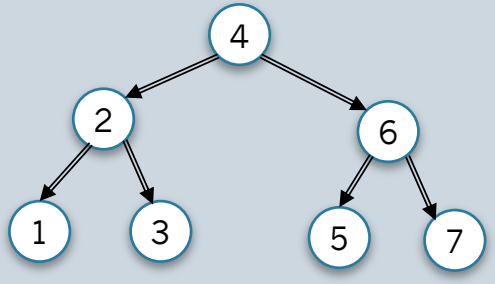
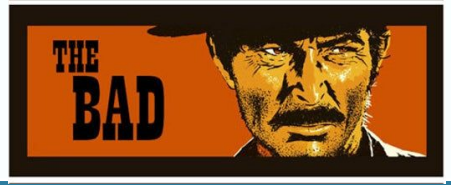
COMP20007 Workshop Week 9

	Preparation: - *have draft papers and pen ready* - open Ed.Week 9 Workshop
1	Why BST, AVL, 2-3 Tree?
2	BST: Rotation, Balance factor: Q 9.1, 9.2
3	AVL Tree: Concepts, Insertion: Q 9.3
4	2-3 Tree: Concepts, Insertion & Deletion: Q 9.4, 9.5
5	B-tree?
LAB	BST: insertion & level traversal (homework) 2-3-4 Tree: insertion & level traversal (homework) Questions for previous week materials

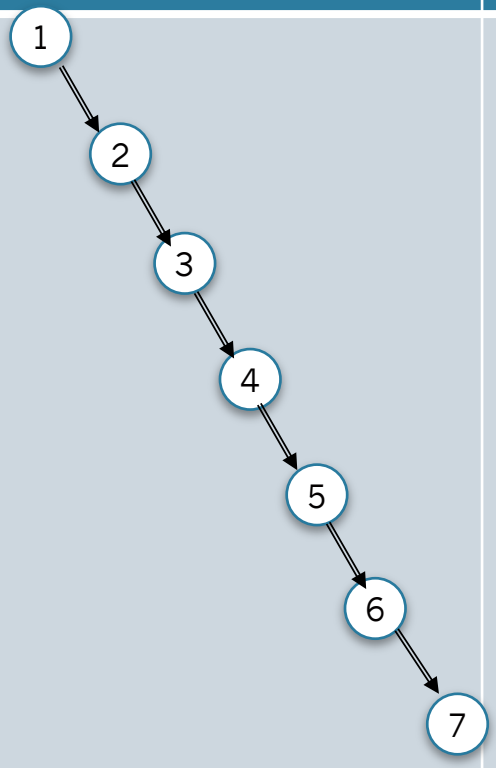
BST efficiency depends on the order of input data

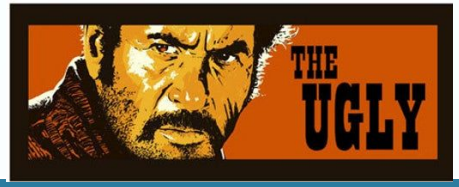
4 6 2 1 3 7 5



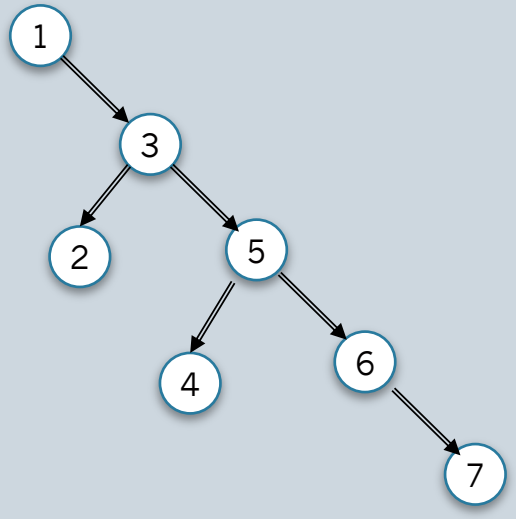


1 2 3 4 5 6 7





1 3 5 2 4 6 7



Want The Good, no matter what's the data input order? Use AVL or 2-3 Trees

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

Using Rotations to rebalance AVL

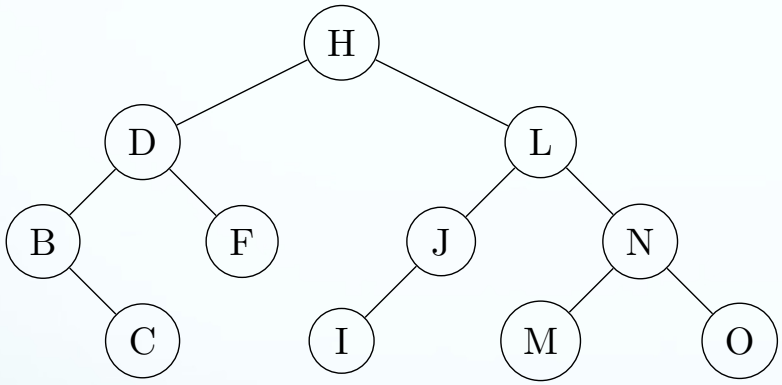
- *At the start:* an empty BST is an AVL
- *Problem:* After a insertion/deletion, the resulted tree might become unbalanced
- *Approach*: use Rotations to rebalance.

- To rotate:

  When? What? How?

What's a balanced BST?

Q 9.2: 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

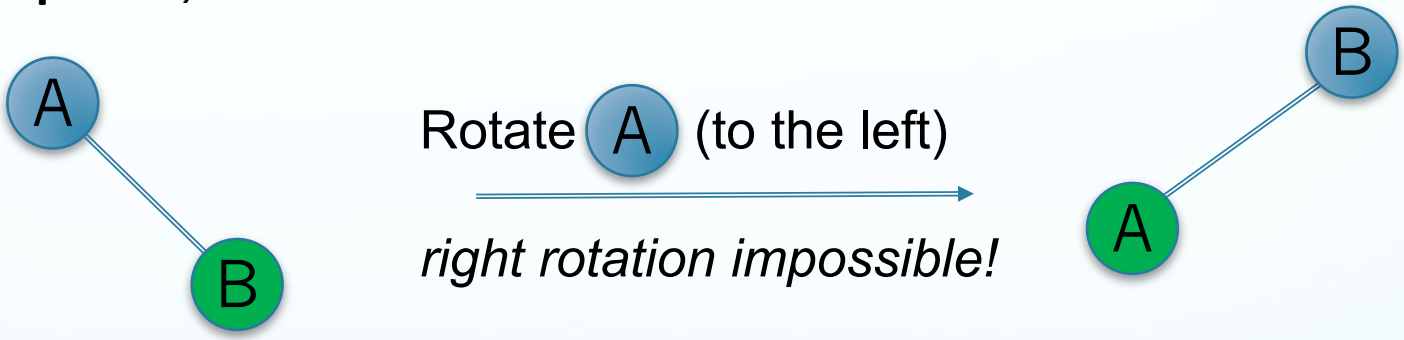
= for each node, the difference of subtree heights is at most 1

