Assignment Overview
Binary Search Trees are often used instead of linear structures, such as arrays or linked lists, because most operations on BSTs are O(height). In this project, you will be implementing a Binary Search Tree class to store integers.
You are given Project6_main() with some tests. Since this file will not be submitted, you can and are encouraged to edit the file and do your own testing. You are also given a skeleton file, BinarySearchTree.py, that contains a Node class and other functions that you will complete.

Assignment Deliverables
Be sure to use the specified file name and to submit your file for grading **via D2L Dropbox** before the project deadline.

- BinarySearchTree.py

Assignment Specifications
Complete the specified functions below, with the given time complexities.

The **Node class** is included in the skeleton file with the following functions and **should not be edited.**

- __init__(self, value, parent=None, left=None, right=None)
  This function initializes a node with a given value. Optional arguments are the parent node, left child, and right child.
- __eq__(self, other)
  This provides comparison ('==') for nodes. If two nodes have value, they are considered equal.
- __repr__(self)
  A node is represented in string form as 'value'. Use str(node) to make into a string.

The **Binary Search Tree class** is partially completed in the skeleton file. **Function signatures or provided functions may not be edited in any way.**
The following functions are provided and may be used as needed.

- __init__(self)
  A BST class has variables root and size. Self.root is the node that is at the root of the tree. If the tree is empty, then self.root is None. Self.size is the number of nodes in the tree.

- __eq__(self, other) and _compare(self, t1, t2)
  These functions are used to compare two BSTs. Two BSTs are equal if all subtrees are equal. They will primarily be used for grading purposes - DO NOT CHANGE.

You must complete and implement the following functions. Take note of the specified return values and input parameters. **Do not change the function signatures.**

- **insert(self, value)**
  This function takes value and inserts it into the BST. If the value already exists in the tree, nothing is changed. When inserting new nodes, lesser values are inserted on the left and greater values are inserted on the right. Nothing is returned.
  Time complexity: O(height)
- **remove(self, value)**
  This function finds and removes a node with the given value. It will be helpful to review the 3 cases of node removal from Lecture 15. When removing a node with 2 children, replace with the minimum of the right subtree. Nothing is returned.
  Please spend time testing and error-checking this function.
  Time complexity: O(height)
- **search(self, value, node)**
  Starting from the specified node, this function recursively searches the subtree for the given value. If the value is found, return the node. If the value is not found, return the potential parent node (nearest leaf node). See Lecture 15 notes and below for examples. If the tree is empty, return None.
  Time complexity: O(height)
- **inorder(self, node, inorder_list)**
  This recursive function performs an in-order traversal of the BST starting at the specified node, and stores the values of the nodes in the parameter inorder_list. The list of values is returned.
  Time complexity: O(n)
- **preorder(self, node, preorder_list)**
  Same as inorder(), but does a pre-order traversal of the BST.
  Time complexity: O(n)
- **postorder(self, node, postorder_list)**
  Same as inorder(), but does a post-order traversal of the BST
  Time complexity: O(n)
- **depth(self, value)**
  This function finds the depth of the node with the specified value. The height of the node is returned. If the value is not in the tree, return -1.
- **height(self, node)**
  This function finds and returns the height of the BST.
  Time complexity: O(n)

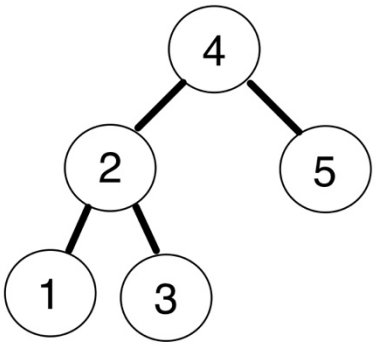
- **min(self, node)**
  Starting from the specified node, this function moves recursively through the subtree to find the node with the minimum value. That node is returned. If the tree is empty, return None.
  Time complexity: O(height)
- **max(self, node)**
  Starting from the specified node, this function moves recursively through the subtree to find the node with the maximum value. That node is returned. If the tree is empty, return None.
  Time complexity: O(height)
- **get_size(self)**
  This function gets and returns the size of the BST.
  Time complexity: O(1)

Assignment Notes

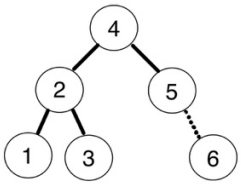
Points will be deducted if your solution has any warnings of type.

- Do not modify the Node class.
- You are provided skeleton code for the BinarySearchTree class and must complete the included methods. You may add more functions in BinarySearchTree class than what is provided**, but may not alter the function signatures in anyway**.
- To test your classes, Project6_main.py is provided and uses some of the required functions. Compare your results to the output on the next page.
- Pre and Post conditions are required on all functions.

Example for search():
Starting with the tree on the left, search for value 6 starting from the root node.



Since 6 is not in the tree, we find the node that would be the parent of 6.



In this case, node with value 5 would be the parent, so that is the node that is returned by search(6).

Results of project6_main.py:

```
-----TEST 1-----
In-order:    [2, 4, 5, 7, 9, 12]
Pre-order:   [5, 4, 2, 9, 7, 12]
Post-order:  [2, 4, 7, 12, 9, 5]
Root:        5
Height:      2

-----TEST 2-----
In-order:    [2, 3, 4, 5, 7, 8, 9]
Root:        5

-----TEST 3-----
In-order:    [2, 3, 6, 7, 8, 9]
Pre-order:   [6, 2, 3, 9, 7, 8]
Post-order:  [3, 2, 8, 7, 9, 6]
Root:        6
Height:      3

-----TEST 4-----
Depth(8):    3
Depth(6):    0
Depth(1):    -1
Height:      3
Min:         2
Max:         9
Size:        6
```