Revised: 7/7/2024

Trees I

Michael C. Hackett
Assistant Professor, Computer Science

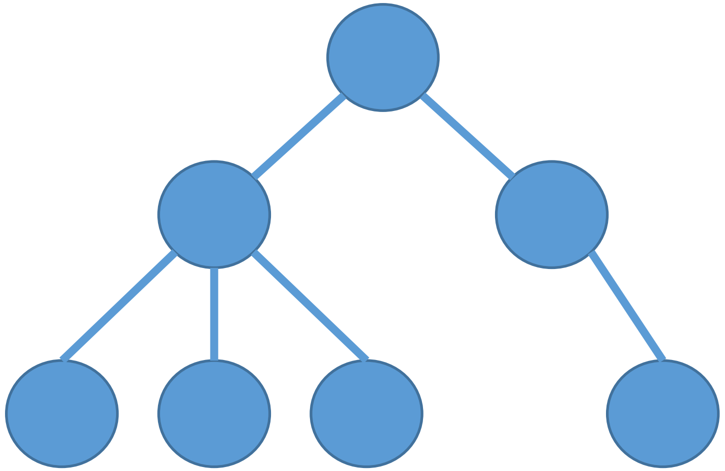
COMMUNITY COLLEGE OF PHILADELPHIA

Lecture Topics

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

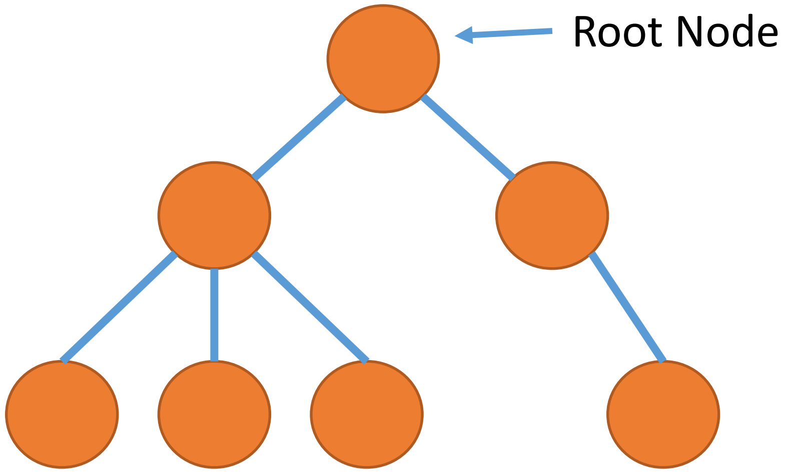
Trees

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



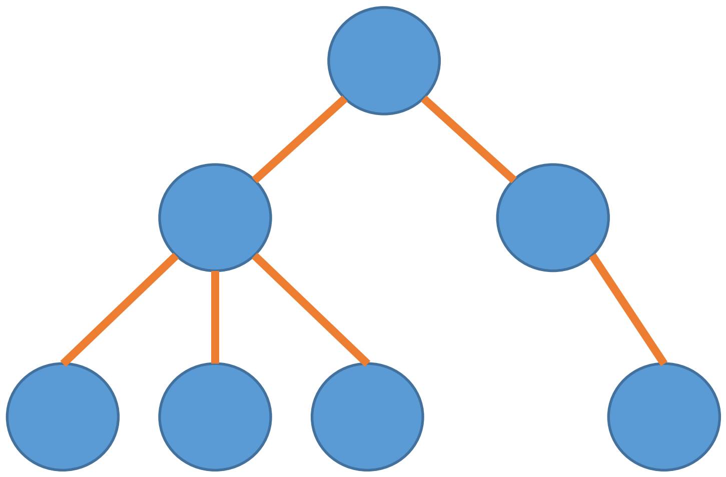
Trees

- Each point in the tree is called a **node** or **vertex**
  - Shown in orange below
- The top-most node is called the **root** node
  - The root node is the tree's starting point



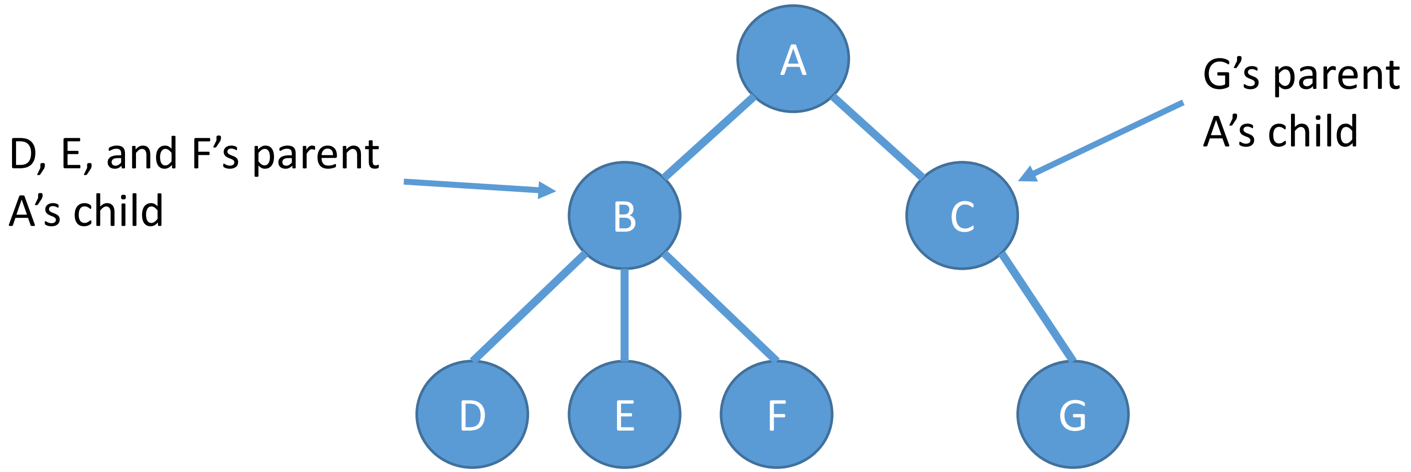
Trees

- The lines connecting the nodes are **edges** or **branches**
  - Shown in orange below



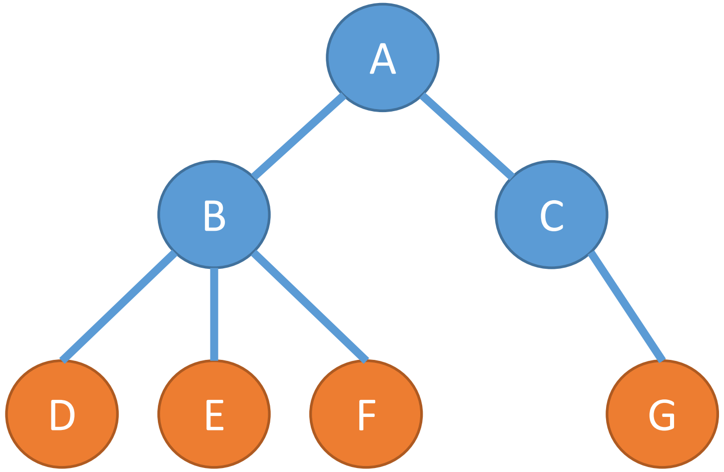
Trees

- **Children** or **child nodes** are nodes that branch from a higher node.
- A node's **parent** is the node it branches from.
  - The root node is the only node in a tree that does not have a parent.



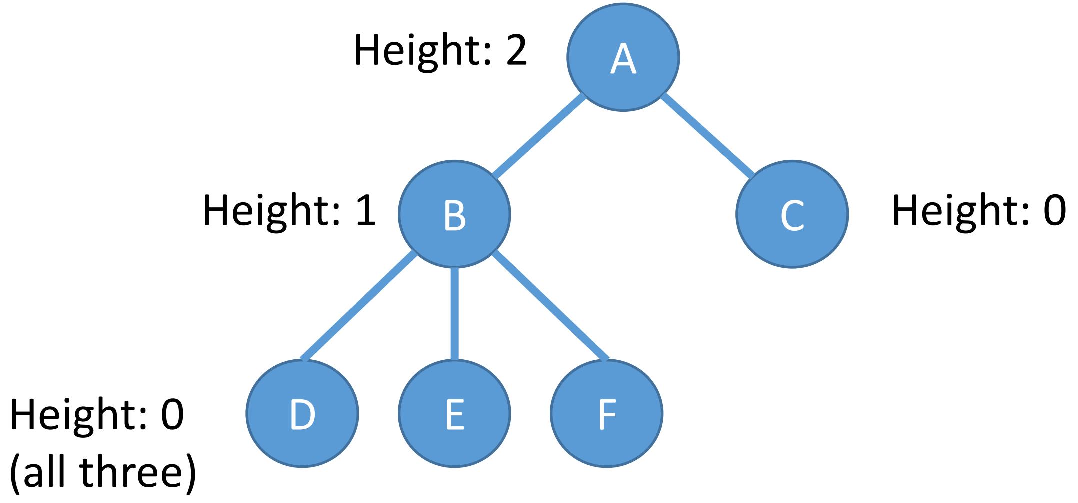
Trees

- A **leaf** or **leaf node** is a node with no children
  - Nodes D, E, F, and G



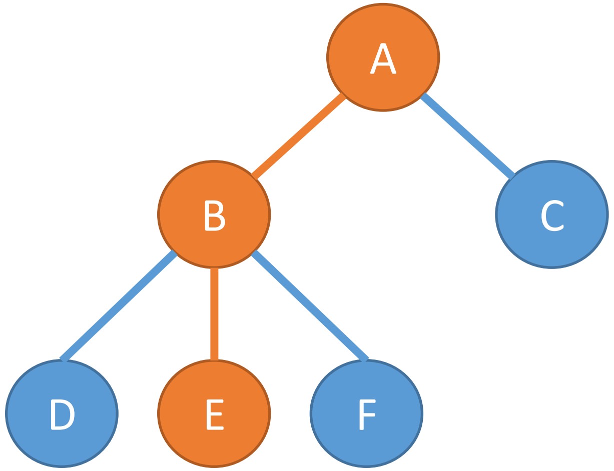
Trees

- Each node's **height** is its distance (number of edges) to its <u>farthest</u> leaf.



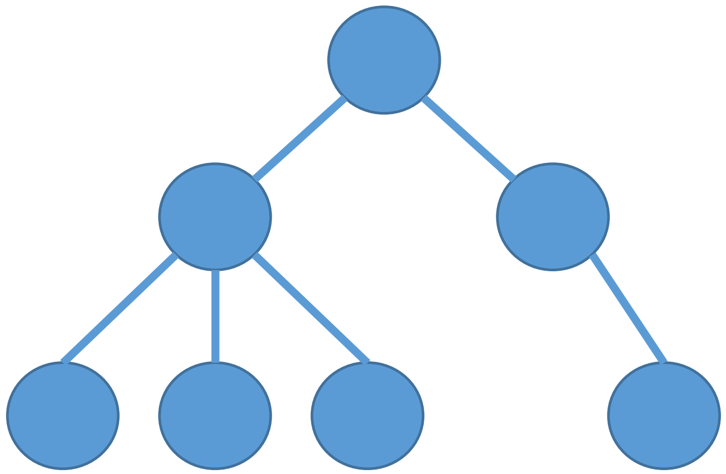
Trees

- A **path** is the edges that connect the root node to any other particular node in the tree.
  - In a tree, paths only begin at the root node
  - Path from node A to node E:

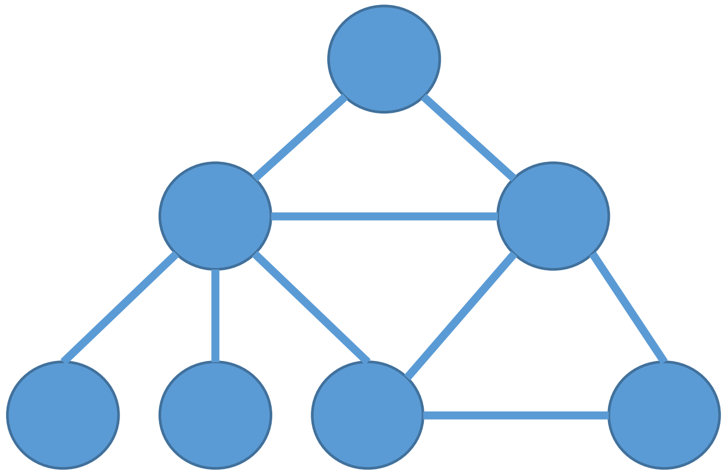


Trees

- There is only **<u>one</u>** path from the root to any node in the tree



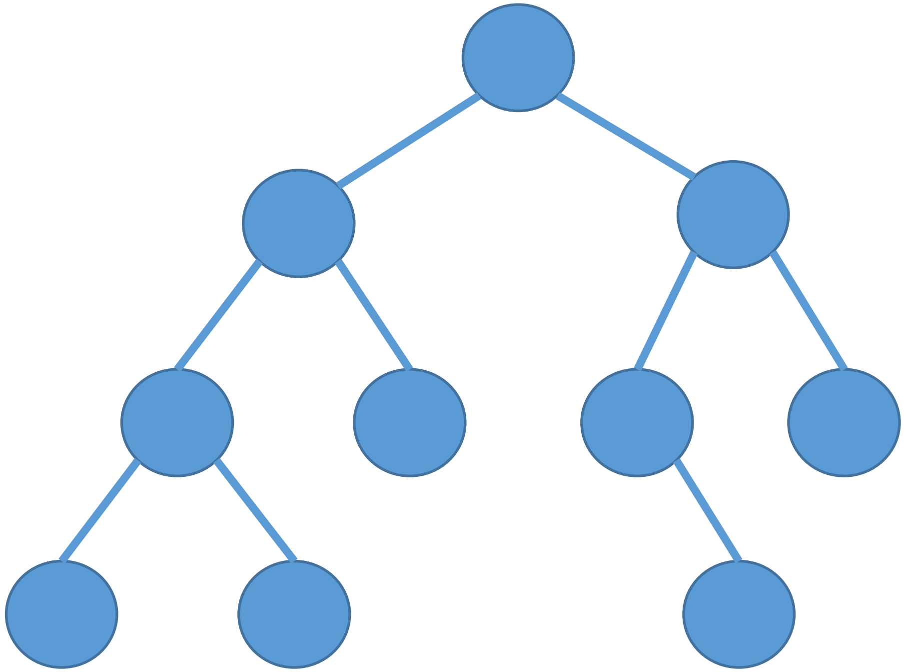
Tree



Not a Tree

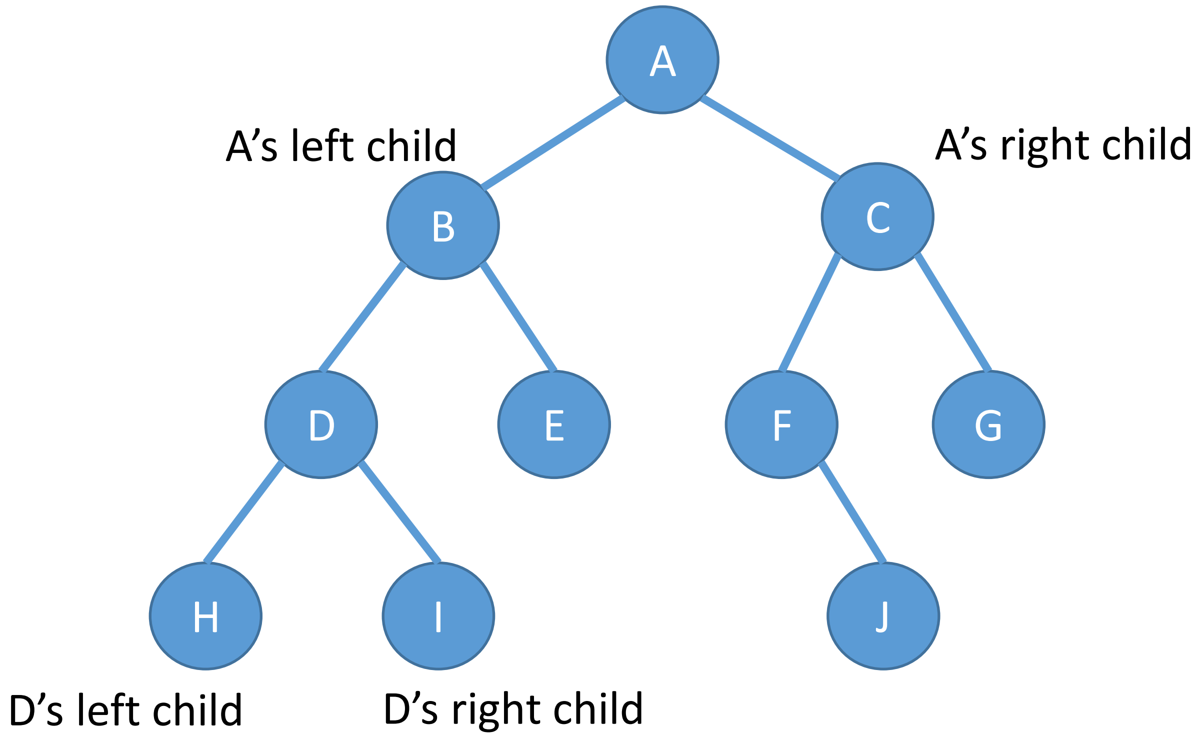
Binary Trees

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



Binary Trees

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



Tree Traversals

- The process of visiting the nodes of a tree is called a **traversal**.

