Binary Search Trees

Data Structures and Algorithms

Warm Up

What is the runtime for get, put, and remove of an ArrayDictionary?

Can you think of a way of making it better?

Finding your partner

Your repository will be titled

project1-NETID1-NETID2

To find your partner, take the NETID that isn't yours, add @uw.edu, and e-mail them!

If that still doesn't work, e-mail the course staff and we'll send an introductory e-mail to the two of you.

Storing Items in an Array

Key	3	4	7	9	10	12	15
Value	"dog"	"cat"	"bird"	"horse"	"oxen"	"ferret"	"moose"

get(key): O(n)

put(key, value): O(n)

remove(): O(n)

Storing Sorted Items in an Array

Key	~~7~~ 1	~~4~~ 2	~~3~~ 3	~~12~~ 4	~~10~~ ~~5~~	~~9~~ 6	~~15~~ 7
Value	"bird"	"cat"	"dog"	"ferret"	~~"oxen"~~	"horse"	"moose"

get(key): O(log n)

put(key, value): O(n)

remove(): O(n)

Storing Sorted Items in an Array

get() – O(logn)

put() – O(n)

remove() – O(n)

Can we do better with insertions and removals?

Trees!

A **tree** is a collection of nodes

- Each node has at most 1 parent and 0 or more children

Root node: the single node with no parent, "top" of the tree

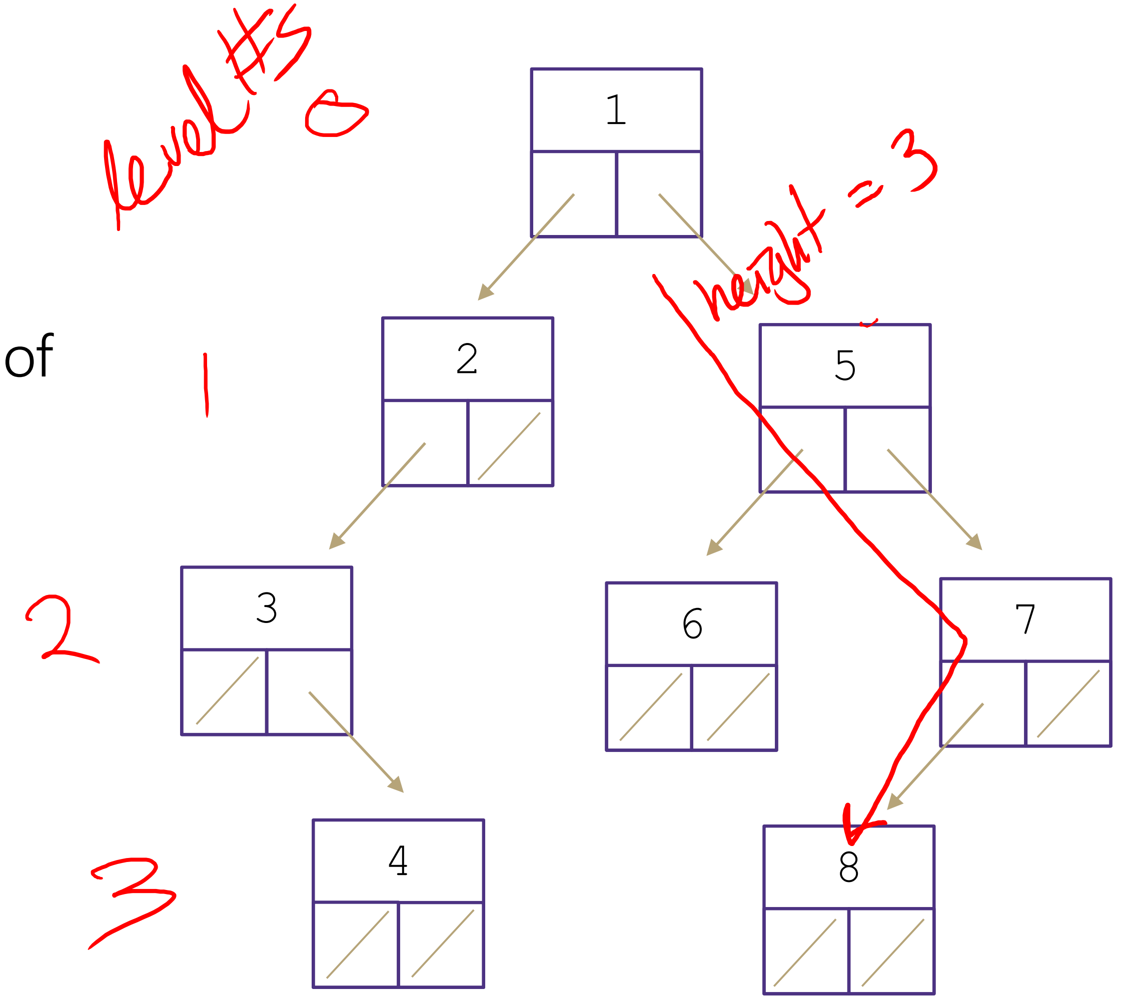
Branch node: a node with one or more children

Leaf node: a node with no children

Edge: a pointer from one node to another

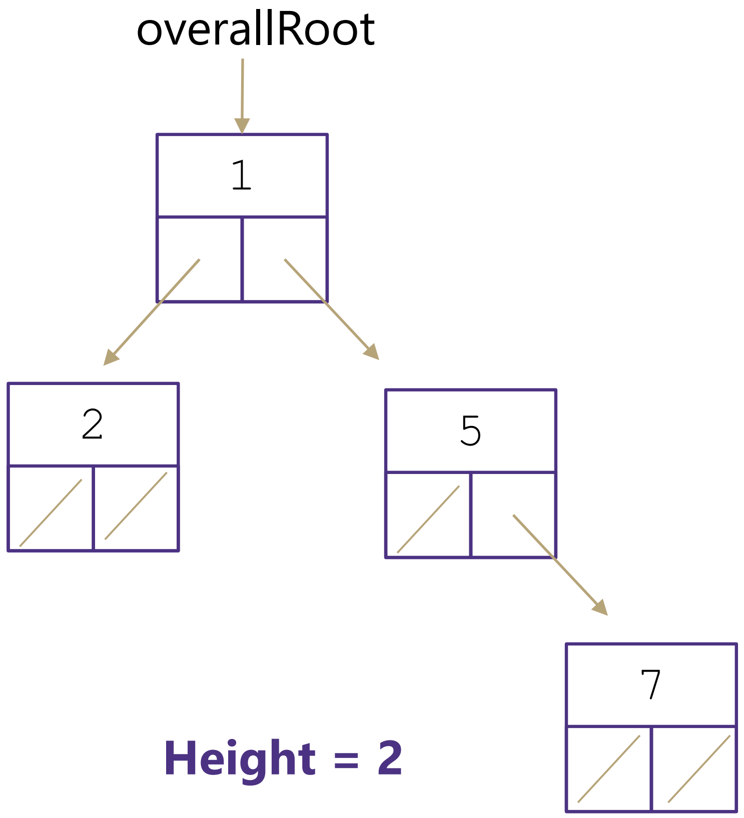
Subtree: a node and all it descendants

Height: the number of edges contained in the longest path from root node to some leaf node

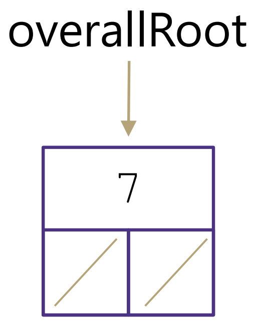


Tree Height

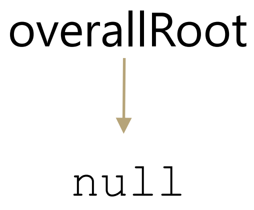
What is the height of the following trees?



Height = 2



Height = 0



Height = -1 or NA

Traversals

traversal: An examination of the elements of a tree.

