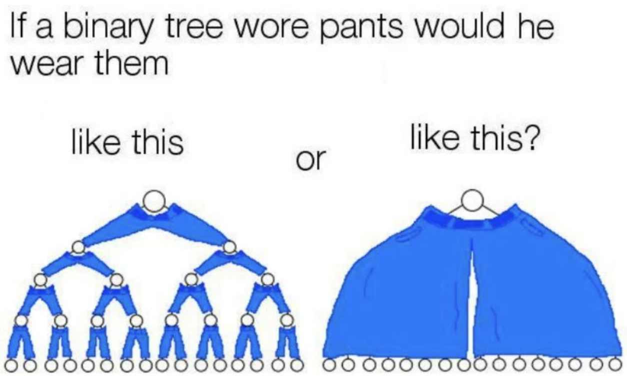
CS 1332R WEEK 4



Iterator/Iterable

Reference/Value Equality

Trees/Binary Trees/BSTs

Pointer Reinforcement

ANNOUNCEMENTS

- ❑

REFRESHER: Iterator/Iterable

- Iterator and Iterable are Java interfaces.

ITERABLE

Data structures that can be iterated over should implement the Iterable interface.

```
Iterator<T> iterator()
```

- *Returns an Iterator object with the following abilities →*

ITERATOR

```
boolean hasNext()

T next()
```

REFRESHER: Iterator/Iterable

Using an **Iterator** created from an **Iterable** object.

Explicit	Implicit

`Iterator<T> it = obj.iterator()`
`while (it.hasNext()) {`
`    T item = it.next();`
`    ...`
`}`
	`for (T item : obj) {`
	`    ...`
	`}`

REFRESHER: Reference vs. Value Equality

- ❑ Both are ways of comparing pieces of data - but what are we comparing?

REFERENCE: `==`

- Compares the **memory locations** of the two objects
- Only compares values in primitive types
  - int
  - double
  - char
  - boolean
  - etc.

VALUE: `.equals()`

- Every Object must implement a .equals() method that compares **data values**
- Will ensure every attribute of the two Objects are equal
- Not used on primitives

Tree ADT

- ❑ Trees are comprised of Nodes.
- ❑ A Node contains data and a reference(s) to its child nodes.
- ❑ **Every node is a root of its own subtree: TREES = RECURSION**

A constructor instantiating the tree <u>with a Collection of data</u>

```
add(T data)

T remove(T data)

int size()

Node<T> getRoot()
```

Tree Properties

EVERY TYPE OF TREE HAS A SHAPE AND/OR ORDER PROPERTY THAT DEFINES IT.

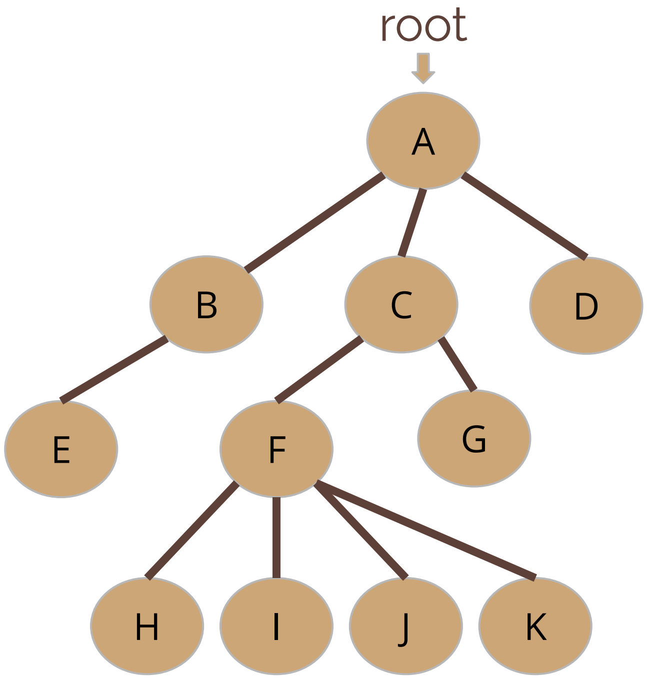
Shape Property

- Rules about the placement and structure of nodes in the tree
- Full, balanced, complete, etc. (will discuss in next slide)

Order Property

- Rules about the relationships between nodes based on ***their data***
- Ex: parent data must be greater than child data

TREE TERMINOLOGY



Ancestor: A is an ancestor of B, F, and J.

