Binary Search Trees
Lecture 21

Menu

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Tables (Maps)

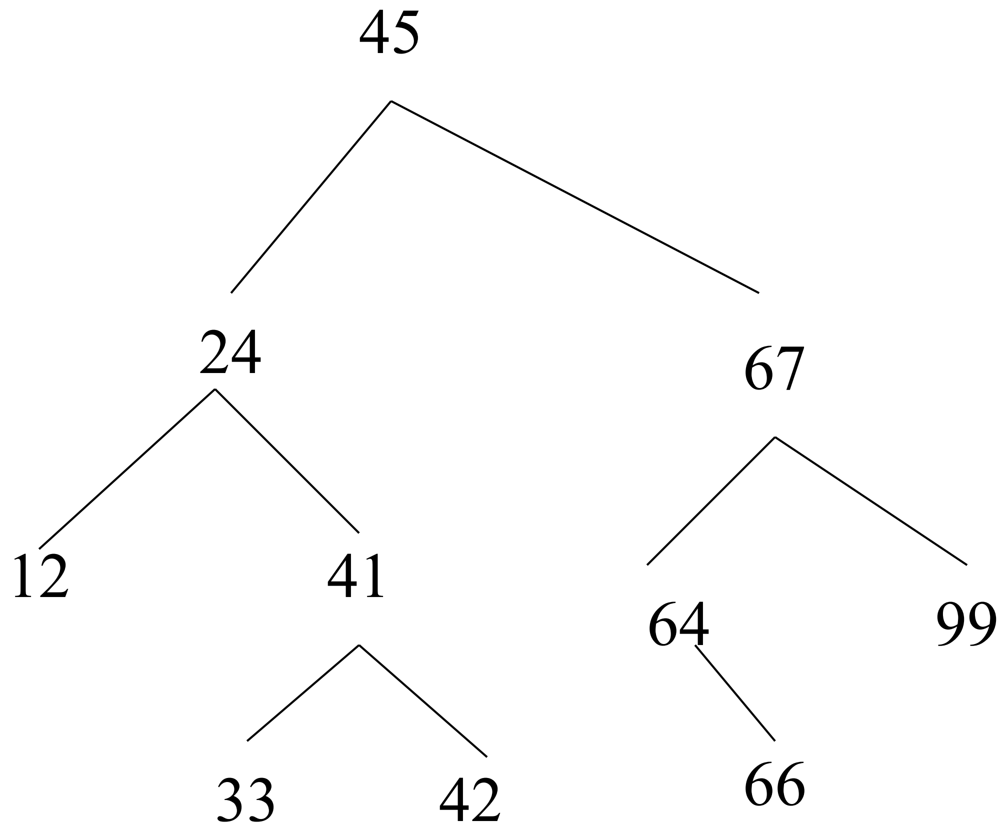
- indexed container
- Associate information with a key
  - *key* is often a character string
  - *info* is any information
- E.g. a phone book
  - key is the name of a person or business
  - info is their phone number & address
- Typical Operations on Tables

  void insert(string key, Object o);
  object lookup(string key);
  void remove(string key);
- Alternative implementations include
  - Search Lists
  - Binary Search Trees
  - Hash Tables

Search Lists

- a linked list with key, info, and next
- **O(N)** in average for insert, lookup, and remove

Binary Search Tree



Binary Search Tree Definitions

- A binary search tree is a binary tree where each node has a key
- The key in the left child (if exists) of a node is less than the key in the parent
- The key in the right child (if exists) of a node is greater than the key in the parent
- The left & right subtrees of the root are again binary search trees

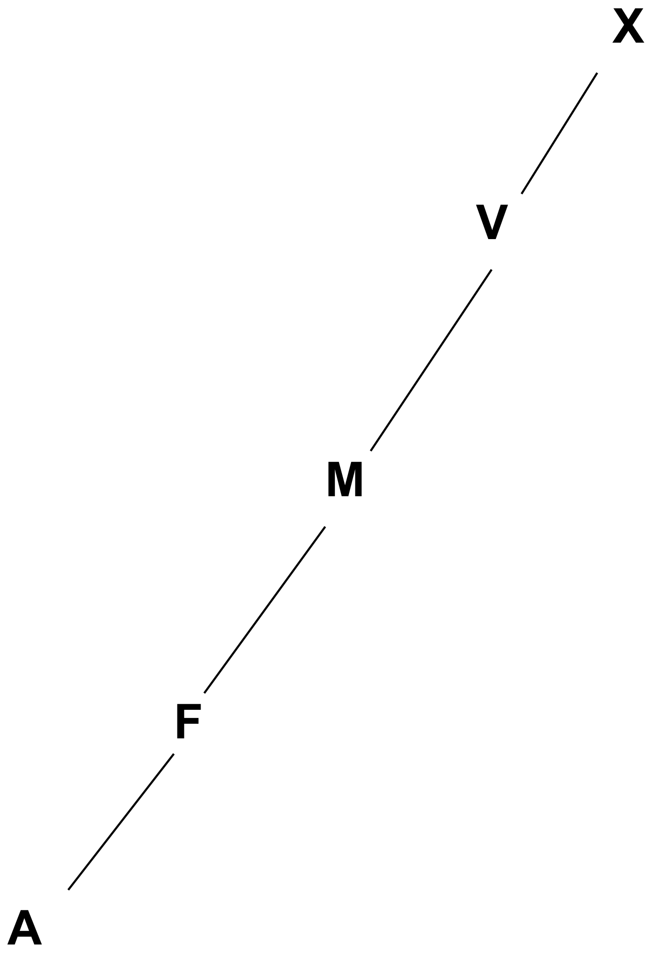
Binary Search Trees (BST)

- similar to a linked list, but two next pointers
- we call them *left* and *right*
- for each node n, with key k
  - n->left contains only nodes with keys < k
  - n->right contains only nodes with keys > k
- **O(log N)** in average for insert, lookup, and remove

Worst Cases

- operations can degenerate to O(N) – worst case!
- degenerates to a linked list
  - when keys are inserted in ascending order
    - all keys are to the right
  - when keys are inserted in descending order
    - all keys are to the left
- ideal is mid first, then successive middles, etc.
- random order also works fairly well

