Lecture 9: Recurrences and AVL Trees

CSE 373: Data Structures and Algorithms

Administrivia

Exercise 1 due tonight

Exercise 2 out tonight

Project 1 part 2 out now

Since we pushed a day late, it's due a day later (Thursday July 18th)

Review session Tuesday at **1:10** in Sieg 134

Practice recurrences!

More Administrivia

Project 2 partner form will come out soon – start looking for a partner!

- You may keep your previous partner or switch.
- **Everyone** needs to fill out the partner form (even if you're keeping the same partner; both members of a partnership must fill out the form).

Some questions are better handled in office hours not on piazza.

If you're failing a test, and need help debugging – go to office hours, not piazza.

If you don't understand why your code is throwing an exception – office hours!

Piazza is great for conceptual questions; for a lot of code-questions we have to say "come see us in person."

The TAs have been lonely at office hours. Ask them for help in person!

Warm Up

Write a recurrence to describe the running time of the following method:

When describing non-recursive work, simplify out any constant factors and lower-order terms.

```
public void nonsense(ArrayList<Integer> arr, int ind){
      if(ind == -1) return;
      for(int j = 0; j < ind; j++){
            for(int k = j+1; k < ind; k++){
                  System.out.println(arr[j] + " " + arr[k]);
            }
      }
      nonsense(toSort, ind-1);
}
```

pollEV.com/cse373su19

Write a recurrence for the running time of `nonsense`

$$\sum_{j=0}^{n-1} \sum_{k=j+1}^{n-1} 1$$

$$\sum_{i=a}^{b} f(x) = \sum_{i=0}^{b} f(x) - \sum_{i=0}^{a-1} f(x)$$

$$\sum_{i=0}^{n-1} c = \underbrace{c + c + \cdots + c}_{n \text{ times}} = cn$$

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

$$\sum_{j=0}^{n-1} \sum_{k=j+1}^{n-1} 1 = \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} 1 - \sum_{k=0}^{j} 1 \right)$$

$$= \sum_{j=0}^{n-1} n - (j+1)$$

$$\sum_{j=0}^{n-1} n - \sum_{j=0}^{n-1} j + \sum_{j=0}^{n-1} 1$$

$$= n^2 - \frac{n(n-1)}{2} + n$$

$$T(n) = \begin{cases} T(n-1) + n^2 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Even though we had lower-order terms initially, this is the recurrence we would try to solve.

We’ll tell you to simplify if we want you to.

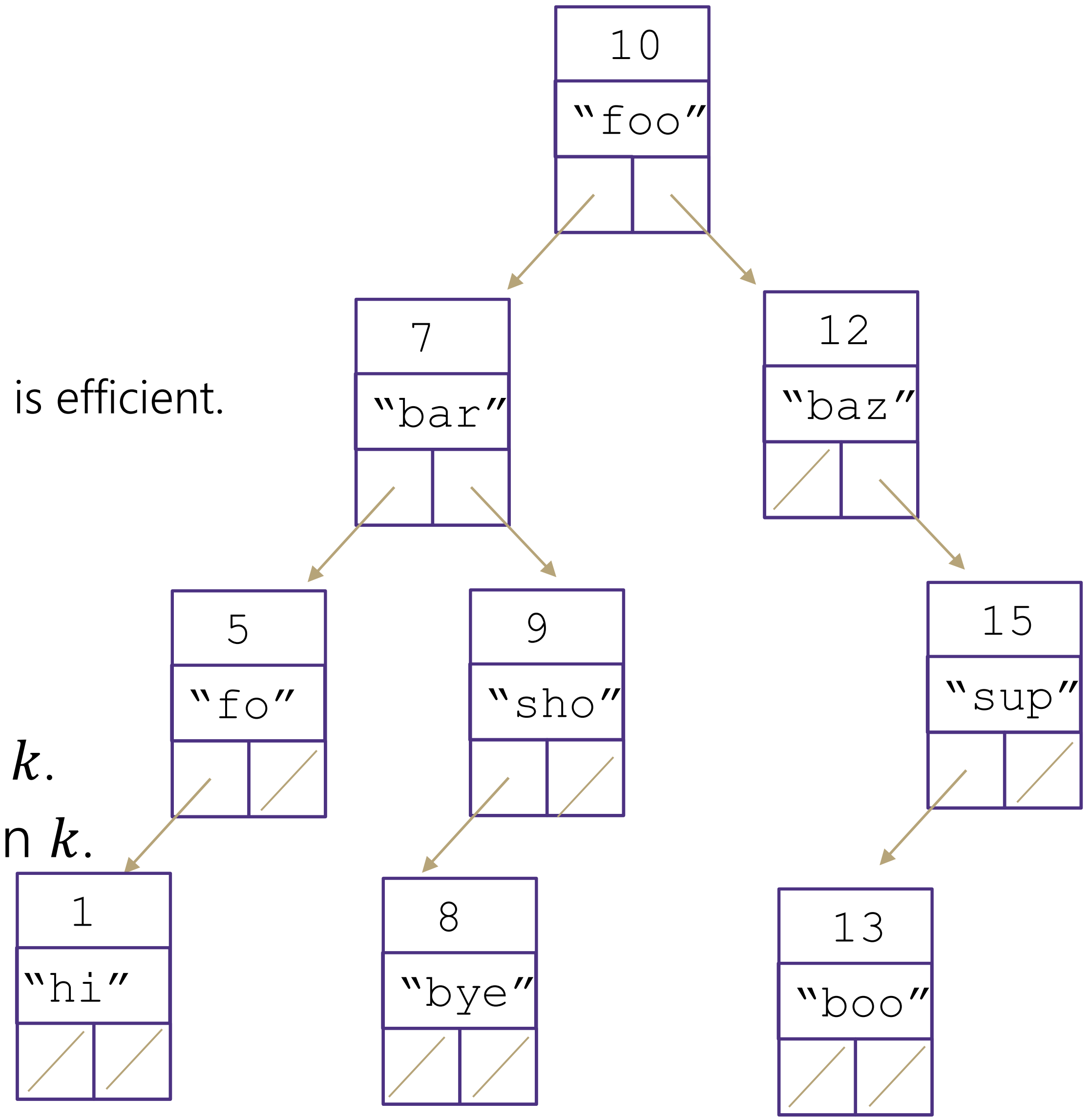
Binary Search Tree

Invariants

- Things that are always true.
- The way you make sure your data structure works and is efficient.

