# TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

# Unit 8 Cheatsheet

## Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as reference as you solve the breakout problems for Unit 8. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 8. In addition to the material below, you will expected to know any required concepts from previous units.

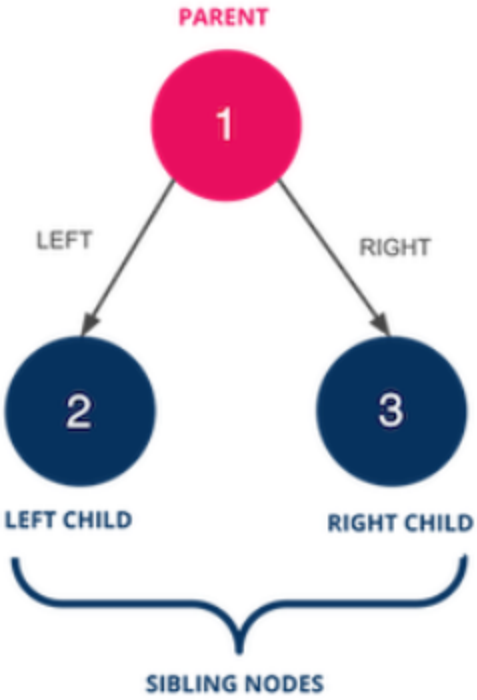
## Standard Concepts

### Binary Trees

Binary trees are a *non-linear* data structure. Similar to a linked list, trees are made up of nodes that store a piece of data as well as references to other nodes.

#### Binary Tree Structure

Each node in a binary tree can point to up to *two* other nodes in the tree. Each of these pointers is known as a **child** of the original **parent** node. The two children are referred to as the left and right children or pointers.




A `TreeNode` class for a binary tree may look like the following:

```
class TreeNode():
     def __init__(self, value, left=None, right=None):
         self.value = value
         self.left = left
         self.right = right
```

Example Usage:

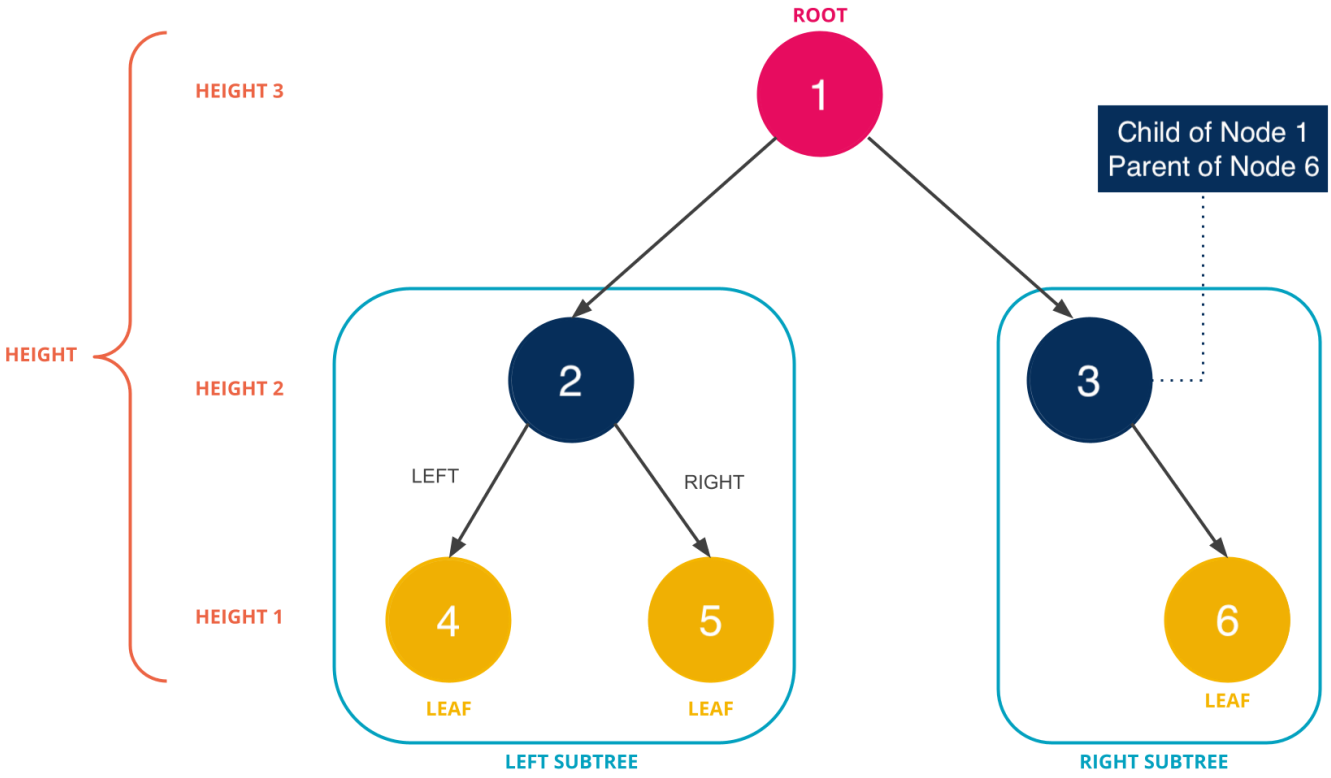
```
# Creates the following tree:
#   1
#  / \
# 2   3

node_one = TreeNode(1)
node_two = TreeNode(2)
node_three = TreeNode(3)

node_one.left = node_two    # Node two is the left child of node one
node_one.right = node_three # Node three is the right child of node one
```