- Traversals always begin at the root node
  - Other nodes cannot be accessed directly

- Two types of tree traversals:
  - Depth-First Traversal
  - Breadth-First Traversal

Depth-First Traversal

- A depth-first traversal begins at the root node and travels down the left path until reaching a leaf
  - Works its way back up the tree, going down the right path of any nodes
  - Travels down the left path until reaching a leaf
  - Process repeats

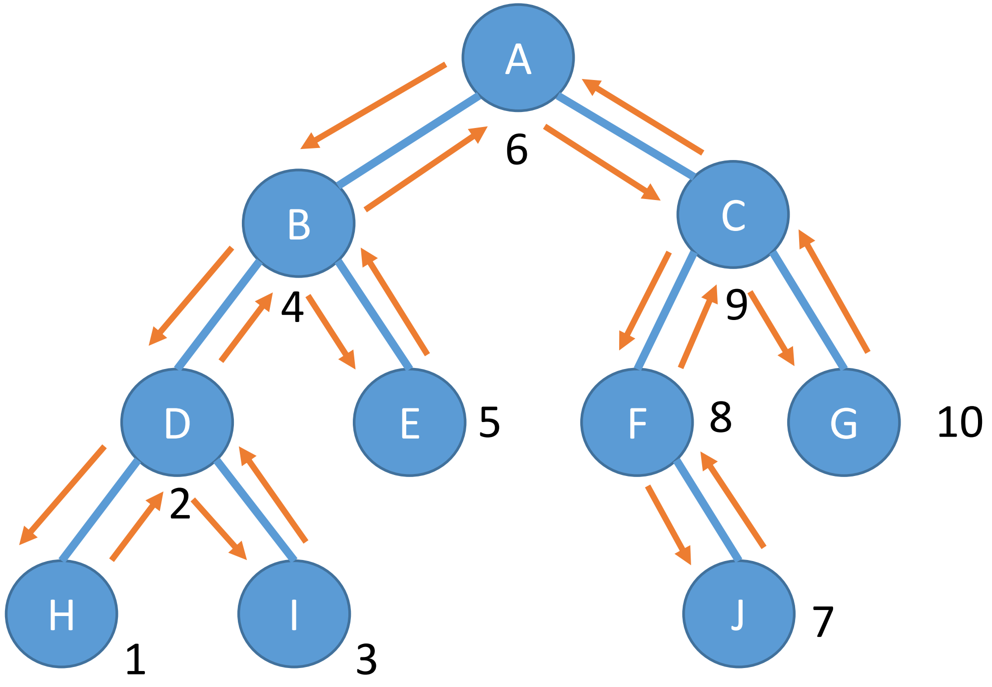
- There are three methods of performing a depth-first traversal: In-Order, Pre-Order, and Post-Order traversals.
  - The path taken by each is always the same.

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.

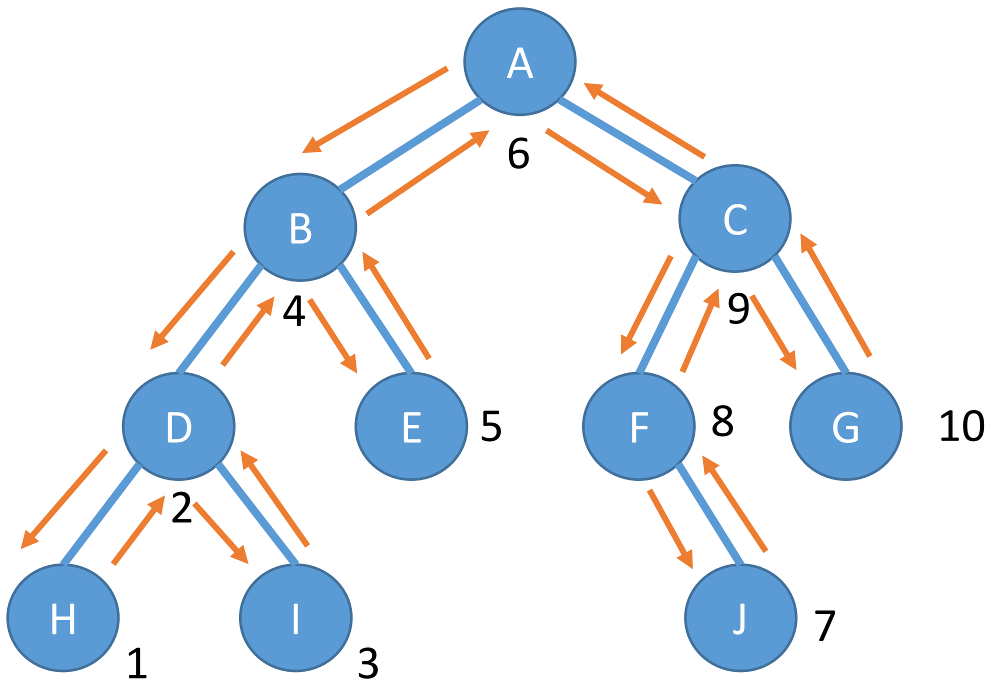
In-Order Traversal

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



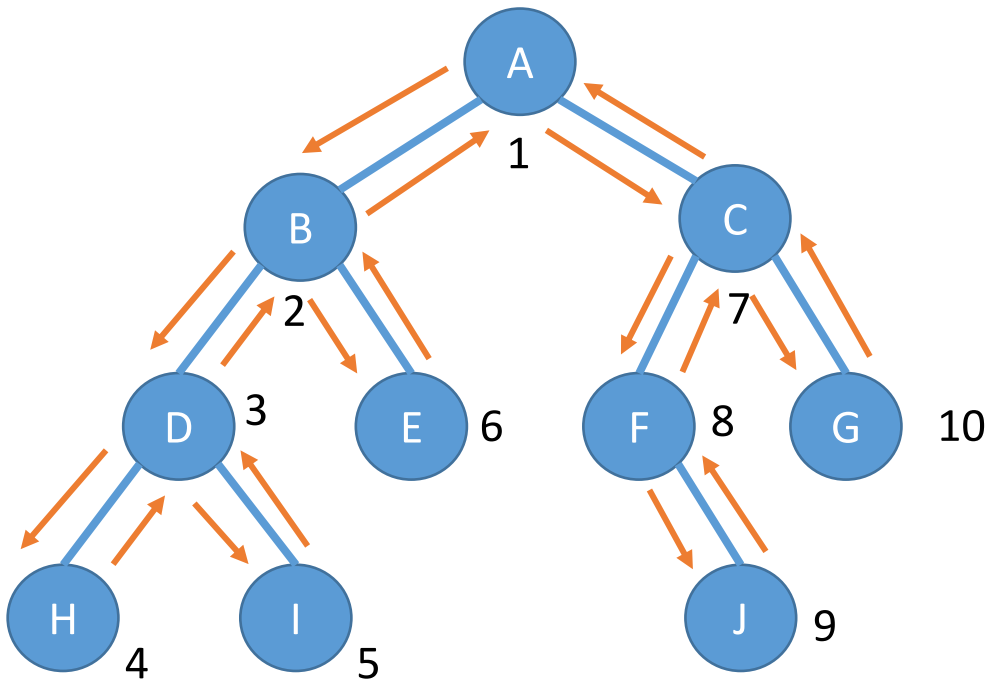
In-Order Traversal

- The value of the node is processed upon the **second** time it is visited
  (or first time if its a leaf)
- **Infix Format**: H D I B E A J F C G



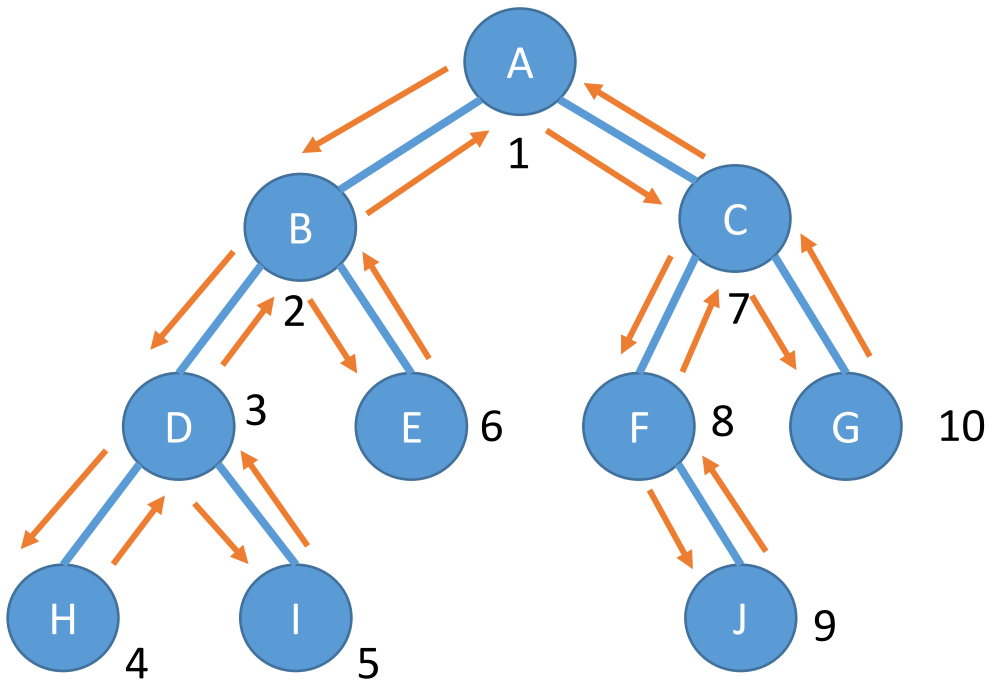
Pre-Order Traversal

- Arrows show the direction of the traversal (same as in-order)
- Numbers indicate when the values of the nodes are processed in the traversal.



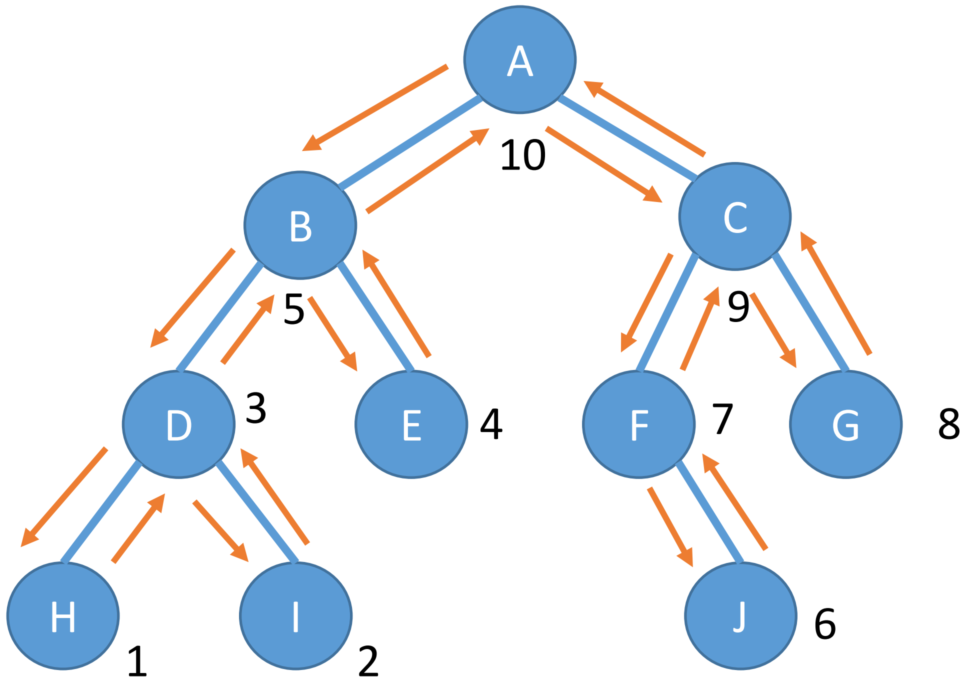
Pre-Order Traversal

- The value of the node is processed upon the **first** time it is visited
- **Prefix Format**: A B D H I E C F J G



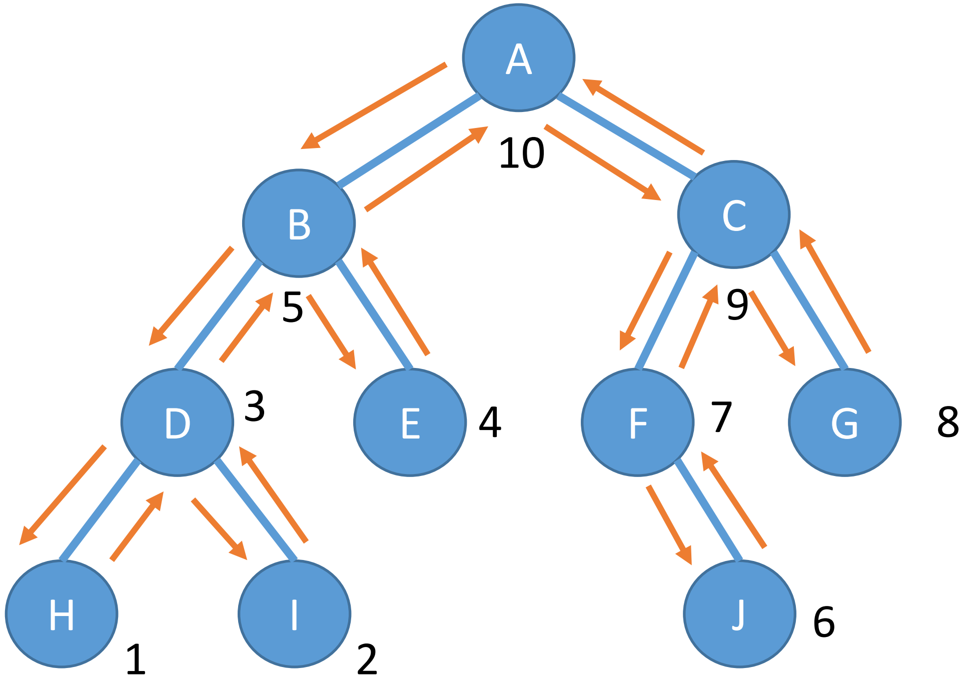
Post-Order Traversal

- Arrows show the direction of the traversal. (Same as in-order and pre-order)
- Numbers indicate when the values of the nodes are processed in the traversal.



