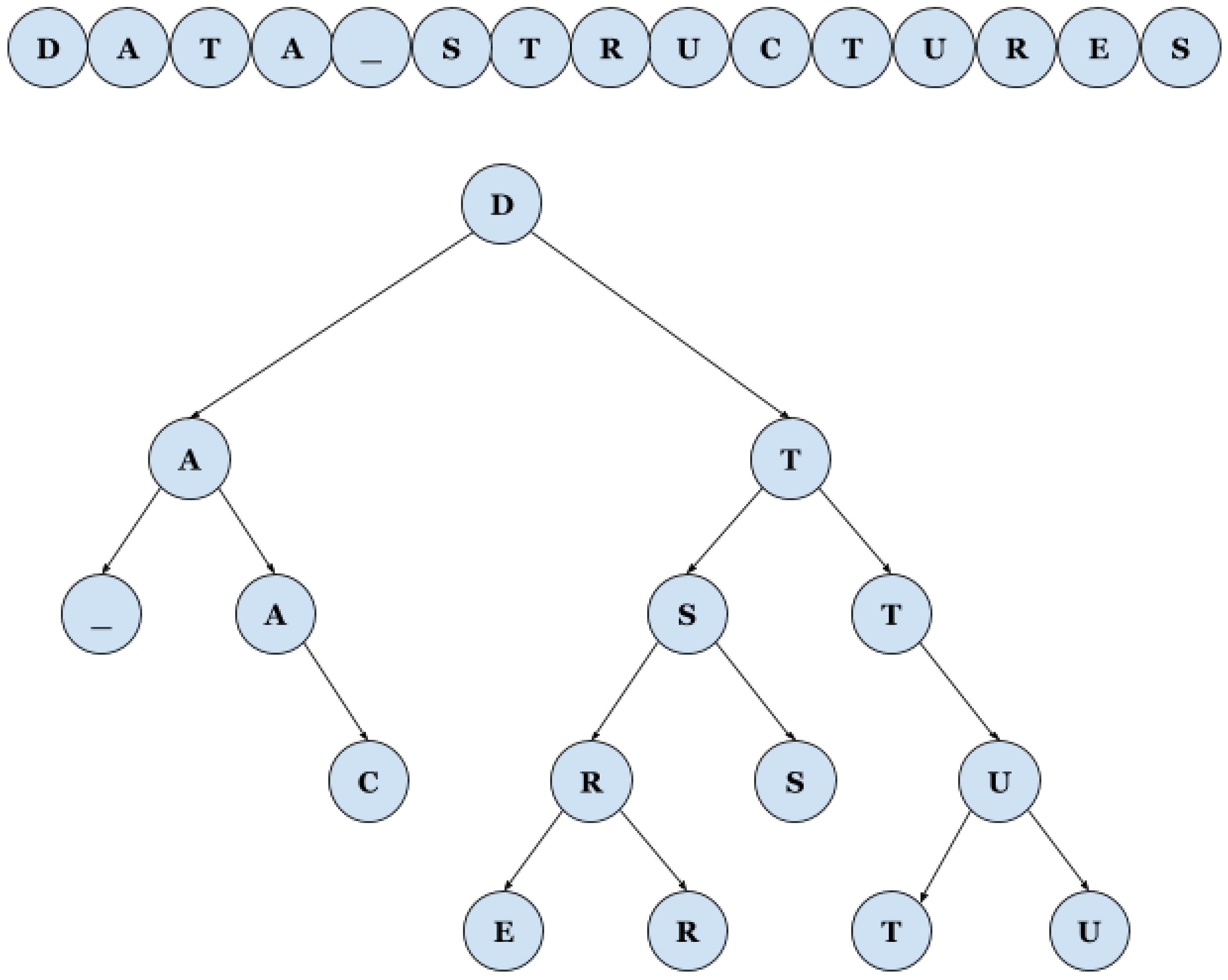
CS261 Data Structures

Assignment 4

v 1.02 (revised 5/10/2020)

Your Very Own Binary Search Tree



Contents

General Instructions 3

Part 1 - Binary Search Tree Implementation

Summary and Specific Instructions 4
add() 5
contains() 6
get_first() 6
remove() 7
remove_first() 8
pre_order_traversal() 10
post_order_traversal() 11
left_child() 12

General Instructions

1. Your project must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. In Gradescope, your code will run through several tests. Any failed tests will provide a brief explanation of testing conditions to help you with troubleshooting. Your goal is to pass all tests.

2. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

3. You are encouraged to develop your own additional tests even though this work won't have to be submitted and won't be graded. **You are not permitted to share any additional tests with other students.** Understanding the functionality of methods and developing unit tests is part of the assignment and must be done individually. Gradescope tests are limited in scope and may not cover all edge cases. Your submission must work on all valid inputs. We reserve the right to test your submission with more tests than Gradescope.

4. Your code must have an appropriate level of comments.

5. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in skeleton code must retain their names and input / output parameters. Variables defined in skeleton code must also retain their names. We will only test your solution by making calls to methods defined in the skeleton code and by checking values of variables defined in the skeleton code. You can add more methods and variables, as needed. However, certains classes and methods can not be changed in any way. Please see comments in the skeleton code for guidance.

6. **Both the skeleton code and code examples** provided in this document **are part of assignment requirements**. They have been carefully selected to demonstrate requirements for each method. Refer to them for the detailed description of expected method behavior, input / output parameters, and handling of edge cases. Code examples may include assignment requirements not explicitly stated elsewhere.

7. For each method, you can choose to implement a recursive or iterative solution. When using a recursive solution, be aware of maximum recursion depths on large inputs. We will specify the maximum input size that your solution must handle.

Part 1 - Summary and Specific Instructions

1. Implement a BinarySearchTree class by completing provided skeleton code. Once completed, your implementation will include the following methods:

   ```
   add()
   contains()
   get_first()
   remove()
   remove_first()
   pre_order_traversal()
   post_order_traversal()
   left_child()
   ```

2. Refer to the skeleton code and input / output examples below for each method for detailed description of described behaviour, input / output parameters etc. Please note that provided examples have been carefully selected and may include assignment requirements not explicitly stated elsewhere.

3. We will test your implementation with different types of objects, not just integers. For objects, you must implement __eq__, __lt__, __gt__, and __str__. (See the Student class in the skeleton code).

4. The number of objects stored in the tree will be between 0 and 900, inclusive.

5. Tree must allow for duplicate values. When comparing objects, values less then current node must be put in the left subtree, values greater than or equal to current node must be put in the **right subtree**.

6. When removing a node, replace it with the leftmost child of the right subtree (aka in-order successor). If the deleted node only has the left subtree, replace the deleted node with the root node of the left subtree.

7. Variables in TreeNode() class are not private. You are allowed to access and change their values directly. You do not need to write any getter or setter methods for the TreeNode() class.

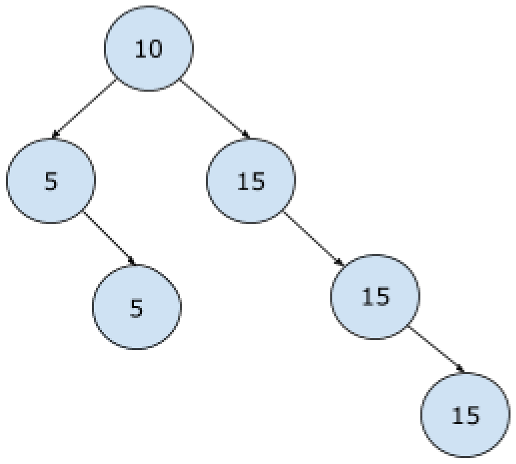
8. Name of the file for Gradescope submission must be bst.py

add(self, new_value: object) -> None:

This method adds new value to the tree, maintaining BST property. Duplicates must be allowed and placed in the right subtree.

Example #1:

```
tree = BST()
print(tree)
tree.add(10)
tree.add(15)
tree.add(5)
print(tree)
tree.add(15)
tree.add(15)
print(tree)
tree.add(5)
print(tree)
```

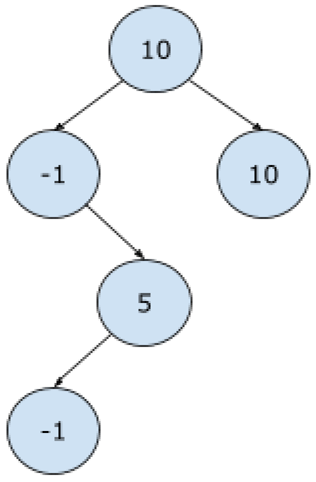


Output:

```
TREE in order {  }
TREE in order { 5, 10, 15 }
TREE in order { 5, 10, 15, 15, 15 }
TREE in order { 5, 5, 10, 15, 15, 15 }
```

