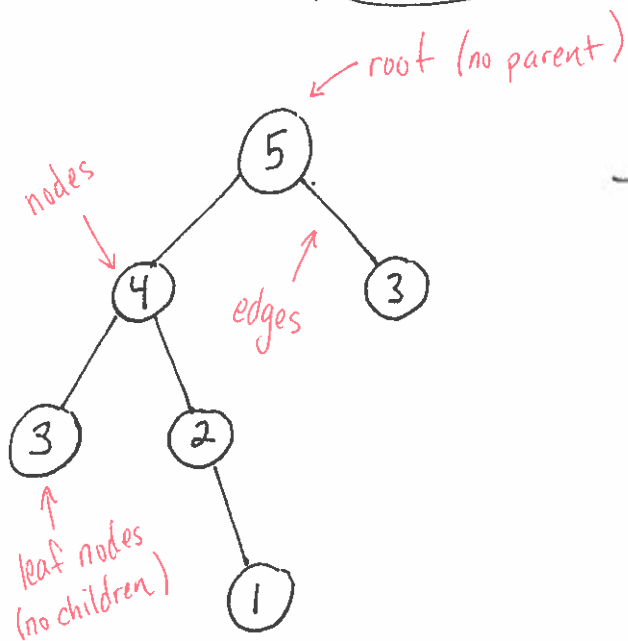


This is a TREE: another linked, node-based data structure!



* Terminology above is review.
We saw those when we covered binary heaps.

Recursive Definition:

A tree is a collection of nodes, which can be empty. If not empty, there is a single root node, r , and zero or more subtrees, $T_1, T_2, \dots, T_k$, whose roots are connected by a (directed) edge from r .

The hierarchical ordering of our diagram conveys parent-child relationships.

A node can have many children, but only one (at most) parent.

* Sources vary on the specifics of some of these definitions! Don't worry too much about that.

(root node has no parent)

Definition:

If tiny details matter for a tree problem in this class, we will happily clarify.

A path is a sequence of edges that connects a sequence of nodes.

(distinct?)*

(distinct?)*

Example: Path from 5 to 1 in the above graph: 5-4-2-1

More on graphs later this quarter!

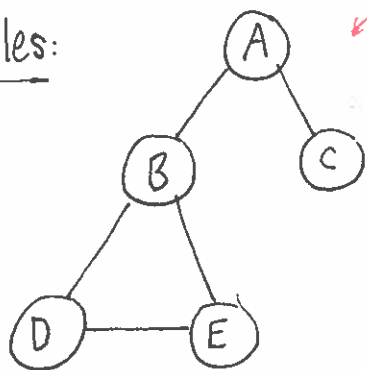
A graph is a collection of nodes and edges. A tree is a type of graph.

A tree cannot have a cycle (a non-empty path from some node to itself).

Another way to think of this:

A node cannot be its own ancestor,
and a node cannot have multiple
parents.

Examples:

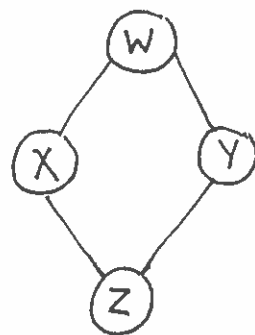


Multiple paths from
A to E now:

A-B-E

A-B-D-E

Not conveying parent-child relationship.

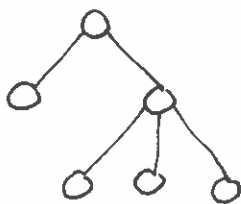


Cannot have
two parents.

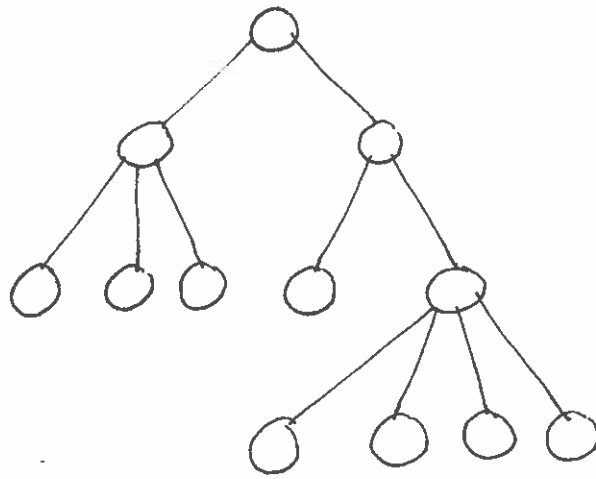
Note that the presence of a cycle means we have more than one
path from some node to at least one other node in our graph.

