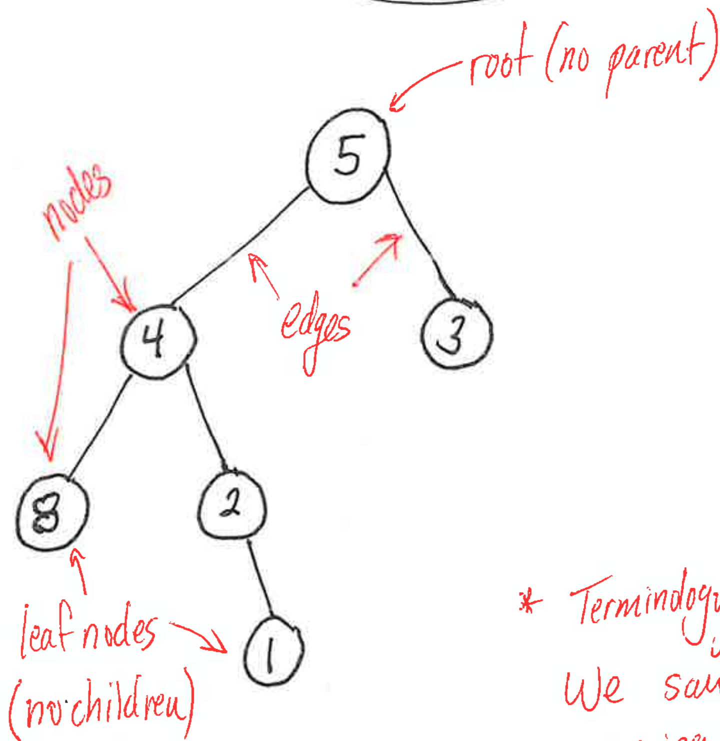
This is a **tree**: ← another linked, node-based structure!



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

* Terminology above is review. We saw those when covering minheaps.

Definitions:

Some sources say the nodes/edges in a path must be distinct (no repeats). Don't worry about that. If that small detail matters for a tree problem in this class, we will clarify.

① **Path**: A path is a sequence of nodes where each successive pair of nodes is connected via an edge.

Example: A path from 5 to 1 in the tree above is 5 4 2 1.

② **Cycle**: A cycle is a non-empty path (it has at least one edge) that starts and ends at the same node but does not repeat any edges.

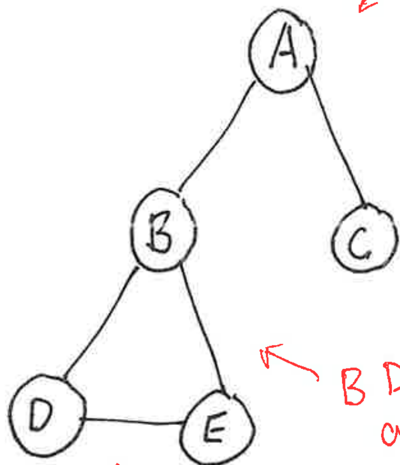
← examples on pg. 3

Some important tree properties:

1. Every non-empty tree has exactly one root node.
2. Every node has exactly one parent, except for the root node, which has no parent.
3. A tree cannot contain a cycle. ← examples on pg. 3
4. A tree must be connected, meaning there must be a path from every node in the tree to every other node in that tree. ↖ example on pg. 4

↑ note that this says path (not edge)

Examples:

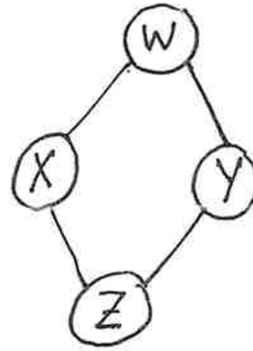


Multiple paths from A to E:

① A B E
② A B D E

B D E B is a cycle.

This edge does not convey a parent-child relationship.



WXW is not a cycle because it uses the same edge twice.

