COMP20003 Workshop Week 1

It's about Search Trees

Binary Trees, Tree Traversal, BST
AVL & Rotations

Lab: bst_insert

- defining a tree node, a tree (or bst)
- bst_insert
- bst_search
- bst_delete

BE PREPARED FOR ASSIGMENT 2 BEFORE NEXT WEEK'S WORKSHOP

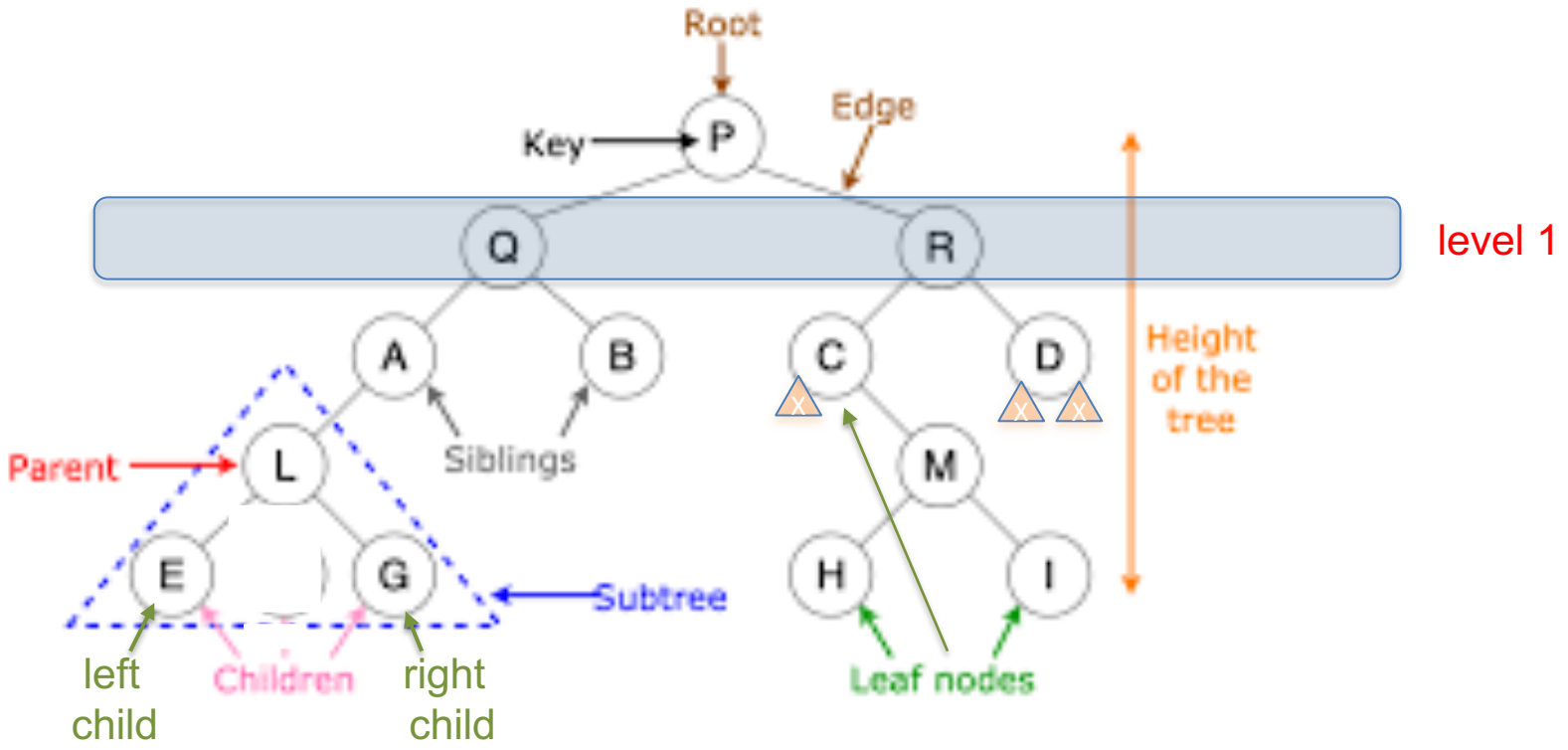
DS for Dictionary so far

	Search	Insert	Delete
Dynamic Arrays	-	+	-
Linked List	-	+	-
Sorted Arrays	+	-	-
Something better?	+	+	+

+: seems good in terms of time efficiency, ie. ”clearly better than linear”

- : not good, it’s linear in terms of time efficiency

Binary Trees: some jargons



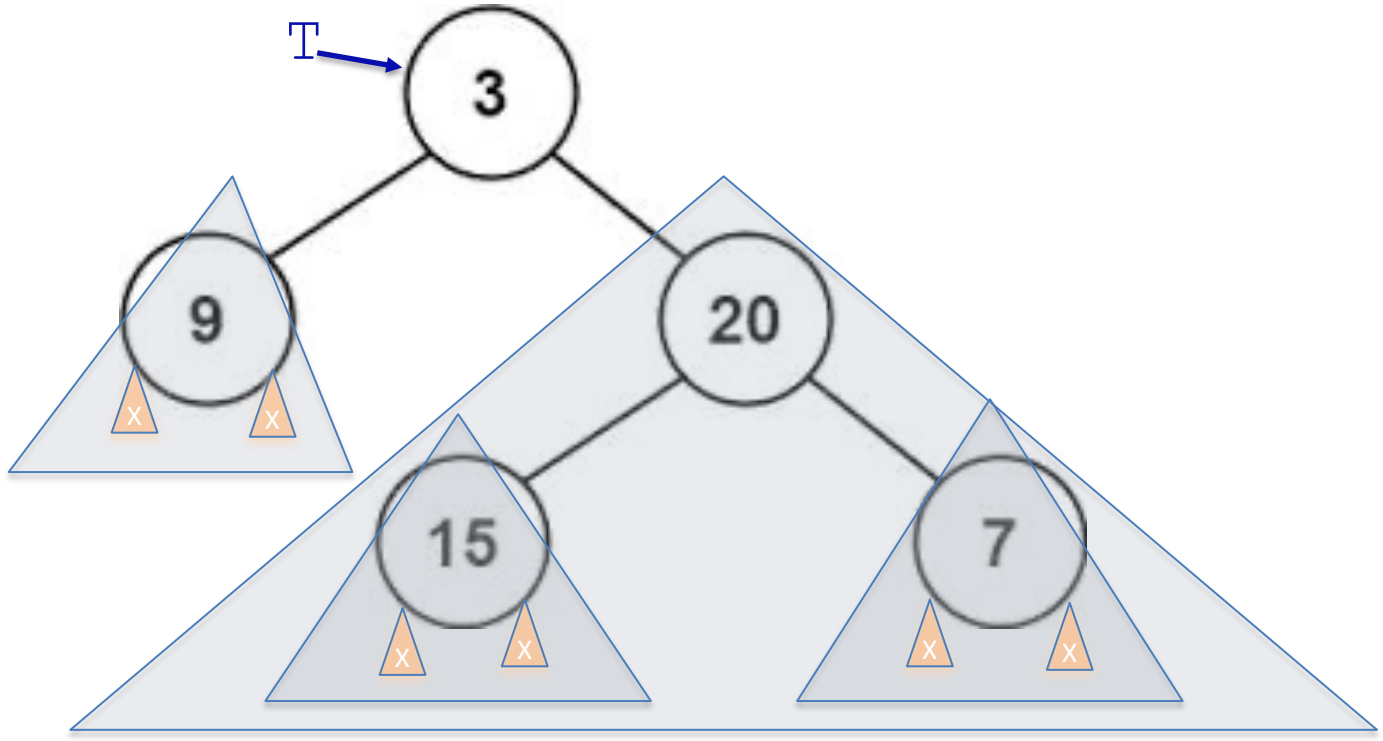
Notes:

△ denotes a NULL pointer, only a few of them drawn here

Binary Tree: Recursive Definition

A binary tree is:

- `NULL`, or
- a node, called the tree's *root node*, that contains:
  - some data, normally including a key,
  - a link to another binary tree called the root's *left child*, and
  - a link to another binary tree called the root's *right child*



Declaring trees: code examples

Model 1: in ED workspace style	Model 2:	Notes

`struct bst {`
`  struct data data;`
`  struct bst *left;`
`  struct bst *right;`
`};`
`struct bst *t= NULL;`
	`typedef struct t_node *tree_t;`
	`struct t_node {`
	`  data_t *data;`
	`  tree_t left;`
	`  tree_t right;`
	`};`
	`tree_t t= NULL;`
	`//t is a pointer because it's a tree_t`
		a tree node has
		• a data
		• a left child/sub-tree
		• a right child/sub-tree
		
		this line creates the empty tree `t`

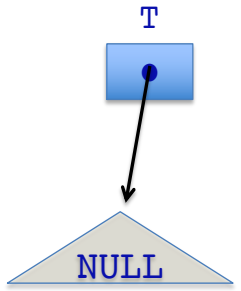
Note: often `data_t` includes a special field `key`. And `data` is:

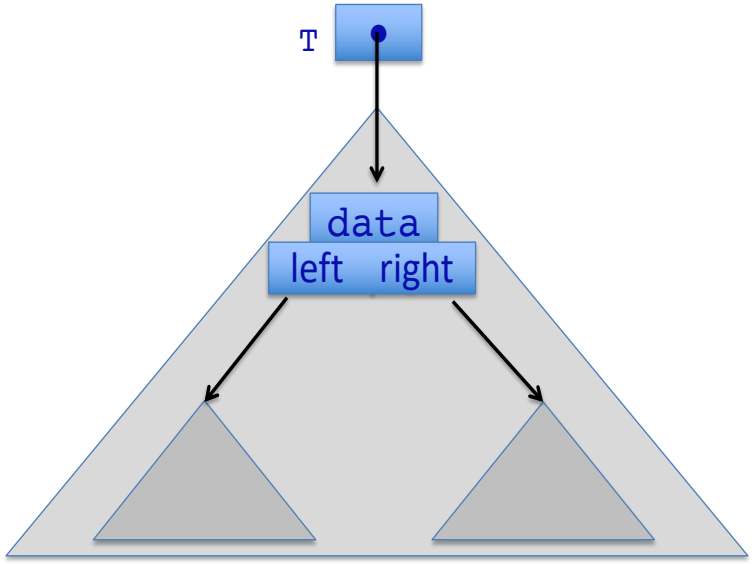
In practice:

- `void *data`

In demonstrations:

- `int key`





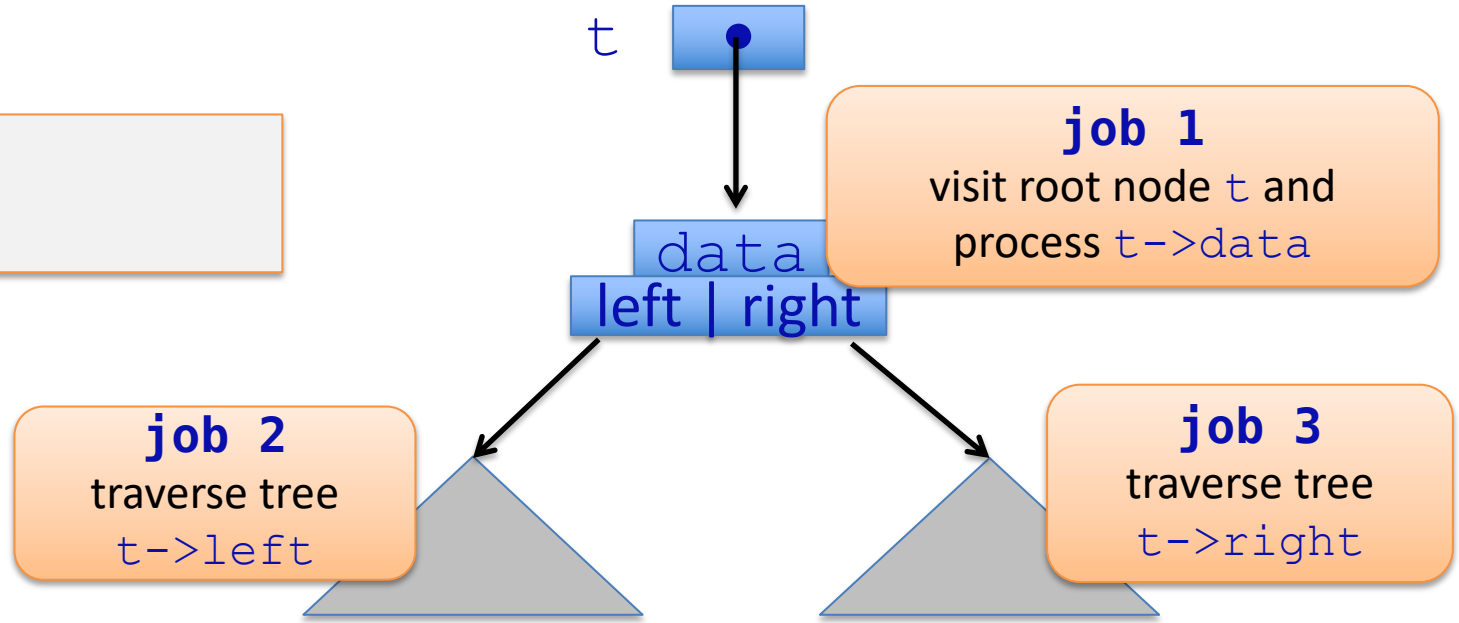
Tree/BST traversal= visiting all nodes of a tree

Tree traversal= visit all nodes of a tree in a systematic way.