Descendant: B, F, and J are descendants of A.

Sibling: D and B are siblings because they have the same parent.

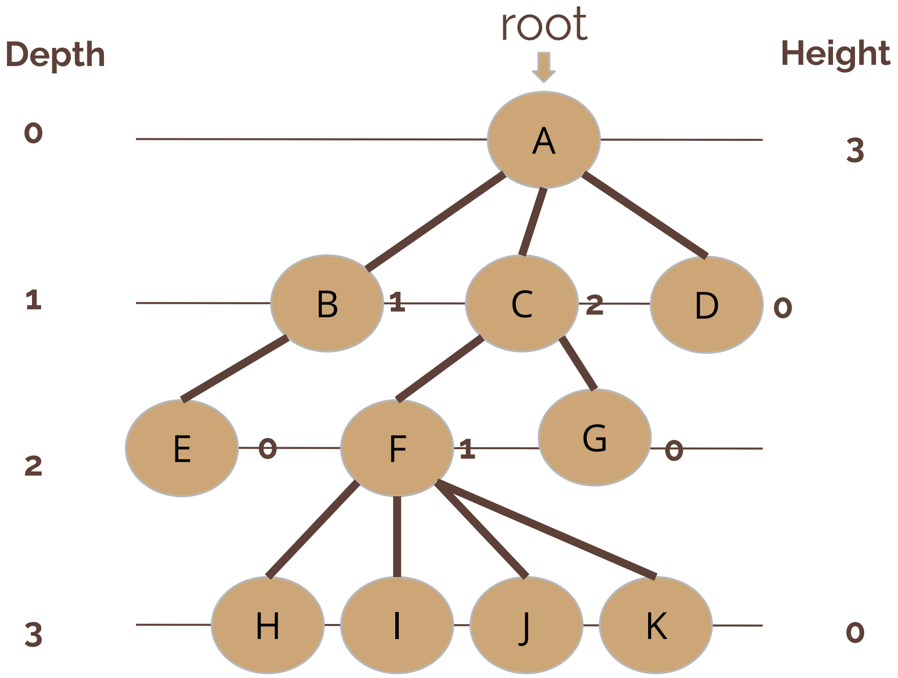
Leaf: A node with zero children (e.g. E, G, D, H)

Inner Node: A node with a parent and at least one child

Edge: the connection between two nodes

Branch: A series of nodes completely connected by edges

TREE TERMINOLOGY



Depth: # of edges it takes to get from the root to the node

Height: # of nodes it takes to get from a node to its deepest child

Let the height of a null node be -1.

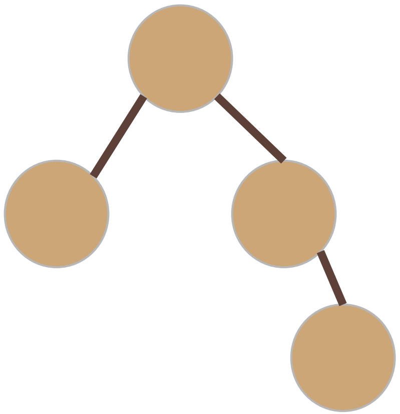
Therefore, the height of a leaf is 0.

```
height(node) = max(height(node.left), height(node, right)) + 1

depth of tree = height of tree = height(root)
```

