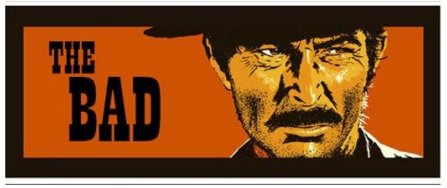
COMP20007 Workshop Week 11

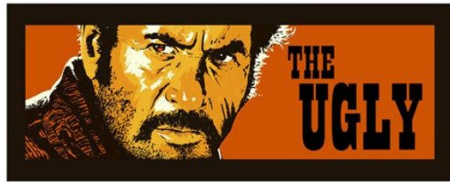
	Preparation: - *have draft papers and pen ready* - open Ed.Assignment2
1	Why BST, AVL, 2-3 Tree?
2	BST: Rotation, Balance factor: Q 11.1, 11.2
3	AVL Tree: Concepts, Insertion, Deletion: Q 11.3, 11.4
4	2-3 Tree: Concepts, Insertion: Q 11.5
5	[Homework, not examinable?] 2-3 Deletion, Deletion - Q 11.6
	B-tree?
LAB	
	Assignment 2
	Revision: Questions for previous week materials

Why AVL, 2-3 Trees?

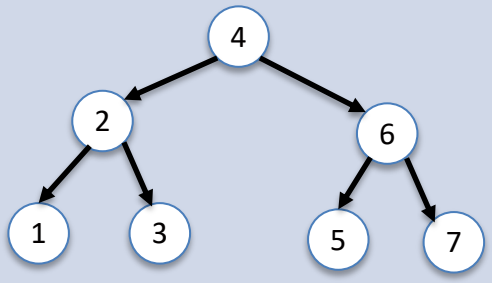
BST efficiency depends on the order of input data

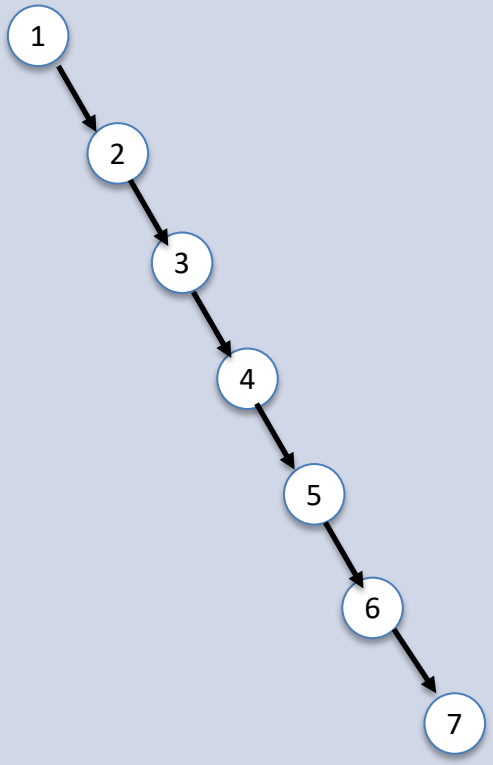


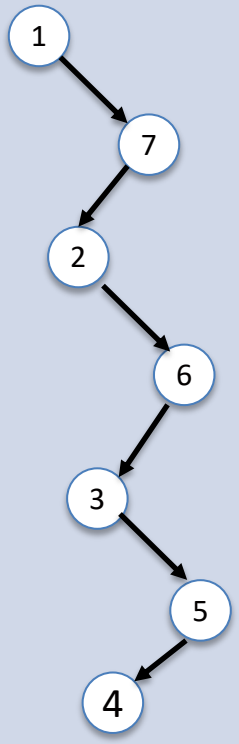
AND



4 6 2 1 3 7 5	1 2 3 4 5 6 7	1 7 2 6 3 5 4







Want The Good, no matter what's the data input order? Use AVL (or …)!

Good-Bad_Ugly Picture Source: https://www.pinterest.com.au/pin/170573904624610413/

Using Rotations to rebalance a BST

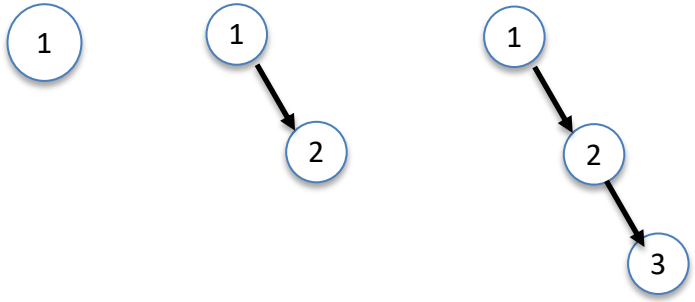
At the start: an empty BST is balanced

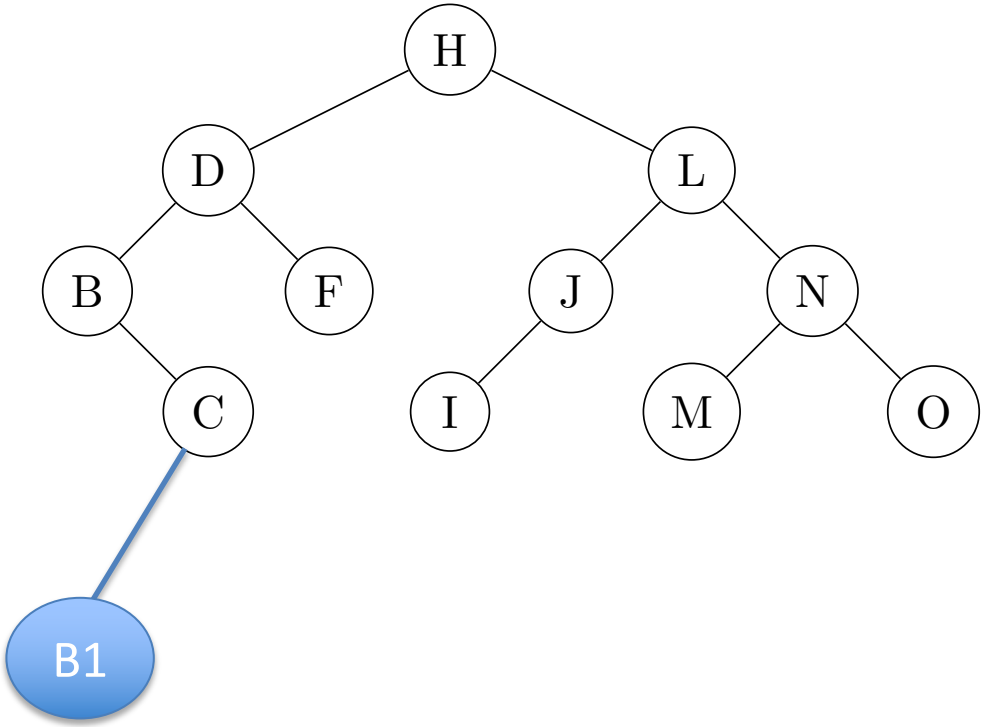
Problem: After a insertion/deletion, the resulted tree might become unbalanced

Approach: use Rotations to rebalance.

To rotate:

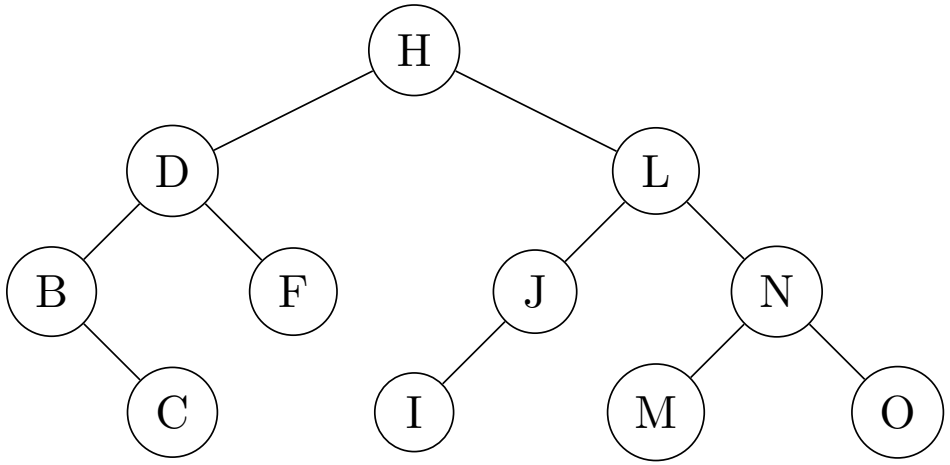
When? What? How?





