CSE221

Lecture 14: AVL Trees

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

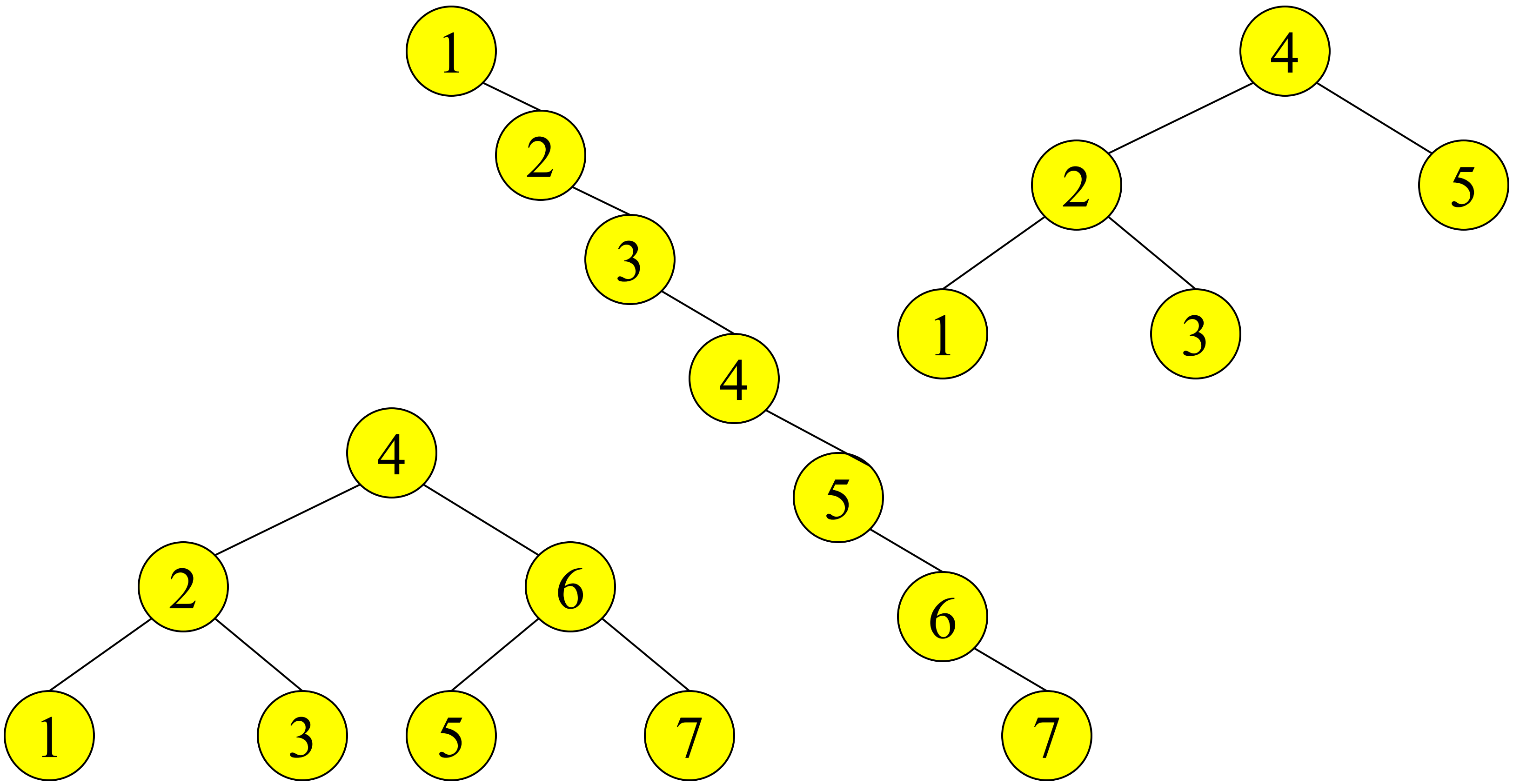
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Dynamic Balanced Binary Search Trees

- Average and maximum search time in binary search trees depend on the height of the tree
- Minimum height of a binary tree with n nodes is log n
- Dynamic balanced binary search tree is to keep the height of a binary search tree O(log n) for search, insert, delete

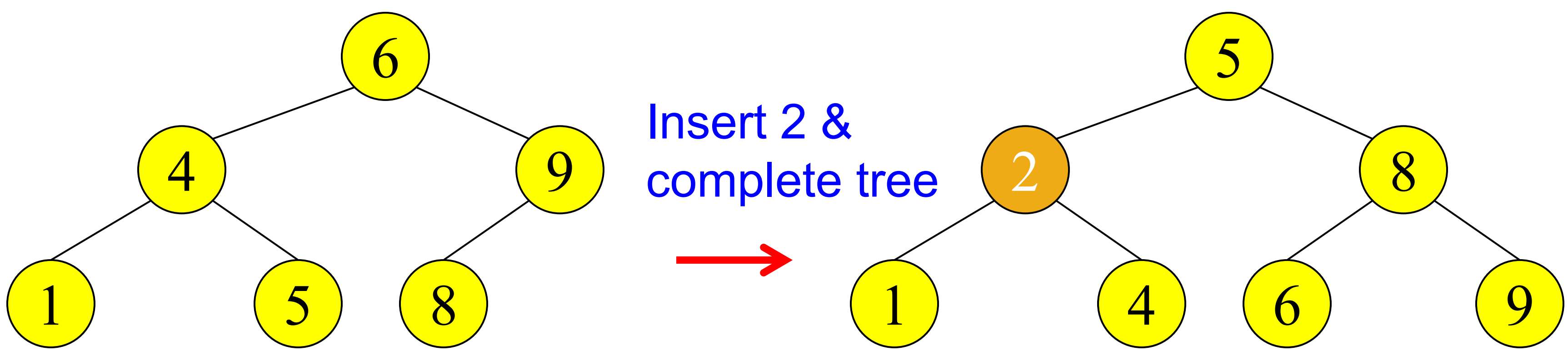
Balanced and Unbalanced BST

- Balanced?



Perfect Balance

- Want a complete tree after every operation
  - tree is full except possibly in the lower right
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL Trees: and early example

- Adelson-Velskii and Landis
- Height-balanced binary tree
  - But not perfectly balanced

Balance factor

- For every node x, define its balance factor

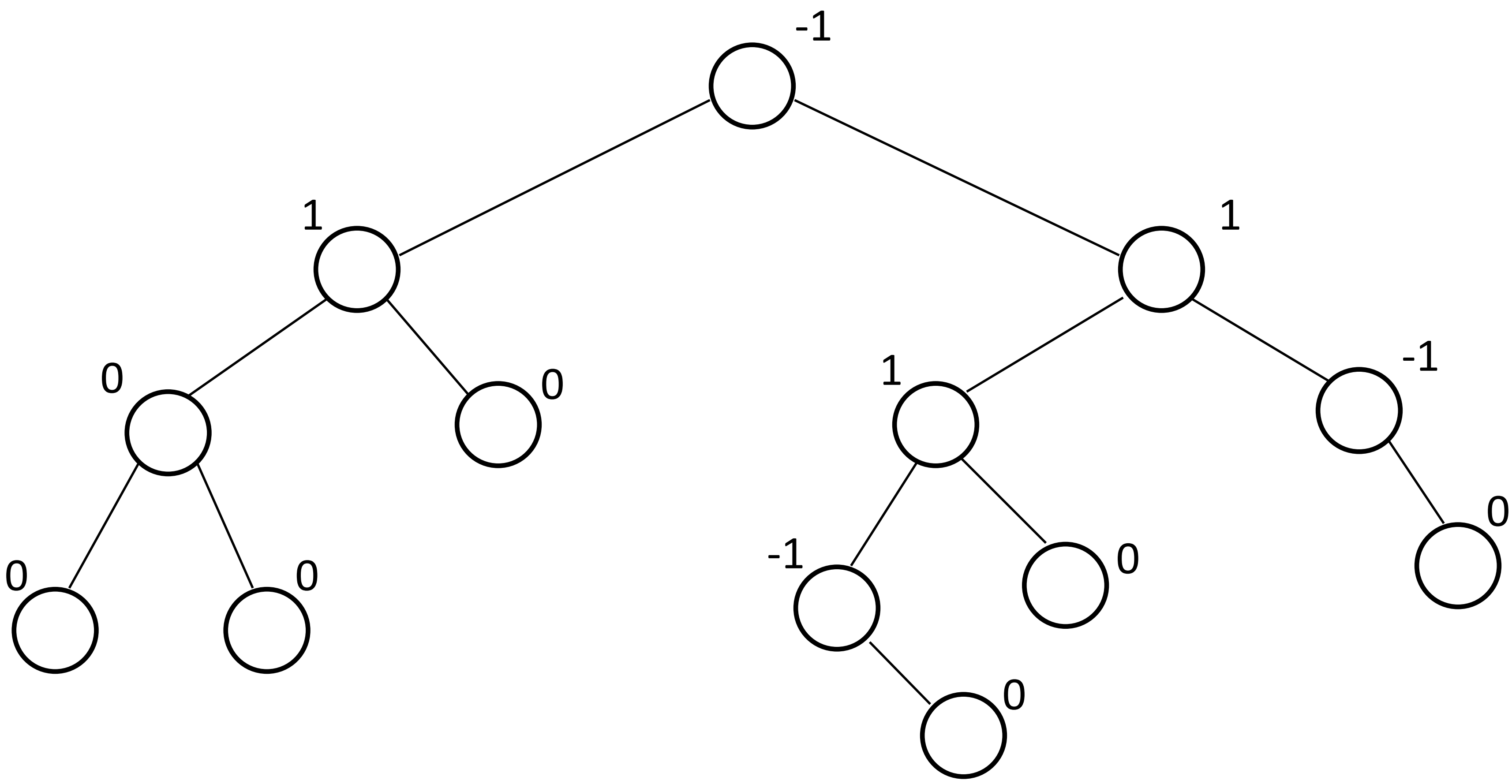
  balance factor of x = $h_{left}(x)-h_{right}(x)$