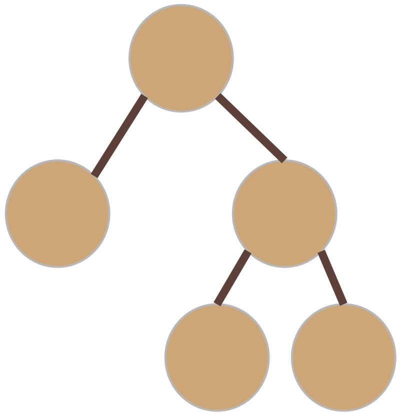
SHAPE PROPERTIES

BALANCED



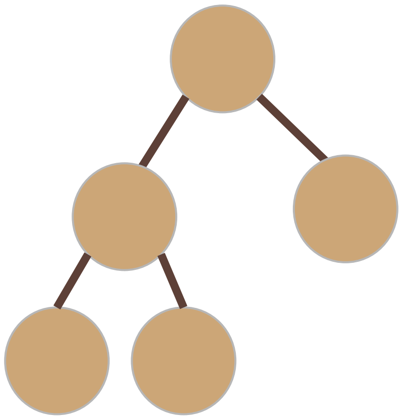
The heights of a node's children cannot differ by more than 1, max height of log(n).

FULL



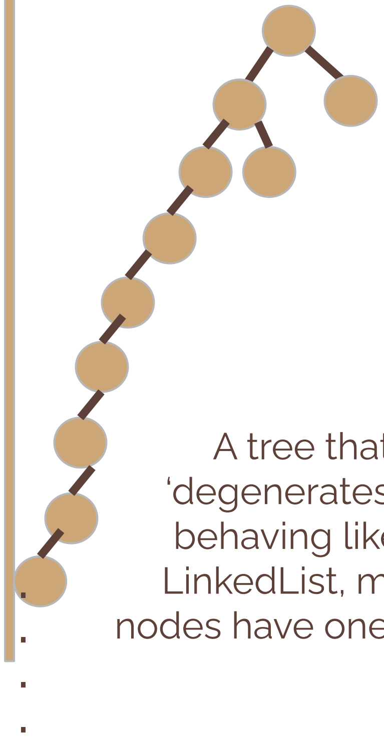
Each node has either 0 or the max # of children.

COMPLETE



Each level must be completely populated, with the last level filled left to right.

DEGENERATE



A tree that 'degenerates' to behaving like a LinkedList, most nodes have one child.

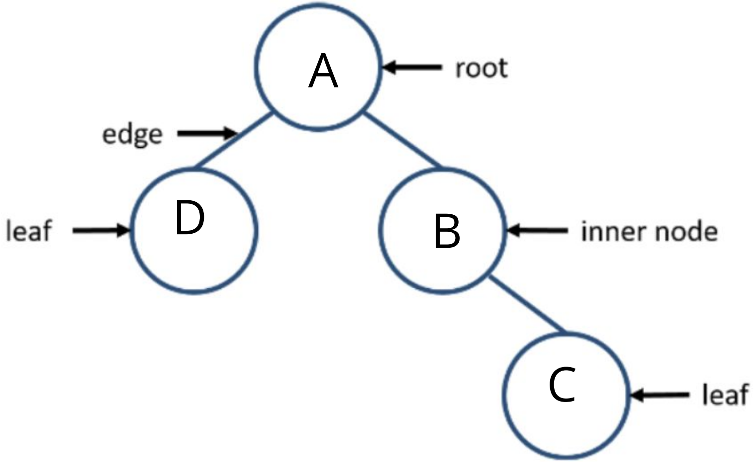
Binary Tree

SHAPE Property

A node cannot have more than two children.

