Data Structures and Algorithms

Lecture 06 – Lowest Common Ancestor (LCA)

(Basics of Trees, BST, AVL Tree)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

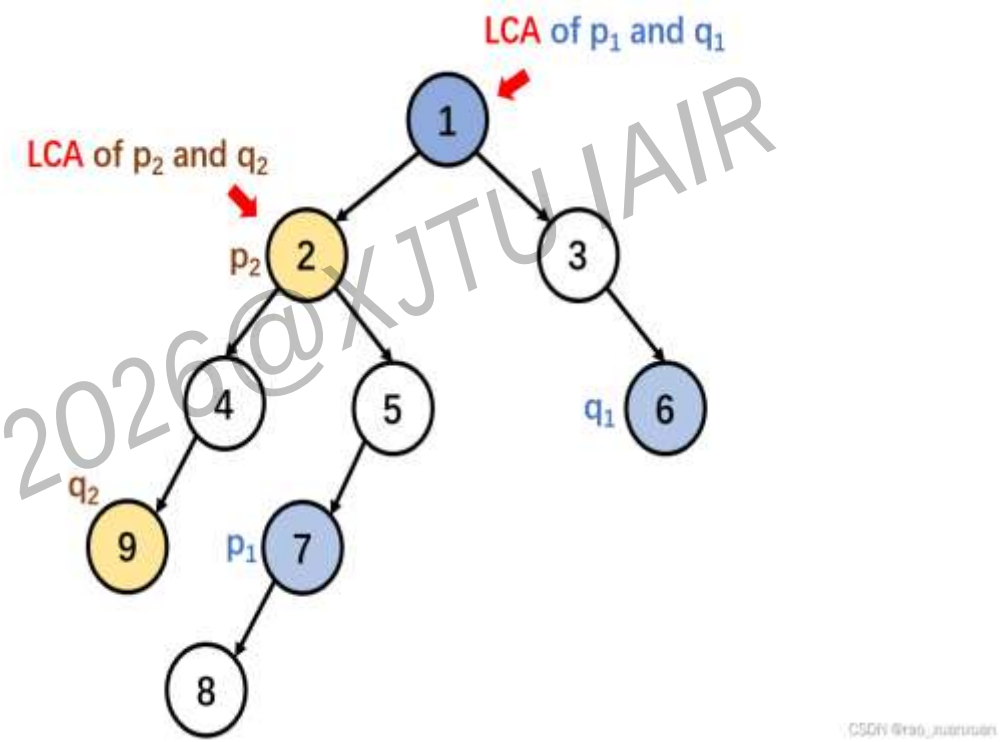
http://gr.xjtu.edu.cn/web/pengjuren

Problem Solving

Cross-departmental collaboration to find the *lowest common ancestor* (LCA) of two nodes in a tree

Solution1：LCA Naïve Implementation

```
def lca_naive(u, v):
    # 先将深度对齐
    while u.depth > v.depth:
        u = u.parent
    while v.depth > u.depth:
        v = v.parent
    # 同时向上移动直到相遇
    while u != v:
        u = u.parent
        v = v.parent
    return u
```



Time Complexity: Each (u,v) pair search LCA is O(N)

Solution2：LCA Binary Lifting

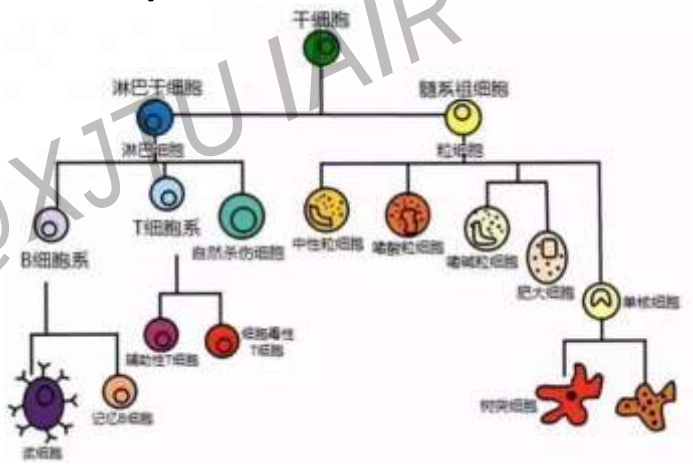
```
# Pre-processing run DFS to set self.up[u][k] k=1 ……⌈log2 n⌉ for every node
def lca_Binarylifting(self, u, v):
        if self.depth[u] < self.depth[v]:
            u, v = v, u
        # 将 u 向上跳到与 v 同深度
        diff = self.depth[u] - self.depth[v]
        for k in range(self.max_log):
            if diff >> k & 1:
                u = self.up[u][k]
        if u == v:
            return u
        # 一起向上跳
        for k in range(self.max_log-1,-1,-1):
            if self.up[u][k] != self.up[v][k]:
                u = self.up[u][k]
                v = self.up[v][k]
        return self.up[u][0]
```


Time Complexity: each (u,v) pair search LCA is O(log N)

Space Complexity: O(Nlog N) for *self.up table*

What is a Tree?

A tree is an important **non-linear data structure** composed of nodes and edges connecting them, used to represent data sets with **hierarchical relationships**.



Formal Definition:

A tree is a finite set T of n ($n \geq 0$) nodes, satisfying:

- When $n = 0$, it is called an *empty tree*.
- When $n > 0$, there exists exactly one node called the ***root***. The remaining nodes can be partitioned into m ($m \geq 0$) disjoint finite sets $T_1, T_2, \ldots, T_m$,each of which is itself a tree, called a ***subtree*** of the root.