  $h_{left/right}(x)$=height of left/right subtree of x

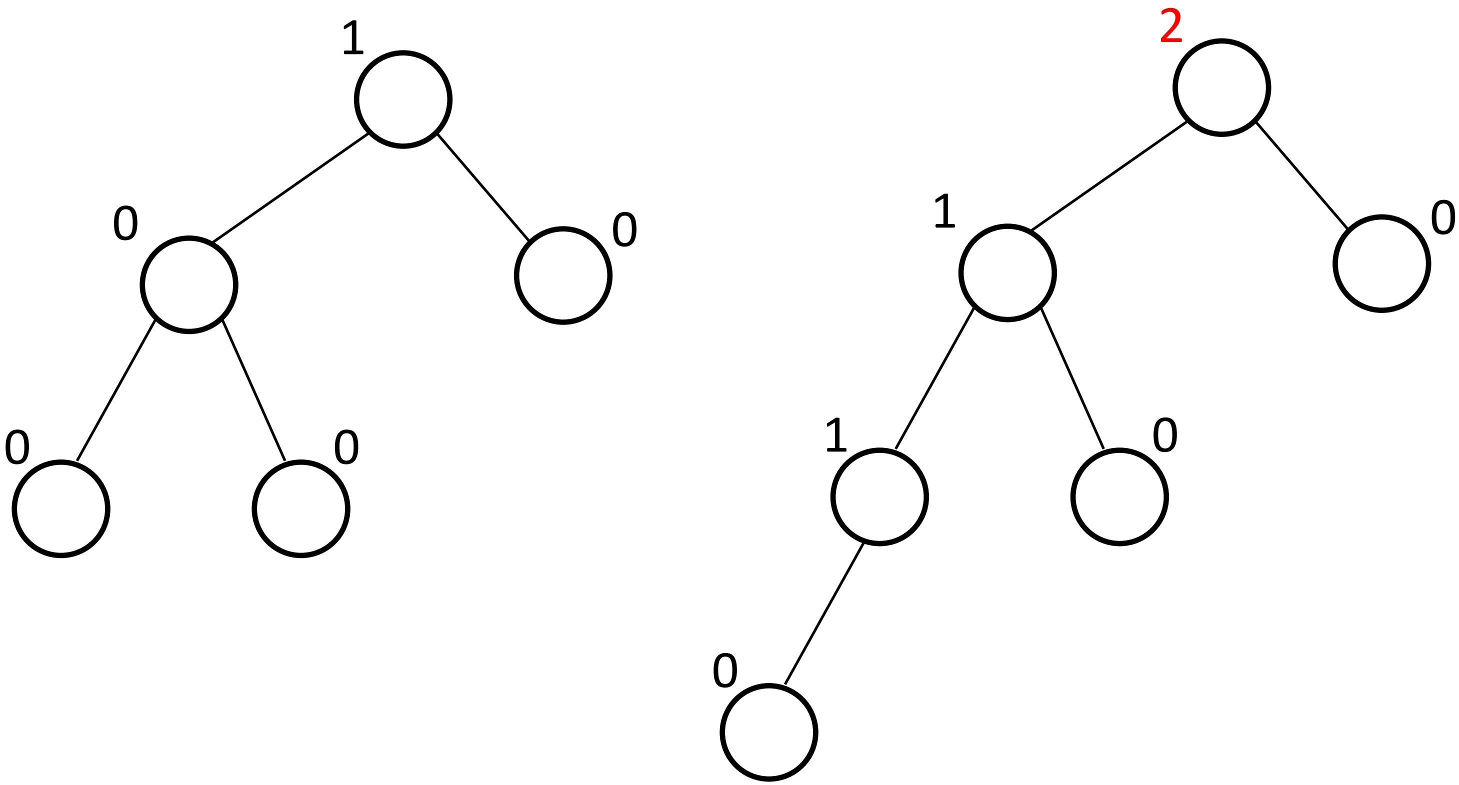
- Balanced → close-to-0 balance factor.

- AVL tree: keeps the balance factor of every node x to be – 1, 0, or 1
  - Height of sibling trees is differ no more than 1

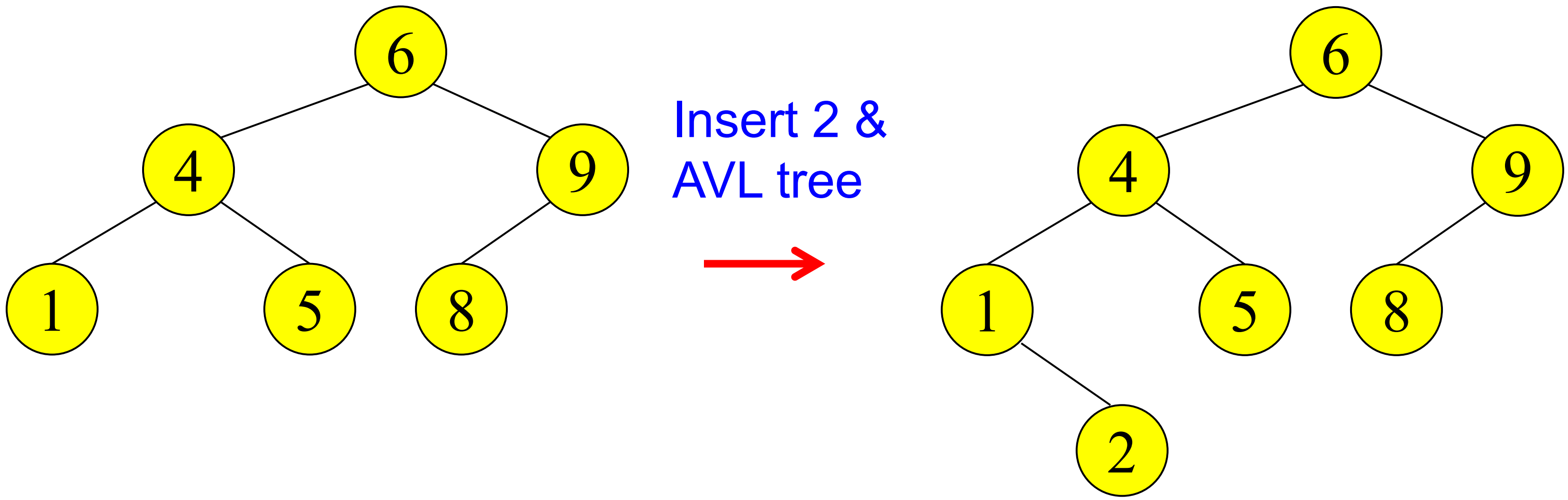
Example of AVL tree



AVL Trees?



Example of AVL tree



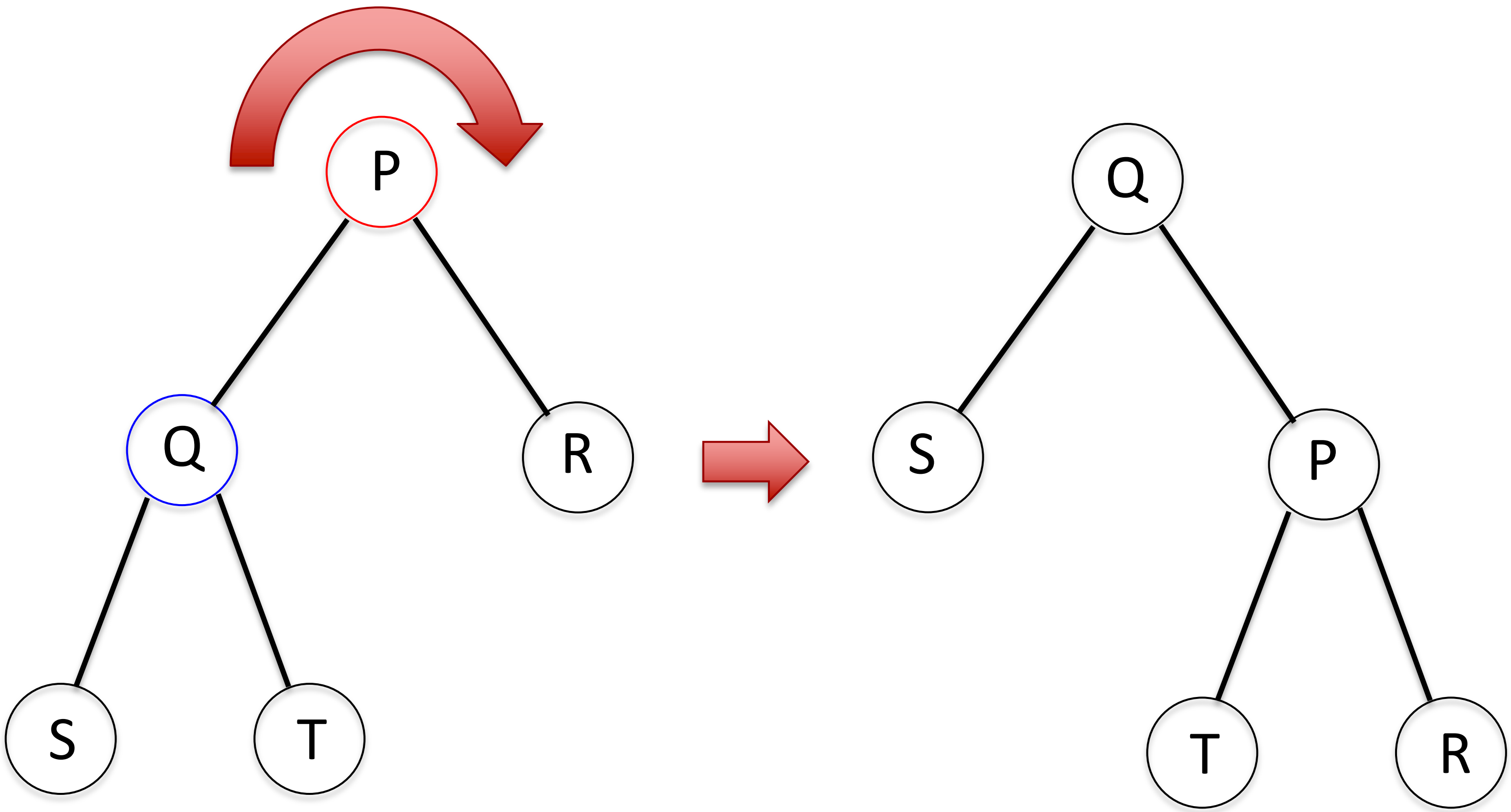
This is not perfectly balanced but height-balanced AVL tree!

