Data Structures

Homework Assignment 7 - Binary Search Tree

Problem 1 – Binary Search Tree - 20 Points
Problem 2 – Binary Search Tree (A/B) - 30 Points
Problem 3 – Binary Search Tree Builder - 20 Points
Problem 4 - Find Pairs - 30 Points

25% of Gradescope Autograder test cases are hidden for this assignment.

Fall 2024 - Revision 1 - *November*

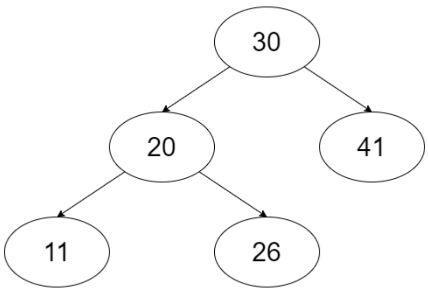
Problem 1 – Binary Search Tree (20 Points)

Given a binary search tree implementation, the member function *is_bst(self),* which returns *True* if a binary tree is a binary search tree. Return *False* otherwise. You cannot modify the input. Your function has to be non-recursive.

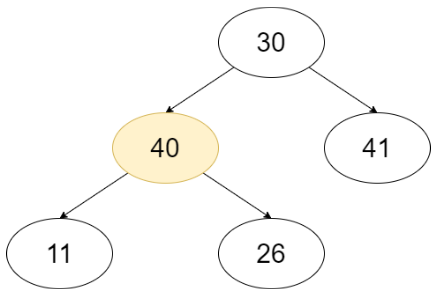
Requirements

- The time complexity requirement of this method is at most $O(n)$
- The space complexity requirement of this method is at most $O(n)$
- Your function has to be non-recursive
- You cannot use lists or any other data structures except queue and tuple
- You are not allowed to use any tree traversal function
- You are not allowed to create a new BinaryTree or BinarySearchTree instance

Example



Valid BinarySearchTree. Function returns true



Invalid BianrySearchTree. Function returns false

Problem 2 – Binary Search Tree (40 Points)

Given a binary search tree implementation, add the following functions:

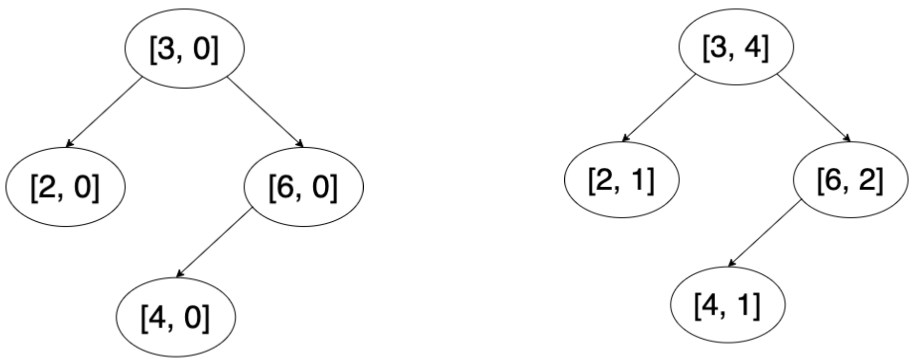
Part A - Calculate Subtree Size - 20 Points

Implement the function *subtree_size(self)*, which determines the size of every subtree in a given binary search tree. Each node's element is a list of two numbers having the following format: *[<node element value>, <size>]. In the beginning, all sizes are zero.* The size of a subtree is defined as the number of left and right children of a tree plus itself.

Requirements

- The time complexity requirement of this method is at most $O(n)$
- The space complexity requirement of this method is at most $O(n)$

Example



Part B - Convert to Binary Search Tree - 20 Points

Implement the function *convert_to_bst(self),* which converts a provided binary tree into a binary search tree without changing the structure of the tree.

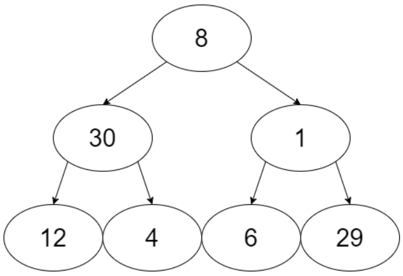
Requirements

- The time complexity requirement of this method is at most $O(nlogn)$
- The space complexity requirement of this method is at most $O(n)$
- You are not allowed to create a new BinaryTree or BinarySearchTree instance

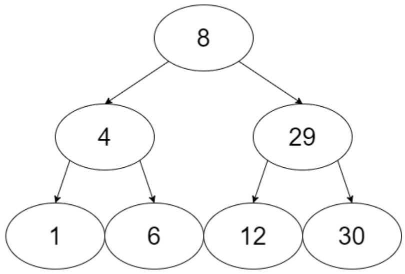
Information

- The time complexity of the Python sort function for list is *O(n*log*n)*
- There are no duplicated elements in the given binary tree.
- The connection of nodes via edges defines the structure of a tree.
- The structure can also be understood as the shape of a tree with its subtrees.
- To solve this problem, you should not rearrange nodes in the given tree.