What's a balanced BST?

Q 11.2: A node's '*balance factor*' is defined as the height of its right subtree minus the height of its left subtree. Calculate the balance factor of each node in the following binary search tree.



Note on balance factor (BF) definition

- Popular (as in lectures):
  BF= height(T.left) - height(T.right)
- Option 2 (here):
  height(T.right) - height(T.left)

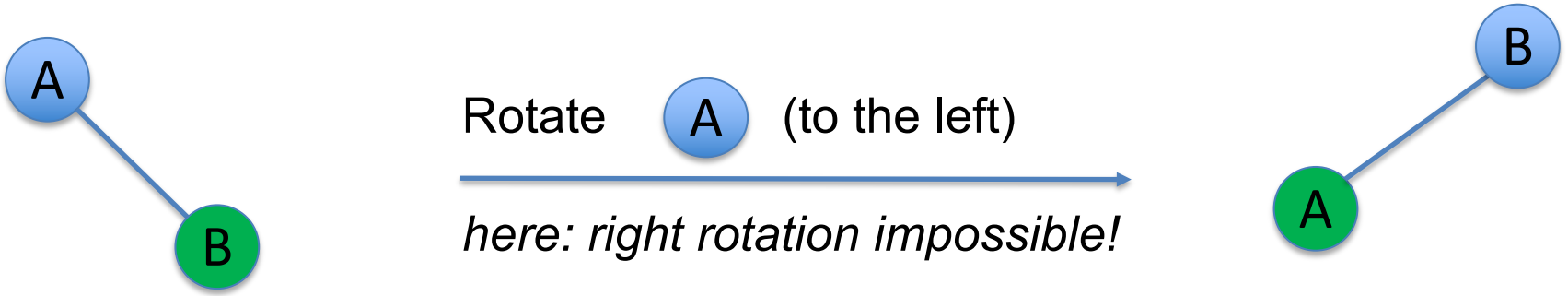
Any of them is OK, but needs **consistency**!
When manually operating we just need absolute value of the height difference.

Balanced tree = when the balance factor of each node is 0, -1, or +1
= for each node, the difference of subtree heights is at most 1

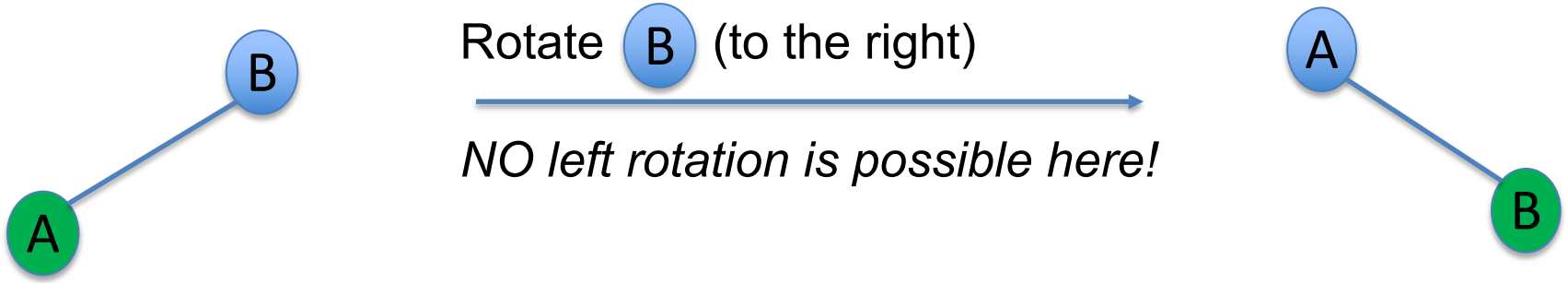
BST: what's a rotation

A *rotation reverses* the parent-child relationship of a parent and a child of it in a BST.

left rotation: rotate parent down to the left (parent becomes the left child)



right rotation: rotate parent down to the right (to become the right child)

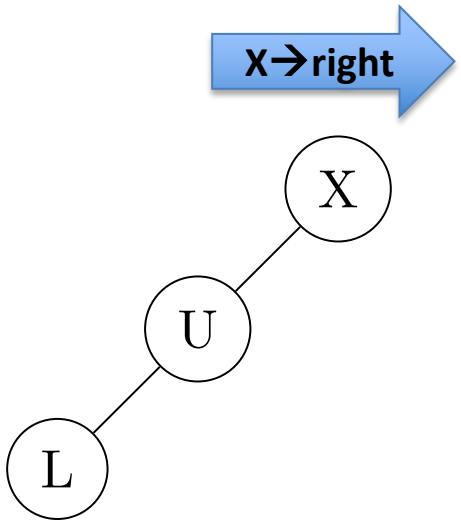


Note: we say that we **rotate the parent node,** although we rotate both parent and child.

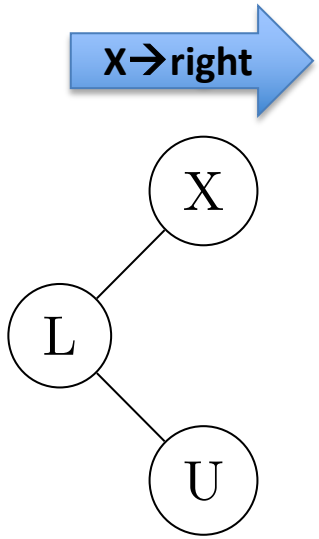
Q 11.1 [class]: Rotation

In the following binary search trees, rotate the ‘X’ node to the right (that is, rotate it and its left child). Do these rotations improve the overall balance of the tree?

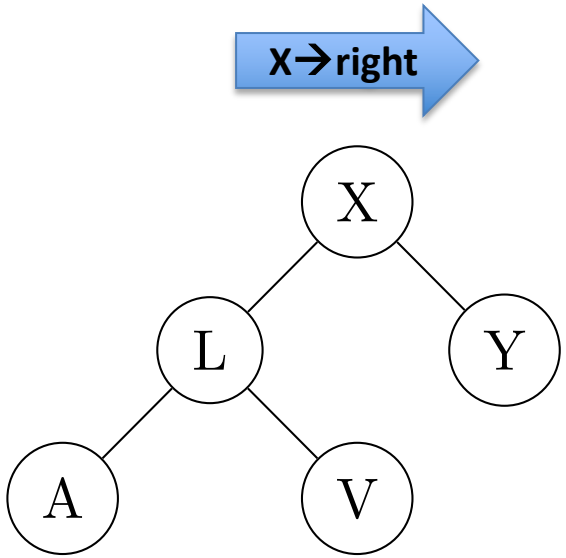
(c)



(b)



(a)

