COMP20003 Workshop Week 5

Binary Search Trees + AVL

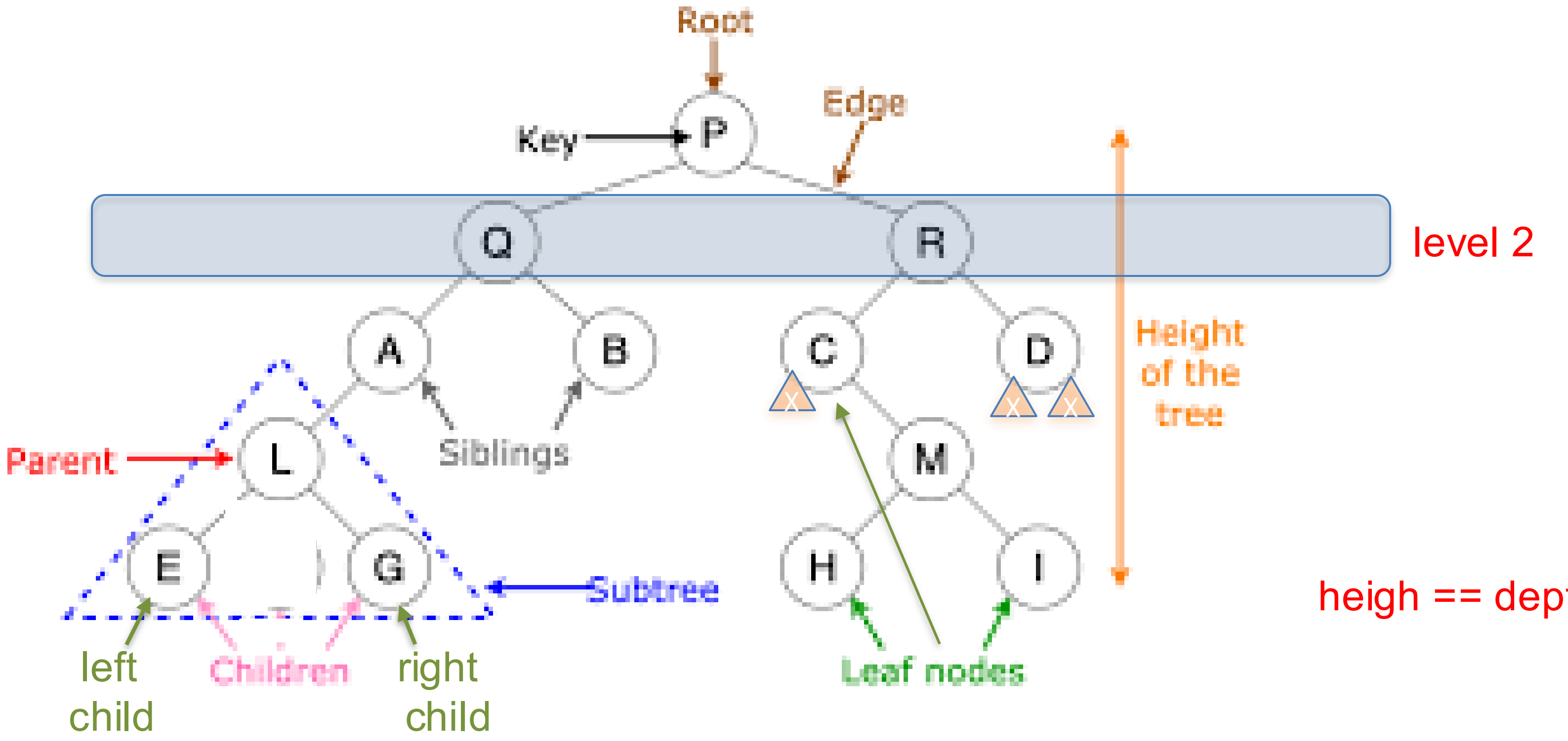
- Binary Trees & Traversal
- BST
- AVL & Rotations

Lab:

- implementing bst_insert
- review Weeks 1-4 using sample MST papers

- Patricia Tries and Assignment 2

Binary Trees: some jargons



heigh == depth == largest level in tree == 5

Notes:

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

Declaring trees: declaration examples

Possible def	Notes

`struct bst {`
`    data_t data;`
`    struct bst *left;`
`    struct bst *right;`
`};`
	a tree node has
	• a data
	• a left child (aka. *left sub-tree*)
	• a right child (aka. *right sub-tree*)

`struct bst *t= NULL;`	this line creates the empty tree t

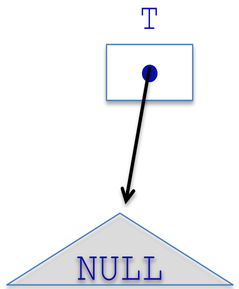
Note: often data_t includes a special field key. And data is:

In practice:

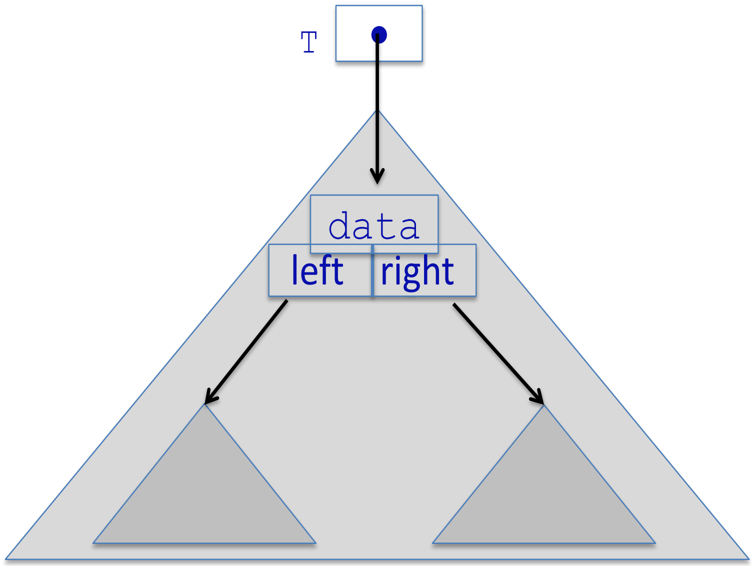
- void *data, or
- data_t *data

In demonstrations:

- int key



an empty tree



a non-empty tree

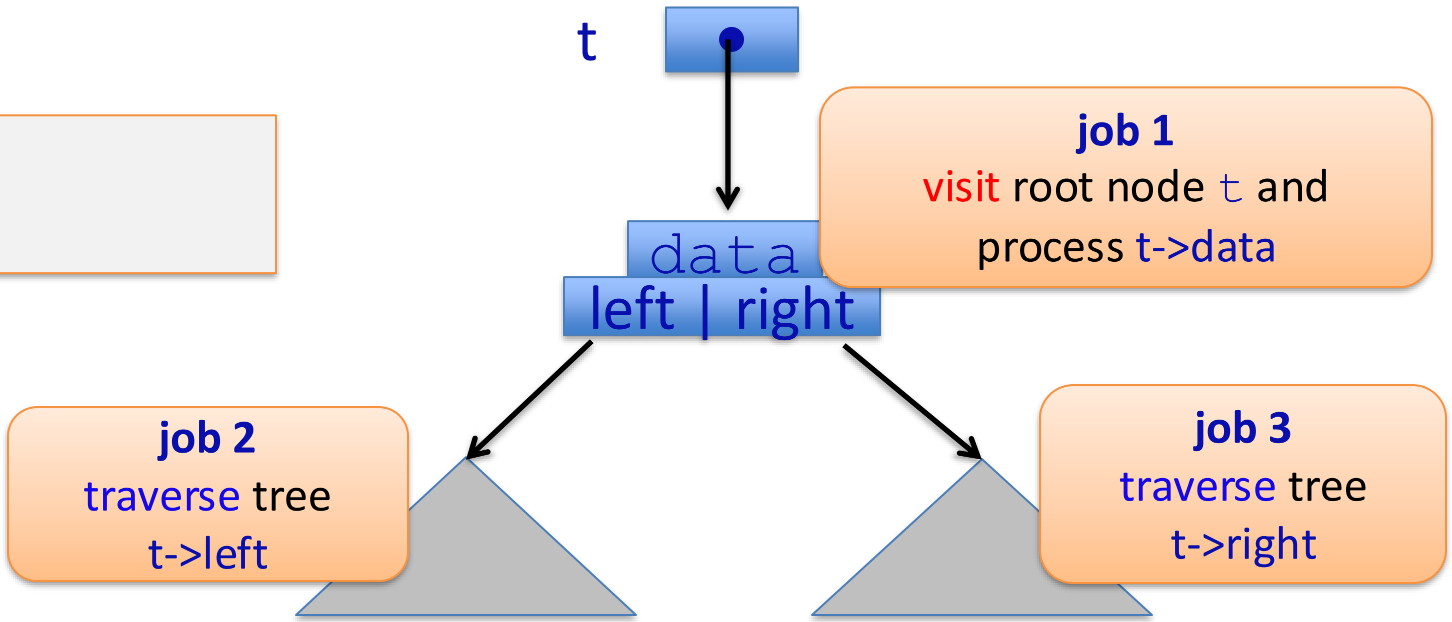
Tree/BST traversal= visiting all nodes of a tree

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

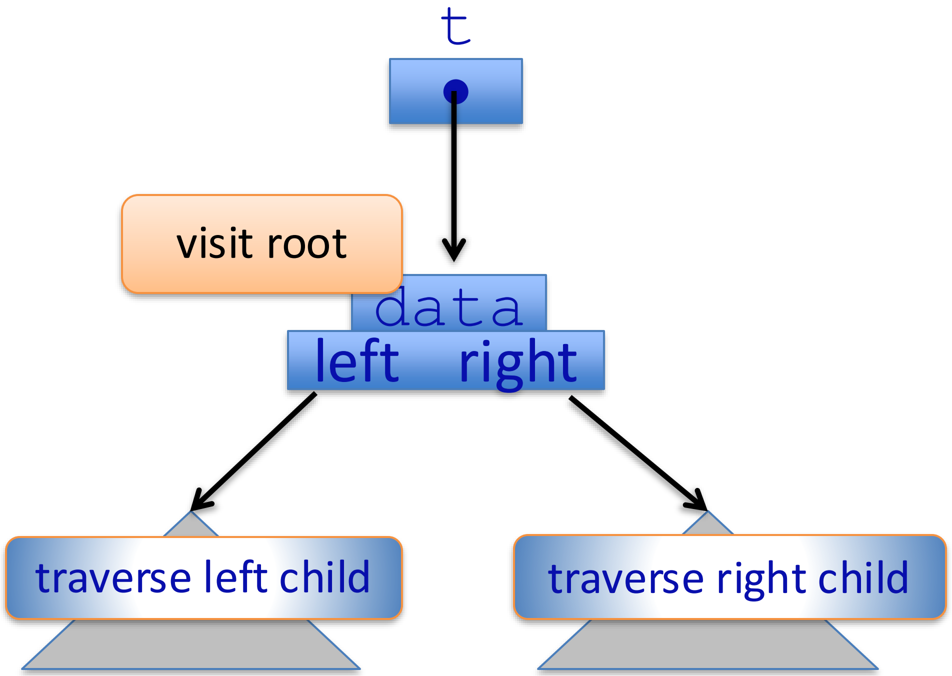


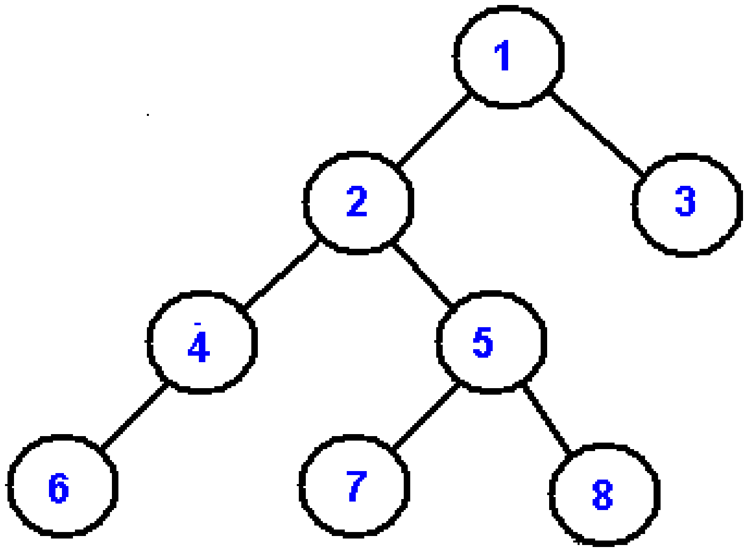
Tree/BST traversal | Example

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

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

Note: Children are normally traversed in the letf-right order, but can also be in the right-left order.

