Dictionaries I

CSE 332 Spring 2025
Lecture 7

Logistics

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

Warm Up

Write a recurrence to represent the running time of this code

```
int Mystery(int n){
      if(n <= 5)
            return 1;
      for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                  System.out.println("hi");
            }
      }
      return n*Mystery(n/2);
}
```

$$T(n) = \begin{cases} 2 & \text{if } n \leq 5 \\ T\left(\frac{n}{2}\right) + 3n^2 + 2n + 2 & \text{otherwise} \end{cases}$$

Outline

Some Last Comments on Tree Method

Two new (old?) ADTs

- Dictionaries
- Sets

Review BSTs

Intro AVL trees

Some Last Comments on Recurrences

Extra video this week with another example of using the tree method coming soon.

You'll end up with a summation (level 0 to level height - 1)

- Sometimes you need a formula to find the closed-form. We have a reference sheet with the ones you need.

There are many possible effects of the multiple levels

- Sometimes the root level is the dominant term
- Sometimes every level is equal
- Sometimes the last level is the dominant term

Don't ignore the levels after the root! In section examples we'll skip the base case level; it's within a constant factor of the level before, so even if it's the biggest one you'll get the correct big-O from the level before.

Our Next ADT

Dictionary ADT

state

Set of (key, value) pairs

behavior

insert(key, value) – inserts (key, value) pair. If key was already in dictionary, overwrites the previous value.

find(key) – returns the stored value associated with key.

delete(key) – removes the key and its value from the dictionary.

Real world intuition:
keys: words
values: definitions

Dictionaries are often called "maps"

Our Next ADT

Set ADT

state

Set of elements

behavior

insert(element) – inserts element into the set.

find(element) – returns true if element is in the set, false otherwise.

delete(key) – removes the key and its value from the dictionary.

Usually implemented as a dictionary with values “true” or “false”

Later in the course we’ll want more complicated set operations like **union(set1, set2)**

Uses of Dictionaries

Dictionaries show up all the time.

There are too many applications to really list all of them:

- Phonebooks
- Indexes
- Databases
- Operating System memory management
- The internet (DNS)
- …

Any time you want to organize information for easy retrieval.

We're going to design two *completely different* implementations of Dictionaries – (and we might do a third at the end of the quarter).

Simple Dictionary Implementations

	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	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

What are the worst case running times for each operation if you have n (key, value) pairs.

Assume the arrays do not need to be resized.

Think about what happens if a repeat key is inserted! (need to replace value)

Simple Dictionary Implementations

	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	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

What are the worst case running times for each operation if you have n (key, value) pairs.

Assume the arrays do not need to be resized.

Think about what happens if a repeat key is inserted!

A Better Implementation

What about BSTs?

Keys will have to be comparable…

	Insert	Find	Delete
Average			
Worst			

A Better Implementation

What about BSTs?

Keys will have to be comparable…

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

Let’s talk about how to implement delete.

Deletion from BSTs

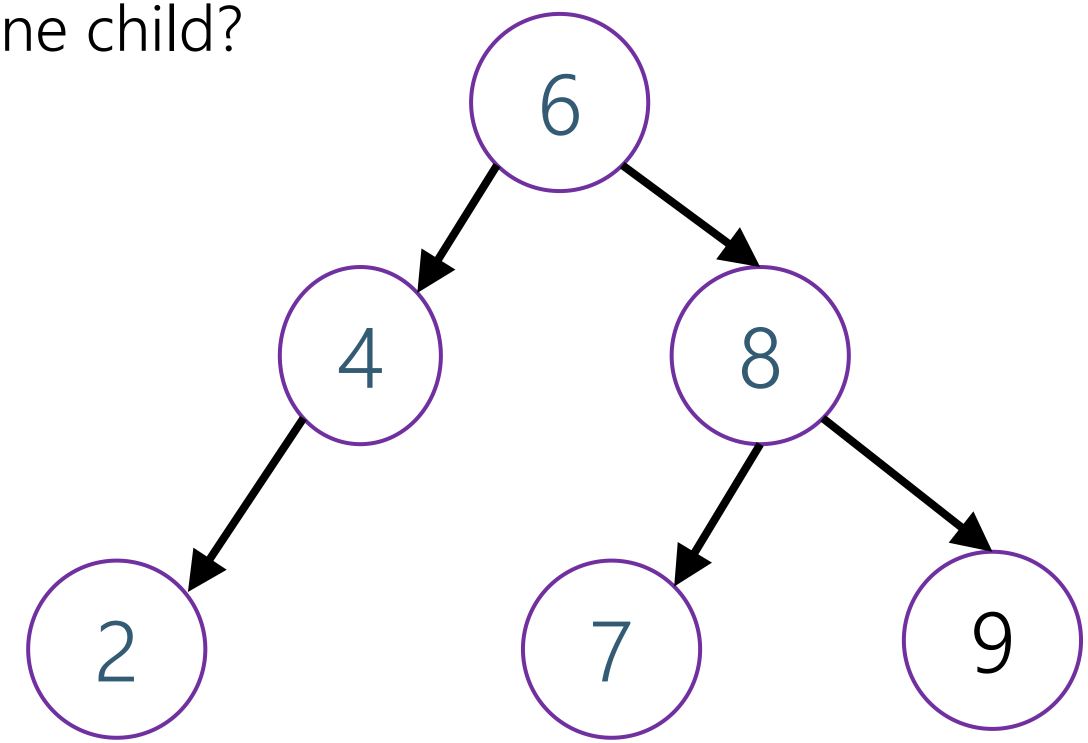
Deleting will have three steps:

- Finding the element to delete
- Removing the element
- Restoring the BST property

Deletion – Easy Cases

What if the elements to delete is:

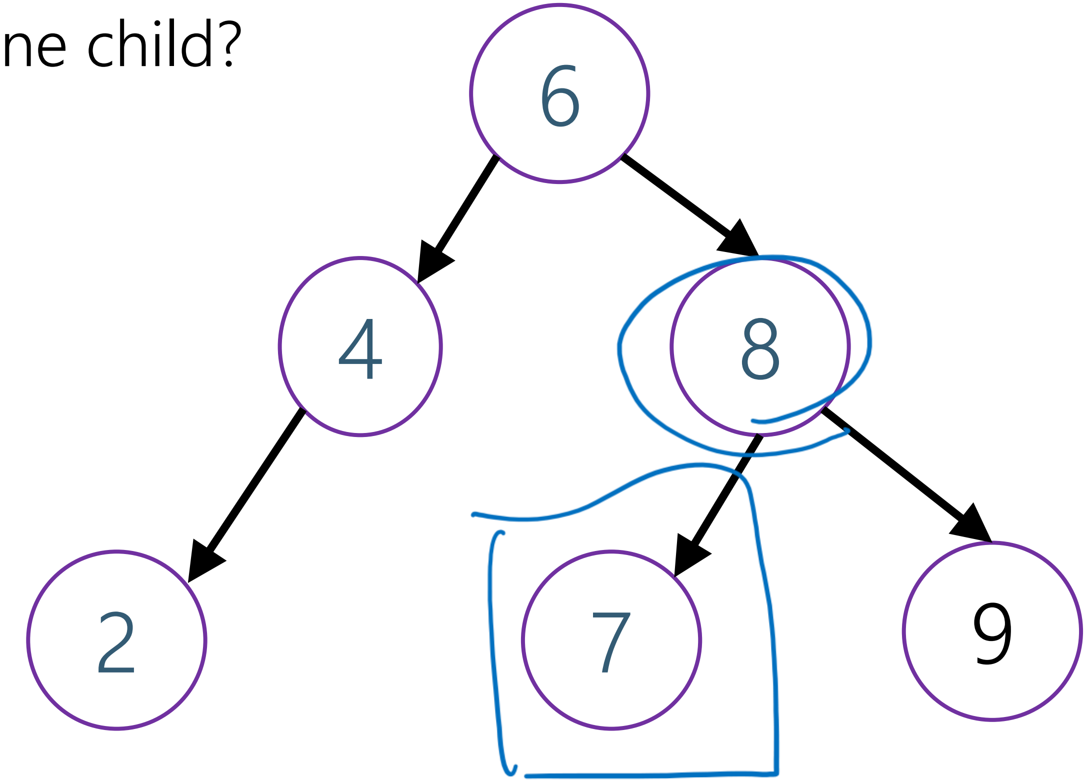
- A leaf?
- Has exactly one child?



Deletion – Easy Cases

What if the elements to delete is:

- A leaf?
- Has exactly one child?



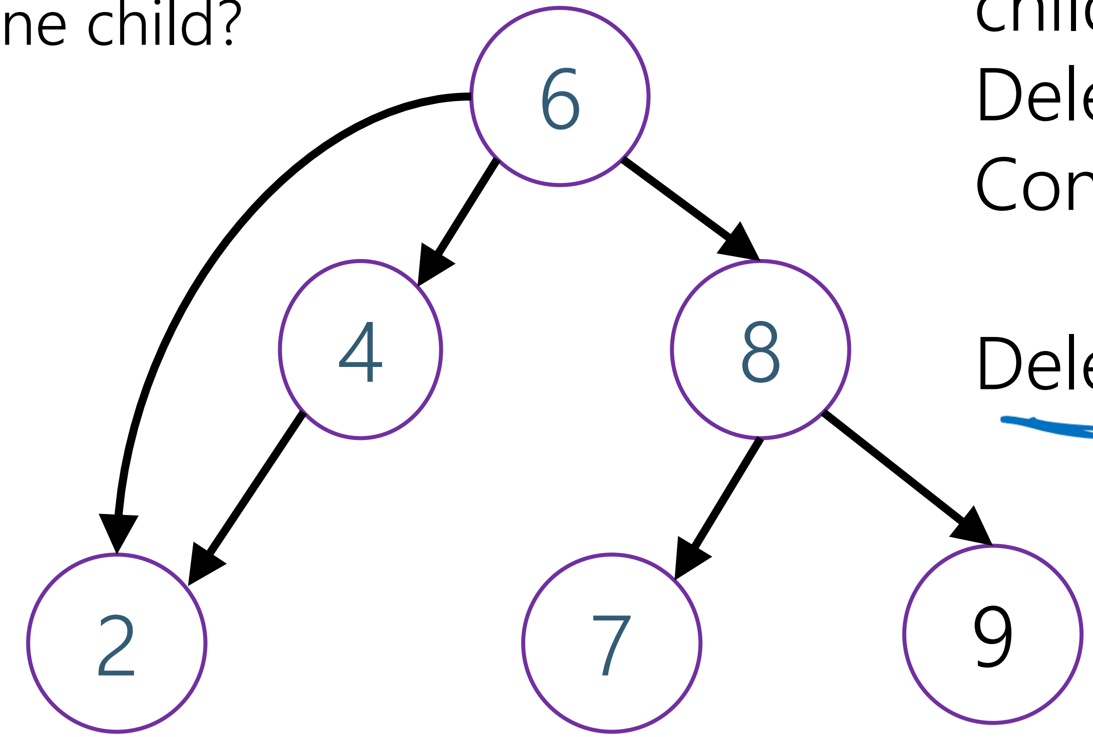
Deleting a leaf:
Just get rid of it.

Delete(7)

Deletion – Easy Cases

What if the elements to delete is:

- A leaf?
- Has exactly one child?



Deleting a node with one child:

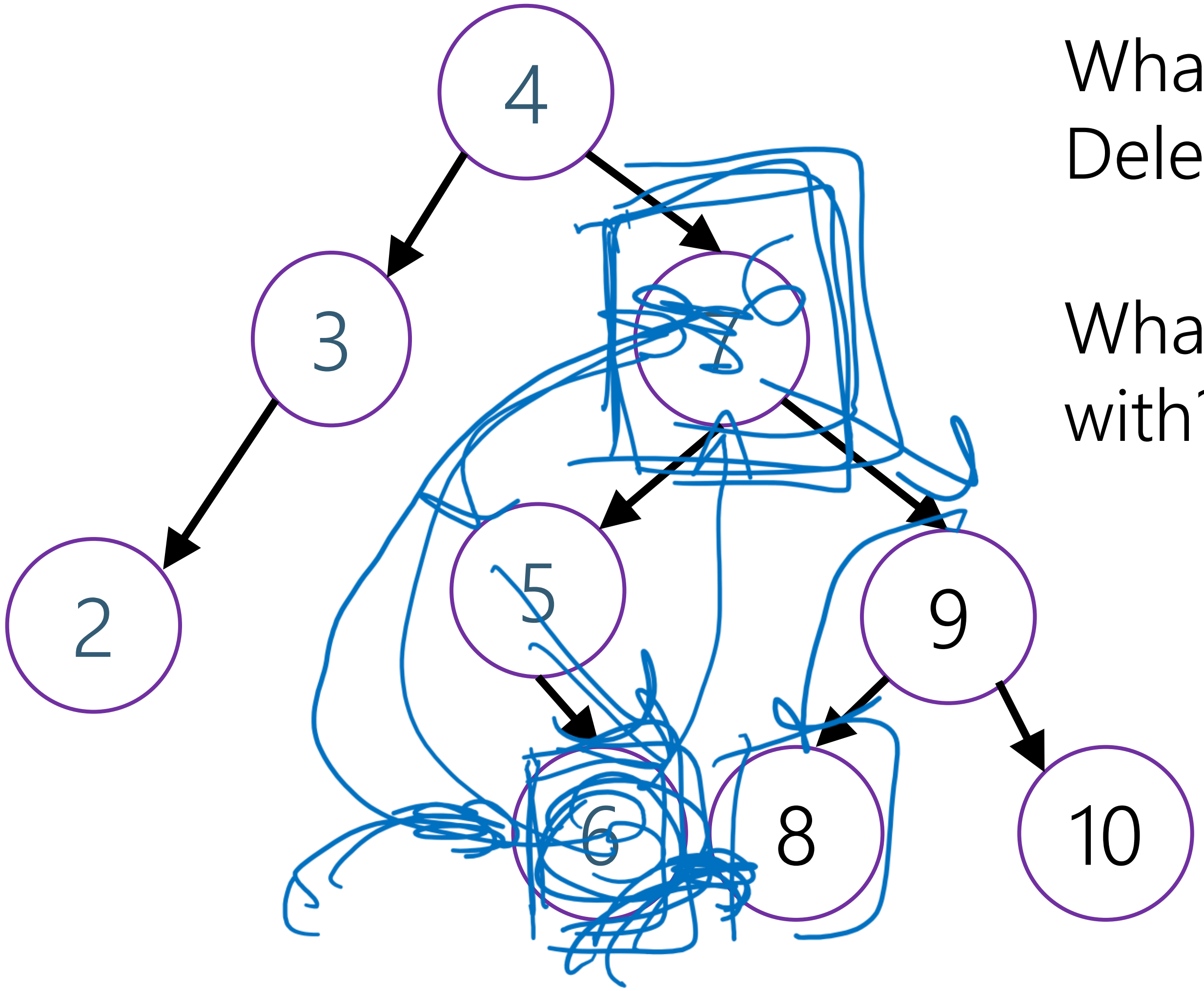
Delete the node

Connect its parent and child

Delete(4)

Deletion – The Hard Case

What happens if the node to delete has two children?



What if we try Delete(7)?

What can we replace it with?

6 or 8

The biggest thing in left subtree or smallest thing in right subtree.

Predecessor/Successor

The predecessor of x is the greatest node less than x.

The successor of x is the smallest node bigger than x.

How do we find?

If x has two children, these are in x's subtree by BST condition

The predecessor must be in x's left subtree (it's smaller), and it has to be as far right as possible in that subtree (since it's biggest).

The successor must be in x's right subtree (it's bigger), and it has to be as far left as possible in that subtree (since it's smallest).

You can find these easily!

Deletion Overall

Locate the node-to-be-deleted.

If it’s an easy case (no children, one child) handle it by rearranging pointers.

Otherwise, find the predecessor or successor

- The predecessor/successor always has at most one child
- If it’s the largest thing in the subtree, there can’t be anything to its right/smallest can’t have anything to its left. (symmetric for smallest in subtree)

Delete predecessor or successor (we’re in one of the easy cases!)

Place predecessor or successor data in location of node-to-be-deleted.

A Better Implementation

What about BSTs?

Keys will have to be comparable.

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

A Better Implementation

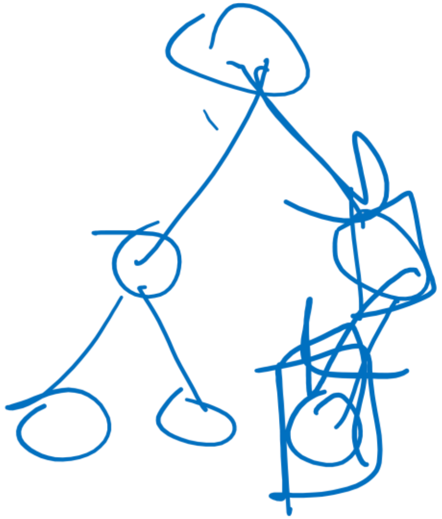
What about BSTs?

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 I:

Let's require the tree to be complete.

It worked for heaps!

What goes wrong?

- When we insert, we'll break the completeness property.

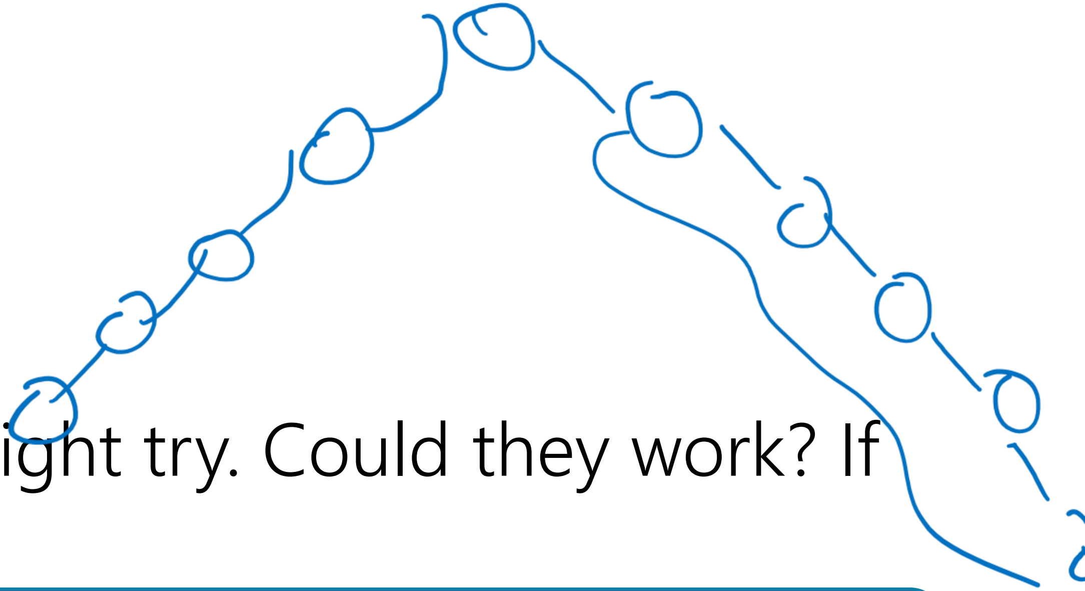
Insertions always add a new leaf, but you can't control where.

- Can we fix it?

Not easily :/

Avoiding the Worst Case

too weak



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 $O(\log n)$.

