Revised: 1/3/2020

Trees I

Michael C. Hackett
Computer Science Department

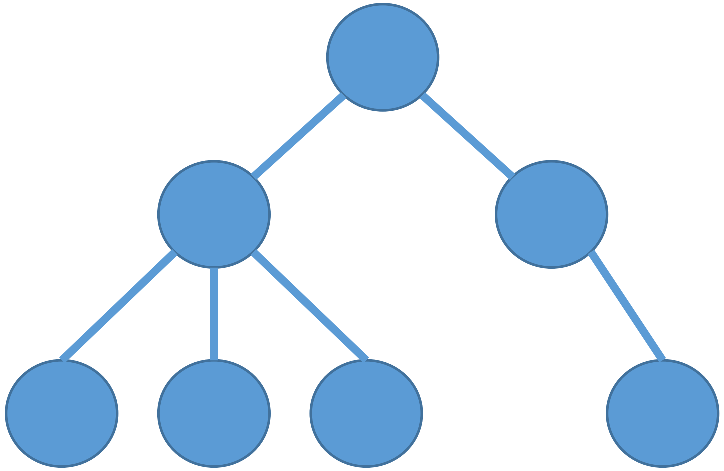
Community College of Philadelphia

Lecture Topics

- Tree Terminology
- Binary Trees
  - Tree Traversals
- Binary Search Trees
- General/N-ary Trees
- Complexity of Trees
- Other Tree Classifications
  - Balanced Binary Trees
  - Full Binary Trees
  - Complete Binary Trees
  - Perfect Binary Trees
- Tree Structure Complexities

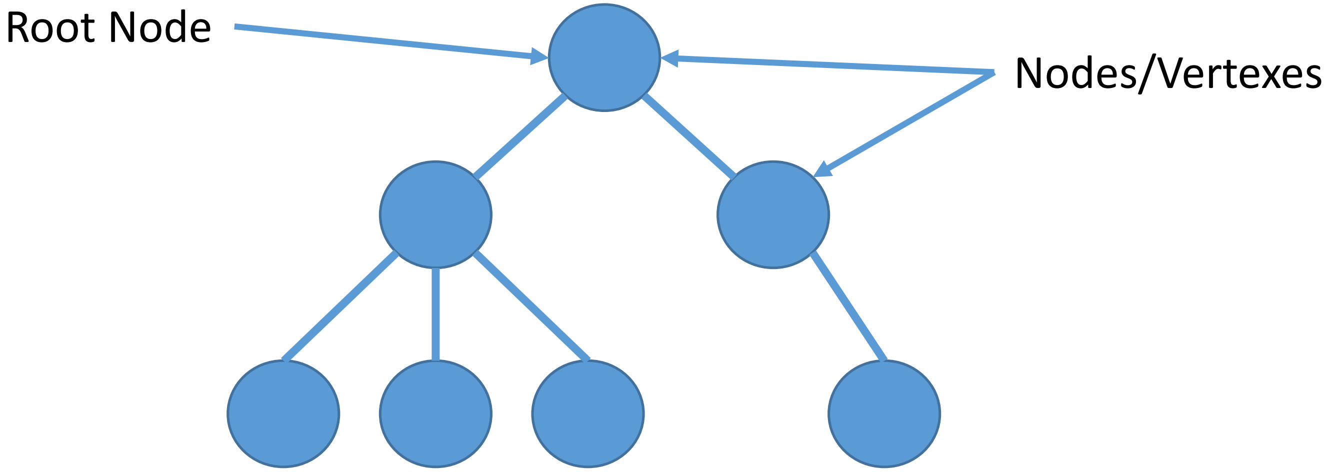
Trees

- A **tree** is a non-linear data structure, where each point in the tree will branch into zero or more points.



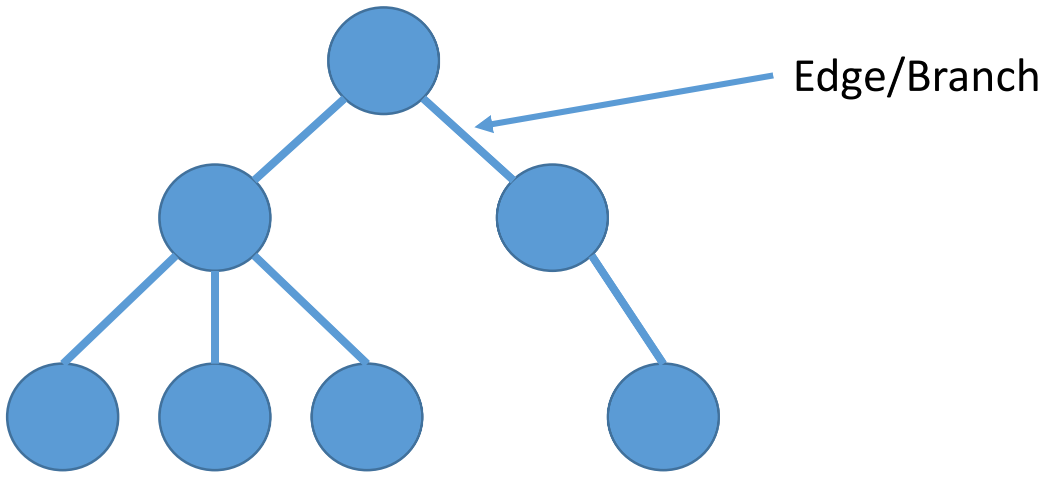
Trees

- Each point in the tree is called a **node** or **vertex**
- The top-most node is called the **root** node



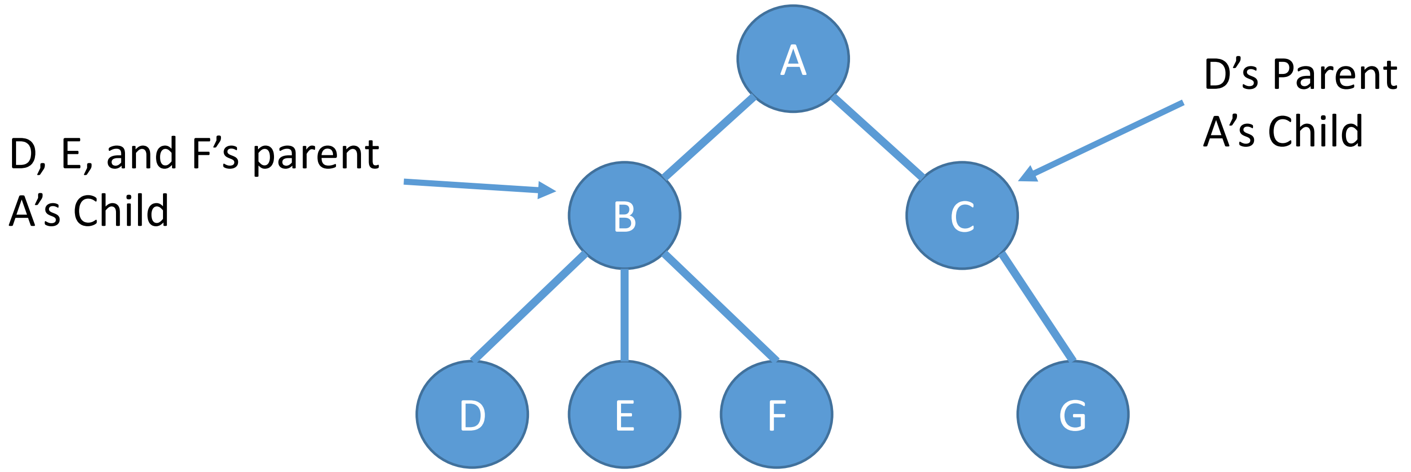
Trees

- The lines connecting the nodes are **edges** or **branches**



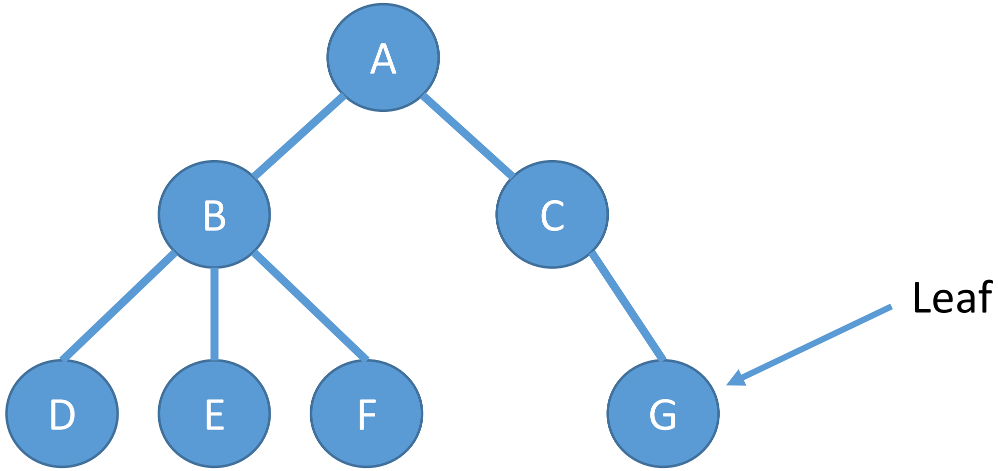
Trees

- **Children** or **child nodes** are nodes that branch from a higher node.
- A node's **parent** is the node it branches from.
  - <u>The root node will not have a parent</u>.



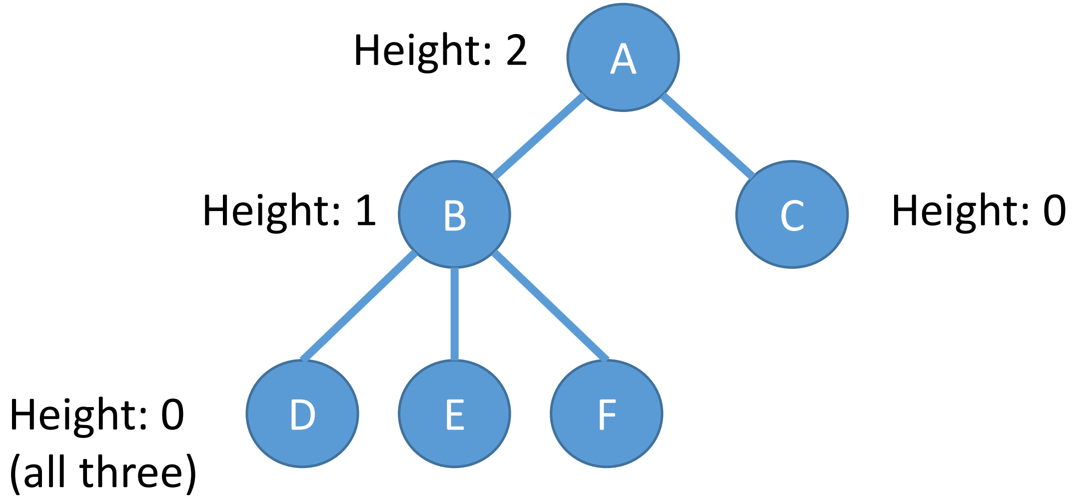
Trees

- A node with no children is called a **leaf** or **leaf node**



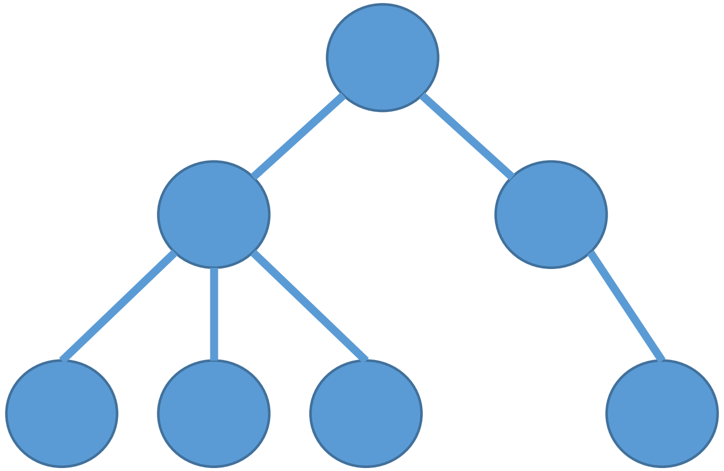
Trees

- A node's **height** is its distance to the farthest leaf.