Example #2:

```
tree = BST()
tree.add(10)
tree.add(10)
print(tree)
tree.add(-1)
print(tree)
tree.add(5)
print(tree)
tree.add(-1)
print(tree)
```



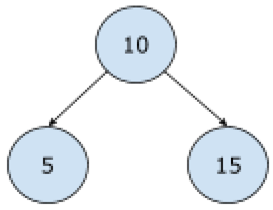
Output:

```
TREE in order { 10, 10 }
TREE in order { -1, 10, 10 }
TREE in order { -1, 5, 10, 10 }
TREE in order { -1, -1, 5, 10, 10 }
```

contains(self, value: object) -> bool:

This method must return True if the value parameter is in the BinaryTree or False if it is not in the tree. Empty trees do not contain any objects.

Example #1:



```
tree = BST([10, 5, 15])
print(tree.contains(15))
print(tree.contains(-10))
print(tree.contains(15))
```

Output:

```
True
False
True
```

Example #2:

```
tree = BST()
print(tree.contains(0))
```

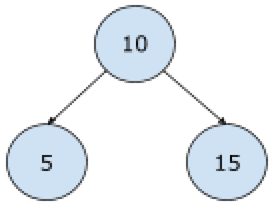
Output:

```
False
```

get_first(self) -> object:

This method must return the object stored at the root node. If the BinaryTree is empty, this method must return None.

Example #1:



```
tree = BST()
print(tree.get_first())
tree.add(10)
tree.add(15)
tree.add(5)
print(tree.get_first())
print(tree)
```

Output:

```
None
10
TREE in order { 5, 10, 15 }
```

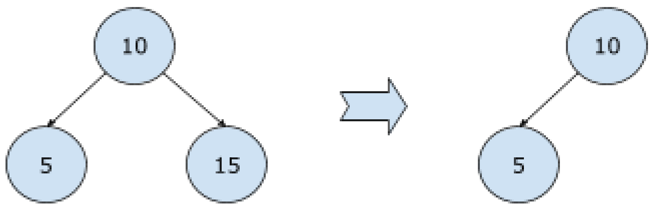
remove(self, value: object) -> bool:

This method should remove the first instance of the object in the BinaryTree. The method must return True if the value is removed from the BinaryTree and otherwise return False.

NOTE: See ‘Specific Instructions’ for explanation of which node replaces the deleted node.

Example #1:

```
tree = BST([10, 5, 15])
print(tree.remove(7))
print(tree.remove(15))
print(tree.remove(15))
```



Output:

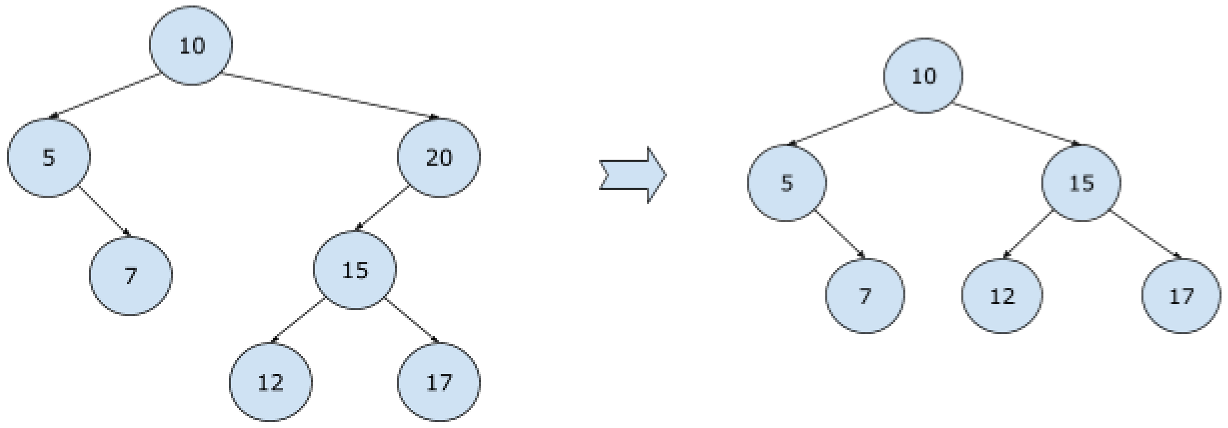
```
False
True
False
```

Example #2:

```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.remove(20))
print(tree)
```

Output:

```
True
TREE in order { 5, 7, 10, 12, 15, 17 }
```



remove_first(self) -> bool:

This method must remove the root node in the BinaryTree. The method must return False if the tree is empty and there is no root node to remove and True if the root is removed.