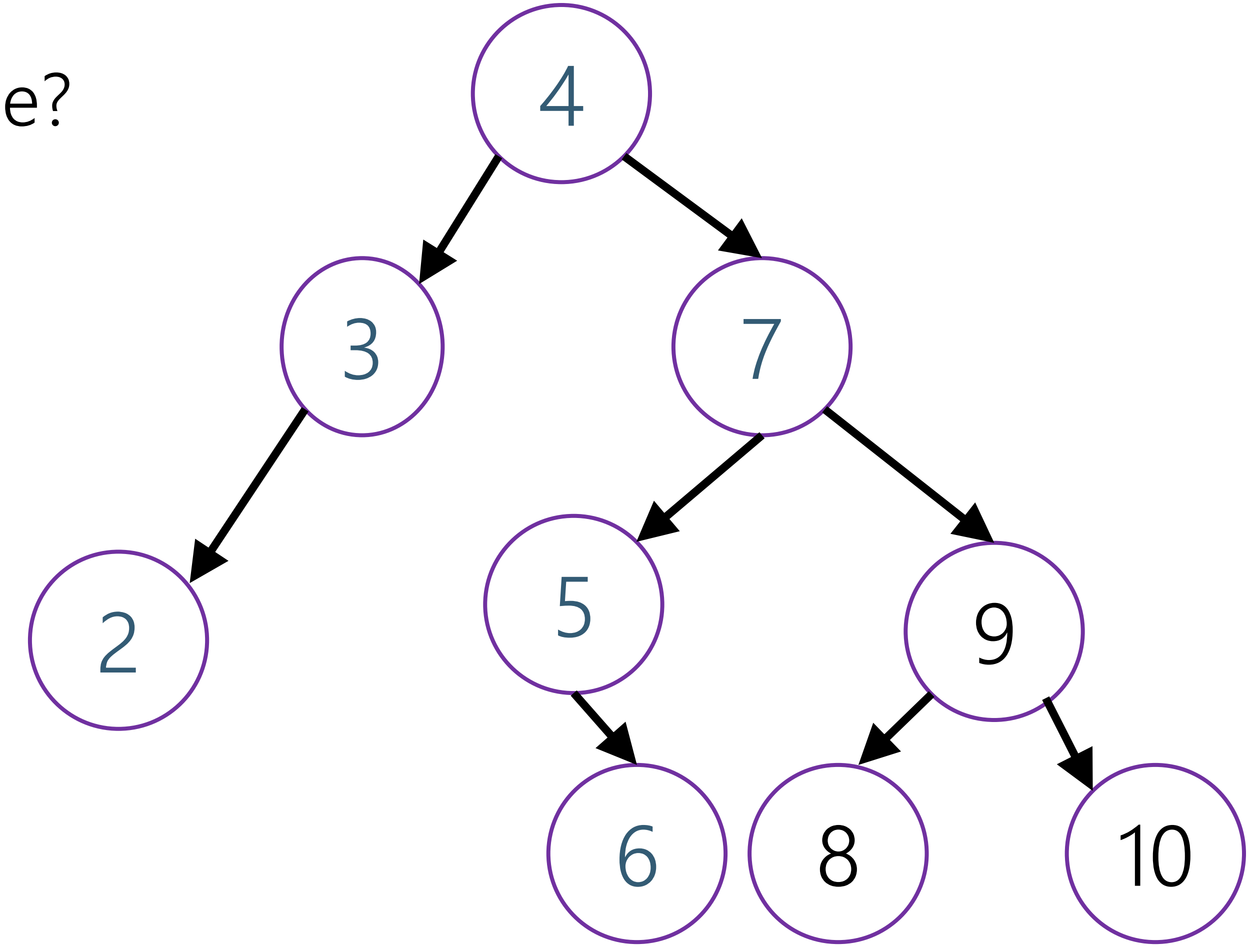
2. We must be able to maintain this property when inserting/deleting

Going to argue 2 first.