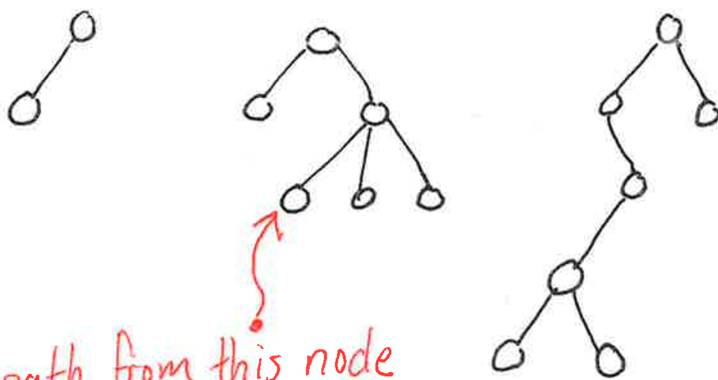
WXZYW is a cycle.

i.e., another way to think about the fact that trees cannot have cycles.

* Another way to think of this: ① a node cannot be its own ancestor, ② a node cannot have multiple parents, and ③ an edge cannot convey anything other than a parent-child relationship (such as a sibling relationship).

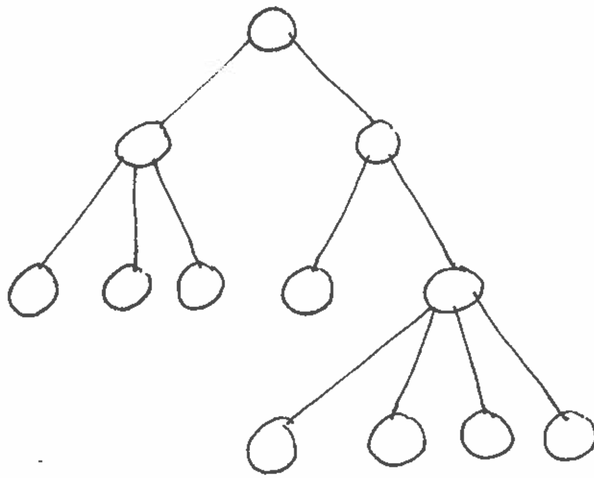
The following is not a tree, because it is not connected.

It is, however, a forest.



There is no path from this node to this one.

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

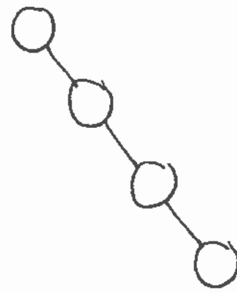


Answer: $n-1$ (for $n > 0$)

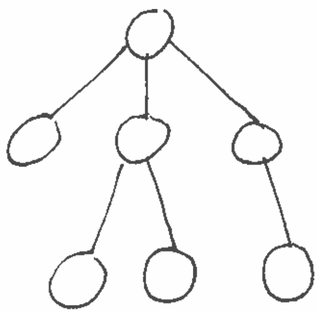
↑ every node has an edge connecting it to its parent, except for the root!