## Tree Terminology

There is a lot of terminology used to discuss tree data structures. Here are some common ones!

- **root**: The top-most node in a tree.
- **leaf**: A node with no children.

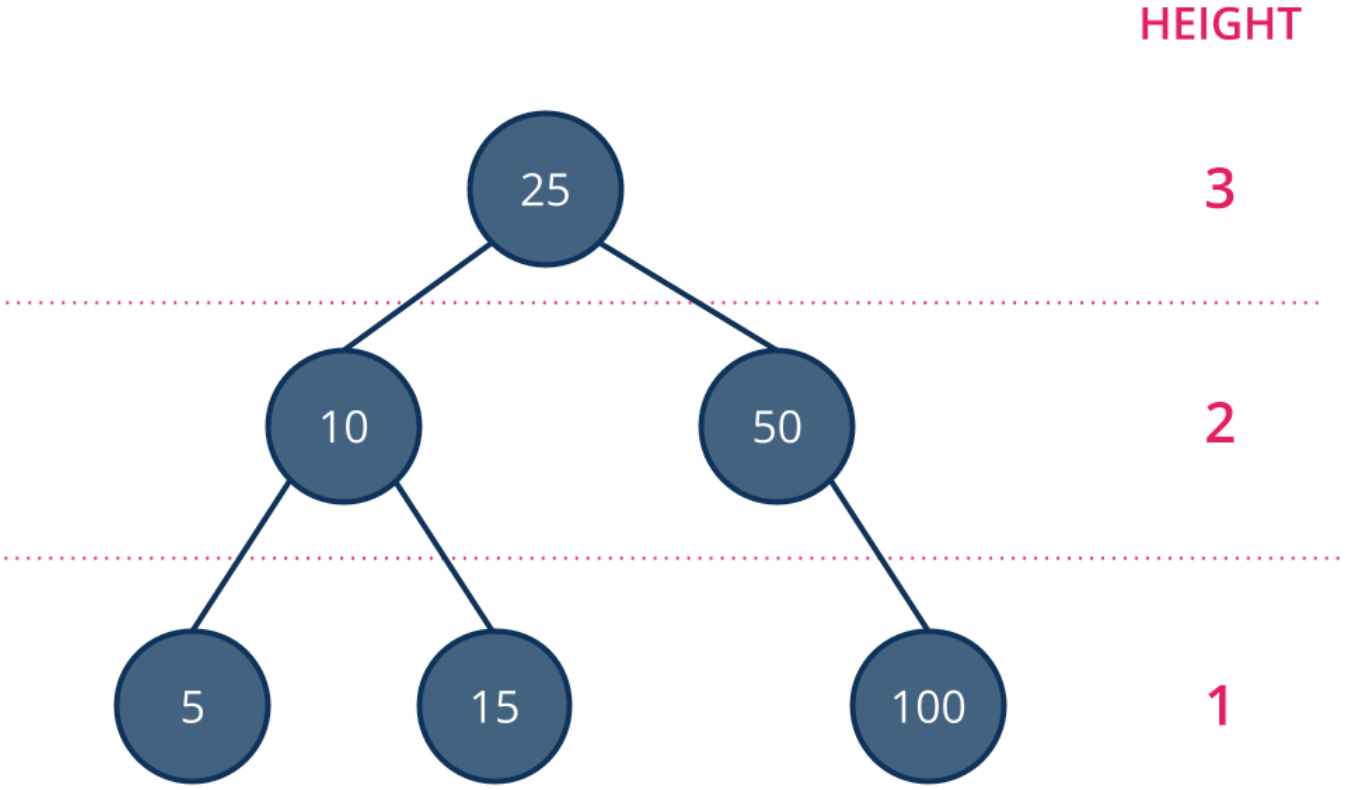
- **edge**: The reference or pointer (path) between a parent and child node.
- **height**: The maximum number of edges it takes to travel from a leaf node to the root node.
- **level**: All the nodes at the same height in the tree.
- **subtree**: The tree formed by any node, it's children, grandchildren, etc. that is not the root node.
- **ancestor**: A node with a greater height (closer to the root) on the path from the current node to the root node.
- **descendant**: A node in the current node's subtree(s).
- **siblings**: Two nodes that share a parent.