Binary Search Tree invariants:

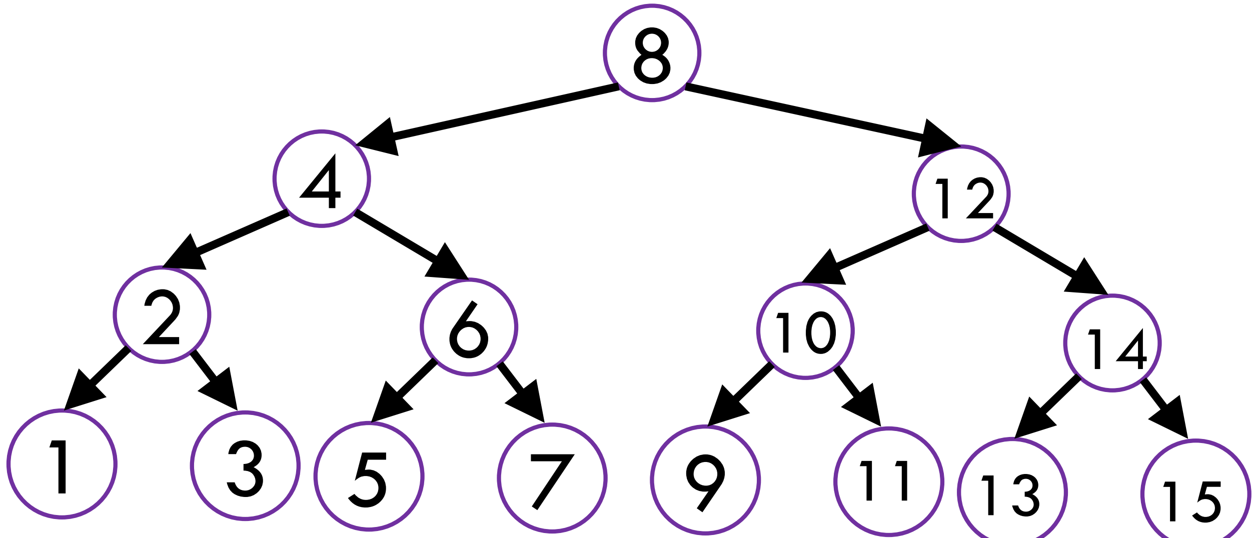
- For every node with key k:
  - The left subtree has only keys smaller than k.
  - The right subtree has only keys greater than k.



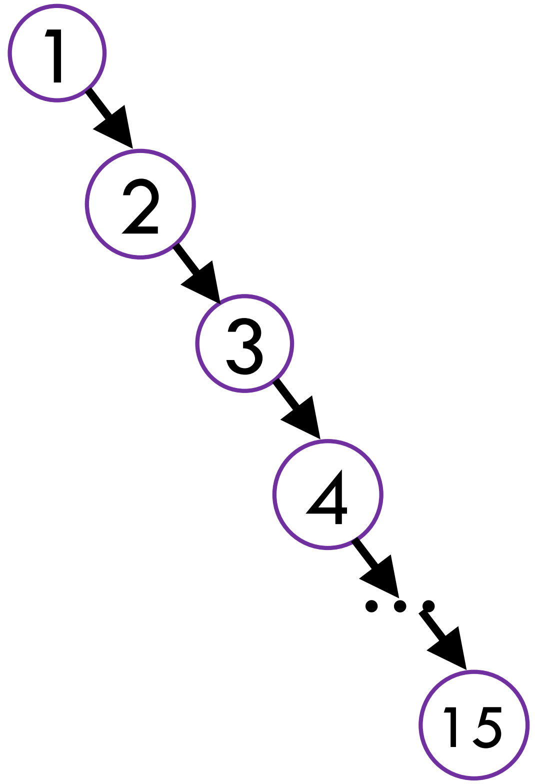
BSTs as dictionaries

Let's figure out the worst case of `get()` for two different states our BST could be in.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



Degenerate – for every node, all of its descendants are in the right subtree.

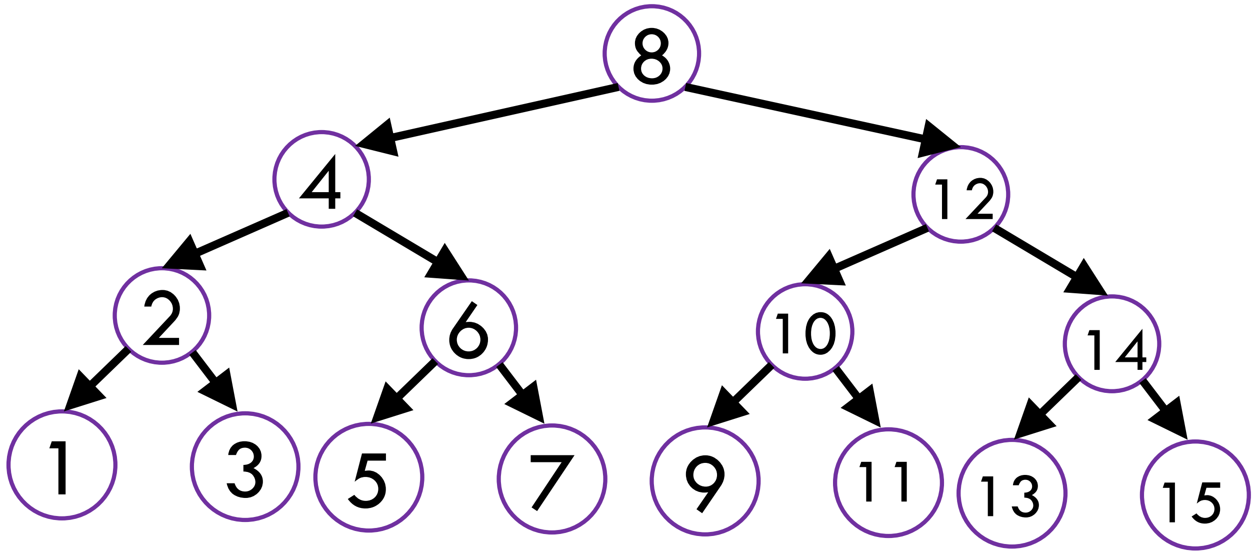


BSTs as dictionaries

Let's figure out the worst case of `get()` for two different states our BST could be in.

`get()` is a recursive method!

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 \text{ if } n > 1 \\ 3 \quad\quad\quad \text{otherwise} \end{cases}$$

$$\mathrm{T(n)} = \Theta(\log n)$$

BSTs as dictionaries

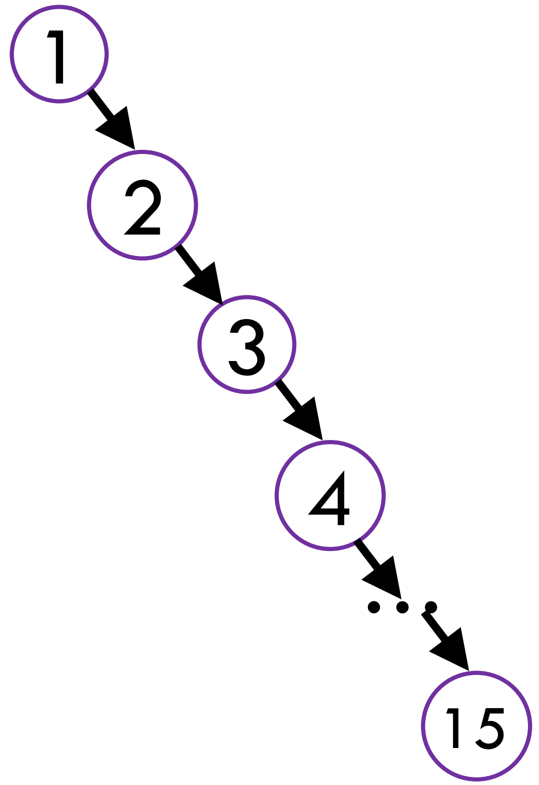
Let's figure out the worst case of `get()` for two different states our BST could be in.

`get()` is a recursive method!

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

$$\mathrm{T(n)} = \Theta(n)$$

Degenerate – for every node, all of its descendants are in the right subtree.



Invariants

Observation: What was important was actually the height of the tree.

- **Height:** number of edges on the longest path from the root to a leaf.

That’s the number of recursive calls we’re going to make

- And each recursive call does a constant number of operations.