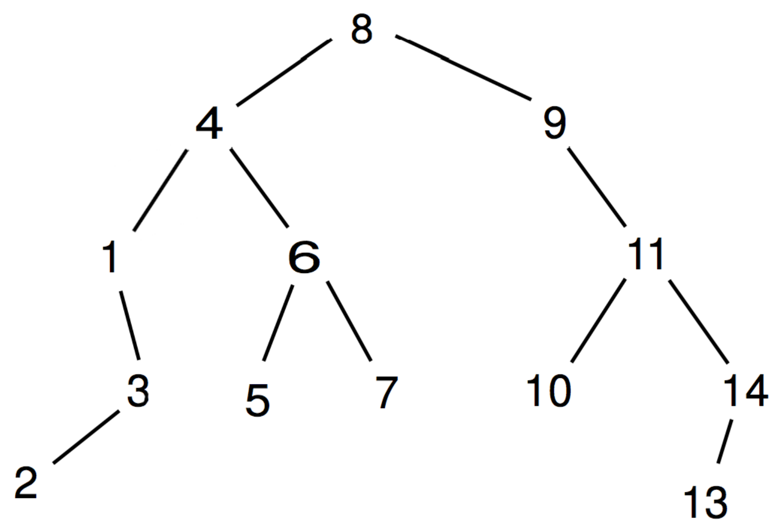




List the nodes in order visited by:

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

BST



Review:
- What's a 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`)

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

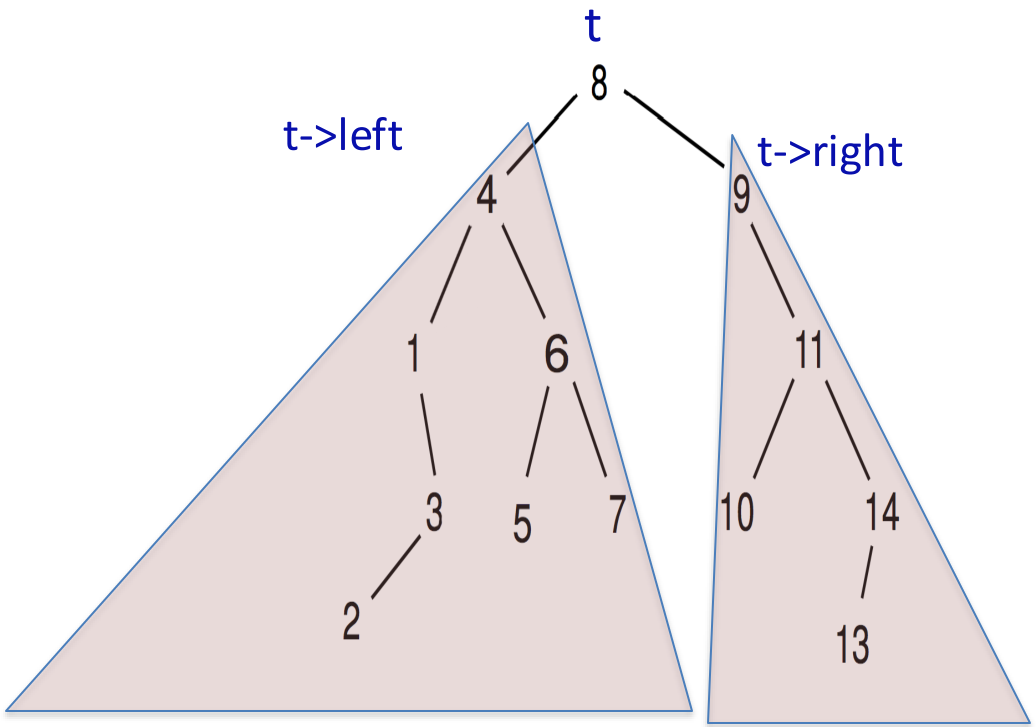
Ex1: Write a C functions for:

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

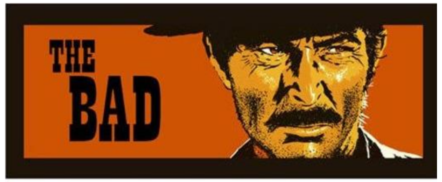
```
??? printIncreasing( ??? ) {

}
```

Ex2 : What traversal order should be used for:

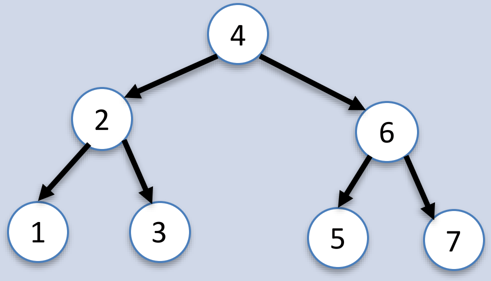
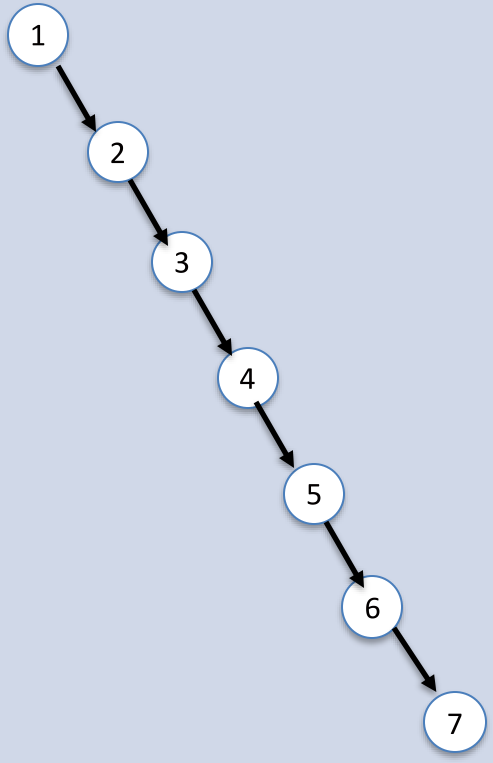
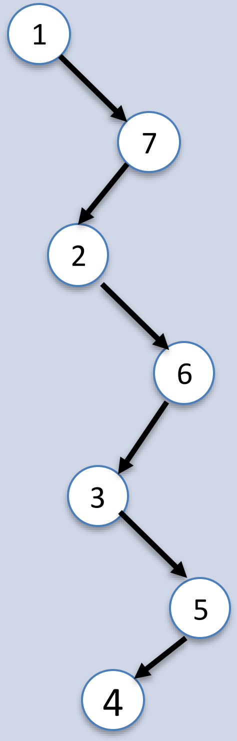
- copying a tree ?
- free a tree ?