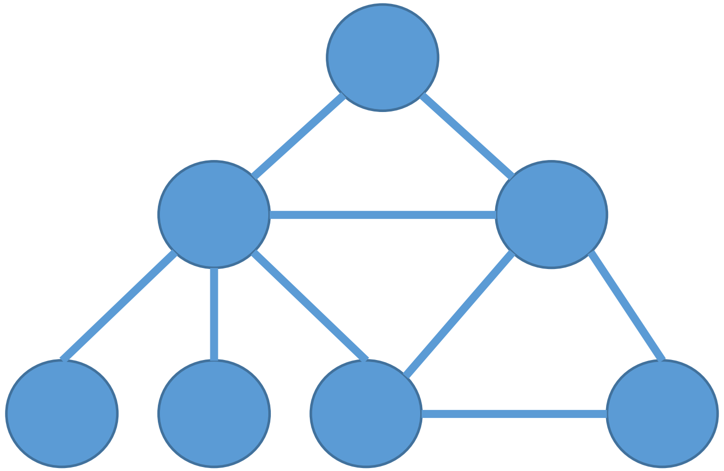


Trees

- Major Characteristic:
  - Only **one** path from the root to any node in the tree



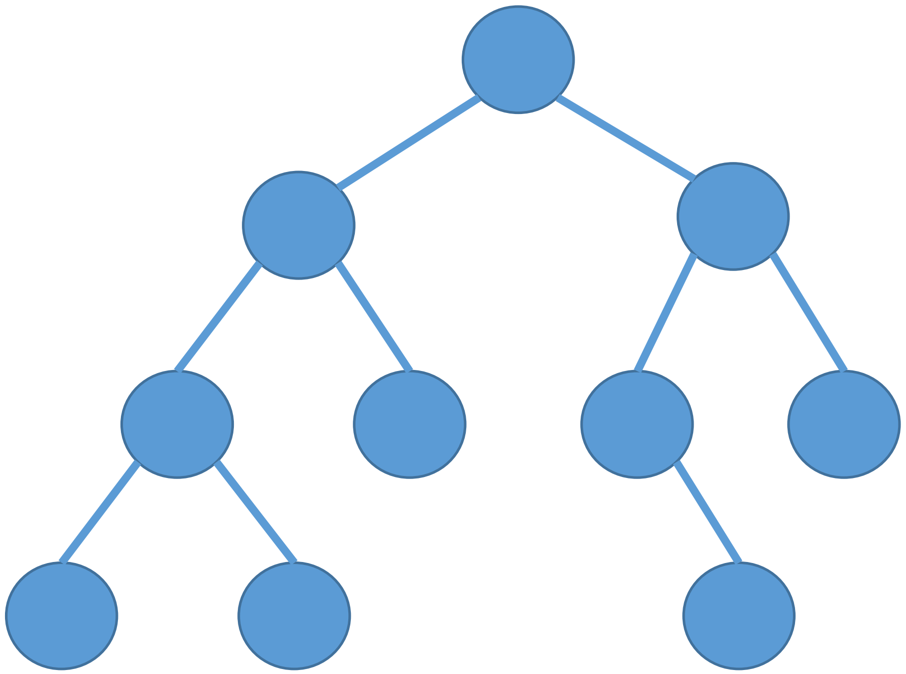
Tree



Not a Tree

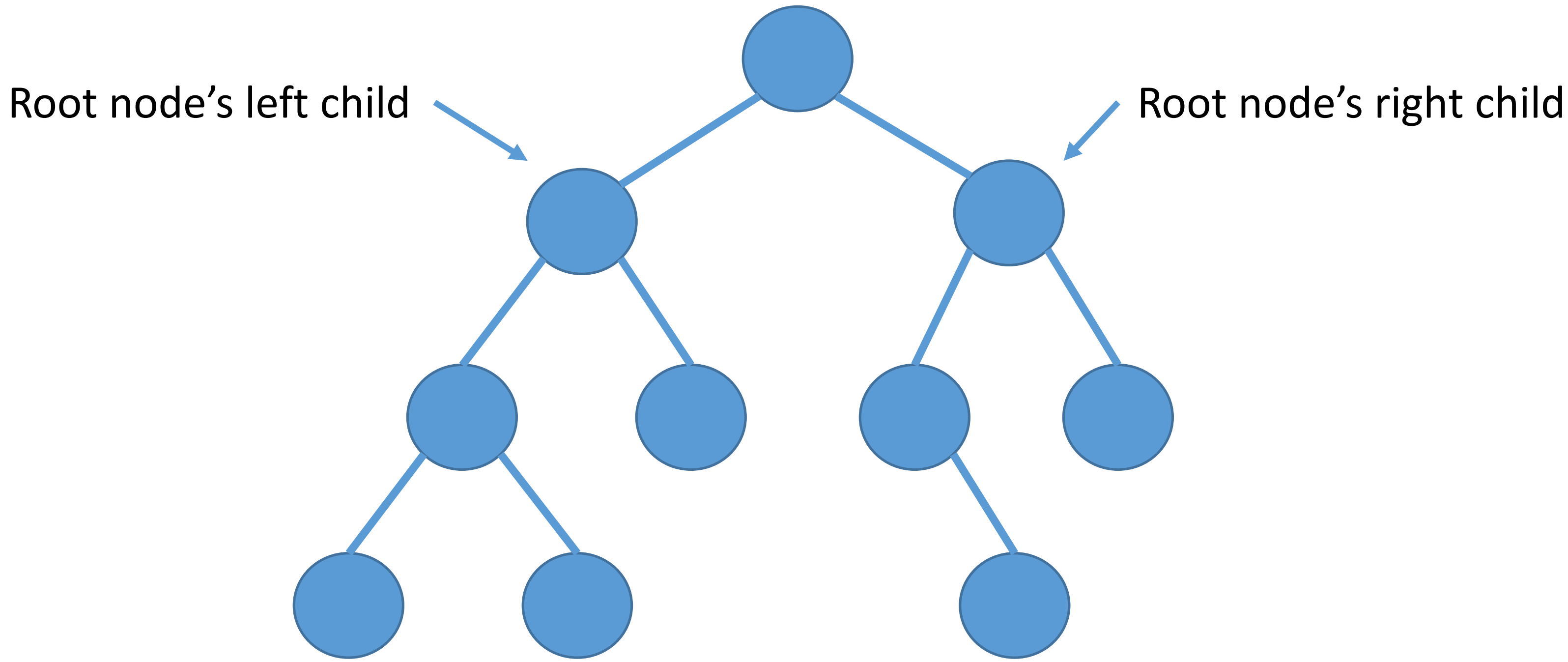
Binary Trees

- While trees can be built without limits for the number of children a node may have, the **binary tree** only allows <u>up to two</u> children for each node.



Binary Trees

- The children of a node are often referred to as the left child and right child

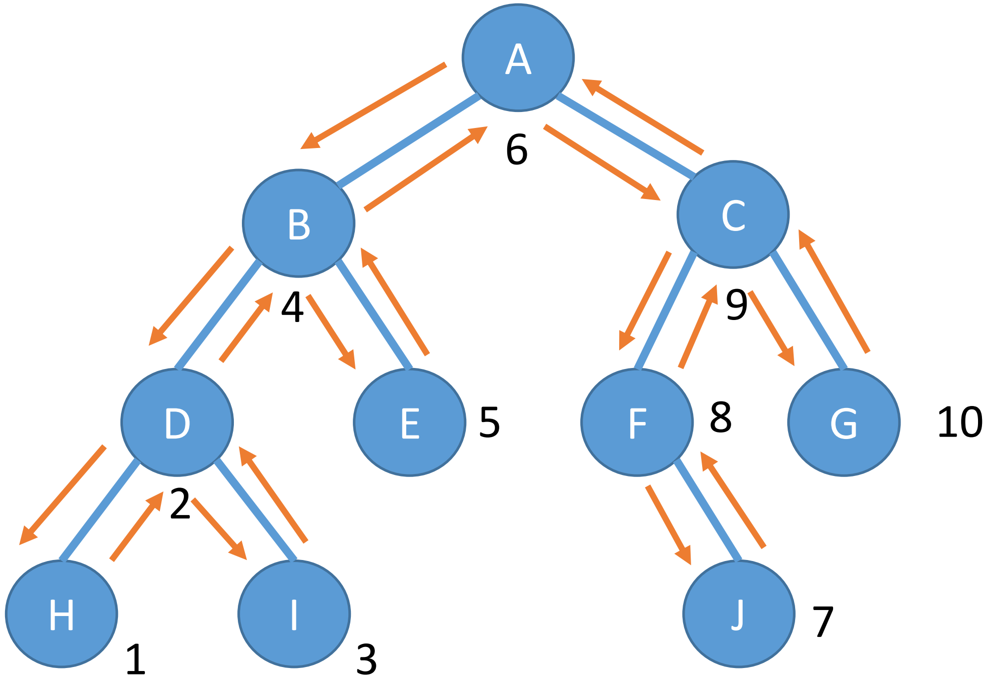


Tree Traversals

- In-Order Traversal
  - Traverse down the left side
  - Use the node’s value/data
  - Traverse down the right side
  - In other words, the value of the node is used upon the **second** time it is visited
- Pre-Order Traversal
  - Use the node’s value/data
  - Traverse down the left side
  - Traverse down the right side
  - In other words, the value of the node is used upon the **first** time it is visited.
- Post-Order Traversal
  - Traverse down the left side
  - Traverse down the right side
  - Use the node’s value/data
  - In other words, the value of the node is used upon the **last** time it is visited.

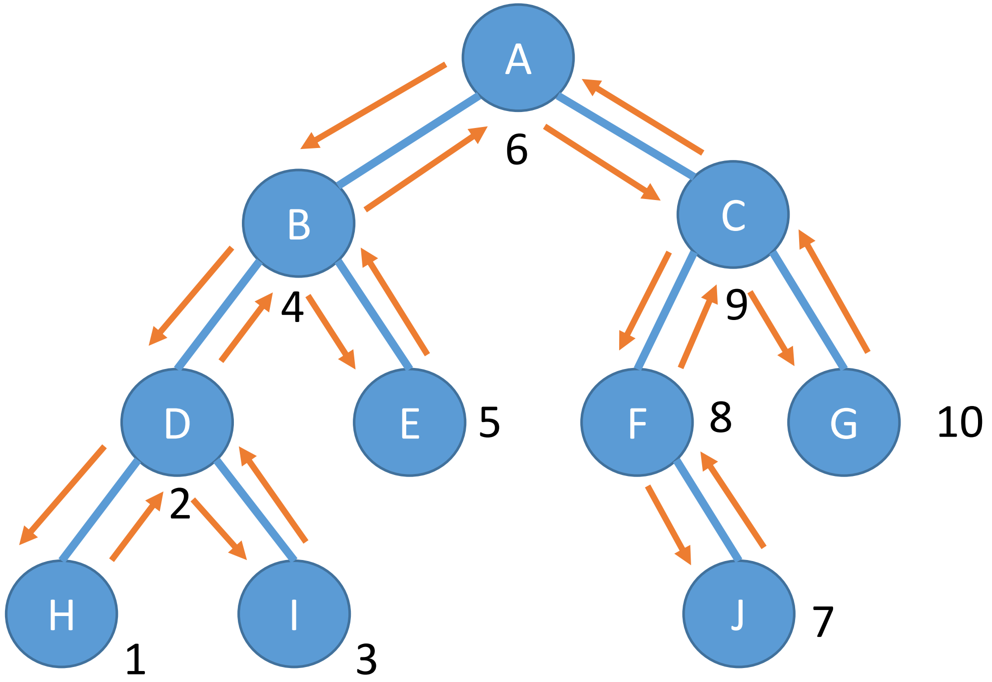
In-Order Traversal

- Arrows show the direction of the traversal.
- Numbers indicate when the values of the nodes are used in the traversal.