Tree Rotation

- Operation on a binary tree that
  - Move one node up and one node down
  - Preserve binary search tree property
- It will also preserve
  - Leaf node order in depth-first search traversal
  - Inorder traversal sequence
    - Does not change binary search tree order
- It is used to
  - Change the shape of the trees, e.g., adjusting height of the trees

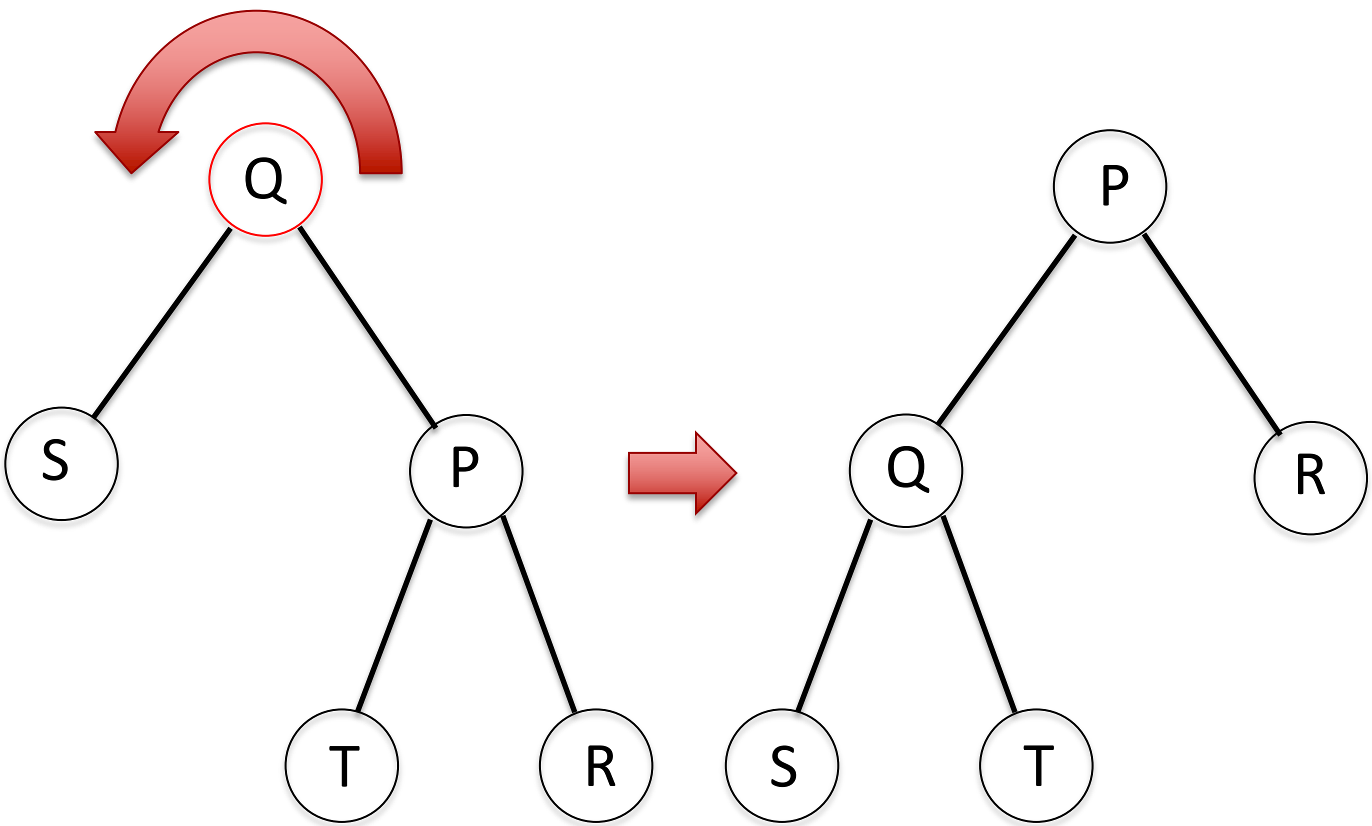
Tree Rotation

- Right rotation on (rooted at) P



Tree Rotation

- Left rotation on (rooted at) Q

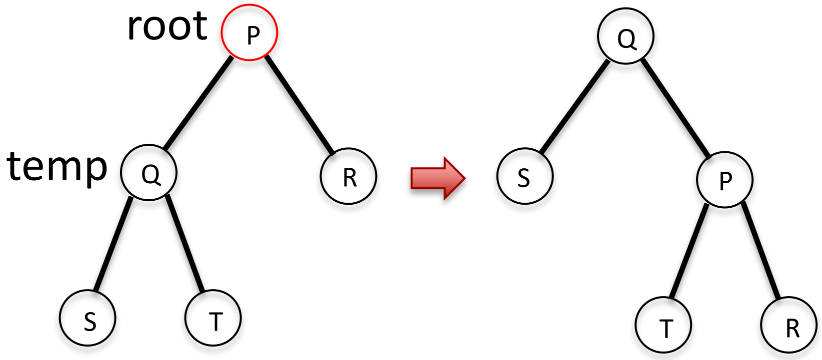


Tree Rotation

- Pseudo code for right rotation

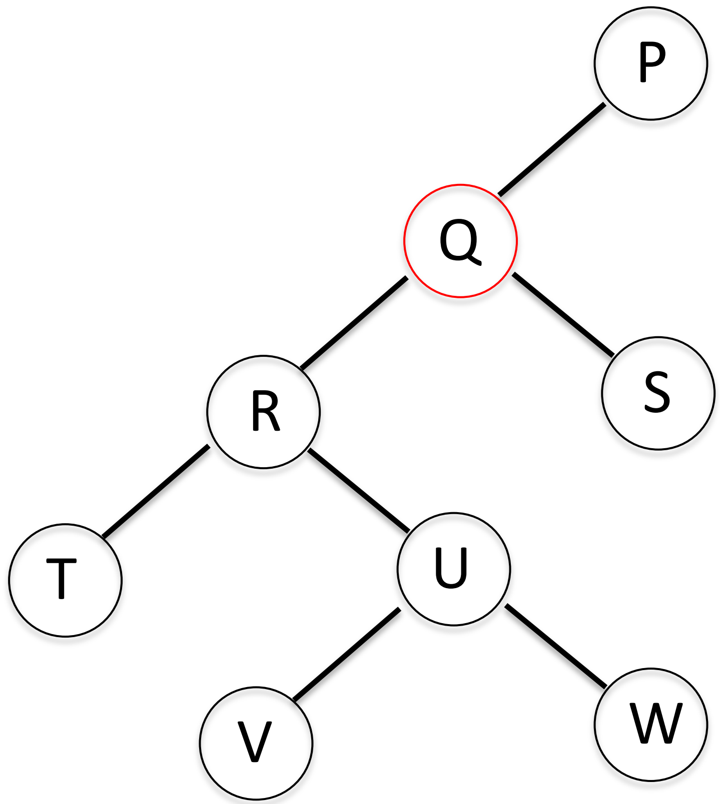
```
temp = root->leftChild
root->leftChild = temp->rightChild
temp->rightChild = root
root = temp
```

?

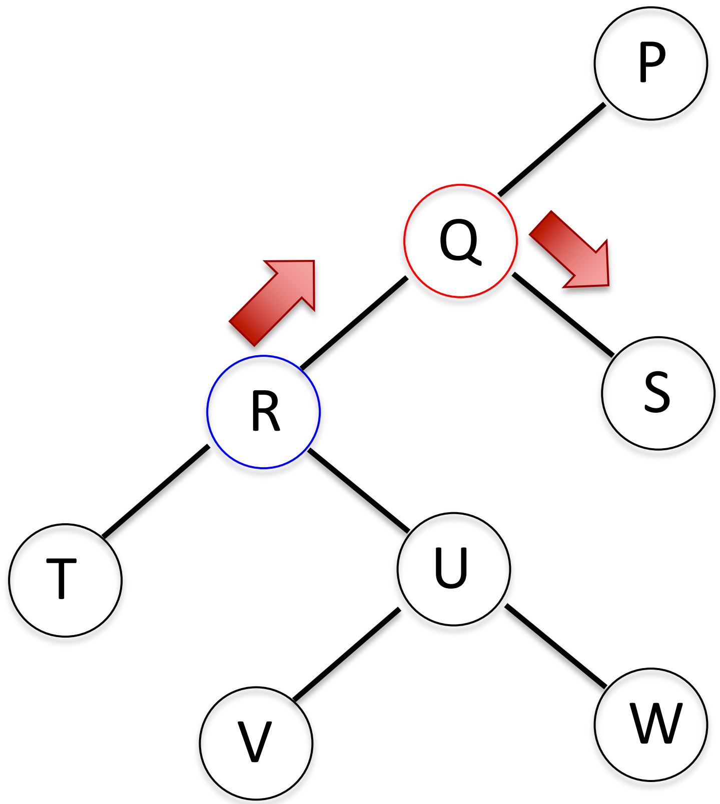


Example

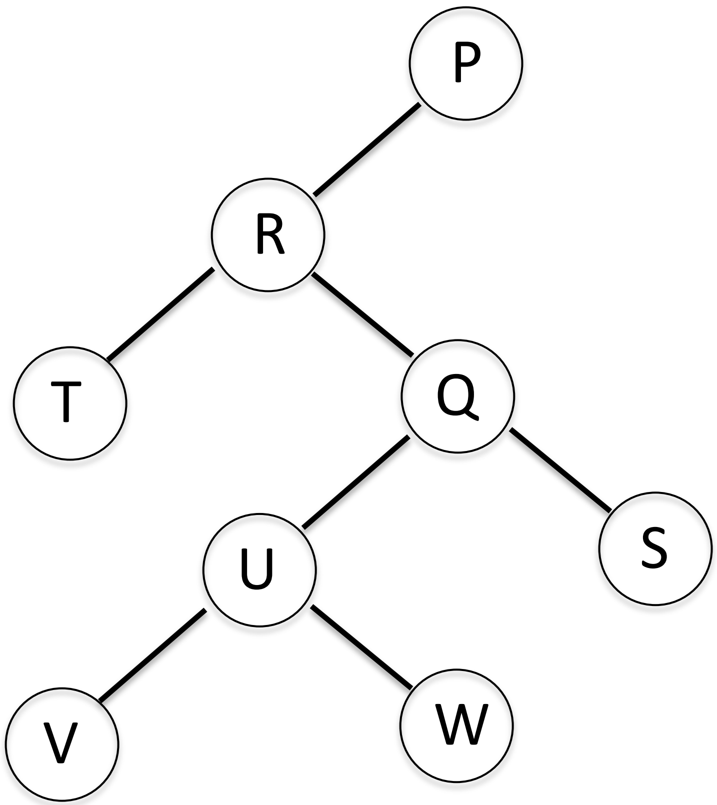
- Right rotate on Q?



