Data Structure & Algorithms

Lec 11: Binary Search Trees
AVL Trees

Fall 2017 - University of Windsor
Dr. Sherif Saad

Agenda

1. Binary Search Tree
2. Balanced Binary Tree
3. AVL Tree - Self Balancing Binary Tree (Part 1)

Learning Outcome

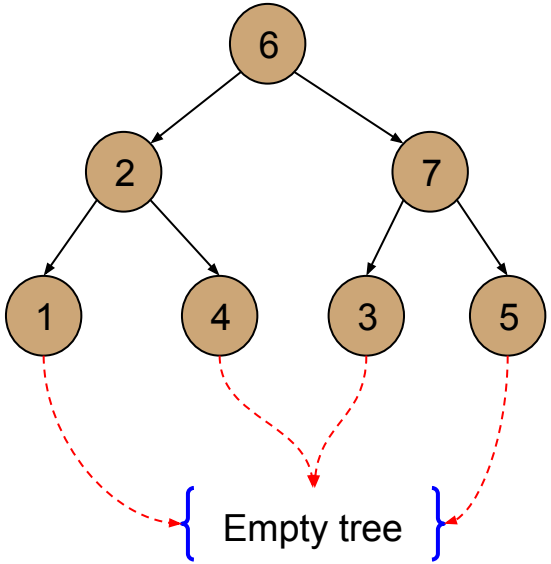
By the end of this class you should be able to:

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.

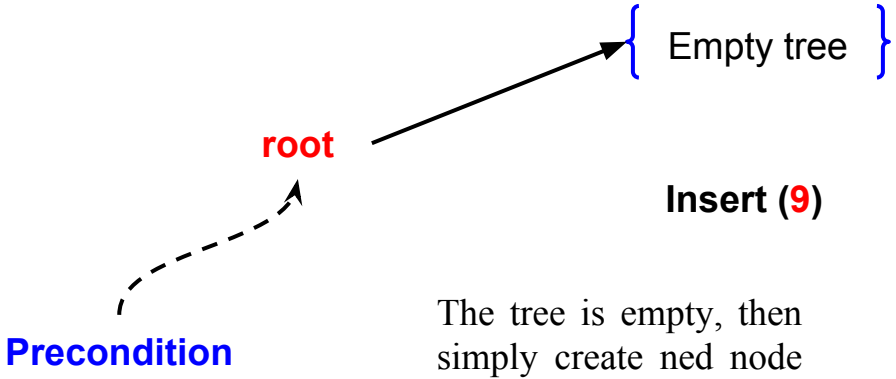
What is Binary Search Tree?

For any BST with a root node with value x, all the nodes in the left subtree of the root contains values < x and all the nodes in the right subtree of the root contains values >= x

In reasonably balanced BST all the insertion, deletion, and search operations can be done in O(log n) time.

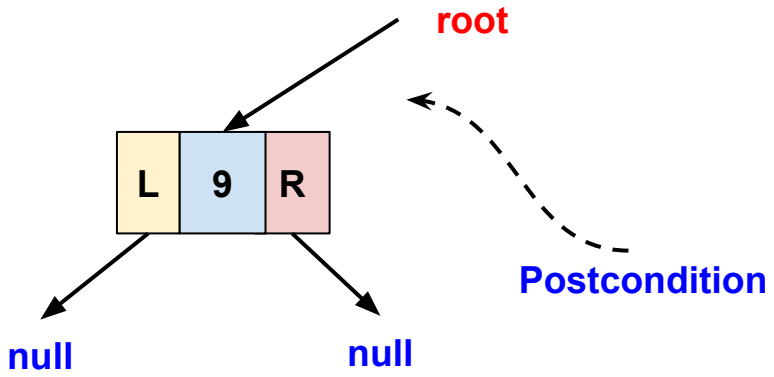


Inserting a Node in a Binary Search Tree

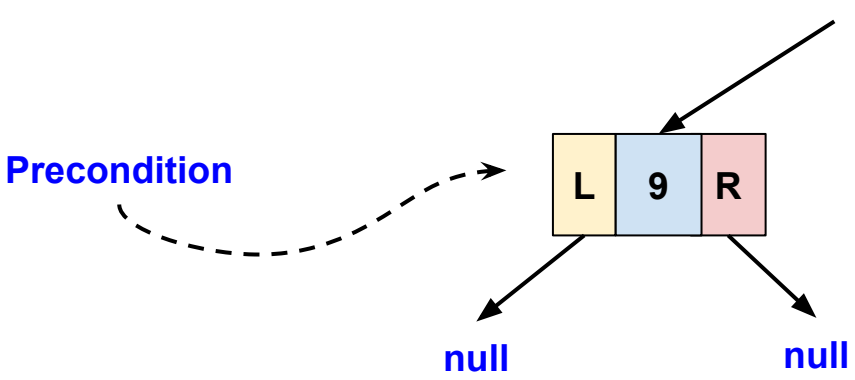


Insert (9)

The tree is empty, then simply create ned node 9 and points the root to 9. (now 9 is currently the root of the tree)

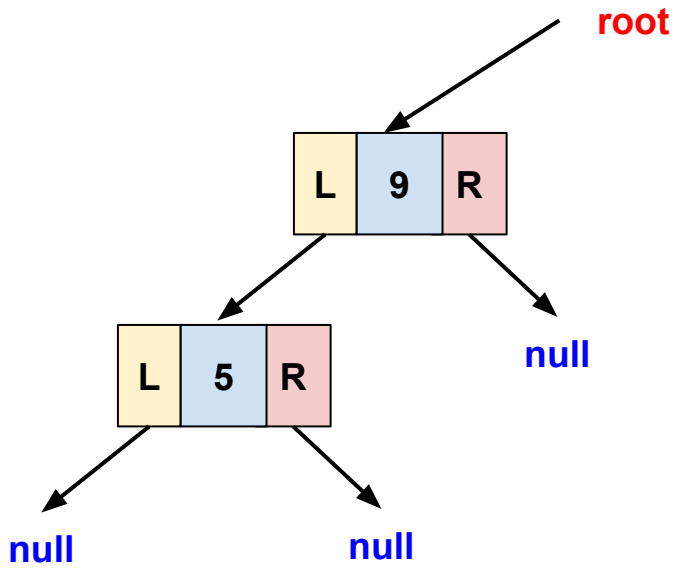


Inserting a Node in a Binary Search Tree

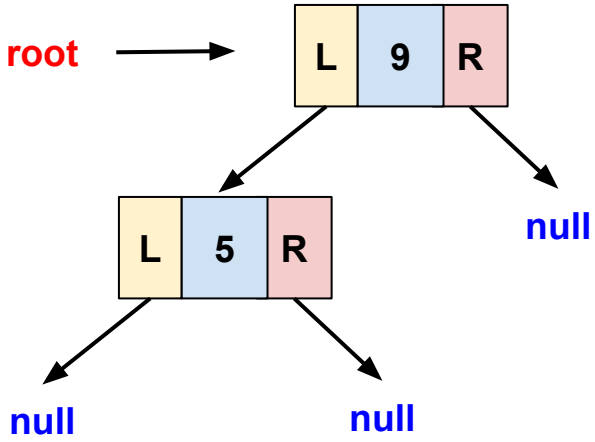


Insert (5)

