

# Calculating the Height of a Binary Tree

In this lesson, you will learn how to calculate the height of a binary tree.

## We'll cover the following

- Height of Tree
  - Height of Node
- Algorithm
- Implementation

## Height of Tree #

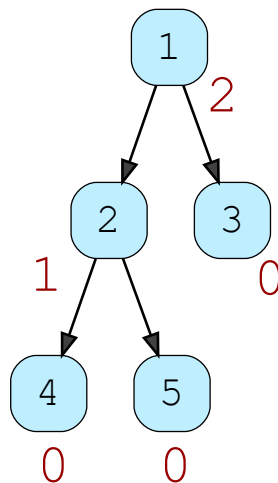
Let's start by defining the height of a tree. The height of a tree is the height of its root node. Now let's see what we mean by the height of a node:

## Height of Node #

The height of a node is the number of edges on the longest path between that node and a leaf. The height of a leaf node is 0.

Recursively defined, the height of a node is one greater than the max of its right and left children's height.

Below is an example of a binary tree labeled with heights of individual nodes:



Height Of the Tree = 2 (Height of the Root Node)

## Algorithm #

In this lesson, we will consider the recursive approach to calculate the height of a tree. The idea is to break down the problem using recursion and traverse through the left and right subtree of a node to calculate the height of that node. Once we get the height of the left and right subtree, we will consider the maximum of the two heights plus one to be the height of the tree.

## Implementation #

Let's move to the implementation in Python:

```
def height(self, node):  
    if node is None:  
        return -1  
    left_height = self.height(node.left)  
    right_height = self.height(node.right)  
  
    return 1 + max(left_height, right_height)
```

height(self, node)

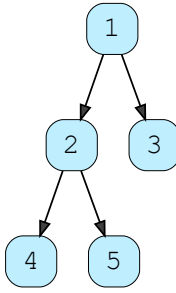
In the code above, our base case is when `node` equals `None`. If `node` equals `None`, we return `-1` on **line 3** as we have gone past the leaf nodes. Once a leaf

node discovers that its left and right children are reporting heights of **-1** each, it will add **1** to **-1** and return **0** as its height.

In **lines 4-5**, we recursively call the **height** method on the left child and the right child. The final height is calculated by adding **1** to the maximum height of the left and right subtree, as height is the longest path between that node and the leaf node. The final height is what is returned from the method.

Let's run this code step by step in the illustration below:

```
def height(node):  
    if node is None:  
        return -1  
    left_height = height(node.left)  
    right_height = height(node.right)  
  
    return 1 + max(left_height, right_height)
```



Node	left_height	right_height

1 of 39



I hope the visuals are helpful to understand the algorithm.

In the code widget, the **height** method is made part of the **BinaryTree** Class. Write your test cases to verify the **height** method. A sample test case has been given to you.

```
class Stack(object):  
    def __init__(self):  
        self.items = []  
  
    def __len__(self):  
        return self.size()  
  
    def size(self):  
        return len(self.items)  
  
    def push(self, item):  
        self.items.append(item)  
  
    def pop(self):  
        if not self.is_empty():  
            return self.items.pop()
```



```
        return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def __str__(self):
        s = ""
        for i in range(len(self.items)):
            s += str(self.items[i].value) + "-"
        return s
```

```
class Queue(object):
    def __init__(self):
        self.items = []

    def __len__(self):
        return self.size()

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def size(self):
        return len(self.items)

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value
```

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

```
class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def search(self, find_val, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_search(self.root, find_val)
        elif traversal_type == "inorder":
            return self.inorder_search(self.root, find_val)
        elif traversal_type == "postorder":
            return self.postorder_search(self.root, find_val)
        else:
            print("Traversal type " + str(traversal_type) + " not recognized.")
            return False
```

```

def print_tree(self, traversal_type):
    # Recursive traversals
    if traversal_type == "preorder":
        return self.preorder_print(tree.root, "")
    elif traversal_type == "inorder":
        return self.inorder_print(tree.root, "")
    elif traversal_type == "postorder":
        return self.postorder_print(tree.root, "")

    # Iterative traversals
    elif traversal_type == "levelorder":
        return self.levelorder_print(tree.root)
    elif traversal_type == "inorder_iterative":
        return self.inorder_iterative(tree.root)
    elif traversal_type == "preorder_iterative":
        return self.preorder_iterative(tree.root)
    elif traversal_type == "postorder_iterative":
        return self.postorder_iterative(tree.root)
    else:
        print("Traversal type " + str(traversal_type) + " not recognized.")
        return False

def levelorder_print(self, start):
    if start is None:
        return
    queue = Queue()
    queue.enqueue(start)

    traversal = ""
    while len(queue) > 0:
        traversal += str(queue.peek()) + "-"
        node = queue.dequeue()

        if node.left:
            queue.enqueue(node.left)
        if node.right:
            queue.enqueue(node.right)

    return traversal

def preorder_search(self, start, find_val):
    if start:
        if start.value == find_val:
            return True
        else:
            return self.preorder_search(start.left, find_val) or \
                   self.preorder_search(start.right, find_val)
    return False

def preorder_print(self, start, traversal):
    """Root->Left-Right"""
    if start:
        traversal += (str(start.value) + "-")
        traversal = self.preorder_print(start.left, traversal)
        traversal = self.preorder_print(start.right, traversal)
    return traversal

def inorder_print(self, start, traversal):
    """Left->Root->Right"""
    if start:
        traversal = self.inorder_print(start.left, traversal)
        traversal += (str(start.value) + "-")

```

```

        traversal = self.inorder_print(start.right, traversal)
    return traversal

def postorder_print(self, start, traversal):
    """Left->Right->Root"""
    if start:
        traversal = self.postorder_print(start.left, traversal)
        traversal = self.postorder_print(start.right, traversal)
        traversal += (str(start.value) + "-")
    return traversal

def height(self, node):
    if node is None:
        return -1
    left_height = self.height(node.left)
    right_height = self.height(node.right)

    return 1 + max(left_height, right_height)

# Calculate height of binary tree:
#      1
#     / \
#    2   3
#   / \
#  4   5
#
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.height(tree.root))

```



This sums up our content on Binary Trees. Get ready as we have a challenge regarding binary trees waiting for you in the next lesson.