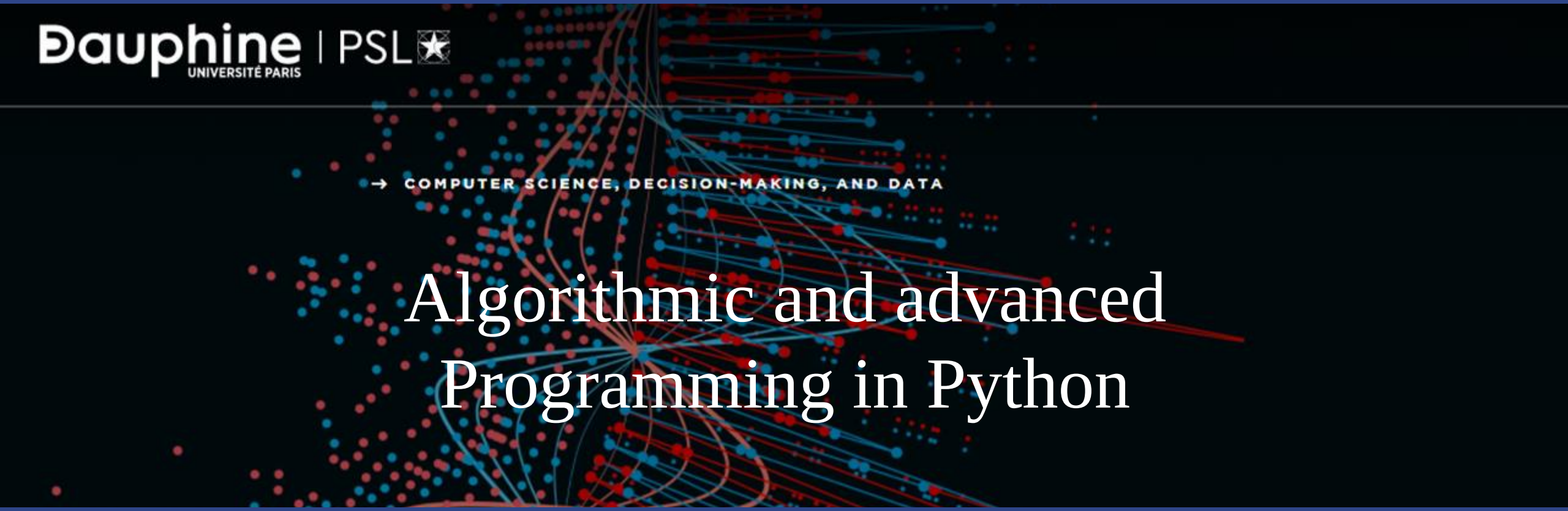


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 6

Outline

1. Generic trees
2. Threaded trees
3. Expression Trees
4. AVL Trees

Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask

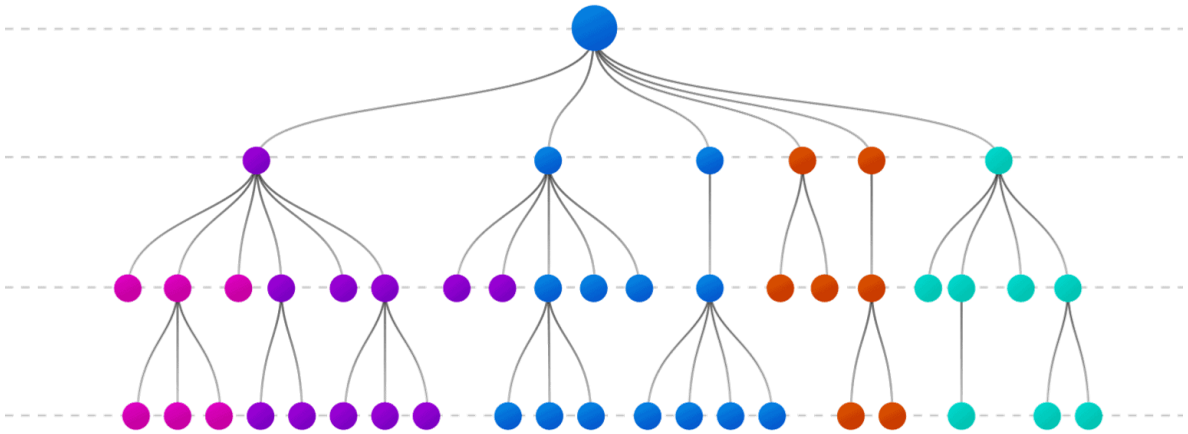
- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Reminder of previous session

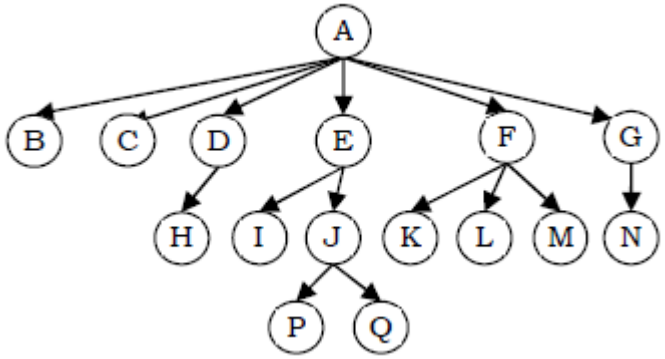
- In Master class 6, we discuss about binary trees
- Question: can you summarize the different type of binary tree traversals?

Generic tree?

- In the previous class, we discussed about binary trees where each node can have a maximum of two children and these are represented easily with two pointers.
- But suppose we have a tree with many children at every node and we do not know how many children a node can have, how do we represent them?



How do we represent the tree?



- In the above tree, there are nodes with 6 children, with 3 children, with 2 children, with 1 child, and with zero children (leaves).
- To present this tree we have to consider the worst case (6 children) and allocate that many child pointers for each node. Based on this, the node representation can be given as:

A naïve implementation

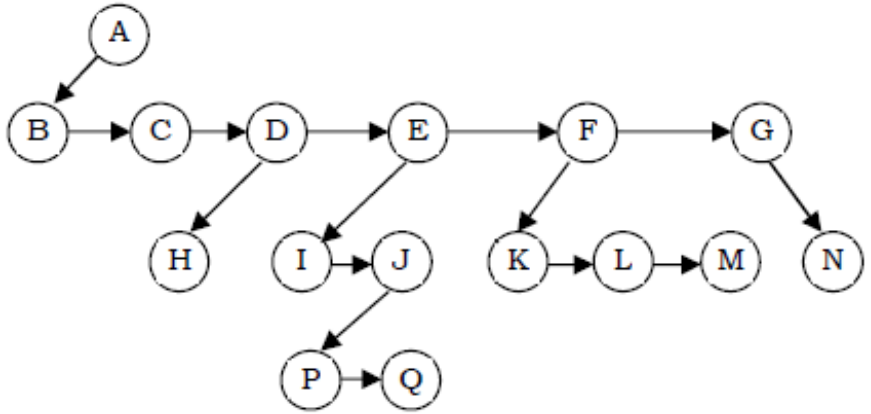
```
@author: eric.benhamou
"""

class TreeNode:
    def __init__(self,data):
        self.data=data
        self.Child1 = None
        self.Child2 = None
        self.Child3 = None
        self.Child4 = None
        self.Child5 = None
        self.Child6 = None
```

- But this is way too naïve.
- Question: why? What can we do to improve?

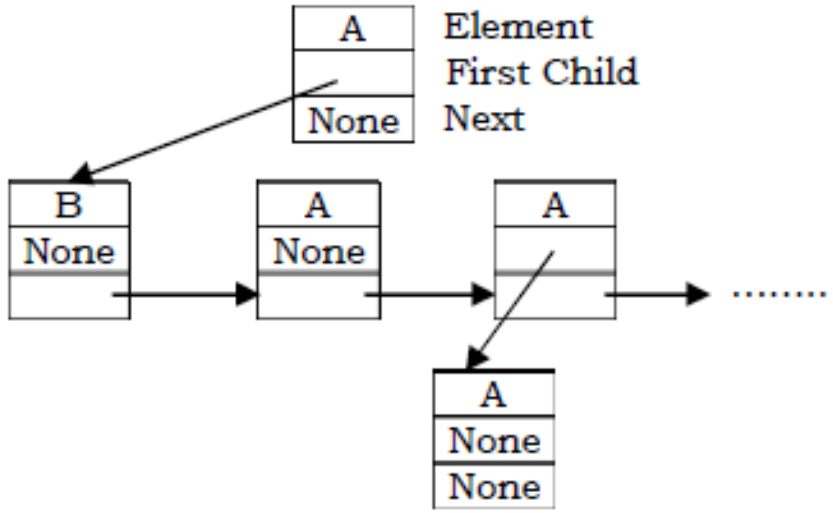
Representation of Generic Trees

- Since our objective is to reach all nodes of the tree, a possible solution to this is as follows:
  - At each node link children of same parent (siblings) from left to right.
  - Remove the links from parent to all children except the first child.