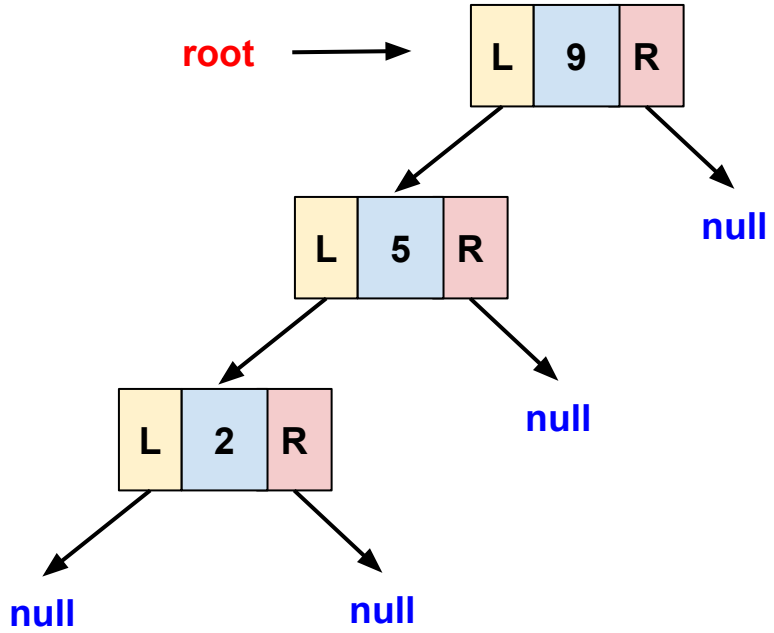
The tree is not empty, navigate the tree until we find the correct location (parent) of the new node.



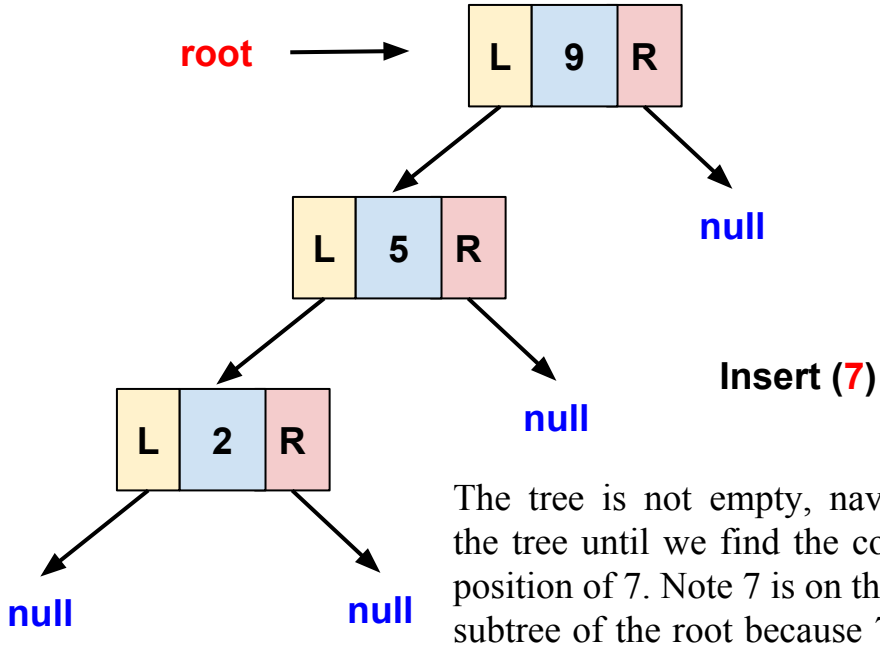
Inserting a Node in a Binary Search Tree



Insert (2)

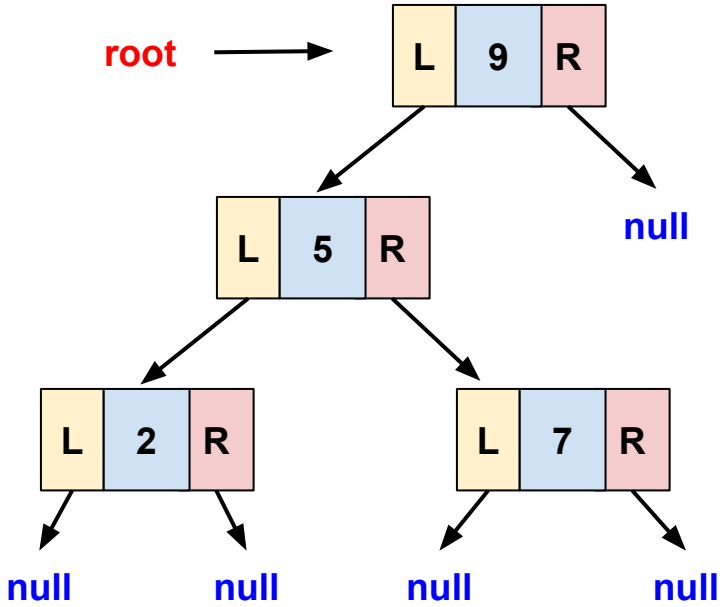


Inserting a Node in a Binary Search Tree



Insert (7)

The tree is not empty, navigate the tree until we find the correct position of 7. Note 7 is on the left subtree of the root because 7 < 9 and 7 is on the right subtree of it parent 5, because 7 > 5