Example

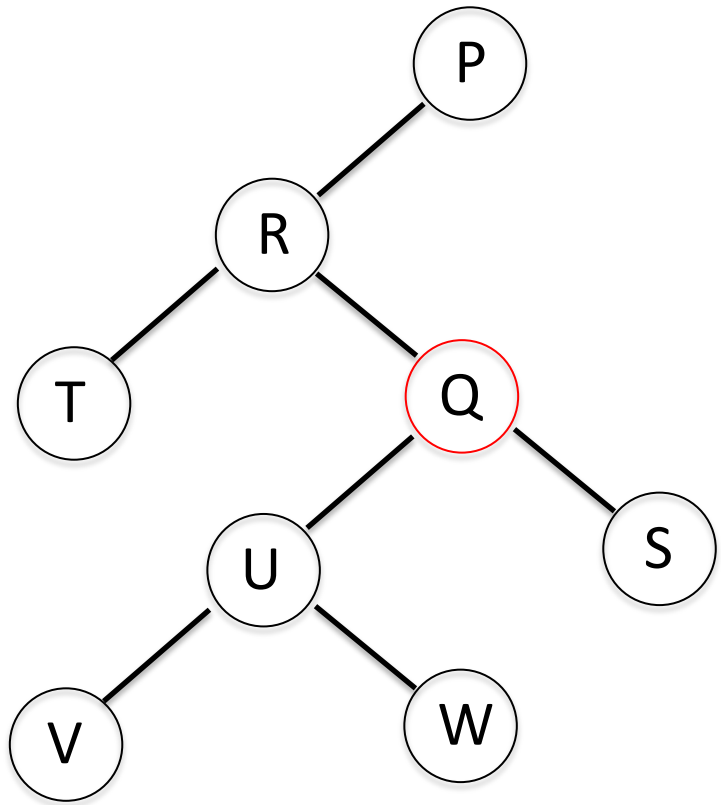


Example

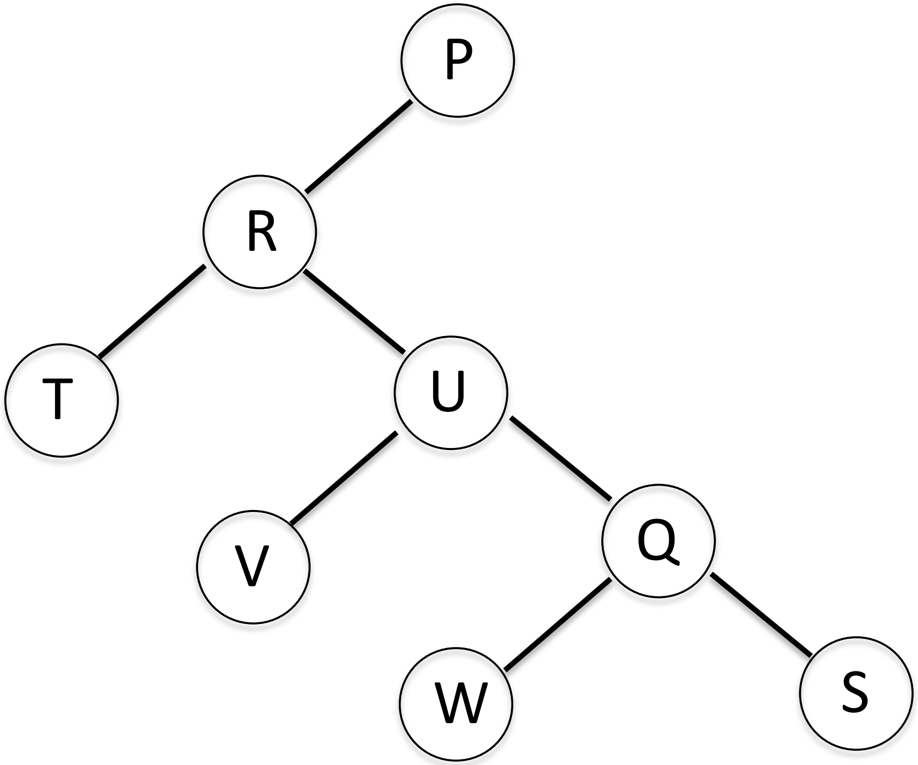


Example

- Right rotate on Q?



Example

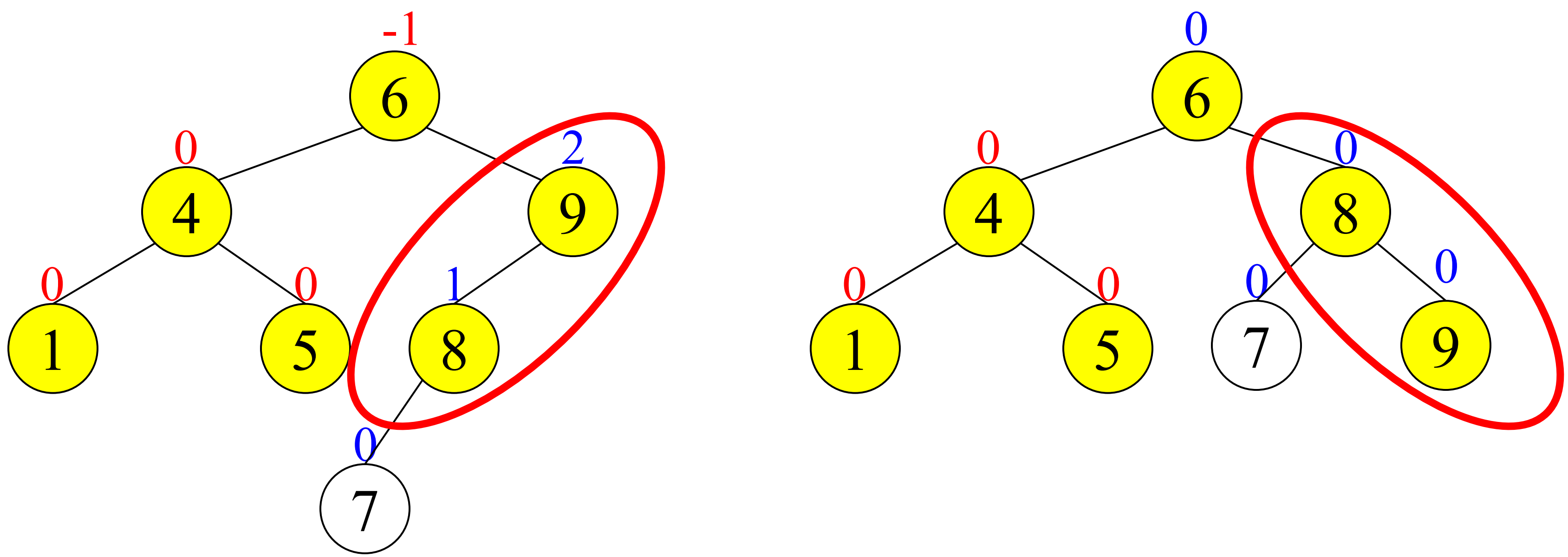


Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  – Every node has -1,0 or 1 before then.
- Only nodes on the path from insertion point to root node have possibly changed in height
  – New ← … ←root
- So after the Insert, go back up to the root node by node, updating heights/balance factor
- If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation Example

