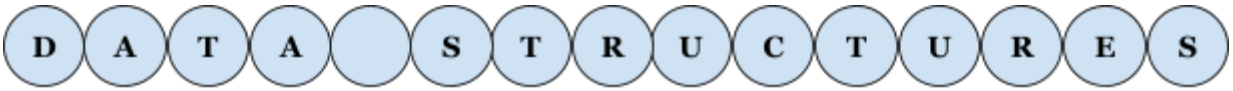
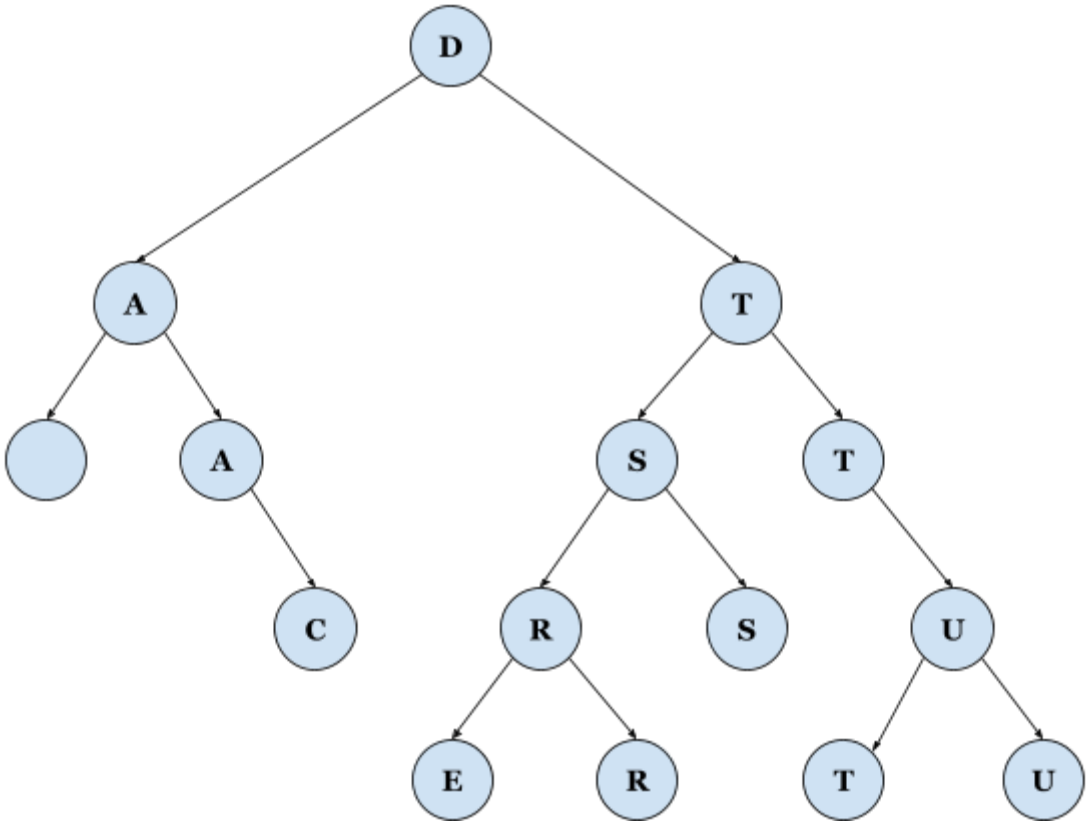
CS261 Data Structures

Assignment 4

Fall 2023

BST/AVL Tree Implementation





Contents

General Instructions

1. Programs in this assignment must be written in Python 3 and submitted to Gradescope before the due date specified on Canvas and in the Course Schedule. You may resubmit your code as many times as necessary before the due date (this is encouraged). Gradescope allows you to choose which submission will be graded. **We will grade the currently activated submission**.

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. You earn points as you pass tests. There are no hidden Gradescope tests and you will know your maximum possible score with each submission.

3. We encourage you to create your own test cases (such as with Python's **unittest** unit testing framework, which is more powerful and efficient than "print statement debugging"), even though this work doesn't have to be submitted and won't be graded. Gradescope tests are limited in scope and may not cover all edge cases. We reserve the right to evaluate your submission with additional methods beyond Gradescope.