A tree must be connected: there must be a path (not edge)
in a tree to every other node in that tree.

The following is not a tree. It is a forest.



A puzzle: How many edges must there be in a tree with n nodes?

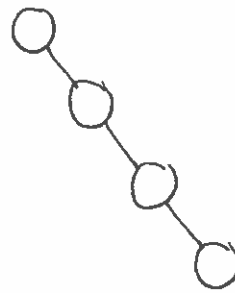


Answer: $n-1$ (for $n > 0$)

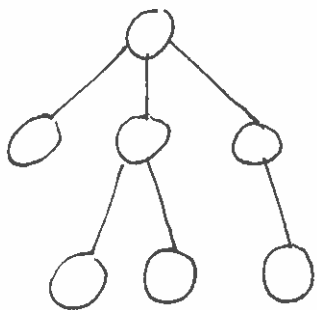
↑
every node has an edge connecting it to its parent, except for the root!

The height of a tree is the greatest distance (in terms of # of edges) from the root to any of the leaf nodes in the tree.

(58) height: 0



height: 3



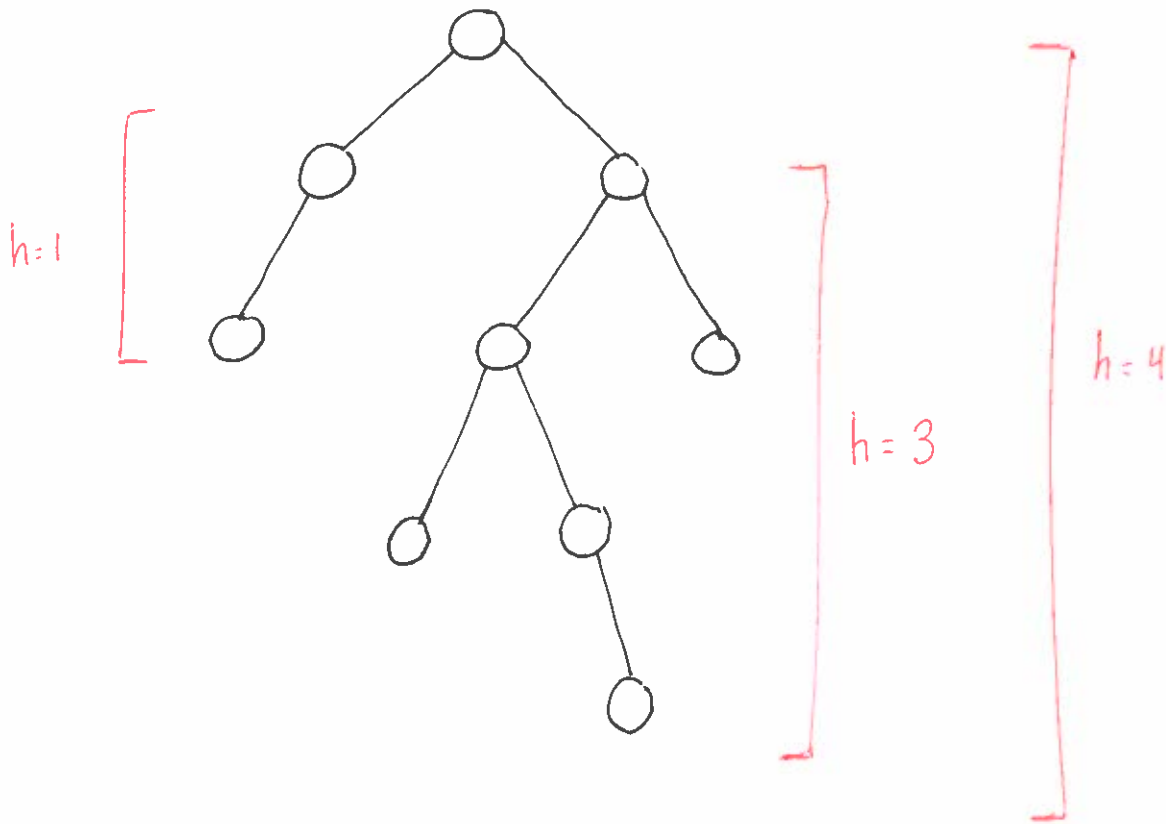
height: 2

height: -1
↑

(the empty tree)

This will come in handy on the following page!