- Question: how would call this representation?

- This representation is sometimes called first child/next sibling representation.
- First child/next sibling representation of the generic tree is shown above. The actual representation for this tree is:



Code

```
@author: eric.benhamou
"""

class TreeNode:
    def __init__(self, data = None, firstChild = None, nextSibling = None):
        self.data = data
        self.firstChild = firstChild
        self.nextSibling = nextSibling
```

- Question: how do this relate to binary tree?

Link with binary tree!

- This representation indeed converts any generic tree to binary representation; in practice we use binary trees. We can treat all generic trees with a first child/next sibling representation as binary trees.

Implementation of a simple N-ary tree?

```
""" Generic n-ary tree node object
    Children are additive; no provision for deleting them.
    The birth order of children is recorded: 0 for the first
    child added, 1 for the second, and so on.
        GenericTree(parent, value=None)    Constructor
        parent                If this is the root node, None, otherwise the
            parent's GenericTree object.
        childList             List of children, zero or more GenericTree objects.
        value                 Value passed to constructor; can be any type.
        birthOrder            If this is the root node, 0, otherwise the
            index of this child in the parent's .childList
        nChildren()           Returns the number of self's children.
        nthChild(n)           Returns the nth child; raises IndexError
            if n is not a valid child number.
        fullPath():           Returns path to self as a list of child numbers.
        nodeId():             Returns path to self as a NodeId.
"""
```

Corresponding code

```
class GenericTree:
    def __init__ (self, parent, value=None):
        self.parent = parent
        self.value = value
        self.childList = []
        if  parent is None:
            self.birthOrder = 0
        else:
            self.birthOrder = len(parent.childList)
            parent.childList.append(self)

    def nChildren (self):
        return len(self.childList)

    def nthChild (self, n):
        return self.childList[n]

    def fullPath (self):
        result = []
        parent = self.parent
        kid = self
        while  parent:
            result.insert (0, kid.birthOrder)
            parent, kid = parent.parent, parent
        return result

    def nodeId (self):
        fullPath = self.fullPath()
        return NodeId (fullPath)
```

And NodeId

```python
class NodeId:
    def __init__ (self, path):
        self.path = path

    def __str__ (self):
        L = map (str, self.path)
        return string.join (L, "/")

    def find (self, node):
        return self.__reFind (node, 0)

    def __reFind (self, node, i):
        if  i >= len(self.path):
            return node.value  # We're there!
        else:
            childNo = self.path[i]
        try:
            child = node.nthChild (childNo)
        except IndexError:
            return None
        return self.__reFind (child, i + 1)

    def isOnPath (self, node):
        if  len(nodePath) > len(self.path):
            return 0  # Node is deeper than self.path

        for  i in range(len(nodePath)):
            if  nodePath[i] != self.path[i]:
                return 0  # Node is a different route than self.path
        return 1
```

Threaded binary tree traversal

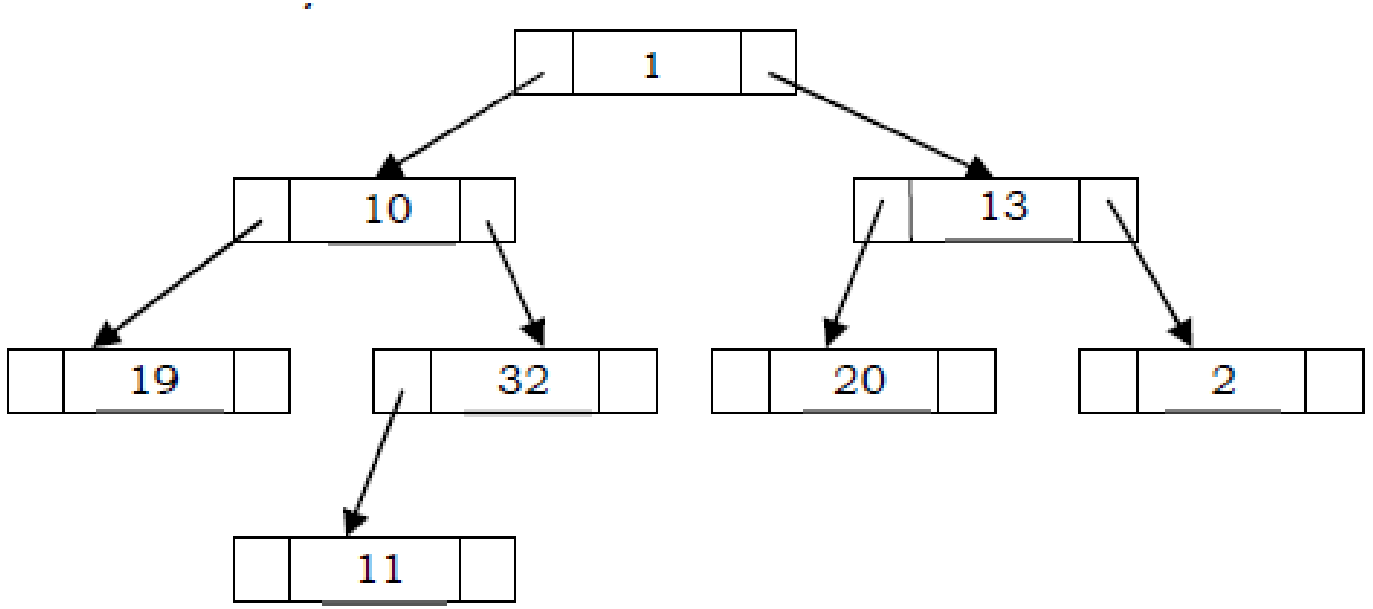
- In earlier class, we have seen that, $preorder$, $inorder\ and\ postorder$ binary tree traversals used stacks and $level\ order$ traversals used queues as an auxiliary data structure.

- In this section we will discuss new traversal algorithms which do not need both stacks and queues. Such traversal algorithms are called
- $threaded\ binary\ tree\ traversals$ or $stack/queue - less\ traversals$

- Question: What can be the issue with regular binary tree traversals?

Hint for the question

- Question: What can be the issue with regular binary tree traversals



Issues with Regular Binary Tree Traversals

- The storage space required for the stack and queue is large.
- The majority of pointers in any binary tree are nil. For example, a binary tree with n nodes has $n + 1$ nil pointers and these were wasted.
- It is difficult to find successor node (preorder, inorder and postorder successors) for a given node.

One solution; threaded binary trees

- To solve these problems, one idea is to store some useful information in NULL pointers. If we observe the previous traversals carefully, stack/queue is required because we have to record the current position in order to move to the right subtree after processing the left subtree. If we store the useful information in nil pointers, then we don't have to store such information in stack/queue.

- The binary trees which store such information in nil pointers are called *threaded binary trees*. From the above discussion, let us assume that we want to store some useful information in nil pointers.

- The next question is what to store?

Threaded Binary Trees

- The common convention is to put predecessor/successor information. That means, if we are dealing with preorder traversals, then for a given node, nil left pointer will contain preorder predecessor information and nil right pointer will contain preorder successor information. These special pointers are called *threads*.

Classifying Threaded Binary Trees

- The classification is based on whether we are storing useful information in both nil pointers or only in one of them
- If we store predecessor information in nil left pointers only, then we can call such binary trees $left\ threaded\ binary\ trees$.
- If we store successor information in nil right pointers only, then we can call such binary trees $right\ threaded\ binary\ trees$.
- If we store predecessor information in nil left pointers and successor information in nil right pointers, then we can call such binary trees $fully\ threaded\ binary\ trees$ or simply $threaded\ binary\ trees$.

- Note: Fo r the remaining discussion we consider only ($fully$) $threaded\ binary\ trees$.

Types of Threaded Binary Trees

- According to you what are the tree types of threaded binary trees?

Types of Threaded Binary Trees

- According to you what are the tree types of threaded binary trees?
- Based on above discussion we get three representations for threaded binary trees.
  - *Preorder Threaded Binary Trees*: nil left pointer will contain PreOrder predecessor information and nil right pointer will contain PreOrder successor information.
  - *Inorder Threaded Binary Trees*: nil left pointer will contain InOrder predecessor information and nil right pointer will contain InOrder successor information.
  - *Postorder Threaded Binary Trees*: nil left pointer will contain PostOrder predecessor information and nil right pointer will contain PostOrder successor information.
- Do you see any major difference between these threaded binary trees?

They are all similar!

- As the representations are similar, for the remaining discussion we will use InOrder threaded binary trees.