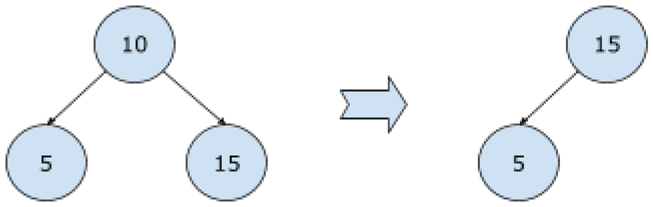
NOTE: See ‘Specific Instructions’ for explanation of which node replaces the deleted node.

Example #1:

```
tree = BST([10, 15, 5])
print(tree.remove_first())
print(tree)
```

Output:

```
True
TREE in order { 5, 15 }
```

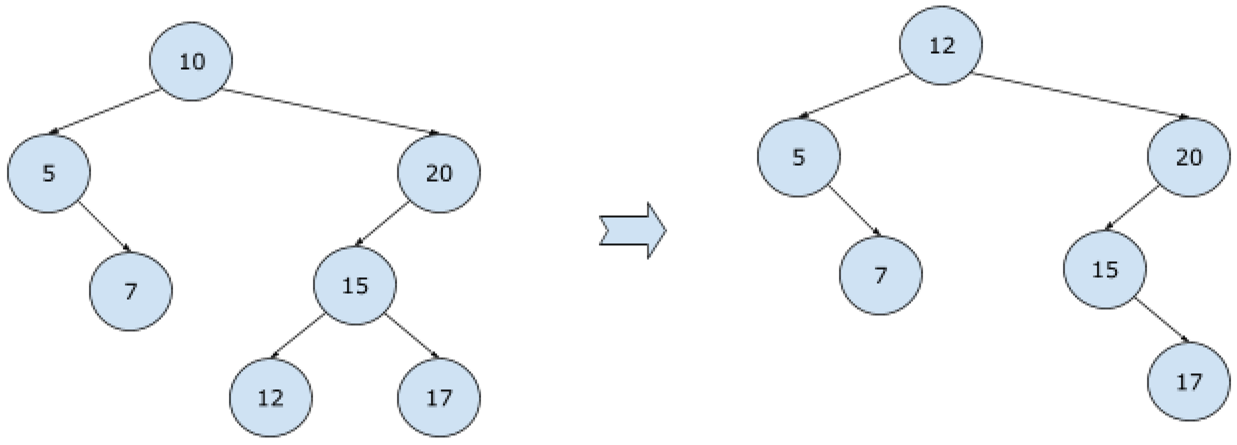


Example #2:

```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.remove_first())
print(tree)
```