A puzzle: How can we define the height of a tree in terms of the heights of its left and right subtrees?



Recursive definition:

$$\text{height}(\text{root}) = \max(\text{height}(\text{root} \rightarrow \text{left}), \text{height}(\text{root} \rightarrow \text{right})) + 1$$

↑
for non-empty tree

$$\text{height}(\text{nullptr}) = -1$$

← empty tree

Add one level when uniting left and right subtrees with parent node! ↓

Let's see how this applies to a single node:

○] height: 0

Left and right subtrees are empty.
Their heights are -1 by definition.

$$\text{So, } \text{height}(\text{root}) = \max(-1, -1) + 1 = 0 \quad \ddot{\smile}$$

A binary tree is a tree in which every node has at most two children.

```
struct TreeNode {
```

```
    int data; // or whatever data type suits our needs
```

```
    TreeNode *left;
```

```
    TreeNode *right;
```

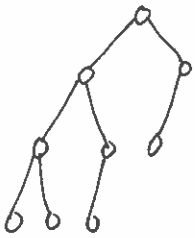
```
};
```

// Oooh! Recursion will be handy here!

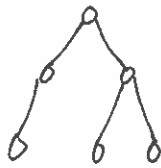
↑ for algorithms that need to explore both subtrees of every node!

Recall that a binary tree is complete if all levels of the tree are completely filled up, except perhaps for the last level, whose nodes must all be flushed to the left (no gaps before or between nodes).

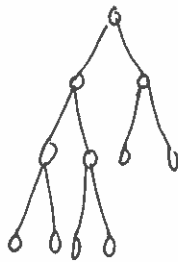
Examples:



Not complete



Not complete



Complete :)



Not complete



Complete :)

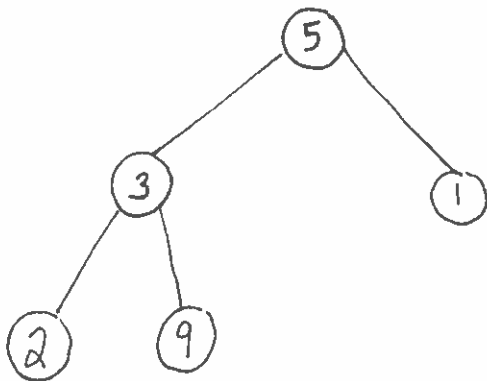
Recall that a minheap/priority queue is (structurally) a complete binary tree.
↑ (it also has an ordering property)

What is the exact height of a complete binary tree with n nodes?