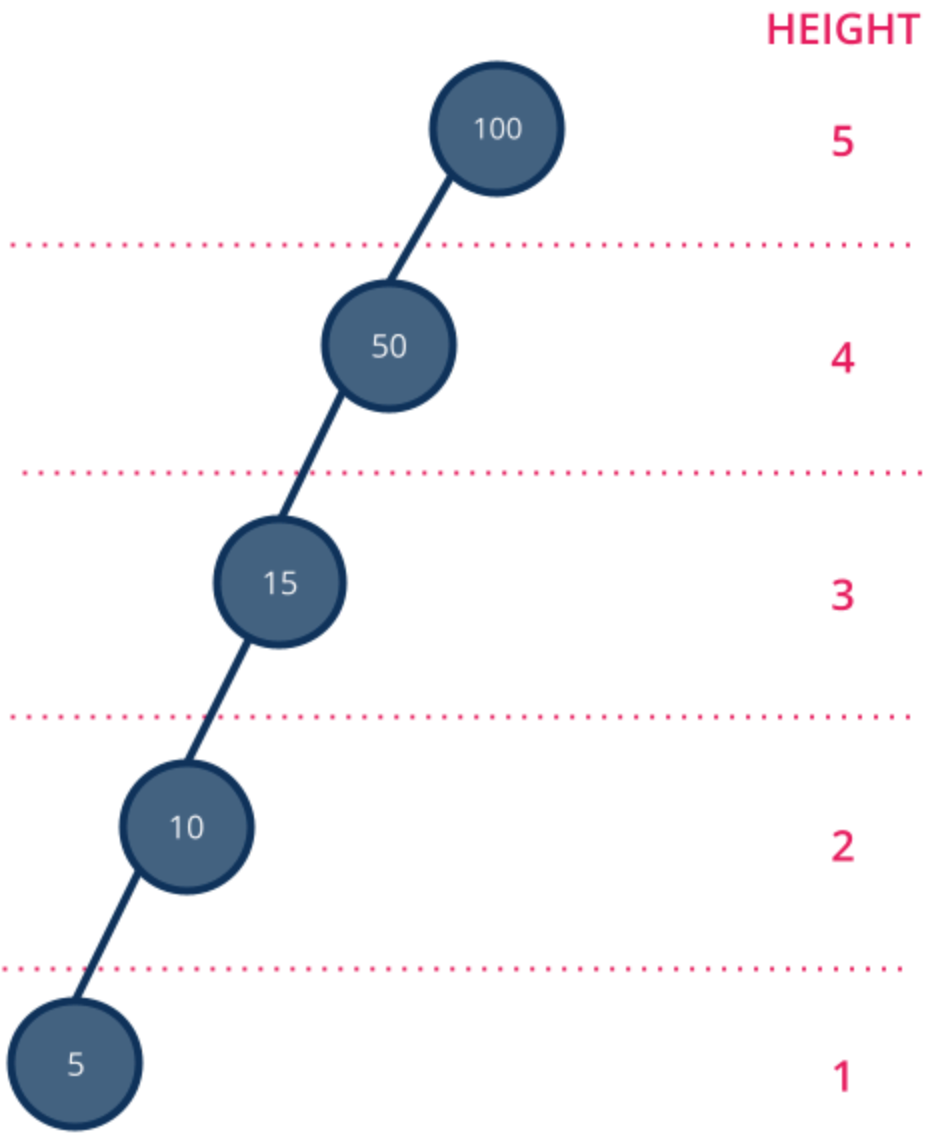

## Balanced Trees

A tree is considered balanced if the height difference between the left and right subtree is at most one and both subtrees are also balanced. In this way the nodes in the tree must be spread fairly evenly.

Below is an example of a balanced tree.




