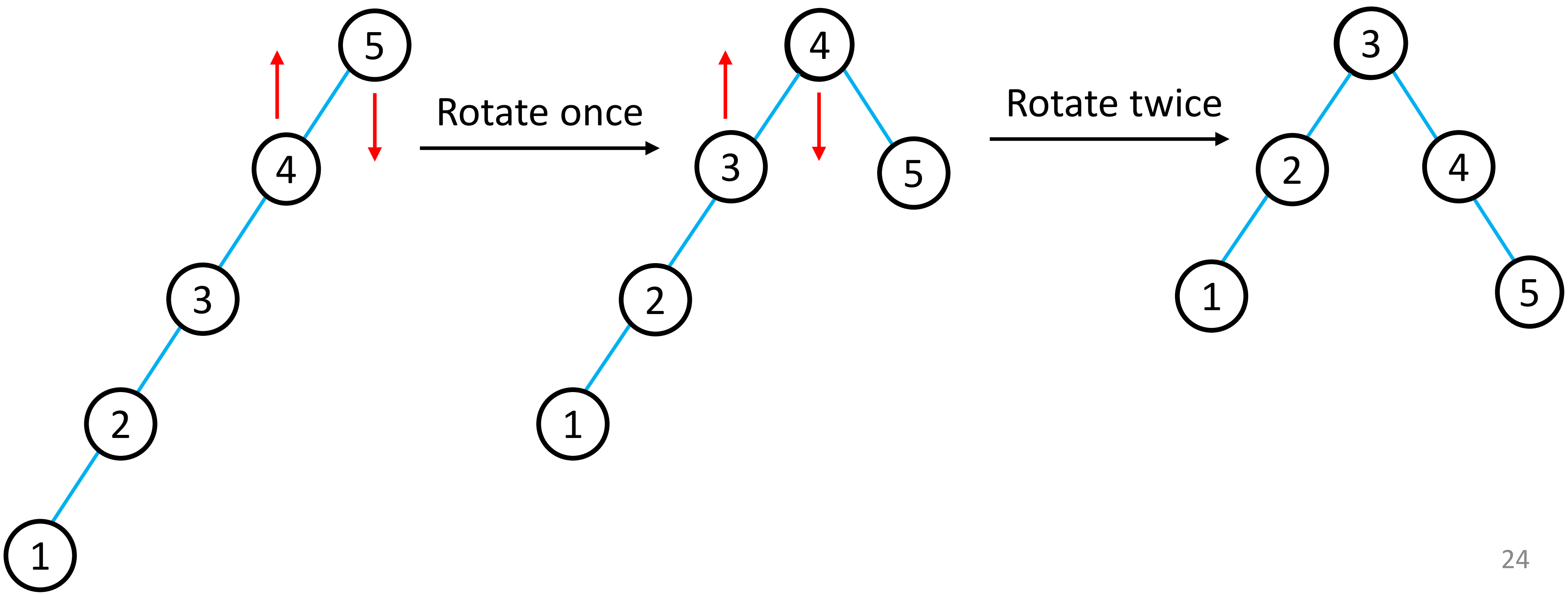
Deleting a Node

```
void DeleteItem(Tree &tree, int Data){
     if (tree == 0) return;
     else if (Data < tree->data)
          DeleteItem(tree->left, Data);
     else if (Data > tree->data)
          DeleteItem(tree->right, Data);
     else { // (Data == tree->data)
          if (tree->left == 0){
               Tree temp = tree;
               tree = tree->right;
               FreeNode(temp);
          }
          else if (tree->right == 0){
               Tree temp = tree;
               tree = tree->left;
               FreeNode(temp);
          }
          else{
               Tree pred = 0;
               FindPredecessor(tree, pred);
               tree->data = pred->data;
               DeleteItem(tree->left, tree->data);
          }
     }
}
```

```
void FindPredecessor(Tree tree, Tree
&predecessor){
     predecessor = tree->left;
     while (predecessor->right != 0)
          predecessor = predecessor->right;
}
```

Rotation

- Rotation is a fundamental technique performed on BSTs.
- A tree rotation moves one node up in the tree and one node down.