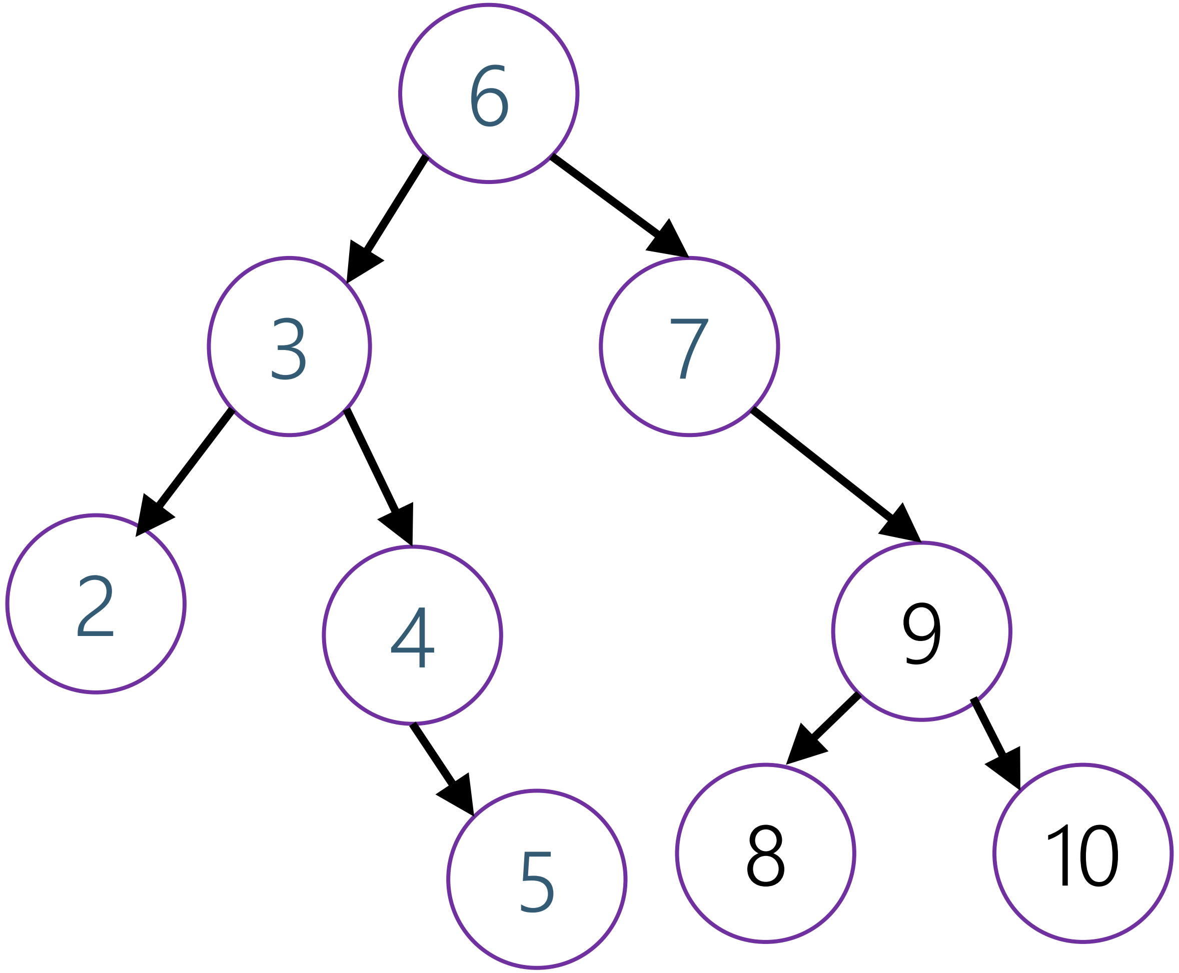
Warm-Up

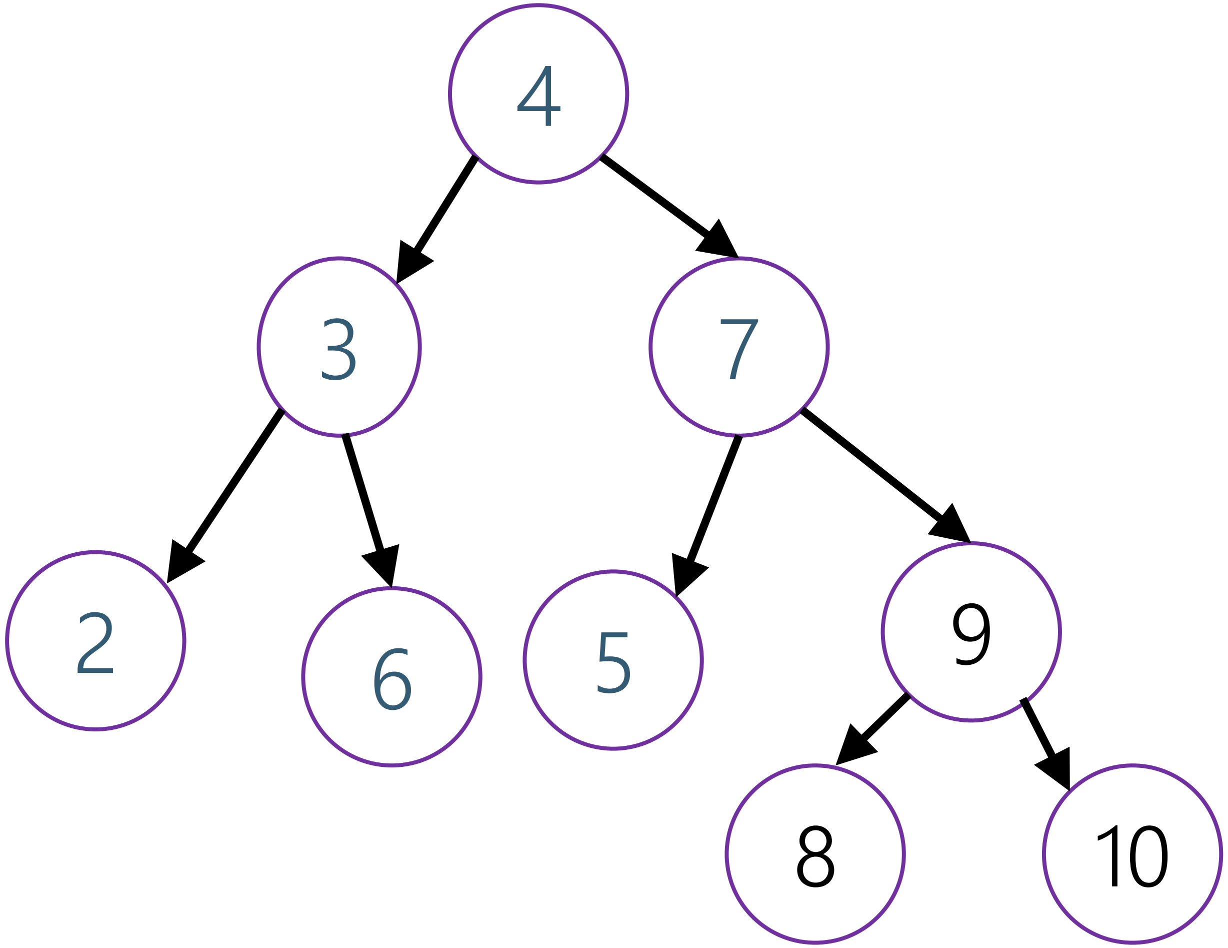
AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



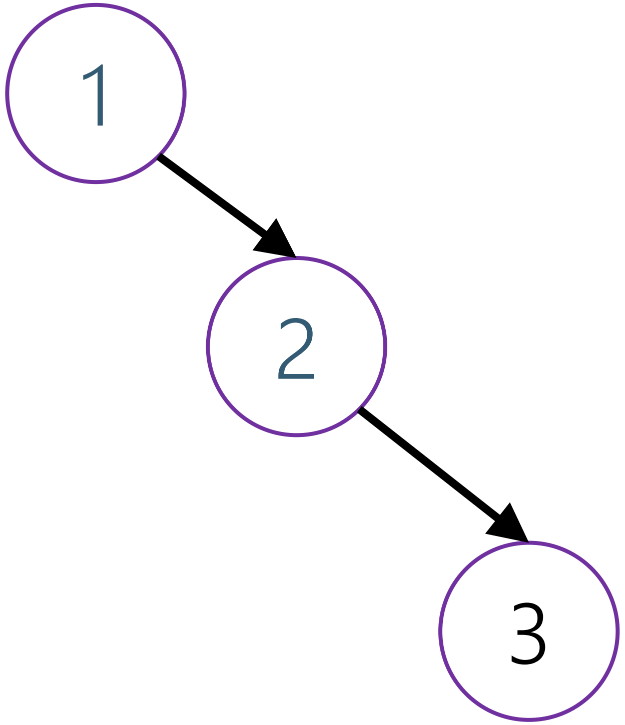
Are These AVL Trees?