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
		

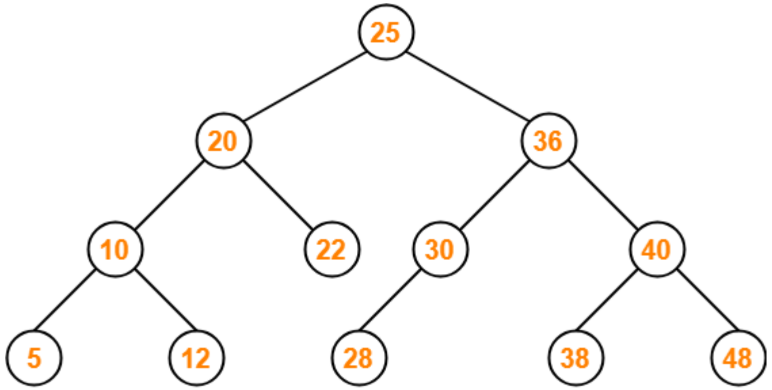
Want The Good, no matter what's the data input order? Use a tree which is always "balanced"!

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

Why AVL? The Good and the Bad of BST

The Good:

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



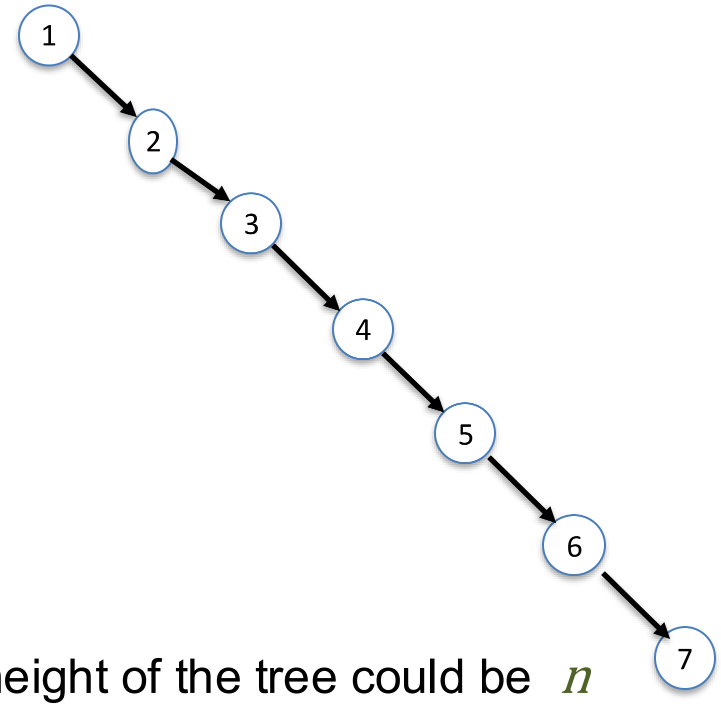
Binary Search Tree

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

The Bad:

in general:

search, insert and delete is $O(n)$



The height of the tree could be n

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

How: re-balance BST whenever it becomes unbalanced

How to know if a node/tree is imbalanced?

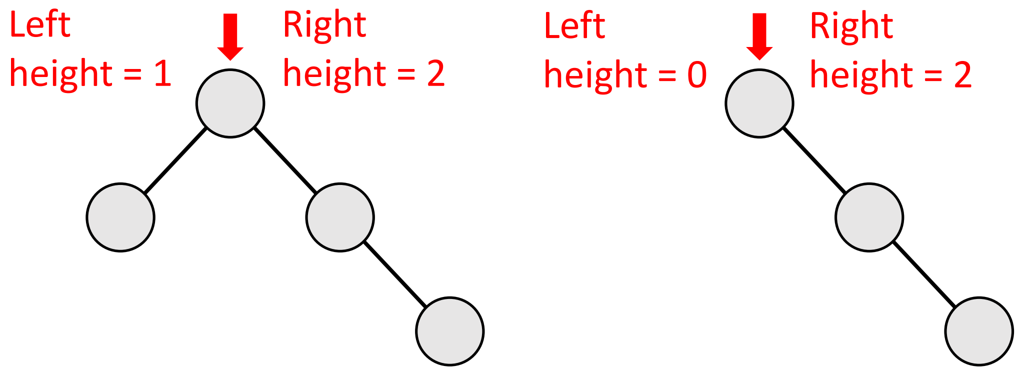
A node is *balanced* iif the heights of its left tree and its right tree differ by at most 1

- using *balance factor* of a node (aka. *counter*)
- counter = left height – right height

A tree is balanced iif each of its nodes is balanced, ie.:

for *each node*: difference= |counter| ≤ 1

- Is this node balanced?

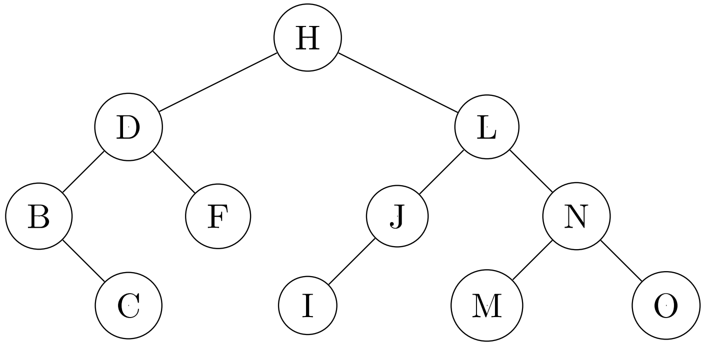


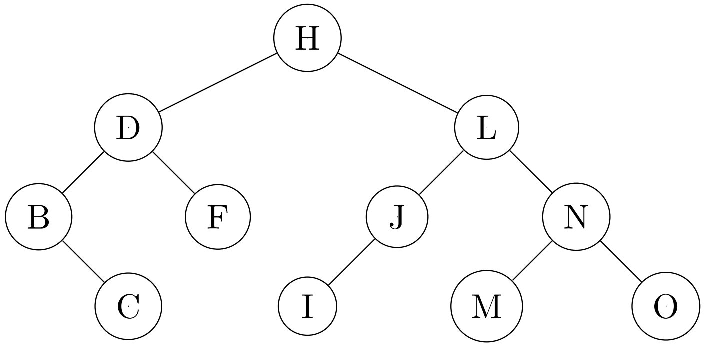
Balanced:
Difference is <=1

Unbalanced:
Difference is >1

fig. from lecture slides

is the tree balanced?