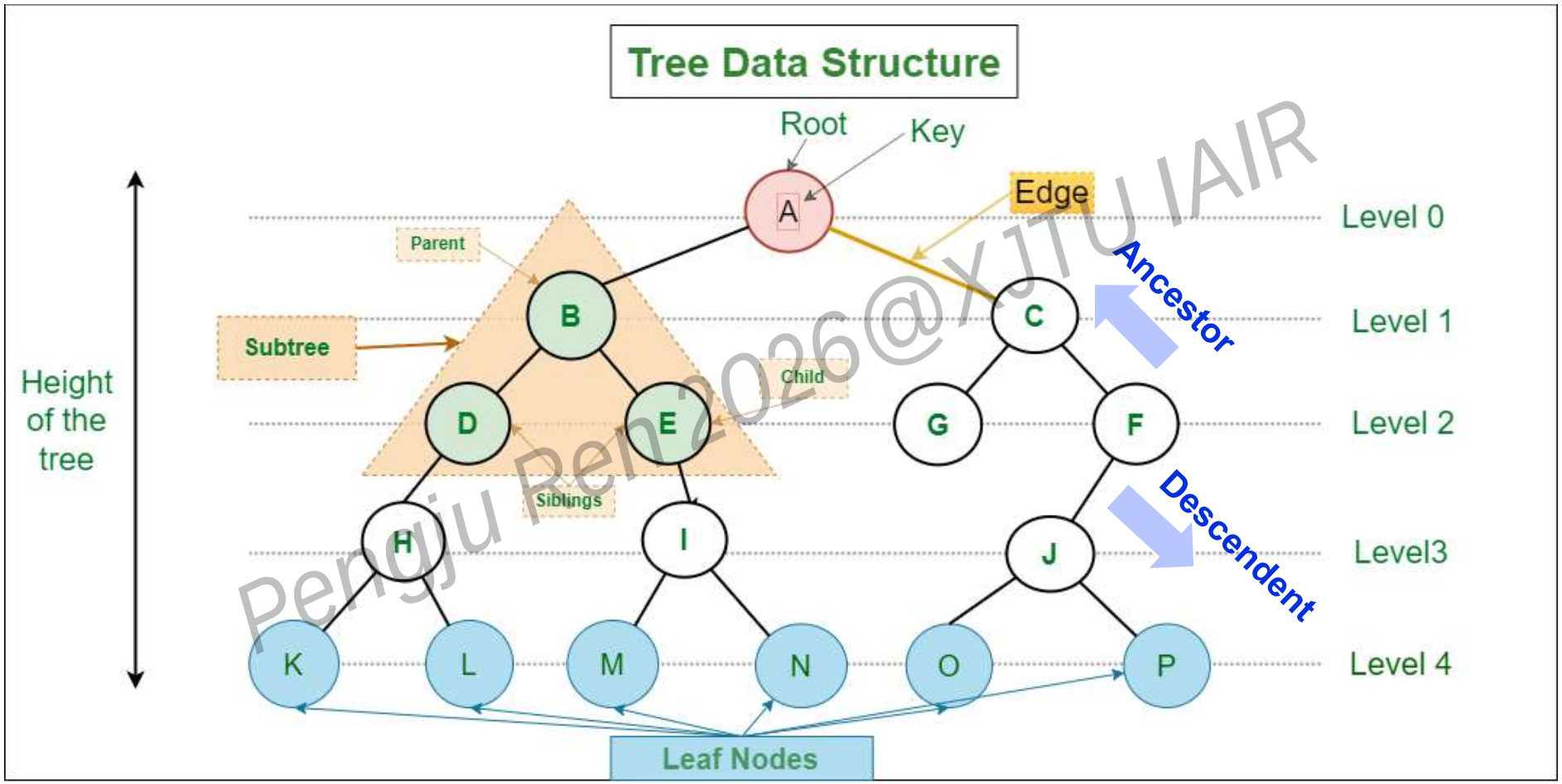
Terminology of Trees

- **Node**: The basic unit of a tree, containing a *data* element and references (*pointers*) to its subtrees.
- **Root**: The only node in a tree that has no parent.
- **Parent**: If node A has a child node B, then A is the parent of B.
- **Child**: Node B is a child of node A iff A is the parent of B.
- **Sibling**: Nodes that share the same parent are called siblings.
- **Leaf**: A node with no children (degree 0).
- **Ancestor**: All nodes on the path from the root to that node (including the node itself).
- **Descendant**: All nodes in the subtree of that node

Terminology of Trees

- **Degree of a Node**: The number of subtrees a node has.
- **Degree of a Tree**: The maximum degree among all nodes in the tree.
- **Level**: Defined starting from the root; the root is at level 1 (some textbooks start from 0), its children at level 2, and so on.
- **Depth**: The length of the path *from the root to that node* (commonly defined with root depth 0).
- **Height**: The length of the path from that *node to the farthest leaf node* (leaf height is 0).
- **Forest**: A set of m ($m \geq 0$) disjoint trees.

Tree Data Structure



Basic Classification of Trees

■ **By Number of Child Nodes**

- **Binary Tree**: Each node has *at most* two subtrees, usually referred to as the left subtree and right subtree. Binary trees are the most commonly used tree structure with rich theory and applications.
- **Multiway Tree**: Nodes can have more than two children, e.g., ternary trees, quadtrees, etc. File systems are typically multiway trees.

■ **By Weighted Nodes/Edges**

- **Weighted Tree**: Nodes or edges carry weights, e.g., Huffman trees (optimal binary trees) used in data compression.

Basic Classification of Trees

■ By Tree Shape/Properties

- **Full Binary Tree**: A binary tree in which every node either is a leaf or has two children, and all leaves are on the same level.
- **Complete Binary Tree**: A binary tree where all levels except possibly the last are completely filled, and all nodes in the last level are as far left as possible. Complete binary trees can be efficiently stored using arrays.
- **Balanced Binary Tree**: The height difference between the left and right subtrees of any node is at most 1 (e.g., *AVL trees*), or it satisfies some other balancing condition (e.g., ***red-black trees***).
- **Binary Search Tree (BST)**: A binary tree where the value of every node in the *left subtree is less than the node's value*, and the value of every node in the *right subtree is greater*. Both left and right subtrees are also BSTs.
- **Heap**: A special kind of complete binary tree, categorized into *max-heaps* (parent value ≥ children values) and *min-heaps* (parent value ≤ children values).

ADT(Abstract Data Structure) of a Tree

Data Objects

- A tree consists of a set of nodes and a set of edges. Each node contains a data element (which can be of any type) and references (pointers) to its child nodes.

Basic Operations

- **Initialize**: Create an empty tree.
- **IsEmpty**: Check if the tree is empty.
- **Get Root**: Return the root node of the tree.
- **Get Parent**: Given a node, return its parent node (if it exists).
- **Get Children**: Given a node, return all its child nodes (or children in a specified order).
- **Insert Subtree**: Insert a new subtree (as a child) under a specified node.
- **Delete Subtree**: Delete the specified node and its entire subtree.

ADT(Abstract Data Structure) of a Tree

Basic Operations

- **Traverse**: Visit all nodes in the tree in a specific order. Common traversal methods include:
  - *Preorder Traversal* (Root → Left Subtree → Right Subtree)
  - *Inorder Traversal* (Left Subtree → Root → Right Subtree) — only for binary trees
  - *Postorder Traversal* (Left Subtree → Right Subtree → Root)
  - *Level Order Traversal* (top to bottom, left to right)
- **Search**: Find a node satisfying a specific condition.
- **Get Depth/Height**: Calculate the depth or height of the tree or a specific node.
- **Get Node Count**: Return the total number of nodes in the tree.

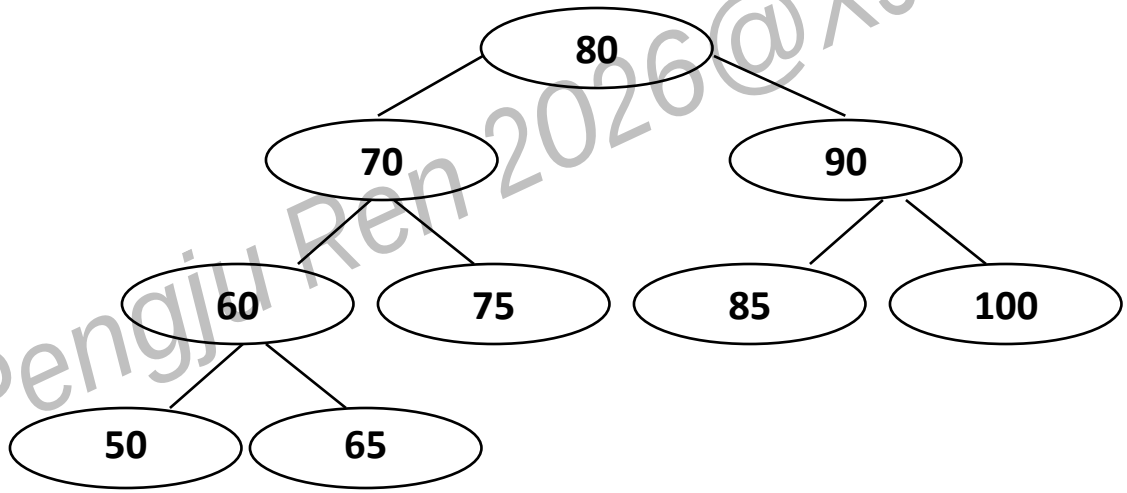
Implementation of the tree ADT

- **Linked Storage**: Each node contains a data field and several pointer fields (pointing to child nodes). Suitable for *general trees.*
- **Array Storage**: For *complete binary trees*, a contiguous array can be used, with parent-child relationships calculated via indices.
- **Adjacency List**: A list or hash table stores the list of children for each node. Suitable for *multiway trees*.

What is a Binary Search Tree?