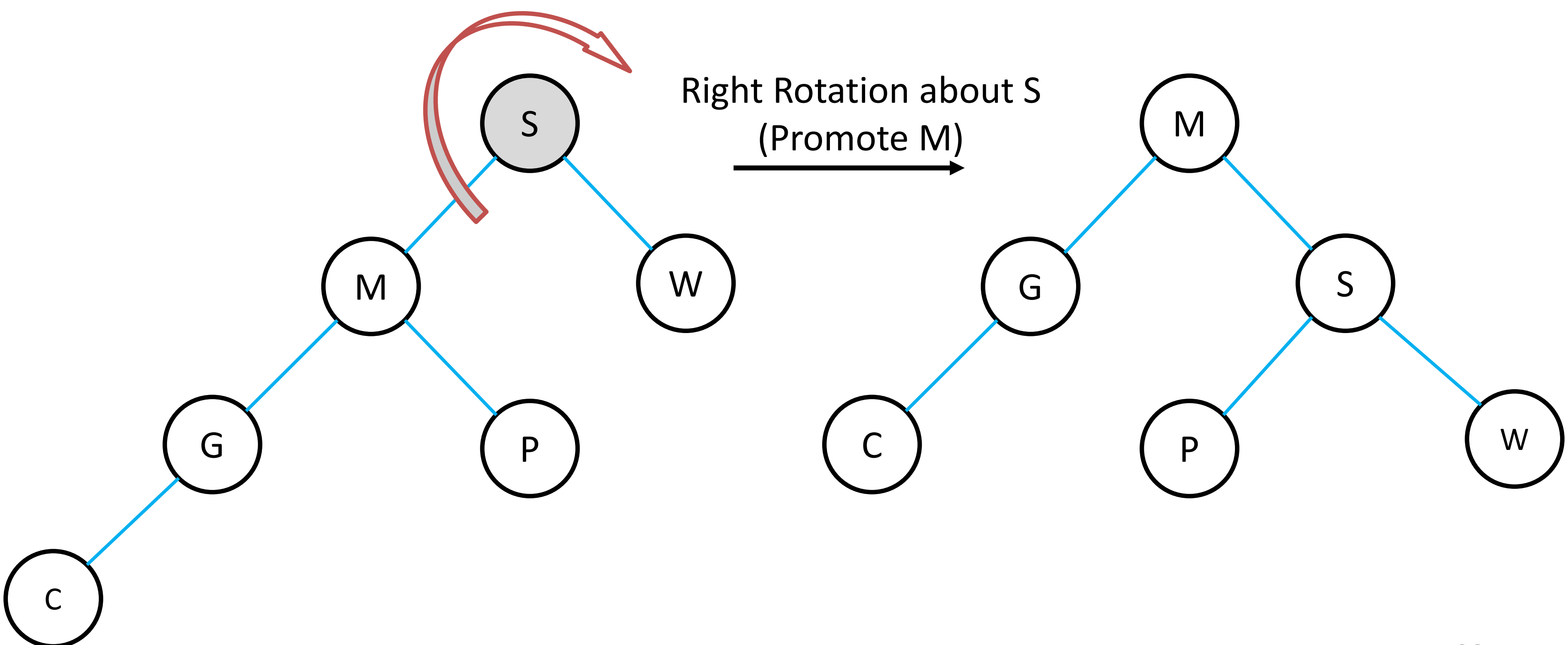


Rotation

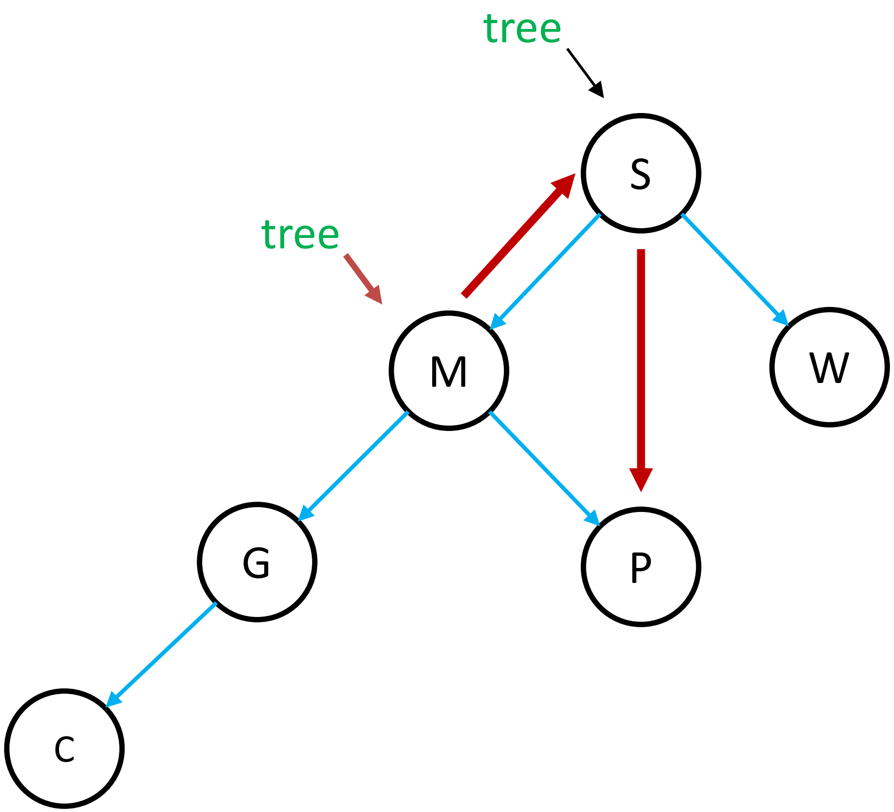
- It is used to change the shape of the tree.
  - In particular, to decrease its height by moving smaller subtrees down and larger subtrees up.
- This results in the improved performance of many tree operations.
- Two types of rotations:
  - Right rotation
  - Left rotation
- After the rotation, the sort order is preserved.
  - The resulting tree is STILL a BST.

Right Rotation

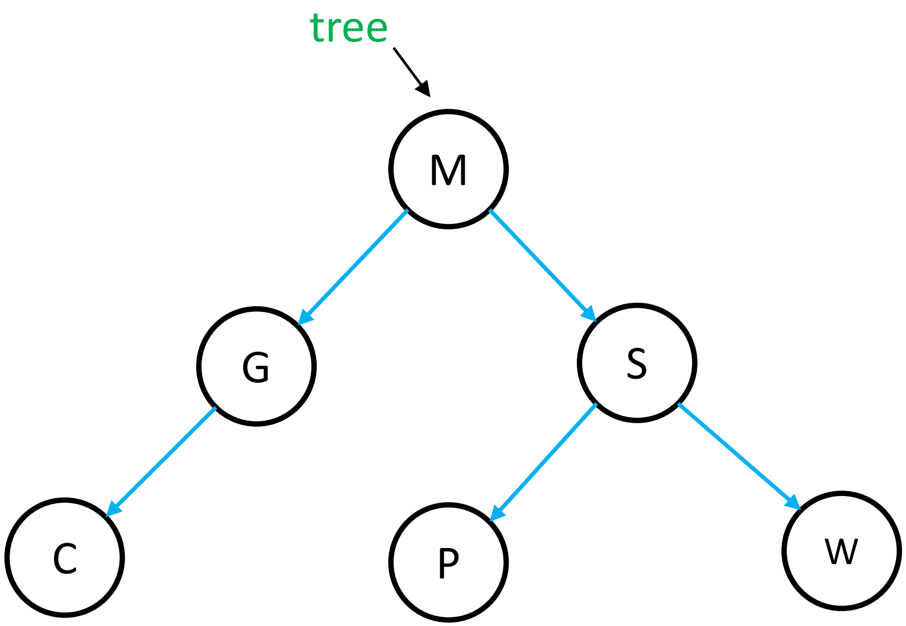
- Rotate right about a node
  - Move the left child up so that it will rotate into its parent's position
  - This is to promote the left child.



How to Implement Right Rotation?