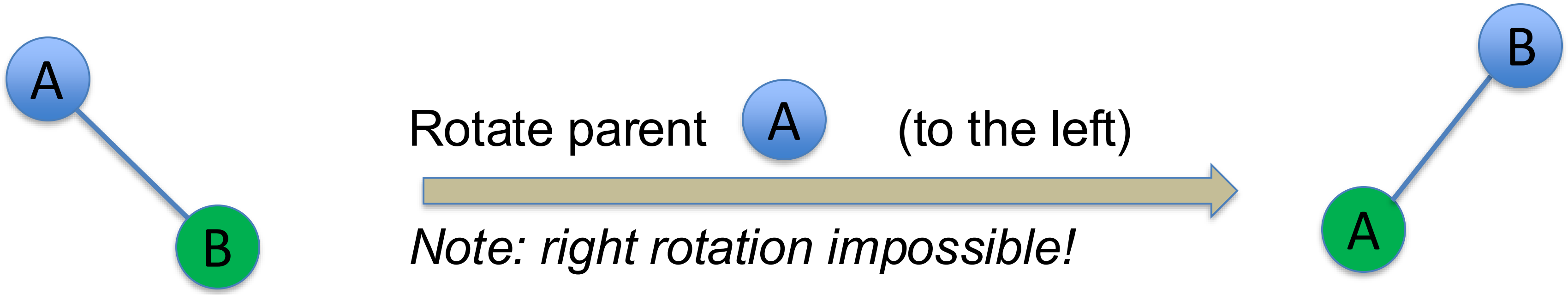


B1

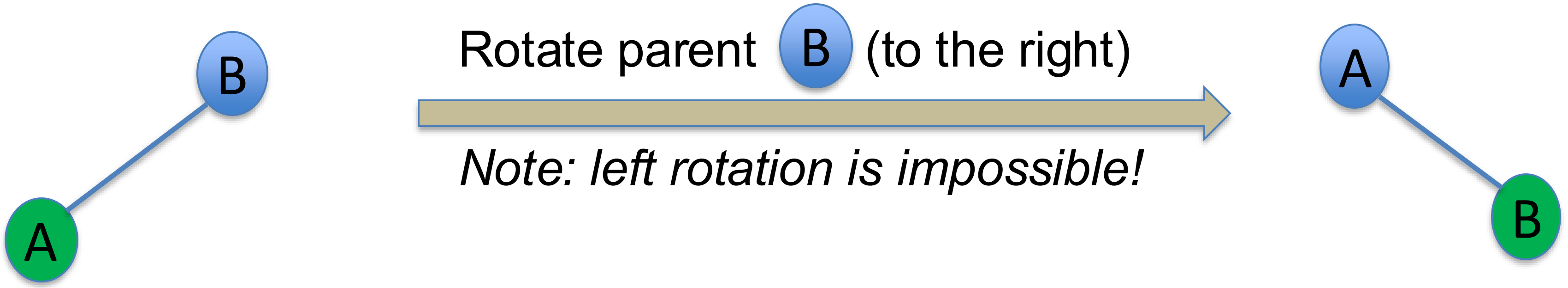
BST: what's a rotation

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 ((left) parent becomes left child)



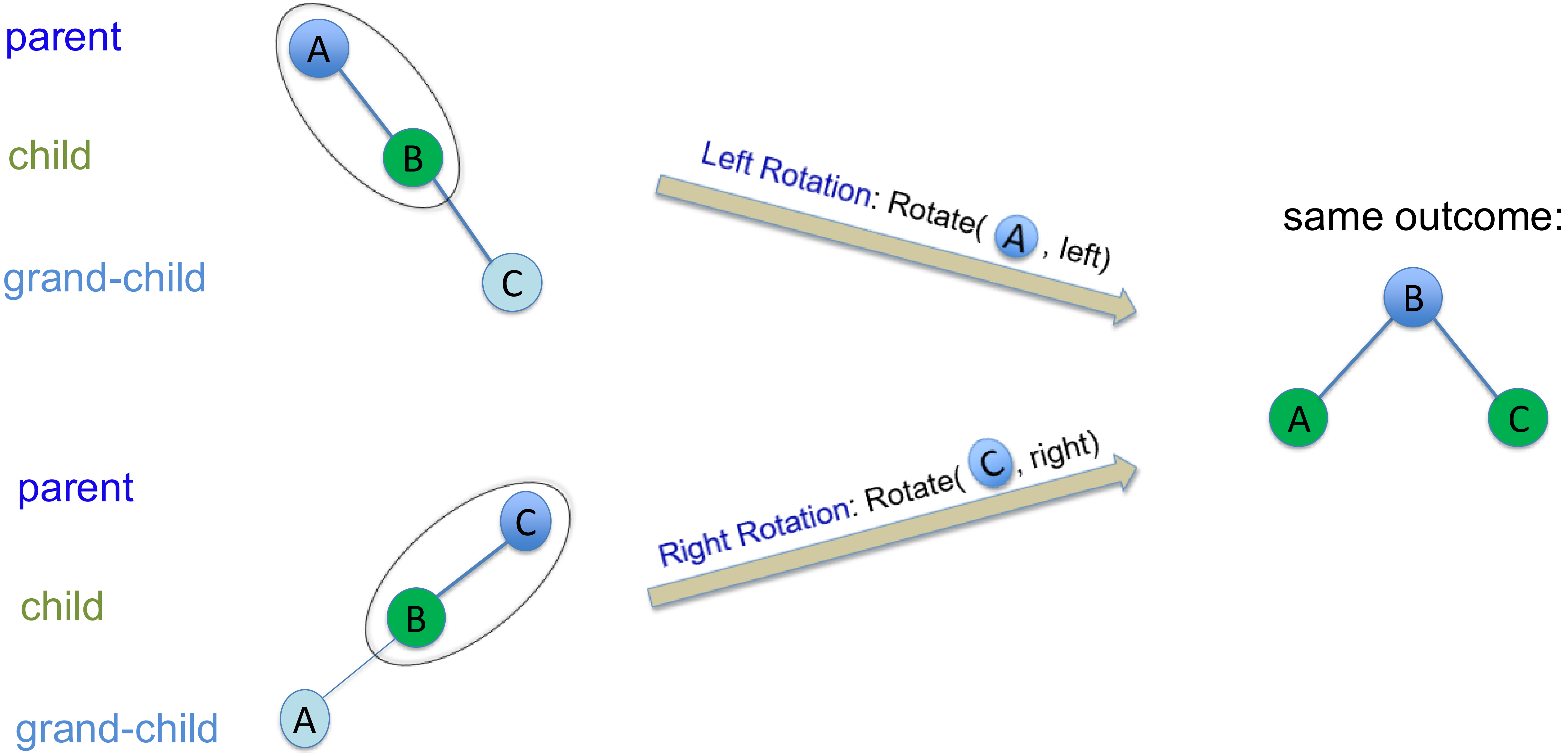
right rotation for parent-leftChild: rotate parent down to the right



Note: we say that we ***rotate the parent node*** = using the child node as the axe and *rotate the parent* node.