Threaded Binary Tree structure

- Any program examining the tree must be able to differentiate between a regular *left*/*right* pointer and a *thread*. To do this, we use two additional fields in each node, giving us, for threaded trees, nodes of the following form:


```python
'''Threaded Binary Tree Class and its methods'''
class ThreadedBinaryTree:
  def __init__(self, data):
    self.data = data  # data
    self.left = None  # left child
    self.LTag = None
    self.right = None  # right child
    self.RTag = None
```

Difference between Binary Tree and Threaded Binary Tree Structures

- According to you what are the major difference between Binary Tree and Threaded Binary Tree Structures?

Difference between Binary Tree and Threaded Binary Tree Structures

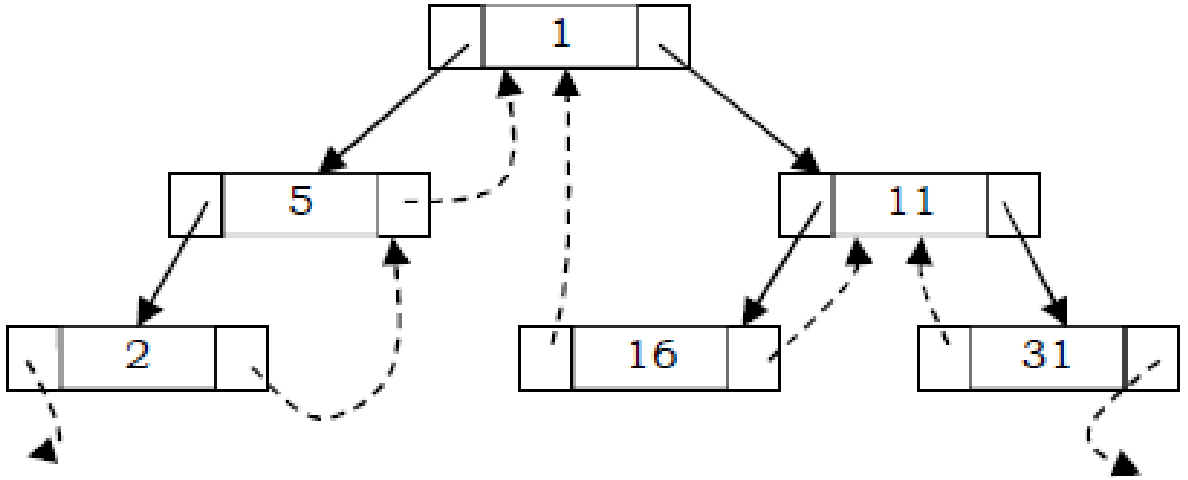
- According to you what are the major difference between Binary Tree and Threaded Binary Tree Structures?

	Regular Binary Trees	Threaded Binary Trees
if LTag == 0	nil	left points to the in-order predecessor
if LTag == 1	left points to the left child	left points to left child
if RTag == 0	nil	right points to the in-order successor
if RTag == 1	right points to the right child	right points to the right child

- Note: Similarly, we can define preorder/postorder differences as well.

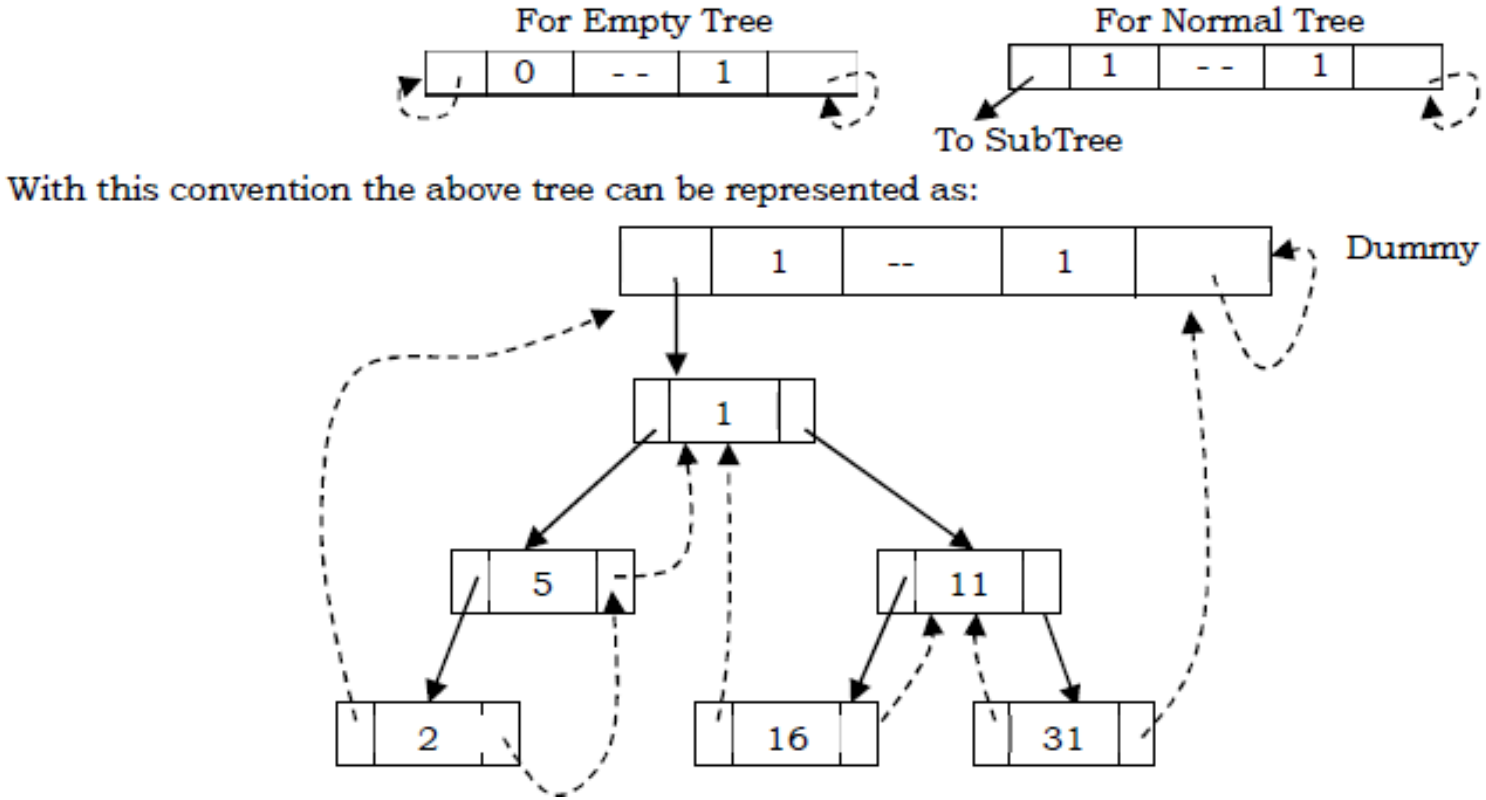
Difference between Binary Tree and Threaded Binary Tree Structures

- As an example, let us try representing a tree in inorder threaded binary tree form. The tree below shows what an inorder threaded binary tree will look like. The dotted arrows indicate the threads. If we observe, the left pointer of left most node (2) and right pointer of right most node (31) are hanging.



What should leftmost and rightmost pointers point to?

- In the representation of a threaded binary tree, it is convenient to use a special node *Dummy* which is always present even for an empty tree. Note that right tag of Dummy node is 1 and its right child points to itself.



Finding Inorder Successor in Inorder Threaded Binary Tree

- To find inorder successor of a given node without using a stack, assume that the node for which we want to find the inorder successor is *P*.

- **Strategy**: If *P* has a no right subtree, then return the right child of *P*. If *P* has right subtree, then return the left of the nearest node whose left subtree contains *P*.

Corresponding code

```
def InorderSuccessor(P):
  if(P.RTag == 0):
    return P.right
  else:
    Position = P.right
    while(Position.LTag == 1):
      Position = Position.left
      return Position
```

- Question: What is the Time Complexity and Space Complexity?

Corresponding code

```
def InorderSuccessor(P):
  if(P.RTag == 0):
    return P.right
  else:
    Position = P.right
    while(Position.LTag == 1):
      Position = Position.left
      return Position
```

- Question: What is the Time Complexity and Space Complexity?
- Time Complexity: O(n). Space Complexity: O(1).