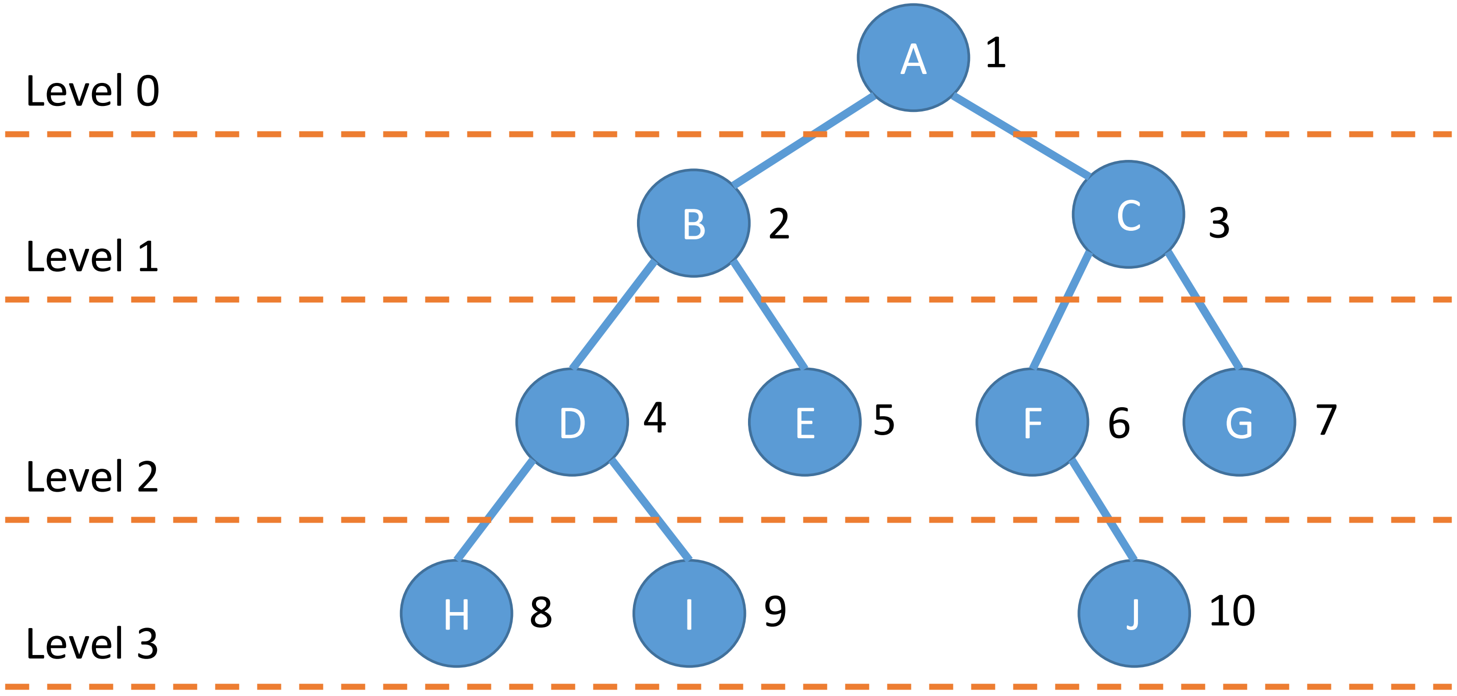
Post-Order Traversal

- The value of the node is processed upon the **last** time it is visited.
- **Postfix Format**: H I D E B J F G C A



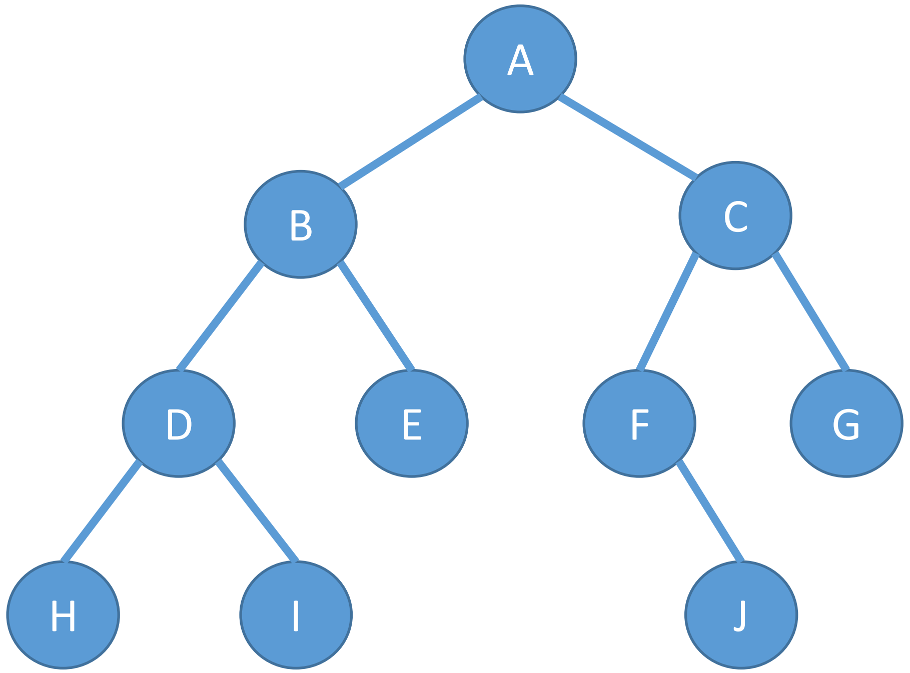
Breadth-First Traversal

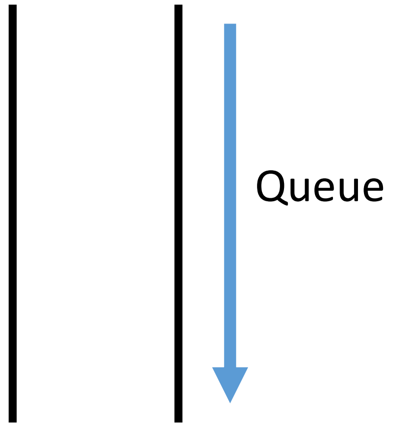
- Using a **breath-first** or **level-order traversal**, the tree is traversed by visiting all nodes at each level of the tree, working its way to the bottom.



Breadth-First Traversal

- Breadth-first traversals often use a queue to complete the traversal.

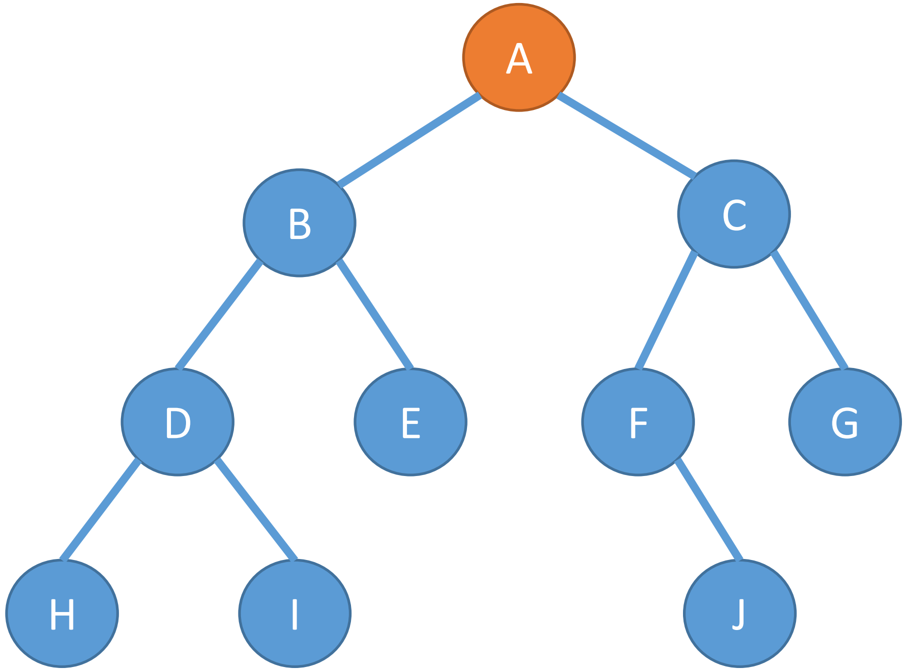


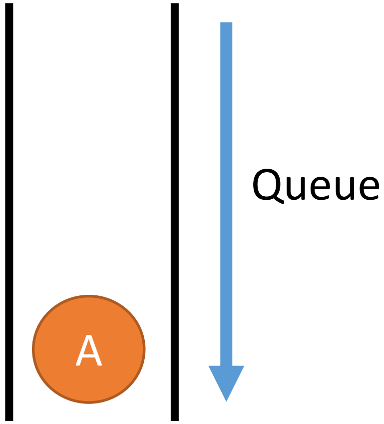


Node values used:

Breadth-First Traversal

- First, the root is added to the queue.

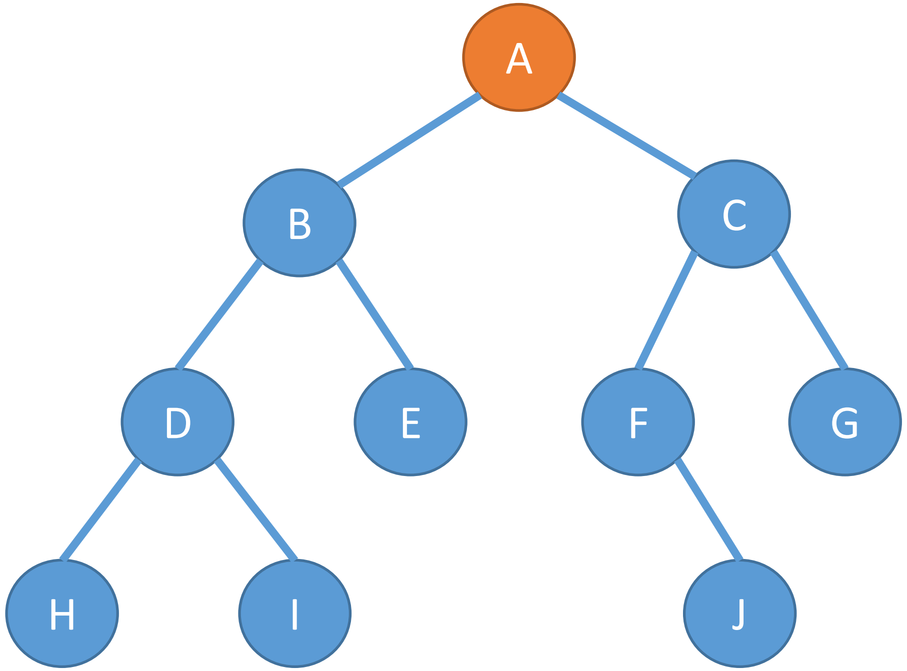


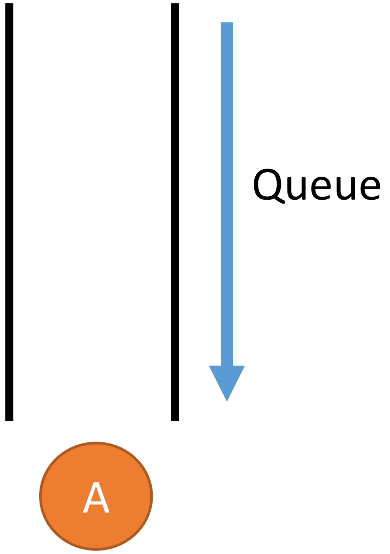


Node values used:

Breadth-First Traversal

- Then, it is popped from the queue

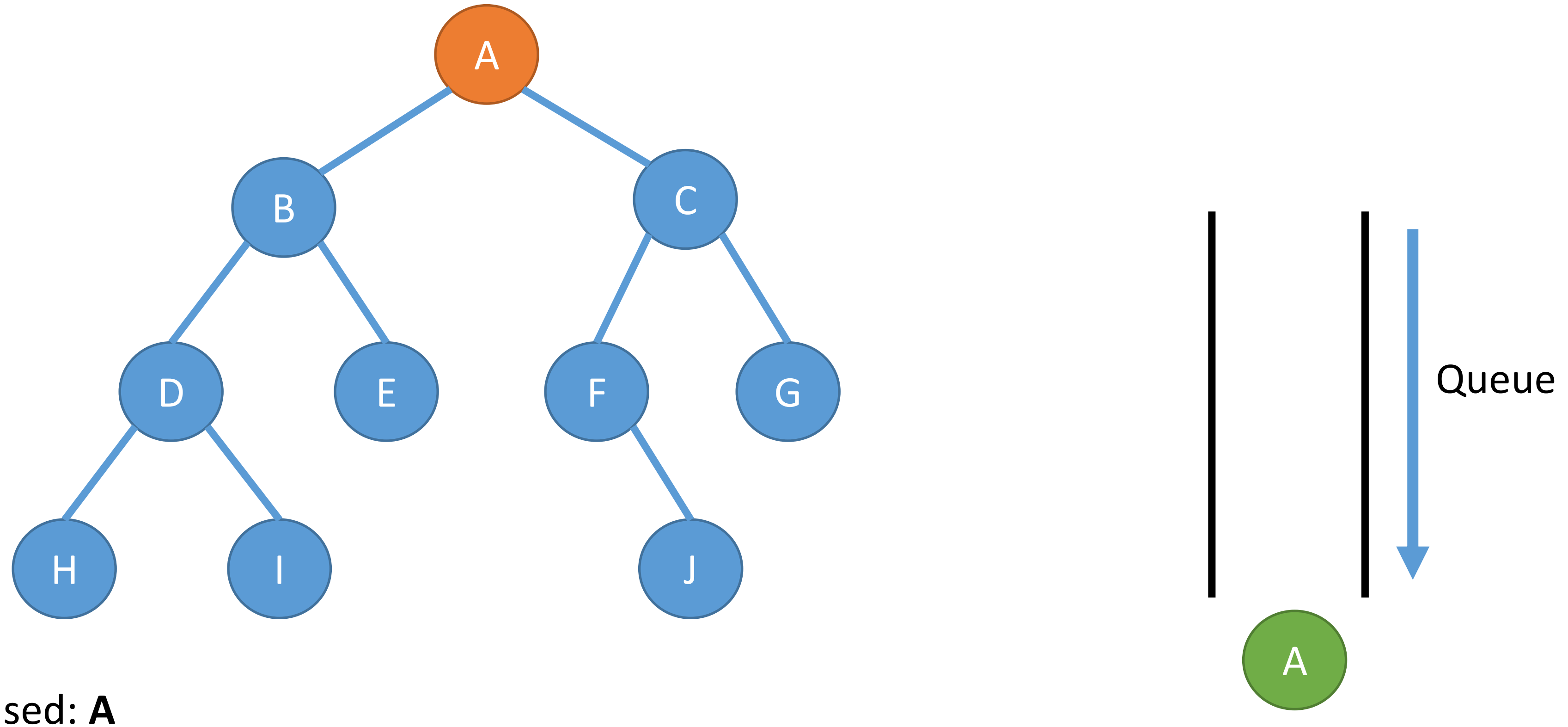




Node values used:

Breadth-First Traversal

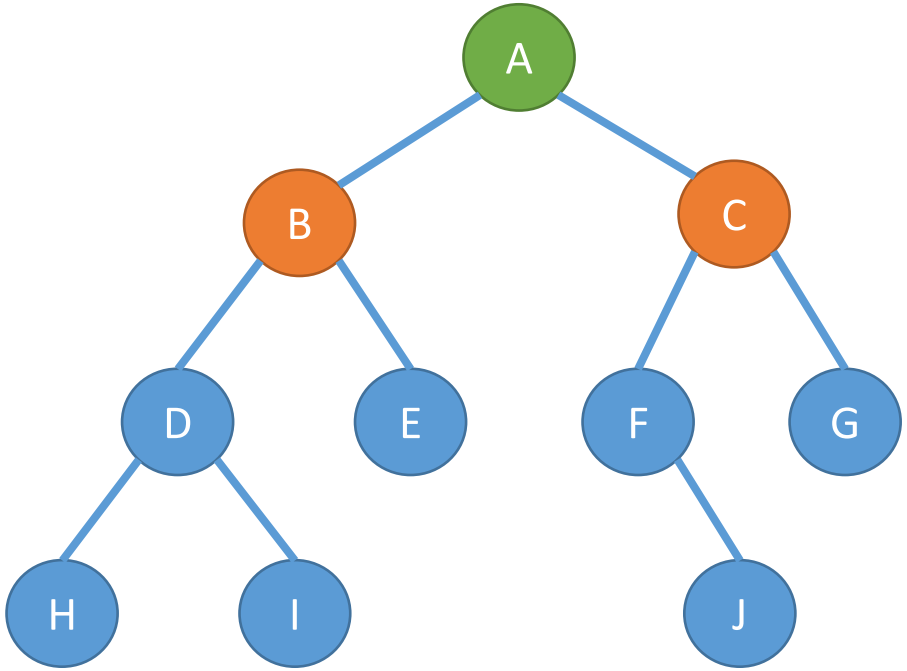
- Its data is processed

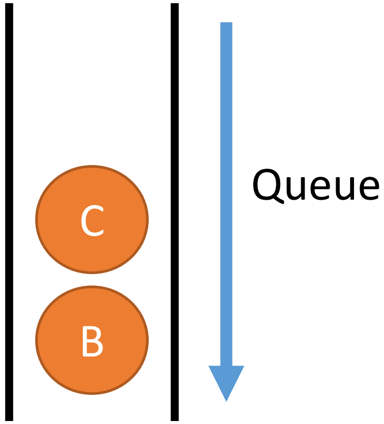


Node values used: **A**

Breadth-First Traversal

- Its children are added to the queue

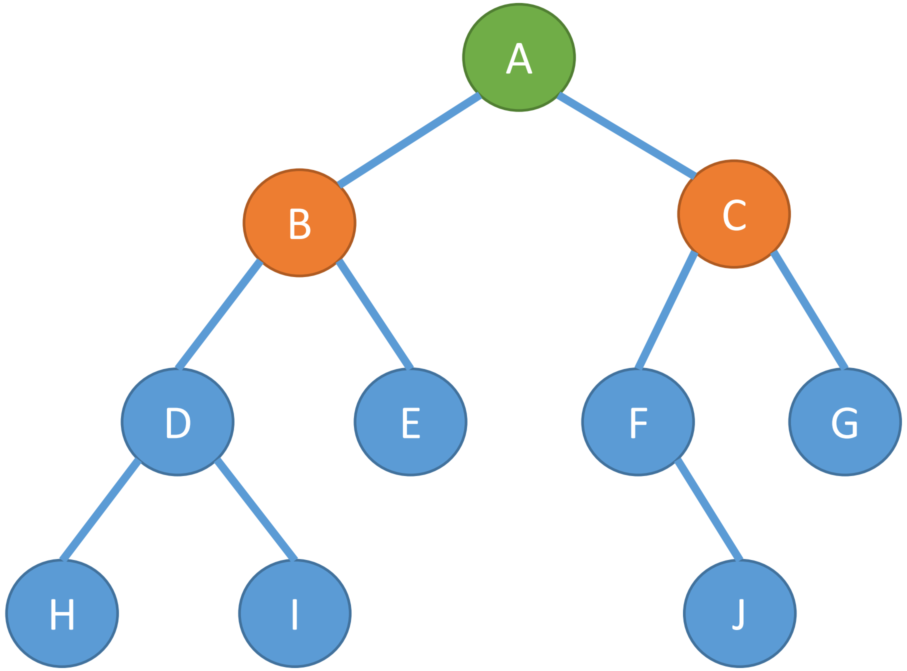


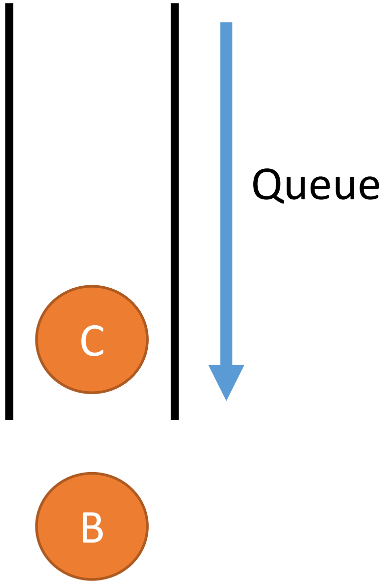


Node values used: **A**

Breadth-First Traversal

- An item is popped from the queue.

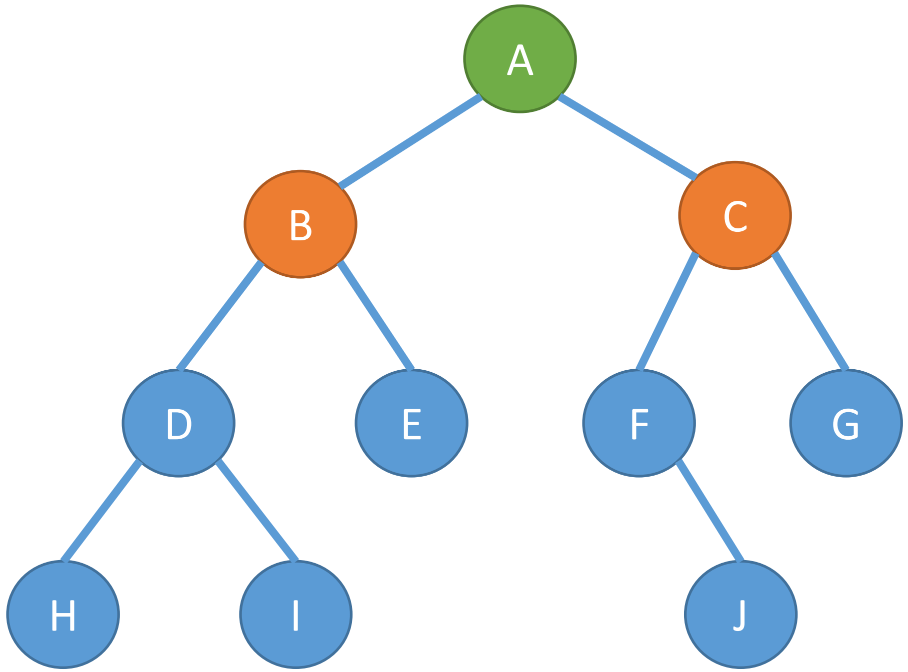




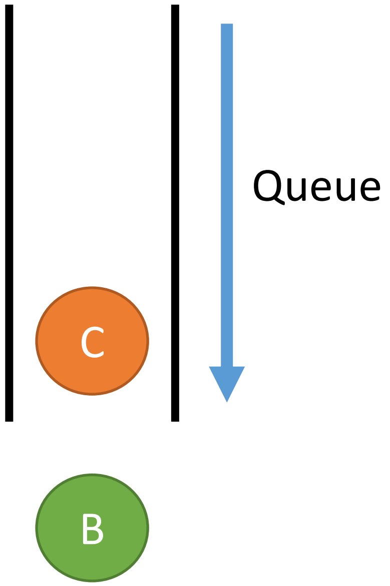
Node values used: **A**

Breadth-First Traversal

- Its data is processed

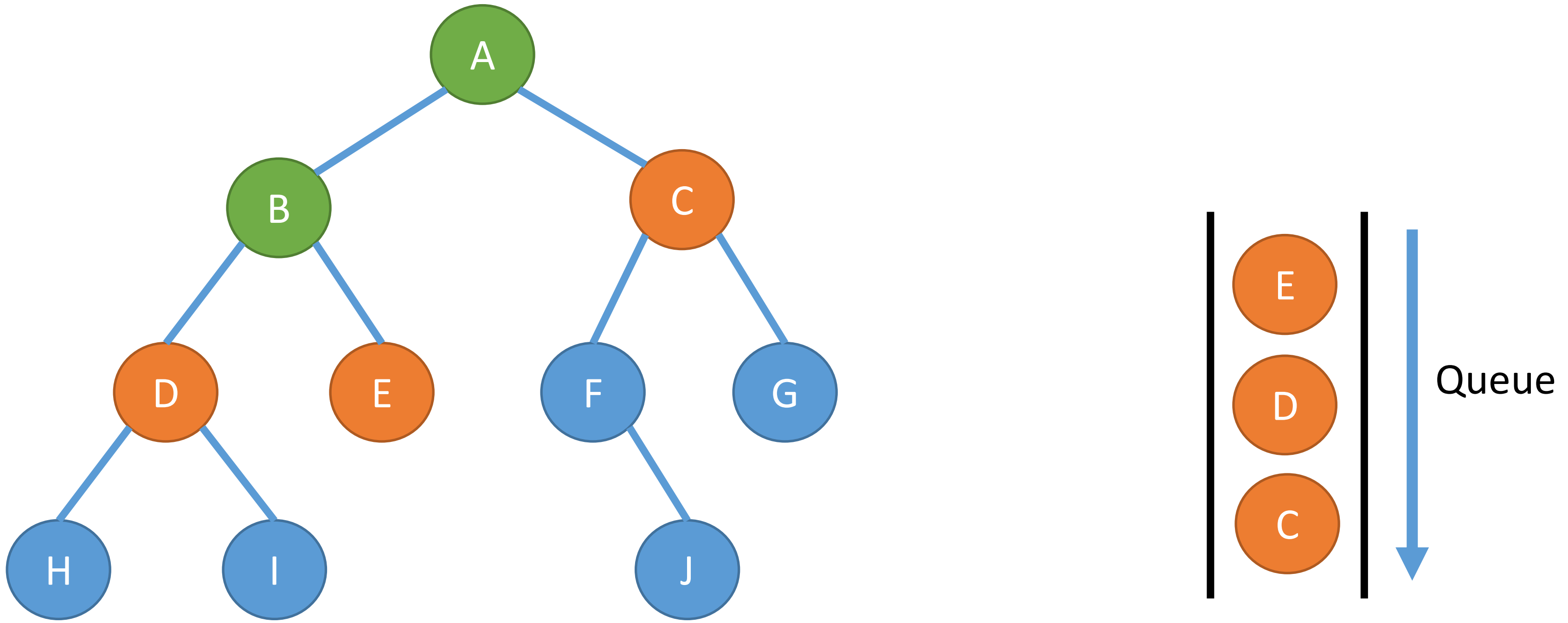


Node values used: **A B**



Breadth-First Traversal

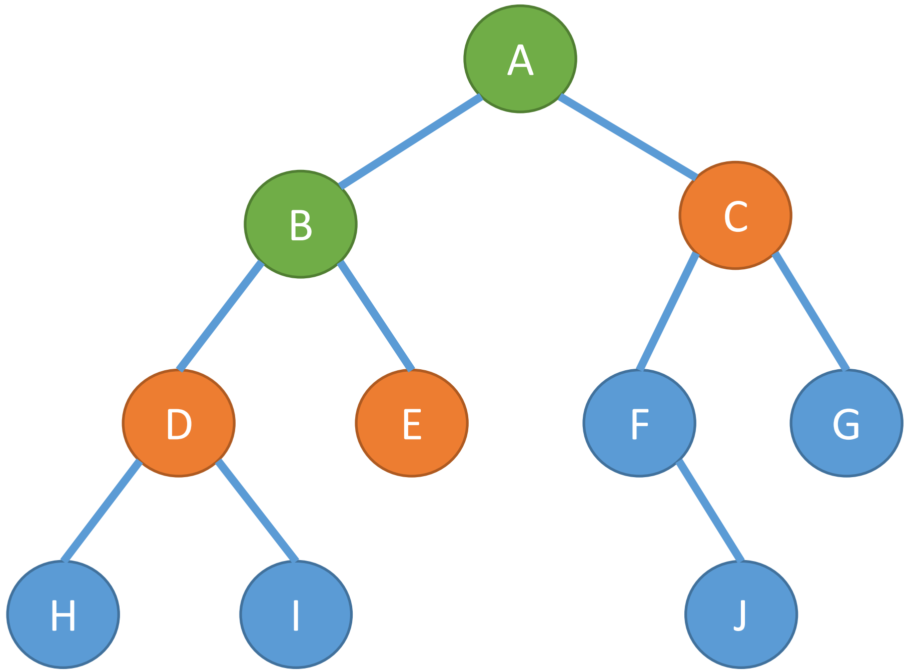
- Its children are added to the queue

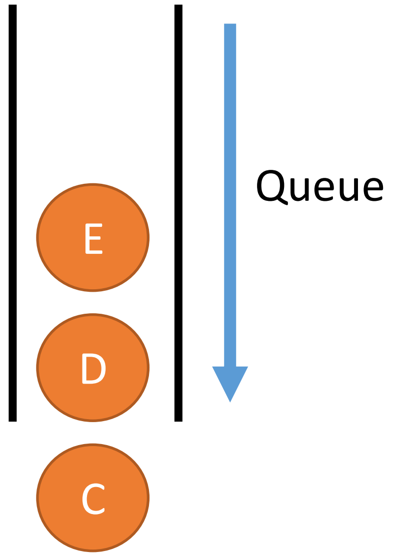


Node values used: **A B**

Breadth-First Traversal

- An item is popped from the queue.