Degenerated Tree

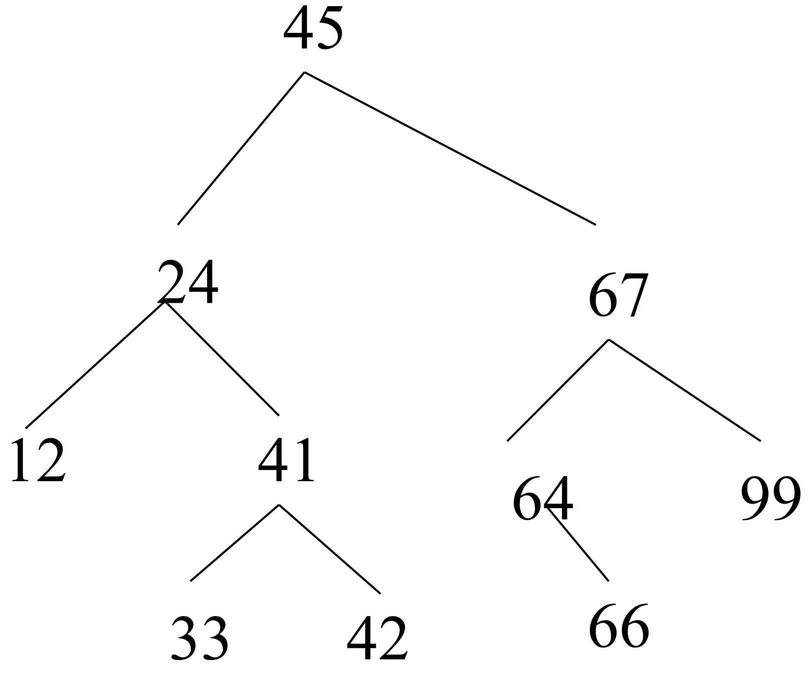


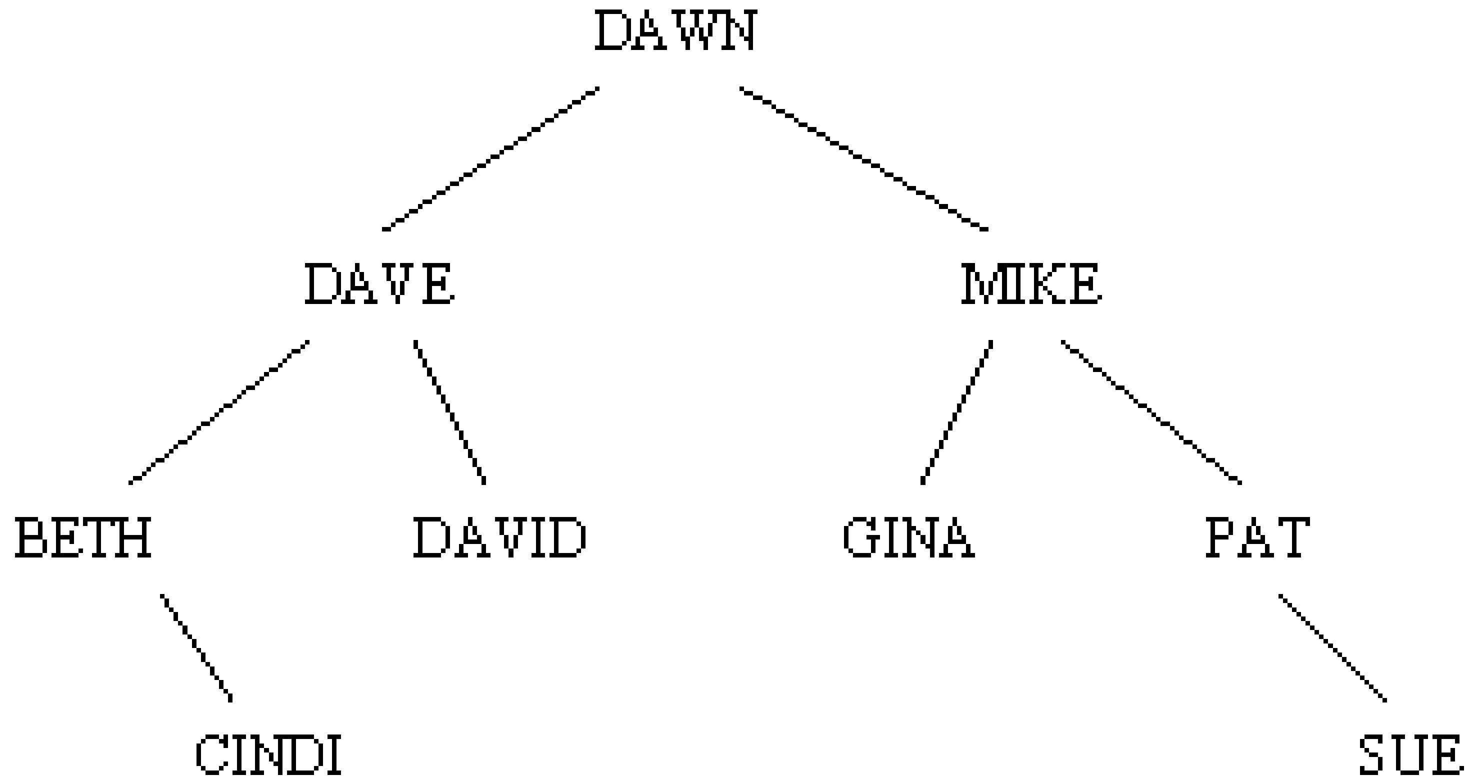
Binary Tree Traversal

- inOrder
- preOrder
- postOrder

inOrder traversal: recursive

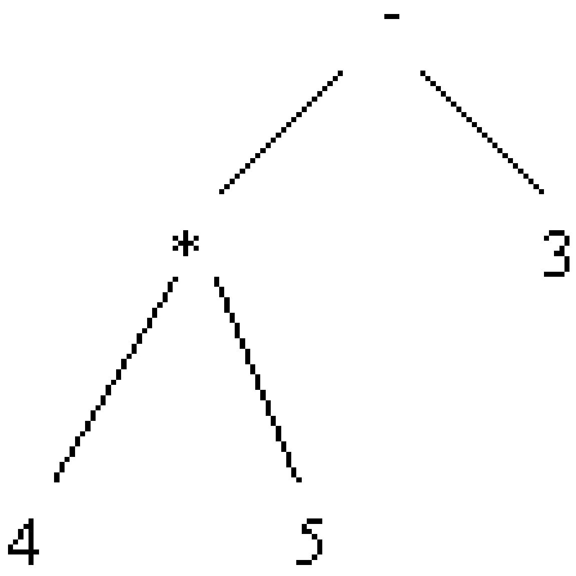
- traverse the left subtree inOrder
- process (display) the value in the node
- traverse the right subtree inOrder