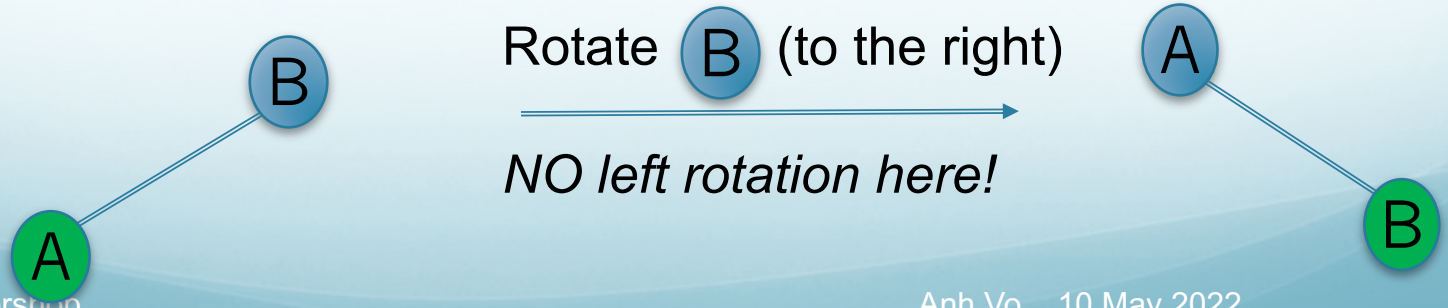
BST: what's a rotation

A rotation reverses the parent-child relationship of 2 parent-child nodes, reserving the BST search property

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

Note: we **rotate a parent**, we do not rotate a child.

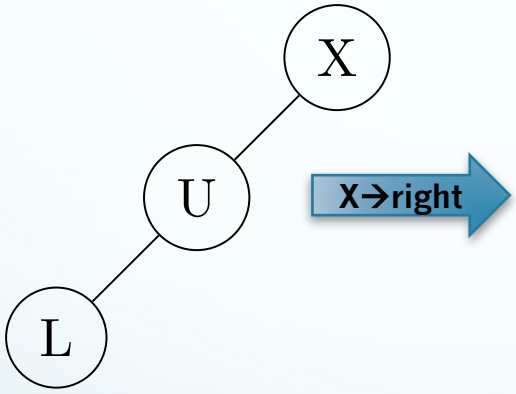




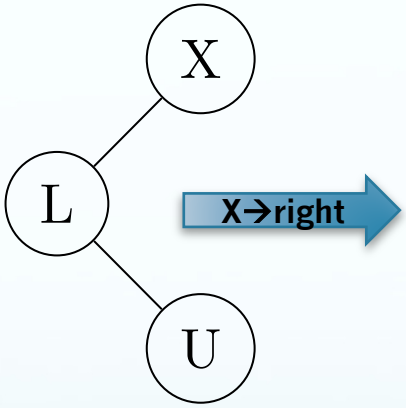
Q 9.1 [class]: Rotation

In the following binary 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)

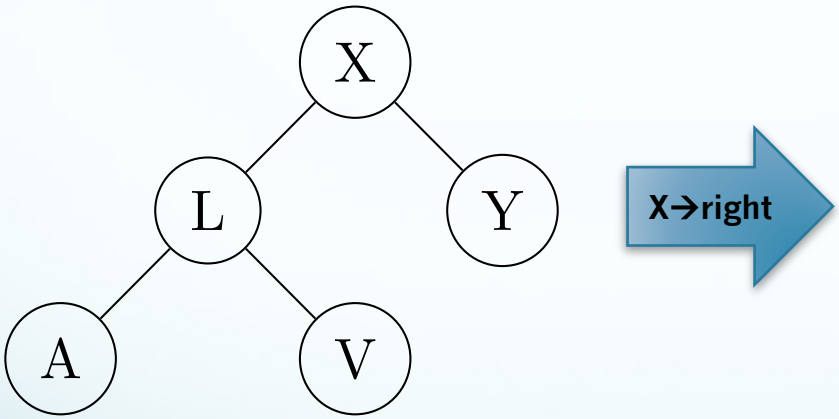


Recall: Only 2 types of rotations: *Right Rotation* (a node and its left child), and *Left Rotation* (a node and its right child)

Problem 1 [class]: Rotation

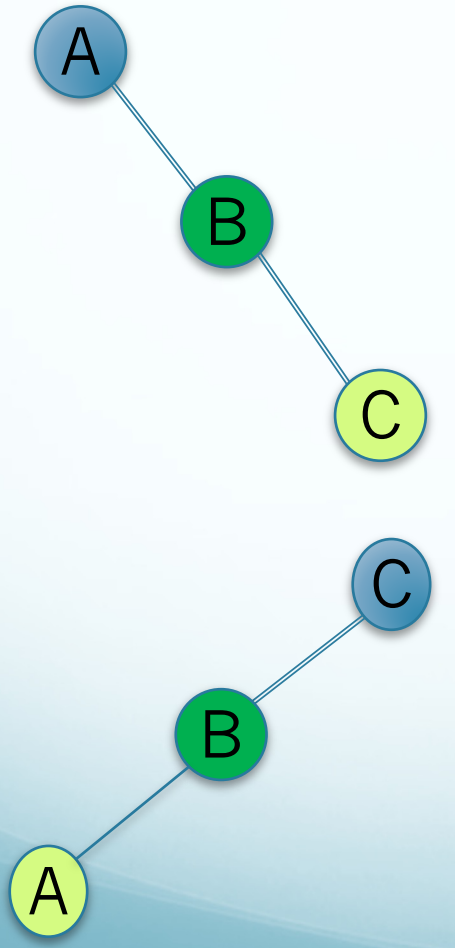
In the following binary trees, rotate the ‘X’ node to the right (that is, rotate it and its left child). Do these rotations improve the overall balance of the tree?