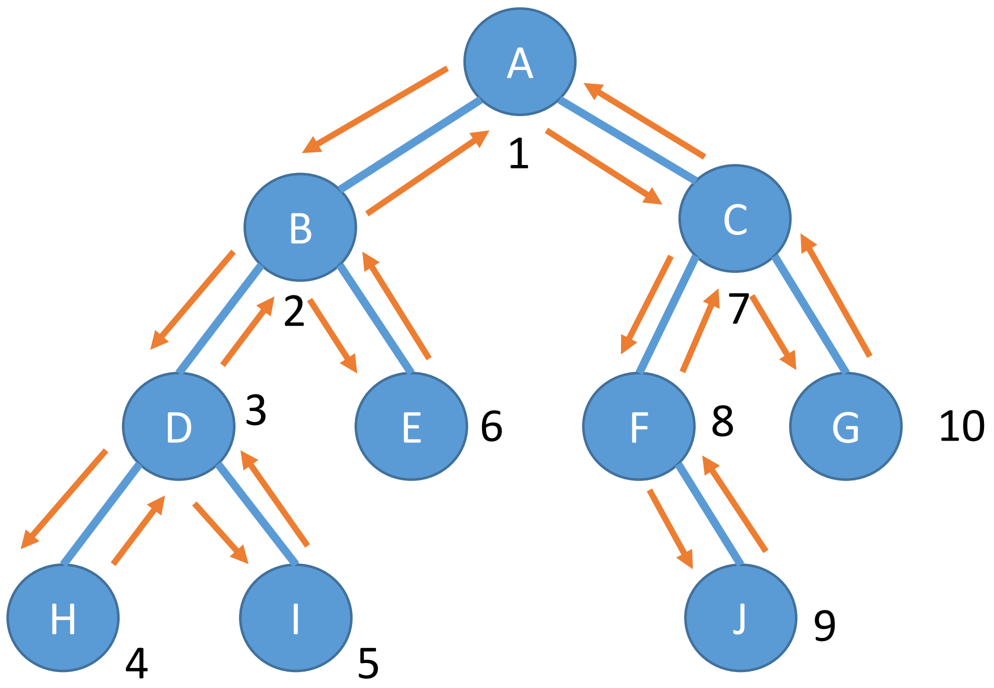
In-Order Traversal

- **Infix Format**: H D I B E A J F C G



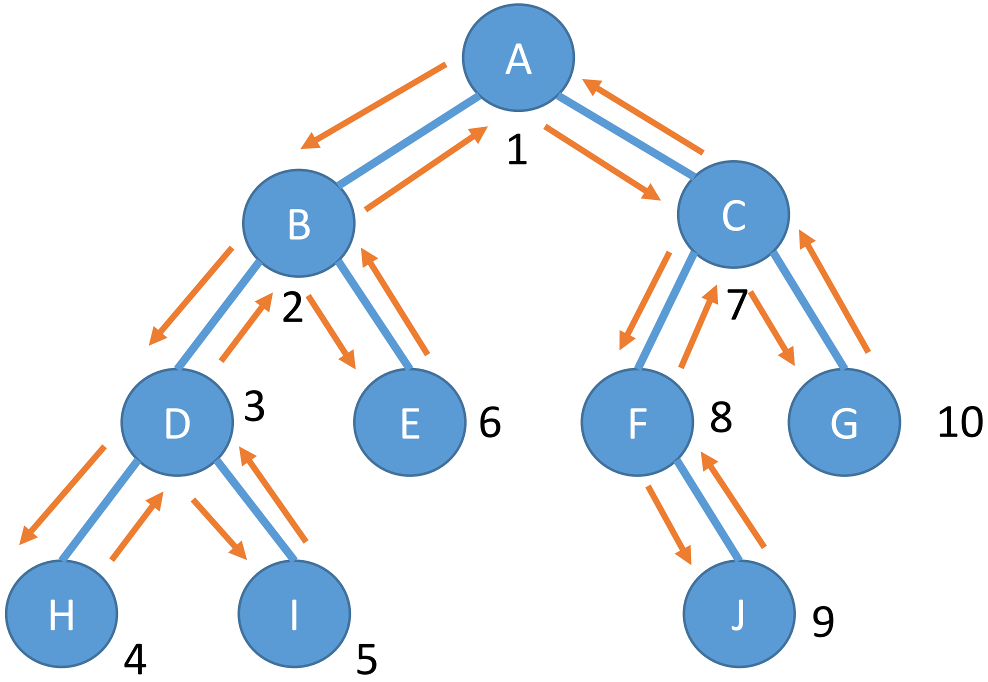
Pre-Order Traversal

- Arrows show the direction of the traversal.
- Numbers indicate when the values of the nodes are used in the traversal.



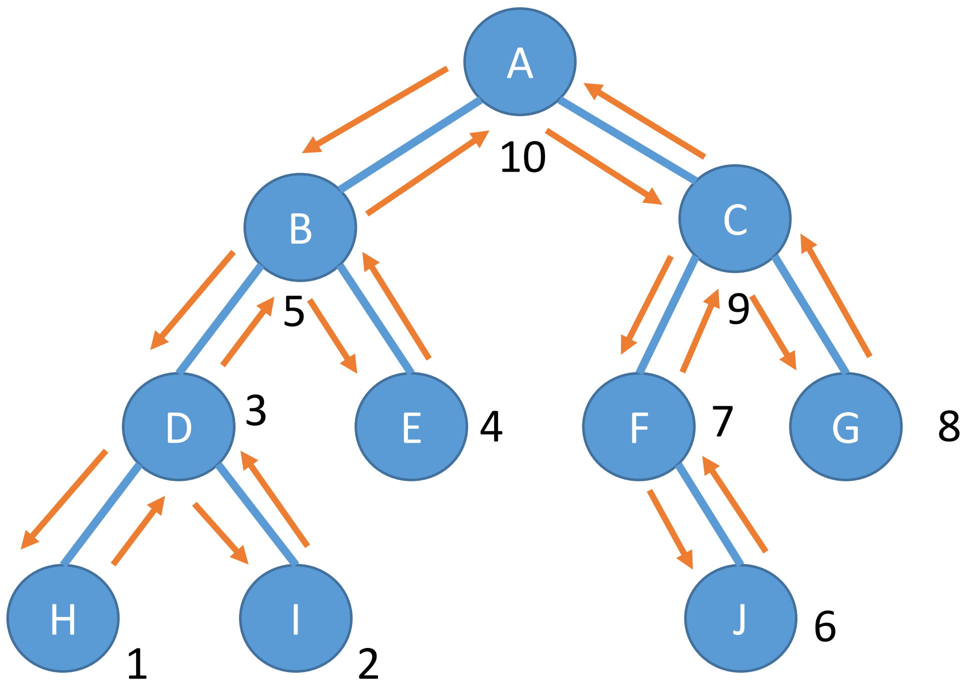
Pre-Order Traversal

- **Prefix Format**: A B D H I E C F J G



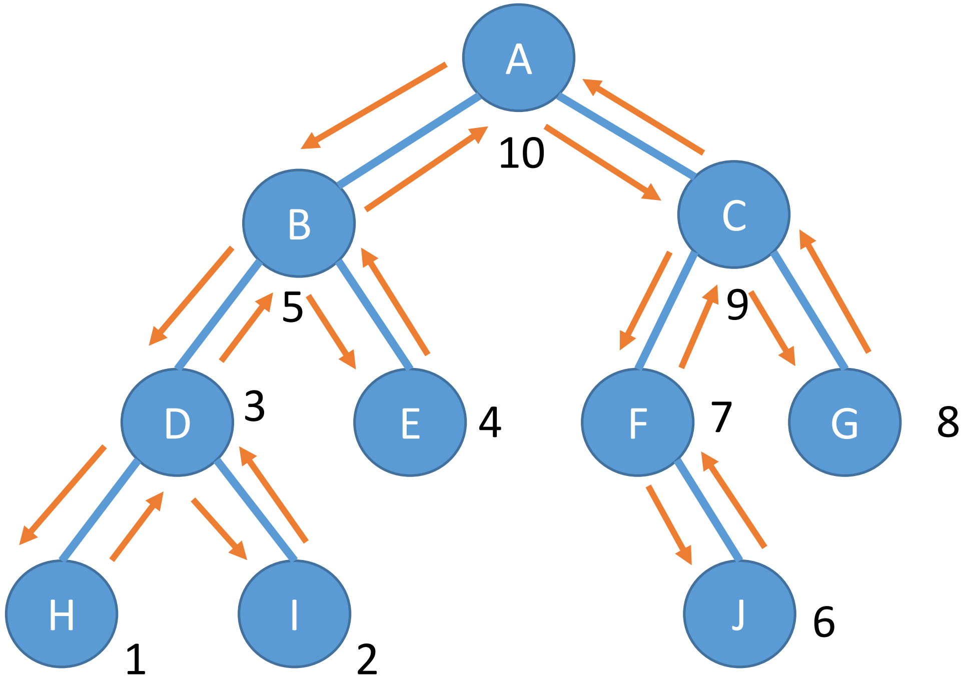
Post-Order Traversal

- Arrows show the direction of the traversal.
- Numbers indicate when the values of the nodes are used in the traversal.



Post-Order Traversal

- **Postfix Format**: H I D E B J F G C A



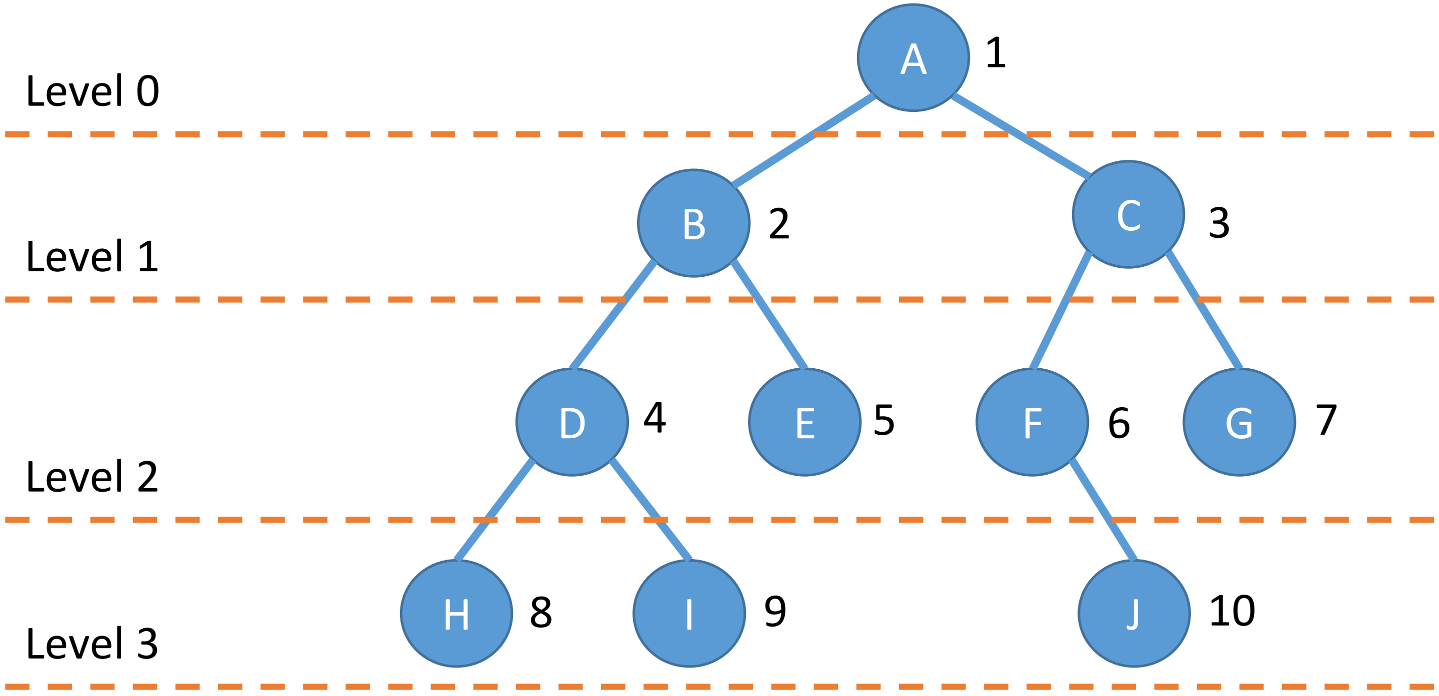
Depth-First Traversal

- In-Order, Pre-Order, and Post-Order are all examples of a **depth-first traversal**.

- The traversal algorithms always start by going to the lowest point on the left side.
  - Regardless of when each node’s data/value is used.
  - Looking at the previous examples, the path taken is always the same.

Breadth-First Traversal

- Using a **breath-first** or **level-order traversal**, the tree is traversed by visiting all nodes at each level of the tree, working its way to the bottom.



