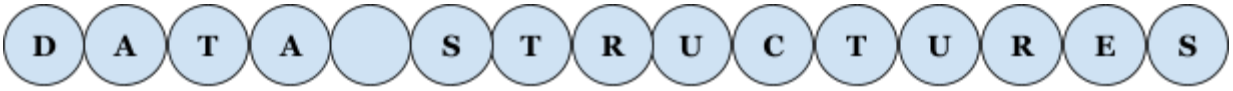
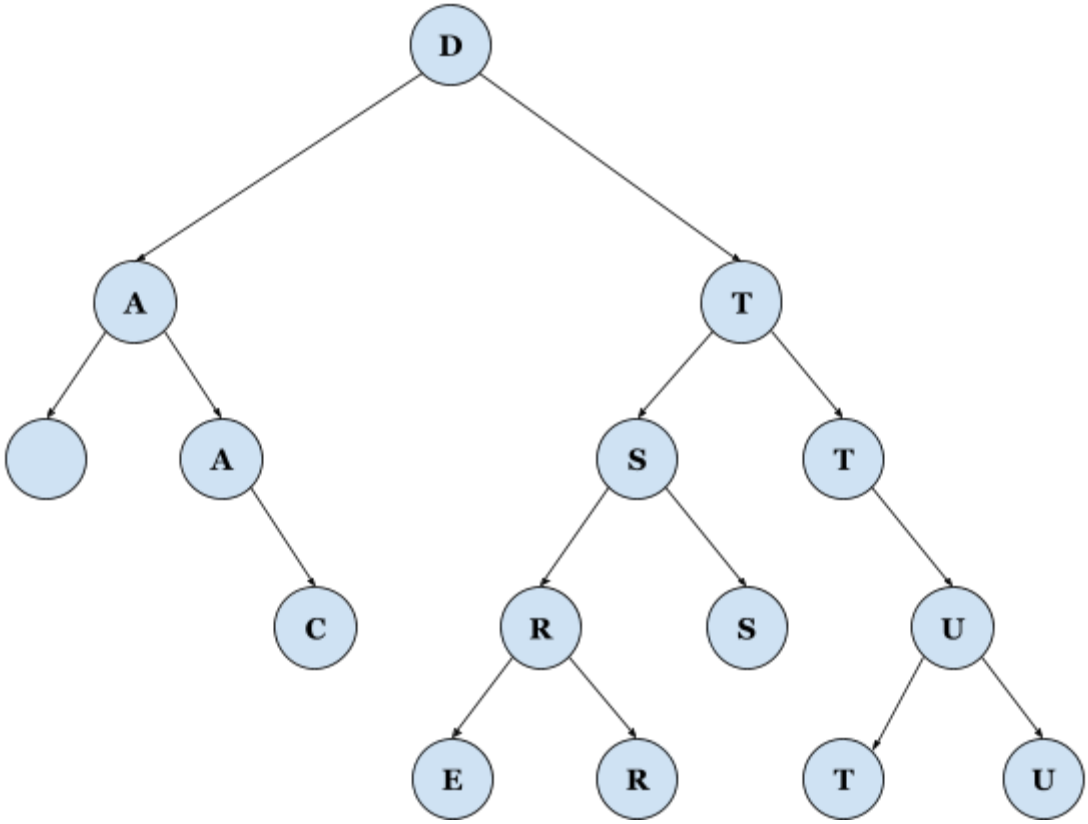
CS261 Data Structures

Assignment 4

v 1.10 (revised 7/16/2021)

Your Very Own BST Tree and Binary Search Practice





Contents

General Instructions

1. Programs in this assignment must be written in Python v3 and submitted to Gradescope before the due date specified in the syllabus. You may resubmit your code as many times as necessary. Gradescope allows you to choose which submission will be graded.

2. 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.

3. We encourage you to create your own test programs and cases even though this work won't have to be submitted and won't be graded. 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. At a minimum, each method should have a descriptive docstring. Additionally, put comments throughout the code to make it easy to follow and understand.

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 helper methods and variables, as needed. You also are allowed to add optional parameters to method definitions (may be especially helpful when writing recursive solutions).

   However, certains classes and methods cannot be changed in any way. Please see comments in the skeleton code for guidance. In particular, content of any methods pre-written for you as part of the skeleton code must not be changed.