4. Unless indicated otherwise, we will test your implementation with different types of objects, not just integers. We guarantee that all such objects will have the correct implementation of:

    a. "rich comparison" methods:

        i. `__eq__()`

        ii. `__lt__()`

        iii. `__gt__()`

        iv. `__ge__()`

        v. `__le__()`

    b. `__str__()`

5. **Your code must have an appropriate level of comments.** At minimum, each method must have a descriptive docstring. Additionally, write comments throughout your code to make it easy to follow and understand any non-obvious code. However, be mindful of the amount of comments in your code. Cluttering comments negatively impact readability and are discouraged. If your code includes an excessive amount of comments, consider refactoring. Please refer to "Styling Your Code" from the "Coding Guides and Tips - Style and Debugging" module for specifics.

6. You will be provided with a starter "skeleton" code, on which you will build your implementation. Methods defined in the skeleton code must retain their names and input/output parameters. Variables defined in the 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. **Both of the BST and AVL skeleton code files include some suggested helper methods.**

   You are allowed to:

   - add more helper methods and variables, as needed
   - add optional default parameters to method definitions
   - modify or add to the basic testing section within the scope of:

     ```
     if __name__ == "__main__":
     ```

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

   50% of points will be deducted from methods with the incorrect time complexity.

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

8. **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.

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

Part 1 - Summary and Specific Instructions

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

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

2. The BST class is constructed using instances of the provided BSTNode class.

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

4. **When removing a node with two subtrees, replace it with the leftmost child of the right subtree (i.e. 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.

5. **The variables in BSTNode 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. **The BST skeleton code includes some suggested helper methods.**

6. **RESTRICTIONS:** You are not allowed to use ANY built-in Python data structures and/or their methods. Your solutions should not call double underscore ("dunder") methods. In case you need 'helper' data structures in your solution, the skeleton code includes prewritten implementations of Queue and Stack classes, which are in separate files and imported in `bst.py` and `avl.py`. 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.

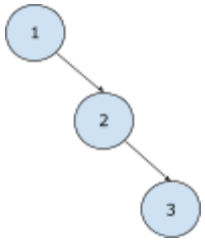
7. **Ensure that your methods meet the specified runtime requirements.**

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

This method adds a new value to the tree. **Duplicate values are allowed.** If a node with that value is already in the tree, the new value should be added to the right subtree of that node. It must be implemented with **O(N)** runtime complexity.

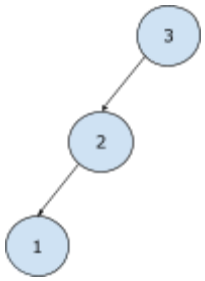
Example #1:

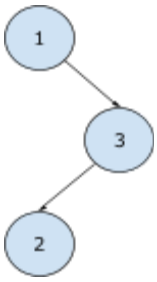
```
test_cases = (
    (1, 2, 3),
    (3, 2, 1),
    (1, 3, 2),
    (3, 1, 2),
)
for case in test_cases:
    tree = BST(case)
    print(tree)
```

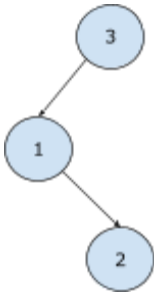


Output:

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







Example #2:

```
test_cases = (
    (10, 20, 30, 40, 50),
    (10, 20, 30, 50, 40),
    (30, 20, 10, 5, 1),
    (30, 20, 10, 1, 5),
    (5, 4, 6, 3, 7, 2, 8),
    (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:
    tree = BST(case)
    print('INPUT  :', case)
    print('RESULT :', tree)
```

Output:

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

Example #3:

```
for _ in range(100):
    case = list(set(random.randrange(1, 20000) for _ in range(900)))
    tree = BST()
    for value in case:
        bst.add(value)
    if not tree.is_valid_bst():
        Raise Exception("PROBLEM WITH ADD OPERATION")
print('add() stress test finished')
```

Output:

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

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

This method removes a value from the tree. The method returns True if the value is removed. Otherwise, it returns False. It must be implemented with **O(N)** runtime complexity.