(a)



AVL: Two Basic Rotations: 1) Single Rotation

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



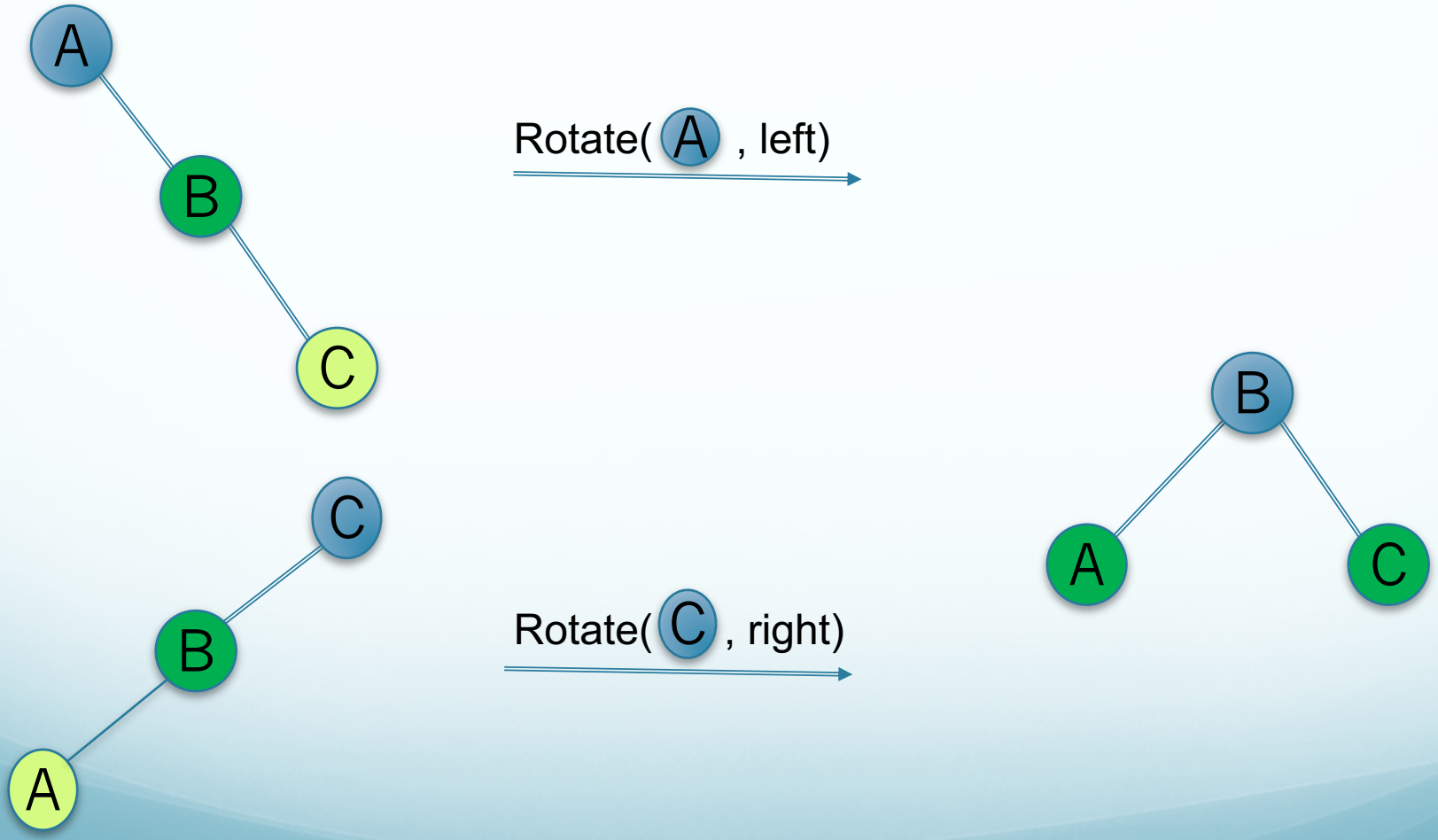
Questions we should ask ourselves:

- Which node (or subtree) is unbalanced?
- Which rotation can be done?

HERE:
→ Rotate the root and hence balance the stick

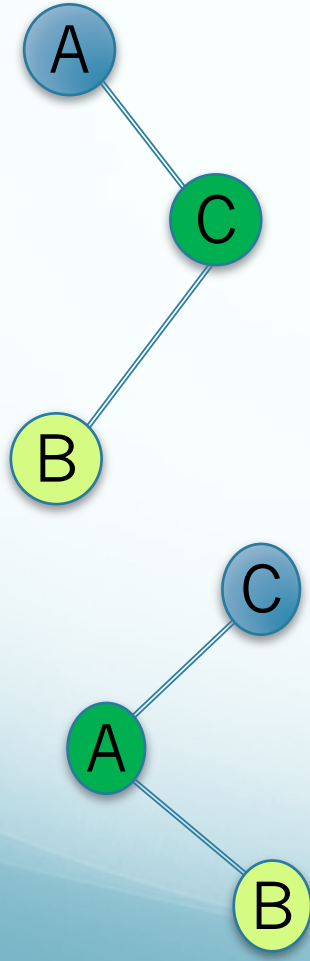
AVL: Two Basic Rotations: 1) Single Rotation

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



AVL: Two Basic Rotations: 2) Double Rotation

Applied when an AVL (subtree) is not a “stick”:



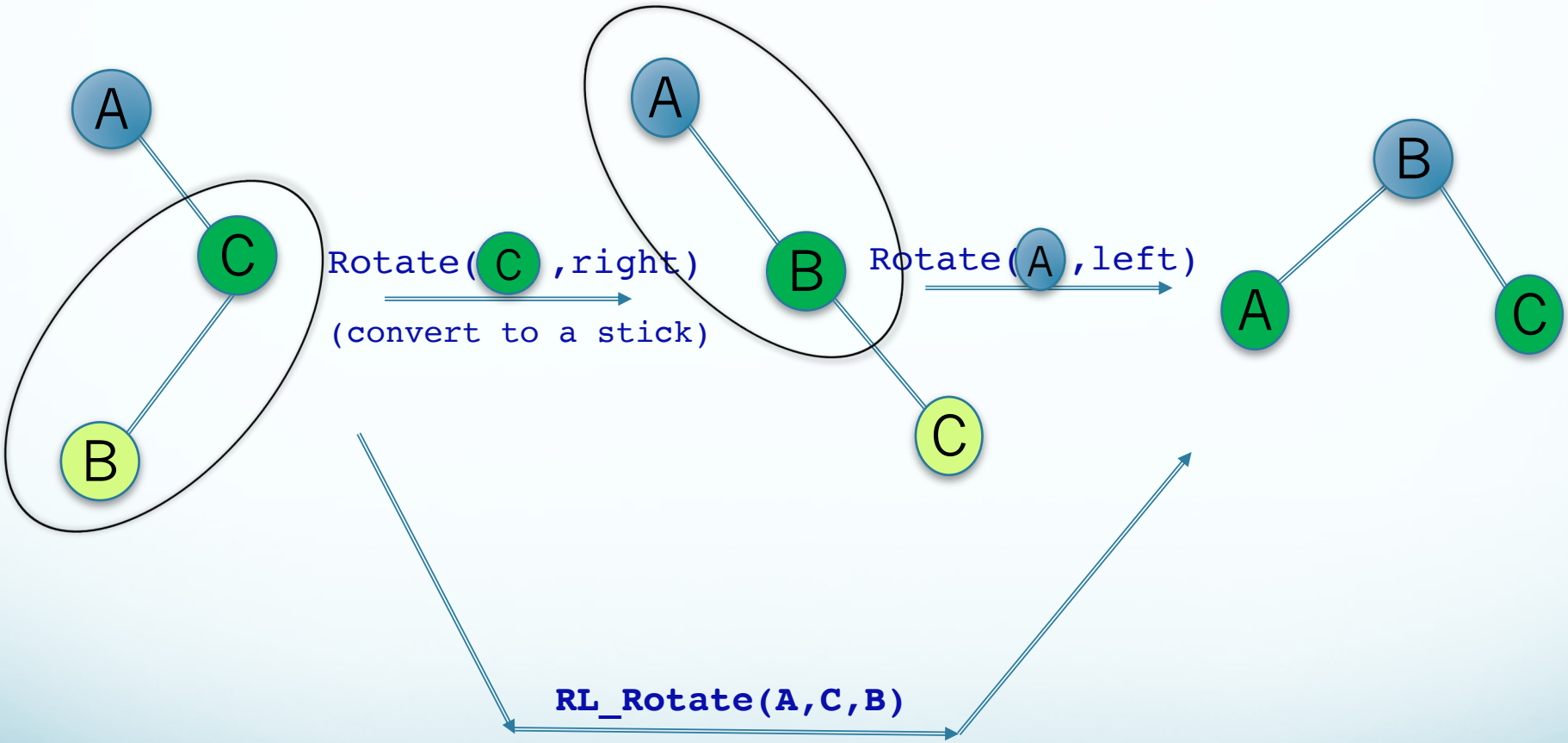
Rotation1:

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

Rotation2:

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

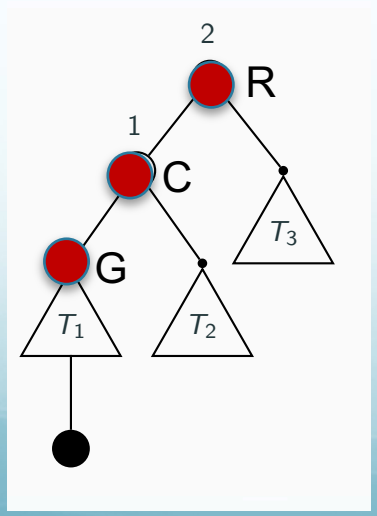
Double Rotation Example: RL rotation

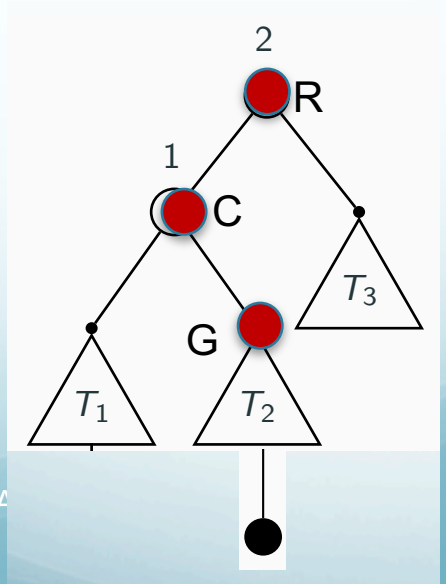


Do it Yourself: Perform `LR_Rotate(C,A,B)` for the other case of the previous page

AVL: Using Rotations to rebalance AVL

- Problem: When inserting/deleting node, AVL might become unbalanced
- Approach: Rotations (Rotate WHAT?, and HOW?)
- Rotate WHAT?
  - Walk up, find the *lowest* subtree R which is unbalanced
- HOW
  - Consider *the first 3 nodes* R→C→G in the path from root R to the just-inserted node
  - Apply a single rotations if that path is a stick, double rotation otherwise





AVL Tree Insertion

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

Class example:

20 10 5 15 30 17 8 2 12 4

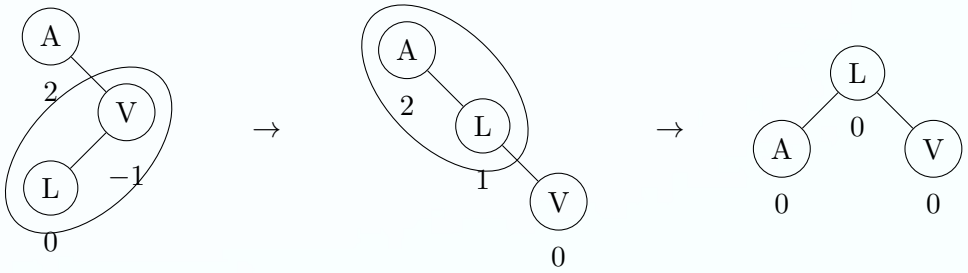
Q9.3 [*group/individual*]:

A V L T R E X M P

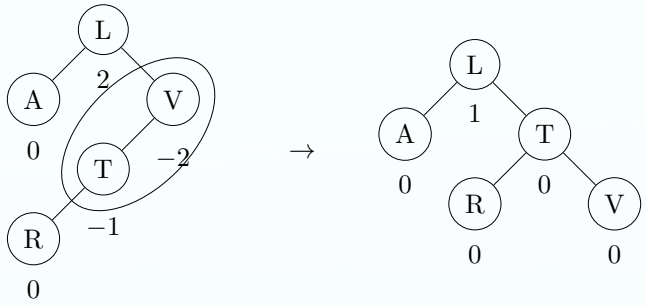
Q 9.3: Check your solution

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

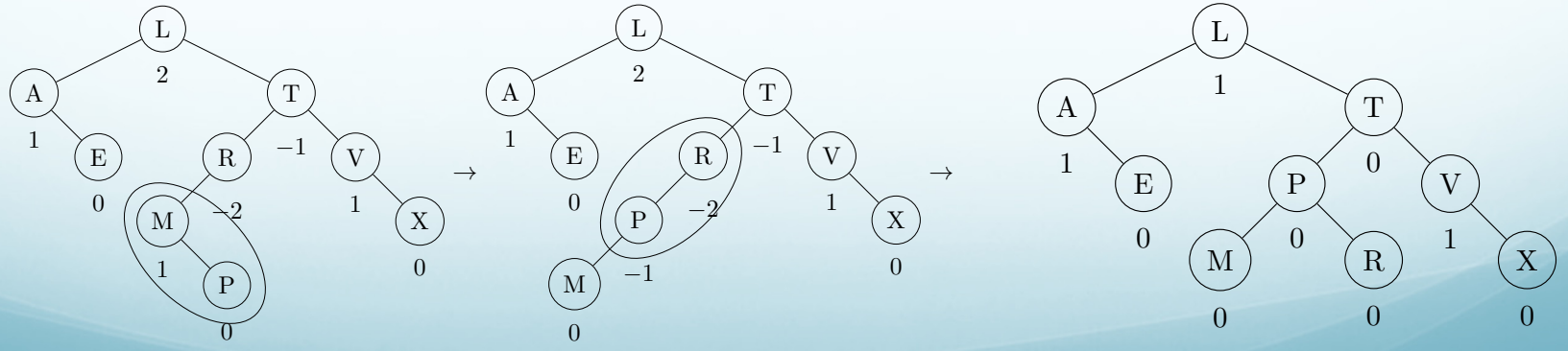
after A V L:



after T R:

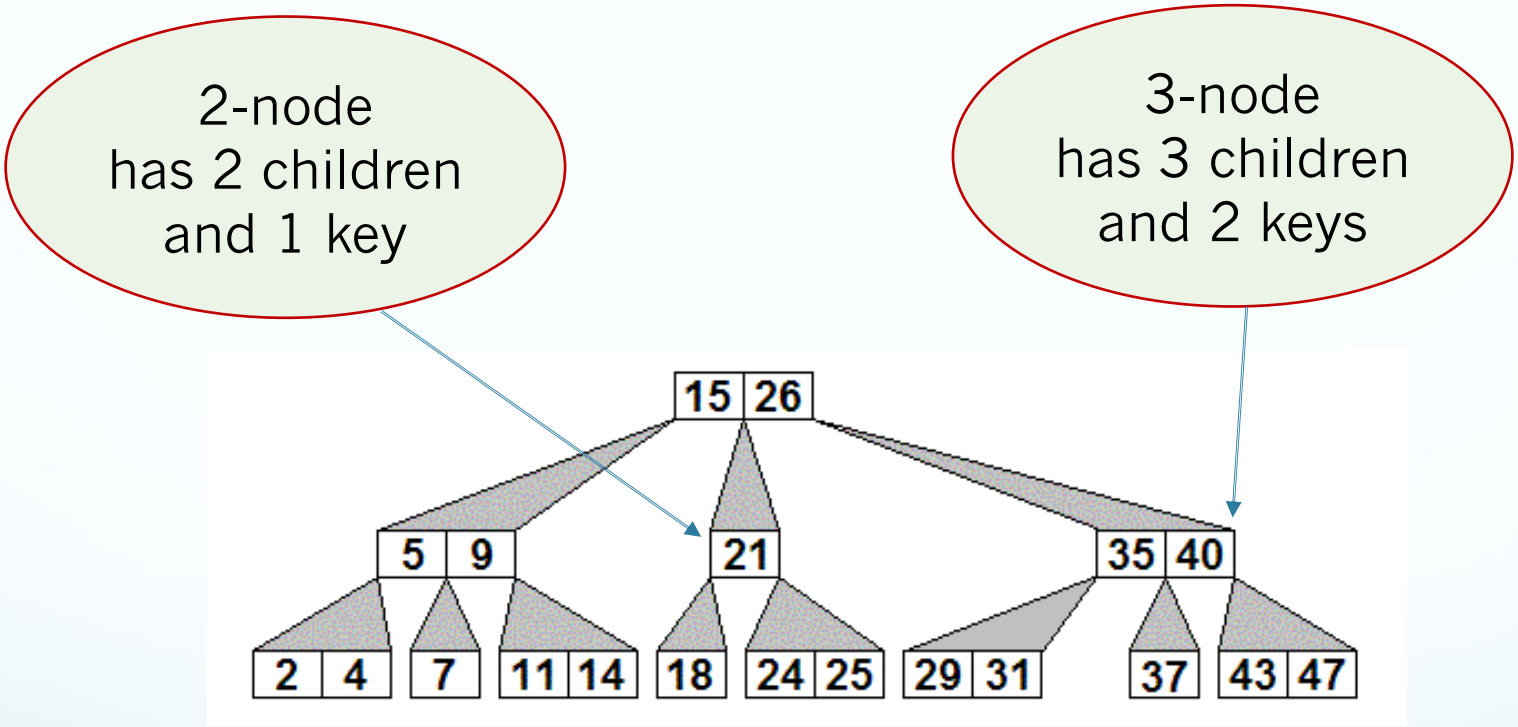


after R E X M P



2-3 Trees

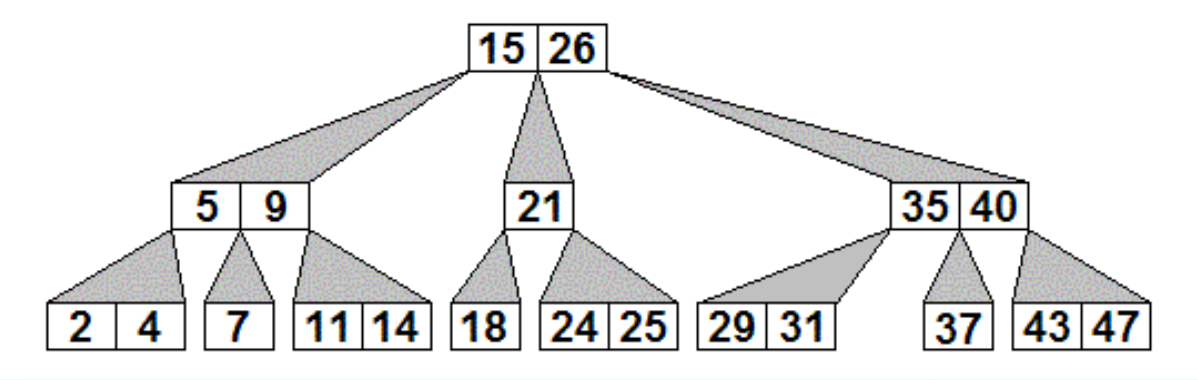
- *What?* It's a search tree, but not a binary! Each node might have 1 or 2 keys/data, and hence 2 or 3 children.