In the left subtree the value of a node is *less or equal* to its parent node's value

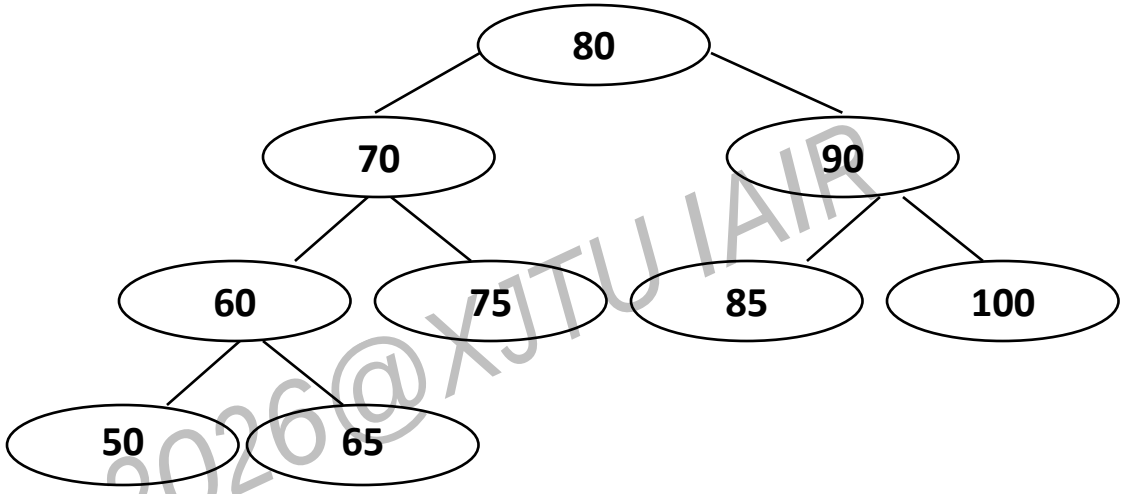
In the right subtree the value of a node is *greater than* its parent node's value

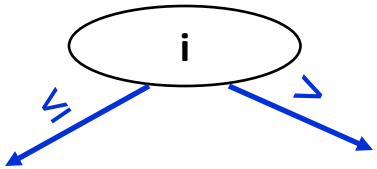


It performs faster than Binary Tree when *searching* , *inserting* and *deleting* nodes

Common Ops on Binary Search Tree

- Creation of Tree
- Traverse all nodes
- Insertion of a node
- Search for a value
- Deletion of a node
- Deletion of tree





What is time complexity of Insertion/Search in BST?

$T(n) = T(n/2) + O(1) \Rightarrow O(\log n)$

Common Ops on Binary Search Tree

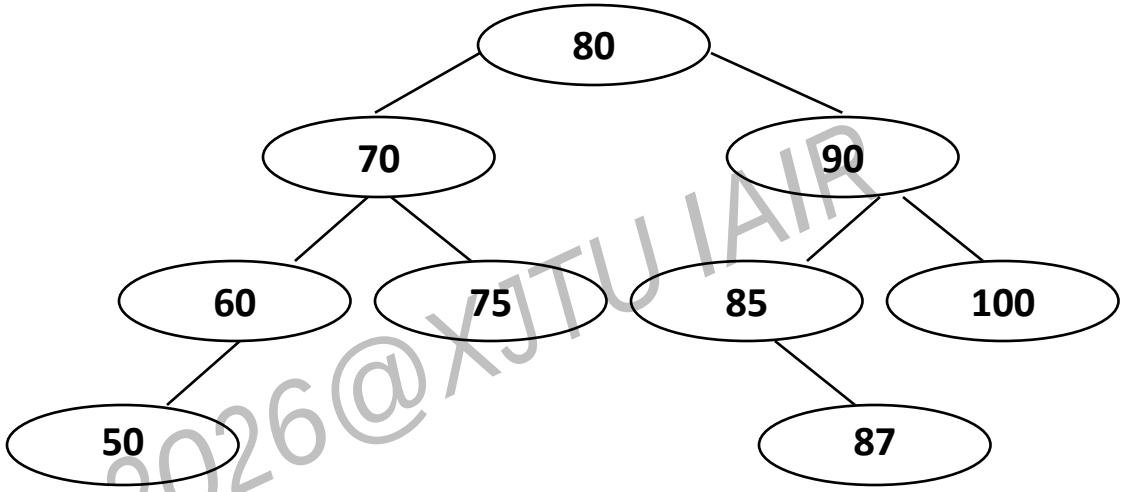
Creation of Tree

Traverse all nodes

Insertion of a node

Search for a value

Deletion of a node



C1 : node to be deleted is a leaf node (node 50、75、87、100)

C2: node to be deleted has one child (node 60)

C3: node to be deleted has two children (node 70, 80, 90)

Deletion of tree

What is time complexity of Insertion/Search in BST?

T(n)= T(n/2)+O(1) ➧ $O(\log n)$

Time and Space complexity of BST

	Time Complexity	Space Complexity
Create Binary Search Tree	$O(1)$	$O(1)$
Insert a node to BST	$\boldsymbol{O(\log n)}/O(n)$	$\boldsymbol{O(\log n)}$
Delete a node from BST	$\boldsymbol{O(\log n)}/O(n)$	$\boldsymbol{O(\log n)}$
Search for a node in BST	$\boldsymbol{O(\log n)}/O(n)$	$\boldsymbol{O(\log n)}$
Traverse BST	$O(n)$	$O(n)$
Delete entire BST	$O(1)$	$O(1)$

PreOrder Traversal of BT

Depth first search

- Preorder traversal

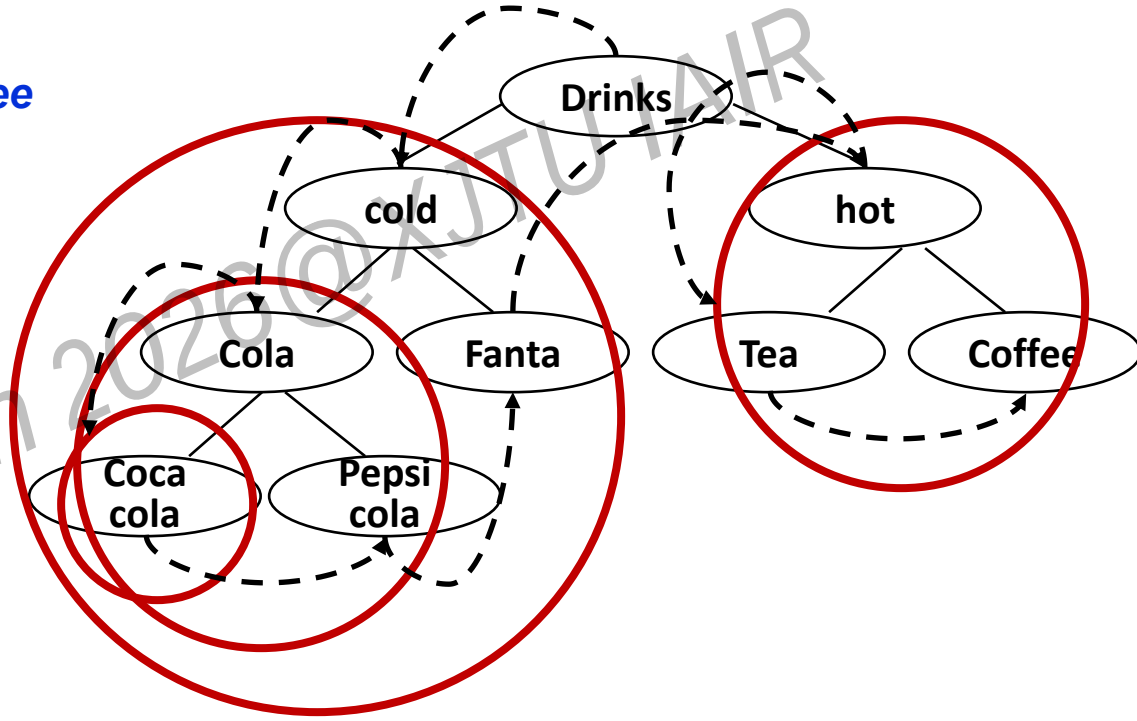
root->left subtree->right subtree

- Inorder traversal

- Post order traversal

Breadth first search

- Level order traversal




```
def preOrderTraversal(rootNode):
    if not rootNode:
        return
    print(rootNode.data)
    preOrderTraversal(rootNode.leftChild)
    preOrderTraversal(rootNode.rightChild)
```