The BST invariant makes it easy to know where to find a `key`

But it doesn’t force the tree to be short.

Let’s add an invariant that forces the height to be short!

Invariants

Why not just make the invariant "keep the height of the tree at most $O(\log n)$" ?

The invariant needs to be easy to maintain.

Every method we write needs to ensure it doesn't break it.
Can we keep that invariant true without making a bunch of other methods slow?

It's not obvious…

Writing invariants is more art than science.

- Learning that art is beyond the scope of the course
- but we'll talk a bit about how you might have come up with a good invariant (so our ideas are motivated).

When writing invariants, we usually start by asking "can we maintain this" then ask "is it strong enough to make our code as efficient as we want?"

Avoiding $\Theta(n)$ behavior

Here are some invariants you might try.
Can you maintain them? If not what can go wrong?

Do you think they are strong enough to make `get` efficient?

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.

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.

Invariant Lessons

Need requirements everywhere, not just at root

Forcing things to be exactly equal is too difficult to maintain.

Avoiding the Degenerate Tree

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 $\Theta(n)$ behavior! We have to check:

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

Bounding the Height

Let $m(h)$ be the minimum number of nodes in an AVL tree of height h.

If we can say $m(h)$ is big, we'll be able to say that a tree with n nodes has a small height.

So…what's $m(h)$?

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

A simpler recurrence

Hey! That's a recurrence!

Recurrences can describe any kind of function, not just running time of code!

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

We could use tree method, but it's a little…weird.

It'll be easier if we change things just a bit:

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

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

Answer the following questions:

1. What is the size of the input on level i?
2. What is the work done by each node on the i^{th} recursive level
3. What is the number of nodes at level i?
4. What is the total work done at the i^th recursive level?
5. What value of i does the last level occur?
6. What is the total work across the base case level?

5 Minutes

Tree Method Practice

1. What is the size of the input on level i? $n - 2i$

2. What is the work done by each node on the i^{th} recursive level? 1

3. What is the number of nodes at level i? 2^i

4. What is the total work done at the i^{th} recursive level? $2^i \cdot 1$

5. What value of i does the last level occur?

   $n - 2i = 0 \rightarrow i = \frac{n}{2}$

6. What is the total work across the base case level? $2^{n/2}$

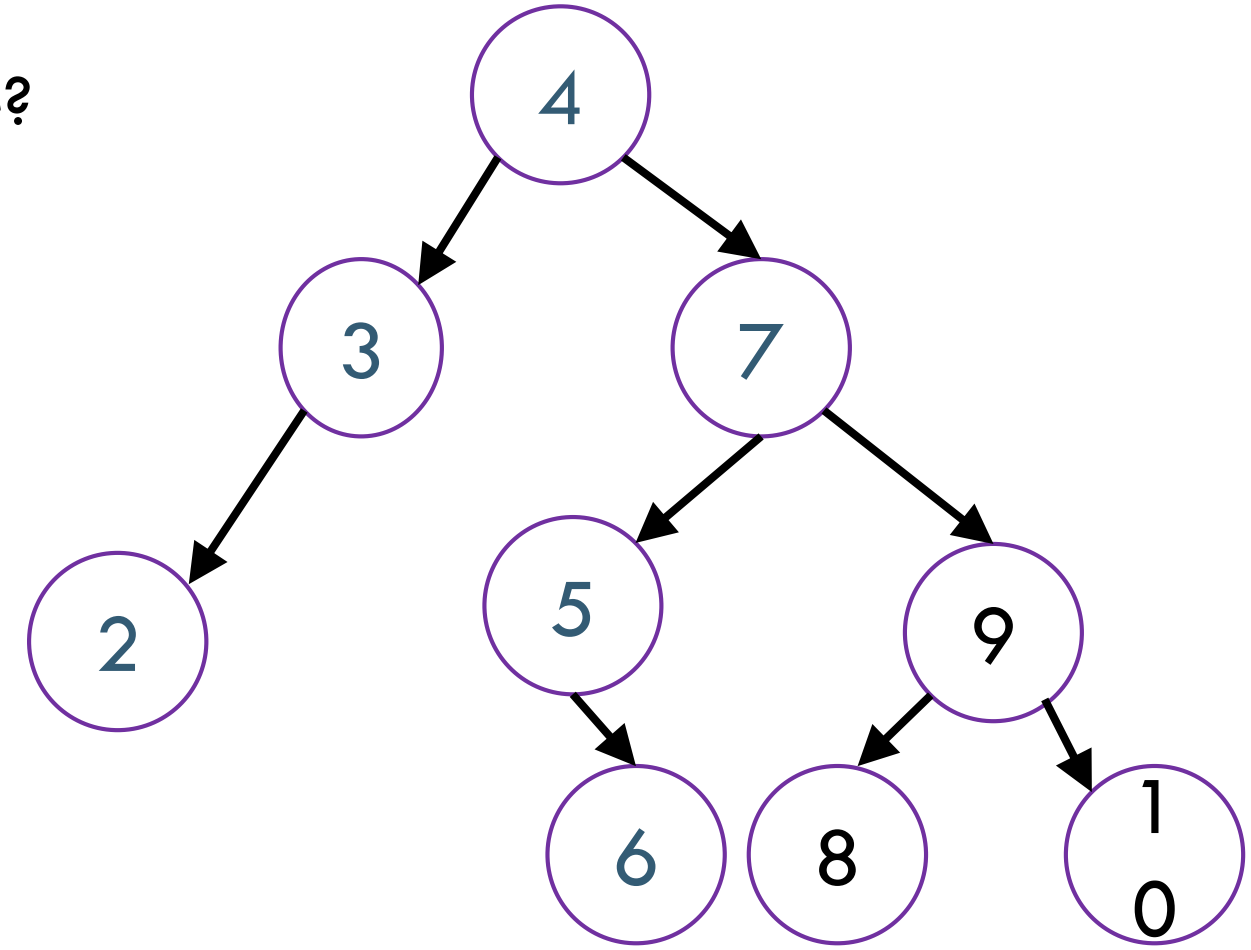
$$m(n) \geq \sum_{i=0}^{\frac{n}{2}-1} 2^i + 2^{n/2} = 2^{n/2} - 1 + 2^{n/2}$$