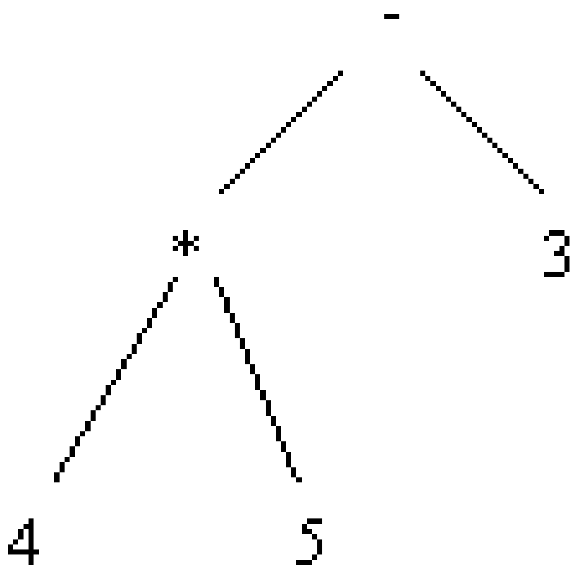


BETH, CINDI, DAVE, DAVID, DAWN, GINA, MIKE, PAT, SUE

Exercise: inorder traversal of the binary expression tree for 4 * 5 - 3

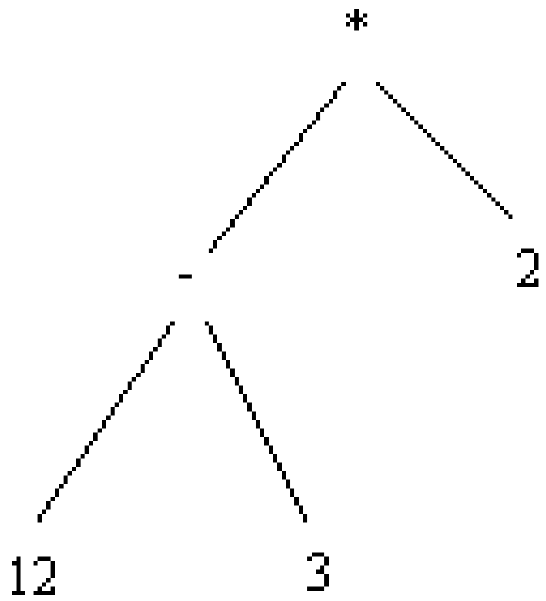


inorder traversal of the binary expression tree for 4 * 5 - 3

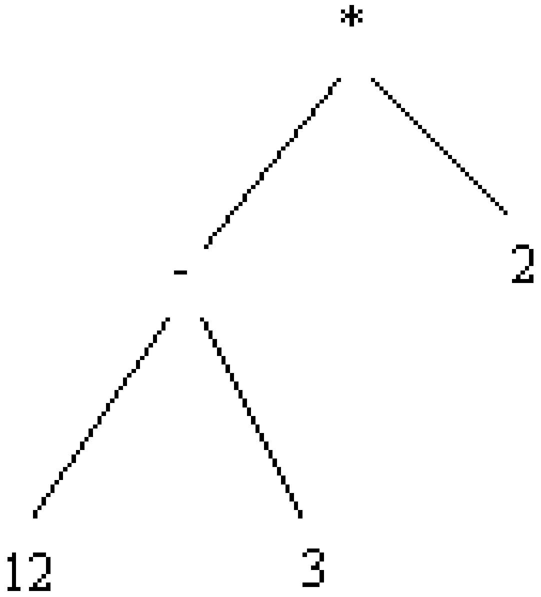


4 * 5 - 3

Exercise: inorder traversal of the binary expression tree for (12-3)*2



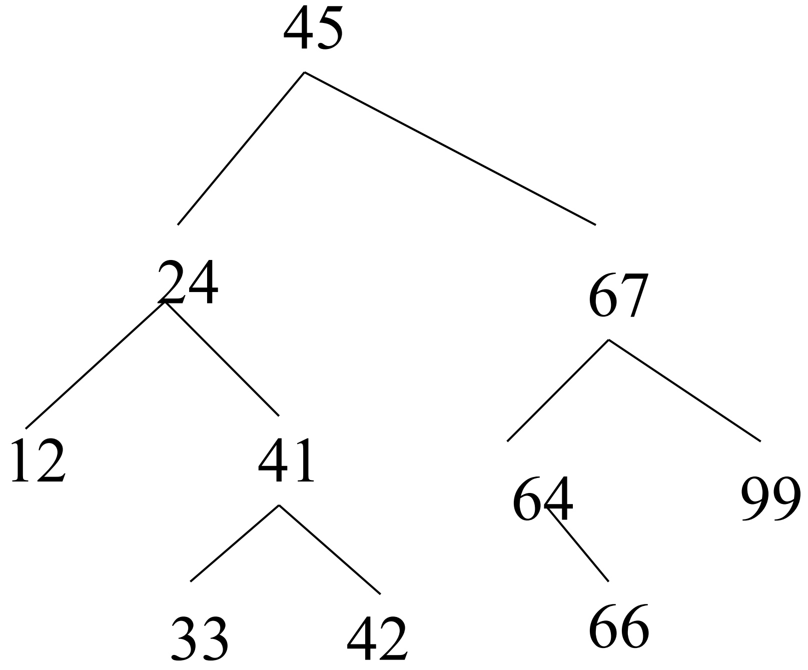
inorder traversal of the binary expression tree for (12-3)*2



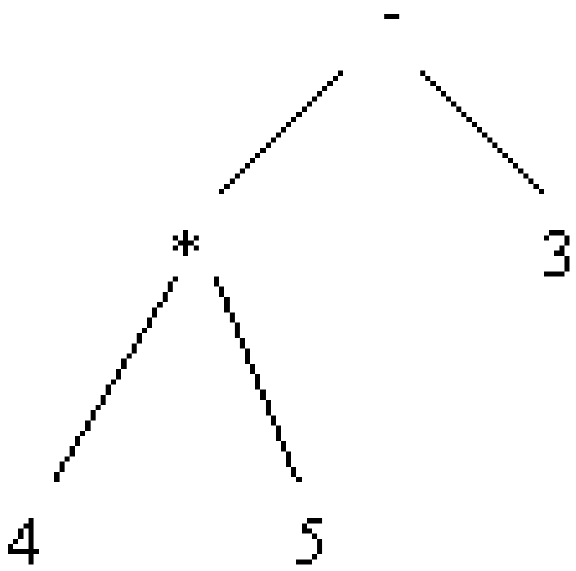
12 – 3*2

preOrder traversal: recursive

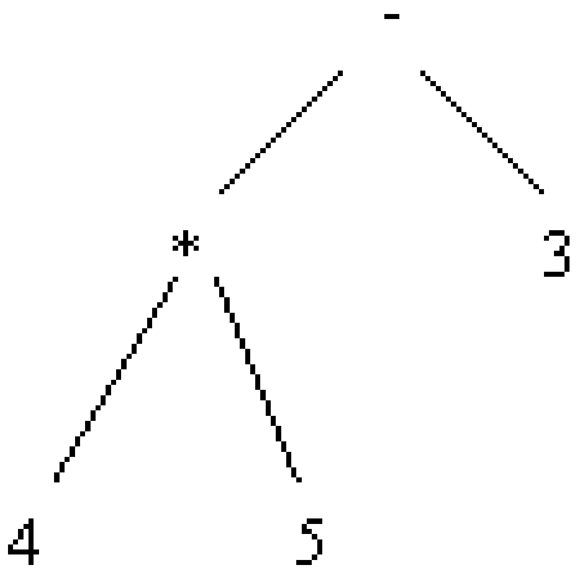
- process (display) the value in the node
- traverse the left subtree preOrder
- traverse the right subtree preOrder
- ?



Exercise: preorder traversal of the binary expression tree for 4 * 5 - 3



preorder traversal of the binary expression tree for 4 * 5 - 3

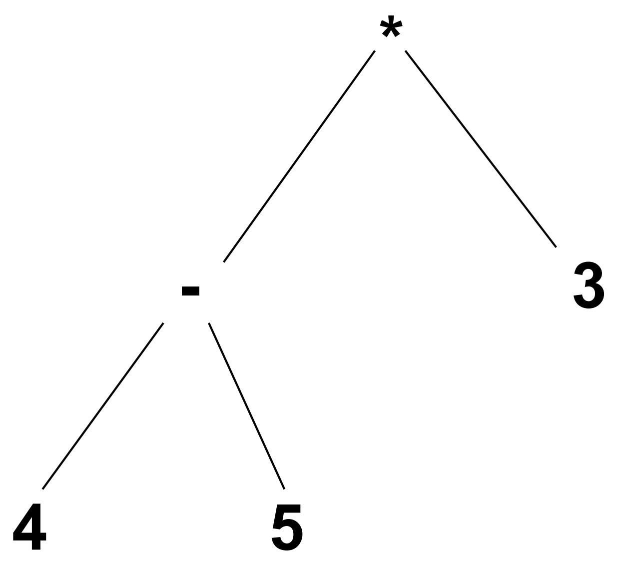


- * 4 5 3

Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

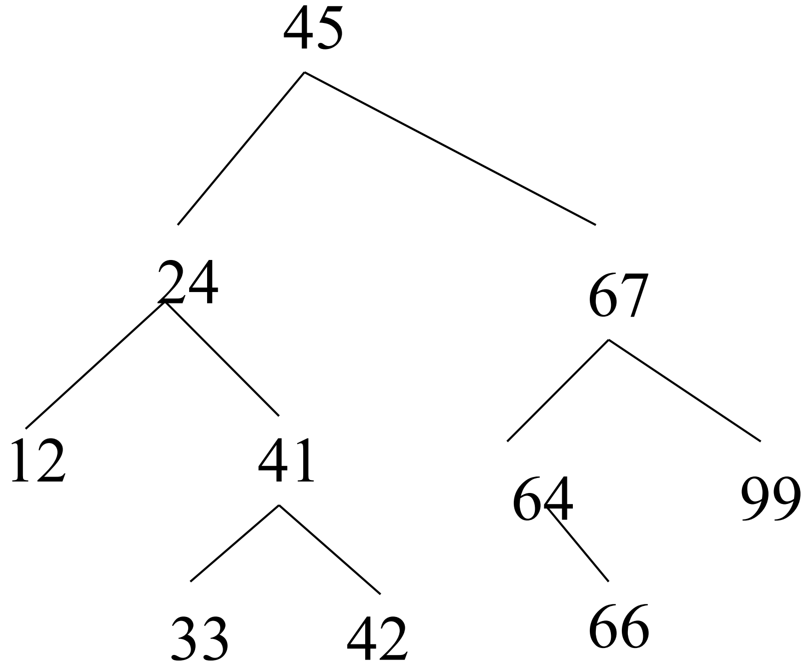
Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

- Correct evaluation in preorder should be: * - 4 5 3
- Corresponding tree:

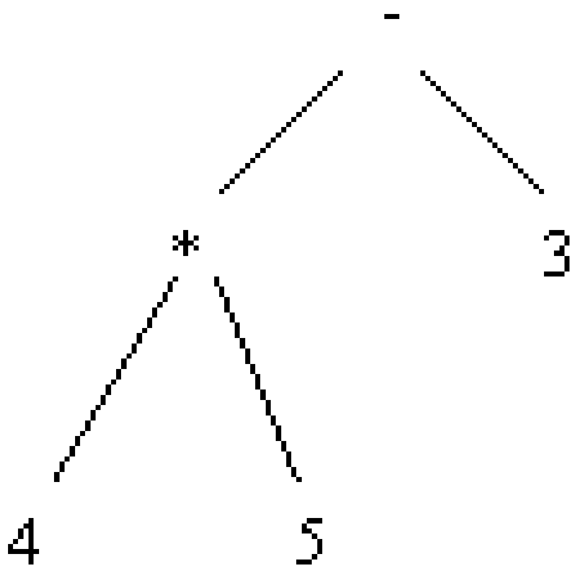


postOrder traversal: recursive

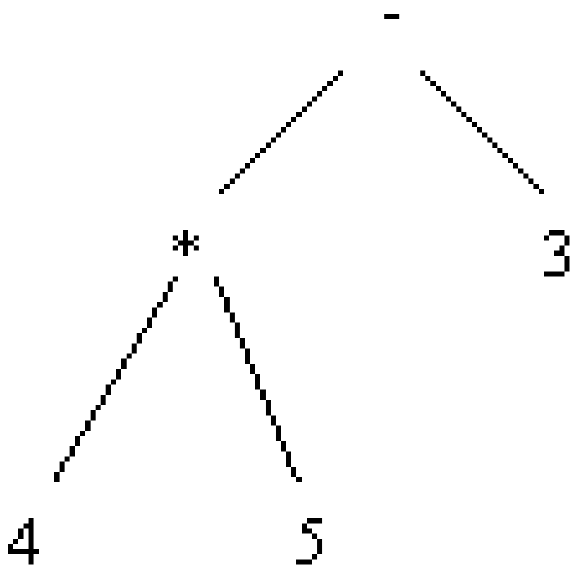
- traverse the left subtree postOrder
- traverse the right subtree postOrder
- process (display) the value in the node
- ?



Exercise: postorder traversal of the binary expression tree for 4 * 5 - 3



postorder traversal of the binary expression tree for 4 * 5 - 3



4 5 * 3 -

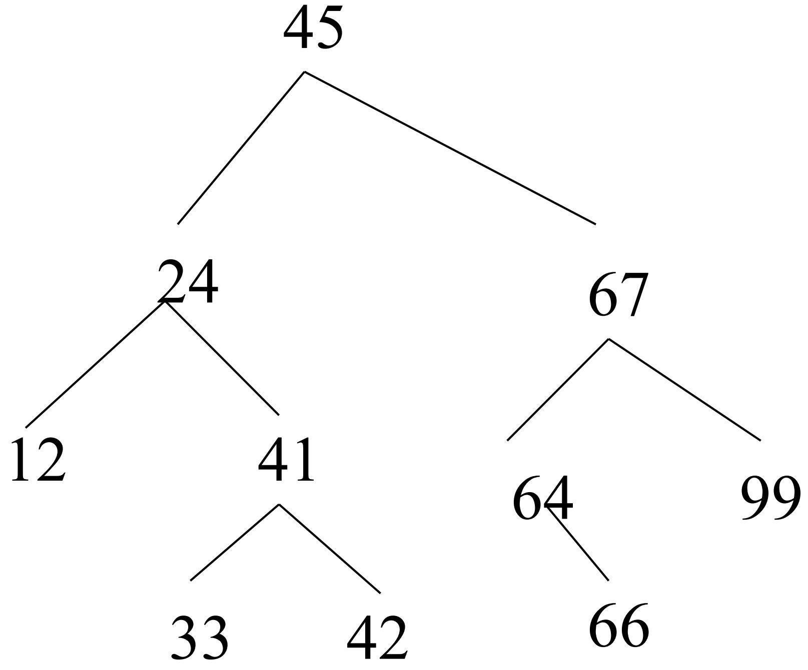
Ex: How do you construct the binary tree for 4-5*3 to be evaluated correctly in postorder?

Breadth-First traversal

- all previous traversals are *Depth-First* traversals
- visit all the nodes at depth 0, then depth 1, etc.
- may use a **queue** to traverse across levels

Breadth-First traversal

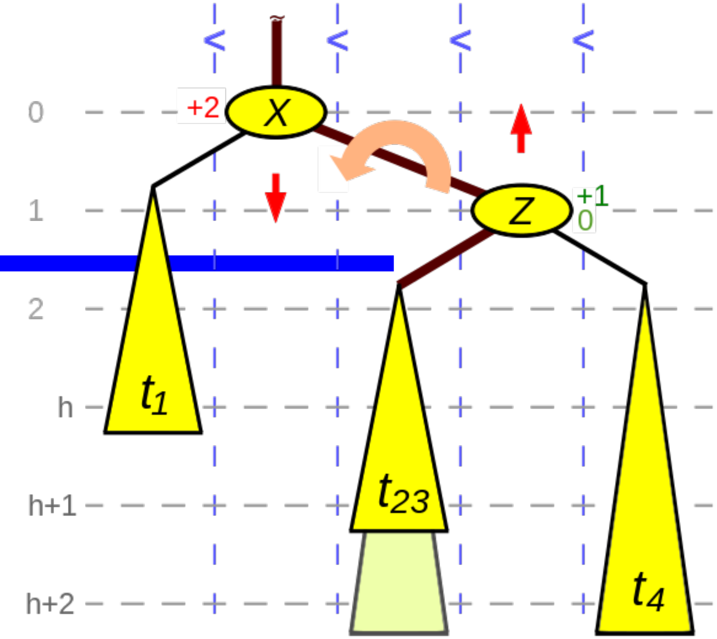
- ?

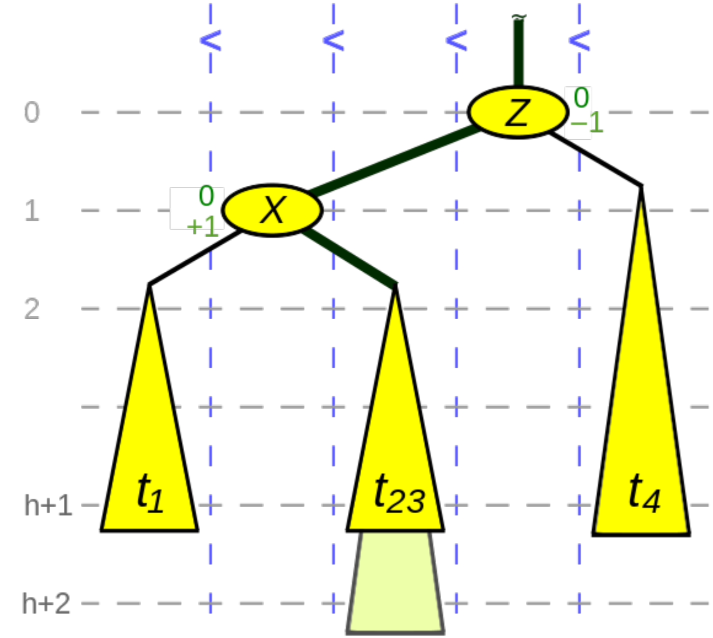


Balanced Search Trees

- use **rotations** to ensure tree is always 'full'
- prevents degenerative cases mentioned above
- truely O(log N) worst case for insert, lookup, remove
- insertion/removal takes more time
- but lookup is faster
- trickier to code correctly

Simple rotation *rotate_Left*(*X*,*Z*)





AVL Trees (Adelson-Velskii and Landis)