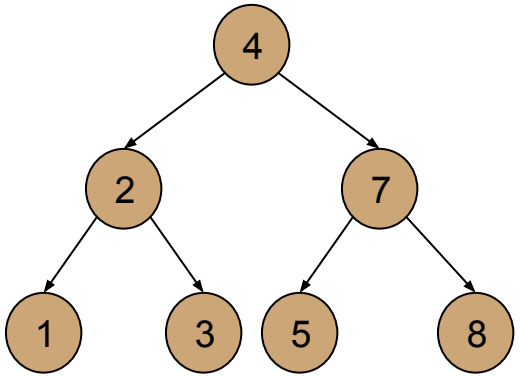


Inserting a Node in a Binary Search Tree

```
1. algorithm Insert(value)
2.     Pre: value has passed custom type checks for type T
3.     Post: value has been placed in the correct location in the tree
4.     if root = ∅
5.         root ← node(value)
6.     else
7.         InsertNode(root, value)
8.     end if
9. end Insert
```

Inserting a Node in a Binary Search Tree

```
1. algorithm InsertNode(current, value)
2.     Pre: current is the node to start from
3.     Post: value has been placed in the correct location in the tree
4.
5.     if value < current.value
6.         if current.Left = null
7.             current.Left ← node(value)
8.         else
9.
10.            InsertNode(current.Left, value)
11.        end if
12.    else
13.        if current.Right = null
14.            current.Right ← node(value)
15.        else
16.            InsertNode(current.Right, value)
17.        end if
18.    end if
19. end InsertNode
```



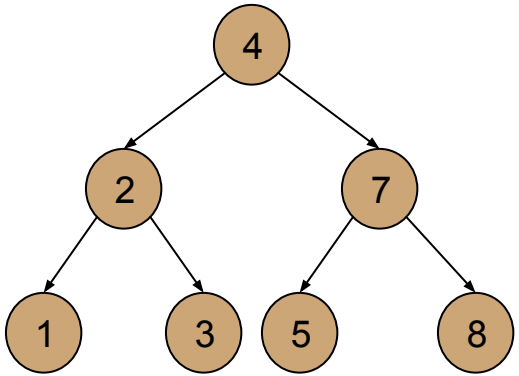
Searching in a Binary Search Tree

Searching a BST is a straightforward process:

1. If the root is null, the tree is empty, and in which case, lookup value is not in the tree
2. If the value of the root equal the look-up value then returns true.
3. If the value less than the root value then we check the left subtree of the root
4. If the value greater than the root value then we check the right subtree of the root.

Searching in a Binary Search Tree

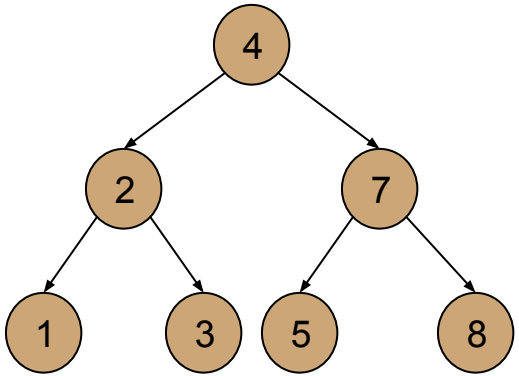
```
1. algorithm Search(root, value)
2.     Pre:  tree exist
3.     Post: value exist or not exist
4.     if root = null
5.         return not_found
6.     end if
7.     if root.Value = value
8.         return found
9.     else if value < root.Value
10.        return Search(root.Left, value)
11.    else
12.        return Search(root.Right, value)
13.    end if
14. end Search
```



Deleting a Node from a Binary Search Tree

When we try to delete a node from a binary search tree, we have four special cases. Could you guess these cases?

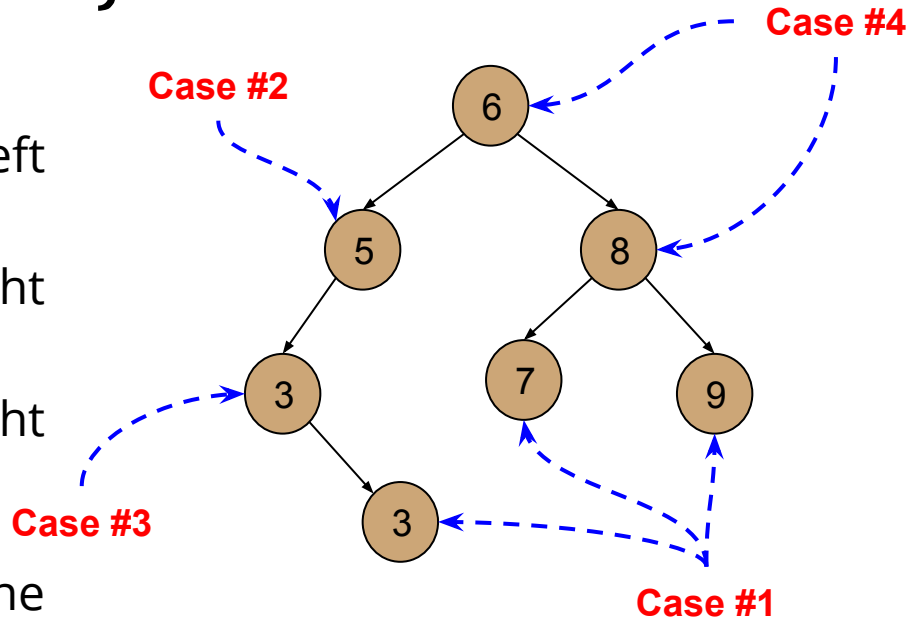
After deleting a node from a BST the BST MUST remain a valid BST



Deleting a Node from a Binary Search Tree

1. Deleting a leaf node.
2. Deleting a node that only has a left subtree (e.g 5)
3. Deleting a node that only has a right subtree.
4. Deleting a node with a left and right subtree (e.g. 6 and 8)

if a value happens to exist more than one in the BST then the first occurrence will be remove

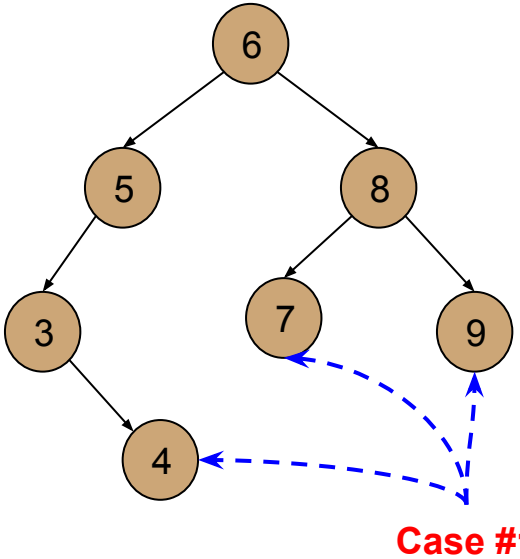


Deleting a Node from a Binary Search Tree

Case 1: we are deleting a leaf node.