InOrder Traversal of BT

Depth first search

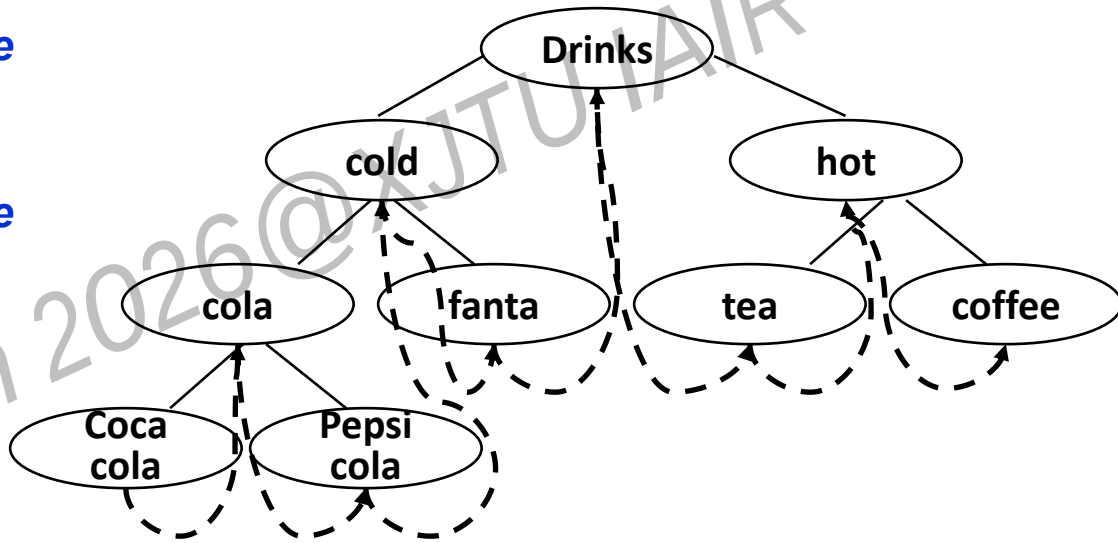
- Preorder traversal

root->left subtree->right subtree

- Inorder traversal

left subtree->root->right subtree

- Post order traversal




```
def inOrderTraversal(rootNode):
    if not rootNode:
        return
    inOrderTraversal(rootNode.leftChild)
    print(rootNode.data)
    inOrderTraversal(rootNode.rightChild)
```

PostOrder Traversal of BT

Depth first search

- Preorder traversal

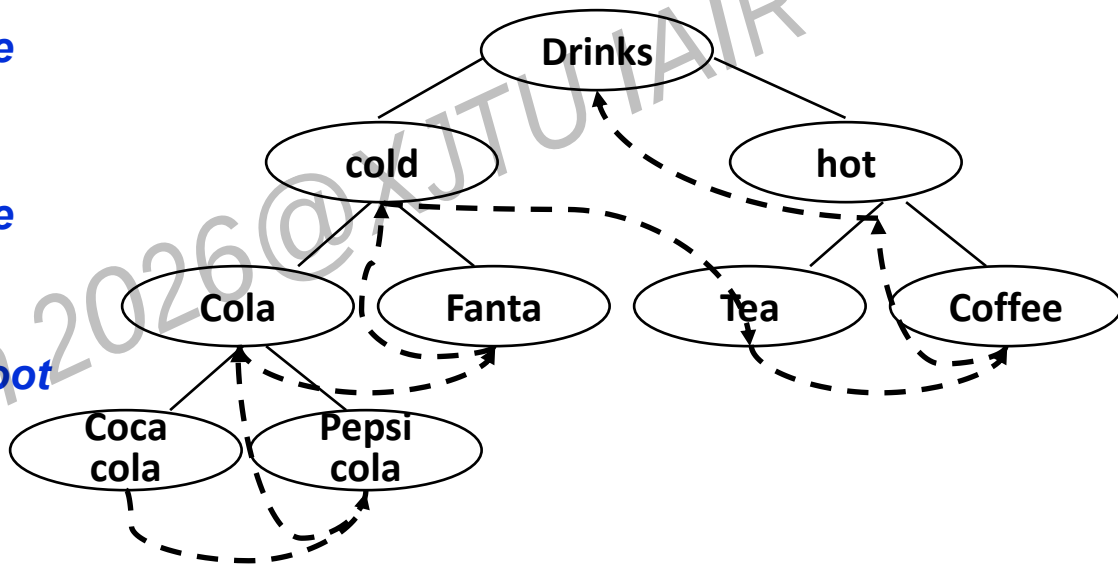
root->left subtree->right subtree

- Inorder traversal

left subtree->root->right subtree

- Post order traversal

left subtree-> right subtree -> root




```
def postOrderTraversal(rootNode):
    if not rootNode:
        return
    postOrderTraversal(rootNode.leftChild)
    postOrderTraversal(rootNode.rightChild)
    print(rootNode.data)
```

