AVL Trees

Data Structures and Parallelism

Announcements

Exercise 1 grades on gradescope.

Regrade policy

Exercises: Submit a regrade request via gradescope

If the initial regrade isn't sufficient, send an email to Rob

- Include a written explanation of why you think the grading isn't correct.
- (especially on autograded section of projects) if you can explain a mistake you made and why it's not worth as many points as it cost you, we'll consider adjusting the rubric.
- tl;dr – we're humans, talk to us.

Project 1 – Will treat late pushes as a late submission, but also try to let us know when you plan to turn in late

More Announcements

Exercises 2,3 due today.

Exercise 4 is out tonight

- Due Wednesday (so you have time to prepare for the midterm after).

Project 2 out tonight.

If you haven't told us you plan to change partners, tell us ASAP.

Old midterms to study from will go up over the weekend.

- In the meantime, there are some old midterms on 19wi's site.

Outline

Last time:

Dictionaries are **extremely** common data structures

Main operations:

- Find
- Insert
- Delete

BSTs are good in the average case, but bad in the worst case.

Today:

How to perform actions on AVL trees.

Avoiding the Worst Case

An AVL tree is a binary search tree that also meets the following rule

AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

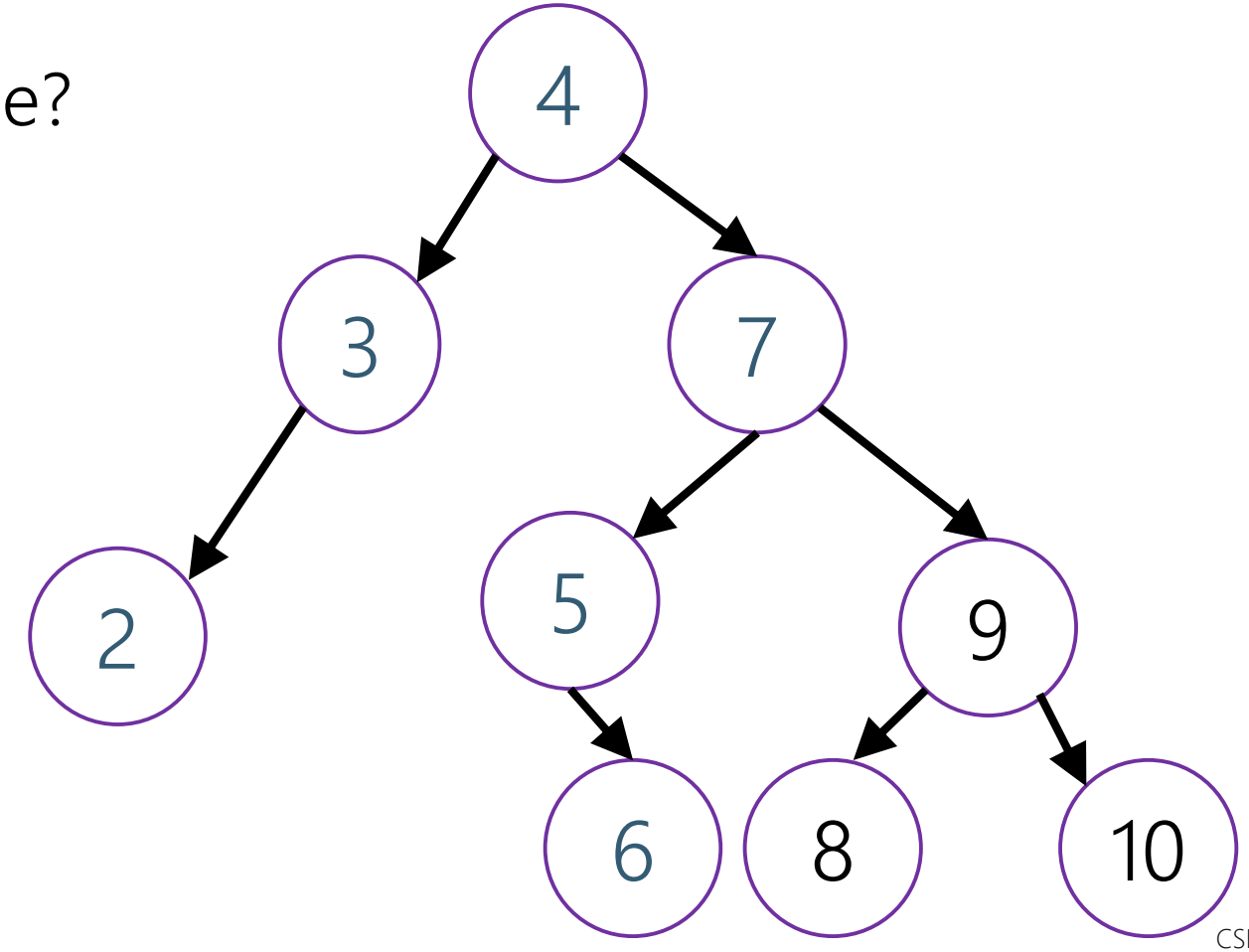
This will avoid the worst case! We have to check:

1. ~~Such a tree must have height $O(\log n)$.~~ Done!
2. We must be able to maintain this property when inserting/deleting

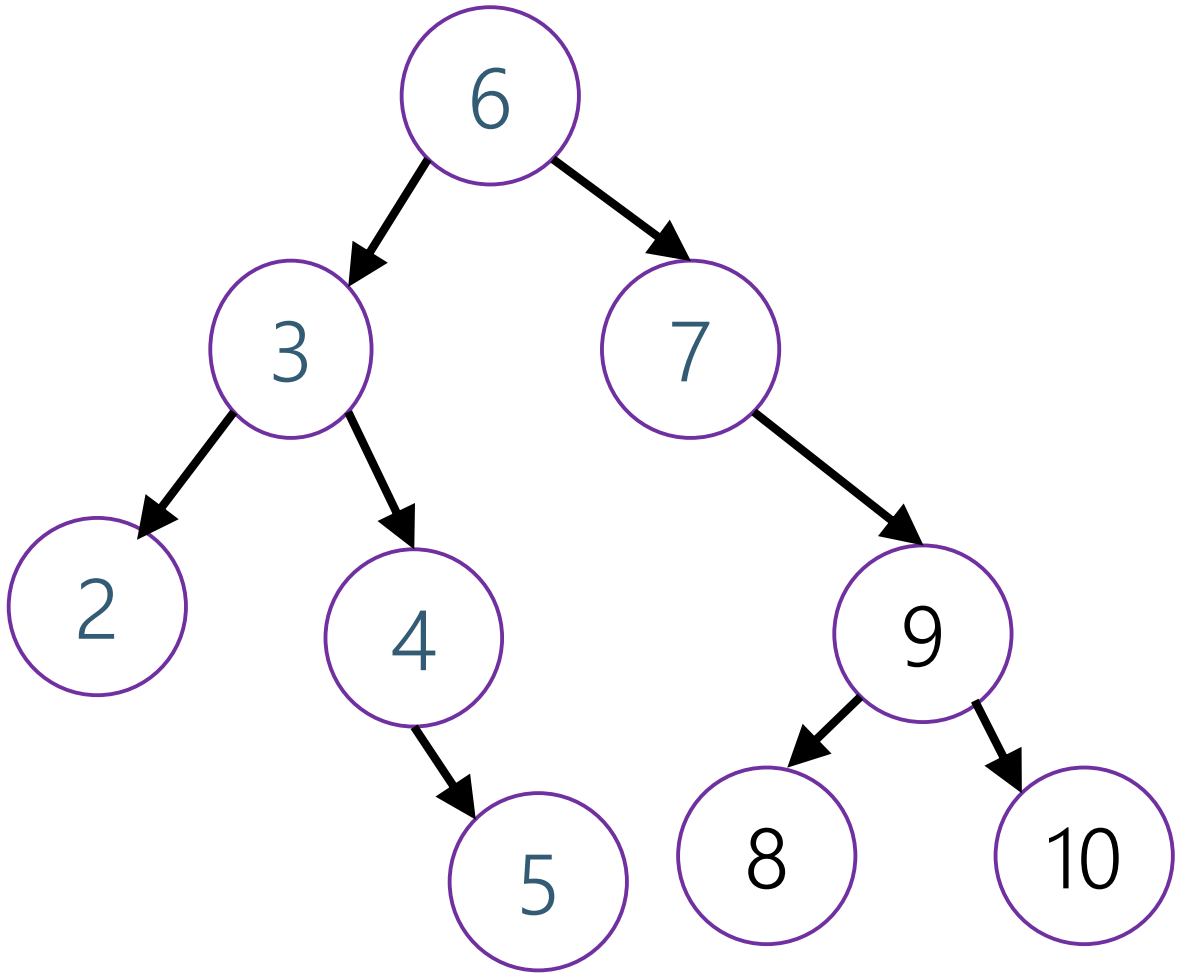
Warm-Up

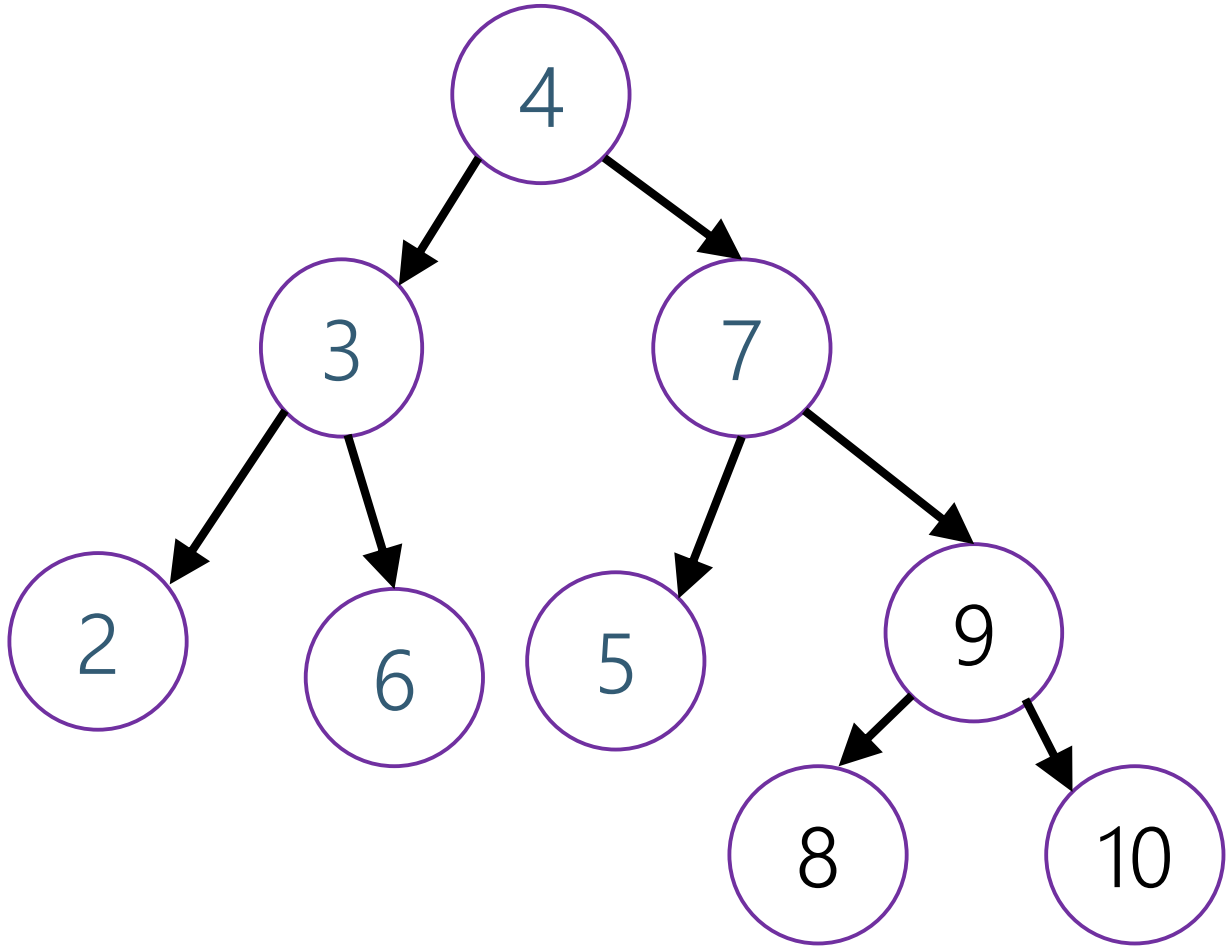
AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



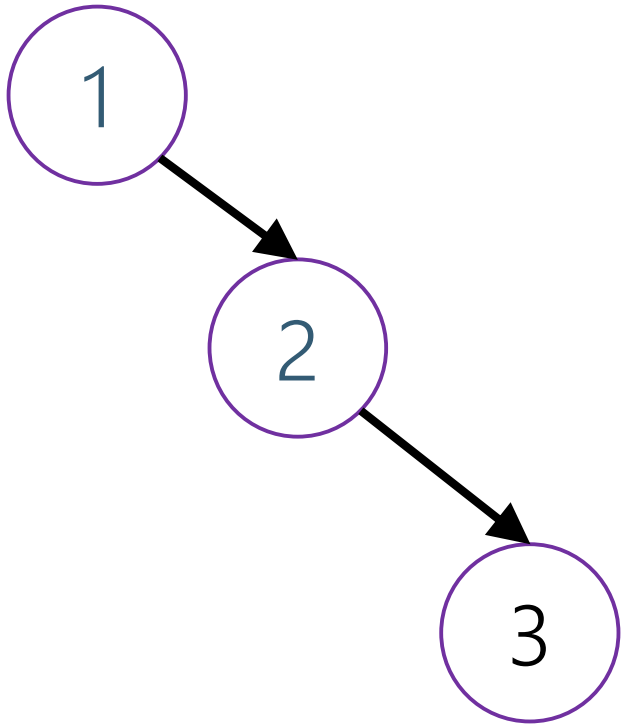
Are These AVL Trees?

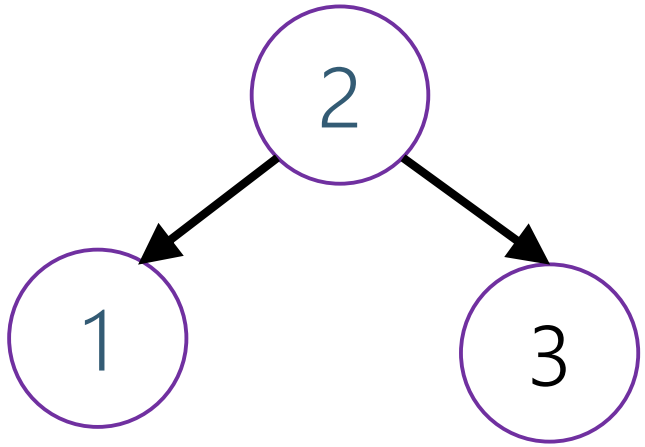




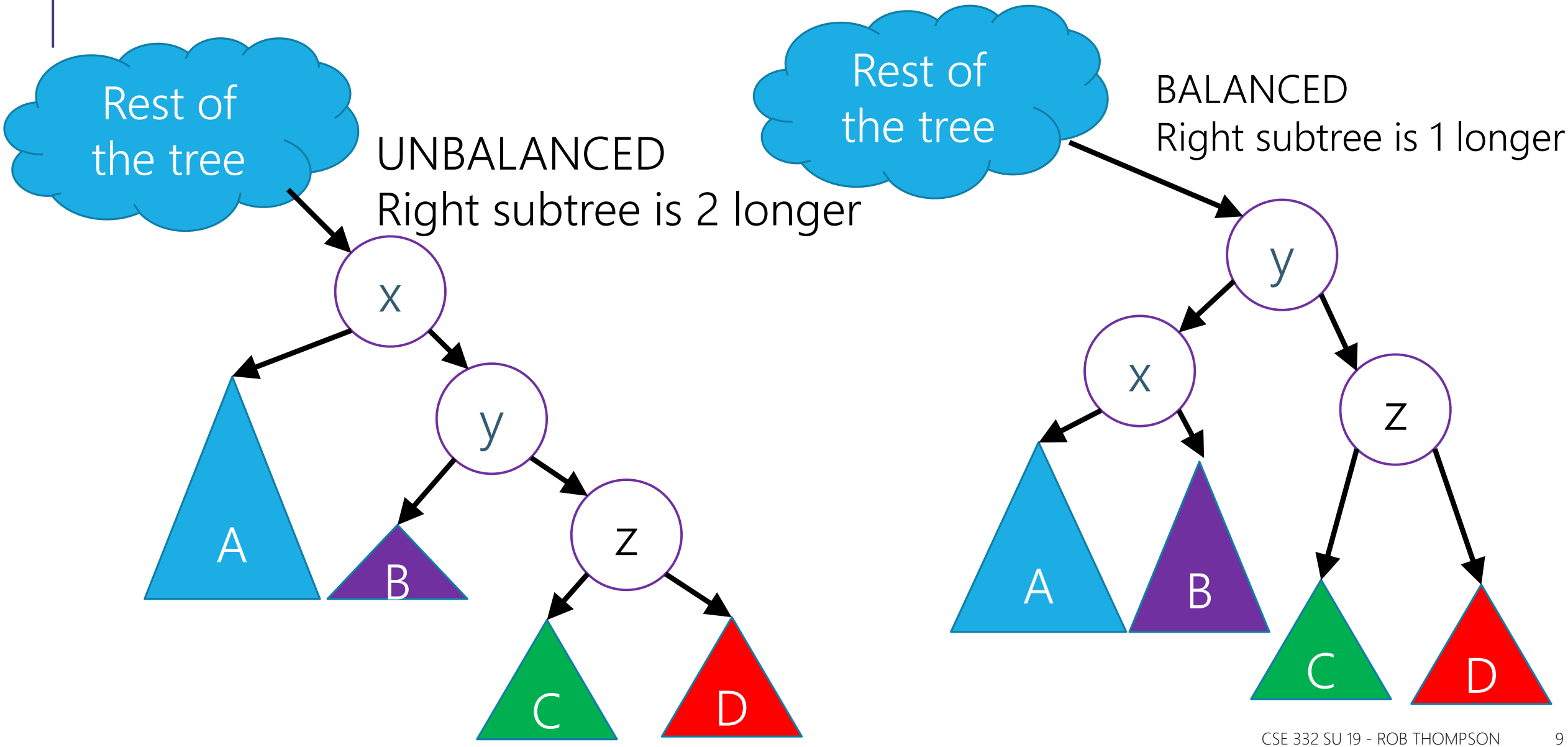
Insertion

What happens if when we do an insertion, we break the AVL condition?

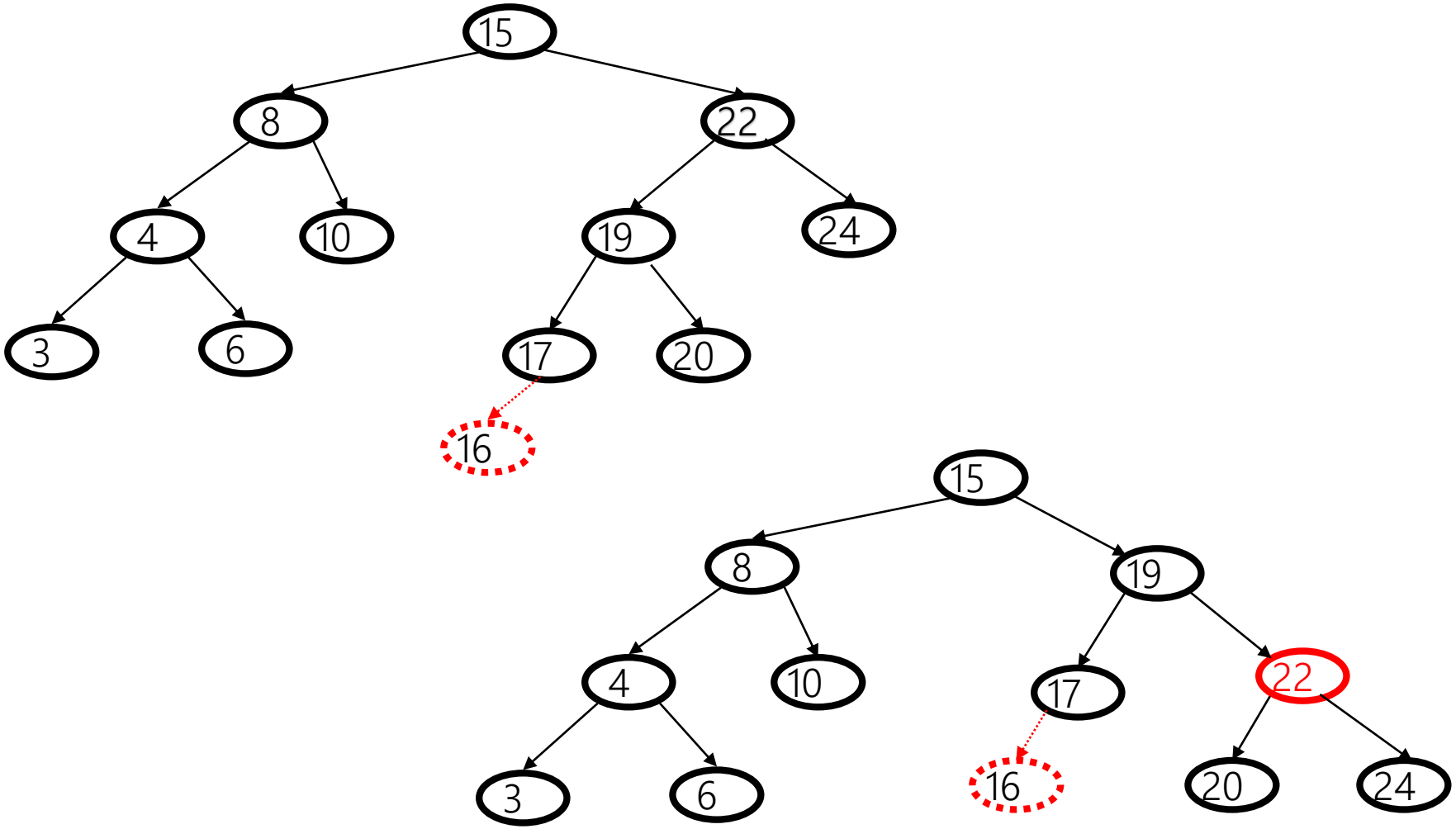




Left Rotation

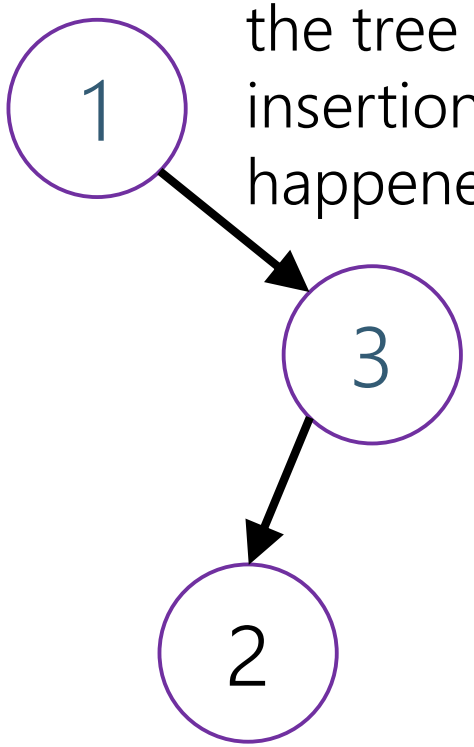


Another example: insert(16)



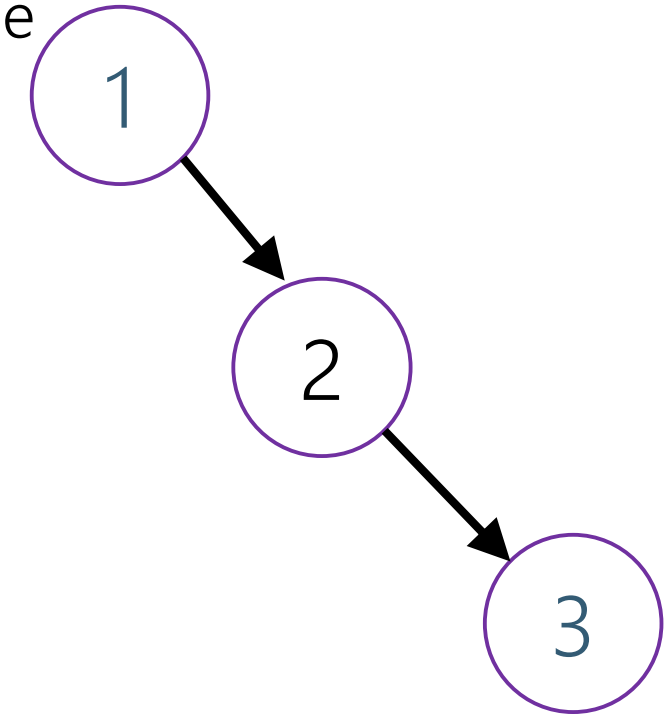
It Gets More Complicated

There's a "kink" in the tree where the insertion happened.

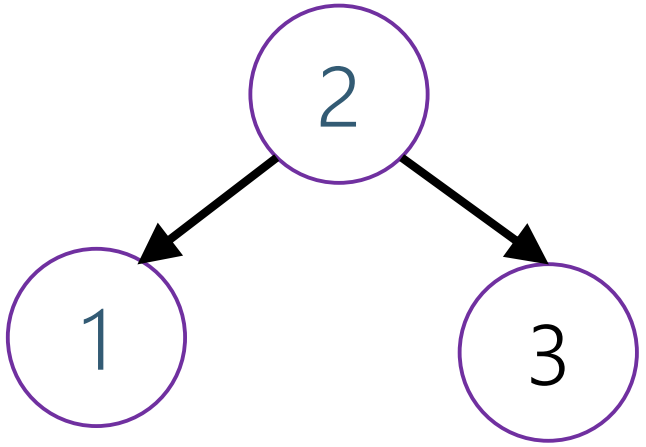


Can't do a left rotation

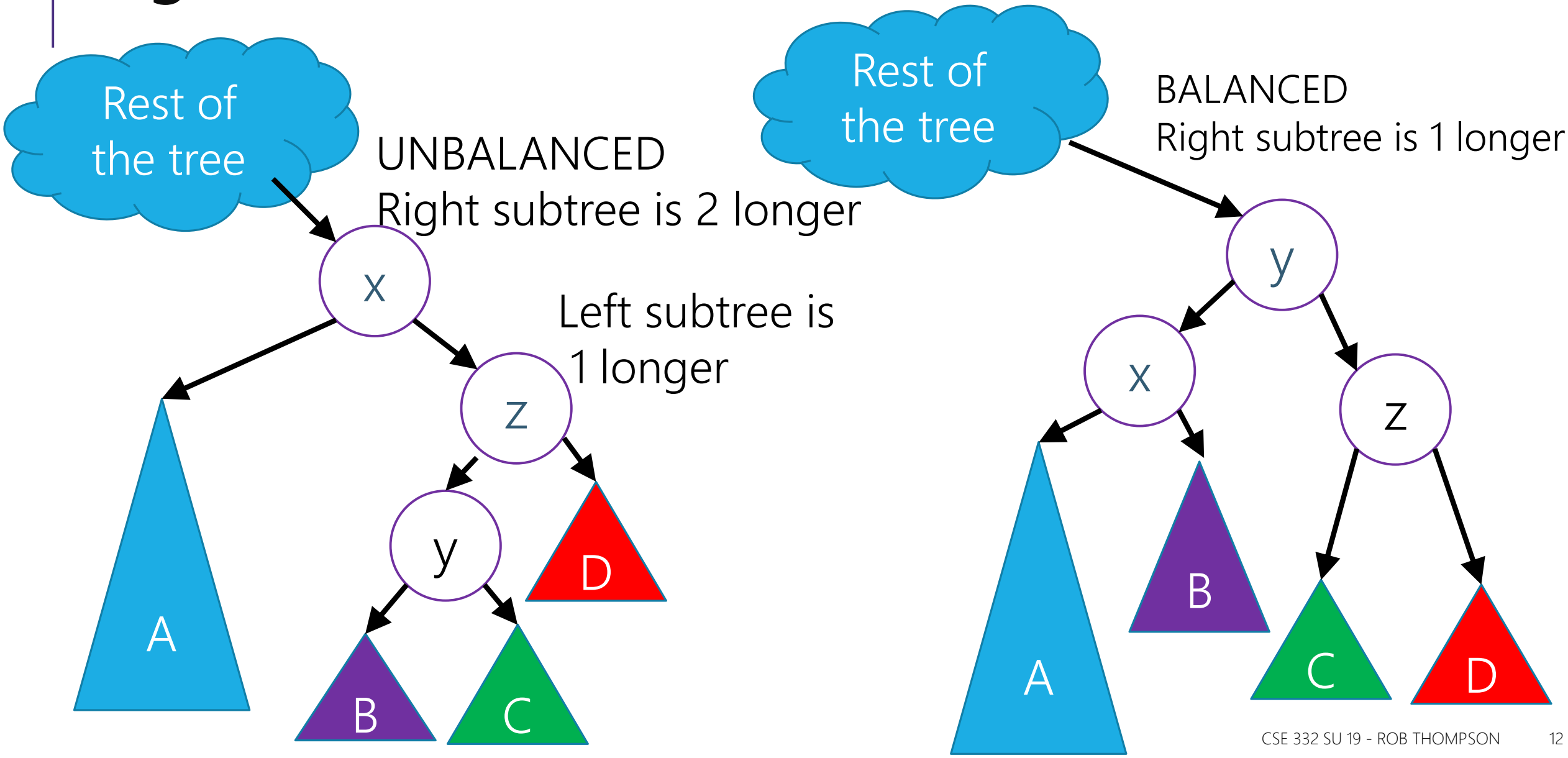
Do a "right" rotation around 3 first.



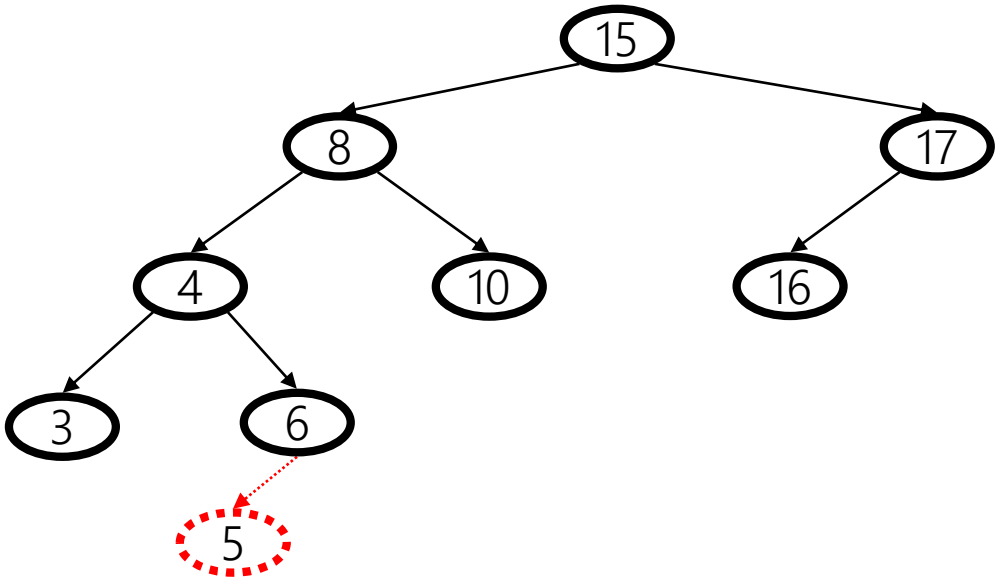
Now do a left rotation.



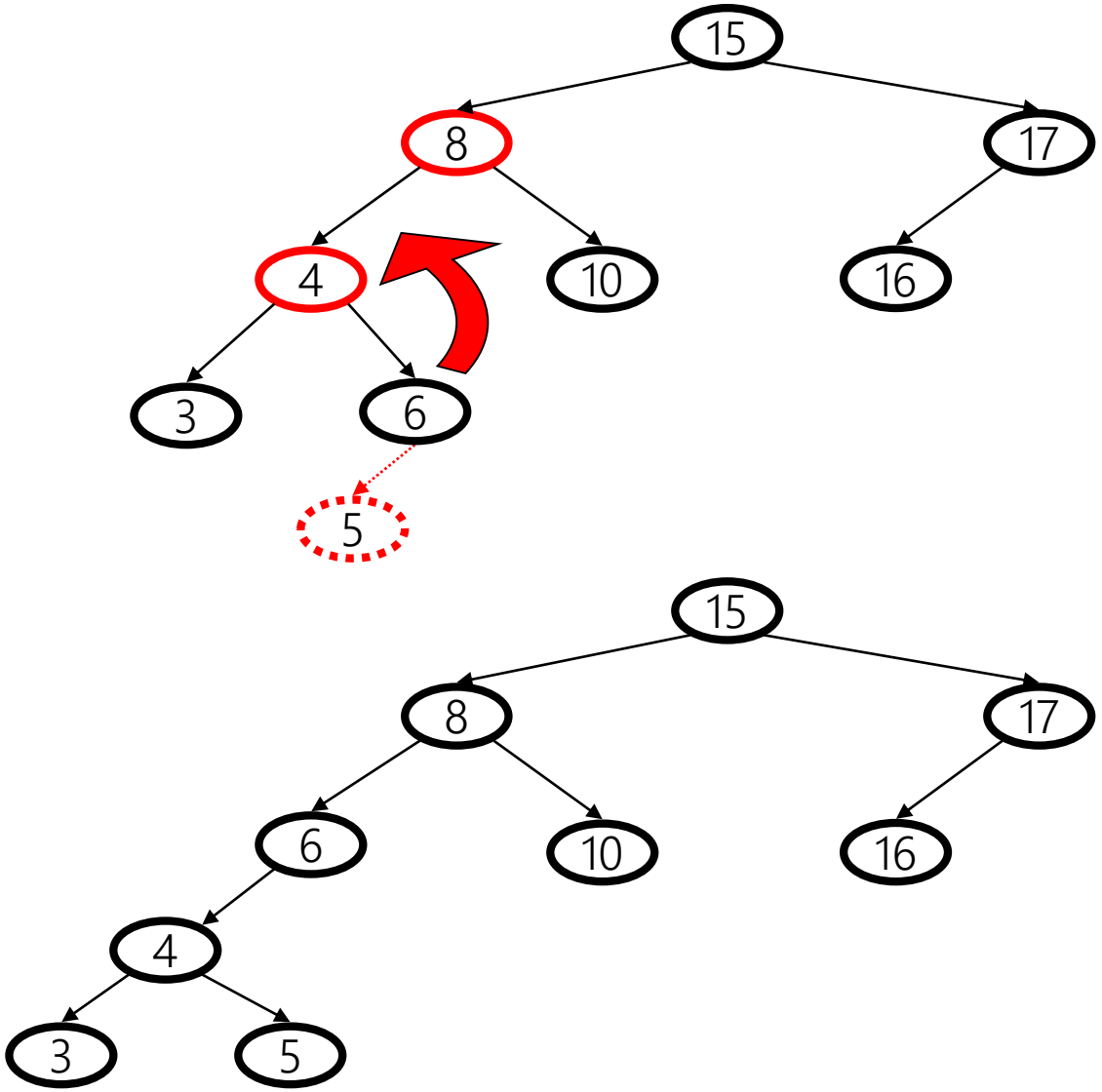
Right Left Rotation



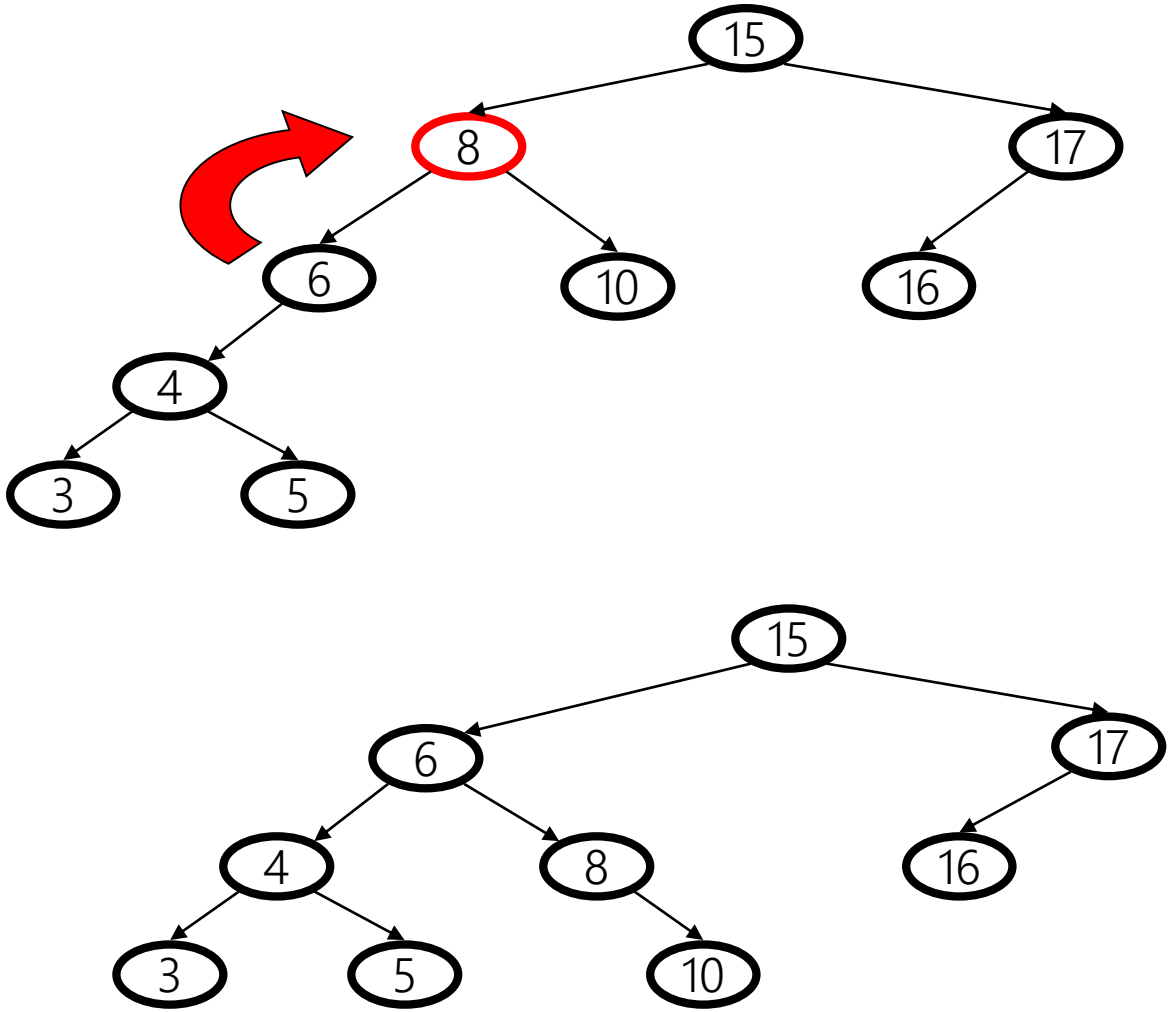
Insert 5



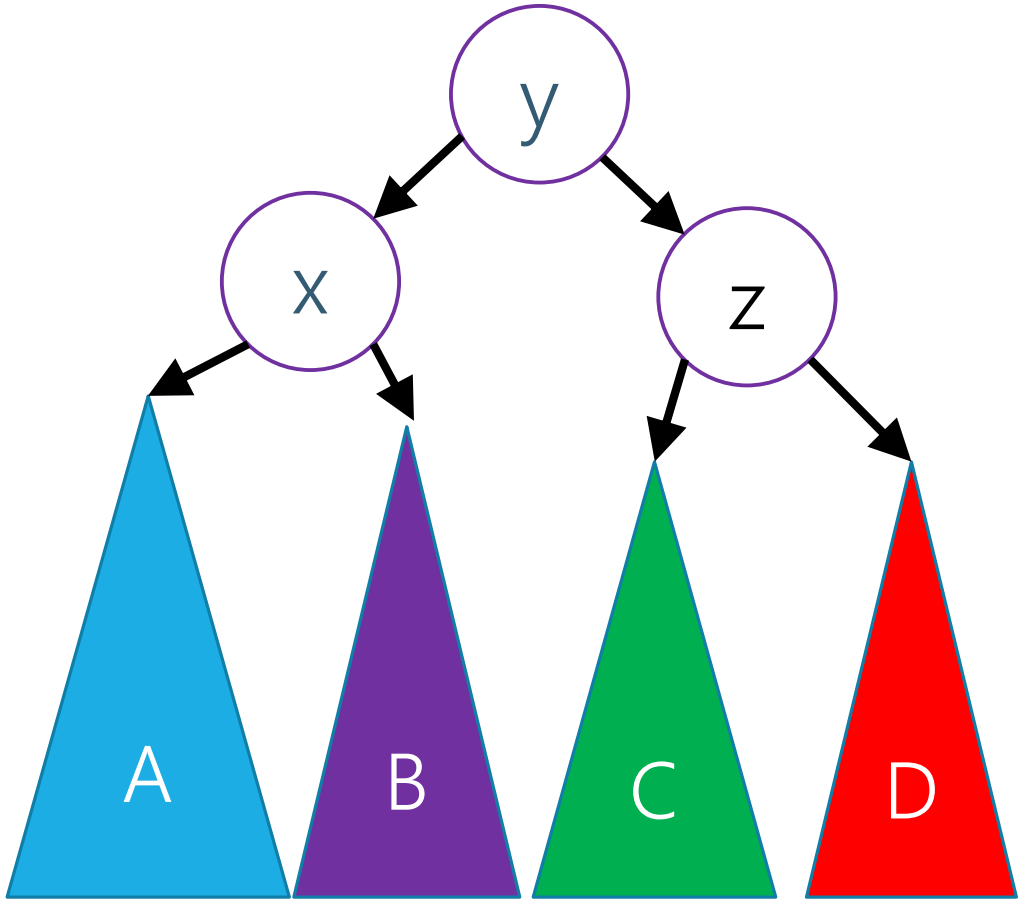
Double rotation, step 1



Double rotation, step 2

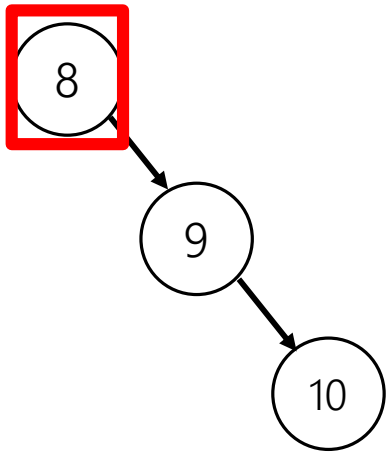


Four Types of Rotations

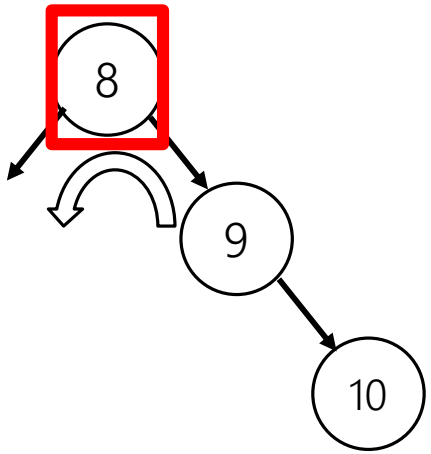


Insert location	Solution
Left subtree of left child (A)	Single right rotation
Right subtree of left child (B)	Double (left-right) rotation
Left subtree of right child (C)	Double (right-left) rotation
Right subtree of right child(D)	Single left rotation

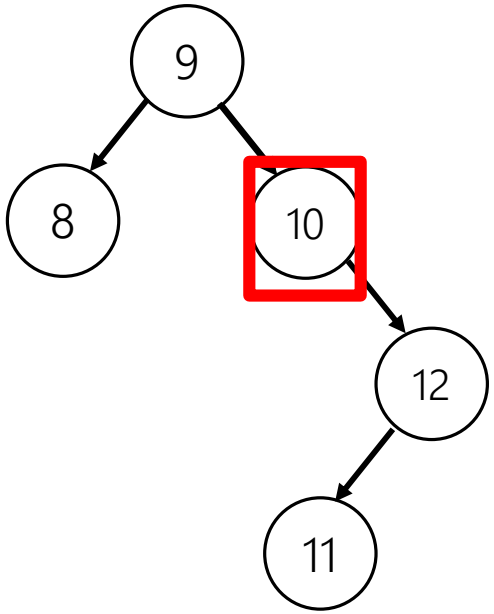
AVL Example: 8,9,10,12,11



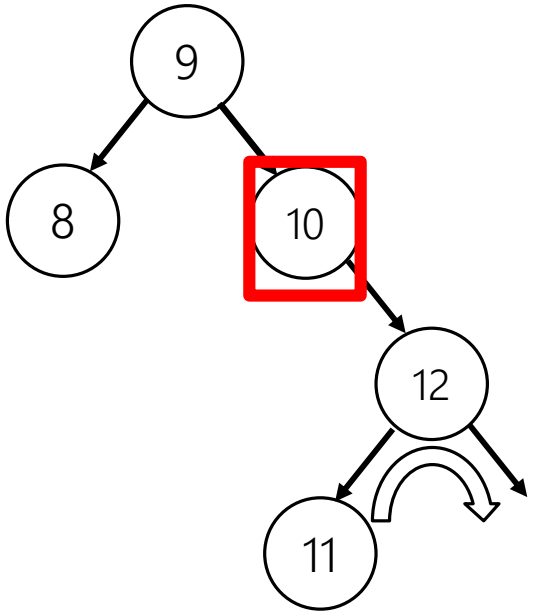
AVL Example: 8,9,10,12,11



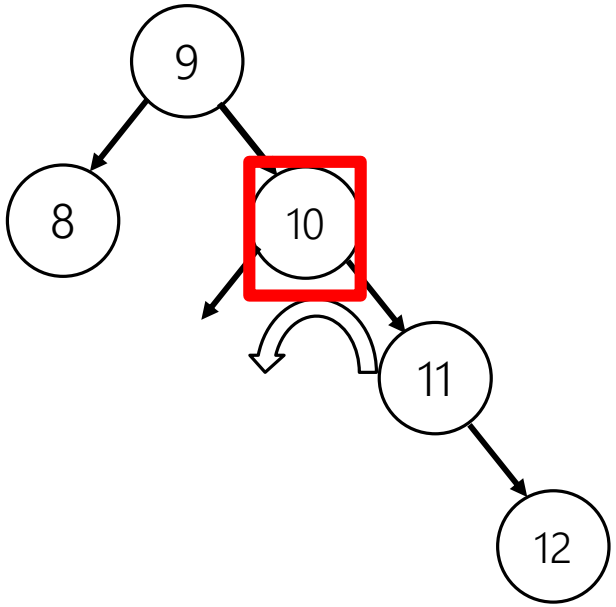
AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

How many rotations might we have to do?

How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

- Just go back up the tree from where we inserted.

How many rotations might we have to do?

- Just a single or double rotation on the lowest unbalanced node.
- A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion.

Deletion

In Project 2: We're letting you off the hook!

A similar set of rotations is possible to rebalance after a deletion.

(may require more than 1, but still constant #)

The textbook (or Wikipedia) can tell you more. You can implement these yourself in "Above and Beyond"

Aside: Traversals

What if, to save space, we didn't store heights of subtrees.

How could we calculate from scratch?

We could use a "traversal"

```
int height(Node curr){
    if(curr==null) return -1;
    int h = Math.max(height(curr.left),height(curr.right));
    return h+1;
}
```

Three Kinds of Traversals

```
InOrder(Node curr){
    InOrder(curr.left);
    doSomething(curr);
    InOrder(curr.right);
}
```

```
PreOrder(Node curr){
    doSomething(curr);
    PreOrder(curr.left);
    PreOrder(curr.right);
}
```

```
PostOrder(Node curr){
    PostOrder(curr.left);
    PostOrder(curr.right);
    doSomething(curr);
}
```

Traversals

If we have n elements, how long does it take to calculate height?

$\Theta(n)$ time.

The recursion tree (from the tree method) IS the AVL tree!

We do a constant number of operations at each node

In general, traversals take $\Theta(n \cdot f(n))$ time,

where `doSomething()` takes $\Theta(f(n))$ time.

Where Were We?

We used rotations to restore the AVL property after insertion.

If h is the height of an AVL tree:

It takes $\mathbf{O}(h)$ time to find an imbalance (if any) and fix it.

So the worst case running time of insert? $\mathbf{\Theta}(h)$.

Deletion? Time to find then rotate, i.e. $\mathbf{\Theta}(h)$.

h is always $O(\log n)$

More Practice

Insert 5,1,3,6,7,2,3

Wrap Up

AVL Trees:

$O(\log n)$ worst case `find`, `insert`, and `delete`.

Pros:

Much more reliable running times than regular BSTs.

Cons:

Tricky to implement

A little more space to store subtree heights

Other Dictionaries

There are lots of flavors of self-balancing search trees

"Red-black trees" work on a similar principle to AVL trees.

"Splay trees"
- Get $O(\log n)$ amortized bounds for all operations.

"Treaps" – a BST and heap in one (!)

Similar tradeoffs to AVL trees.

Next time: A completely different idea for a dictionary

Goal: $O(1)$ operations on average, in exchange for $O(n)$ worst case.