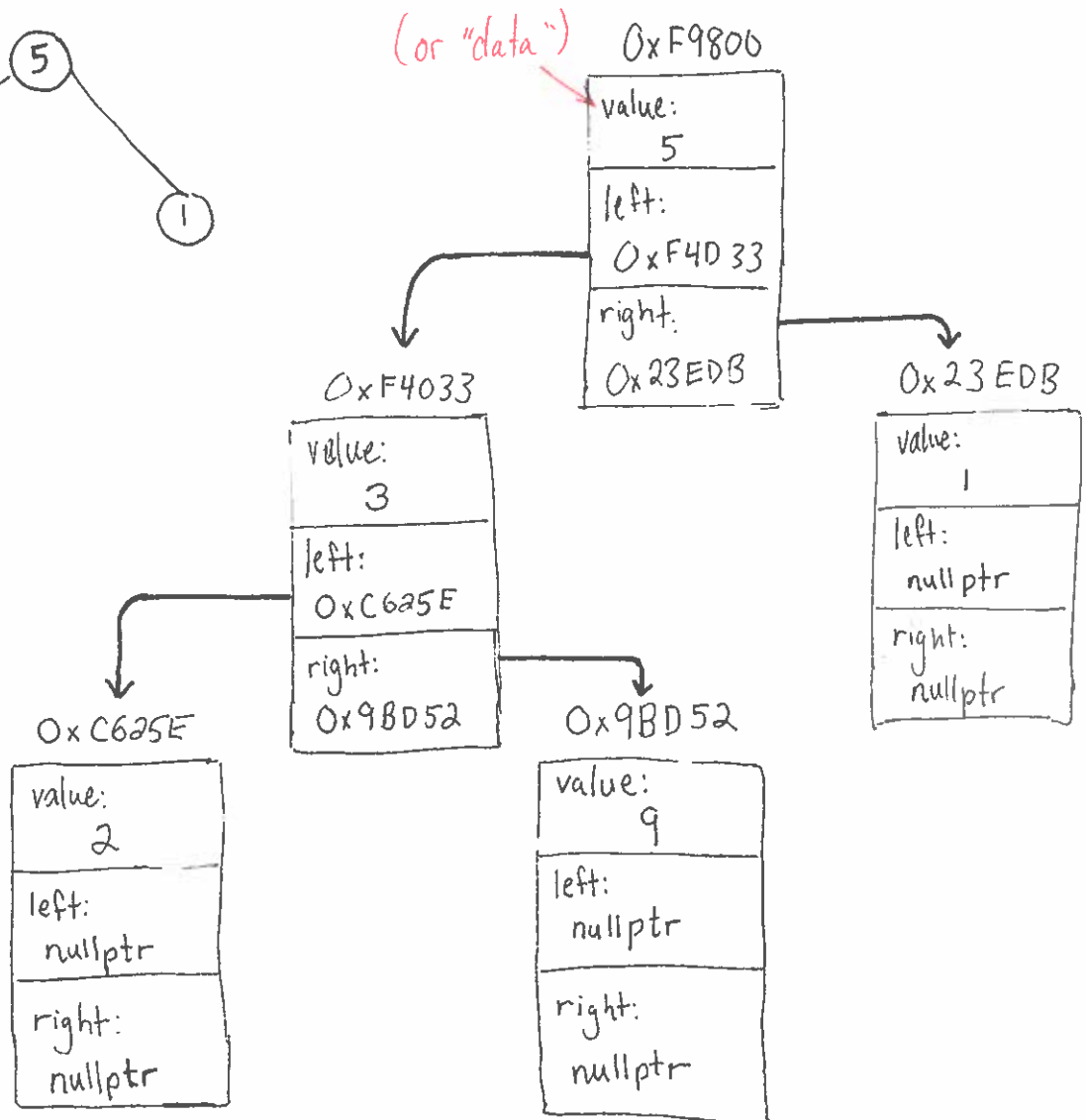
Answer: $\lfloor \log_2 n \rfloor$ (The derivation of this was covered in our binary heap lecture.)
← This is also the minimum height of a binary tree with n nodes! Max is $(n-1)$.

Let's take a peek behind the curtain at a binary tree in memory:

Abstract View:



Behind the Curtain:



We would also have a dedicated variable to store the address of the root node, like so.

root:

0xF9800

```
struct TreeNode {  
    int data;  
    TreeNode *left;  
    TreeNode *right;  
};
```

Check your understanding: (from previous page)

Given root
0xF9800, how do we get to the data field for the node containing 9?

Answer: root → left → right → data

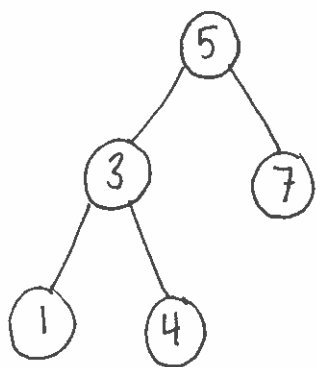
If we have a `TreeNode` constructor function that takes a single integer to be added to a new node, how do we give the node containing 1 a left child containing 4?

Answer: root → right → left = new `TreeNode(4)`;

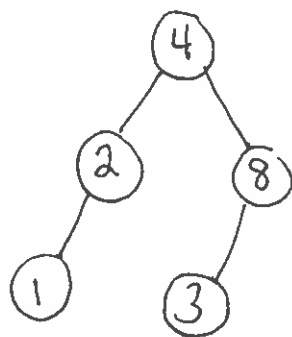
A binary search tree (BST) is a binary tree with the following ordering property:

For every node, x , the values in the left subtree of x must be less than or equal to the value at node x , and the values in its right subtree must be greater than the value at node x .

