Binary Tree Traversals

- When storing data in a binary tree (e.g. a binary search tree), we'll often want to perform some kind of operation or processing on all of the nodes of the tree.
    - For example, we might simply want to print the value stored at each node, or we might want to do something more complicated, like update the values, perform a calculation over all the values, etc.

- In general, we'll want to do this in a systematic way, perhaps even in a particular order. A ***tree traversal*** is a mechanism to help us achieve this. A tree traversal is specifically a method for visiting each node in a tree exactly once and performing some operation or processing at each node when it's visited.
    - Importantly, tree traversals are general methods and are not restricted to any particular kind of tree. However, here, we'll focus on tree traversals for binary trees.

- There are two high-level types of tree traversal:
    - ***Depth-first*** – A depth-first traversal explores a tree subtree by subtree, visiting all of a node's descendants before visiting any of its siblings (and their descendants). In other words, a depth-first traversal moves as far *downward* in the tree as it can go before moving *across* in the tree.
    - ***Breadth-first*** – A breadth-first traversal explores a tree level by level, visiting every node at a given depth in the tree before moving downward and traversing the nodes at the next-deepest level. In other words, a breadth-first traversal moves as far *across* the tree as it can go before moving *down* in the tree.

- We'll explore each of these kinds of traversal below.

Depth-first traversals

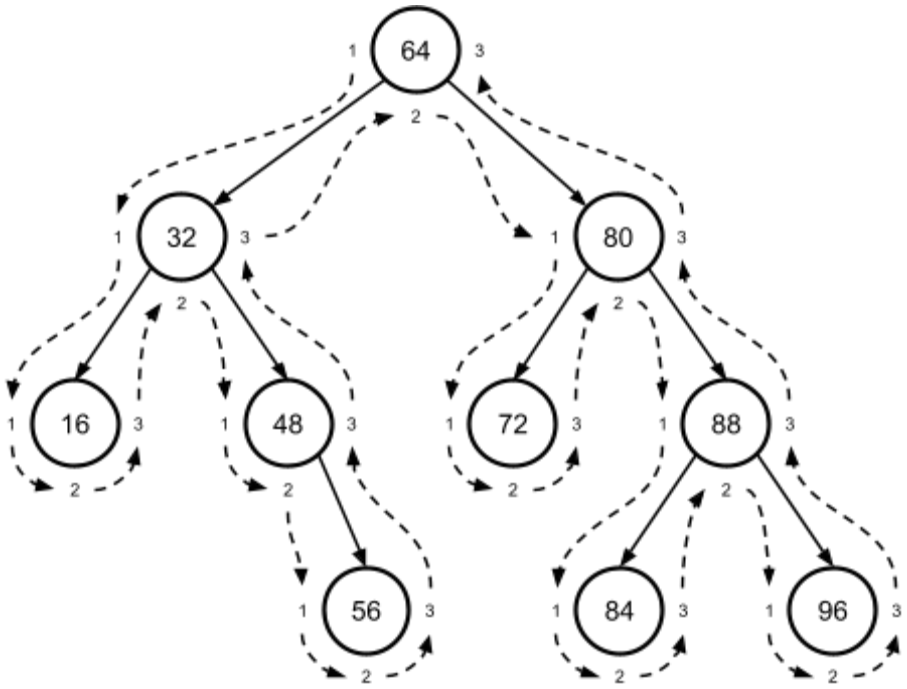
- A depth-first traversal explores a tree subtree by subtree, moving downward in the tree as far as possible before moving across in the tree.

- There are several different kinds of depth-first BST traversal. Each of the ones we'll explore here does the following three specific things at every node, which we'll denote using the letters N, L, and R:

- **N** – visit/process the current node itself
- **L** – traverse the left subtree of the current node
- **R** – traverse the right subtree of the current node

- Each specific kind of depth-first BST traversal we'll explore does these same three things, just in a different order. In general, we won't be concerned about traversals that go from right to left (i.e. in which R comes before L). That leaves us with the following three kinds of depth-first traversal:
  - ***Pre-order traversal*** – NLR – process the current node before traversing either of its subtrees
  - ***In-order traversal*** – LNR – traverse the current node's left subtree before processing the node itself, and then traverse the node's right subtree
  - ***Post-order traversal*** – LRN – traverse both of the current node's subtrees (left, then right) before processing the node itself

- We can visualize these traversals by imagining a process that "walks" around the perimeter of the tree, starting at the root node and proceeding counter-clockwise. During its "walk", this process will encounter each node exactly 3 times: (1) before walking either of the node's subtrees; (2) after walking the node's left subtree but before its right subtree; and (3) after walking the node's right subtree. This walk can be visualized as in the following example:



- In this visualization, the three visits to each node (1), (2), and (3) are depicted, with visit (1) occurring on the left of the node, visit (2) occurring at the bottom of the node, and visit (3) occurring on the right side of the node.