Below is an example of an unbalanced tree.




Whether or not a tree is balanced can affect the time complexity of our algorithms. It is usually much faster to reach the bottom of a tree or find a particular node in a balanced tree than it is in an unbalanced tree!

## Tree Traversal

Tree traversals almost always start from the root of the tree. But how do we decide which child we want to visit first? Now that we are working with a non-linear data structure, there are multiple paths that we can take to visit all the nodes of a tree. Oftentimes, the strategy you'll use will depend on the data you are trying to find or the operation you want to perform on the tree.

The most common type of traversal is a *depth first search* traversal. This means that we choose one of the root node's children to visit, then visit its grandchildren and any further descendants before ever exploring the root's second child and its descendants. In other words, we fully explore a single subtree before traversing the second subtree.

There are three standard types of depth first search tree traversals:

- **Preorder**: Current, Left, Right
- **Inorder**: Left, Current, Right

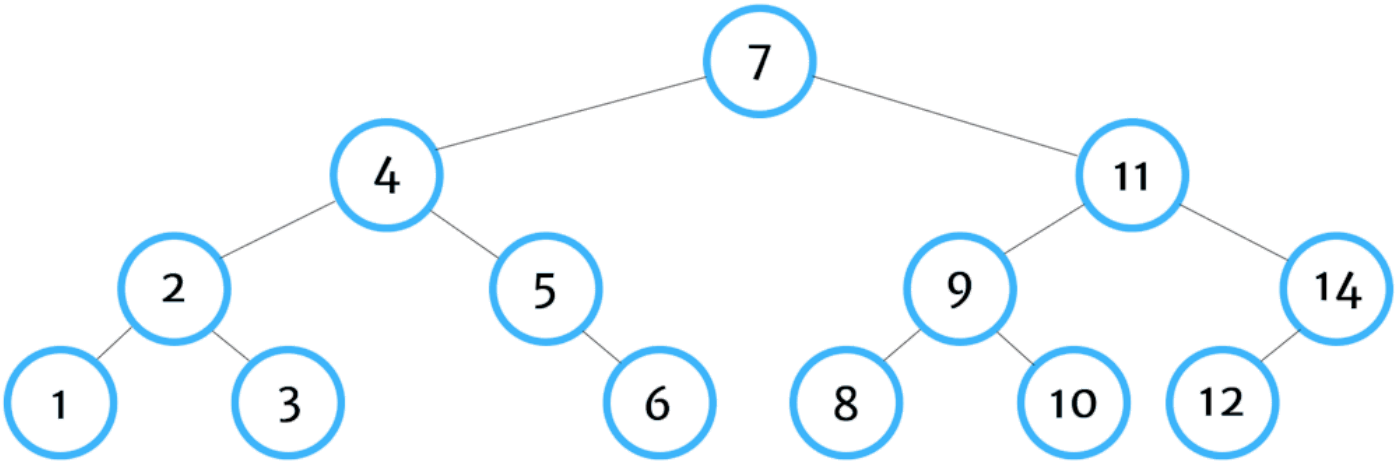
- **Postorder**: Left, Right, Current

**Preorder**

A preorder traversal works as follows:

```
visit the current node
traverse the current node's left subtree
traverse the current node's right subtree
```

Preorder traverses nodes in the order they were added to the tree, so it is commonly used to save a copy of the tree structure to memory or send it across a network.




Source: via Tyler Willis

When we start our traversal, the current node is our `root` node. So we start by visiting node `7` in the example above. Then preorder says to traverse its left subtree, by updating the current node to the `root`'s left child node `4`. Now we start the process over as if `4` is the root of its own tree.

This means that per preorder protocol we visit the current node first so we visit node `4`. Then preorder tells us to traverse the current node `4`'s left subtree. So we update the current node to node `2`.

Once again, we visit the current node, node `2`, then visit it's left subtree, by updating the current node to node `2`'s left child, node `1`.