We need to find the reference (pointer | address) of the node we want to delete.

```
1. nodeToDelete = FindNodeRef(value)
2. If nodeToDelete.Left is null &&  nodeToDelete.Right is null:
3.     nodeToDelete = null
```



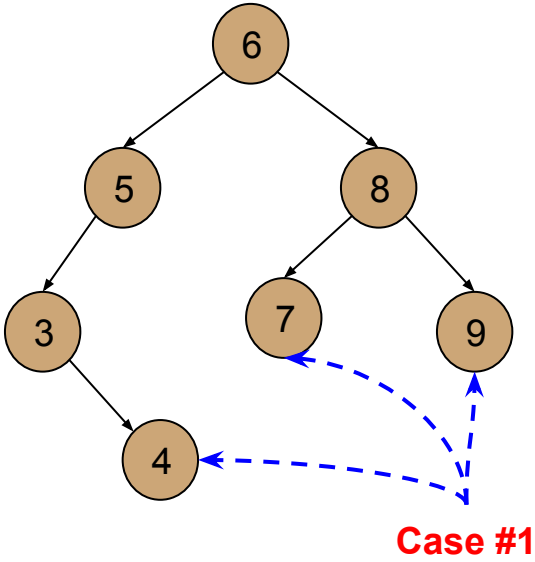
Not really accurate| precise, Why??

Deleting a Node from a Binary Search Tree

Case 1: we are deleting a leaf node.

We need to find the reference (pointer | address) of the node we want to delete and its parent node

```
1. nodeToDelete = FindNodeRef(value)
2. parent = FindParent(nodeToDelete)
3. If nodeToDelete.Left is null &&  nodeToDelete.Right is null:
4.     if nodeToDelete.value > parent.value:
5.         parent.right = null
6.     else:
7.         parent.left = null
8.     Free(nodeToDelete)
```

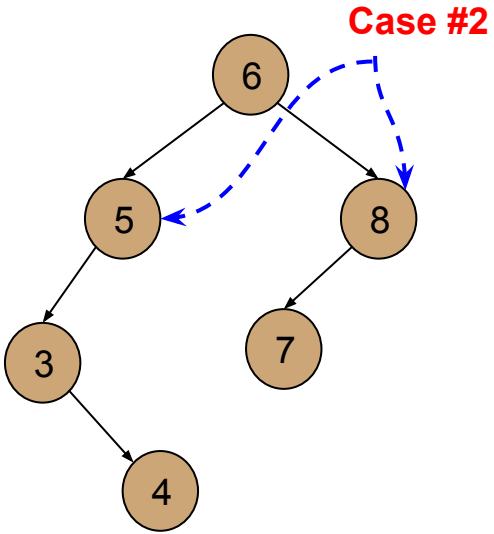


Deleting a Node from a Binary Search Tree

Case 2: we are deleting a node with only left sub tree

We need to find the reference (pointer | address) of the node we want to delete and its parent node

```
1. nodeToDelete = FindNodeRef(value)
2. parent = FindParent(nodeToDelete)
3. If nodeToDelete.Left is not Null &&  nodeToDelete.Right is Null:
4.     if nodeToDelete.value <  parent.value:
5.         parent.Left  = nodeToDelete.Left
6.     else:
7.         parent.Right = nodeToDelete.Left
8.     Free(nodeToDelete)
```

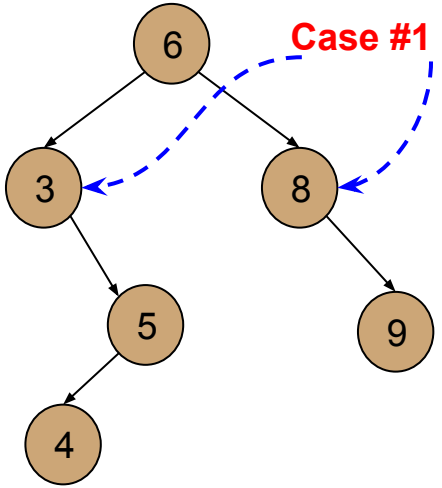


Deleting a Node from a Binary Search Tree

Case 3: we are deleting a node with only right subtree

We need to find the reference (pointer | address) of the node we want to delete and its parent node

```
1. nodeToDelete = FindNodeRef(value)
2. parent = FindParent(nodeToDelete)
3. If nodeToDelete.Left is Null &&  nodeToDelete.Right is not Null:
4.     if nodeToDelete.value <  parent.value:
5.         parent.Left  = nodeToDelete.Right
6.    else:
7.        parent.Right = nodeToDelete.Right
8.    Free(nodeToDelete)
```

