Data Structure & Algorithms

Lec 12: Binary Search Trees
AVL Trees

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. AVL Tree Rotation
2. AVL Insertion
3. AVL Deletion

Learning Outcome

By the end of this class you should be able to:

1. Explain and implement the BST basic operations.
2. Identify if a given tree is balanced or not.
3. Recognize if a binary tree is an AVL tree or not.
4. Explain and implemen single tree location.

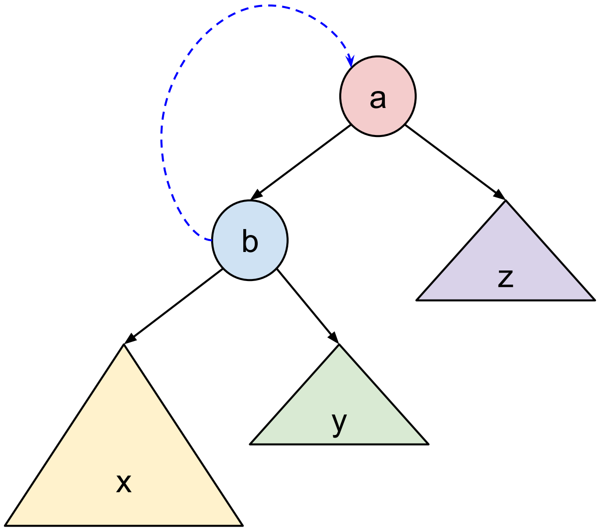
AVL Rotation

To solve an imbalance after inserting or deleting a node (if any), the AVL a perform one or more rotation operation. There are 4 types of rotation operations.

1. Left rotation (single)
2. Right rotation (single)
3. Left-Right rotation (double)
4. Right-Left rotation (double)

Direction of Rotation

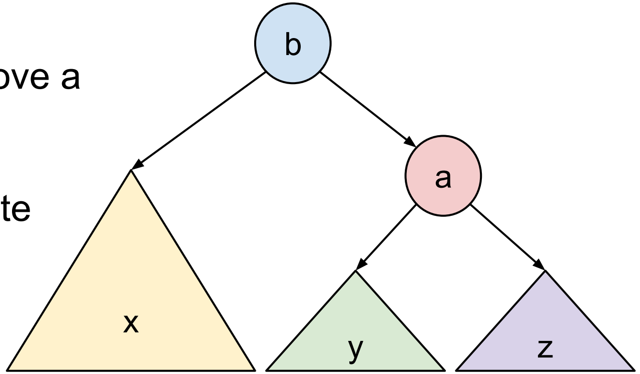
Is it left rotation or right rotation?



b and a will switch places, because the tree is imbalanced at a.

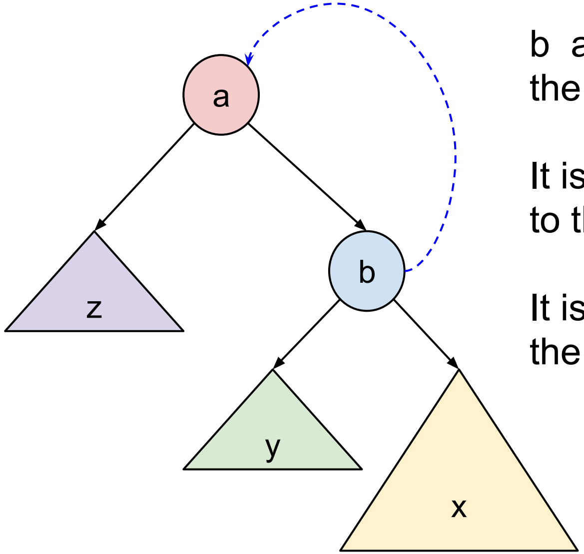
It is right rotation because we move a to the right (right subtree)

It is left rotation because we rotate the left subtree of a



Direction of Rotation

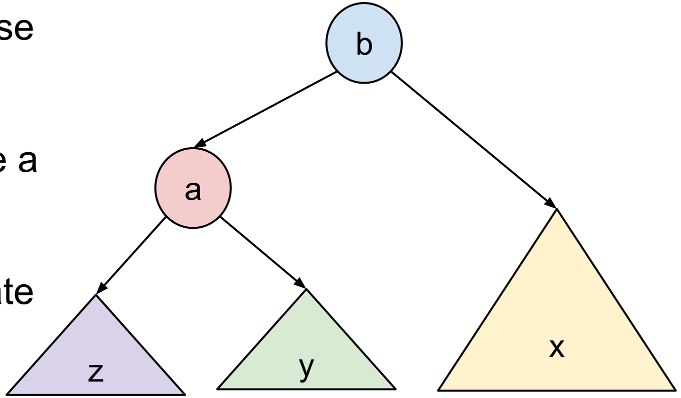
Is it left rotation or right rotation?



b and a will switch places, because the tree is imbalanced at a.