ORDER Property

None



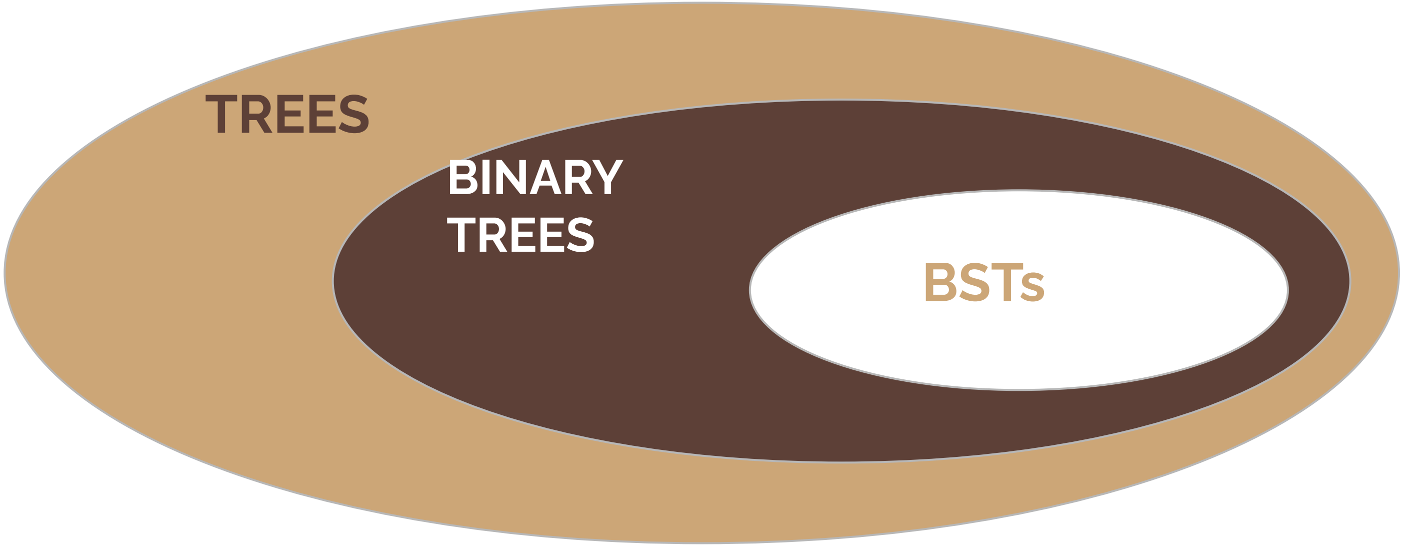
Binary Search Tree

SHAPE Property

A node cannot have more than two children.

ORDER Property

1. The left child's data must be less than the parent's data.
2. The right child's data must be greater than the parent's data.



Binary Search Tree

SHAPE Property

A node cannot have more than two children.

ORDER Property

1. The left child's data must be less than the parent's data.
2. The right child's data must be greater than the parent's data.

➔ We must be able to compare the data stored in a BST, therefore the data type **must implement Comparable<T>**.

- use obj1.compareTo(obj2) to compare two Objects
  - **obj1 < obj2 ⇒** a number **< 0**
  - **obj1 > obj2 ⇒** a number **> 0**
  - **obj1 == obj2 ⇒ 0**

Binary Search Tree: Search

- ➔ Search is the primary purpose of a binary *search* tree
- ➔ In a balanced tree, our search process is capped at **O(logn)** - the fastest search we have seen yet.

Compare the data you want to find with the current node's data. **There are 4 cases**:

1. If **data < currentNode.data**, go **left**.
2. If **data > currentNode.data**, go **right**.
3. If **data == currentNode.data**, you have found your data. Do what you need to do with it.
4. If **currentNode == null**, the data you are looking for is not in the tree.

Binary Search Tree: Add

- All data added to the tree is added as a leaf node.
- When adding, we must **maintain the order property**.

1. Search for the data you would like to add.
2. Eventually, when **currentNode == null**, you know you have reached the spot where you need to add the new data. Create a node with the new data.