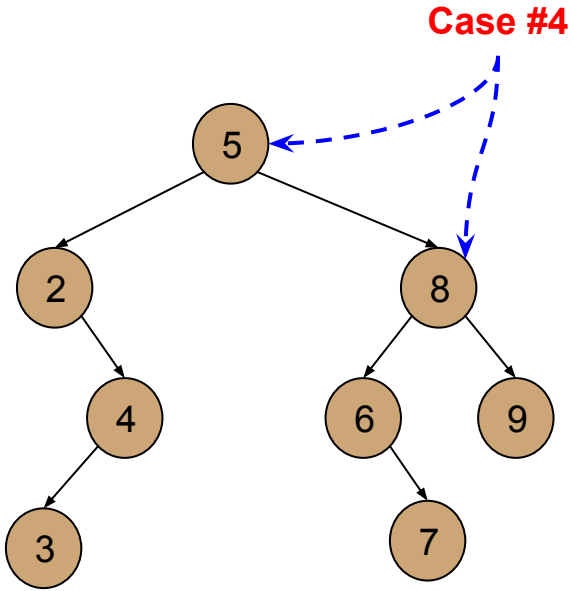


Deleting a Node from a Binary Search Tree

Case 4: Deleting a node with a left and right subtree

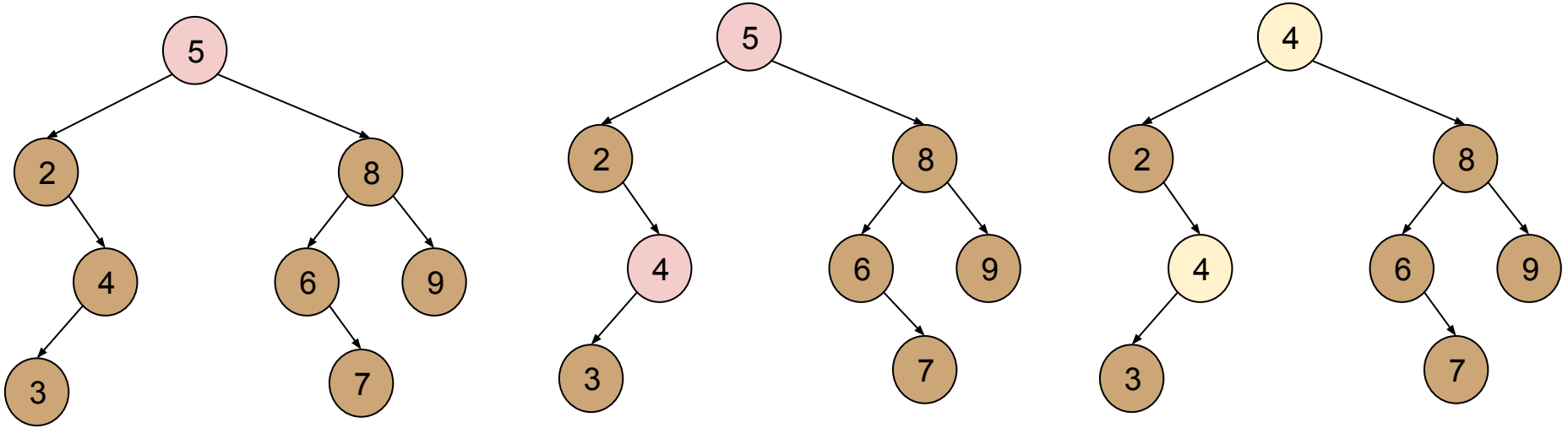
We find the node with the largest value in the left subtree of the node to be deleted. Then we replace the value of the node to be deleted with the largest value in its left subtree.

Notice that the node with maximum value has no right child and, therefore, it's removal may result in first or second cases only.



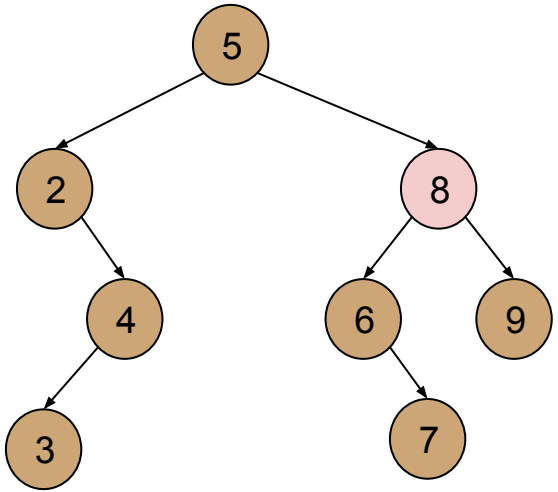
Deleting a Node from a Binary Search Tree

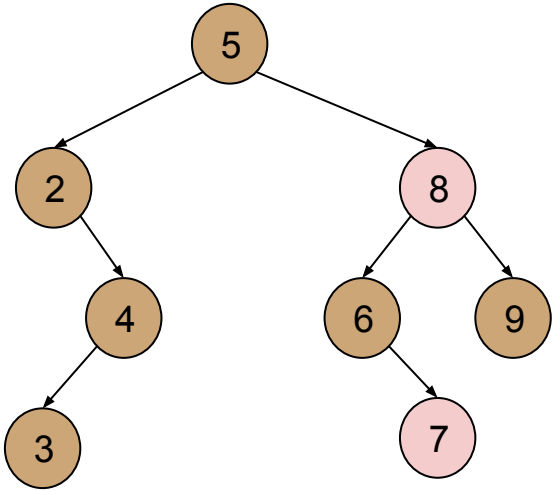
Case 4: Deleting a node with a left and right subtree

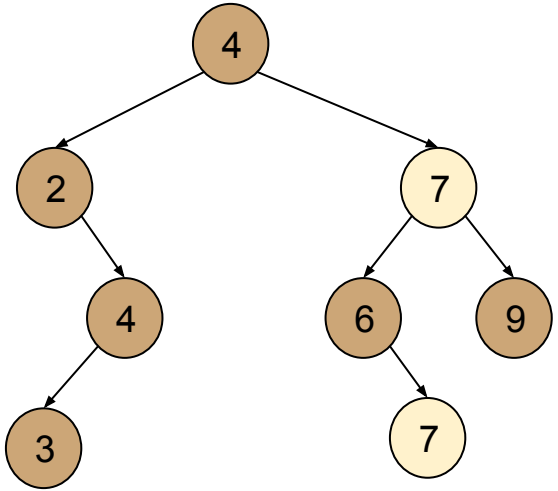


Deleting a Node from a Binary Search Tree

Case 4: Deleting a node with a left and right subtree



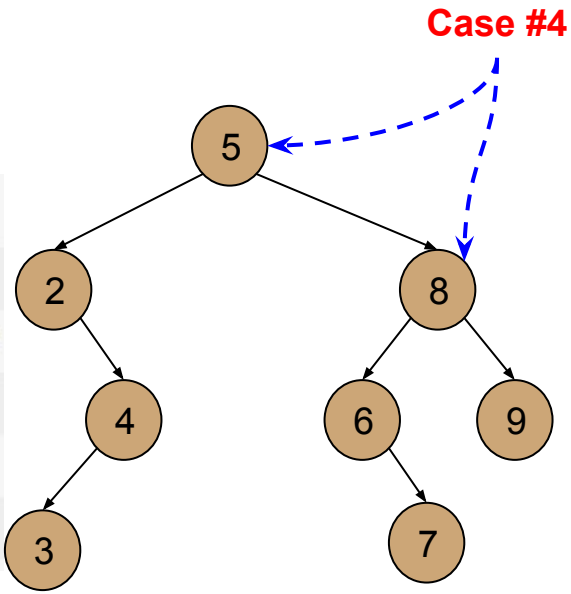




Deleting a Node from a Binary Search Tree

Case 4: Deleting a node with a left and right subtree

```
1. nodeToDelete = FindNodeRef(value)
2. parent = FindParent(nodeToDelete)
3. If nodeToDelete.Left is not Null &&  nodeToDelete.Right is not Null:
4.       maxNode = FindMaxNode(nodeToDelete.left)
5.     nodeToDelete.Value = maxNode.Value
6. delete(maxNode)
```



Deleting a Node from a Binary Search Tree

There are two special cases when deleting a node from a BST we did not explicitly, what are these cases?

