CS261 Data Structures

Assignment 5

Fall 2021

AVL Tree Implementation

Contents

General Instructions

1. The program 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 any non-obvious code.

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 default parameters to method definitions.

   However, certain 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. **Methods may be implemented iteratively or recursively at your discretion.** 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.

8. 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__.

Summary and Specific Instructions

1. Implement the AVL class by completing the provided skeleton code in the file `avl.py`. Once completed, your implementation will include the following methods:

   ```
   add(), remove()
   contains(), inorder_traversal()
   find_min(), find_max()
   is_empty(), make_empty()
   ```

2. When reviewing the provided skeleton code, please note that the TreeNode class has a couple of important attributes: `parent` (designed to store a pointer to the parent of the current node); and `height` (designed to store the height of the subtree rooted at the current node). Your implementation must correctly maintain all three node pointers (`left`, `right` and `parent`), as well as the `height` attribute of each node.

3. 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__.

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

5. **When removing a node with two subtrees, replace it with the leftmost child of the right subtree (a.k.a. the inorder 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. **The variables in TreeNode and AVL 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. 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.

8. There are no specified runtime requirements for this assignment, however we encourage you to try and implement efficient solutions as outlined in the pseudocode in the course modules on Canvas.

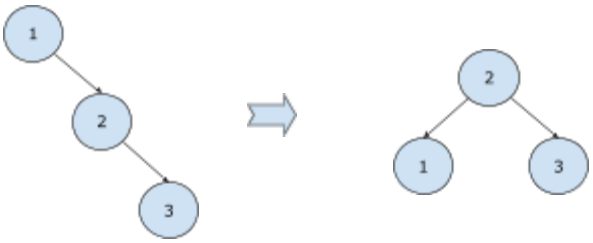
9. You may not use any imports beyond the ones included in the assignment source code provided.

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

This method adds a new value to the tree while maintaining its AVL property. **Duplicate values are not allowed.** If the value is already in the tree, the method should not change the tree.

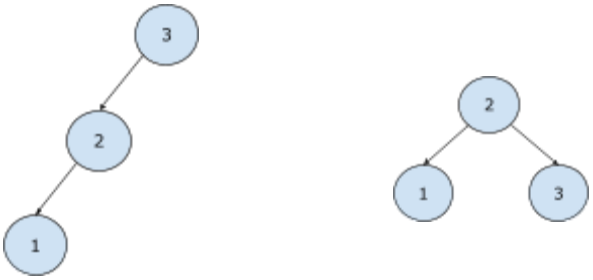
Example #1:

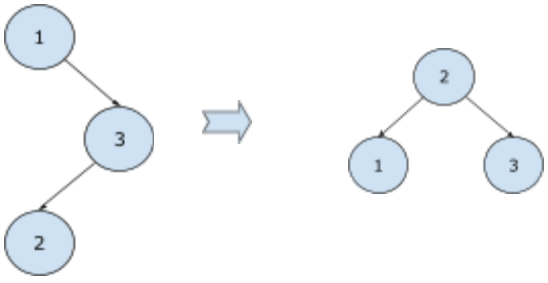
```
test_cases = (
    (1, 2, 3),          #RR
    (3, 2, 1),          #LL
    (1, 3, 2),          #RL
    (3, 1, 2),          #LR
)
for case in test_cases:
    avl = AVL(case)
    print(avl)
```

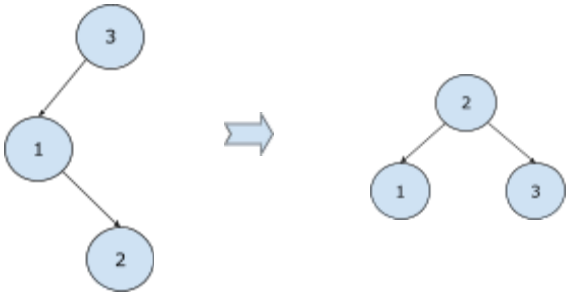


Output:

```
AVL pre-order { 2, 1, 3 }
AVL pre-order { 2, 1, 3 }
AVL pre-order { 2, 1, 3 }
AVL pre-order { 2, 1, 3 }
```







Example #2:

```
test_cases = (
    (10, 20, 30, 40, 50),   # RR, RR
    (10, 20, 30, 50, 40),   # RR, RL
    (30, 20, 10, 5, 1),     # LL, LL
    (30, 20, 10, 1, 5),     # LL, LR
    (5, 4, 6, 3, 7, 2, 8),  # LL, RR
    (range(0, 30, 3)),
    (range(0, 31, 3)),
    (range(0, 34, 3)),
    (range(10, -10, -2)),
    ('A', 'B', 'C', 'D', 'E'),
    (1, 1, 1, 1),
)
for case in test_cases:
    avl = AVL(case)
    print('INPUT  :', case)
    print('RESULT :', avl)
```