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

Example #1:

```
test_cases = (
    ((1, 2, 3), 1),
    ((1, 2, 3), 2),
    ((1, 2, 3), 3),
    ((50, 40, 60, 30, 70, 20, 80, 45), 0),
    ((50, 40, 60, 30, 70, 20, 80, 45), 45),
    ((50, 40, 60, 30, 70, 20, 80, 45), 40),
    ((50, 40, 60, 30, 70, 20, 80, 45), 30),
)
for case, del_value in test_cases:
    tree = BST(case)
    print('INPUT  :', tree, "DELETE:", del_value)
    tree.remove(del_value)
    print('RESULT :', tree)
```

Output:

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

Example #2:

```
test_cases = (
    ((50, 40, 60, 30, 70, 20, 80, 45), 20),
    ((50, 40, 60, 30, 70, 20, 80, 15), 40),
    ((50, 40, 60, 30, 70, 20, 80, 35), 20),
    ((50, 40, 60, 30, 70, 20, 80, 25), 40),
)
for case, del_value in test_cases:
    tree= BST(tree)
    print('INPUT  :', tree, "DELETE:", del_value)
    tree.remove(del_value)
    print('RESULT :', tree)
```

Output:

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

Example #3:

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

Output:

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

Example #4:

```
case = range(0, 34, 3)
tree = BST(case)
for _ in case[:-2]:
    root_value = tree.get_root().value
    print('INPUT  :', tree, root_value)
    tree.remove(root_value)
    if not tree.is_valid_bst():
        raise Exception("PROBLEM WITH REMOVE OPERATION")
    print('RESULT :', tree)
```

Output:

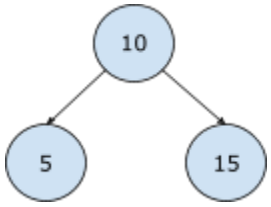
```
INPUT  : BST pre-order { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33 } 0
RESULT : BST pre-order { 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33 }
INPUT  : BST pre-order { 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33 } 3
RESULT : BST pre-order { 6, 9, 12, 15, 18, 21, 24, 27, 30, 33 }
INPUT  : BST pre-order {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

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

This method returns True if the value is in the tree. Otherwise, it returns False. If the tree is empty, the method should return False. It must be implemented with **O(N)** runtime complexity.

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
```

inorder_traversal(self) -> Queue:

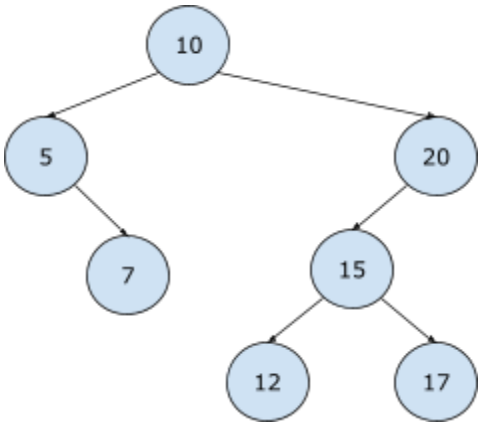
This method will perform an inorder traversal of the tree and return a Queue object that contains the values of the visited nodes, in the order they were visited. If the tree is empty, the method returns an empty Queue. It must be implemented with **O(N)** runtime complexity.

Example #1:

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

Output:

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

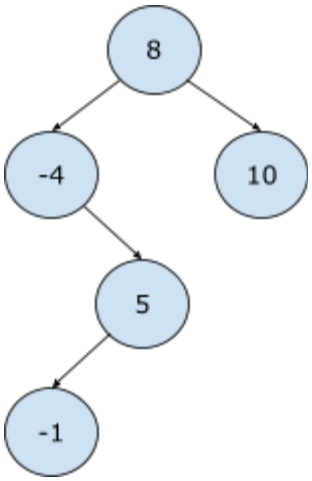


Example #2:

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

Output:

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

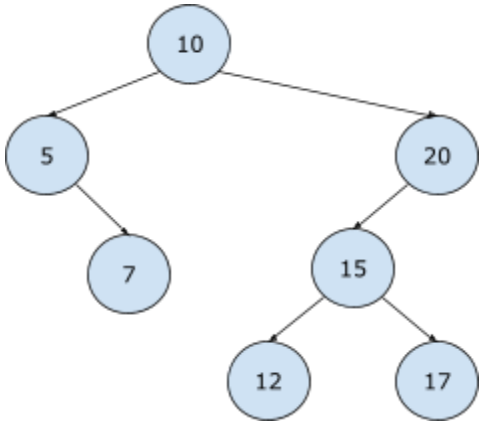


find_min(self) -> object:

This method returns the lowest value in the tree. If the tree is empty, the method should return None. It must be implemented with **O(N)** runtime complexity.