Find Node with Given Value

```
1. algorithm FindNode(root, value)
2.     Pre: Tree exist
3.     Post: a reference to the node of value if found; otherwise null
4.     if root =Null
5.         return Null
6.     end if
7.     if root.Value = value
8.         return root
9.     else if value < root.Value
10.        return FindNode(root.Left, value)
11.    else
12.        return FindNode(root.Right, value)
13.    end if
14. end FindNode
```

Find Parent of a Given Node

```
1. algorithm FindParent(value, root)
2.     Pre: the tree exist and not empty
3.     Post: a reference to the parent node of value if found otherwise Null
4.     if value = root.Value
5.         return Null
6.     end if
7.     if value < root.Value
8.         if root.Left = Null
9.             return Null
10.        else if root.Left.Value = value
11.            return root
12.        else
13.            return FindParent(value, root.Left)
14.        end if
15.    else
16.        if root.Right = Null
17.            return Null
18.        else if root.Right.Value = value
19.            return root
20.        else
21.            return FindParent(value, root.Right)
22.        end if
23.    end if
```

Find the Maximum or Minimum Value in BST

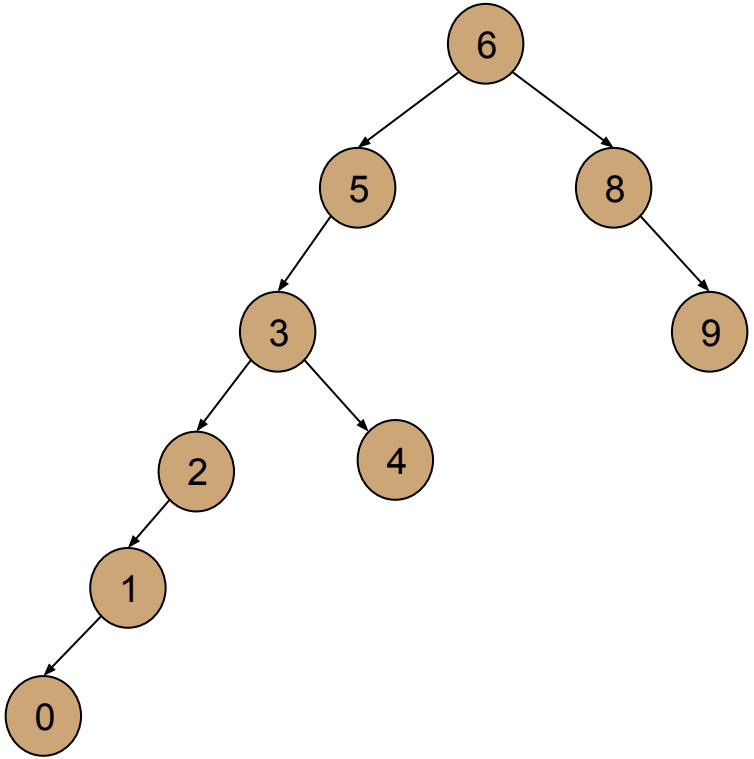
```
1. algorithm FindMin(root)
2.     Pre: tree exist
3.     Post: a reference to the node of minimumvalue or Null
4.      "Use recursive to find the minimum "
5.
6. end FindMin
7.
8. algorithm FindMax(root)
9.     Pre: tree exist
10.    Post: a reference to the node of maximum value or Null
11.     "Use recursive to find the maximum"
12.
13. end FindMin
```

Balancing Binary Tree

The worst-case performance for a BST is O(n) this happen when we have a skewed BST.

BST insertion, deletion, and search operations can be done in O(log n) time, if the BST is a balanced binary tree.

What is a balanced binary tree?



Balanced Binary Tree

A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1 (Weight-Balance)

A binary tree is balanced if for any two leaves the difference of the depth is at most 1 (Height Balance)

Note: Height-balance does not imply Weight-Balance. Could you give an example?

Balanced Binary Tree

There are many algorithms for keeping binary search trees balanced

- Height-Balanced Trees (including AVL Tree)
- Weight-Balanced Trees (many applications in functional programming)
- Red-Black Trees
- Splay Trees
- B-Tress (2-4 Trees)

Techniques For Balancing Binary Trees

Our goal is to keep the tree balanced after any insertion or deletion operation.

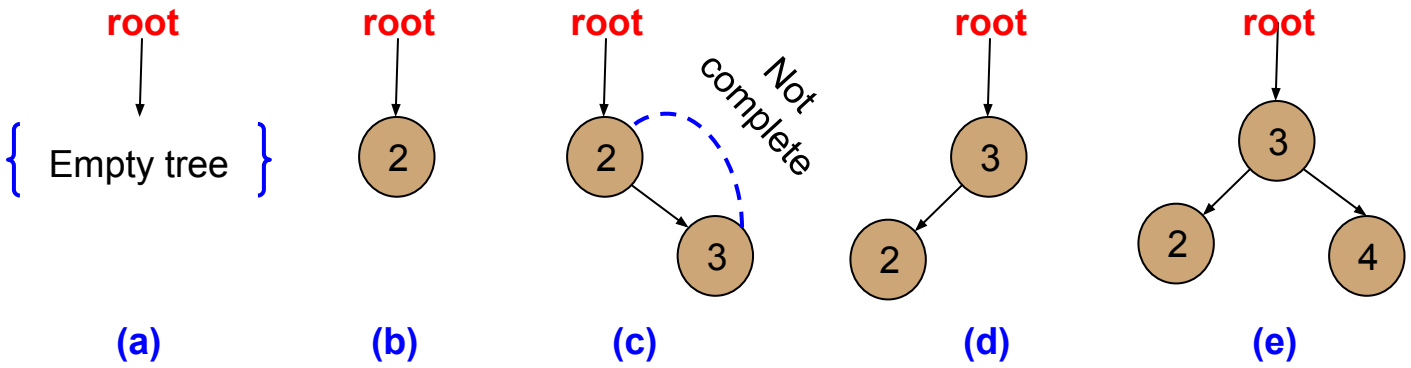
We want to be able to balance the tree in at most O(log n) after each operation.