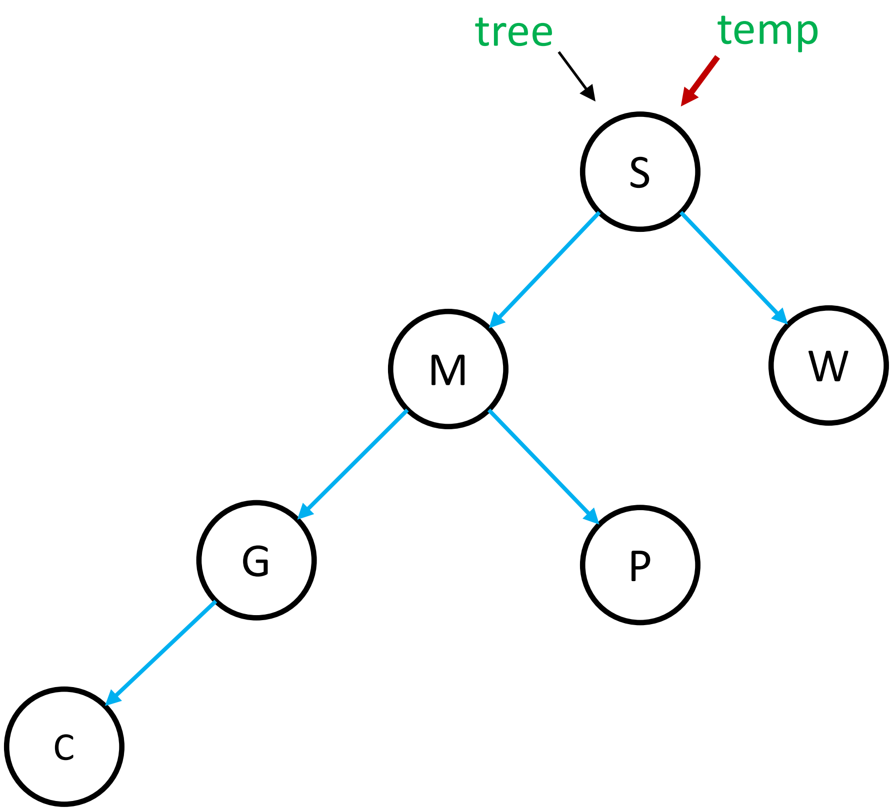
Before rotation



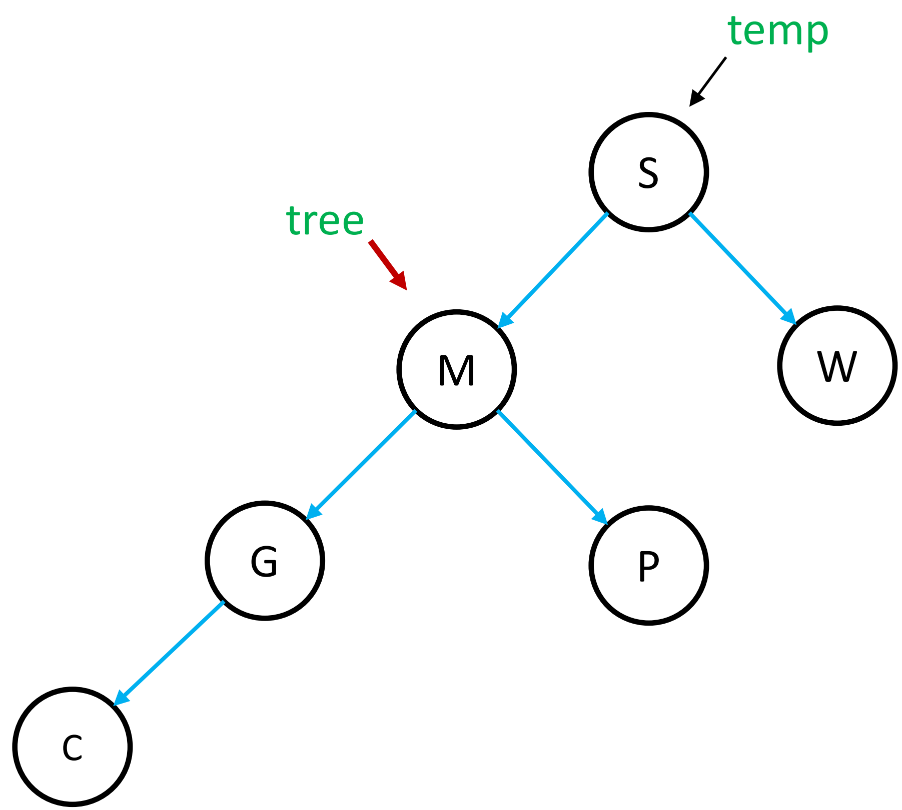
After rotation

Step by Step

`1.Tree temp = tree;`

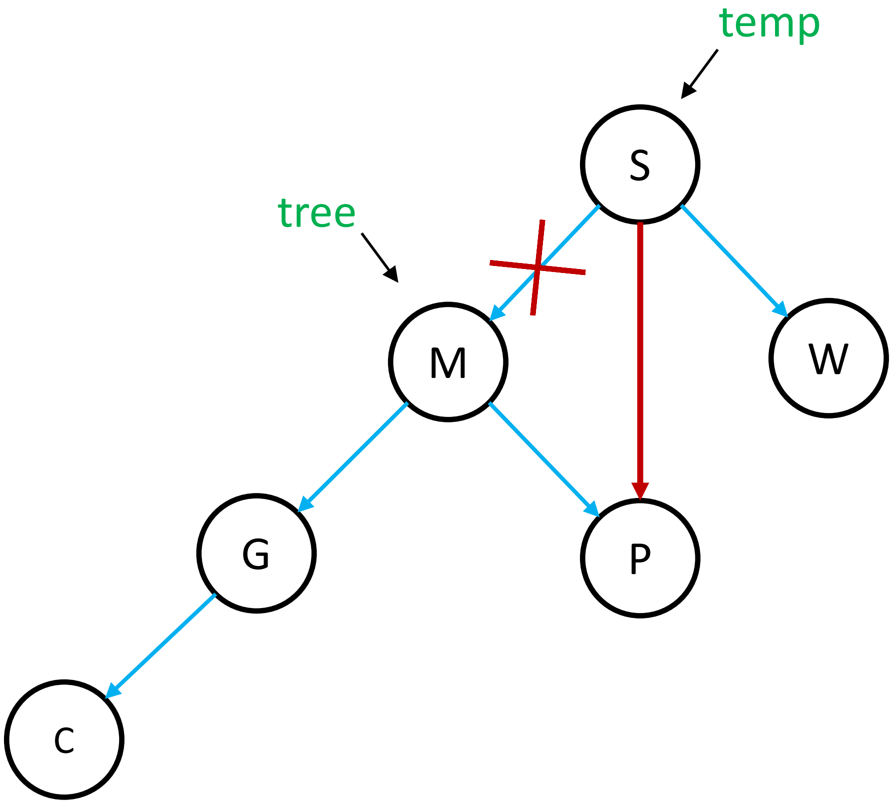