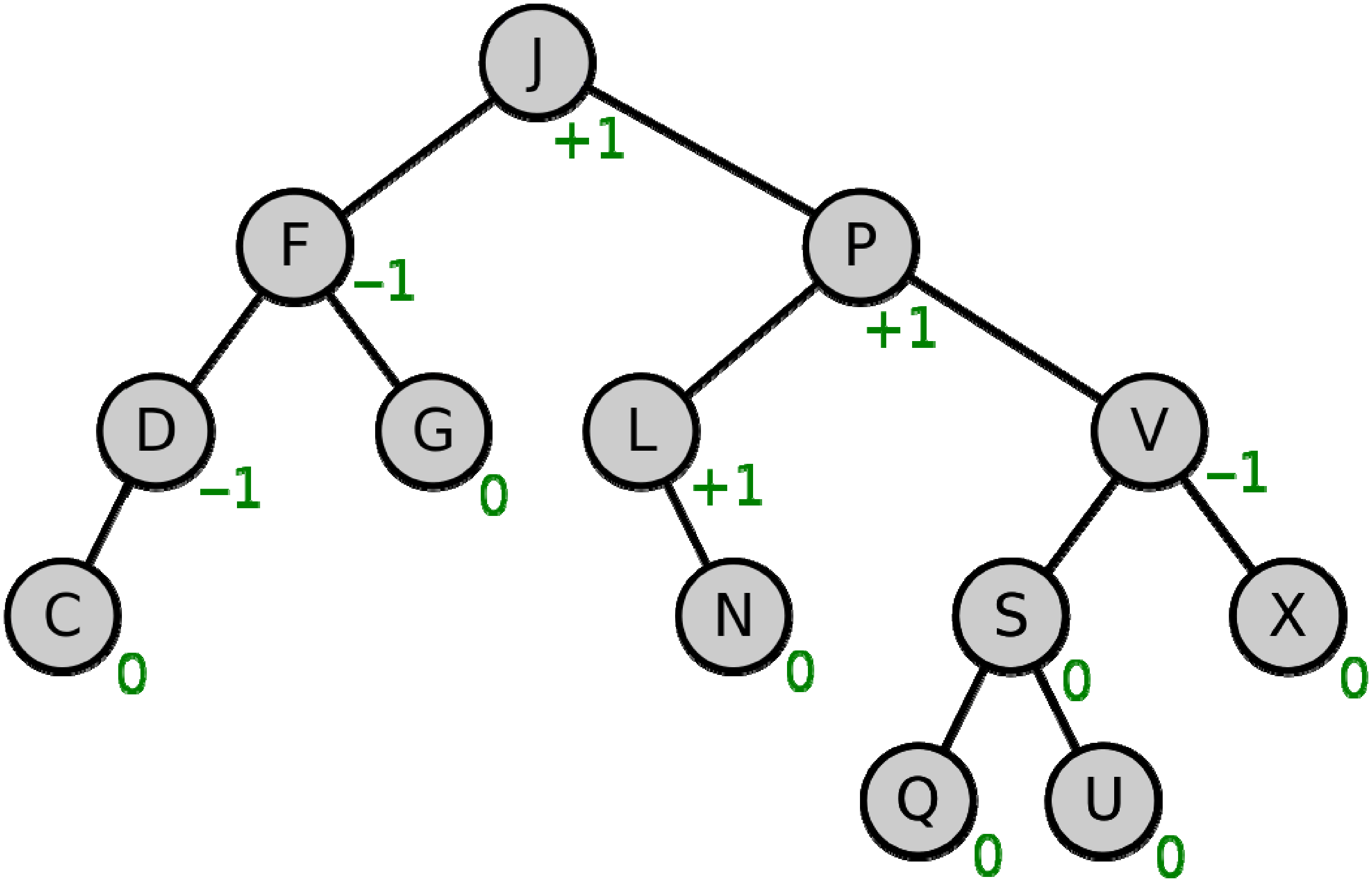
- An AVL Tree is a form of binary search tree
- Unlike a binary search tree, the worst case scenario for a search is O(log n).
- AVL data structure achieves this property by placing restrictions on the difference in height between the sub-trees of a given node - height balanced to within 1
- and re-balancing the tree if it violates these restrictions.

AVL Tree Balance Requirements

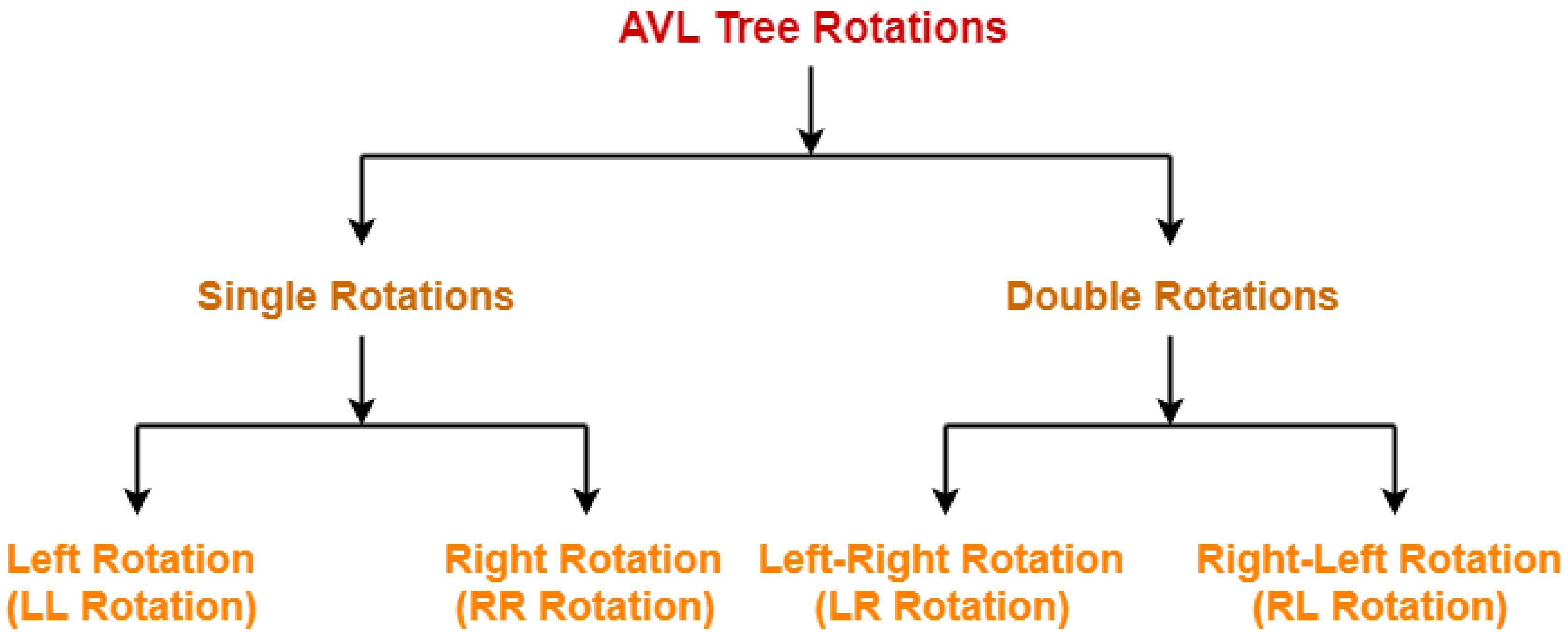
- A node is only allowed to possess one of three possible states:
- **Left-High (balance factor -1)**
  The left-sub tree is one level taller than the right-sub tree
- **Balanced (balance factor 0)**
  The left and right sub-trees both have the same heights
- **Right-High (balance factor +1)**
  The right sub-tree is one level taller than the left-sub tree

- If the balance of a node becomes -2 or +2 it will require re-balancing.
- This is achieved by performing a **rotation** about this node
- **Rotation does not break the existing properties for a search tree**