$$m(n) \geq 2^{n/2}$$

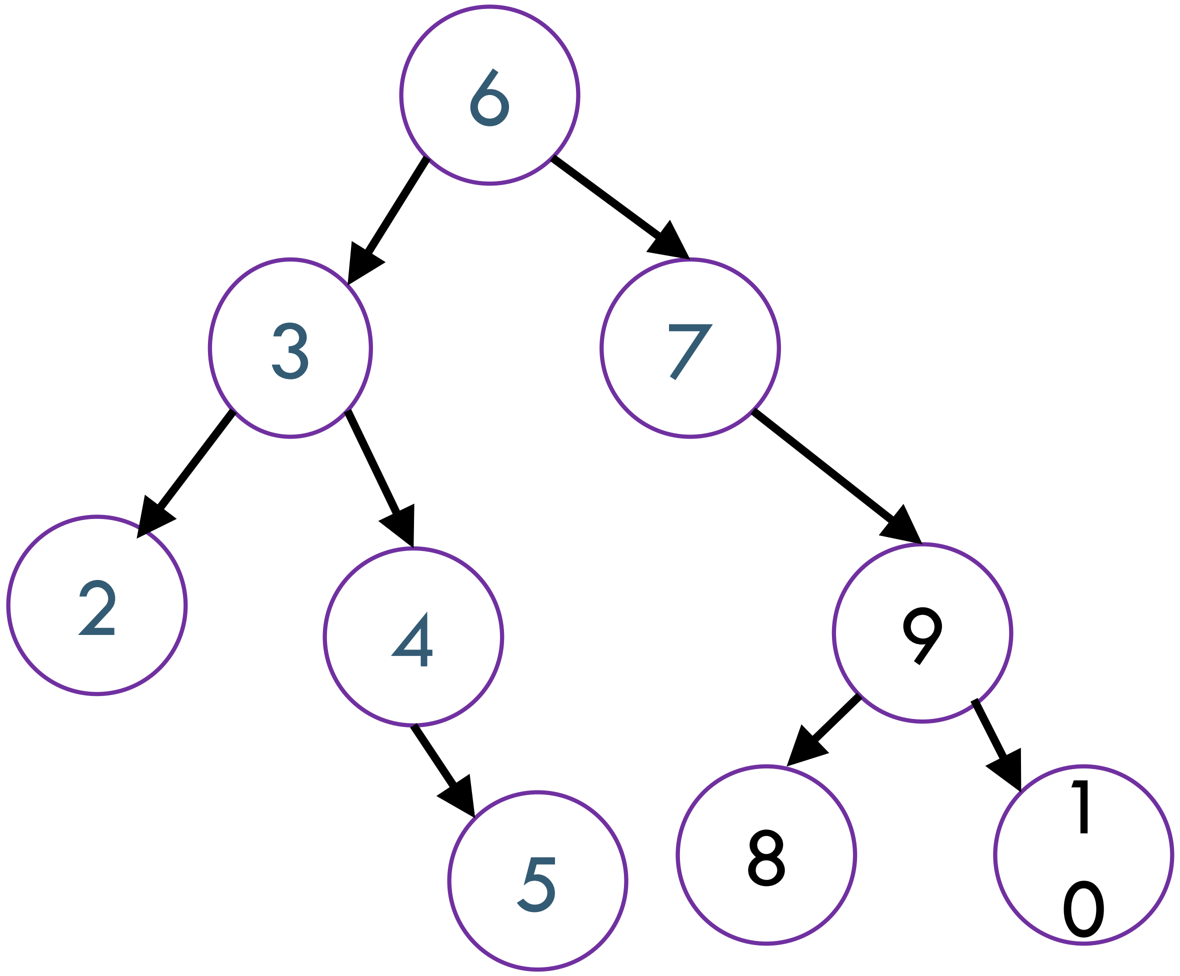
Warm-Up

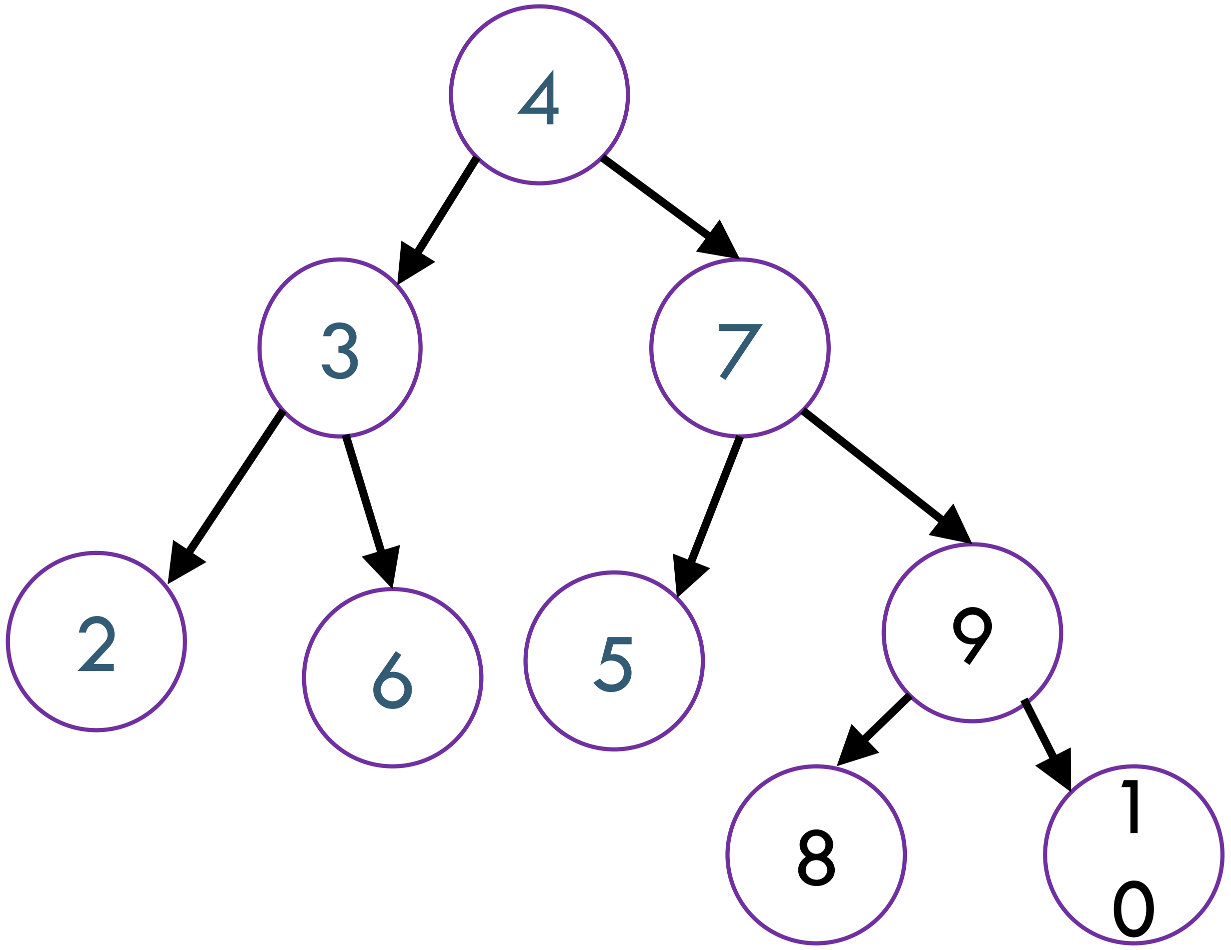
AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



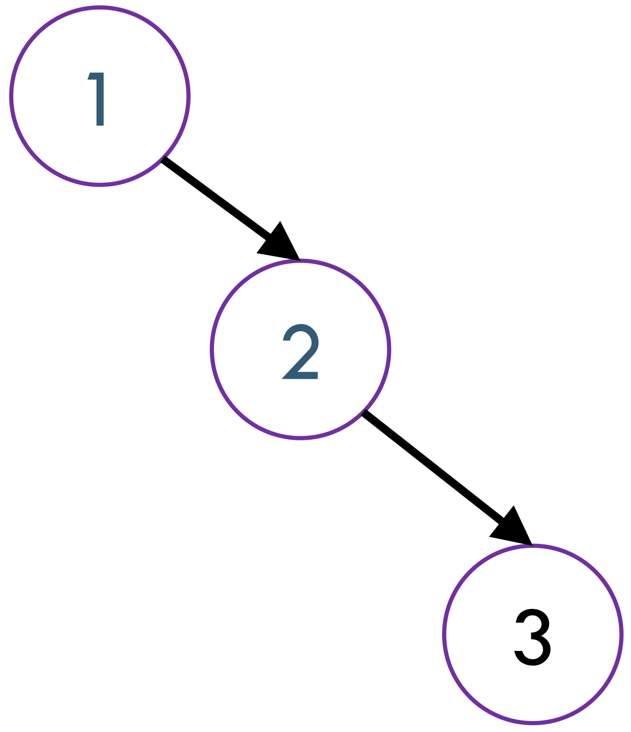
Are These AVL Trees?