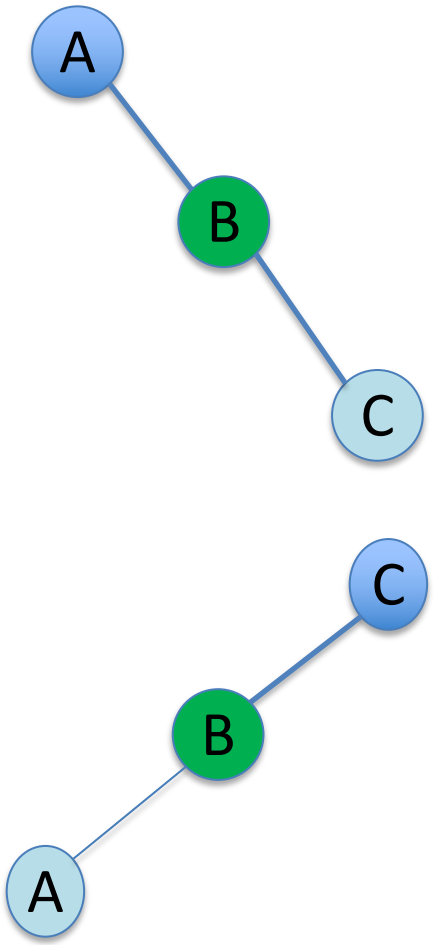


Recall: Only 2 types of rotations: *Right Rotation* (a node and its left child), and *Left Rotation* (a node and its right child)

AVL= balanced BST
using rotation to rebalance when the BST becomes un-balanced

AVL: Two Basic Rotations: 1) Single Rotation

Applied when an unbalanced AVL subtree (or tree) is a ”stick”:



Questions we should ask ourselves when doing a rotation:

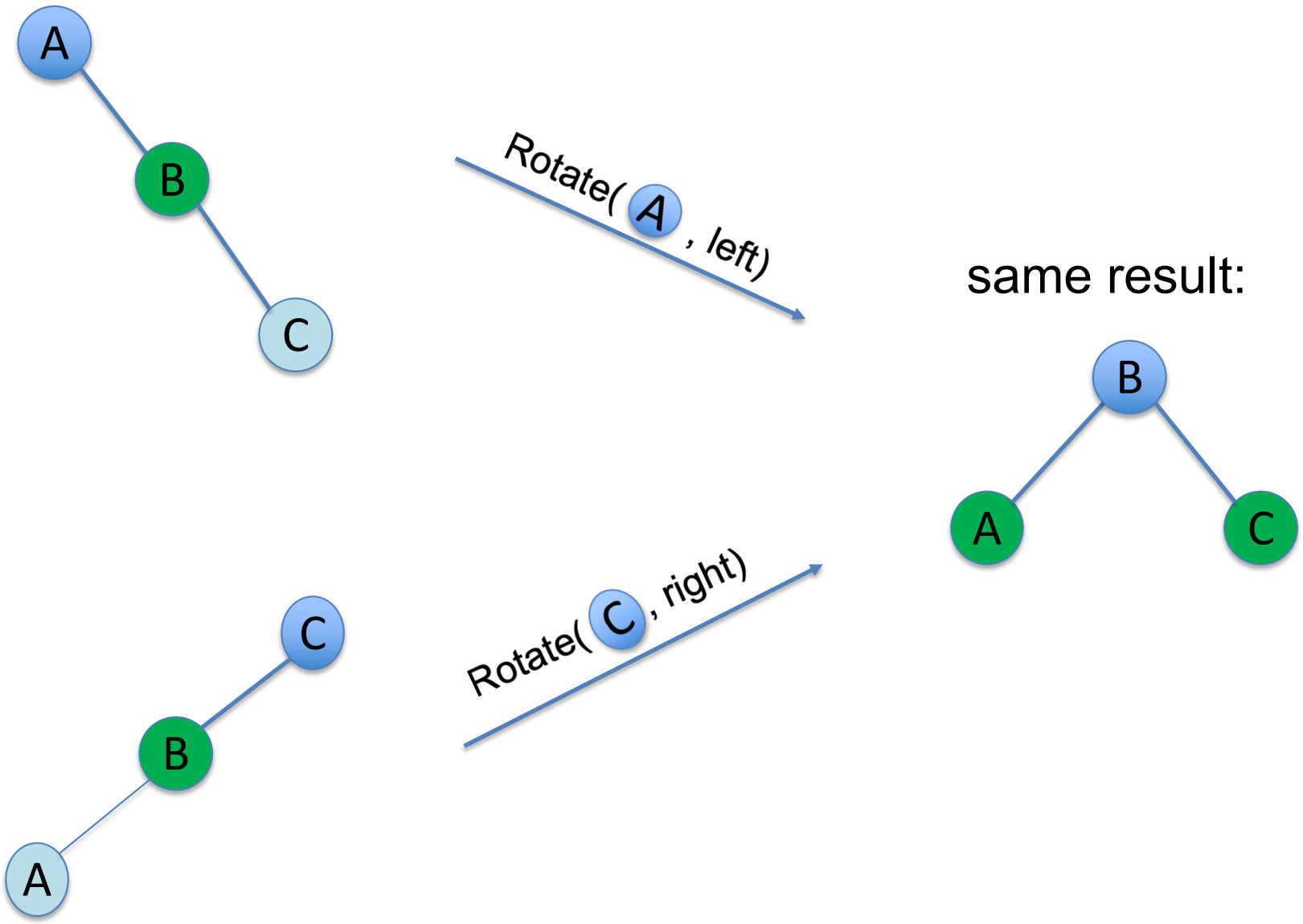
- Which subtree is unbalanced? Which node is the *unbalanced root* of that unbalanced subtree?
- Which rotation should be done for re-balancing?

HERE: we have an un-balanced stick:
(unbalanced root)-child-grandchild

HOW:
→ Rotate the root down and hence balance the stick

AVL: Two Basic Rotations: 1) Single Rotation

Applied when an AVL (subtree) is a ”stick”. Two cases:



The “Rotation” also includes the re-arrangement of all involved nodes. In particular:

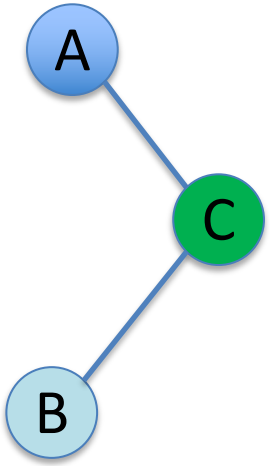
- not changing the children of the 3 involved nodes if possible,
- or, if otherwise, insert the “abducted” child to the new root (ie. root “B” in the picture)

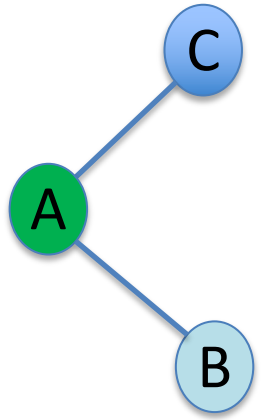
Examples: ...

AVL: Two Basic Rotations: 2) Double Rotation

Applied when an unbalanced 3-node AVL subtree has a non-stick (that is, zig-zag) form.

Two cases: (a) (b)





We do 2 rotations to re-balance the non-stick unbalanced AVL.