Example #1:
```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.find_min())
```

Output:
```
5
```



Example #2:
```
tree = BST([8, 10, -4, 5, -1])
print(tree.find_min())
```

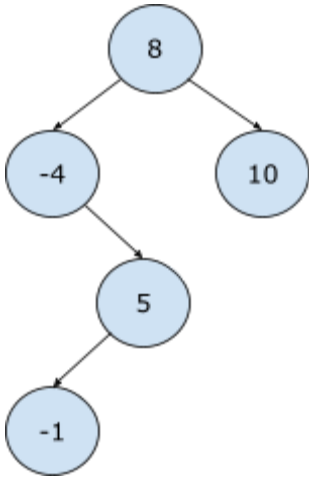
Output:
```
-4
```

find_max(self) -> object:

This method returns the highest value in the tree. If the tree is empty, the method should return None. It must be implemented with **O(N)** runtime complexity.

Example #1:
```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.find_max())
```

Output:
```
20
```



Example #2:
```
tree = BST([8, 10, -4, 5, -1])
print(tree.find_max())
```

Output:
```
10
```

is_empty(self) -> bool:

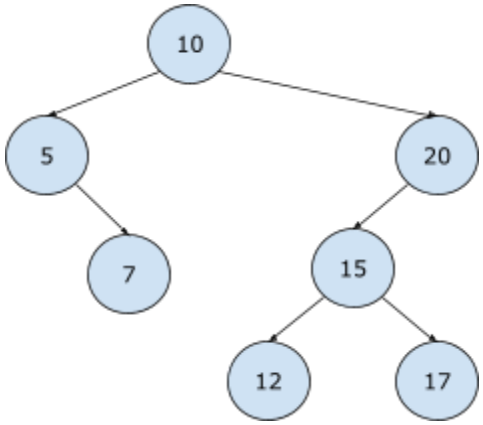
This method returns True if the tree is empty. Otherwise, it returns False. It must be implemented with **O(1)** runtime complexity.

Example #1:
```
tree = BST([10, 20, 5, 15, 17, 7, 12])
print(tree.is_empty())
```

Output:
```
False
```

Example #2:
```
tree = BST()
print(tree.is_empty())
```

Output:
```
True
```



make_empty(self) -> None:

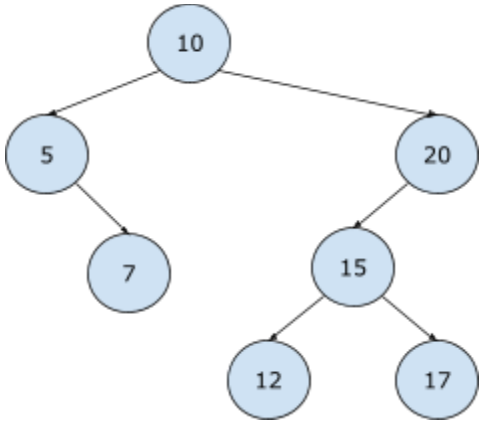
This method removes all of the nodes from the tree. It must be implemented with **O(1)** runtime complexity.

Example #1:
```
tree = BST([10, 20, 5, 15, 17, 7, 12])
tree.make_empty())
print(tree)
```

Output:
```
AVL pre-order { }
```

Example #2:
```
tree = BST()
tree.make_empty())
print(tree)
```

Output:
```
AVL pre-order { }
```



Part 2 - Summary and Specific Instructions

1. Implement the AVL class (a subclass of BST) by completing the provided skeleton code in the file `avl.py`. Once completed, your implementation will include overridden version of the following methods:

   ```
   add(), remove()
   ```

   And it will inherit the following methods from BST:

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

2. When reviewing the provided skeleton code, please note that the AVLNode class (a subclass of BSTNode) has two important added attributes: `parent` (to store a pointer to the parent of the current node) and `height` (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. Your tree must use the AVLNode class.

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

4. **When removing a node with two subtrees, replace it with the leftmost child of the right subtree (i.e. 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.

5. **The variables in AVLNode 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. **The AVL skeleton code includes some suggested helper methods.**

6. **RESTRICTIONS:** You are **NOT** allowed to use ANY built-in Python data structures and/or their methods. Your solutions should not call double underscore ("dunder") methods. In case you need 'helper' data structures in your solution, the skeleton code includes prewritten implementations of Queue and Stack classes, which are in separate files and imported in `bst.py` and `avl.py`. 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.