- Prefect or Strict Balance: the tree must always be balanced perfectly.
- Pretty Good Balance: almost balanced but not perfect. (some imbalance is allowed)
- Adjust on Access: self-adjusting the tree to reach good balance
- Do Nothing: relies on the randomness of data to keep depth to approximately log n.

Perfect or Strict Balance

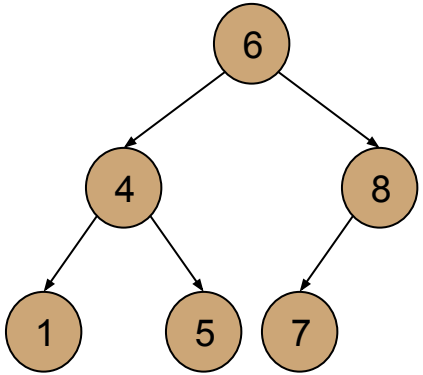
After every operation (insert or delete) the result must be a complete tree. The perfect balance requirement is expensive.

Assume we will insert the following values {2, 3, 4} in an empty binary tree

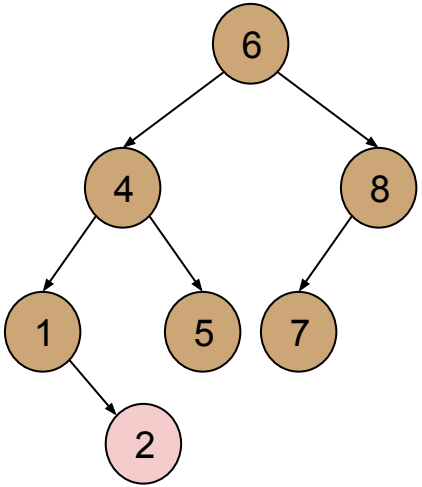


Perfect or Strict Balance

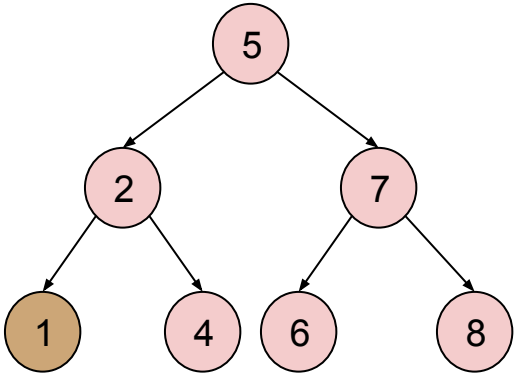
Let us assume we want to insert value 2 in the following complete binary tree



(a)



(b)



(c)

AVL Tree

The first self-balancing binary search tree (data structure)

AVL trees are height-balanced binary search trees.

AVL trees are not perfect balance trees.

Invented in 1962 by two Russian mathematicians Georgii Adelson-Velsky (1922-2014) and Evgenii Mikhailovich Landis (1921-1997) AVL Tree.

AVL Trees

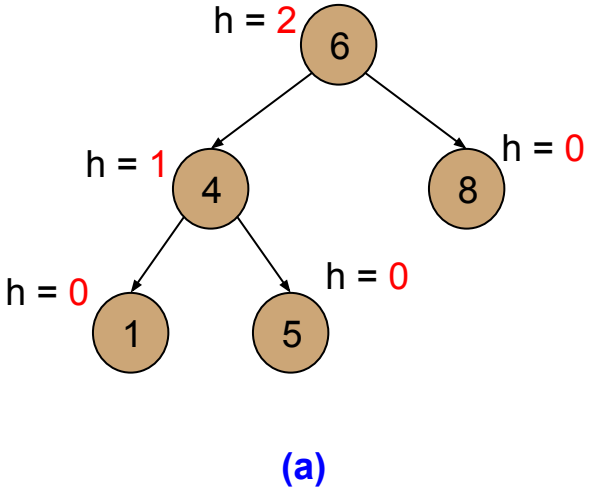
AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1.

For each node, the height of the left and right subtrees can differ by no more than 1.

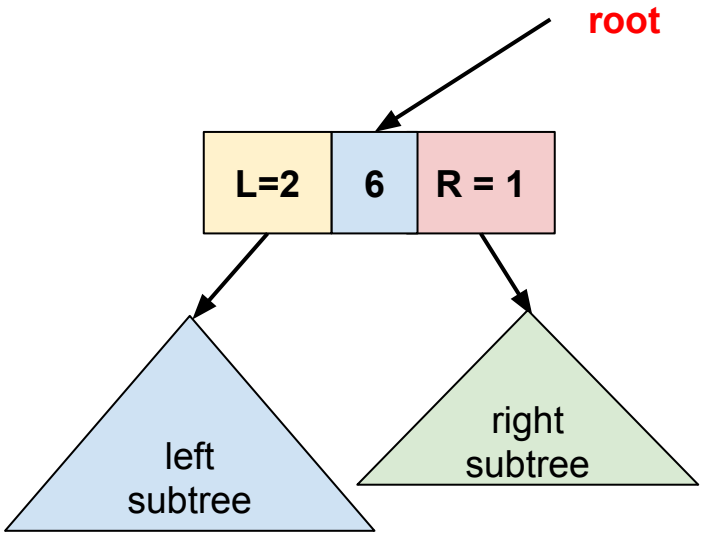
We need to store or calculate the height of each node

This difference is called the Balance Factor. Where the Balance factor, ***BalanceFactor(T) = height(T.left) - height(T.right)***

AVL Tree and Node Heights

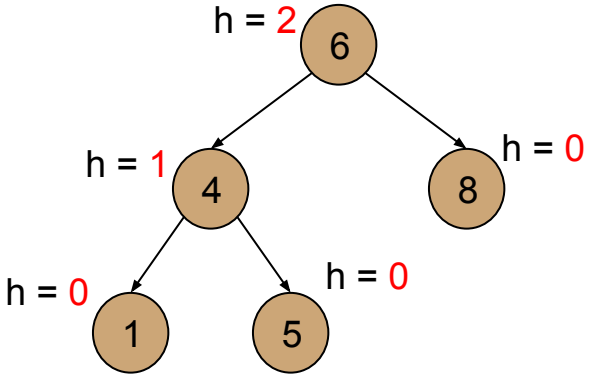


(a)