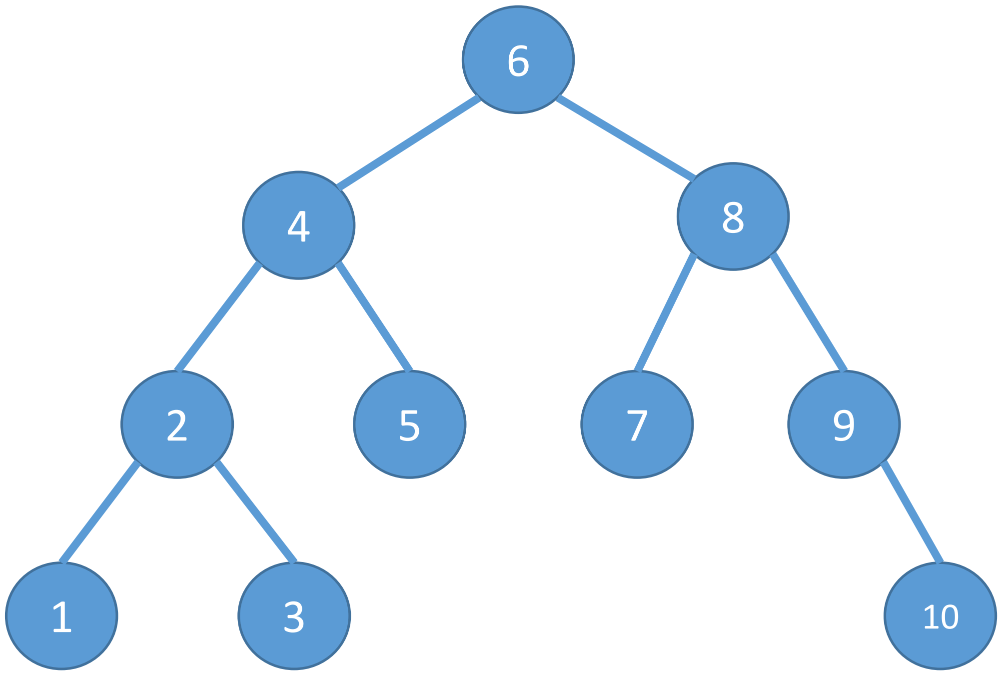
Binary Search Trees

- A **binary search tree** (or **BST**) is a binary tree where the node are comparatively added to preserve the natural ordering of the values stored in the tree's nodes.

- For each node:
  - Its left child node's value will be less than the node's data
    - As will all of its children
  - Its right child node's value will be greater than the node's data
    - As will all of its children

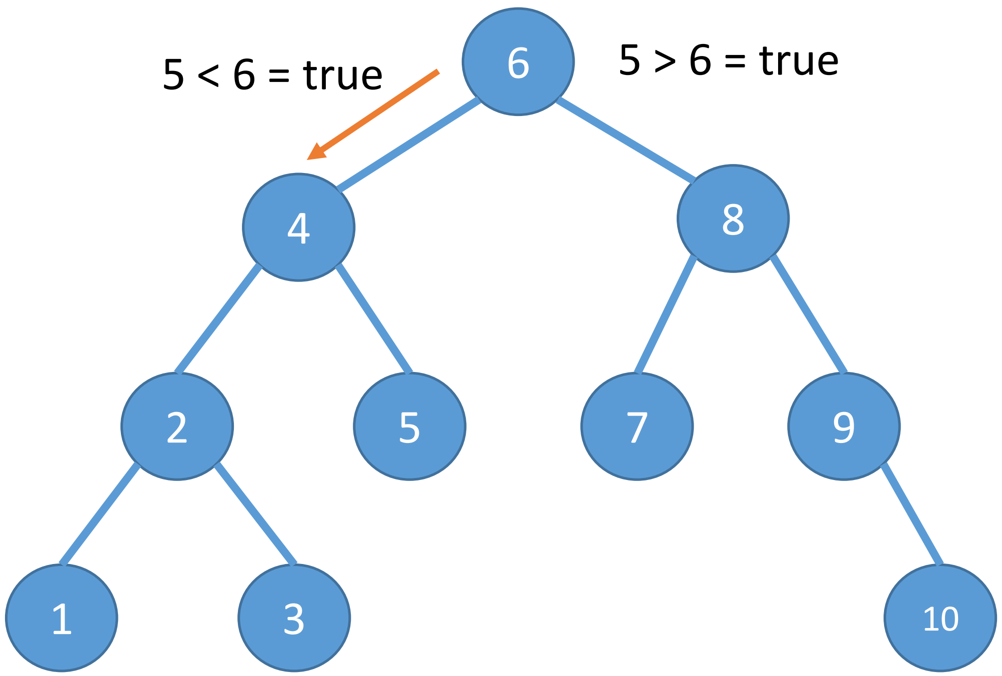
Binary Search Trees

- Left child's value is less than the parent's value
- Right child's value is greater than the parent's value



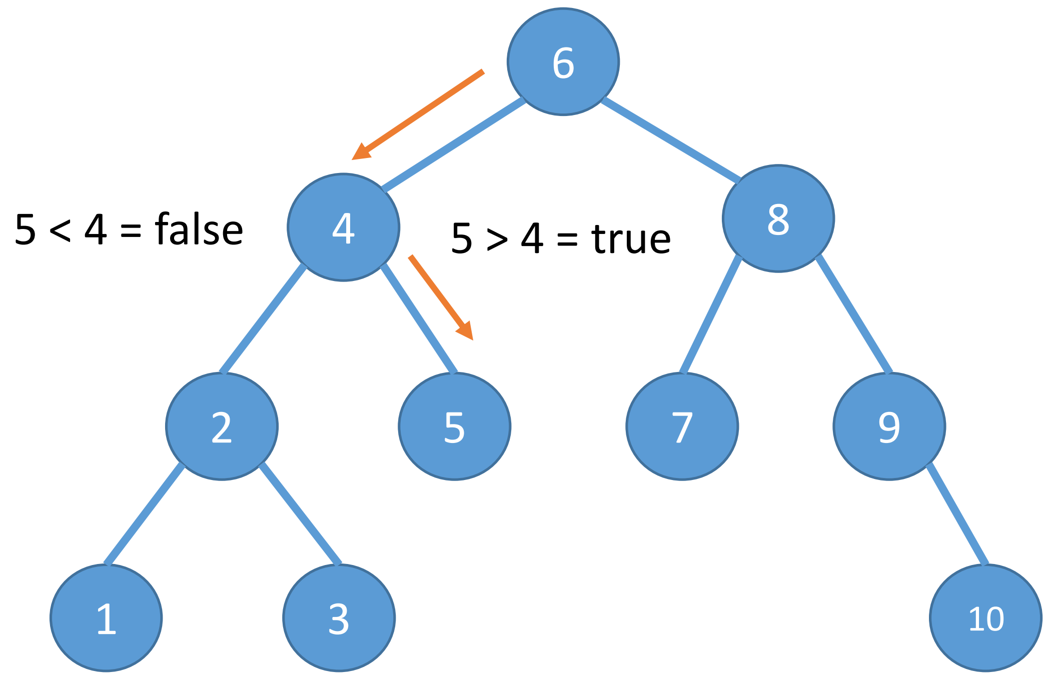
Binary Search Trees

- To search the tree for a value, simple > or < comparisons are used
  - Searching for 5



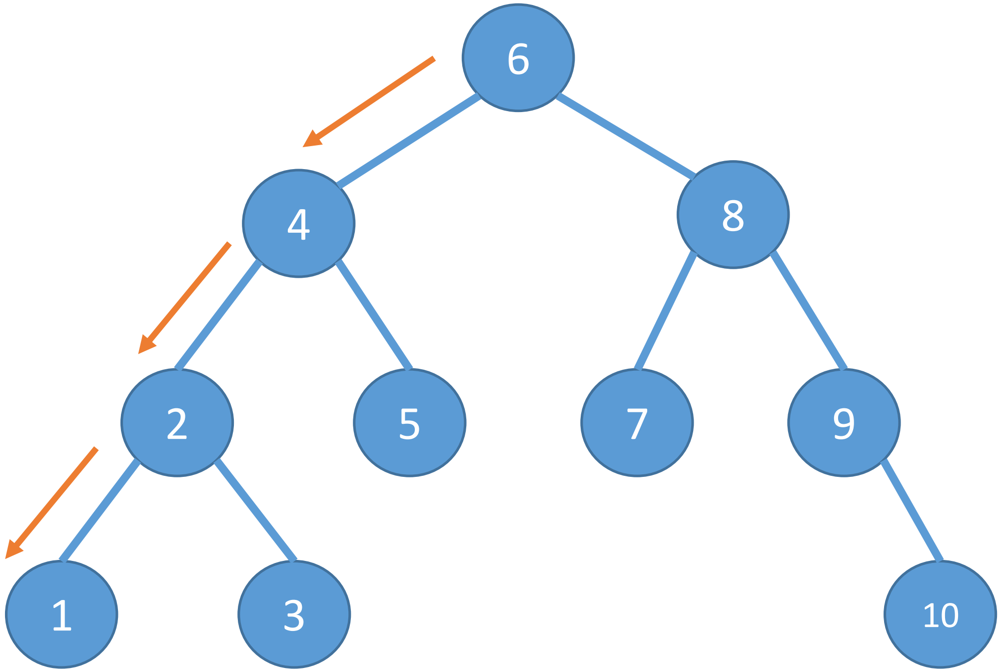
Binary Search Trees

- To search the tree for a value, simple > or < comparisons are used
  - Searching for 5
- A similar process is used for adding new nodes to a BST



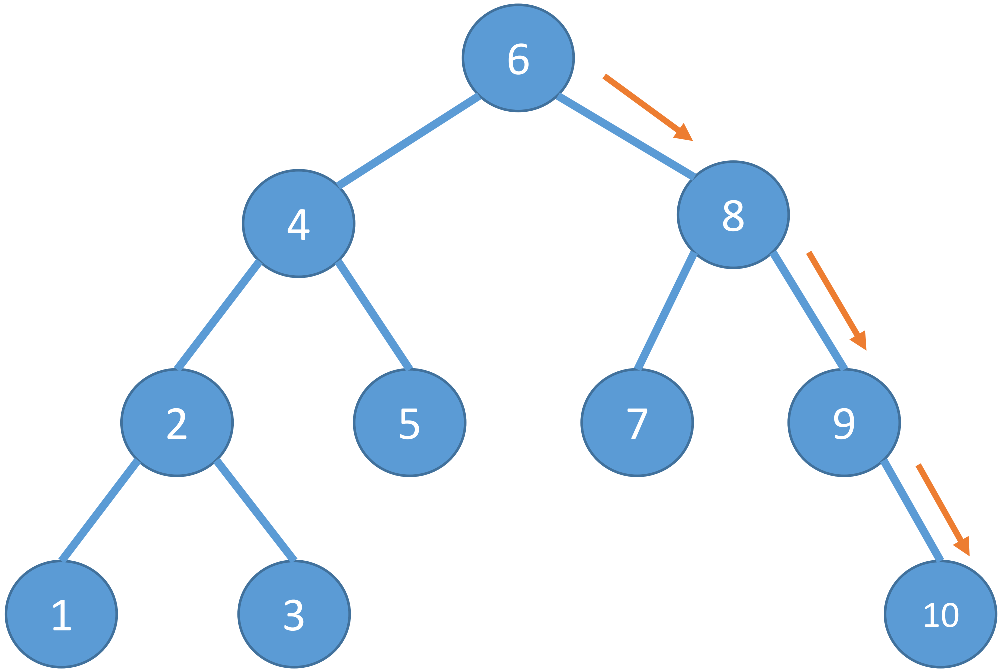
Binary Search Trees

- The smallest (min) value in a BST is always the left-most leaf.



Binary Search Trees

- The largest (max) value in a BST is always the right-most leaf.



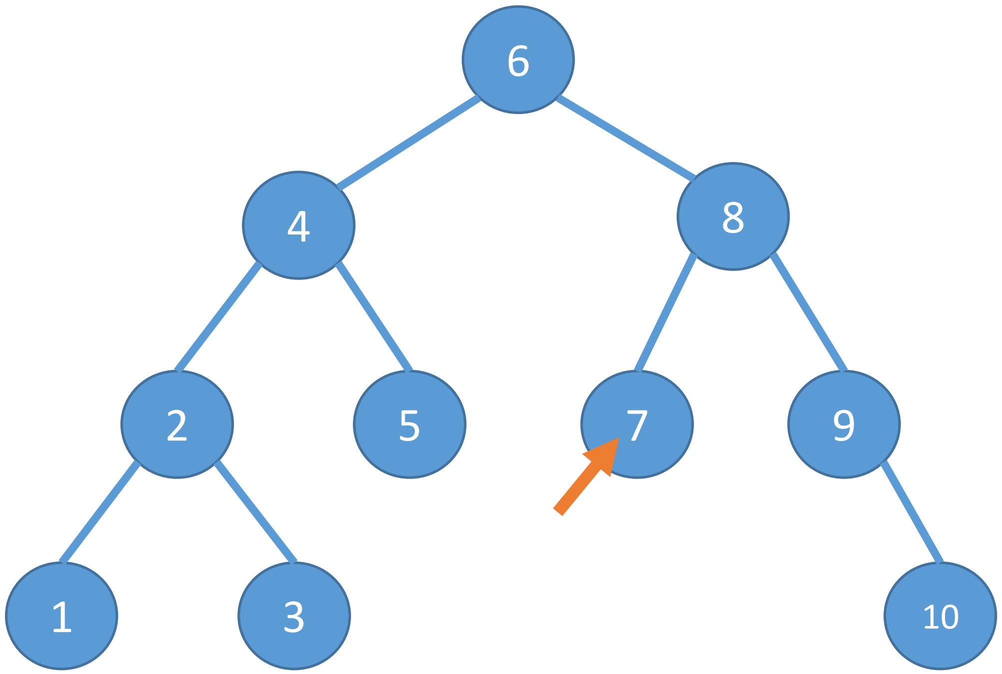
Binary Search Trees

- To remove a node, we need to determine its **successor**- the node that will replace it.