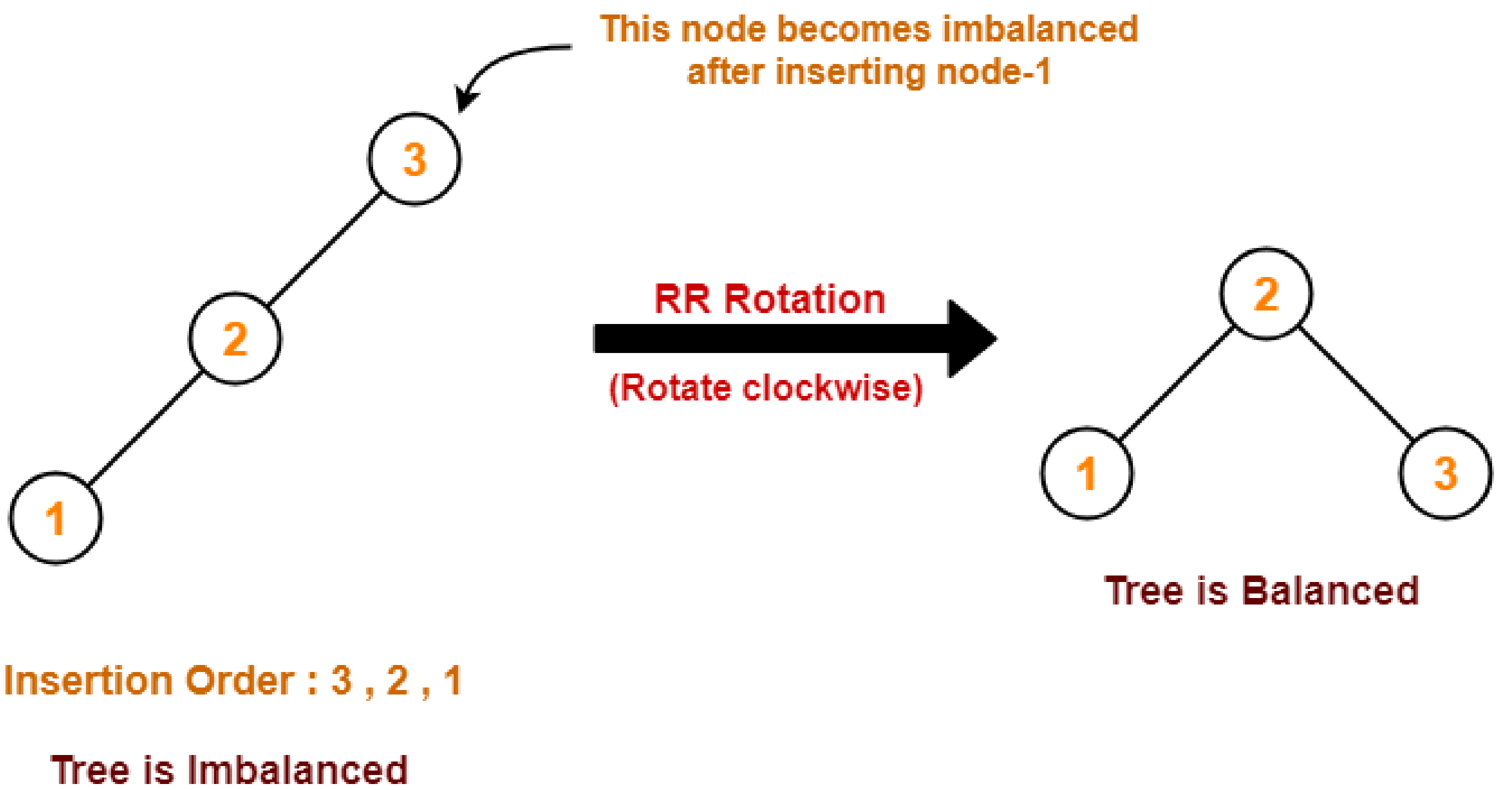
AVL tree with balance factors



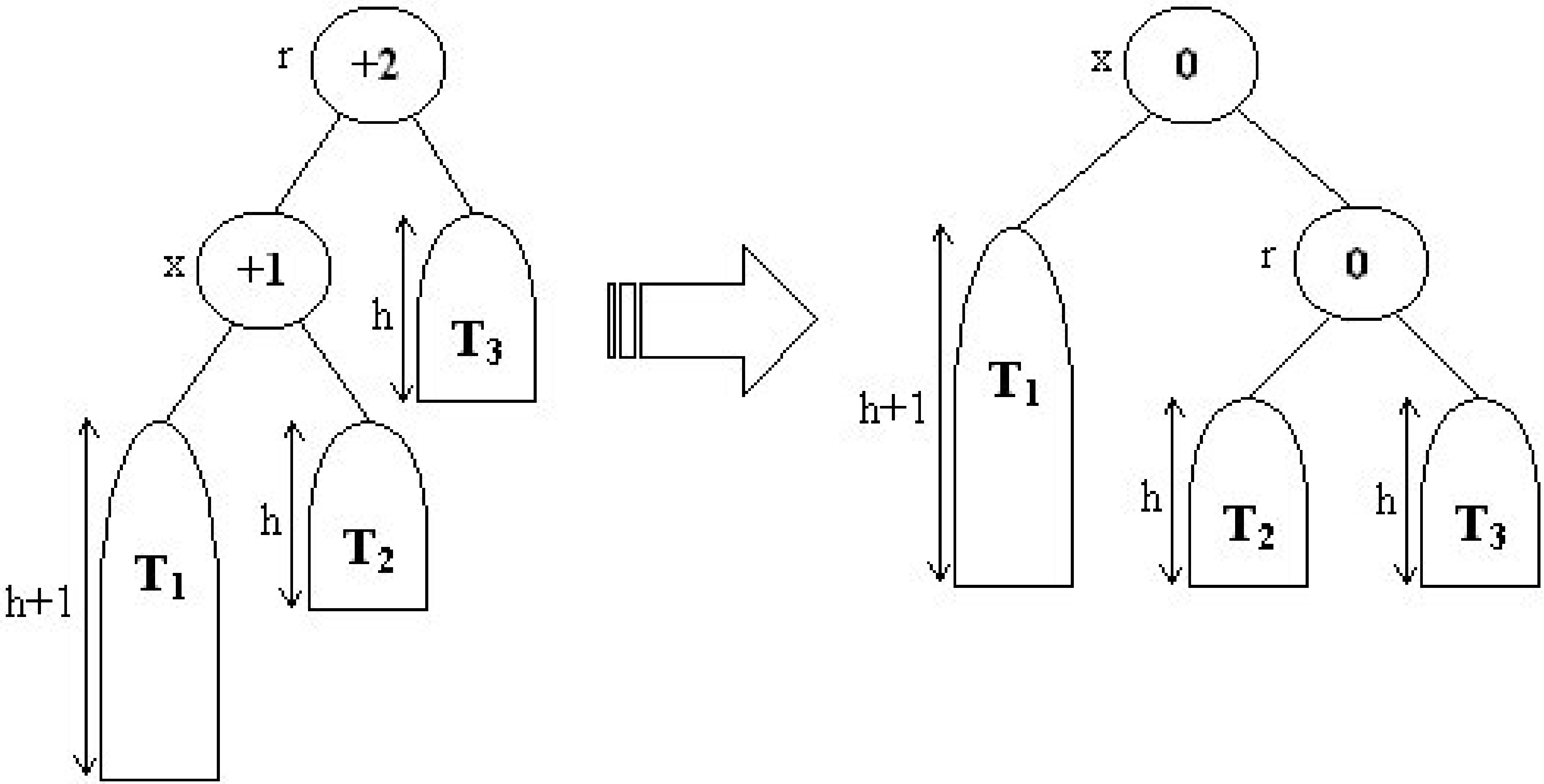
AVL Tree Rotations



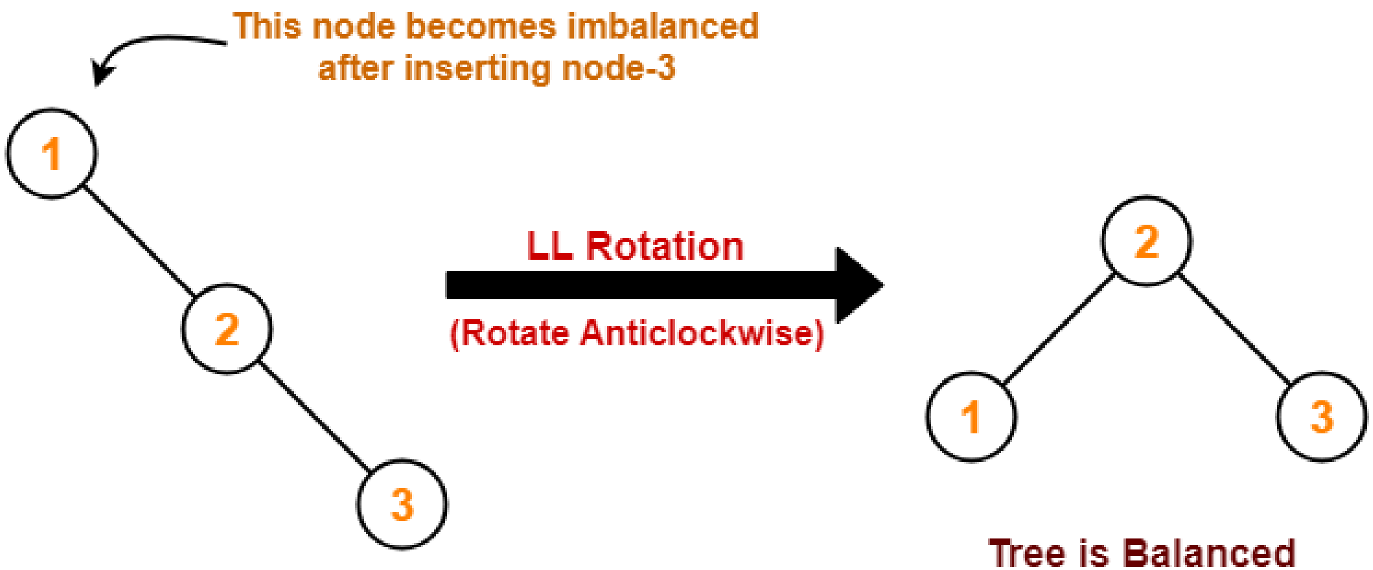
AVL Rotation - RR



AVL Rotation - RR



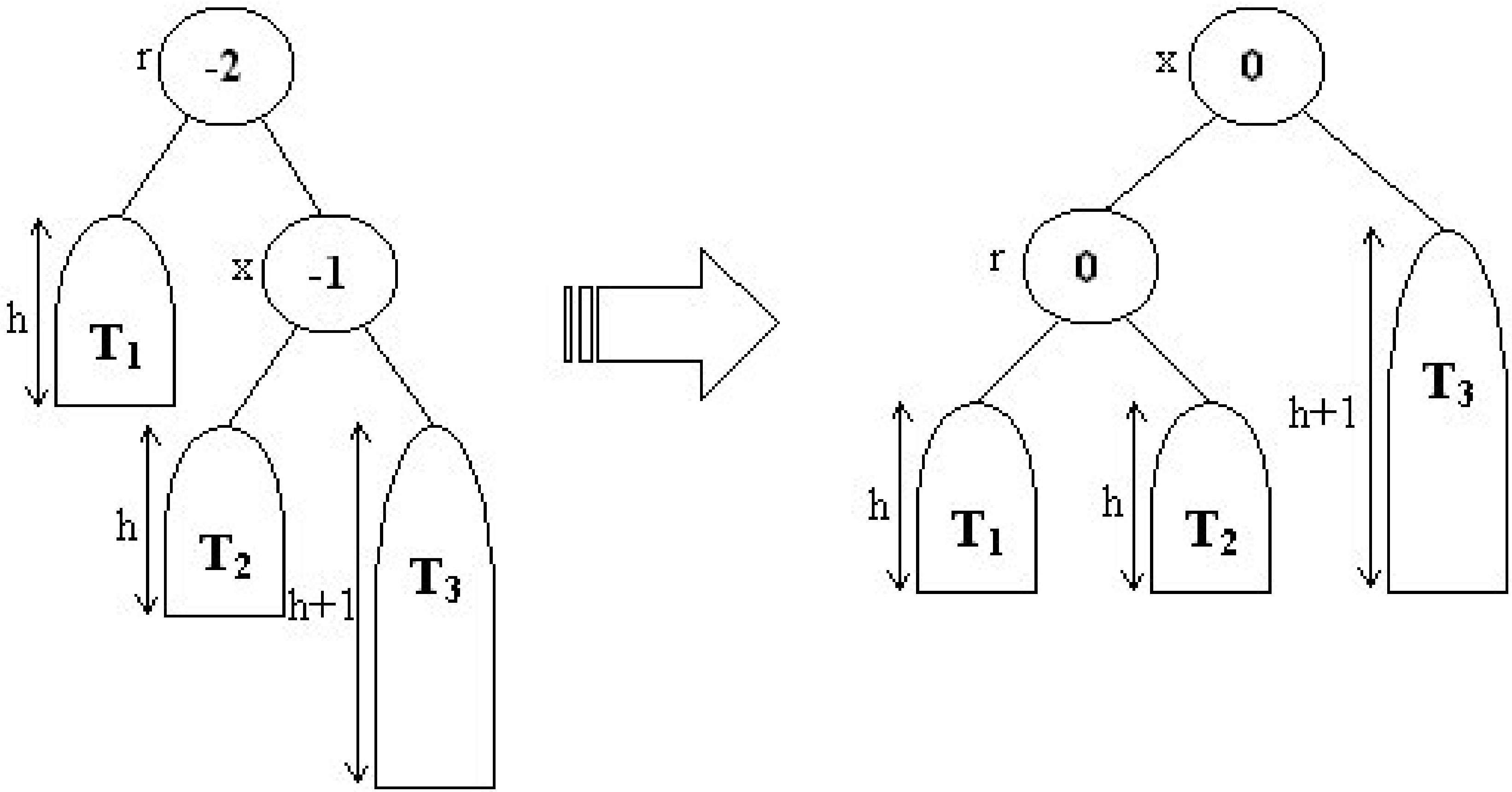
AVL Rotation - LL



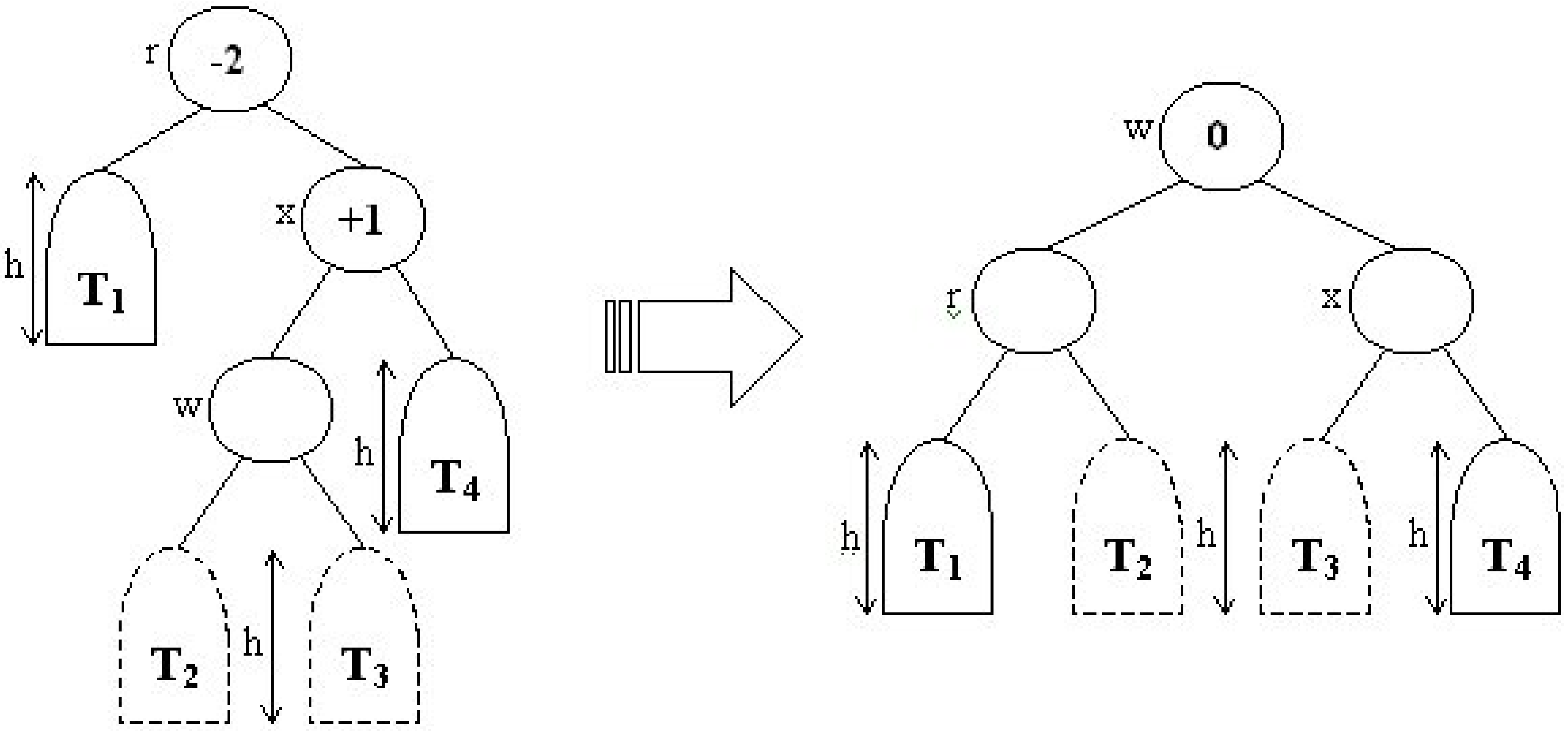
Insertion Order : 1 , 2 , 3

Tree is Imbalanced

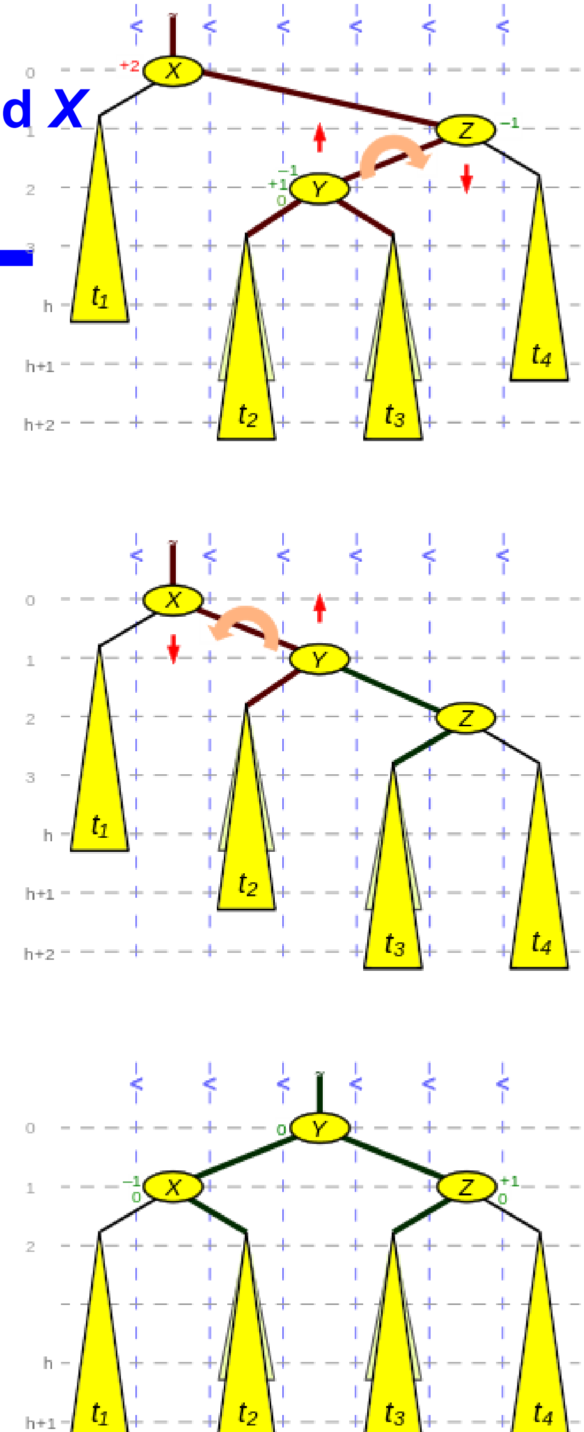