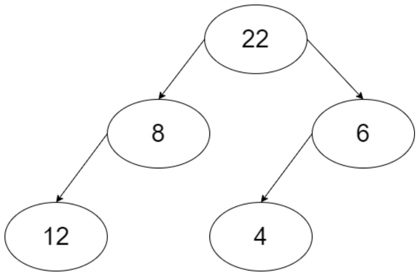
Examples



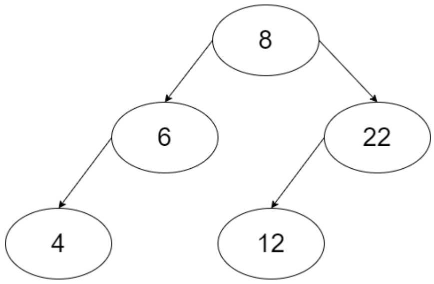
A binary tree with randomly ordered elements



The same binary tree as a binary search tree



A binary tree with randomly ordered elements



The same binary tree as a binary search tree

Problem 3 – Binary Search Tree Builder (30 Points)

Implement the member function *same(self, i1, i2)*, which verifies whether two sets of keys build the same binary search tree without building a binary search tree. Return *True* if both sets describe the same tree. Return *False* otherwise.

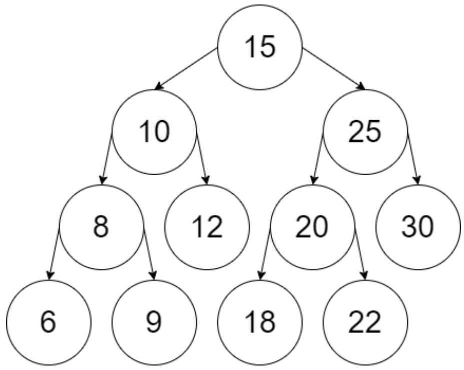
Requirements

- The time complexity requirement of this method is at most $O(n^2)$
- The space complexity requirement of this method is at most $O(n^2)$

Example

```
# You can find this tree on the
right
i1 = [15, 25, 20, 22, 30, 18, 10, 8,
9, 12, 6]
i2 = [15, 10, 12, 8, 25, 30, 6, 20,
18, 9, 22]

res = Solution().same(i1,i2)
print(res)  # Should print true
```



Problem 4 – Find Pairs (30 Points)

Part A - Find sum In Array - 10 Points

Implement the function *array_sum(self, arr, sum1),* which finds the sum *sum1* of two numbers in a given array. Your function has to return a tuple as the first elements forming the sum. The left tuple element is smaller than the right tuple element. The provided array is sorted, and all values are unique.

Requirements

- The time complexity requirement of this method is at most $O(n)$
- The space complexity requirement of this method is at most $O(1)$
- You cannot use Python lists or any other built-in data structures
- Your function has to be recursive

Example

```
arr = [1, 8, 9, 12, 23, 54, 88]
res = Solution().array_sum(arr, 55)
print(res) # Should print (1, 54)
```

Part B - Find Sum In Tree - 20 Points

Implement the recursive member function *pairs(self, sum1)*, which finds node pairs for a provided sum in a binary search tree. Return the first possible pair if a pair exists. Return *None* otherwise. Please note that the binary search tree member functions *before* and *after* have an amortized time complexity of *O(1)*. A node's successor or predecessor is usually its direct child or parent node. Only in exceptional cases is a subtree upward/downward traversal of *O(h)* required, where *h* is the tree's height.

Requirements

- The time complexity requirement of this method is at most $O(n)$
- The space complexity requirement of this method is at most $O(1)$
- You cannot use Python lists or any other built-in data structures
- Your function has to be recursive

Example

```
res = Solution().pairs(41)
print(res) # Should print (30, 11) or (11, 30)

res = Solution().pairs(100)
print(res) # Should print None

res = Solution().pairs(37)
print(res) # Should print (11, 26) or (26, 11)
```