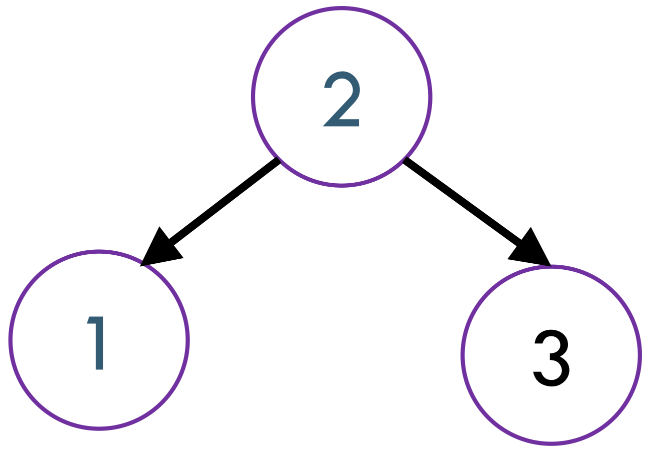




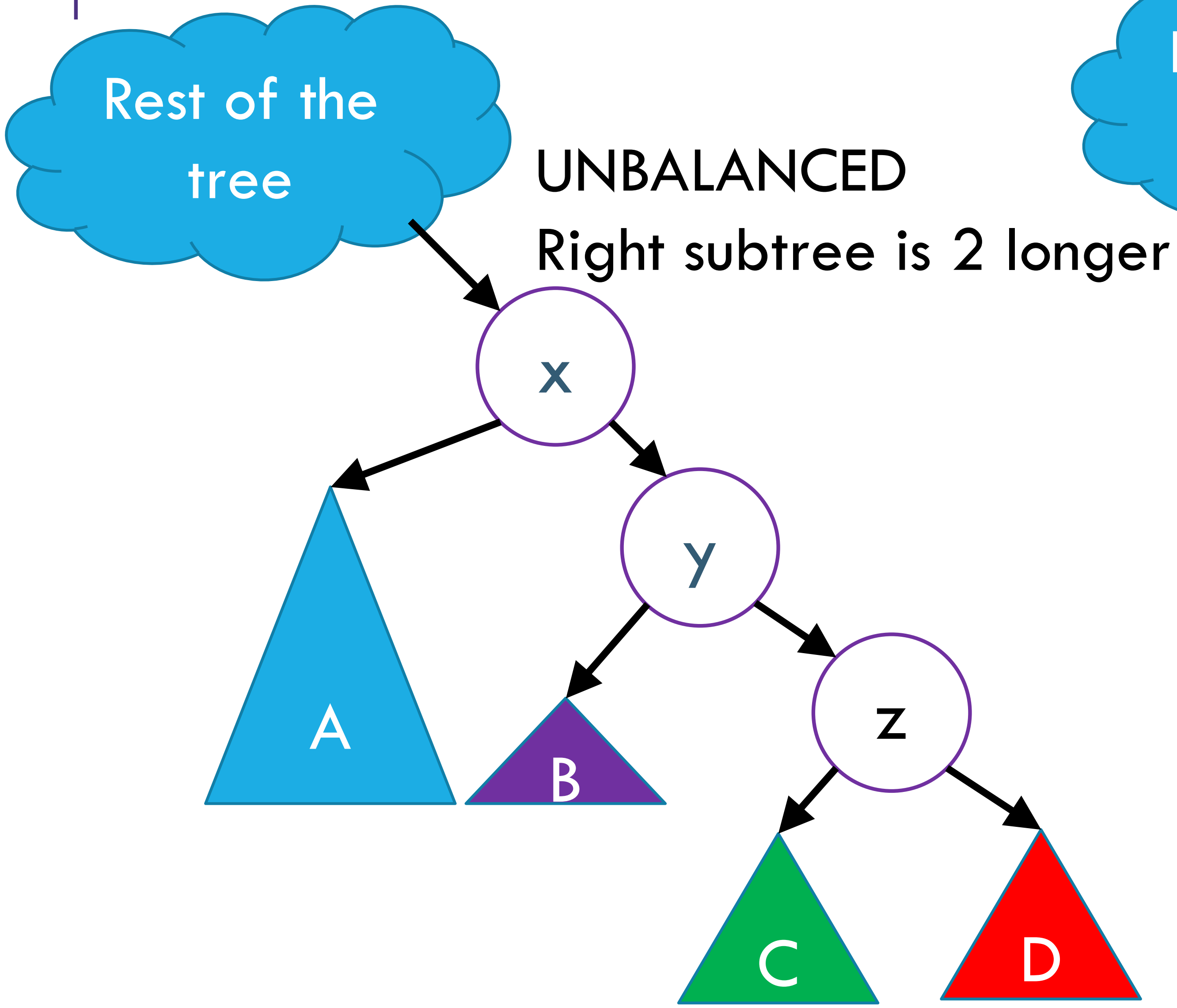
Insertion

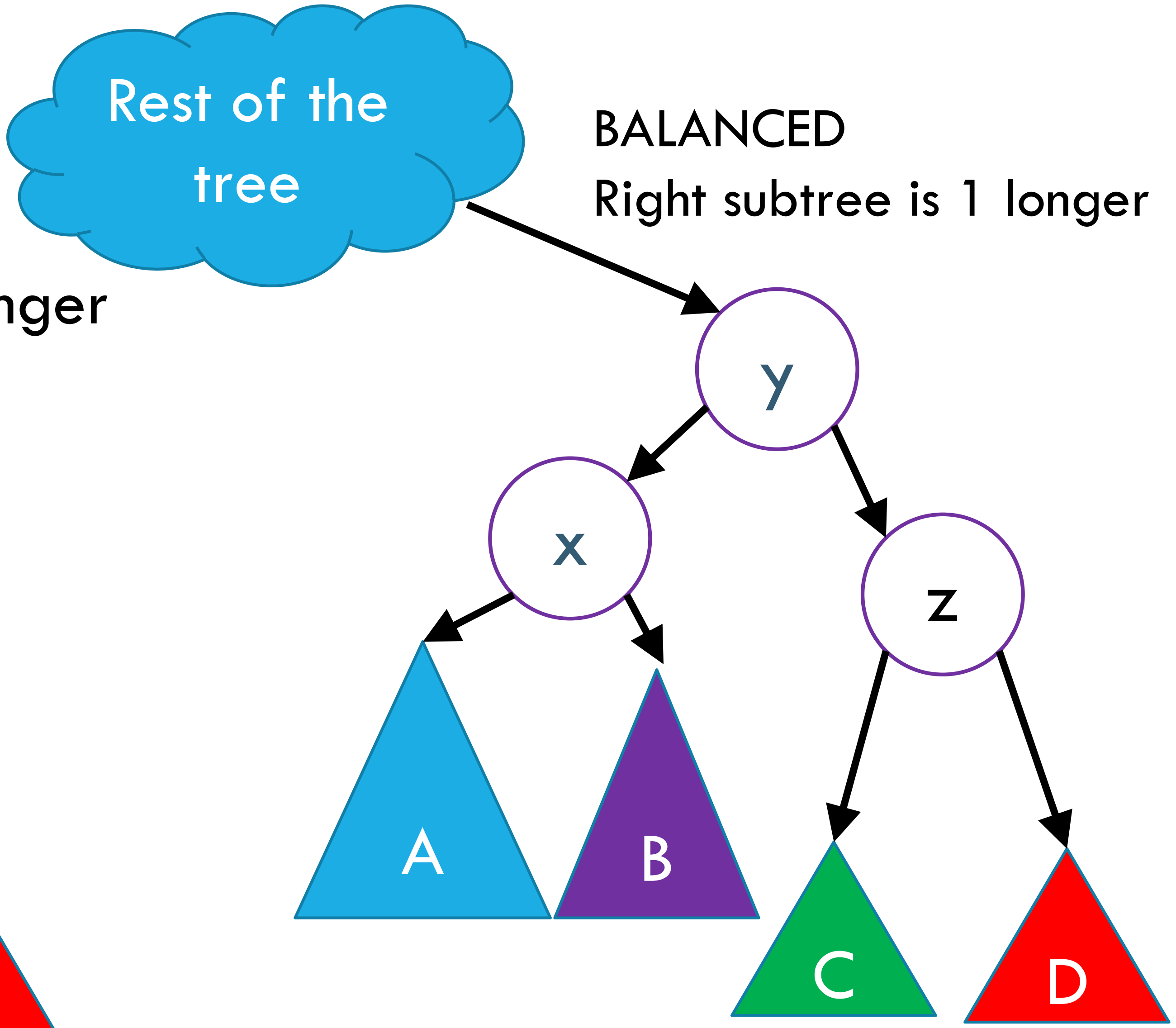
What happens if when we do an insertion, we break the AVL condition?