Balance factor = height$_{left}$ - height$_{right}$

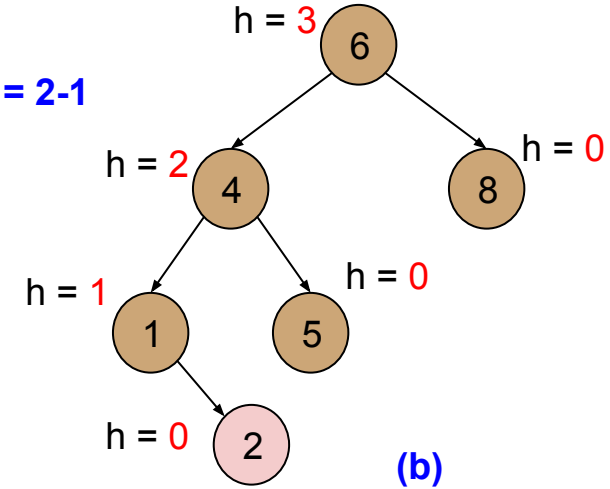
AVL Tree and Node Heights



(a)

Balance factor = 2-1

Balance factor = 3-1



(b)

Balancing AVL Tree

When inserting a new value (node) in an AVL the balance factor could become 2 or -2 for at least one node.

Only the nodes on the path from the insertion location to the root node could be affected (change their height)

After we insert a node we need to check for possible imbalance, by going back to root node by node and update the heights and calculate the balance factor.

If we found imbalance we adjust the tree by applying rotation around the imbalanced node

AVL Tree Rotation

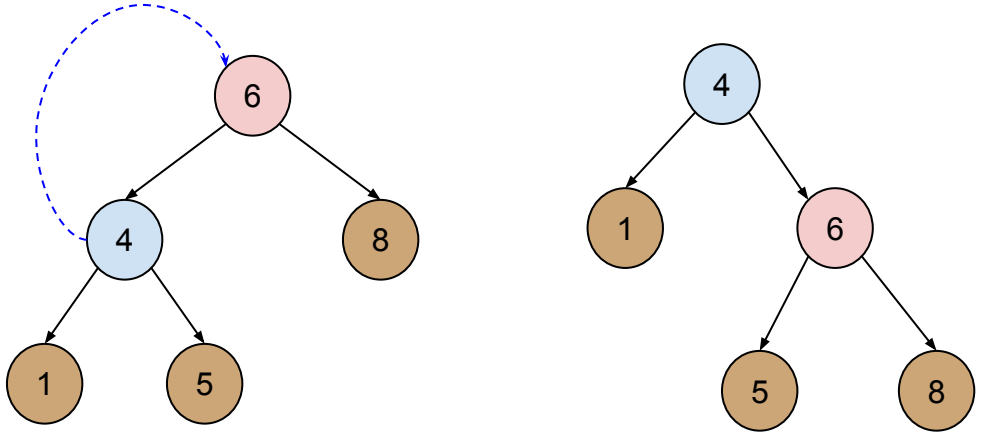
To balance an AVL tree after insertion or deletion operation, the AVL tree perform the following four kinds of rotations:

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

AVL Tree : Single Rotation

Left Rotation: when applying leaf rotation on node x, node x, and its parent w switch locations.

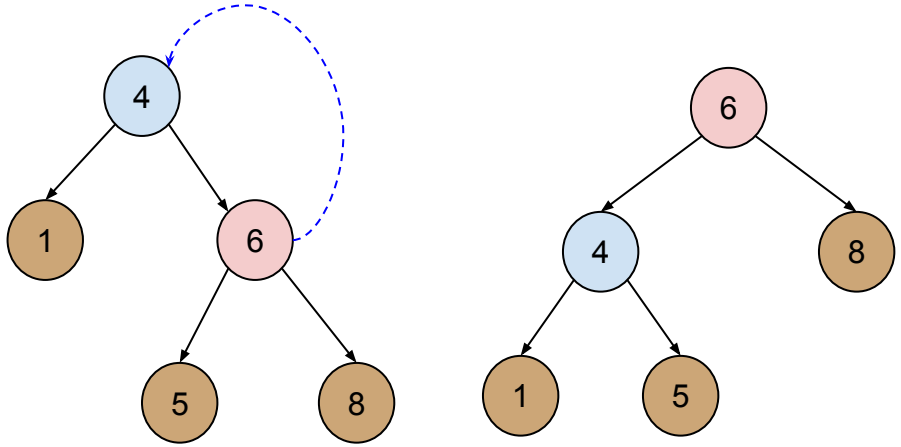
In this case, w becomes the right subtree of x and x right subtree becomes the left subtree of w



AVL Tree : Single Rotation

Right Rotation: when applying right rotation on node x, node x, and its parent w switch locations.

In this case w becomes the left subtree of x and x left subtree becomes the right subtree of x



AVL Single Rotation

What exactly rotation does?

Changes the structure without interfering with the order of the elements. A tree rotation moves one node up in the tree and one node down.

There are left and right rotations both of them decrease the height of the tree by moving smaller subtrees down and larger subtrees up.

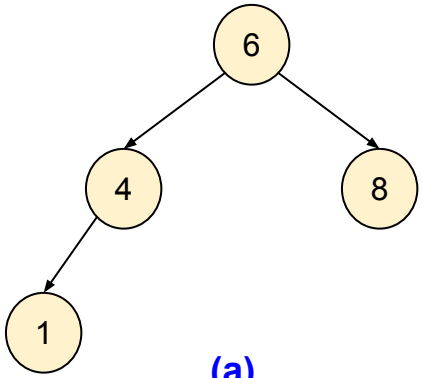
It is possible that we need more than one rotation (double rotation) to restore the balance of the tree.

AVL Tree: Left Rotation

```
1. algorithm LeftRotation(childNode, parentNode)
2.    Pre: childNode is the root of the left subtree of parentNode
3.    Post: the right subtree of childNode become the left subtree of
4.    parentNode and childNode is the parent of parentNode
5.          rightTemp ← childNode.Right
6.    childNode.Right ←parentNode
7.    parentNode.Left ←rightTemp
8. end LeftRotation
```

Self-Assessment

Given the tree in figure (a) is it an AVL tree? If we insert 0 in this tree what is the structure of the tree.



(a)