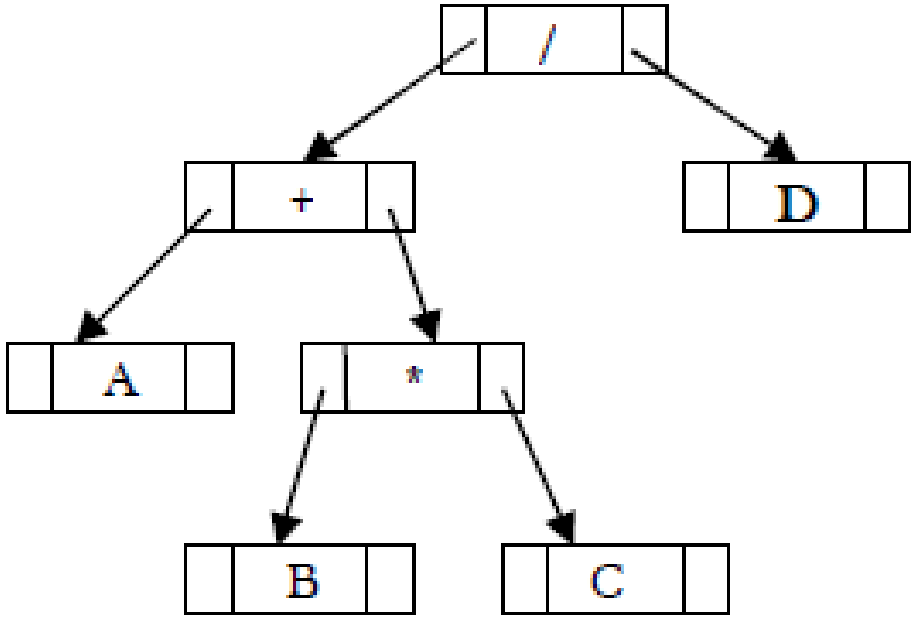
Expression Trees

- A tree representing an expression is called an *expression tree*. In expression trees, leaf nodes are operands and non-leaf nodes are operators.

- That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of binary expression. But for a unary operator, one subtree will be empty.

Expression trees

- The figure below shows a simple expression tree for (A + B * C) / D.

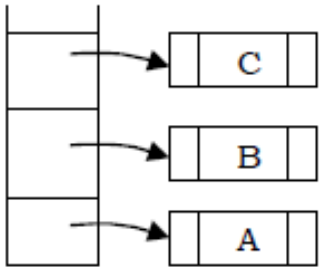


Expression trees examples

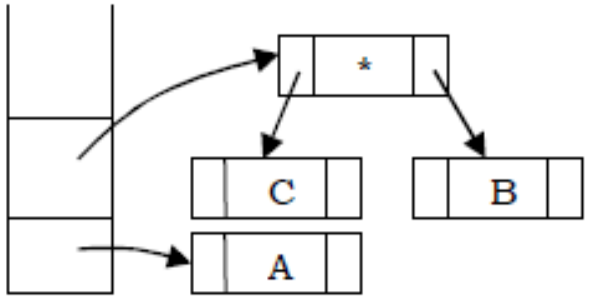
- Example: Assume that one symbol is read at a time. If the symbol is an operand, we create a tree node and push a pointer to it onto a stack. If the symbol is an operator, pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

- As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.

Steps

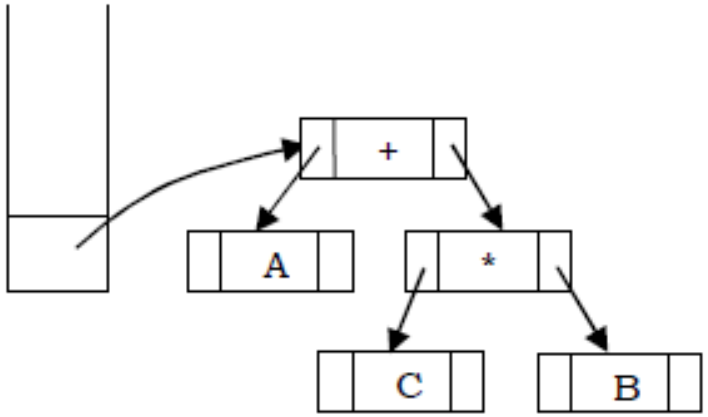


- Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



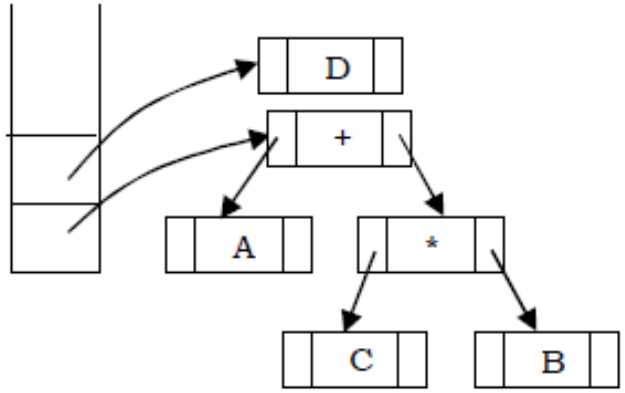
Steps

- Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.

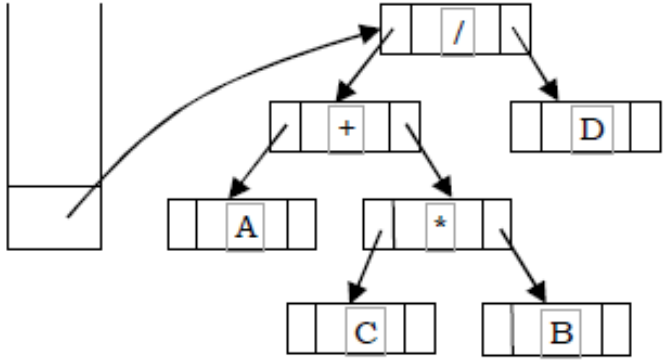


- Next, an operand ‘D’ is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

Next



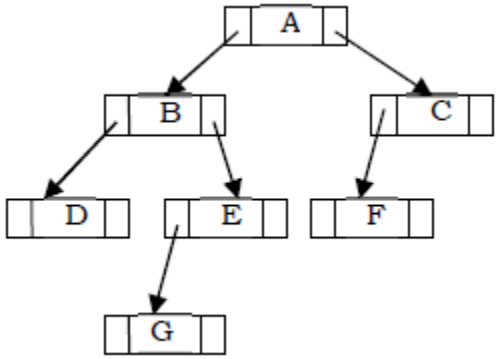
- Finally, the last symbol (‘/’) is read, two trees are merged and a pointer to the final tree is left on the stack.



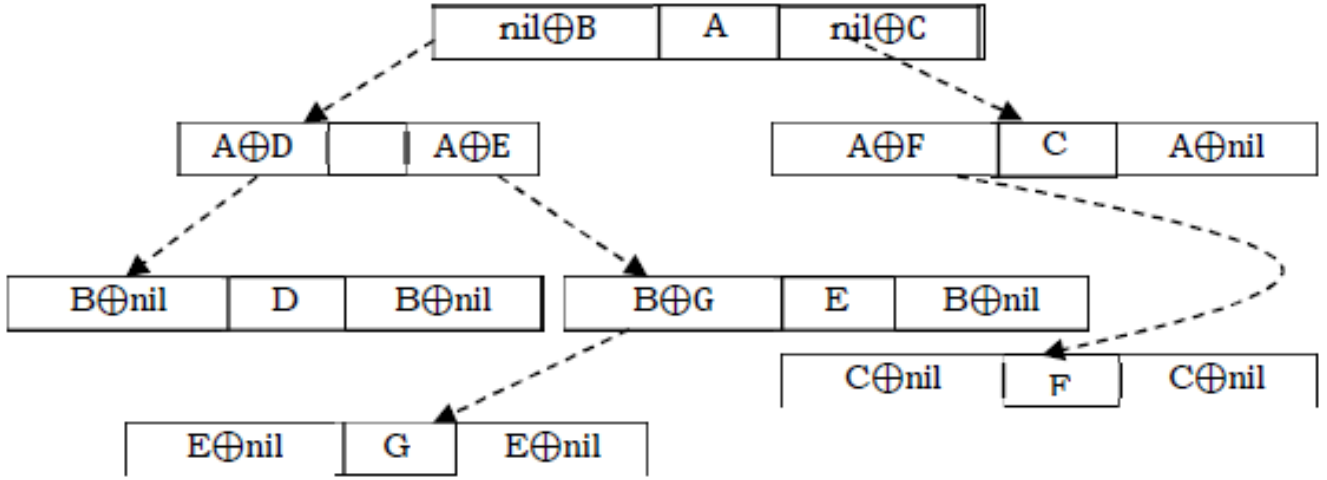
XOR Trees

- This concept is similar to *memory efficient doubly linked lists* of *Linked Lists* chapter. Also, like threaded binary trees this representation does not need stacks or queues for traversing the trees. This representation is used for traversing back (to parent) and forth (to children) using $\oplus$ operation. To represent the same in XOR trees, for each node below are the rules used for representation:
  - Each nodes left will have the $\oplus$ of its parent and its left children.
  - Each nodes right will have the $\oplus$ of its parent and its right children.
  - The root nodes parent is nil and also leaf nodes children are nil nodes.