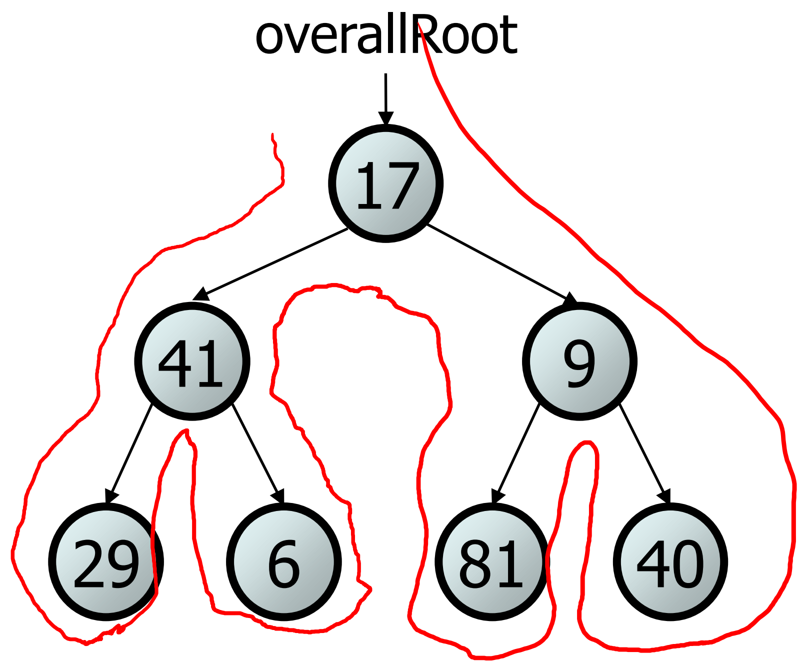
- A pattern used in many tree algorithms and methods

Common orderings for traversals:

- **pre-order**: process root node, then its left/right subtrees
  - `17 41 29 6 9 81 40`
- **in-order**: process left subtree, then root node, then right
  - `29 41 6 17 81 9 40`
- **post-order**: process left/right subtrees, then root node
  - `29 6 41 81 40 9 17`

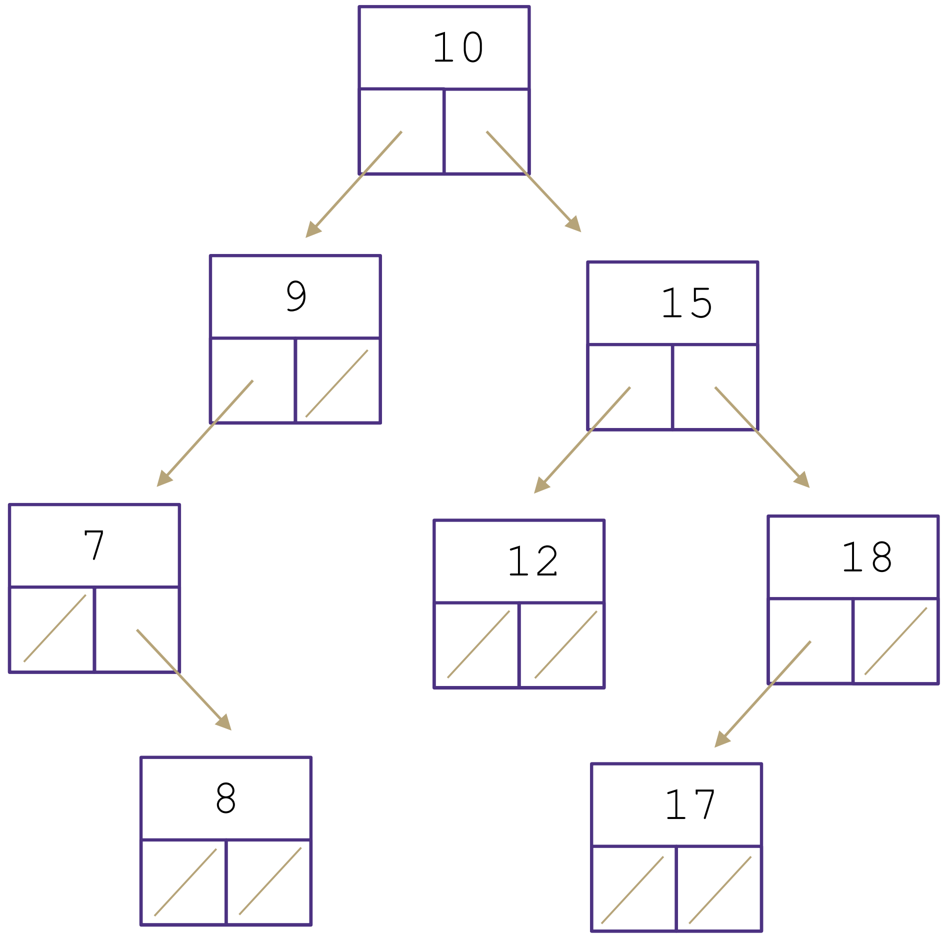
Traversal Trick: Sailboat method

- Trace a path around the tree.
- As you pass a node on the proper side, process it.
  - pre-order: left side
  - in-order: bottom
  - post-order: right side



Binary Search Trees

A binary search tree is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.



Implement Dictionary

Binary Search Trees allow us to:

- quickly find what we're looking for
- add and remove values easily

Dictionary Operations:

Runtime in terms of height, "h"

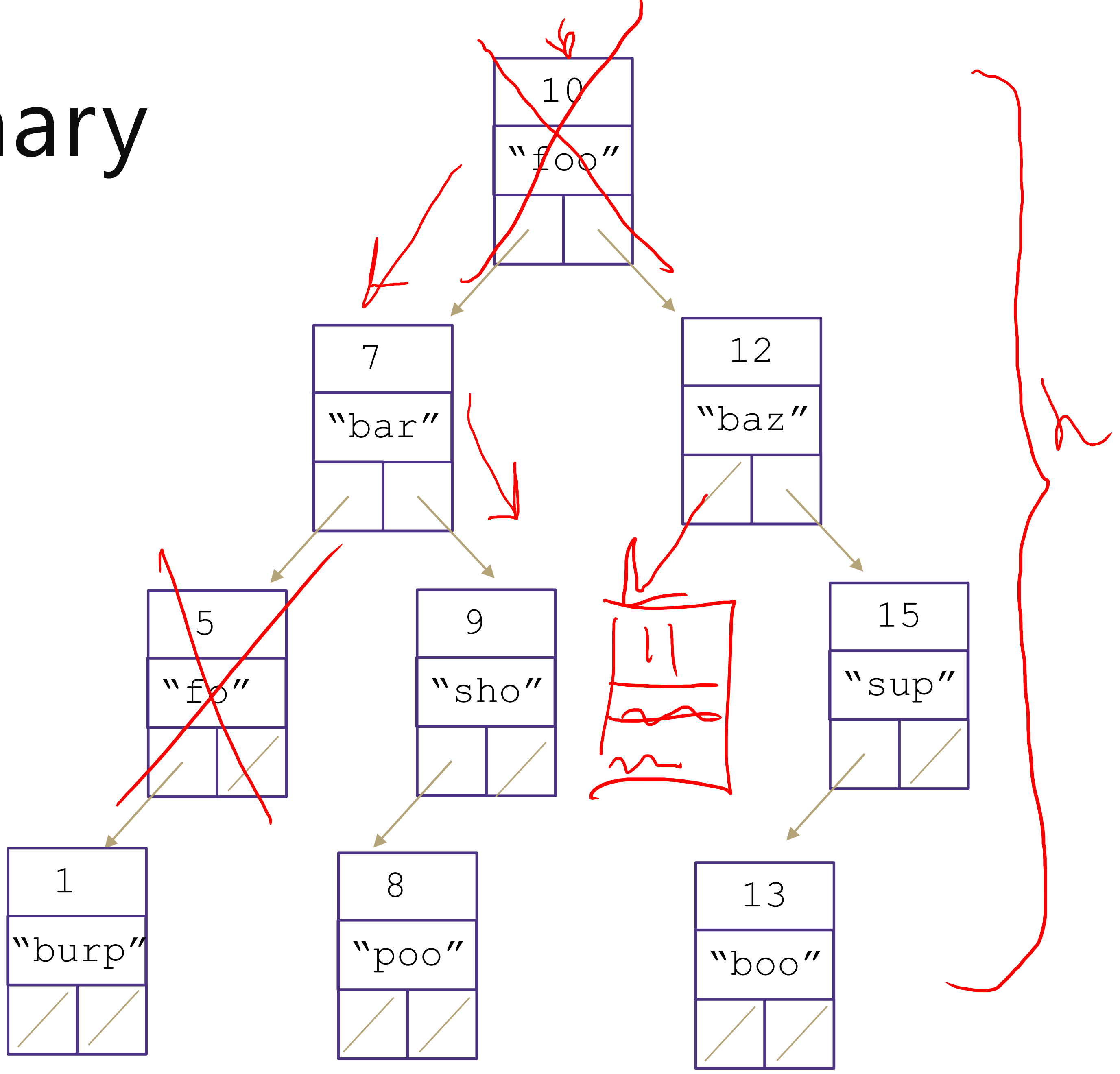
get() – O(h)

put() – O(h)

remove() – O(h)

What do you replace the node with?