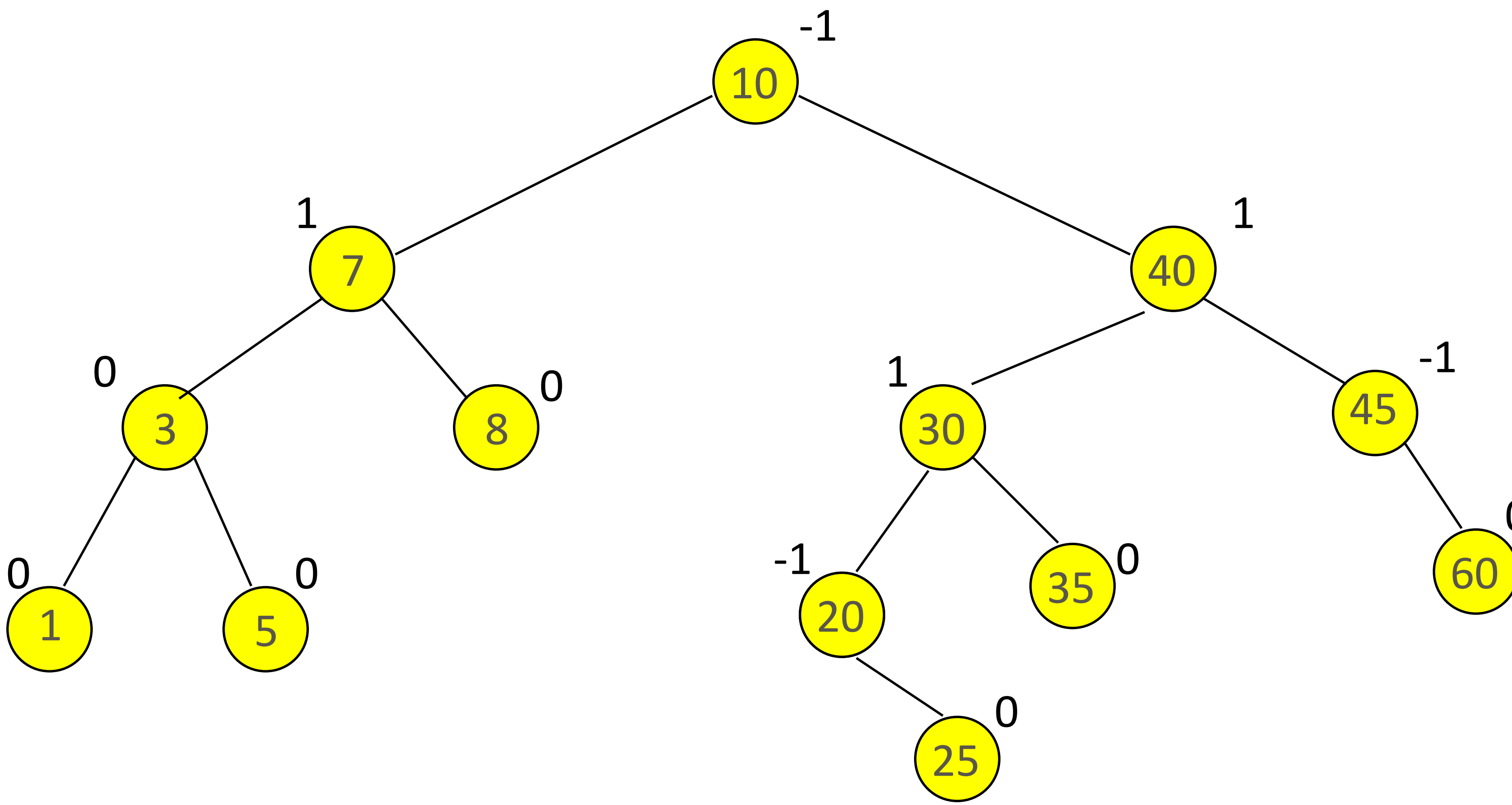
- Right rotation on 9



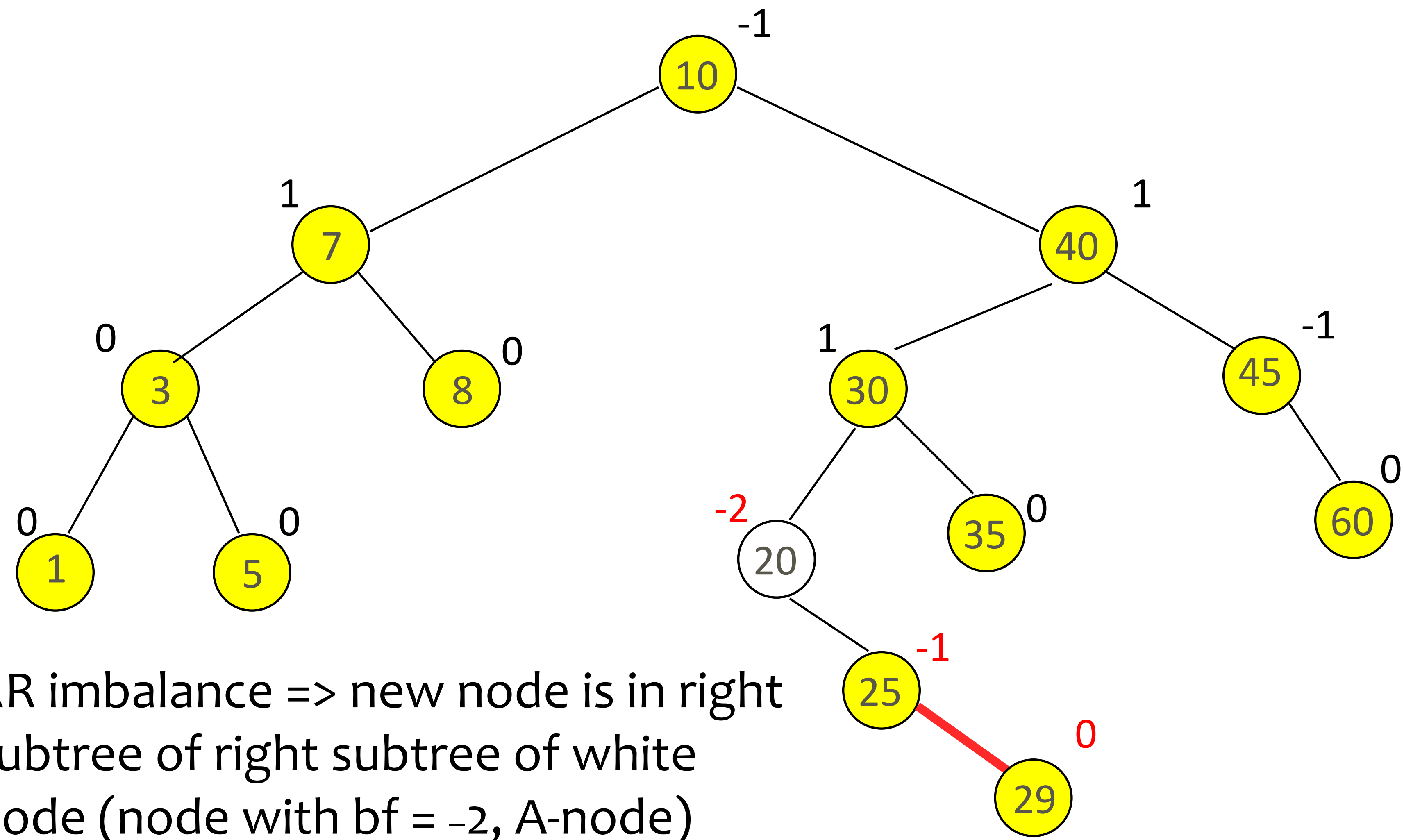
A-Node

- Let A be the nearest ancestor of the newly inserted node whose balance factor becomes +2 or –2 following the insert
- Balance factor of nodes between new node and A is 0 before insertion
- Rotation must be done on A-node

Insert 29

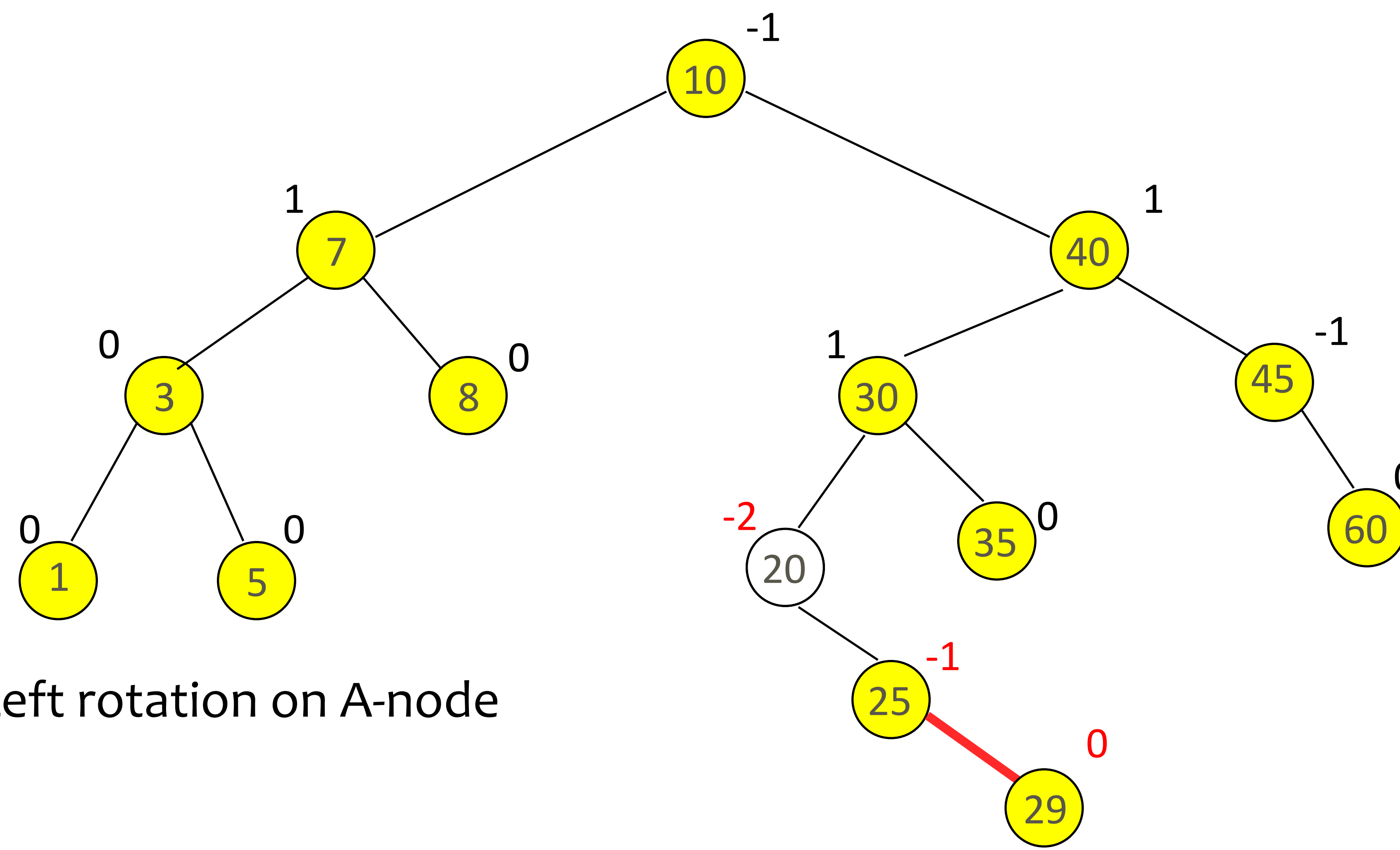


Insert 29



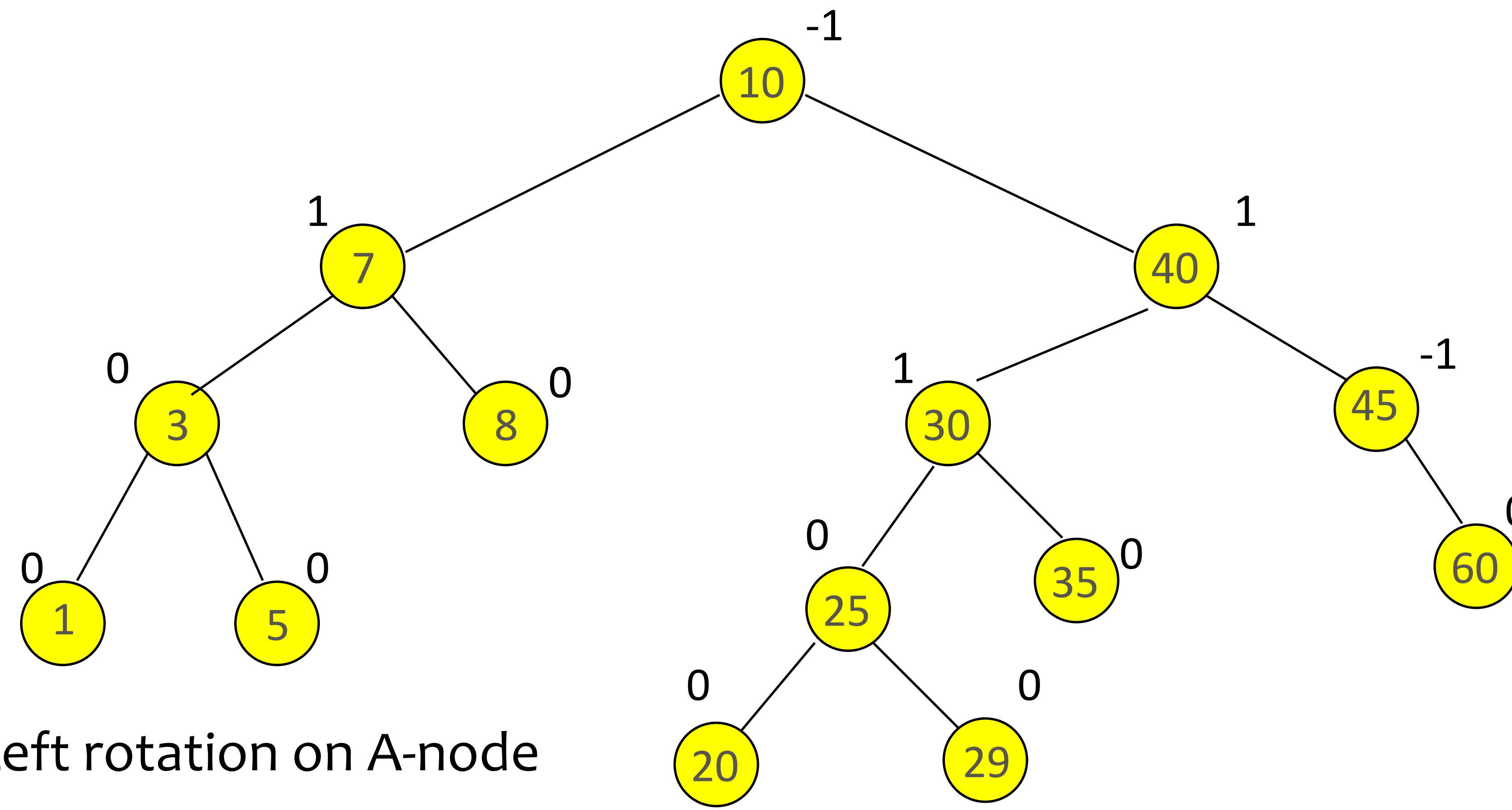
RR imbalance => new node is in right subtree of right subtree of white node (node with bf = –2, A-node)