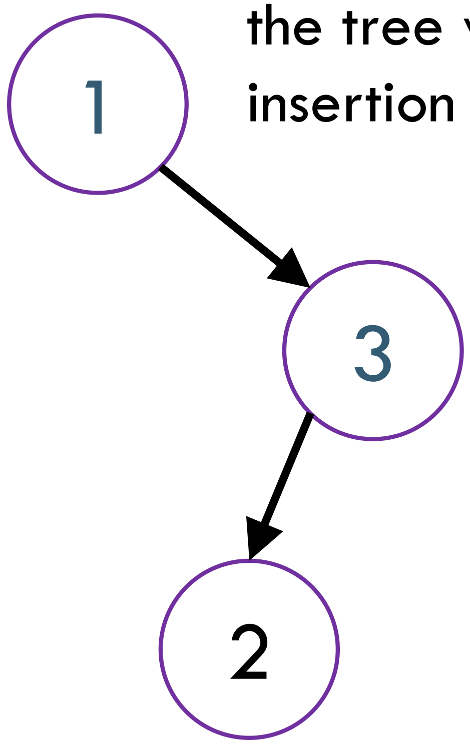
Left Rotation



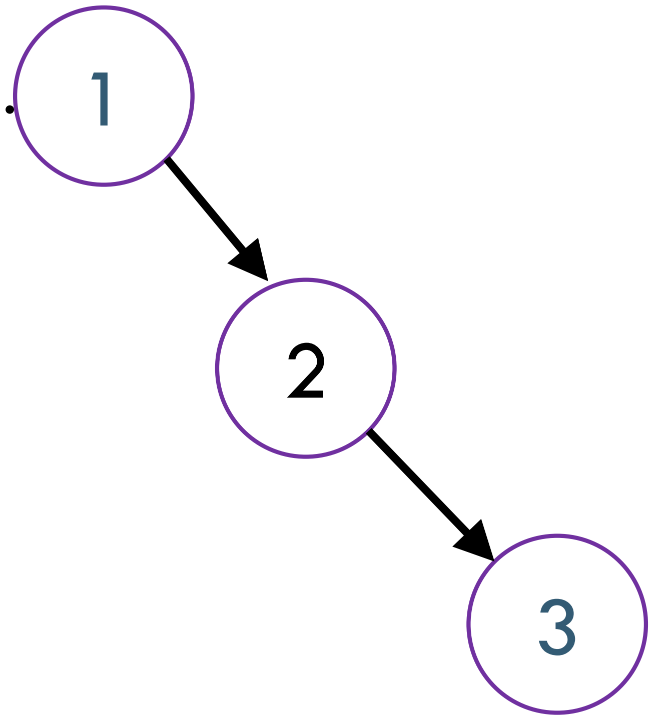


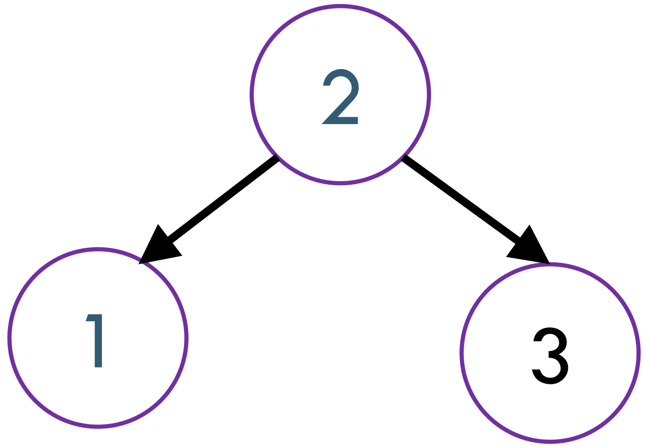
It Gets More Complicated

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



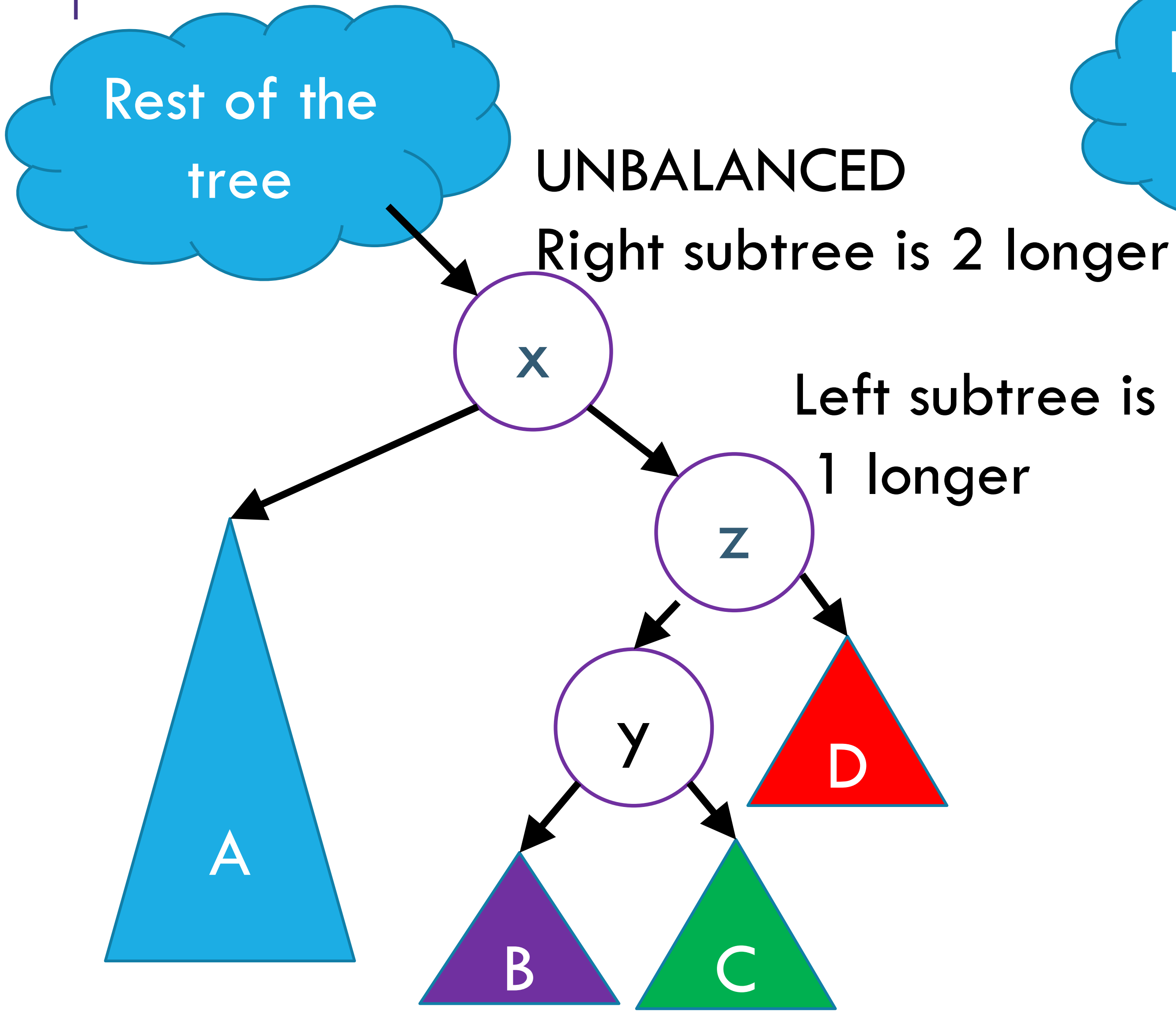
Can't do a left rotation
Do a "right" rotation around 3 first.