Largest in left sub tree or smallest in right sub tree

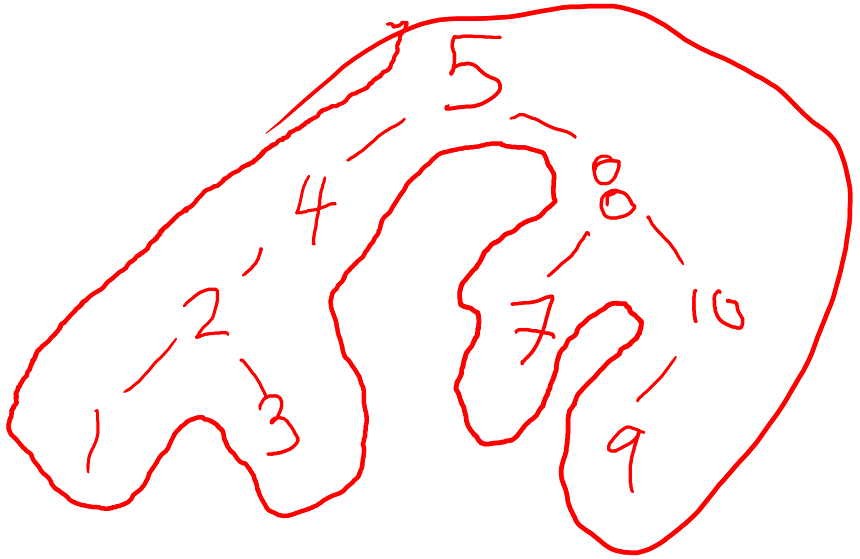


Practice

What will the binary search tree look like if you insert nodes in the following order:

5, 8, 7, 10, 9, 4, 2, 3, 1

What is the pre-order traversal order for the resulting tree?



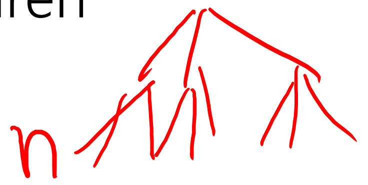
Height in terms of Nodes

For "balanced" trees $h \approx \log_c(n)$ where c is the maximum number of children

Balanced binary trees $h \approx \log_2(n)$

Balanced trinary tree $h \approx \log_3(n)$

Thus for balanced trees operations take $\Theta(\log_c(n))$



Unbalanced Trees

Is this a valid Binary Search Tree?

Yes, but…

We call this a degenerate tree

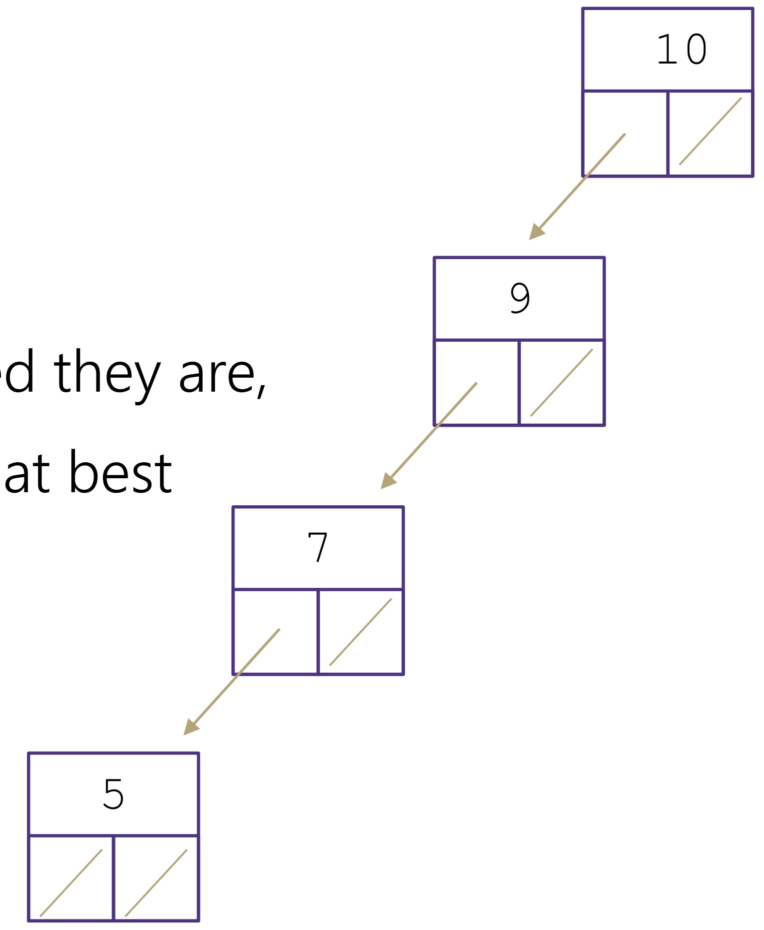
For trees, depending on how balanced they are,

Operations at worst can be O(n) and at best

can be O(logn)

How are degenerate trees formed?

- insert(10)
- insert(9)
- insert(7)
- insert(5)

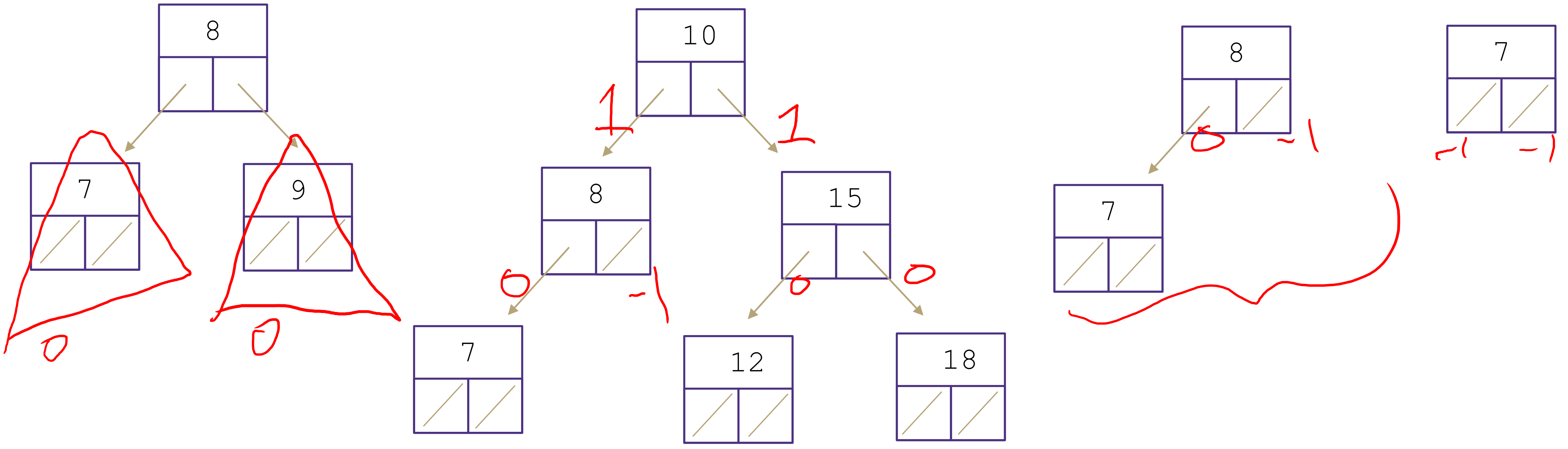


Measuring Balance

Measuring balance:

For each node, compare the heights of its two sub trees

Balanced when the difference in height between sub trees is no greater than 1



Meet AVL Trees

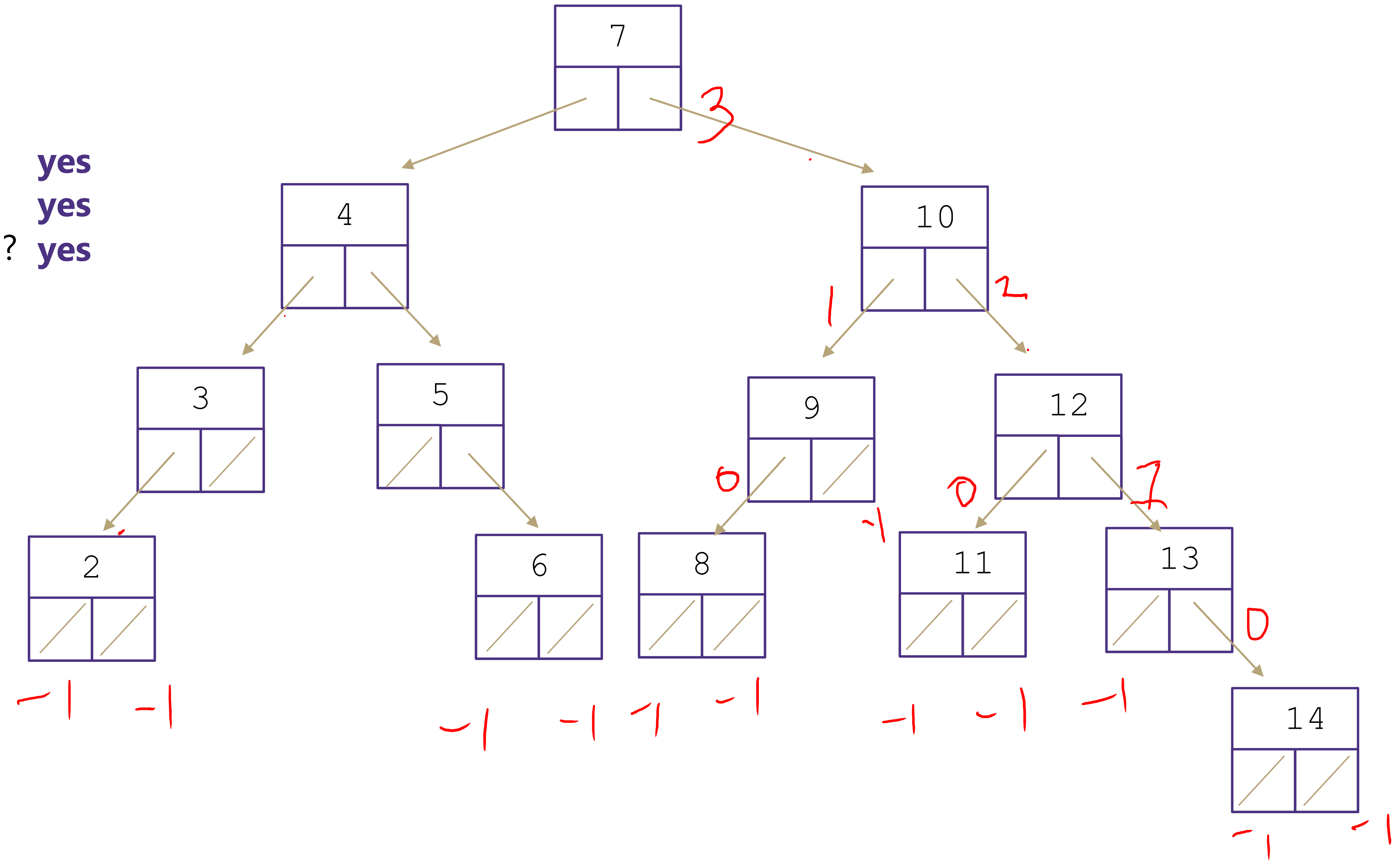
AVL Trees must satisfy the following properties:

- binary trees: all nodes must have between 0 and 2 children
- binary search tree: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- balanced: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

Is this a valid AVL tree?

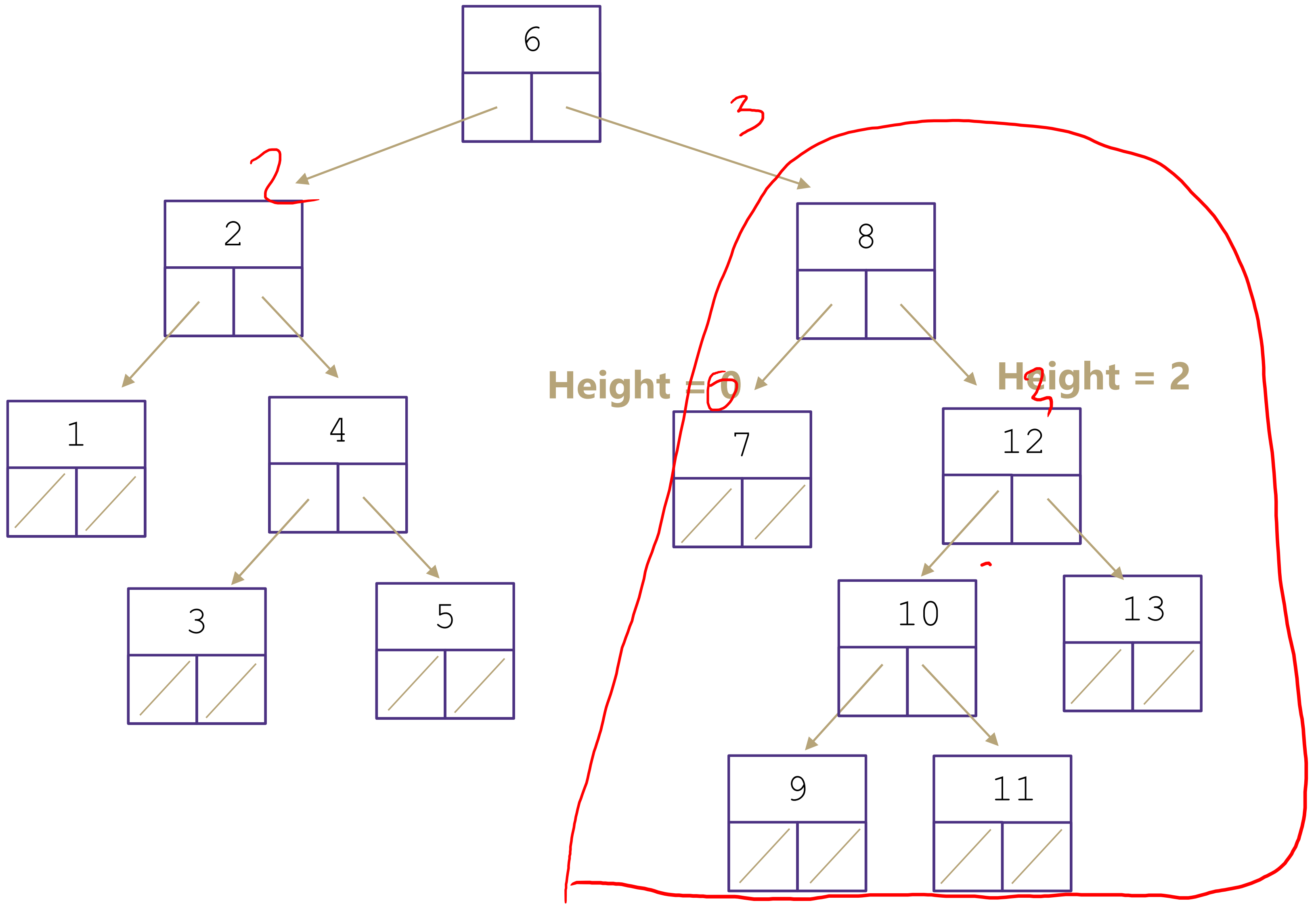
Is it…
- Binary **yes**
- BST **yes**
- Balanced? **yes**



Is this a valid AVL tree?

