Unit 2 Build Week Pt. 2

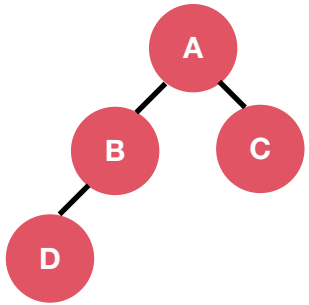
AGENDA

- Binary Trees
- There is a unit assessment you need to take before the end of B week

Binary Trees

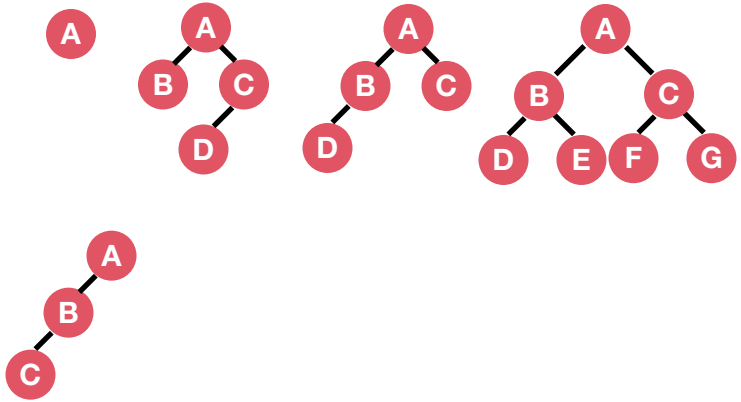
BINARY TREES: REVIEW

- Comprised of nodes
- Each node can have [0, 2] children
- Can have different properties (BST, balanced, complete, perfect etc.)
- Can be represented in a couple of ways



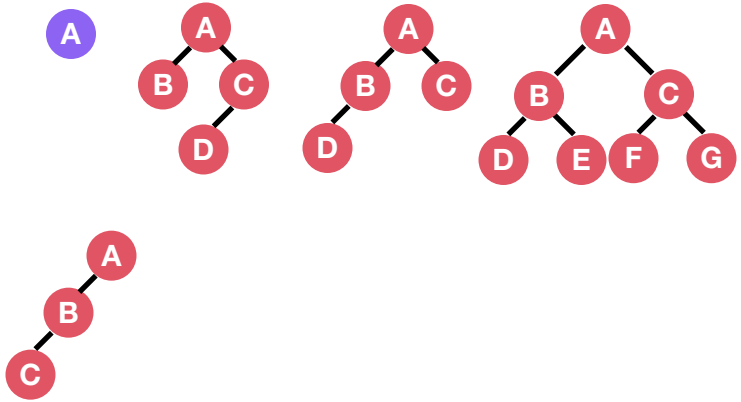
BINARY TREES: REVIEW

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



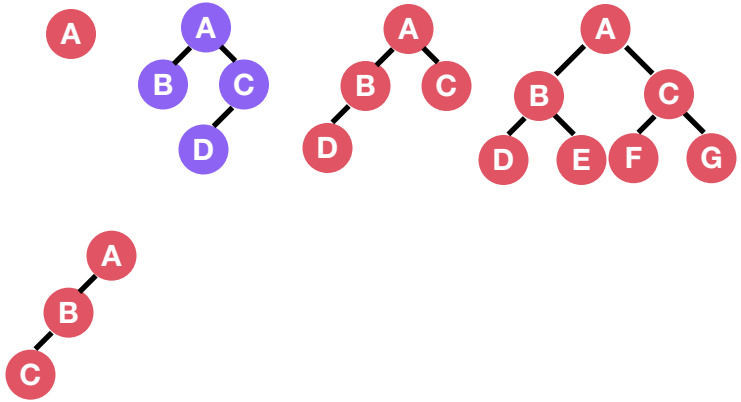
WHAT CAN A TREE LOOK LIKE?

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



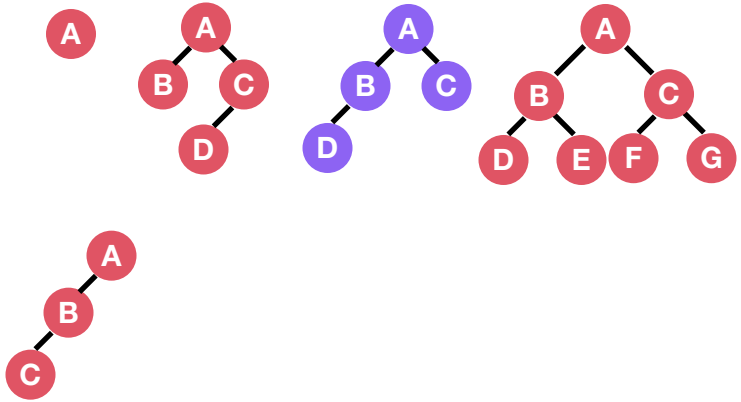
WHAT CAN A TREE LOOK LIKE?