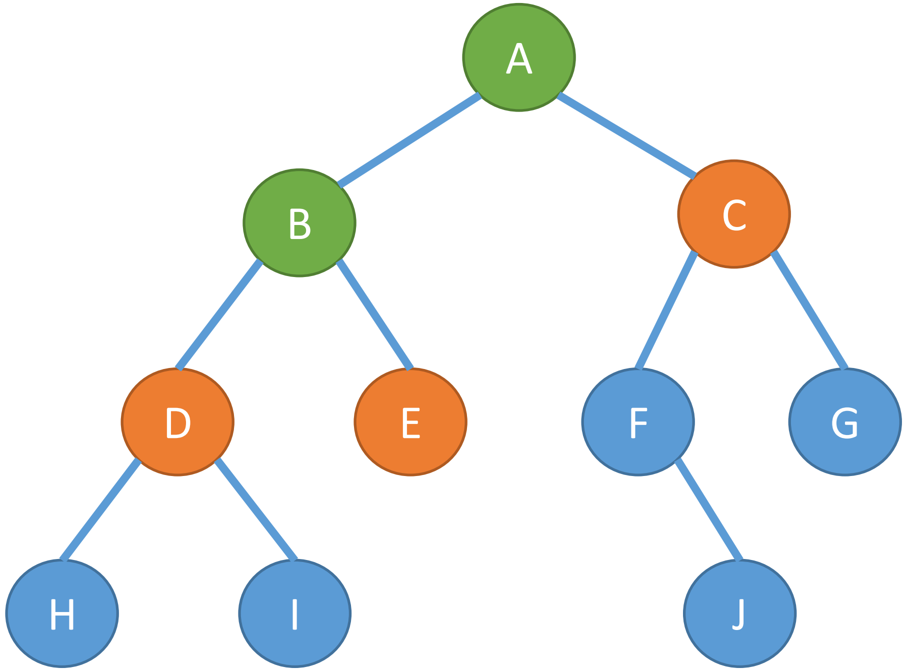


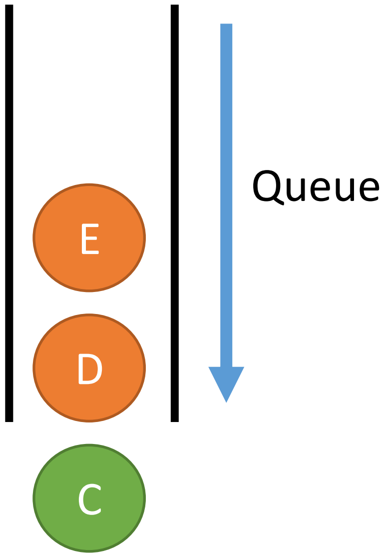


Node values used: **A B**

Breadth-First Traversal

- Its data is processed

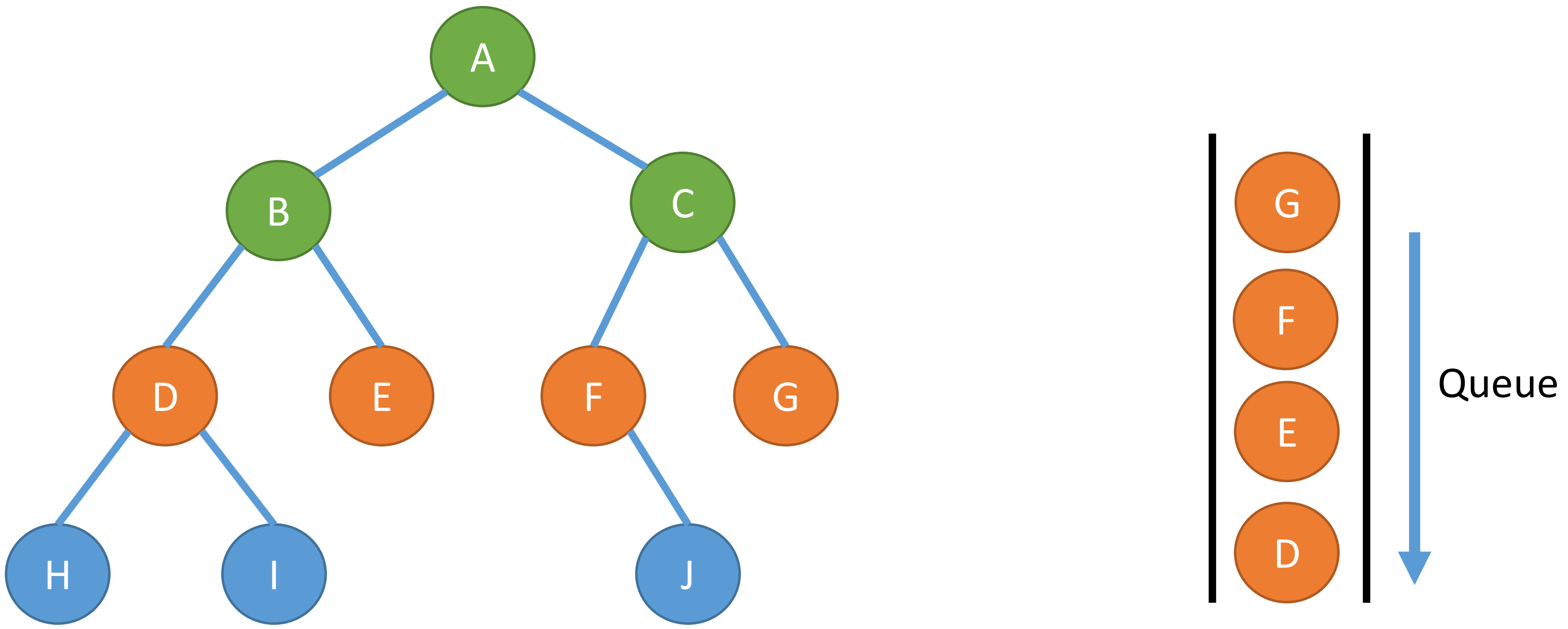




Node values used: **A B C**

Breadth-First Traversal

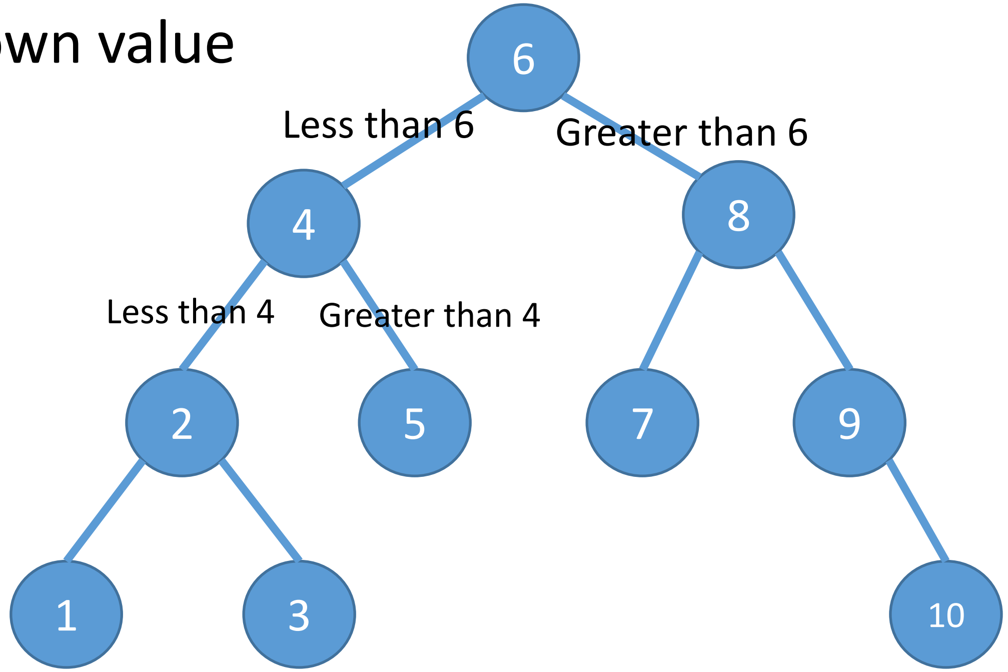
- Its children are added to the queue
  - The process repeats until the queue is empty



Node values used: **A B C**

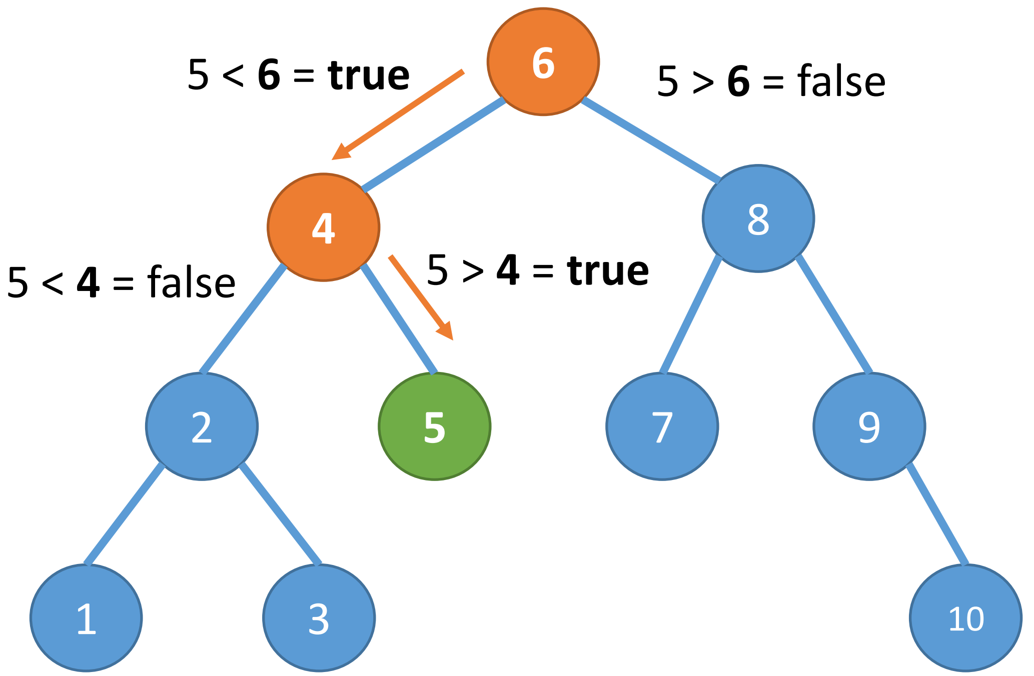
Binary Search Trees

- A **binary search tree** (or **BST**) is a binary tree where, for each node:
  - Its left child's value is **less than** its own value
    - As will all its children
  - Its right child's value is **greater than** its own value
    - As will all its children



Binary Search Trees

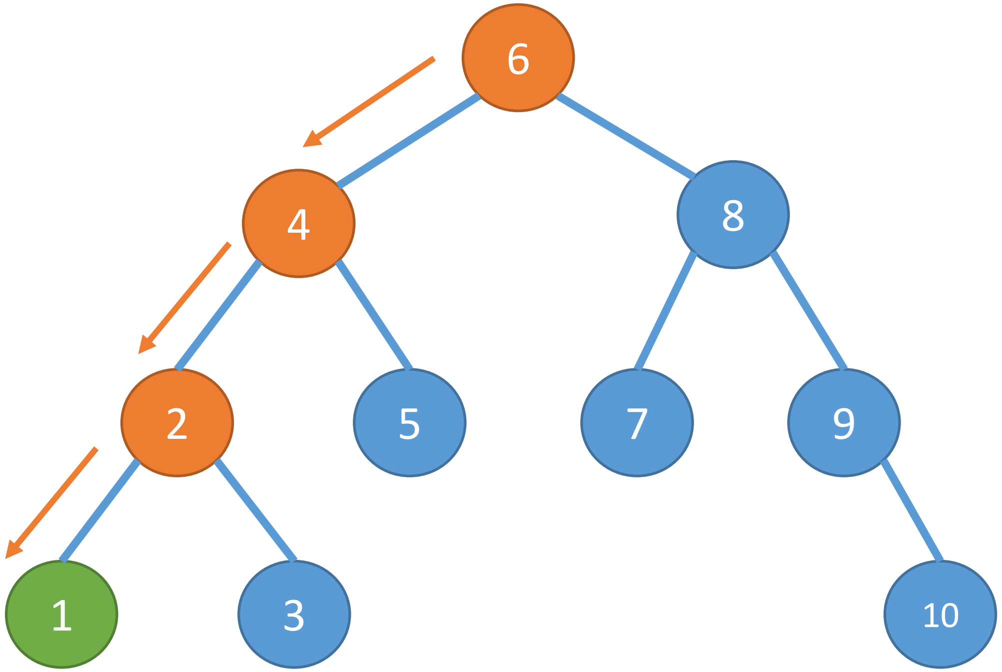
- This allows simple less-than or greater-than comparisons to search the tree for a value.
  - Searching for 5:



A similar process is used for adding new nodes to a BST

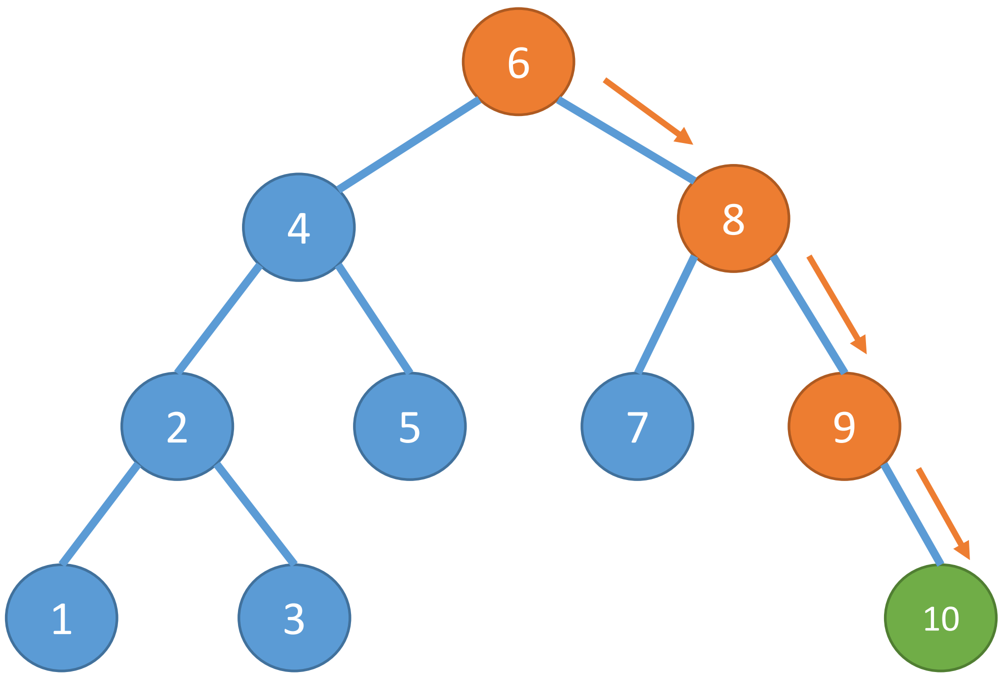
Binary Search Trees

- 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.



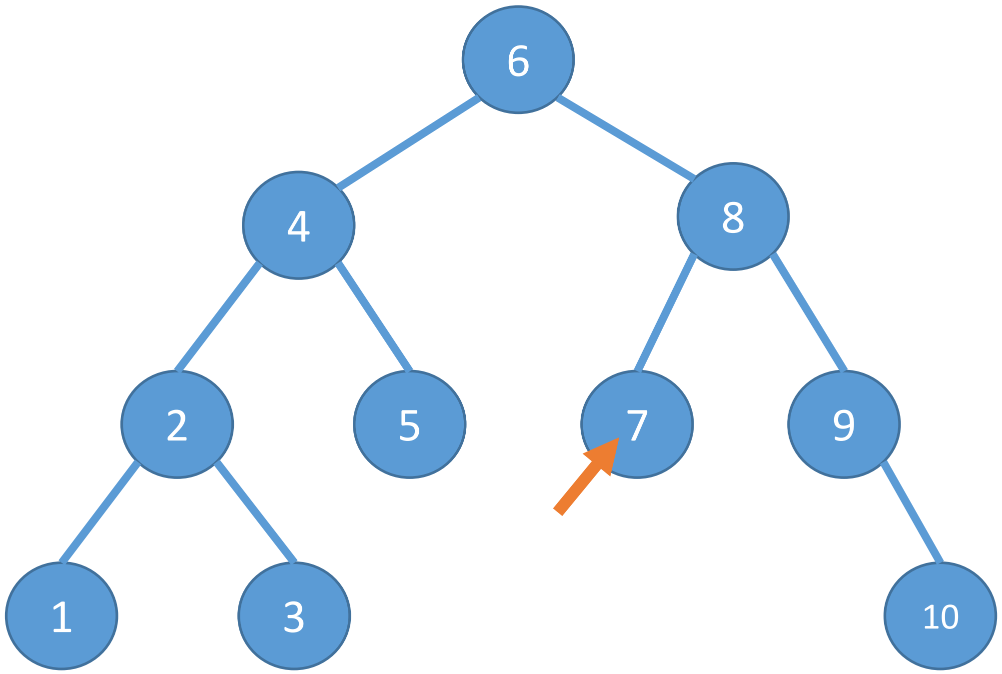
Binary Search Trees

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

- If the node to remove..
  - **Has no children (is a leaf)** – 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 trickiest scenario) The smallest value down the right branch of the node is the successor.