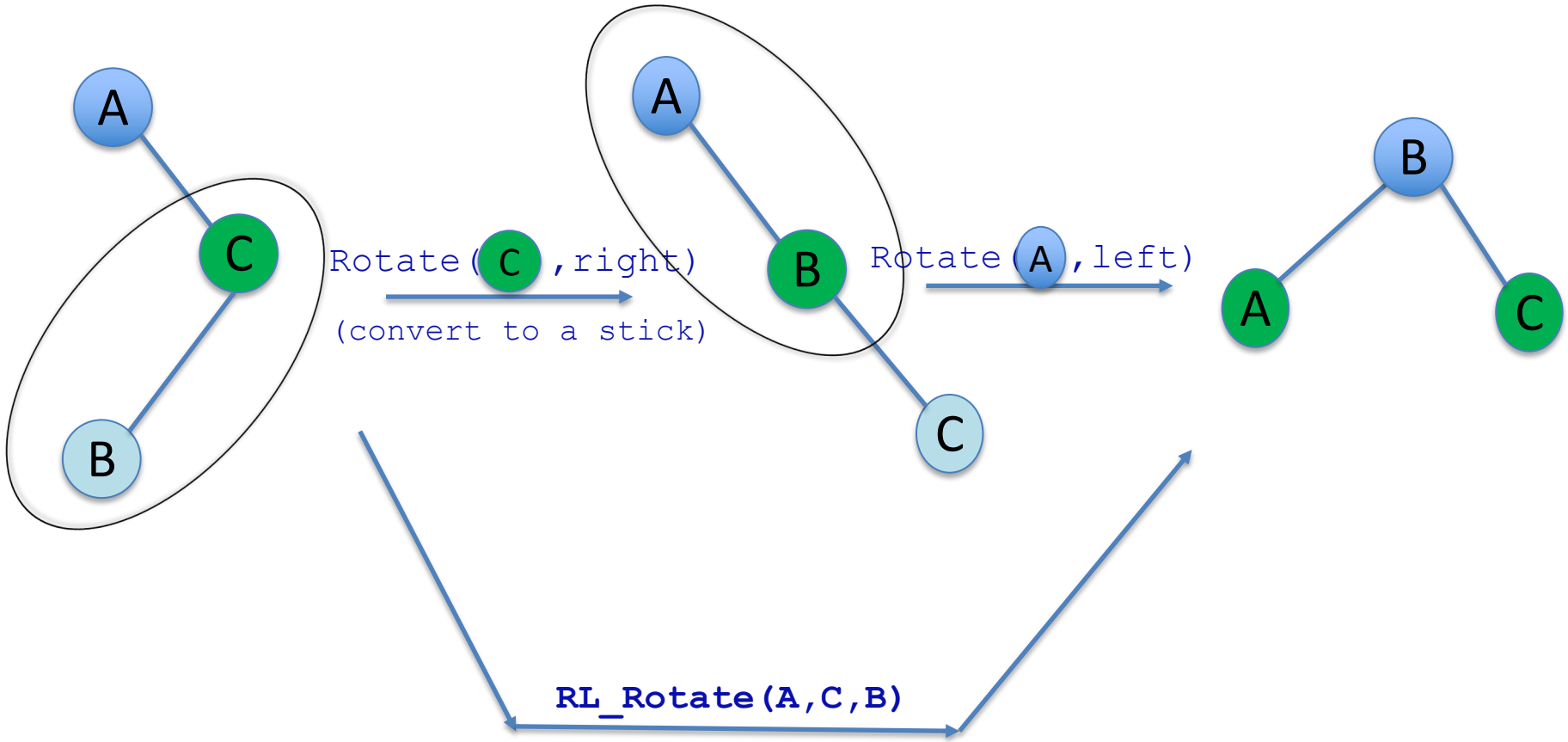
Rotation1:

- Rotate the **child** of the unbalanced root and turn the tree to a stick

Rotation2:

- Rotate the **unbalanced root** of the new stick.

Double Rotation Example: RL rotation



Do it Yourself: Perform `LR_Rotate(C,A,B)` for the other case of the previous page

AVL: Using Rotations to rebalance AVL, Q 11.3

Problem: When inserting/deleting node, AVL might become unbalanced

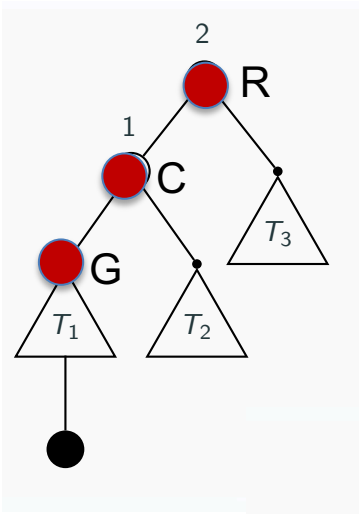
Approach: Rotations (Rotate WHAT?, and HOW?)

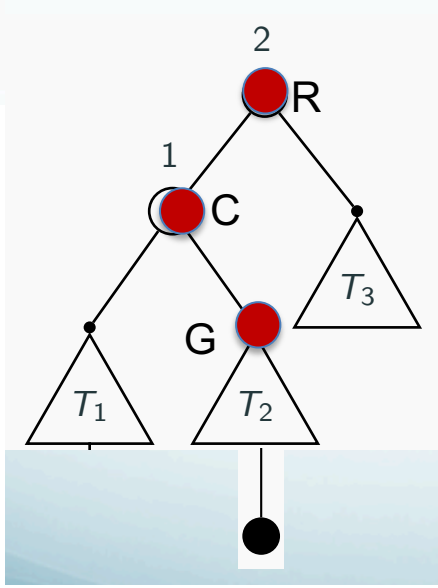
Rotate WHAT?

- From bottom, walk up, find the *lowest* subtree R which is unbalanced, R is the unbalanced root

HOW

- Consider *the last 3 nodes* G, C, R in the above walking
- Apply a single rotations if R→C→G is a stick, double rotation otherwise





Insert the following keys into an initially-empty AVL Tree.

Class example:

20 10 5 15 30 17 8 2 12 4

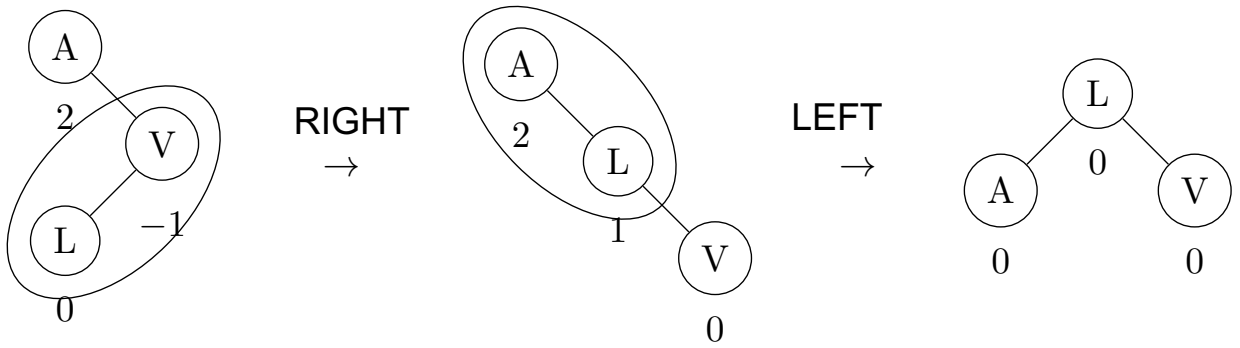
Q11.3 [*group/individual*]:

A V L T R E X M P

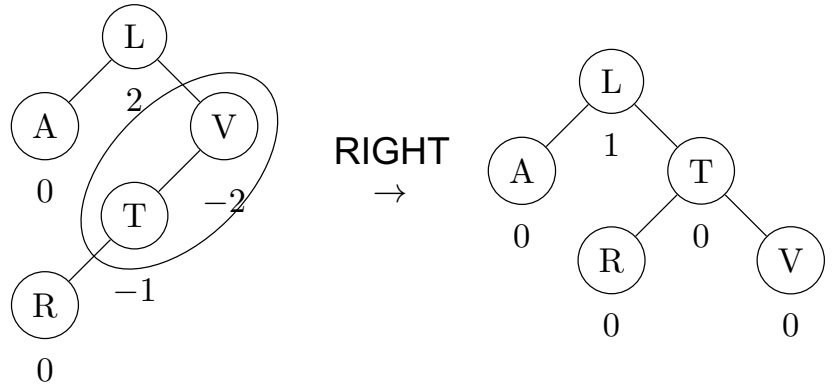
Q 11.3: Check your solution

Insert the following letters into an initially-empty AVL Tree: A V L T R E X M P

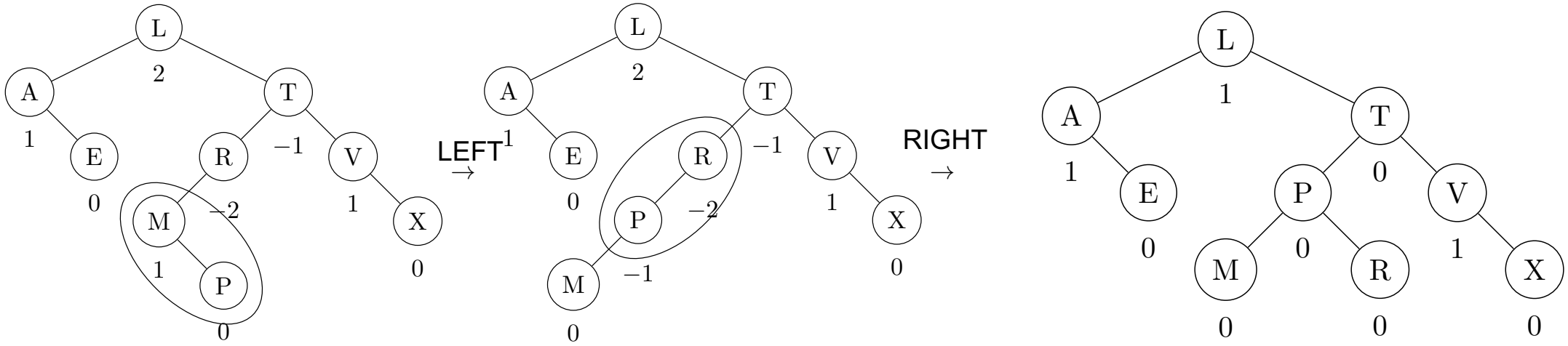
after A V L:



after T R:



after R E X M P

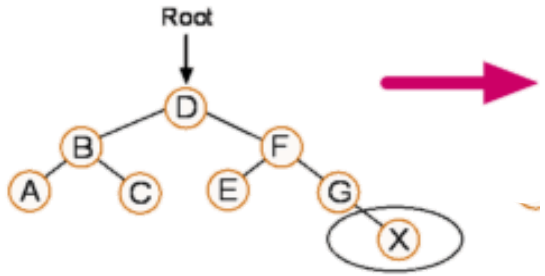


Deletion in BST

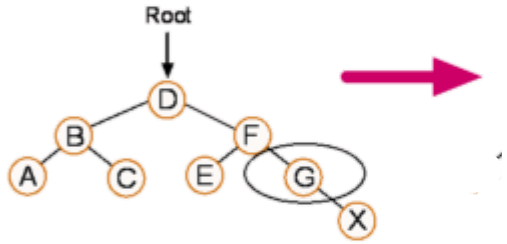
After deletion, do minimal work to keep the new tree valid, ie.:

- connected as a binary tree
- satisfying condition of a BST

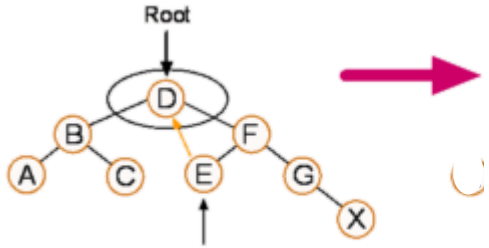
Case 1: delete no-child node (X)



Case 2: delete node single-child node (G)

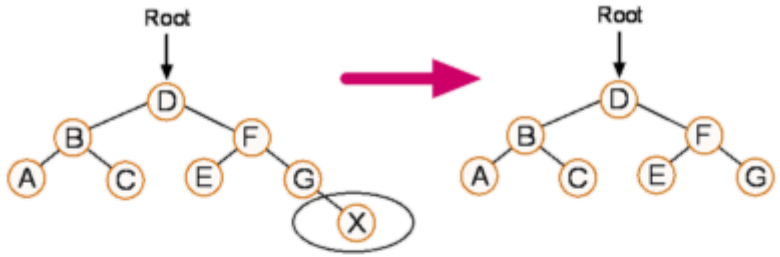


delete two-children node (D):

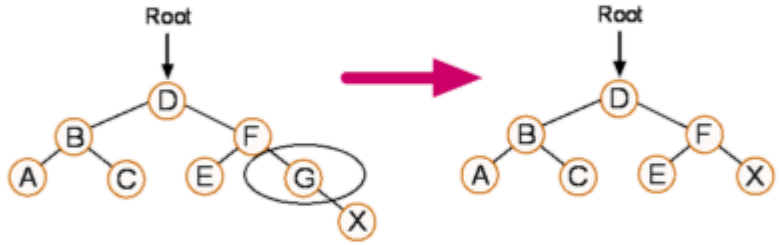


Check: Deletion in BST

Case 1: delete no-child node (X)



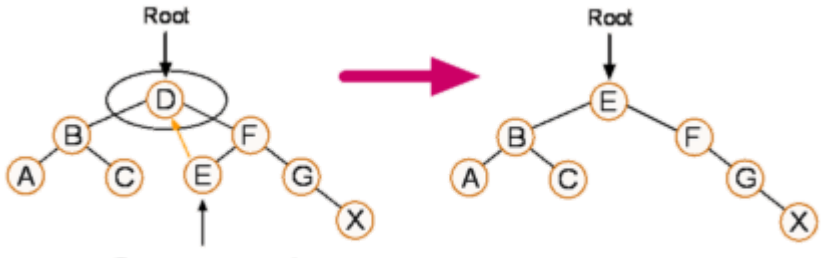
Case 2: delete node single-child node (G)



To delete two-children node (D) : 2 options

- option 1: swap with C (the largest of the left, aka. the rightmost of left sub-tree, the in-order predecessor)
- option 2: swap with E (the smallest of the right, aka. the leftmost of right sub-tree, the in-order successor)

then, delete the newly-positioned D

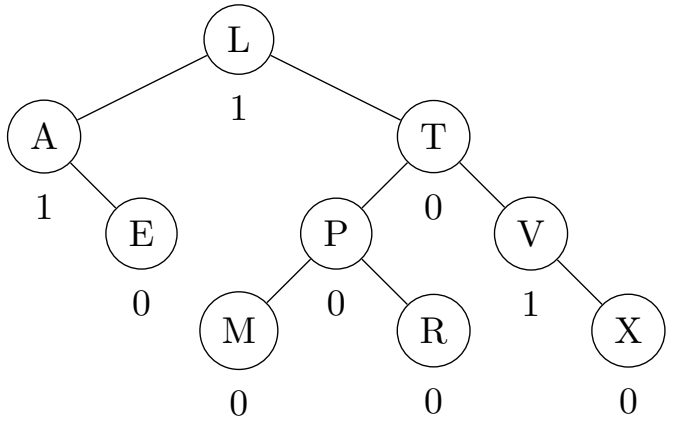


Deletion in AVL, Q11.4

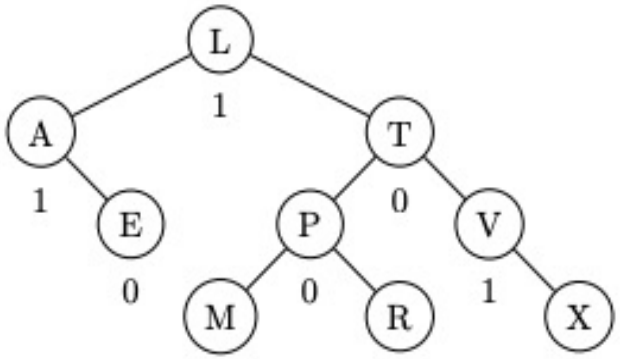
Same as deletion in BST, but two extra steps:

- Walk upwards starting from deleted node
- Rotate at each node if unbalanced.