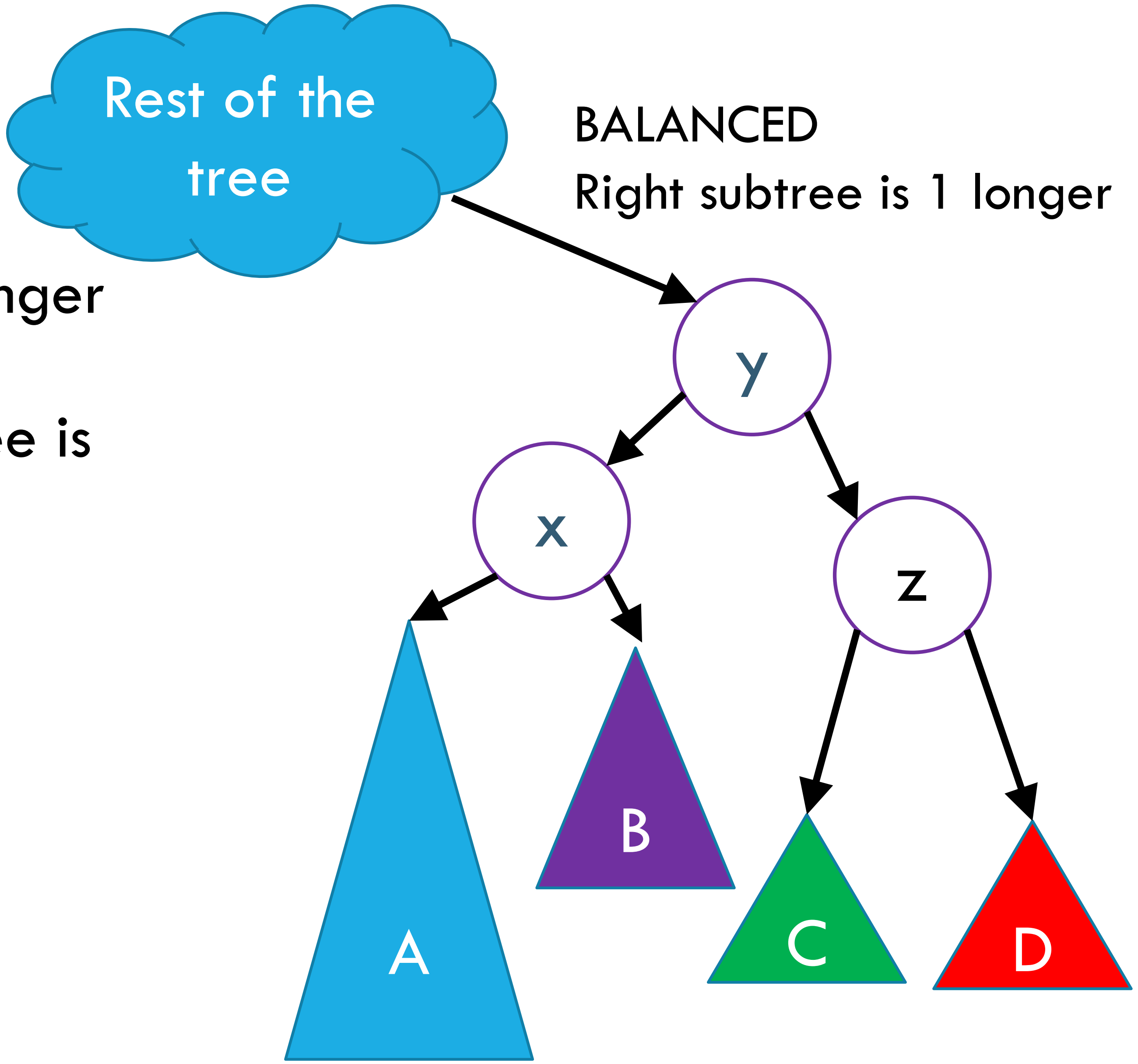




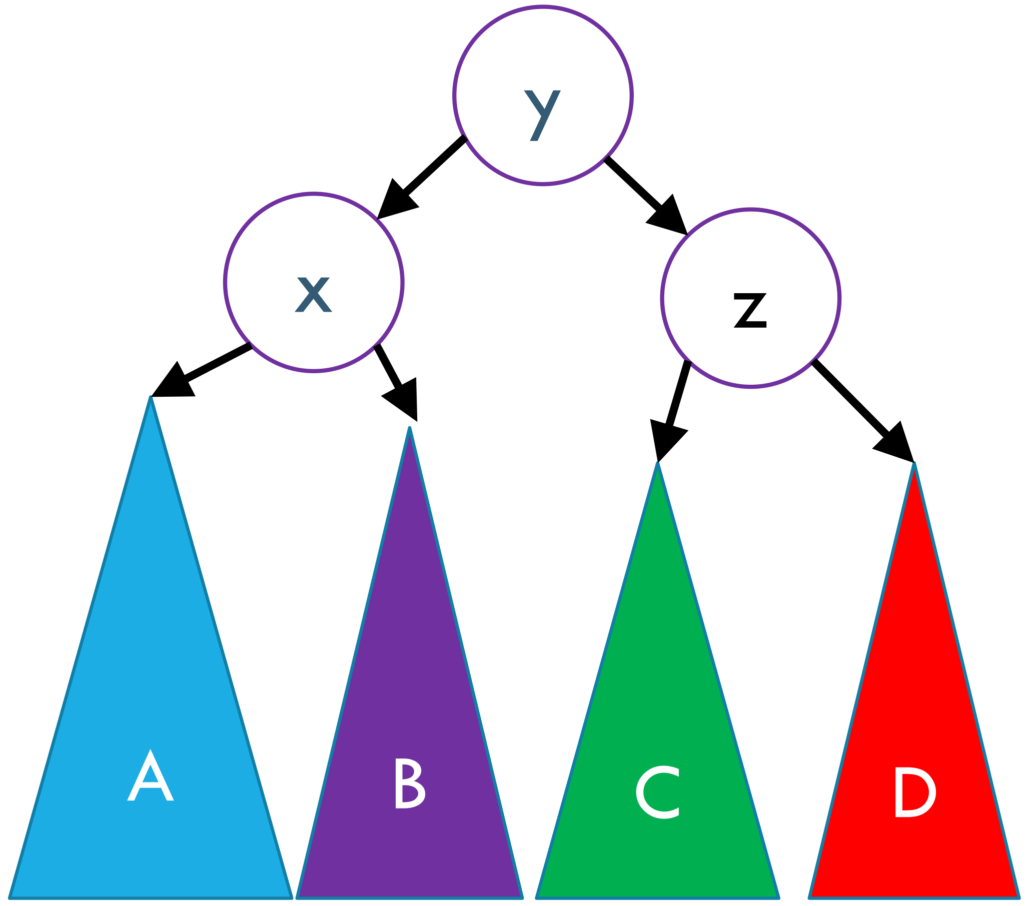
Now do a left rotation.

Right Left Rotation



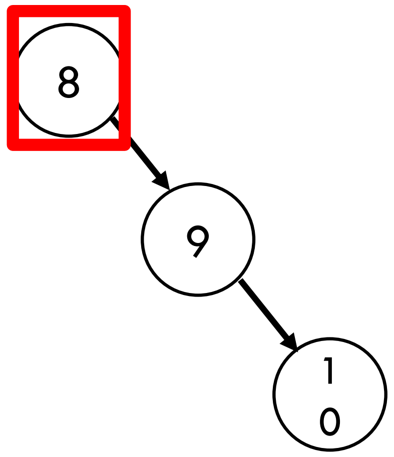


Four Types of Rotations

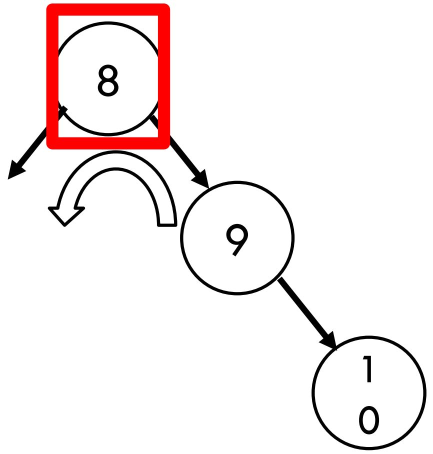


Insert location	Solution
Left subtree of left child (A)	Single right rotation
Right subtree of left child (B)	Double (left-right) rotation
Left subtree of right child (C)	Double (right-left) rotation
Right subtree of right child(D)	Single left rotation

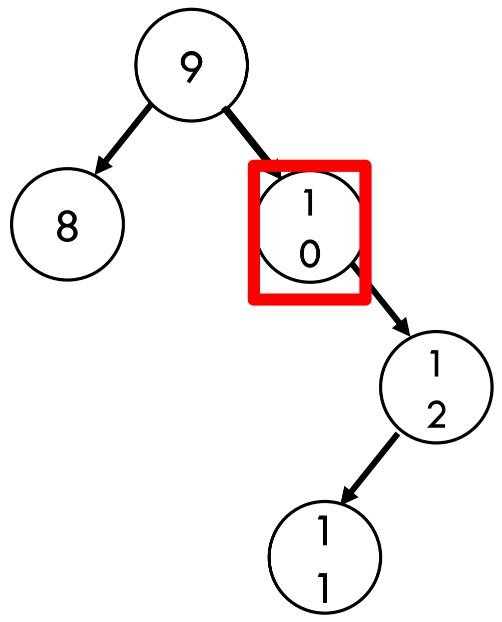
AVL Example: 8,9,10,12,11



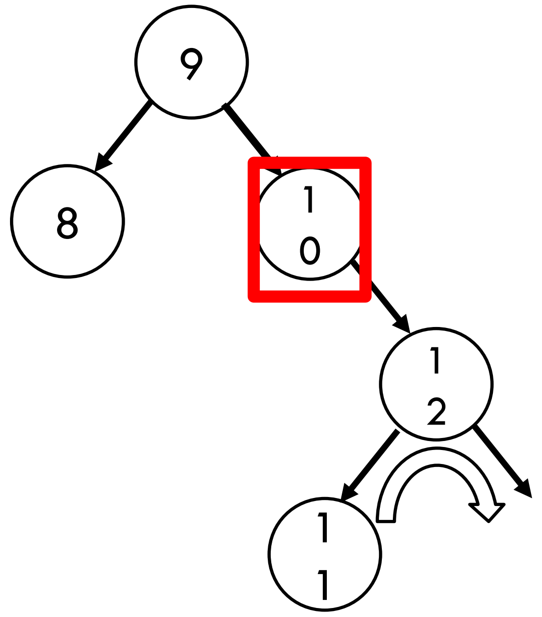
AVL Example: 8,9,10,12,11



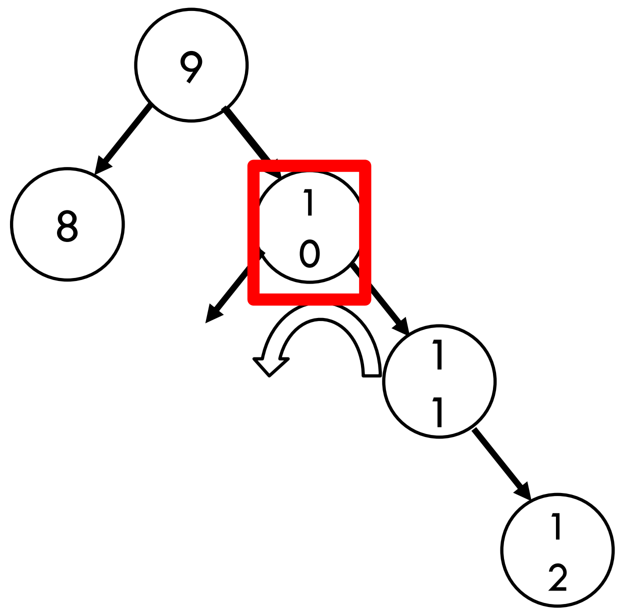
AVL Example: 8,9,10,12,11



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: 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. 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** $\Theta\left(f(n)\right)$ **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 $O(h)$ time to find an imbalance (if any) and fix it.

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

Deletion? With lazy deletion just the time to find, i.e. $\Theta(h)$.

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 unrolling or recursion trees.

Let’s try unrolling

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

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

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

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

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

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

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

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

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

…

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 number of nodes is $O(\log n)$.

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

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.

“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.

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 unrolling or recursion trees.

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 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 $\frac{1+\sqrt{5}}{2} \approx 1.62$

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.

On whiteboard:

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