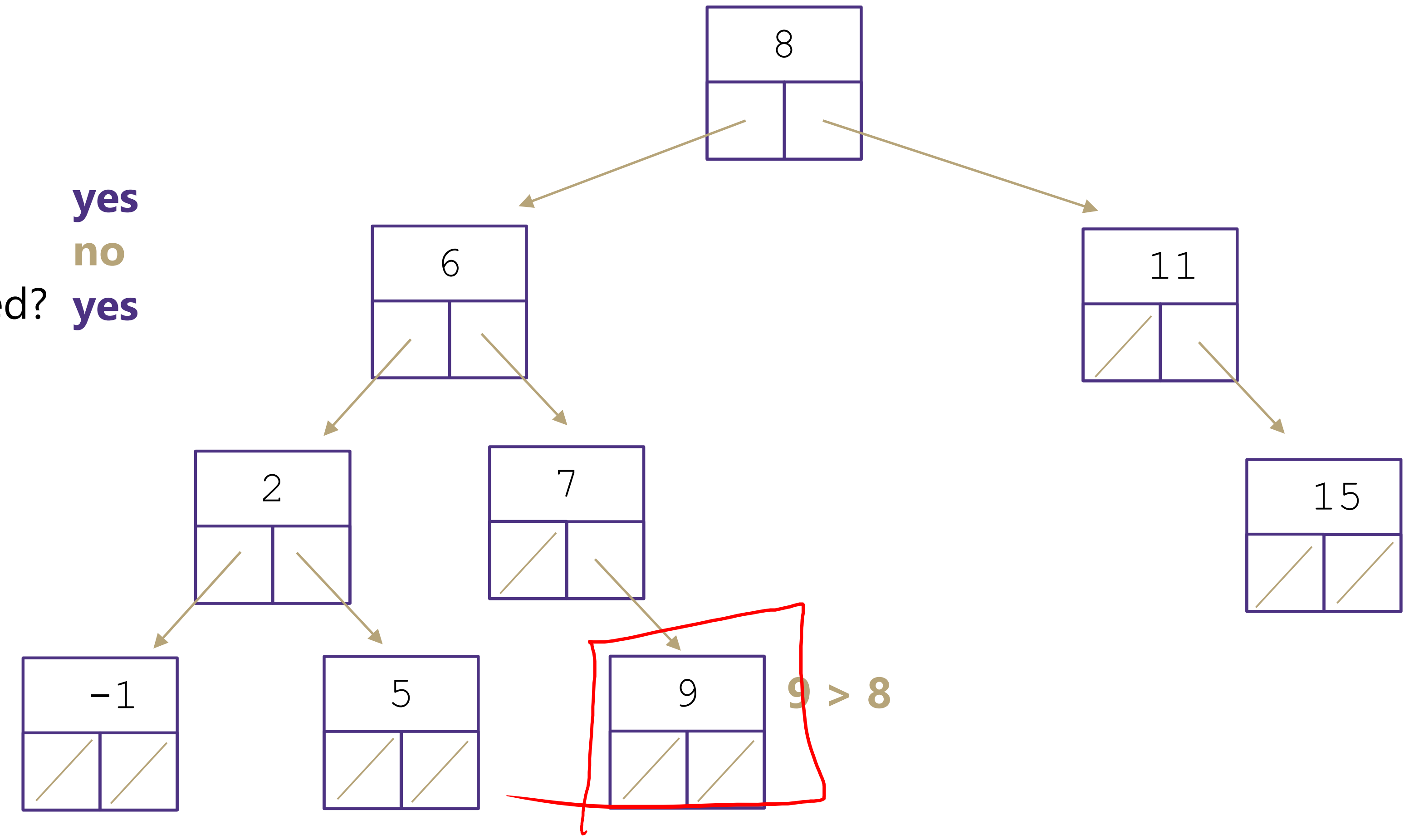
Is it…

- Binary **yes**
- BST **yes**
- Balanced? **no**



Is this a valid AVL tree?

Is it…
- Binary **yes**
- BST **no**
- Balanced? **yes**



Implementing an AVL tree dictionary

Dictionary Operations:

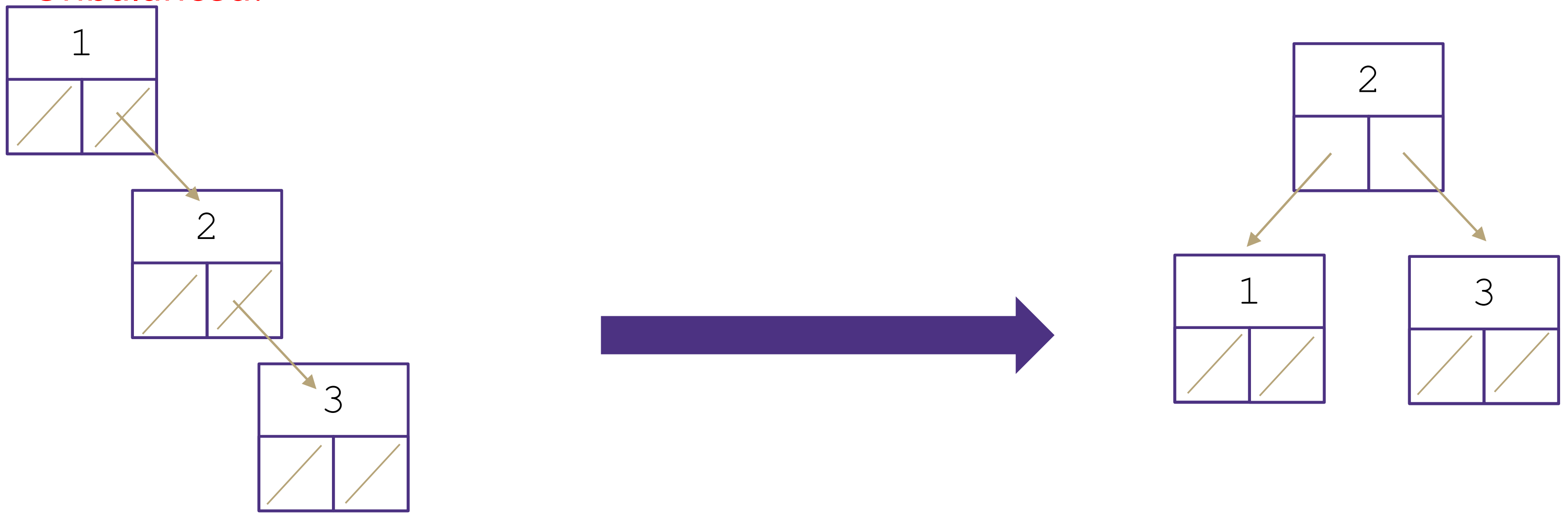
get() – same as BST

containsKey() – same as BST

put() - Add the node to keep BST, fix AVL property if necessary

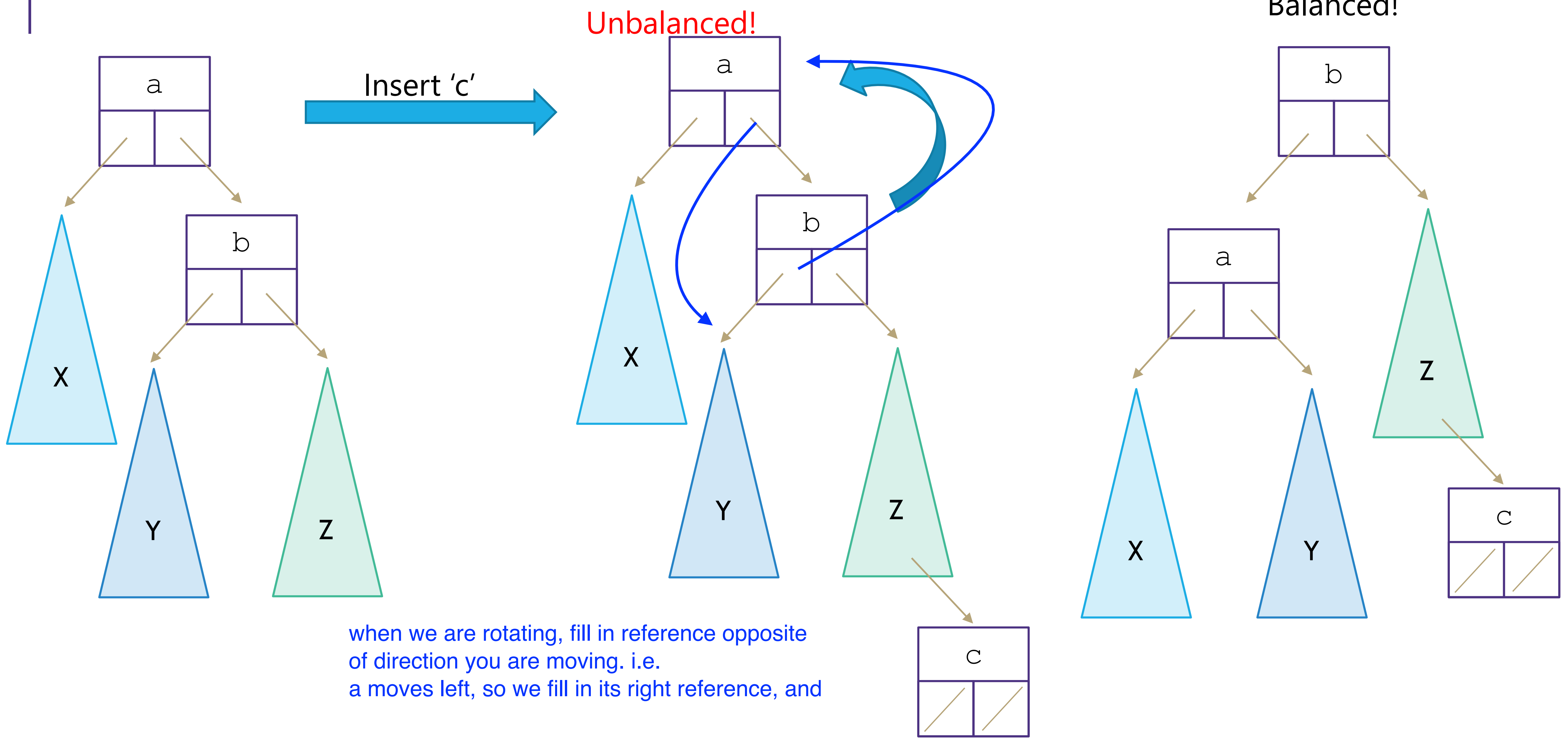
remove() - Replace the node to keep BST, fix AVL property if necessary

Unbalanced!



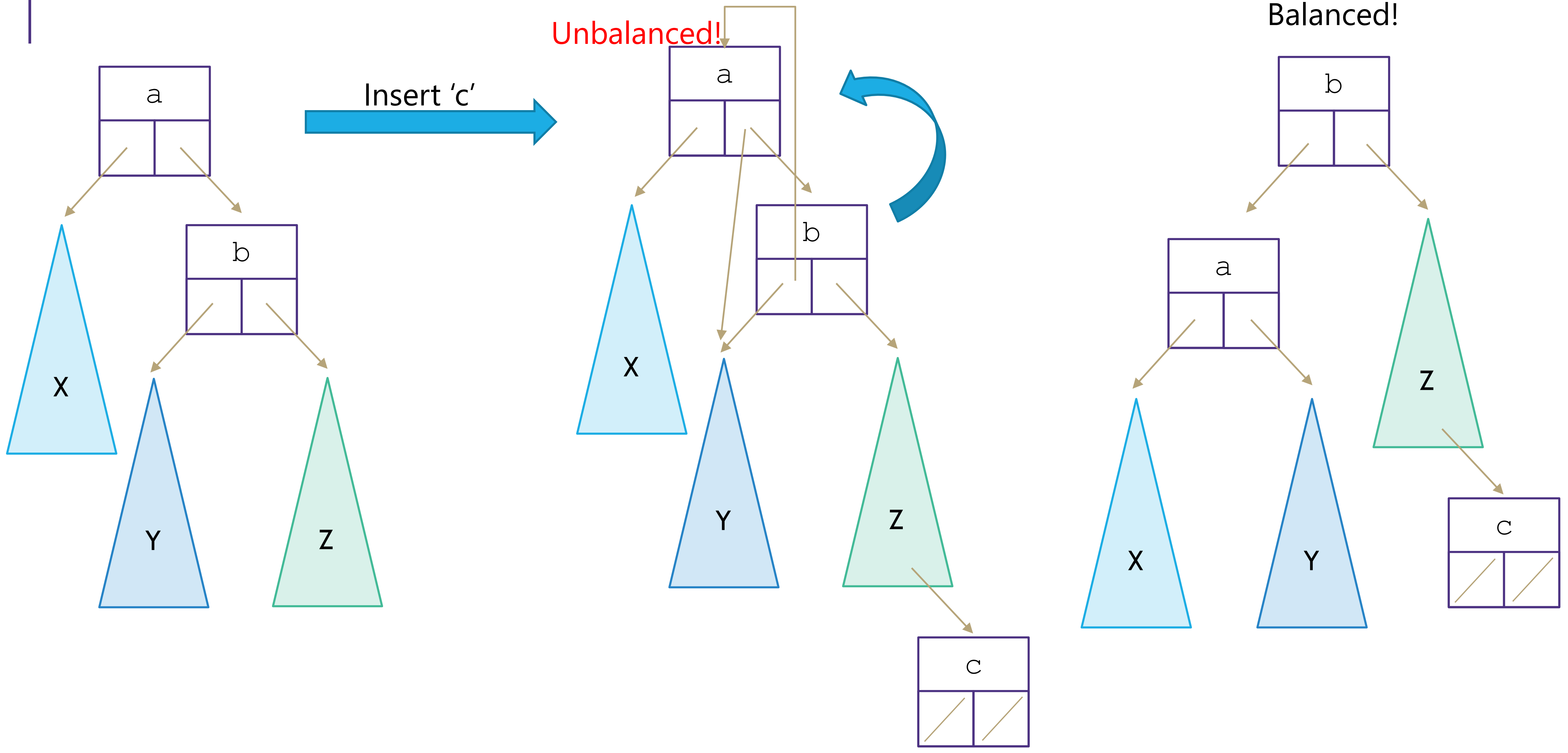
Rotations!

what we know at this point as that anything in a right subtree is greater than the root, and anything to the left is lesser.
We can move the Y triangle to a's right reference BECAUSE its in the right subtree, and therefore necessarily larger than a. Furthermore, we can move a to b's left reference because b is to the right of a and necessarily LARGER