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



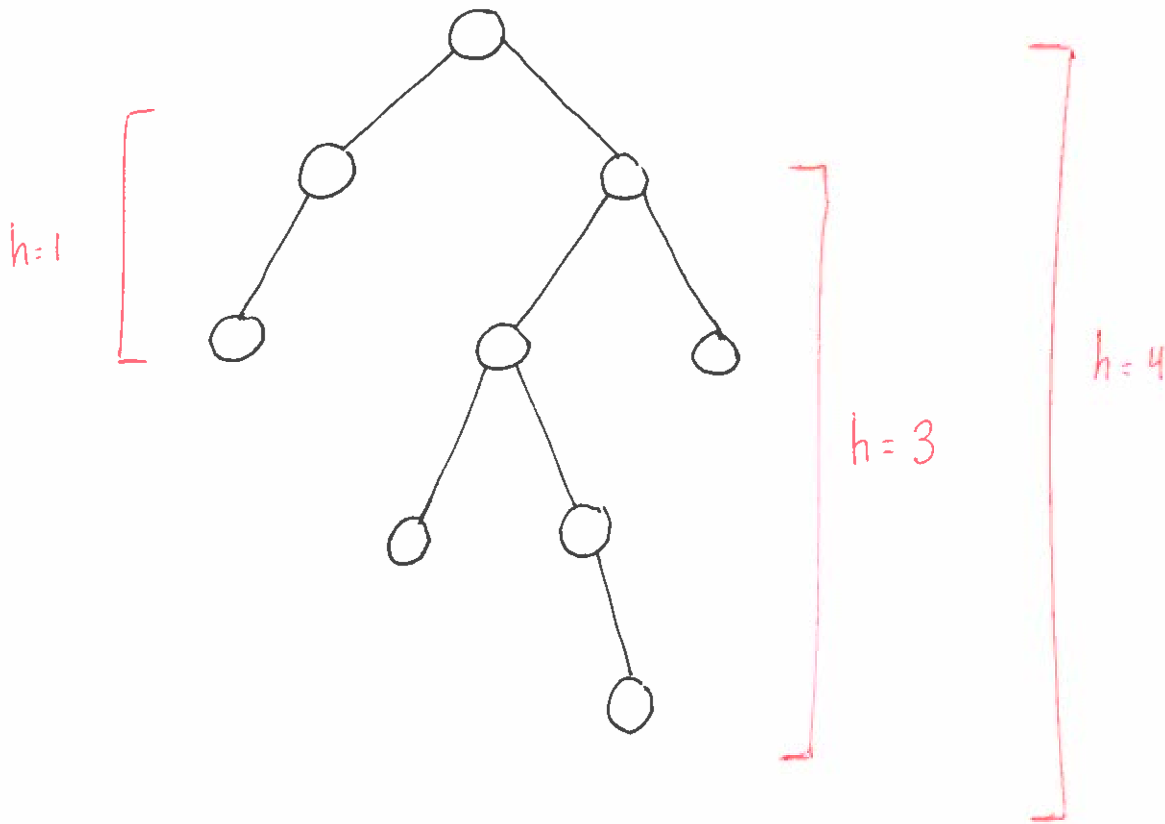
height: 2

height: -1

↑ This will come in handy on the following page!

(the empty tree)

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



Recursive definition:

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

height(root) = max(height(root->left), height(root->right)) + 1

↑ for non-empty tree

height(nullptr) = -1

↖ empty tree

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

○] height: 0

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

So, height(root) = max(-1, -1) + 1 = 0 :)

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

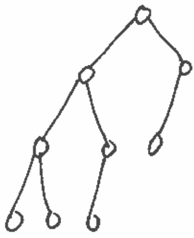
```
struct TreeNode {
    int data;          // or whatever data type suits our needs

    TreeNode *left;    // Oooh! Recursion will be handy here!
    TreeNode *right;   //
};
```

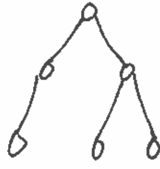
↑ for algorithms that need to explore both subtrees of every node!

Recall that a binary tree is <u>complete</u> 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).

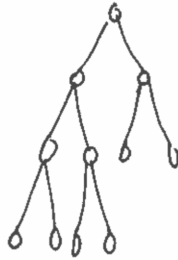
Examples:



Not complete



Not complete



Complete :)

Not complete

Complete :)

Recall that a minheap / priority queue is (structurally) a complete binary tree.

↖ (it also has an <u>ordering</u> property)

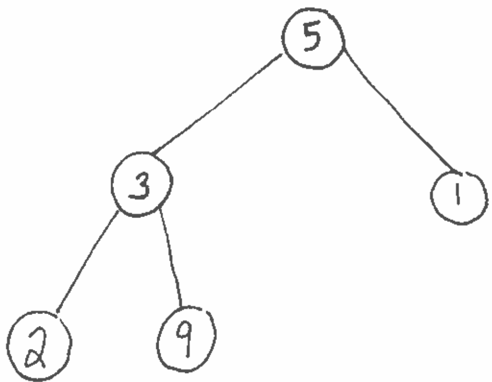
What is the exact height of a complete binary tree with n nodes?