balanced: all leaf nodes are in the same level

2-3 Trees: Insertion

- How to insert:
  - start from root, go down and insert to a leaf
  - if the new leaf has <=2 keys, it's ok
  - 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

2-3 Tree Insertion

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

Class example:

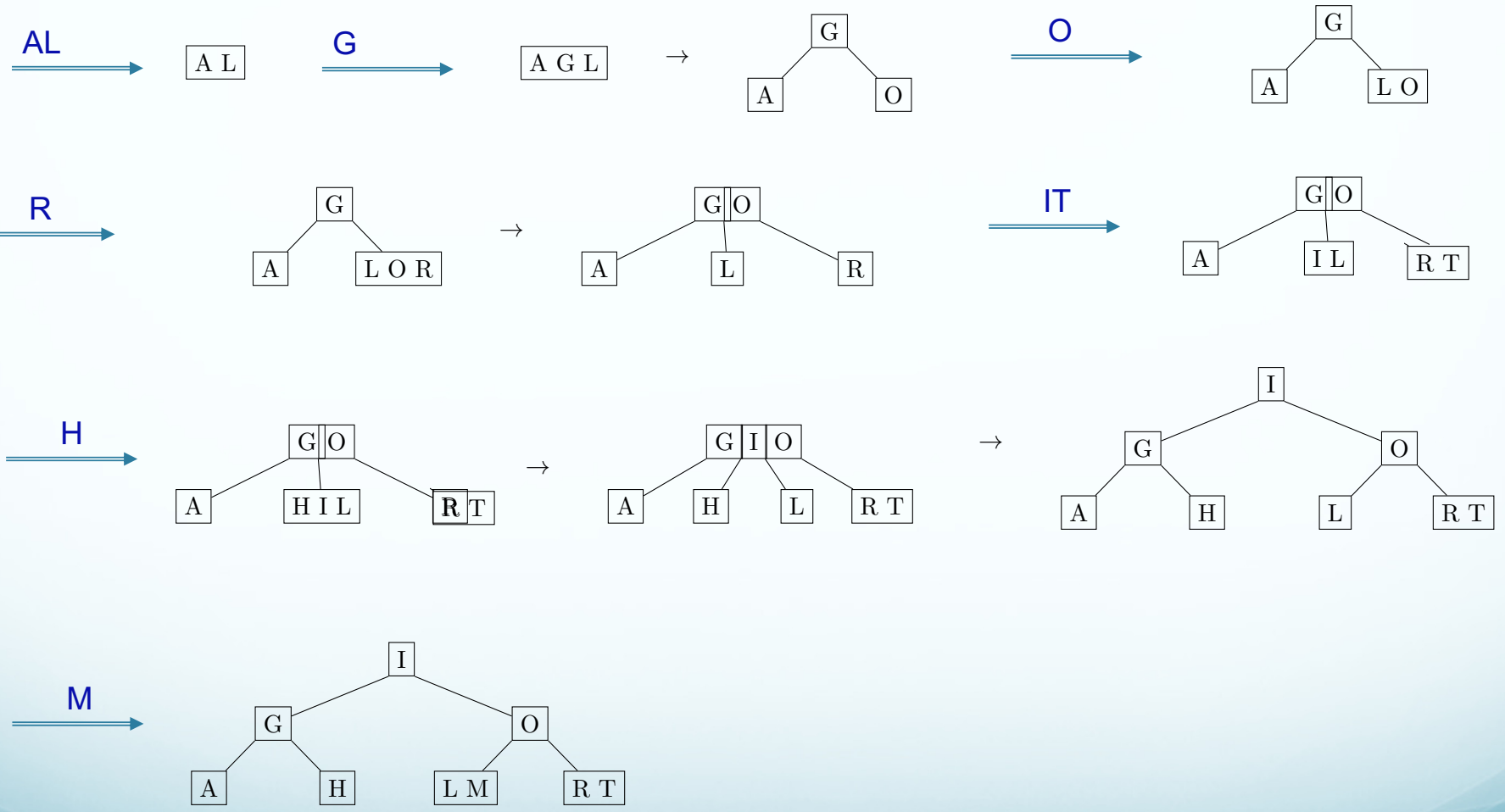
20 10 5 15 30 17 8 2 12 4

DYI Q 9.4: insert the keys into an initially empty 2-3 tree

A L G O R I T H M

Q 9.4: Check your solution

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

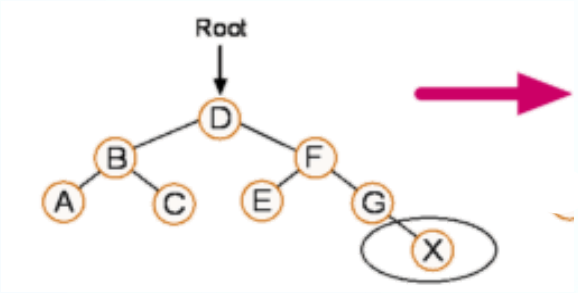


Deletion in BST

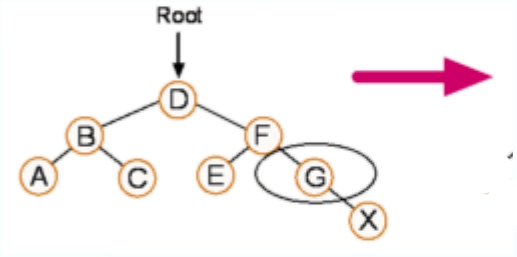
After deletion, do minimal work to keep the new tree valid, ie.:

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

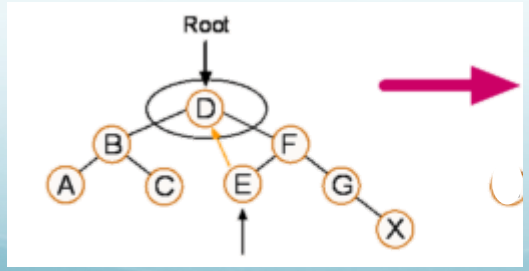
delete X which has no child



delete G which has only one child

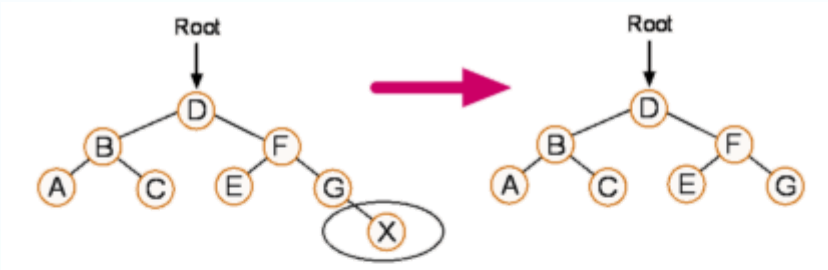


delete D, which has 2 children:

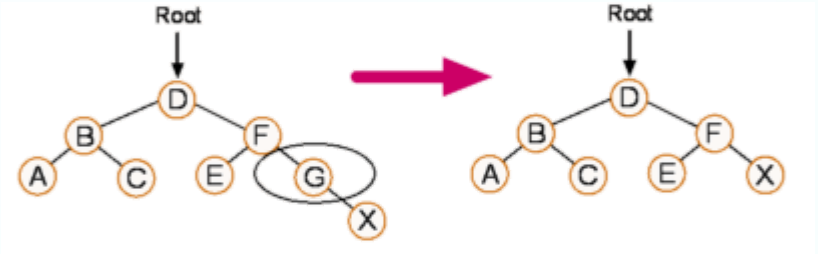


Check: Deletion in BST

delete X which has no child



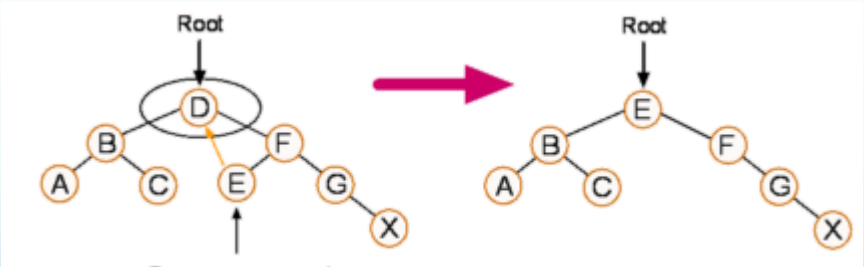
delete G which has only one child



To delete D, which has 2 children: 2 options

- option 1: swap with C (the largest of the left, aka. the rightmost left child)
- option 2: swap with E (the smallest of the right, aka. the leftmost right child)

then, delete the new D



Deletion in 2-3 Trees

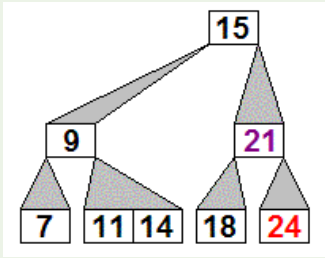
Keep the 2-3 property after deletion by

- 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

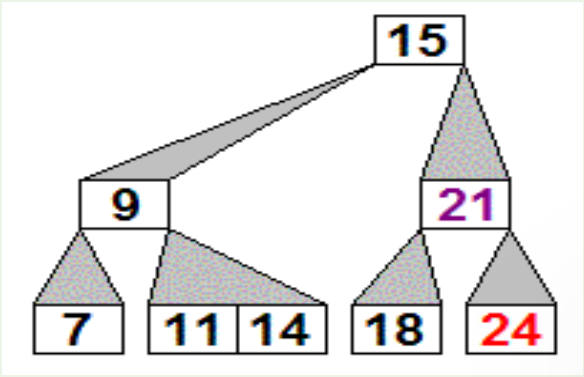
step 1:



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

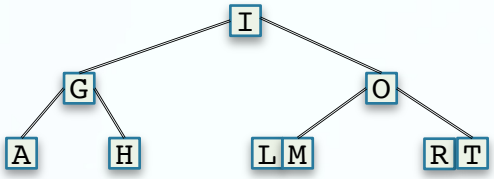
step 2:

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



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

Question 9.5: Delete `I`, then `L`, then `A` from the tree



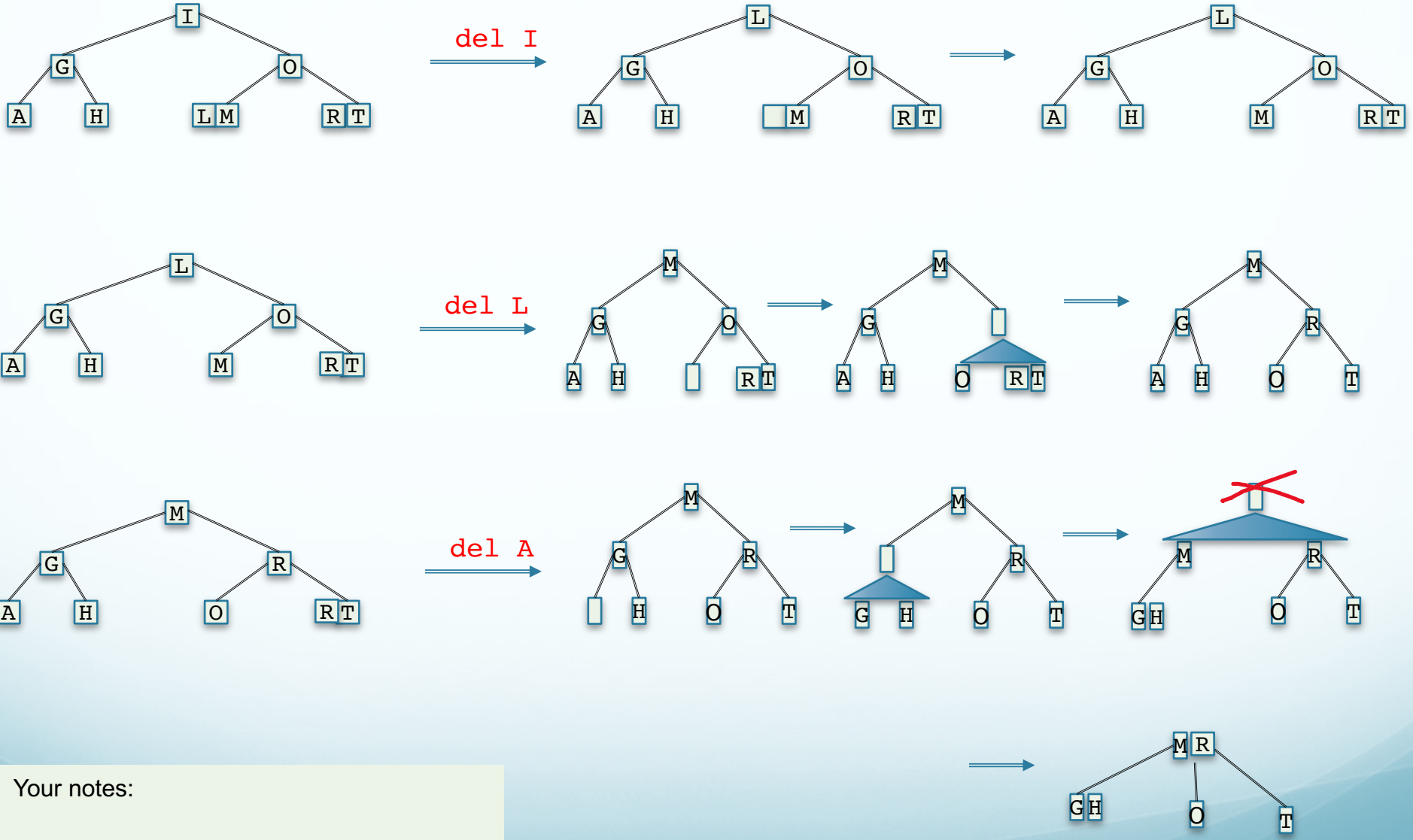
del I

del L

del A

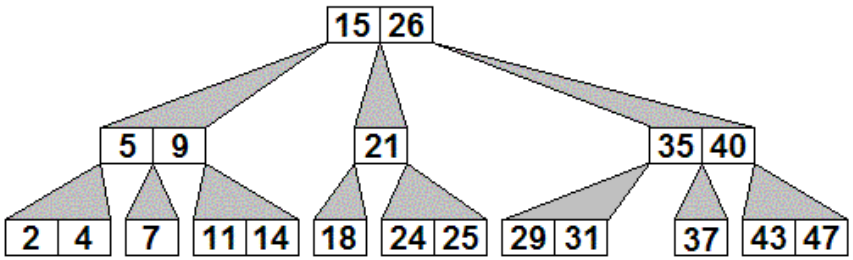
Your notes:

Check Q 9.5: Delete I, then L, then A from the tree

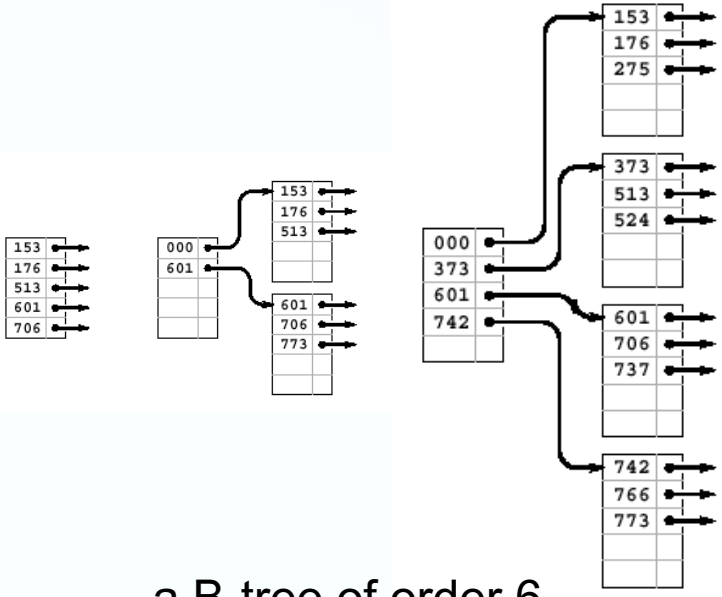


Your notes:

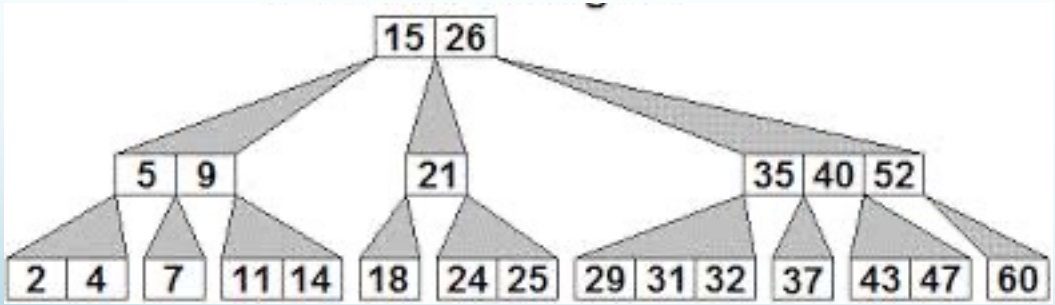
2-3 Trees, 2-3-4 Trees, B-Tree



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

Lab

- Q&A on previous week materials, and/or
- [Optional] Implement BST: insert, build tree from data, printing the tree. Note:
  - Build your program from scratch
  - But you can use last week list and queue modules
- And/or: [Optional] Implement 2-3-4 Tree with the above operations
- For easy printing of trees, first build a complete tree using data:

```
50 30 80 20 40 60 90 15 25 35 45 55 65
```

Simple defs for binary trees

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

```
??? insert(??? t, int key)
```

Example of creating a tree

```
tree_t t= NULL;

// insert 10 to tree t
```

Lab

- Q&A on previous week materials, and/or
- [Optional] Implement BST: insert, build tree from data, printing the tree. Note:
  - Build your program from scratch
  - But you can use last week list and queue modules
- And/or: [Optional] Implement 2-3-4 Tree with the above operations
- For easy printing of trees, first build a complete tree using data:

```
50 30 80 20 40 60 90 15 25 35 45 55 65
```

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
```