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 an 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 BST class by completing provided skeleton code in the file `bst.py`. Once completed, your implementation will include the following methods:

   ```
   add(), contains(), remove()
   get_first(), remove_first()
   pre_order_traversal()
   in_order_traversal()
   post_order_traversal()
   by_level_traversal()
   size(), height()
   count_leaves()
   count_unique()
   is_complete(), is_full(), is_perfect()
   ```

2. We will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have correct implementation of methods __eq__, __lt__, __gt__, __ge__, __le__ and __str__.

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

4. 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.

5. When removing a node, replace it with the leftmost child of the right subtree (aka in-order successor). You do not need to 'recursively' continue this process.

   If the deleted node only has one subtree (either right or left), replace the deleted node with the root node of that subtree.

6. Variables in TreeNode and BST classes 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 them.

7. RESTRICTIONS: You are not allowed to use ANY built-in Python data structures and/or their methods. You are not to use any imported modules for your implementation but you may use the `random` module for testing.

   In case you need 'helper' data structures in your solution, skeleton code includes prewritten implementation of Queue and Stack classes. You are allowed to create and use objects from those classes in your implementation.

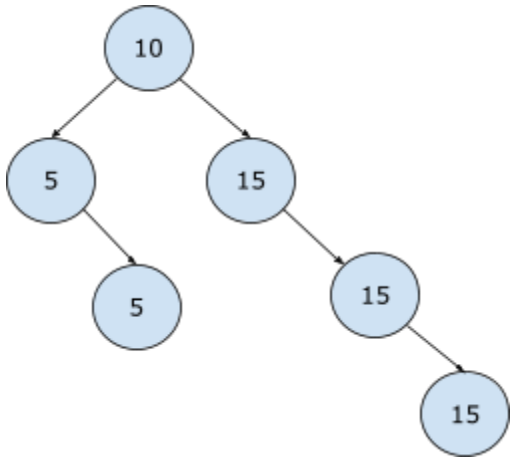
   You are not allowed to directly access any variables of the Queue or Stack classes. All work must be done only by using class methods.

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

This method adds new value to the tree, maintaining its 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)
```

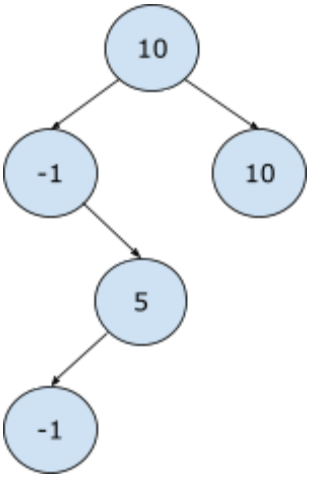


Output:

```
TREE pre-order {  }
TREE pre-order { 10, 5, 15 }
TREE pre-order { 10, 5, 15, 15, 15 }
TREE pre-order { 10, 5, 5, 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:

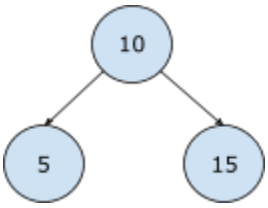
```
TREE pre-order { 10, 10 }
TREE pre-order { 10, -1, 10 }
TREE pre-order { 10, -1, 5, 10 }
TREE pre-order { 10, -1, 5, -1, 10 }
```

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

This method returns True if the value parameter is in the BinaryTree or False if it is not in the tree. If the tree is empty, the method should return False.

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:

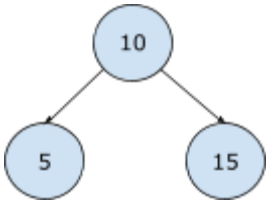
```
False
```

get_first(self) -> object:

This method returns the value stored at the root node. If the BinaryTree is empty, this method returns 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 pre-order { 10, 5, 15 }
```

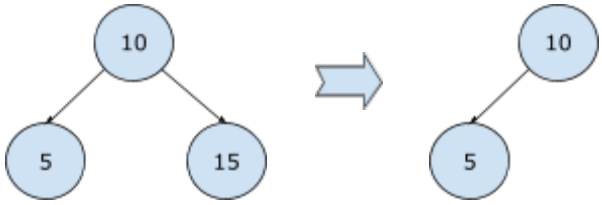
remove(self, value: object) -> bool:

This method should remove the first instance of the value 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:

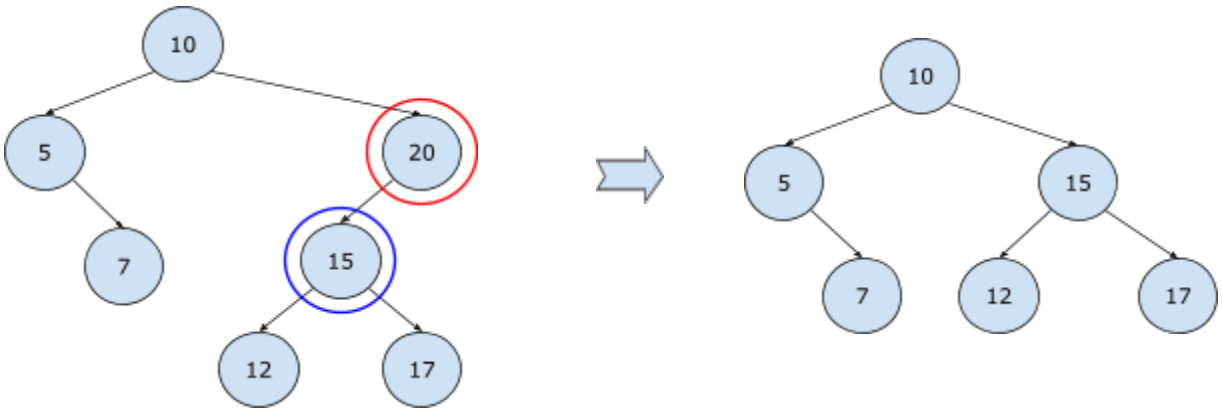
```
False
True
False
```

Example #2:

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

Output:

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

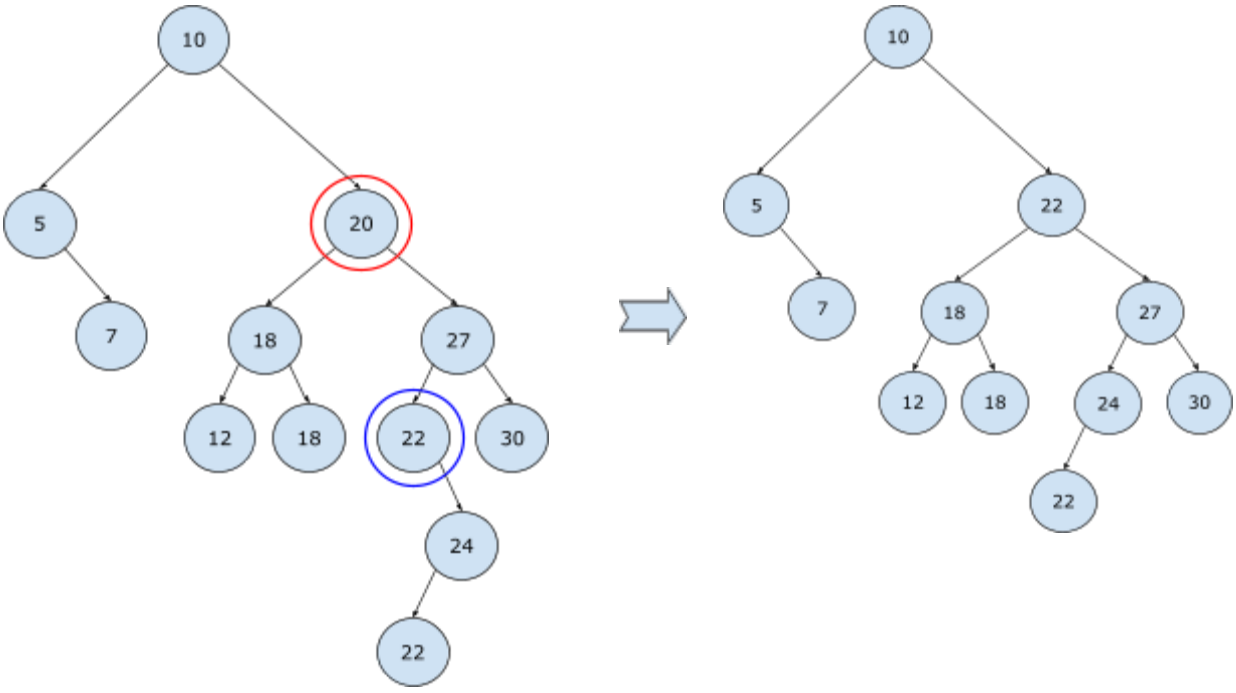


Example #3:

```
tree = BST([10, 5, 20, 18, 12, 7, 27, 22, 18, 24, 22, 30])
print(tree.remove(20))
print(tree)
# comment out the following lines
# if you have not yet implemented traversal methods
print(tree.pre_order_traversal())
print(tree.in_order_traversal())
print(tree.post_order_traversal())
print(tree.by_level_traversal())
```

Output:

```
True
TREE pre-order { 10, 5, 7, 22, 18, 12, 18, 27, 24, 22, 30 }
QUEUE { 10, 5, 7, 22, 18, 12, 18, 27, 24, 22, 30 }
QUEUE { 5, 7, 10, 12, 18, 18, 22, 22, 24, 27, 30 }
QUEUE { 7, 5, 12, 18, 18, 22, 24, 30, 27, 22, 10 }
QUEUE { 10, 5, 22, 7, 18, 27, 12, 18, 24, 30, 22 }
```



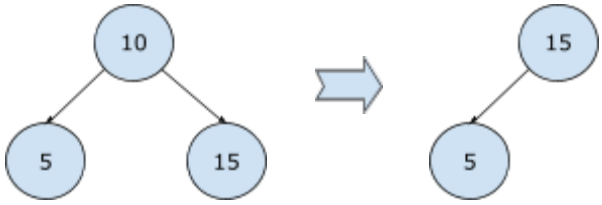
remove_first(self) -> bool:

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:

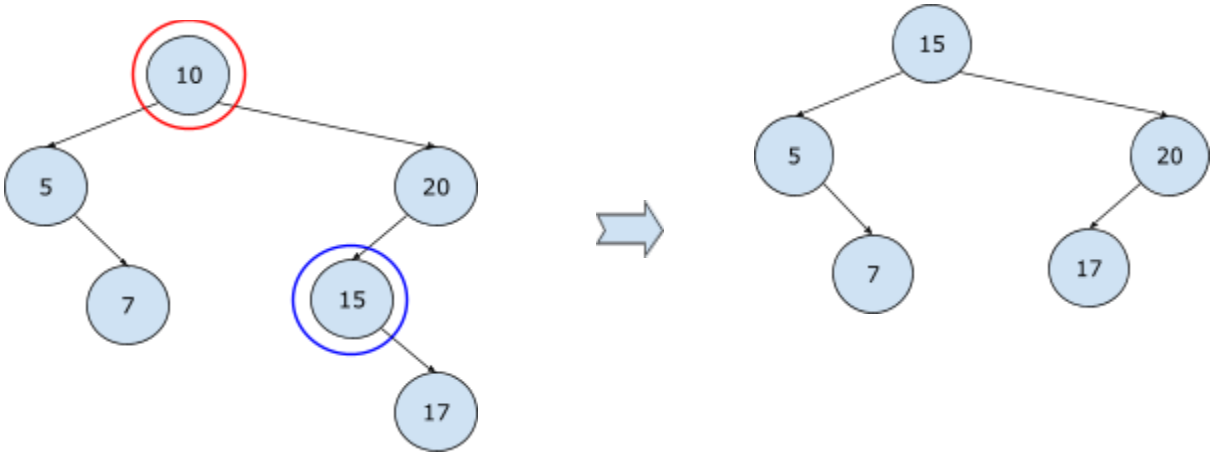
```
True
TREE pre-order { 15, 5 }
```

Example #2:

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

Output:

```
True
TREE pre-order { 15, 5, 7, 20, 17 }
```

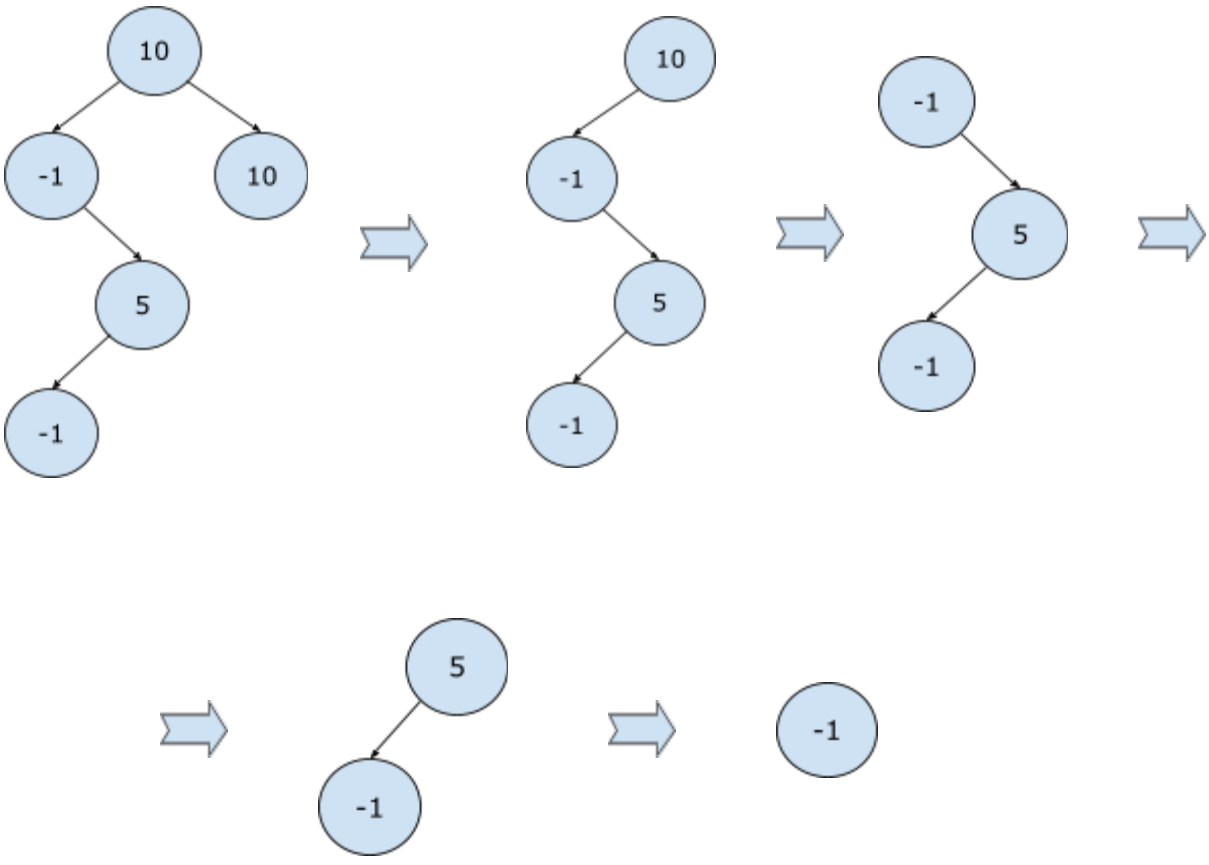


Example #3:

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

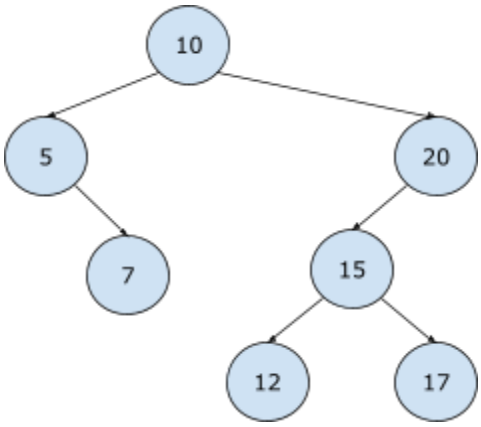


pre_order_traversal(self) -> Queue:
in_order_traversal(self) -> Queue:
post_order_traversal(self) -> Queue:
by_level_traversal(self) -> Queue:

These methods will perform pre-order, in-order, post-order, or by-level traversal of the tree, respectively, and return a Queue object that contains values of visited nodes, in the order they were visited. If the tree has no nodes, these methods should return an empty Queue.

Example #1:

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

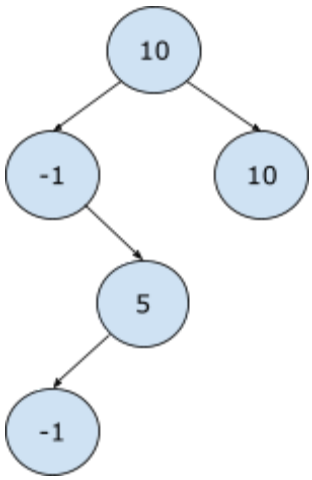


Output:

```
QUEUE { 10, 5, 7, 20, 15, 12, 17 }
QUEUE { 5, 7, 10, 12, 15, 17, 20 }
QUEUE { 7, 5, 12, 17, 15, 20, 10 }
QUEUE { 10, 5, 20, 7, 15, 12, 17 }
```

Example #2:

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



Output:

```
QUEUE { 10, -1, 5, -1, 10 }
QUEUE { -1, -1, 5, 10, 10 }
QUEUE { -1, 5, -1, 10, 10 }
QUEUE { 10, -1, 10, 5, -1 }
```

size(self) -> int:

This method returns the total number of nodes in the tree. See comprehensive examples 1 and 2 below for more details.

height(self) -> int:

This method returns the height of the binary tree. An empty tree has a height of -1. Tree consisting of just a single root node should return a height of 0. See comprehensive examples 1 and 2 below for more details.

count_leaves(self) -> int:

This method returns the number of nodes in the tree that have no children. If the tree is empty, this method should return 0. See comprehensive examples 1 and 2 below for more details.

count_unique(self) -> int:

This method returns the count of unique values stored in the tree. If all values stored in the tree are distinct (no duplicates), this method will return the same result as the size() method. See comprehensive examples 1 and 2 below for more details.

is_complete(self) -> bool:

This method returns True if the current tree is a 'complete binary tree'. Empty tree is considered complete. A tree consisting of a single root node is complete. See comprehensive examples 1 and 2 below for more details.

is_full(self) -> bool:

This method returns True if the current tree is a 'full binary tree'. Empty tree is considered 'full'. A tree consisting of a single root node is 'full'. See comprehensive examples 1 and 2 below for more details.

is_perfect(self) -> bool:

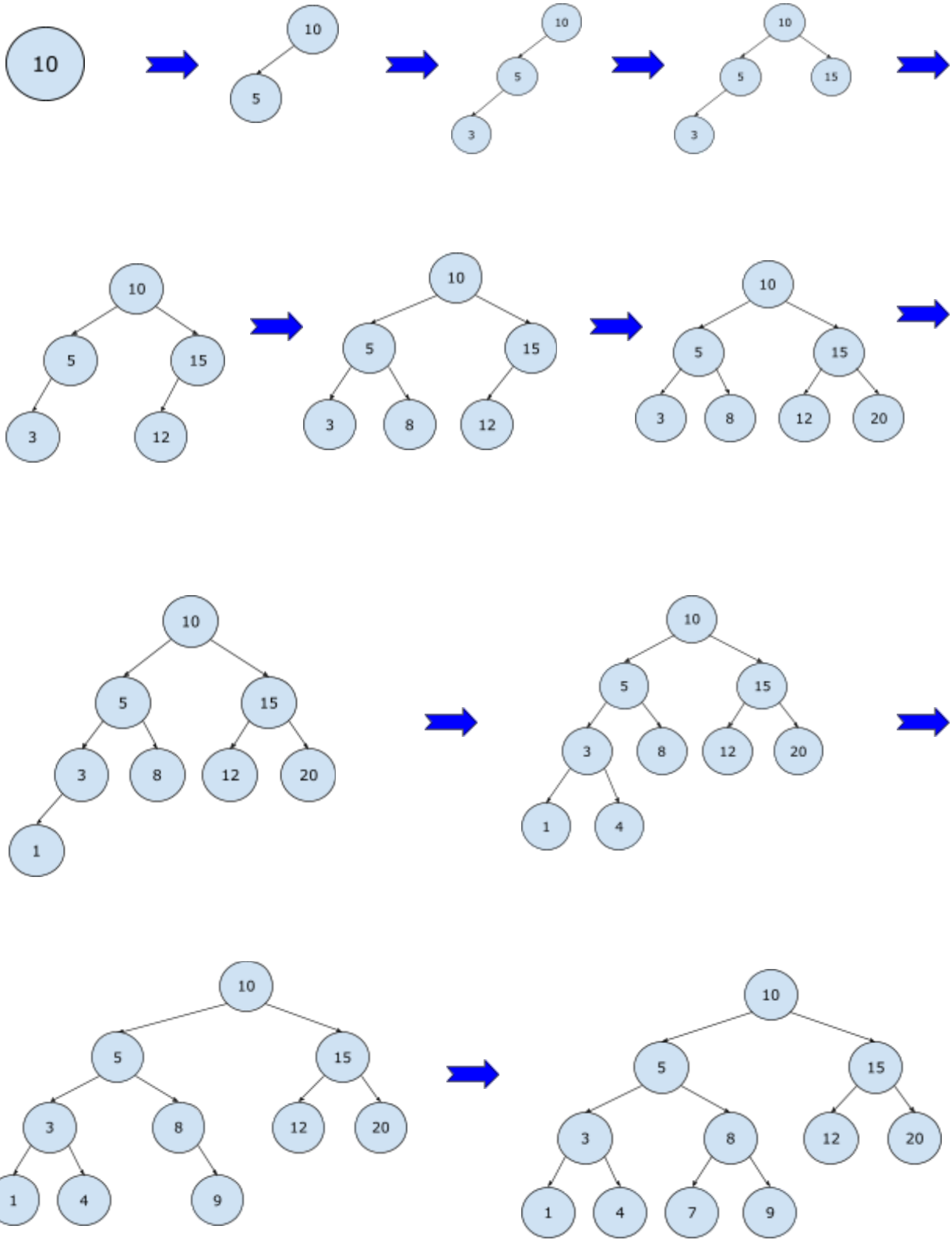
This method returns True if the current tree is a 'perfect binary tree'. Empty tree is considered 'perfect'. A tree consisting of a single root node is 'perfect'. See comprehensive examples 1 and 2 below for more details.

Comprehensive Example #1:

```
tree = BST()
header = 'Value   Size  Height   Leaves   Unique   '
header += 'Complete?  Full?    Perfect?'
print(header)
print('-' * len(header))
print(f'  N/A {tree.size():6} {tree.height():7} ',
      f'{tree.count_leaves():7} {tree.count_unique():8}  ',
      f'{str(tree.is_complete()):10}',
      f'{str(tree.is_full()):7} ',
      f'{str(tree.is_perfect())}')

for value in [10, 5, 3, 15, 12, 8, 20, 1, 4, 9, 7]:
    tree.add(value)
    print(f'{value:5} {tree.size():6} {tree.height():7} ',
          f'{tree.count_leaves():7} {tree.count_unique():8}  ',
          f'{str(tree.is_complete()):10}',
          f'{str(tree.is_full()):7} ',
          f'{str(tree.is_perfect())}')
print()
print(tree.pre_order_traversal())
print(tree.in_order_traversal())
print(tree.post_order_traversal())
print(tree.by_level_traversal())
```

Output:

```
Value   Size  Height   Leaves   Unique   Complete?  Full?    Perfect?
-------------------------------------------------------------------------
  N/A      0      -1        0        0   True       True     True
   10      1       0        1        1   True       True     True
    5      2       1        1        2   True       False    False
    3      3       2        1        3   False      False    False
   15      4       2        2        4   True       False    False
   12      5       2        2        5   False      False    False
    8      6       2        3        6   True       False    False
   20      7       2        4        7   True       True     True
    1      8       3        4        8   True       False    False
    4      9       3        5        9   True       True     False
    9     10       3        5       10   False      False    False
    7     11       3        6       11   True       True     False