AVL: Two Basic Rotations: 1) Single Rotation

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



AVL: Two Basic Rotations: 2) Double Rotation

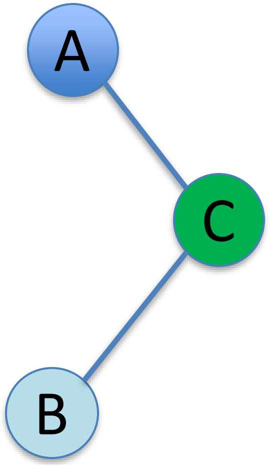
Applied when an unbalanced 3-node AVL subtree has a non-stick (that is, zig-zag) form.

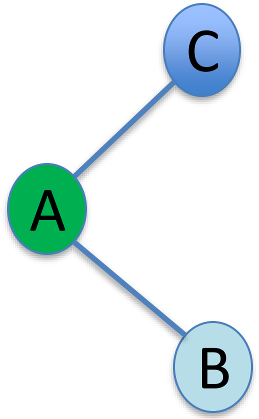
Two cases: (a) (b)

parent

child

grand-child





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

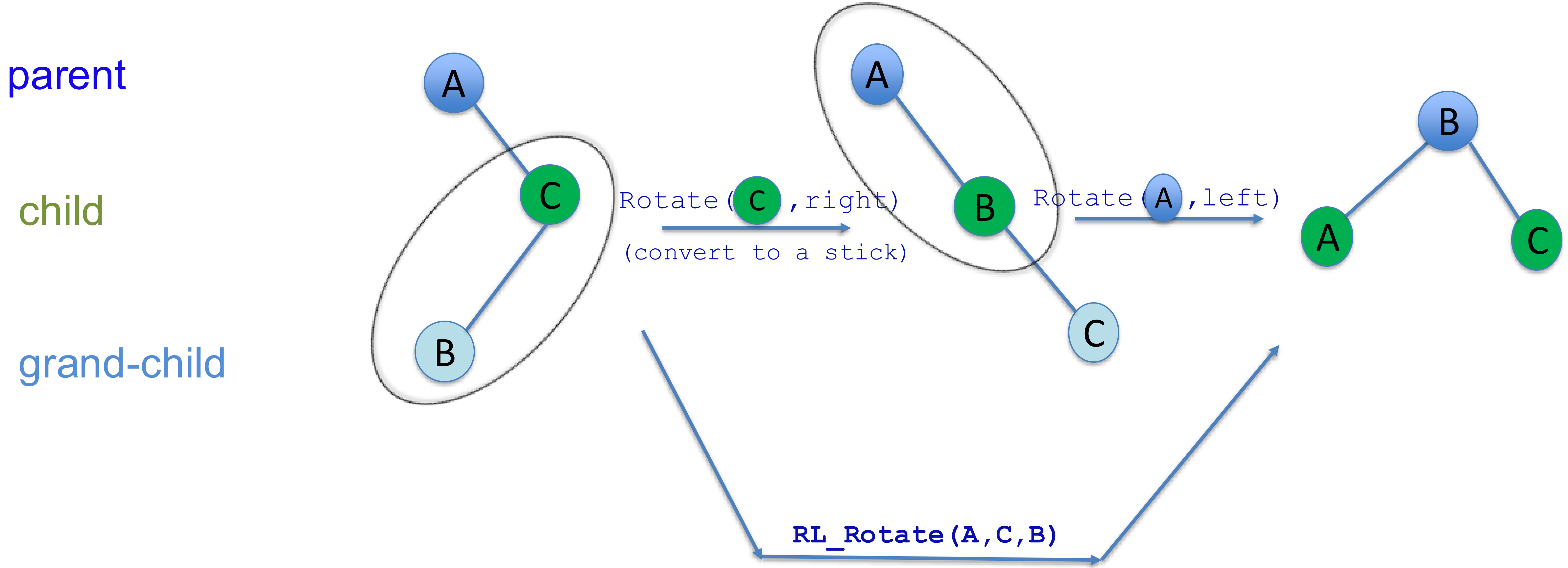
Rotation1:

- Rotate the **Child (the middle node)** of the unbalanced root and turn the tree to a stick

Rotation2:

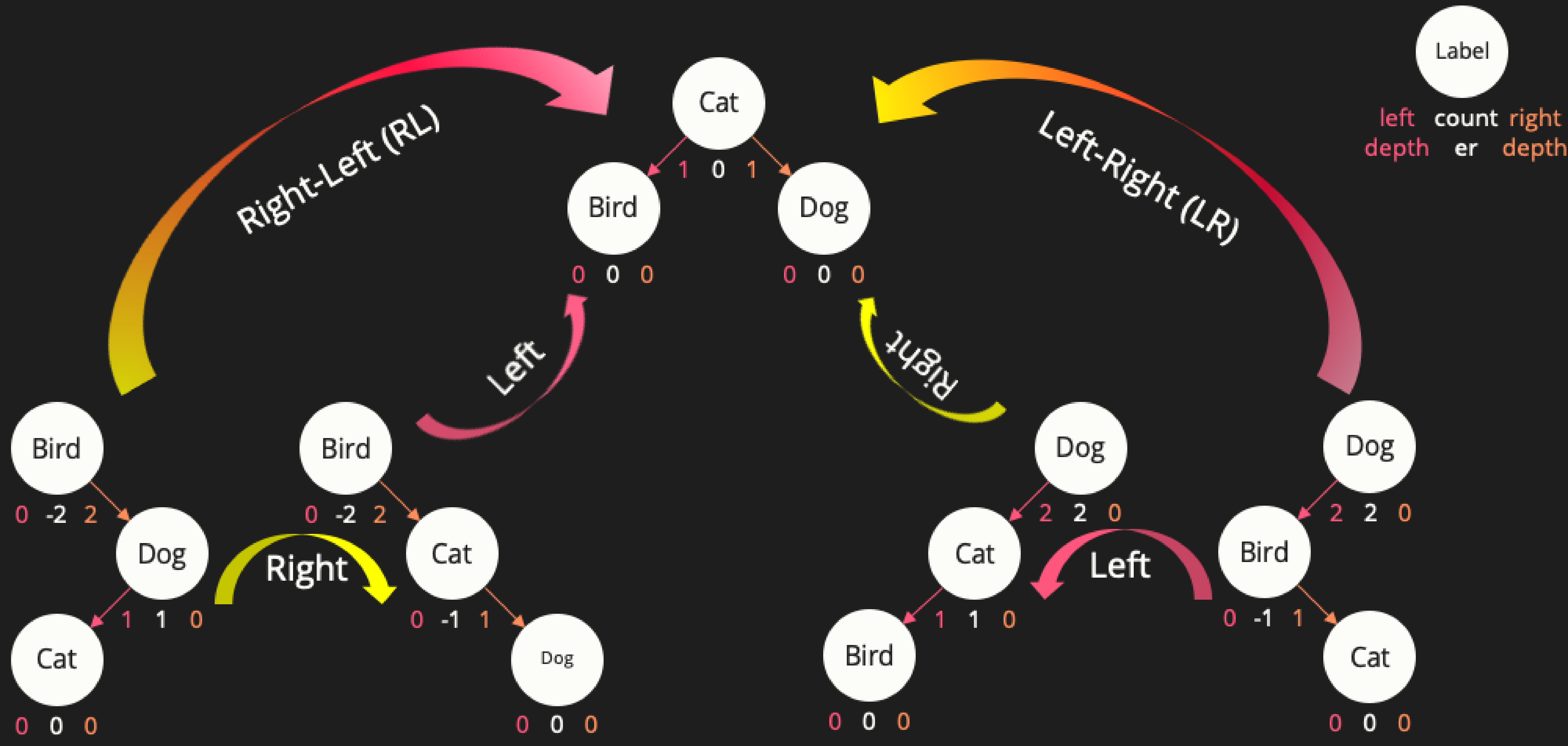
- Rotate the **unbalanced root** of the new stick.