Output:

```
True
TREE in order { 5, 7, 12, 15, 17, 20 }
```

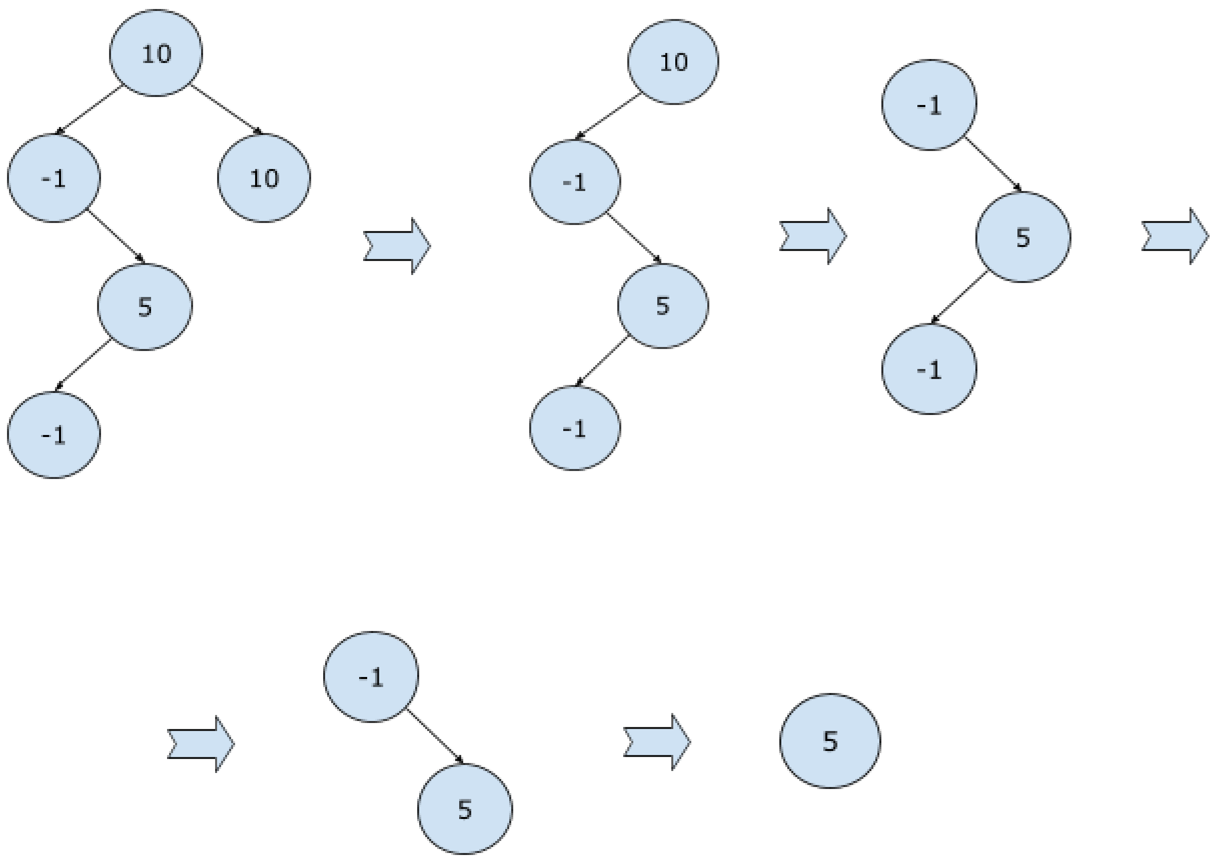


Example #3:

```
tree = BST([10, 10, -1, 5, -1])
print(tree.remove_first(), tree)
print(tree.remove_first(), tree)
print(tree.remove_first(), tree)
print(tree.remove_first(), tree)
print(tree.remove_first(), tree)
print(tree.remove_first(), tree)
```

Output:

```
True TREE in order { -1, -1, 5, 10 }
True TREE in order { -1, -1, 5 }
True TREE in order { -1, 5 }
True TREE in order { 5 }
True TREE in order {  }
False TREE in order {  }
```



pre_order_traversal(self) -> []:

This method will perform pre-order traversal of the tree and return a list of visited nodes.

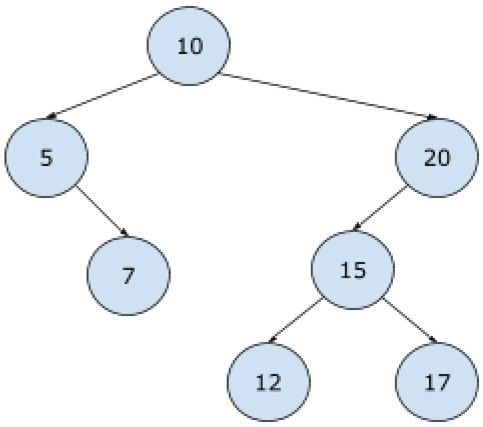
HINT: Skeleton code includes pre-written code for in_order_traversal() method. Please review that code for ideas on how to implement this method. The pre-written implementation of in_order_traversal() is recursive with no use of helper methods. You are welcome to implement the pre_order_traversal() as you see fit (similar to in_order_traversal() or using an iterative approach or recursively with helper methods…etc).