How do we connect the new node back to the rest of the tree?

pointer reinforcement

EXAMPLE: csvistool.com

Implementation Methods

LOOK AHEAD

- Look at the nodes ahead and check if either is null before manipulating
- Messier code
- More mistakes/room for error

POINTER REINFORCEMENT

- Recursive technique
- Less efficient…
- Cleaner code :)
- HIGHLY RECOMMEND

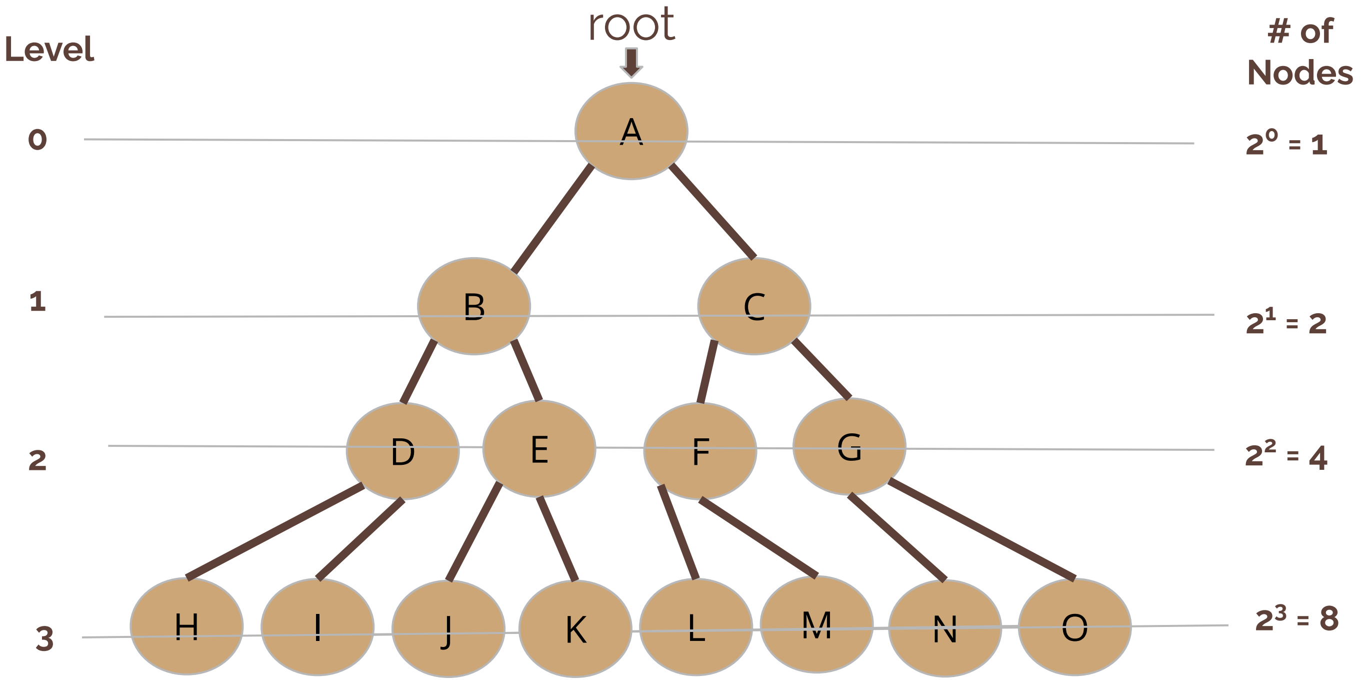
Pointer Reinforcement Example

```
private Node addExample(Node curr, T data):
    if curr is null:
        return new Node()
    if data > curr.data:
        curr.right = addExample(curr.right, data)

    return curr
```

Binary Search Tree: Efficiencies

	Adding	Accessing	Height
Average	*O(log n)*	*O(log n)*	*O(n)*
Worst	*O(n)*	*O(n)*	*O(n)*



LEETCODE PROBLEMS

[235. Lowest Common Ancestor]

[230. Kth Smallest Element in a BST]

Any questions?

Name
Office Hours
Contact

Name
Office Hours
Contact

Let us know if there is anything specific you want out of recitation!