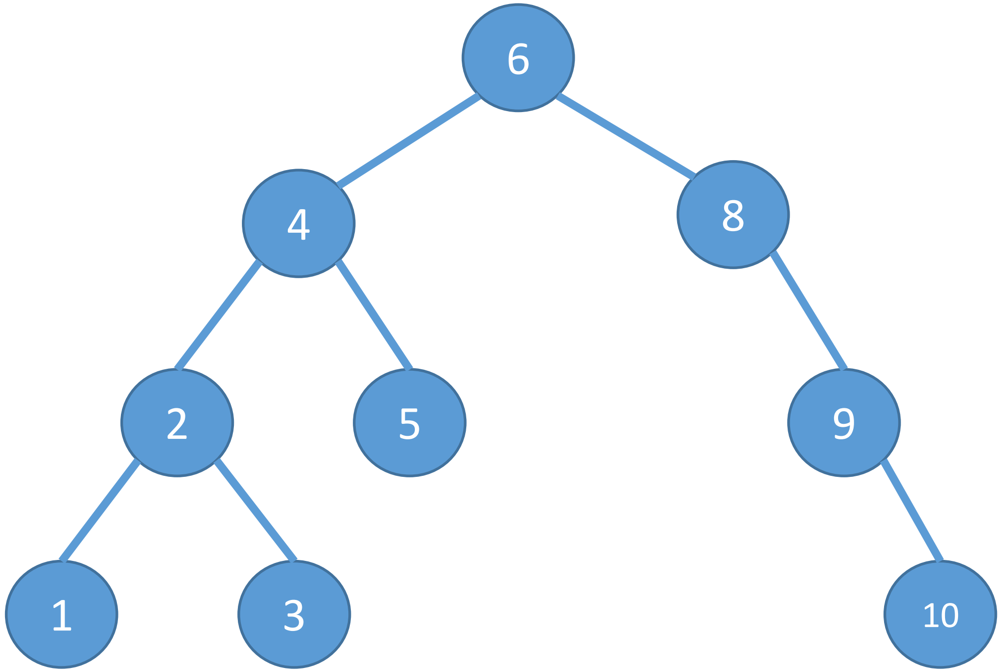
Binary Search Trees

- Removing the node containing 7…



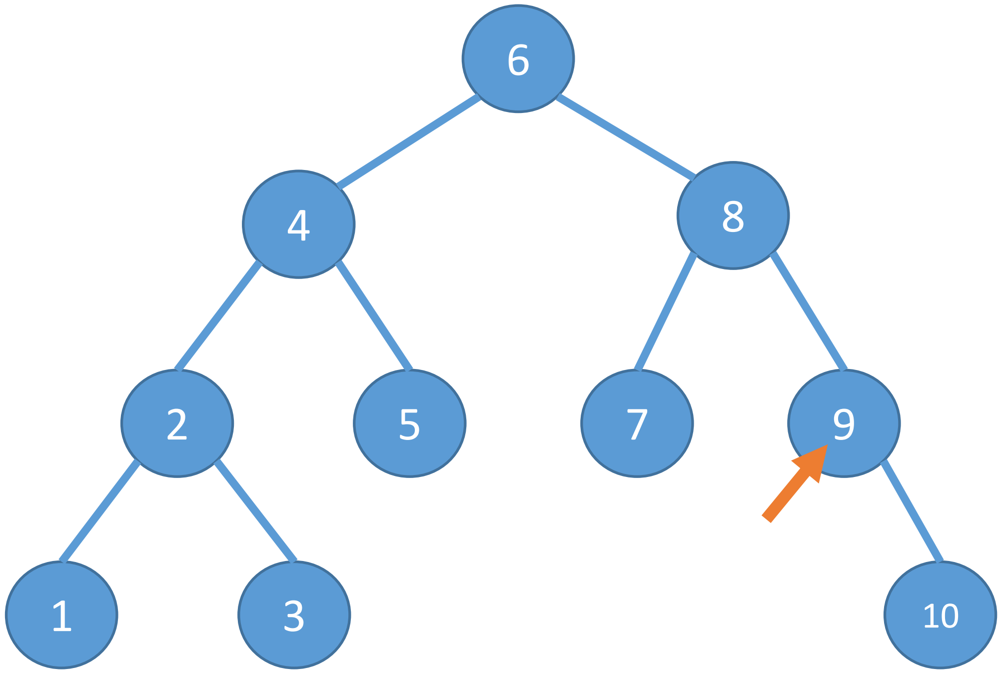
Binary Search Trees

- Removing the node containing 7… No successor, safe to simply remove



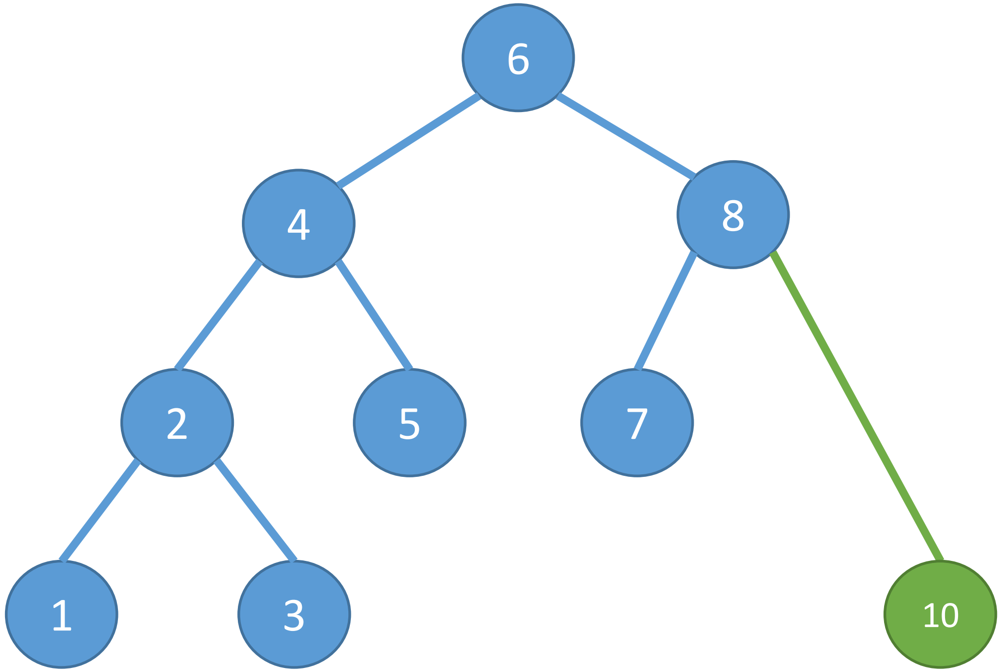
Binary Search Trees

- Removing the node containing 9…



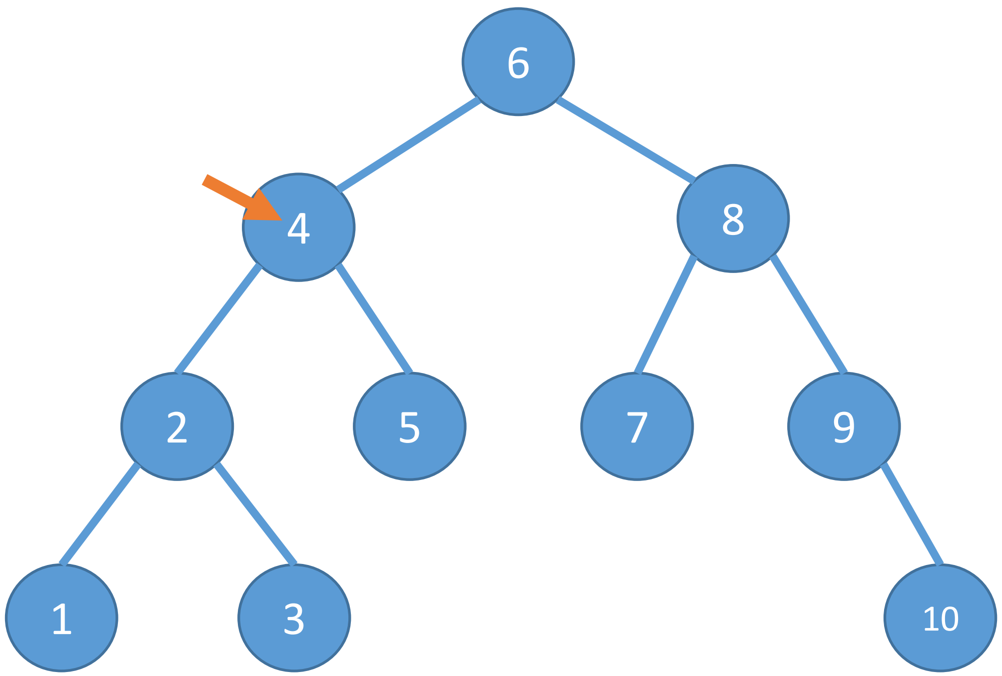
Binary Search Trees

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



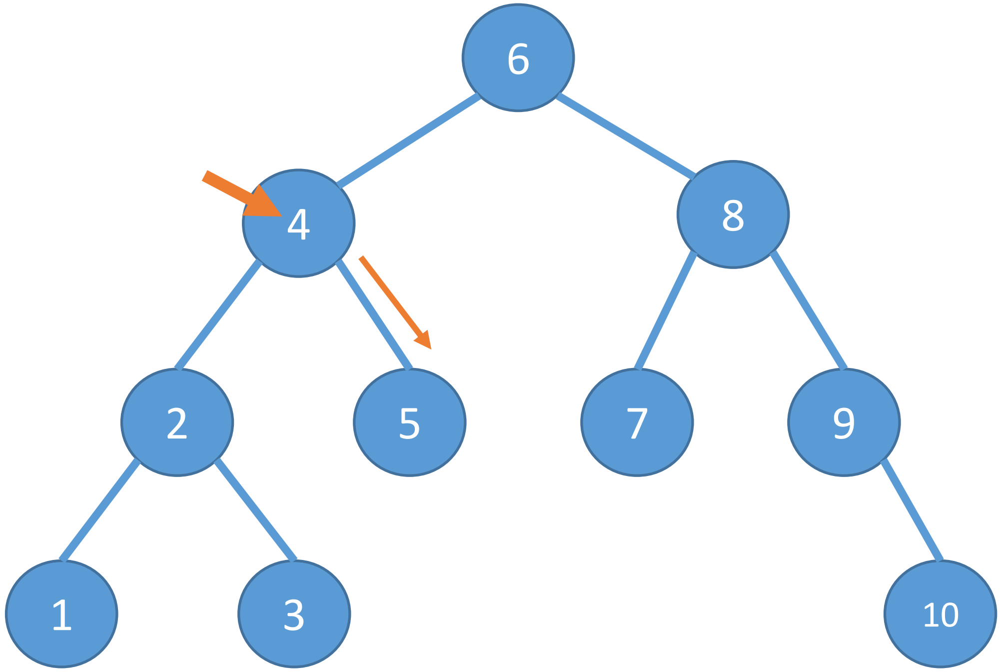
Binary Search Trees

- Removing the node containing 4…
  - Has two children



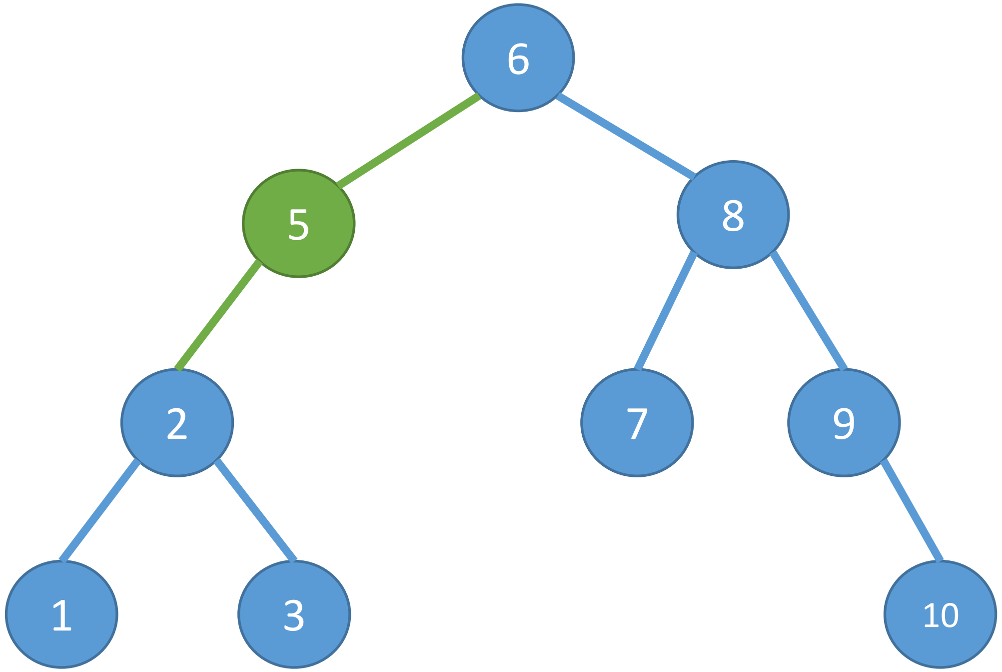
Binary Search Trees

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



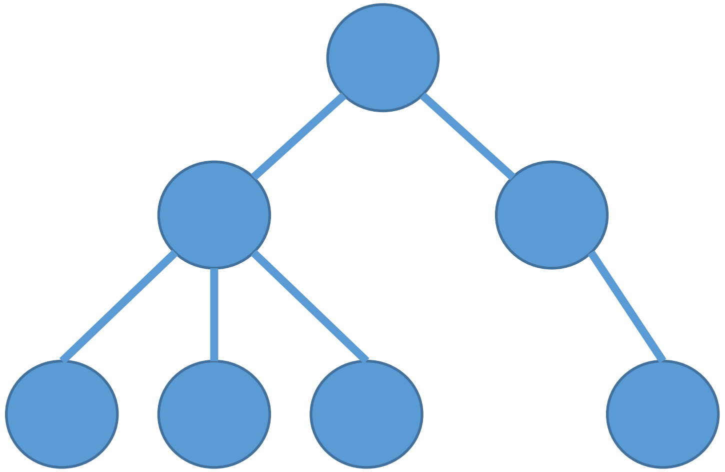
Binary Search Trees

- 5 is the smallest (and only) node, so that is its successor.



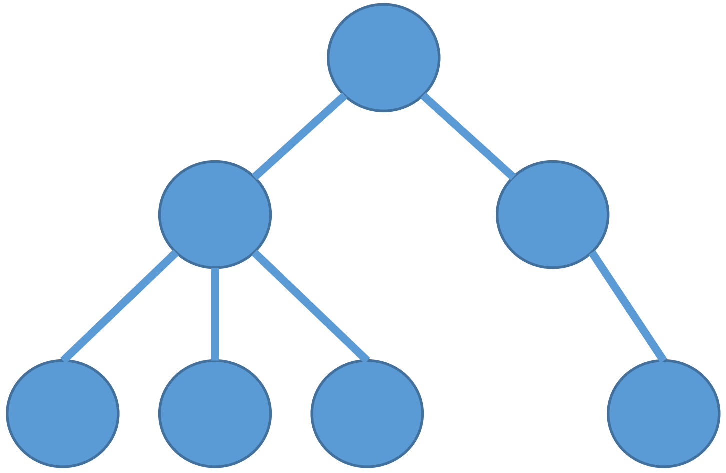
General Trees

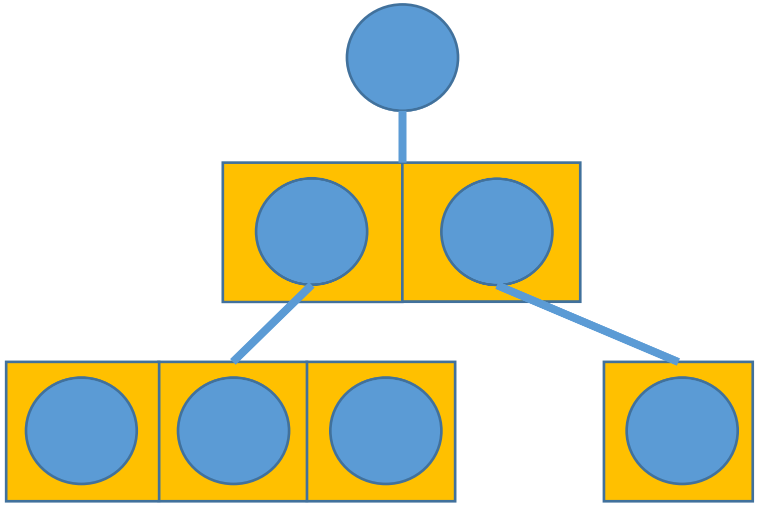
- A **general tree** (or **n-ary tree**) is a tree where each node may have any number of children.



General Trees

- Nodes don't have a fixed number of children (e.g., max 2 children in a binary tree).
- Instead, each node maintains a list structure of its children.