It is left rotation because we move a to the left (left subtree)

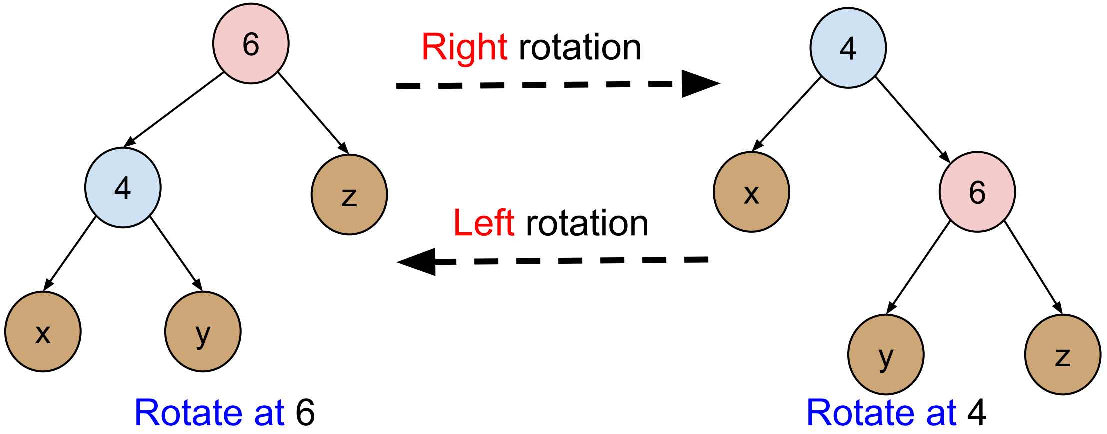
It is right rotation because we rotate the right subtree of a



Direction of Rotation

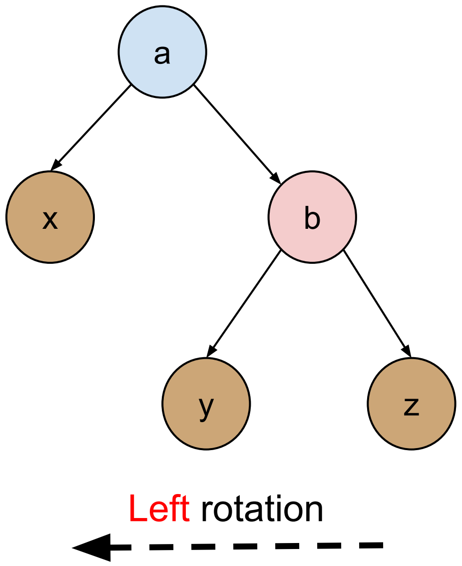
The right and left rotation are symmetric operations.

We will use the directional movement of the rotating node to describe the rotation direction. If the node is moving to the right subtree, then it is a right rotation if the node is moving to the left subtree then it is a left rotation



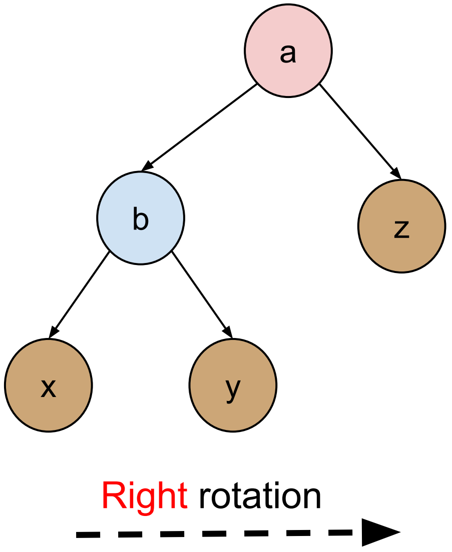
Left Rotation

1. b becomes the parent of a.
2. a takes the ownership of b's left subtree (y)
3. b's right subtree remain the right subtree of b
4. a's left subtree remain the left subtree of a
5. a becomes the left child of b



Right Rotation

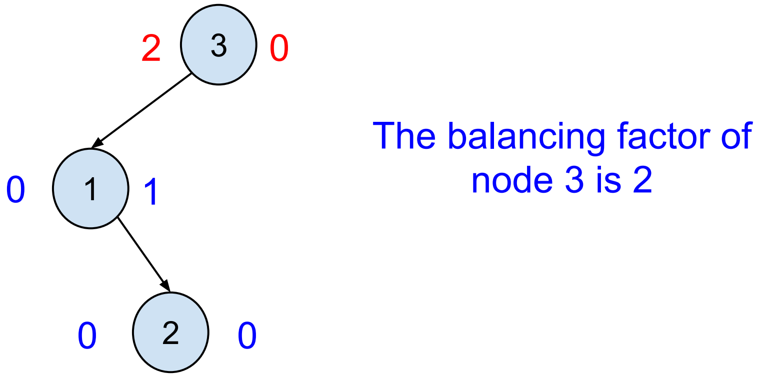
1. b becomes the parent of a.
2. a takes the ownership of b's right subtree (y)
3. b's left subtree remain the left subtree of b
4. a's right subtree remain the right subtree of a
5. a becomes the right child of b



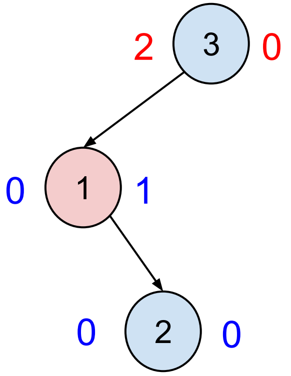
Left-Right (LR) Rotation

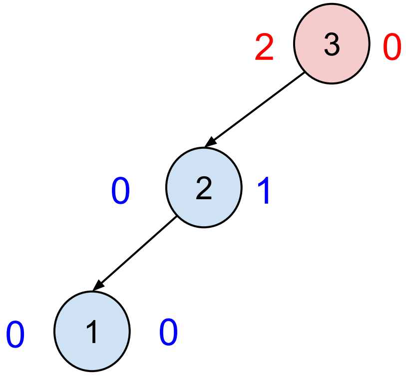
Sometimes a single left rotation is not sufficient to balance an unbalanced tree. In this case we will need to perform a double rotation. A single left rotation and then a right left rotation.