7. **Ensure that your methods meet the specified runtime requirements.**

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

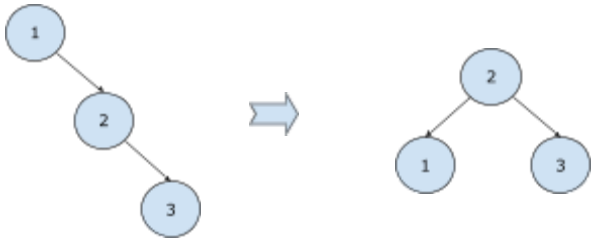
This method adds a new value to the tree while maintaining its AVL property. **Duplicate values are not allowed.** If the value is already in the tree, the method should not change the tree. It must be implemented with **O(log N)** runtime complexity.

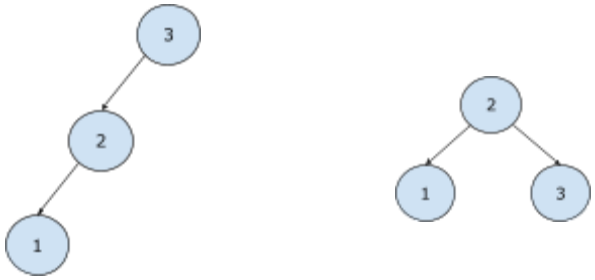
Example #1:

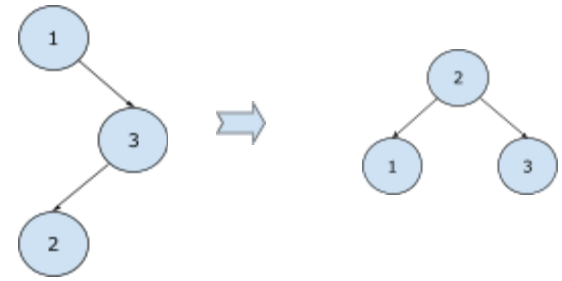
```
test_cases = (
    (1, 2, 3),          #RR
    (3, 2, 1),          #LL
    (1, 3, 2),          #RL
    (3, 1, 2),          #LR
)
for case in test_cases:
    tree = AVL(case)
    print(tree)
```

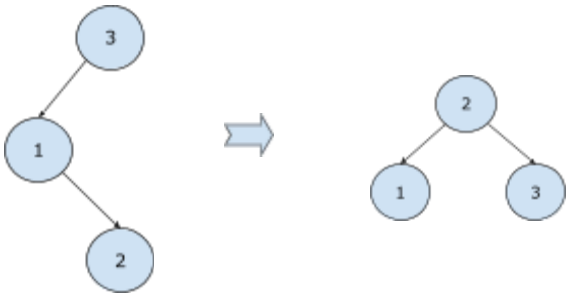
Output:

```
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:
    tree = AVL(case)
    print('INPUT  :', case)
    print('RESULT :', tree)
```

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)))
    tree = AVL()
    for value in case:
        tree.add(value)
    if not tree.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 removes the value from the AVL tree. The method returns True if the value is removed. Otherwise, it returns False. It must be implemented with **O(log N)** runtime complexity.

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 case, del_value in test_cases:
    tree = AVL(case)
    print('INPUT  :', tree, "DELETE:", del_value)
    tree.remove(del_value)
    print('RESULT :', tree)
```

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 case, del_value in test_cases:
    tree = AVL(case)
    print('INPUT  :', tree, "DELETE:", del_value)
    tree.remove(del_value)
    print('RESULT :', tree)
```

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)
tree = AVL(case)
for del_value in case:
     print('INPUT  :', tree, del_value)
     tree.remove(del_value)
     print('RESULT :', tree)
```

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)
tree = AVL(case)
for _ in case[:-2]:
    root_value = tree.get_root().value
    print('INPUT  :', tree, root_value)
    tree.remove(root_value)
    print('RESULT :', tree)
```

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)))
    tree = AVL(case)
    for value in case[::2]:
        tree.remove(value)
    if not tree.is_valid_avl():
        raise Exception("PROBLEM WITH REMOVE OPERATION")
print('Stress test finished')
```

Output:

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