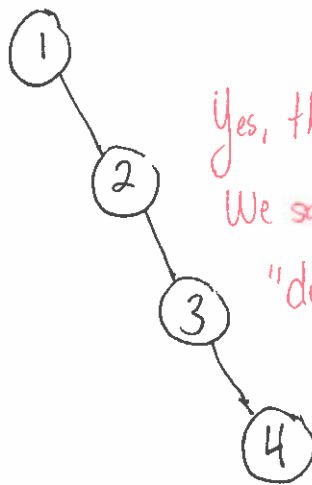
Examples:



BST ☺



Not a BST!
(3 cannot go in the right subtree of 4!)



Yes, this is a BST.
We say it has
"degenerated into a
linked list."

↖ This conveys the
max height of
a BST.

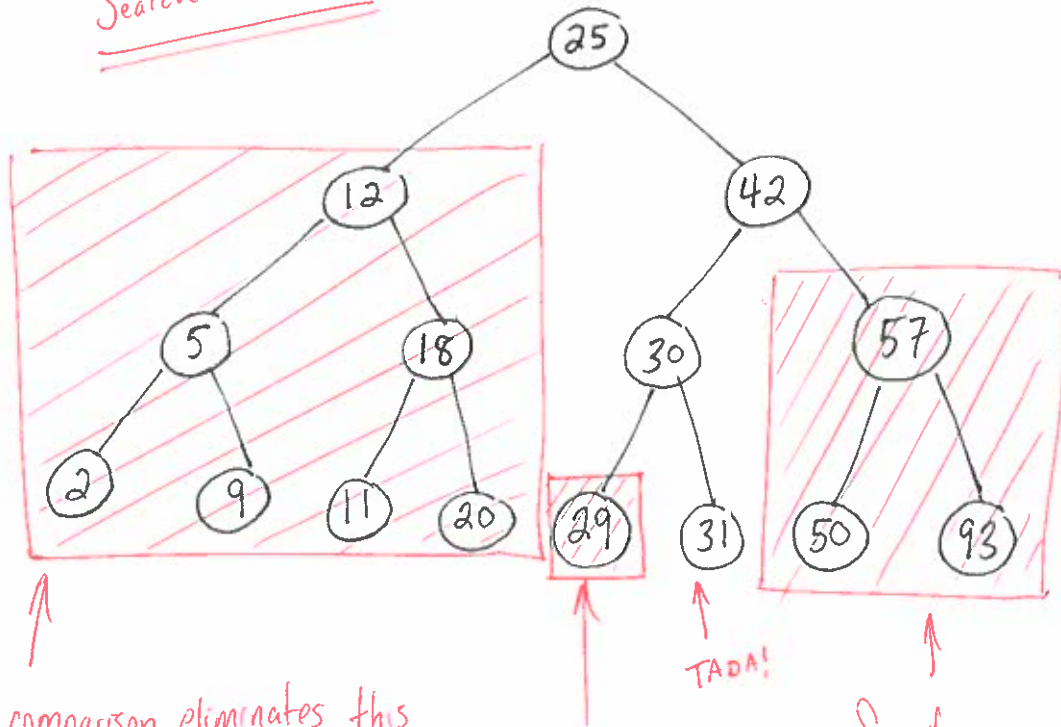
The ordering property gives rise to a search algorithm!

What is the maximum height of a tree with n nodes? Answer: $n-1$

What is the minimum height of a tree with n nodes? Answer: $\lceil \log_2 n \rceil$

Search algorithm for BSTs:

Search for 31:



First comparison eliminates this subtree from consideration ($\sim 1/2$ of our nodes!); since $31 > 25$, it would have to be in 25's right subtree (if it's present)