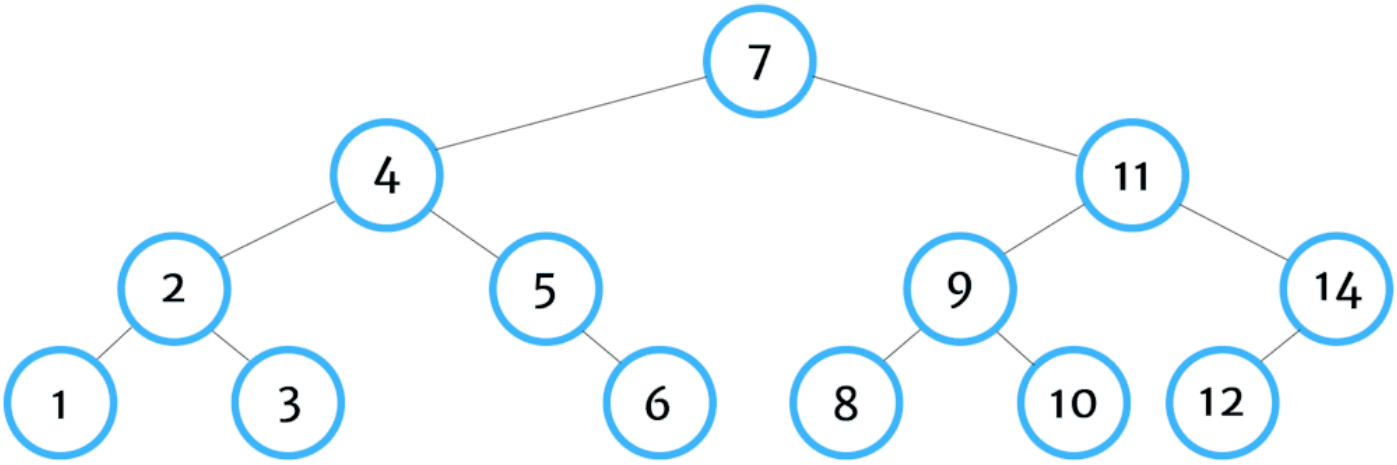
We visit the current node `1` first, and then go to visit it's left subtree. Since it has no left subtree, we move on to the next step which is to visit the right subtree. It has no right subtree either, so we know we've hit a leaf node and move back up to node `1`'s parent node, node `2`.

Now that we've traversed node `2`'s complete left subtree, we move to traverse it's right subtree and visit node `3`. We will again check if node `3` has left and right children, find that it doesn't, and return to node `2`. We've traversed node `2`'s left and right subtrees at this point, so we have done a complete preorder traversal of the subtree formed by node `2`, and we move back to node `4` so we can traverse its right subtree. The process continues until we have done a complete traversal of the `root` node `7`'s left and right subtrees.

**Inorder**

```
traverse the current node's left subtree
visit the current node
traverse the current node's right subtree
```

Inorder begins by traversing the leftmost node and ends by traversing the rightmost node. In a binary search tree which is a specific type of binary tree that maintains nodes in a specific order, this means that nodes are processed from smallest to largest, so it can be useful to print nodes in sorted order.




Source: via Tyler Willis

When we start our traversal, the current node is our `root` node. So we start by going to node `7`'s left subtree. The root of node `7`'s subtree is `4`, so we update the current node to node `4`.

Now we start the inorder pattern over as if `4` was the root of its own tree. So we visit `4`'s left subtree and update the current node to node `2`.

Again, per inorder protocol, we visit node `2`'s left subtree and update the current node to node `1`.

We attempt to visit node `1`'s left subtree but find it doesn't have one. The next step in the inorder pattern is to visit the current node, so we mark node `1` as visited. Then we attempt to visit node `1`'s right subtree but find it doesn't have one. Now we have completed an inorder traversal of the subtree formed by node `1`, so we go back to it's parent node, node `2`.

We've now visited node `2`'s complete left subtree, so we can move on to the next step in an inorder traversal to visit node `2` itself. Then we move to visit its right subtree, node `3`.