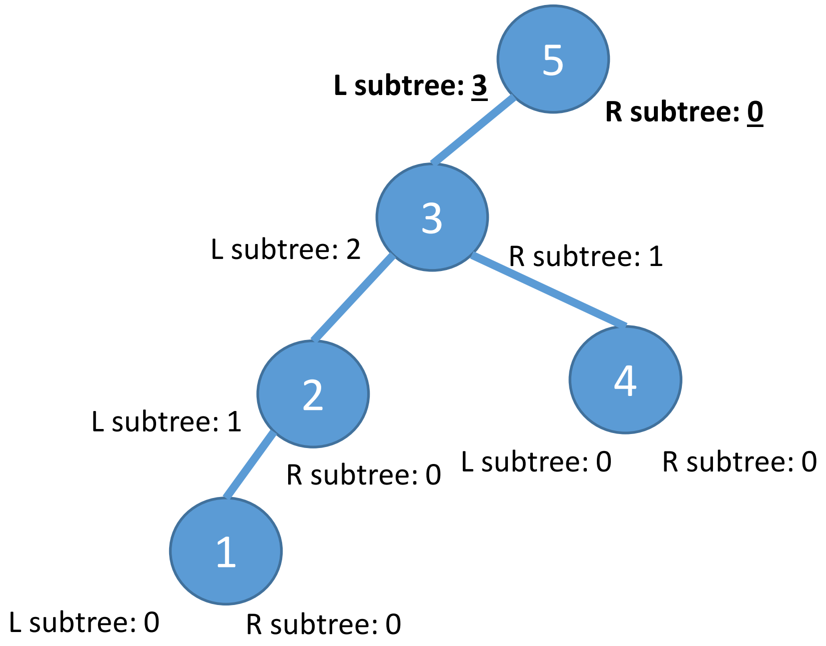


Tree Complexity

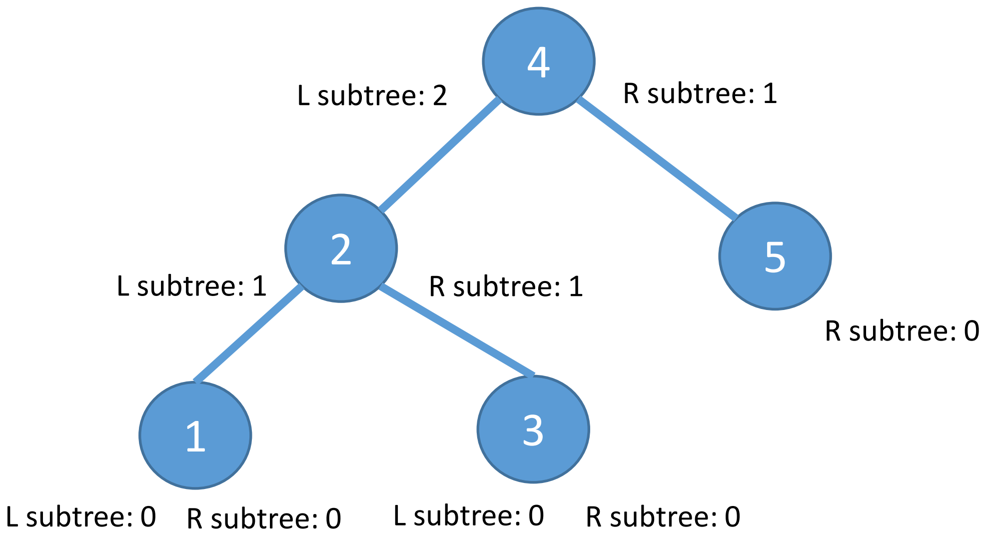
- Traversing any tree using depth-first or breadth-first traversals will have O(n) complexity as each node needs to be visited/processed.
  - Just as traversing an array or linked list.

- 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



Unbalanced BST

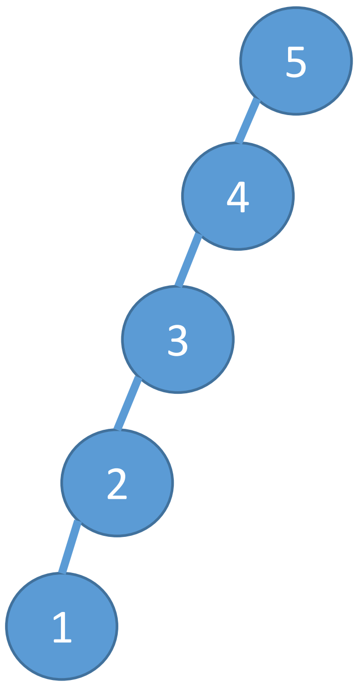


Balanced BST

<u>Every node's left subtree and right subtree must differ by <= 1</u>

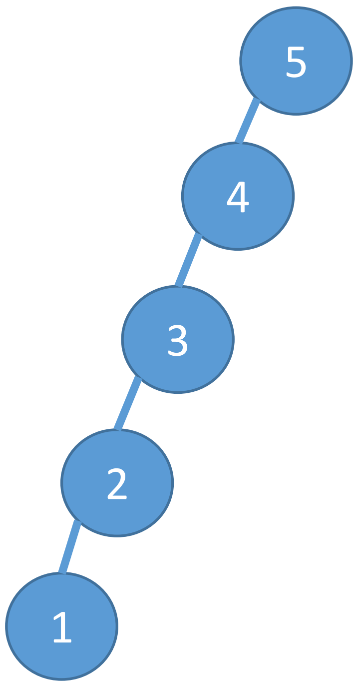
Tree Complexity

- 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 tree*.
  - Clearly unbalanced.



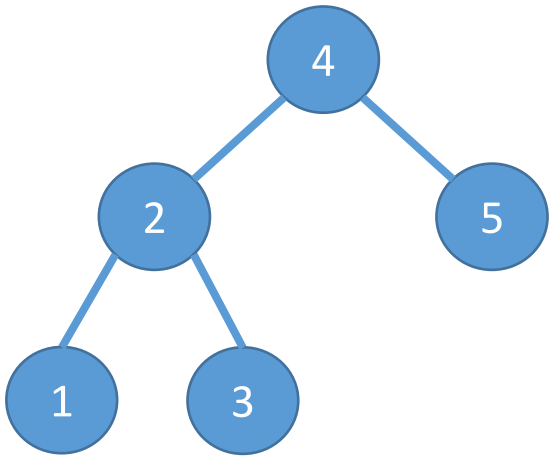
Tree Complexity

- 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 I no different than a singly linked list



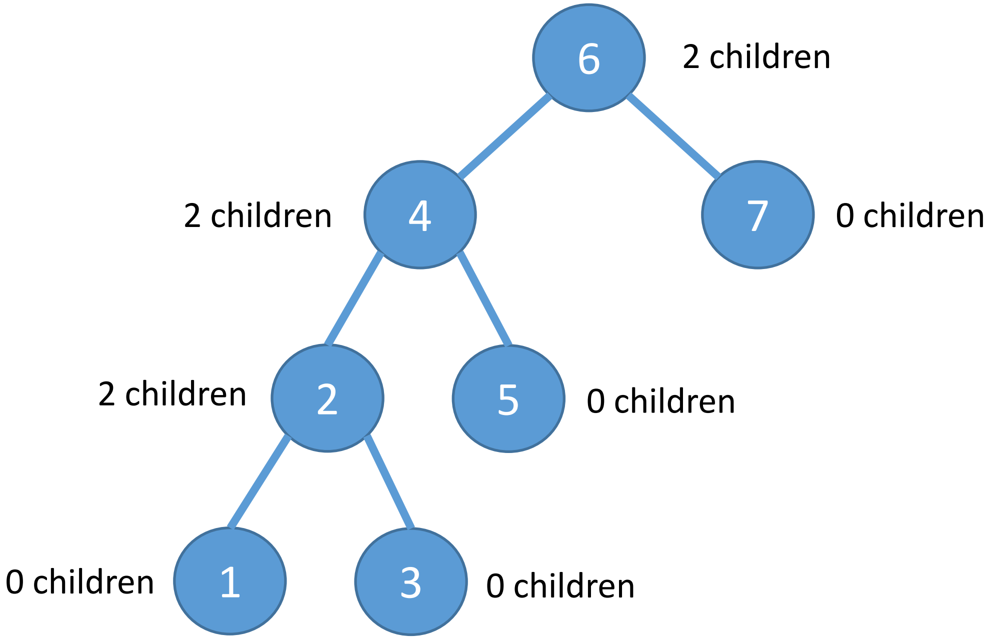
Tree Complexity

- Here are the 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.