- If the node to remove..
  - Has no children – Safe to remove
  - Has only a right child – The right child is the successor
  - Has only a left child – The left child is the successor
  - Has both a right and left child – The smallest value down the right side of the node is the successor.

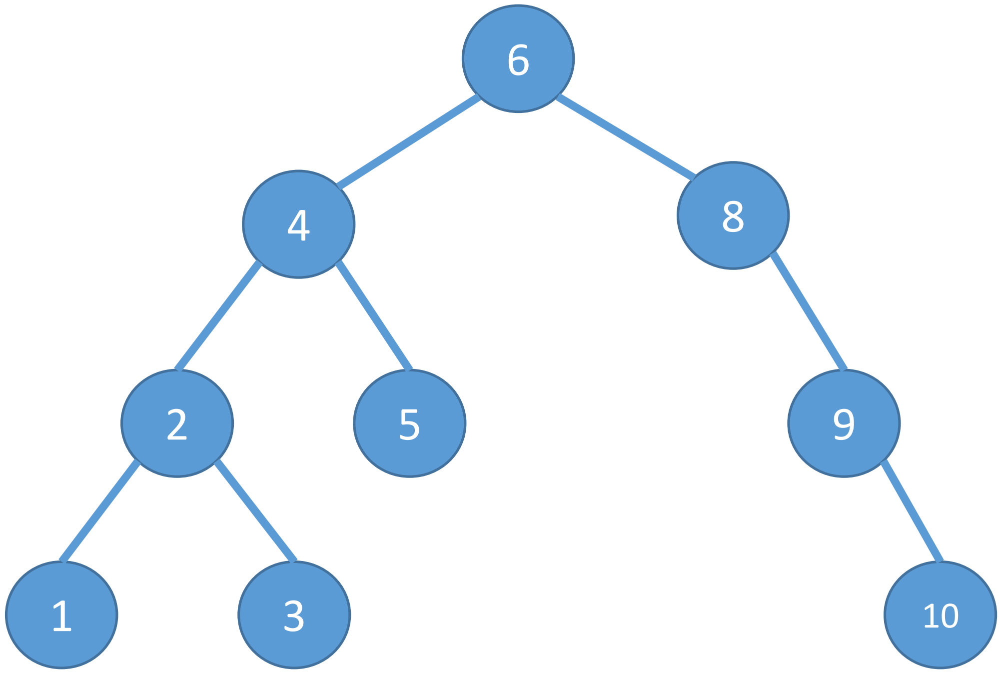
Binary Search Trees

- Removing the node containing 7…



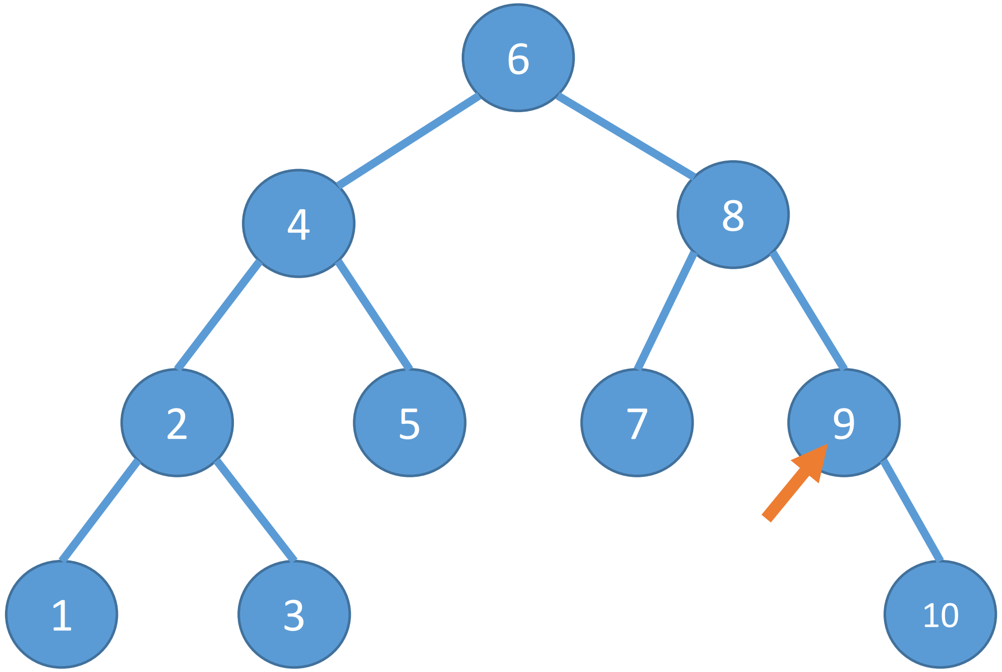
Binary Search Trees

- Removing the node containing 7… No successor



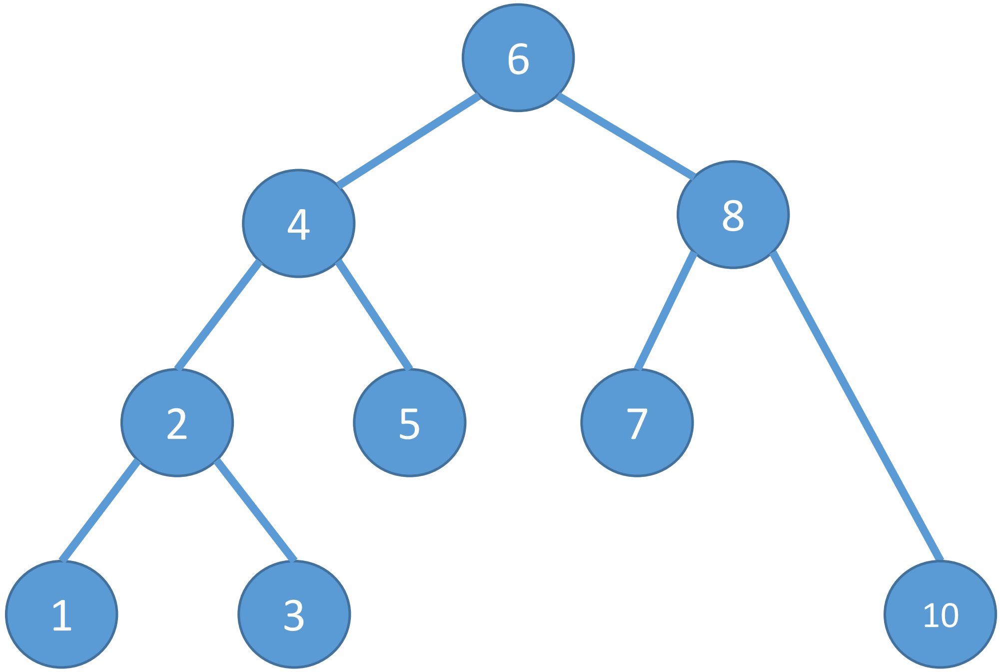
Binary Search Trees

- Removing the node containing 9…



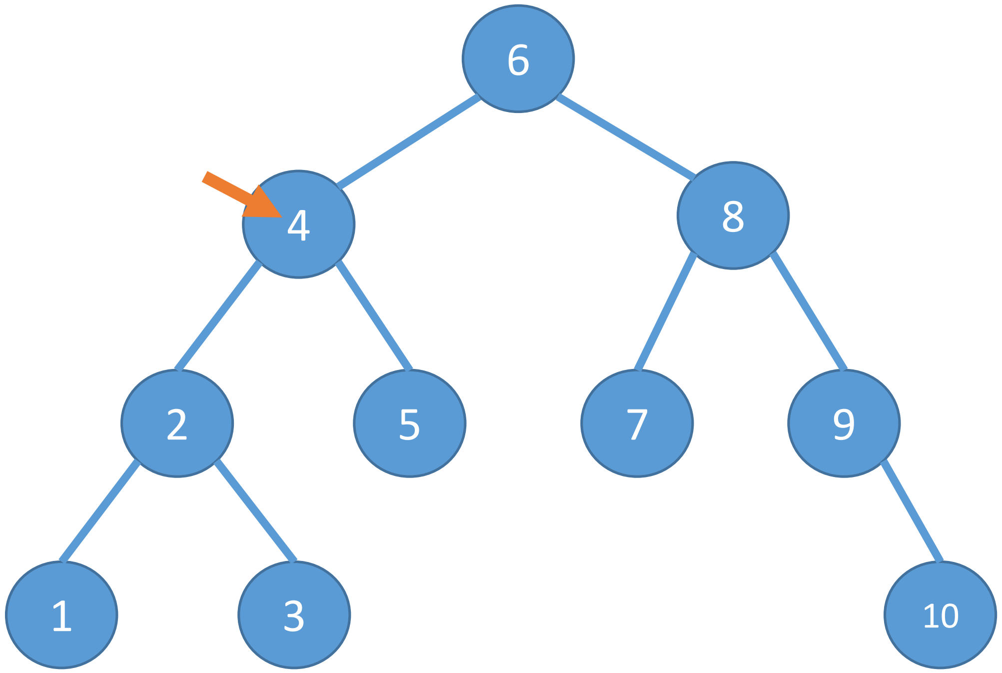
Binary Search Trees

- Removing the node containing 9… (didn't have a left child) the node containing 10 is its successor.