LevelOrder Traversal of BT

Depth first search

- Preorder traversal

root->left subtree->right subtree

- Inorder traversal

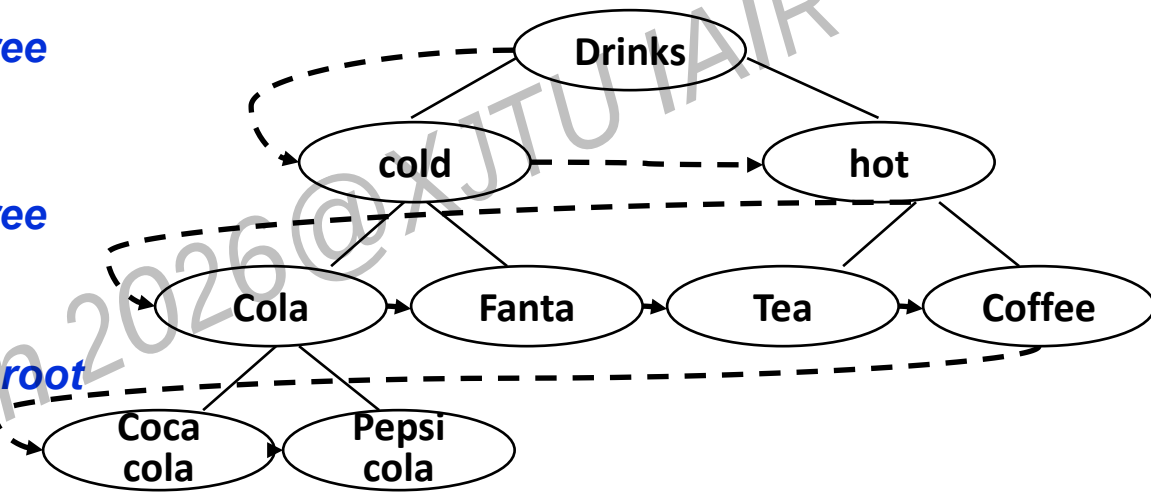
left subtree->root->right subtree

- Post order traversal

left subtree-> right subtree -> root

Breath first search

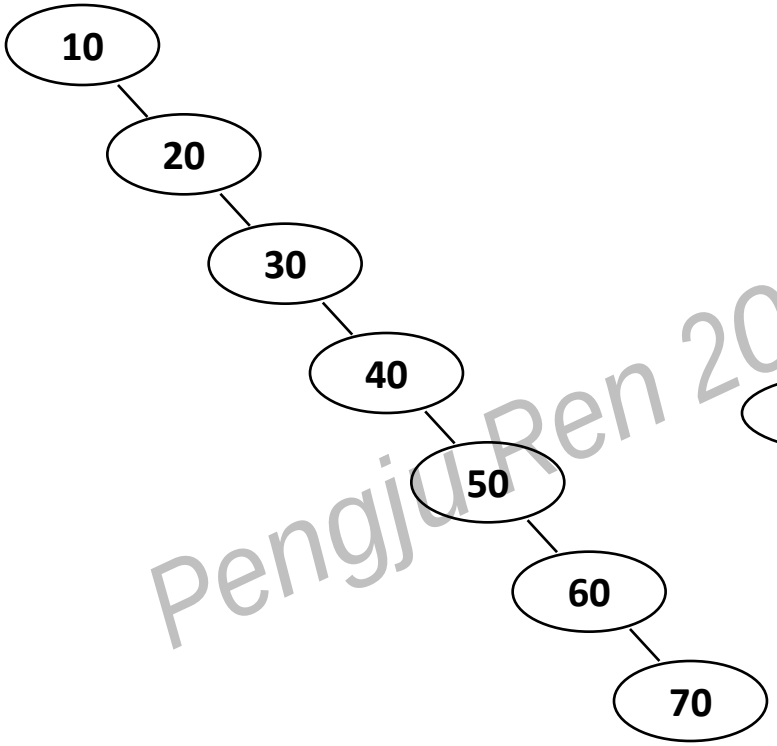
(Level Ordered Traversal)

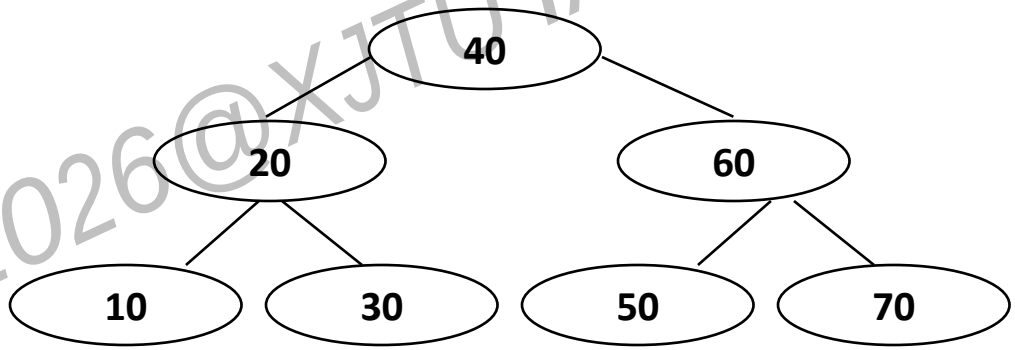



```
def levelOrderTraversal(rootNode):
    if not rootNode:
        return
    else:
        customQueue = queue.Queue()
        customQueue.enqueue(rootNode)
        while not(customQueue.isEmpty()):
            root = customQueue.dequeue()
            print(root.value.data)
            if root.value.leftChild is not None:
               customQueue.enqueue(root.value.leftChild)
            if root.value.rightChild is not None:
               customQueue.enqueue(root.value.rightChild)
```

The motivation of AVL Tree

Insert 10, 20, 30, 40, 50, 60, 70





What is an AVL tree?

An AVL tree is a *self-balancing* Binary Search Tree(BST) where the difference between heights of left and right subtrees *cannot be more than 1 for all nodes*

Node : 80

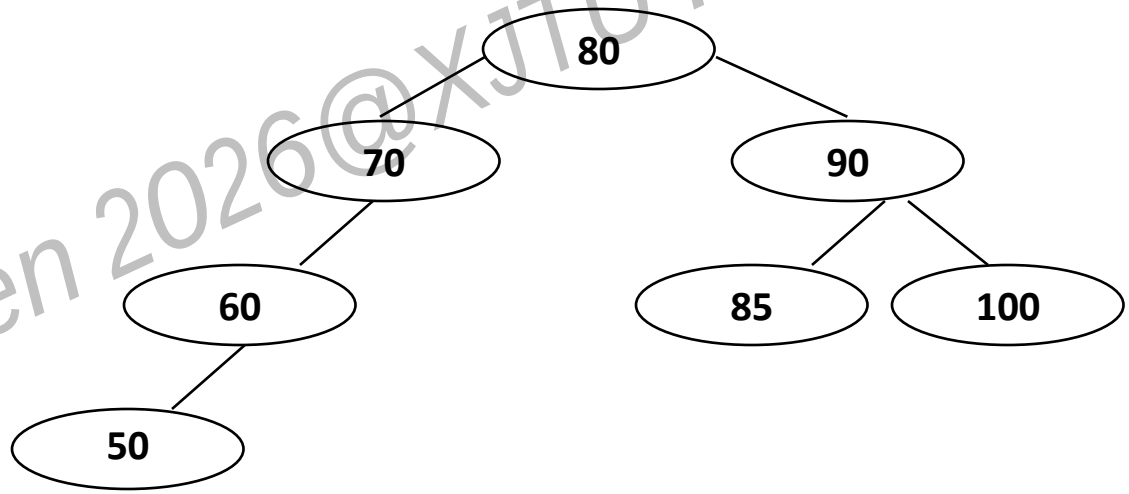
Height of left Subtree = 3

Height of right Subtree = 2

Node : 70

Height of left Subtree = 2

Height of right Subtree = 0



If at any time heights of left and right subtrees differ by more than 1, then *rebalancing* is done to restore AVL property, called *rotation*