Double Rotation Example: RL rotation



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

AVL Basic Rotations



AVL: Using Rotations to rebalance AVL

Problem: When *inserting* a node, AVL might become unbalanced

Approach: Rotate to re-balance

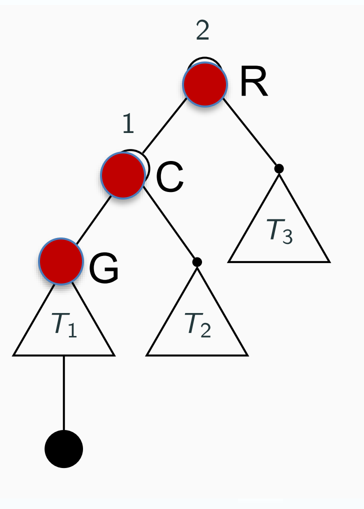
Related questions: Rotate WHAT?, HOW?)

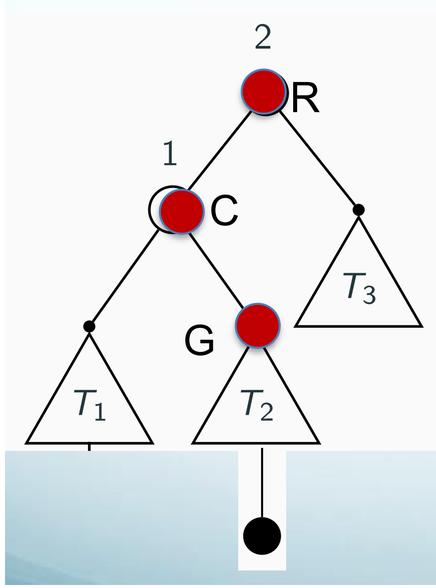
Rotate WHAT?

- Walk up from the new node, find the *lowest* subtree Root which is unbalanced

HOW

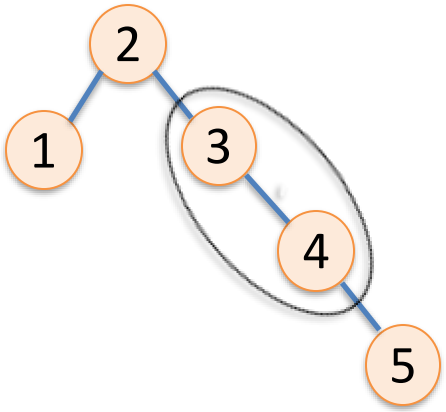
- Consider *the first 3 nodes* R→Child→Grand-child in the path from root R to the new node
- Apply a single (Left or Right) Rotation if that path is a stick, double (LR or RL) Rotation otherwise
- *Note*: when doing manually, focus on rotating the red nodes alone, and add the other nodes later



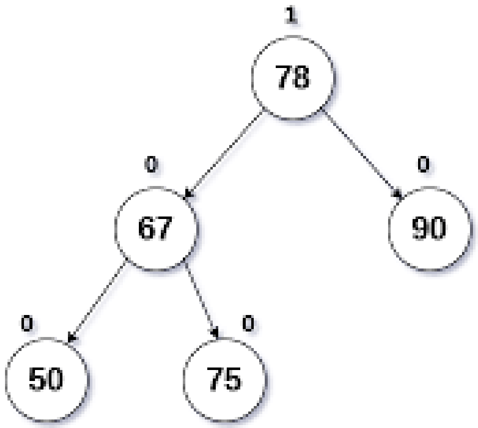


Examples: do rotation to keep the BST balanced after insertion

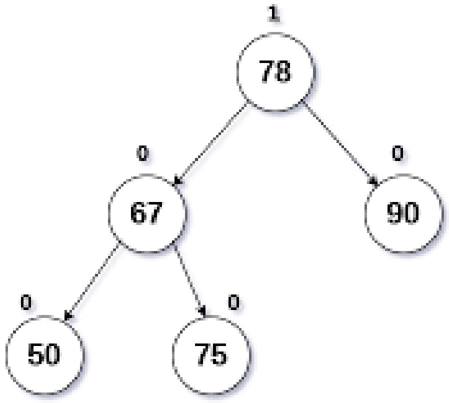
need rebalancing? if yes, what rotation on which node?



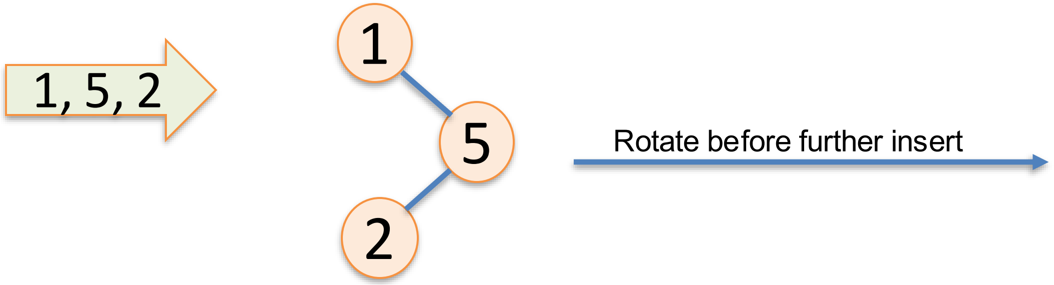
same question for the following tree after: insert 60?