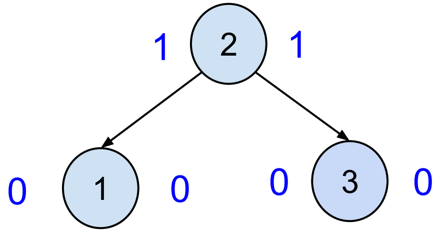
Example insert the values {3, 1, 2} in an empty tree.



left-Right (LR) Rotation





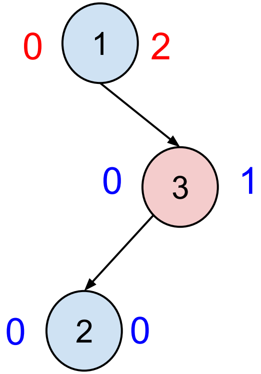


Left rotation on node 1

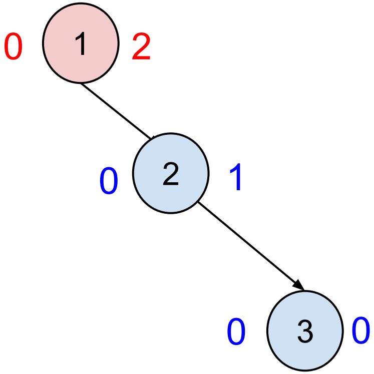
Right rotation on node 3

Right-Left (RL) Rotation

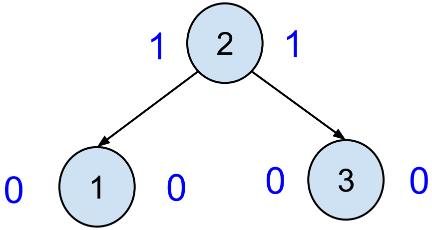
Example insert the values {1, 3, 2} in an empty tree.