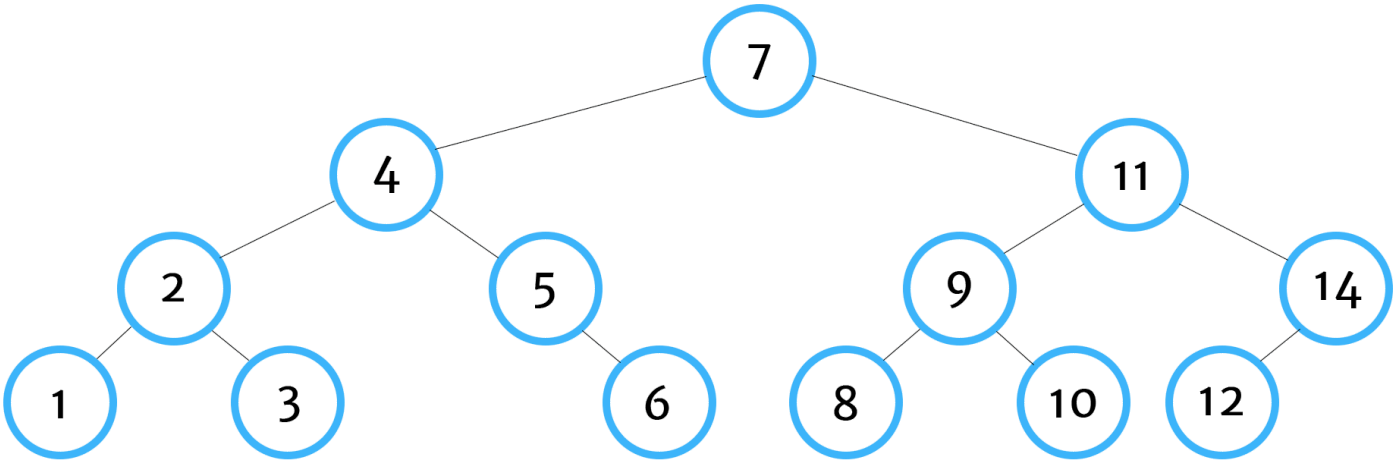
Node `3` doesn't have a left subtree, so we move on to marking node `3` as visited, find it also doesn't have a right subtree and so go back to its parent node, node `2`.

We have now done a complete inorder traversal of the subtree formed by node `2`, and we move back to node `4` so we can traverse its right subtree. The process continues until we have done a complete traversal of the `root` node `7`'s left subtree, node `7` itself, and node `7`'s right subtrees.

**Postorder**

```
traverse the current node's left subtree
traverse the current node's right subtree
visit the current node
```

Postorder traverses all of the root node's descendants before processing the root itself. This makes it useful for deleting a binary tree.




Source: via Tyler Willis

When we start our traversal, the current node is our `root` node. So we start by going to node `7`'s left subtree. The root of node `7`'s subtree is `4`, so we update the current node to node `4`.

Now we start the postorder pattern over as if `4` was the root of its own tree. So we visit `4`'s left subtree and update the current node to node `2`.

Again, per postorder protocol, we visit node `2`'s left subtree and update the current node to node `1`.

We attempt to visit node `1`'s left subtree but find it doesn't have one. The next step in the postorder pattern is to visit the current node's right subtree. We find that node `1` also does not have a right subtree, so we move to the final step which is to visit the current node, node `1`, itself. Now we have completed an inorder traversal of the subtree formed by node `1`, so we go back to it's parent node, node `2`.

We've now visited node `2`'s complete left subtree, so we can move on to the next step in an postorder traversal and visit its right subtree, node `3`.

Node `3` doesn't have a left subtree or a right subtree, so we mark it as visited and move back to the parent node, node `2`. Now that we have traversed node `2`'s left and right subtrees, we can mark node `2` itself as visited.

We have now done a complete postorder traversal of the subtree formed by node `2`, and we move back to node `4` so we can traverse its right subtree. The process continues until we have done a complete traversal of the `root` node `7`'s left and right subtrees, and visit node `7` itself.