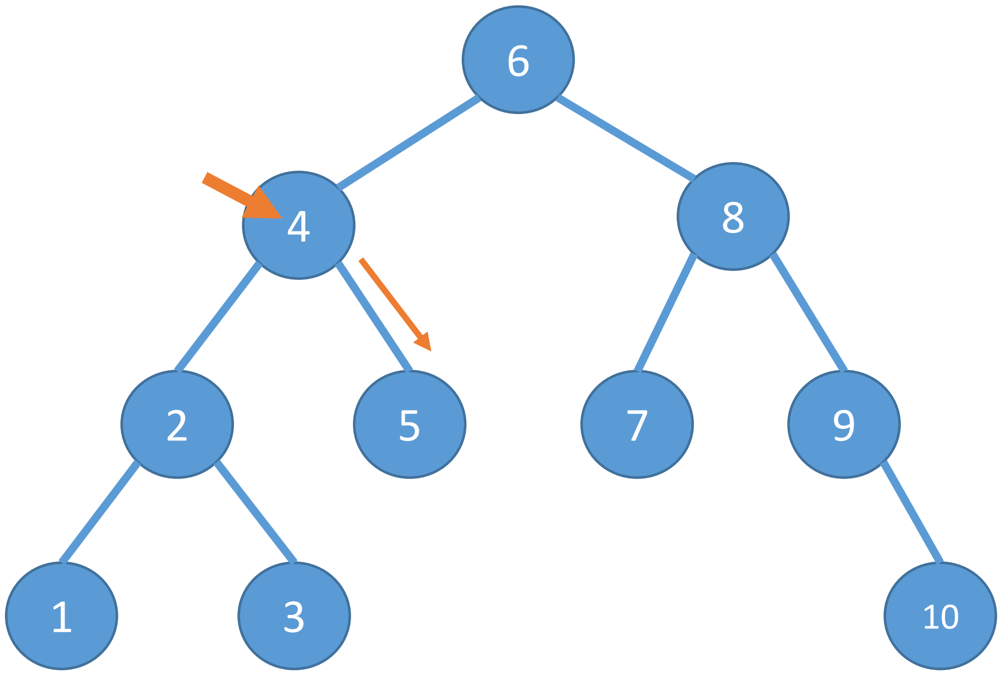
Binary Search Trees

- Removing the node containing 4…



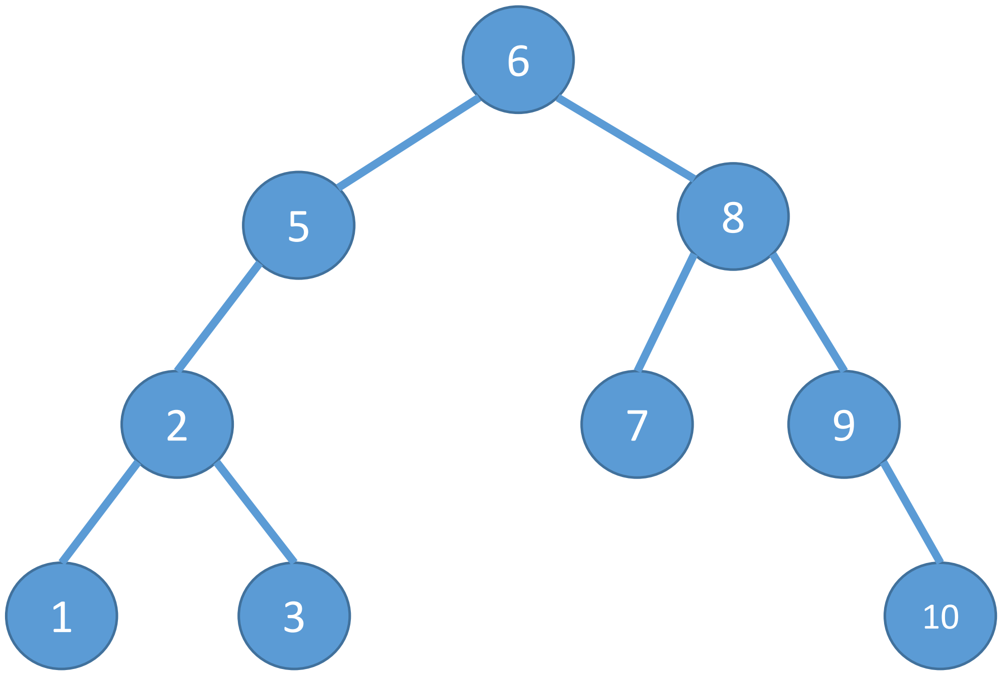
Binary Search Trees

- Removing the node containing 4… Goes down its right side looking for the smallest value (only one node to check)…



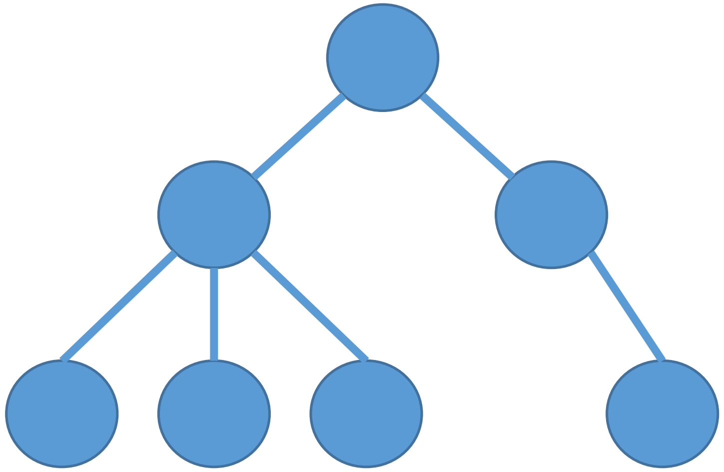
Binary Search Trees

- Removing the node containing 4… Goes down its right side looking for the smallest value (only one node to check)… 5 is the smallest, so that is its successor.



N-ary Trees

- An **n-ary tree** (or **general tree**) is a tree where each node may have any number of children.
  - The first tree shown at the beginning of the lecture was such a tree.



N-ary Trees

- Since we don't know how many children each node has, it won't have left or right children like a binary tree.
- Instead, each node maintains a list structure of its children.

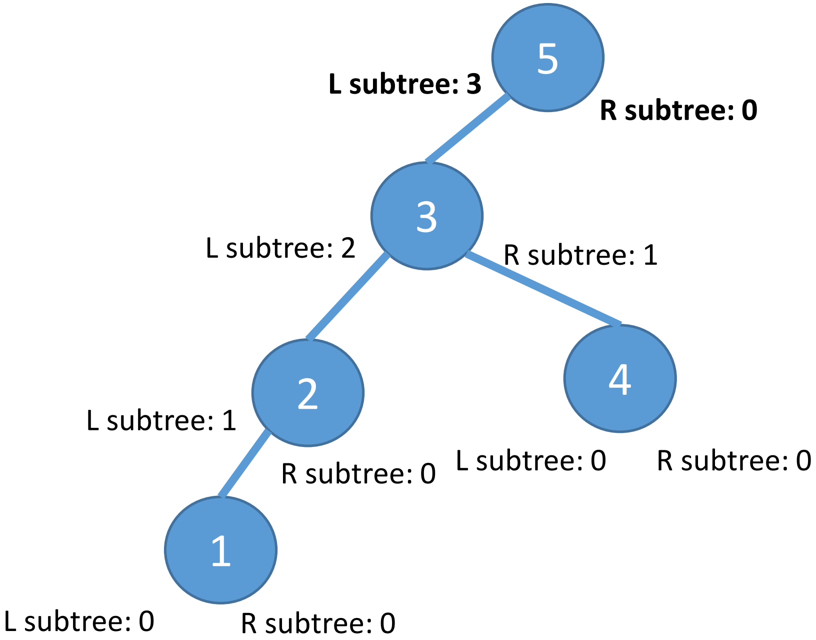
Tree Complexity

- Traversing any tree will have O(n) complexity as each node needs to be visited one.
  - Just as traversing and array or linked list.

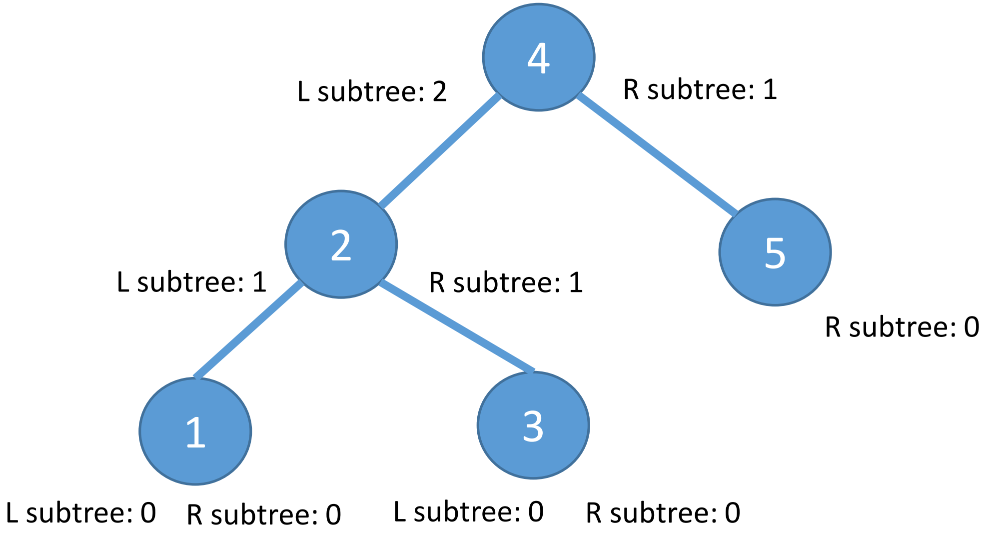
- How the tree is structured will have an impact on insert, removal, and search.
  - A binary tree is **balanced** when every node's left subtree and right subtree differ by, at most, one level

Tree Complexity

Every node's left subtree and right subtree must differ by, at most, 1 for a balanced tree



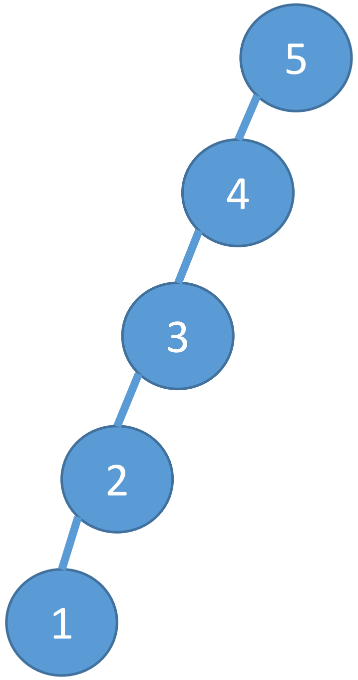
Unbalanced BST



Balanced BST

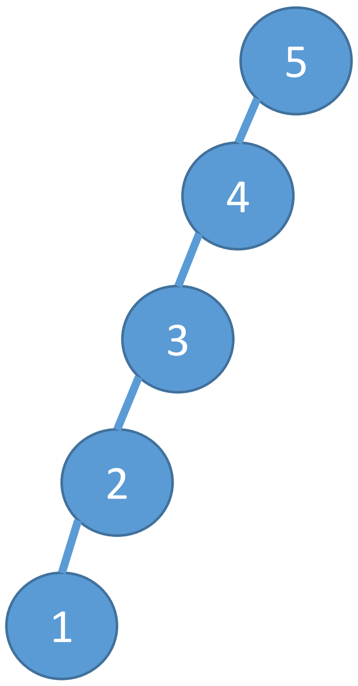
Tree Complexity

- Balanced trees ensure the best performance and efficiency.
- Consider this (valid) BST:
  - This example, where each node has at most one child, is called a *pathological* or *degenerate tree*.
  - Clearly unbalanced.



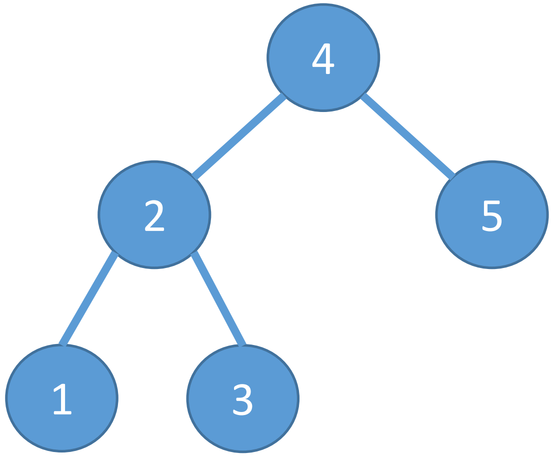
Tree Complexity

- The smallest value in the tree will be the final leaf.
- To get to it, we need to visit every node in the tree.
  - O(n)
- Same for searching for a value.
- Essentially, this tree would behave like a linked list (a linear data structure)



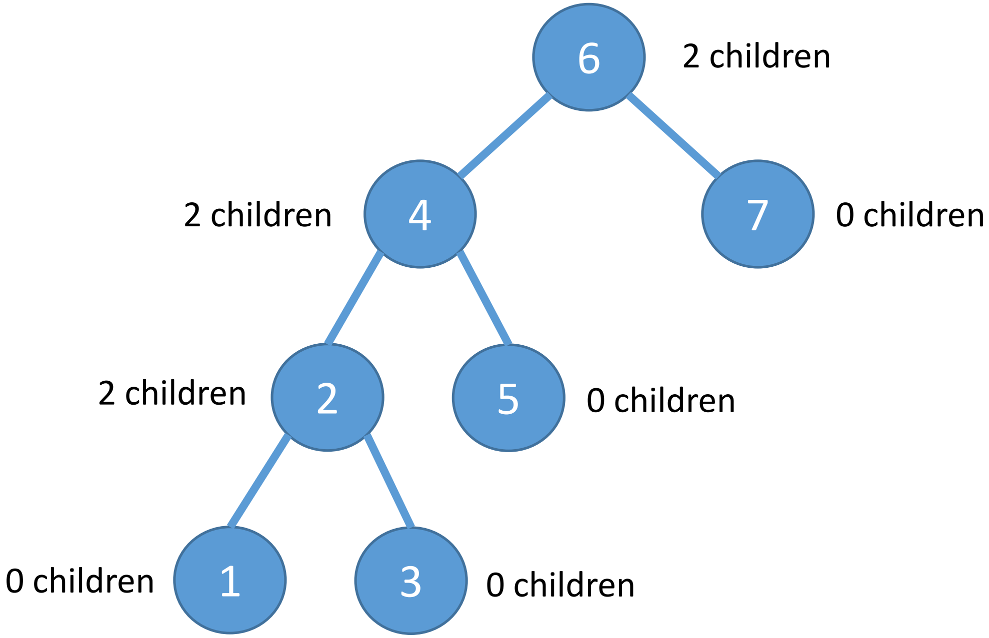
Tree Complexity

- Here is the same tree (same nodes) but balanced a little better.
- The complexity of any operation on this balanced tree is O(h)
  - Where h is the height of the tree.