- The three kinds of depth-first BST traversal listed above correspond to these three visits to the nodes during this "walk" of the tree. Specifically:
    - A pre-order traversal performs processing at each node at visit (1).
    - An in-order traversal performs processing at each node at visit (2).
    - A post-order traversal performs processing at each node at visit (3).

- In the example BST depicted above, each traversal processes the tree's nodes in a different order.
    - Pre-order – 64, 32, 16, 48, 56, 80, 72, 88, 84, 96
    - In-order – 16, 32, 48, 56, 64, 72, 80, 84, 88, 96
    - Post-order – 16, 56, 48, 32, 72, 84, 96, 88, 80, 64

- Notice that the in-order BST traversal processes the nodes in sorted order. This will always be the case for a BST.
    - Understanding this should help clarify the notion of an in-order successor, which we encountered when exploring the remove operation in BSTs.

- Each of these traversals has a simple, elegant recursive implementation. Here, for example, is pseudocode for a recursive in-order BST traversal function, which is called on a node in the BST. The initial call is made on the BST's root node:

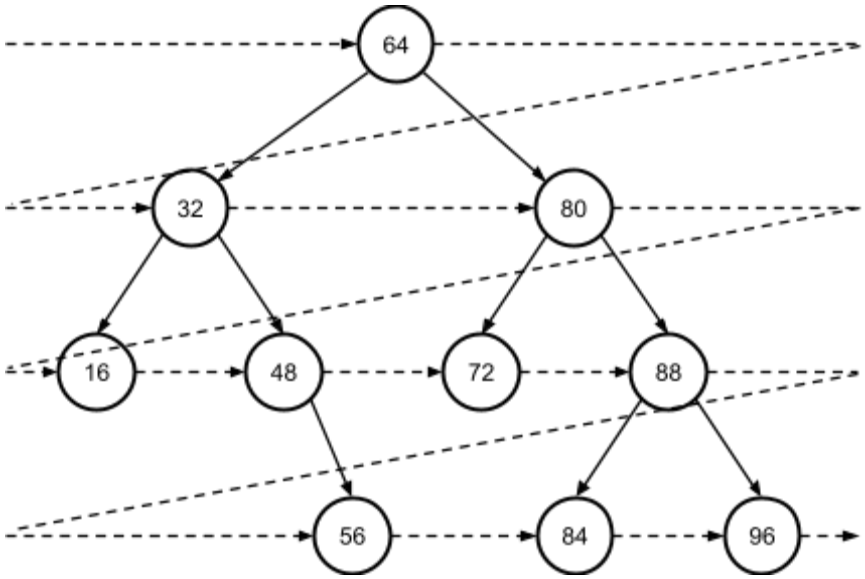
```
inOrder(N):
     if N is not NULL:
          inOrder(N.left)
          process N
          inOrder(N.right)
```

- The specific processing performed by this function at each node is left undefined here. It could be as simple as just printing out the key and/or value at each node, or it could involve more complex computations.

- A pre-order BST traversal and a post-order BST traversal have recursive implementations nearly identical to the one above. The pre-order traversal simply moves processing of **`N`** before the recursive call on **`N.left`**, while the post-order traversal moves processing of **`N`** *after* the recursive call on **`N.right`**.

Breadth-first traversal

- There is one main kind of breadth-first traversal, called a ***level-order traversal***, which processes a tree level by level. Specifically, a level-order traversal begins at the root node and proceeds by processing all of the nodes at the current depth *d* in left-to-right order before moving downward to process all of the nodes at depth *d+1* in left-to-right order:



- In this example, the nodes are specifically processed by a level-order traversal in this order: 64, 32, 80, 16, 48, 72, 88, 56, 84, 96.

- The implementation of a level-order traversal uses a queue to order the nodes:

```
levelOrder(bst):
     q ← new, empty queue
     enqueue(q, bst.root)
     while q is not empty:
          N ← dequeue(q)
          if N is not NULL:
               process N
               enqueue(q, N.left)
               enqueue(q, N.right)
```