Q11.4: Delete T, V, X, E in the previous AVL (for two-children node, swap with the smallest of the right sub-tree)

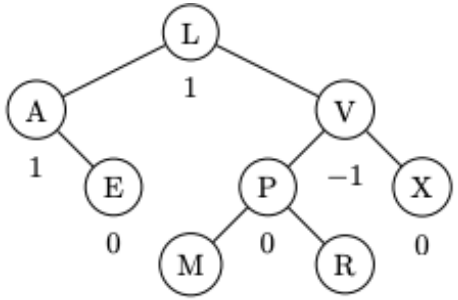


Q11.4: check your answer (delete T, V, X, E)



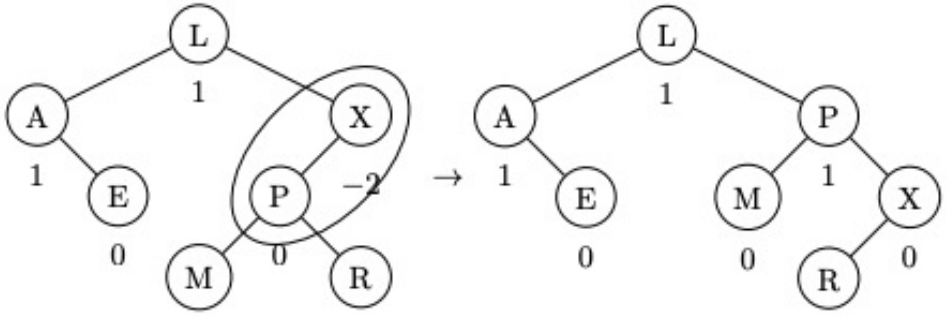
Delete T (having 2 children)

- swap T with V
- del renewed T (that has 1 child X) by replacing with X
- new tree is balanced!



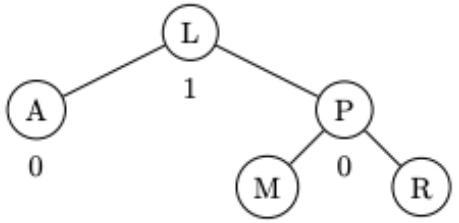
Delete V (having 2 children)

- swap V with X
- del renewed V which has no child
- need to re-balance X with a Left Rotation



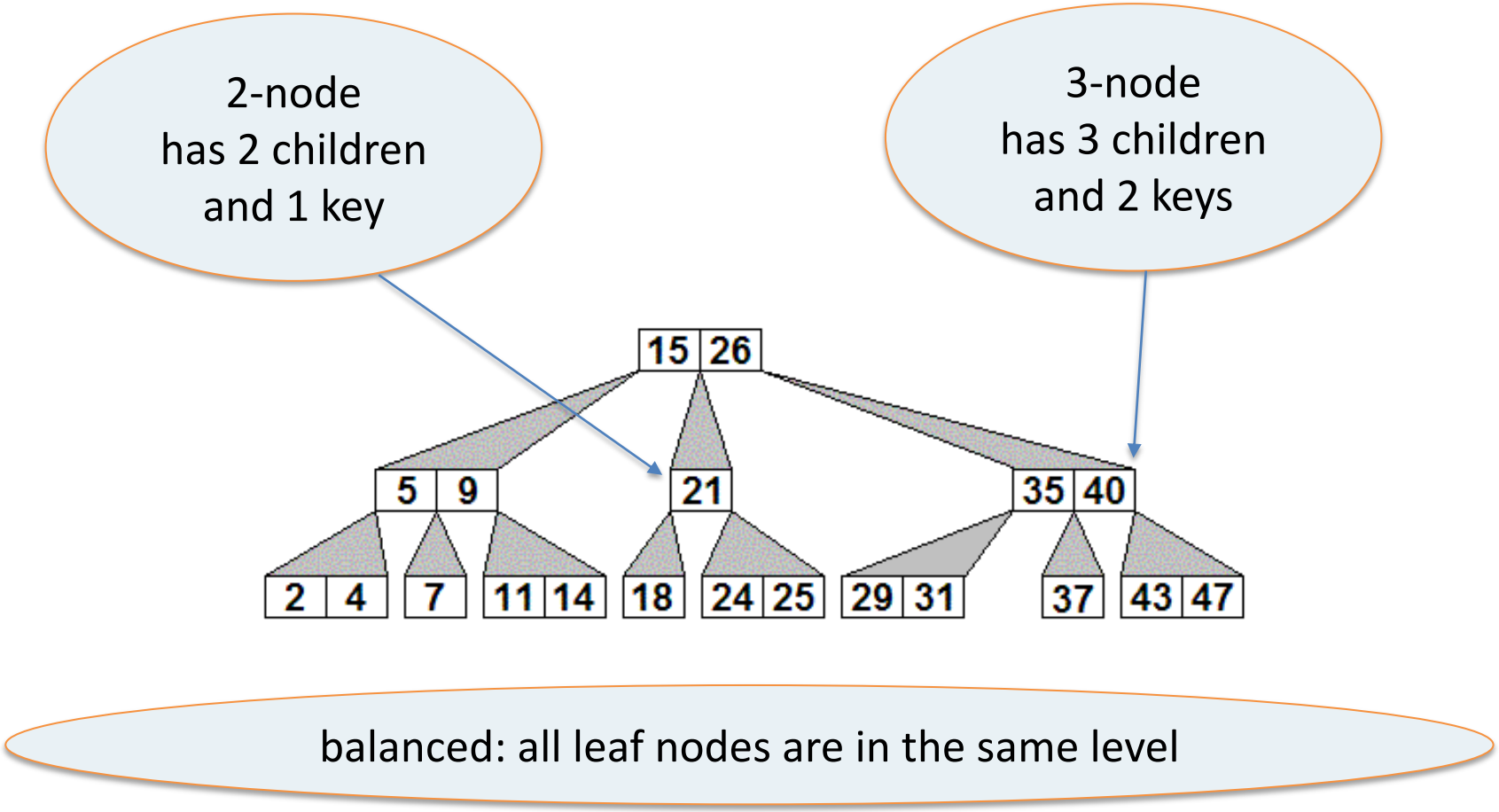
Delete X, and E (no child nodes)

- tree remains balanced after each deletion



2-3 Trees

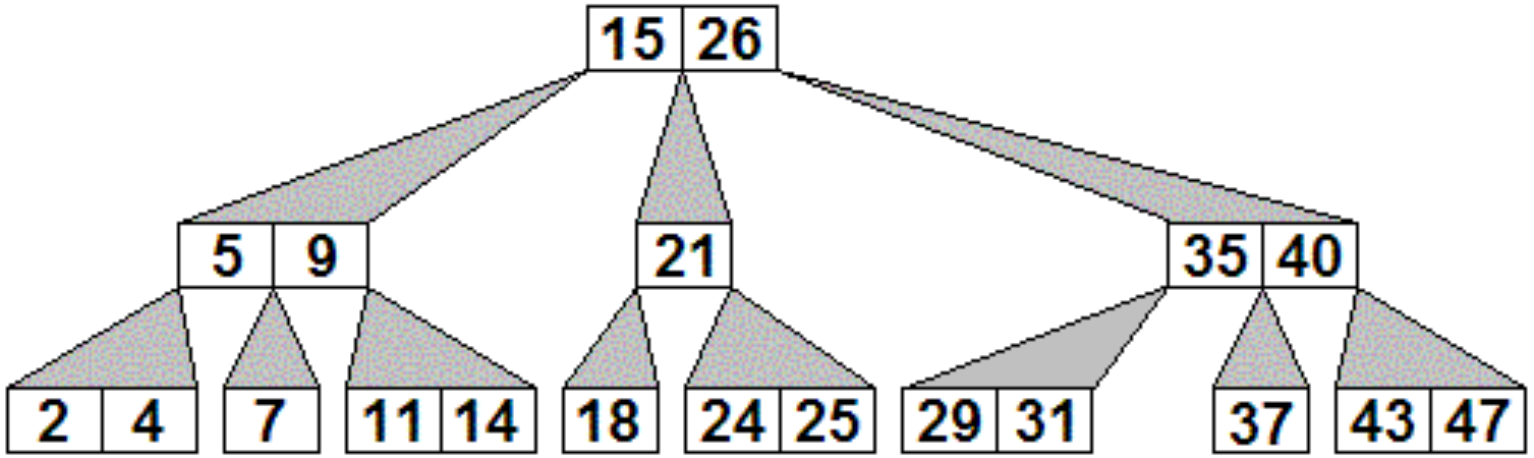
What? It's a search tree, but not a binary tree! Each node might have 1 or 2 keys/data, and hence 2 or 3 children.