Example #1:

```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.pre_order_traversal())
```

Output:

```
[10, 5, 7, 20, 15, 12, 17]
```

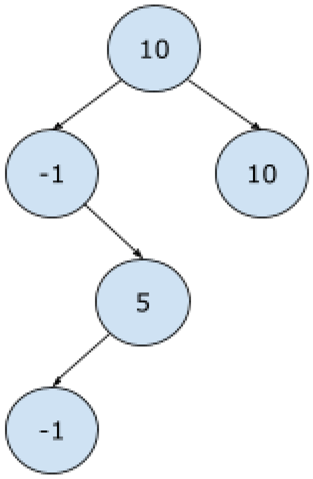


Example #2:

```
tree = BST([10, 10, -1, 5, -1])
print(tree.pre_order_traversal())
```

Output:

```
[10, -1, 5, -1, 10]
```



post_order_traversal(self) -> []:

This method will perform post-order traversal of the tree and return a list of visited nodes.

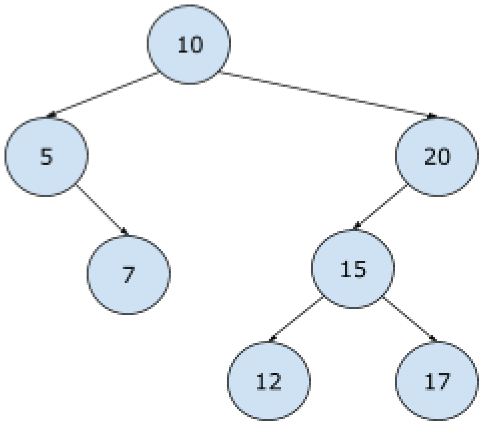
HINT: Skeleton code includes pre-written code for in_order_traversal() method. Please review that code for ideas on how to implement this method. The pre-written implementation of in_order_traversal() is recursive with no use of helper methods. You're welcome to implement post_order_traversal() as you see fit (similar to in_order_traversal() or using iterative approach or recursively with helper methods…etc).

Example #1:

```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.post_order_traversal())
```

Output:

```
[7, 5, 12, 17, 15, 20, 10]
```

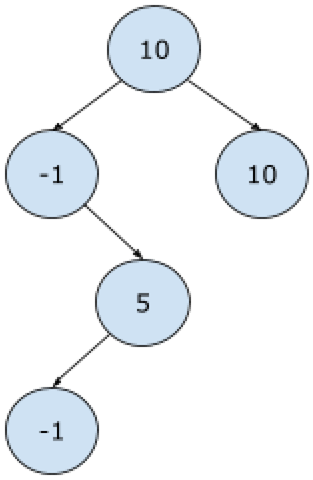


Example #2:

```
tree = BST([10, 10, -1, 5, -1])
print(tree.post_order_traversal())
```

Output:

```
[-1, 5, -1, 10, 10]
```



left_child(self, subtree_root: TreeNode) -> TreeNode

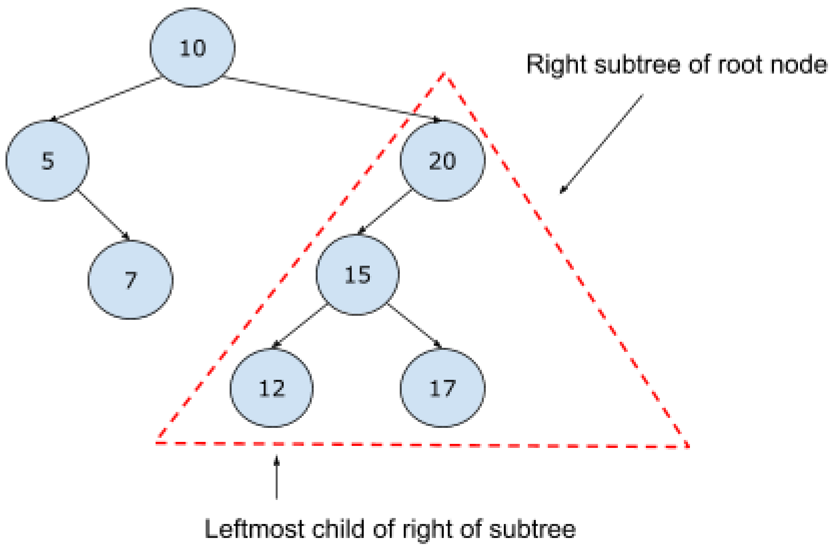
This helper method should return the leftmost node in the given subtree.

Example #1:

```
tree = BST([10, 20, 5, 15, 17, 7, 12])
root_node = tree.root
left_child = tree.left_child(root_node.right)
print(left_child.val)
```

Output:

```
12
```


10
5
20
7
15
12
17
Right subtree of root node
Leftmost child of right of subtree