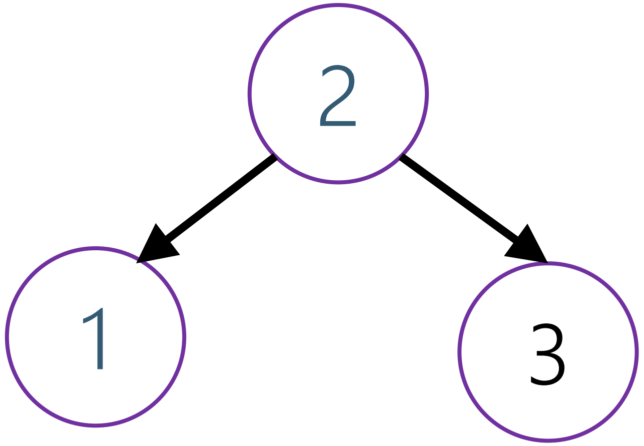




Insertion

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





Insertion

After you insert, check each node back up for balance

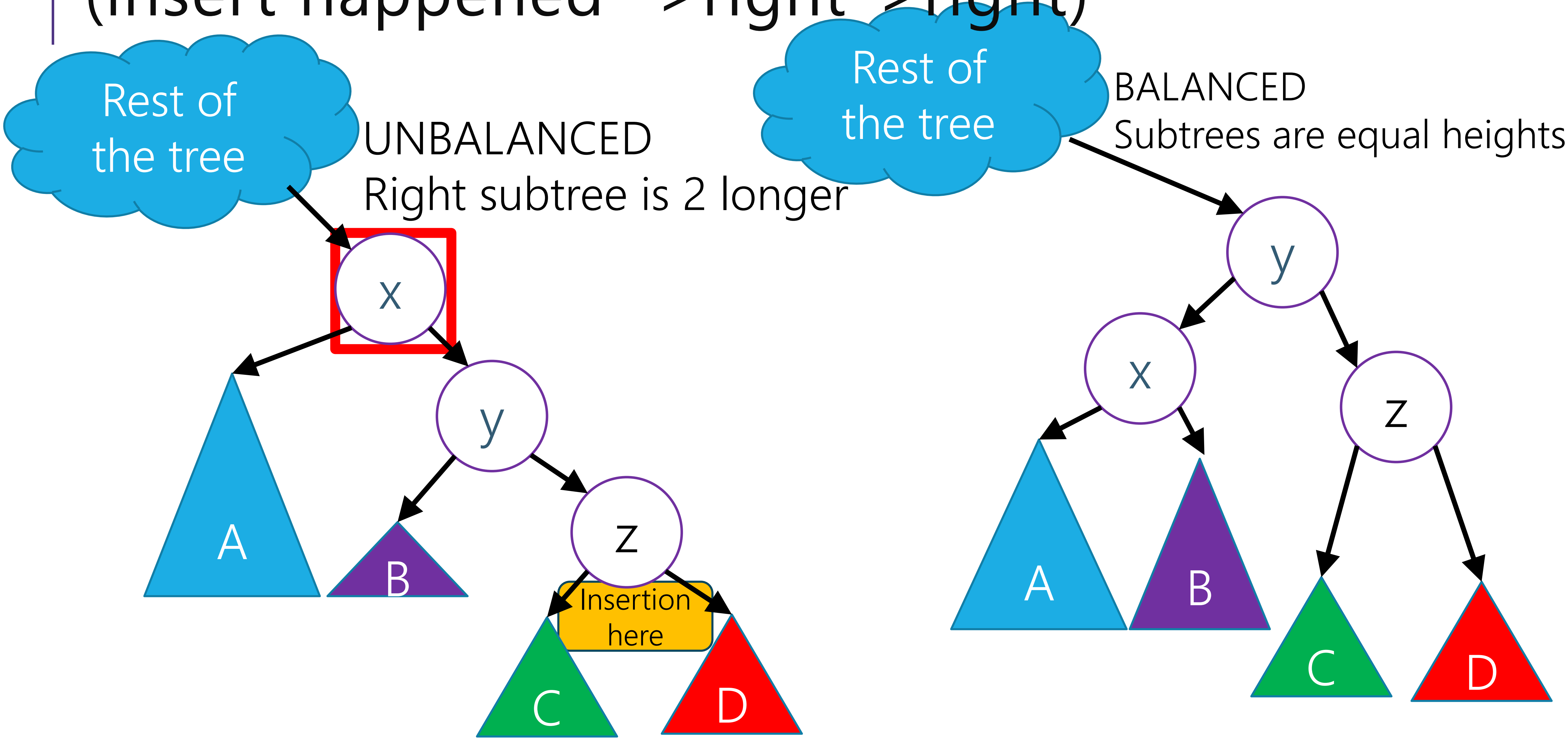
You rebalance at the lowest problem node

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



UNBALANCED
Right subtree is 2 longer

BALANCED
Subtrees are equal heights

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.

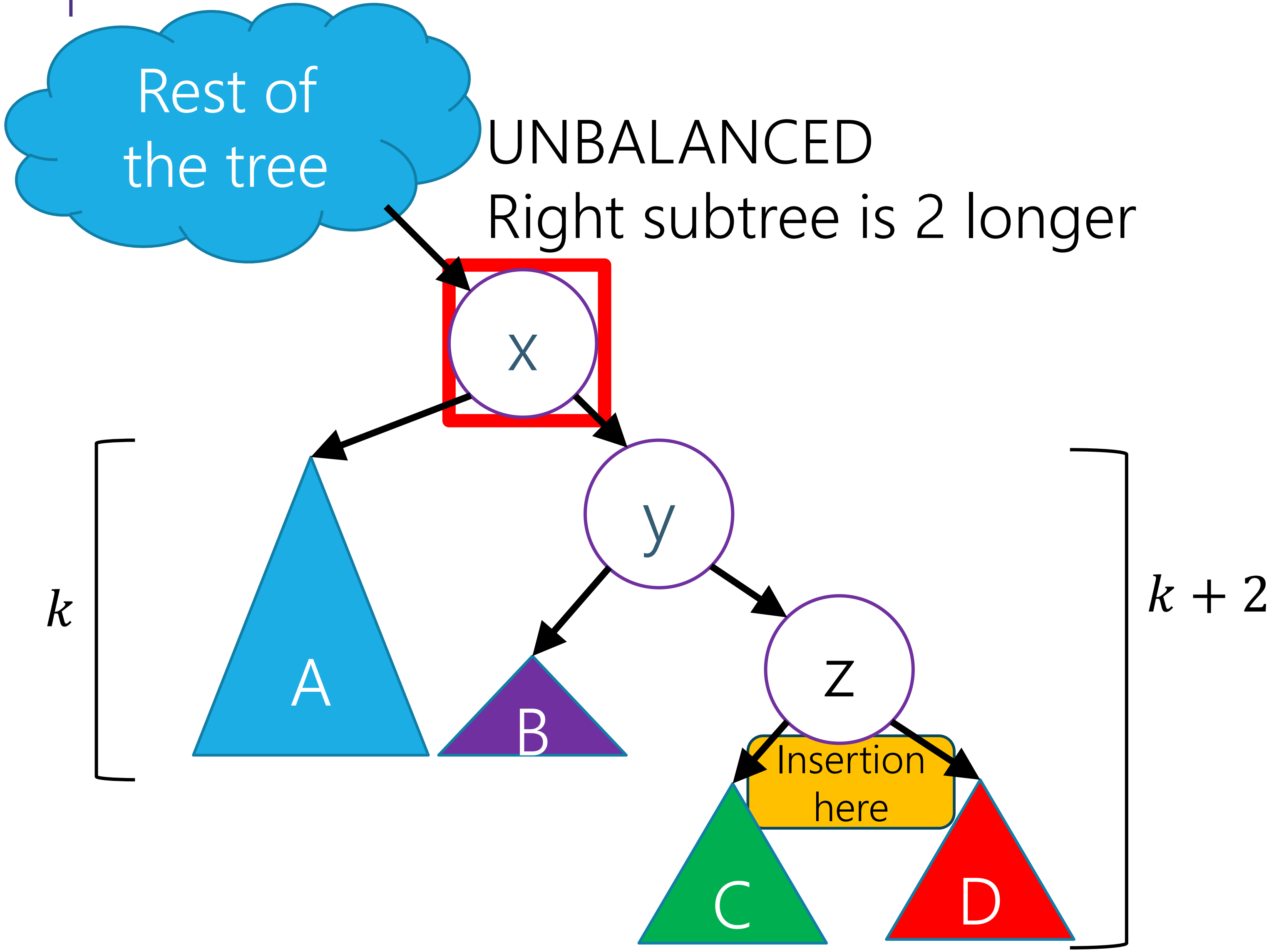
Rotation lengthened left subtree by 1, shrunk right subtree by 1.