Right rotation on node 3

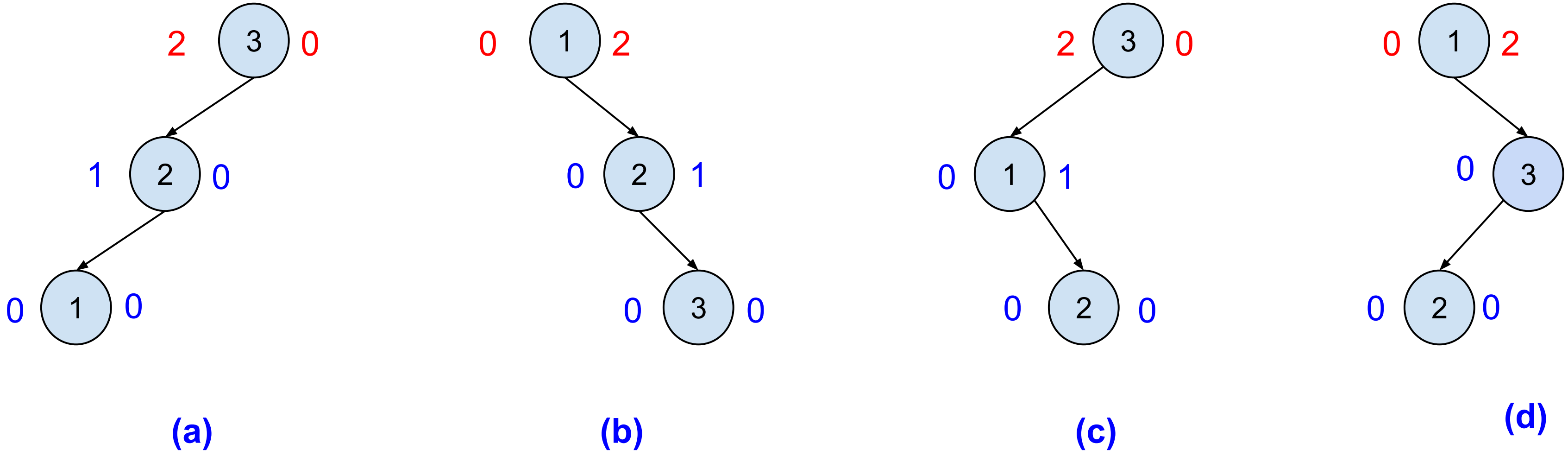


Left rotation on node 2



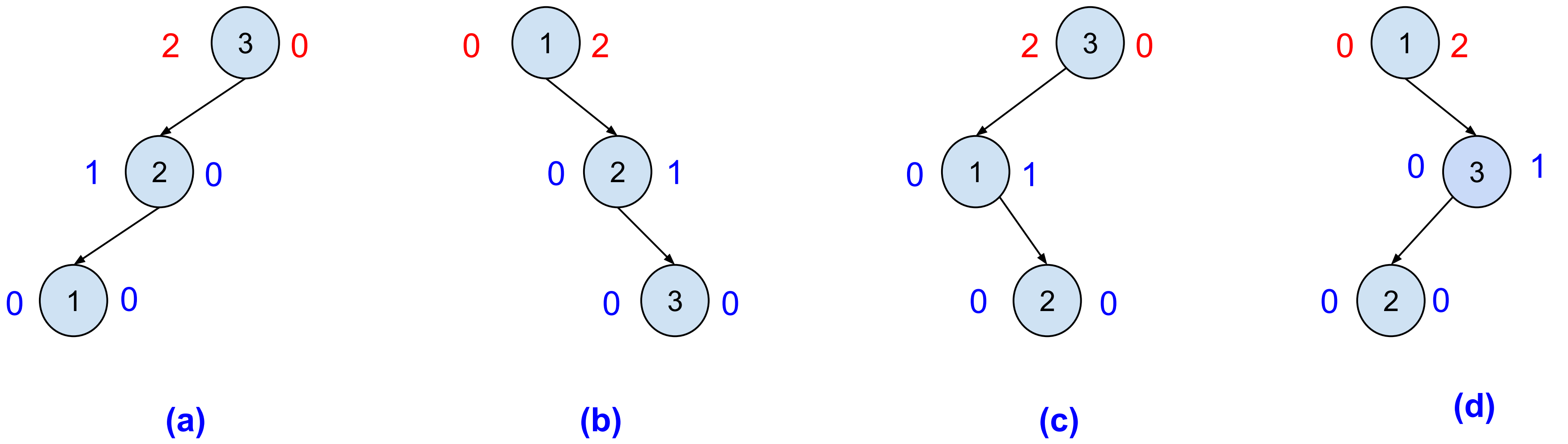
Selecting the Rotation Operation

Given unbalanced AVL tree how we determine which type of rotation we need



Selecting the Rotation Operation

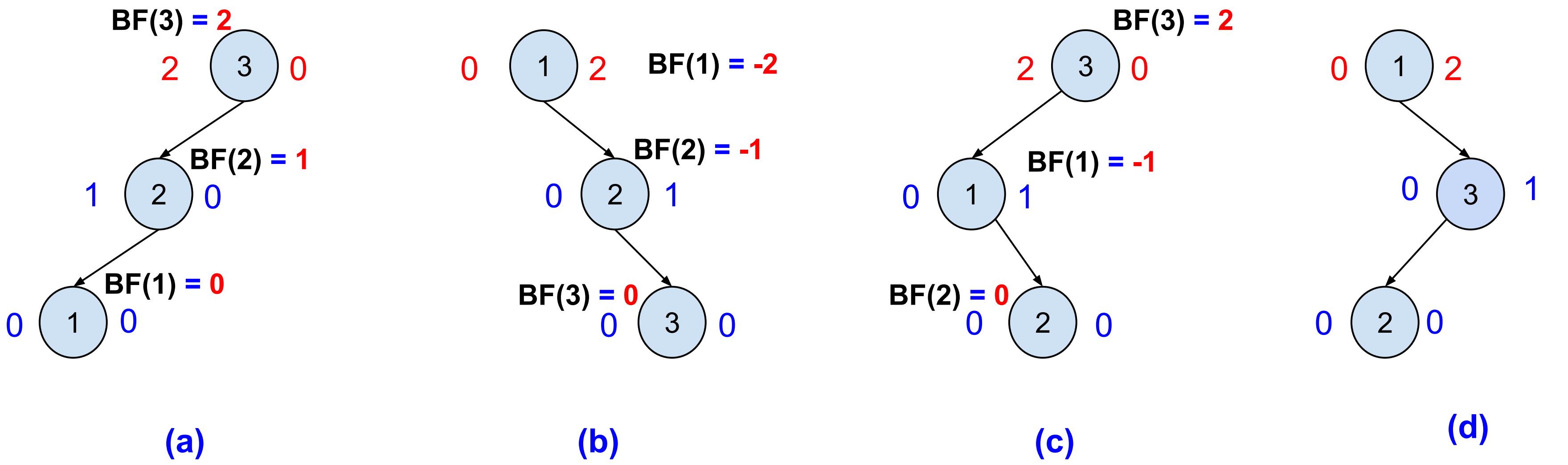
Given unbalanced AVL tree how we determine which type of rotation we need



BalanceFactor(T) = height(T.left) - height(T.right)

Selecting the Rotation Operation

Given unbalanced AVL tree how we determine which type of rotation we need



BalanceFactor(T) = height(T.left) - height(T.right)

Selecting the Rotation Operation

If the tree is left heavy and its subtree is left heavy →
    right rotation

If the tree is left heavy and its subtree is right heavy →
    left-right rotation

If the tree is right heavy and its subtree is right heavy →
    left rotation

If the tree is right heavy and its subtree is left heavy →
    right-left rotation

AVL Tree Insertion and Balancing

After inserting a node in balanced AVL tree we need at most two rotations to rebalance the tree.

Inserting in an an AVL tree take O(log n) and any rotation operation is O(1) fixed number of pointers modifications.

The time required to insert an element in an AVL tree is O (log n).

After every insertion operation, we need to check if the tree is balanced or not and it an imbalance occurred we need to apply the appropriate rotations to fix it.