Common Ops on AVL Tree

Creation of Tree

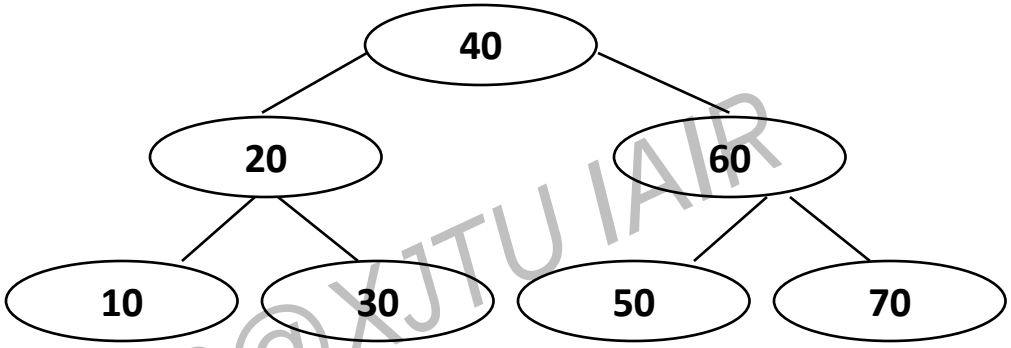
Traverse all nodes

Search for a value

Insertion of a node

Deletion of a node

Deletion of tree



Cause topology modification

time complexity of Traverse and Search in AVL is $O(\log n)$

Common Ops on AVL Tree

Creation of Tree

Traverse all nodes

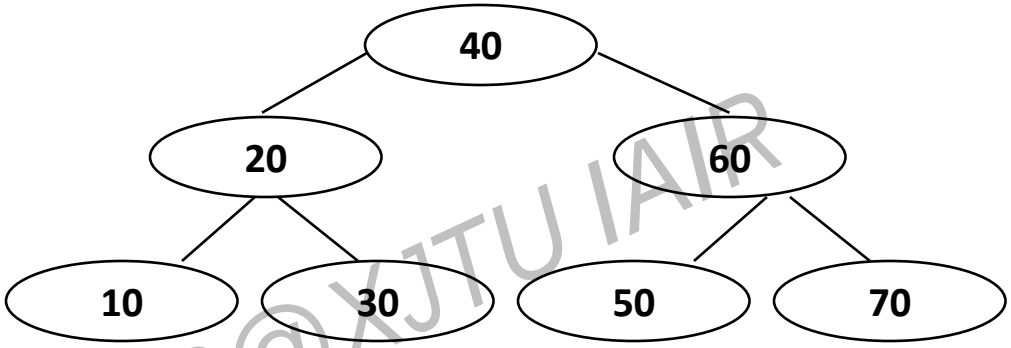
Search for a value

Insertion of a node

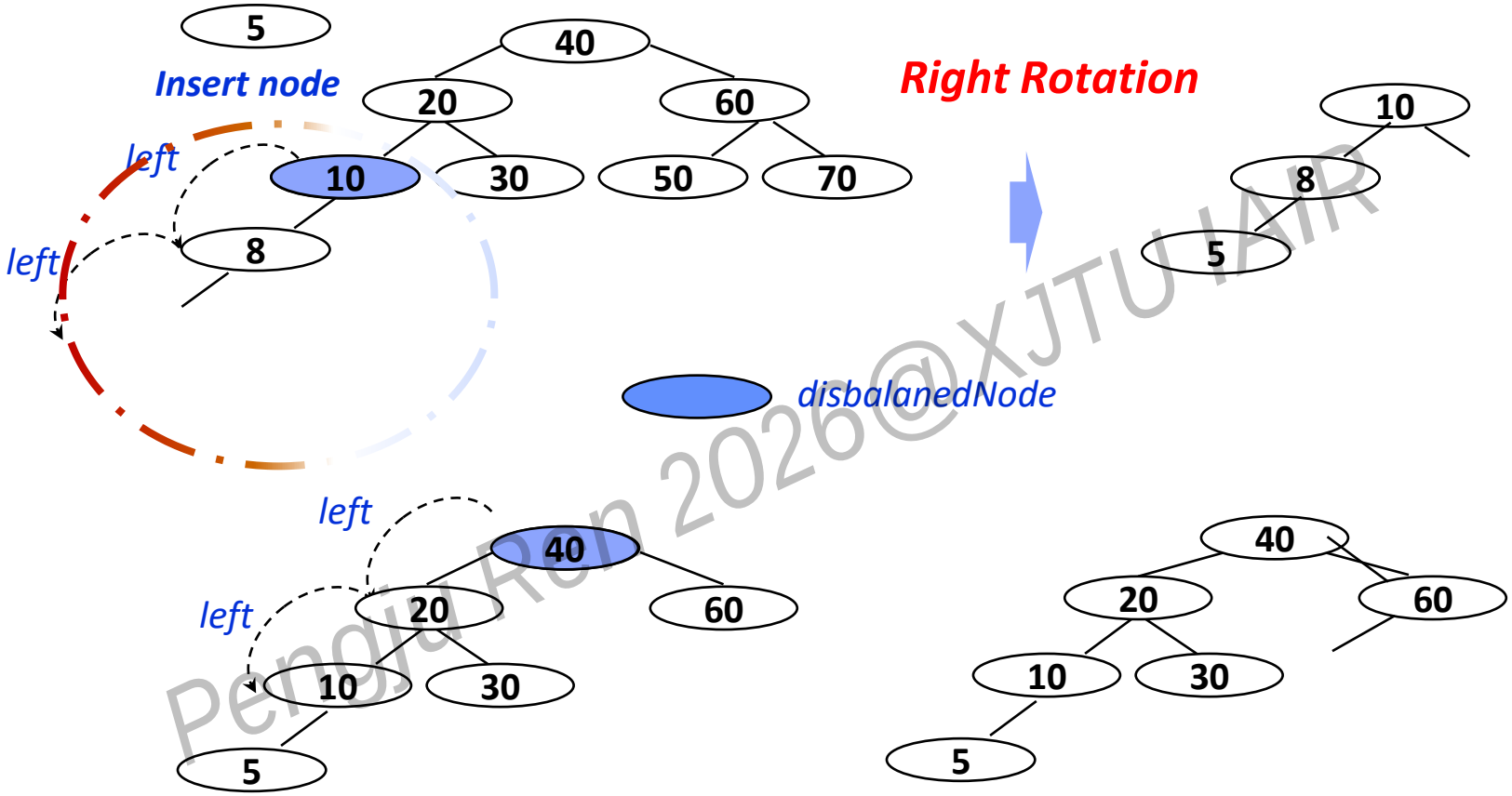
- *C1: Rotation is not required*
- *C2: Rotation is required*
  - *Left-left Cond*
  - *Left-right Cond*
  - *Right-left Cond*
  - *Right-right Cond*

Deletion of a node

Deletion of tree



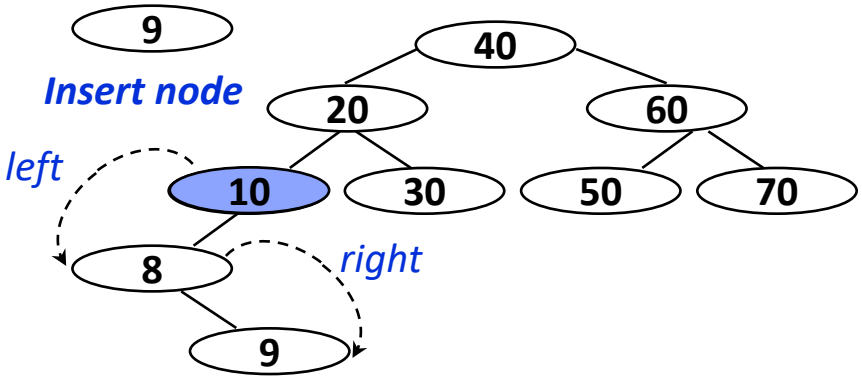
Insertion of a node in AVL Tree (left-left)