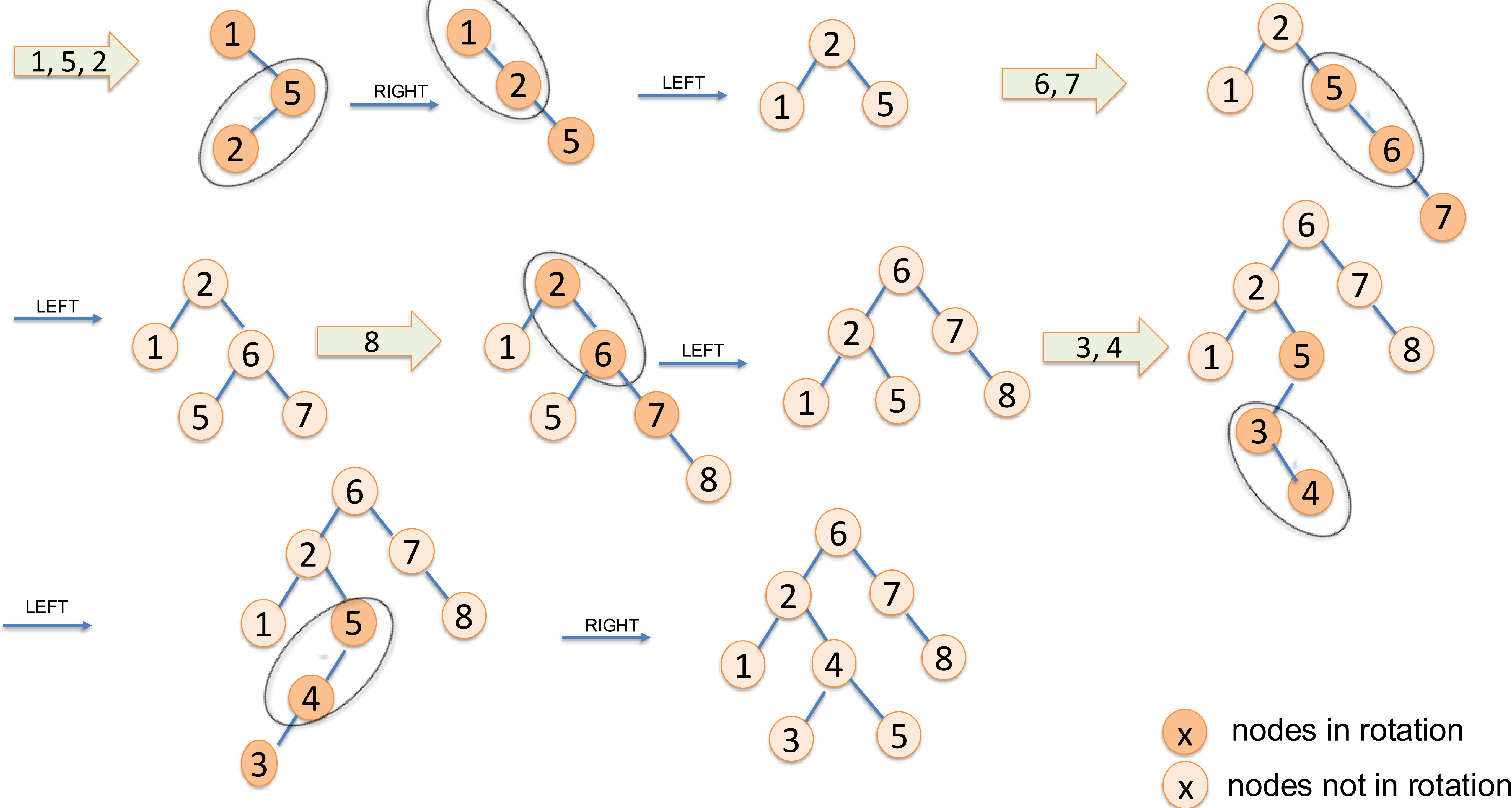
insert 70?



W5.4: Insert the following keys into an initially empty AVL tree: 1 5 2 6 7 8 3 4

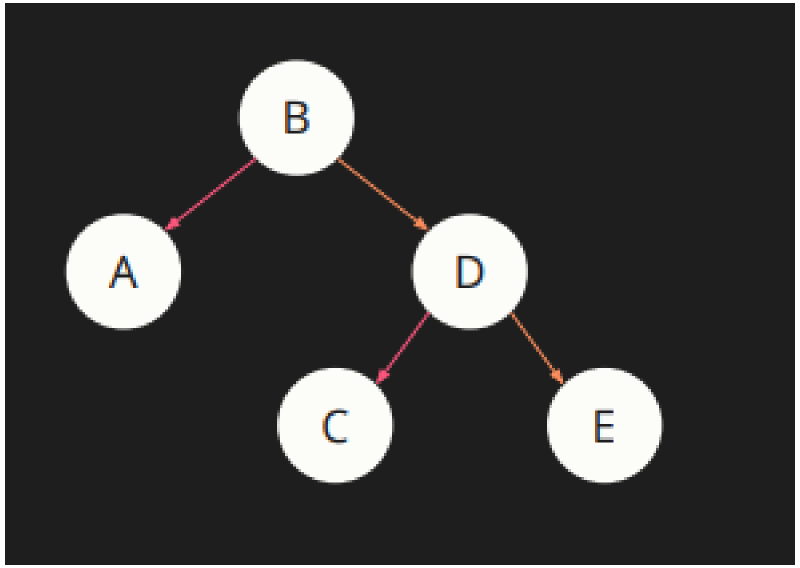


W5.4: Insert the following keys into an initially empty AVL tree: 1 5 2 6 7 8 3 4



Do Peer Activity W5.10 and then fill in the table

W5.10: What rebalancing rotation needs to be done after inserting node *F* into this AVL tree?



A	LR rotation on B–D–C
B	L rotation on B–D
C	RL rotation on B–D–C
D	R rotation on B–D

Operation	Case	Complexity for	
		BST	AVL
Insert	Average	O(log n)	O(log n)
	General	O( )	O( )
Search	Average	O( )	O( )
	General	O( )	O( )
Delete	Average	O( )	O( )
	General	O( )	O( )

Lab : How to implement bstInsert?

Start by discussing with your neighbours:

What should be the function header?

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

LAB Discussion: bstInsert? Is this code correct? Why?

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

tree_t *bstInsert(tree_t *t, int key) {
  if (t==NULL) {
    //  t= malloc a new node and set its value to {key,NULL,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
```

- Do & Finish A1
- Get Week 4 ✓
- Questions on sample MST papers W5.2.(a,b,c)
- Questions on A2 ?

Another Search Tree: Patricia Trie for Bit Strings

Insert {“A”,1}, {“B”,2}, {“A”,3}, {“ABBA”, 4}, {“AA”,5}

Notes: ASCII for ‘A’ is 0100 0001 (valued 65)

‘B’ 0100 0010

Bit pattern of “A” is 0100 0001 0000 0000 (array of 16 bits)