`2.tree = tree->left;`

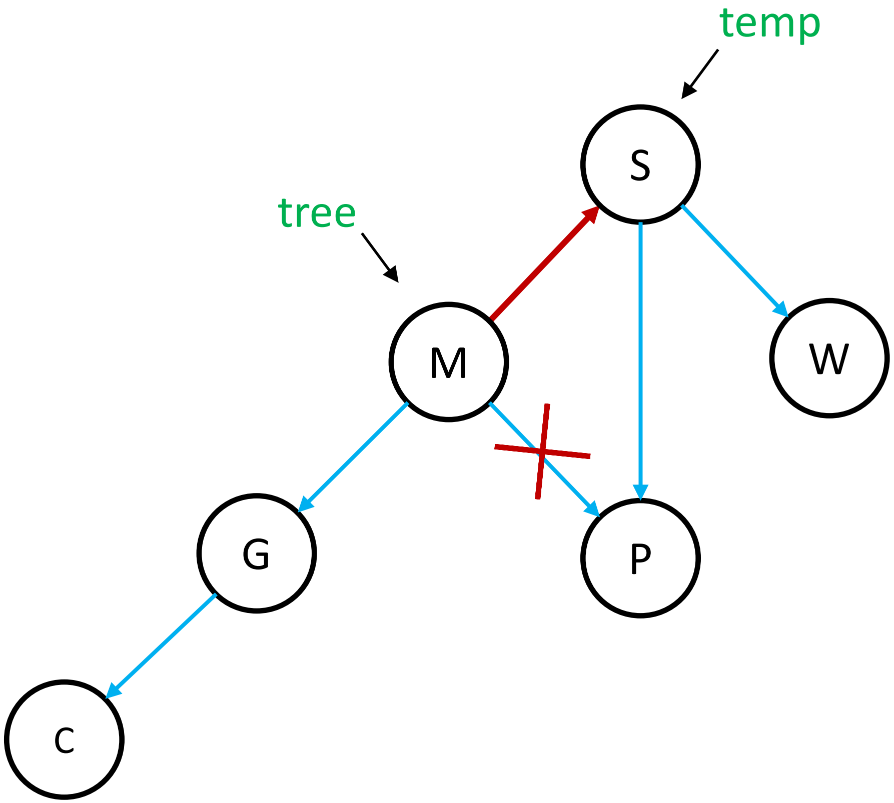


Step by Step

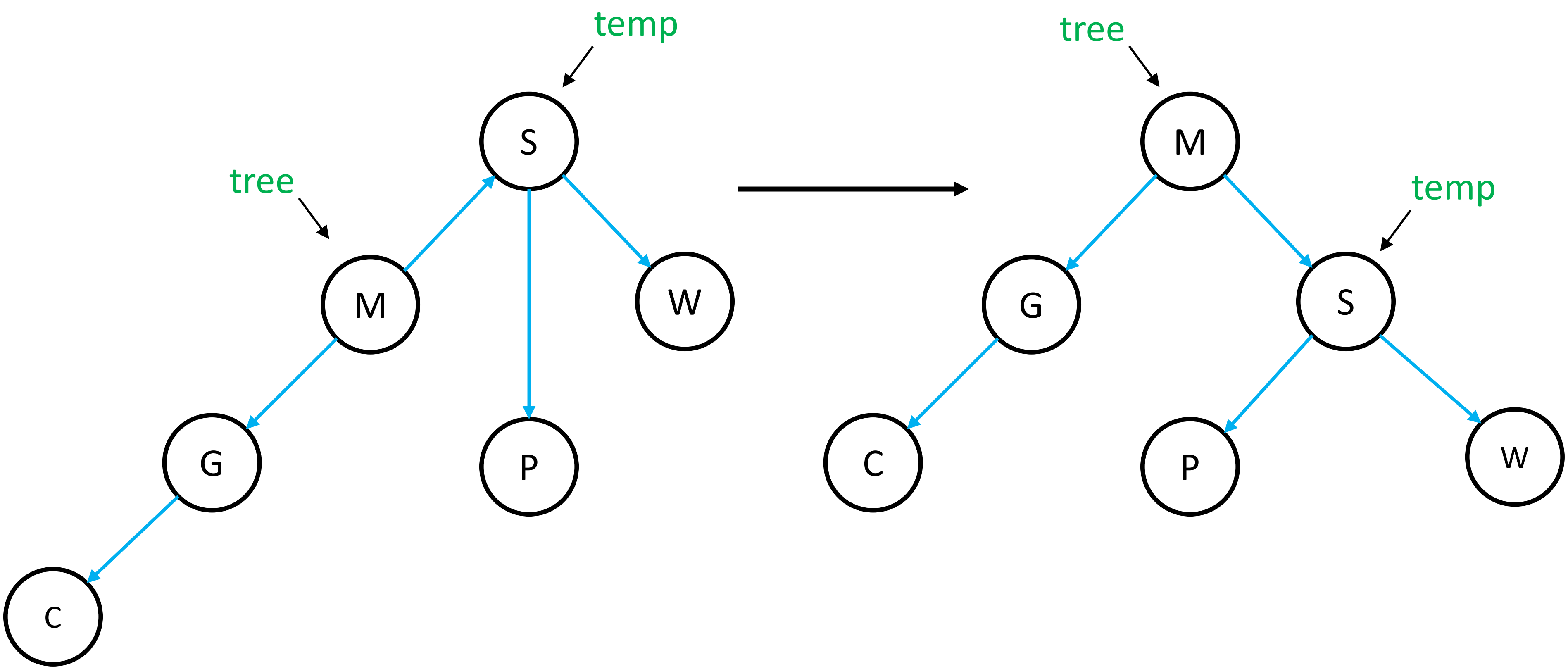
```
temp->left = tree->right;
```




```
tree->right = temp;
```