▼ 💡 **Why do we traverse left to right?**

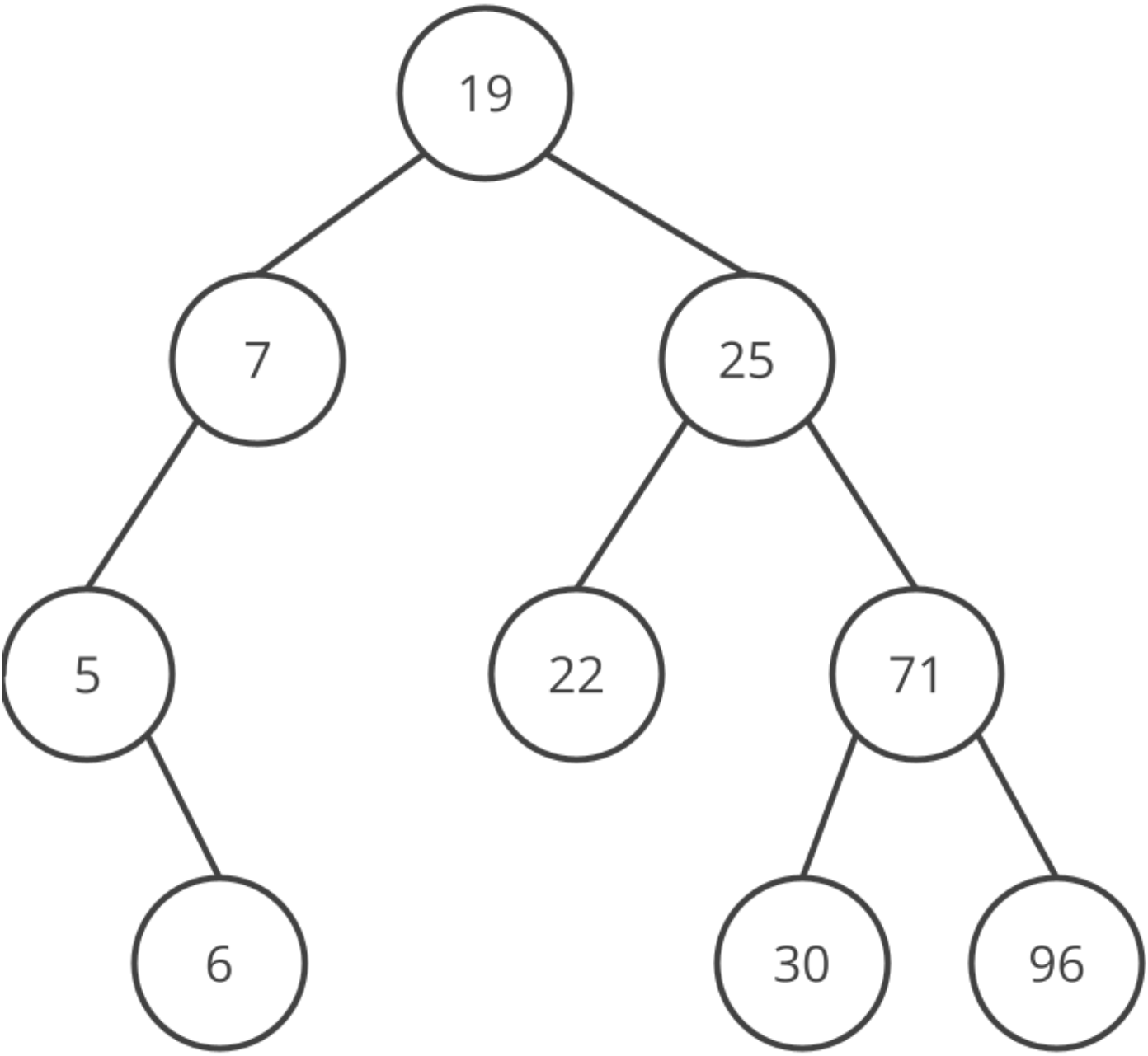
Computer science as a field began in western cultures where people read from left to right. This led to a cultural bias when designing traversal algorithms. There is no logical reason we can't traverse trees from right to left!

## `nary Search Trees

A binary search tree (BST) is a very common type of binary tree that organizes nodes in a specific way that makes searching, inserting, and deleting nodes from the tree very efficient ( `O(log n)` time complexity).