<u>Answer:</u> $\lfloor \log_2 n \rfloor$ (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:

(or "data")

0xF9800
value: 5
left: 0xF4D33
right: 0x23EDB

0xF4033
value: 3
left: 0xC625E
right: 0x9BD52

0x23EDB
value: 1
left: nullptr
right: nullptr

0xC625E
value: 2
left: nullptr
right: nullptr

0x9BD52
value: 9
left: nullptr
right: nullptr

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 `0x F9800`, how do we get to the data field for the node containing 9?

Answer: `root -> left -> right -> data`

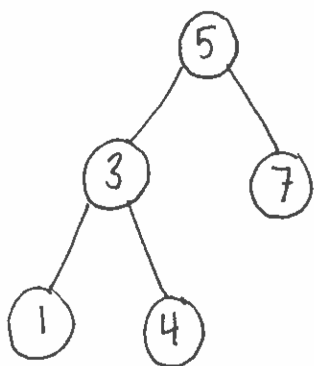
If we have a TreeNode constructor function that takes a single integer to be added to a new node, how do we give the node containing 1 a left child containing 4?

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

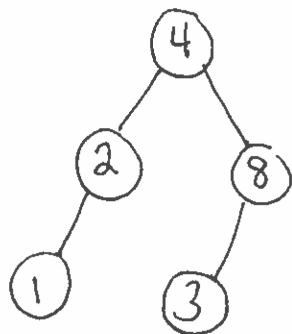
A <u>binary search tree</u> (<u>BST</u>) is a binary tree with the following ordering property:

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

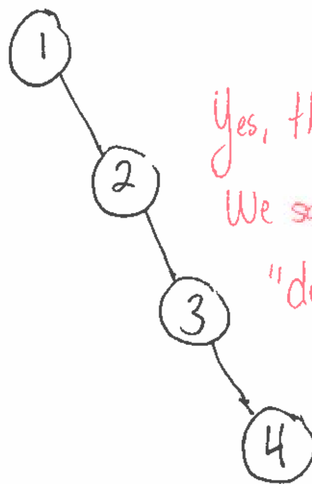
<u>Examples:</u>



BST :)



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



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

This conveys the max height of a BST.

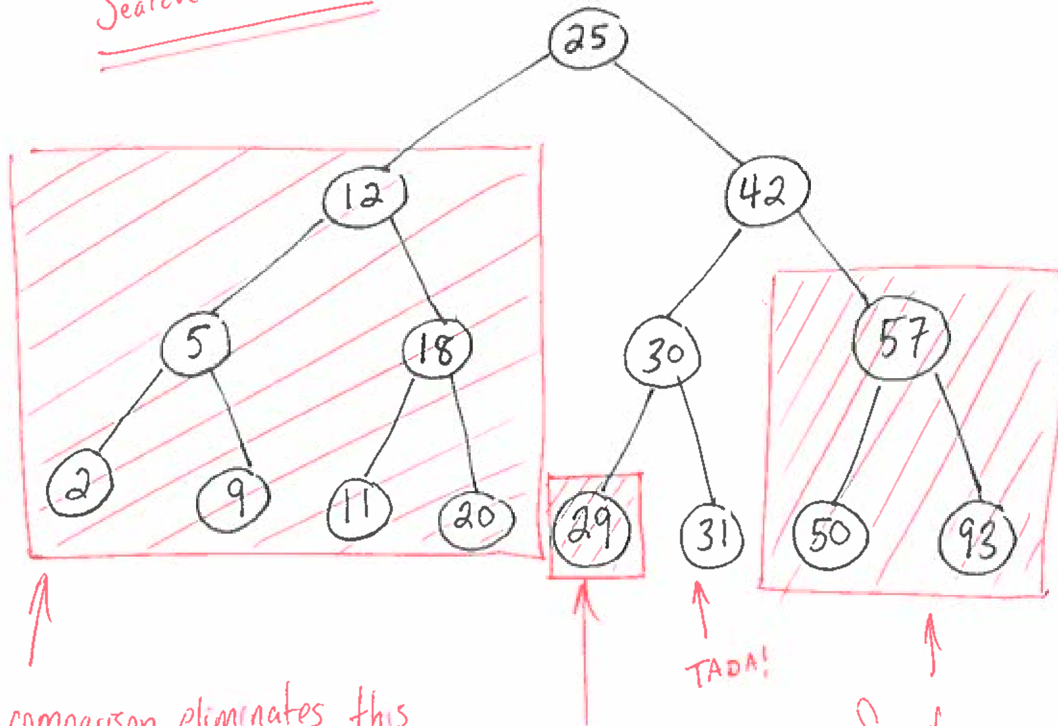
The ordering property gives rise to a search algorithm!