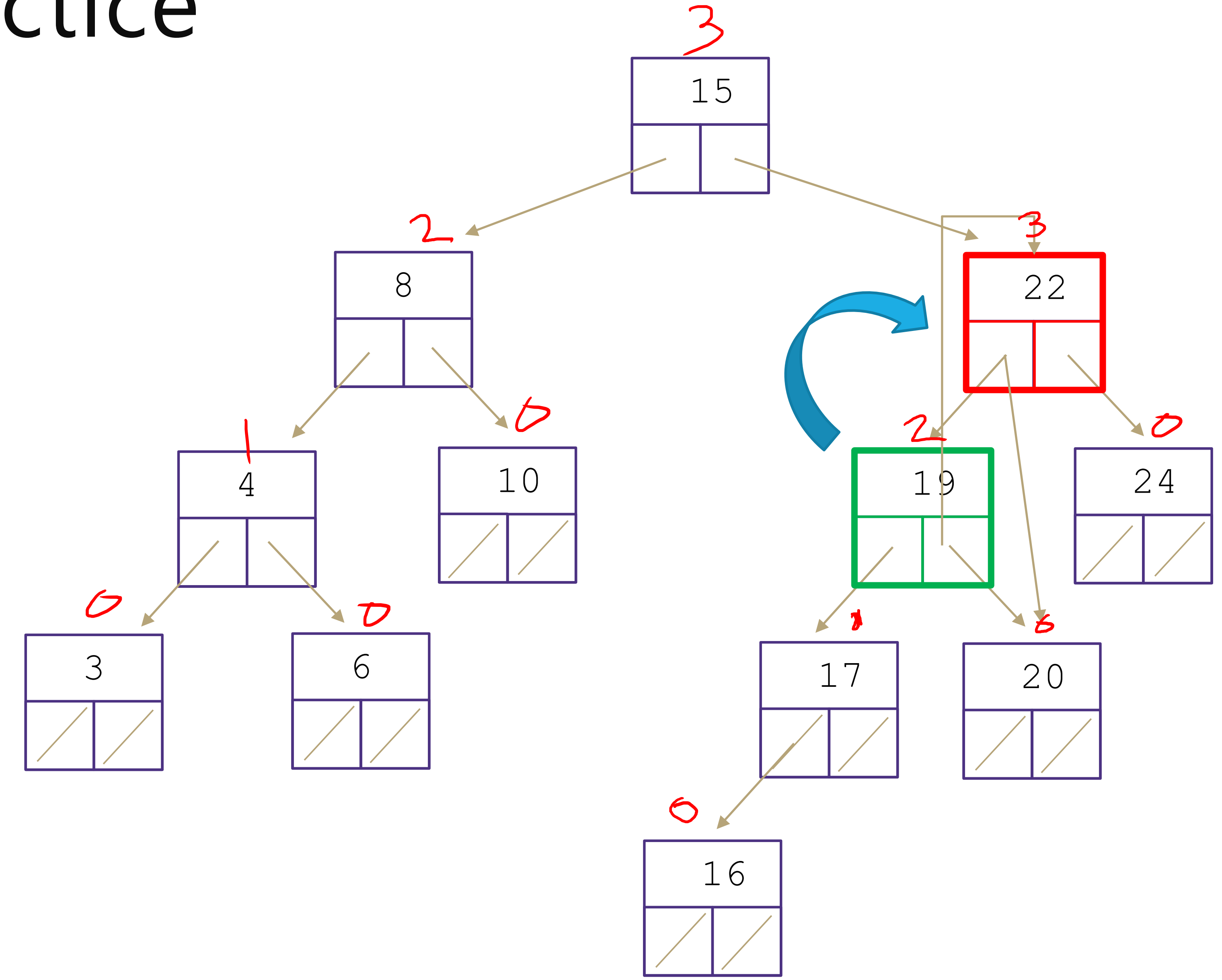
when we are rotating, fill in reference opposite of direction you are moving. i.e.
a moves left, so we fill in its right reference, and

Rotations!



Practice

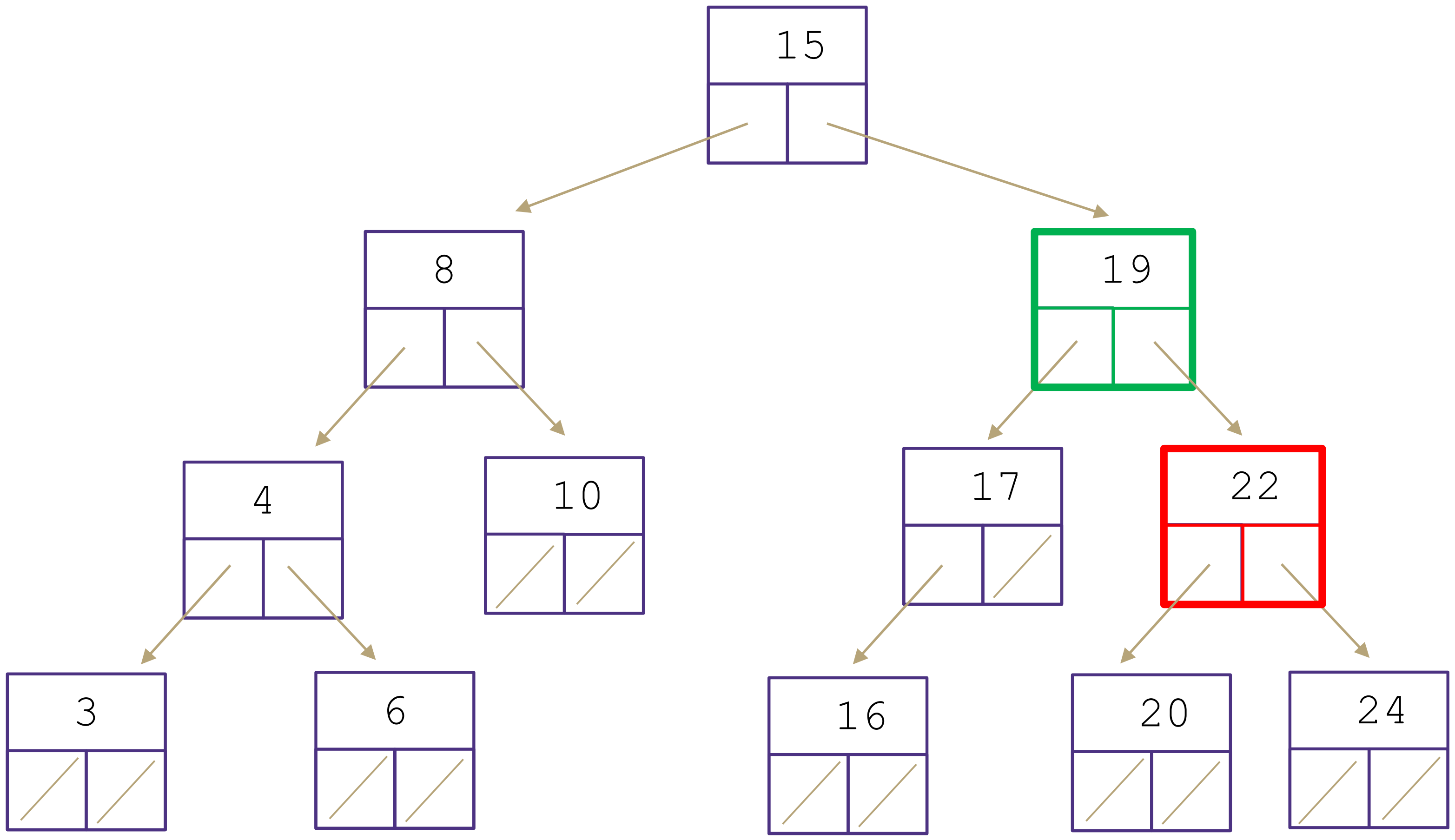
put(16);



find subtree that is too high and move it up; in this case, 22's left subtree is too large. As it is the left subtree, we change the RIGHT reference of 19, the root of the subtree, to be its parent, and move 22's LEFT referenc to 19's old right reference. Also make sure 15's right reference connects to 19

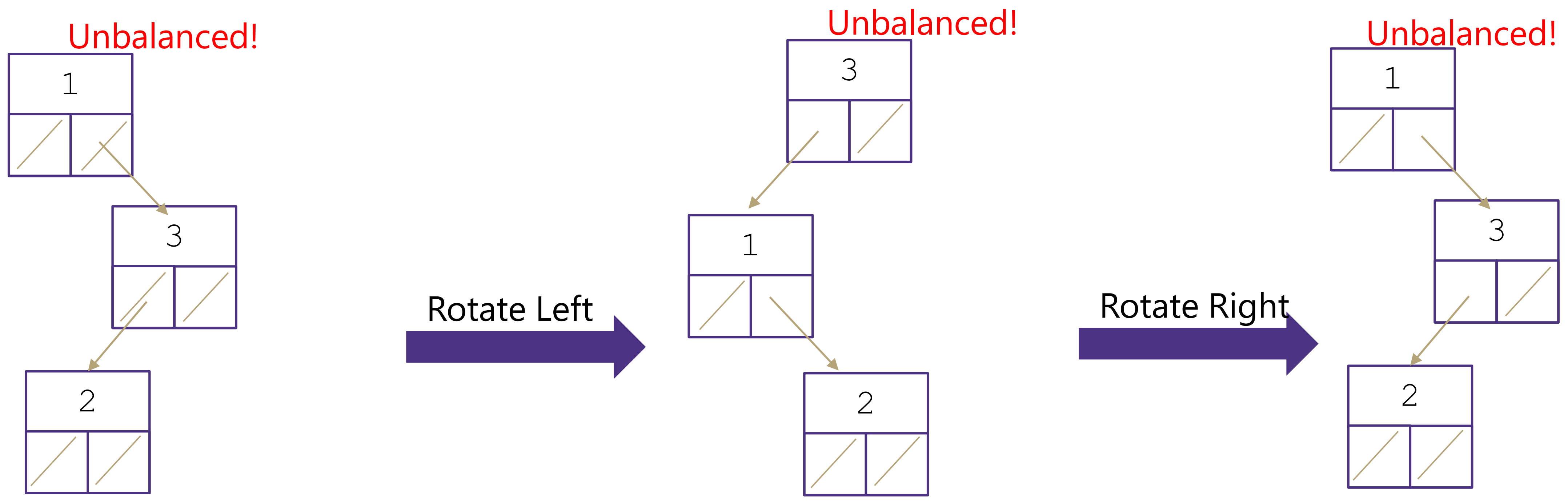
Practice

put(16);