For a non-empty tree , there are 3 **jobs** , and they can be done in any order!

traverse left/right tree:

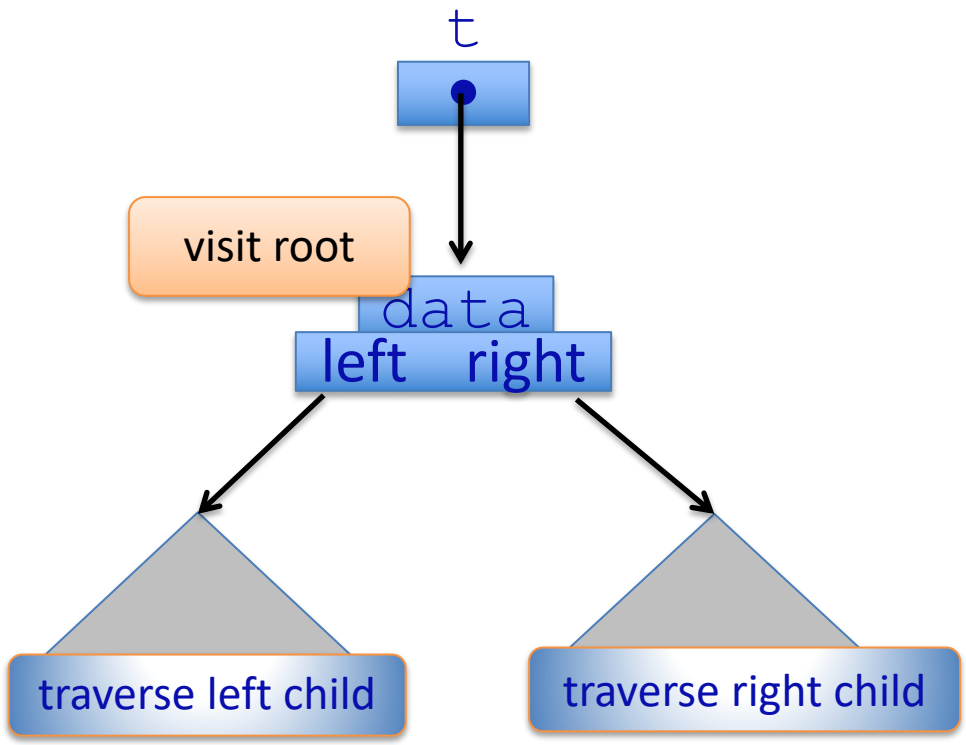
- normally done recursively

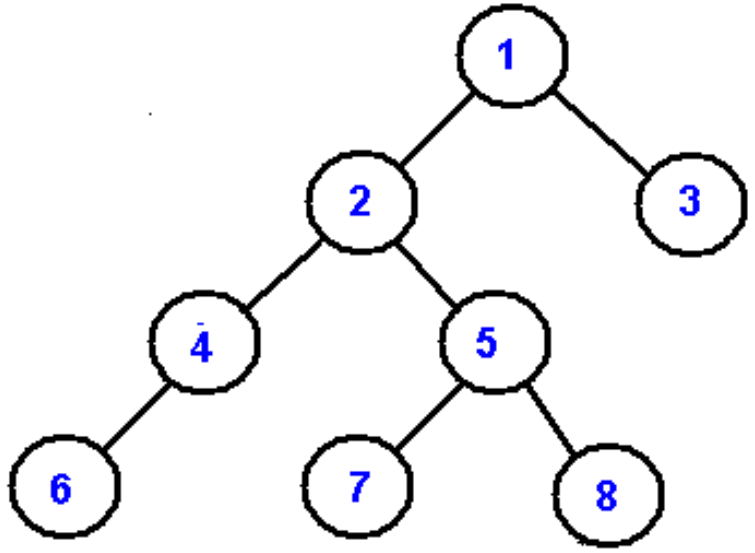


Tree/BST traversal | Example

Depending on when to visit the root node, we have:

- *pre-order* (visit root *before* children),
- *post-order* (visit root *after* children), and
- *in-order* (visit root *in between* children)

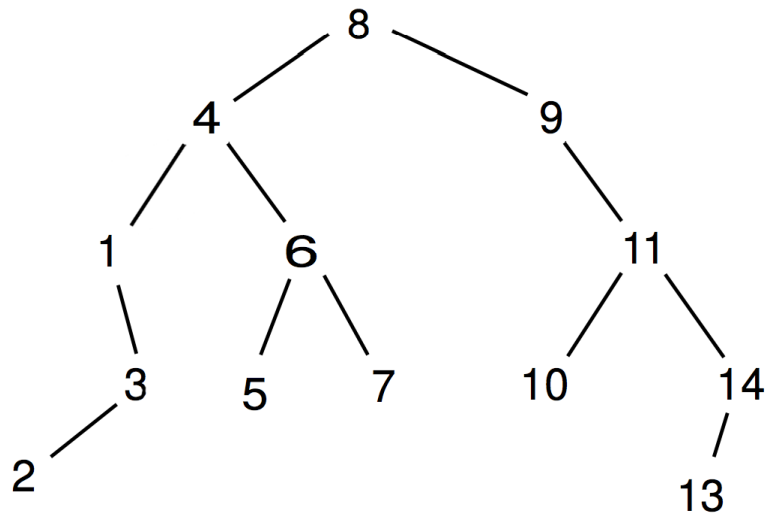




List the nodes in order visited by:

- in-order :
- pre-order :
- post-order :

BST



How to: search? insert? delete?
Complexity of
search (for a key)= ?,
insert (node with a given key)= ?,
delete (node of a given key) = ?