Output:

```
INPUT  : (10, 20, 30, 40, 50)
RESULT : AVL pre-order { 20, 10, 40, 30, 50 }
INPUT  : (10, 20, 30, 50, 40)
RESULT : AVL pre-order { 20, 10, 40, 30, 50 }
INPUT  : (30, 20, 10, 5, 1)
RESULT : AVL pre-order { 20, 5, 1, 10, 30 }
INPUT  : (30, 20, 10, 1, 5)
RESULT : AVL pre-order { 20, 5, 1, 10, 30 }
INPUT  : (5, 4, 6, 3, 7, 2, 8)
RESULT : AVL pre-order { 5, 3, 2, 4, 7, 6, 8 }
INPUT  : range(0, 30, 3)
RESULT : AVL pre-order { 9, 3, 0, 6, 21, 15, 12, 18, 24, 27 }
INPUT  : range(0, 31, 3)
RESULT : AVL pre-order { 9, 3, 0, 6, 21, 15, 12, 18, 27, 24, 30 }
INPUT  : range(0, 34, 3)
RESULT : AVL pre-order { 21, 9, 3, 0, 6, 15, 12, 18, 27, 24, 30, 33 }
INPUT  : range(10, -10, -2)
RESULT : AVL pre-order { 4, -4, -6, -8, 0, -2, 2, 8, 6, 10 }
INPUT  : ('A', 'B', 'C', 'D', 'E')
RESULT : AVL pre-order { B, A, D, C, E }
INPUT  : (1, 1, 1, 1)
RESULT : AVL pre-order { 1 }
```

Example #3:

```
for _ in range(100):
    case = list(set(random.randrange(1, 20000) for _ in range(900)))
    avl = AVL()
    for value in case:
        avl.add(value)
    if not avl.is_valid_avl():
        raise Exception("PROBLEM WITH ADD OPERATION")
print('add() stress test finished')
```

Output:

```
add() stress test finished
```

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

This method should remove the value from the AVL tree. The method must return True if the value is removed from the AVL Tree; otherwise return False.

NOTE: See 'Specific Instructions' for an explanation of which node replaces the deleted node.

Example #1:

```
test_cases = (
    ((1, 2, 3), 1),                             # no AVL rotation
    ((1, 2, 3), 2),                             # no AVL rotation
    ((1, 2, 3), 3),                             # no AVL rotation
    ((50, 40, 60, 30, 70, 20, 80, 45), 0),
    ((50, 40, 60, 30, 70, 20, 80, 45), 45),     # no AVL rotation
    ((50, 40, 60, 30, 70, 20, 80, 45), 40),     # no AVL rotation
    ((50, 40, 60, 30, 70, 20, 80, 45), 30),     # no AVL rotation
)
for tree, del_value in test_cases:
    avl = AVL(tree)
    print('INPUT  :', avl, "DELETE:", del_value)
    avl.remove(del_value)
    print('RESULT :', avl)
```

Output:

```
INPUT  : AVL pre-order { 2, 1, 3 } DEL: 1
RESULT : AVL pre-order { 2, 3 }
INPUT  : AVL pre-order { 2, 1, 3 } DEL: 2
RESULT : AVL pre-order { 3, 1 }
INPUT  : AVL pre-order { 2, 1, 3 } DEL: 3
RESULT : AVL pre-order { 2, 1 }
INPUT  : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 0
RESULT : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 45
RESULT : AVL pre-order { 50, 30, 20, 40, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 40
RESULT : AVL pre-order { 50, 30, 20, 45, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 30
RESULT : AVL pre-order { 50, 40, 20, 45, 70, 60, 80 }
```

Example #2:

```
test_cases = (
    ((50, 40, 60, 30, 70, 20, 80, 45), 20),     # RR
    ((50, 40, 60, 30, 70, 20, 80, 15), 40),     # LL
    ((50, 40, 60, 30, 70, 20, 80, 35), 20),     # RL
    ((50, 40, 60, 30, 70, 20, 80, 25), 40),     # LR
)
for tree, del_value in test_cases:
    avl = AVL(tree)
    print('INPUT  :', avl, "DELETE:", del_value)
    avl.remove(del_value)
    print('RESULT :', avl)
```

Output:

```
INPUT  : AVL pre-order { 50, 30, 20, 40, 45, 70, 60, 80 } DEL: 20
RESULT : AVL pre-order { 50, 40, 30, 45, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 15, 40, 70, 60, 80 } DEL: 40
RESULT : AVL pre-order { 50, 20, 15, 30, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 40, 35, 70, 60, 80 } DEL: 20
RESULT : AVL pre-order { 50, 35, 30, 40, 70, 60, 80 }
INPUT  : AVL pre-order { 50, 30, 20, 25, 40, 70, 60, 80 } DEL: 40
RESULT : AVL pre-order { 50, 25, 20, 30, 70, 60, 80 }
```