Insert 29



Left rotation on A-node

Insert 29



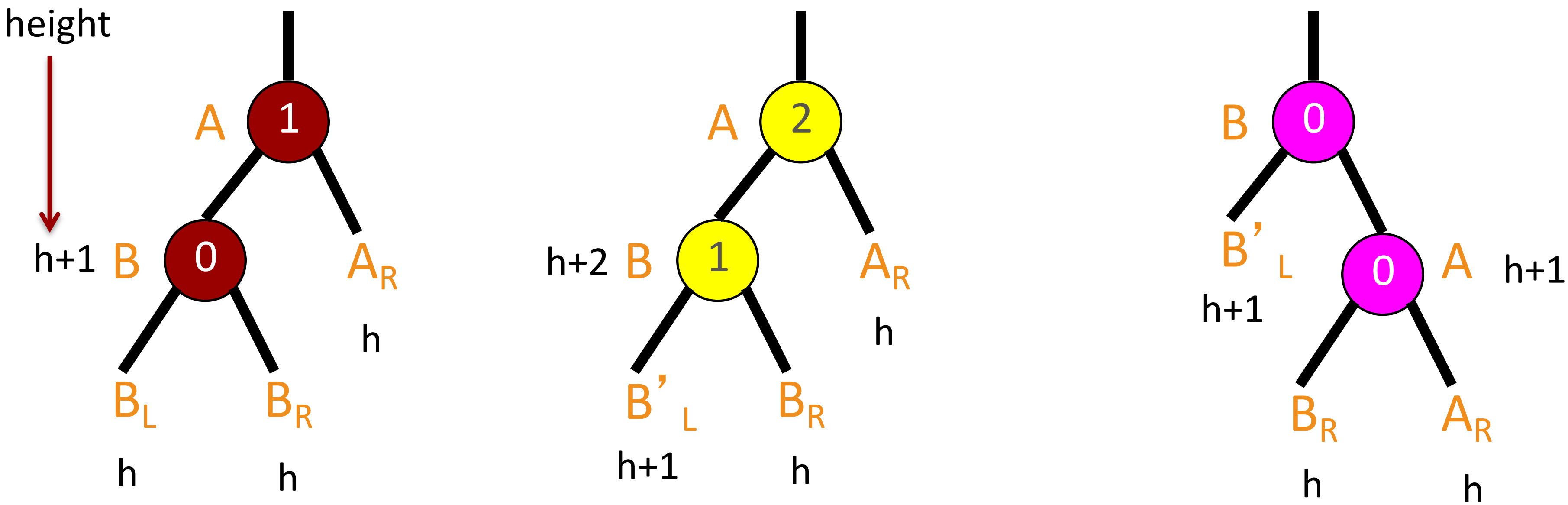
Left rotation on A-node

Imbalance Types

- RR … newly inserted node is in the right subtree of the right subtree of A
- LL … left subtree of left subtree of A
- RL… left subtree of right subtree of A
- LR… right subtree of left subtree of A

LL Rotation

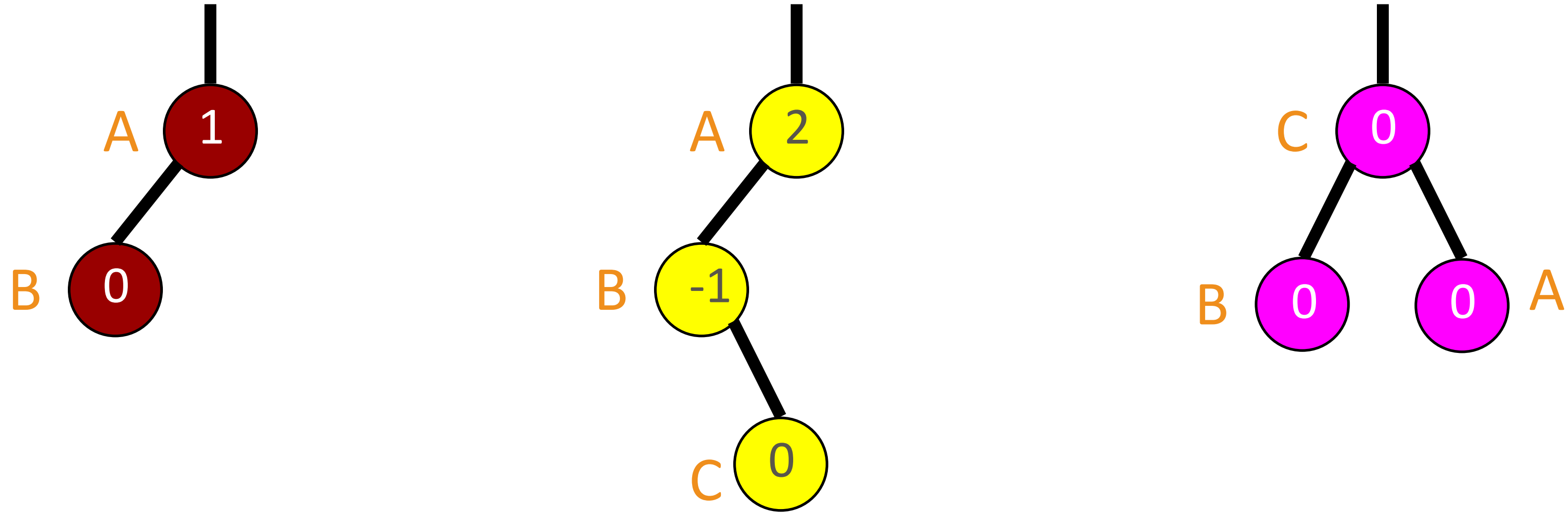
- Right rotation on A



Before insertion After insertion After rotation

LR Rotation (case 1)

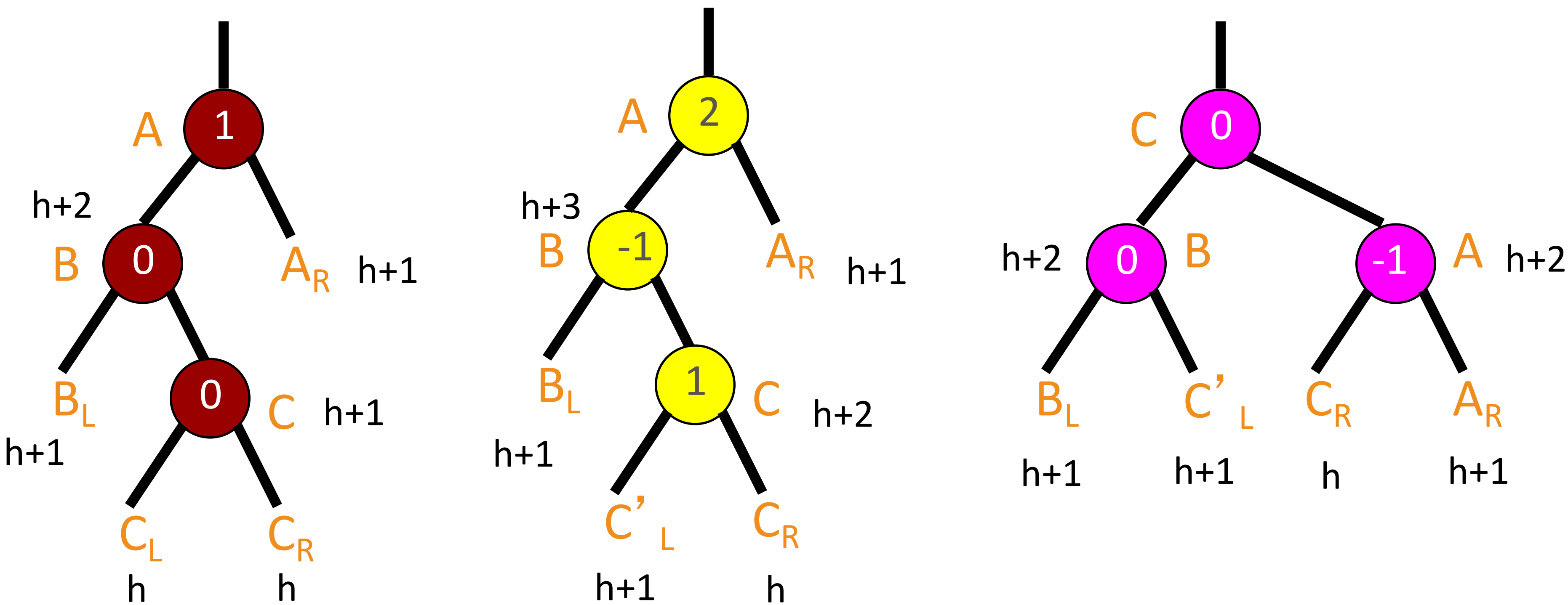
- Left – Right rotations
  - Left on B, right on A