Third comparison eliminates this subtree from consideration

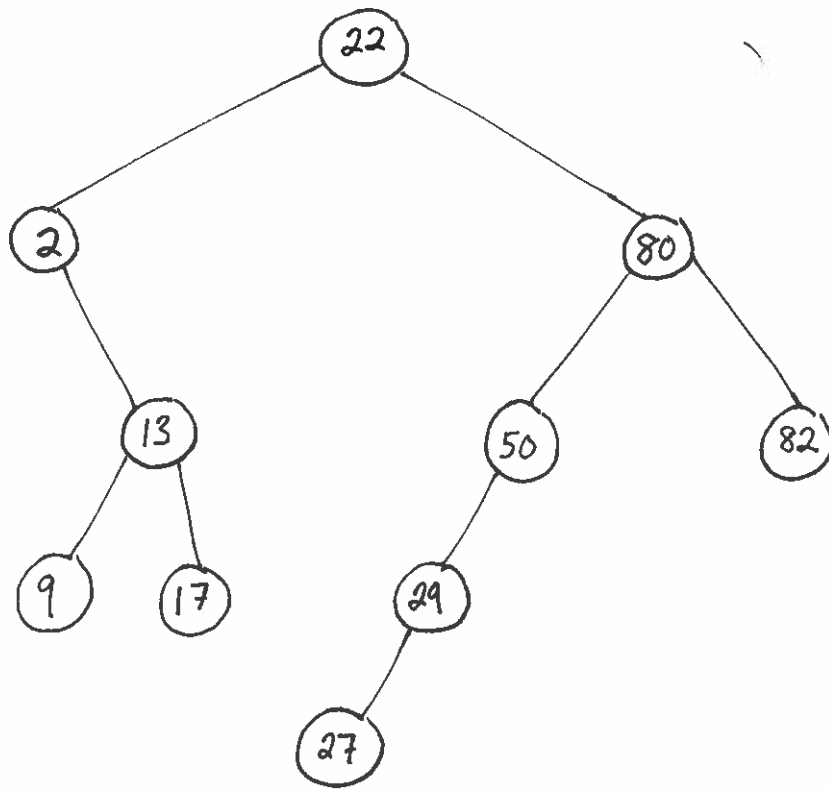
Second comparison eliminates this subtree from consideration ($\sim 1/2$ of our remaining nodes - so, about $1/4$ of all the nodes in our tree)

- * If our BST is balanced and bushy and full, our search algorithm has a worst-case runtime of $O(\log n)$ and emulates, to some degree, binary search through a sorted vector/array.

Let's insert the following values into a BST:

22, 2, 80, 82, 50, 29, 13, 27, 17, 9

* We insert these one by one, starting at the root and always going left ($\leq$) or right ($>$) as appropriate for the values in question. Our resulting tree is as follows:



Runtimes for BST operations:

* Some sources prefer to give these in terms of h , the height of the tree. We will explore that in our next set of lecture notes.

	Best-case	Worst-case	Average-case
search	$O(1)$	$O(n)$	$O(\log n)$
insertion	$O(1)$	$O(n)$	$O(\log n)$
deletion	$O(1)$	$O(n)$	$O(\log n)$

↑

← We will discuss the deletion algorithm next time.

These happen even if the tree is arbitrarily large! Remember, we do not restrict the size of our input (the BST, in this case) when considering best-case runtimes.

The $O(n)$ runtimes are encountered when our BST has devolved into a linked list and we end up having to traverse the entire list for the given operation.

The average-case $O(\log n)$ runtimes are based on the expected height of a BST (in the statistical sense of "expected," as in, "expected value"), which is $O(\log n)$. That can be demonstrated empirically or proven formally. The proof is beyond the scope of this class but is readily Googleable.

Traversal Algorithms

These allow us to visit each node in a binary tree in a prescribed order.

On the following pages, we cover four common traversal algorithms:

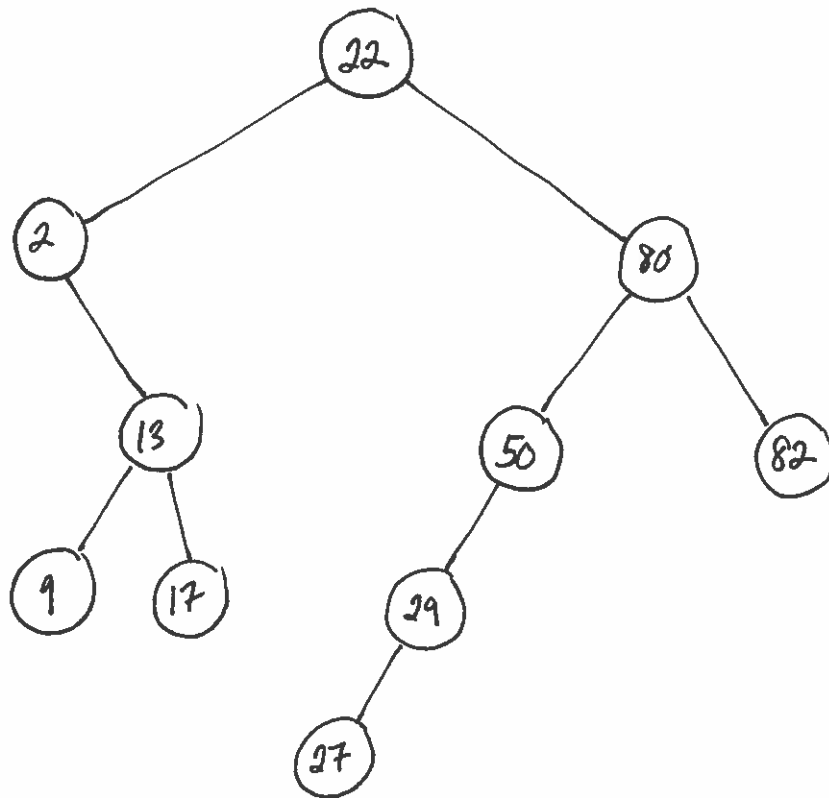
1. pre-order traversal
2. post-order traversal
3. in-order traversal
4. level-order traversal