XOR Trees



- Based on the above rules and discussion, the tree can be represented as:



XOR Trees

- The major objective of this presentation is the ability to move to parent as well to children. Now, let us see how to use this representation for traversing the tree. For example, if we are at node B and want to move to its parent node A, then we just need to perform $\oplus$ on its left content with its left child address (we can use right child also for going to parent node).
- Similarly, if we want to move to its child (say, left child D) then we have to perform $\oplus$ on its left content with its parent node address. One important point that we need to understand about this representation is: When we are at node B, how do we know the address of its children D? Since the traversal starts at node root node, we can apply $\oplus$ on root's left content with nil.
- As a result we get its left child, B. When we are at B, we can apply $\oplus$ on its left content with A address.

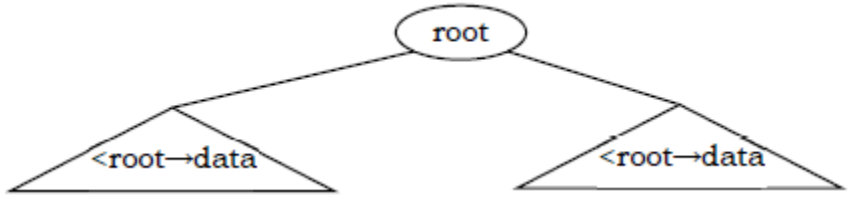
Binary Search Trees (BSTs)

- Why Binary Search Trees?
- In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data.
- As a result, to search for an element we need to check both in left subtree and in right subtree. Due to this, the worst-case complexity of search operation is O(n).
- In this section, we will discuss another variant of binary trees: Binary Search Trees (BSTs).
- As the name suggests, the main use of this representation is for $searching$. In this representation we impose restriction on the kind of data a node can contain.
- As a result, it reduces the worst-case average search operation to O($logn$).

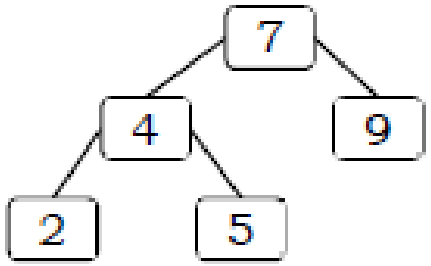
Binary Search Tree Property

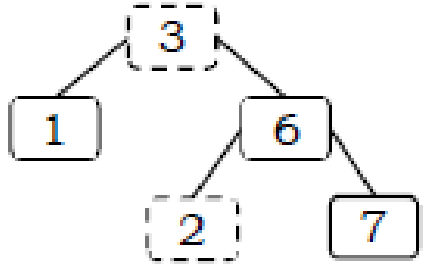
- In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.
  - The left subtree of a node contains only nodes with keys less than the nodes key.
  - The right subtree of a node contains only nodes with keys greater than the nodes key.
  - Both the left and right subtrees must also be binary search trees.

Binary Search Tree Property



- **Example**: The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it's not satisfying the binary search tree property).





Binary Search Tree Declaration

- There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

```
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data
```

Operations on Binary Search Trees

- Main operations: Following are the main operations that are supported by binary search trees:
  - Find/ Find Minimum / Find Maximum element in binary search trees
  - Inserting an element in binary search trees
  - Deleting an element from binary search trees

- Auxiliary operations: Checking whether the given tree is a binary search tree or not:
  - Finding k^{th} -smallest element in tree
  - Sorting the elements of binary search tree and many more

Important Notes on Binary Search Trees

- Since root data is always between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.

Important Notes on Binary Search Trees

- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it.
- The same is the case with the right subtree.. Because of this, binary search trees take less time for searching an element than a regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.

Important Notes on Binary Search Trees

- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.

- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(logn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(n) worst-case time.

Finding an Element in Binary Search Trees

- Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is same as nodes data then we return current node.

- If the data we are searching is less than nodes data then search left subtree of current node; otherwise search right subtree of current node. If the data is not present, we end up in a NULL link.

Finding an Element in Binary Search Trees

- Time Complexity: O(n), in worst case (when BST is a skew tree). Space Complexity: O(n), for recursive stack. *Non recursive* version of the above algorithm can be given as:

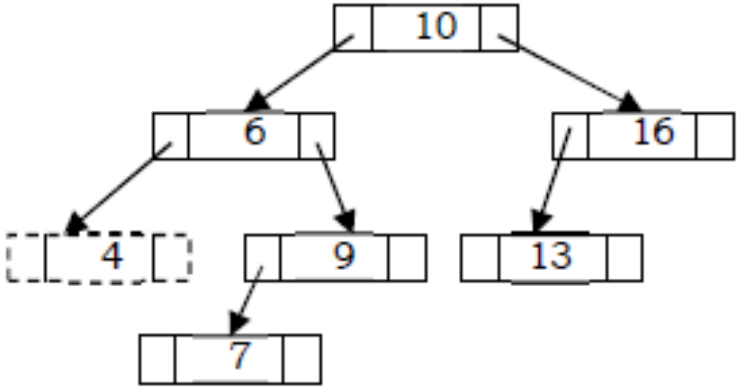
```
def Find(self, root, data):
  currentNode = root
  while currentNode:
    if data == currentNode.get_data():
        return currentNode
    if key < currentNode.get_data():
        currentNode = currentNode.getLeft()
    else:
        currentNode = currentNode.getRight()
  return None
```

- Time Complexity: O(n). Space Complexity: O(1).

Finding Minimum Element in Binary Search Trees

- In BSTs, the minimum element is the left-most node, which does not has left child. In the BST below, the minimum element is **4**.

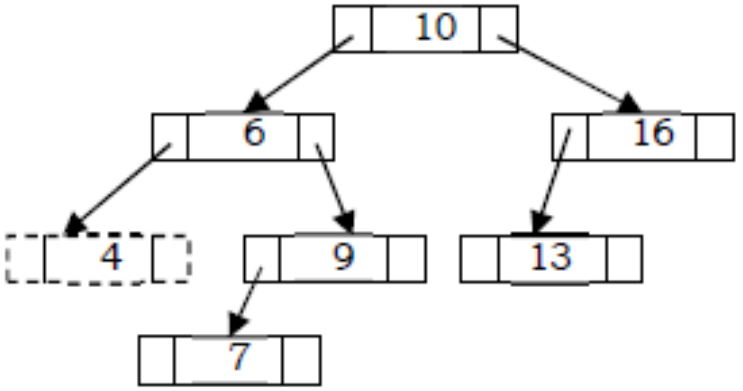
```
def findMin(root):
  currentNode = root
  if currentNode == None:
    return None
  while currentNode.getLeft() != None:
    currentNode = currentNode.getLeft()
  return currentNode
```



Finding Maximum Element in BST

- In BSTs, the maximum element is the right-most node, which does not have right child. In the BST below, the maximum element is **16**.

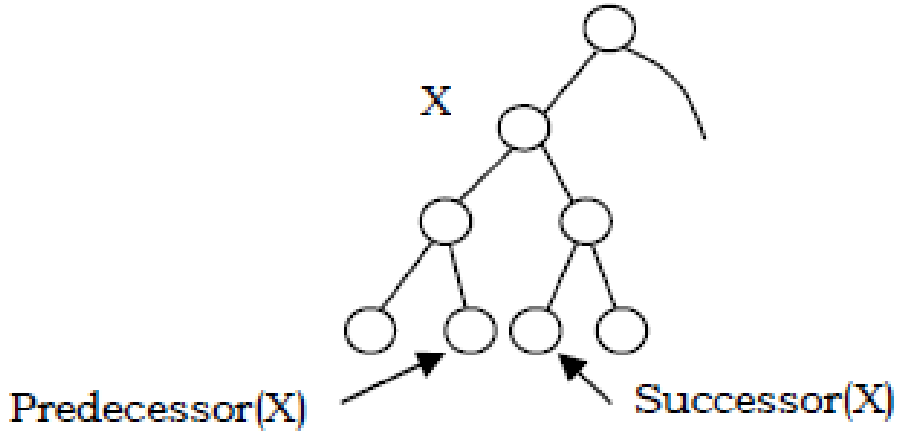
```
# Search the key from node, iteratively
def findMax(root):
  currentNode = root
  if currentNode == None:
    return None
  while currentNode.right != None:
    currentNode = currentNode.right
  return currentNode
```



- Time Complexity: O(n), in worst case (when BST is a *right skew* tree). Space Complexity: O(n), for recursive stack.

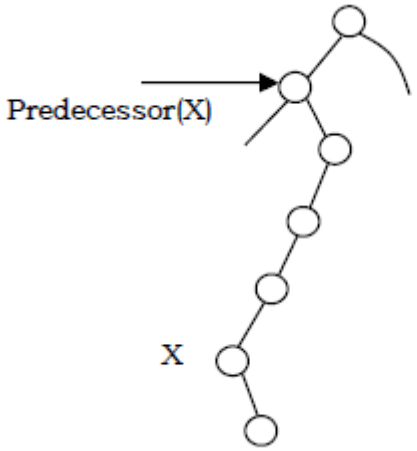
Where is Inorder Predecessor and Successor?