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



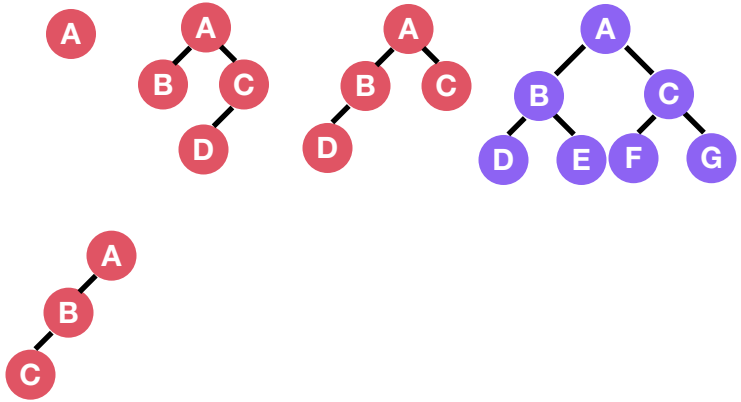
WHAT CAN A TREE LOOK LIKE?

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



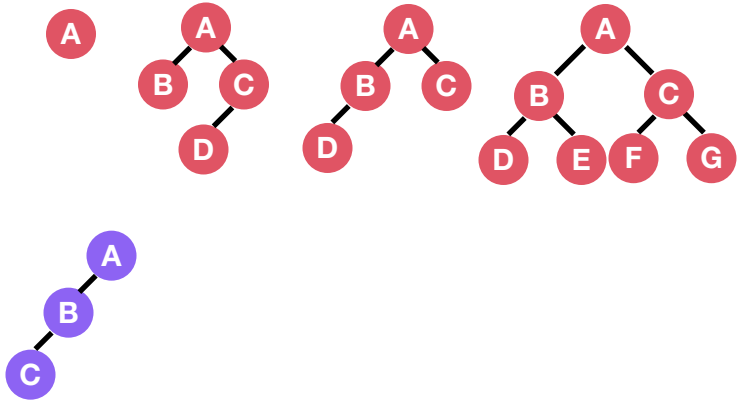
WHAT CAN A TREE LOOK LIKE?

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



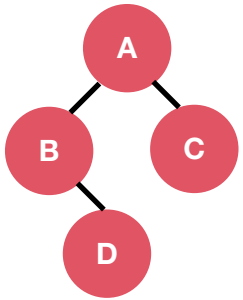
WHAT CAN A TREE LOOK LIKE?

- The appearance of a binary tree can look many different ways
- Nodes can also have duplicate values
- Realizing this is helpful when creating test cases!



REPRESENTING BINARY TREES

- Usually represented as a class
  - With properties: *value, left, and right*
  - This is usually how trees are represented in interview questions

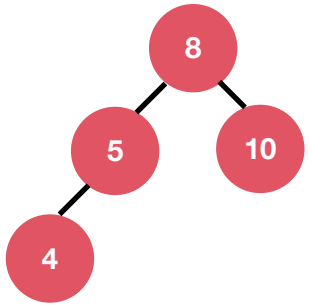


DIFFERENT PROPERTIES OF BINARY TREES

- Most common:
  - Balanced/Non-balanced
  - Binary Search Tree (BST)
- Not as common:
  - Full, Perfect, Complete, Degenerate etc.
- Binary Trees can have multiple properties
  - Balanced BST, Non-balanced BST, Complete BST etc.
- *Hint: Tree properties are usually hints! Always ask your interviewer if a tree has special properties*

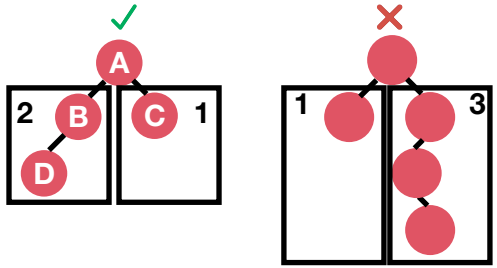
BINARY SEARCH TREES (BST)

- Left subtree contains children <= root
- Right subtree contains children > root
- Better lookup and insert performance than non-BSTs
  - *Caveat: only if they are balanced*
- *Hint: The properties of a BST are usually needed to get the optimal solution*