Exercise (supposing data is just `int key`)

Ex1: Write a C functions for:

- printing a BST's keys in increasing order
- printing a BST's keys in decreasing order

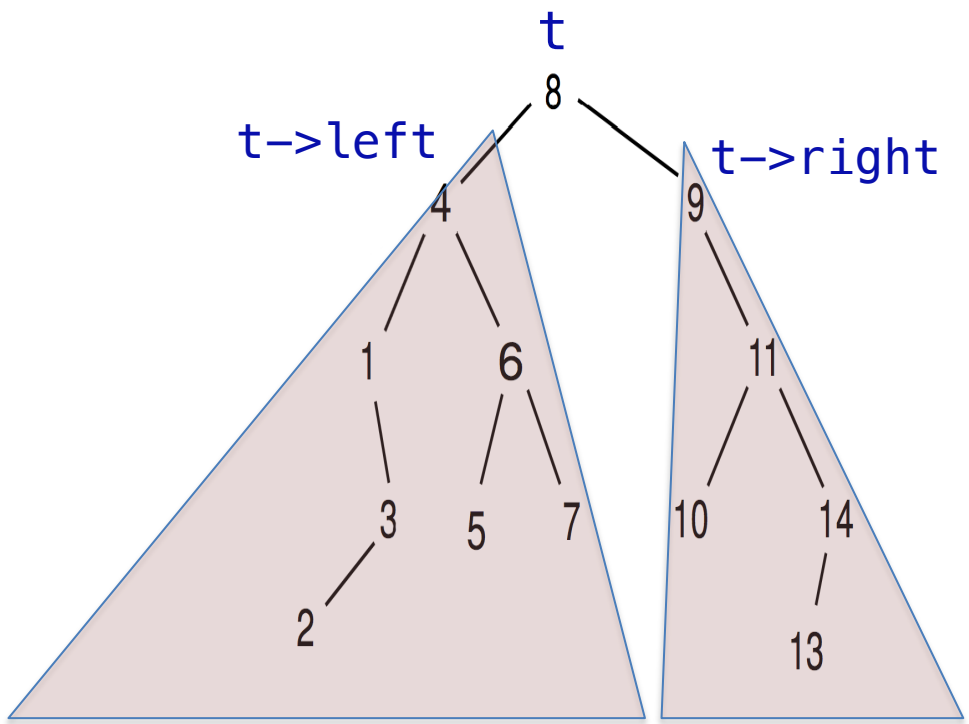
```
void printIncreasing(    ) {

}
```

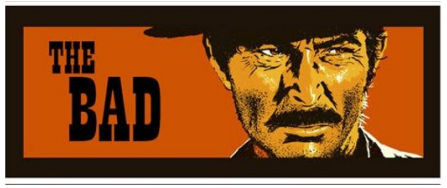
Ex2: What traversal order should be used for:

- copying a tree ?
- free a tree ?

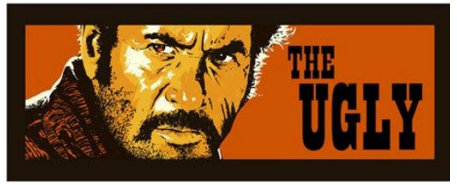
```
typedef struct bst* tree_t;
struct bst {
  int key;
  tree_t left;
  tree_t right;
};
```



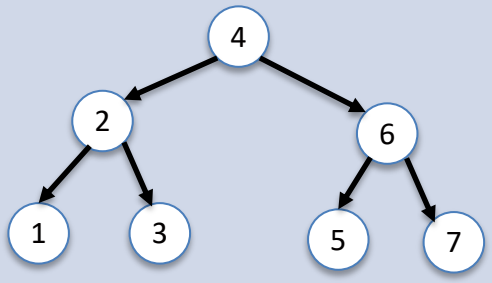
BST efficiency depends on the order of input data

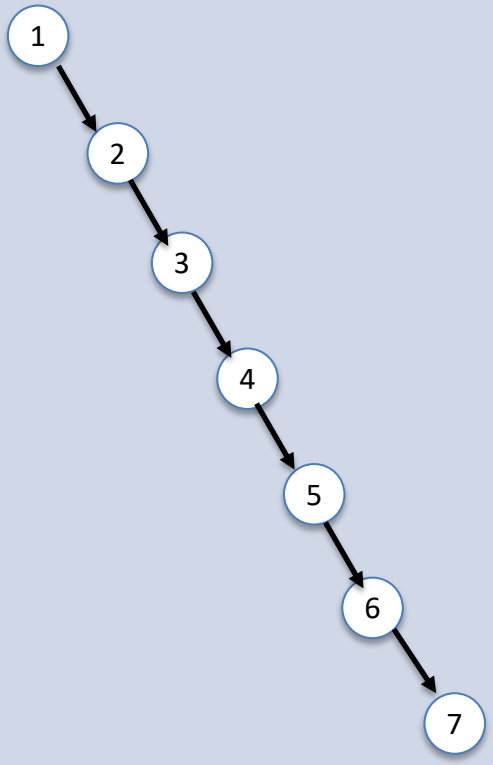


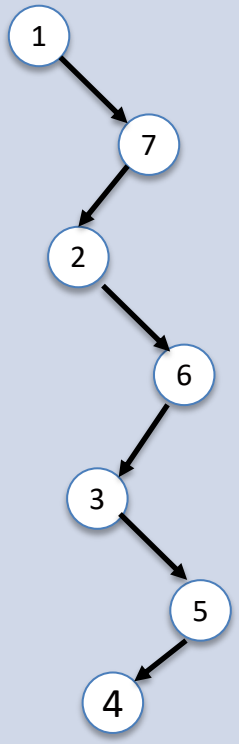
AND



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







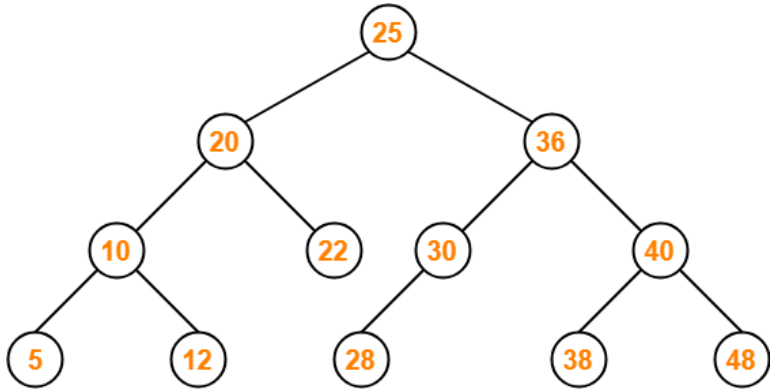
Want The Good, no matter what's the data input order? Use AVL (or …)!