We will see some applications of these in our next lecture. 😊

pre-order traversal

↑
root is processed before
left and right subtrees.

1. process the root of this (sub) tree
2. traverse left subtree with pre-order traversal
3. traverse right subtree with pre-order traversal



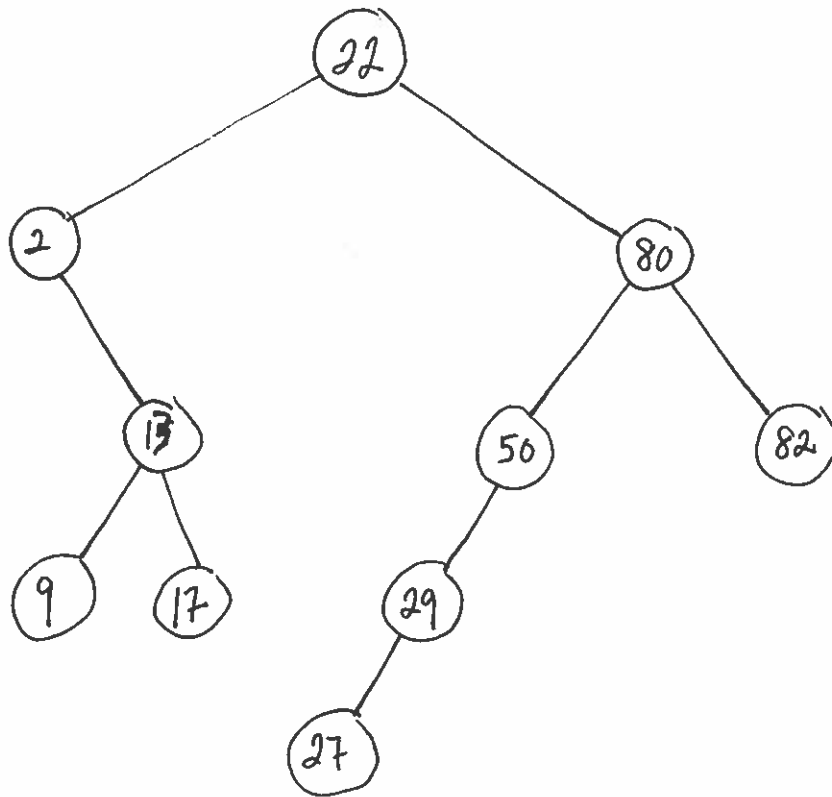
pre-order traversal: 22, 2, 13, 9, 17, 80, 50, 29, 27, 82

↖ * In a pre-order traversal, the root is always the first node to be processed.

post-order traversal:

↑
root is processed after
left and right subtrees

1. traverse left subtree with post-order traversal
2. traverse right subtree with post-order traversal
3. process the root of this (sub)tree