BALANCED BINARY TREES

- Height of left and right subtree of every node differs at most by 1
- Height = The max distance of any node from the root
- *Hint: Balanced/non-balanced trees impact the runtime complexity for trees, especially BSTs*



https://xlinux.nist.gov/dads/HTML/height.html

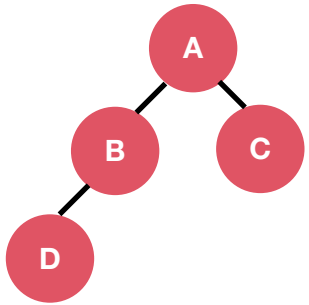
TIME/SPACE COMPLEXITY

- Time complexities usually differ depending on:
  - If the tree is a BST or not
  - If the tree is balanced/non-balanced
- **Remember your time and space complexities, interviewers will ask!**
- Always preface your assumptions
  - *"If this BST was balanced/non-balanced, the runtime would be..."*

	Non-balanced BST	Balanced BST
Get	O(n)	O(logn)
Insert	O(n)	O(logn)
Remove	O(n)	O(logn)

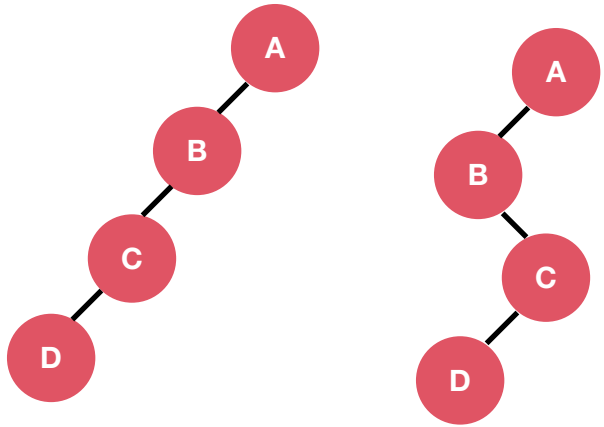
COMPLETE BINARY TREES

- Every level is filled, except the last level
- Last level has all nodes as far left as possible
- A complete binary tree is also balanced



DEGENERATE BINARY TREES

- Also called a *pathological* tree
- Every node has at most 1 child
- This is usually why runtime/space complexity is O(n) for trees (especially BSTs)

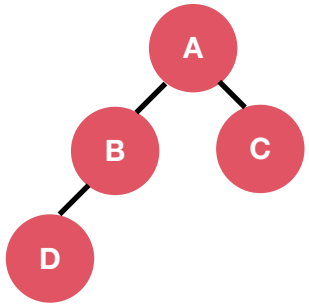


TREE TRAVERSALS

- Binary Tree problems require you to traverse the tree and manipulate/get values from it
- Two ways to traverse:
  - Depth-first search (DFS)
    - In-order, Pre-order, Post-order
  - Breadth-first search (BFS)
    - Level-order
- *Hint: Doing these traversals should be muscle memory!*

DEPTH-FIRST SEARCH (DFS)

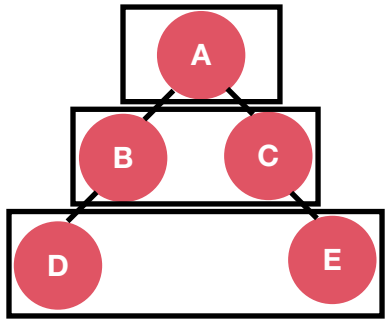
- Can be implemented recursively or iteratively
- In-order (left, current, right)
  - *Hint: In-order = Left to right*
  - *This traversal in a BST gives you the sorted order*
  - Reverse In-order (right, current, left)
- Pre-order (current, left, right)
  - *Hint: Pre = Before*
  - Reverse Pre-order (current, right, left)
- Post-order (left, right, current)
  - *Hint: Post = After*
  - Reverse Post-order (right, left, current)
- *Keywords: max, deepest, longest*



In-order: [D, B, A, C]
Pre-order: [A, B, D, C]
Post-order: [D, B, C, A]

BREADTH-FIRST SEARCH (BFS)

- Traverse the tree in a level-order fashion
- Can be implemented using a queue
- *Keywords: level, row, closest, minimum, width, diameter*



Level-order: [A, B, C, D, E]

BINARY TREES AND UPER

- Understand
  - Does the tree have any special properties?
  - What are some cases I need to watch out for?
- Plan
  - What type of traversal should I do?
  - Should I use recursion/iterative? What's my base/recursive case? What should I do w/ the nodes/subtrees?

PROBLEM WALKTHROUGH: MIN-DEPTH

- Find the minimum depth of a binary tree
- Leetcode Link

Min Depth Demo