Check AVL Tree Balance

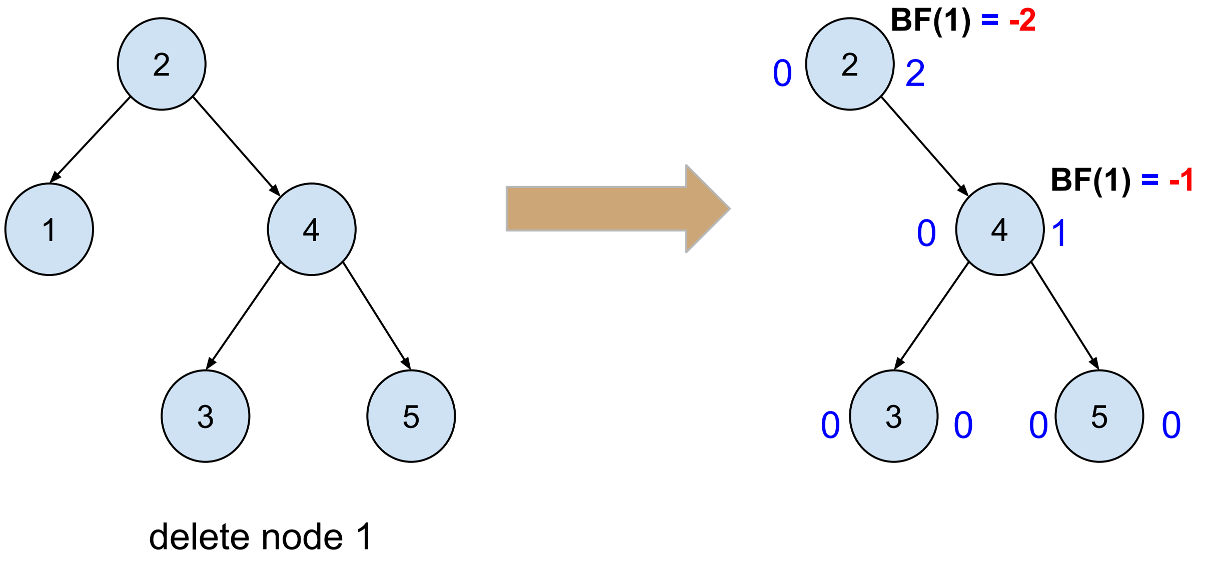
```
algorithm InsertNode(current, value)
Pre: current is the node to start from
Post: value has been placed in the correct location and the tree is a valid AVL tree
if value < current.Value
    if current.Left = null
        current.Left ← node(value)
    else
        InsertNode(current.Left, value)
    end if
else
    if current.Right = null
        current.Right ← node(value)
    else
        InsertNode(current.Right, value)
    end if
end if
CheckBalance(current)
end InsertNode
```

Check AVL Tree Balance

```
algorithm CheckBalance(current)
   Pre: current is the node to start from balancing
   Post: current height has been updated and tree is rebalanced if needed
   if current.Left = null and current.Right = null
      current.Height = 0;
   else
      current.Height = Max(Height(current.Left),Height(current.Right)) + 1
   end if
   if Height(current.Left) - Height(current.Right) > 1
      if Height(current.Left.Left) - Height(current.Left.Right) > 0
         RightRotation(current)
      else
      LeftAndRightRotation(current)
   end if
   else if Height(current.Left) - Height(current.Right) < −1
         if Height(current.Right.Left) - Height(current.Right.Right) < 0
            LeftRotation(current)
         else
         RightAndLeftRotation(current)
         end if
   end if
```