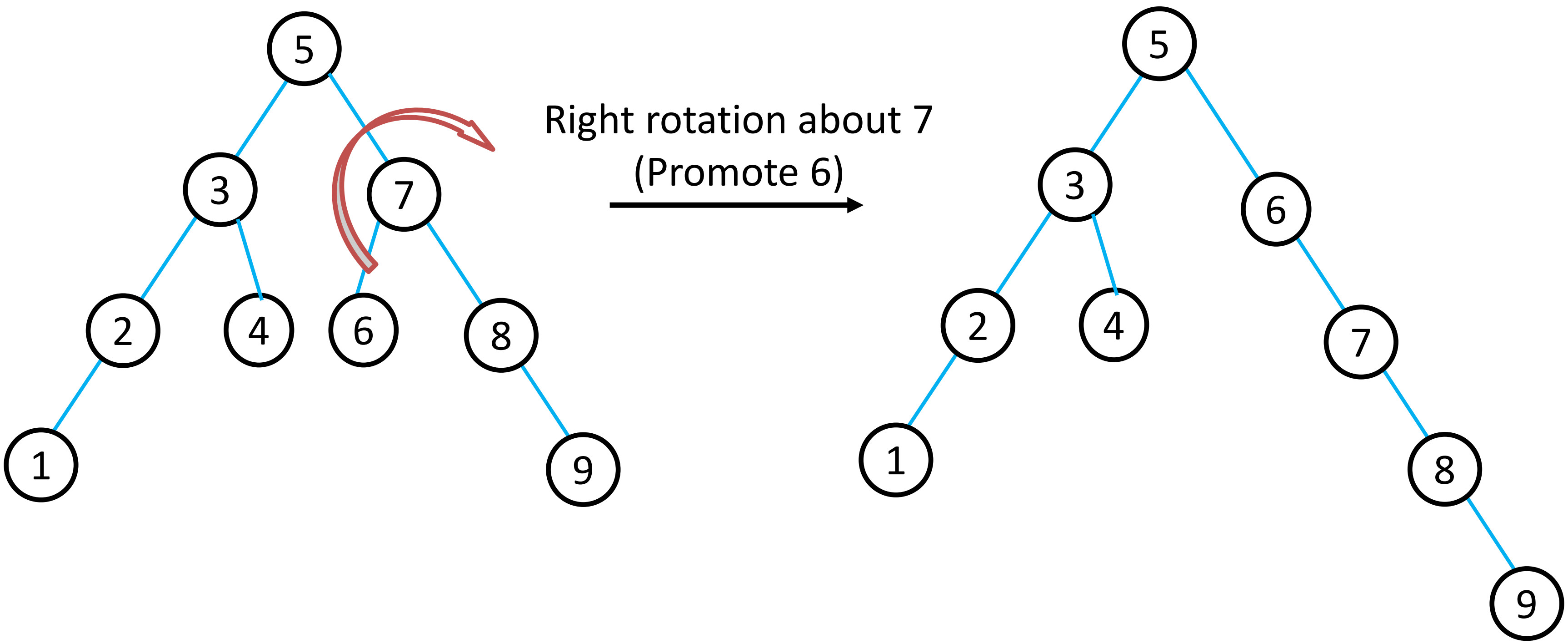


Adjusting the Diagram



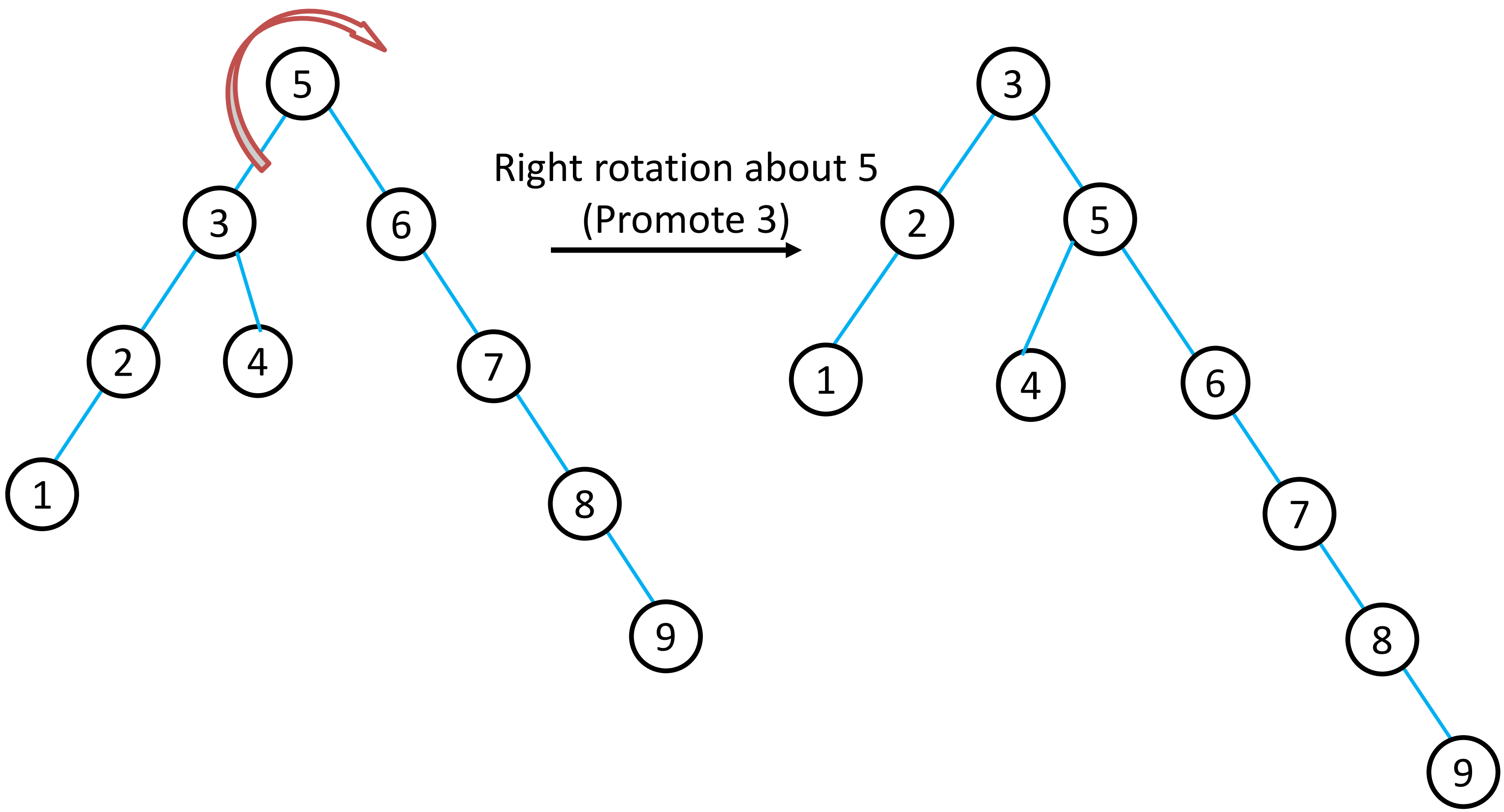
Right Rotation: Another Example

- Right rotate the following BST twice, first about 7 and then 5.



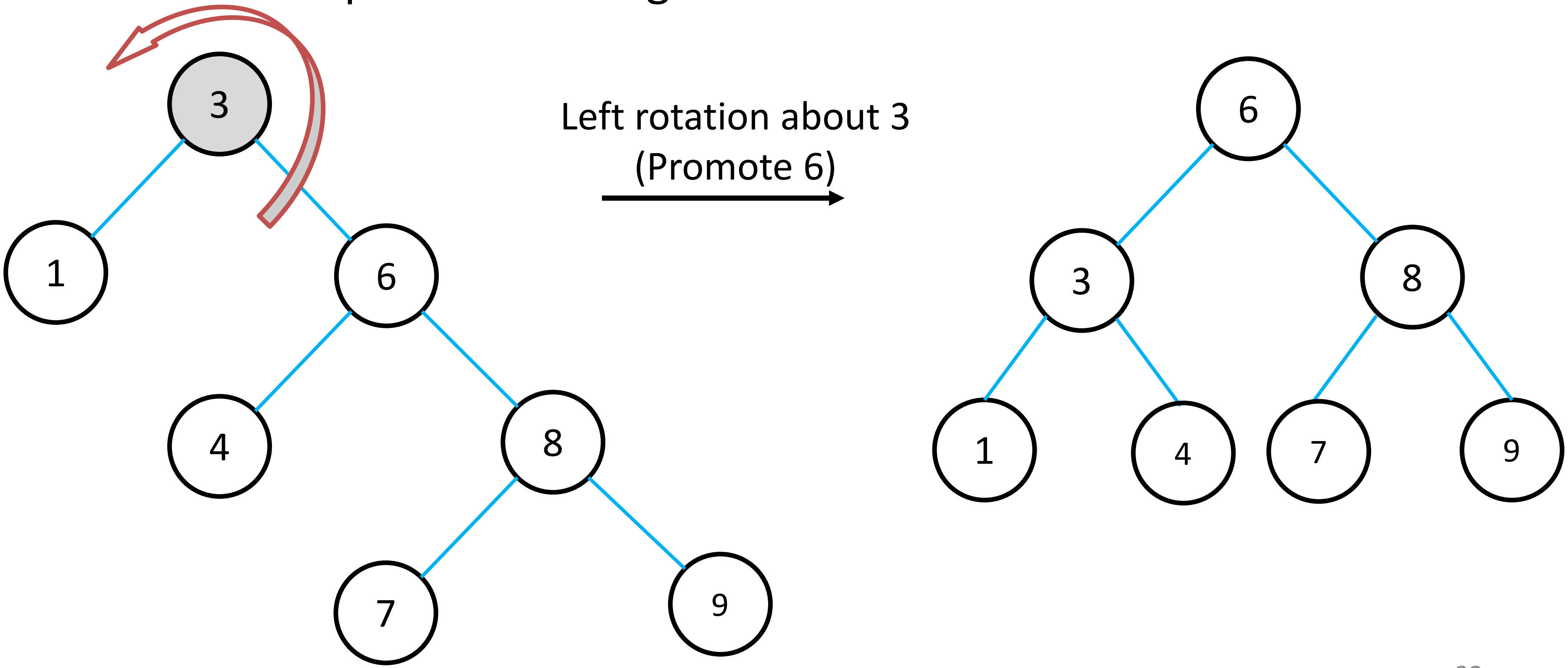
Right Rotation: Another Example

- Right rotate the following BST twice, first about 7 and then 5.