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

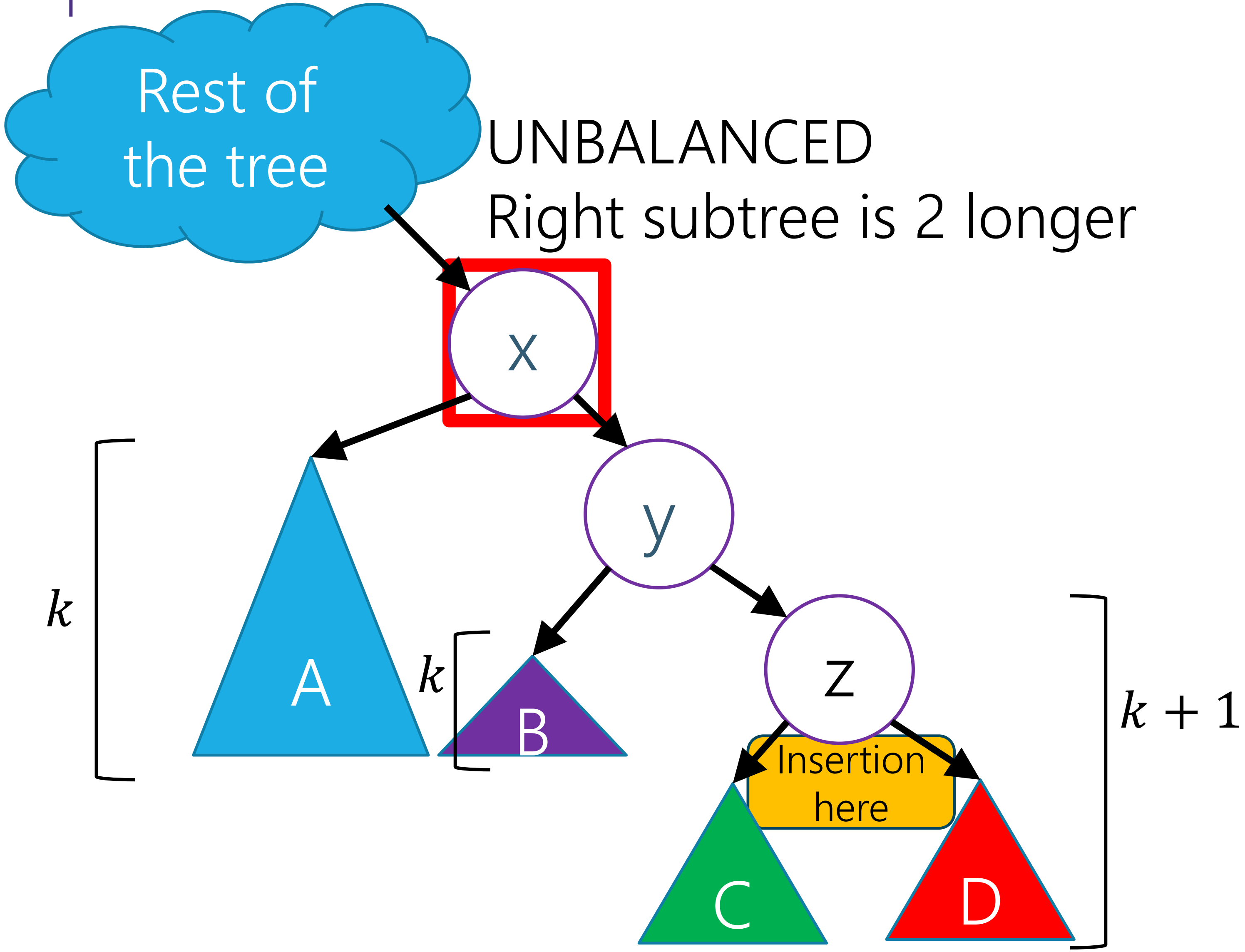
Left Single Rotation (insert happened ->right->right)



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.