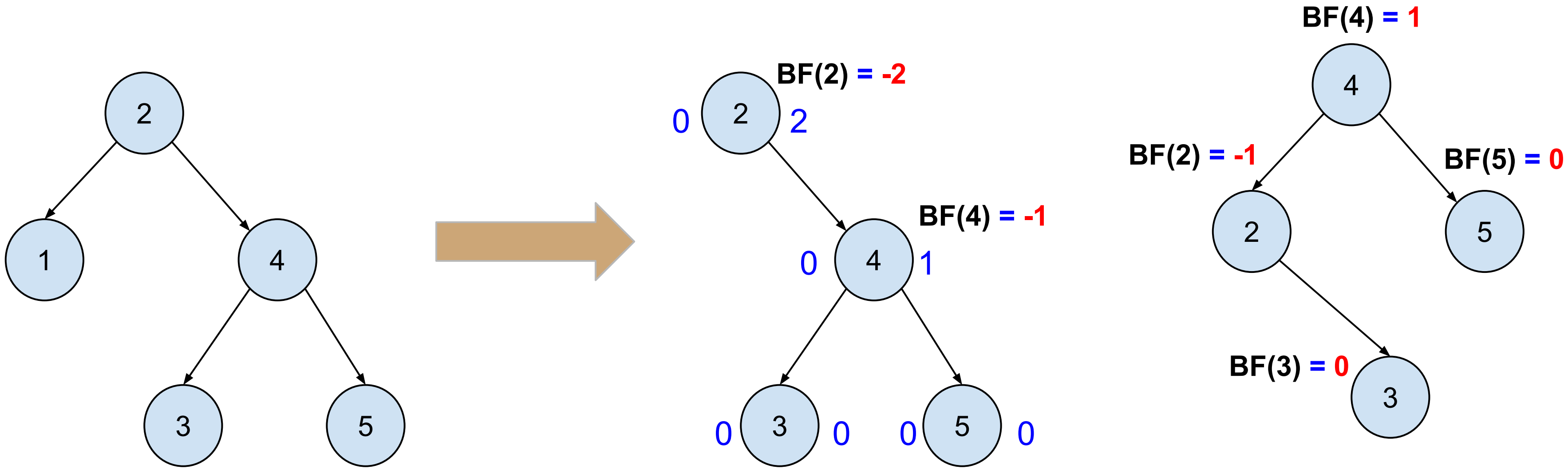
Deleting a node in AVL Tree

Imbalance might occur after deleting a node from an AVL tree



Deleting a node in AVL Tree

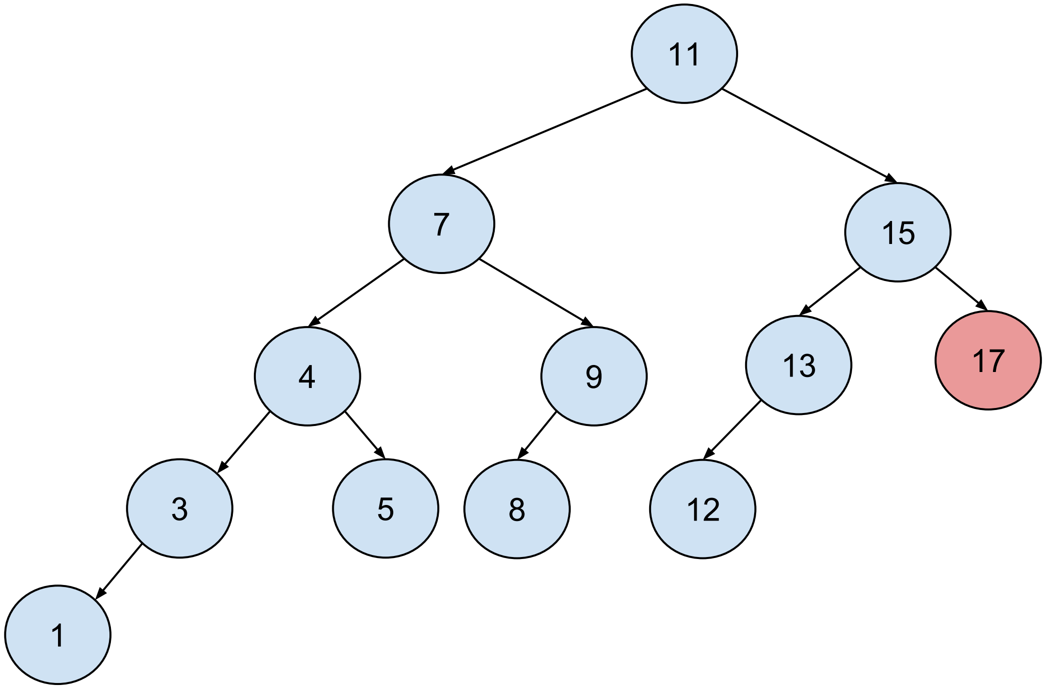
Imbalance might occur after deleting a node from an AVL tree