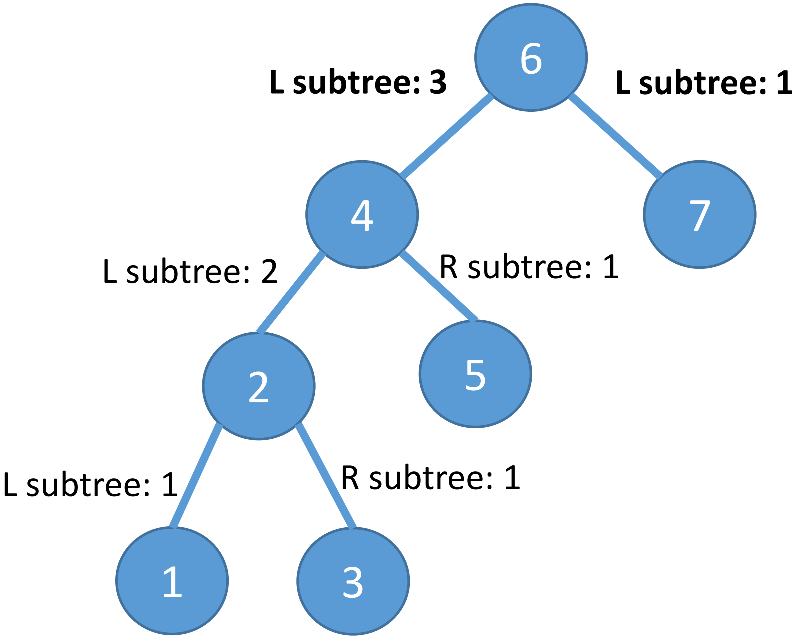


Full Binary Trees

- A **full binary tree** is when every node has either 0 or 2 children
  - The tree below is full, but not balanced



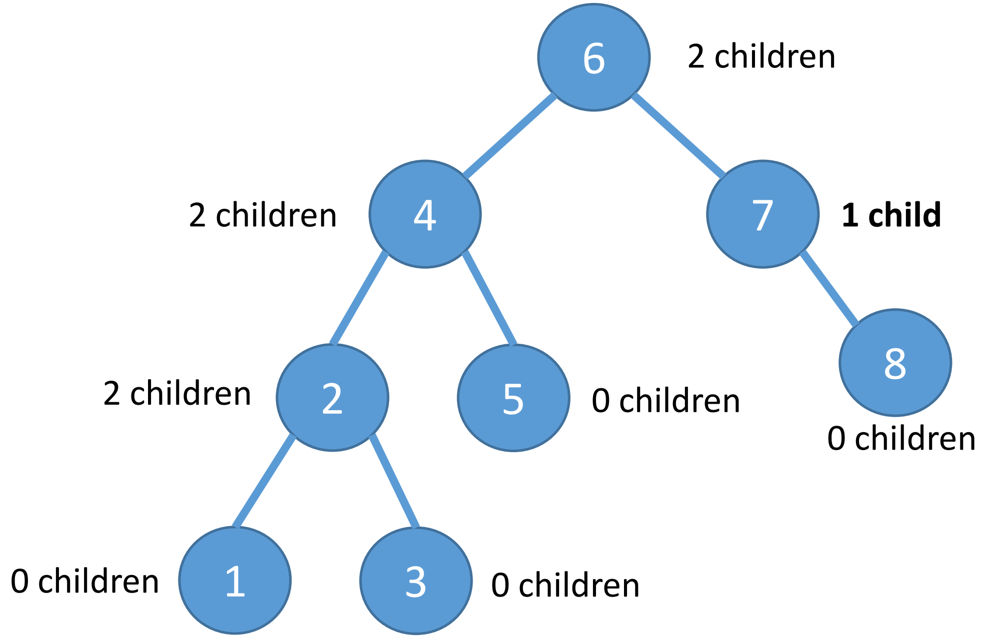
Full



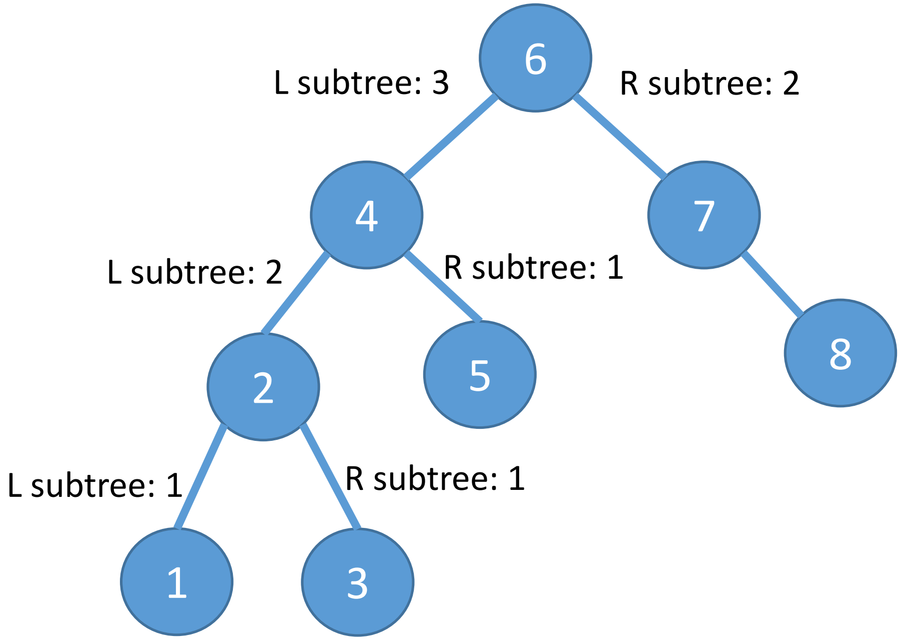
Unbalanced

Full Binary Trees

- An example of a BST that is balanced, but not full



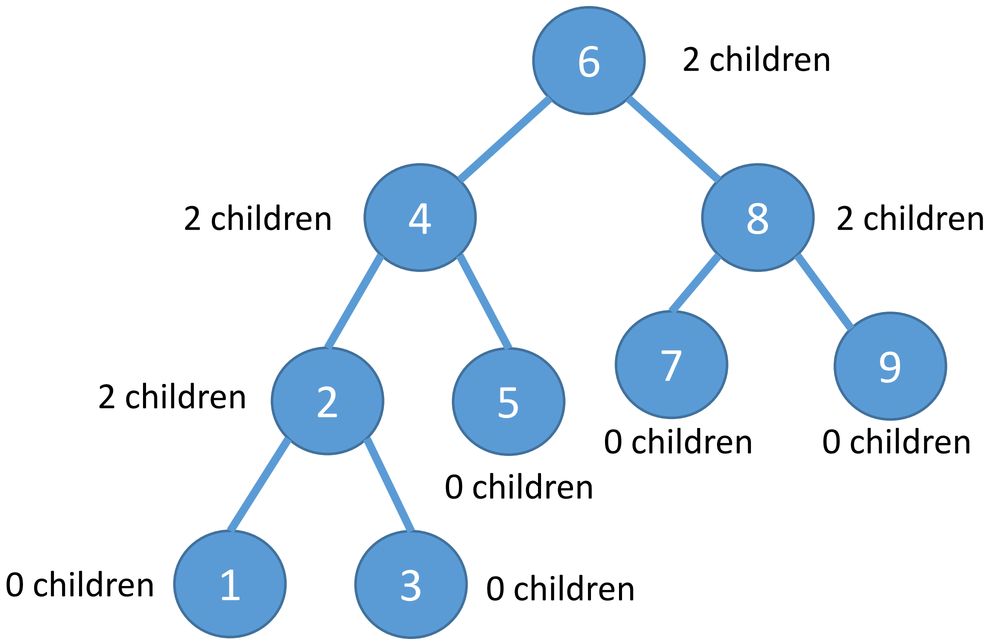
Not Full



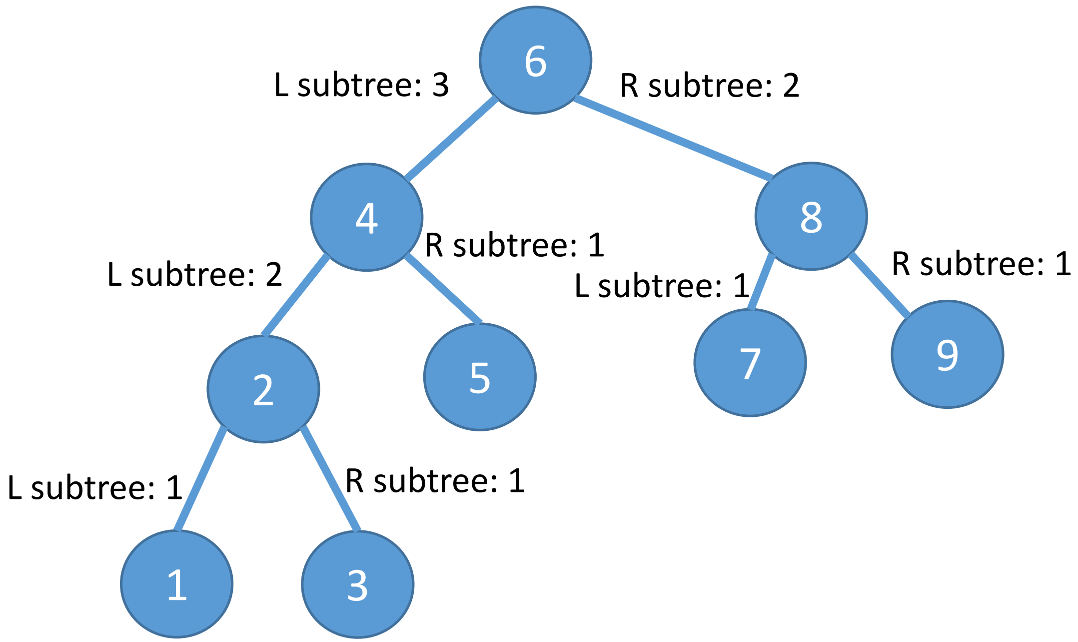
Balanced

Full Binary Trees

- An example of a BST that is balanced **and** full



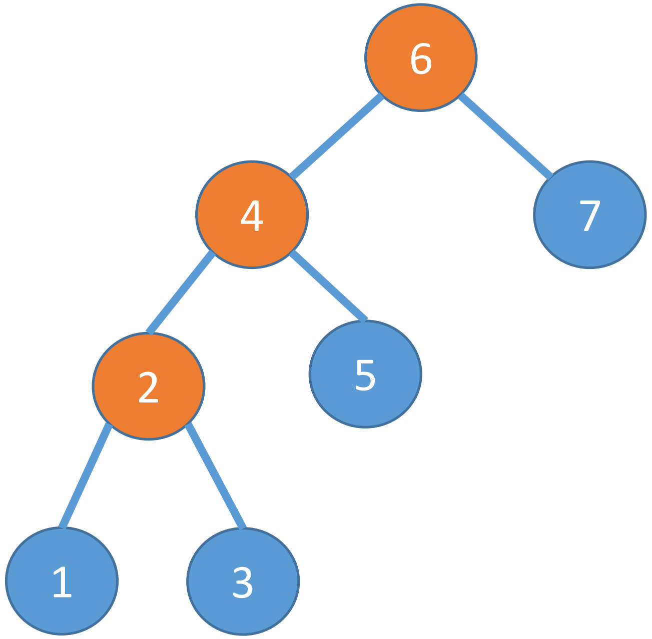
Full



Balanced

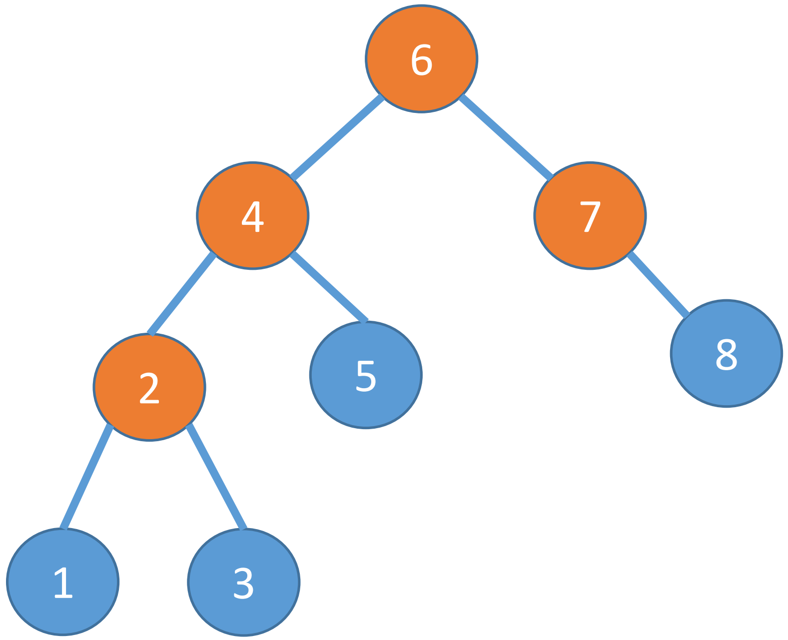
Full Binary Trees

- Checking for a Full Tree
  - Number of parents + 1 = Number of leaves
  - 3 + 1 = 4