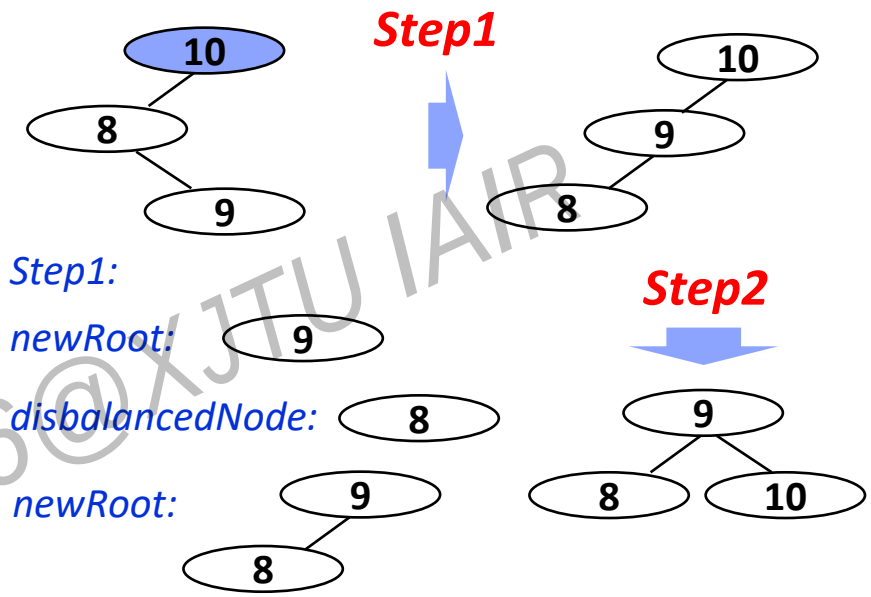



```
rotateRight(disbalancedNode):
  newRoot=disbalancedNode.leftChild
  disbalancedNode.leftChild = newRoot.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

What is time/space complexity of this?

Insertion of a node in AVL Tree(left-right)





Algo of Left Right Conditon:

1) Rotate left *disbalancedNode.leftChild*
2) Rotate right *disbalancedNode*

Step1: `rotateleft(disbalancedNode):`

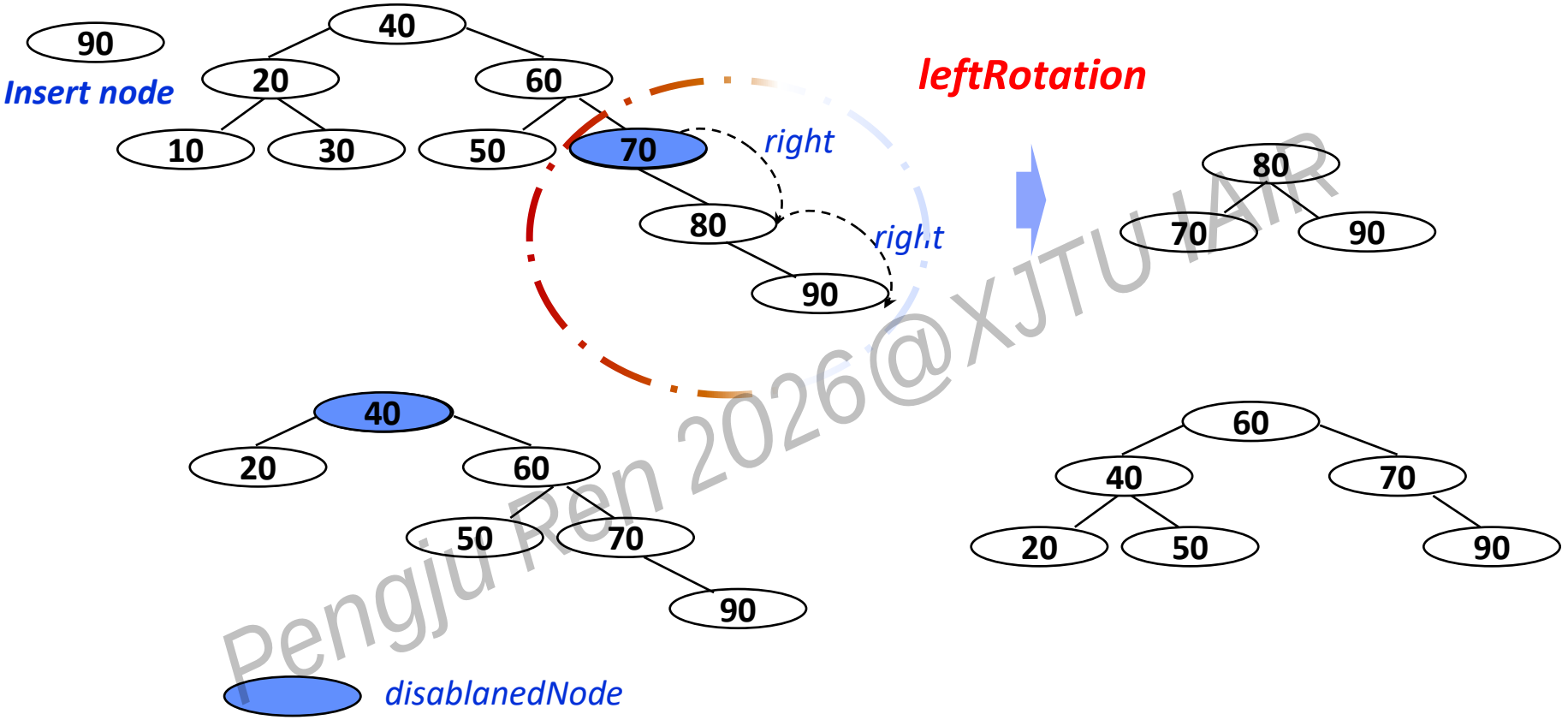
```
newRoot=disbalancedNode.rightChild
disbalancedNode.rightChild = newRoot.leftChild
newRoot.leftChild = disbalancedNode
update height of disbalancedNode and newRoot
return newRoot
```

What is time/space complexity of this?

Step2: `rotateRight(disbalancedNode):`

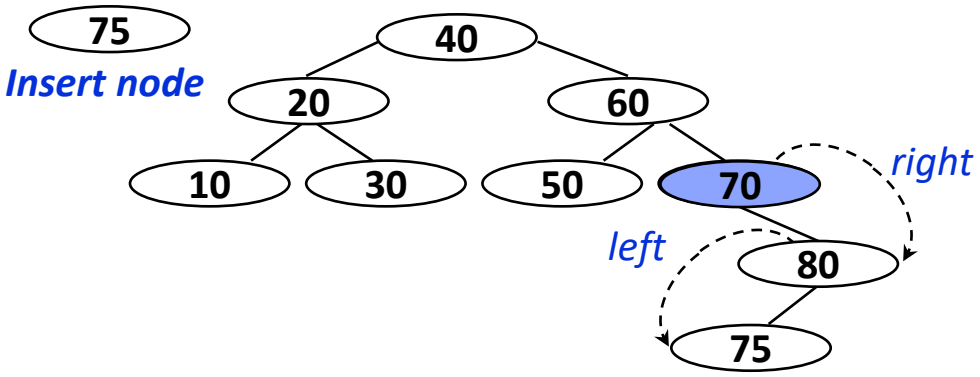
```
newRoot=disbalancedNode.leftChild
disbalancedNode.leftChild = newRoot.rightChild
newRoot.rightChild = disbalancedNode
update height of disbalancedNode and newRoot
return newRoot
```

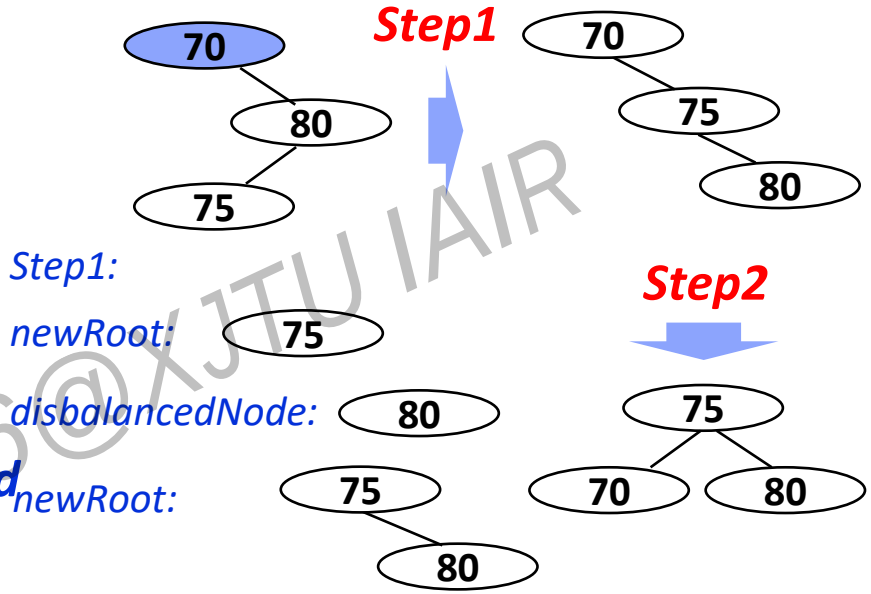
Insertion of a node in AVL Tree (right-right)




```
rotateleft(disbalancedNode):
  newRoot=disbalancedNode.rightChild
  disbalancedNode.rightChild = newRoot.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

Insertion of a node in AVL Tree (right-left)





Algo of right-left Conditon:

1) Rotate right *disbalancedNode.rightChild*
2) Rotate left *disbalancedNode*

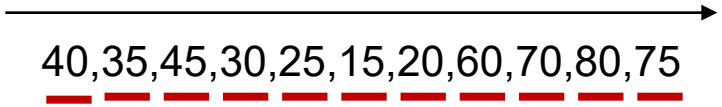
Step1:

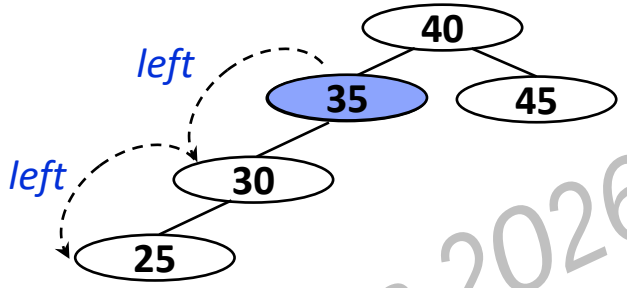
```
rotateRight(disbalancedNode):
  newRoot=disbalancedNode.leftChild
  disbalancedNode.leftChild = newRoot.leftChild.rightChild
  newRoot.rightChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

Step2:

```
rotateleft(disbalancedNode):
  newRoot=disbalancedNode.rightChild
  disbalancedNode.rightChild = newRoot.rightChild.leftChild
  newRoot.leftChild = disbalancedNode
  update height of disbalancedNode and newRoot
  return newRoot
```

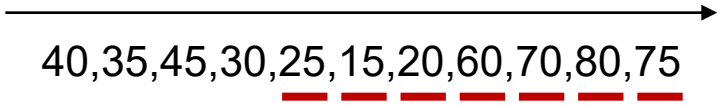
Insert a node in AVL Tree (all together)

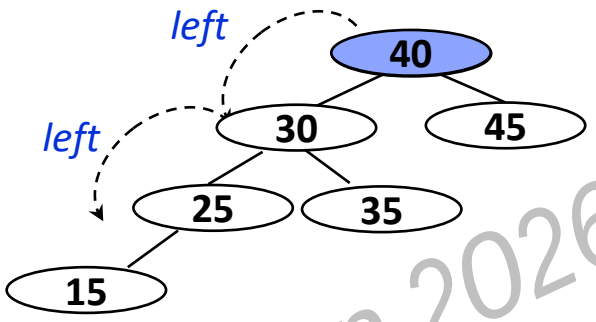




- ☐ Left-left condition
- ☐ Left-right condition
- ☐ Right right condition
- ☐ Right left condition

Insert a node in AVL Tree (all together)



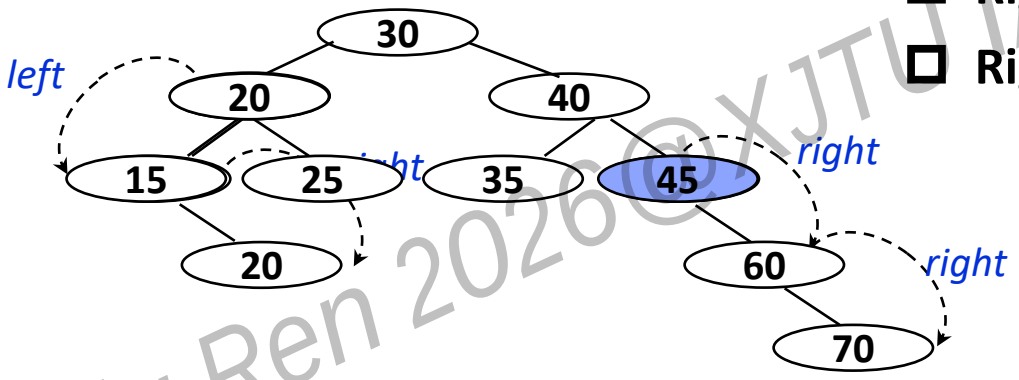


- Left-left condition
- Left-right condition
- Right right condition
- Right left condition

Insert a node in AVL Tree (all together)

40,35,45,30,25,15,20,60,70,80,75

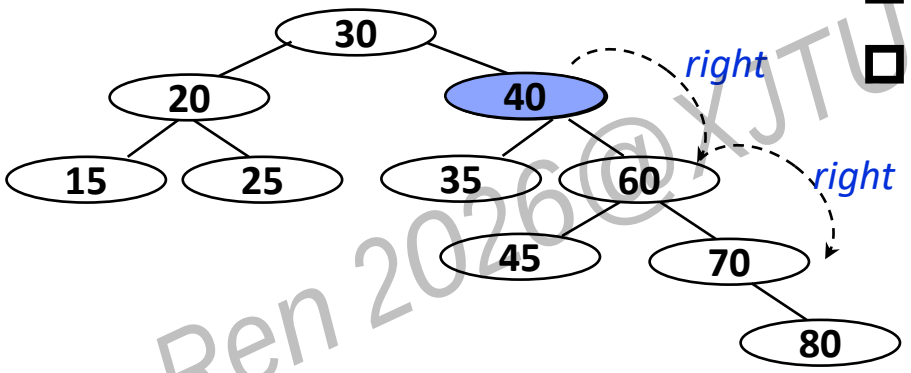
- Left-left condition
- Left-right condition
- Right right condition
- Right left condition



Insert a node in AVL Tree (all together)

40,35,45,30,25,15,20,60,70,80,75

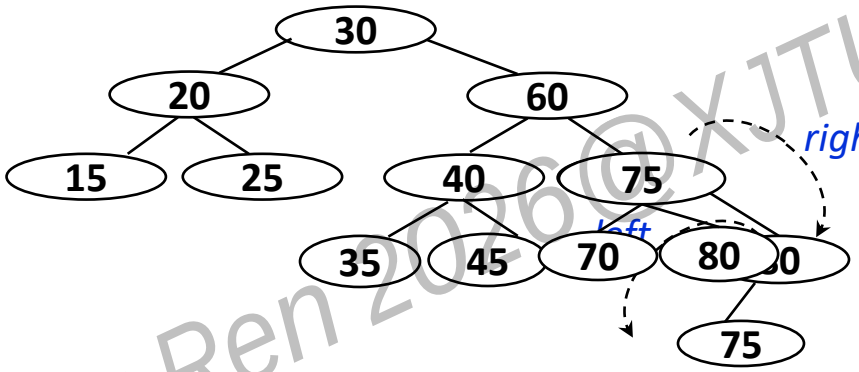
- Left-left condition
- Left-right condition
- Right right condition
- Right left condition



Insert a node in AVL Tree (all together)

40,35,45,30,25,15,20,60,70,80,75

- Left-left condition
- Left-right condition
- Right right condition
- Right left condition



Delete a node in AVL Tree

C1: Rotation is not required

C2: Rotation is required

- *Left-left Cond*
- *Left-left Cond*
- *Left-left Cond*
- *Left-left Cond*

Delete entire AVL Tree

```
rootNode = None
rootNode.leftChild = None
rootNode.rightChild = None
```

Time and Space complexity of AVL Tree

	Time Complexity	Space Complexity
Create AVL	$O(1)$	$O(1)$
Insert a node to AVL	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(\log n)}$
Delete a node from AVL	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(\log n)}$
Search for a node in AVL	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(\log n)}$
Traverse AVL	$O(n)$	$O(n)$
Delete entire AVL	$O(1)$	$O(1)$

Red-Black Tree/Splay Tree

Summary

- **LCA (*lowest common ancestor):* Binary Lifting is more efficient, highlighting the importance of algorithm *preprocessing* and the trade-off between space and time complexities.**
- **Tree: Very important non-linear data structure, can be used to solve many problems**
- **BST(Binary Search Tree): Used for efficient searching and sorting**
- **AVL: a *self-balancing* Binary Search Tree(BST), the height difference between the left and right subtrees of any node is at most 1**

Homework 6

Insert the sequence [50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45, 55, 65, 75, 85] into the AVL tree, outputting the *inorder traversal* after each insertion to observe the tree's evolution.

Then delete the keys [20, 60, 30, 80] in order, outputting the *levelorder traversal* after each deletion to observe the tree's evolution.

Submission requirements:

1: Description of the implementation approach
2: Performance test results and analysis
3: Difficulties encountered and solutions

Homework naming: HW6-Class X-stu ID-name, e.g. HW6-Class01-12345678-张三

Submission email: chengyuma@stu.xjtu.edu.cn

Submission time: 4.12 24:00