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

What is the <u>minimum</u> height of a tree with n nodes? <u>Answer:</u> $\lfloor \log_2 n \rfloor$

Search algorithm for BSTs:

Search for 31:



First comparison eliminates this subtree from consideration (~ 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 (~1/2 of our remaining nodes — so, about 1/4 of all the nodes in our tree)

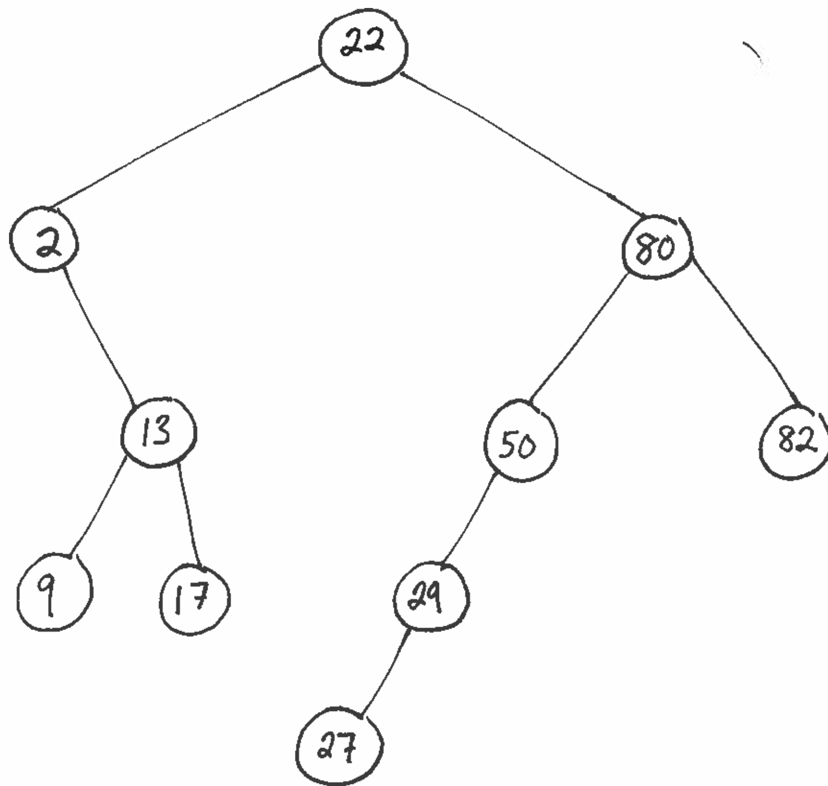
* If our BST is balanced and bushy and full, our search algorithm has a worst-case runtime of $O(\log n)$ and emulates, to some degree, binary search through a sorted vector/array.

rch for 31

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 (≤) 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:

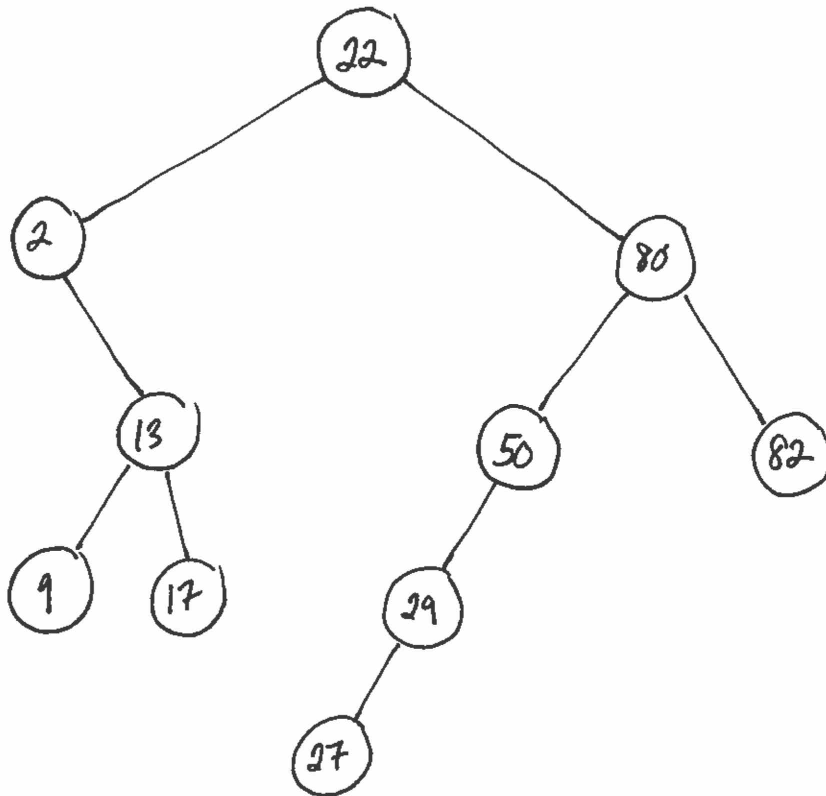
1. pre-order traversal
2. post-order traversal
3. in-order traversal
4. level-order traversal

We will see some applications of those 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



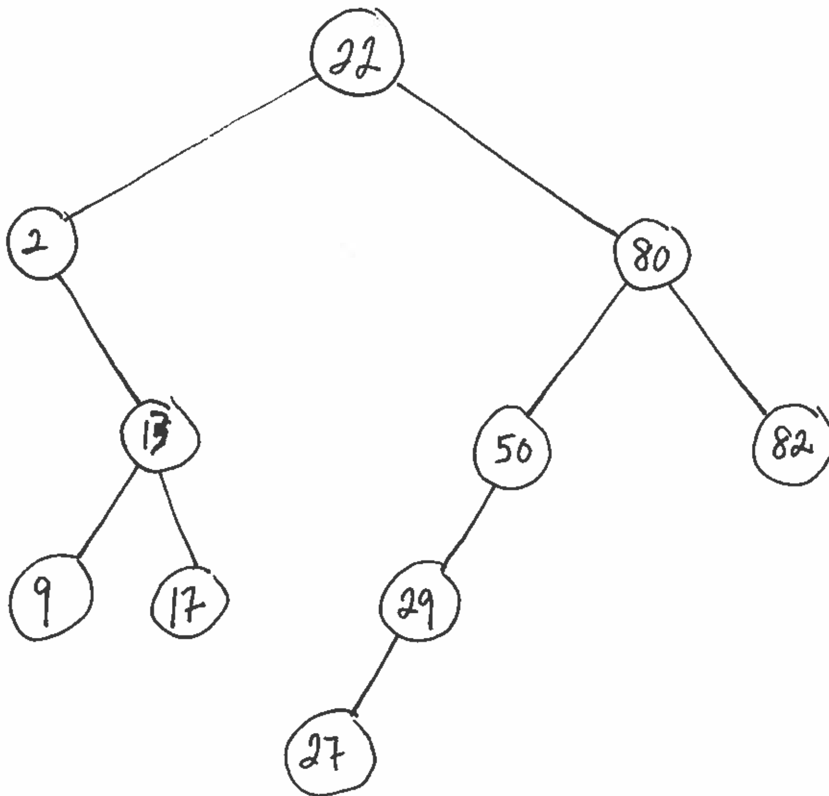
pre-order traversal: 22, 2, 13, 9, 17, 80, 50, 29, 27, 82

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

post-order traversal:

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

root is processed *after* left and right subtrees



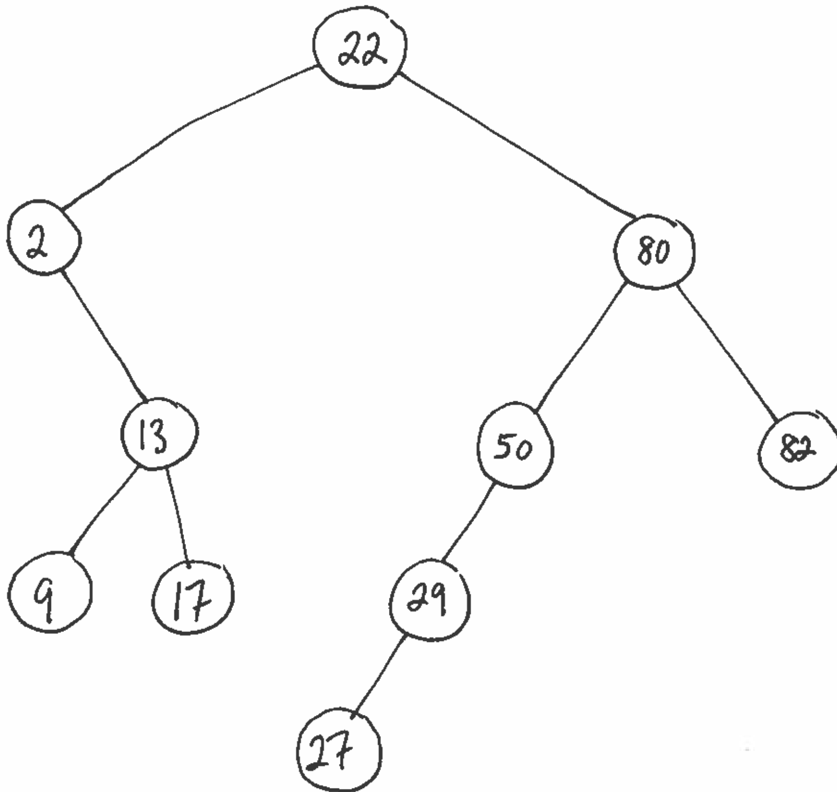
post-order traversal: 9, 17, 13, 2, 27, 29, 50, 82, 80, 22

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

in-order traversal:

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

↑ root is processed in between left and right subtrees.



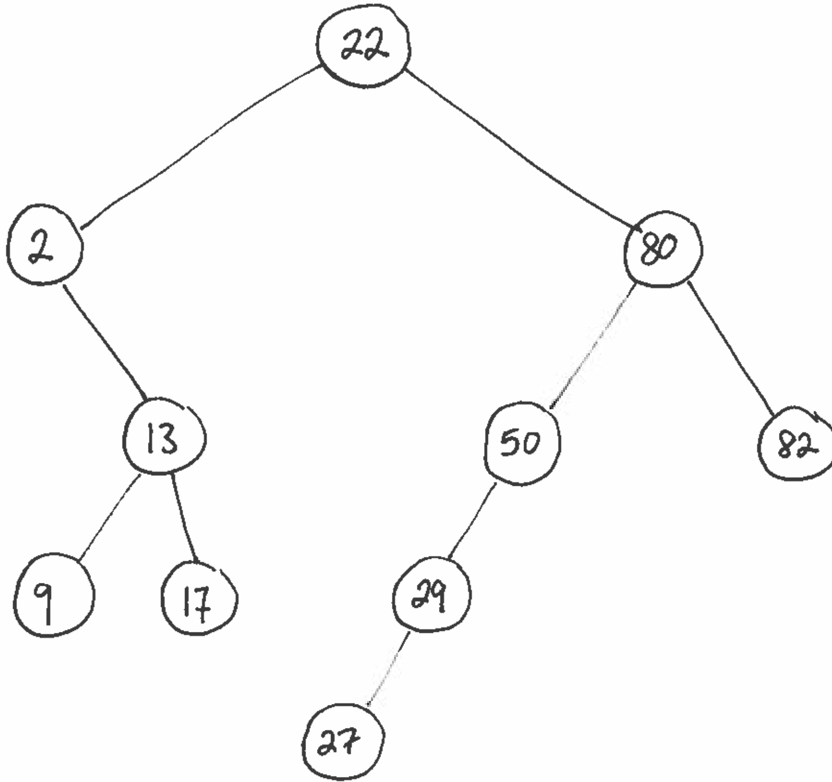
in-order traversal: 2, 9, 13, 17, 22, 27, 29, 50, 80, 82

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

↑ we go level by level



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