Before insertion After insertion After rotation

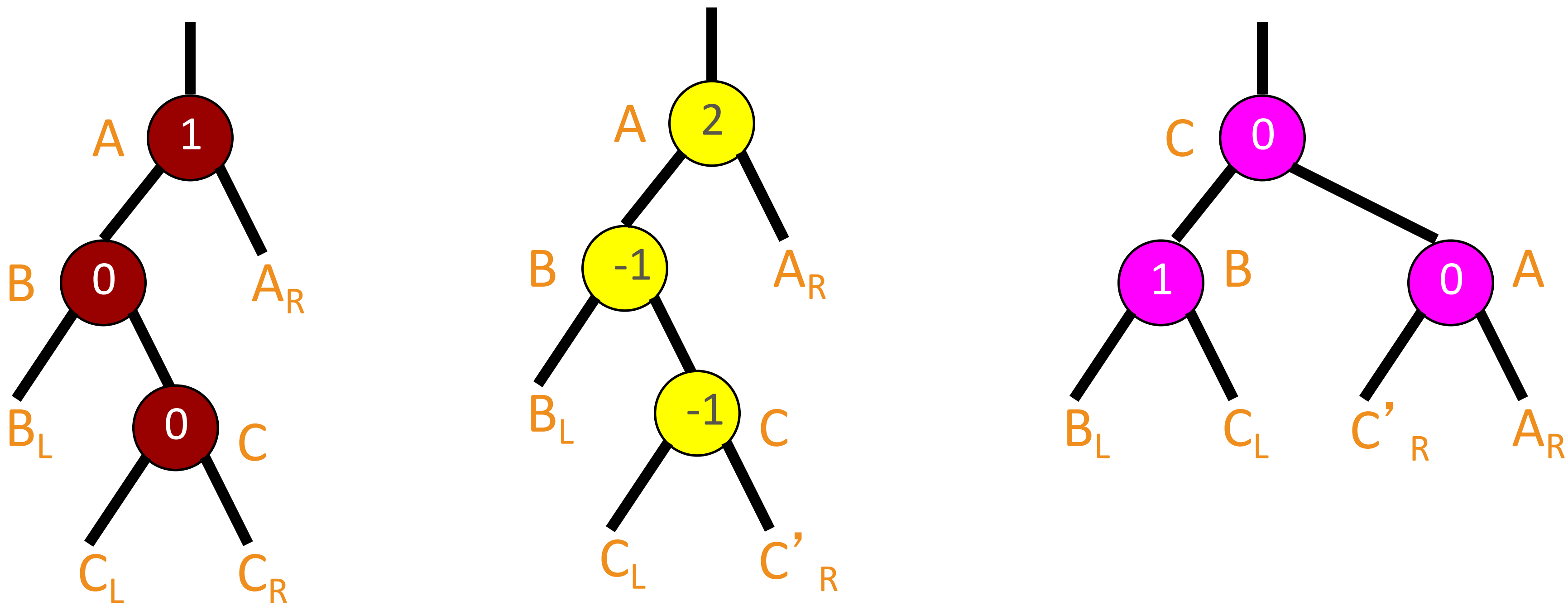
LR Rotation (case 2)

- Left – Right rotations
  - Left on B, right on A



LR Rotation (case 3)

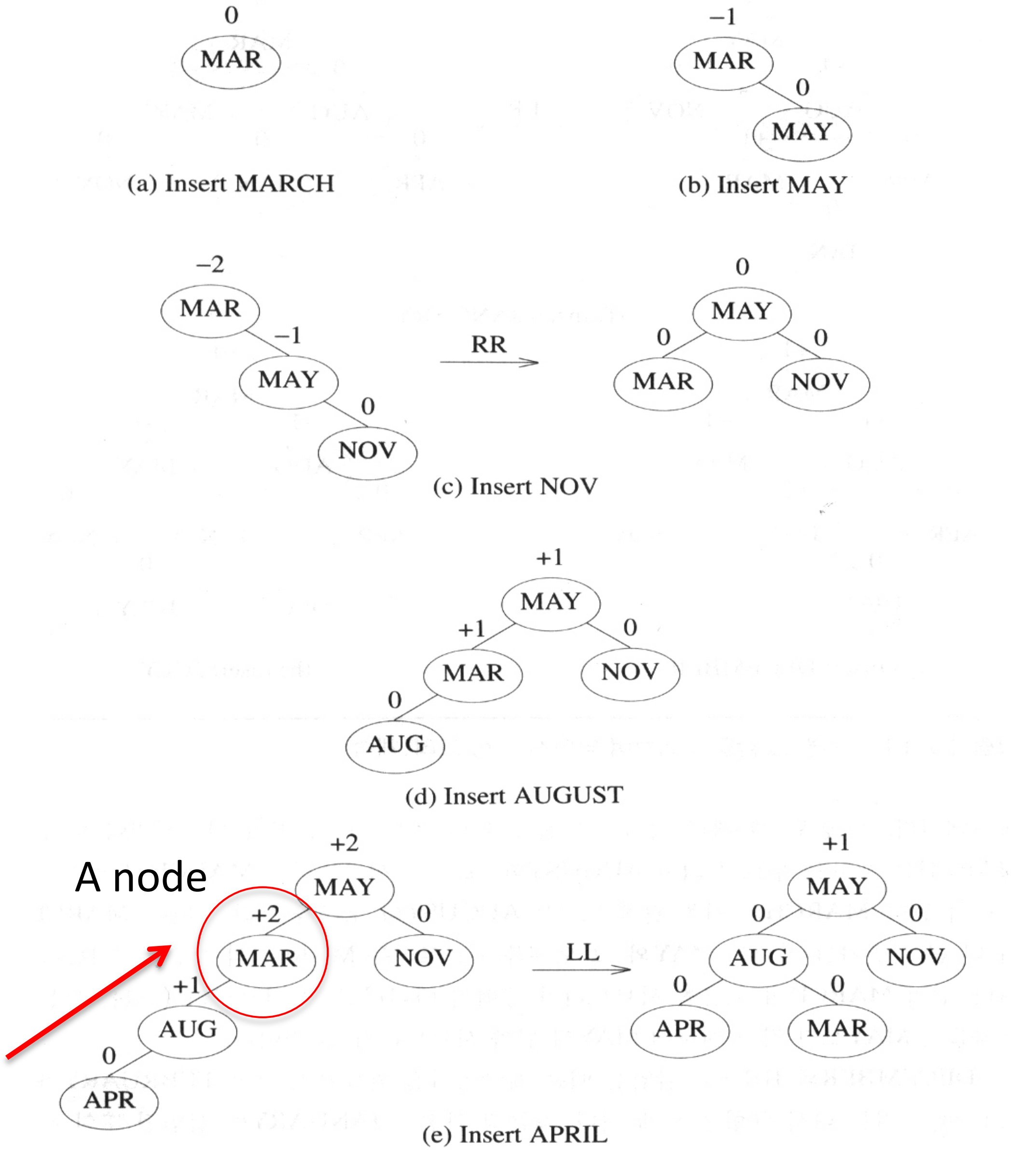
- Left – Right rotations
  - Left on B, right on A

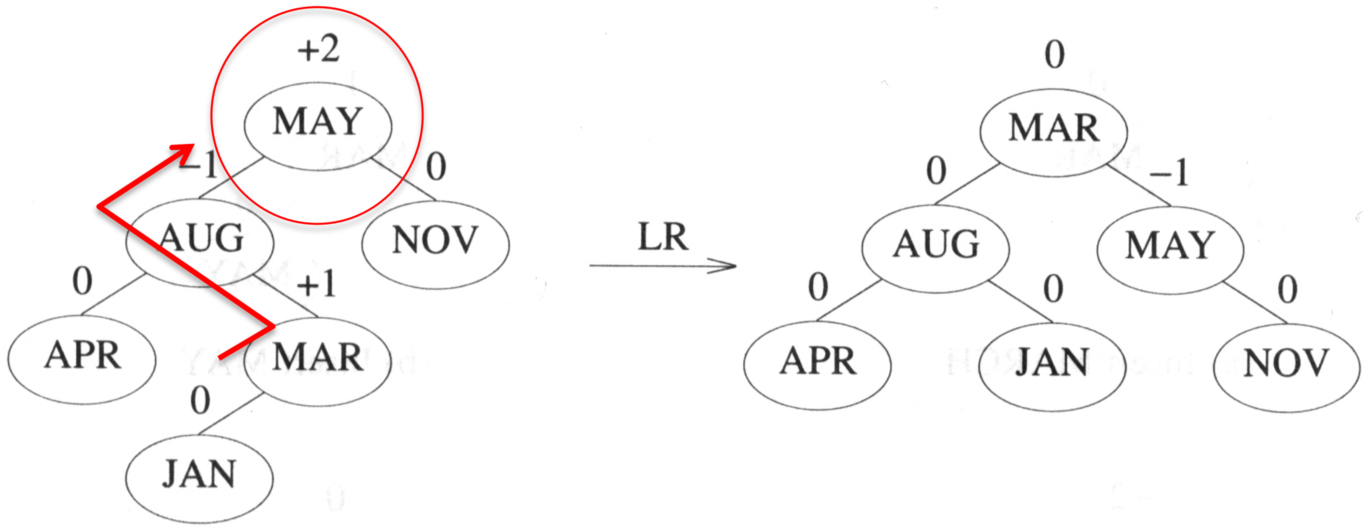


Single & Double Rotations

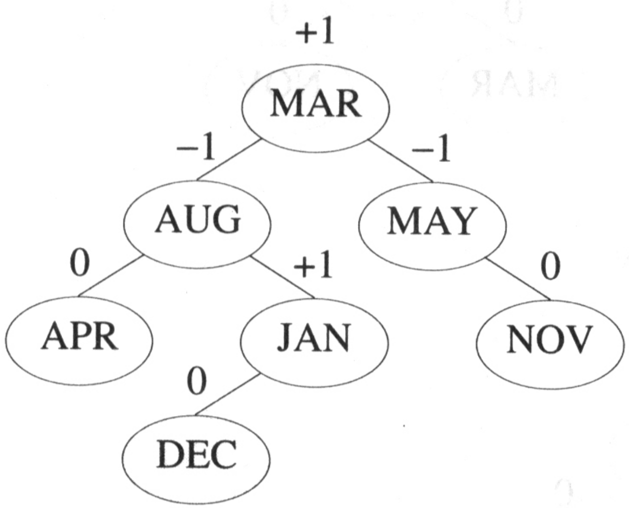
- Single rotation
  - LL and RR
- Double rotation
  - LR and RL
  - LR is RR followed by LL
  - RL is LL followed by RR