- Where is Inorder Predecessor and Successor? Where is the inorder predecessor and successor of node *X* in a binary search tree assuming all keys are distinct?
- If *X* has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.



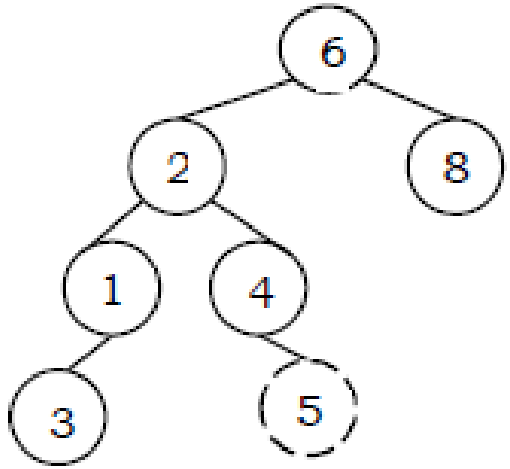
Where is Inorder Predecessor and Successor?

- If it does not have a left child, then a node's inorder predecessor is its first left ancestor.



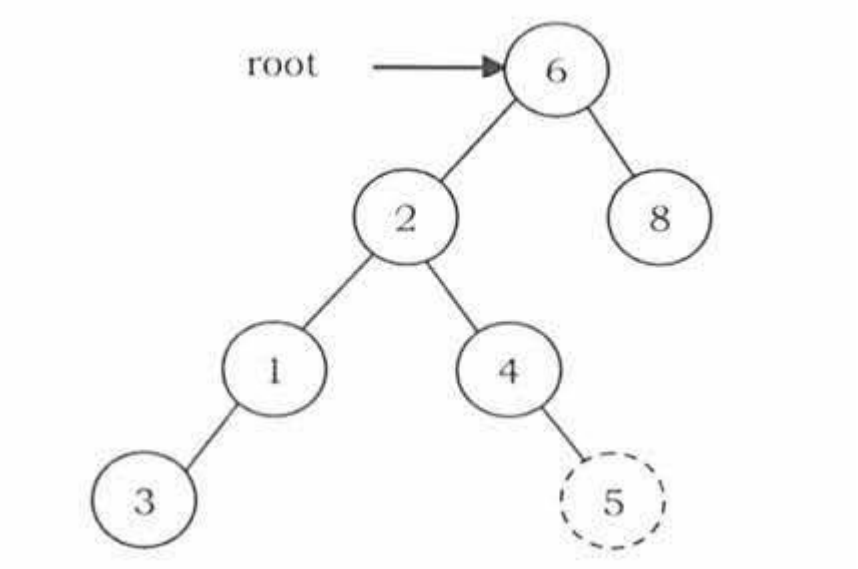
Inserting an Element from Binary Search Tree

- To insert $data$ into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of $find$ operation. While finding the location, if the $data$ is already there then we can simply neglect and come out. Otherwise, insert $data$ at the last location on the path traversed.



Inserting an Element from Binary Search Tree

- As an example let us consider the following tree. The dotted node indicates the element (5) to be inserted. To insert 5, traverse the tree using $find$ function. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.



Inserting node code

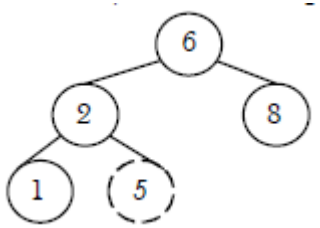
```
def insertNode(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.left == None:
                root.left = node
            else:
                insertNode(root.left, node)
        else:
            if root.right == None:
                root.right = node
            else:
                insertNode(root.right, node)
```

- Note: In the above code, after inserting an element in subtrees, the tree is returned to its parent. As a result, the complete tree will get updated.
- Time Complexity: O(n). Space Complexity: O(n), for recursive stack. For iterative version, space complexity is O(1).

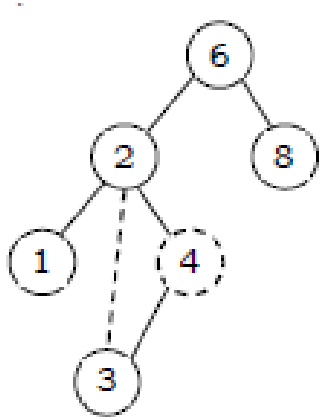
Deleting an Element from Binary Search Tree

- The delete operation is more complicated than other operations. This is because the element to be deleted may not be the leaf node. In this operation also, first we need to find the location of the element which we want to delete.
- Once we have found the node to be deleted, consider the following cases:
- If the element to be deleted is a leaf node: return nil to its parent. That means make the corresponding child pointer nil. In the tree below to delete 5, set nil to its parent node 2.

Deleting node



- If the element to be deleted has one child: In this case we just need to send the current node's child to its parent. In the tree below, to delete 4, 4 left subtree is set to its parent node 2.

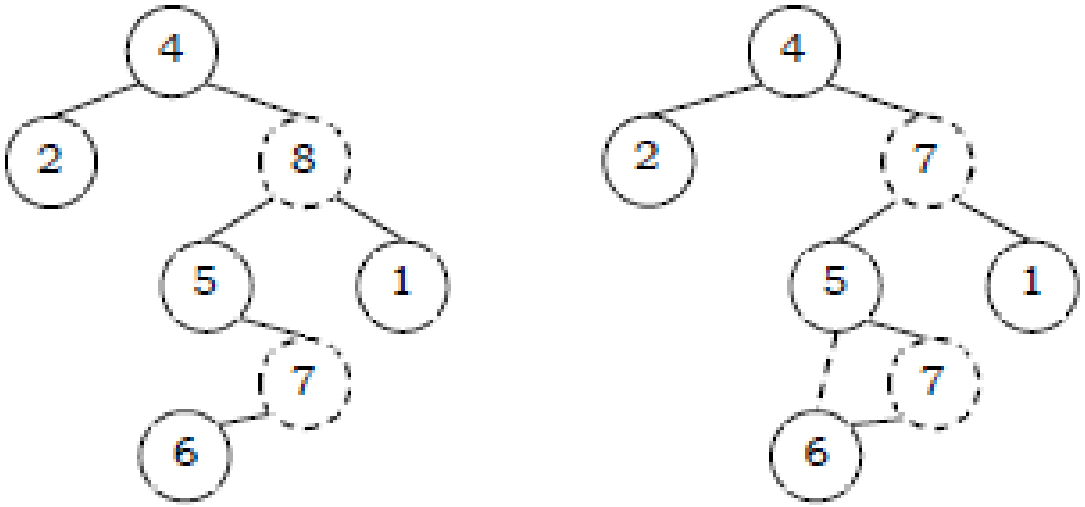


Deleting node

- If the element to be deleted has both children: The general strategy is to replace the key of this node with the largest element of the left subtree and recursively delete that node (which is now empty). The largest node in the left subtree cannot have a right child, so the second $delete$ is an easy one.

Deleting node

- As an example, let us consider the following tree. In the tree below, to delete 8, it is the right child of the root. The key value is 8. It is replaced with the largest key in its left subtree (7), and then that node is deleted as before (second case).

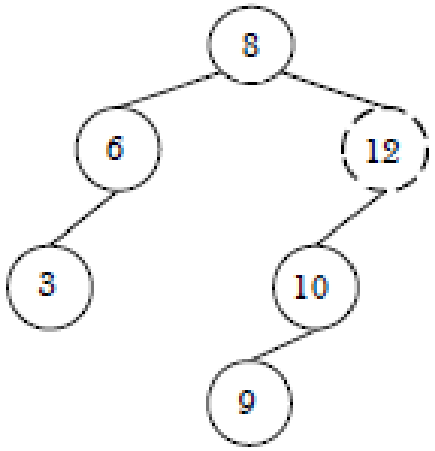


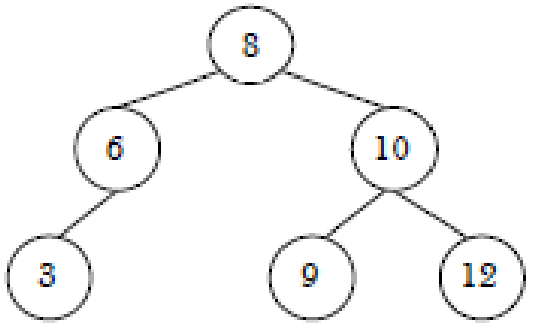
AVL (Adelson-Velskii and Landis) Trees

- In $HB(k)$, if $k = 1$ (if balance factor is one), such a binary search tree is called an $AVL\ tree$. That means an AVL tree is a binary search tree with a $balance$ condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

- A binary tree is said to be an AVL tree, if:
  - It is a binary search tree, and
  - For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.