START WITH LOWEST NODE THAT IS AN ISSUE

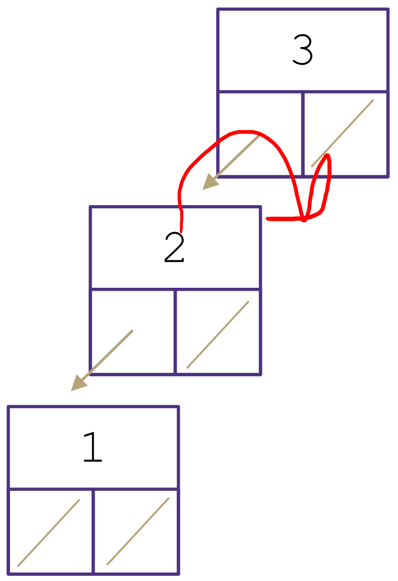
So much can go wrong

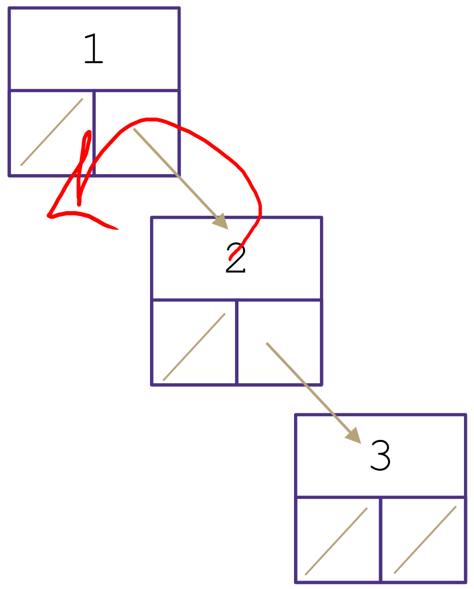


Two AVL Cases

start with lowest node that is an issue

Line Case
Solve with 1 rotation



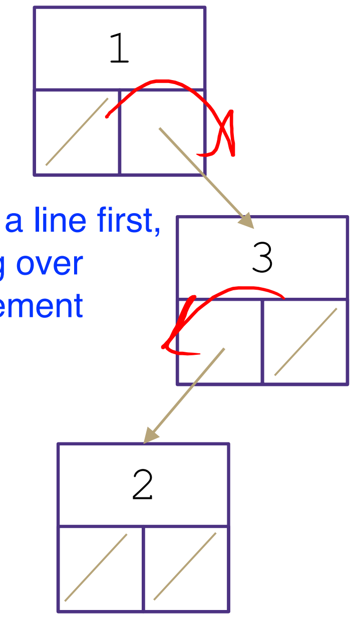


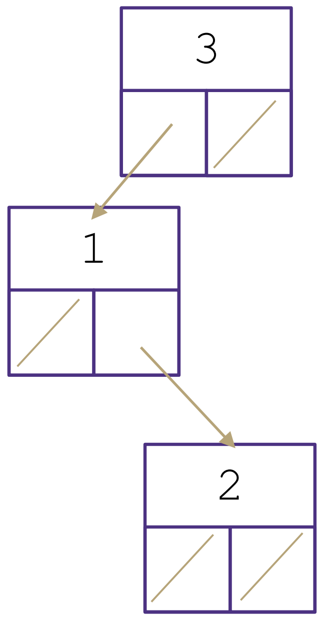
Rotate Right
Parent's left becomes child's right
Child's right becomes its parent

Rotate Left
Parent's right becomes child's left
Child's left becomes its parent

Kink Case
Solve with 2 rotations

make into a line first,
by rotating over
bottom element

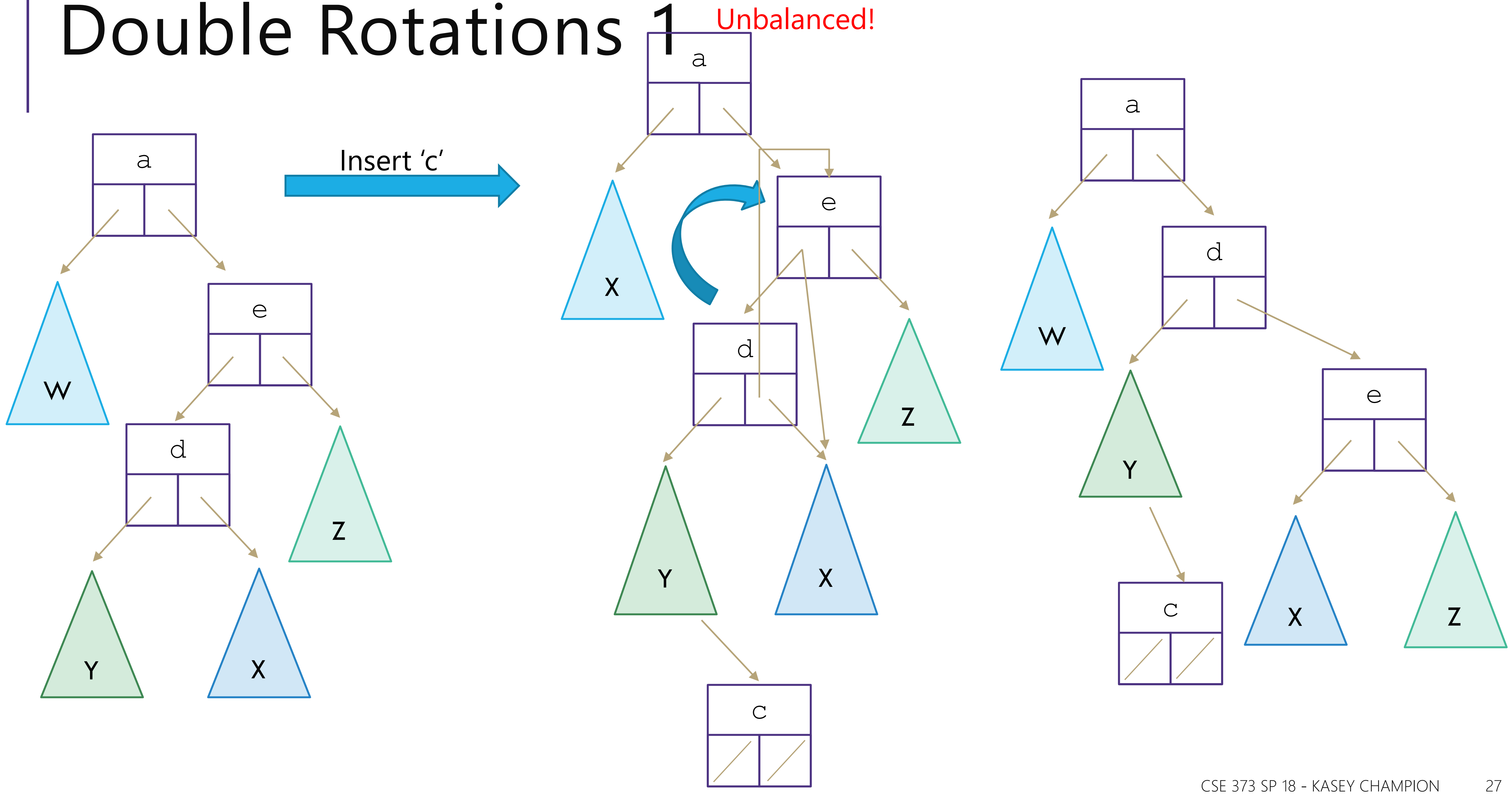




Rotate subtree left
Rotate root tree right

Rotate subtree right
Rotate root tree left

Double Rotations 1


a
W
e
d
Y
X
Z
Insert 'c'
Unbalanced!
a
X
e
d
Y
X
c
Z
a
W
d
Y
c
e
X
Z