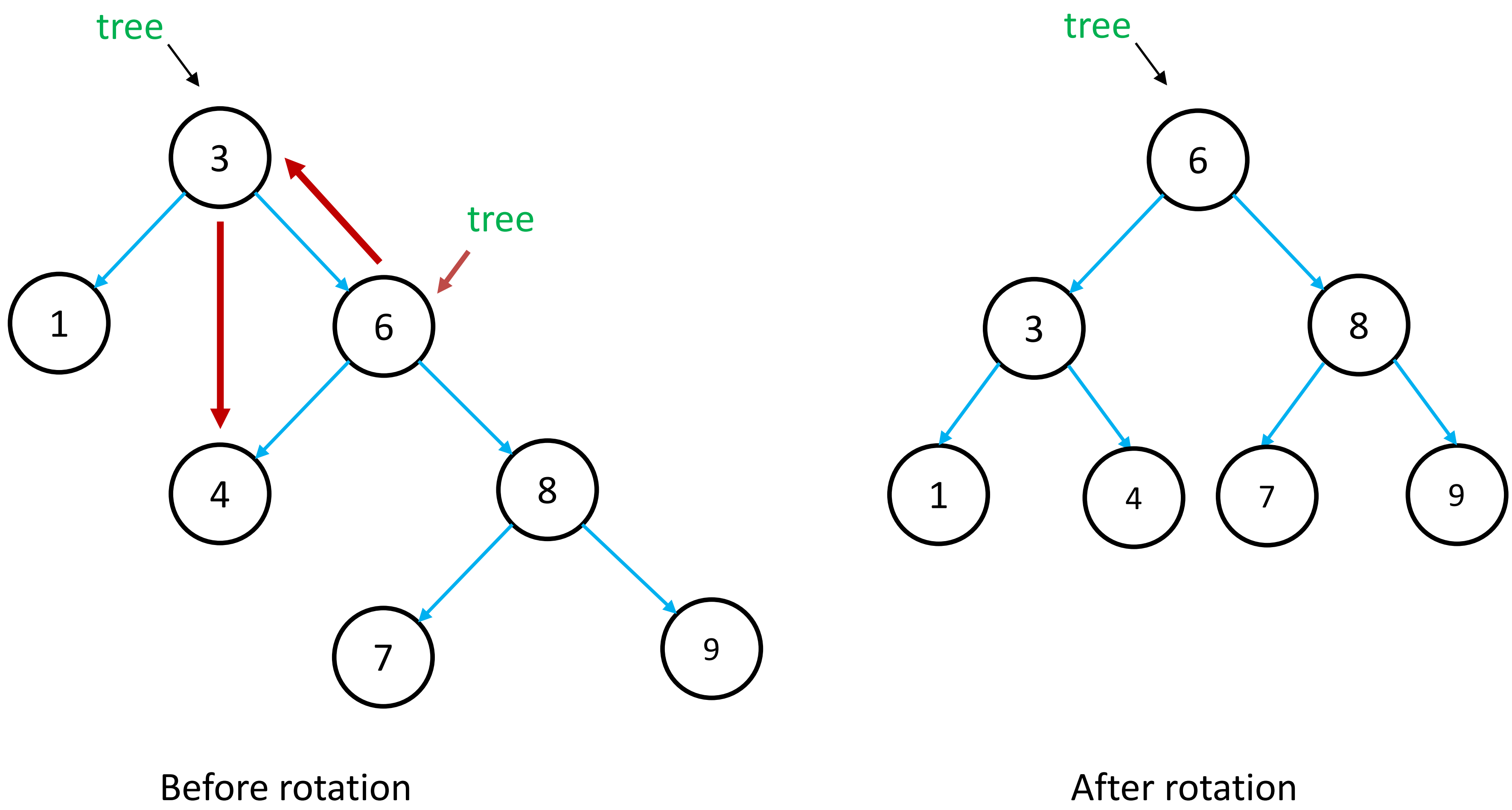


Left Rotation

- Rotate left around a node
  - Move the right child up so that it will rotate into its parent's position.
  - This is to promote the right child.

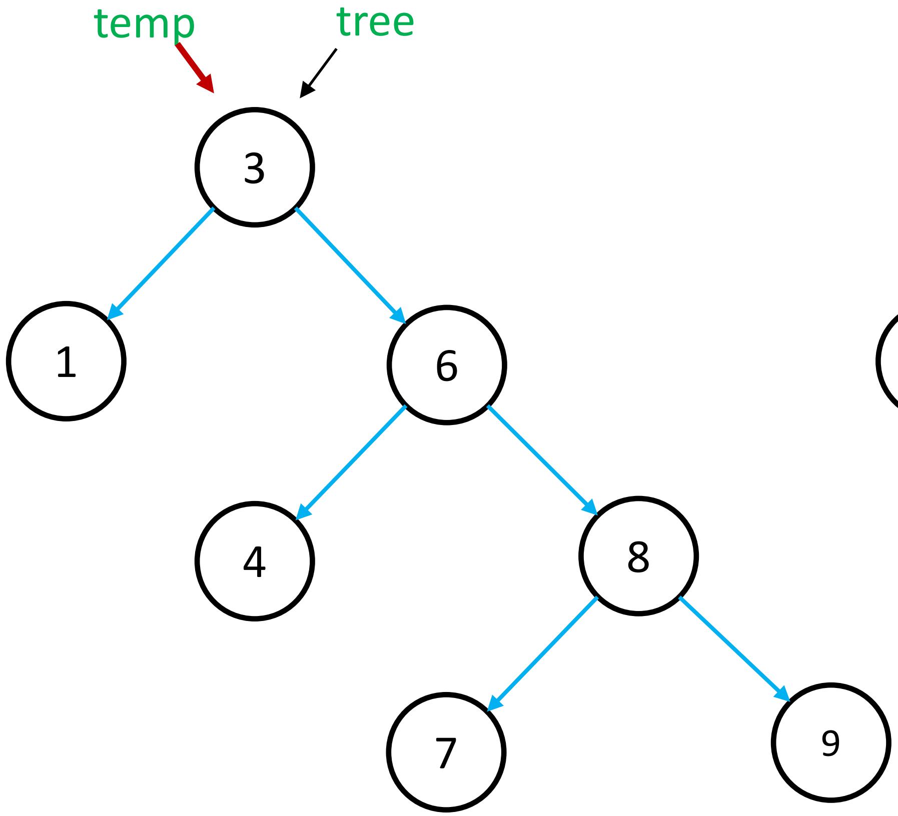


How to Implement Left Rotation?