delete node 1

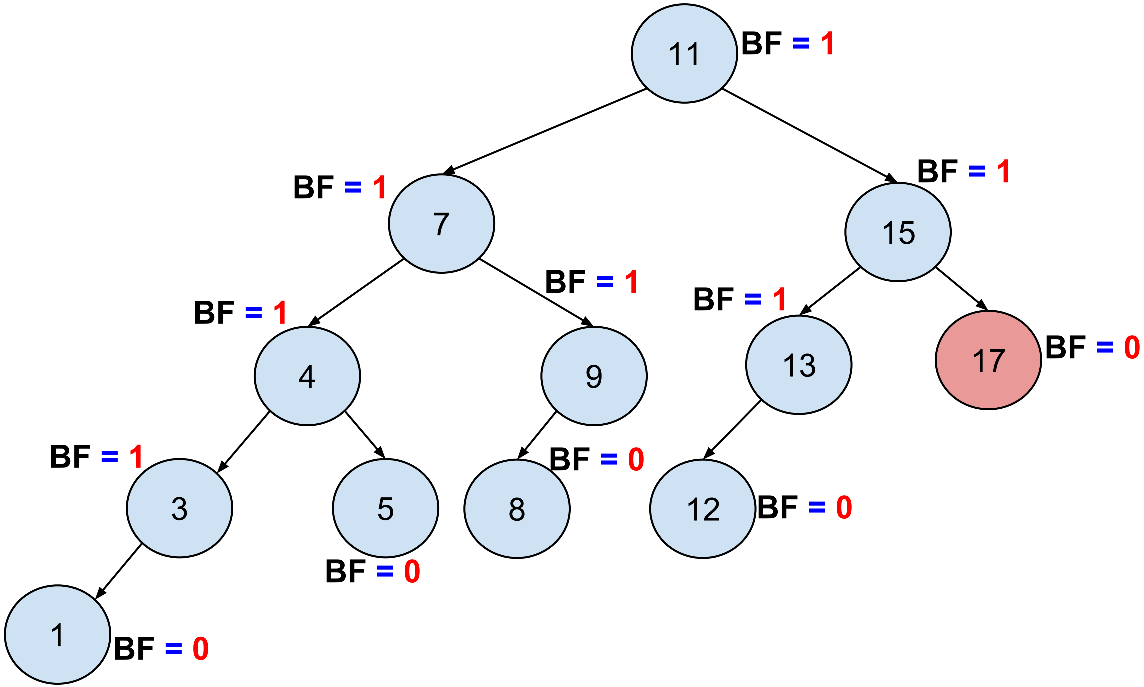
Left rotation

Deleting a node in AVL Tree



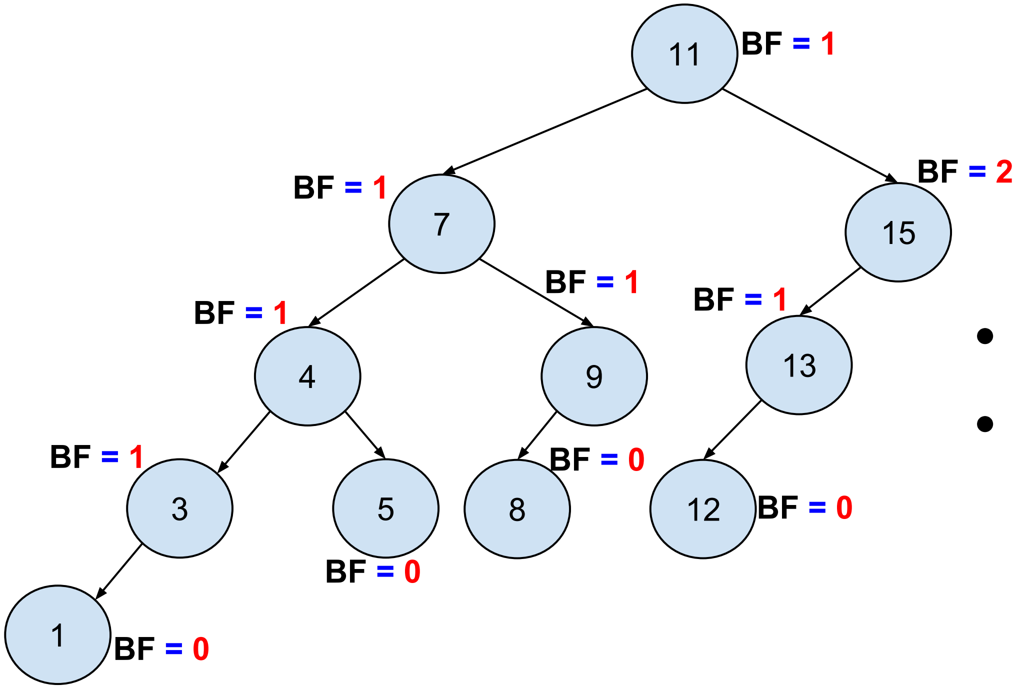
delete node 17

Deleting a node in AVL Tree



delete node 17

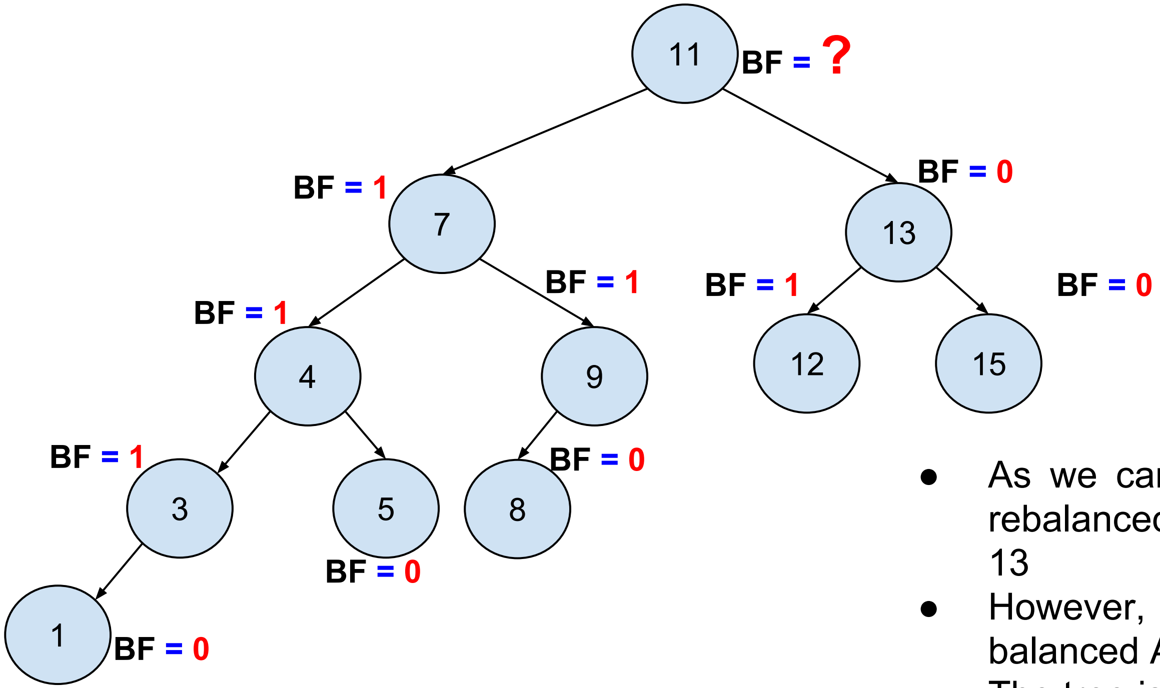
Deleting a node in AVL Tree



After deleting node 17

- The subtree is imbalanced at node 15
- The tree is left heavy and its subtree is left heavy so we need one right rotation on node 15 to rebalance the subtree 15