Alphabetical order

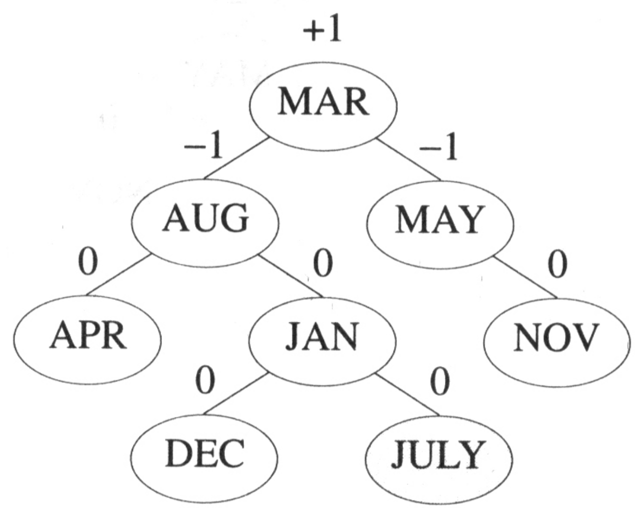




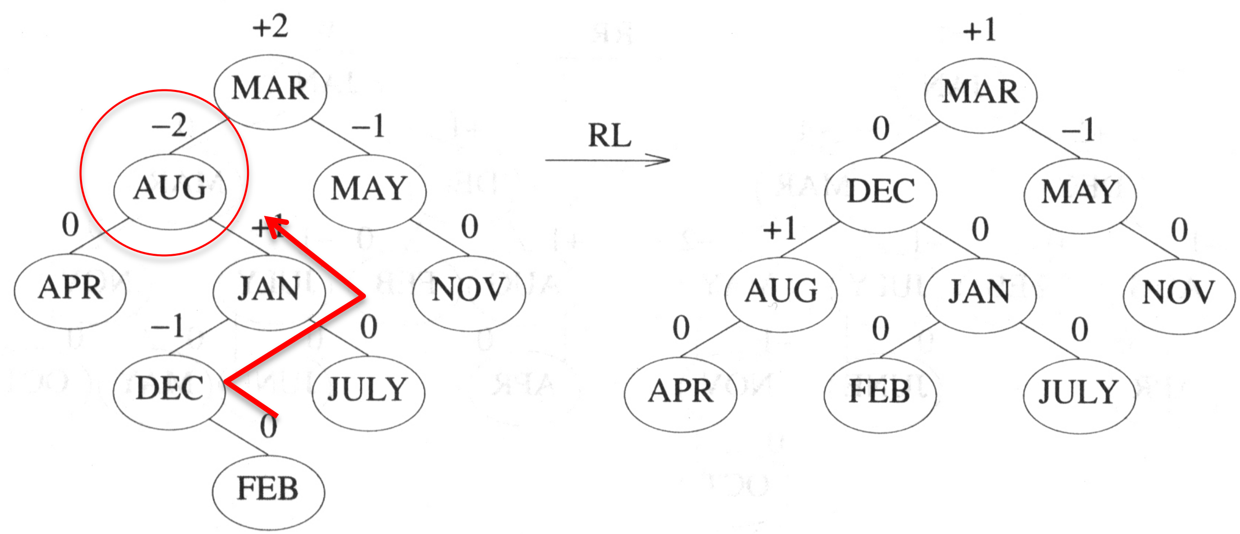
(f) Insert JANUARY



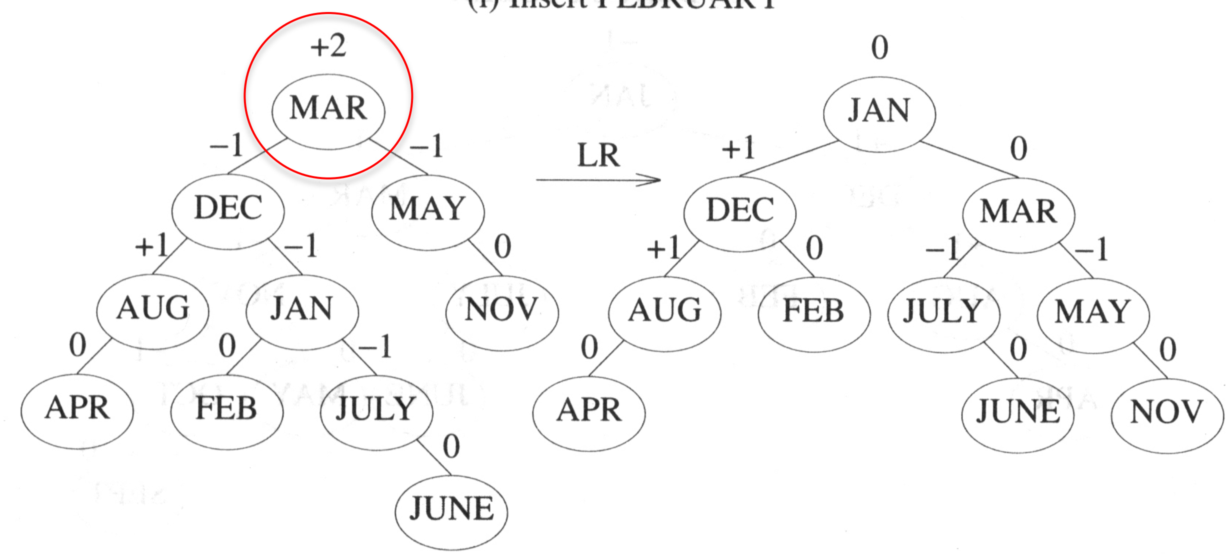
(g) Insert DECEMBER

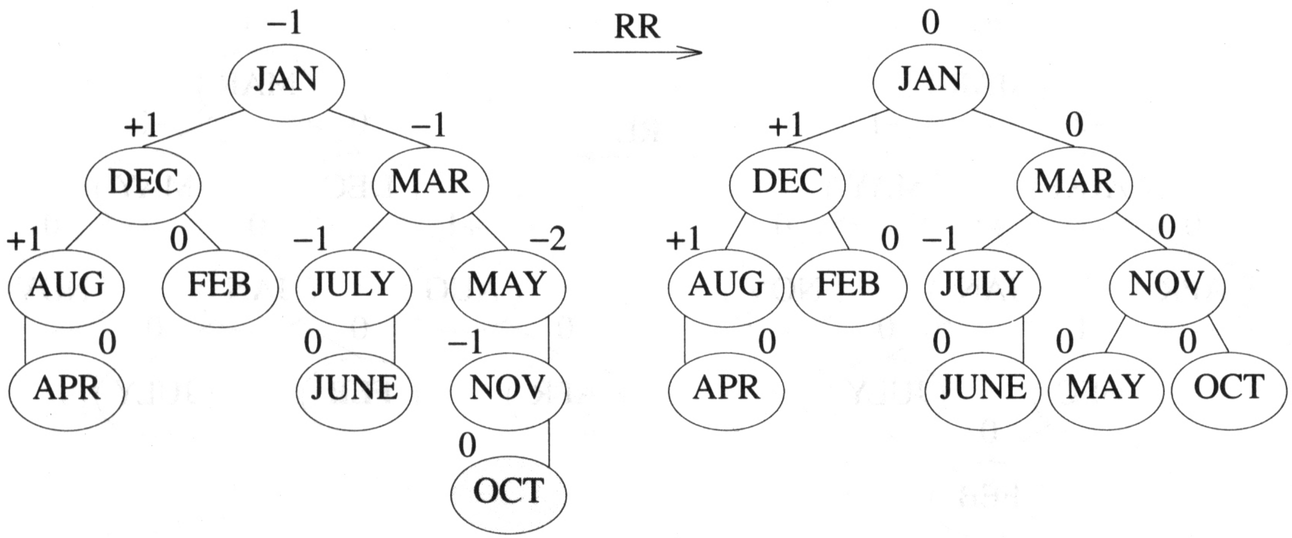


(h) Insert JULY

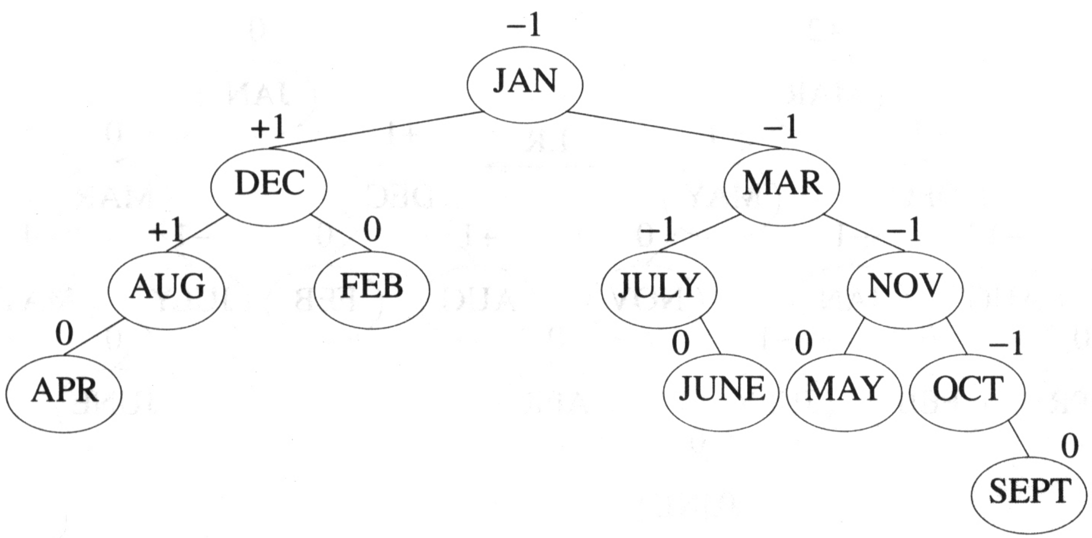


(i) Insert FEBRUARY





(k) Insert OCTOBER



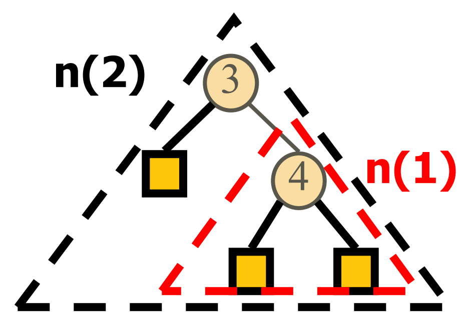
(l) Insert SEPTEMBER

Height of an AVL Tree

- Fact: The height of an AVL tree storing n keys is O(log n).
- Proof: Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

  $n(h) > 2^i n(h-2i)$
- Let i=h/2-1, then we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)



Number Of Rebalancing Rotations

- Insert : at most 2 rotations
  - 2 rotations reduce the height of the A node.
  - Ancestors of the A node has -1, 0 or 1 after the rotations.
- Delete : at most O(log n) rotations
  - May need to rotate every node between the deleted root node.

Discussion

- AVL trees manage strict height-balanced structure
  - Height is bounded by O(log n)
    - Search, insertion, deletion are O(log n)
  - Fast for lookup intensive problems
  - Insert, delete can be slower than lookup

Questions?

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY