AVL Trees

CSE 332 Spring 2025
Lecture 8

Logistics

	Monday	Tuesday	Wednesday	Thursday	Friday
This Week	Ex 2 due		TODAY		Ex 3 due Ex 4 out
Next Week	Ex 5 out				Ex 4 due

A Better Implementation

What about BSTs?

Keys will have to be comparable.

	Insert	Find	Delete
Average	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

We're in the same position we were in for heaps
BSTs are great on average,
but we need to avoid the worst case.

Avoiding the Worst Case

Take II:

Here are some other requirements you might try. Could they work? If not what can go wrong?

Root Balanced: The root must have the same number of nodes in its left and right subtrees

Recursively Balanced: Every node must have the same number of nodes in its left and right subtrees.

Root Height Balanced: The left and right subtrees of the root must have the same height.

Avoiding the Worst Case

Take III:

The AVL condition

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

This actually works. To convince you it works, we have to check:

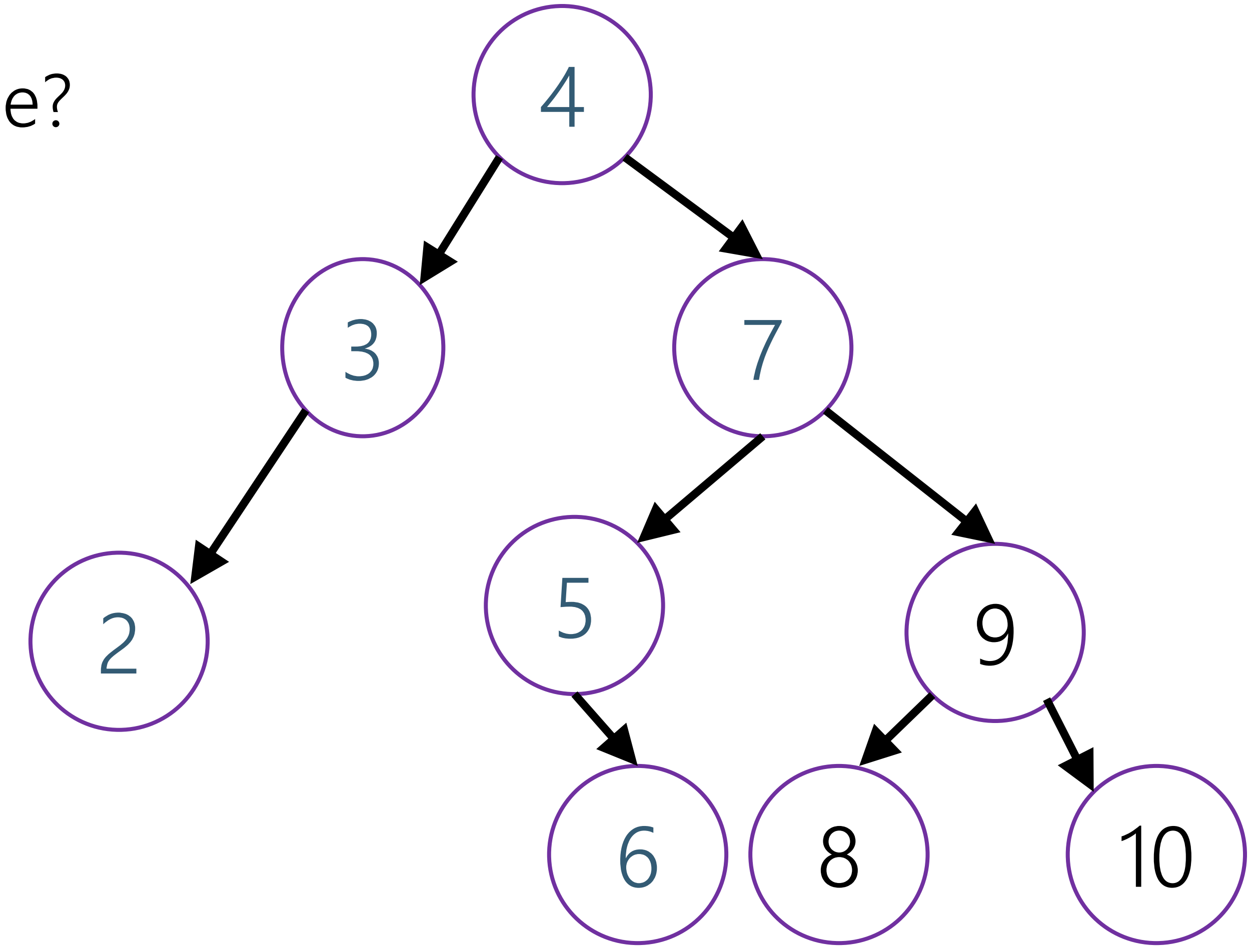
1. Such a tree must have height $O(\log n)$.

2. We must be able to maintain this property when inserting/deleting

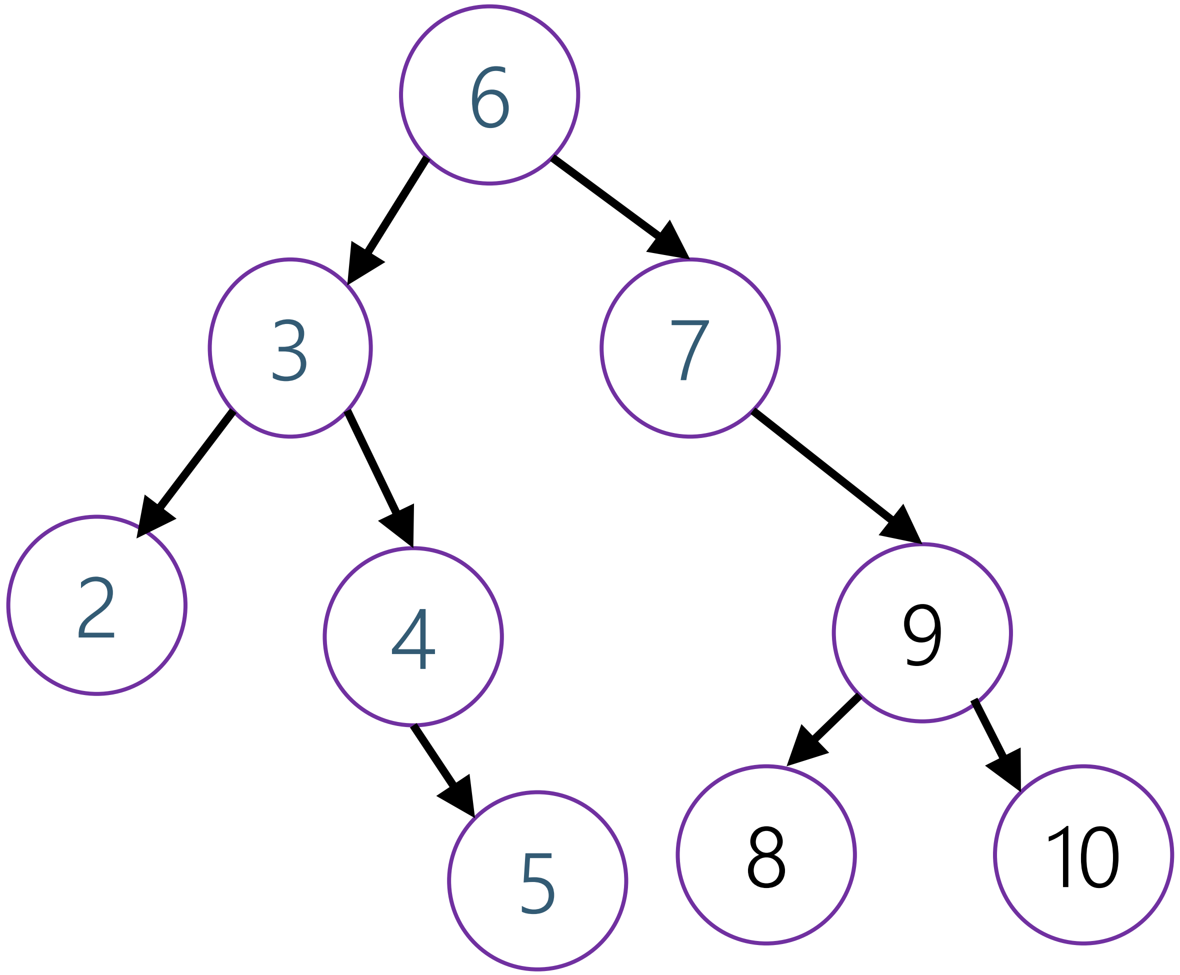
Warm-Up

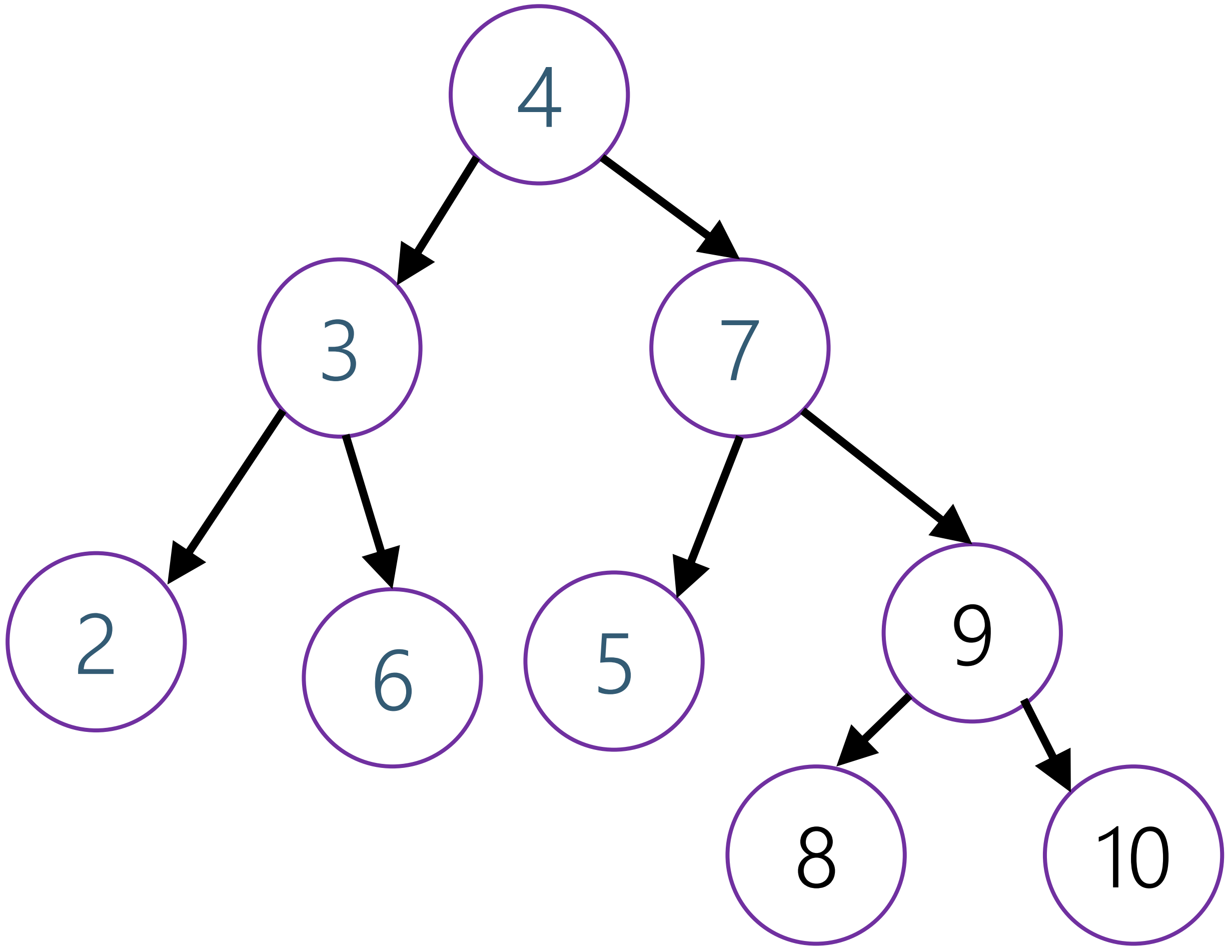
AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



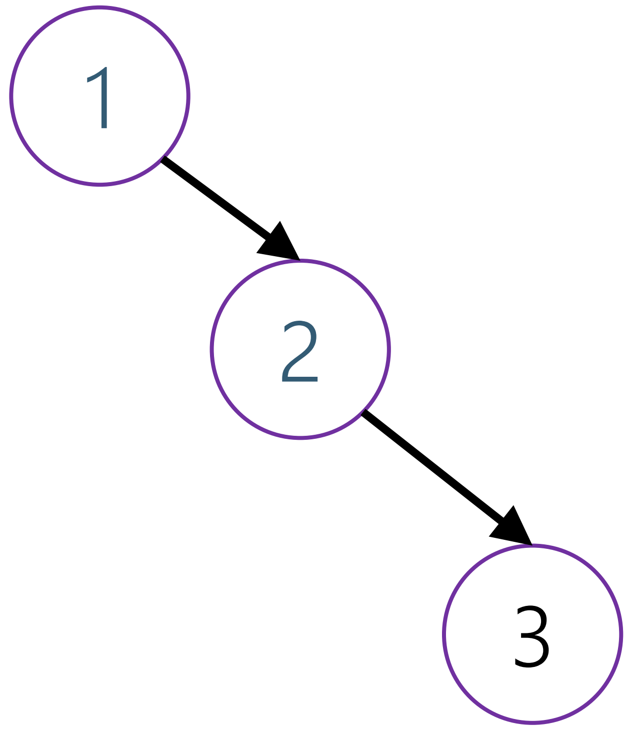
Are These AVL Trees?

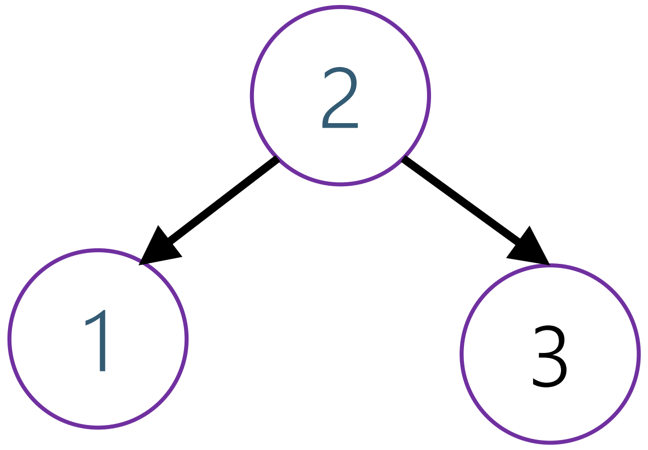




Insertion

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





Insertion

After you insert, check each node back up for balance

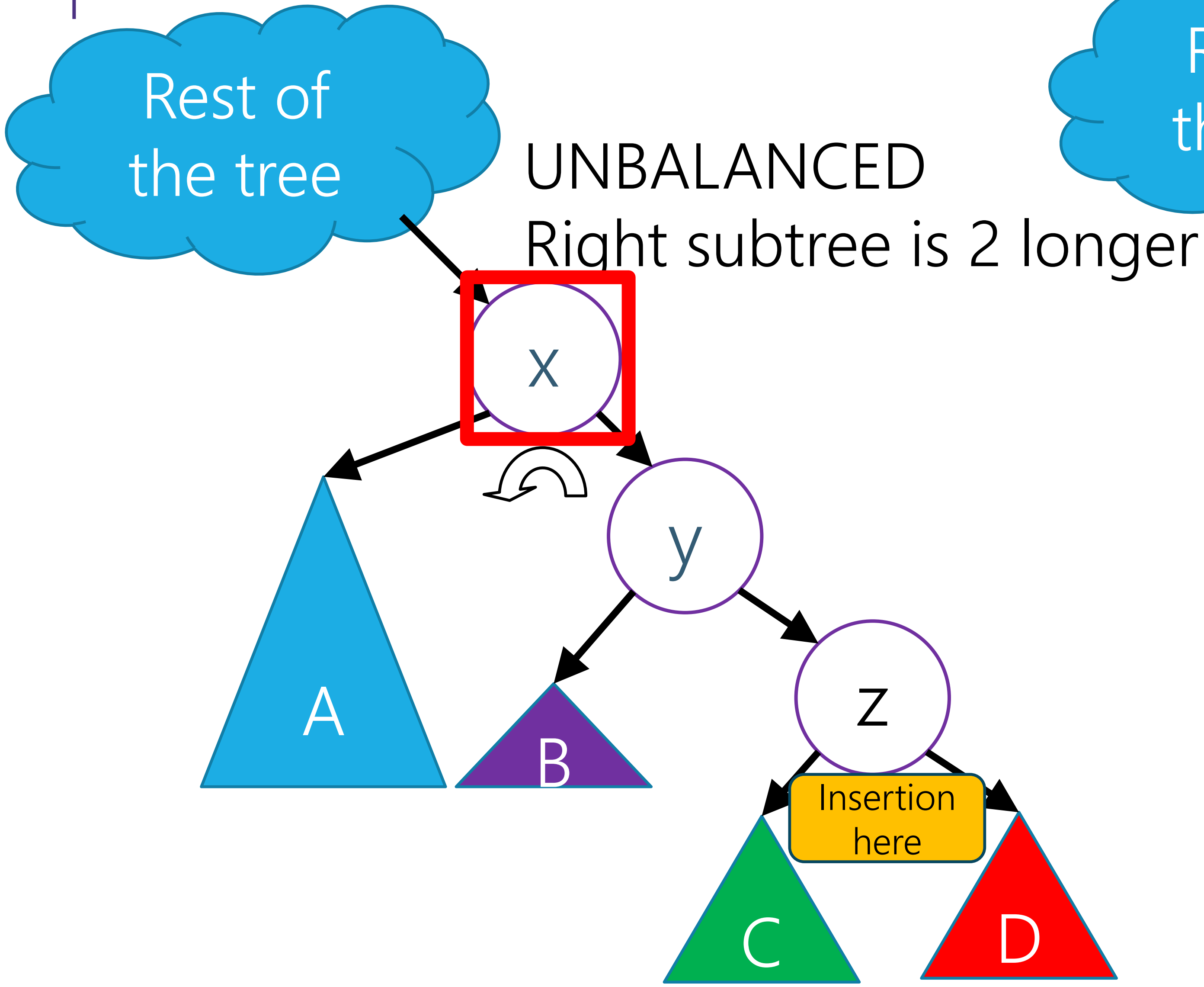
You rebalance at the lowest problem node

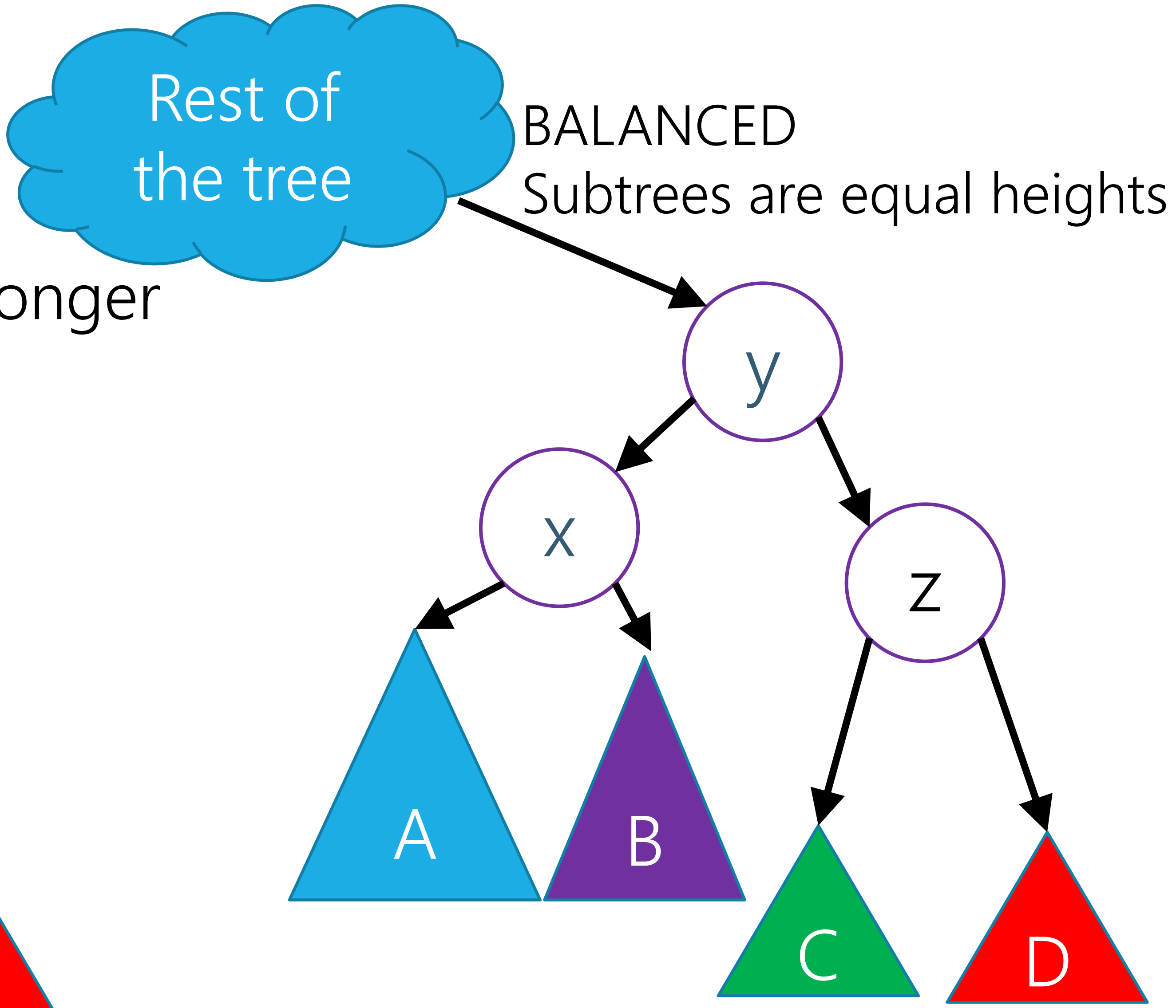
- The lowest node at which the AVL condition is violated---heights differ by 2.

Then ask “what directions were the insertion from here” for two levels from lowest problem node

- There’s a case for each of the 4 combinations:
- Left-left
- Left-right
- Right-left
- Right-right

Left Rotation





Left Single Rotation

From lowest-problem node, insertion happened in right child's right subtree.

Let x be problem node, y be x's right child, z be y's right child.

Let $a = \mathrm{x.left}, \mathrm{b} = \mathrm{y.left}, \mathrm{c} = \mathrm{z.left}, \mathrm{d} = \mathrm{z.right}$ //only need some of these.

Let $y.\mathrm{left}$=x; $y.\mathrm{right} = z$.

Let x.right=b, //don't need to update x.left, already correct value

//Don't need to update z's pointers

Let x's parent point to y. // will have to implement this via recursion.

Is it a BST Still?

From pre-insertion ordering, we know

$x < y < z$

$A < x$

$x < B < y$

$y < C < z$

$z < D$

Post-insertion, x, y, z in appropriate spots relative to each other

A left of x, B between x and y. C between y and z, D right of z

So we’re still a BST! :D

Are We balanced now?

Intuitively: z got longer, so it must be the too-long side.

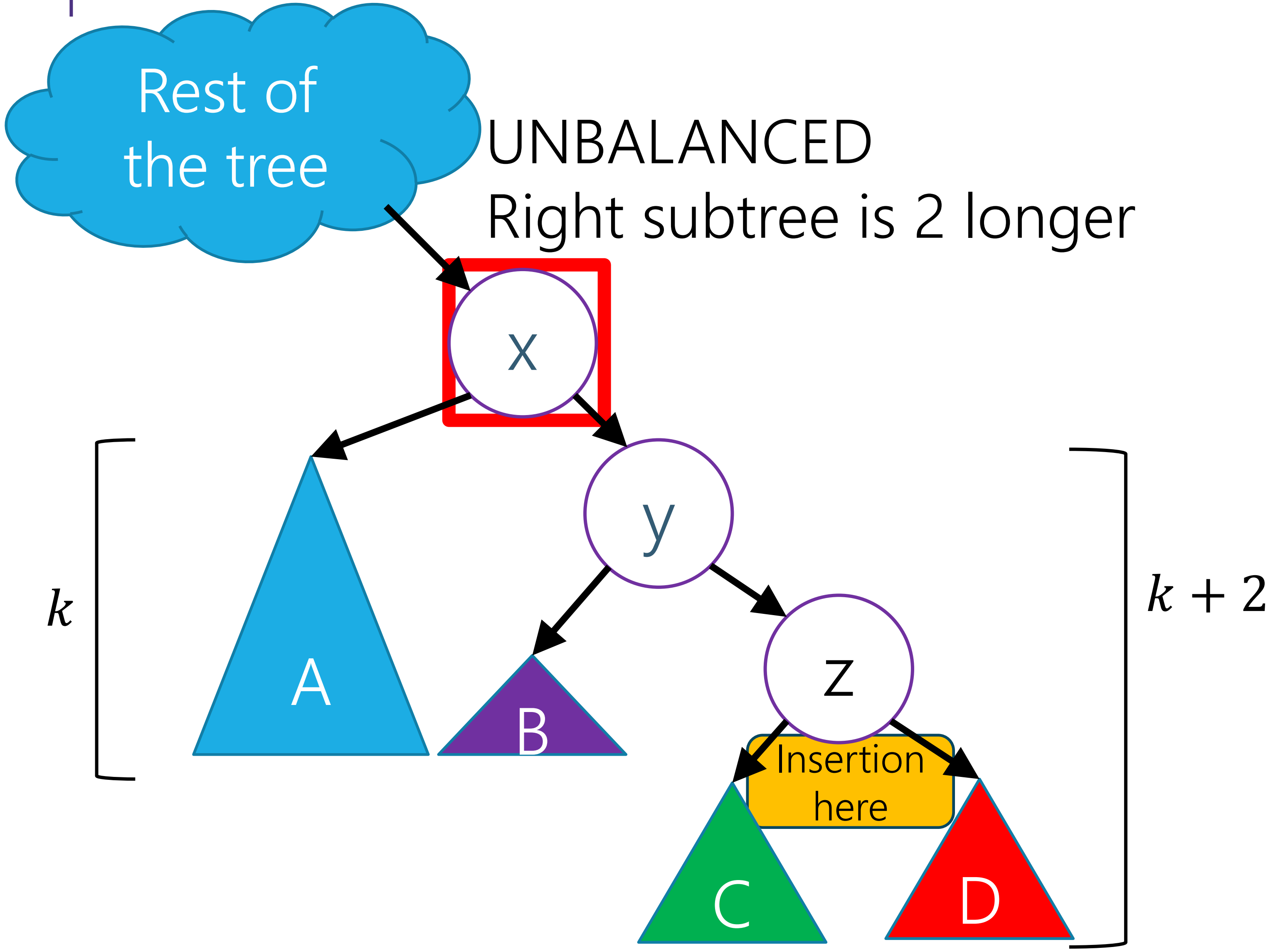
Rotation lengthened left subtree by 1, shrunk right subtree by 1.

Subtrees differed by 2 (since invariant kept us off-by-one before), now we are balanced!

The tree now has its pre-insertion height, so our ancestors don’t have to worry!

Sketch of balance proof

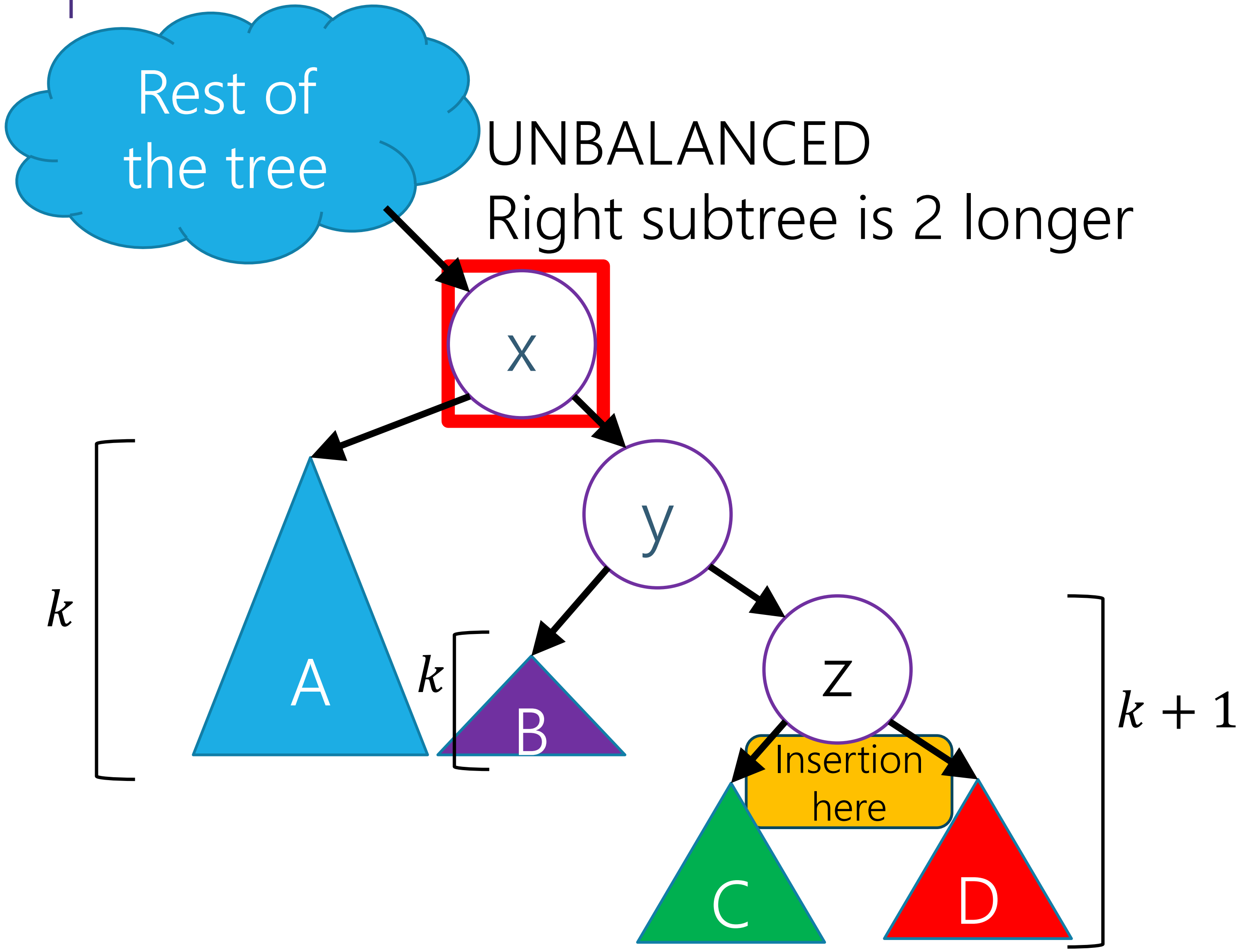
Left Single Rotation (insert happened ->right->right)



Let height of A be k.
Insertion adjusts heights by ≤ 1.

Since x is imbalanced now, right subtree must have height $k+2$ after insertion.

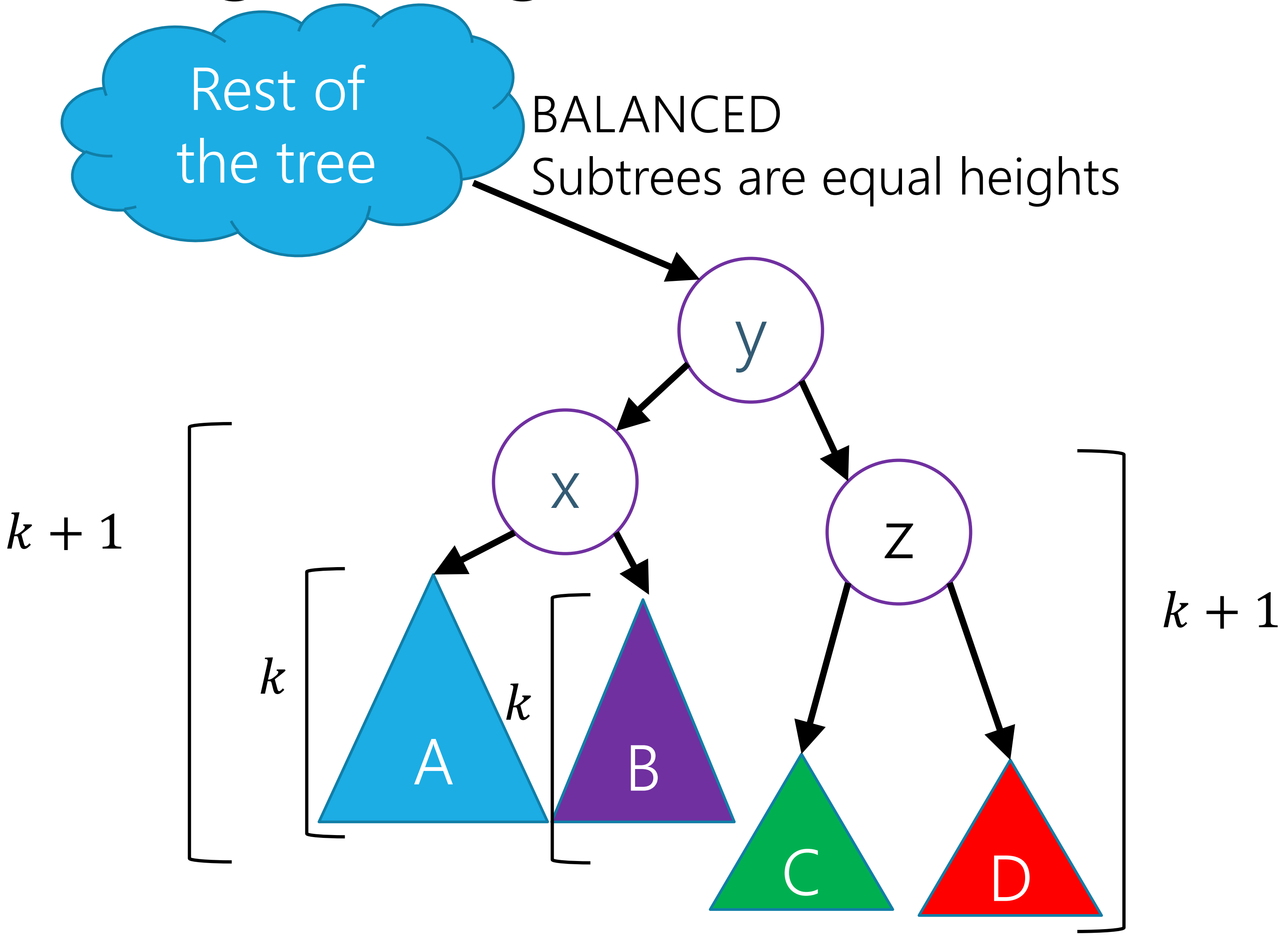
Left Single Rotation (insert happened ->right->right)



Cl: z has height $k+1$
y had height $k+2$, and right subtree must be longer side to cause imbalance).
Cl: B has height k.
y was balanced before insertion. For an imbalance to happen at x, z's increase in height must have increased height of tree rooted at y. So B must have height $\leq k$. But y is balanced now (x is lowest imbalanced), so height $\geq k$
$\Rightarrow$height is k.

Left Single Rotation (insert happened ->right->right)

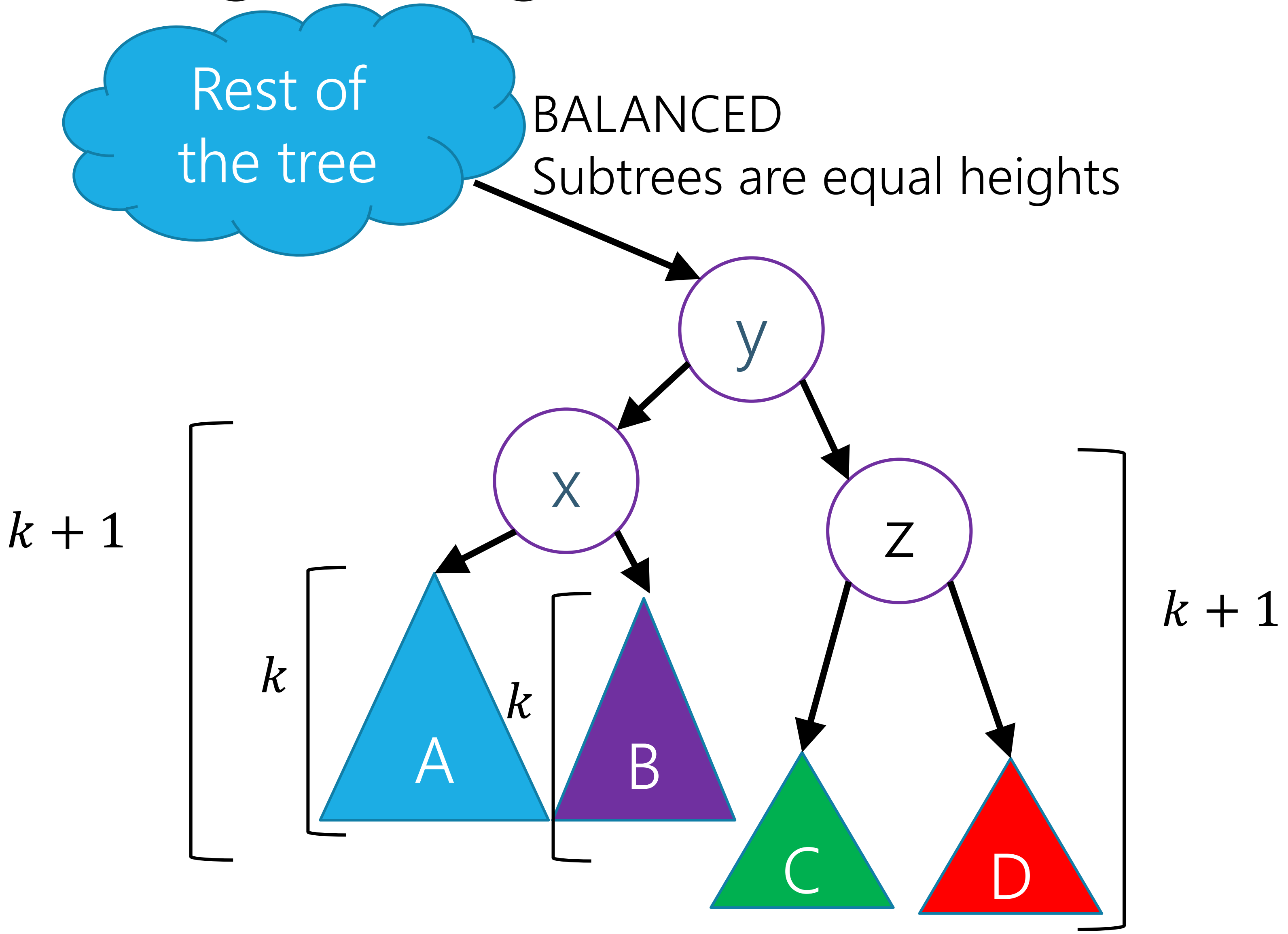
y is balanced (both subtrees height $k+1$).
x is balanced (both subtrees height k).
z is balanced (was not a problem node in recursion, and haven't rearranged its descendants).



Left Single Rotation (insert happened ->right->right)

What about y's parent?
Post rotation, this tree has height $k+2$
Post-insertion, pre-rotation, tree had height $k+3$, so pre-insertion, pre-rotation, tree had height $k+2$.

The height is the same now, so our parent must be balanced! No need to even check after this---no more rotations needed.



The Other Cases

There are other cases

There are three other cases, depending on where the insertion happened

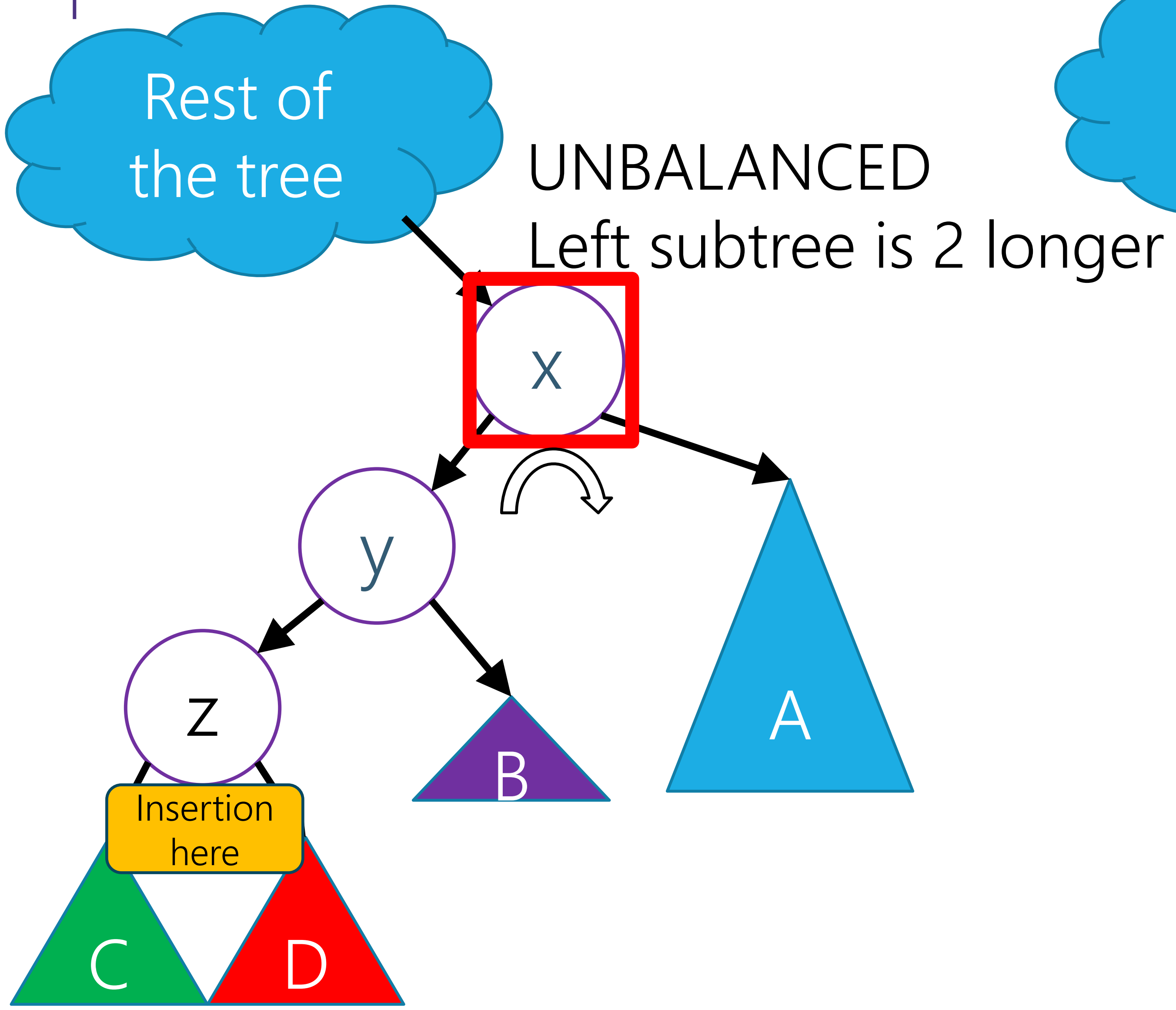
A left rotation handles when the insertion was
lowest imbalanced node -> right -> right

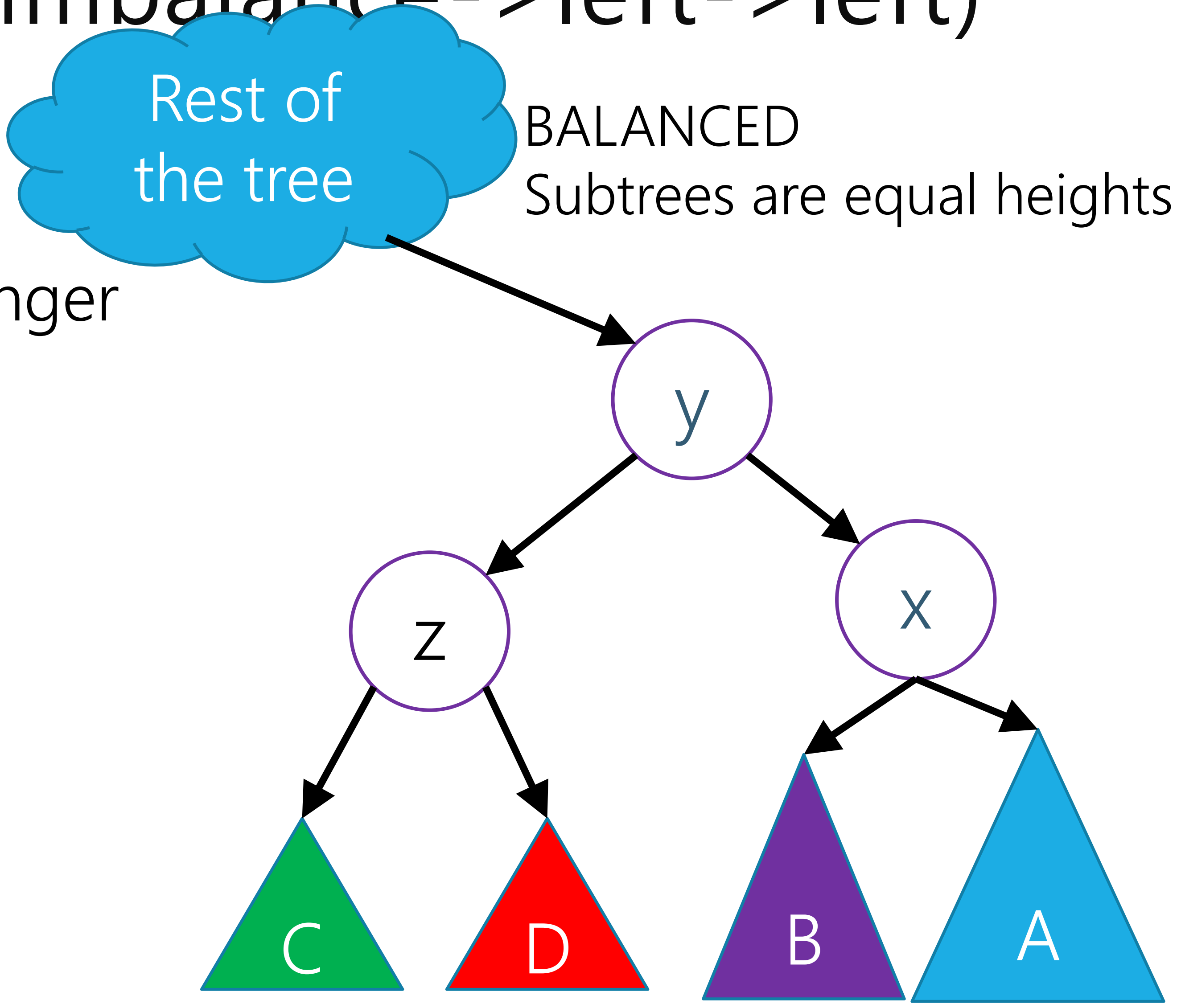
What if it was somewhere else?

Easiest is symmetric one

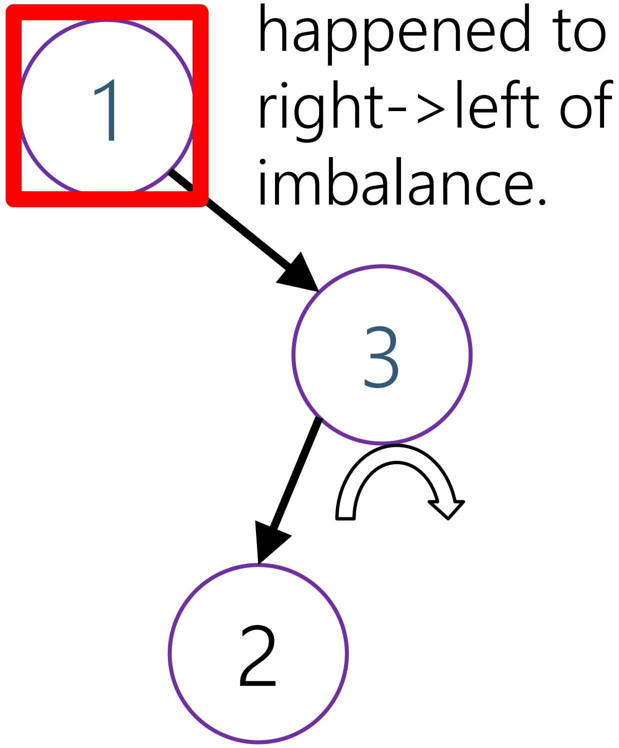
Insertion was
Lowest imbalanced node -> left -> left

Single Right Rotation (Insertion happened (imbalance->left->left)

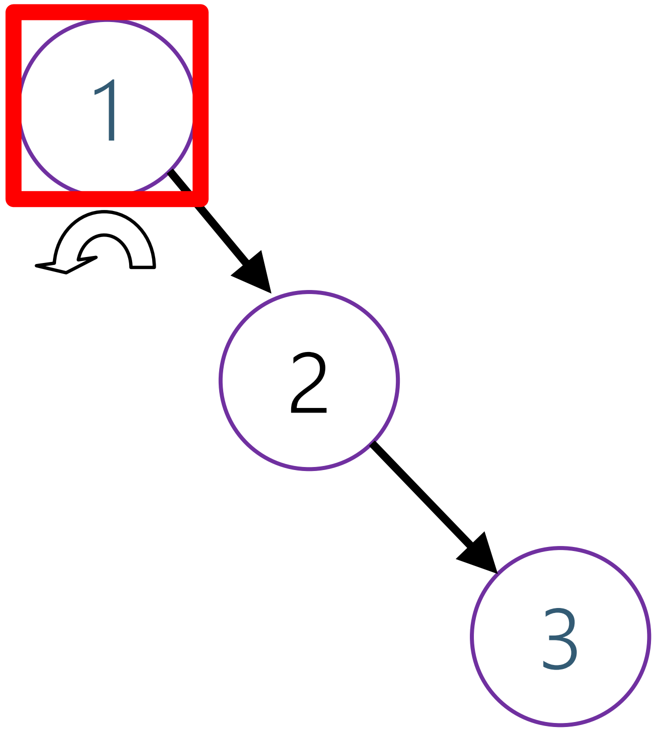


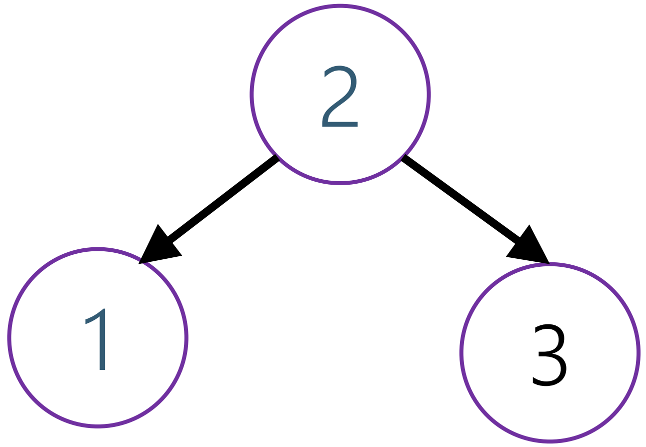


It Gets More Complicated



Insertion happened to right->left of imbalance.

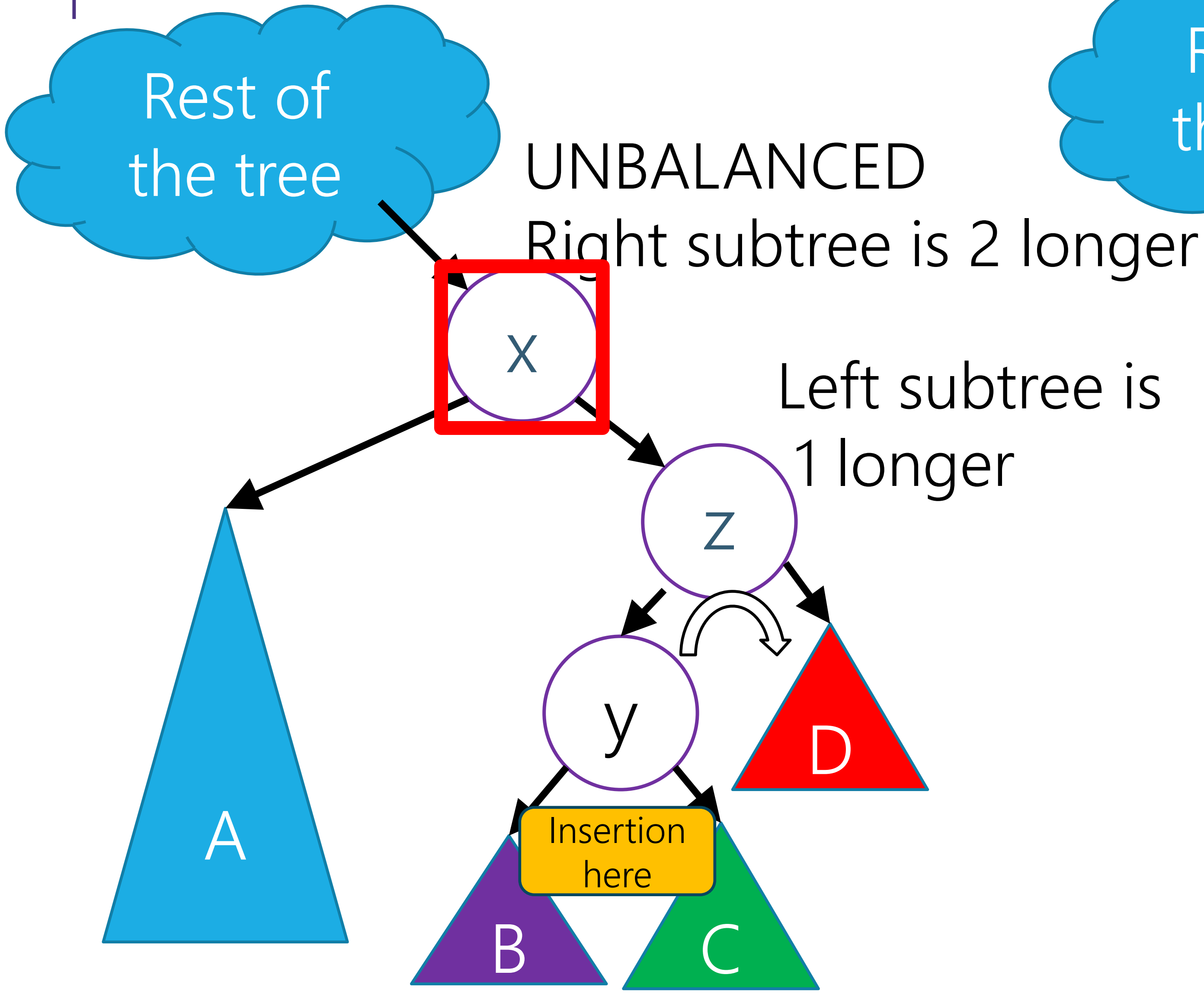


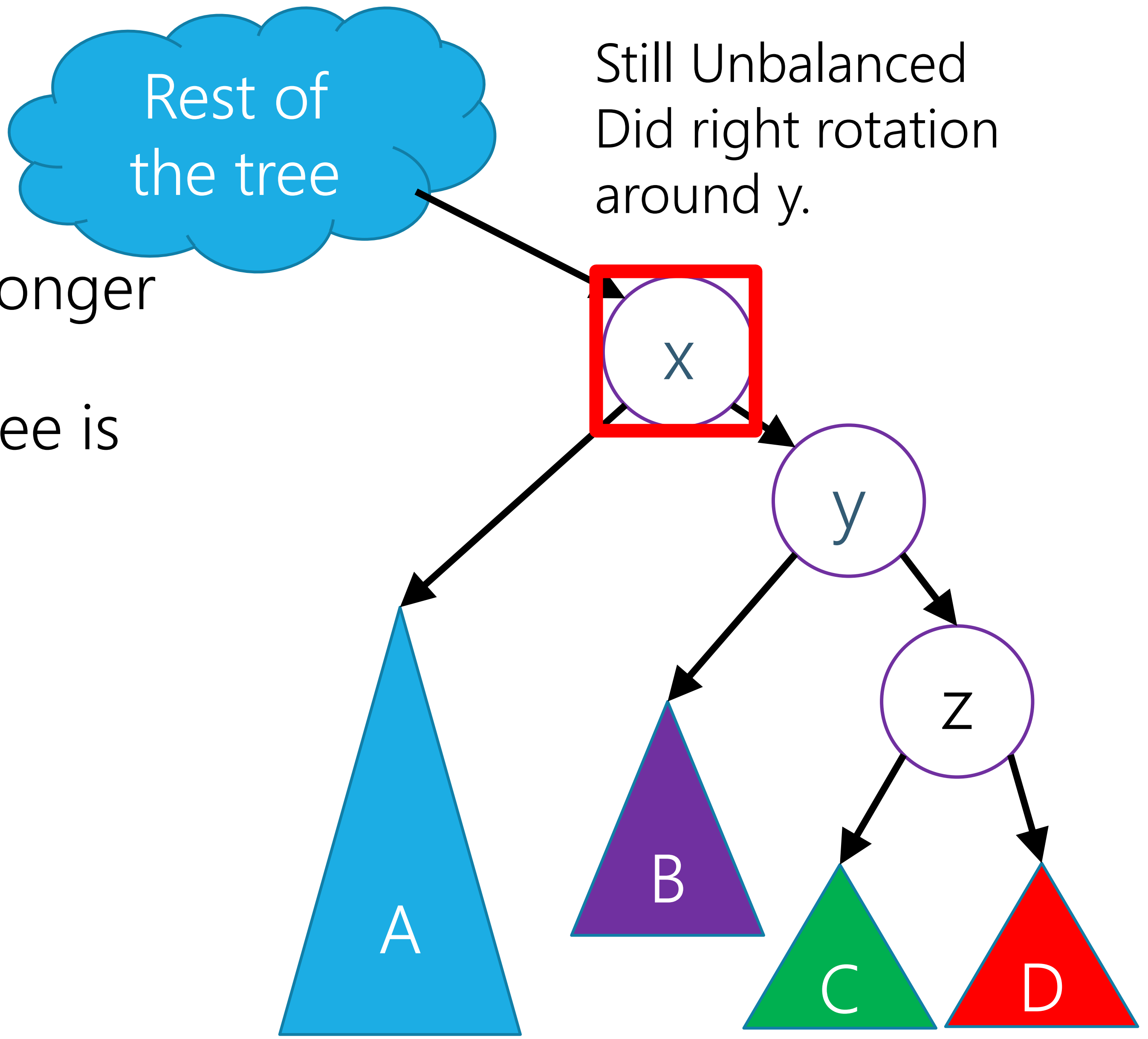


Can't do a left rotation
Do a "right" rotation around 3 first.
i.e. 3 is x, 2 is y, (null is z) from last slide.

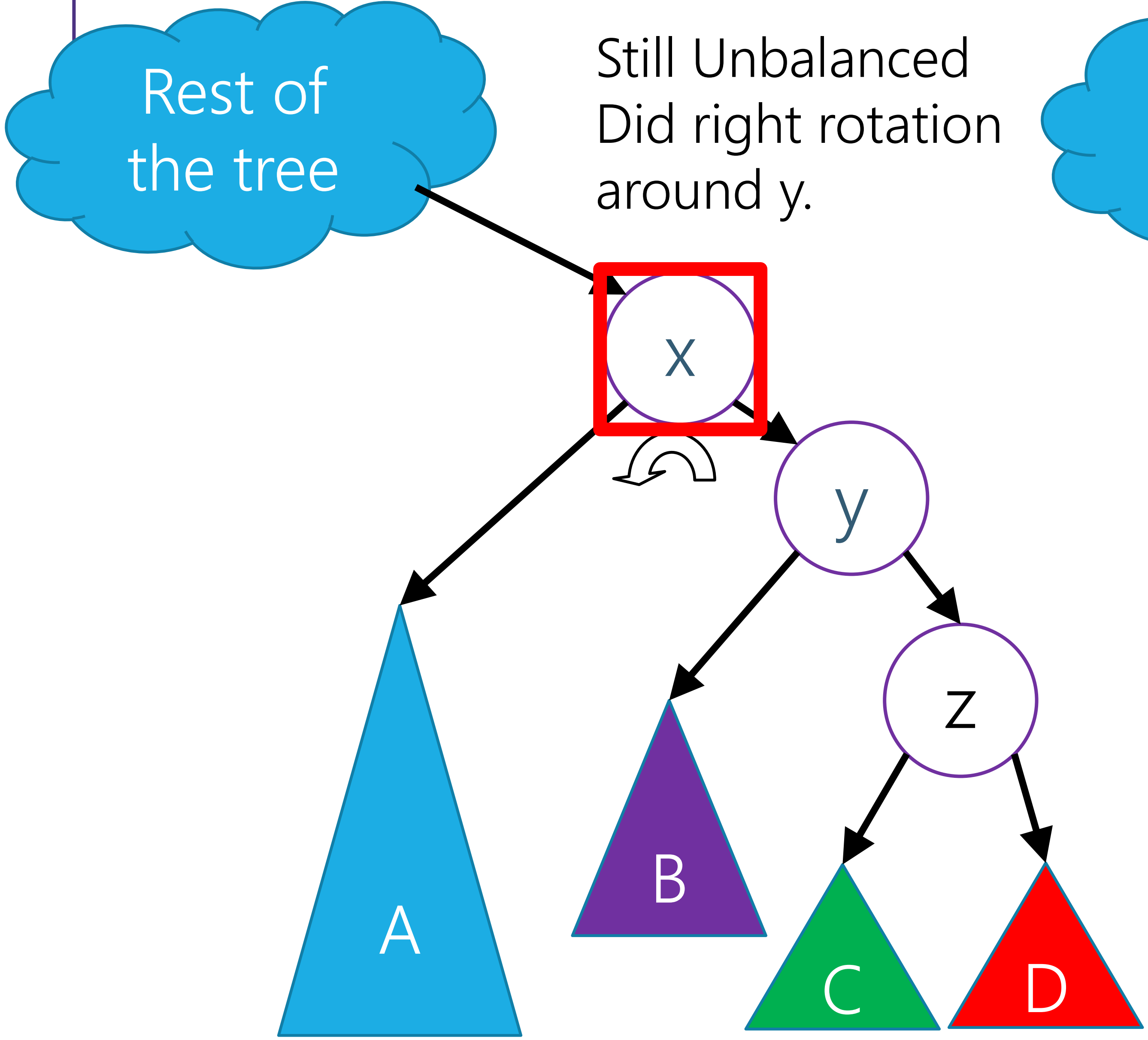
Now longer subtree is right->right of imbalance, do a left rotation around 1.
1 is x, 2 is y, 3 is z from 2 slides ago.

Right Left Rotation

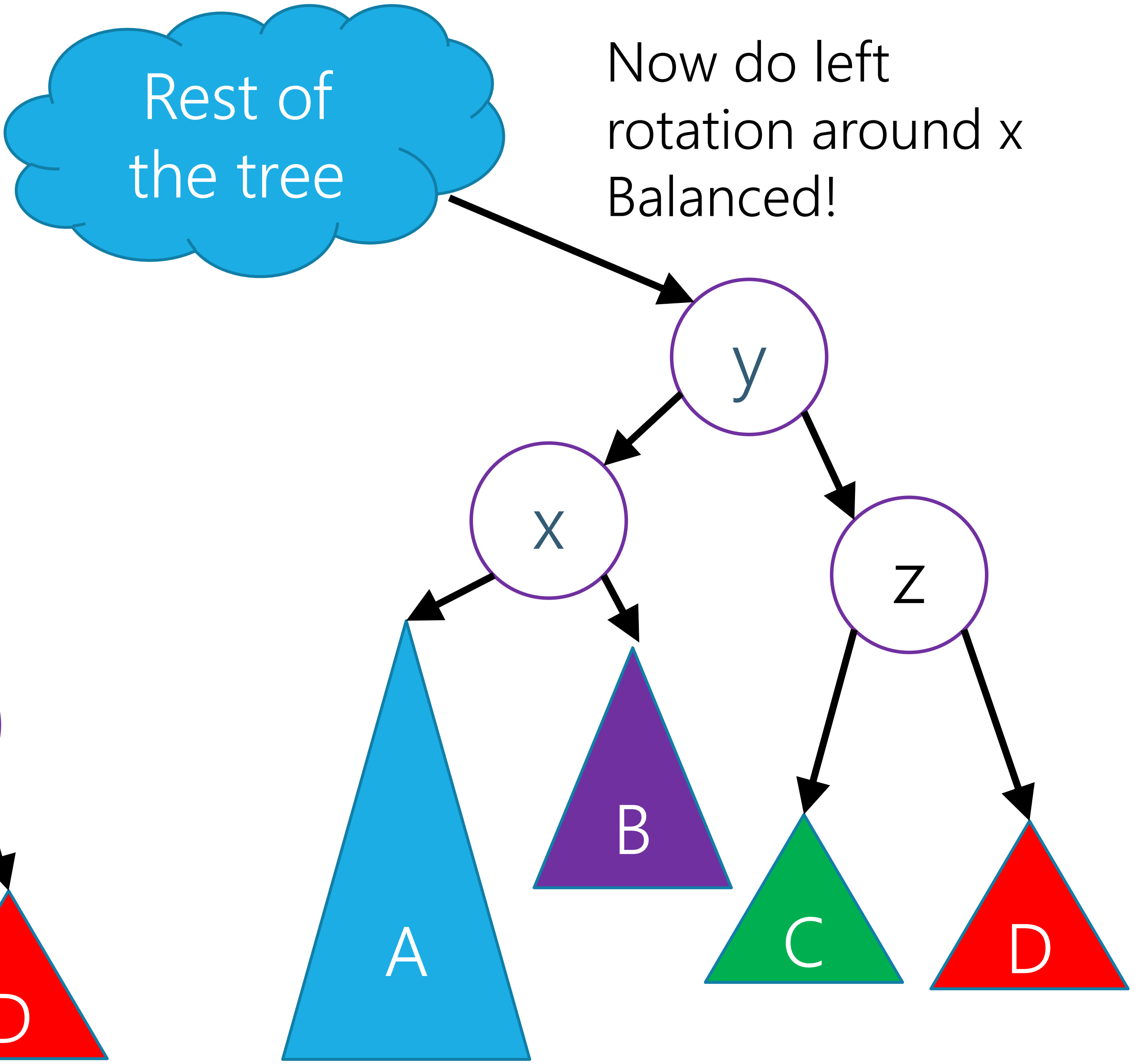




Right Left Rotation

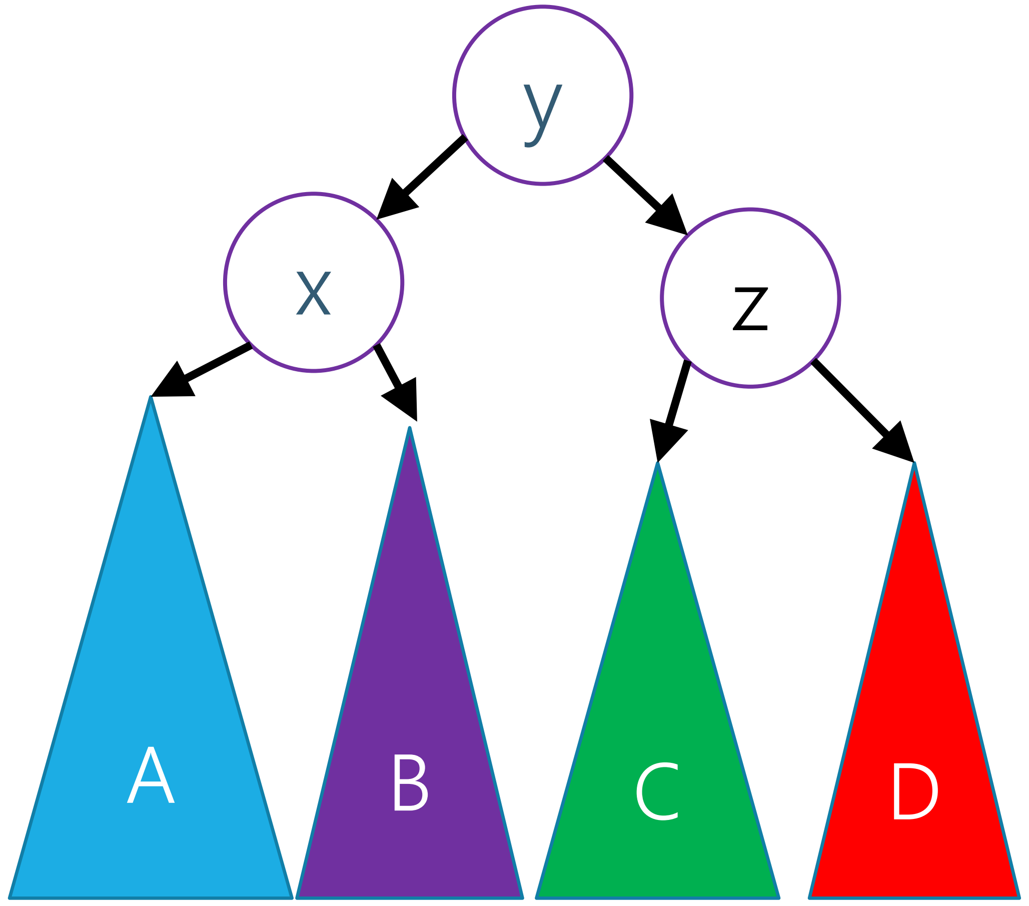


Still Unbalanced
Did right rotation around y.



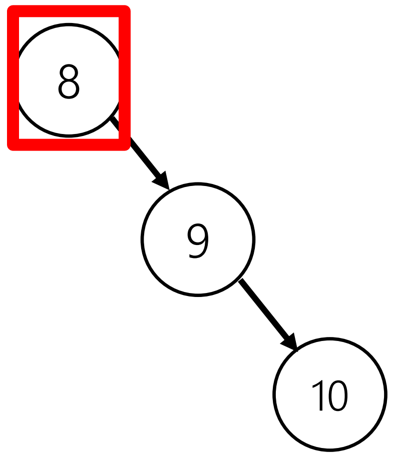
Now do left rotation around x
Balanced!

Four Types of Rotations

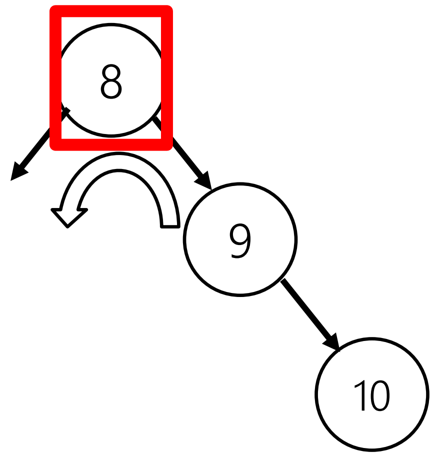


Insert location (relative to lowest imbalanced node)	Solution
Left subtree of left child (A)	Single right rotation
Right subtree of left child (B)	Double (left-right) rotation
Left subtree of right child (C)	Double (right-left) rotation
Right subtree of right child(D)	Single left rotation

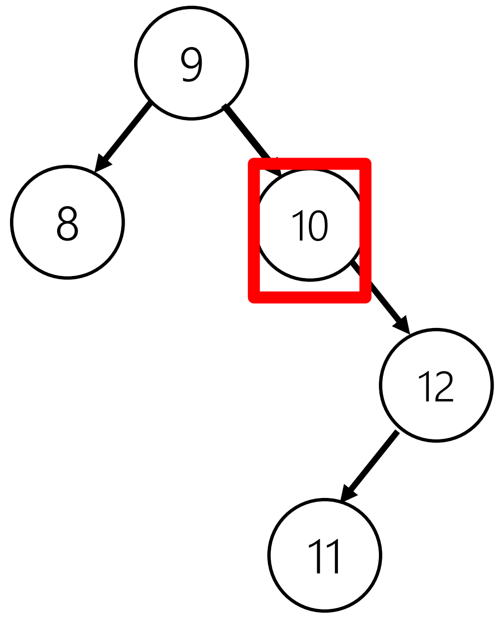
AVL Example: 8,9,10,12,11



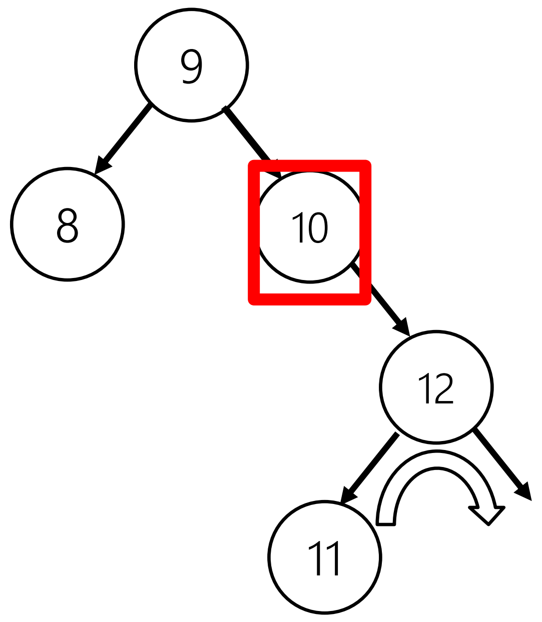
AVL Example: 8,9,10,12,11



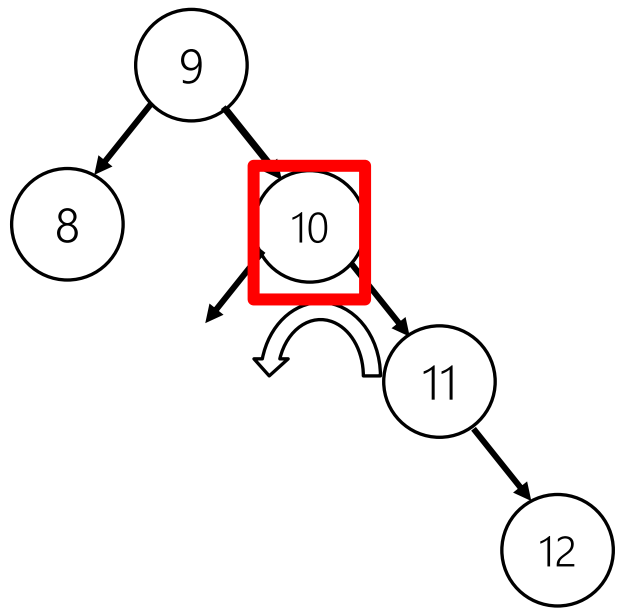
AVL Example: 8,9,10,12,11



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.

Some Related Topics

Lazy deletion, formally arguing that we have height log n

Aside: Lazy Deletion

Lazy Deletion: A general way to make `delete`() more efficient. (specifically, as efficient as `find`())

Don't remove the entry from the structure, just "mark" it as deleted.

Every node/array-index/etc. gets a Boolean.

Benefits:

- Much simpler to implement
- More efficient to delete (no need to shift values on every single delete)

Drawbacks:

- Extra space:
  - For the flag
  - More drastically, data structure grows with all insertions, not with the current number of items.
- Sometimes makes other operations more complicated.

Simple Dictionary Implementations

	Insert	Find	Delete
Unsorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Unsorted Array	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Sorted Linked List	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Sorted Array	$\Theta(m)$	$\Theta(\log m)$	$\Theta(\log m)$

We can do slightly better with lazy deletion, let m be the total number of elements ever inserted (even if later lazily deleted)
Think about what happens if a repeat key is inserted!

Deletion

In Exercise: Just do lazy deletion!

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

- The textbook (or Wikipedia) can tell you more.
- The delete rotations are more involved—you may have to rotate multiple times up the tree. But you'll still do $\Theta(\log n)$ rotations in total

Formally bounding height

Where Were We?

We used rotations to restore the AVL property after insertion.

If h is the height of an AVL tree:

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

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

Is h always $O(\log n)$? YES! These are all $\Theta(\log n)$. Let's prove it!

Bounding the Height

Suppose you have a tree of height h, meeting the AVL condition.

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

What is the minimum number of nodes in the tree?

If $h = 0$, then 1 node

If $h = 1$, then 2 nodes.

In general?

Bounding the Height

In general, let $N()$ be the minimum number of nodes in a tree of height h, meeting the AVL requirement.

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

Bounding the Height

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

We can try a recursion tree…

Bounding the Height

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

When unrolling we'll quickly realize:

- Something with Fibonacci numbers is going on.
- It's going to be hard to exactly describe the pattern.

The real solution (using deep math magic beyond this course) is

$N(h) \geq \phi^h - 1$ where ϕ is $\frac{1+\sqrt{5}}{2} \approx 1.62$

The Proof

To convince you that the recurrence solution is correct, I don't need to tell you where it came from.

I just need to prove it correct via induction.

We'll need this fact: $\phi + 1 = \phi^2$

It's easy to check by just evaluating $\left(\frac{1+\sqrt{5}}{2}\right)^2$

The Proof

$$N(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ N(h-1) + N(h-2) + 1 & \text{otherwise} \end{cases}$$

$$\phi + 1 = \phi^2$$

Base Cases: $\phi^0 - 1 = 0 < 1 = N(0)$ $\phi^1 - 1 = \phi - 1 \approx 0.62 < 2 = N(1)$

Inductive Step

Inductive Hypothesis: Suppose that $N(h) > \phi^h - 1$ for $h < k$.

Inductive Step: We show $N(k) > \phi^k - 1$.

$N(k) = N(k-1) + N(k-2) + 1$	definition of $N()$
$> \phi^{k-1} - 1 + \phi^{k-2} - 1 + 1$	by IH (note we need a strong hypothesis here)
$= \phi^{k-1} + \phi^{k-2} - 1$	algebra
$= \phi^{k-2}(\phi + 1) - 1$	
$= \phi^{k-2}(\phi^2) - 1$	fact from last slide
$= \phi^{k+1} - 1$	

What’s the point?

The number of nodes in an AVL tree of height h is always at least $\phi^h - 1$

So in an AVL tree with n elements, the height is always at most $\log_\phi(n+1)$

In big-O terms, that’s enough to say the height is $\Theta(\log n)$.

So our AVL trees really do have $\Theta(\log n)$ worst cases for insert, find, and delete!

Is the improvement worth it?

Wrap Up

Was this…worth it?

That was a lot of work (and there are going to be a lot of pointers to move around to actually make those trees…)

Is going from n to $\log n$ a good enough improvement?

Wrap Up

	Insert	Find	Delete
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Does It Matter?

$\Theta(\log n)$ is the most common running time between $\Theta(1)$ and $\Theta(n)$.

Which of these times is $\log n$ between?

$\Theta(1)$	A	$\Theta(n^{0.01})$	B	$\Theta(n^{0.1})$	C	$\Theta(n^{0.5})$	D	$\Theta(n)$

Fill out the poll everywhere so Robbie knows how much to explain
Pollev.com/robbie

Does It Matter?

We sped up from $\Theta(n)$ to $\Theta(\log n)$.

Suppose we could handle an input of size k with the $\Theta(n)$ algorithm. If the constant factors are small (and similar) we'll be able to handle an input of size 2^k in the same time.

Is squaring the size of the input an issue?

For $\Theta(n)$ algorithms, yes, it will almost square the time it takes.

- "Almost" is because the constant factors don't change.

For $\Theta(\log n)$ algorithms, no. You'll at most double the time it takes.

- "At most" because the lower order terms change differently.

Does It Matter?

Let's say you have an algorithm that takes n milliseconds or $\log_2 n$ milliseconds. How long does it take to run when n is…

	1000	10000	100000	1000000	10000000	100000000
$\boldsymbol{log_2(n)}$	0.010 s	0.013s	0.017s	0.020s	0.023s	0.027s
$\boldsymbol{n}$	1 second	10 seconds	2 minutes	17 minutes	2.7 hours	1 day

$\log_2(number\ of\ particles\ in\ the\ universe) \approx 268.$

Constant factors do matter, (and for reasons you'll learn in 351, constant factors get worse as your dataset gets bigger) but you can't make a dataset so big that the $\log_2(n)$ part will be what makes it intractable.

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.

"Scapegoat trees"

"Treaps" – a BST and heap in one (!)

Similar tradeoffs to AVL trees.

Next week: A completely different idea for a dictionary

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

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

Aside: Traversals

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,

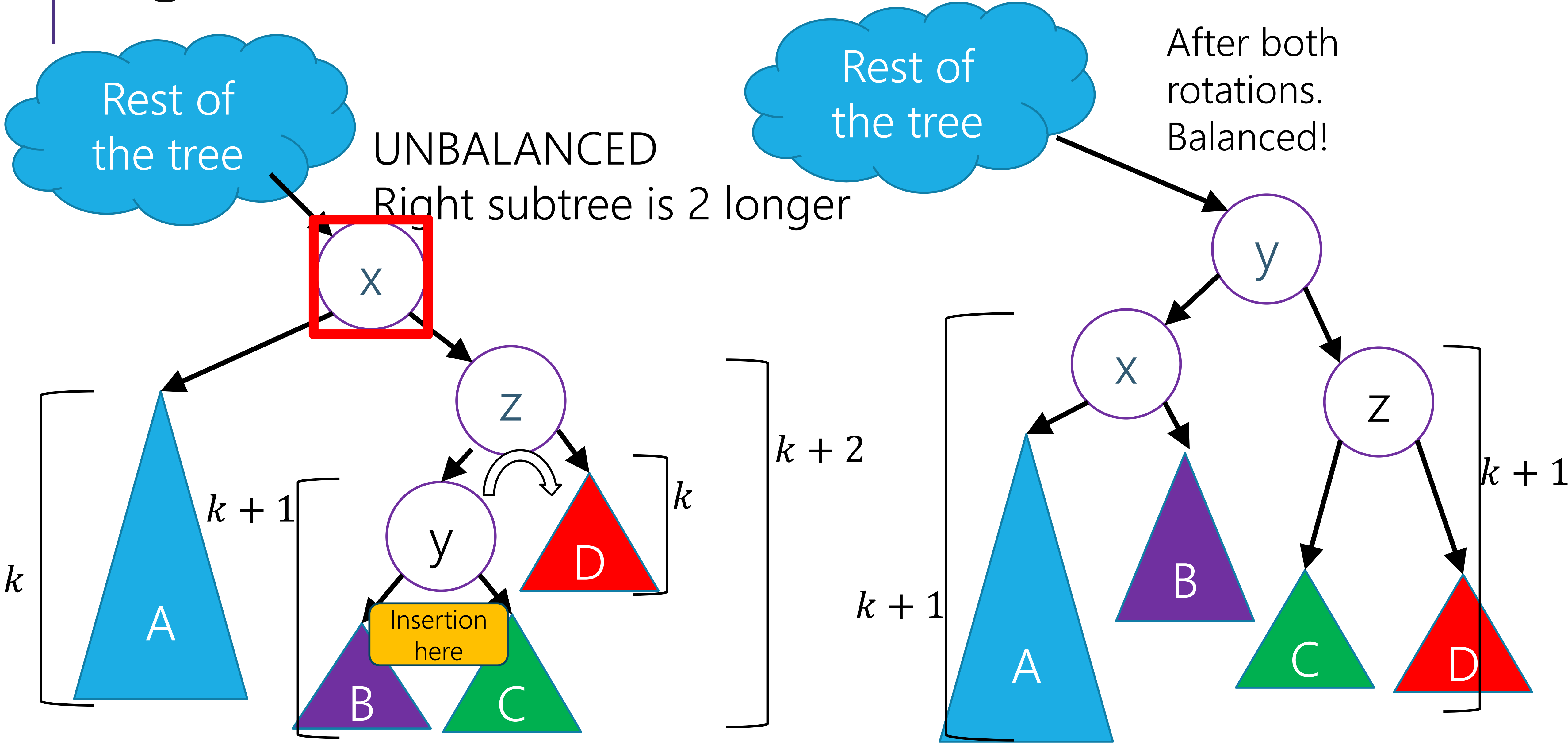
where `doSomething()` takes $\Theta\left(f(n)\right)$ time.

Optional: Some more math slides

Right Left Rotation

UNBALANCED
Right subtree is 2 longer

After both
rotations.
Balanced!



What happens if you try to solve the $N()$ recurrence?

$N(h) = N(h-1) + N(h-2) + 1$

$= N(h-2) + N(h-3) + 1 + N(h-2) + 1$

$= 2N(h-2) + N(h-3) + 1 + 1$

$= 2(N(h-3) + N(h-4) + 1) + N(h-3) + 1 + 1$

$= 3N(h-3) + 2N(h-4) + 2 + 1 + 1$

$= 3(N(h-4) + N(h-5) + 1) + 2N(h-4) + 2 + 1 + 1$

$= 5N(h-4) + 3N(h-5) + 3 + 2 + 1 + 1$

$= 5(N(h-5) + N(h-6) + 1) + 3N(h-5) + 3 + 2 + 1 + 1$

$= 5N(h-6) + 8N(h-5) + 5 + 3 + 2 + 1 + 1$

$\dots$