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

The Good and the Bad of BST

The Good:

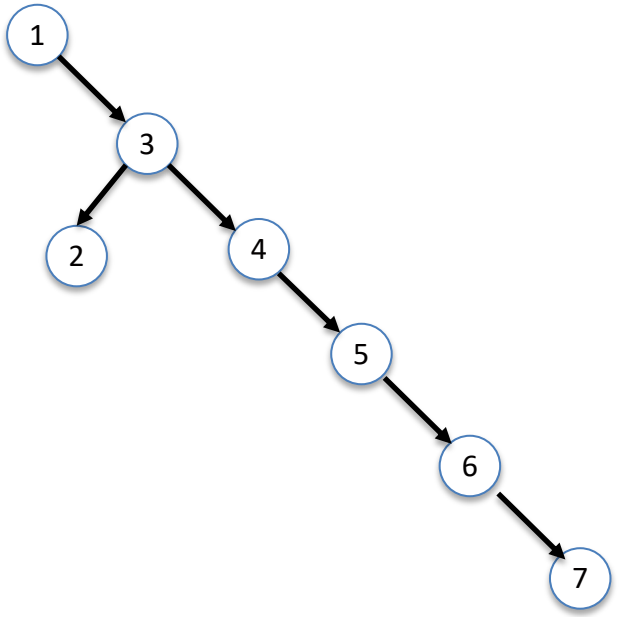
The Best and Average performance for `search, insert` and `delete` is `O(log n)`



Binary Search Tree

The height of the tree is around $\log_2 n$ *in average*

The Bad: in general, and in the worst cases `search, insert` and `delete` is `O(n)`



The height of the tree could be around `n`

Using Rotations to rebalance AVL

AVL= a BST which is always balanced ➔ O(log n) for search/insert/delete

How: re-balance BST when it becomes unbalanced

Using Rotations to rebalance a BST

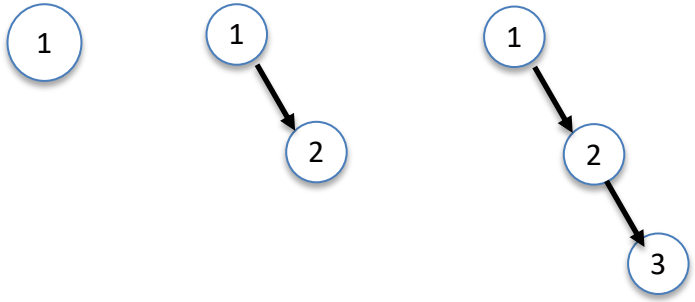
At the start: an empty BST is balanced

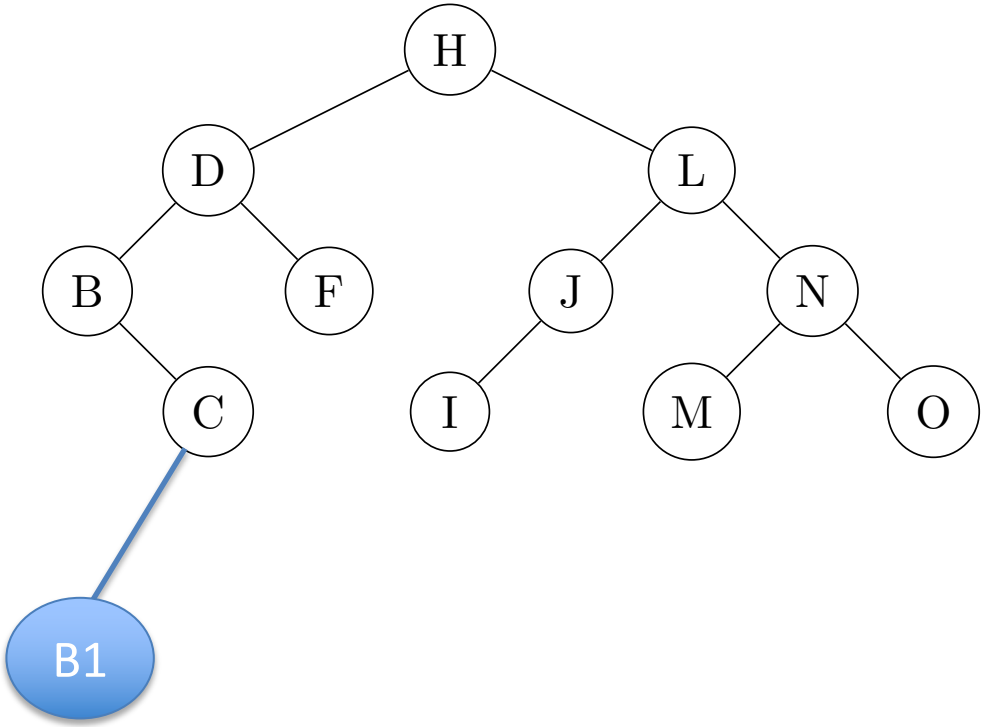
Problem: After a insertion/deletion, the resulted tree might become unbalanced

Approach: use Rotations to rebalance if needed after each insertion/deletion.

To rotate:

When? What? How?





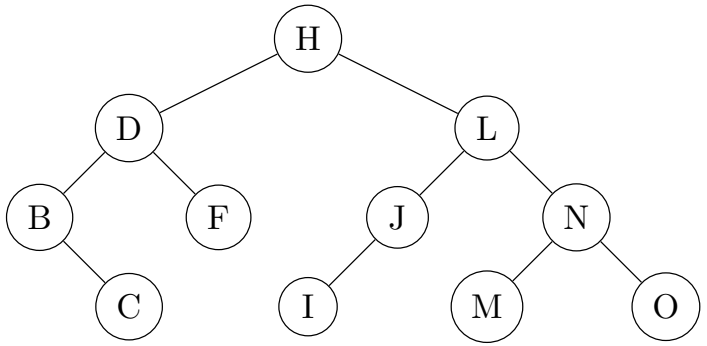
How to know if a node/tree is imbalanced?

A node is *balanced* iif the heights of its left tree and its right tree differ by at most 1

balance factor of a node = height of left – height of right

A tree is balanced iif each of its nodes is balanced.

is the tree balanced?



- Is this node balanced?