post-order traversal: 9, 17, 13, 2, 27, 29, 56, 82, 80, 22

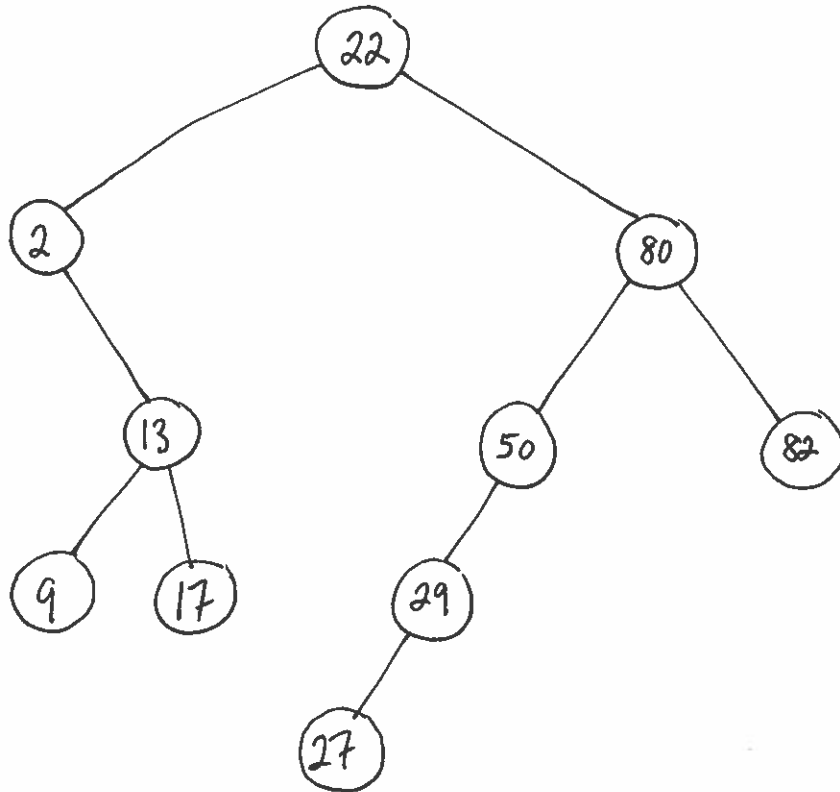


* In a post-order traversal, the root is always the last node to be processed.

in-order traversal:

↑
root is processed in between
left and right subtrees.

1. traverse the left subtree with in-order traversal
2. process the root of this (sub)tree
3. traverse the right subtree with in-order traversal



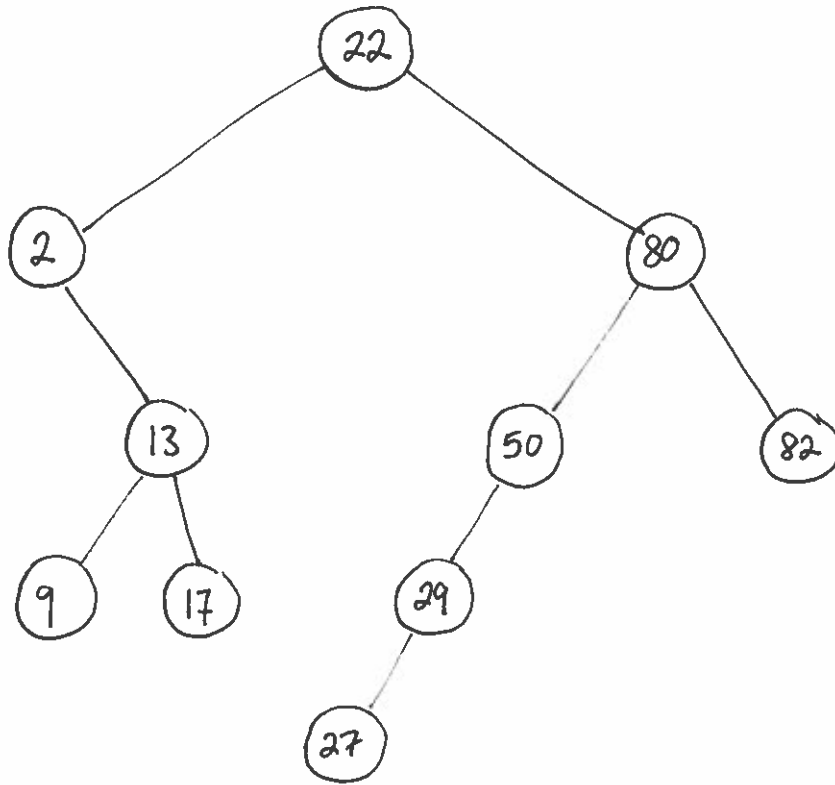
in-order traversal: 2, 9, 13, 17, 22, 27, 29, 50, 80, 82

↑
* Performing an in-order traversal on a binary search tree (not just an arbitrary binary tree) processed the values in our BST in sorted order.

level-order traversal:

↑
we go level by level

Go one level at a time, processing each node from left to right before moving on to the next level.



level-order traversal: 22, 2, 80, 13, 50, 82, 9, 17, 29, 27