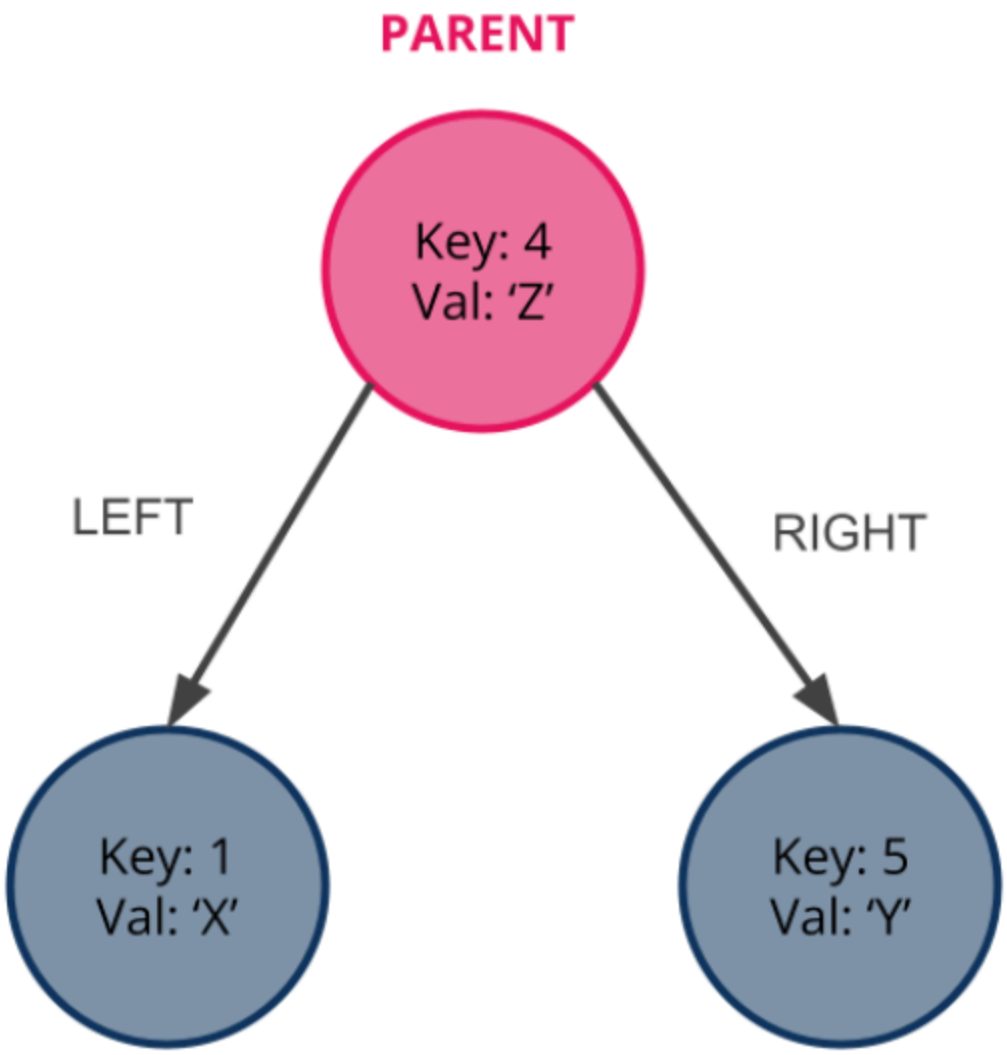
A binary search tree maintains the following rules:

- All nodes in the left subtree of a node contain values less than the node's value
- All nodes in the right subtree of a node contain values greater than the node's value




In the tree above, we can see that all nodes in the root's right subtree are greater than 19, and all nodes in the root's left subtree are less than 19. The same properties hold for any subtree we consider. All nodes in node 25's right subtree are greater than 25, and the only node in its left subtree is less than 25.

The `TreeNode` class for a binary search tree often maintains an additional property `key` . The `key` is an integer used to maintain the sorted order of the tree: all nodes in the left subtree must have a smaller key and all nodes in the right subtree must have a greater key. The `value` is then free to be any piece data (a string, list, another integer) irrespective of the key.




```
class TreeNode():
     def __init__(self, key, value=0, left=None, right=None):
        self.key = key
        self.value = value
        self.left = left
        self.right = right
```

Example Usage:

```


# Creates the following tree :
# By Key:        By Value:
#     1              Yoda
#    / \            /   \
#   2   3         Luke  R2D2


z = Node(2, 'Yoda')
x = Node(1, 'Luke')
y = Node(3, 'R2D2')
```

### Binary Search Tree Operations

We can use the properties of a binary search tree to perform insert, delete, and search operations faster.

Let's take the search operation for example. Because the nodes are maintained in sorted order, every time we visit a node we can compare the key of the node we are visiting to the key of the node we are searching for. If the node is not the node we are looking for, we can determine whether our target node will be in the current node's left or right subtree.

If our target node's key is less than the current node's key, we should look in the left subtree. If it is greater, we should look in the right subtree.

In this way, with each node we visit, we halve the amount of remaining nodes we need to look through (assuming the tree is balanced). Notice this is just like the binary search algorithm we did on sorted lists from Unit 7!

Example Search Implementation:

```
class TreeNode():
     def __init__(self, key, value=0, left=None, right=None):
        self.key = key
        self.value = value
        self.left = left
        self.right = right

def search_bst(node, key):
    # Base case: node is None or we find the key
    if node is None or node.key == key:
        return node

    # If the key to be found is less than the current node's key, search in the left subtree
    if key < node.key:
        return search_bst(node.left, key)

    # If the key to be found is greater than the current node's key, search in the right subtree
    return search_bst(node.right, key)
```

## Bonus Concepts & Syntax

The following concepts are nice to know and may improve your code readability or help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit. These concepts are **not in scope for either the Standard or Advanced Unit 8 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

- **Throwaway Variable** Used to ignore values. Commonly used when a function returns multiple values but the user is only interested in one or when the loop variable is not needed inside the body of the for loop.
- **Inner Functions** Specialized Python syntax often used to create helper functions