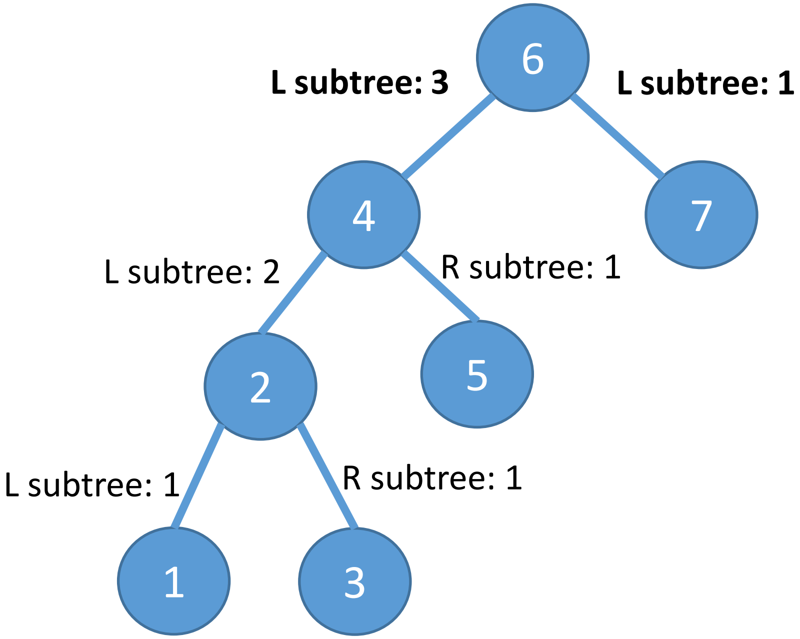


Full Binary Trees

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



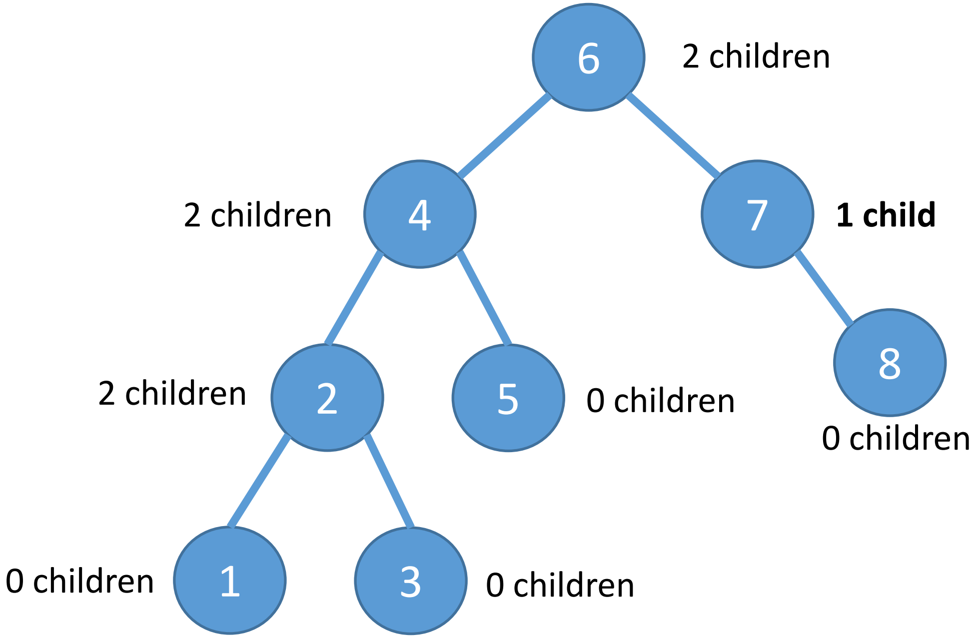
Full



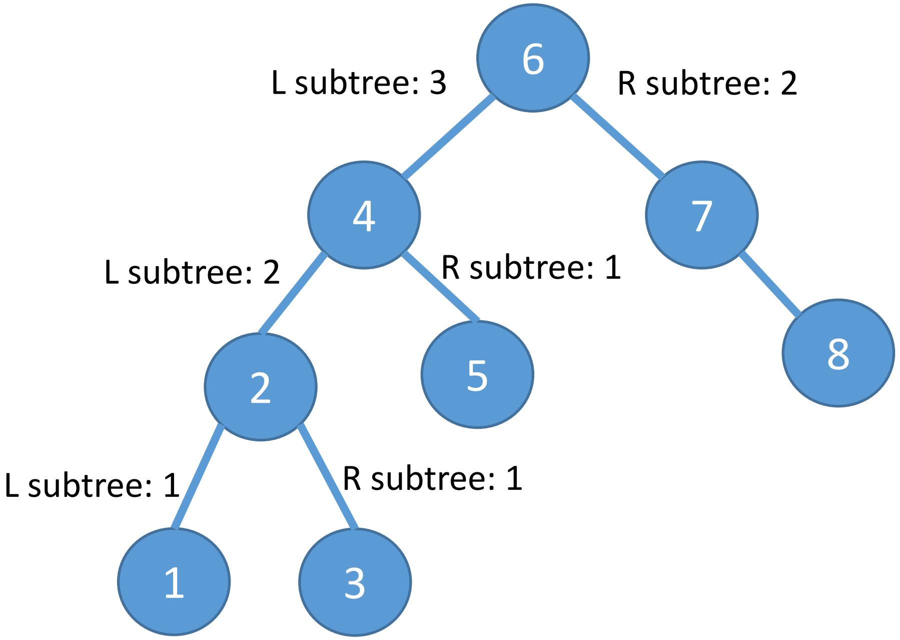
Unbalanced

Full Binary Trees

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



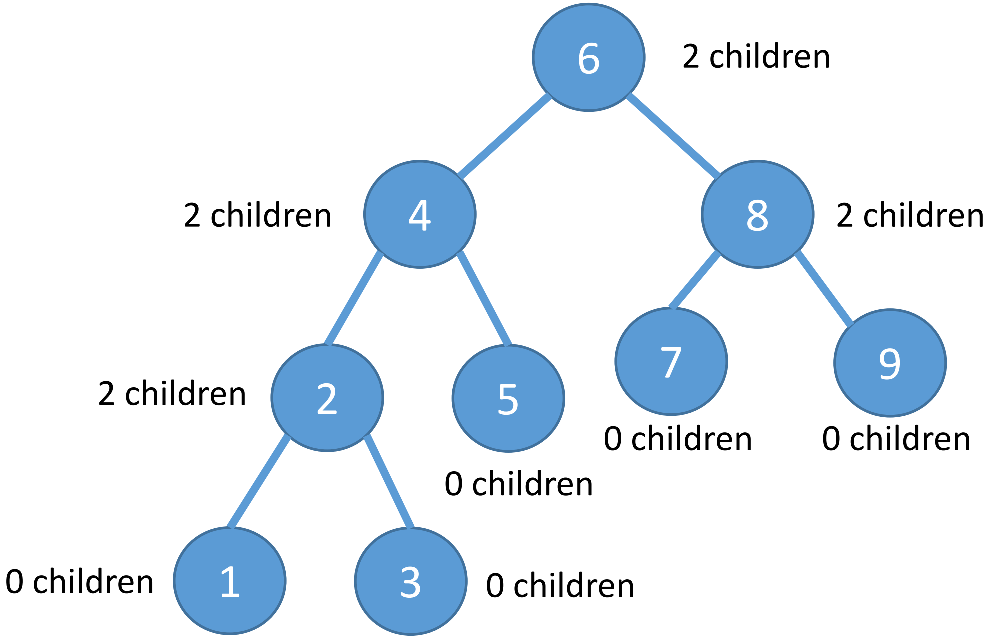
Not Full



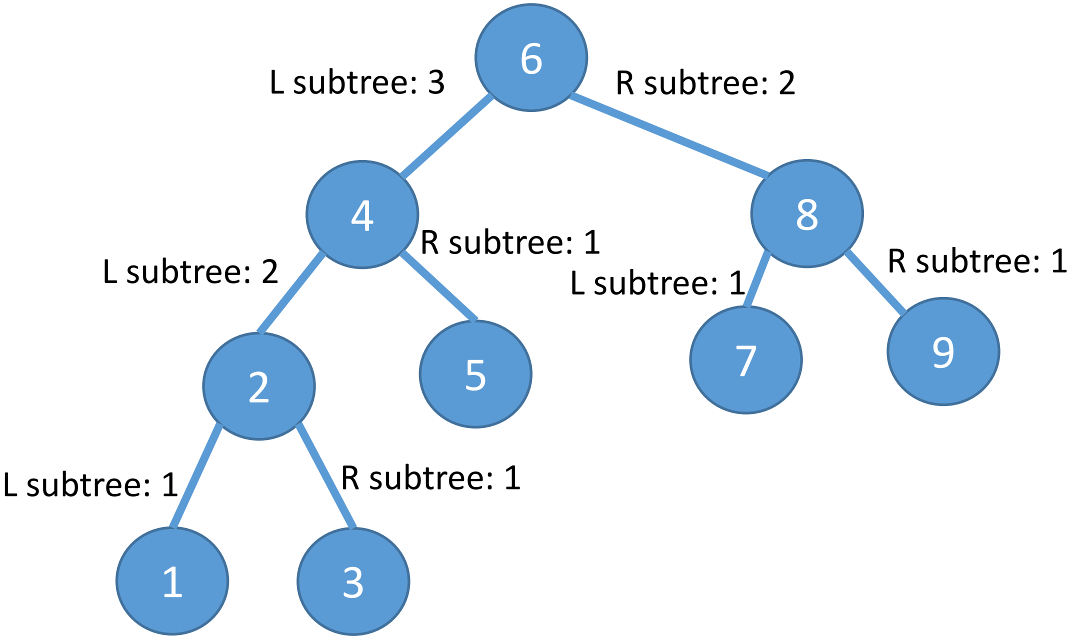
Balanced

Full Binary Trees

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



Full

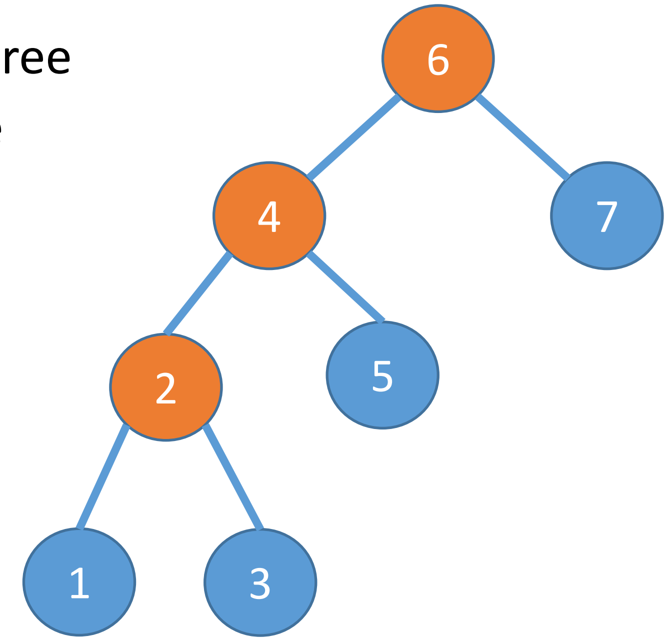


Balanced

Full Binary Trees

- Checking for a Full Tree
  - P + 1 = L
    - P is the number of parent nodes in the tree
    - L is the number of leaf nodes in the tree

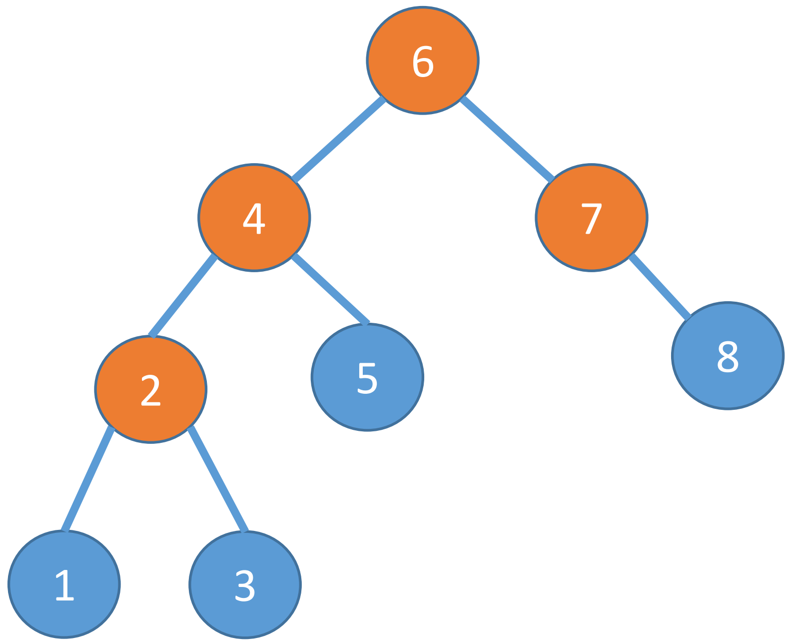
  - 3 + 1 = 4
  - 4 = 4 (**Full**)



Full Binary Trees

- Checking for a Full Tree
  - P + 1 = L

  - 4 + 1 = 4
  - 5 = 4 (**Not Full**)

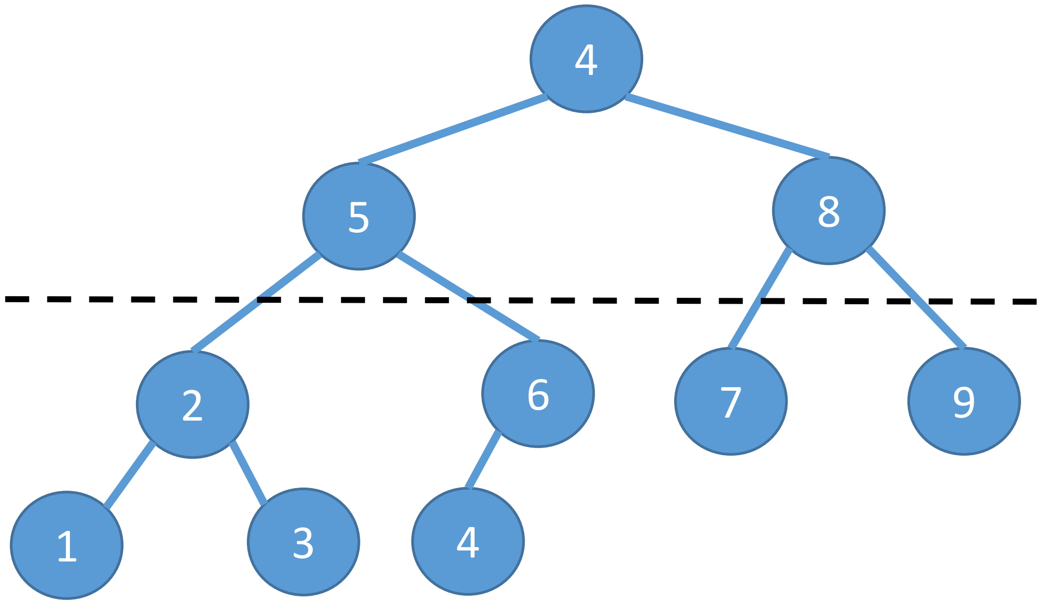


Complete Binary Trees

- 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 leaves are far left



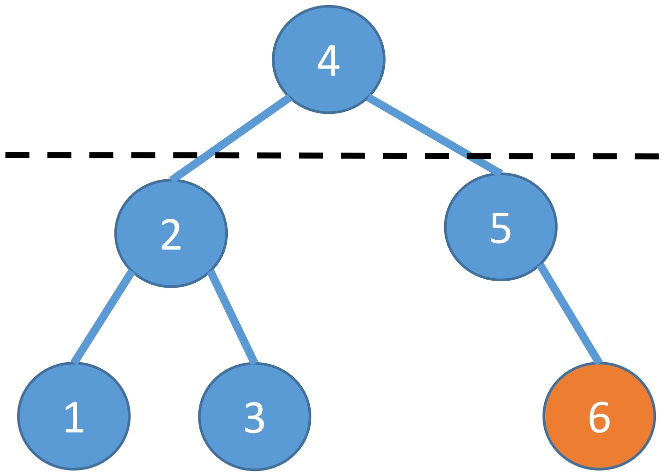
Balanced and Complete, but not Full

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 leaves are far left



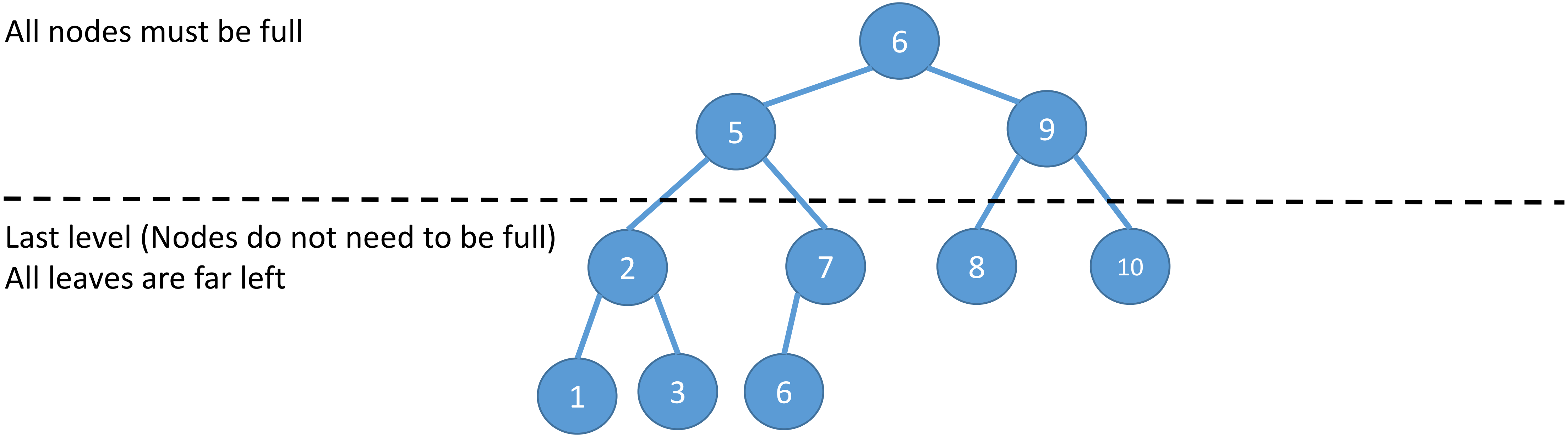
Balanced, not full, not complete

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 leaves are far left



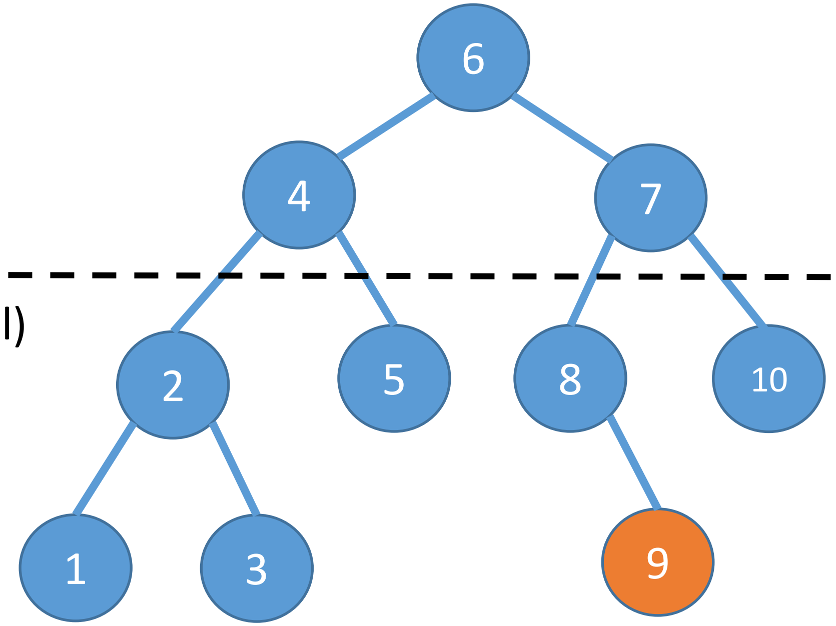
Balanced, not full, complete

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 leaves are far left



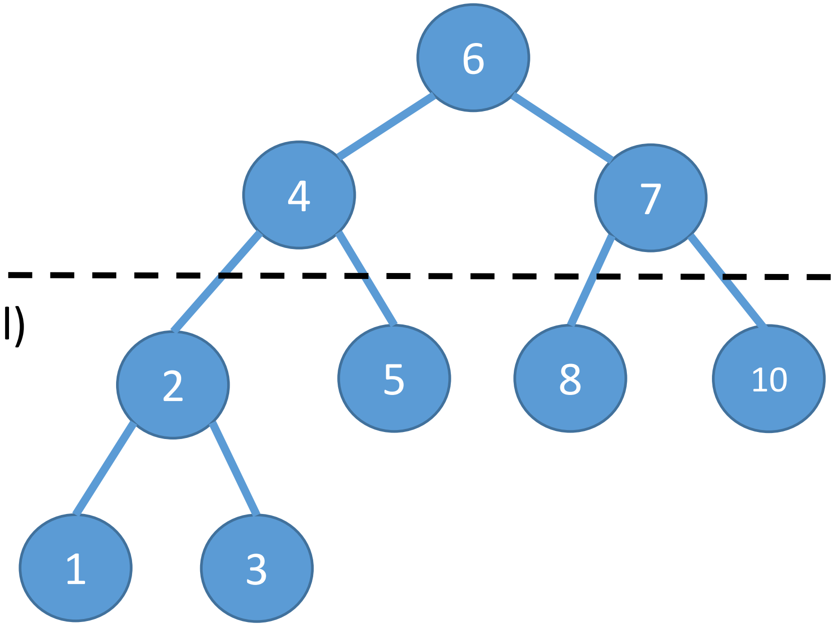
Balanced, not full, not complete

Complete Binary Trees

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

All nodes must be full

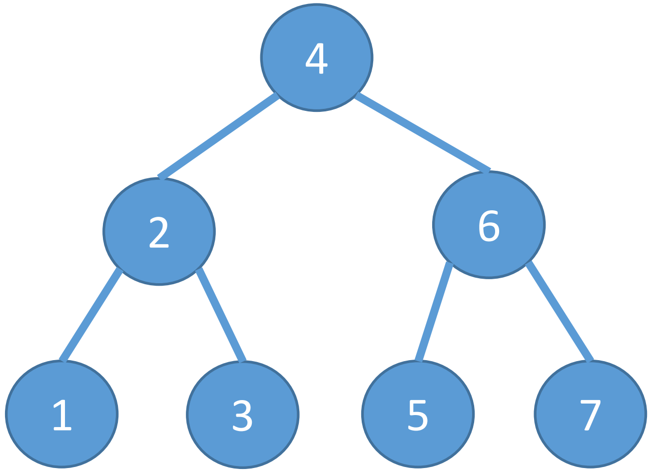
Last level (Nodes do not need to be full)
All leaves are far left



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

- Perfect BSTs perform in O(log n)
- Balanced BSTs will perform between O(log n) at best and O(h) at worst, depending on how balanced it is
- 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 BST	O(h)	O(h)	O(h)	O(h)	O(h)
Perfect BST	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)