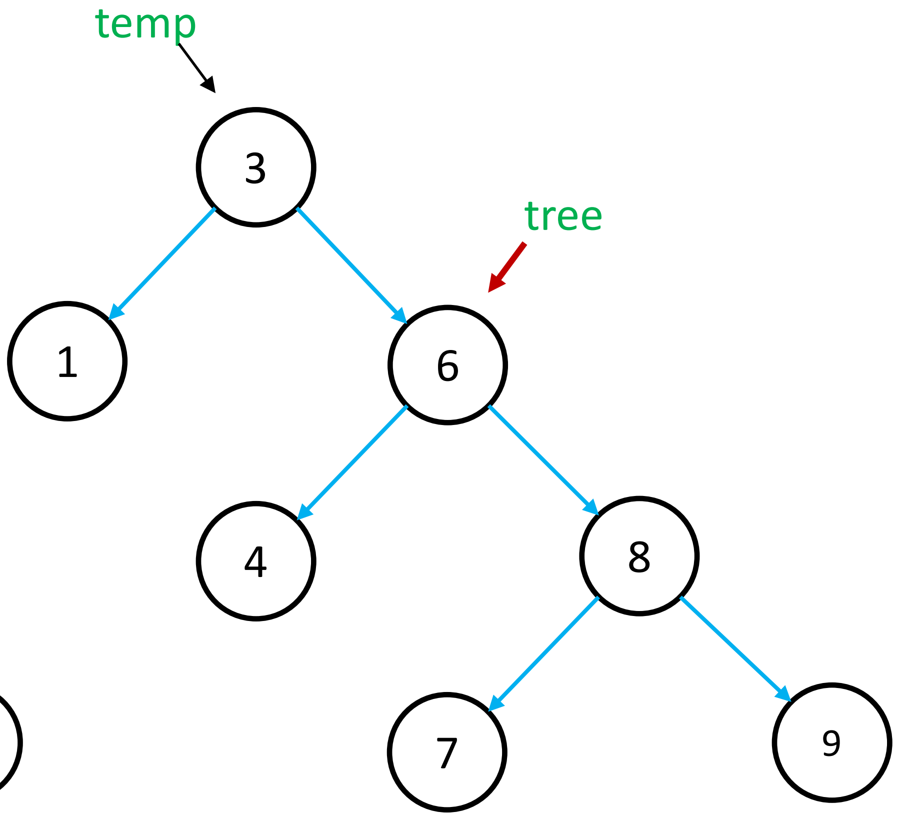


Step by Step

`1.Tree temp = tree;`

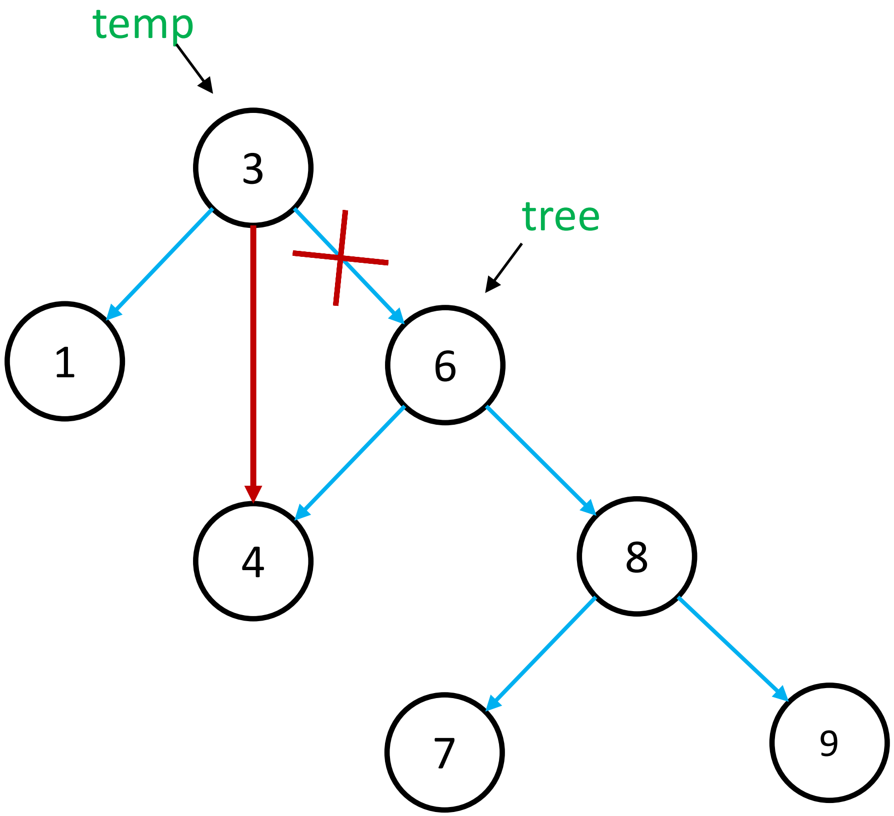


`2.tree = tree->right;`

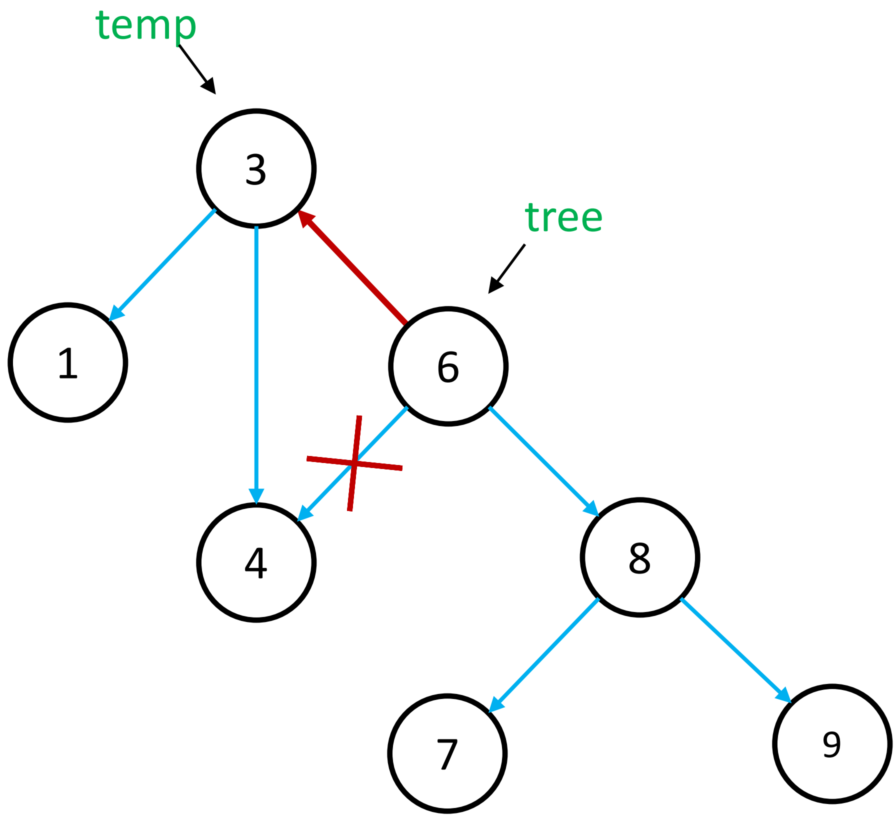


Step by Step

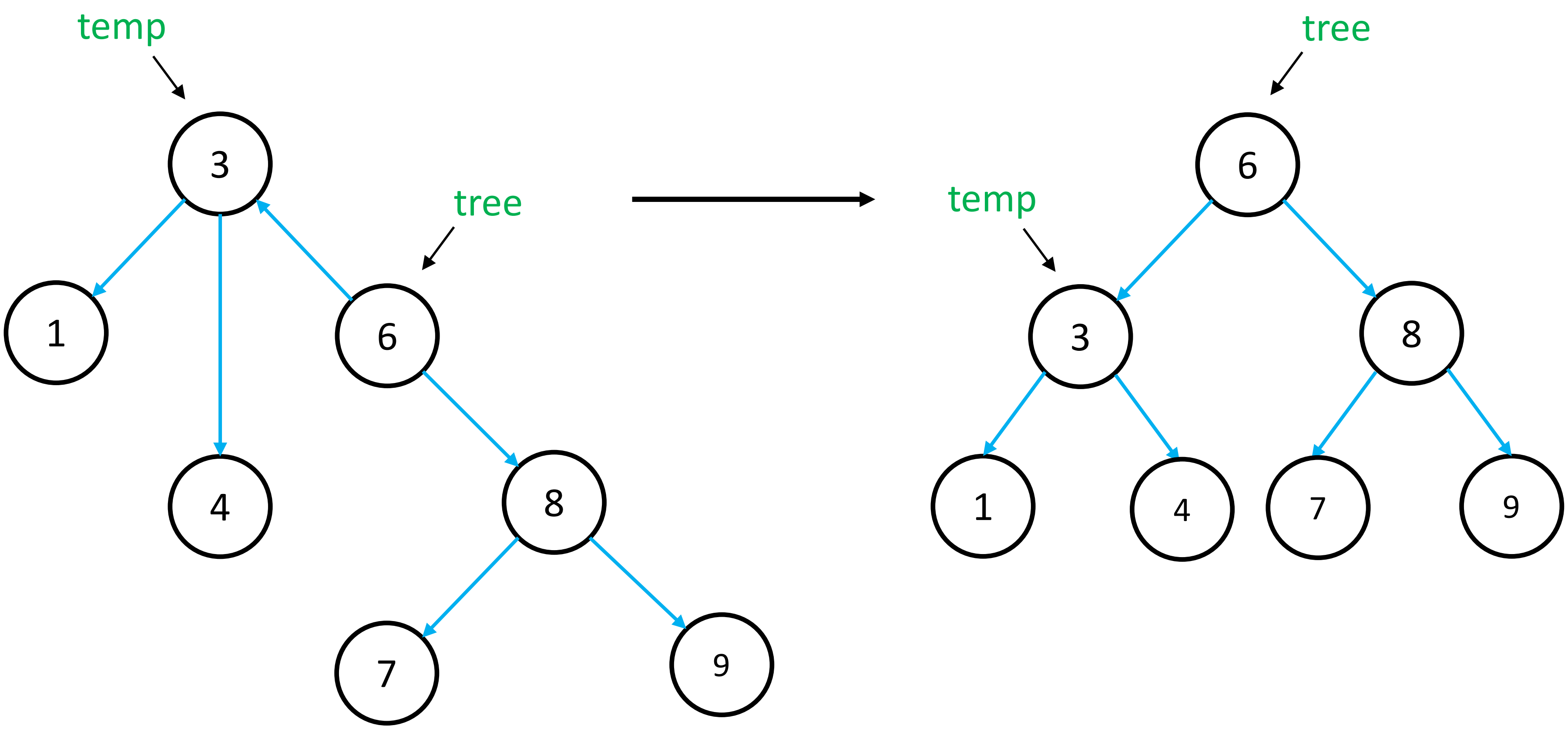
```
temp->right = tree->left;
```




```
tree->left = temp;
```



Adjusting the Diagram



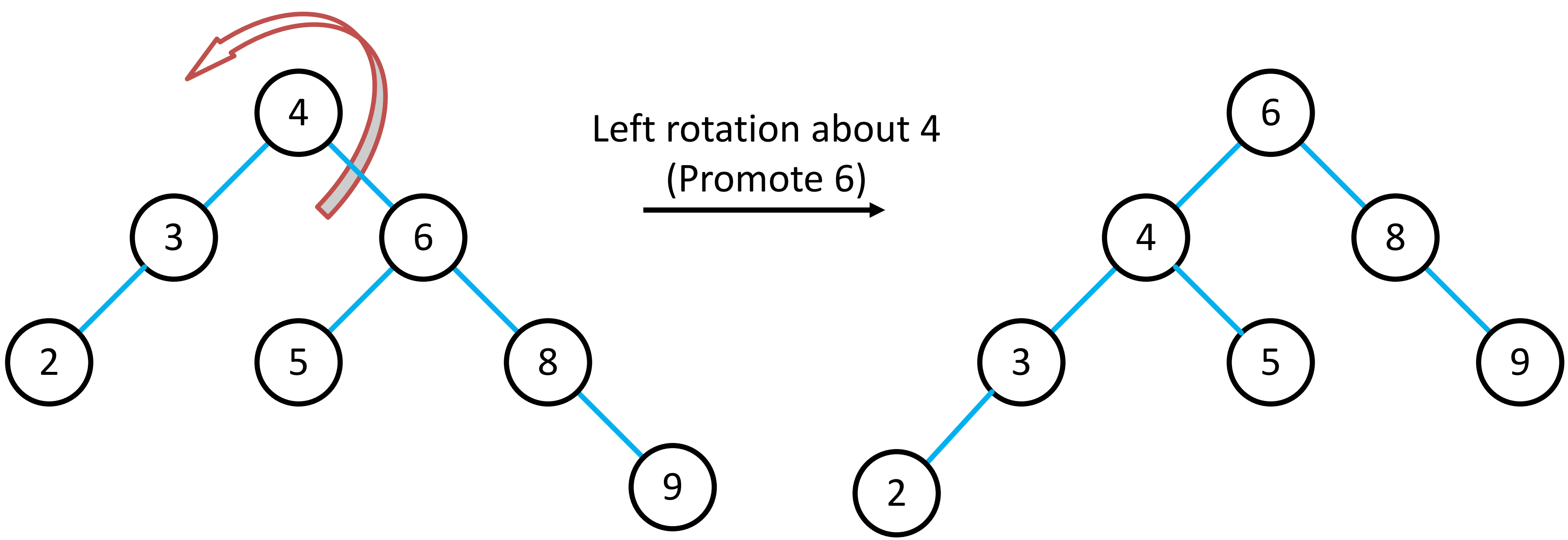
Right and Left Rotation

```
void RotateRight(Tree &tree){
    Tree temp = tree;
    tree = tree->left;
    temp->left = tree->right;
    tree->right = temp;
}
```

```
void RotateLeft(Tree &tree){
    Tree temp = tree;
    tree = tree->right;
    temp->right = tree->left;
    tree->left = temp;
}
```

Left Rotation: Another Example

- Left rotate the following BST about 4.



Summary

- Binary Search Tree Definition
- Binary Search Tree Operations
  - Finding an item
  - Insertion
  - Deletion
  - Rotation