AVL Trees

- As an example, among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.
-

Minimum/Maximum Number of Nodes

- For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get the minimum number of nodes with height h, we should fill the tree with the minimum number of nodes possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

$$\mathrm{N}(h) = N(h-1) + N(h-2) + 1$$

- In the above equation:
  - $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
  - $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
  - In the above expression, "1" indicates the current node.

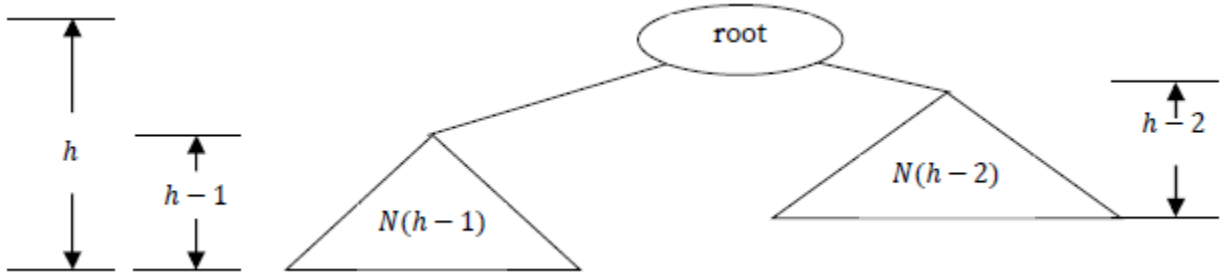
- We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = \mathrm{O}(1.618^{\mathrm{h}}) \Longrightarrow h = 1.44\ logn \approx \mathrm{O}(logn)$$

- The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = \mathrm{O}(2^{\mathrm{h}}) \Longrightarrow h = logn \approx \mathrm{O}(logn)$$

- In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is O($logn$).



AVL Tree Declaration

- Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we also include the height as part of the declaration.

```
class AVLNode:
    def __init__(self, data, balanceFactor, left, right):
        self.data = data
        self.balanceFactor = 0
        self.left = left
        self.right = right
```

Finding the Height of an AVL tree

```
def height(self):
    return self.recHeight(self.root)

def recHeight(self, root):
    if root == None:
        return 0
    else:
        leftH = self.recHeight(root.left)
        rightH = self.recHeight(root.right)
        if leftH > rightH:
            return 1 + leftH
        else:
            return 1 + rightH
```

Rotations

- When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a subtree by 1.
- So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2.
- This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node X.

Rotations

- **Observation**: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered, because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to the root of the tree.

- While moving to the root, we need to consider the first node that is not satisfying the AVL property. From that node onwards, every node on the path to the root will have the issue.

- Also, if we fix the issue for that first node, then all other nodes on the path to the root will automatically satisfy the AVL tree property. That means we always need to care for the first node that is not satisfying the AVL property on the path from the insertion point to the root and fix it.

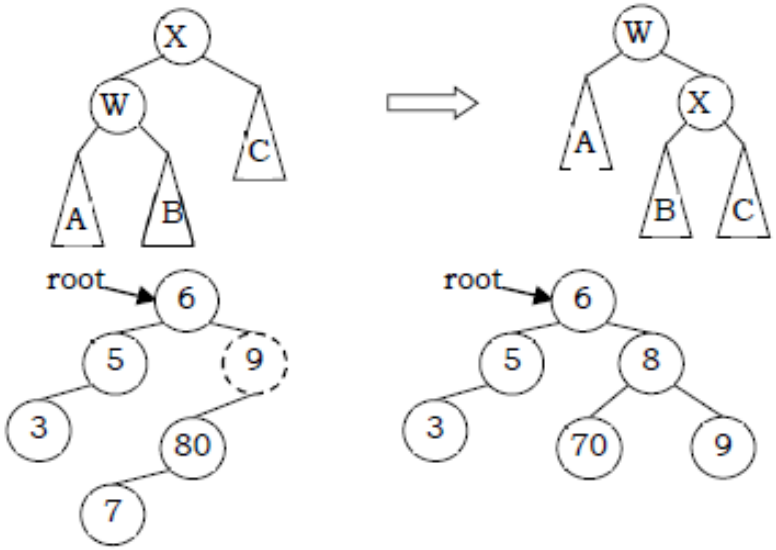
Types of Violations

- The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required. Let us assume the node that must be rebalanced is *X*. Since any node has at most two children, and a height imbalance requires that *X's* two subtree heights differ by two, we can observe that a violation might occur in four cases:
    1. An insertion into the left subtree of the left child of *X*.
    2. An insertion into the right subtree of the left child of *X*.
    3. An insertion into the left subtree of the right child of *X*.
    4. An insertion into the right subtree of the right child of *X*.

Single Rotations (case 1 & 4)

- Left Left Rotation (LL Rotation) **[Case-1]**:
- In the case below, node *X* is not satisfying the AVL tree property. As discussed earlier, the rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

Single Rotations



- For example, in the figure above, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.
- Time Complexity: O(1). Space Complexity: O(1).

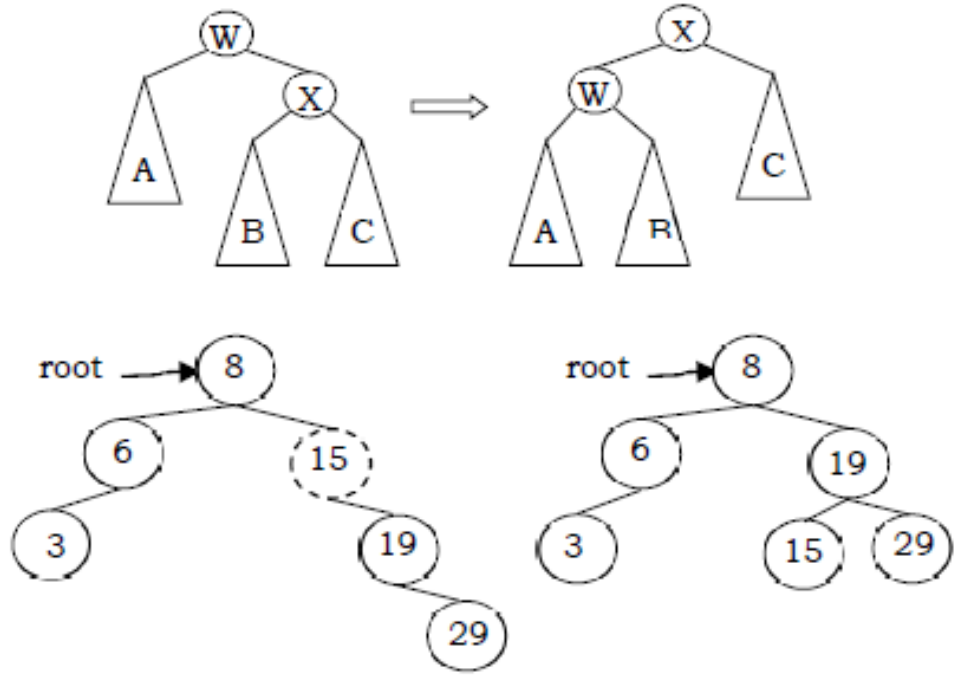
Code

```python
def rightLeftRotate(self, root):
        X = root.right
        if X.balanceFactor == 1:
            root.balanceFactor = 0
            X.balanceFactor = 0
            root = self.singleRightRoate(r)
        else:
            Y = X.left
            if Y.balanceFactor == -1:
                root.balanceFactor = 0
                X.balanceFactor = 1
            elif Y.balanceFactor == 0:
                root.balanceFactor = 0
                X.balanceFactor = 0
            else:
                root.balanceFactor = -1
                X.balanceFactor = 0
            Y.balanceFactor = 0
            root.right = self.singleLeftRotate(X)
            root = self.singleRightRoate(root)
        return root
```

```python
def rightLeftRotate(self, root):
        X = root.left
        if X.balanceFactor == -1:
            root.balanceFactor = 0
            X.balanceFactor = 0
            root = self.singleLeftRotate(root)
        else:
            Y = X.right
            if Y.balanceFactor == -1:
                root.balanceFactor = 1
                X.balanceFactor = 0
            elif Y.balanceFactor == 0:
                root.balanceFactor = 0
                X.balanceFactor = 0
            else:
                root.balanceFactor = 0
                X.balanceFactor = -1
            Y.balanceFactor = 0
            root.left = self.singleRightRoate(X)
            root = self.singleLeftRotate(root)
        return root
```

Right Right Rotation (RR Rotation) [Case-4]