Example #3:

```
case = range(-9, 16, 2)
avl = AVL(case)
for del_value in case:
     print('INPUT  :', avl, del_value)
     avl.remove(del_value)
     print('RESULT :', avl)
```

Output:

```
INPUT  : AVL pre-order { 5, -3, -7, -9, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -9
RESULT : AVL pre-order { 5, -3, -7, -5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 5, -3, -7, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -7
RESULT : AVL pre-order { 5, -3, -5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 5, -3, -5, 1, -1, 3, 9, 7, 13, 11, 15 } -5
RESULT : AVL pre-order { 5, 1, -3, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 5, 1, -3, -1, 3, 9, 7, 13, 11, 15 } -3
RESULT : AVL pre-order { 5, 1, -1, 3, 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 5, 1, -1, 3, 9, 7, 13, 11, 15 } -1
RESULT : AVL pre-order { 5, 1, 3, 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 5, 1, 3, 9, 7, 13, 11, 15 } 1
RESULT : AVL pre-order { 9, 5, 3, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 9, 5, 3, 7, 13, 11, 15 } 3
RESULT : AVL pre-order { 9, 5, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 9, 5, 7, 13, 11, 15 } 5
RESULT : AVL pre-order { 9, 7, 13, 11, 15 }
INPUT  : AVL pre-order { 9, 7, 13, 11, 15 } 7
RESULT : AVL pre-order { 13, 9, 11, 15 }
INPUT  : AVL pre-order { 13, 9, 11, 15 } 9
RESULT : AVL pre-order { 13, 11, 15 }
INPUT  : AVL pre-order { 13, 11, 15 } 11
RESULT : AVL pre-order { 13, 15 }
INPUT  : AVL pre-order { 13, 15 } 13
RESULT : AVL pre-order { 15 }
INPUT  : AVL pre-order { 15 } 15
RESULT : AVL pre-order {  }
```

Example #4:

```
case = range(0, 34, 3)
avl = AVL(case)
for _ in case[:-2]:
    print('INPUT  :', avl.size(), avl, avl.root)
    avl.remove(avl.root.value)
    print('RESULT :', avl)
```

Output:

```
INPUT  : AVL pre-order { 21, 9, 3, 0, 6, 15, 12, 18, 27, 24, 30, 33 } 21
RESULT : AVL pre-order { 24, 9, 3, 0, 6, 15, 12, 18, 30, 27, 33 }
INPUT  : AVL pre-order { 24, 9, 3, 0, 6, 15, 12, 18, 30, 27, 33 } 24
RESULT : AVL pre-order { 27, 9, 3, 0, 6, 15, 12, 18, 30, 33 }
INPUT  : AVL pre-order { 27, 9, 3, 0, 6, 15, 12, 18, 30, 33 } 27
RESULT : AVL pre-order { 9, 3, 0, 6, 30, 15, 12, 18, 33 }
INPUT  : AVL pre-order { 9, 3, 0, 6, 30, 15, 12, 18, 33 } 9
RESULT : AVL pre-order { 12, 3, 0, 6, 30, 15, 18, 33 }
INPUT  : AVL pre-order { 12, 3, 0, 6, 30, 15, 18, 33 } 12
RESULT : AVL pre-order { 15, 3, 0, 6, 30, 18, 33 }
INPUT  : AVL pre-order { 15, 3, 0, 6, 30, 18, 33 } 15
RESULT : AVL pre-order { 18, 3, 0, 6, 30, 33 }
INPUT  : AVL pre-order { 18, 3, 0, 6, 30, 33 } 18
RESULT : AVL pre-order { 30, 3, 0, 6, 33 }
INPUT  : AVL pre-order { 30, 3, 0, 6, 33 } 30
RESULT : AVL pre-order { 3, 0, 33, 6 }
INPUT  : AVL pre-order { 3, 0, 33, 6 } 3
RESULT : AVL pre-order { 6, 0, 33 }
INPUT  : AVL pre-order { 6, 0, 33 } 6
RESULT : AVL pre-order { 33, 0 }
```

Example #5:

```
for _ in range(100):
    case = list(set(random.randrange(1, 20000) for _ in range(900)))
    avl = AVL(case)
    for value in case[::2]:
        avl.remove(value)
    if not avl.is_valid_avl():
        raise Exception("PROBLEM WITH REMOVE OPERATION")
print('Stress test finished')
```

Output:

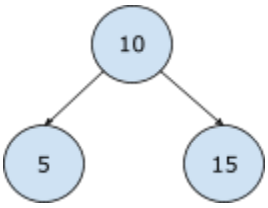
```
remove() stress test finished
```

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

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

Example #1:

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



Output:

```
True
False
True
```

Example #2:

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

Output:

```
False
```

inorder_traversal(self) -> Queue:

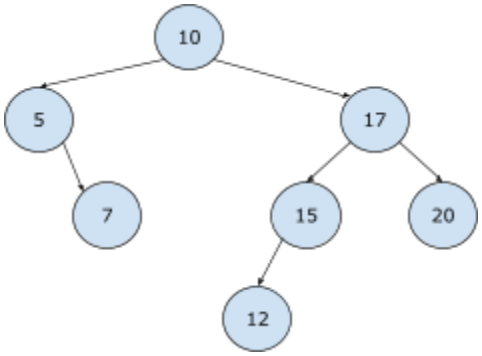
This method will perform an inorder traversal of the tree and return a Queue object that contains the values of the visited nodes, in the order they were visited. If the tree is empty, the methods should return an empty Queue.

Example #1:

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

Output:

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

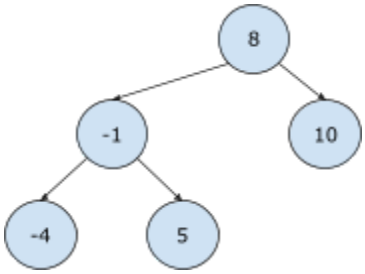


Example #2:

```
tree = AVL([8, 10, -4, 5, -1])
print(tree.inorder_traversal())
```

Output:

```
QUEUE { -4, -1, 5, 8, 10 }
```

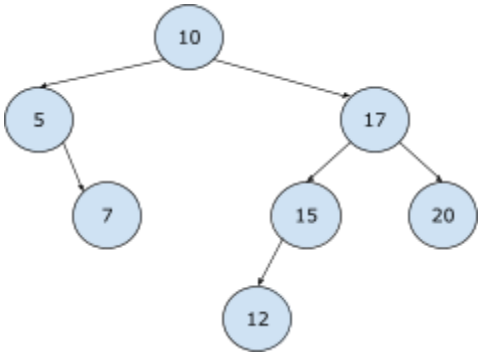


find_min(self) -> object:

This method returns the lowest value in the tree. If the tree is empty the method should return None.

Example #1:

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



Output:

```
5
```

Example #2:

```
tree = AVL([8, 10, -4, 5, -1])
print(tree.find_min())
```

Output:

```
-4
```

find_max(self) -> object:

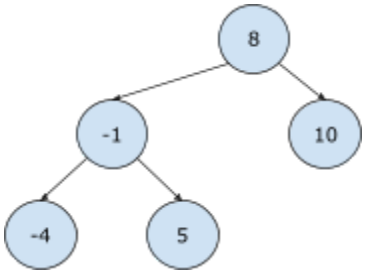
This method returns the highest value in the tree. If the tree is empty the method should return None.

Example #1:

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

Output:

```
20
```



Example #2:

```
tree = AVL([8, 10, -4, 5, -1])
print(tree.find_max())
```

Output:

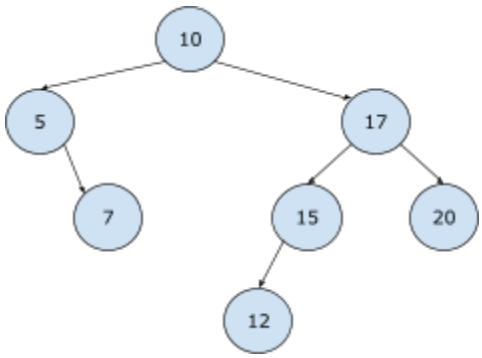
```
10
```

is_empty(self) -> bool:

This method returns True if the tree is empty, otherwise the method should return False.

Example #1:

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



Output:

```
False
```

Example #2:

```
tree = AVL()
print(tree.is_empty())
```

Output:

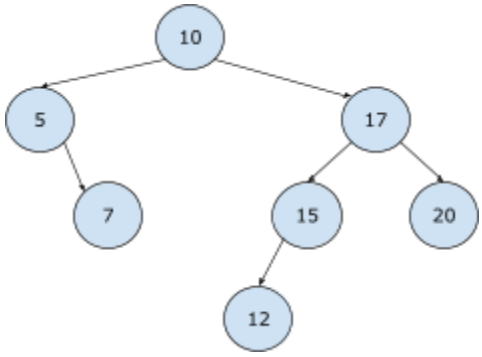
```
True
```

make_empty(self) -> None:

This method removes all of the nodes from the tree.

Example #1:

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



Output:

```
AVL pre-order { }
```

Example #2:

```
tree = AVL()
tree.make_empty())
print(tree)
```

Output:

```
AVL pre-order { }
```