QUEUE { 10, 5, 3, 1, 4, 8, 7, 9, 15, 12, 20 }
QUEUE { 1, 3, 4, 5, 7, 8, 9, 10, 12, 15, 20 }
QUEUE { 1, 4, 3, 7, 9, 8, 5, 12, 20, 15, 10 }
QUEUE { 10, 5, 15, 3, 8, 12, 20, 1, 4, 7, 9 }
```

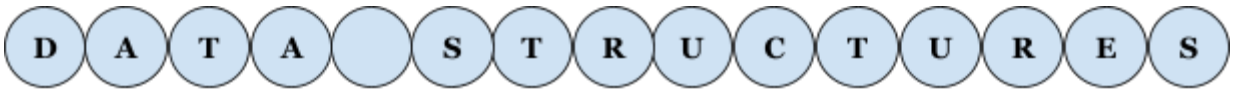

10
10
5
10
5
3
10
5
15
3
10
5
15
3
12
10
5
15
3
8
12
10
5
15
3
8
12
20
10
5
15
3
8
12
20
1
10
5
15
3
8
12
20
1
4
10
5
15
3
8
12
20
1
4
9
10
5
15
3
8
12
20
1
4
7
9

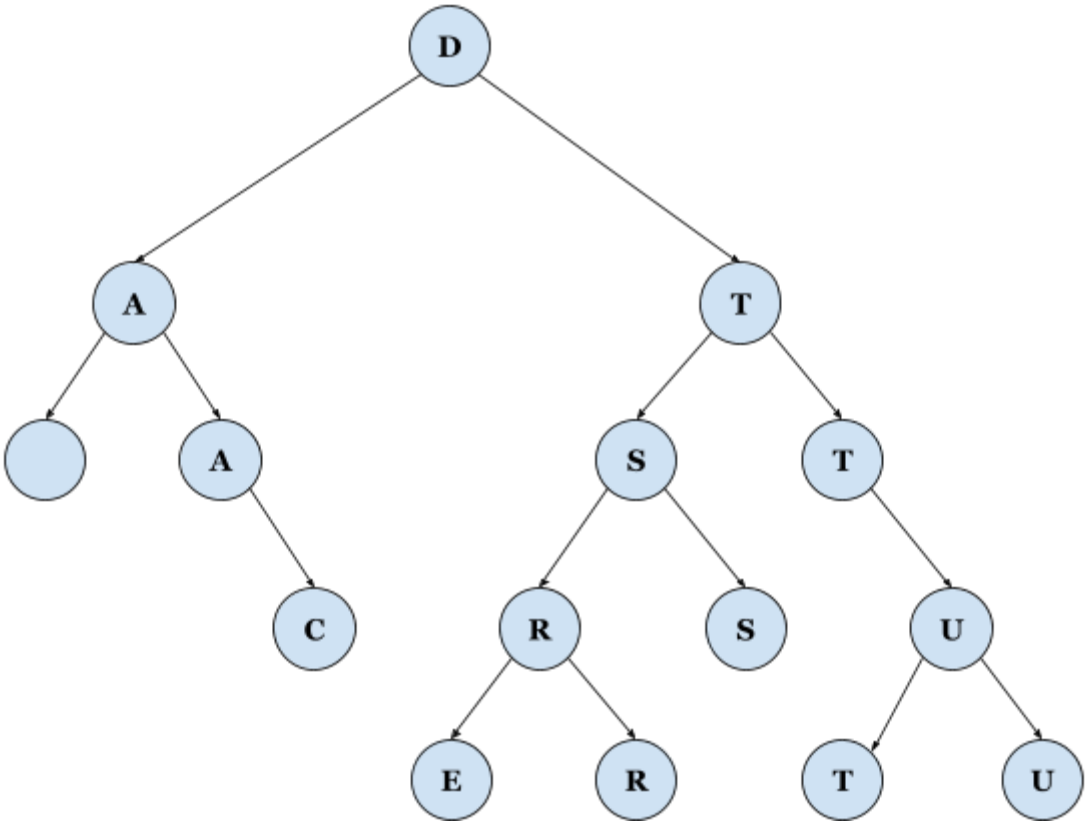
Comprehensive Example #2:

```
tree = BST()
header = 'Value   Size  Height   Leaves   Unique   '
header += 'Complete?  Full?    Perfect?'
print(header)
print('-' * len(header))
print(f'N/A   {tree.size():6} {tree.height():7} ',
      f'{tree.count_leaves():7} {tree.count_unique():8}  ',
      f'{str(tree.is_complete()):10}',
      f'{str(tree.is_full()):7} ',
      f'{str(tree.is_perfect())}')

for value in 'DATA STRUCTURES':
    tree.add(value)
    print(f'{value:5} {tree.size():6} {tree.height():7} ',
          f'{tree.count_leaves():7} {tree.count_unique():8}  ',
          f'{str(tree.is_complete()):10}',
          f'{str(tree.is_full()):7} ',
          f'{str(tree.is_perfect())}')
print('', tree.pre_order_traversal(), tree.in_order_traversal(),
      tree.post_order_traversal(), tree.by_level_traversal(),
      sep='\n')
```

Output:

```
Value   Size  Height   Leaves   Unique   Complete?  Full?    Perfect?
-------------------------------------------------------------------
N/A        0      -1        0        0   True       True     True
D          1       0        1        1   True       True     True
A          2       1        1        2   True       False    False
T          3       1        2        3   True       True     True
A          4       2        2        3   False      False    False
           5       2        3        4   True       True     False
S          6       2        3        5   True       False    False
T          7       2        4        5   True       True     True
R          8       3        4        6   False      False    False
U          9       3        4        7   False      False    False
C         10       3        4        8   False      False    False
T         11       4        4        8   False      False    False
U         12       4        5        8   False      False    False
R         13       4        5        8   False      False    False
E         14       4        6        9   False      False    False
S         15       4        7        9   False      False    False

QUEUE { D, A,  , A, C, T, S, R, E, R, S, T, U, T, U }
QUEUE {  , A, A, C, D, E, R, R, S, S, T, T, T, U, U }
QUEUE {  , C, A, A, E, R, R, S, S, T, U, U, T, T, D }
QUEUE { D, A, T,  , A, S, T, C, R, S, U, E, R, T, U }
```


D A T A S T R U C T U R E S


D
A
T
A
S
T
C
R
S
U
E
R
T
U