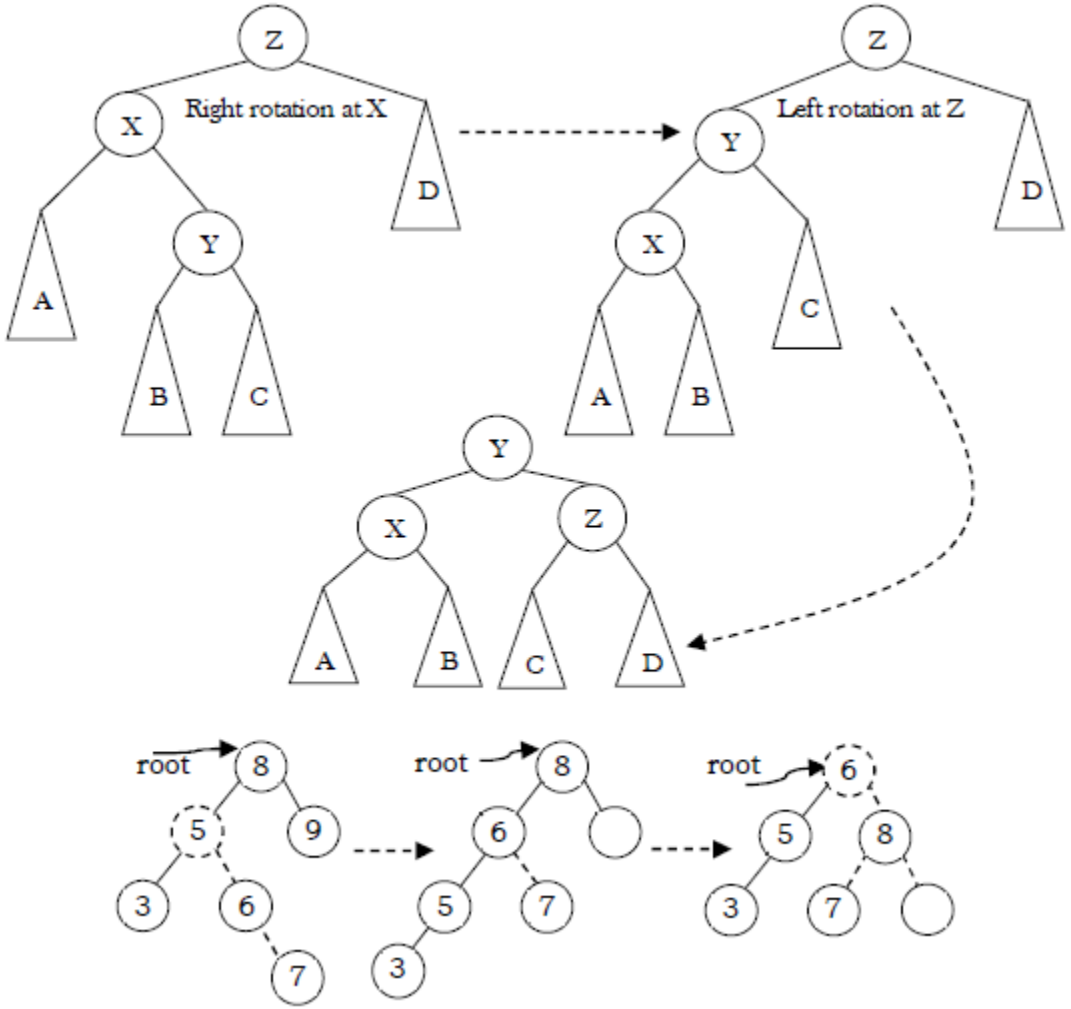
- Right Right Rotation (RR Rotation) [Case-4]: In this case, node *X* is not satisfying the AVL tree property.



Double Rotations (case 2 & 3)

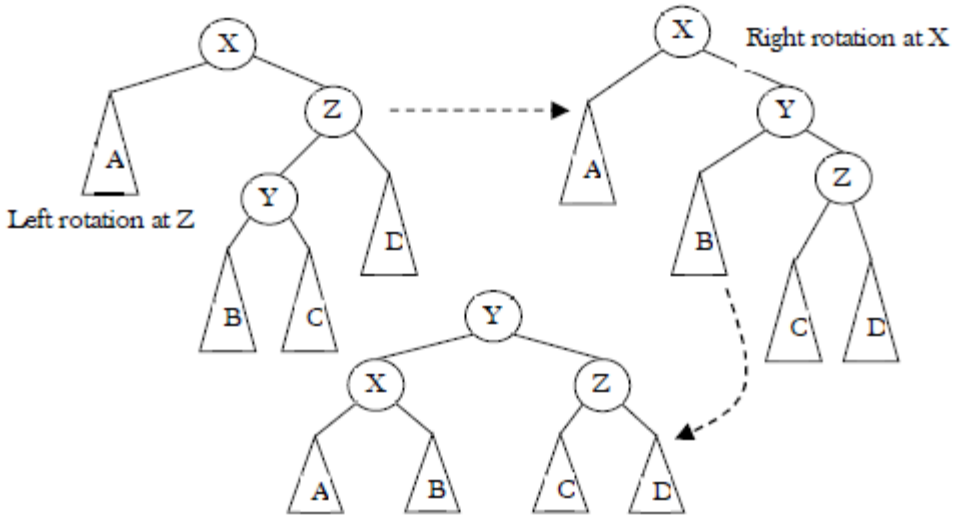
- Left Right Rotation (LR Rotation) **[Case-2]:** For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations. As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right-side tree is the one after double rotation.

Double Rotations

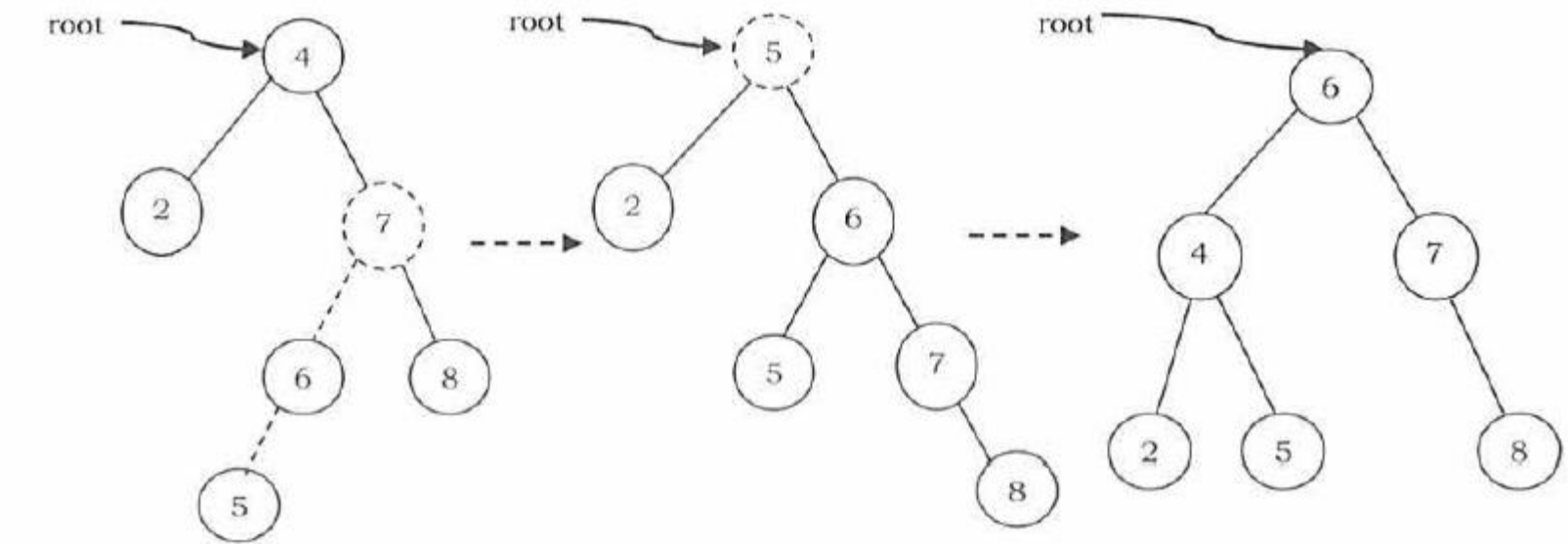


Right Left Rotation case 3

- [Case-3]: Similar to case-2, we need to perform two rotations to fix this scenario.



- As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right-side tree is the one after the double rotation.



Insertion an Element into an AVL tree

- Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. After inserting the element, we just need to check whether there is any height imbalance. If there is an imbalance, call the appropriate rotation functions.
- The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.
- The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required. As discussed in the previous section, depending upon the type of insertion, the rotations are categorized into four categories.

Deleting an Element from AVL tree

- Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations.

Single Rotations

- **Left Left Rotation (LL Rotation) [Case-1]:** In the case below, node *X* is not satisfying the AVL tree property. As discussed earlier, the rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

In Lab session

- You will play with the concepts and starts getting more and more familiar with trees
- This can be useful for your project

- Lab is done by Remy Belmonte