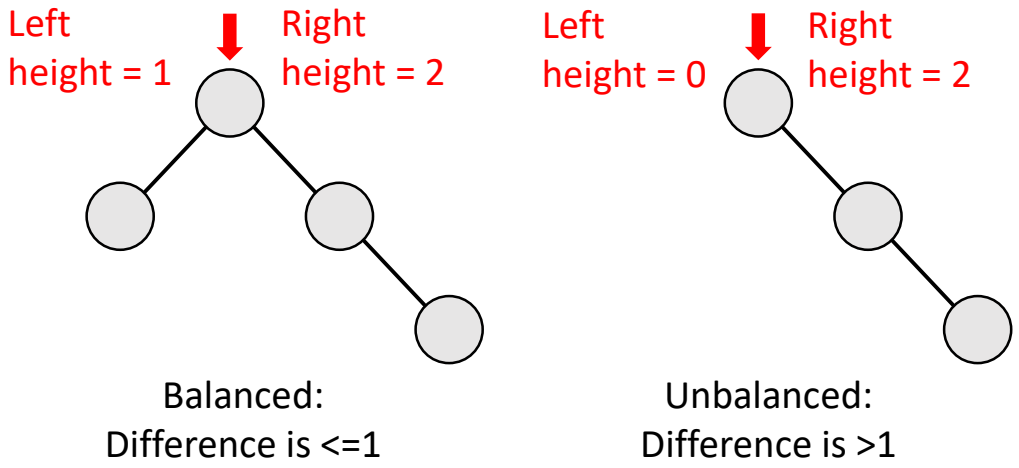
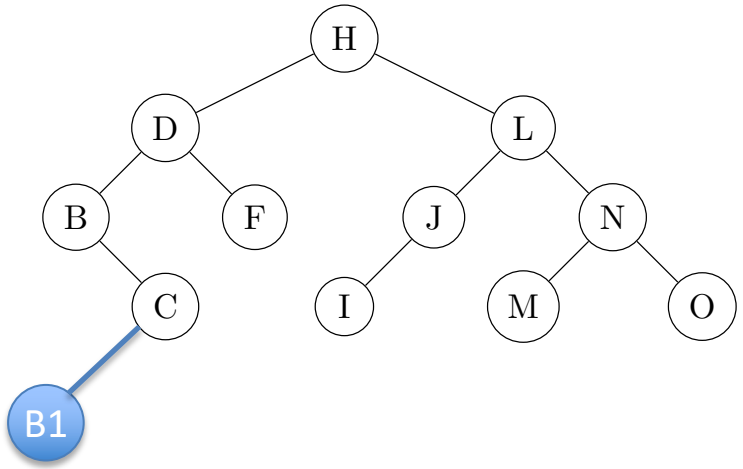


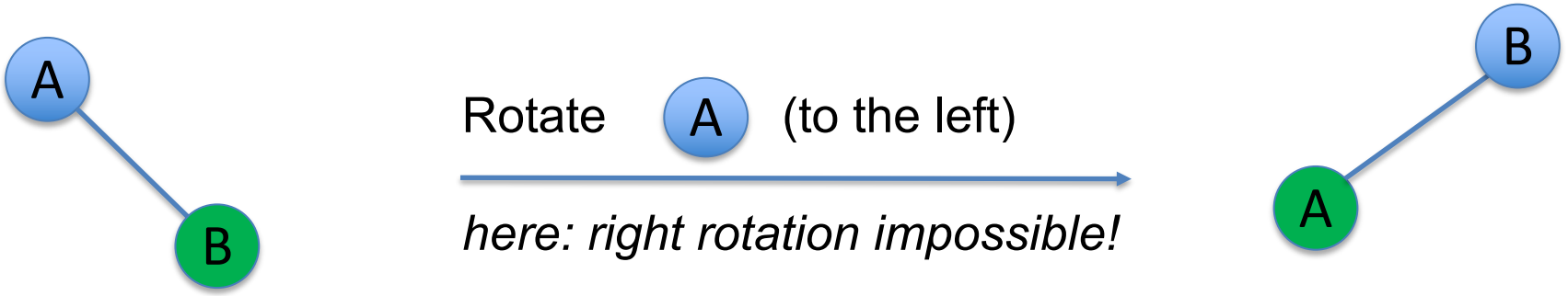
fig. from lecture slides



BST: what's a rotation

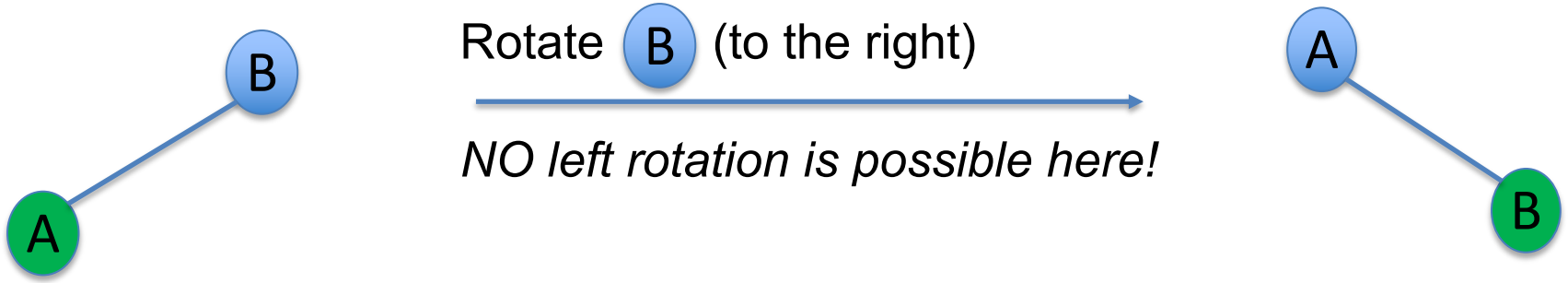
A *rotation reverses* the parent-child relationship of a parent and a child, but still maintaining the BST property.

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



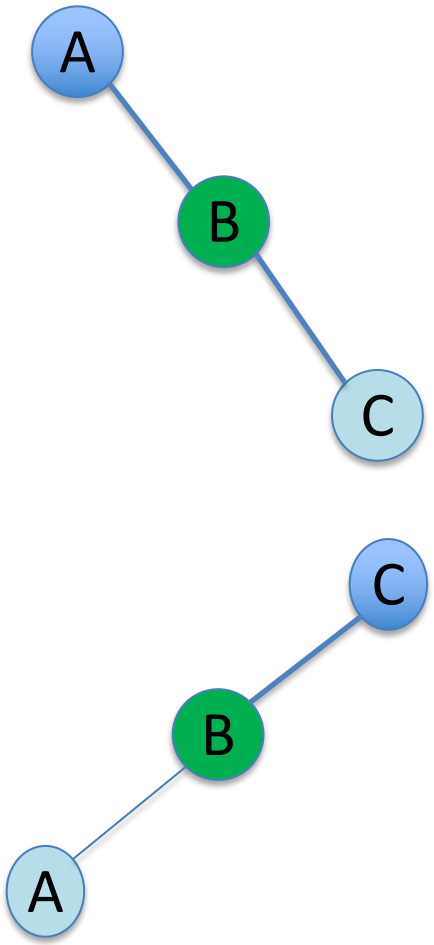
Note: we say that we ***rotate the parent node,*** although we rotate both parent and child.

right rotation for parent-leftChild: rotate parent down to the right



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:

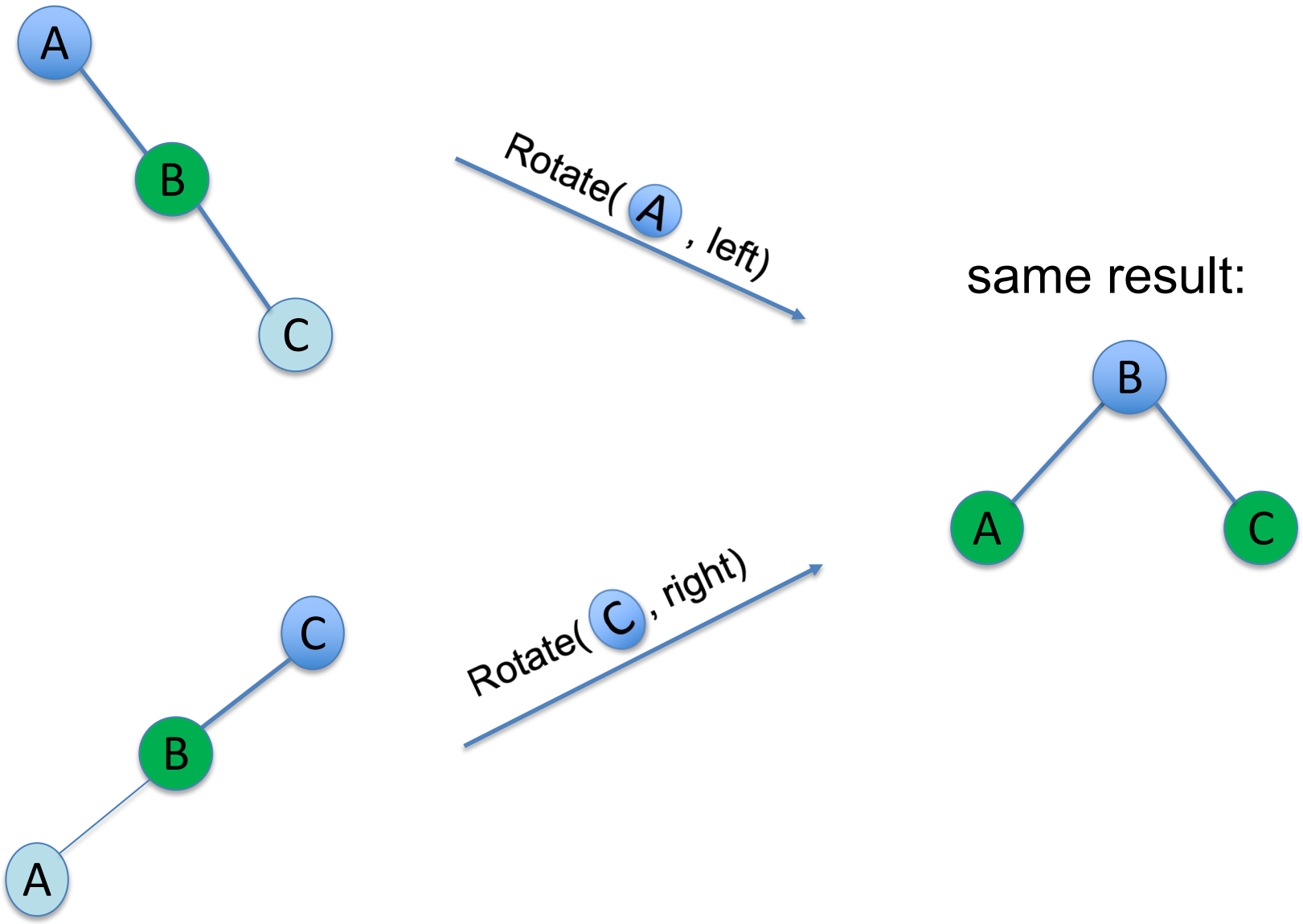
- Which subtree is unbalanced? Which node is the *unbalanced root* of that unbalanced subtree?
- Which rotation should be done for re-balancing?

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:

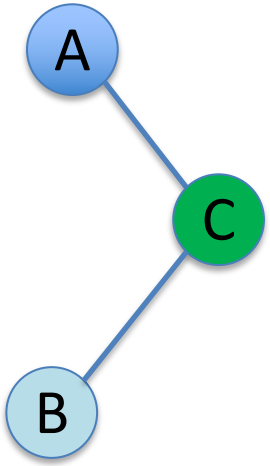
- not changing the children of the 3 involved nodes if possible,
- or, if otherwise, insert the “lost” child to the new root (ie. root “B” in the picture)

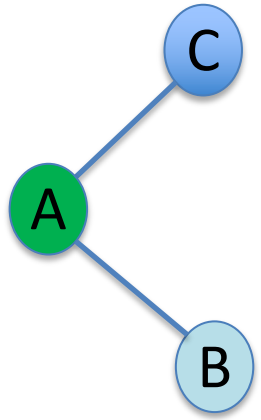
Examples: ...

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.

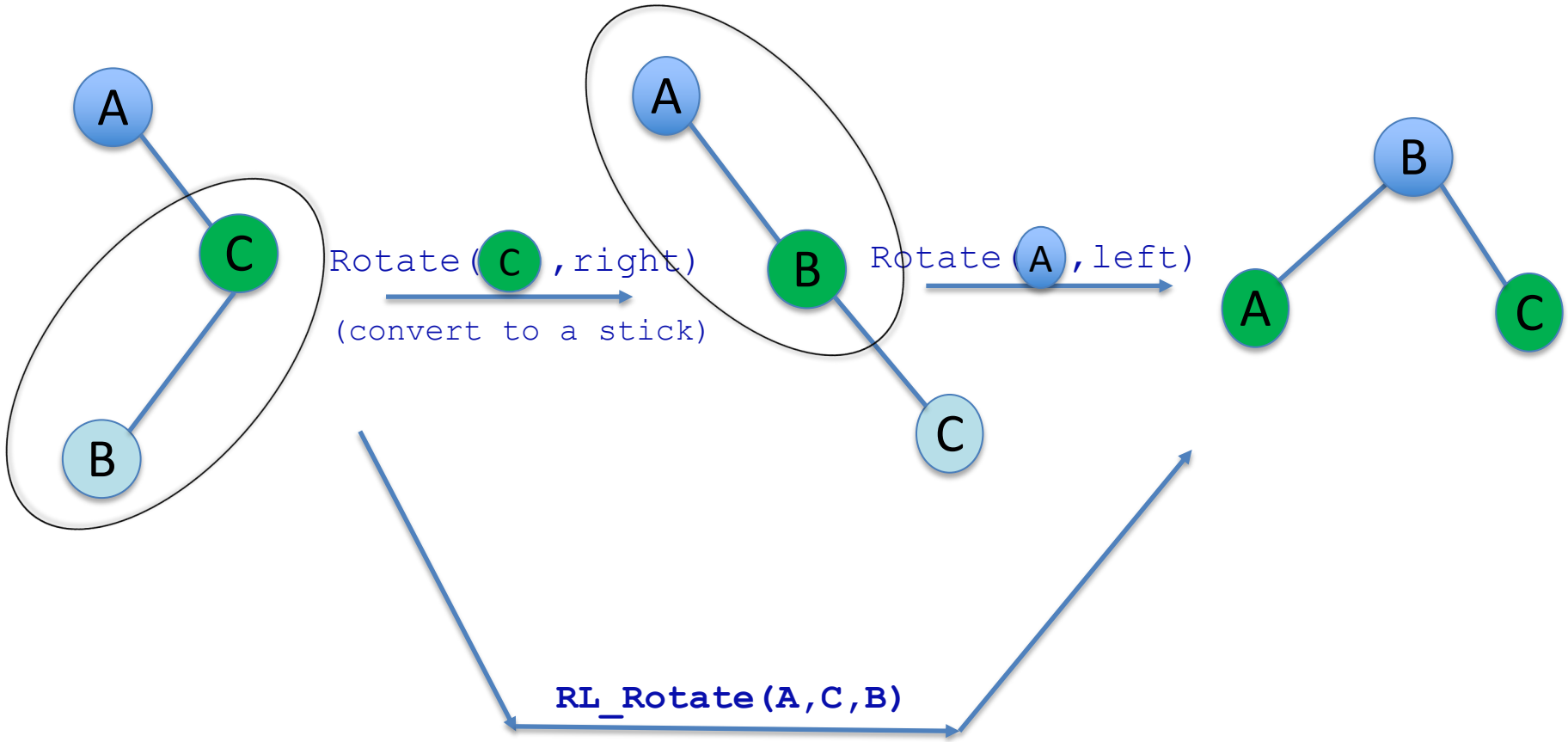
Rotation1:

- Rotate the **child (the middle node)** 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

Problem: When inserting a node, AVL might become unbalanced

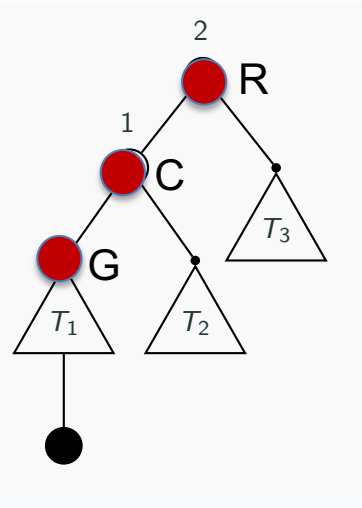
Approach: Rotate to re-balance (Rotate WHAT?, and HOW?)

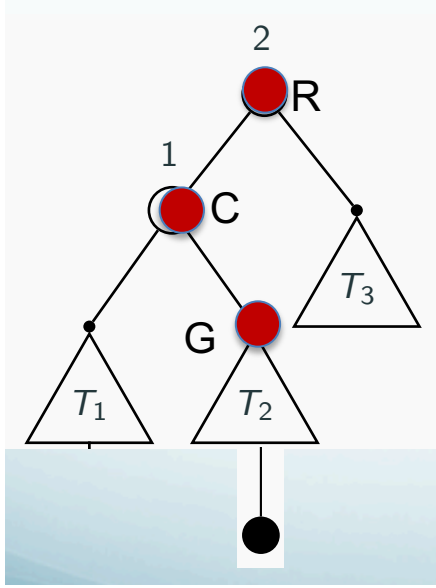
Rotate WHAT?

- Walk up, find the *lowest* subtree R which is unbalanced

HOW

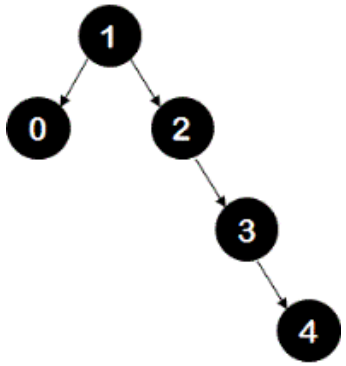
- Consider *the first 3 nodes* R→C→G in the path from root R to the just-inserted node
- Apply a single rotations if that path is a stick, double rotation otherwise



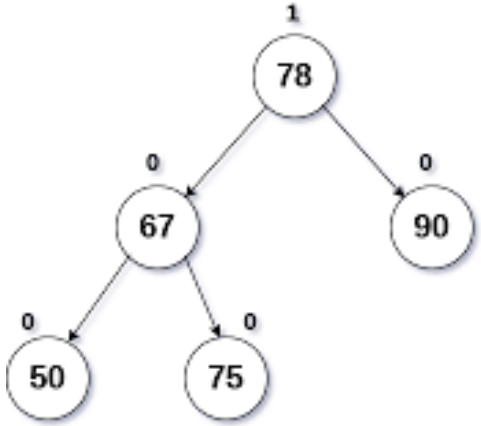


Examples: do rotation to keep the BST balanced after insertion

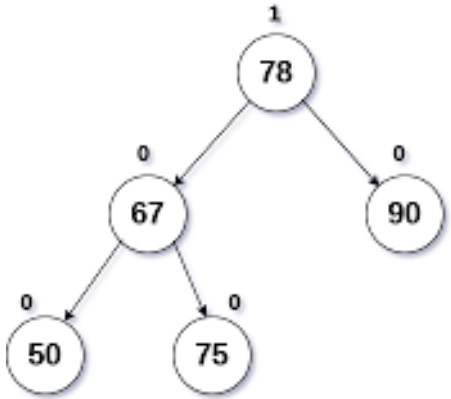
need rebalancing? if yes, what rotation on which node?



same question for the following tree after: insert 60?



insert 70?



Is AVL better than BST?
Do Peer Activities W5.10 and fill in the table below

Operation	Case	Search Trees BST | AVL	
Insert	Average	O( )	O( )
	Worst	O( )	O( )
Search	Average	O( )	O( )
	Worst	O( )	O( )
Delete	Average	O( )	O( )
	Worst	O( )	O( )

Lab : How to implement bstInsert?

Work out different versions of Function Header, then do W5.

```
???   bstInsert( ???     );
```

LAB Discussion: How to implement bstInsert? Is this code correct? Why?

```
tree_t bstInsert(tree_t t, int key) {
  if (t==NULL) {
    t= malloc(*t); assert(t);
    t->key= key;
    t->left= t->right= NULL;

  } else if (key < t->key)
        bstInsert(t->left, key);
  else
        bstInsert(t->right, key);
  return t;
}
```

Example of use:

```
tree_t t= NULL;
for (i=1; i<=5; i++) t= bstInsert(t, (i*10)%7);
// that will insert 3,6,2,5,1
```

Notes: Don't like `tree_t`? Replace all `tree_t` with `struct bst *`

Next week: prepare for the A2 Q&A