2-3 Trees: Insertion

How to insert:

- start from root, go down and *insert new data to a **leaf node***
- if the new leaf has <=2 keys, it's good, done
- if the new leaf has 3 keys: promote the median key to the parent (the promoting might continue several levels upward)



insert 8 is easy: node [7] become [7,8]
insert 45 make node [43,47] be [43,45,47]
→ promote 45, parent become [35,40,45]
→ promote 40, root become [15,26,40]
→ promote 26 to a new root

Insert the following keys into an initially-empty 2-3 Tree.

Class example:

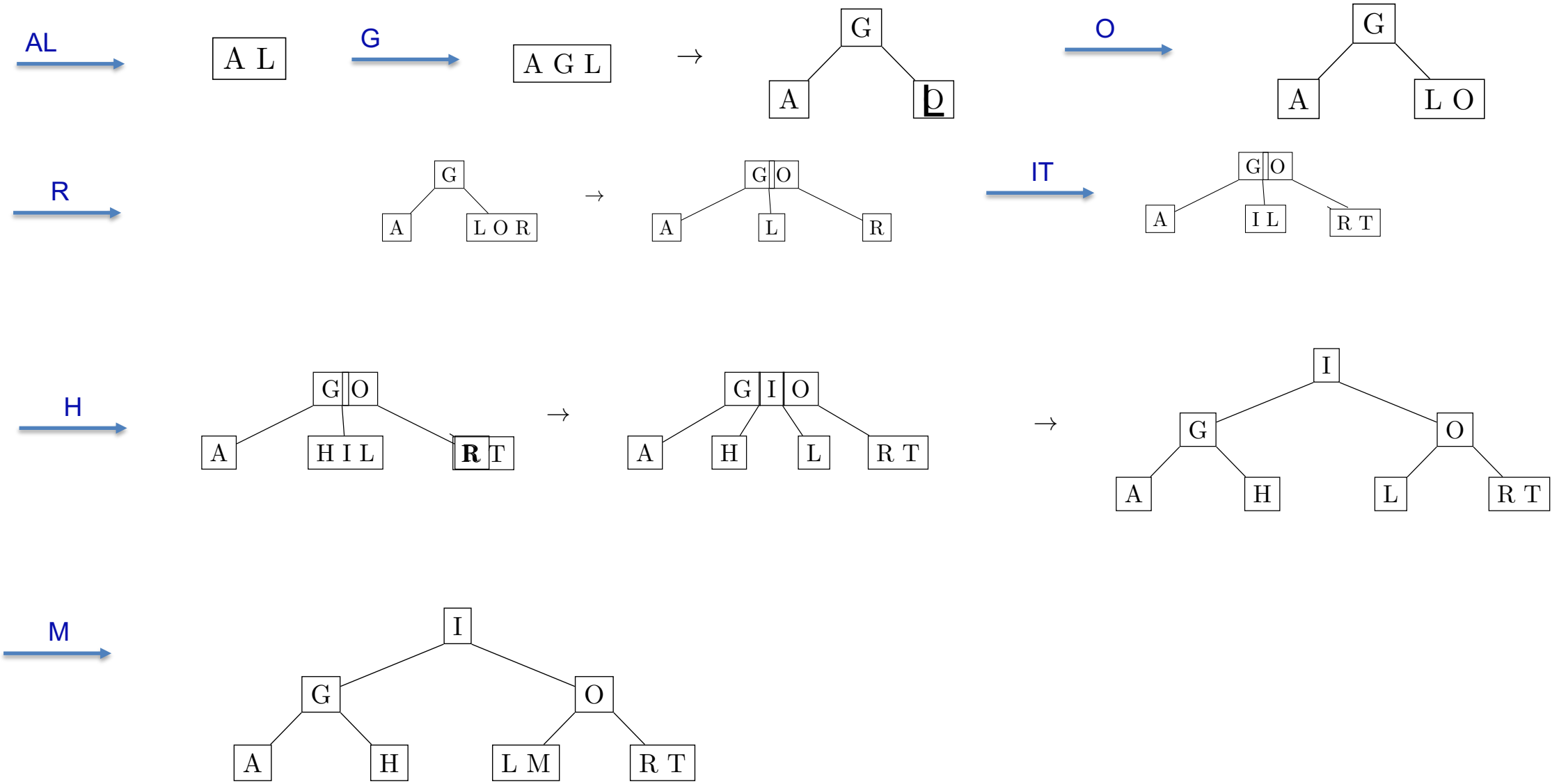
20 10 5 15 30 17 8 2 12 4

Q 11.5: insert the keys into an initially empty 2-3 tree

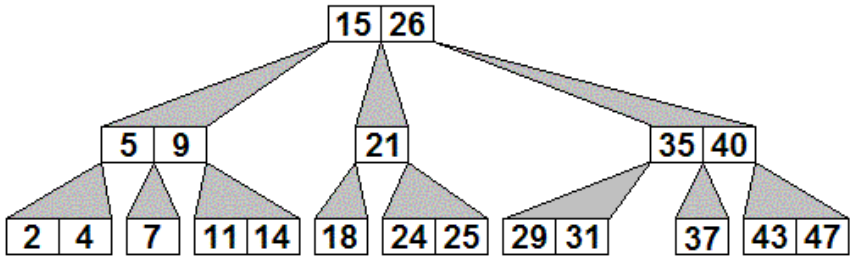
A L G O R I T H M

Q 11.5: Check your solution

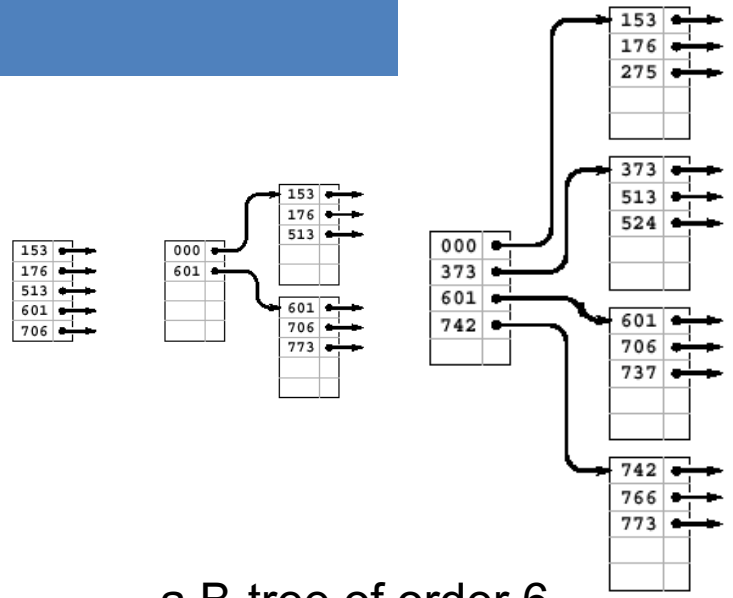
Insert the following keys into an initially-empty 2-3 Tree: A L G O R I T H M