AVL Rotation - LL



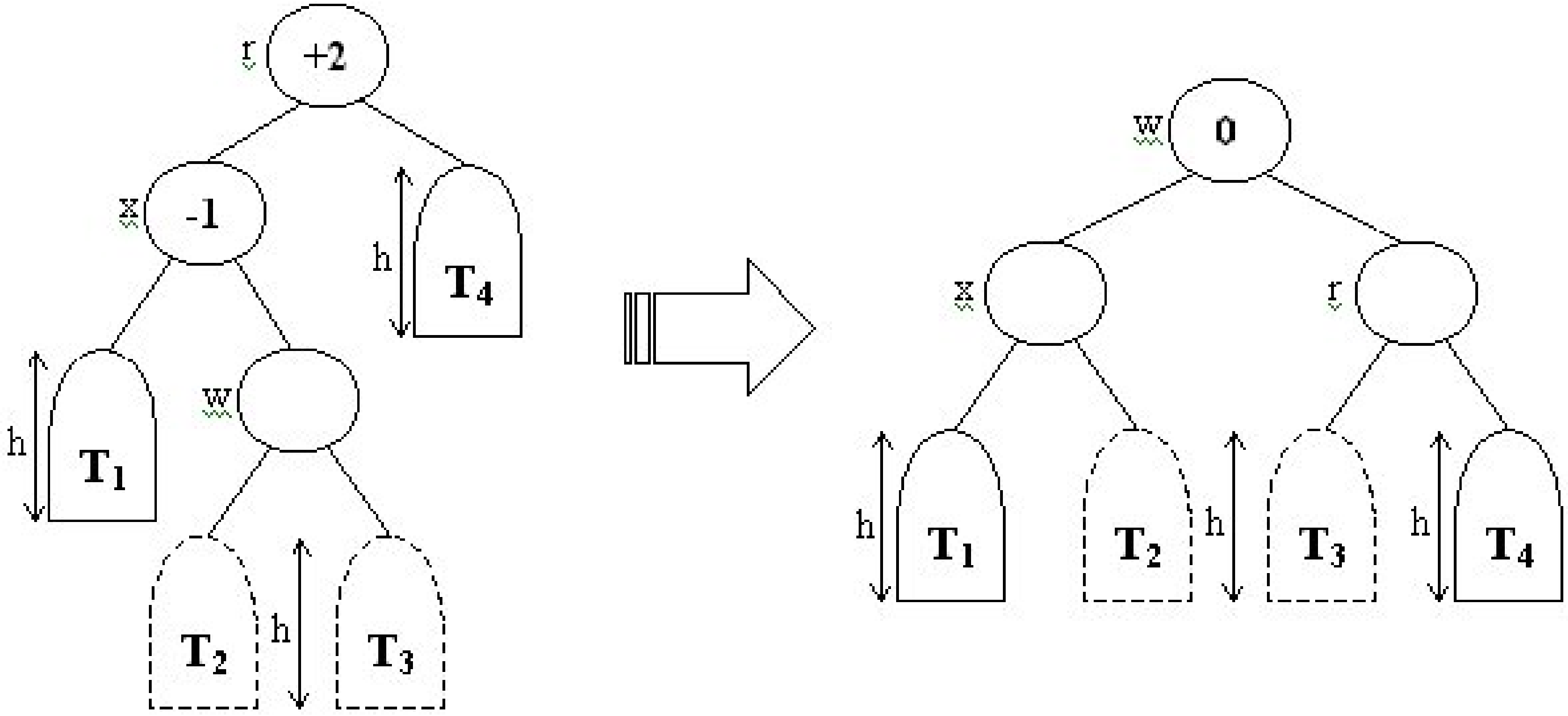
AVL Rotation - RL



AVL Double rotation *rotate_RightLeft*(*X*,*Z*) = *rotate_Right* around *Z* followed by *rotate_Left* around *X*



AVL Rotation - LR



AVL Tree Insertion

- AVL requires two passes for insertion:
- one pass down tree (to determine insertion)
- one pass back up to update heights and rebalance

AVL Tree Insertion

- Animation showing the insertion of several elements into an AVL tree. It includes left, right, left-right and right-left rotations.

AVL Time complexity in big O notation

Algorithm	Average	Worst case
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)
Space	O(n)	O(n)

Summary

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Readings

- [Mar07] Read 4.2, 4.3, 4.4, 4.8, 9.6
- [Mar13] Read 4.2, 4.3, 4.4, 4.8