Full Binary Trees

- Number of parents + 1 = Number of leaves
  - 4 + 1 **!=** 4
  - **Not Full**

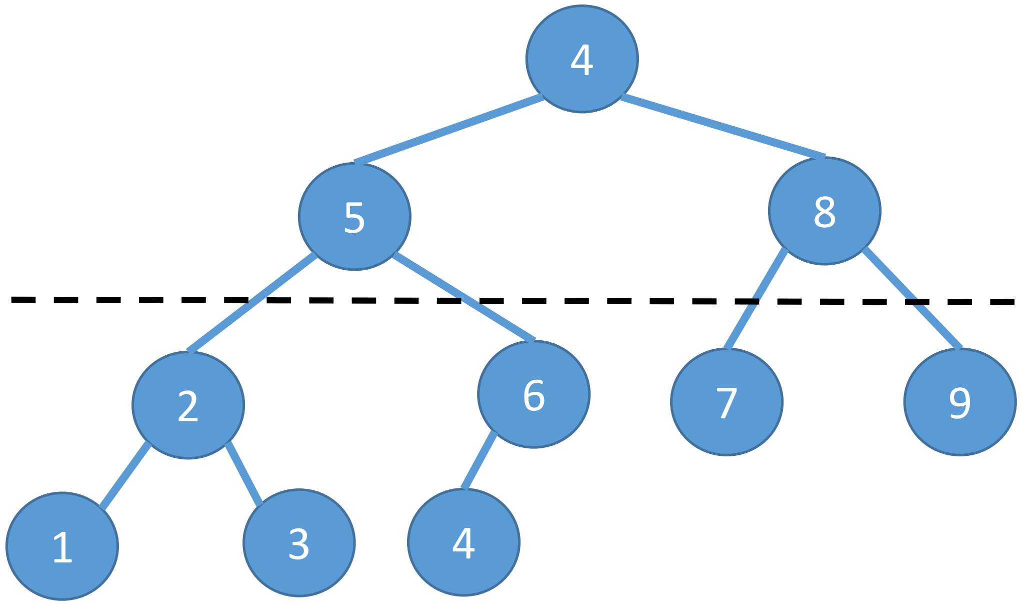


Complete Binary Trees

- A **complete binary tree** is when every level is filled (except for the last level) and the leaves are as far left as possible.

All nodes must be full

Last level (Nodes do not need to be full)
All nodes are far left



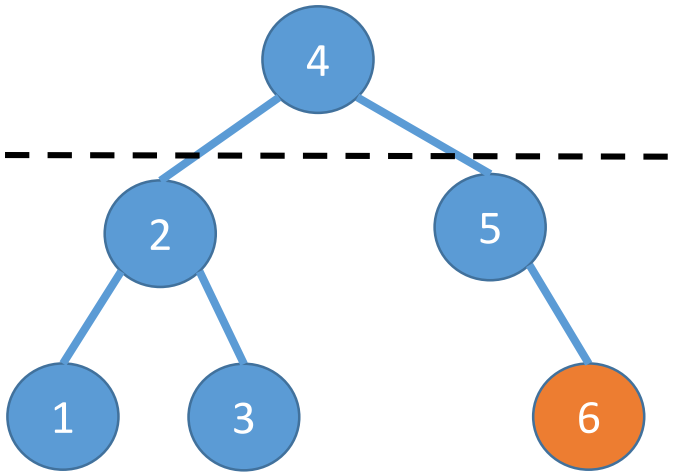
Balanced and Complete, but not Full

Complete Binary Trees

- An example of a BST that is balanced but not complete.
  - Not full, either

All nodes must be full

Last level (Nodes do not need to be full)
All nodes are far left



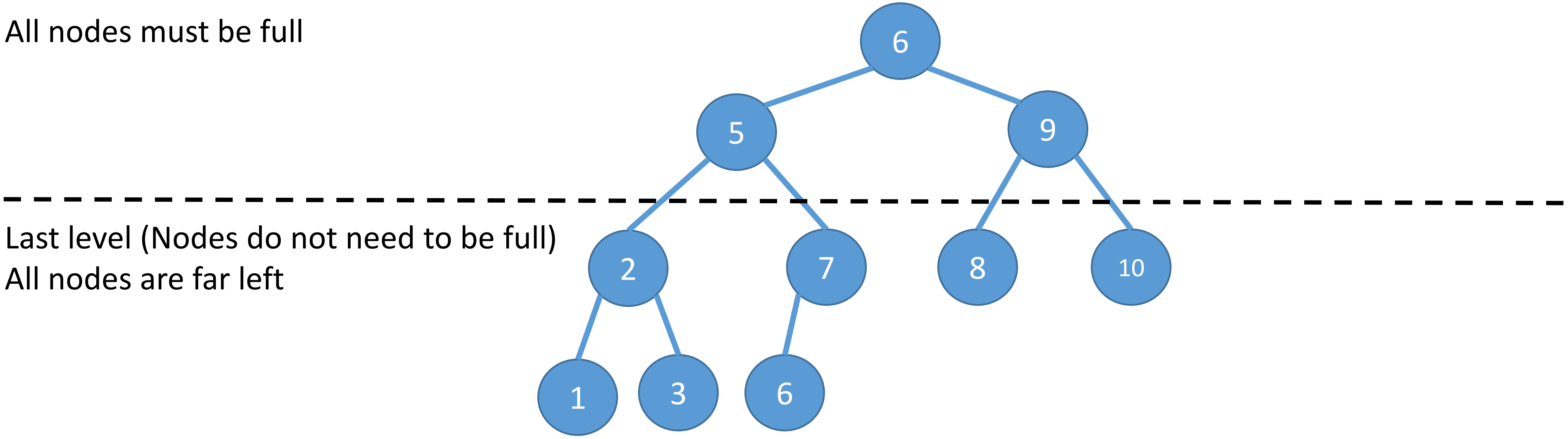
Balanced, not full, not complete

Complete Binary Trees

- An example of a BST that is balanced, not full, but complete.

All nodes must be full

Last level (Nodes do not need to be full)
All nodes are far left



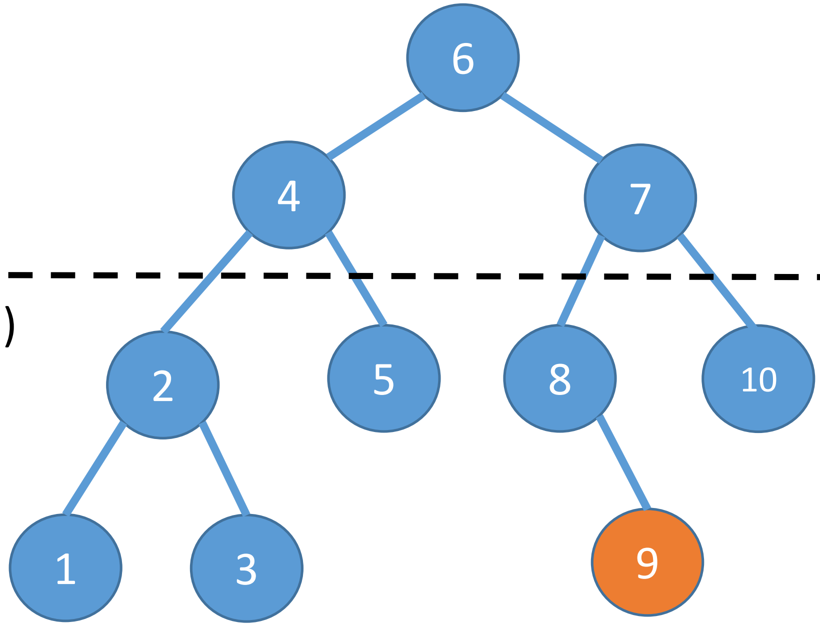
Balanced, not full, complete

Complete Binary Trees

- An example of a BST that is balanced, not full, and not complete.

All nodes must be full

Last level (Nodes do not need to be full)
All nodes are far left



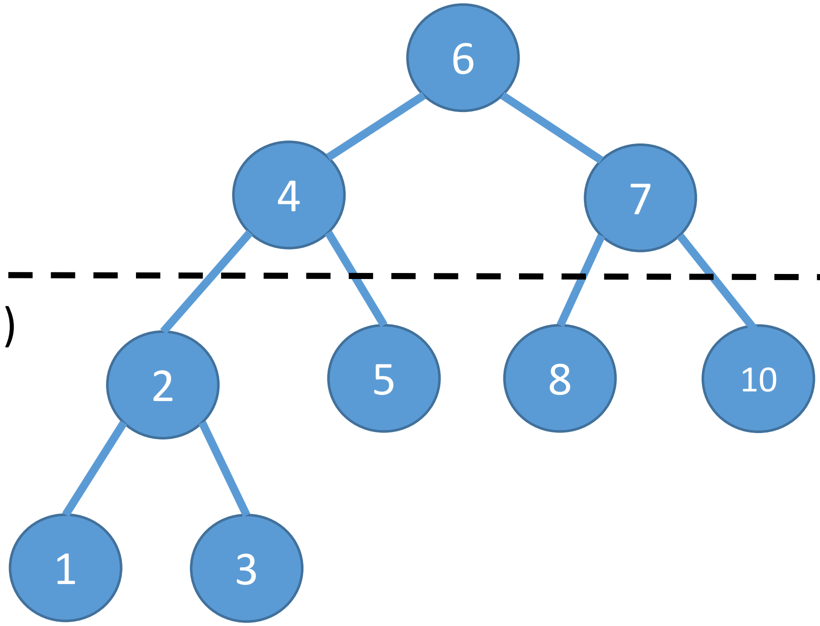
Balanced, not full, not complete

Complete Binary Trees

- An example of a BST that is balanced, full, and complete.

All nodes must be full

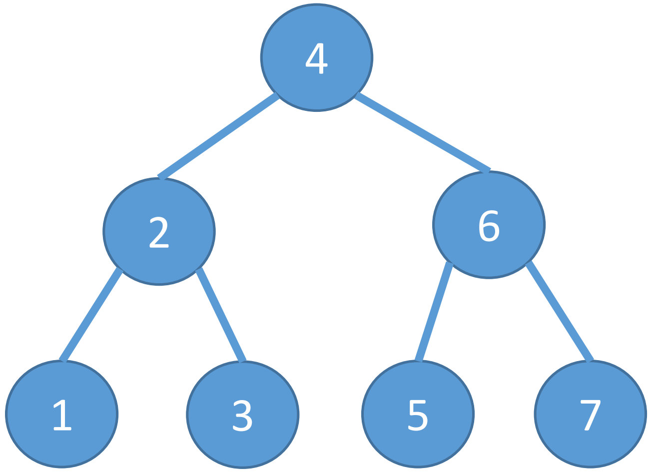
Last level (Nodes do not need to be full)
No lone right leaves



Balanced, full, complete

Perfect Binary Trees

- A **perfect binary tree** is when every node has two children and the leaves are all at the same level.
  - Always full, complete, and balanced.



Balanced, Complete, Full, and Perfect

Tree Structure Complexities

- Perfectly balanced trees perform in O(log n), balanced trees will perform somewhere between O(log n) and O(h), very unbalanced trees perform closer to O(n)

Structure	Insertion	Removal	Search	Find Min	Find Max
Pathological Tree	O(n)	O(n)	O(n)	O(n)	O(n)
Balanced Tree	O(h)	O(h)	O(h)	O(h)	O(h)
Perfect Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)