Deleting a node in AVL Tree

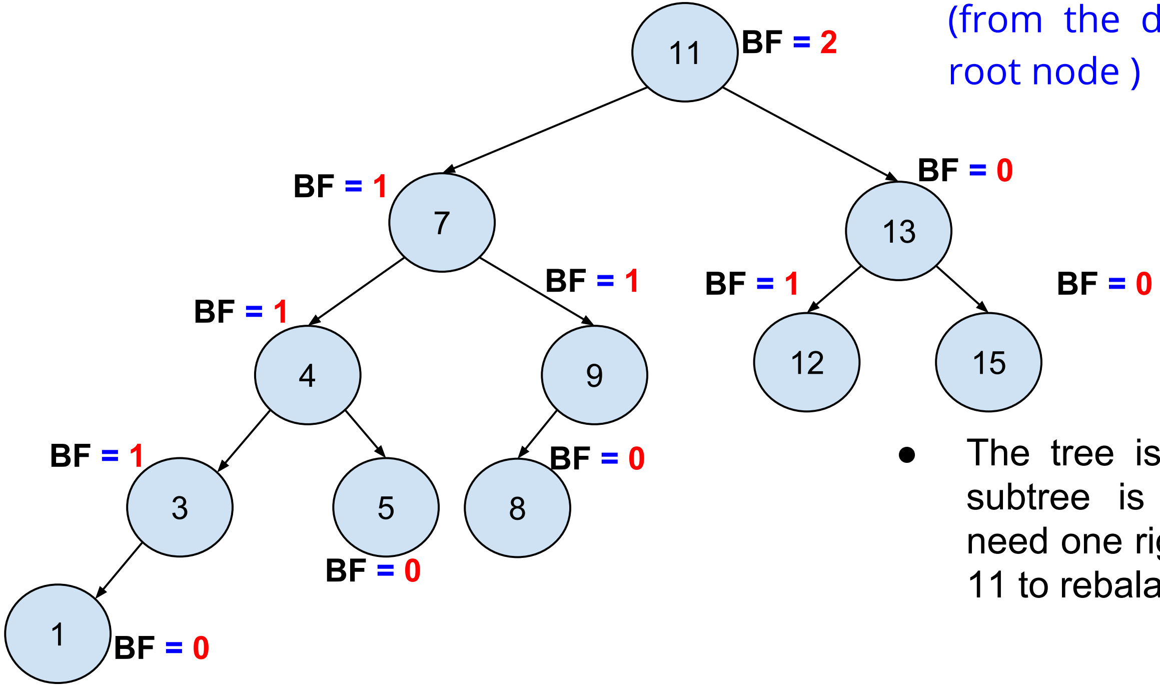


After right rotate on 15

- As we can see the subtree is rebalanced and its new root is 13
- However, the result is not a balanced AVL tree.
- The tree is imbalanced at node 11

Deleting a node in AVL Tree

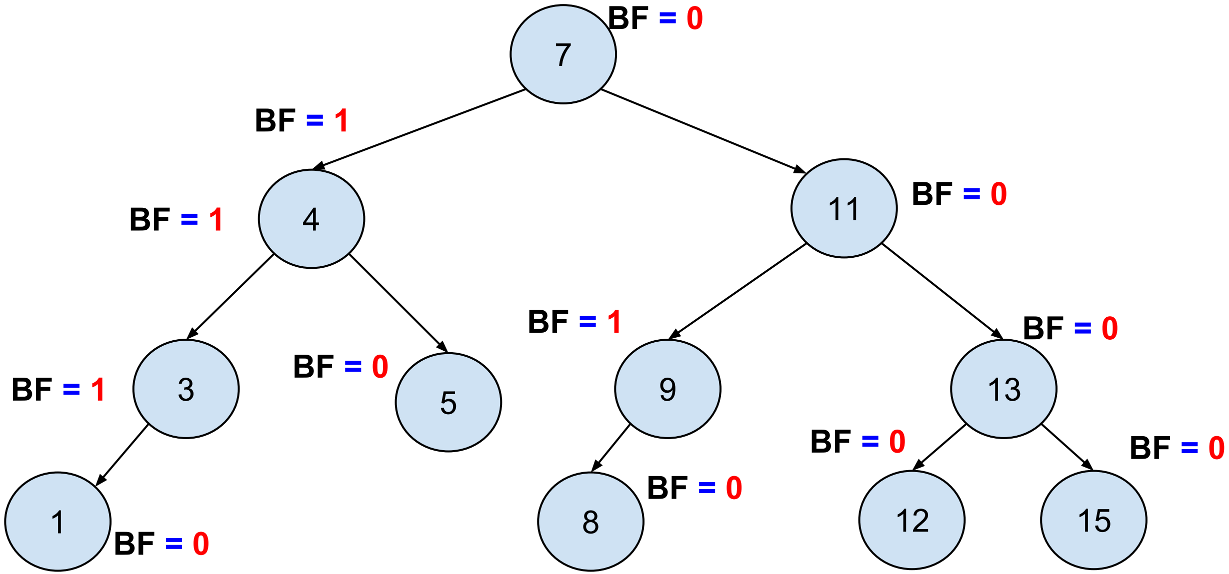
Imbalance propagate upward (from the delete position to the root node)



After deleting node 17

- The tree is left heavy and its subtree is left heavy so we need one right rotation on node 11 to rebalance the tree

Deleting a node in AVL Tree



Deleting a node in AVL Tree

Deleting a node in an AVL tree is more complex than inserting a node.

The imbalance may propagate upward so that many rotations may be needed.

The maximum number of rotation operations when deleting a node is O(log n)

We need to store the path from the root of the tree to the node to be deleted in a stack then after deleting the node we check all the node in this path for imbalance condition.

To delete a node from an AVL tree, it takes O (log n)

Deleting a node in AVL Tree

```
algorithm Delete(value)
Pre: the tree exist
Post: remove the value and the tree is a valid AVL tree
nodeToRemove ← root
parent = null
path = Stack()
while nodeToRemove != null and nodetToRemove.Value= value
    parent = nodeToRemove
    if value < nodeToRemove.Value
        nodeToRemove ← nodeToRemove.Left
    else
        nodeToRemove ← nodeToRemove.Right
    push(nodeToRemove, path)
 end while
 apply the regular binary search tree delete procedure (we check the four cases)
 while ! isEmpty(path)
     CheckBalance(pop(path))
end while
end InsertNode
```

AVL Tree Implementation

```
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};
```

Self-Assessment

Insert the following keys {8, 3, 1, 7, 5, 2, 4, 6, 9 }in an empty AVL tree

How many rotation operations required to balance the tree?

How many left rotation, right rotation, left-right rotation, right-left rotation are required?

Delete node 7 then node 3

You can use this AVL applet to check your answer

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html