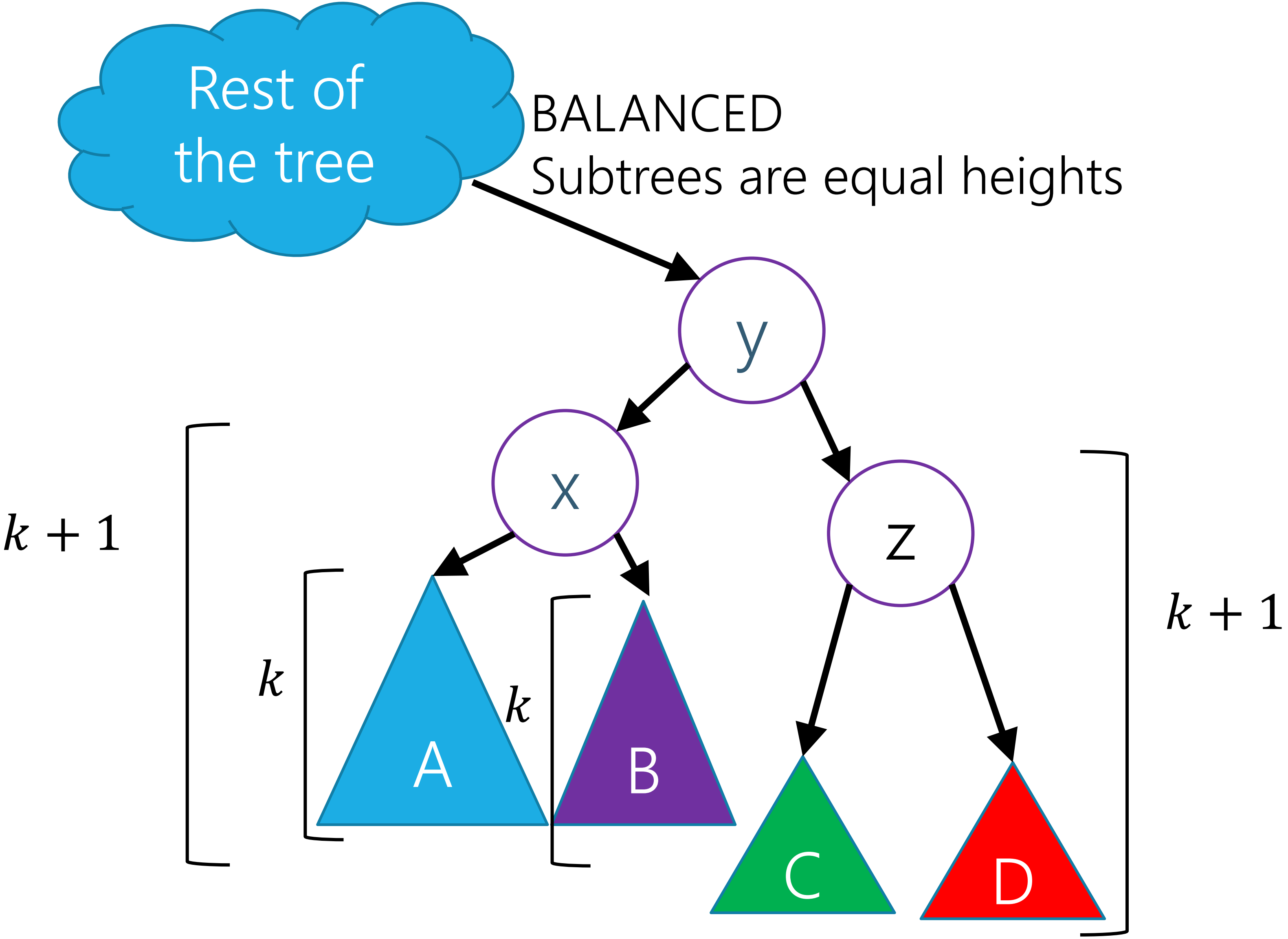
Left Single Rotation (insert happened ->right->right)



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)

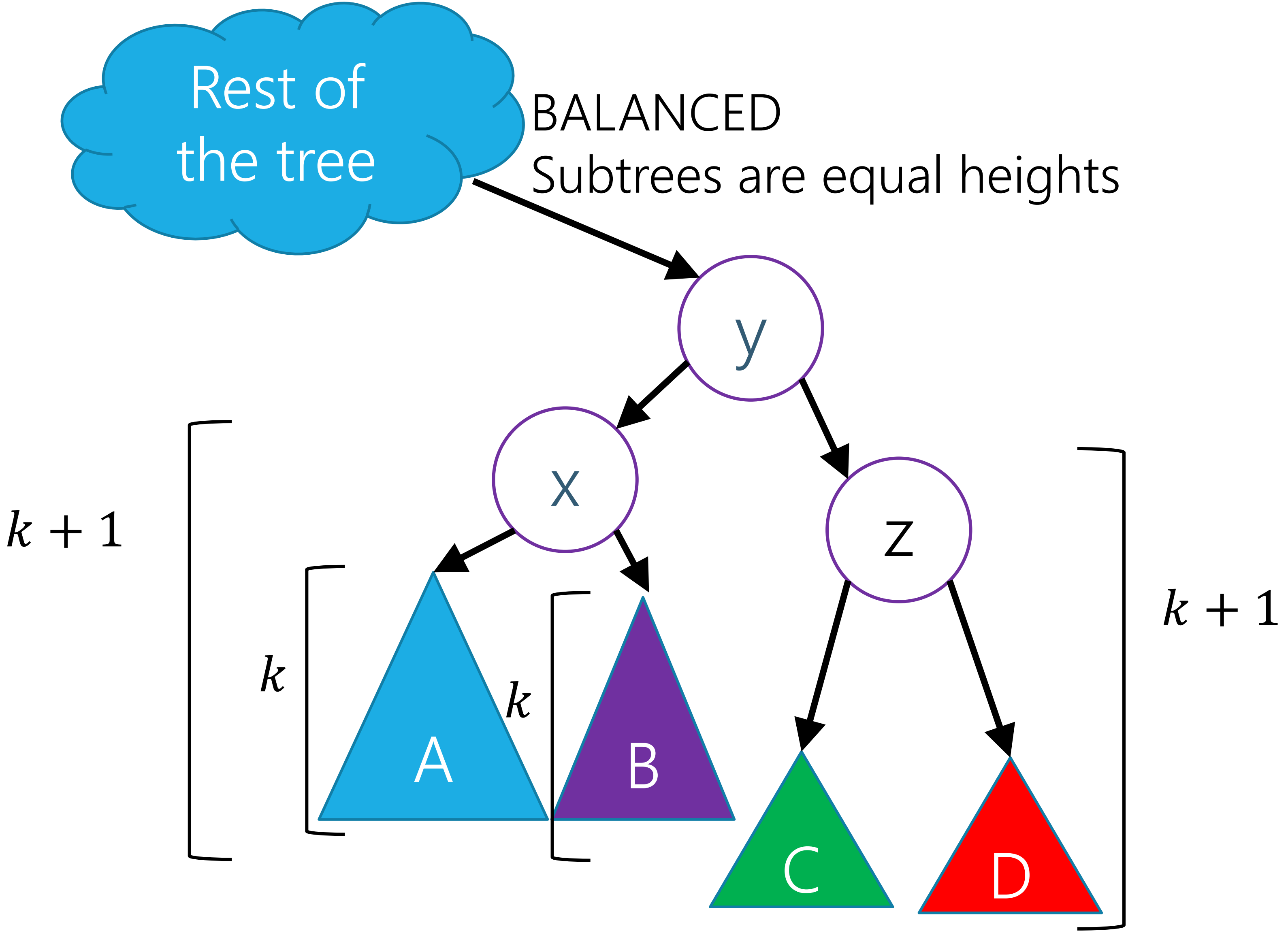
y is balanced (both subtrees height $k+1$).
x is balanced (both subtrees height k).
z is balanced (was not a problem node in recursion, and haven't rearranged its descendants).



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

We’ll sketch them next time.

Sometimes you need more than one rotation, but you never need more than two!

Efficient?

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 the tree method.

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 we do we'll quickly realize:

- Something with Fibonacci numbers is going on.
- It's really 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 $\frac{1+\sqrt{5}}{2} \approx 1.62$

Lazy Deletion

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.

Benefits:

- Much simpler to implement
- More efficient (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!