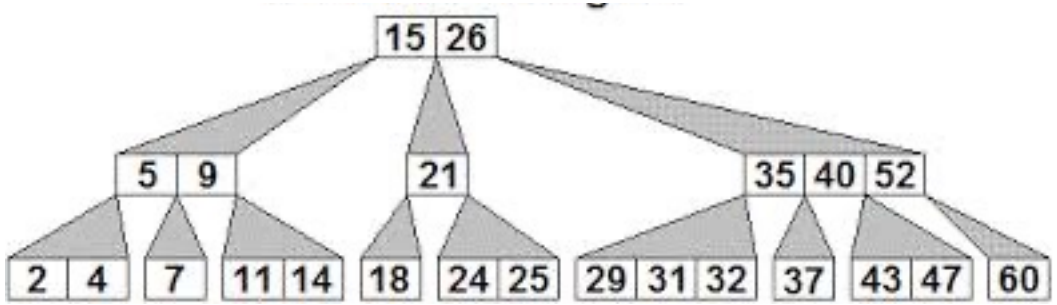
2-3 Trees, 2-3-4 Trees, B-Tree



2-3 trees= B-trees of order 3
(order= max number of children)



a B-tree of order 6



2-3-4 trees= B-trees of order 4

B-tree principles

- Always insert at leaves
- When a node full: promote the median data to the node's parent [and walk up further if needed]

Image sources: ?? and http://anh.cs.luc.edu/363/notes/06DynamicDataStructures.html

Additional Slides

Including:

- Deletion in 2-3 Trees [probably not examinable, check with the lecturers]
- Lab on BST insert

Deletion in 2-3 Trees

Keep the 2-3 property after deletion by

Step 1: If the deleted key not in a leaf node: swap it with the rightmost left key or the left most right key (similar to BST)

Step 2: Turn the deleted key into a "hole" and try to remove it.

Stop if the removal is possible (=lucky). Otherwise repeat:

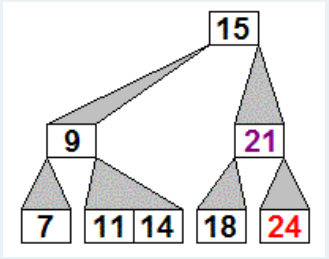
- "moving up" by swapping the hole with a valid parent key, merge the key's children into one node and promote the middle key to the hole if applicable

until:

- the new parent node is a valid 2-3 node: job done
- the new parent doesn't have any key, but is the root: remove the root

Note: Be cunning! If have more than 1 choices, choose the simpler one!

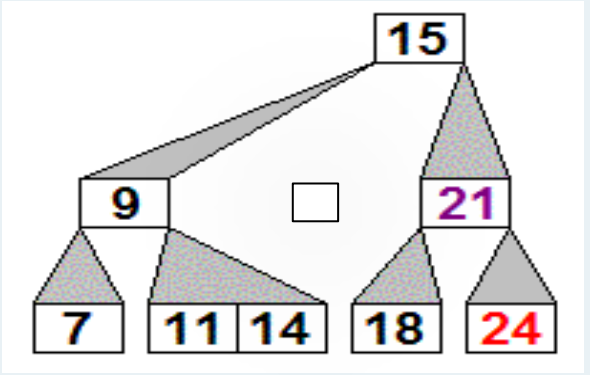
step 1:



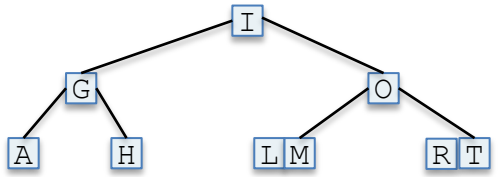
del 21: swap 21 with 18 or 24
del 15: swap 15 with 14 or 18

step 2:

HOLE at 11 or 14: lucky!
HOLE at 9: promote 11, lucky!
HOLE at 21: swap HOLE up to 15 …



Question 11.6: Delete I, then L, then A from the tree



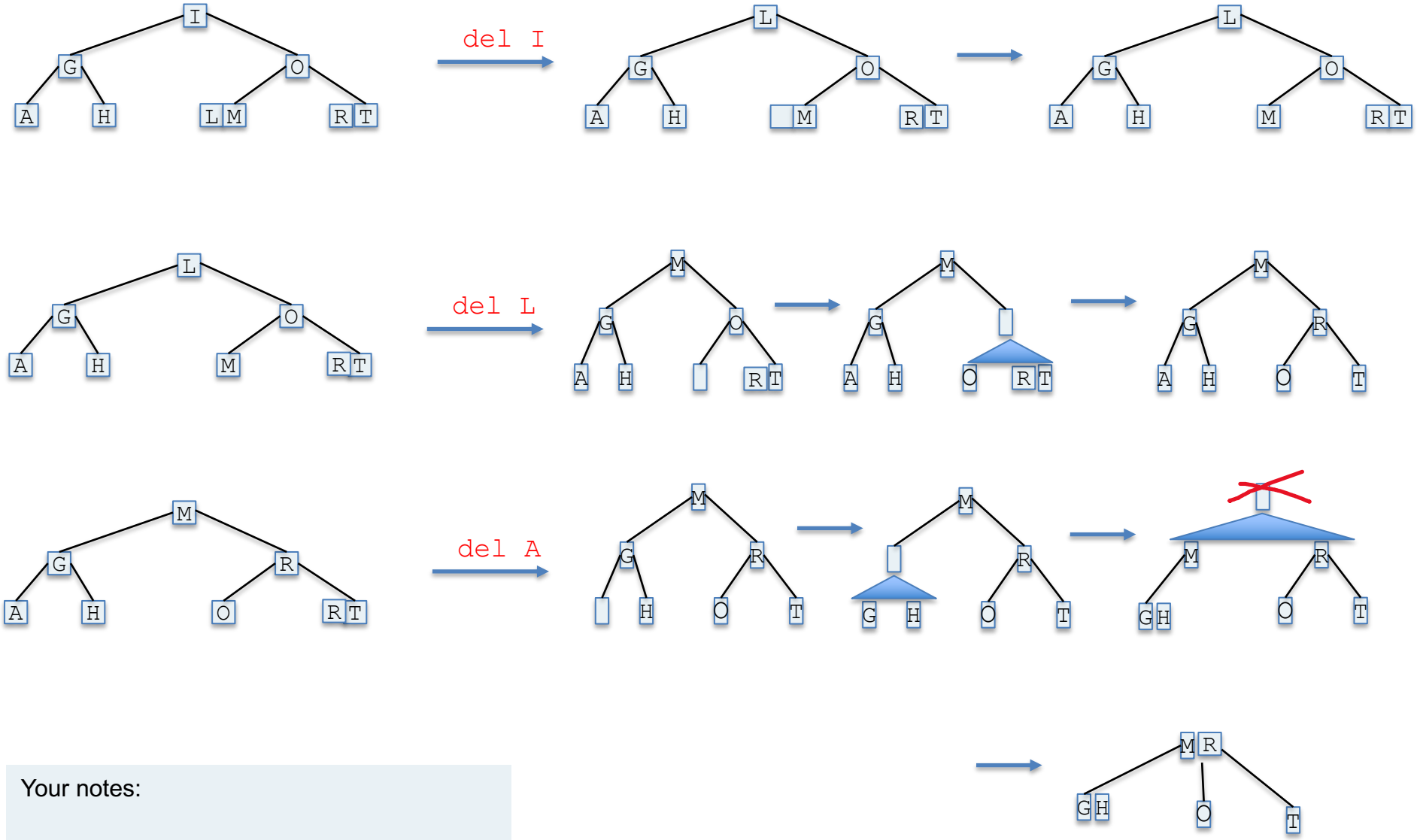
del I

del L

del A

Your notes:

Check Q 11.5: Delete I, then L, then A from the tree



Your notes:

Lab

Discuss: BST-insert

Assignment 2: Q&A

Q&A on previous week materials, revision, and/or A2

[Optional] Implement BST: insert, build tree from data, printing the tree. Note:

- Build your program from scratch
- But you can use the list and queue modules from previous weeks

Simple defs for binary trees

```
typedef struct treenode *tree_t;
struct treenode {
  int key;
  tree_t left, right;
} ;
```

```
tree_t insert(tree_t t, int key);
//OR
void insert(tree_t *t, int key);
```

Example of creating a tree

```
tree_t t= NULL;

// insert 10 to tree t
t= insert(t, 10); //OR
insert(&t, 10);
// depending on insert header
```